深入解析Python语言，透析应用开发的方方面面

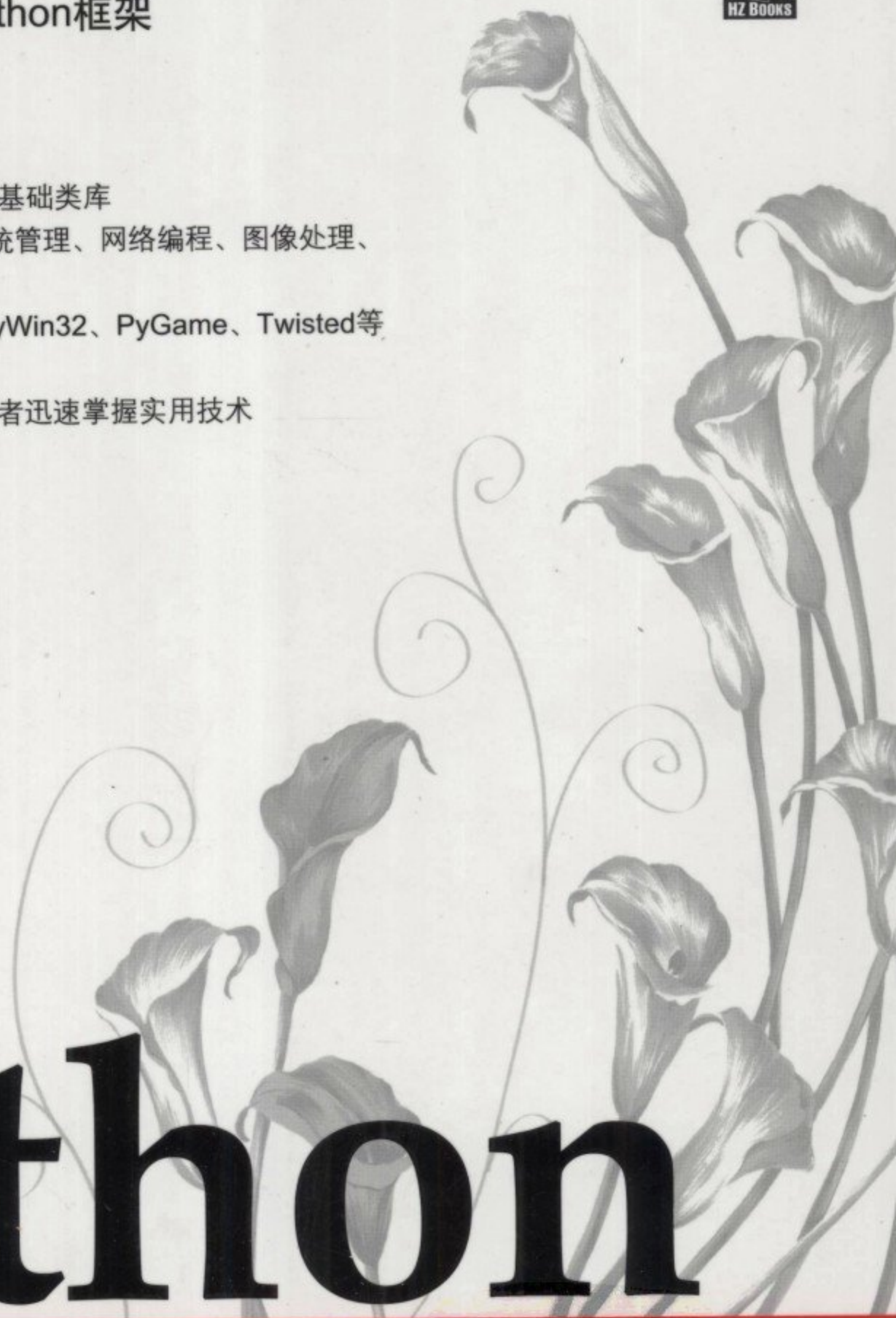
涵盖9大应用领域的各类Python框架

HZ BOOKS

本书特色

- ⊙ 详细讲解Python语言的基本要点和基础类库
- ⊙ 内容覆盖GUI、数据库、Web、系统管理、网络编程、图像处理、游戏开发等领域
- ⊙ 深入讲解常用开发框架Django、PyWin32、PyGame、Twisted等框架工具包
- ⊙ 通过丰富的典型应用实例，引领读者迅速掌握实用技术
- ⊙ ***9.5***小时多媒体视频讲解

CD-ROM



Python

开发技术详解

周伟 宗杰 等编著

机械工业出版社
China Machine Press

更多资源请访问稀酷客(www.ckook.com)

Python
开发技术详解

内容概述

- ⊙ 安装和环境 GUI显示
- ⊙ Python典型语法
- ⊙ 模块和函数
- ⊙ 内置数据结构
- ⊙ 字符串和文本操作
- ⊙ 正则表达式
- ⊙ 异常的捕获和处理
- ⊙ 面向对象的设计
- ⊙ 数据库编程
- ⊙ WxPython库的使用
- ⊙ HTML和XML应用
- ⊙ Django框架
- ⊙ 测试驱动开发
- ⊙ IPython系统管理
- ⊙ PIL库的图像处理
- ⊙ 常用网络协议应用
- ⊙ PyGame游戏开发
- ⊙ Python的扩展与嵌入
- ⊙ COM技术应用
- ⊙ GUI开发
- ⊙ XRC的使用
- ⊙ 系统剪贴板操作
- ⊙ HTTP获取
- ⊙ HTML文档解析
- ⊙ CGI
- ⊙ XML文档解析
- ⊙ SAX和DOM
- ⊙ Django框架
- ⊙ 测试驱动开发
- ⊙ 进程和线程
- ⊙ IPython
- ⊙ 系统管理应用
- ⊙ 网络编程
- ⊙ Twisted框架
- ⊙ Scapy
- ⊙ 图像处理：PIL
- ⊙ 游戏编程：PyGame
- ⊙ 扩展和嵌入应用
- ⊙ Windows下的Python

CD-ROM

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计·锡 彬

上架指导：计算机/程序设计

ISBN 978-7-111-27015-7

9 787111 270157

定价：69.00元（附光盘）

更多资源请访问稀酷客(www.ckook.com)

Python

开发技术详解

周伟 宗杰 等编著

机械工业出版社
China Machine Press

更多资源请访问稀酷客(www.ckook.com)

Python 是目前最流行的动态脚本语言之一。本书共 27 章，由浅入深、全面系统地介绍了利用 Python 语言进行程序开发的知识和技巧，包括 Python 的安装和环境配置、Python 的基本语法、模块和函数、内置数据结构、字符串和文件的处理、正则表达式的使用、异常的捕获和处理、面向对象的语言特性和设计、Python 的数据库编程、wxPython 库的使用、HTML 应用、XML 应用、Django 网页开发框架的使用方法、测试驱动开发模式应用、Python 进程和线程、Python 系统管理、网络模块、Python 图像处理和游戏开发、Python 扩展和嵌入以及 Windows 下的 Python 开发等。为了便于读者学习，本书每个章节都提供了详尽的例子，结合实例讲解各个知识点。

本书适合 Python 爱好者、大中专院校的学生、社会培训班的学生以及用 Python 语言进行系统管理、GUI 开发、Web 开发、数据库编程、网络编程的人员使用。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

Python 开发技术详解/周伟等编著. —北京：机械工业出版社，2009.8

ISBN 978-7-111-27015-7

Ⅰ.P…　Ⅱ.周…　Ⅲ.软件工具—程序设计　Ⅳ.TP311.56

中国版本图书馆 CIP 数据核字（2009）第 065986 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：盛东亮

三河市明辉印装有限公司印刷

2009 年 8 月第 1 版第 1 次印刷

184mm×260mm 1/16 · 34.5 印张

标准书号：ISBN 978-7-111-27015-7

ISBN 978-7-89451-082-2（光盘）

定价：69.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

更多资源请访问稀酷客(www.ckook.com)

前　言

当前，Python 已经成为最流行的程序设计语言之一，被越来越多的人作为首选语言来学习和应用。作为一种解释型的语言，Python 具有内置的高级数据结构和简单有效的面向对象编程机制。同时，其语法简洁而清晰，类库丰富而强大，非常适合进行快速原型开发。另外，Python 可以运行在多种系统平台下，从而使得只需要编写一次代码，就可以在多种系统平台下实现同等的功能。

与相同类型的 Perl 语言相比，年轻的 Python 语言的优势非常突出。随着各大公司支持力度的增大，其应用领域越来越广。现在，围绕着 Python 的相关技术包括 Web 开发和图形界面开发等都在飞速发展，相关的资源也越来越丰富。

为了使广大读者既能了解 Python 语言的基础知识，又能将 Python 语言应用于一些特定领域（如 Web 开发），本书全面地介绍了用 Python 语言进行程序开发的相关知识。在学习完本书之后，相信读者能够掌握 Python 语言，并可以使用 Python 语言进行实际项目的开发。

本书的特点

本书主要有以下特点。

1. 循序渐进，由浅入深

为了方便读者学习，本书在介绍 Python 的发展历史及其特点、开发环境的搭建、Python 的开发工具等基础上，在后续章节中结合具体的实例，逐步介绍了 Python 语言的基本语法以及基础类库、常用库和工具的使用方法，以及用 Python 语言进行程序开发的相关知识等内容。

2. 技术全面，内容充实

本书所讲的内容在保证实用的前提下，详细介绍了 Python 语言各个方面的知识。同时，介绍了用 Python 语言进行程序开发的相关知识，无论是用 Python 语言进行系统管理，还是进行 GUI 开发、Web 开发、数据库编程、网络编程等的读者，都可以从本书中找到应用 Python 的相关知识。

3. 对比讲解，理解深刻

考虑到有很多读者具备 Java 的开发经验，因此本书注意结合 Python 与 Java 语法的异同点进行讲解。同时本书注意对 Python 中与 Java 语法相似的函数和方法进行对比。希望通过这种讲解的方式，能帮助读者加深、加快对 Python 语法要点的理解。

4. 分析原理，步骤清晰

每种程序设计语言都有自身的独特魅力。掌握一门技术首先需要理解原理，本书注意把握各

更多资源请访问稀酷客(www.ckook.com)

个知识点的原理，总结实现的思路和步骤。读者可以根据具体步骤实现书中的例子，将理论知识与实践相结合，这样更利于学习。

5. 代码完整，讲解详尽

书中的每个知识点都有相应的实例代码，并对关键的代码部分进行了注释说明。每段代码的后面都有详细的分析，并给出了代码运行后的结果。读者可以参照运行结果阅读源程序，以便于加深理解。

本书的内容

本书共27章，各章的主要内容如下。

第一篇　Python 语言基础知识

第1章：本章从Python能做什么讲起，介绍了Python的特征和优势， Python的发展历史及其特点、开发环境的搭建、Python的开发工具等基础知识，读者如果是一个Python新手，本章将把读者带入Python世界。

第2章：详细介绍了Python的语法知识，深入讲解了Python的编码规则、变量和常量的定义及其使用方法、数据类型、运算符和表达式。通过本章的学习，读者能掌握Python编码的一些规范和Python的语法知识。

第3章：详细介绍了Python中的控制语句、循环语句以及一些习惯用法，结合实例讲解了Python进行结构化编程的要点。

第4章：主要介绍了Python的内置数据结构——元组、列表、字典和序列。分别介绍了这些内置数据结构的特点、区别及其使用方法。

第5章：讲解了Python中模块和函数的概念。重点介绍了Python的常用内置模块、函数的参数、递归函数、lambda函数、generator函数等。

第6章：重点介绍了Python中字符串的处理，包括字符串的格式化、比较、合并、截取、查找、替换等。讲解了正则表达式的概念以及使用re模块处理正则表达式的方法。

第7章：主要介绍了Python对文件的基本操作，包括文件的创建、读写、删除、复制、查找、替换和比较。重点介绍了Python对目录遍历的实现方法以及Python的流对象。

第8章：主要介绍了面向对象程序设计的要点，重点讲解了Python实现面向对象的特性以及Python中的设计模式。

第9章：详细介绍了Python对异常的处理、异常的捕获和抛出、自定义异常等内容。讲解了使用PythonWin、Eclipse调试Python程序的方法。

第10章：介绍了Python的数据库编程，重点讲解了使用ODBC、DAO、ADO、Python等专用模块连接数据库以及Python的持久化。最后结合SQLite数据库的实例讲解了Python对数据库的操作。

第二篇　Python 的 GUI 程序设计

第11章：介绍Python下的GUI开发。主要介绍了当前Python中使用比较多的Tkinter、PyGTK和PyQt等GUI开发库。然后对WxPythonGUI开发库及其开发环境进行了介绍，包括其封装以及界面设计工具等。

第12章：介绍了WxPython的基本运行机制，包括程序的组成和事件驱动模型原理及其使

更多资源请访问稀酷客(www.ckook.com)

用方法。同时，对 WxPython 中的基本组件进行了介绍。

第 13 章：讲解了 WxPython 的基本组件，包括文本框、按钮、单选框、多选框、下拉列表和布局管理器等控件的基本属性、方法和事件。

第 14 章：介绍了 WxPython 菜单与窗口组件。包括多级菜单、位图菜单、上下文菜单的创建，Frame 窗口、MDI 窗口、HTML 窗口的创建等内容。重点讲解了对话框的应用以及对话框和窗口的交互。

第 15 章：介绍了 WxPython 库中的高级控件，包括常见的表格、列表、树型和定时器控件。重点讲解了这些控件的样式、事件及应用场合。

第 16 章：介绍了 WxPython 框架的高级功能，包括 HTML 显示、XRC 的使用方法和系统剪贴板的操作。

第三篇　Python 的 Web 开发

第 17 章：介绍了 Python 的 HTML 应用。详细介绍了 URL 的解析以及 HTML 资源的获取方法，同时对 CGI 的使用和 HTML 文档的解析方法进行了介绍。

第 18 章：讲解了 Python 的 XML 操作。包括 XML 的各种内容实体如节点等，并对 XML 文档的 SAX 和 DOM 这两种处理方式进行了详细地讲解。

第 19 章：Python 的 Web 开发是近年来发展迅速的一个领域。在本章中对 Python 中常用的 Web 开发框架进行了介绍。同时在介绍 MVC 模式的基础上，对 Django 框架进行了详细地讲解，包括其基本使用和高级使用方法。

第 20 章：介绍了 Python 中的测试框架。主要讲解 Python 中的 unittest 和 doctest 这两种测试框架，且都有相应的实例，以帮助读者快速掌握。

第四篇　Python 的其他应用

第 21 章：介绍了 Python 下进程和线程的概念。主要内容包括进程和线程的创建和管理等。同时，着重讲解了多线程环境下的数据同步机制。

第 22 章：介绍了 Python 的系统管理。其中主要讲解 IPython 的应用。内容不仅包括 IPython 的介绍，还包括常见的系统管理应用。

第 23 章：讲解了 Python 中和网络编程相关的知识，包括服务器端和客户端的通信。随后着重介绍了网络中的异步通信方式，最后对 Twisted 框架进行了介绍。

第 24 章：讲解了 Python 开发中常见的网络应用。主要包括文件传输、邮件的接收和发送、远程登录以及简单网络管理功能的实现。最后还对使用 Scapy 分析网络数据进行了介绍。

第 25 章：讲解了 Python 图像处理和游戏编程。主要涉及 PIL 和 PyGame 这两个包。通过各种实例，使读者能够迅速地掌握相关知识并用于实践。

第 26 章：介绍了 Python 语言的扩展和嵌入应用。这两种技术使得 Python 语言有了更进一步的发展空间。

第 27 章：在本章中对 Windows 下的 Python 进行了介绍。主要介绍了组件对象模型和 Python 对其支持。同时，使用实例讲解了 Windows 下的各种 Python 应用。

适合的读者

- Python 爱好者

更多资源请访问稀酷客(www.ckook.com)

- 大中专院校的学生
- 社会培训班的学生
- 高等教育学校的学生
- 系统管理员
- GUI 开发人员
- Web 开发人员
- 网络编程人员

本书作者

本书主要由周伟、宗杰编著，其他参与编著和资料整理的人员有冯华君、刘博、刘燕、叶青、张军、张立娟、张艺、彭涛、徐磊、戎伟、朱毅、李佳、李玉涵、杨利润、杨春娇、武鹏、潘中强、王丹、王宁、王西莉、石淑珍、程彩红、邵毅、郑丹丹、郑海平、顾旭光。在此对他们的辛勤工作表示感谢！

更多资源请访问稀酷客(www.ckook.com)

目　录

前　言

第一篇　Python 语言基础知识

第 1 章　Python 概述 …… 1
1.1　Python 简介 …… 1
1.2　Python 的特征 …… 2
1.3　第一个 Python 程序 …… 4
1.4　搭建开发环境 …… 5
1.4.1　Python 的下载和安装 …… 5
1.4.2　交互式命令行的使用方法 …… 5
1.5　Python 的开发工具 …… 6
1.5.1　PythonWin 的使用方法 …… 6
1.5.2　Eclipse IDE 集成开发环境 …… 7
1.5.3　EditPlus 编辑器环境的配置 …… 7
1.6　不同平台下的 Python …… 9
1.7　小结 …… 9
第 2 章　Python 的基本语法 …… 10
2.1　Python 的文件类型 …… 10
2.2　Python 的编码规则 …… 11
2.2.1　命名规则 …… 11
2.2.2　代码缩进与冒号 …… 13
2.2.3　模块导入的规范 …… 15
2.2.4　使用空行分隔代码 …… 16
2.2.5　正确的注释 …… 16
2.2.6　语句的分隔 …… 19
2.3　变量和常量 …… 20
2.3.1　变量的命名 …… 20
2.3.2　变量的赋值 …… 21
2.3.3　局部变量 …… 22
2.3.4　全局变量 …… 23

更多资源请访问稀酷客(www.ckook.com)

2.3.5 常量 …… 25
2.4 **数据类型** …… 25
2.4.1 数字类型 …… 25
2.4.2 字符串类型 …… 27
2.5 **运算符与表达式** …… 28
2.5.1 算术运算符和算术表达式 …… 28
2.5.2 关系运算符和关系表达式 …… 30
2.5.3 逻辑运算符和逻辑表达式 …… 31
2.5.4 运算符的优先级 …… 31
2.6 **小结** …… 32
第3章 Python 的控制语句 …… 33
3.1 **结构化程序设计** …… 33
3.2 **条件语句** …… 34
3.2.1 if 语句 …… 34
3.2.2 if... elif... else 语句 …… 35
3.2.3 if 语句的嵌套 …… 36
3.2.4 实现 switch 语句的功能 …… 37
3.3 **循环语句** …… 39
3.3.1 while 循环 …… 39
3.3.2 for 循环 …… 40
3.3.3 break 和 continue 语句 …… 42
3.4 **结构化程序示例** …… 43
3.5 **小结** …… 45
第4章 内置数据结构 …… 46
4.1 **元组** …… 46
4.1.1 元组的创建 …… 46
4.1.2 元组的访问 …… 47
4.1.3 元组的遍历 …… 49
4.2 **列表** …… 50
4.2.1 列表的创建 …… 50
4.2.2 列表的使用方法 …… 52
4.2.3 列表的查找、排序与反转 …… 53
4.2.4 用列表实现堆栈和队列 …… 54
4.3 **字典** …… 55
4.3.1 字典的创建 …… 55
4.3.2 字典的访问 …… 56
4.3.3 字典的方法 …… 59
4.3.4 字典的排序与复制 …… 62
4.3.5 全局字典——sys. modules 模块 …… 63
4.4 **序列** …… 64

更多资源请访问稀酷客(www.ckook.com)

4.5 **小结** …… 66
第5章 模块与函数 …… 67
5.1 **Python 程序的结构** …… 67
5.2 **模块** …… 67
5.2.1 模块的创建 …… 68
5.2.2 模块的导入 …… 68
5.2.3 模块的属性 …… 69
5.2.4 模块的内置函数 …… 70
5.2.5 自定义包 …… 73
5.3 **函数** …… 75
5.3.1 函数的定义 …… 75
5.3.2 函数的参数 …… 76
5.3.3 函数的返回值 …… 78
5.3.4 函数的嵌套 …… 80
5.3.5 递归函数 …… 82
5.3.6 lambda 函数 …… 84
5.3.7 Generator 函数 …… 85
5.4 **小结** …… 86
第6章 字符串与正则表达式 …… 87
6.1 **字符串的操作** …… 87
6.1.1 字符串的格式化 …… 87
6.1.2 字符串的转义符 …… 89
6.1.3 字符串的合并 …… 90
6.1.4 字符串的截取 …… 91
6.1.5 字符串的比较 …… 92
6.1.6 字符串的反转 …… 94
6.1.7 字符串的查找和替换 …… 95
6.1.8 字符串与日期的转换 …… 96
6.2 **正则表达式** …… 97
6.2.1 正则表达式简介 …… 97
6.2.2 使用 sys. re 模块处理正则表达式 …… 100
6.3 **小结** …… 104
第7章 文件的处理 …… 105
7.1 **文件的基本操作** …… 105
7.1.1 文件的打开或创建 …… 105
7.1.2 文件的读取 …… 107
7.1.3 文件的写入 …… 109
7.1.4 文件的删除 …… 109
7.1.5 文件的复制 …… 111
7.1.6 文件的重命名 …… 111

更多资源请访问稀酷客(www.ckook.com)

7.1.7 文件内容的查找和替换 …… 113
7.1.8 文件的比较 …… 114
7.1.9 配置文件的访问 …… 115
7.2 **目录的基本操作** …… 117
7.2.1 目录的创建与删除 …… 117
7.2.2 目录的遍历 …… 118
7.3 **文件和流** …… 121
7.3.1 Python 的流对象 …… 121
7.3.2 模拟 Java 的输入、输出流 …… 122
7.4 **文件处理示例——文件属性浏览程序** …… 123
7.5 **小结** …… 124
第 8 章 面向对象编程 …… 126
8.1 **面向对象编程概述** …… 126
8.2 **类和对象** …… 127
8.2.1 类和对象的区别 …… 127
8.2.2 类的定义 …… 127
8.2.3 对象的创建 …… 128
8.2.4 Python 对象的体系结构 …… 128
8.3 **属性和方法** …… 129
8.3.1 类的属性 …… 129
8.3.2 类的方法 …… 131
8.3.3 内部类的使用 …… 132
8.3.4 __init__方法 …… 133
8.3.5 __del__方法 …… 134
8.3.6 垃圾回收机制 …… 135
8.3.7 类的内置方法 …… 136
8.3.8 方法的动态特性 …… 139
8.4 **继承** …… 140
8.4.1 使用继承 …… 141
8.4.2 抽象类的模拟 …… 143
8.4.3 多态性 …… 144
8.4.4 多重继承 …… 145
8.4.5 更好的继承方式 …… 146
8.5 **运算符的重载** …… 148
8.6 **Python 与设计模式** …… 149
8.6.1 设计模式简介 …… 149
8.6.2 设计模式示例——Python 实现工厂方法 …… 150
8.7 **小结** …… 151
第 9 章 异常处理与程序调试 …… 152
9.1 **异常的处理** …… 152

更多资源请访问稀酷客(www.ckook.com)

9.1.1 Python 中的异常 …… 152
9.1.2 try…except 的使用方法 …… 153
9.1.3 try…finally 的使用方法 …… 155
9.1.4 使用 raise 抛出异常 …… 155
9.1.5 自定义异常 …… 156
9.1.6 assert 语句的使用方法 …… 157
9.1.7 异常信息 …… 157
9.2 **使用 PythonWin 调试程序** …… 158
9.3 **使用 Eclipse for Python 调试程序** …… 160
9.3.1 新建工程 …… 160
9.3.2 配置调试 …… 161
9.3.3 设置断点 …… 163
9.4 **小结** …… 164
第 10 章 Python 的数据库编程 …… 165
10.1 **Python 环境下的数据库编程** …… 165
10.1.1 通过 ODBC 访问数据库 …… 165
10.1.2 使用 DAO 对象访问数据库 …… 167
10.1.3 使用 ActiveX Data Object 访问数据库 …… 168
10.1.4 Python 连接数据库的专用模块 …… 170
10.2 **Python 的持久化模块** …… 172
10.2.1 用 dbhash 模块读写数据 …… 172
10.2.2 shelve 模块读写数据 …… 173
10.3 **嵌入式数据库 SQLite** …… 174
10.3.1 sqlite 的命令行工具 …… 174
10.3.2 使用 sqlite3 模块访问 sqlite 数据库 …… 175
10.4 **小结** …… 177

第二篇 Python 的 GUI 程序设计

第 11 章 WxPython 的开发环境 …… 179
11.1 **Python 所支持的 GUI 开发库** …… 179
11.1.1 Tkinter …… 179
11.1.2 PyGTK …… 182
11.1.3 PyQT …… 185
11.2 **WxPython 介绍** …… 186
11.2.1 下载和安装 WxPython …… 187
11.2.2 第一个 WxPython 程序 …… 187
11.2.3 WxPython 的封装：Pythoncard …… 188
11.3 **WxPython 的开发工具** …… 189
11.3.1 使用 wxGlade 开发 WxPython 程序 …… 190
11.3.2 使用 wxFormBuilder 开发 WxPython 程序 …… 190

更多资源请访问稀酷客(www.ckook.com)

11.3.3 使用 Boa-constructor 开发 WxPython 程序 …… 191
11.4 **小结** …… 191
第 12 章 WxPython 框架初步 …… 192
12.1 **WxPython 的程序结构** …… 192
12.1.1 WxPython 应用程序的组成 …… 192
12.1.2 WxPython 窗口的组成 …… 194
12.1.3 WxPython 应用程序的运行过程 …… 196
12.2 **WxPython 中的事件驱动编程** …… 197
12.2.1 事件驱动的编程模型 …… 197
12.2.2 事件的触发 …… 197
12.2.3 事件的绑定和处理 …… 199
12.3 **WxPython 的常用组件** …… 201
12.3.1 对话框 …… 201
12.3.2 工具栏 …… 201
12.3.3 状态栏 …… 202
12.4 **小结** …… 203
第 13 章 WxPython 库中的基本组件 …… 204
13.1 **静态文本框** …… 204
13.2 **文本框** …… 206
13.2.1 单行文本框 …… 206
13.2.2 多行文本框 …… 208
13.3 **按钮控件** …… 209
13.3.1 普通按钮 …… 209
13.3.2 位图按钮 …… 210
13.4 **单选框** …… 210
13.4.1 单选框的选择 …… 210
13.4.2 单选框的分组 …… 212
13.5 **多选框** …… 212
13.5.1 多选框 …… 213
13.5.2 多选框事件 …… 213
13.6 **列表控件** …… 215
13.6.1 列表框 …… 215
13.6.2 下拉列表 …… 216
13.6.3 可编辑的下拉列表 …… 217
13.7 **使用 Sizers 布局组件** …… 218
13.7.1 Sizers 布局管理器 …… 218
13.7.2 Grid Sizer 布局 …… 218
13.7.3 Flex Grid Sizer 布局 …… 219
13.7.4 Grid Bag sizer 布局 …… 220
13.7.5 Box Sizer 布局 …… 221

更多资源请访问稀酷客(www.ckook.com)

13.8 **小结** …… 221
第 14 章 WxPython 库中的菜单、窗口与对话框控件 …… 222
14.1 **菜单** …… 222
14.1.1 创建菜单 …… 222
14.1.2 菜单的事件 …… 224
14.1.3 多级菜单 …… 225
14.1.4 菜单的快捷键 …… 226
14.1.5 位图菜单 …… 227
14.1.6 上下文菜单 …… 228
14.2 **窗口** …… 229
14.2.1 Frame 窗口 …… 229
14.2.2 带工具栏和状态栏的 Frame 窗口 …… 230
14.2.3 MDI 窗口 …… 231
14.2.4 MiniFrame 窗口 …… 232
14.3 **对话框** …… 232
14.3.1 对话框的创建 …… 233
14.3.2 提示对话框 …… 234
14.3.3 文本输入对话框 …… 235
14.3.4 特殊对话框 …… 236
14.3.5 窗口与对话框的交互 …… 236
14.4 **小结** …… 238
第 15 章 WxPython 库中的高级控件 …… 239
15.1 **表格控件** …… 239
15.1.1 表格控件的创建 …… 239
15.1.2 通过 PyGridTable 创建表格 …… 240
15.1.3 表格控件的单元格 …… 241
15.1.4 处理单元格的事件 …… 243
15.2 **高级列表控件** …… 245
15.2.1 列表控件的创建 …… 245
15.2.2 列表控件的排序 …… 246
15.2.3 带位图的列表控件 …… 247
15.3 **树型控件** …… 248
15.3.1 树型控件的创建 …… 248
15.3.2 树型控件的事件 …… 249
15.3.3 树表控件 …… 250
15.4 **定时器控件** …… 251
15.5 **小结** …… 253
第 16 章 WxPython 库中的高级功能 …… 254
16.1 **WxPython 下的 HTML 显示** …… 254
16.1.1 HTML 的显示 …… 254

更多资源请访问稀酷客(www.ckook.com)

16.1.2 基于 HtmlWindow 的简单浏览器 …… 256
16.2 **WxPython 中的 XRC** …… 257
16.2.1 使用 XRC …… 257
16.2.2 XRC 文件 …… 258
16.3 **剪贴板的操作** …… 260
16.3.1 WxPython 中的剪贴板对象 …… 260
16.3.2 处理剪贴板对象 …… 260
16.3.3 剪贴板操作实例 …… 261
16.4 **小结** …… 262

第三篇 Python 的 Web 开发

第 17 章 Python 的 HTML 应用 …… 263
17.1 **HTML 介绍** …… 263
17.1.1 HTML 的历史 …… 263
17.1.2 SGML、HTML 和 XHTML 三者的关系 …… 264
17.1.3 HTML 的标签 …… 264
17.1.4 HTML 的框架组成 …… 265
17.2 **URL 的处理** …… 266
17.2.1 统一资源定位符 URL …… 266
17.2.2 URL 的解析 …… 267
17.2.3 URL 的拼合 …… 269
17.2.4 URL 的分解 …… 270
17.2.5 URL 的编码与解码 …… 271
17.2.6 中文的编码与解码 …… 273
17.2.7 查询参数的编码 …… 274
17.3 **CGI 的使用** …… 276
17.3.1 CGI 介绍 …… 276
17.3.2 获取 CGI 环境信息 …… 277
17.3.3 解析用户的输入 …… 280
17.4 **获取 HTML 资源** …… 282
17.4.1 使用 urlopen 和 urlretrieve 获取 HTTP 资源 …… 282
17.4.2 分析返回资源的相关信息 …… 287
17.4.3 自定义获取资源方式 …… 288
17.4.4 使用 urllib2 模块获取资源 …… 291
17.4.5 使用 httplib 模块获取资源 …… 293
17.5 **HTML 文档的解析** …… 295
17.5.1 使用 HTMLParser 模块 …… 295
17.5.2 sgmllib 的 HTML 文档处理 …… 297
17.5.3 HTML 文档元素的处理 …… 297
17.5.4 使用 htmllib 处理 HTML 文档 …… 299

更多资源请访问稀酷客(www.ckook.com)

17.6 小结 …… 302
第 18 章 Python 和 XML …… 303
18.1 XML 介绍 …… 303
18.1.1 XML 的演进历史 …… 303
18.1.2 XML 的优点和限制 …… 304
18.1.3 XML 技术的 Python 支持 …… 305
18.2 XML 文档概览和验证 …… 305
18.2.1 XML 文档的基本概念 …… 306
18.2.2 XML 文档的结构良好性验证 …… 306
18.2.3 XML 文档的有效性验证 …… 309
18.3 XML 文档的结构 …… 311
18.3.1 XML 的元素和标签 …… 311
18.3.2 元素的属性 …… 313
18.3.3 XML 的文字 …… 315
18.3.4 字符实体 …… 317
18.3.5 CDATA 段 …… 318
18.3.6 注释 …… 320
18.3.7 处理指令 …… 320
18.3.8 XML 定义 …… 321
18.4 使用 SAX 处理 XML 文档 …… 321
18.4.1 SAX 介绍 …… 321
18.4.2 SAX 处理的组成部分 …… 322
18.5 使用 DOM 处理 XML 文档 …… 326
18.5.1 DOM 介绍 …… 327
18.5.2 xml.dom 模块中的接口操作 …… 328
18.5.3 对 XML 文档的操作 …… 335
18.6 小结 …… 340
第 19 章 Python 的 Web 开发——Django 框架的应用 …… 341
19.1 常见的 Web 开发框架 …… 341
19.1.1 Zope …… 341
19.1.2 TurboGears …… 342
19.1.3 Django …… 343
19.1.4 其他的 Web 开发框架 …… 344
19.1.5 各种开发框架的选择 …… 345
19.2 MVC 模式 …… 345
19.2.1 MVC 模式介绍 …… 346
19.2.2 MVC 模式的优点和缺点 …… 347
19.2.3 Django 框架中的 MVC …… 347
19.3 Django 开发环境的搭建 …… 348
19.3.1 Django 框架的安装 …… 348

更多资源请访问稀酷客(www.ckook.com)

19.3.2 数据库的配置 …… 348
19.4 **Django 框架的应用** …… 349
19.4.1 Web 应用的创建 …… 349
19.4.2 Django 中的开发服务器 …… 350
19.4.3 创建数据库 …… 352
19.4.4 生成 Django 应用 …… 353
19.4.5 创建数据模型 …… 354
19.4.6 URL 设计 …… 354
19.4.7 创建视图 …… 356
19.4.8 模板系统 …… 357
19.4.9 发布 Django 项目 …… 358
19.5 **Django 框架的高级应用** …… 359
19.5.1 管理界面 …… 359
19.5.2 生成数据库数据 …… 361
19.5.3 Session 功能 …… 362
19.5.4 国际化 …… 364
19.6 **小结** …… 365
第 20 章 敏捷方法学在 Python 中的应用——测试驱动开发 …… 366
20.1 **测试驱动开发** …… 366
20.1.1 测试驱动开发模式（TDD） …… 366
20.1.2 TDD 的优势 …… 368
20.1.3 TDD 的使用步骤 …… 368
20.2 **unittest 测试框架** …… 369
20.2.1 unittest 模块介绍 …… 369
20.2.2 构建测试用例 …… 370
20.2.3 构建测试固件 …… 371
20.2.4 组织多个测试用例 …… 373
20.2.5 构建测试套件 …… 374
20.2.6 重构代码 …… 375
20.2.7 执行测试 …… 376
20.3 **使用 doctest 模块进行测试** …… 380
20.3.1 doctest 模块介绍 …… 380
20.3.2 构建可执行文档 …… 380
20.3.3 执行 doctest 测试 …… 382
20.4 **小结** …… 384

第四篇 Python 的其他应用

第 21 章 Python 中的进程和线程 …… 385
21.1 **进程和线程** …… 385
21.1.1 进程和线程的概念 …… 385

更多资源请访问稀酷客(www.ckook.com)

21.1.2 Python 中对于进程和线程处理的支持 …… 386
21.2 **Python 下的进程编程** …… 387
21.2.1 进程的运行环境 …… 387
21.2.2 创建进程 …… 388
21.2.3 终止进程 …… 389
21.3 **使用 subprocess 模块管理进程** …… 390
21.3.1 使用 Popen 类管理进程 …… 390
21.3.2 调用外部系统命令 …… 393
21.3.3 替代其他进程创建函数 …… 393
21.4 **进程间的信号机制** …… 393
21.4.1 信号的处理 …… 394
21.4.2 信号的使用规则 …… 396
21.5 **多线程概述** …… 396
21.5.1 什么是多线程 …… 397
21.5.2 线程的状态 …… 397
21.5.3 Python 中的线程支持 …… 397
21.6 **生成和终止线程** …… 398
21.6.1 使用 thread 模块 …… 398
21.6.2 使用 threading. Thread 类 …… 400
21.7 **管理线程** …… 402
21.7.1 线程状态转移 …… 402
21.7.2 主线程对子线程的控制 …… 403
21.7.3 线程中的局部变量 …… 403
21.8 **线程之间的同步** …… 404
21.8.1 临界资源和临界区 …… 404
21.8.2 锁机制 …… 406
21.8.3 条件变量 …… 407
21.8.4 信号量 …… 409
21.8.5 同步队列 …… 410
21.8.6 线程同步小结 …… 411
21.9 **小结** …… 412
第 22 章 基于 Python 的系统管理 …… 413
22.1 **增强的交互式环境 IPython** …… 413
22.1.1 IPython 介绍 …… 413
22.1.2 IPython 的安装 …… 414
22.1.3 IPython 的启动 …… 415
22.1.4 IPython 的环境配置 …… 416
22.1.5 IPython 的退出 …… 416
22.2 **和 IPython 的简单交互** …… 416
22.2.1 IPython 中的输入与输出 …… 417

更多资源请访问稀酷客(www.ckook.com)

22.2.2 输出提示符的区别 …… 417
22.2.3 输出提示符不同的原因 …… 418
22.2.4 IPython 中的 In 和 Out …… 420
22.3 **IPython 中的 magic 函数** …… 420
22.3.1 Magic 函数的使用和构造 …… 420
22.3.2 目录管理 …… 422
22.3.3 对象信息的收集 …… 426
22.4 **IPython 适合于系统管理的特点** …… 428
22.4.1 Tab 补全 …… 428
22.4.2 历史记录功能 …… 429
22.4.3 执行外部的系统命令和运行脚本文件 …… 432
22.4.4 对象的查看和自省 …… 435
22.4.5 直接编辑代码 …… 438
22.4.6 设置别名和宏 …… 440
22.5 **使用 Python 进行文件管理** …… 441
22.5.1 文件的比较 …… 441
22.5.2 文件的归档 …… 443
22.5.3 文件的压缩 …… 445
22.6 **使用 Python 定时执行任务** …… 447
22.6.1 使用休眠功能 …… 447
22.6.2 使用 sched 来定时执行任务 …… 447
22.7 **小结** …… 448
第 23 章 Python 和网络编程 …… 449
23.1 **网络模型介绍** …… 449
23.1.1 OSI 简介 …… 449
23.1.2 TCP/IP 简介 …… 450
23.2 **Socket 应用** …… 451
23.2.1 Socket 基础知识 …… 451
23.2.2 Socket 的工作方式 …… 451
23.3 **服务器端和客户端通信** …… 452
23.3.1 服务器端的构建 …… 452
23.3.2 客户端的构建 …… 456
23.4 **异步通信方式** …… 458
23.4.1 使用 Fork 方式 …… 458
23.4.2 使用线程方式 …… 459
23.4.3 异步 IO 方式 …… 460
23.4.4 使用 asyncore 模块 …… 463
23.5 **Twisted 网络框架** …… 466
23.5.1 认识 Twisted 框架 …… 466
23.5.2 Twisted 框架下服务器端的实现 …… 467

更多资源请访问稀酷客(www.ckook.com)

23.5.3 Twisted 框架下服务器端的其他处理 …… 468
23.6 小结 …… 470
第 24 章 常见的 Python 网络应用 …… 471
24.1 使用 FTP 传输文件 …… 471
24.1.1 FTP 的工作原理和 Python 库的支持 …… 471
24.1.2 FTP 的登录和退出 …… 472
24.1.3 FTP 的数据传输 …… 473
24.2 使用 POP3 获取邮件 …… 474
24.2.1 POP3 协议介绍 …… 474
24.2.2 poplib 模块的使用方法 …… 475
24.3 使用 SMTP 发送邮件 …… 477
24.3.1 SMTP 协议介绍 …… 477
24.3.2 smtplib 模块的使用 …… 478
24.4 使用 Telnet 协议远程登录 …… 479
24.4.1 Telnet 协议介绍和 Python 库的支持 …… 479
24.4.2 telnetlib 模块的使用方法 …… 480
24.5 使用 SNMP 管理网络 …… 481
24.5.1 SNMP 协议的组成 …… 481
24.5.2 PySNMP 框架介绍及其使用方法 …… 482
24.6 网络分析 …… 483
24.6.1 网络分析概述 …… 483
24.6.2 使用 Scapy 在网络中抓包分析 …… 484
24.7 小结 …… 486
第 25 章 图像处理和游戏开发 …… 487
25.1 图像处理的基本概念 …… 487
25.1.1 Python 下的图像处理包 …… 487
25.1.2 PIL 支持的图像文件格式 …… 488
25.1.3 图像处理中的其他概念 …… 489
25.2 图像的基本处理 …… 490
25.2.1 图像的读写操作 …… 490
25.2.2 获取图像信息 …… 491
25.2.3 图像文件格式的转换 …… 492
25.2.4 图像的裁剪和合成 …… 494
25.2.5 图像的变换 …… 496
25.3 图像处理的高级应用 …… 498
25.3.1 图像的通道操作 …… 498
25.3.2 图像的增强处理 …… 501
25.3.3 PIL 中的内置滤镜 …… 503
25.4 使用 PyGame 进行游戏开发 …… 504
25.4.1 Python 游戏开发包：PyGame …… 504

更多资源请访问稀酷客(www.ckook.com)

25.4.2 创建基本的 Pygame 窗口 …… 506
25.4.3 事件系统 …… 508
25.5 **小结** …… 509
第 26 章 Python 语言的扩展与嵌入 …… 510
26.1 **Python 语言的扩展** …… 510
26.1.1 Python 扩展简介 …… 510
26.1.2 一个 C 扩展的例子 …… 511
26.1.3 模块方法表和初始化函数 …… 512
26.1.4 编译和测试 …… 513
26.2 **Python 语言的嵌入** …… 515
26.2.1 Python 嵌入简介 …… 515
26.2.2 一个 Python 嵌入的例子 …… 516
26.2.3 更好的嵌入 …… 517
26.3 **小结** …… 519
第 27 章 Windows 下的 Python …… 520
27.1 **组件对象模型** …… 520
27.1.1 组件对象模型介绍 …… 520
27.1.2 COM 结构 …… 521
27.1.3 COM 对象的交互 …… 521
27.2 **Python 对 COM 技术的支持** …… 522
27.2.1 Python 中的 Windows 扩展：PyWin32 …… 522
27.2.2 客户端 COM 组件 …… 523
27.2.3 实现 COM 组件 …… 524
27.3 **Windows 下的常见 Python 应用** …… 528
27.3.1 对 Word 的自动访问 …… 528
27.3.2 对 Excel 的自动访问 …… 529
27.3.3 对 Powerpoint 的自动访问 …… 530
27.3.4 对 Outlook 的自动访问 …… 532
27.4 **小结** …… 532

更多资源请访问稀酷客(www.ckook.com)

第一篇
Python 语言基础知识

第 1 章　Python 概述

Python（蟒蛇）是一种动态解释型的编程语言。Python 功能强大、简单易学，支持面向对象、函数式编程。Python 可以在 Windows、UNIX 等多种操作系统上使用，也可以在 Java、.NET 开发平台上使用。Python 的简洁性、易用性使得程序的开发过程变得简捷，特别适用于快速应用开发。本章的知识点：

- Python 的特征
- Python 开发环境的配置
- Python 程序的编写
- 不同开发平台对 Python 的支持

1.1　Python 简介

Python 语言是由 Guido van Rossum 在 1989 年开发的，于 1991 年年初发表。Guido van Rossum 曾是 CWI 公司的一员，使用解释性编程语言 ABC 开发应用程序，这种语言在软件开发上有许多局限性。由于要完成系统管理方面的一些任务，需要获取 Amoeba 机操作系统所提供的系统调用能力，虽然可以设计 Amoeba 的专用语言去实现这个任务，但是 van Rossum 计划设计一门更通用的程序设计语言。Python 就此诞生了。

Python 语言已经诞生了 20 年，但是并没有成为程序开发领域的主流程序设计语言。这是因为 Python 语言的动态性，程序解释执行的速度比编译型语言慢。随着 Python 语言的不断优化以及计算机硬件技术的迅猛发展，动态语言已经越来越受到软件界的重视。其中的代表性语言有 Python、Ruby、SmallTalk、Groovy 等。

众所周知，Java 是软件行业认可的程序开发语言。Java 比C++ 更容易使用，内部结构也相对简单。而 Python 的语法特性使得程序设计变得更轻松，Python 能编写出比 Java 可读性更强的代码。随着 Jython 等解释器的出现，使得 Python 可以在 Java 虚拟机上运行。这样，Python 就可以使用 Java 丰富的应用程序包。Python 与读者熟知的 JavaScript 非常相似，都是解释执行，而且语法结构有很多相似之处。JavaScript 是浏览器端的脚本语言，而 Python 也可以用于 Web 方面的开发。

Python 吸收了 Perl、Tcl 等脚本语言的优点，使得 Python 具备 Tcl 的扩展性，同时又具备

更多资源请访问稀酷客(www.ckook.com)

Perl 的文本解析和匹配能力。Python 与 Lisp 也有相似之处，Python 可以实现函数式的编程模型。

1.2　Python 的特征

程序设计语言在不断地发展，从最初的汇编语言到后来的 C、Pascal 等语言，发展到现在的 C++ 、Java 等高级编程语言。程序设计的难度在不断的减小，软件的开发和设计已经形成了一套标准，开发工作已经不再是复杂的任务。最初只能使用机器码编写代码，而现在可以使用具有良好调试功能的 IDE 环境编程。Python 使用 C 语言开发，但是 Python 不再有 C 语言中的指针等复杂的数据类型。Python 的简洁性使得软件的代码大幅度的减少，开发任务进一步简化。程序员关注的重点不再是语法特性，而是程序所要实现的任务。Python 语言有许多重要的特性，而且有些特性是富有创造性的。

1. 面向对象的特性

面向对象的程序设计降低了结构化程序设计的复杂性，使得程序设计更贴近现实生活。结构化程序设计把数据和逻辑混合在一起，不便于程序的维护。面向对象的程序设计抽象出对象的行为和属性，把行为和属性分离开，但又合理的组织在一起。Python 语言具有很强的面向对象特性，而且简化了面向对象的实现。它消除了保护类型、抽象类、接口等面向对象的元素，使得面向对象的概念更容易理解。

2. 简单性

Python 语言的保留字比较少。它没有分号、begin、end 等标记，代码块使用空格或制表键缩进的方式分隔代码。Python 的代码简洁、短小，易于阅读。Python 简化了循环语句，使程序结构很清晰，方便阅读。详细说明请参考第 2 章的相关内容。表 1-1 列出了 Python2.5 中所有的保留字。

表 1-1　Python2.5 中所有的保留字

保　留　字	说　　明
and	用于表达式运算，逻辑与操作
as	用于类型转换
assert	断言，用于判断变量或条件表达式的值是否为真
break	中断循环语句的执行
class	用于定义类
continue	继续执行下一次循环
def	用于定义函数或方法
del	删除变量或序列的值
elif	条件语句，与 if、else 结合使用
else	条件语句，与 if、elif 结合使用。也可用于异常和循环语句
except	except 包含捕获异常后的操作代码块，与 try、finally 结合使用
exec	用于执行 Python 语句
for	for 循环语句
finally	用于异常语句，出现异常后，始终要执行 finally 包含的代码块。与 try、except 结合使用
from	用于导入模块，与 import 结合使用
global	定义全局变量
if	条件语句，与 else、elif 结合使用

更多资源请访问稀酷客(www.ckook.com)

（续）

保 留 字	说　明
import	用于导入模块，与 from 结合使用
in	判断变量是否在序列中
is	判断变量是否为某个类的实例
lambda	定义匿名函数
not	用于表达式运算，逻辑非操作
or	用于表达式运算，逻辑或操作
pass	空的类、方法或函数的占位符
print	打印语句
raise	异常抛出操作
return	用于从函数返回计算结果
try	try 包含可能会出现异常的语句，与 except、finally 结合使用
while	while 循环语句
with	简化 Python 的语句
yield	用于从函数依次返回值（参考 5.3.7 节的内容）

可以使用以下代码查看 Python 中的保留字：

```
from keyword import kwlist
print kwlist
```

3. 内置的数据结构

Python 提供了一些内置的数据结构，这些数据结构实现了类似 Java 中集合类的功能。Python 的数据结构包括元组、列表、字典等。内置的数据结构简化了程序的设计。元组相当于“只读”的数组，列表可以作为可变长度的数组使用，字典相当于 Java 中的 HashTable 类型。内置数据结构的使用方法详见第 4 章。

4. 健壮性

Python 提供了异常处理机制，能捕获程序的异常情况。此外，Python 的堆栈跟踪对象能够指出程序出错的位置和出错的原因。异常机制能够避免不安全退出的情况，同时能帮助程序员调试程序。详细说明请参考第 9 章的相关内容。

5. 跨平台性

Python 会先编译为与平台相关的二进制代码，然后再解释执行，这种方式和 Java 类似。Python编写的应用程序可以运行在 Windows、UNIX、Linux 等不同的操作系统上。在一种操作系统上编写的 Python 代码只需要做少量的修改，就可以移植到其他操作系统上。

6. 可扩展性

Python 是采用 C 开发的语言，因此可以使用 C 扩展 Python，可以给 Python 添加新的模块、新的类。同时 Python 可以嵌入到 C、C++ 语言开发的项目中，使程序具备脚本语言的特性。

7. 动态性

Python 与 JavaScript、PHP、Perl 等语言类似。Python 不需要另外声明变量，直接赋值即可创建一个新的变量。

更多资源请访问稀酷客(www.ckook.com)

8. 强类型语言

Python 的变量创建后会对应一种数据类型，Python 会根据赋值表达式的内容决定变量的数据类型。Python 在内部建立了管理这些变量的机制，出现在同一个表达式中的不同类型的变量需要做类型转换。

9. 应用广泛

Python 语言应用于数据库、网络、图形图像、数学计算、Web 开发、操作系统扩展等领域。Python 有许多第三方库支持。例如，PIL 库用于图像处理、NumPy 库用于数学计算、WxPython 库用于 GUI 程序的设计、Django 库用于 Web 应用程序的开发等。

1.3　第一个 Python 程序

Python 的源代码文件以“py”作为文件名后缀。下面编写一段简单的 Python 程序。

【例 1-1】创建用于输出“hello world”字符串的程序。并以“hello. py”为文件名保存起来。

```
if__name__ == "__main__":
    print "hello world"
```

- 第 1 行代码相当于 C 语言中的 main（）函数，是 Python 程序的入口。详细用法请参考第 5 章的相关内容。
- 第 2 行代码使用 print 语句输出字符串“hello world”。

输出结果如下所示：

```
hello world
```

Python 的 print 语句用于输出字符串的内容，即把双引号中的内容输出到控制台。Python 的输入、输出是通过“流”实现的，【例 1-1】中的 print 语句把字符串的内容输出到标准输出流，这里将输出到控制台。流也可以把结果输出到文件、打印机等其他终端，关于流的详细用法请参考 7. 3 节的内容。

print 语句类似于 C 语言中的 printf 函数。在使用 print 语句时可以加“%”，用于自定义输出。通过元组可以输出多个变量的值，元组中变量的顺序必须与字符串中对应的定制符一致，定制符用于说明输出变量的类型。例如：

```
age = 30
name = "bill"
print '% s is % d years old' % (name, age)
```

其中“% s”、“% d”为定制符，字符串类型的 name 与% s 对应，整型 age 与% d 对应。元组是 Python 内置的数据结构，元组的用法请参考 4. 1 节的内容。

Python 程序的运行非常简单，命令格式如下所示：

```
python python_file_path + python_file. py
```

其中，python_ file. py 表示 Python 的源代码文件，python_ file_ path 表示 python_ file. py 所在的路径。在 DOS 窗口中输入如图 1-1 所示的命令，即可运行 hello. py 文件。

这种运行方式不够直观，而且不便于程序的调试。后面将介绍如何在编辑器 EditPlus 和开发工具 PythonWin、Eclipse 中运行 Python 程序。

更多资源请访问稀酷客(www.ckook.com)

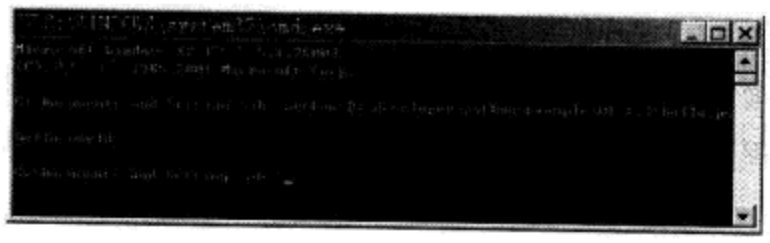


图 1-1　Python 程序的运行

1.4　搭建开发环境

Python 开发环境的安装和配置非常简单。Python 可以在多个平台进行安装和开发，ActivePython 是非常流行的 Python 安装包。本节将介绍 ActivePython 的安装和 Python 交互式命令行的使用方法。

1.4.1　Python 的下载和安装

在 UNIX 系统上，Python 的可执行文件被安装在/usr/local/bin 目录中，库文件被安装在/usr/local/python 目录中。在 Windows 环境中，Python 可以被安装到任何目录中。读者可以到官方网站 www. python. org 下载 Python2. 5，官方网站提供了 Windows、UNIX 等不同操作系统的 Python 安装。

用户也可以下载 ActivePython 组件包安装，下载地址为 http://www. activestate. com/。ActivePython是对 Python 核心和常用模块的二进制包装。它是 ActiveState 公司发布的 Python 开发环境。ActivePython 使 Python 的安装更加容易，并且可以应用在各种操作系统上。ActivePython 包含了一些常用的 Python 扩展，以及 Windows 环境的编程接口。

ActivePython 基本上包含了程序开发中所需要的组件包。如果是 Windows 用户，下载 ActivePython-2. 5. 0. 0-win32-x86. msi 二进制安装包，双击运行即可完成安装。如果是 UNIX、Linux 用户，解压 ActivePython- 2. 5. 0. 0-linux-x86. tar. gz 包即可。ActivePython 安装完成后，在 DOS 窗口中输入“python”，将显示计算机中安装的Python 版本信息。同时会启动 Python 的交互式命令行窗口，如图 1-2 所示。

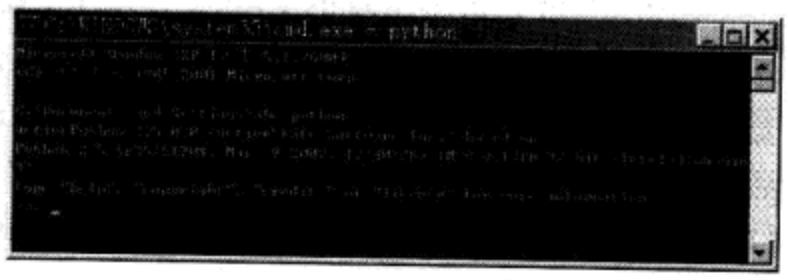


图 1-2　Python 的命令行窗口

注意　Python 命令带有一些参数，可以在 DOS 窗口中输入“python-h”查看每个参数的含义。

1.4.2　交互式命令行的使用方法

ActivePython 安装成功后，通过【开始】|【程序】|【ActiveState ActivePython2. 5】|【Python Interactive Shell】命令启动令行程序，或者在 DOS 窗口中输入“python”启动命令行程序，如图 1-3 所示。

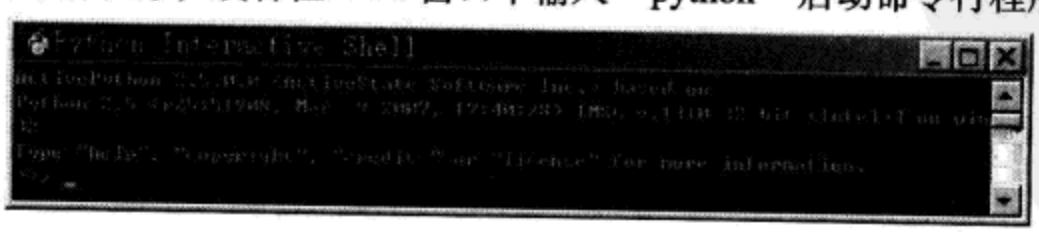


图 1-3　启动 Python 命令行窗口

通过命令行可以直接向解释器输入语句，并输出程序的运行结果。命令行窗口中的“>>>”提示符的后面可以输入 Python 语句。例如，可以使用 print 语句输出字符串“hello world”，如图

更多资源请访问稀酷客(www.ckook.com)

1-4 所示。

当然也可以在命令行窗口中输入多行 Python 代码。例如，可以把【例 1-1】中的代码输入到命令行窗口中。当输入完程序的最后一行，按两次回车键后即可结束程序，并输出程序的运行结果，如图 1-5 所示。



图 1-4　Python 命令行窗口的使用方法



图 1-5　在命令行窗口中输入多行代码

注意　如果要退出交互式命令行环境，按 Ctrl + Z 快捷键，然后按回车键即可。

1.5　Python 的开发工具

Python 的开发工具非常丰富，有许多强大的智能开发环境（IDE，Integrated Development Environment）工具，如 Komodo、PythonWin、Eclipse 等。这些工具不仅支持图形化操作，而且具备编辑、调试等功能。此外，文本编辑器也可作为 Python 的开发环境，如 EditPlus、Vi 等。

1.5.1　PythonWin 的使用方法

PythonWin 是最早出现的 Python 开发工具之一。PythonWin 的发行版本包括 Windows 应用程序接口和 COM 组件模型。PythonWin 是采用C++ 开发的，可以编辑和调试程序。ActivePython 安装完成后，单击【开始】|【程序】|【ActiveState ActivePython2.5】|【PythonWin Editor】命令，运行 PythonWin。PythonWin 运行后将出现图形化的命令行窗口，如图 1-6 所示。

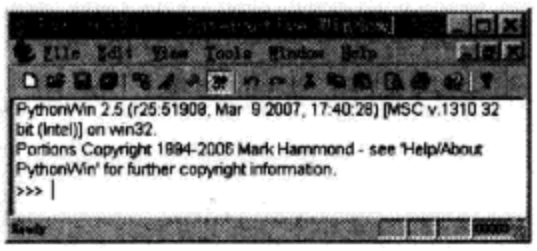


图 1-6　PythonWin 的命令行窗口

PythonWin 被安装在 Python 根目录下的\Lib\site-packages\pythonwin 子目录中，也可以在该子目录中运行“Pythonwin. exe”来启动 PythonWin。在命令行中进行多行代码的编辑十分不便，所以建议在 IDE 中进行编辑。单击【File】|【New】命令，在弹出的对话框中选择“Python Script”后确定，即可编写 Python 代码。单击【File】|【Open】命令，打开 1.3 节所编写的 hello. py，如图 1-7 所示。

PythonWin 包括编辑窗口和命令行窗口。在编辑窗口中运行 hello. py，输出结果将显示在命令行窗口中，如图 1-8 所示。

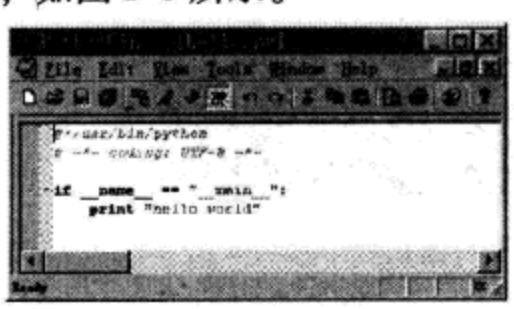


图 1-7　使用 PythonWin 编辑 Python 程序

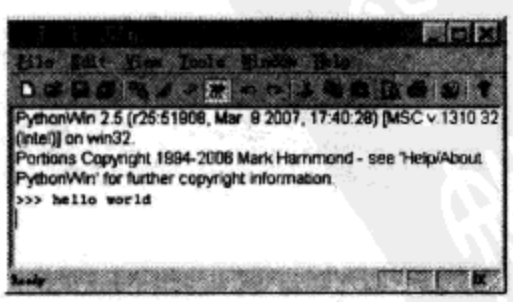


图 1-8　PythonWin 运行程序的输出结果

PythonWin 还支持断点的设置、单步调试等功能。其调试功能将在第9 章的9.2 节进行介绍。

更多资源请访问稀酷客(www.ckook.com)

1.5.2　Eclipse IDE 集成开发环境

Eclipse 是 Java 开发的一个集成开发环境，而且是一个开源项目。Eclipse 的扩展性非常强，它不仅可以作为 Java 的 IDE 使用，而且还可以开发大量的插件支持其他类型的语言，如 C、C++、Python、PHP 等。如果要在 Eclipse 平台上开发 Python，需要下载 PyDev 这个插件。easyeclipse 网站提供了 eclipse 的各种插件下载，而且可以获取单独运行的 Eclipse for Python。下载地址为 www. easyeclipse. org。

Eclipse 的功能非常强大，它实现了 Python 代码的语法加亮、代码提示和代码补全等智能化的功能。此外 Eclipse 提供了比 PythonWin 更强大的调试能力，Eclipse 还支持 Jython、Pyunit、团队开发等其他功能。

在 Eclipse 中，源代码被组织到项目（project）中。Eclipse 用户界面的结构划分为视图（View）与编辑器（Editor）。视图与编辑器包括：源代码大纲视图、Java 源代码编辑器、Python 源代码编辑器和文件系统导航视图。Eclipse 用户界面包含各种视图（perspective）。视图是通常在执行某种类型活动时使用的一组窗口。Eclipse 中的标准视图包括：Debug、Java Browsing、Java、Java Type Hierarchy、Plug-in Development、CVS Repository Exploring、Resource 和 Install/Update。Eclipse for Python 提供了一个 Pydev 视图。当启动调试模式时，eclipse 会自动切换到 Debug 视图。第 9 章中的 9.3 节将介绍 Eclipse for Python 的配置和调试方法。可以在 Eclipse 的开发环境中编写输出字符串“hello world”的程序，如图 1-9 所示。

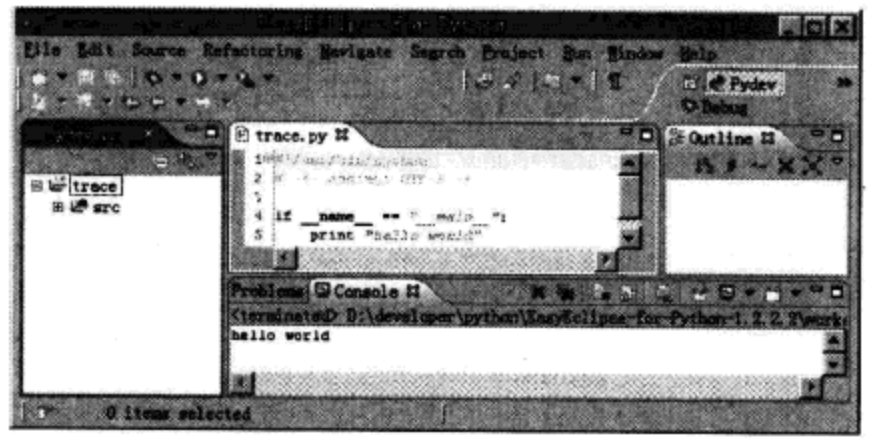


图 1-9　Eclipse 的开发环境

注意　安装 Pydev 之前，需要先在计算机中安装 Python。

1.5.3　EditPlus 编辑器环境的配置

Python 也可以使用编辑器进行开发。例如，文本编辑软件 EditPlus 也能成为 Python 的编辑、执行环境，甚至可以用于调试程序。EditPlus 具备语法加亮、代码自动缩进等功能。本节介绍一下如何配置 EditPlus 编辑器的开发环境。

1. 添加 Python 群组

首先启动 EditPlus，然后单击【工具】|【配置用户工具】命令，打开【参数】对话框。在【参数】对话框中单击【添加工具】按钮，在弹出的菜单中选择【程序】命令。新建的群组名称命名为“Python”，分别在【菜单文本】文本框中输入“python”，在【命令】文本框中输入 Python 的安装路径，在【参数】文本框中输入“$（FileName）”，在【起始目录】文本框中输入“$（FileDir）”。勾选【捕获输出】选项，Python 程序运行后的输出结果将显示在 Edit Plus 的输出栏中，否则，运行 Python 程序后将弹出命令行窗口，并把结果输出到命令行中。

设置完成后的对话框如图 1-10 所示。单击【确定】按钮，新建一个 Python 文件，【工具】菜单下将会出现【python】选项。单击【python】选项或按快捷键 Ctrl + 1，就可以运行 Python 程序。

2. 设置 Python 高亮和自动完成

EditPlus 不仅可以作为 Python 的开发环境，还支持 Java、C#、PHP、HTML 等其他类型的语言。不同语言的语法高亮显示和自动完成的特征各不相同。为了实现语法加亮和自动完成功能，

更多资源请访问稀酷客(www.ckook.com)

需要下载 python. acp 和 python. stx 这两个特征文件。下载地址为 http：//www. editplus. com/files/pythonfiles. zip。下载后把文件 python. acp 和 python. stx 解压到 EditPlus 的安装目录下。acp 后缀的文件表示自动完成的特征文件，stx 后缀的文件表示语法加亮的特征文件。在编写 Python 代码之前，需要先在 EditPlus 中设置这些特征文件。

（1）选择【文件】|【设置与语法】选项，在【文件类型】列表中选择【python】选项，【描述】文本框中输入“python”，【扩展名】文本框中输入“py”，如图 1-11 所示。

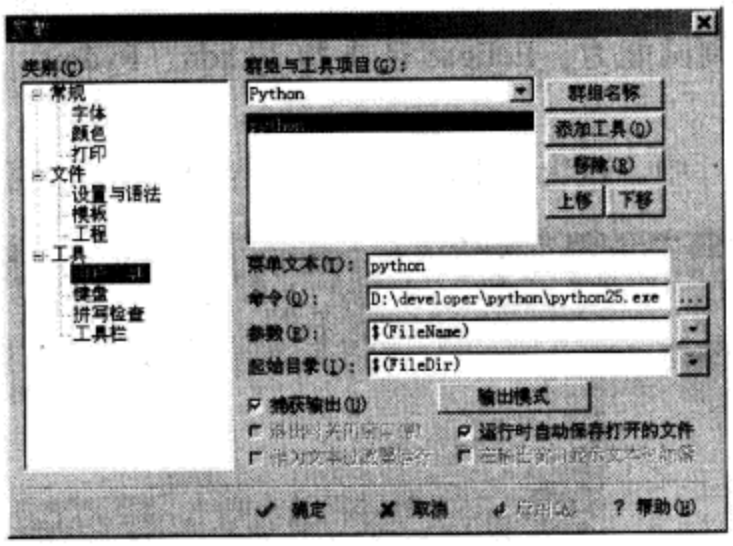


图 1-10　在 EditPlus 中添加对 Python 的支持

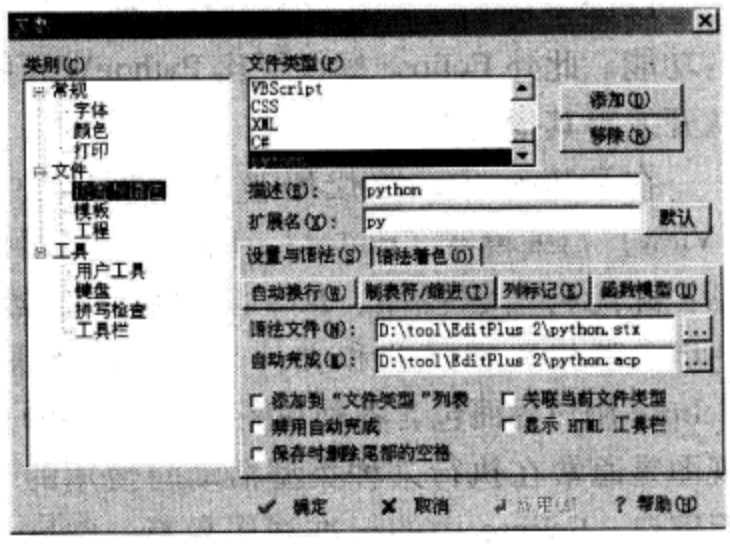


图 1-11　设置 Python 的特征文件

（2）在【设置与语法】选项卡中，在【语法文件】文本框中输入 python. stx 的路径，在【自动完成】文本框中输入 python. acp 的路径。

（3）Python 的语法中没有使用 begin、end 或 {} 区分代码块，而是使用冒号和代码缩进的方式区分代码之间的层次关系。单击【制表符/缩进】按钮，打开【制表符与缩进】对话框。设置 Python 代码的缩进方式，如图 1-12 所示。在使用 IDE 工具时，输入冒号代码会自动缩进，EditPlus 也可以设置该功能。在【制表符】和【缩进】文本框中分别输入空格的个数，一般设置为“4”。把【启用自动缩进】选项选中，在【自动缩进开始】文本框中输入“:”。单击【确定】按钮保存设置。

（4）单击【函数模型】按钮，打开【函数模型】对话框，如图 1-13 所示。在【函数模型正则表达式】文本框中输入“[\t] * def[\t]. + :”。单击【确定】按钮保存设置。

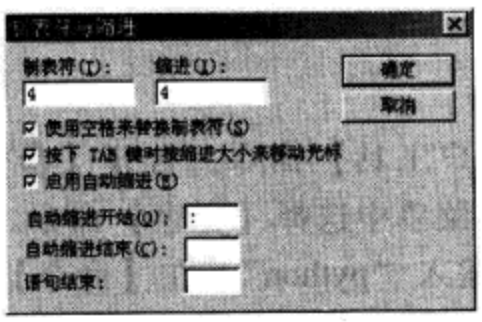


图 1-12　Python 代码的缩进方式

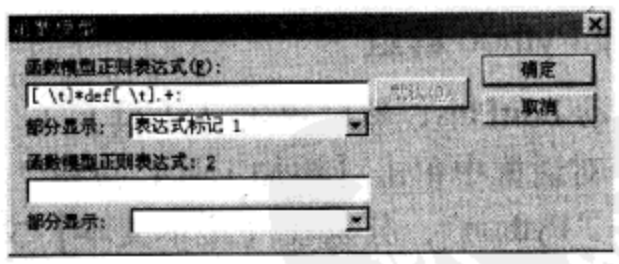


图 1-13　设置函数模型

至此，EditPlus 的 Python 开发环境就设置完成了。EditPlus 还可以建立 Python 文件的模板，以后每次新建 Python 文件都可以在模板的基础上编写代码。编写 Python 代码经常要使用中文，同时也要考虑跨平台的功能，因此可以建立名为“template. py”的模板文件。template. py 的内容如下所示。

```
#! /usr/bin/python
# -* - coding: UTF-8 -*-
```

更多资源请访问稀酷客(www.ckook.com)

- 第 1 行代码使 Python 程序可以在 UNIX 平台上运行。
- 第 2 行代码设置编码集为 UTF-8，使 Python 代码可以支持中文。

注意　在 EditPlus 中通过快捷键 Ctrl + F11 可以查看当前 python 文件中的函数列表。

运行 Python 程序前，需要先保存 Python 程序。下面使用 EditPlus 编写一段 Python 程序并输出结果，如图 1-14 所示。

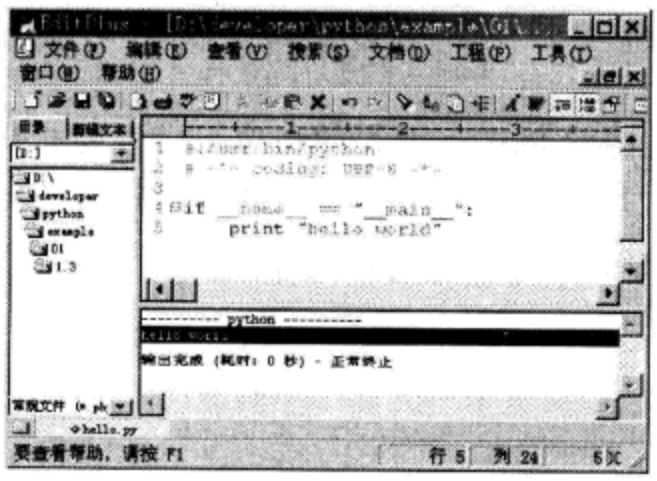


图 1-14　EditPlus 运行 Python 程序

注意　本书第一篇的代码均在 EditPlus 环境下编辑执行，笔者使用的 EditPlus 版本为 v2.31 汉化版。

1.6　不同平台下的 Python

Java 和 .NET 是目前软件行业非常成熟的两大开发平台。Python 可以在这两大开发平台上使用，也可以用 Java、C#扩展 Python。

1. Jython

Jython 是完全采用 Java 编写的 Python 解析器。虽然 Jython 解释器的实现和性能与 Python 的解释器相比还有些差距，但是 Jython 使得 Python 完全可以应用在 Java 开发平台下。Jython 使得 Python 程序可以在 Java 虚拟机上运行，同时 Python 可以访问 Java 下的类库和包。Jython 也为 Java提供了完善的脚本环境，Python 在 Java 应用中可以作为中间层服务的实现语言。Jython 使得 Java 可以扩展 Python 模块，反过来也可以使用 Python 编写 Java 应用程序。

2. IronPython

IronPython 是 Python 在 .NET 平台上的实现。IronPython 提供了交互式的控制台，该控制台支持动态编译。它使得 Python 程序员可以访问所有的 .NET 库，而且完全兼容 Python 语言。IronPython 必须提供对 .NET 2.0 版本的支持。IronPython 的出现使得可以在 .NET 平台下编写 Python 代码，同时又可以调用丰富的 .NET 类库框架。

1.7　小结

本章讲解了 Python 的历史、特性、安装、开发环境等方面的知识。Python 既可以使用 IDE 工具编写程序，也可以使用 EditPlus 等文本编辑器进行开发。本章重点讲解了 Python 程序的基本特点、EditPlus 编辑器下 Python 环境的设置以及交互式命令行的使用方法。Python 可用的开发工具很多，例如 PythonWin、Eclipse、Komodo 等，读者可以根据使用习惯进行选择。

更多资源请访问稀酷客(www.ckook.com)

第 2 章　Python 的基本语法

Python 的语法非常简练，因此 Python 编写的程序可读性强，容易理解。本章将向读者介绍 Python 的基本语法及其概念，并与目前流行的程序开发语言进行比较。Python 的语法与其他高级语言有很多不同之处，它使用了一些标记作为语法的一部分，例如空格缩进、冒号等。本章的知识点：

- Python 文件的扩展名
- Python 的编码规则
- 数据类型
- 变量和常量的定义和使用方法
- 运算符及表达式

2.1　Python 的文件类型

Python 的文件类型主要分为 3 种，分别是源代码、字节代码和优化代码。这些代码都可以直接运行，不需要进行编译或者连接。这正是 Python 语言的特性，Python 的文件通过 python. exe 或 pythonw. exe 解释运行。

1. 源代码

Python 源代码的文件以“py”为扩展名，由 python. exe 解释，可在控制台下运行。用 Python 语言写的程序不需要编译成二进制代码，可以直接运行源代码。“pyw”是程序开发图形用户接口（Graphical user interface）的源文件的扩展名，作为桌面应用程序，这种文件是专门用于开发图形界面的，由 pythonw. exe 解释运行。“py”和“pyw”类型的文件可以用文本工具打开，并可以编辑文件的内容。

2. 字节代码

Python 源文件经过编译后生成扩展名为“pyc”的文件，“pyc”是编译过的字节文件。这种文件不能使用文本编辑工具打开或修改。“pyc”文件是与平台无关的。因此 Python 的程序可以运行在 Windows、UNIX 和 Linux 等操作系统上。“py”文件直接运行后即可得到“pyc”类型的文件或通过脚本生成该类型的文件。例如，下面这段脚本可以把 hello. py 文件编译为 hello. pyc 文件：

```
import py_compile
py_compile.compile('hello.py')
```

将此脚本保存为以“py”为扩展名的文件，运行后即可得到 hello. pyc 文件。

3. 优化代码

经过优化的源文件生成扩展名为“pyo”的文件。“pyo”类型的文件需要命令行工具生成。

更多资源请访问稀酷客(www.ckook.com)

“pyo”文件也不能使用文本编辑工具打开或修改。例如，用下面的步骤把 hello. py 文件编译成 hello. pyo 文件。

（1）启动命令行窗口，进入 hello. py 文件所在的目录：

```
cd /D D:\developer\python\example\02\2.1。
```

D：\ developer \ python \ example \ 02 \ 2. 1 是笔者编写的 hello. py 文件所在的目录，请读者根据自己的环境进行修改。

（2）在命令行中输入“python - O - m py_compile hello. py”，然后按回车键：

```
python - O - m py_compile hello. py
```

- 参数“-O”表示生成优化代码。
- 参数“-m”表示把导入的 py_ compile 模块作为脚本运行。编译 hello. pyo 需要调用 py_ compile 模块中的 compile( )方法。
- 参数“hello. py”是待编译的文件名。

查看 hello. py 文件所在的目录可以发现，此时目录中生成了一个名为 hello. pyo 的文件。

2. 2　Python 的编码规则

Python 语言有自己独特的编码规则，包括命名规则、代码书写规则等。本节将详细介绍 Python 中常用的规则，并解释这些规则的原理和由来。

2. 2. 1　命名规则

Python 语言有一套自己的命名规则，用户也可以借鉴 Java 语言的命名规则以形成自己编码的规则。命名规则并不是规定，只是一种习惯用法。变量名的首字符必须是字母或下划线，首字符之外的字符可以由字母、数字或下划线组成，并且不能使用 Python 的保留字。下面介绍几种常见的规则。

1. 变量名、包名、模块名

变量名通常由字母和下划线组成，包名、模块名通常用小写字母，如【例 2-1】所示。

【例 2-1】演示变量名、包名、模块名的规范写法。

```
# 变量、模块名的命名规则
# Filename: ruleModule.py

_rule = "rule information"
```

- 第 1 行代码是程序代码的注释，关于 Python 语言的注释请参见 2. 2. 5 节。
- 第 2 行代码也是程序的注释，用于声明模块的名称，模块名用小写字母。也可以不指定模块名，以 py 后缀的文件就是一个模块。模块名就是文件名。
- 第 4 行代码定义了一个全局变量_rule。

2. 类名、对象名

类名首字母用大写，其他字母采用小写。对象名用小写字母。类的属性和方法名以对象作为前缀，对象通过操作符“.”访问属性和方法。类的私有变量、私有方法以两个下划线作为前缀。如【例 2-2】所示。

【例 2-2】演示类的定义和实例化的规范写法。

更多资源请访问稀酷客(www.ckook.com)

```
class Student:                                    # 类名用大写字母
__name = ""                                      # 私有实例变量前必须有两个下划线
def __init__(self, name):
    self.__name = name                           # self 相当于 Java 中的 this
def getName(self):                               # 方法名首字母用小写,其后每个单词的首字母用大写
    return self.__name

if __name__ == "__main__":
 student = Student("borphi")                     # 对象名用小写字母
print student.getName()
```

- 第 1 行代码定义了一个名为 Student 的类，类名首字母大写。
- 第 2 行代码定义了一个私有的实例变量，变量名前有两个下划线。
- 第 4 行代码使用 self 前缀说明__name 变量属于 Student 类。
- 第 5 行代码定义了一个公有的方法，方法名首字母小写，其后的单词“Name”首字母大写。函数的命名规则与此相同。
- 第 9 行代码创建了一个 student 对象，对象名小写。

说明　关于面向对象的编程知识将在第 8 章详细介绍，读者此时只要知道类、对象、属性以及方法的书写方式即可。

3. 函数名

函数名通常采用小写，并用下划线或单词首字母大写来增加名称的可读性，导入的函数以模块名作为前缀。为了演示导入函数前缀的写法，下面使用了生成随机数的模块 random。该模块有一个函数 randrange( )。该函数可以根据给定的数字范围生成随机数。randrange( )的声明如下所示：

```
randrange(start, stop[, step])
```

- 参数 start 表示生成随机数所在范围的开始数字。
- 参数 stop 表示生成随机数所在范围的结束数字，但不包括数字 stop。
- 参数 step 表示从 start 开始往后的步数。生成的随机数在［start，stop － 1］的范围内，取值等于 start ＋ step。

例如：

```
randrange(1, 9, 2)
```

随机数的范围在 1、3、5、7 之间取得。

【例 2-3】 演示函数的规范写法，其中定义了一个函数 compareNum( )，该函数用于比较两个数字的大小，并返回对应的结果。

```
# 函数中的命名规则
import random
def compareNum(num1, num2):
    if(num1 > num2):
        return 1
    elif(num1 == num2):
        return 0
    else:
        return -1
num1 = random.randrange(1, 9)
num2 = random.randrange(1, 9)
print "num1 =", num1
print "num2 =", num2
print compareNum(num1, num2)
```

更多资源请访问稀酷客(www.ckook.com)

- 第 2 行代码导入了 random 模块。
- 第 4 行代码定义了一个函数 compareNum( )，参数 num1 、num2 为待比较的两个变量。
- 第 5 行到第 10 行代码比较两个数的大小，根据比较结果返回不同的结果。
- 第 11 行、第 12 行代码调用 random 模块的 randrange( )返回两个随机数。
- 第 13 行、第 14 行代码输出随机数，不同的机器、不同的执行时间得到的随机数均不相同。
- 第 15 行代码调用 compareNum( )，并把产生的两个随机数作为参数传入。

良好的命名习惯可以提高编程效率，使代码阅读者在不了解文档的情况下，也能理解代码的内容。

下面以变量的命名为例，说明如何定义有母实际含义的名称。许多程序员对变量的命名带有随意性，如使用 i、j、k 等单个字母。代码阅读者并不知道这些变量的实际含义，需要阅读文档或仔细查看源代码才能了解其含义。

【例 2-4】演示变量命名不规范的写法。

```
# 不规范的变量命名写法
sum = 0
i = 2000
j = 1200
sum = i + 12 * j
```

- 这段代码定义了一个求和变量 sum，以及两个变量 i、j。如果只看代码片段，并不知道该运算的含义是什么，需要通读整个函数或功能模块才能理解此处各个变量的含义。

【例 2-5】演示符合命名规则的写法。

```
# 规范的变量命名写法
sumPay = 0
bonusOfYear = 2000
monthPay = 1200
sumPay = bonusOfYear + 12 * monthPay
```

- bonusOfYear 表示年终奖金、monthPay 表示月薪，因此 sumPay 表示全年的薪水。使变量的名字有实际含义可以节省阅读程序的时间，能更快地理解程序的含义。

注意　变量的命名应尽可能的表达此变量的作用，尽量避免使用缩写，以便使任何人都能理解变量名的含义。不要过多的担心变量名的长度，长的变量名往往能更表达清楚该变量的作用。

以上讨论的命名方式同样适用于模块名、类名、方法名、属性名等。遵守命名规则会带来很多好处。统一的命名规则便于程序开发团队合作开发同一个项目；便于统一代码的风格，理解不同程序员编写的代码；命名规范的变量名可以使程序代码更容易被理解；避免项目中随意命名变量的情况，促进程序员之间的交流。规则并不是绝对的，统一规则、表达清楚名称的含义才是制定规则的目的。

2.2.2　代码缩进与冒号

代码缩进是指通过在每行代码前键入空格或制表符的方式，表示每行代码之间的层次关系。任何编程语言都需要代码缩进规范程序的结构，采用代码缩进的编程风格有利于代码的阅读和理解。对于 C、C++ 、Java 等语言，代码缩进只是作为编程的一种良好习惯而延承下来。对于 Python而言，代码缩进是一种语法，Python 语言中没有采用花括号或 begin... end... 分隔代码块，而是使用冒号和代码缩进来区分代码之间的层次。

使用 IDE 开发工具或 EditPlus 等编辑器书写代码时，编辑器会自动缩进代码、补齐冒号，提

更多资源请访问稀酷客(www.ckook.com)

高编码效率。

【例 2-6】演示代码中的条件语句采用代码缩进的语法。

```
# Filename: code.py
x = 1
if x == 1:
    print "x =", x                    # 代码缩进
else:
    print "x =", x                    # 代码缩进
    x = x + 1                         # 代码缩进
print "x =", x
```

- 第 2 行代码创建了变量 x，并赋值为 1。在赋值运算符的两侧各添加一个空格，这是一种良好的书写习惯，提高了程序的可读性。
- 第 3 行代码使用了条件语句 if，判断 x 的值是否等于 1。if 表达式后输入了一个冒号，冒号后面的代码块需要缩进编写。本行代码与第 1 行代码处于同一个层次，所以直接从最左端书写代码。
- 第 4 行代码表示 x 的值等于 1 时输出结果。当 if 条件成立时，程序才能执行到第 4 行，所以第 4 行代码位于第 3 行代码的下一个层次。在编码时，首先在最左端输入 4 个空格或制表键，然后再书写 print 语句。输出结果：

```
x = 1
```

- 第 5 行代码的 else 保留字后是一段新的代码块。当 x 的值不等于 1 时，程序将执行第 6、第 7 行代码。
- 第 6 行、第 7 行代码采用缩进式的代码风格。
- 第 8 行代码输出结果：

```
x = 1
```

python 对代码缩进的要求非常严格。如果程序中不采用代码缩进的编码风格，将抛出一个 IndentationError 异常。

【例 2-7】演示错误的缩进方式。

```
x = 0
if x == 1:
print "x =", x
else:
    print "x =", x                    # 代码缩进
    x = x + 1                         # 代码缩进
print "x =", x
```

- 第 3 行没有缩进代码，python 不能识别出代码的层次关系，python 误认为 if x == 1：语句后面没有代码块。代码运行后输出如下错误信息：

```
IndentationError: expected an indented block
```

注意　如果缩进的代码前只有一个空格或几个制表符也是符合语法要求的，但是不推荐使用这种写法。最佳的方式是编码前统一代码的书写规则，所有代码前的空格数保持一致，最好使用 4 个空格缩进。

每行代码缩进的情况不同，代码执行的结果也不同。例如，在【例 2-6】中，else 子句包含的两条语句都采用了代码缩进的格式，这两条语句组成了一个代码块。

【例 2-8】把【例 2-6】中的代码修改为如下的内容，则变量 x 的值将有所变化。

更多资源请访问稀酷客(www.ckook.com)

```
x = 1
if x == 1:
    print "x =", x          #代码缩进
else:
    print "x =", x          #代码缩进
x = x + 1
print "x =", x
```

- 第 6 行代码与第 1 行代码处于同一个层次，是主程序必须执行的的代码，因此变量 x 的值为 2。
- 第 7 行代码的输出结果：

```
x = 2
```

注意　当程序出现问题时，程序员首先要检查代码的书写格式，看是否因为代码缩进的问题导致了不期望的计算结果。

2.2.3　模块导入的规范

模块是类或函数的集合，用于实现某个功能。模块的导入和 Java 中包的导入的概念很相似，都使用 import 语句。在 Python 中，如果需要在程序中调用标准库或其他第三方库的类时，需要先使用 import 或 from... import... 语句导入相关的模块。

1. import 语句

【例 2-9】演示使用 import 语句导入 sys 模块，并打印相关内容的方法。

```
#规范导入方式
import sys
print sys.path
print sys.argv
```

- 第 2 行代码使用 import 语句导入了 sys 模块，sys 模块是处理系统环境的函数的集合。
- 第 4 行代码输出 Python 环境下的查找路径的集合，Python 默认情况下会查找 sys.path 返回的目录列表。列表是 Python 内置的数据结构，定义了一组数据，通常用来作为参数或返回值。关于列表的知识请参考第 4 章的内容。本行代码的输出结果：

```
['D:\\developer\\python\\example\\02\\2.2\\2.2.3','C:\\WINDOWS\\system32\\python25.
zip', 'D:\\developer\\python\\DLLs','D:\\developer \\python \\lib','D:\\developer\\python
\\lib\\plat-win',
    'D:\\developer\\python\\lib\\lib-tk','D:\\developer \\python',
    'D:\\developer\\python\\lib\\site-packages','D:\\developer\\python\\lib\\site-
packages\\win32','D:\\developer\\python\\l ib\\site-packages\\win32\\lib',
    'D:\\developer\\python\\lib\\site-packages\\Pythonwin',
    'D:\\developer\\python\\lib\\site-packages\\wx-2.8 -msw-unicode']
```

- 第 5 行代码的 sys.argv 是存储输入参数的列表。默认情况下，argv 自带的参数是文件名。输出结果如下：

```
['import.py']
```

2. from... import... 语句

首先解释一下命名空间的概念。命名空间是指标识符的上下文。相同名称的标识符可在多条命名空间中定义，命名空间中可以定义任何标识符，并且保证这些标识符不会与任何已有的标识符发生冲突。

使用 from... import... 语句导入与 import 语句导入有所不同，区别是前者只导入模块中的一部

更多资源请访问稀酷客(www.ckook.com)

分内容，并在当前的命名空间中创建导入对象的引用；而后者在当前程序的命名空间中创建导入模块的引用，从而可以使用“类名．属性”的方式调用。例如 import sys 后，可以在当前文件中调用 sys. path。

【例 2-10】使用 from... import... 语句导入指定内容。实现【例 2-9】中代码相同的功能。

```
#不规范导入方式
from sys import path
from sys import argv
print path
print argv
```

第 5 行、第 6 行代码直接调用 path、argv 列表的内容，没有模块名的限定，这种写法不够规范。如果程序比较复杂，导入了很多模块，阅读程序的人并不了解 path、argv 来自哪个模块。而 sys. path、sys. argv 的写法可以清楚地知道 path、argv 来自 sys 模块。

2. 2. 4　使用空行分隔代码

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

【例 2-11】创建一个类 A，并在类 A 中定义 funX( )和 funY( )这两个方法。

```
class A:
    def funX(self):
        print "funY()"
    def funY(self):
        print "funY()"
if __name__ == "__main__":
    a = A()
    a.funX()
    a.funY()
```

- 第 4 行代码插入了一个空行，便于程序员阅读代码，区分方法之间的间隔。
- 第 7 行代码也是一个空行。因为下面的 if 语句是主程序的入口，用于创建类 A 的对象，并调用其方法。

空行与代码缩进不同，空行并不是 Python 语法的一部分。书写时不插入空行，Python 解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。记住：空行也是程序代码的一部分。

2. 2. 5　正确的注释

注释是用于说明代码实现的功能、采用的算法、代码的编写者以及代码创建和修改的时间等信息。注释是代码的一部分，注释起到了对代码补充说明的作用。C、C++ 、Java 均采用“//”或“/**/”作为注释的标记，Python 的注释方式有所不同。如果只对一行代码注释，使用“#”加若干空格开始，后面是注释的内容，以回车作为注释的结束。如果对一段代码进行注释，也使用“#”，段落之间以一个“#”行分隔。Python 一般会忽略“#”行的内容，跳过“#”行执行后面的内容。特殊含义的注释例外，读者可以参考【例 4-3】中的代码说明。下面这段代码演示了 Python 注释的用法。

【例 2-12】演示 Python 注释的使用方法。

```
# note - show Python's note
# Copyright (C) 2008 bigmarten
#
# This program is free software. you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation
```

更多资源请访问稀酷客(www.ckook.com)

```
#
#############################################################################
#
# Version is 1.0
#
# Its contents are calculate payment。
#
#############################################################################
#规范的变量命名
sumPay = 0                                    # 年薪
bonusOfYear = 2000                            #年终奖金
monthPay = 1200                               #月薪
sumPay = bonusOfYear + 12 * monthPay          # 年薪 =年终奖金 + 12 * 月薪
```

- 第 1 行代码对本文件进行摘要说明。
- 第 2 行代码声明了版权信息。
- 第 3 行代码是个空行，说明下面将另起一段，注释其他的内容。
- 第 4 行到第 6 行说明程序的许可信息。
- 第 8 行到第 14 行代码说明程序的版本和实现的功能。
- 第 16 行开始的代码实现程序的功能，并对每行代码进行单行注释。

Python 还有一些特殊的注释，以完成一些特别的功能，如中文注释、程序的跨平台等。

（1）中文注释：如果需要在代码中使用中文注释，必须在 Python 文件的最前面加上如下注释说明。

```
# -* - coding: UTF - 8 - * -
```

（2）跨平台注释：如果需要使 Python 程序运行在 windows 以外的平台，需要在 Python 文件的最前面加上如下注释说明。

```
#! /usr/bin/python
```

此外，注释也用于调试程序。由于文本编辑器不具备调试功能，因此可以使用注释辅助调试。如果一个程序很长，可以把程序分成若干个部分编写。为了更好的调试目前正在编写的程序模块，可以将那些已经编译通过的部分注释掉，或者把多余代码注释掉，以便把主要精力集中在当前编写的逻辑上。

【例 2-13】编写一个比较两个数字大小的函数。

```
def compare Num(num1, num2):
    if(num1 > num2):
        return str(num1) + " > " + str(num2)
    elif(num1 < num2):
        return str(num1) + " = " + str(num2)
    elif(num1 == num2):
        return str(num1) + " = " + str(num2)
    else:
        return ""
```

本段代码中的 str( )实现了数字类型到字符串类型的转换。

编译后显示如下内容：

```
------python------
输出完成(耗时：1 秒) - 正常终止
```

说明这个函数编译通过，至少在语法上没有任何错误。

更多资源请访问稀酷客(www.ckook.com)

【例 2-14】接上例。编写【例 2-13】中函数的调用程序，以证明【例 2-13】中程序的逻辑是否正确。在【例 2-13】的代码段后面添加如下代码。

```
num1 = 2
num2 = 1
print compareNum(num1, num2)
num1 = 2
num2 = 2
print compareNum(num1, num2)
num1 = 1
num2 = 2
print compareNum(num1, num2)
```

运行程序，发现第 3 行的输出结果有误：

```
2 > 1
2 = 2
1 = 2
```

第 1 行和第 2 行的输出结果证明函数 compareNum( ) 中 num1 > num2 和 num1 == num2 的判断是正确的，于是想到程序可能是 num1 < num2 的条件判断的逻辑出现了错误。为了证明这一点，注释掉 num1 < num2 的分支判断语句。

【例 2-15】接【例 2-13】。注释后的 compareNum( ) 的代码。

```
def compareNum(num1, num2):
    if(num1 > num2):
        return str(num1) + " > " + str(num2)
    # elif(num1 < num2):
    #     return str(num1) + " = " + str(num2)
    elif(num1 == num2):
        return str(num1) + " = " + str(num2)
    else:
        return ""
```

此时，程序编译通过，证明逻辑错误就在 num1 < num2 的分支语句中。仔细检查注释的语句，发现 return 语句的返回值表达式写错了，其中的“ < ”误写为了“ = ”。

【例 2-16】接【例 2-13】，将【例 2-13】中的 compareNum( ) 改为正确的写法。

```
def compareNum(num1, num2):
    if(num1 > num2):
        return str(num1) + " > " + str(num2)
    elif(num1 < num2):
        return str(num1) + " < " + str(num2)
    elif(num1 == num2):
        return str(num1) + " = " + str(num2)
    else:
        return ""
```

再次运行整个程序，输出正确结果：

```
2 > 1
2 = 2
1 < 2
```

合理的注释方法可以检查程序中的错误。这段代码演示了注释问题语句的做法，并检查这些语句错误的过程。另一种用法是，将正确语句加上注释，单独运行可能有问题的语句，检查语句错误。这两种用法可以根据实际情况分别应用，后者更适合于代码比较长的情况。注释对于程序非常重要，表 2-1 说明了注释的用法和作用。

更多资源请访问稀酷客(www.ckook.com)

表 2-1　注释的用法及其作用

注释的用法	作　用
单行注释	说明一行语句的作用
块注释	说明一段代码的作用或整个程序文件的功能
程序的跨平台	#! /usr/bin/python
中文注释的支持	# -* - coding: UTF-8 -* -
调试程序	注释可以帮助程序员调试程序

2.2.6　语句的分隔

在 C、Java 等语言的语法中规定，必须以分号作为语句结束的标识。Python 也支持分号，同样用于一条语句的结束标识。但在 Python 中分号的作用已经不像 C、Java 中那么重要了，Python 中的分号可以省略，主要通过换行来识别语句的结束。

例如，以下两行代码是等价的：

```
print "hello world!"
print "hello world!";
```

❑　第 1 行代码的输出结果：

```
hello world!
```

❑　第 2 行代码的输出结果：

```
hello world!
```

如果要在一行中书写多条句，就必须使用分号分隔每个语句，否则 Python 无法识别语句之间的间隔：

```
#使用分号分隔语句
x=1; y=1 ; z=1
```

第 2 行代码有 3 条赋值语句，语句之间需要用分号隔开。如果不隔开语句，Python 解释器将不能正确解释，提示语法错误：

```
SyntaxError: invalid syntax
```

注意　分号不是 Python 推荐使用的符号，Python 倾向于使用换行符作为每条语句的分隔，简单直白是 Python 语法的特点。通常一行只写一条语句，这样便于阅读和理解程序。一行写多条语句的方式是不好的习惯。

Python 同样支持多行写一条语句，Python 使用“ \ ”作为换行符。在实践中，一条语句写在多行也是非常常见的。

【例 2-17】把 SQL 语句作为参数传递给函数，由于 SQL 的语句一般非常长，为了阅读方便，因此需要换行书写。

```
#字符串的换行
#写法一
sql = "select id,name \
from dept \
where name = 'A'"
print sql
#写法二
sql = "select id,name " \
"from dept " \
"where name = 'A'"
print sql
```

更多资源请访问稀酷客(www.ckook.com)

- 写法一只使用了一对双引号，把 SQL 语句分为 select、from、where 等 3 部分分别书写。
- 第 6 行代码输出结果：

```
select id,name from dept where name = 'A'
```

- 写法二使用了 3 对双引号，select、from、where 分别对应一对双引号。
- 第 11 行代码输出结果：

```
select id,name from dept where name = 'A'
```

第二种写法比第一种写法的可读性更强，可以使用空格和制表符对齐语句，使代码显得更工整。对于简短的语句不推荐换行的写法，这种写法只会造成阅读的复杂性。下面这段程序是不合理的换行写法：

```
# 一条语句写在多行
print \
"hello world!"
```

- 第 2 行～第 3 行代码是一个整体，调用 print 输出“hello world!”，这种情况不适合分行书写。

2.3　变量和常量

变量是计算机内存中的一块区域，变量可以存储规定范围内的值，而且值可以改变。常量是一块只读的内存区域，常量一旦初始化就不能修改。

2.3.1　变量的命名

变量名由字母、数字或下划线组成。变量名的首字符必须是字母或下划线，首字符之外的字符可以由字母、数字或下划线组成。例如：

```
# 正确的变量命名
var_1 = 1
print var_1
_var1 = 2
print _var1
```

- 第 2 行代码定义了一个名为 var_1 的变量，该变量的初始值为 1。这个变量以字母开头，后面的字符由字母、下划线和数字组成。
- 第 3 行代码输出结果：

```
1
```

- 第 4 行代码定义了一个名为_var1 的变量，该变量的初始值为 2。这个变量以下划线开头，后面的字符由字母和数字组成。
- 第 5 行代码输出结果：

```
2
```

【例 2-18】演示错误的变量命名方式。

```
# 错误的变量命名
1_var = 3
print 1_var
 $var = 4
print $var
```

更多资源请访问稀酷客(www.ckook.com)

- 第 2 行代码定义了一个名为 1_var 的变量，该变量以数字开头，后面的字符由字母、下划线组成。
- 第 3 行代码，变量以数字开头，不符合变量命名的规则。提示如下错误：

```
SyntaxError: invalid syntax
```

- 第 4 行代码定义了一个名为 $ var 的变量，该变量以 $ 符号开头。
- 第 5 行代码，变量以 $ 符号开头，不符合变量命名的规则。提示如下错误：

```
SyntaxError: invalid syntax
```

2.3.2　变量的赋值

Python 中的变量不需要声明，变量的赋值操作即是变量声明和定义的过程。每个变量在内存中创建，都包括变量的标识、名称和数据这些信息。例如：

```
x = 1
```

上面的代码创建了一个变量 x，并且赋值为 1，如图 2-1 所示。

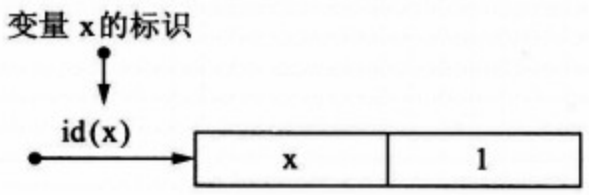


图 2-1　变量的内部结构

Python 中一次新的赋值，将创建一个新的变量。即使变量的名称相同，变量的标识并不相同。

【例 2-19】演示 Python 的变量声明以及赋值操作。

```
# 一次新的赋值操作,将创建一个新的变量
x = 1
print id(x)
x = 2
print id(x)
```

- 第 2 行代码定义了一个名为 x 的变量，该变量的初始值为 1。
- 第 3 行代码调用函数 id，输出变量 x 的标识。每个变量都有一个内部标识，便于变量的维护和管理。输出结果：

```
11229424
```

- 第 4 行代码再次定义了一个 x 的变量，该变量的初始值为 2。该变量与前面的变量 x 并不是同一个变量。
- 第 5 行代码输出变量 x 的标识：

```
11229412
```

如果变量没有赋值，Python 将认为该变量不存在。例如：

```
print y
```

运行后，解释器提示：

```
NameError: name 'y' is not defined
```

在变量 y 没有赋值的前提下，不能直接输出 y 的值。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。Python 支持对一些变量同时赋值的操作，例如：

更多资源请访问稀酷客(www.ckook.com)

```
# 给多个变量赋值
a = (1, 2, 3)
(x, y, z) = a
print "x =", x
print "y =", y
print "z =", z
```

- 第 2 行代码定义了一个序列 a，这个序列有 3 个值：1、2、3。
- 第 3 行代码，把序列 a 的值分别赋值给序列（x，y，z）中的变量 x、y、z。
- 第 4 行代码输出变量 x 的值。输出结果：

```
x = 1
```

- 第 5 行代码输出变量 y 的值。输出结果：

```
y = 2
```

- 第 6 行代码输出变量 z 的值。输出结果：

```
z = 3
```

通过序列的装包和拆包操作，实现了同时给多个变量赋值。关于序列的概念请参看第 4 章的内容。

2.3.3　局部变量

局部变量是只能在函数或代码段内使用的变量。函数或代码段一旦结束，局部变量的生命周期也将结束。局部变量的作用范围只在局部变量被创建的函数内有效。例如，如果文件 1 的 fun()中定义了一个局部变量，则该局部变量只能被 fun()访问。局部变量不能被 fun2()访问，也不能被文件 2 访问，如图 2-2 所示。

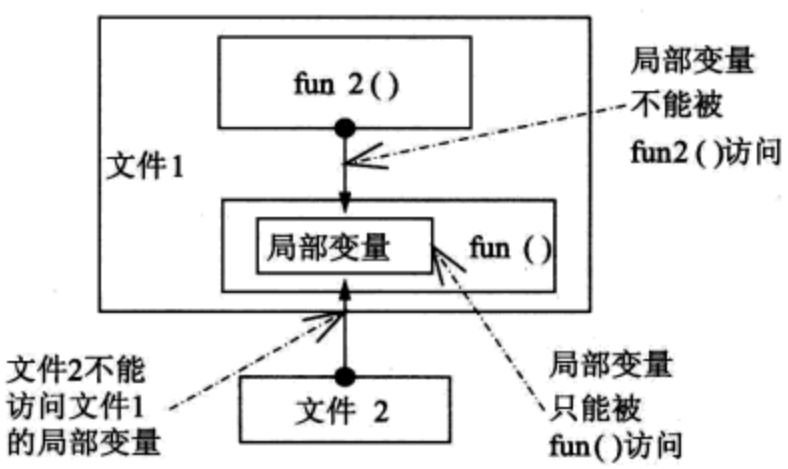


图 2-2　局部变量的作用范围

下面定义了一个函数 fun()，该函数中定义了一个局部变量：

```
# 局部变量
def fun():
    local = 1
    print local
local = local - 1
```

- 第 2 行代码定义了一个函数 fun()。
- 第 3 行代码定义了一个局部变量 local。
- 第 4 行代码输出 local 的值：

```
1
```

- 第 5 行代码执行算术运算。此时已超出 local 变量的作用范围。运行后会提示错误：name 'local' is not defined。

更多资源请访问稀酷客(www.ckook.com)

注意　Python 创建的变量就是一个对象，Python 会管理变量的生命周期。Python 对变量的回收采用的也是垃圾回收机制。

2.3.4　全局变量

全局变量是能够被不同的函数、类或文件共享的变量，在函数之外定义的变量都可以称为全局变量。全局变量可以被文件内部的任何函数和外部文件访问。例如，如果文件 1 中定义了一个全局变量，文件 1 中的函数 fun( )可以访问该全局变量。此外，该全局变量也能被文件 1 以外的文件访问，如图 2-3 所示。

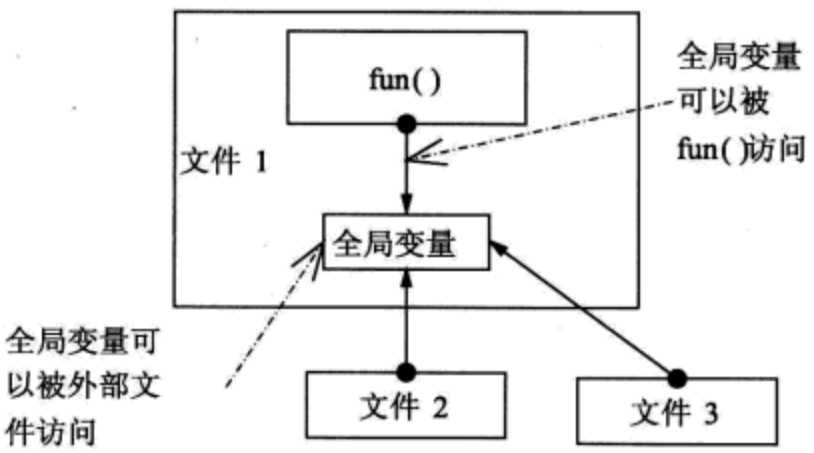


图 2-3　全局变量的作用范围

全局变量通常在文件的开始处定义。

【例 2-20】定义两个全局变量_a、_b 和两个函数 add( )、sub( )，这两个函数调用全局变量执行加法和减法计算。

```
# 在文件的开头定义全局变量
_a = 1
_b = 2
def add():
    global _a
    _a = 3
    return "_a + _b =", _a + _b
def sub():
    global _b
    _b = 4
    return "_a - _b =", _a - _b
print add()
print sub()
```

- 第 2 行代码定义了一个名为_a 的全局变量，这个变量的作用范围从定义处到文件的结尾。之所以使用下划线是为了区分局部变量，让程序员对全局变量的出现引起重视。
- 第 3 行代码定义了一个名为_b 的全局变量。同样，变量_b 的作用范围从定义处到文件的结尾。
- 第 4 行代码定义了一个函数 add( )，用于执行加法计算。
- 第 5 行代码引用全局变量_a。这里使用了 global 保留字，global 用于引用全局变量。
- 第 6 行代码对全局变量_a 重新赋值。
- 第 7 行代码返回_a + _b 的值。
- 第 8 行代码定义了一个函数 sub( )，用于执行减法运算。函数内的实现方式和 add( )相同。
- 第 12 代码调用函数 add( )。输出结果：

```
('_a + _b =', 5)
```

更多资源请访问稀酷客(www.ckook.com)

- 第 13 行代码调用函数 sub( )。输出结果：

```
('_a - _b =', -1)
```

如果不使用 gloabal 保留字引用全局变量，而直接对_a、_b 赋值，将得到不正确的结果：

```
# 错误的使用全局变量
_a = 1
_b = 2
def add():
    _a = 3
    return "_a + _b =", _a + _b
def sub():
    _b = 4
    return "_a - _b =", _a - _b
print add()
print sub()
```

- 第 5 行代码中的_a 并不是前面定义的全局变量，而是函数 add （） 中的局部变量。虽然输出的结果相同，但是运算的对象并不相同。
- 第 6 行代码中的_b 还是前面定义的全局变量_b。
- 第 8 行代码，这里的_b 是局部变量。
- 第 10 行代码的输出结果：

```
('_a + _b =', 5)
```

- 第 11 行代码的输出结果：

```
('_a - _b =', -3)
```

注意　变量名相同的两个变量可能并不是同一个变量，变量的名称只是起标识的作用。变量出现的位置不同，变量的含义也不同。

可以把全局变量放到一个专门的文件中，这样便于统一管理和修改。如创建一个名为 gl. py 的文件：

```
# 全局变量
_a = 1
_b = 2
```

pl. py 文件创建了两个全局变量_ a 和_ b。

再创建一个调用全局变量的文件 use_ global. py：

```
# 调用全局变量
import gl
def fun():
    print gl._a
    print gl._b
fun()
```

- 第 2 行代码导入前面创建的文件 gl. py，即模块 gl。
- 第 3 行代码定义了一个函数 fun( )，该函数调用全局变量_a 和_b。这里不需要使用 global 引用 gl. py 中的全局变量，因为前导符可以定位导全局变量_a 和_b。
- 第 4 行代码输出_a 的值，使用前导符 gl 定位。输出结果：

```
1
```

- 第 5 行代码输出_ b 的值，使用前导符 gl 定位。输出结果：

更多资源请访问稀酷客(www.ckook.com)

```
2
```

- 第6行代码调用fun()。

应该尽量避免使用全局变量。不同的模块都可以自由的访问全局变量，可能会导致全局变量的不可预知性。对于gl.py中的全局变量，如果程序员甲修改了_a的值，程序员乙同时也要使用_a，这时可能导致程序中的错误。这种错误是很难发现和更正的。

全局变量降低了函数或模块之间的通用性，不同的函数或模块都要依赖于全局变量。同样，全局变量降低了代码的可读性，阅读者可能并不知道调用的某个变量是全局变量。

2.3.5 常量

常量是指一旦初始化后就不能修改的固定值。例如，数字“5”、字符串“abc”都是常量。C++中使用const保留字指定常量，Java使用static和final保留字指定常量，而Python并没有提供定义常量的保留字。Python是一门功能强大的语言，可以自己定义一个常量类来实现常量的功能。

【例2-21】演示在《Python Cookbook》一书中定义的常量模块const。

```
class _const:
    class ConstError(TypeError): pass
    def __setattr__(self,name,value):
        if self.__dict__.has_key(name):
            raise self.ConstError, "Can't rebind const(%s)"% name
        self.__dict__[name] = value
import sys
sys.modules[__name__] = _const()
```

- 这个类定义了一个方法__setattr__()，和一个异常类型ConstError，ConstError类继承自TypeError。通过调用类自带的字典__dict__，判断定义的常量是否包含在字典中。如果字典中包含此常量，将抛出异常，否则，给新创建的常量赋值。
- 最后两行代码的作用是把const类注册到sys.modules这个全局字典中。

在use_const.py中调用const，定义常量：

```
import const
const.magic = 23
const.magic = 33
```

- 第1行代码导入了const模块。
- 第2行代码定义了一个常量magic。
- 第3行代码修改常量magic的值，抛出异常：

```
const.ConstError: Can't rebind const(magic)
```

2.4 数据类型

数据类型是构成编程语言语法的基础。不同的编程语言有不同的数据类型，但都具有常用的几种数据类型。Python有几种内置的数据类型——数字、字符串、元组、列表和字典。本节将重点介绍数字和字符串数据类型。

2.4.1 数字类型

Python的数字类型分为整型、长整型、浮点型、布尔型、复数类型。Python没有字符类型。使用Python编写程序时，不需要声明变量的类型，整型、长整型可以用二进制、八进制、十六

更多资源请访问稀酷客(www.ckook.com)

进制数。由于 Python 不需要显式的声明变量的类型，变量的类型由 Python 内部管理，在程序的后台实现数值与类型的关联，以及类型转换等操作。Python 与其他高级语言定义变量的方式及内部原理有很大的不同。在 C 或 Java 中，定义一个整型的变量，可以采用如下方式：

```
int i = 1;
```

在 Python 中，定义整型变量的表达方式更简练：

```
i = 1
```

在 Python 中，定义整型变量只需要采用赋值表达式即可，程序员不需要关心赋值变量的大小，Python 会根据值的大小自动转换为长整型。Python 其他数字类型变量的定义方法与此类似。

C 语言分为单精度和双精度浮点类型，而 Python 只有双精度浮点类型。Python 根据变量的值自动判断变量的类型，程序员不需要关心变量究竟是什么类型，只要知道创建的变量存放了一个数。以后的工作只是对这个数值进行操作，Python 会对这个数的生命周期负责。

更重要的一点是，C 或 Java 只是创建了一个 int 型的普通变量，而 Python 创建的是一个整型对象，并且 Python 自动完成了整型对象的创建工作，不再需要通过构造函数创建。Python 内部没有普通类型，任何类型都是对象。如果 C 或 Java 需要修改变量 i 的值，只要重新赋值即可，而 Python 并不能修改对象 i 的值。例如：

```
#下面的两个 i 并不是同一个对象
i = 1
print id(i)
i = 2
print id(i)
```

如果需要查看变量的类型，可以使用 Python 定义的 type 类。type 是__builtin__模块的一个类，该类能返回变量的类型或创建一个新的类型。__builtin__模块是 Python 的内联模块，内联模块不需要使用 import 语句，由 Python 解释器自动导入。后面还会接触到更多内联模块的类和函数。关于数据类型可用的函数，读者可参考 5.2.4 节的内容。

【例 2-22】演示返回各种变量的类型。

```
#整型
i =1
print type(i)
#长整型
l =9999999990
print type(l)
#浮点型
f =1.2
print type(f)
#布尔型
b =True
print type(b)
```

- 第 3 行代码输出结果：<type 'int'>。
- 第 6 行代码输出结果：<type 'long'>。
- 第 9 行代码输出结果：<type 'float'>。
- 第 12 行代码输出结果：<type 'bool'>。

Python 对于科学计算的应用也很方便，因为 Python 内置了复数类型（Java、C#等高级语言没有提供复数类型）：

```
#复数类型
c = 7 + 8j
print type(c)
```

更多资源请访问稀酷客(www.ckook.com)

- 第 3 行调用 type 函数，该函数用于返回任意对象的数据类型。代码输出结果：< type 'complex' > 。

注意　复数类型的写法与数学中的写法相同，如果写为 c = 7 + 8i，Python 不能识别其中的“i”，将提示语法错误。

2.4.2　字符串类型

在 Python 中有 3 种表示字符串类型的方式——单引号、双引号、三引号。单引号和双引号的作用是相同的，程序员可以根据自己的习惯使用。PHP 程序员可能更习惯使用单引号表示字符串，C、Java 程序员则习惯使用双引号表示字符串。

【例 2-23】演示单引号和双引号的作用是等价的。

```
# 单引号和双引号的作用是等价
str = "hello world!"
print str
str = 'hello world! '
print str
```

- 第 3 行代码输出结果：hello world!；第 5 行代码输出结果：hello world!。

三引号的用法是 Python 特有的语法，三引号中可以输入单引号、双引号或换行等字符：

```
# 三引号的用法
str = '''he say "hello world! "'''
print str
```

- 第 3 行代码的三引号中带有双引号，双引号也会被输出。输出结果：he say"hello world!" 。

三引号的另一种用法是制作文档字符串。Python 的每个对象都有一个属性__doc__，这个属性用于描述该对象的作用：

```
# 三引号制作 doc 文档
class Hello:
    """hello class"""
    def printHello():
        """print hello world"""
        print "hello world!"
print Hello.__doc__
print Hello.printHello.__doc__
```

- 第 2 行代码定义了一个名为 Hello 的类。
- 第 3 行代码是对 Hello 类的描述，该字符串将被存放在类的__doc__属性中。
- 第 4 行代码定义了一个方法 printHello( )。
- 第 5 行代码描述了 printHello( )，并把字符串存放在该函数的__doc__属性中。
- 第 6 行代码输出结果：hello world!。
- 第 7 行代码输出 Hello 的__doc__属性的内容。输出结果：hello class。
- 第 8 行代码输出 printHello( )的__doc__属性的内容。输出结果：print hello world。

如果要输出含有特殊字符（单引号、双引号等）的字符串，需要使用转义字符。Python 中转义字符为“\”，和 C、Java 中的转义字符符号相同。转义操作只要在特殊字符的前面加上“\”即可。下面这段代码说明了特殊字符的转义用法：

```
# 转义字符
str = 'he say:\'hello world! \"'
print str
```

更多资源请访问稀酷客(www.ckook.com)

- 第 2 行代码中的单引号是特殊字符，需要在“'”前加上转义字符。
- 第 3 行代码的输出结果：

```
he say:'hello world! '
```

使用双引号或三引号可以直接输出含有特殊字符的字符串，不需要使用转义符：

```
# 直接输出特殊字符
str = "he say:'hello world! '"
print str
str = '''he say:'hello world! ' '''
print str
```

- 第 2 行代码中使用了双引号表示字符串变量 str，因此 Python 能够识别出该双引号内部的单引号是作为输出的字符。
- 第 3 行代码的输出结果：

```
he say:'hello world! '
```

- 第 4 行代码使用三引号表示字符串变量 str，注意，最后一个单引号后面留有一个空格，这个空格是为了让 Python 识别出三引号留下的。如果不留下这个空格，4 个单引号连在一起，Python 解释器不能正确识别三引号，将提示如下错误：

```
SyntaxError: EOL while scanning single-quoted string
```

- 第 5 行代码的输出结果：

```
he say:'hello world! '
```

注意　输出的字符串中如果含有单引号，需要使用双引号表示字符串；输出的字符串中如果含有双引号，则需要使用单引号表示字符串。

2.5　运算符与表达式

Python 的运算符号包括赋值运算符、算术运算符、关系运算符和逻辑运算符。赋值运算符最简单，使用“=”表示。表达式是将不同类型的数据（包括常量、变量、函数）用运算符号按一定的规则连接起来的式子。表达式是学习编写 Python 语句的基础，本节将分别介绍 Python 中的各种表达式的使用方法。

2.5.1　算术运算符和算术表达式

算术运算符包括四则运算符、求模运算符和求幂运算符。Python 中的算术运算符和表达式如表 2-2 所示。

表 2-2　Python 中的算术运算符和表达式

算术运算符	算术表达式	说　明
+	x + y	加法运算
-	x - y	减法运算
*	x * y	乘法运算
/	x / y	除法运算
%	x % y	求模运算
* *	x 的 y 次方	求幂运算

更多资源请访问稀酷客(www.ckook.com)

注意 与C、Java语言不同，Python不支持自增运算符和自减运算符。例如，i++、i--是错误的语句，但是类似i+=1的语句是允许的。

【例2-24】演示Python中算术运算符的使用方法。

```
print "1 + 1 =", 1 + 1
print "2 - 1 =", 2 - 1
print "2 * 3 =", 2 * 3
print "4 / 2 =", 4 / 2
print "1 / 2 =", 1 / 2
print "1 / 2 =", 1.0 / 2.0
print "3 % 2 =", 3 % 2
print "2 * * 3 =", 2 * * 3
```

- 第1行代码的输出结果：1 + 1 = 2。
- 第2行代码的输出结果：2 - 1 = 1。
- 第3行代码的输出结果：2 * 3 = 6。
- 第4行代码的输出结果：4 / 2 = 2。
- 第5行代码输出的结果并不是期望的“0.5”。Python认为数字1和2是整型，整型相除后输出的结果将被截断，小数部分会被抛弃，因此输出值为“0”。输出结果：1 / 2 = 0。
- 第6行代码中的被除数是1.0，除数是2.0。Python把这两个数作为浮点型处理，因此相除后可以得到正确的结果。输出结果：1 / 2 = 0.5。
- 第7行代码，求模的值为3除以2后的余数。输出结果：3 % 2 = 1。
- 第8行代码的输出结果：2 * * 3 =8。

注意 不同版本的Python执行“1/2”算术表达式的结果略有不同，Python2.5之前的版本执行结果可能等于0.5。

如果要使“1/2”进行浮点数计算，需要在文件的最前面导入division模块。代码如下：

```
from __future__ import division
```

Python的算术表达式具有结合性和优先性。结合性是指表达式按照从左往右、先乘除后加减的原则。即从表达式的左边开始计算，先执行乘法和除法运算，再执行加法和减法运算。例如：

```
a + b * c % d
```

以上表达式先执行b * c，然后执行b * c % d，最后执行加法运算。

优先级是指先执行圆括号内的表达式，再按照结合性的原则进行计算。例如：

```
(a + b) * (c % d)
```

以上表达式先计算a + b的值，然后计算c % d的值，最后把两个值相乘。

【例2-25】演示算术运算的优先级。

```
# 算术运算的优先级
a = 1
b = 2
c = 3
d = 4
print "a + b * c % d =", a + b * c % d
print "(a + b) * (c % d) =", (a + b) * (c % d)
```

- 第6行代码的输出结果：

```
a + b * c % d = 3
```

- 第7行代码的输出结果：

更多资源请访问稀酷客(www.ckook.com)

```
(a + b) * (c % d) = 9
```

2.5.2　关系运算符和关系表达式

关系运算符是对两个对象进行比较的符号。Python 中的关系运算符和表达式如表 2-3 所示。

表 2-3　Python 中的关系运算符和表达式

关系运算符	关系表达式	说　明
<	x < y	小于
>	x > y	大于
<=	x <= y	小于等于
> =	x > = y	大于等于
==	x == y	等于
! =或< >	x ! = y或x < > y	不等于

【例 2-26】演示关系表达式的逻辑输出。

```
#关系表达式
print 2 > 1
print 1 <= 2
print 1 == 2
print 1 ! = 2
print 1 < > 2
```

❑　第 2 行代码，2 > 1 的逻辑关系成立。输出结果：

```
True
```

❑　第 3 行代码，1 <= 2 的逻辑关系成立。输出结果：

```
True
```

❑　第 4 行代码，1 == 2 的逻辑关系不成立。输出结果：

```
False
```

❑　第 5 行代码，1 ! = 2 的逻辑关系成立。输出结果：

```
True
```

❑　第 7 行代码，1 < > 2 的逻辑关系成立。输出结果：

```
True
```

不同的关系运算符优先级别不同。其中 < 、<= 、 > 、 > = 这 4 个运算符的优先级别相同， == 、! = 的优先级别相同。而 < 、<= 、 > 、 > = 的优先级别大于 == 、! = 的优先级别。例如：a > = b == c 等价于（a > = b）== c。

关系运算符的优先级低于算术运算符。

【例 2-27】演示关系运算符的优先级别。

```
#关系表达式的优先级别
print "1 + 2 < 3 - 1 =>", 1 + 2, "<", 3 - 1, "=>", 1 + 2 < 3 - 1
print "1 + 2 <= 3 > 5 % 2 =>", 1 + 2, "<=", 3, ">", 5 % 2, "=>", 1 + 2 <= 3 > 5 % 2
```

❑　第 2 行代码，先执行 1 + 2 = 3，然后执行 3 - 1 = 2，最后比较 3 < 2。输出结果：

```
1 + 2 < 3 - 1 => 3 < 2 => False
```

❑　第 3 行代码，先执行 1 + 2 = 3，然后执行 5 % 2 = 1，最后比较 3 <= 3 > 1。输出结果：

更多资源请访问稀酷客(www.ckook.com)

```
1 + 2 <= 3 > 5 % 2 => 3 <= 3 > 1 => True
```

2.5.3　逻辑运算符和逻辑表达式

逻辑表达式是用逻辑运算符和变量连接起来的式子。任何语言的逻辑运算符都一般分为 3 种——逻辑与、逻辑或和逻辑非。C、Java 语言的逻辑运算符用 &&、‖、! 表示，Python 采用 and、or、not。表 2-4 列出了 Python 中的逻辑运算符和表达式。

表 2-4　Python 中的逻辑运算符和表达式

逻辑运算符	逻辑表达式	说　明
And	x and y	逻辑与，当 x 为 Ture 时，才计算 y
Or	x || y	逻辑或，当 x 为 False 时，才计算 y
Not	not x	逻辑非

【例 2-28】演示逻辑表达式的运算。

```
# 逻辑运算符
print not True
print False and True
print True and False
print True or False
```

- 第 2 行代码，True 的逻辑非为 False。输出结果：False。
- 第 3 行代码，检测到 and 运算符左侧的 False，就直接返回 False。输出结果：False。
- 第 4 行代码，检测到 and 运算符左侧为 True，然后继续检测右侧，右侧的值为 False，于是返回 False。输出结果：False。
- 第 5 行代码，or 运算符的左侧为 True，于是返回 True。输出结果：True。

逻辑非的优先级大于逻辑与和逻辑或的优先级，而逻辑与和逻辑或的优先级相等。逻辑运算符的优先级低于关系运算符，必须先计算关系运算符，然后再计算逻辑运算符。

【例 2-29】演示逻辑运算符、关系运算符和算术运算符的优先级别。

```
# 逻辑表达式的优先级别
print "not 1 and 0 =>", not 1 and 0
print "not (1 and 0) =>", not (1 and 0)
print "(1 <= 2) and False or True =>", (1 <= 2) and False or True
print "(1 <= 2) or 1 > 1 + 2 =>", 1 <= 2, "or", 1 > 2, " =>", (1 <= 2) or (1 < 2)
```

- 第 2 行代码，先执行 not 1，再执行 and 运算。输出结果：not 1 and 0 => False。
- 第 3 行代码，先执行括号内的 1 and 0，再执行 not 运算。输出结果：not（1 and 0）=> True。
- 第 4 行代码，先执行 1 <= 2 的关系运算表达式，再执行 and 运算，最后执行 or 运算。输出结果：（1 <= 2）and False or True => True
- 第 5 行代码，先执行 1 <= 2 的关系运算表达式，再执行表达式 1 > 1 + 2，最后执行 or 运算。输出结果：（1 <= 2）or 1 > 1 + 2 => True or False => True。

2.5.4　运算符的优先级

Python 运算符在同一个表达式中具有不同的优先级。算术运算符的优先级大于关系运算符的优先级，关系运算符的优先级大于逻辑运算符的优先级。如果表达式中包含多种类型的运算符，Python 会根据运算符的优先级从高到低进行计算。表 2-5 从最低的优先级到最高的优先级依次列出了 Python 所有的运算符。Python 会首先计算表 2-5 中较下面的运算符，然后再计算表上

更多资源请访问稀酷客(www.ckook.com)

部的运算符。表 2-5 引用自《Python 参考手册》。

表 2-5　Python 运算符说明

运　算　符	说　明
lambda	Lambda 表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in，not in	成员测试
is，is not	同一性测试
<，<=，>，>=,!=，==	比较
|	按位或
^	按位异或
&	按位与
<<，>>	移位
+，-	加法与减法
*，/,%	乘法、除法与取余
+x，-x	正负号
~x	按位翻转
**	指数
x. attribute	属性参考
x［index］	下标
x［index：index］	寻址段
f（arguments...）	函数调用
（experession，...）	绑定或元组显示
［expression，...］	列表显示
{key：datum，...}	字典显示
‘expression，...’	字符串转换

为了使代码具有更好的可读性，最好使用圆括号来分组表达式。例如，（x > y）&&（! z）。显而易见，先计算两个圆括号中的表达式，然后再计算 && 操作。

2.6　小结

本章讲解了 Python 的基本语法和基本概念，包括 Python 的文件类型、编码规则、变量、数据类型以及运算符和表达式等内容。重点讲解了 Python 的编码规则，命名规则、缩进式的写法、注释等。Python 中的变量与 C、Java 有很大的不同，本章通过比较的方法阐述了 Python 变量的分类和特性。最后讲解了运算符和表达式的知识，这些知识是编写 Python 代码的基础。

更多资源请访问稀酷客(www.ckook.com)

第 3 章　Python 的控制语句

本章将介绍 Python 中控制语句的使用方法。控制语句包括条件语句和循环语句。控制语句根据条件表达式控制程序的执行。第 2 章中的一些例子使用到了条件语句，读者应该有个大概的印象。本章将进一步讨论控制语句的概念以及结构化编程的知识。本章的知识点：

- if 语句
- if... else... 语句
- while 循环
- for 循环
- 中断语句
- 结构化编程

3.1　结构化程序设计

结构化程序设计是以模块化设计为中心，将待开发的软件系统划分为若干个相互独立的模块，这样使完成每一个模块的工作变得明确，为设计一些较大的软件打下良好的基础。要完成一项工作任务，需要先设计，然后再对设计具体实现。例如，施工图纸就是一个设计，工程师制作图纸的过程就是设计的过程，工人根据图纸施工的过程就是实现的过程。程序设计也是这样，首先需要明确需要完成的目标，确定要做的步骤，然后再根据每个步骤去编写代码。

现实世界的事物是复杂的，为了方便描述客观世界中的问题的处理步骤，可以以图形的方式来表达。程序流程图就是程序员用于设计的利器，程序流程图可以描述每个任务的要求以及实现步骤，程序流程图对任何编程语言都是通用的。图 3-1 描述了判断某个数字是属于正数、负数或零的流程。

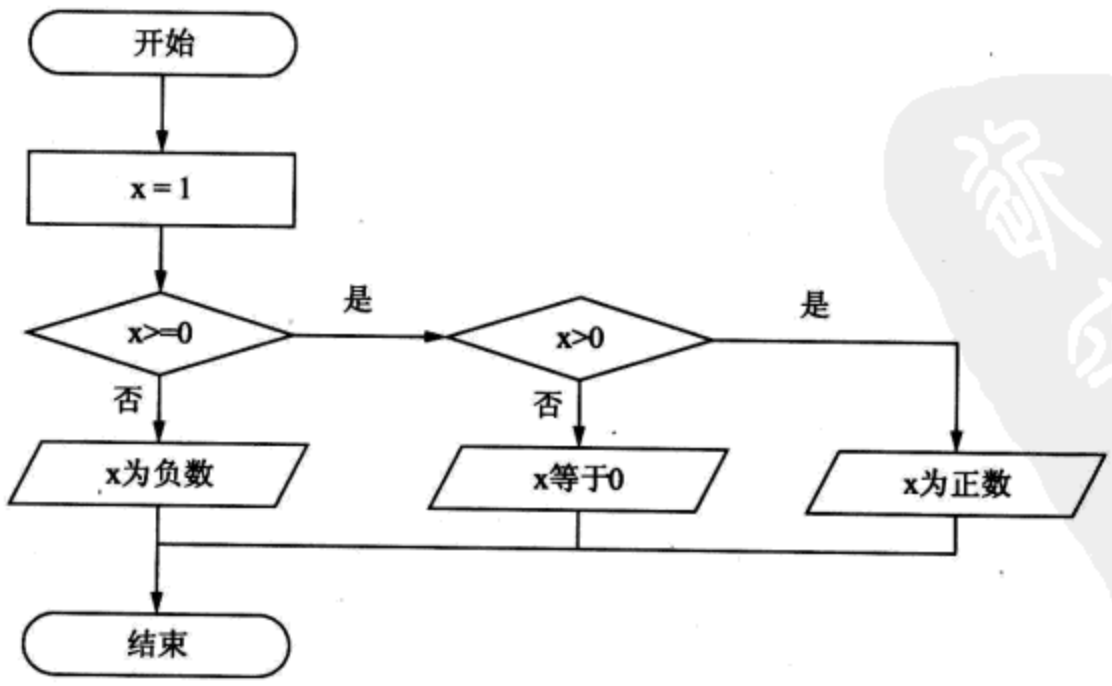


图 3-1　流程图示例

更多资源请访问稀酷客(www.ckook.com)

如图3-1 所示，首先输入 x 的值，然后判断 x 是否大于等于0。如果大于0，则执行 x > 0 的分支流程，否则，输出“x 为数字为负数”。对于 x > 0 的分支，如果 x 大于 0，输出“x 为正数”，否则，输出“x 等于 0”。

结构化程序设计提倡结构的清晰、设计的规范。结构化程序设计的主要方法是——自顶向下、逐步细化。把需要解决的问题分成若干个任务来完成，再对每个任务进行设计，逐步细化。例如，房屋的装修。首先是确定装修方案，以及装修任务（水电施工、水泥工程、家具施工等）。然后再对每个任务进行细分，确定子任务以及任务之间的施工顺序。确定好方案后，就可以具体实施了。实施的过程就是逐个完成子任务的过程。对于具体实现需要编写结构化的程序来完成，结构化程序设计分为 3 种结构——顺序结构、判断结构、循环结构。顺序结构非常简单，只有一条流程。下面将讨论另外两种结构的设计和实现。

3.2　条件语句

条件语句是指根据条件表达式的不同计算结果，使程序执行到不同的代码块。Python 中的条件语句有 if 语句和 if else 语句。

3.2.1　if 语句

if 语句用于检测某个条件是否成立。如果成立，则执行 if 语句内的代码，否则跳过 if 语句，执行后面的代码。if 语句的格式如下所示：

```
if(表达式):
语句1
else:
语句2
```

if 语句的执行过程如下：如果表达式的布尔值为真，则执行语句 1，否则，执行语句 2。其中的 else 子句可以省略，表达式两侧的括号也可以省略。

在讲解 if 语句之前，先介绍一下 Python 中的控制台输入函数。C 语言中使用 scanf( ) 和 getchar( ) 捕获用户输入，而 Java 语言的 System. in 包提供了控制台输入的方法。Python 也提供了类似功能的函数：input( ) 和 raw_input( )。raw_input （） 捕获用户的原始输入。raw_input( ) 函数的声明如下所示。

```
raw_input([prompt]) -> string
```

参数 prompt 是控制台中输出的提示文字，提示用户输入。返回值为字符串。如果输入的是数字，返回的还是字符串，使用前需要调用 int( ) 转换一下。下面这段代码说明了字符串和数字类型的转换：

```
x = raw_input("x:")
x = int(x)
x = x + 1
```

如果不调用 int( ) 把字符串转换为数字，而直接计算表达式 x = x + 1，将提示如下错误。

```
TypeError: cannot concatenate'str' and'int' object
```

input( ) 支持用户输入数字或表达式，不支持输入字符串。input( ) 函数的声明如下所示：

```
input([prompt]) -> value
```

参数 prompt 是控制台中输出的提示文字，提示用户输入。返回数字类型的值。例如：

更多资源请访问稀酷客(www.ckook.com)

```
x = input("x:")
print x
x = raw_input("x:")
print x
```

- 第1行代码，如果输入的是表达式1 + 1，input()将计算表达式的结果并返回。
- 第2行代码输出结果：2。
- 第3行代码，同样输入表达式1 + 1，raw_input()把输入的表达式作为字符串捕获。
- 第4行代码输出结果：1 + 1。

【例3-1】演示if语句的执行流程。

```
#执行if语句内的程序
a = input("a:")
b = input("b:")
if(a > b):
    print a, " > ", b
```

- 第2行代码定义了变量a。
- 第3行代码定义了变量b。
- 第4行代码判断变量a、b是否相等。
- 第5行代码，假设a=2、b=1。输出结果：2>1。

【例3-2】演示如果不满足if语句内的条件，程序将跳过if语句，执行后面的代码。

```
#跳过if语句
a = input("a:")
b = input("b:")
if(a > b):
    print a, " > ", b
print a, " < ", b
```

- 第4行代码中变量a的值小于变量b的值。因此，程序跳转到第6行执行。
- 第6行代码，假设a=1、b=2。输出结果：1<2。

【例3-3】接上例。将【例3-2】的代码改写为if else结构的语句。

```
#if else语句
a = input("a:")
b = input("b:")
if(a > b):
    print a, " > ", b
else:
    print a, " < ", b
```

- 第4行代码中变量a的值小于变量b的值。因此，程序跳转到else子句。
- 第7行代码，假设a=1、b=2。输出结果：1<2。

注意 else子句后需要加一个冒号，使Python解释器能识别出else子句对应的代码块。Java程序员可能不习惯这种语法，往往会忽略else子句后的冒号。

3.2.2 if... elif... else语句

if... elif... else语句是对if... else...语句的补充。当程序的条件分支很多时，可以使用这种语句。if... elif... else语句相当于C、Java中的if... elseif..... else语句。该语句的格式如下所示：

```
if(表达式1): 语句1
elif(表达式2): 语句2
```

更多资源请访问稀酷客(www.ckook.com)

```
...
elif(表达式 n): 语句 n
else: 语句 m
```

if... elif... else 语句的执行过程：首先判断表达式 1 的值是否为真。如果为真，则执行语句 1，否则，程序执行到 elif 子句，判断表达式 2 的值是否为真。如果表达式 2 的值为真，则执行语句 2，否则，程序进入下面一个 elif 子句，依次类推。如果所有的表达式都不成立，程序执行 else 子句的代码。其中的 else 子句可以省略，表达式两侧的括号也可以省略。

【例 3-4】通过判断学生的分数，以确定学生获得的成绩等级。

```
# if elif else 语句
score = input("score:")
if(score > = 90) and (score <= 100):
    print "A"
elif(score > = 80) and (score < 90):
    print "B"
elif(score > = 60) and (score < 80):
    print "C"
else:
    print "D"
```

- 第 2 行代码定义了一个变量 score，这个变量表示学生的分数。假设输入的值为“70”。
- 第 3 行代码，分数大于等于 90 并且小于等于 100，则等级评定为“A”。
- 第 5 行代码，分数大于等于 80 并且小于 90，则等级评定为“B”。
- 第 8 行代码，由于第 7 行代码中的条件表达式成立，因此程序跳转到第 8 行。输出结果：C。
- 第 9 行代码，当前面的条件表达式都不成立时，程序流转到 else 子句。

3.2.3　if 语句的嵌套

if 语句的嵌套是指在 if 语句中可以包含一个或多个 if 语句。嵌套的格式如下所示：

```
If(表达式 1):
    if(表达式 2): 语句 1
    elif(表达式 3): 语句 2
    ...
else: 语句 3
elif(表达式 n):
    ...
else:
...
```

【例 3-5】演示嵌套的条件语句。如果 x 的值大于 0，则 y 的值等于 1；如果 x 的值等于 0，则 y 的值等于 0；如果 x 的值小于 0，则 y 的值等于 -1。

```
x = -1
y = 99
if(x > = 0):
    if(x > 0):              #嵌套的 if 语句
        y = 1
    else:
        y = 0
else:
    y = -1
print "y =", y
```

- 第 2 行代码定义了一个变量 y。为了不和最终可能的输出结果 1、0、-1 重复，设置其初始值为 99。
- 第 3 行代码判断变量 x 的值。如果大于等于 0，则执行下面嵌套的 if 语句。

更多资源请访问稀酷客(www.ckook.com)

- 第4行代码判断x的值是否大于0。如果大于0，则执行第5行代码，否则执行第7行代码。
- 第8行代码，如果变量x的值小于0，则执行第9行代码。
- 第9行代码，由于变量x的值为-1，因此y的值等于-1。
- 第10行代码的输出结果：-1。

嵌套语句可以组合出很多写法，但是要注意把所有的分支情况都考虑到。下面的这种写法是错误的：

```
# 错误的嵌套语句
x = -1
y = 99
if(x < > 0):
    if(x > 0):          #嵌套的 if 语句
        y = 1
else:
    y = 0
print "y =", y
```

- 第4行代码判断变量x的是否等于0。如果不等于0，则执行if语句下面的代码块，否则执行else子句的代码。由于x的值等于-1，因此程序跳转到第5行。
- 第5行代码判断变量x的值是否大于0。如果大于0，则变量y的值设置为1。由于这里没有考虑变量x小于0的情况，所以程序直接跳转到第9行。
- 第9行代码，变量y的值并没有被改变，程序的分支结构没有考虑到x小于0的情况，所以最终输出的不是期望中的结果。输出结果：99。

注意 编写条件语句时，应该尽量避免使用嵌套语句。嵌套语句不便于阅读，而且可能会忽略一些可能性。

3.2.4 实现switch语句的功能

switch语句用于编写多分支结构的程序，类似于if... elif... else语句。C语言中switch语句的结构如下所示：

```
switch(表达式) {
case 常量表达式1: 语句1
case 常量表达式2: 语句2
...
case 常量表达式n: 语句n
default: 语句m
}
```

switch语句表示的分支结构比if... elif... else语句更清晰，代码可读性更高。但是Python并没有提供switch语句，Python可以通过字典实现switch语句的功能。

实现方法分为两步。首先，定义一个字典。字典是由键值对组成的集合（字典的使用方法详见第4章）。其次，调用字典的get()获取相应的表达式。

【例3-6】通过算术运算的符号，获取算术运算表达式。

```
# 使用字典实现 switch 语句
from __future__ import division
x = 1
y = 2
operator = "/"
result = {
```

更多资源请访问稀酷客(www.ckook.com)

```
    "+" : x + y,
    "-" : x - y,
    "* " : x * y,
    "/" : x / y
}
print result.get(operator)
```

- 第 3、4 行代码定义了两个操作数 x、y。
- 第 5 行代码定义了运算符变量 operator，该变量用于存放算术运算符。
- 第 6 行代码定义了一个字典 result。该字典的 key 值由“+”、“-”、“*”、“/”四则运算符组成。value 值由对应的算术表达式组成。
- 第 12 行代码调用 get( ) 方法，get( ) 的参数即变量 operator 的值。由于 operator 的值为“/”，因此将执行除法运算。输出结果：0.5。

另一种使用 switch 分支语句的方案是创建一个 switch 类，处理程序的流程。这种实现方法比较复杂，需要涉及到面向对象、for 循环、中断语句、遍历等知识，实现步骤分为如下的 4 步。

（1）创建一个 switch 类，该类继承自 Python 的祖先类 object。调用构造函数 init ( ) 初始化需要匹配的字符串，并需要定义两个成员变量 value 和 fall。value 用于存放需要匹配的字符串，fall 用于记录是否匹配成功，初始值为 false，表示匹配不成功。如果匹配成功，程序往后执行。

（2）定义一个 match( ) 方法，该方法用于匹配 case 子句。这里需要考虑 3 种情况：首先是匹配成功的情况，其次是匹配失败的默认 case 子句，最后是 case 子句中没有使用 break 中断的情况。

（3）重写__iter__( ) 方法，定义了该方法后才能使 switch 类用于循环语句中。__iter__( ) 调用 match( ) 方法进行匹配。通过 yield 保留字，使函数可以在循环中迭代。此外，调用 StopIteration 异常中断循环。Python 中的循环都是通过异常 StopIteration 中断的。这样 switch 类就构造完成了。

（4）编写调用代码，在 for... in... 循环中使用 switch 类。

【例 3-7】演示实现 switch 语句的功能。

```
class switch(object):
    def __init__(self, value):          # 初始化需要匹配的值 value
        self.value = value
        self.fall = False               # 如果匹配到的 case 语句中没有 break,则 fall 为 true。
    def __iter__(self):
        yield self.match                # 调用 match 方法 返回一个生成器
        raise StopIteration             # StopIteration 异常来判断 for 循环是否结束
    def match(self, * args):            # 模拟 case 子句的方法
        if self.fall or not args:       # 如果 fall 为 true,则继续执行下面的 case 子句
                                        # 或 case 子句没有匹配项,则流转到默认分支。
            return True
        elif self.value in args:        # 匹配成功
            self.fall = True
            return True
        else:                           # 匹配失败
            return False
operator = "+"
x = 1
y = 2
for case in switch(operator):           # switch 只能用于 for in 循环中
    if case('+'):
        print x + y
        break
    if case('-'):
        print x - y
```

更多资源请访问稀酷客(www.ckook.com)

```
        break
    if case('* '):
        print x * y
        break
    if case('/'):
        print x / y
        break
    if case():                          # 默认分支
        print ""
```

- 第1行到第18行代码定义了switch类。定义了__init__()、__iter__()、match()方法。
- 第23行代码在for... in...循环中调用switch类，变量operator作为switch类的参数传递给构造函数。变量operator的值等于“+”，程序执行到第24行。
- 第24行代码定义了一个字典result。该字典的key值由“+”、“-”、“*”、“\”四则运算符组成。value值由对应的算术表达式组成。
- 第25行代码输出x + y的结果。输出结果：0.5。
- 第26行代码使用break语句中断switch分支结构，程序流转到文件的末尾。

本章3.3.3节将继续以这个程序为例，探讨当case子句中没有break语句时的执行情况。

注意 switch语句造成代码不易维护，使源文件臃肿。面向对象的设计中常常对switch语句进行重构，把switch语句分解为若干个类。当然，对于分支流程简单的switch，可以使用字典来实现。使用字典更容易管理switch，而switch类回到了C、Java的老路上，而且写法更复杂了，并不值得推荐。

3.3 循环语句

循环语句是指重复执行同一个代码块。循环语句是由循环体及循环终止条件两部分组成，一组被重复执行的语句称为循环体，循环的终止条件决定重复执行的次数。Python中的循环语句有while语句和for语句。

3.3.1 while循环

在Python语言中，while循环的格式如下所示：

```
while(表达式):
...
else:
...
```

while循环的执行过程：当循环表达式为true时，依次执行while中的语句。直到循环表达式的值为false，程序的流程转到else语句。其中else子句可以省略，表达式两侧的括号也可以省略。

注意 while循环中的else子句也属于循环的一部分，最后一次循环结束后将执行else子句。

【例3-8】演示while循环的使用方法。程序首先要求输入5个数字，然后依次输出这5个数字。

```
# while 循环
numbers = raw_input("输入几个数字,用逗号分隔:").split(",")
print numbers
x = 0
while x < len(numbers):
    print numbers[x]
    x + = 1
```

更多资源请访问稀酷客(www.ckook.com)

- 第 2 行代码使用 raw_input( ) 捕获输入。按照提示输入 5 个数字，并用逗号分隔。raw_input( ) 根据输入的逗号，生成一个列表。
- 第 3 行代码输出列表 numbers 的内容。
- 第 4 行代码定义变量 x，其值为 0。
- 第 5 行代码通过列表的长度遍历列表 numbers。
- 第 6 行代码输出列表中的值。
- 第 7 行代码，每次循环使变量 x 增 1。

【例 3-9】演示 else 子句在 while 循环中的使用。当变量 x 的值大于 0 时，执行循环，否则输出变量 x 的值。

```
# 带 else 子句的 while 循环
x = input("输入 x 的值:")
i = 0
while(x < > 0):
    if(x > 0):
        x - = 1
    else:
        x + = 1
    i = i + 1
    print "第% d 次循环:" % i, x
else:
    print "x 等于 0:", x
```

- 第 2 行代码输入变量 x 的值。
- 第 3 行代码定义变量 i，变量 i 表示循环的次数。
- 第 4 行代码，根据 x < > 0 的条件循环。如果 x 不等于 0，则执行第 5 行代码，否则执行 else 子句的代码。
- 第 5 行代码判断变量 x 的值是否大于 0。
- 第 6 行代码，如果 x 的值大于 0，则每次循环都减 1。
- 第 7 行代码判断变量 x 的值是否小于 0。
- 第 8 行代码，如果 x 的值小于 0，则每次循环都加 1。
- 第 9 行代码，每次循环使变量 i 的值加 1。
- 第 11 行代码，循环结束，else 子句输出变量 x 的值。输出结果：

```
0
```

在使用循环语句时，应注意循环表达式的布尔值，避免死循环的出现。死循环是指循环条件永远为真。例如：

```
i = 1
while i > 0:
    i = i + 1
    print i
```

在以上代码中，变量 i 的值永远都大于 0，所以是一个死循环。

3.3.2　for 循环

for 循环用于遍历一个集合，依次访问集合中的每个项目。for 循环的格式如下所示：

```
for 变量 in 集合:
…
else:
…
```

更多资源请访问稀酷客(www.ckook.com)

for... in... 循环的执行过程：每次循环从集合中取出一个值，并把该值赋值给变量。集合可以是元组、列表、字典等数据结构。其中 else 子句可以省略。

注意　for 循环中的 case 子句也属于循环的一部分，最后一次循环结束后将执行 else 子句。

for... in... 循环通常与 range( )函数一起使用，range( )返回一个列表，for... in... 遍历列表中的元素。range( )函数的声明如下：

```
range([start,] stop[, step]) -> list of integers
```

- range( )返回一个递增或递减的数字列表，列表的元素值由 3 个参数决定。
- 参数 start 表示列表开始的值，默认值为“0”。
- 参数 stop 表示列表结束的值，该参数不可缺少。
- 参数 setp 表示步长，每次递增或递减的值，默认值为“1”。
- range( )返回一个递增或递减的数字列表，列表的元素值由以上 3 个参数决定。例如，range（5，－1，－1）返回的列表为［5，4，3，2，1，0］。

【例 3-10】遍历 range( )生成的列表，过滤出正数、负数和 0。

```
# for in 语句
for x in range( -1, 2):
    if x > 0:
        print "正数:",x
    elif x == 0 :
        print "零:",x
    else:
        print "负数:",x
else:
    print "循环结束"
```

- 第 2 行代码遍历 range（－1，2）生成的列表。Range（－1，2）返回的列表为［－1，0，1］。每次循环变量 x 的值依次为－1、0、1。
- 第 3 行代码判断变量 x 的值是否大于 0。
- 第 4 行代码，输出正数的值。输出结果：正数：1。
- 第 5 行代码判断变量 x 的值是否等于 0。
- 第 6 行代码，输出结果：零：0。
- 第 8 行代码，输出负数的值。输出结果：负数：－1。
- 第 9 行代码，此时并没有结束 for 循环，else 子句执行后循环才结束。输出结果：负数：－1。

在 C、Java 语言中，支持如下结构的 for 语句：

```
for(表达式1; 表达式2; 表达式3)
    语句块
```

Python 不支持这样的 for 循环。如果需要编写类似功能的循环，可以使用 while 循环。例如：

```
x = 0
while x < 5:
    print x
    x = x + 2
```

while 循环的写法比较琐碎，需要比较判断。也可以使用 for 循环，借助 range( )函数来实现。例如：

```
for x in range(0, 5, 2):
    print x
```

更多资源请访问稀酷客(www.ckook.com)

- 输出的数字在［0，5］这个区间，不包括5。每次循环 x 的值加 2。输出结果：

```
0
2
4
```

- 这里只用了两行代码即实现了传统 for 循环。如果要用条件表达式作为循环的条件，可以构造 range( ) 函数来实现。

3.3.3　break 和 continue 语句

break 和 continue 语句用于控制语句的跳转，下面分别介绍 break 和 continue 语句的使用方法。

1. break 语句

break 语句可以使程序跳出 switch 语句，也可以结束循环语句，分别说明如下。

（1）break 语句可以使程序跳出 switch 语句。例如，3.2.4 节【例3-7】模拟 switch 分支结构中使用了 break 语句：

```
operator = "+"
x = 1
y = 2
for case in switch(operator):          # switch 只能用于 for in 循环中
    if case('+'):
        print x + y
        break
    if case('-'):
        print x - y
        break
    if case('* '):
        print x * y
        break
    if case('/'):
        print x / y
        break
    if case():                         # 默认分支
        print ""
```

- 第 7 行代码中使用了 break。当变量 operator 的值为“+”，则执行表达式 x + y。然后中断 switch 分支结构，后面的 case 分支都不会执行。此时输出结果为 3。后面的 break 作用相同。当匹配到某个 case 后，程序将跳出 switch。
- 如果第一个 case 不使用 break 子句，程序将输出两个值，分别是 3 和 -1。因为输出表达式 x + y 后，分支结构并没有中断，程序将流转到下面一个 case。然后继续计算表达式 x - y 的值，遇到后面的 break 语句才退出分支结构。

（2）在循环结构中，break 语句可以提前结束循环。

【例 3-11】在 0 ~ 99 的数中查找用户输入的值。

```
x = input("输入 x 的值:")
y = 0
for y in range(0, 100):
    if x == y:
        print "找到数字:", x
        break
else:
    print "没有找到"
```

- 第 1 行代码捕获用户输入的值，并把该值赋值给变量 x。
- 第 2 行代码定义一个变量 y，变量 y 用于暂存需要遍历的列表的值。

更多资源请访问稀酷客(www.ckook.com)

- 第3行代码使用 for... in... 循环遍历 range（0，100）返回的列表。range（0，100）的返回值为［0、1、2... 99］。
- 第4行代码判断输入的值是否等于列表中的值。如果条件成立，输出查找到的数字，并立即退出循环。循环结束，后面的 else 子句将不会被执行。
- 第7行代码，当没有找到输入的值时，else 子句的代码将被执行。

注意 break 语句不能运行在循环体或分支语句之外，否则，Python 解释器将提示如下错误：

```
SyntaxError: 'break'outside loop'
```

2. continue 语句

continue 语句用于循环中的控制，当程序执行到 continue 语句时，程序将转到下一次循环。break 语句会直接中断循环，而 continue 语句则是停止本次循环，进入下一次循环，循环语句并没有中止。

【例 3-12】演示 continue 语句的使用方法。

```
x = input("输入 x 的值:")
y = 0
for y in range(0, 100):
    if x < > y:
        print "y = ", y
        continue
    else:
        print "x = ", x
        break
```

- 第6行代码，如果当前循环的次数与用户输入的数字不相等，则进入下一次循环。
- 第9行代码，如果当前循环的次数与用户输入的数字相等，则中断循环。

3.4 结构化程序示例

本节将结合结构化程序设计的思路，讲解使用 Python 实现冒泡排序的方法。冒泡排序将使用前面讲解的条件判断和循环语句等知识。冒泡排序是数据结构中的一种排序算法，学过数据结构课程的读者应该不陌生。

冒泡排序的基本思想是，将需要排序的元素看作是一个个“气泡”，最小的“气泡”最快浮出水面，排在前面。较小的“气泡”排在第二个位置，依次类推。冒泡排序需要对数列循环若干次，例如数列中有 i 个元素。第一遍循环，自底向上检查一遍这个数列，比较相邻的两个元素。如果较小的元素在数列的下面，把较小的元素排在前面。依次比较之后，就把最大的元素置于底部了，第二遍循环就不需要比较最后一个元素了。依次类推，第 n 遍循环只需要从第一个元素开始，比较 i－n 次。经过 i－1 遍的处理后，数列就排序完成了。

假设有一个数列［23，12，9，15，6］，这个数列的排序过程如图 3-2 所示。

在实现中将用到 xrange( )。前面提到了 range( ) 函数生成迭代集合。此外，还可以使用 xrange( )。xrange( ) 的用法和 range( ) 相同，xrange( ) 的声明如下所示：

```
xrange ( [start,] stop [, step]) -> xrange object
```

xrange 是一个类，而且返回的是一个 xrange 对象。使用 xrange( ) 进行遍历，每次遍历只返回一个值。range( ) 返回的是一个列表，一次性计算并返回所有的值。因此，xrange( ) 的执行效率要高于 range( )。

更多资源请访问稀酷客(www.ckook.com)

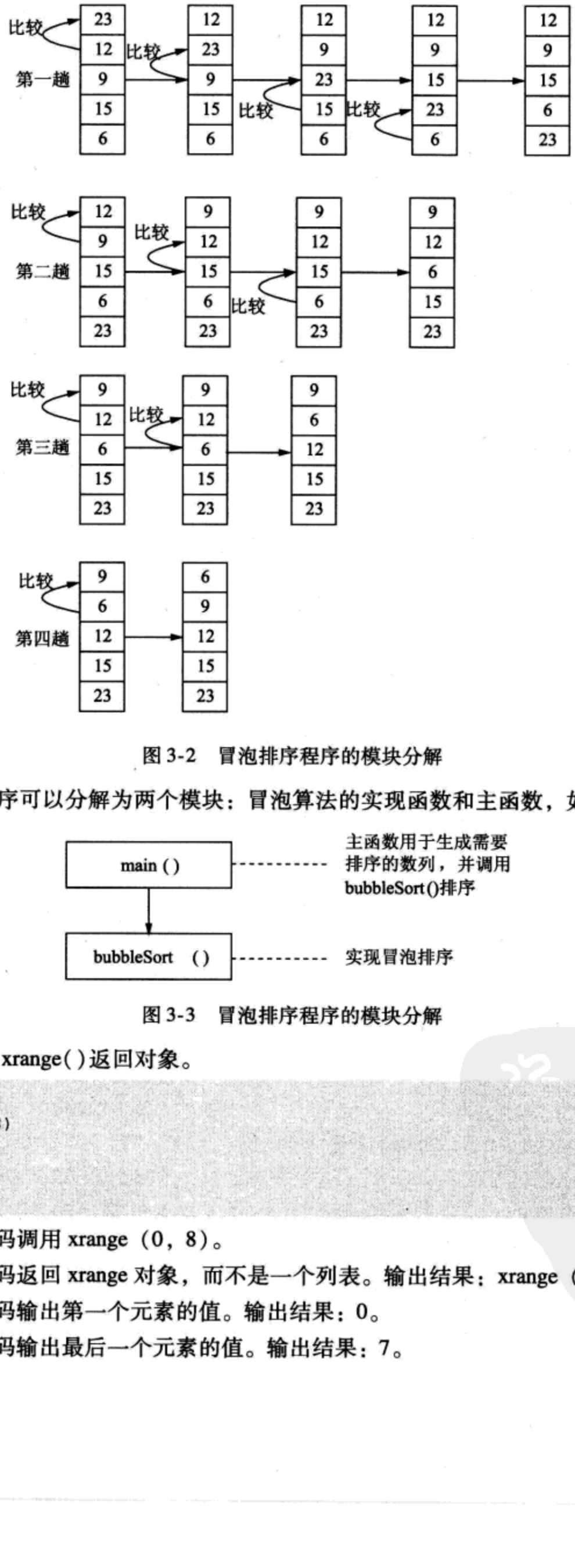


图 3-2　冒泡排序程序的模块分解

冒泡排序的程序可以分解为两个模块：冒泡算法的实现函数和主函数，如图 3-3 所示。

图 3-3　冒泡排序程序的模块分解

【例 3-13】用 xrange( )返回对象。

```
# xrange()
x = xrange(0,8)
pirnt x
print x[0]
print x[7]
```

- 第 2 行代码调用 xrange（0，8）。
- 第 3 行代码返回 xrange 对象，而不是一个列表。输出结果：xrange（8）。
- 第 4 行代码输出第一个元素的值。输出结果：0。
- 第 5 行代码输出最后一个元素的值。输出结果：7。

更多资源请访问稀酷客(www.ckook.com)

【例 3-14】实现冒泡排序算法。

```
# 冒泡排序
def bubbleSort(numbers):                                    # 冒泡算法的实现
    for j in xrange(len(numbers) - 1, -1, -1):
        for i in xrange(j):
            if numbers[i] > numbers[i +1]:                  # 把数值小的数字放到顶端
                numbers[i], numbers[i +1] = numbers[i +1], numbers[i]
            print numbers
def main():                                                 # 主函数
    numbers = [23, 12, 9, 15, 6]
    bubbleSort(numbers)
if __name__ == '__main__':
    main()
```

- 第 2 行代码定义 bubbleSort( )实现冒泡排序。
- 第 3 行代码确定循环每趟的比较次数。
- 第 4 行代码循环比较相邻的两个元素。
- 第 5 行代码判断相邻两个元素的大小。
- 第 6 行代码把数值较小的数排到前面。
- 第 7 行代码输出每趟的比较结果。输出结果：

```
[12, 23, 9, 15, 6]
[12, 9, 23, 15, 6]
[12, 9, 15, 23, 6]
[12, 9, 15, 6, 23]
[9, 12, 15, 6, 23]
[9, 12, 15, 6, 23]
[9, 12, 6, 15, 23]
[9, 12, 6, 15, 23]
[9, 6, 12, 15, 23]
[6, 9, 12, 15, 23]
```

3.5　小结

本章介绍了条件语句和循环语句的使用方法，并结合结构化程序设计的原理讲解了这些语句的要点。Python 与 C、Java 的 if 语句用法非常相似，只是格式稍有不同。Python 使用字典替代 switch 语句，使编写的程序更简洁。while 循环和 for 循环都有一个 else 子句，这个语法是 C 和 Java 中没有的。最后通过分析冒泡排序的算法，讲解了结构化程序设计的步骤和方法。

在实际应用中，许多问题都需要使用条件语句和循环语句进行控制，几乎所有的程序都要涉及判断、循环。这些语法和概念是学习一门编程语言的基础，也是最基本的要求。本章的例子中用到了列表、字典等数据结构，这些数据结构是 Python 所特有的。

更多资源请访问稀酷客(www.ckook.com)

第 4 章　内置数据结构

数据结构是用来存储数据的逻辑结构，合理使用数据结构才能编写出优秀的代码。本章将向读者介绍 Python 提供的几种内置数据结构——元组、列表、字典和序列。内置数据结构是Python语言的精华，也是使用 Python 进行程序开发的基础。本章的知识点：

- 元组、列表与字典的创建和使用方法
- 元组的遍历
- 元组和列表的“解包”操作
- 列表的排序、查找和反转
- 字典的特性
- 序列的含义

4.1　元组

元组是 Python 中内置的一种数据结构。元组由不同的元素组成，每个元素可以存储不同类型的数据，如字符串、数字甚至元组。元组是“写保护”的，即元组创建后不能再做任何修改操作，元组通常代表一行数据，而元组中的元素代表不同的数据项。

4.1.1　元组的创建

Tuple（元组）由一系列元素组成，所有元素被包含在一对圆括号中。创建元组时，可以不指定元素的个数，相当于不定长的数组，但是一旦创建后就不能修改元组的长度。元组创建的格式如下所示：

```
tuple_name = (元素1, 元素2, ...)
```

【例 4-1】演示元组的初始化。

```
tuple_name = ("apple", "banana", "grape", "orange")
```

【例 4-1】代码创建了一个名为 tuple_ name 的元组，该元组由 4 个元素组成，元素之间使用逗号分隔。

如果需要定义一个空的元组，表达方式更简单。创建空的元组只需要一对空的圆括号：

```
tuple_name = ()
```

如果创建的元组只包含一个元素，可能会错误地忽略该元素后的逗号，以致 Python 无法区分变量 tuple_ name 是元组还是表达式，Python 会误认为圆括号中的内容为表达式，因此 tuple［0］输出的结果并非期望的值。如下面错误的写法：

```
tuple_name = ("apple")
print tuple_name [0]
```

更多资源请访问稀酷客(www.ckook.com)

运行这段代码并不会提示任何错误，代码将输出字母“a”，而不是期望的“apple”。正确的写法如下所示：

```
tuple_name = ("apple",)
print tuple_name [0]
```

此时，将输出 tuple_name 元组中唯一的元素“apple”。所以，创建一个有唯一元素的元组，需要注意在该元素的后面添加一个逗号，使 Python 能正确识别出元组中的元素。

注意　元组的索引是从 0 开始计数的，因此 tuple［0］获得的是元组 tuple 中第 1 个元素。Python 中其他的数据结构也遵循这个规则。

4.1.2　元组的访问

元组中元素的值通过索引访问，索引是一对方括号中的数字，索引也称之为“下标”。元组的访问格式如下所示：

```
tuple_name [n]
```

其中，tuple_name［n］表示访问元组的第 n 个元素，索引 n 的值可以为 0、正整数或负整数。前面一节已经提到了元组的输出方法，例如：

```
print tuple_name [1]
```

如果将上面这行代码放在【例 4-1】代码的后面，将输出字符串“banana”，print 语句表示输出，tuple_name［1］表示访问 tuple_name 的第 2 个元素。

可以把元组理解为数组，使用过 C 或 Java 语言编程的读者应该不会陌生，数组的下标是从 0 开始计数，元组的下标同样也是从 0 开始计数的。

元组创建后其内部元素的值不能被修改，如果修改元组中的某个元素，运行时将报错。例如：

```
tuple_name [0] = "a"
```

对 tuple_ name 元组中第 1 个元素进行赋值，运行代码后错误信息如下：

```
TypeError: 'tuple_name ' object does not support item assignment
```

可见，元组中的元素不支持赋值操作。

注意　元组不能添加或删除任何元素。因此，元组不存在任何添加、删除元素的方法。

元组的索引还有一些特殊的用法，这些用法对于获取元组的值非常方便，例如负数索引和分片索引。这两个特性是 Python 的特有用法，元组类似 C 或 Java 语言中的数组，但 C、Java 并不支持负数索引。负数索引从元组的尾部开始计数，最尾端的元素索引表示为“－1”，次尾端的元素索引表示为“－2”，依次类推。负索引与元素的对应关系如图 4-1 所示，tuple［－1］的值为 value4，tuple［－2］的值为 value3。

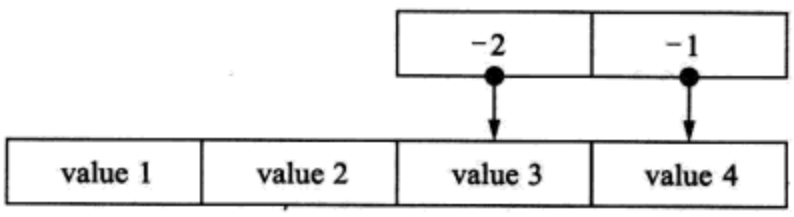


图 4-1　负索引计数图示

分片（slice）是元组的一个子集，分片是从第 1 个索引到第 2 个索引（不包含第 2 个索引所

更多资源请访问稀酷客(www.ckook.com)

指向的元素）所指定的所有元素。分片索引可以为正数或负数，两个索引之间用冒号分隔。分片的格式如下所示：

```
tuple_name [m:n]
```

其中，m、n 可以是 0、正整数或负整数。

分片索引与元素的对应关系如图 4-2 所示，tuple name［1：3］将返回（value2，value3，value4）。

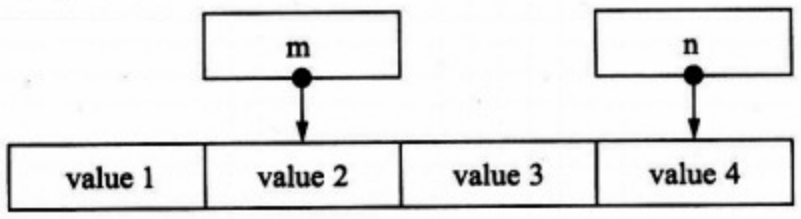


图 4-2　分片索引计数图示

【例 4-2】演示负数索引和分片索引的使用方法。

```
tuple = ("apple", "banana", "grape", "orange")
print tuple[-1]
print tuple[-2]
tuple2 = tuple[1:3]
tuple3 = tuple[0:-2]
tuple4 = tuple[2:-2]
print tuple2
print tuple3
print tuple4
```

- 元组的索引可以用负数表示，第 2 行、第 3 行代码将分别输出“orange”和“grape”。
- 使用分片截取后，将得到一个新的元组。第 4 行到第 6 行代码分别创建了 3 个元组。
- tuple2 输出结果：('banana', 'grape')。
- tuple3 输出结果：('apple', 'banana')。
- tuple4 输出结果：()。

元组还可以由其他元组组成。例如，二元元组可以表示为：

```
tuple = (('t1', 't2'), ('t3', 't4'))
```

tuple 是一个二元元组，该元组由（'t1', 't2'）和（'t3', 't4'）组成。

【例 4-3】演示元组间的包含关系以及如何访问二元元组中的元素。

```
fruit1 = ("apple", "banana")
fruit2 = ("grape", "orange")
tuple = (fruit1, fruit2)
print tuple
print "tuple[0][1] =", tuple[0][1]
print "tuple[1][1] =", tuple[1][1]
print "tuple[1][1] =", tuple[1][2]
```

- 第 3 行代码创建了一个复合元组 tuple，tuple 由元组 fruit1 和 fruit2 组成。元组 tuple 的输出结果：(('apple', 'banana'), ('grape', 'orange'))。
- tuple［0］［1］表示访问 tuple 元组中第 1 个元组的第 2 个元素，即 fruit1 元组中的第 2 个元素，可以把 tuple 理解为二维数组。tuple［0］［1］输出结果：tuple［0］［1］= banana。
- tuple［1］［1］表示访问 tuple 元组中第 2 个元组的第 2 个元素，即 fruit2 元组中的第 2 个元素。tuple［1］［1］输出结果：tuple［1］［1］= orange。

更多资源请访问稀酷客(www.ckook.com)

- 同理，tuple［1］［2］将访问 fruit2 元组中的第 3 个元素，因为此元素不存在，元组的索引访问越界。运行后将输出错误信息：IndexError：tuple index out of range。

tuple 元组的存储结构如图 4-3 所示。

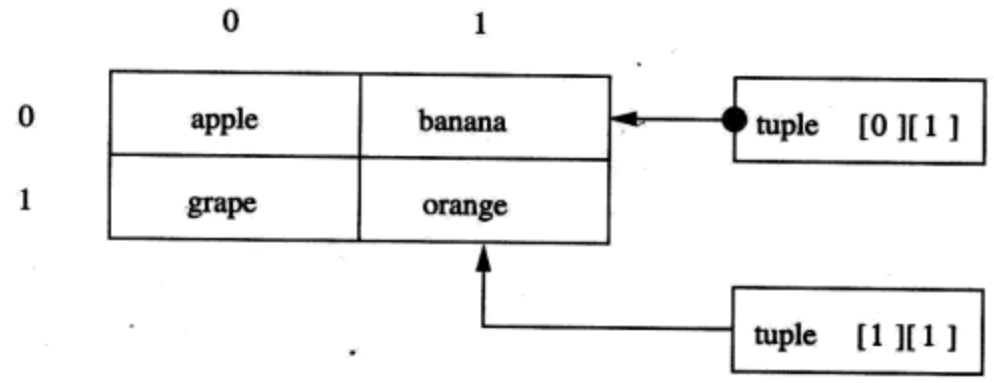


图 4-3　tuple 元组的存储结构

在 Python 中，将创建元组的过程称之为“打包”。相反，元组也可以执行“解包”的操作。“解包”可以将元组中的各个元素分别赋值给多个变量。这样，避免了使用循环遍历的方法去获取每个元素的值，降低了代码的复杂性，表达方式更自然。

【例 4-4】演示元组“打包”和“解包”的过程。

```
# -* - coding: UTF-8 -* -

#打包
tuple = ("apple", "banana", "grape", "orange")
#解包
a, b, c, d = tuple
print a, b, c, d
```

- 第 1 行代码指定了 UTF－8 字符集。由于本段代码使用了中文注释，为了能使 Python 编译器正确识别中文字符，所以需要在代码顶部添加这样一段特殊的注释。
- 第 4 行代码创建了一个 tuple 元组，元组的创建即“打包”的过程。
- 第 6 行代码使用了一个赋值操作，赋值号的右边是 tuple 元组，赋值号的左边有 4 个变量，元组 tuple 中的 4 个元素分别赋值给这 4 个变量。
- 第 7 行代码输出 4 个变量的值，输出的结果为：apple banana grape orange。

4.1.3　元组的遍历

元组的遍历是指通过循环语句依次访问元组中各元素的值。遍历元组需要用到range( )和len( )这两个函数。range( )和 len( )都是 python 的内建函数，这些函数可直接调用，不需要import语句导入模块。内建函数是 python 自动导入的函数，相当于 java 中的 lang 包。

len( )计算出 tuple 元组中元素的个数，range( )返回一个由数字组成的列表，关于列表的概念请参考 4.2 节，range( )的声明如下所示：

```
range([start,] stop[, step]) -> list of integers
```

- range( )返回一个递增或递减的数字列表，列表的元素值由 3 个参数决定。
- 参数 start 表示列表开始的值，默认值为“0”。
- 参数 stop 表示列表结束的值，该参数不可缺少。
- 参数 setp 表示步长，每次递增或递减的值，默认值为“1”。

例如，range（5）返回的列表为［0，1，2，3，4］。

更多资源请访问稀酷客(www.ckook.com)

【例 4-5】演示二元元组的遍历。

```
tuple = (("apple","banana"),("grape",
"orange"),("watermelon",),("grapefruit",))
for i in range(len(tuple)):
    print "tuple[% d] :" % i, "" ,
    for j in range(len(tuple[i])):
        print tuple[i][j], "" ,
    print
```

- 第 1 行代码创建了一个由 4 个子元组组成的二元元组 tuple。
- 第 2 行到第 6 行代码遍历 tuple 元组，输出元组中各个元素的值。tuple 元组的遍历结果：

```
tuple[0] :  apple  banana
tuple[1] :  grape  orange
tuple[2] :  watermelon
tuple[3] :  grapefruit
```

此外，还可以使用 map( )对 tuple 元组进行“解包”，得到每个子元组，然后对每个子元组进行遍历，输出 tuple 元组中所有元素的值。map( )的声明如下所示：

```
map(function, sequence[, sequence, ...]) -> list
```

- map( )返回一个由自定义函数 function 处理后的列表。
- 参数 function 是程序员创建的函数，该函数用于处理参数 sequence。如果 function 的值为“None”，则 map( )返回一个由参数 sequence 组成的元组或列表。
- 参数 sequence 表示序列，元组和列表都是序列。序列的概念将在 4.4 节介绍。

【例 4-6】演示用 map( )实现元组的遍历。

```
tuple = (("apple","banana"),("grape",
"orange"),("watermelon",),("grapefruit",))
k = 0
for a in map(None,tuple):
    print "tuple[% d] :" % k, "" ,
    for x in a:
        print x, "" ,
    print
    k + = 1
```

- 第 1 行代码创建了一个由 4 个子元组组成的二元元组 tuple。
- 第 3 行代码对 tuple 元组“解包”，把“解包”后的子元组赋给变量 a。
- 第 5 行到第 7 行代码对元组 a 进行“解包”，“解包”后的值调用 print 输出。第 7 行代码表示打印完一个元组后回车。tuple 元组的遍历结果：

```
tuple[0] :  apple  banana
tuple[1] :  grape  orange
tuple[2] :  watermelon
tuple[3] :  grapefruit
```

4.2　列表

列表（List）是 Python 中非常重要的数据类型，通常作为函数的返回类型。列表和元组相似，也是由一组元素组成，列表可以实现添加、删除和查找操作，元素的值可以被修改。

4.2.1　列表的创建

List（列表）是 Python 内置的一种数据结构。它由一系列元素组成，所有的元素被包含在一

更多资源请访问稀酷客(www.ckook.com)

对方括号中。列表创建后，可以执行添加、删除或修改操作。使用过 Java 语言的读者可能想到了 Java 语言中的 List 接口，其中的 ArrayList 类继承自 List 接口，实现了动态数组的功能，可以添加或删除任意类型的对象。Python 中列表的作用和 ArrayList 类相似，用法更灵活。列表的创建格式如下所示：

```
list_name = [元素 1, 元素 2, ...]
```

列表的添加可以调用 append( )，该方法的声明如下所示：

```
append(object)
```

其中，object 可以是元组、列表、字典或任何对象。append( ) 每次添加的新元素将排列在列表的尾部。

列表的删除可以调用 remove( )，该方法的声明如下所示：

```
remove(value)
```

该方法将删除元素 value。如果 value 不在列表中，python 将抛出 ValueError 异常。

【例 4-7】演示列表的创建、添加和删除操作。

```
list = ["apple", "banana", "grape", "orange"]
print list
print list[2]
list.append("watermelon")
list.insert(1, "grapefruit")
print list
list.remove("grape")
print list
list.remove("a")
print list.pop ()
print list
```

- 第 1 行代码创建了一个 list 列表，该列表由 4 个元素组成。
- 第 2 行代码输出 list 列表中的内容：['apple', 'banana', 'grape', 'orange']。
- 第 3 行代码输出 list 列表中的第 3 个元素。
- 第 4 行代码调用 list 的 append( )，append( ) 追加了一个“watermelon”元素到列表的末尾。
- 第 5 行代码调用 insert( )，insert( ) 将元素插入到指定的索引位置。这里将“grapefruit”插入到 list 的第 2 个位置。此时的 list 列表为：

```
['apple','grapefruit', 'banana','grape','orange', 'watermelon']
```

- 第 7 行代码调用 remove( )，移除了“grape”元素，移除后的 list 列表为：

```
['apple', 'grapefruit', 'banana', 'orange', 'watermelon']
```

- 第 9 行代码移除“a”元素，由于列表中不存在此元素，因此程序运行后提示如下错误：

```
ValueError: list.remove(x): x not in list
```

- 第 10 行代码调用了 pop( )，pop( ) 取出列表中的最后一个元素，即“弹出”最后一个进入列表的元素“watermelon”。此时的 list 列表为：

```
['apple', 'grapefruit', 'banana', 'orange']
```

更多资源请访问稀酷客(www.ckook.com)

注意　如果 list 列表中存在两个相同的元素，此时调用 remove( )移除同名元素，将只删除 list 列表中位置靠前的元素。例如，如果 list 中存在两个“grape”元素，执行语句 list. remove (" grape") 后，其中一个“grape”将被删除，而且此元素是首次出现的那个“grape”元素。

4.2.2　列表的使用方法

列表的使用方法与元组十分相似，同样支持负索引、分片以及多元列表等特性，但是列表中的元素可以修改，而且存在一些处理列表的方法。

【例 4-8】演示负索引、分片和遍历二元列表的方法。

```
list = ["apple", "banana", "grape", "orange"]
print list[-2]
print list[1:3]
print list[-3:-1]
list = [["apple","banana"],["grape",
"orange"],["watermelon"],["grapefruit"]]
for i in range(len(list)):
    print "list[% d] :" % i, "" ,
    for j in range(len(list[i])):
        print list[i][j], "" ,
    print
```

- 第 2 行代码输出结果：grape。
- 第 3 行代码输出结果：['banana', 'grape']。
- 第 4 行代码输出结果：['banana', 'grape']。
- 第 6 行到第 10 行代码遍历二元列表，输出 list 列表中各元素的值。list 列表中，各子列表的长度是不同的，这并不妨碍 list 列表的遍历，list 列表的输出结果：

```
list[0] :  apple  banana
list[1] :  grape  orange
list[2] :  watermelon
list[3] :  grapefruit
```

列表实现了连接操作的功能，列表的连接提供了两种方式，一种是调用 extend( )连接两个不同的列表，另一种是使用运算符“ + ”或“ + = ”。

【例 4-9】演示列表的连接功能。

```
list1 = ["apple", "banana"]
list2 = ["grape", "orange"]
list1. extend(list2)
print list1
list3 = ["watermelon"]
list1 = list1 + list3
print list1
list1 + = ["grapefruit"]
print list1
list1 = ["apple", "banana"] * 2
print list1
```

- 第 3 行代码调用 extend( )，第 4 行输出结果：['apple', 'banana', 'grape', 'orange']。
- 第 6 行代码使用“ + ”运算符连接两个列表，第 7 行输出结果：['apple', 'banana', 'grape', 'orange', 'watermelon']。
- 第 8 行代码使用“ + = ”运算符，第 9 行输出结果：['apple', 'banana', 'grape', 'orange', 'watermelon', 'grapefruit']。

更多资源请访问稀酷客(www.ckook.com)

- 第 10 行代码使用“ * ”运算符，连接了两个相同的［'apple', 'banana'］元组，第 11 行输出结果：［'apple', 'banana', 'apple', 'banana'］。

4.2.3　列表的查找、排序与反转

list（列表）除可以进行添加、删除操作外，还提供了查找元素的方法。List（列表）的查找提供了两种方式，一种是使用 index 方法返回元素在列表中的位置，另一种方法是使用保留字“in”来判断元素是否在列表中。

【例 4-10】演示列表的查找方法。

```
list = ["apple", "banana", "grape", "orange"]
print list.index("grape")
print list.index("orange")
print "orange" in list
```

- 第 2 行代码返回元素“grape”对应的索引值，输出值为“2”。
- 同理，第 3 行代码输出值为“3”。
- 第 4 行代码判断元素“orange”是否在 list 列表中，返回值为“True”。

列表提供了排序和反转的方法，下面这段代码展示了列表的排序和反转：

```
list = ["banana", "apple", "orange", "grape"]
list.sort()
print "Sorted list:", list
list.reverse()
print "Reversed list:", list
```

- 第 2 行代码调用 sort( )，sort( )提供了参数 reverse 用于控制元素排序的顺序。如果不提供参数，默认采用首字母升序排列。如果设置参数 reverse = True，则采用首字母降序排列。这里元素按照首字母升序，排序后第 3 行的打印结果：

```
Sorted list: ['apple', 'banana', 'grape', 'orange']
```

- 第 4 行代码调用 reverse( )，颠倒列表中元素的排列顺序，反转后的结果：

```
Reversed list: ['orange', 'grape', 'banana', 'apple']
```

注意　在 Python 中，列表是由类 list 实现的，使用函数 help（list）查看 list 类的定义，可以快速了解列表所包含的方法，Help 函数同样适用于其他 Python 类。

表 4-1 列出了列表的常用方法。

表 4-1　列表的常用方法

方 法 名	说　明
append（object）	在列表的末尾添加一个对象 object
insert（index，object）	在指定的索引 index 处插入一个对象 object
remove（value）	删除列表中首次出现的 value 值
pop（［index］）	删除索引 index 指定的值；如果 index 不指定，则删除列表中最后一个元素
extend（iterable）	将 iterable 指定的元素添加到列表的末尾
index（value，［start，［stop］］）	返回 value 现在出现在列表中的索引
sort（cmp = None，key = None，reverse = False）	列表的排序
reverse( )	列表的反转

更多资源请访问稀酷客(www.ckook.com)

4.2.4　用列表实现堆栈和队列

堆栈和队列是数据结构中常用的数据结构，使用列表的 append( )、pop( )方法可以模拟这两种数据结构。append( )、pop( )的使用方法请参见 4.2.1 小节。

1. 堆栈的实现

堆栈是指最先进入堆栈的元素最后才输出，符合“后进先出”的原则。栈的插入、弹出是通过栈首指针控制的。当插入一个新的元素时，指针移到新元素的位置；当弹出一个元素时，指针移到下面一个元素的位置，即原堆栈倒数第 2 个元素的位置，该元素成为栈顶元素。

堆栈的实现方法是：调用 append( )可以把一个元素添加到堆栈的顶部，调用 pop( )方法可以把堆栈中的最后一个元素弹出来。

假设有一个堆栈［"apple"，"banana"，"grape"］，要向堆栈中添加一个新的元素“orange”。列表实现堆栈的原理如图 4-4 所示。

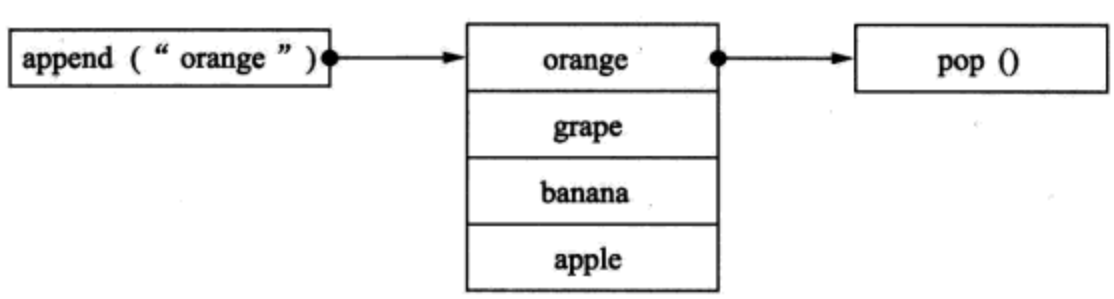


图 4-4　列表实现堆栈的原理

“apple”是列表中第 1 个进入的元素，所以在堆栈的最底端。调用 append（"orange"）后，程序把“orange”元素插入到堆栈的顶部。此时栈的指针移动到元素“orange”，栈中包含 4 个元素，“orange”在堆栈的顶部。然后调用 pop( )，弹出顶部的元素“orange”，栈的指针移到“grape”。

【例 4-11】演示用列表模拟堆栈的方法。

```
# 堆栈的实现
list = ["apple", "banana", "grape"]
list.append("orange")
print list
print "弹出的元素:", list.pop()
print list
```

- 第 2 行代码创建了一个堆栈 list。
- 第 3 行代码向堆栈中添加一个元素“orange”。
- 第 4 行代码输出添加新元素后堆栈中的内容。输出结果：['apple', 'banana', 'grape', 'orange']。
- 第 5 行代码弹出堆栈中最顶部的元素。输出结果：弹出的元素：orange。
- 第 6 行代码输出顶部元素被弹出后堆栈中的内容。输出结果：['apple', 'banana', 'grape']。

2. 队列的实现

队列是指最先进入队列的元素最早输出，符合“先进先出”的原则。队列的插入、弹出是通过队首指针和队尾指针控制的。当插入一个新的元素时，队尾指针移到新元素的位置；当弹出一个元素时，队首指针移到原队列中第 2 个元素的位置，该元素成为队列的第 1 个元素。队列也是通过调用 append( )和 pop( )方法实现的，但是 pop( )的调用方式有所不同，通过调用 pop（0）

更多资源请访问稀酷客(www.ckook.com)

弹出队列最前面的元素。假设有一个队列［" apple" ," banana" ," grape" ］，要向队列中添加一个新的元素“orange”。列表实现队列的原理如图 4-5 所示。

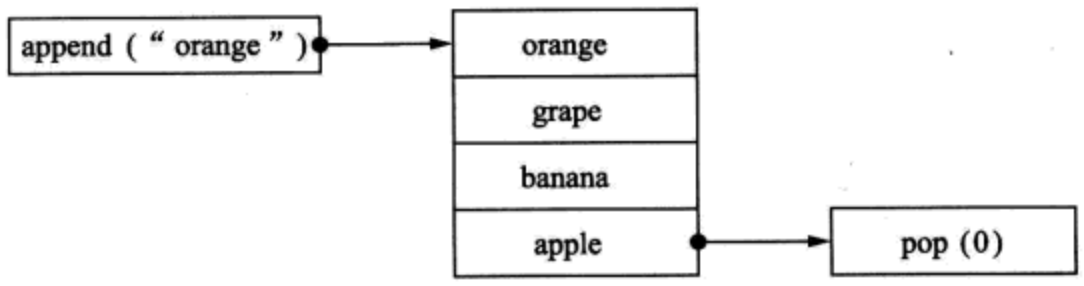


图 4-5　列表实现队列的原理

“apple”是队列中第 1 个进入的元素，所以置于队列的最前端。调用 append(" orange")后，程序把“orange”元素插入到队列的尾部，队尾指针移到元素“orange”。此时队列中包含 4 个元素，“orange”置于队列的尾部。然后调用 pop（0），弹出队列最前面的元素“apple”，队首指针移到元素“banana”。从而实现了“先进先出”的队列结构。

【例 4-12】演示用列表模拟队列的方法。

```
#队列的实现
list = ["apple", "banana", "grape"]
list.append("orange")
print list
print "弹出的元素:", list.pop(0)
print list
```

- 第 2 行代码创建了一个队列 list。
- 第 3 行代码向队列中添加一个元素“orange”。
- 第 4 行代码输出添加新元素后队列中的内容。输出结果：

```
['apple', 'banana', 'grape', 'orange']
```

- 第 5 行代码弹出队列最前面的元素。输出结果：

```
弹出的元素: apple
```

- 第 6 行代码输出最前面的元素被弹出后队列中的内容。输出结果：

```
['banana', 'grape', 'orange']
```

堆栈和队列是“数据结构”这门课程探讨的内容，读者可以参考相关的书籍进一步学习。

4.3　字典

字典（Dictionary）是 Python 中重要的数据类型，字典是由“键 - 值”对组成的集合，字典中的“值”通过“键”来引用，本节将介绍字典的定义、访问、排序等方法。

4.3.1　字典的创建

字典由一系列的“键-值”（key-value）对组成，“键-值”对之间用“逗号”隔开，并且被包含在一对花括号中。字典与 Java 语言中的 HashMap 类作用类似，都是采用“键-值”对映射的方式存储数据。例如，在开发管理系统时，通常把栏目编号作为“键”、栏目名称作为“值”存储在字典结构中，通过栏目编号引用栏目名称。

字典的创建和使用方法非常简单，创建字典的格式如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
dictionary_name = {key1 : value1, key2 : value2, ...}
```

其中，key1、key2 等表示字典的 key 值，value1、value2 等表示字典的 value 值。

如果需要创建一个空的字典，只需要一对花括号即可，代码如下所示：

```
dictionary_name = {}
```

【例 4-13】演示字典的创建和访问方法，字典通过一对中括号和索引来访问指定的元素。

```
dict = {"a" : "apple", "b" : "banana", "g" : "grape", "o" : "orange"}
print dict
print dict["a"]
```

- 第 1 行代码创建字典 dict，使用字母引用对应的值。
- 第 2 行代码输出字典的结果：{'a': 'apple', 'b': 'banana', 'o': 'orange', 'g': 'grape'}。

注意　代码中书写的顺序不是字典实际存储的顺序，字典将根据每个元素的 Hashcode 值进行排列。

- 第 3 行代码通过索引“a”获取对应的值“apple”，输出的结果：apple。

注意　字典的“键”是区分大小写的。例如，dict["a"] 与 dict["A"] 分别指向不同的值，应分别对待。

创建字典时，也可以使用数字作为索引：

```
dict = {1 : "apple", 2 : "banana", 3 : "grape", 4 : "orange"}
print dict
print dict[2]
```

- 第 1 行代码创建字典 dict，这里使用数字引用对应的值。
- 字典 dict 输出的结果：{1: 'apple', 2: 'banana', 3: 'grape', 4: 'orange'}。
- 第 3 行代码通过索引“2”获取对应的值“banana”，输出的结果：banana。

前面多次使用了 print 函数输出结果，print( ) 的使用非常灵活，也可以在 print( ) 中使用字典，下面这行代码演示了字典在 print( ) 中的使用：

```
print "% s, % (a)s, % (b)s" % {"a":"apple", "b":"banana"}
```

- 其中隐式的创建了字典 {"a":"apple","b":"banana"}，这个字典用来定制 print( ) 中的参数列表。“% s”输出这个字典的内容，“% (a)s”获取字典中对应 key 值“a”的 value 值，“% (b)s”获取字典中对应 key 值“b”的 value 值。输出结果：{'a': 'apple', 'b': 'banana'}，apple，banana。

4.3.2　字典的访问

字典的访问与元组、列表有所不同，元组和列表是通过数字索引来获取对应的值，而字典是通过 key 值获取相应的 value 值。访问字典元素的格式如下所示：

```
value = dict[key]
```

字典的添加、删除和修改非常简单，添加或修改操作只需要编写一条赋值语句，例如：

```
dict["x"] = "value"
```

如果索引“x”不在字典 dict 的 key 列表中，字典 dict 将添加一条新的映射（x : value）；如果索引“x”已在字典 dict 的 key 列表中，字典 dict 将直接修改索引 x 对应的 value 值。

字典与列表不同，字典没有 remove( )。字典元素的删除，可以调用 del( ) 实现，del( ) 属于

更多资源请访问稀酷客(www.ckook.com)

内建函数，直接调用即可。列表可以调用 pop( )弹出列表中一个元素，字典也有一个 pop( )方法，该方法的声明和作用与列表的 pop( )有些不同。字典的 pop( )的声明如下所示：

```
D.pop(k[,d]) -> v
```

- pop( )必须指定参数才能删除对应的值。其中，参数 k 表示字典的索引，如果字典 D 中存在索引 k，返回值 v 等于 D［k］，如果字典 D 中没有找到索引 k，返回值为 d。

如果需要清空字典中所有的内容，可以调用字典的 clear( )。

【例 4-14】演示字典的常用操作。

```
#字典的添加、删除、修改操作
dict = {"a" : "apple", "b" : "banana", "g" : "grape", "o" : "orange"}
dict["w"] = "watermelon"
del(dict["a"])
dict["g"] = "grapefruit"
print dict.pop("b")
print dict
dict.clear()
print dict
```

- 第 2 行代码创建字典 dict，这里使用字母索引引用对应的值。
- 字典 dict 输出的结果：{'a': 'apple', 'b': 'banana', 'g': 'grape', 'o': 'orange'}。

由于字典是无序的，因此字典中没有 append( )、remove( )等方法。如果需要向字典插入新的元素，可以调用 setdefault( )。这个方法的使用将在 4.3.3 节介绍。

字典的遍历有多种方式，最直接的方式是通过“for… in…”语句完成遍历的任务，下面这段代码演示了字典的遍历操作：

```
#字典的遍历
dict = {"a" : "apple", "b" : "banana", "g" : "grape", "o" : "orange"}
for k in dict:
    print "dict[%s] =" % k,dict[k]
```

- 第 3 行代码使用了“for… in…”语句循环访问字典 dict。值得注意的是，变量 k 获取的是字典 dict 的 key 值，并没有直接获得 value 值。因此打印输出时，通过 dict［k］来获取 value 值。
- 第 4 行代码依次输出结果：

```
dict[a] = apple
dict[b] = banana
dict[o] = orange
dict[g] = grape
```

此外，还可以使用字典的 items( )方法实现字典的遍历操作，items( )返回一个由若干元组组成的列表。下面这段代码演示了 items( )的使用方法：

```
#字典 items()的使用
dict = {"a" : "apple", "b" : "banana", "c" : "grape", "d" : "orange"}
print dict.items()
```

本段程序将输出以下结果：

```
[('a','apple'), ('c', 'grape'), ('b','banana'), ('d', 'orange')]
```

可见，items( )把字典中每对 key 和 value 组成了一个元组，并把这些元组存放在列表中返回。

【例 4-15】用字典的 items( )方法实现字典的遍历。

更多资源请访问稀酷客(www.ckook.com)

```
#调用 items()实现字典的遍历
dict = {"a" : "apple", "b" : "banana", "g" : "grape", "o" : "orange"}
for (k, v) in dict.items():
    print "dict[%s]=" % k, v
```

- 第3行代码中的变量 k 和 v 分别与字典 dict 中的 key 和 value 值对应，遍历输出的结果：

```
dict[a] = apple
dict[b] = banana
dict[o] = orange
dict[g] = grape
```

字典的遍历操作还可以使用 iteritems()、iterkeys()、itervalues()实现，这些方法将返回一个遍历器对象，通过这个对象实现遍历输出。iteritems()的声明如下所示：

```
D.iteritems() -> an iterator over the (key, value) items of D
```

Iterkeys()的声明如下所示：

```
D.iterkeys() -> an iterator over the keys of D
```

Itervalues()的声明如下所示：

```
D.itervalues() -> an iterator over the values of D
```

【例 4-16】使用 iteritems()、iterkeys()、itervalues()实现字典的遍历。

```
#调用 iteritems()实现字典的遍历
dict = {"a" : "apple", "b" : "banana", "o" : "grape", "g" : "orange"}
print dict.iteritems()
for k, v in dict.iteritems():
    print "dict[%s] =" % k, v
for (k, v) in zip(dict.iterkeys(), dict.itervalues()):
print "dict[%s] =" % k, v
```

- 第3行代码调用 iteritems()，返回字典的遍历器对象。变量 k 将获取字典的 key 值，变量 v 将获取字典的 value 值。输出结果：

```
<dictionary-itemiterator object at 0x00B49B60 >
```

- 第4行代码中的变量 k、v 与遍历器对象指定的元组（key、value）对应，遍历输出的结果：

```
dict[a] = apple
dict[b] = banana
dict[o] = orange
dict[g] = grape
```

- 第6行代码中的变量 k 与 iterkeys()指向的 key 列表中的值对应，变量 v 与 itervalues()指向的 value 列表中的值对应。遍历输出的结果：

```
dict[a] = apple
dict[b] = banana
dict[o] = orange
dict[g] = grape
```

前面的代码都采用字符串作为字典的 value 值，元组、列表甚至字典都可以作为字典的 value 值，使用元组、列表或字典作为 value 值创建的字典，称之为混合型字典。结合前面学习的知识，下面将说明如何访问混合型字典的内容。

混合型字典的创建格式如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
dict =  {"key1" : (tuple), "key2" : [list], "key3" : [dictionary] ...}
```

【例 4-17】演示混合型字典的创建和访问方法。

```
#使用列表、字典作为字典的值
dict = {"a" : ("apple",), "bo" : {"b" : "banana", "o" : "orange"}, "g" : ["grape","grape-
fruit"]}
print dict["a"]
print dict["a"][0]
print dict["bo"]
print dict["bo"]["o"]
print dict["g"]
print dict["g"][1]
```

- 第 2 行代码创建了一个包含元组、列表和字典的混合型字典。
- 第 3 行代码返回嵌套的元组，输出结果：('apple',)。
- 第 4 行代码使用了双下标访问元组的第 1 个元素，输出结果：apple。
- 第 5 行代码返回嵌套的字典，输出结果：{'b': 'banana', 'o': 'orange'}。
- 第 6 行代码使用双下标访问嵌套字典中 key 值“o”对应的 value 值，输出结果：orange。
- 第 7 行代码返回嵌套的列表，输出结果：['grape', 'grapefruit']。
- 第 8 行代码返回嵌套列表中的第 2 个元素的值，输出结果：grapefruit。

4.3.3　字典的方法

前面已经使用了一些字典的方法，本节将详细介绍字典的常用方法，使用这些常用方法可以极大地提高编程效率。

1. Keys( )和 values( )方法

Keys( )和 values( )方法将分别返回字典的 key 列表和 value 列表。

【例 4-18】演示 keys( )和 values( )的使用方法。

```
dict = {"a" : "apple", "b" : "banana", "c" : "grape", "d" : "orange"}
#输出 key 的列表
print dict.keys ()
#输出 value 的列表
print dict.values ()
```

- 第 3 行代码返回由字典的 key 值组成的列表，输出结果：['a', 'c', 'b', 'd']。
- 第 5 行代码返回由字典的 value 值组成的列表，输出结果：['apple', 'grape', 'banana', 'orange']。

2. get( )方法

前面已经提到，要获取字典中的某个 value 值，可以使用 dict［key］的结构访问。另一种获取 value 值的办法是使用字典的 get( )方法，get( )的声明如下：

```
D.get(k[,d]) -> D[k]
```

- 参数 k 表示字典的键值，参数 d 可以作为 get( )的返回值，参数 d 可以默认，默认值为“None”。
- get( )相当于一条 if... else... 语句，参数 k 如果在字典 D 中，get( )将返回 D［k］；参数 k 如果不在字典 D 中，则返回参数 d。get( )的等价代码如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
#get()的等价语句
D = {" key1" : " value1", " key2" : " value2"}
if " key1" in D:
    print D [" key1"]
else:
    print " None"
```

由于在字典 D 中查找到了 key 值“key1”，所以输出结果为“value1”。

【例 4-19】演示 get( )方法的使用。

```
#字典中元素的获取方法
dict = {"a" : "apple", "b" : "banana", "c" : "grape", "d" : "orange"}
print dict
print dict.get("c", "apple")
print dict.get("e", "apple")
```

- 第 4 行代码由于字典 dict 存在索引“c”，所以返回 key 值“c”对应的 value 值，输出结果：

```
grape
```

- 第 5 行代码由于字典 dict 中不存在索引“e”，所以返回参数“apple”，输出结果：

```
apple
```

- 采用 get( )访问字典中的 value 值减少了代码的长度，而且表达方式容易理解，避免了 if 语句带来的维护代价。

3. update( )方法

如果需要添加新的元素到已经存在的字典中，可以调用字典的 update( )方法。update( )把一个字典中的 key 和 value 值全部拷贝到另一个字典中，update( )相当于一个合并函数。update( )的声明如下所示：

```
D.update(E) -> None.
```

如果把字典 E 的内容合并到字典 D 中，update( )的等价代码如下所示：

```
#update () 的等价语句
D = {"key1" : "value1", "key2" : "value2"}
E = {"key3" : "value3", "key4" : "value4"}
for k in E:
    D[k] = E[k]
print D
```

- 第 4 行代码通过 for... in... 循环语句访问字典 E，并把字典 E 中的内容逐个添加到字典 D 中。字典 D 更新后的结果：

```
{'key3': 'value3', 'key2': 'value2', 'key1': 'value1', 'key4': 'value4'}
```

如果字典 E 中含有字典 D 中相同的键值，字典 E 的值将覆盖字典 D 中的值。例如：

```
#字典 E 中含有字典 D 中的 key
D = {"key1" : "value1", "key2" : "value2"}
E = {"key2" : "value3", "key4" : "value4"}
for k in E:
    D[k] = E[k]
print D
```

字典 D 和字典 E 都含有一个名为“key2”的键，但是对应的 value 值并不相同。字典 D 更新后的结果：

更多资源请访问稀酷客(www.ckook.com)

```
{'key2': 'value3', 'key1': 'value1', 'key4': 'value4'}
```

字典的 update( )也存在同样的问题，如果某些 key 在目标字典中已经存在，则更新后新字典中的值将覆盖原有的值。

【例 4-20】演示 update( )方法的使用。

```
#字典的更新
dict = {"a" : "apple", "b" : "banana"}
print dict
dict2 = {"c" : "grape", "d" : "orange"}
dict.update(dict2)
print dict
```

- 第 5 行代码调用 update( )，把字典 dict2 的内容更新到字典 dict，字典 dict 原有的内容保持不变。字典 dict 的输出结果：

```
{'a': 'apple','c': 'grape', 'b': 'banana', 'd': 'orange'}
```

注意　字典不属于序列，所以字典没有顺序性。update( )调用后，字典中各元素的排列顺序是无序的。关于序列的概念将在 4.4 节介绍。

4. setdefault( )方法

字典的 setdefault( )方法可以创建新的元素并设置默认值，setdefault( )的声明如下所示：

```
D.setdefault(k[,d]) -> D.get(k,d)
```

- setdefault( )与 get( )的使用有些相似，参数 k 表示字典的键值，参数 d 表示 D［k］的默认值。参数 d 可以省略，默认值为“None”。参数 k 的值如果在字典 D 中，setdefault( )将返回 get（k，d）获得的结果；参数 k 的值如果不在字典 D 中，字典 D 将添加新的元素 D［k］，并调用 get（k，d）返回参数 d 的值。

【例 4-21】演示 setdefault( )方法的使用。

```
# 设置默认值
dict = {}
dict.setdefault("a")
print dict
dict["a"] = "apple"
dict.setdefault("a", "default")
print dict
```

- 第 3 行代码添加了一个 key 值为“a”，且设置默认 value 值为“None”。Setdefault("a")的返回值结果为：None。
- 第 4 行代码输出结果：{'a': None}。
- 第 5 行代码更新 dict［"a"］的值为“apple”。
- 第 6 行代码再次调用 setdefault( )，设置默认的 value 值为“None”。
- 第 7 行代码输出字典 dict 的内容，由于设置了 dict［"a"］的值为“apple”，因此再次调用 setdefault( )也不会影响 value 值，所以 dict［"a"］的值仍为“apple”，而非“None”。字典 dict 的输出结果：

```
{'a':'apple'}
```

注意　虽然 setdefault（k[,d]）调用了 get（k,d），但是两者还是有区别的。如果字典中不存在 k，setdefault( )将创建一个新的元素添加到字典中，然后再调用 get( )返回 D［k］的值。

更多资源请访问稀酷客(www.ckook.com)

表 4-2 列出了字典中常用的一些方法。

表 4-2　字典中的常用方法

方 法 名	说　明
items( )	返回（key，value）元组组成的列表
iteritems( )	返回指向字典的遍历器
setdefault（k[,d]）	创建新的元素并设置默认值
pop（k[,d]）	删除索引 k 对应的 value 值，并返回该值
get（k[,d]）	返回索引 k 对应的 value 值
keys( )	返回字典中 key 的列表
values( )	返回字典中 value 的列表
update（E）	把字典 E 中数据扩展到原字典中
copy( )	复制一个字典中所有的数据

4.3.4　字典的排序与复制

在开发过程中，通常需要对字典的顺序进行排序、复制等操作，下面将分别介绍字典排序和复制的方法。

1. 字典的排序

前面已经提到，列表的排序可以使用 sorted( )实现，字典的排序同样可以使用该函数。

【例 4-22】演示用 sorted( )实现字典的排序。

```
#调用 sorted()排序
dict = {"a" : "apple", "b" : "grape", "c" : "orange", "d" : "banana"}
print dict
#按照 key 排序
print sorted(dict. items (), key = lambda d: d[0])
#按照 value 排序
print sorted(dict. items (), key = lambda d: d[1])
```

- 第 3 行代码输出字典 dict 的结果：

```
{'a' :'apple', 'b':'grape', 'c' : 'orange', 'd' :'banana'}
```

- 第 5 行代码把 dict. items( )作为需要排序的集合，items( )前面已经提到，它可以用于字典的遍历，并返回（key，value）元组组成的列表。参数 key 表示排序的“依据”，d[0] 表示 items( )中的 key，即按照 key 值进行排序。lambda 可以创建匿名函数，用于返回一些计算结果，lambda 的使用请参考第 5 章的内容。排序后的结果：

```
[('a','apple'), ('b', 'grape'), ('c','orange'), ('d', 'banana')]
```

- 第 7 行代码按照 dict. items( )中的 value 值进行排序，排序后的结果：

```
[('a', 'apple'), ('d', 'banana'), ('b', 'grape'), ('c', 'orange')]
```

2. 字典的复制

前面提到了 update( )，这个方法把字典 A 的内容复制到字典 B 中，且字典 B 中原有的内容保持不变，从而实现了字典 B 的扩展。如果需要把字典 A 的内容复制到字典 B 中，并清楚字典 B 中原有的内容，可以使用 copy( )。copy( )的声明如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
D.copy() -> a shallow copy of D
```

copy()实现了字典的浅拷贝操作，后面还会介绍深拷贝的概念及其与浅拷贝的区别，目前读者只需知道如何使用该方法即可。

【例 4-23】演示字典的复制操作。

```
#字典的浅拷贝
dict = {"a" : "apple", "b" : "grape"}
dict2 = {"c" : "orange", "d" : "banana"}
dict2 = dict.copy()
print dict2
```

- 第 4 行代码把字典 dict 的内容复制给 dict2。字典 dict2 的输出结果：{'a': 'apple', 'b': 'grape'}。

深拷贝能够拷贝对象内部所有的数据和引用，引用相当于 C 语言中指针的概念，Python 语言中不存在指针，但是在变量的内存结构中是通过引用来维护变量的。由于浅拷贝只是复制数据，数据的引用没有被复制，因此新的数据和旧的数据使用同一块内存空间。

例如，字典 B 浅拷贝字典 A 的数据，如果字典 B 的数据发生添加、删除或修改操作，字典 A 的数据也将发生变化；相反，如果字典 B 深拷贝字典 A 的数据，字典 B 的数据既使发生变化，也不会影响到字典 A。

深拷贝和浅拷贝可以应用于 Python 的任何对象，不只是限于字典。在 Python 中可以使用 copy 模块来实现对象的深拷贝和浅拷贝，其中的 deepcopy()用于深拷贝操作，copy()用于浅拷贝操作。

【例 4-24】演示浅拷贝和深拷贝的区别。

```
#字典的深拷贝
import copy
dict = {"a" : "apple", "b" : {"g" : "grape","o" : "orange"}}
dict2 = copy.deepcopy(dict)
dict3 = copy.copy(dict)
dict2["b"]["g"] = "orange"
print dict
dict3["b"]["g"] = "orange"
print dict
```

- 第 2 行代码导入了 copy 模块，在使用 deepcopy()和 copy()之前，必须先导入 copy 模块。
- 第 4 行代码字典 dict2 深拷贝字典 dict。
- 第 5 行代码字典 dict3 浅拷贝字典 dict。注意，copy.copy()的作用等价于 dict.copy()。
- 第 6 行代码改变了字典 dict2 中的数据。第 7 行代码中的字典 dict 并不会受字典 dict2 的影响，字典 dict 的数据保持不变。字典 dict 的输出结果：{'a': 'apple', 'b': {'o': 'orange', 'g': 'grape'}}。
- 第 8 行代码改变了字典 dict3 中的数据。第 9 行代码中字典 dict 受字典 dict3 的数据改变的影响。字典 dict 的输出结果：

```
{'a':'apple', 'b': {'o': 'orange', 'g':'orange'}}
```

4.3.5　全局字典——sys.modules 模块

sys.modules 是一个全局字典，该字典是 python 启动后就加载在内存中的。每当程序员导入新的模块，sys.modules 都将记录这些模块。字典 sys.modules 对于加载模块起到了缓存的作用。当某个模块第一次导入，字典 sys.modules 将自动记录该模块。当第 2 次再导入该模块时，python 会直接到字典中查找，从而加快了程序运行的速度。

更多资源请访问稀酷客(www.ckook.com)

字典 sys. modules 具有字典所拥有的一切方法，可以通过这些方法了解当前的环境加载了哪些模块。

【例 4-25】调用字典的 keys( )和 values( )方法，keys( )返回当前环境下加载的模块，values( )返回这些模块的引用路径。

```
import sys
print sys. modules. keys ()
print sys. modules. values ()
print sys. modules["os"]
```

- 第 2 行代码 keys( )返回 sys 模块以及 python 自动加载模块。输出结果：

```
['copy_reg','sre_compile', 'locale','_sre','__main__', 'site','__builtin__', 'operator', 'encodings',
'os. path', 'encodings. encodings', 'encodings. gbk', 'errno', 'encodings. codecs', 'sre_constants',
're', 'ntpath', 'UserDict', 'encodings. _multibytecodec', 'nt', 'stat', 'zipimport', 'warnings', 'en-
codings. types', '_codecs', '_multibytecodec', 'sys','codecs', 'types', '_types', '_codecs_cn', '_lo-
cale', 'signal', 'linecache','encodings. aliases', 'exceptions', 'sre_parse', 'os', 'encodings. _co-
decs_cn']
```

- 第 3 行代码 values( )返回的模块引用。
- 第 4 行代码返回索引“os”对应的引用，输出结果：

```
<module'os' from 'D: \developer \python \lib \os. pyc' >
```

下面这段代码实现了导入模块的过滤：

```
import sys
d = sys. modules. copy ()
import copy,string
print zip(set(sys. modules) - set(d))
```

- 第 2 行代码调用 copy( )，把当前导入的模块信息保存到字典 d 中。
- 第 3 行代码导入模块 copy、string，此时的字典 sys. modules 包含了原有的模块和新导入的模块。
- 第 4 行代码调用 set( )把字典 sys. modules、字典 d 存入 set 集合中，set 集合实现了减法运算符，set（sys. modules）- set（d）将返回在字典 d 中不存在，而在字典 sys. modules 中存在的模块。然后调用 zip( )对 set 集合“解包”，返回一个列表。该列表就是使用 import copy，string 语句导入的模块。输出结果：[（'copy ',),（'strop ',),（'string ',)]。

4.4　序列

序列是具有索引和切片能力的集合。元组、列表和字符串具有通过索引访问某个具体的值，或通过切片返回一段切片的能力，因此元组、列表和字符串都属于序列。

【例 4-26】演示序列的索引功能。

```
#索引操作
tuple = ("apple", "banana", "grape", "orange")
list = ["apple", "banana", "grape", "orange"]
str = "apple"
print tuple[0]
print tuple[-1]
print list[0]
print list[-1]
print str[0]
print str[-1]
```

更多资源请访问稀酷客(www.ckook.com)

❑ 第 5 行、第 6 行代码访问元组的第 1 个元素和最后一个元素。输出结果：

```
apple
orange
```

❑ 第 7 行、第 8 行代码访问列表的第 1 个元素和最后一个元素。输出结果：

```
apple
orange
```

❑ 第 9 行、第 10 行代码访问字符串的第 1 个字符和最后一个字符。输出结果：

```
a
e
```

【例 4-27】演示序列的分片功能。

```
#分片操作
tuple = ("apple", "banana", "grape", "orange")
list = ["apple", "banana", "grape", "orange"]
str = "apple"
print tuple[:3]
print tuple[3:]
print tuple[1:-1]
print tuple[:]
print list[:3]
print list[3:]
print list[1:-1]
print list[:]
print str[:3]
print str[3:]
print str[1:-1]
print str[:]
```

❑ 第 5 行至第 8 行代码依次输出：

```
('apple', 'banana','grape')
('orange',)
('banana', 'grape')
('apple', 'banana','grape', 'orange')
```

❑ 第 9 行至第 12 行代码依次输出：

```
['apple', 'banana', 'grape´]
['orange',]
['banana','grape']
['apple', 'banana', 'grape', 'orange']
```

❑ 第 13 行至第 16 行代码依次输出：

```
app
le
ppl
apple
```

注意　分片 seq [:3] 表示从序列第 1 个元素到第 3 个元素的值，分片 [:] 获得整个序列的值。

元组和列表都具有序列的特性，但是它们的区别也很明显。元组是只能读的一组数据，而且元组没有提供排序和查找的方法。列表的数据既可以读，也可以写，而且提供了丰富的操作方法，支持排序、查找操作。元组中的数据通常具有不同的含义，例如坐标（x，y）可以表示为一个元组，x 轴坐标和 y 轴坐标的含义是不同的。列表中的数据通常具有相同的含义，例如 [（x1，y1），（x2，y2）] 表示为一个列表，这个列表中有两组坐标值，每组坐标表示的意义是

更多资源请访问稀酷客(www.ckook.com)

相同的，它们都表示直角坐标系中的某个位置。表 4-3 说明了元组和列表的区别。

表 4-3　元组和列表的区别

	支持负索引	支持分片	支持添加、删除、修改	支持排序、查找	数据的组成
元组	是	是	否	否	一组不同含义的数据
列表	是	是	是	是	一组相同含义的数据

4.5　小结

本章介绍了 python 内置的几种数据结构，这些内置的数据结构是使用 python 进行程序开发的基础。其中，重点讲解了使用列表实现堆栈和队列的原理以及实现过程，以及列表、字典中常用方法的使用。最后通过具体的例子阐述了元组、列表和序列之间的关系，描述了元组和列表的区别。

更多资源请访问稀酷客(www.ckook.com)

第 5 章　模块与函数

本章将介绍 Python 中模块和函数的概念。结构化程序设计可以把复杂的问题分解为若干个子任务，针对子任务定义实现的模块和函数，本章将详细讨论 Python 模块和函数的特性。最后还将介绍 Python 的函数化编程。本章的知识点：

- 模块的创建和使用方法
- Python 中的内置模块
- Python 中的常用模块
- 函数的创建和使用方法
- lambda 函数
- generator 函数
- 函数化程序设计

5.1　Python 程序的结构

Python 的程序由包（package）、模块（module）和函数组成。包是由一系列模块组成的集合。模块是处理某一类问题的函数和类的集合。图 5-1 描述了包、模块、类和函数之间的关系。图中的函数和类表示零个或多个。

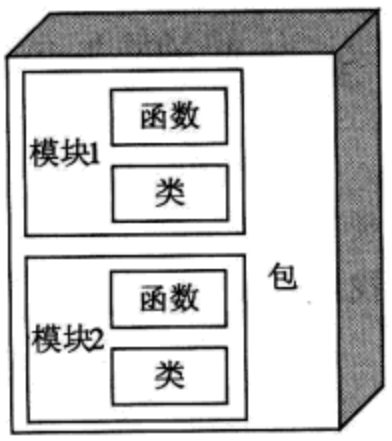


图 5-1　包、模块、类和函数之间的关系

包就是一个完成特定任务的工具箱，Python 提供了许多有用的工具包，如字符串处理、图形用户接口、Web 应用、图形图像处理等。使用这些工具包，可以提高程序员的程序开发效率、减少编程的复杂度、达到代码重用的效果。这些自带的工具包和模块安装在 Python 的安装目录下的 Lib 子目录中。

例如，Lib 目录中的 xml 文件夹就是一个包，这个包用于完成 XML 的应用开发。xml 包中有几个子包：dom、sax、etree 和 parsers。文件 __init__.py 是 xml 包的注册文件，如果没有该文件，Python 将不能识别 xml 包。在系统字典中定义了 xml 包。

注意　包必须至少含有一个__init__.py 文件，该文件的内容可以为空。__init__.py 用于标识当前文件夹是一个包。

5.2　模块

模块是 Python 中的一个重要概念，Python 的程序是由一个个模块组成的。前面的代码中已经接触到了模块，一个 Python 文件就是一个模块。下面将介绍模块的概念和特性。

更多资源请访问稀酷客(www.ckook.com)

5.2.1　模块的创建

模块把一组相关的函数或代码组织到一个文件中。一个文件即是一个模块。模块由代码、函数或类组成。例如，创建一个名为“myModule. py”的文件，即定义了一个名为“myModule”的模块。在该模块中定义一个函数 func( )和一个类 MyClass。MyClass 类中定义一个方法 myFunc( )：

```
# 自定义模块
def func():
    print "MyModule. func()"

class MyClass:
    def myFunc(self):
        print "MyModule. MyClass. myFunc()"
```

然后在 myModule. py 所在的目录下创建一个名为“call_ myModule. py”的文件。在该文件中调用 myModule 模块的函数和类：

```
#调用自定义模块的类和函数
import myModule

myModule. func()
myClass = myModule. MyClass()
myClass. myFunc()
```

- 第 2 行代码导入模块 myModule。
- 第 4 行调用模块的函数。调用时需要加前缀 myModule，否则 Python 不知道 func( )所在模块的命名空间。输出结果：MyModule. func( )。
- 第 5 行代码创建类 MyClass 的实例 myClass。这里也需要使用前缀 myModule 来调用类。
- 第 6 行调用类的方法 myFunc( )。输出结果：MyModule. MyClass. myFunc( )。

注意　myModule. py 和 call_ myModule. py 必须放在同一个目录下，或放在 sys. path 所列出的目录下，否则，Python 解释器找不到自定义的模块。

当 Python 导入一个模块时，Python 首先查找当前路径，然后查找 lib 目录、site-packages 目录（Python \ Lib \ site-packages）和环境变量 PYTHONPATH 设置的目录。如果导入的模块没有找到，在以上路径搜索一下是否含有这个模块。可以通过 sys. path 语句搜索模块的查找路径。

5.2.2　模块的导入

在使用一个模块中的函数或类之前，首先要导入该模块。前面的代码示例中已经多次使用过模块的导入，模块的导入使用 import 语句。模块导入的格式如下所示：

```
import module_name
```

这条语句可以直接导入一个模块。调用模块的函数或类时，需要以模块名作为前缀。关于类的定义请参考 8. 2. 2 节的内容。从模块中调用函数和类的格式如下所示：

```
module_name. func()
```

如果不想在程序中使用前缀符，可以使用 from... import... 语句将模块导入。from... import... 语句的格式如下所示：

```
from module_name import function_name
```

但是 from... import... 语句容易造成代码可读性差等问题，2. 2. 3 节比较了 import 语句和 from...

更多资源请访问稀酷客(www.ckook.com)

import... 语句的不同。导入模块下所有的类和函数，可以使用如下格式的 import 语句：

```
from module_name import *
```

如果只需要导入模块下某个类或函数，import 语句如下所示：

```
from module_name import function_name
```

此外，同一个模块文件支持多条 import 语句。例如，定义一个名为 myModule 的模块。该模块定义一个全局变量 count 和一个函数 func()。每次调用函数 func()，使变量 count 的值加 1：

```
count = 1

def func():
    global count
    count = count + 1
    return count
```

可以多次导入 myModule 模块，以便查看变量 count 的结果：

```
import myModule
print "count =", myModule.func()
myModule.count = 10
print "count =", myModule.count

import myModule
print "count =", myModule.func()
```

- 第 1 行代码导入模块 myModule。
- 第 2 行代码调用模块中的函数 func()。此时变量 count 的值等于2。输出结果：count = 2。
- 第 3 行代码给模块 myModule 中的变量 count 赋值，此时变量 count 的值等于 10。
- 第 4 行代码获取变量 count 的值。输出结果：count = 10。
- 第 6 行代码再次导入模块 myModule，变量 count 的初始值为 10。
- 第 7 行代码调用 func()，变量 count 的值加 1。输出结果：count = 11。

Python 中的 import 语句比 Java 的 import 语句更灵活。Python 的 import 语句可以置于程序中任意的位置，甚至可以放在条件语句中。在上面的代码段后添加如下语句：

```
# import 置于条件语句中
if myModule.count > 1:
    myModule.count = 1
else:
    import myModule
print "count =", myModule.count
```

- 第 2 行代码判断 myModule.count 的值是否大于 1。
- 第 3 行代码：如果 count 的值大于 1，则把变量 count 的值置为 1。由于前面代码段中变量 count 的值为 11，所以变量 count 的值被赋值为 1。
- 第 5 行代码，如果 count 的值小于等于 1，则导入 myModule 模块。
- 第 6 行代码输出变量 count 的值。输出结果：count = 1。

5.2.3　模块的属性

模块有一些内置属性，用于完成特定的任务，如__name__、__doc__。每个模块都有一个名称。例如，__name__用于判断当前模块是否是程序的入口，如果当前程序正在被使用，__name__的值为“__main__”。通常给每个模块都添加一个条件语句，用于单独测试该模块的功

更多资源请访问稀酷客(www.ckook.com)

能。例如，创建一个模块 myModule：

```
if__name__ =='__main__':
    print 'myModule 作为主程序运行'
else:
    print 'myModule 被另一个模块调用'
```

- 第 1 行代码判断本模块是否作为主程序运行。单独运行模块 myModule，输出结果如下所示：

```
myModule 作为主程序运行
```

创建另一个模块 call_ myModule。这个模块很简单，只要导入模块 myModule 即可：

```
import myModule
print __doc__
```

- 运行模块 call_ myModule，输出结果：myModule 被另一个模块调用。第 2 行代码调用了模块另一个属性__doc__。由于该模块没有定义文档字符串，所以输出结果为 None。输出结果：None。

属性__doc__可以输出文档字符串的内容，文档字符串的创建方法请参考 2. 4. 2 节。

5. 2. 4　模块的内置函数

Python 提供了一个内联模块 buildin，该模块中定义了一些软件开发中经常用到的函数，利用这些函数可以实现数据类型的转换、数据的计算、序列的处理等功能。下面将介绍内联模块中常用的函数。

1. apply( )

apply( )可以实现调用可变参数列表的函数，把函数的参数存放在一个元组或序列中。apply( )的声明如下所示：

```
apply(func [, args [,kwargs ]])
```

- 参数 func 是自定义的函数名称。
- 参数 args 是一个包含自定义函数参数的列表或元组。args 如果省略，则表示被执行的函数没有任何参数。
- 参数 kwargs 是一个字典。字典中的 key 值为函数的参数名称，value 值为实际参数的值。
- apply( )的返回值就是 func( )的返回值。

【例 5-1】演示 apply( )调用自定义函数的方法。

```
def sum(x=1, y=2):
    return x + y

print apply(sum, (1, 3))
```

- 第 1 行代码定义了函数 sum( )，并定义了两个形式参数。第 4 行代码通过 apply( )调用 sum（1，3）。输出结果：4。
- apply( )的作用可以用“*”和“**”参数符号替代，“*”和“**”的使用方法请参考 5. 3. 2 节。

注意　apply( )的元组参数是有序的。如本例所示，元素的顺序必须和 sum( )形式参数的顺序一致。

2. filter( )

filter( )可以对某个序列做过滤处理，对自定义函数的参数返回的结果是否为“真”来过滤，

更多资源请访问稀酷客(www.ckook.com)

并一次性返回处理结果。filter( )的声明如下所示：

```
filter(func or None, sequence) -> list, tuple, or string
```

- 参数 func 是自定义的过滤函数，在函数 func（item）中定义过滤的规则。如果 func 为“None”，则过滤项 item 都为“真”，返回所有的序列元素。
- 参数 sequence 为待处理的序列。
- filter( )的返回值是由 func( )的返回值组成的序列，返回的类型与参数 sequence 的类型相同。例如，参数 sequence 为元组，则返回的类型也是元组。

【例 5-2】演示 filter( )过滤序列的功能，从给定的列表中过滤出大于 0 的数字。

```
def func(x):
    if x > 0:
        return x
print filter(func, range(-9, 10))
```

- 第 5 行代码，使用 range( )生成待处理的列表，然后把该列表的值依次传入 func( )。func 返回结果给 filter( )，最后 filter( )把返回的值组成一个列表返回。输出结果如下所示：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

注意　filter( )中的过滤函数 func( )的参数不能为空，否则，没有可以存储 sequence 元素的变量，func( )也不能处理过滤。

3. reduce( )

对序列中元素的连续操作可以通过循环来处理（例如对某个序列中的元素累加操作）。Python提供的 reduce( )也可以实现连续处理的功能。reduce( )的声明如下所示：

```
reduce(func, sequence[, initial]) -> value
```

- 参数 func 是自定义的函数，在函数 func( )中实现对参数 sequence 的连续操作。
- 参数 sequence 为待处理的序列。
- 参数 initial 可以省略，如果 initial 不为空，则 initial 的值将首先传入 func( )进行计算。如果 sequence 为空，则对 initial 的值进行处理。
- reduce ( )的返回值是 func( )计算后的结果。

【例 5-3】实现对一个列表中的数字进行累加的操作。

```
def sum(x, y):
    return x + y

print reduce(sum, range(0, 10))
print reduce(sum, range(0, 10), 10)
print reduce(sum, range(0, 0), 10)
```

- 第 1 行代码，定义了一个 sum( )函数，该函数需要两个参数，执行累加操作。
- 第 4 行代码，对 0 +1 +2 +3 +4 +5 +6 +7 +8 +9 执行累加计算。输出结果：45。
- 第 5 行代码，对 10 +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 执行累加计算。输出结果：55。
- 第 6 行代码，由于 range（0，0）返回空列表，所以返回结果就是 10。输出结果：10。

reduce( )还可以对数字进行乘法、阶乘等复杂的累计计算。

注意　如果 reduce( )进行累计计算，必须在 sum 中定义两个参数，分别对应加法运算符两侧的操作数。

更多资源请访问稀酷客(www.ckook.com)

4. map( )

第 4 章中使用了 map( )对 tuple 元组进行“解包”操作，调用时设置 map( )的第一个参数为“None”。map( )的功能非常强大，可以对多个序列的每个元素都执行相同的操作，并组成列表返回。map( )的声明如下所示：

```
map(func, sequence[, sequence, ...]) -> list
```

- 参数 func 是自定义的函数，实现对序列每个元素的操作。
- 参数 sequence 为待处理的序列，参数 sequence 的个数可以是多个。
- map( )的返回值是对序列元素处理后的列表。

【例 5-4】实现列表中数字的幂运算。

```
def power(x): return x * * x
print map(power, range(1, 5))
def power2(x, y): return x * * y
print map(power2, range(1, 5), range(5, 1, -1))
```

- 第 1 行代码，定义了一个 power( )函数，实现了数字的幂运算。
- 第 2 行代码，把数字 1、2、3、4 依次传入函数 power 中，计算后的结果组成一个列表返回。输出结果：[1, 4, 27, 256]。
- 第 3 行代码，定义了一个 power( )2 函数，计算 x 的 y 次幂。
- 第 4 行代码，提供了两个列表参数。依次计算 1^5、2^4、3^3、4^2，计算后的结果组成一个列表返回。输出结果：[1, 16, 27, 16]。

注意　如果 map( )中提供多个序列，则每个序列中的元素一一对应进行计算。如果每个序列的长度不相同，则短的序列后补充“None”，再进行计算。

常用的内联函数如表 5-1 所示。

表 5-1　内置模块的函数

函　数	说　明
abs（x）	返回 x 的绝对值
apply（func［, args［, kwargs ］］）	把函数的参数放置在序列中传入函数
bool（［x］）	把一个值或表达式转换为 bool 类型。如果表达式 x 为值，返回 true，否则，返回 false
cmp（x, y）	比较 x，y 的大小
delattr（obj, name）	等价于 del obj. name
eval（s［, globals［, locals］］）	计算表达式的值
float（x）	把数字或字符串转换为 float 类型
hash（object）	返回一个对象的 hash 值
help（［object］）	返回内联函数的帮助说明
id（x）	返回一个对象的标识
input（［prompt］）	接受控制台的输入，并把输入的值转换为数字
int（x）	把数字或字符串转换为整型
len（obj）	对象包含的元素个数
range（［start,］ end［, step］）	生成一个列表并返回
raw_ input（［prompt］）	接受控制台的输入，返回字符串类型
reduce（func, sequence［, initial］）	对序列的值进行累计计算

更多资源请访问稀酷客(www.ckook.com)

（续）

函　数	说　明
round（x，n=0）	四舍五入函数
set（［iterable］）	返回一个 set 集合
sorted（iterable［，cmp［，key［，reverse］］］）	返回一个排序后的列表
sum（iterable［，start=0］）	返回一个序列的和
type（obj）	返回一个对象的类型
xrange（start［，end［，step］］）	功能和 range()类似，但是一次返回一个值
zip（seq1［，seq2，...］）	把 n 个序列作为列表的元素返回

5.2.5　自定义包

包就是一个至少包含__init__.py 文件的文件夹。Python 包和 Java 包的作用是相同的，都是为了实现程序的重用。把实现一个常用功能的代码组合到一个包中，调用包提供的服务从而实现重用。例如，定义一个包 parent。在 parent 包中创建两个子包 pack 和 pack2。pack 包中定义一个模块 myModule，pack2 包中定义一个模块 myModule2。最后在包 parent 中定义一个模块 main，调用子包 pack 和 pack2，如图 5-2 所示。

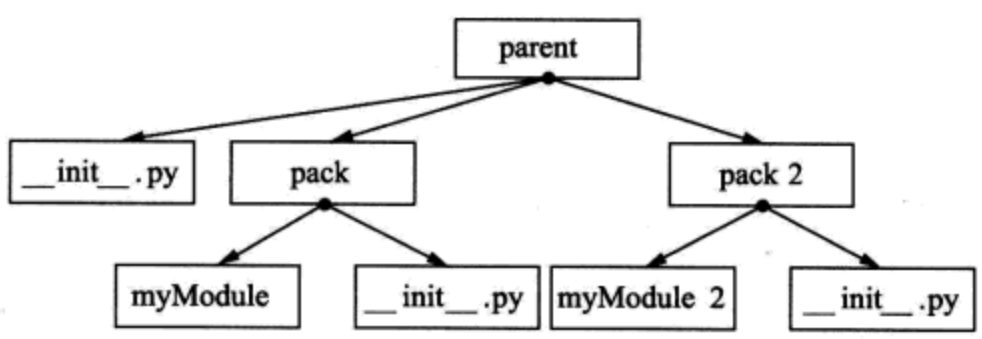


图 5-2　包与模块的树形关系图

包 pack 的__init__.py 程序如下所示：

```
if __name__ == '__main__':
    print '作为主程序运行'
else:
    print 'pack 初始化 '
```

这段代码初始化 pack 包，这里直接输出一段字符串。当 pack 包被其他模块调用时，将输出“pack 包初始化”。包 pack 的 myModule 模块如下所示：

```
def func():
    print "pack.myModule.func()"

if __name__ =='__main__':
    print 'myModule 作为主程序运行'
else:
    print 'myModule 被另一个模块调用'
```

当 pack2 包被其他模块调用时，将首先执行__init__.py 文件。pack2 包的__init__.py 程序如下所示：

```
if __name__ == '__main__':
    print '作为主程序运行'
else:
    print 'pack2 初始化'
```

更多资源请访问稀酷客(www.ckook.com)

包 pack2 的 myModule2 模块如下所示：

```
def func2 ():
    print "pack2. myModule2. func ()"

if __name__ == '__main__':
    print 'myModule2 作为主程序运行'
else:
    print 'myModule2 被另一个模块调用'
```

下面的 main 模块调用了 pack、pack2 包中的函数：

```
from pack import myModule
from pack2 import myModule2

myModule. func()
myModule2. func2()
```

- 第 1 行代码从 pack 包中导入 myModule 模块，myModule 模块被 main 模块调用，因此输出字符串“myModule 被另一个模块调用”。输出结果如下所示：

```
pack 初始化
myModule 被另一个模块调用
```

- 第 2 行代码从 pack2 包中导入 myModule2 模块。输出结果如下所示：

```
pack2 初始化
myModule2 被另一个模块调用
```

- 第 4 行代码调用 myModule 模块的函数 func( )。输出结果如下所示：

```
pack. myModule. func ()
```

- 第 5 行代码调用 myModule2 模块的函数 func2( )。输出结果如下所示：

```
pack2. myModule2. func ()
```

__init__. py 也可以用于提供当前包的模块列表。例如，在 pack 包的__init__. py 文件前面添加一行代码：

```
__all__ = ["myModule"]
```

__all__用于记录当前 pack 包所包含的模块。其中方括号中的内容是模块名的列表，如果模块数量超过两个，使用逗号分开。同理，在 pack2 包也添加一行类似的代码：

```
__all__ = ["myModule2"]
```

这样就可以在 main 模块中一次导入 pack、pack2 包中所有的模块。修改后的 main 模块如下所示：

```
from pack import *
from pack2 import *

myModule. func ()
myModule2. func2 ()
```

- 第 1 行代码，首先执行 pack 包的__init__. py 文件，然后在__all __属性中查找 pack 包含的模块。如果 pack 包的__init__. py 文件不使用__all __属性记录模块名，main 模块调用时将不能识别 myModule 模块。Python 将提示如下错误。

```
NameError: name 'myModule' is not defined
```

- 第 2 行代码与第 1 行代码的作用相同。

更多资源请访问稀酷客(www.ckook.com)

5.3　函数

函数是一段可以重复多次调用的代码，通过输入的参数值，返回需要的结果。前面的例子已经多次使用了 Python 的内置函数，而且还自定义了一些函数。Python 的函数有许多新的特性，下面将一一介绍。

5.3.1　函数的定义

函数的定义非常简单，使用保留字 def 定义。函数在使用前必须定义，函数的类型即返回值的类型。Python 函数定义的格式如下所示：

```
def 函数名(参数1[=默认值1],参数2[=默认值2]...):
    ...
return 表达式
```

函数名可以是字母、数字或下划线组成的字符串，但是不能以数字开头。函数的参数放在一对圆括号中，参数的个数可以有一个或多个，参数之间用逗号隔开，这种参数称之为形式参数。Python 支持形式参数的默认值语法，默认值的赋值是可选的。括号后面以冒号结束，冒号下面就是函数的主体。

【例 5-5】3.2.4 节使用字典实现了 switch 语句，现在把这段代码封装到函数中。它涉及 3 个参数：两个操作数、一个运算符。

```
# 函数的定义
from __future__ import division
def arithmetic(x, y, operator):
    result = {
        "+" : x + y,
        "-" : x - y,
        "* " : x * y,
        "/" : x / y
    }
```

- 第 3 行代码定义函数 arichmetic( )，x、y 是四则运算的两个操作数，operator 是运算符。这 3 个参数的值是从实际参数传递过来的。
- 第 4 行到第 9 行代码是函数的主体，实现了操作数的运算。

arichmetic( )已经创建成功，剩下的就是函数的调用的问题了。函数可以在当前文件中调用，也可以在其他模块中调用。函数调用的格式如下所示：

```
函数名(实参1,实参2...)
```

函数的调用采用函数名加一对圆括号的方式，圆括号内的参数是传递给函数的具体值。函数调用中的实参列表分别与函数定义中的形参列表对应。图 5-3 说明了实际参数和形式参数的对应关系。

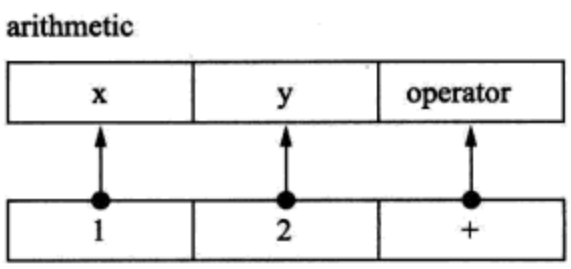


图 5-3　实际参数和形式参数的对应关系

更多资源请访问稀酷客(www.ckook.com)

arichmetic( )的调用如下所示：

```
# 函数的调用
print arithmetic(1, 2, " + ")
```

arichmetic( )写在 print 语句的后面，直接输出函数的返回值。输出结果：3。

注意　实际参数必须与形式参数一一对应，参数的顺序和参数的类型必须一致，否则将出现错误计算。如果参数提供默认值，顺序可以不一致。

5.3.2　函数的参数

在 C、C++ 中，参数的传递有值传递和引用传递两种方式。而 Python 中的任何变量都是对象，所以参数只支持引用传递的方式。Python 通过名字绑定的机制，把实际参数的值和形式参数的名称绑定在一起。即把形式参数传递到函数所在的局部命名空间中，形式参数和实际参数指向内存中的同一个存储空间。

函数的参数支持默认值。当某个参数没有传递实际的值时，函数将使用默认参数计算。例如可以给 arichmetic( )的参数都提供一个默认值：

```
# 函数的默认参数
def arithmetic(x = 1, y = 1, operator = " + "):
    result = {
        " + " : x + y,
        " - " : x - y,
        "* " : x * y,
        "/" : x / y
    }
    return result.get(operator)      # 返回计算结果

print arithmetic(1, 2)
print arithmetic(1, 2, " - ")
print arithmetic(y = 3, operator = " - ")
print arithmetic(x = 4, operator = " - ")
print arithmetic(y = 3, x = 4, operator = " - ")
```

- 第 2 行代码使用赋值表达式的方式定义参数的默认值。
- 第 11 行代码，参数 x、y 的值分别赋值为“1”、“2”，参数 operator 使用默认值“ + ”。输出结果：3。
- 第 12 行代码，提供了 3 个实际参数，这 3 个值将分别覆盖形式参数的默认值。输出结果：－1。
- 第 13 行代码，指定参数 y、operator 的值。输出结果：－2。这里必须使用赋值表达式的方式传递参数，否则，Python 解释器将误认为 x = 3、y = " - "。因此下面的写法是错误的。

```
print arithmetic(3, " - ")
```

- 第 14 行代码，指定参数 x、operator 的值。输出结果：3。
- 第 15 行代码，使用赋值表达式传递参数，可以颠倒参数列表的顺序。输出结果：1。

参数可以是变量，也可以是元组、列表等内置数据结构：

```
# 列表作为参数传递
def arithmetic(args = [], operator = " + "):
    x = args[0]
    y = args[1]
    result = {
        " + " : x + y,
```

更多资源请访问稀酷客(www.ckook.com)

```
        "-" : x - y,
        "* " : x * y,
        "/" : x / y
    }

print arithmetic([1, 2])
```

- 第2行代码把参数x、y合并为一个参数，通过args列表传递x、y的值。
- 第3行、第4行代码，从列表中取出参数值分别赋值给变量x、y。
- 第12行代码，把列表［1，2］传递给arichmetic()。输出结果：3。

由于参数实现了名字绑定的机制，所以在使用默认参数时，可能会出现预期之外的结果：

```
def append(args = []):
    args. append(0)
    print args

append()
append([1])
append()
```

- 第1行代码定义了一个append()函数，参数是一个带默认值的列表。
- 第2行代码在列表中追加一个元素“0”。
- 第5行代码调用append()，使用默认的列表。输出结果：[0]。
- 第6行代码，传递了一个列表［1］，append()中追加一个元素“0”。输出结果：[1，0]。
- 第7行代码再次调用append()，此时使用的列表还是第一次调用的args，因此args在原有的基础上将再次追加一个元素“0”。输出结果：[0，0]。

为了避免这个问题，可以在append()中添加一个条件判断语句。如果列表args中没有任何元素，则先把args列表置空，然后再添加元素：

```
def append(args = []):
    if len(args) <= 0:
        args = []
    args. append(0)
    print args

append()
append([1])
append()
```

- 第2行代码使用len()判断列表args的长度是否大于0。如果小于等于0，则把args置为空列表。即取消函数参数的绑定。
- 第4行代码在列表中追加一个元素“0”。
- 第7行代码调用append()，使用默认的列表。输出结果：[0]。
- 第8行代码，传递了一个列表［1］，append()中追加一个元素“0”。输出结果：[1，0]。
- 第9行代码调用append()，通过len（args）的判断，取消了参数的名字绑定。输出结果：[0]。

在程序的开发过程中，常常需要传递可变长度的参数。在函数的参数前使用标识符“*”可以实现这个要求。“*”可以引用元组，把多个参数组合到一个元组中：

```
#传递可变参数
def func(* args):
    print args
func(1, 2, 3)
```

更多资源请访问稀酷客(www.ckook.com)

- 第 2 行代码，在参数 args 前使用标识符“ * ”。
- 第 4 行代码调用函数 func( )。由于参数使用了“ * args”的形式，因此传入的实际参数被“打包”到一个元组中。其中的参数“1”、“2”、“3”成为 args 元组的元素。输出结果：(1, 2, 3)。

Python 还提供了另一个标识符“ * * ”。在形式参数前面添加“ * * ”，可以引用一个字典。根据实际参数的赋值表达式生成字典。例如，下面这段代码实现了在一个字典中匹配元组的元素。定义函数时，设计两个参数。一个是待匹配的元组，表示为“ * t”。另一个是字典，表示为“ * d”。函数调用时，实际参数分成两部分。一部分参数是若干个数字或字符串，另一部分参数是赋值表达式，如图 5-4 所示。

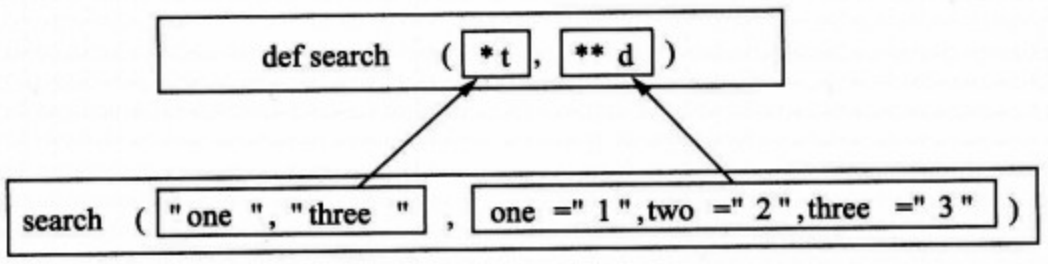


图 5-4　“ * ”、“ * * ”与实际参数的对应关系

【例 5-6】演示传递可变参数。

```
#传递可变参数
def search(* t, * * d):
    keys = d.keys ()
    values = d.values ()
    print keys
    print values
    for arg in t:
        for key in keys:
            if arg == key:
                print "find:",d[key]

search("one", "three", one = "1",two = "2",three = "3")
```

- 第 2 行代码中的“ * t”与第 12 行代码中的“one”、“three”对应。“one”、“three”组成一个元组 t。“ * * d”与 one = " 1" , two = " 2" , three = " 3" 对应，生成一个字典{one: " 1" , two: " 2" , three: " 3" }。
- 第 5 行代码输出结果：[three', 'two', 'one']。
- 第 6 行代码输出结果：['3', '2', '1']。
- 第 7 行到第 10 行代码在字典 d 中查找元组 t 中的值。如果找到，则输出。输出结果如下所示：

```
find: 1
find: 3
```

注意　“ * ”必须写在“ * * ”的前面，这是语法规定。

5.3.3　函数的返回值

函数的返回使用 return 语句，return 后面可以是变量或表达式。

【例 5-7】接【例 5-5】。为 arithmetic( ) 函数添加 return 语句，使其更加完善。

更多资源请访问稀酷客(www.ckook.com)

```
from __future__ import division
def arithmetic(x, y, operator):
    result = {
        "+" : x + y,
        "-" : x - y,
        "* " : x * y,
        "/" : x / y
    }
    return result.get(operator)      # 返回计算结果
```

- 第9行代码调用字典的get()，获得对应的表达式，并把计算后的结果返回。

在C和Java语言中，如果函数主体没有使用return语句返回，而在赋值语句中调用函数，程序编译后会出现错误。Python没有这个语法限制，即使函数没有返回值，依然可以获得返回值。例如：

```
# 没有return语句的函数返回None
def func():
    pass

print func()
```

- 第2行代码定义了一个函数func()，函数的主体没有任何实现代码，pass保留字相当于一个占位符。
- 第5行代码输出func()的返回值，因为没有return语句，所以返回值为“None”。输出结果：None。

“None”是Python中的一个对象，不属于数字也不属于字符串。当函数中的return语句不带任何参数时，返回的结果也是“None”：

```
def func ():
    return

print func ()
```

如果需要返回多个值，可以把这些值“打包”到元组中。在调用时，对返回的元组“解包”即可。下面这段代码实现了输入变量的反转。例如，输入0、1、2，返回2、1、0：

```
# return返回多个值
def func(x, y, z):
    l = [x, y, z]
    l.reverse()
    numbers = tuple(l)
    return numbers

x, y, z = func(0, 1, 2)
print x, y, z
```

- 第2行代码定义了一个函数func()，该函数对传入的3个参数反转后，返回这3个值。
- 第3行代码把3个参数“打包”到一个列表中。
- 第4行代码反转列表。
- 第5行代码把列表装到一个元组中。
- 第6行代码返回元组。即返回了3个数字。
- 第8行代码调用func()，获得返回的元组，并“解包”到3个变量中。
- 第9行代码输出3个变量的值。

稍微改进一下代码，还可以得到第二种解决办法：

更多资源请访问稀酷客(www.ckook.com)

```
def func(x, y, z):
    l = [x, y, z]
    l.reverse ()
    a, b, c = tuple(l)
    return a, b, c

x, y, z = func(0, 1, 2)
print x, y, z
```

- 第 4 行代码“解包”list，把反转后的值分别赋值给变量 a、b、c。
- 第 5 行代码，return 后可以跟逗号间隔的表达式，返回多个值。
- 第 7 行代码调用 func( )，把 a、b、c 分别赋值给 x、y、z。

在一个函数中也可以使用多个 return 语句。例如，if - else 语句的各个分支中，返回不同的结果：

```
# 多个 return 语句
def func(x):
    if x > 0:
        return "x > 0"
    elif x == 0:
        return "x == 0"
    else:
        return "x < 0"

print func( -2)
```

- 当传入的参数大于 0 时，返回“x > 0”；当传入的参数等于 0 时，返回“x == 0”；当传入的参数小于 0 时，返回“x < 0”。

注意　不推荐在一个函数中使用多个 return 语句，return 语句过多往往会造成程序的复杂，这时就需要对代码进行重构了。

如果函数中有多个 return 语句，可以通过增加一个变量的方法来减少 return 语句：

```
# 多个 return 语句的重构
def func(x):
    if x > 0:
        result = "x > 0"
    elif x == 0:
        result = "x == 0"
    else:
        result = "x < 0"
    return result

print func( -2)
```

- 第 4 行、第 6 行、第 8 行代码，增加了一个变量 result，通过赋值语句来记录程序分支的状况。
- 第 9 行代码返回 result 的值，这样就使得每个分支的结果都可以使用一个 return 语句返回。

5.3.4　函数的嵌套

函数的嵌套是指在函数的内部调用其他函数。C、C++ 只允许在函数体内部嵌套，Python 不仅支持函数体内的嵌套，还支持函数定义的嵌套。例如，计算表达式（x + y） * （m - n）的值。可以把计算步骤分为 3 步：先计算表达式 x + y，其次计算表达式 m - n，最后计算前面两步结果的乘积。因此，可以设计 3 个函数。第一个函数 sum( )计算 x + y 的值，第二个函数 sub( )计算 m - n 的值，第三个函数计算前面两者的乘积，如图 5-5 所示。

更多资源请访问稀酷客(www.ckook.com)

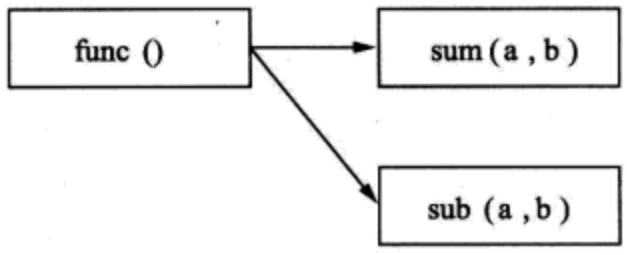


图5-5　函数的嵌套

【例5-8】演示函数之间的调用操作。

```
#嵌套函数
def sum(a, b):
    return a + b
def sub(a, b):
    return a - b
def func():
    x = 1
    y = 2
    m = 3
    n = 4
    return sum(x, y) * sub(m, n)

print func()
```

- 第2行代码定义了函数sum()，sum()带有a、b两个参数。参数a、b用于计算表达式x + y的值。
- 第3行代码计算a + b的值，即返回x + y的结果。
- 第4行代码定义了函数sub()，sub()带有a、b两个参数。
- 第5行代码计算a - b的值，即返回m - n的结果。
- 第11行代码，在return语句中调用sum()、sub()，并执行乘法运算。
- 第13行代码，调用函数func()。输出结果如下所示：

```
-3
```

注意　函数嵌套的层数不宜过多，否则容易造成代码的可读性差、不易维护等问题。一般函数的嵌套调用控制在3层以内为好。

上面这段代码也可以换一种形式实现，即把函数sum()、sub()放到func()的内部，如图5-6所示。

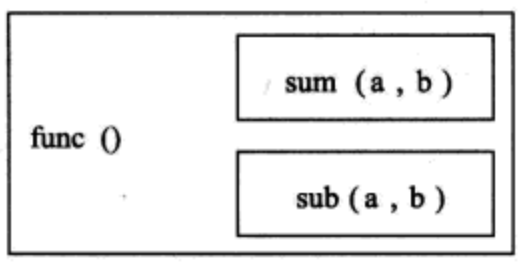


图5-6　在函数内部定义函数

【例5-9】演示在函数func()的内部定义sum()和sub()函数的方法。

```
#嵌套函数
def func():
    x = 1
    y = 2
    m = 3
```

更多资源请访问稀酷客(www.ckook.com)

```
    n = 4
    def sum(a, b):                  # 内部函数
        return a + b
    def sub(a, b):                  # 内部函数
        return a - b
    return sum(x, y) * sub(m, n)

print func ()
```

- 第 7 行代码，在 func( )内部定义了 sum( )。
- 第 9 行代码，在 func( )内部定义了 sub( )。
- 第 11 行代码，调用 sum( )和 sub( )，然后执行乘法运算。输出结果：-3。

内部函数 sum( )、sub( )也可以直接使用外部函数 func( )定义的变量，如图 5-7 所示。

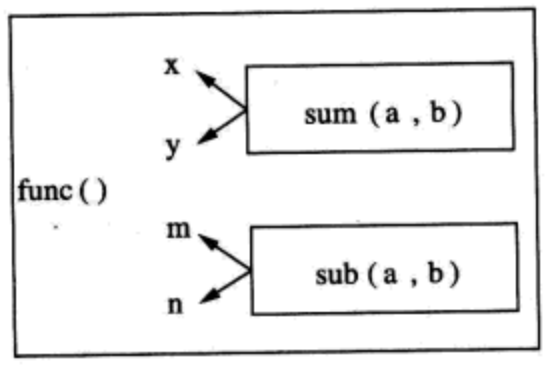


图 5-7　内部函数引用外部函数的变量

【例 5-10】演示在内部函数 sum( )和 sub( )中引用外部函数 func( )的变量。

```
# 嵌套函数,直接使用外层函数的变量
def func ():
    x = 1
    y = 2
    m = 3
    n = 4
    def sum () :                    # 内部函数
        return x + y
    def sub () :                    # 内部函数
        return m - n
    return sum () * sub ()

print func ()
```

- 第 7 行代码，函数 sum( )没有任何参数。
- 第 8 行代码，在 sum( )内部使用外部变量 x、y。
- 第 9 行代码，函数 sub( )也没有任何参数。
- 第 10 行代码，在 sub( )内部使用外部变量 m、n。
- 第 11 行代码，计算 sum( ) * sub( )的值。输出结果：-3。

注意　尽量不要在函数内部定义函数。这种方式不便于程序的维护，容易造成逻辑上的混乱。而且嵌套定义函数的层次越多，程序维护的代价越大。

5.3.5　递归函数

递归函数可以在函数主体内直接或间接地调用自己，即函数的嵌套是函数本身。递归是一种程序设计方法，使用递归可以减少重复的代码，使程序变得简洁。递归的过程分为两个阶段——递推和回归。递归函数的原理如下：

第一阶段，递归函数在内部调用自己。每一次函数调用又重新开始执行此函数的代码，直到某一级递归程序结束。

更多资源请访问稀酷客(www.ckook.com)

第二阶段，递归函数从后往前返回。递归函数从最后一级开始返回，直到返回到第一次调用的函数体内。即递归逐级调用完毕后，再按照相反的顺序逐级返回。

注意　递归函数需要编写递归结束的条件，否则递归程序将无法结束。一般通过判断语句来结束程序。

计算阶乘是一个经典的递归实现。首先，回顾一下阶乘的计算公式。

$$n! \begin{cases} n=1 & 1 \\ n>1 & n*(n-1)*(n-2)...*1 \end{cases}$$

例如，计算5！的结果。在设计程序时，可以根据n是否等于1进行判断。每次递归调用，传入参数n－1。直到n＝1时，返回1！等于1。再依次返回2！、3！、4！、5！的计算结果。图5-8演示了阶乘的计算过程。

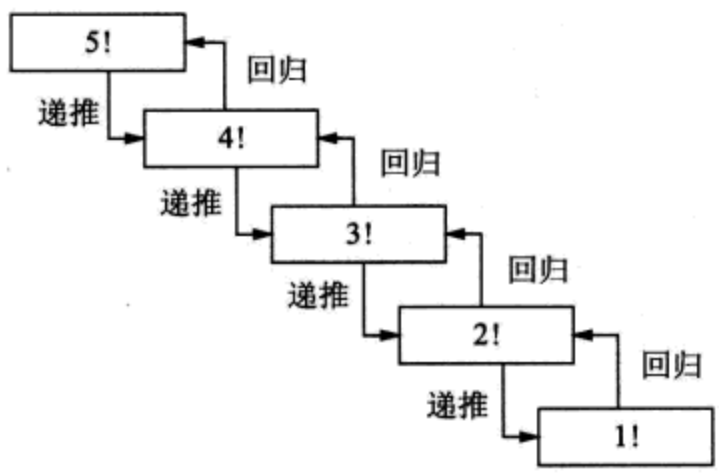


图5-8　使用递归计算阶乘的过程

【例5-11】用递归函数实现阶乘的计算过程。

```
# 计算阶乘
def refunc(n):
    i = 1
    if n > 1:                          # 判断递归是否结束
        i = n
        n = n * refunc(n-1)            # 递推
    print "% d! =" % i, n
    return n                           # 回归

refunc(5)
```

- 第2行代码定义了递归函数，递归函数的定义和普通函数没有什么区别。
- 第3行代码定义了一个变量i，用于print语句的输出。
- 第4行代码，对传递的参数n进行判断。如果n大于1，则函数可以继续递推下去，否则就返回当前计算的结果。
- 第5行代码把n的值赋值给i，用i来记录当前递推的数字。
- 第6行代码调用函数refunc()自身，传递参数n－1。
- 第7行代码输出阶乘的计算结果。
- 第8行代码返回每一级阶乘的计算结果。
- 第10行代码调用递归函数refunc。每次递归的输出结果如下所示：

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

更多资源请访问稀酷客(www.ckook.com)

注意　每次调用递归函数都会复制函数中所有的变量，再执行递归函数。程序需要较多的存储空间，这对程序的性能会有一定的影响。因此，对于没有必要进行递归的程序，最好用其他的方法进行改进。

可以使用前面提到的 reduce( )快速实现阶乘的运算：

```
# 使用 reduce 计算阶乘
print "5! =", reduce(lambda x, y: x * y, range(1, 6))
```

使用 recude( ) 只需要一行代码就可以计算出 5!。

5.3.6　lambda 函数

lambda 函数用于创建一个匿名函数，函数名未和标识符进行绑定。使用 lambda 函数可以返回一些简单的运算结果。lambda 函数的格式如下所示：

```
lambda 变量 1,变量 2... : 表达式
```

其中，变量列表用于表达式的计算。lambda 属于函数，因此变量列表后需要一个冒号。通常把 lambda 赋值给一个变量，变量就可作为函数使用。例如：

```
# 赋值
func = lambda 变量 1,变量 2.., : 表达式
# 调用
func ()
```

这样就把 lambda 和变量 func 绑定在一起了，变量 func 的名字就是函数名。

【例 5-12】接【例 5-10】。对【例 5-10】计算（x + y） * （m - n）的程序进行改写。把其中的函数 sum( )和 sub( )用 lambda 函数来替代。

```
# lambda
def func ():
    x = 1
    y = 2
    m = 3
    n = 4
    sum = lambda x, y : x + y
    print sum
    sub = lambda m, n : m - n
    print sub
    return sum(x, y) * sub(m, n)

print func ()
```

- 第 7 行代码定义了 lambda 函数，实现计算表达式 x + y。并把 lambda 函数赋值给变量 sum。
- 第 8 行代码输出变量 sum 的值，sum 保存了 lambda 函数的地址。输出结果如下所示。

```
<function <lambda> at 0x00B4D3B0>
```

- 第 9 行、第 10 行代码的作用与第 7 行、第 8 行的原理相同。
- 第 11 行代码计算 sum( )与 sub( )的乘积。输出结果：-3。

注意　lambda 也称之为表达式。lambda 中只能使用表达式，不能使用判断、循环等多重语句。

【例 5-12】中是把 lambda 赋值给一个变量使用，也可以把 lambda 直接作为函数使用：

```
# lambda 的函数用法
print (lambda x: -x)(-2)
```

更多资源请访问稀酷客(www.ckook.com)

第2行代码定义了匿名函数 lambda x：-x，用于返回数字的绝对值。函数的参数为-2，输出结果：2。

5.3.7　Generator 函数

生成器（Generator）的作用是一次产生一个数据项，并把数据项输出。Generator 函数可以用在 for 循环中遍历。Generator 函数所具有的每次返回一个数据项的功能，使得迭代器的性能更佳。Generator 函数的定义如下所示：

```
def 函数名(参数列表):
    ...
    yield 表达式
```

Generator 函数的定义和普通函数的定义没有什么区别，只要在函数体内使用 yield 生成数据项即可。Generator 函数可以被 for 循环遍历，而且可以通过 next()方法获得 yield 生成的数据项。

【例 5-13】演示 Generator 函数的使用方法。

```
# 定义 Generator 函数
def func(n):
    for i in range(n):
        yield i
# 在 for 循环中输出
for i in func(3):
    print i
# 使用 next () 输出
r =  func(3)
print r.next ()
print r.next ()
print r.next ()
print r.next ()
```

- 第2行代码定义了 Generator 函数，参数 n 表示一个数字，该函数依次生成0到 n 个数字。
- 第4行代码调用 yield 生成数字。
- 第6行代码遍历函数 func()，依次输出 yield 生成的数字。输出结果：

```
0
1
2
```

- 第9行代码把 func()赋值给列表 r。
- 第10行到第12行代码，输出 r. next()的值。输出结果：

```
0
1
2
```

- 第13行代码再次调用 r. next()。由于 n 的值等于“3”，因此生成3个数字后，已经没有数据可生成。Python 解释器抛出异常 StopIteration。

yield 保留字与 return 语句的返回值和执行原理都不相同。yield 生成值并不会中止程序的执行，返回值后程序继续往后执行。return 返回值后，程序将中止执行。

【例 5-14】演示 yield 和 return 语句的区别。

```
#yield 与 return 区别
def func(n):
    for i in range(n):
        return i
```

更多资源请访问稀酷客(www.ckook.com)

```
def func2(n):
    for i in range(n):
        yield i

print func(3)
f = func2(3)
print f
print f.next()
print f.next()
```

- 第 4 行代码直接返回 i 的值，循环语句将被终止，整个程序到此结束。
- 第 7 行代码循环生成 n 个数字，循环语句不会被终止。
- 第 9 行代码的输出结果：0。
- 第 11 行代码，yield 并没有返回任何值，而是返回了函数 func2( )的地址。输出结果：< generator object at 0x00B4C5D0 > 。
- 第 12 行代码，调用 f. next( )才能获得返回值。输出结果：0。
- 第 13 行代码的输出结果：1。

Generator 函数可以返回元素的值，而序列也可以获取元素的值。但是两者还是存在很大的区别。Generator 函数一次只返回一个数据项，占用更少的内存。每次生成数据都要记录当前的状态，便于下一次生成数据。数据的访问是通过 next( )方法实现的。当访问越界时，Generator 函数会抛出异常 StopIteration。序列一次返回所有的数据，元素的访问是通过索引完成的。当访问越界时，序列提示 list index out of range 错误。

当程序需要较高的性能或一次只需要一个值进行处理时，使用 Generator 函数。当需要获取一次性一组元素的值时，使用序列。

5.4　小结

本章介绍了包、模块和函数之间的逻辑关系、Python 程序的组成结构、模块和函数的使用方法。重点讲解了模块的内置函数、常用模块的使用方法、函数的特性等知识。Python 的参数传递使用引用传递的方式，并且支持返回多个值。递归函数是经常使用到的函数调用方法，Python 中还可以使用 generator 函数替代改造递归函数，提高递归的性能。lambda 函数是一个非常实用的工具，它可以定义一些小函数，将它们用于表达式或函数的返回值中非常方便。

实际应用中，许多任务都是由若干个函数组成的。这些函数把任务细化，实现明确的功能，最后把这些函数组合起来形成一个完整的模块。

更多资源请访问稀酷客(www.ckook.com)

第 6 章　字符串与正则表达式

本章将介绍 Python 中字符串和正则表达式的概念。字符串是程序开发中常用的数据类型，字符串的处理是实际应用中经常面对的问题。Python 提供了功能强大的字符串模块，本章将详细介绍字符串模块中函数的使用方法。正则表达式专门用于匹配应用中的数据，能够简化字符串的处理程序，Python 提供了 re 模块用来匹配正则表达式。最后将介绍字符串函数和正则表达式的应用场合。本章的知识点：

- 字符串的格式化
- 字符串的截取、合并、过滤等操作
- 字符串的查找
- 正则表达式的语法
- Python 的正则表达式模块

6.1　字符串的操作

字符串是 Python 的一种基本类型，字符串的操作包括字符串的格式化、字符串的截取、过滤、合并、查找等操作。下面将一一介绍这些内容。

6.1.1　字符串的格式化

C 语言使用函数 printf( )、sprintf( ) 格式化输出结果，Python 也提供了类似的功能。Python 将若干值插入到带有“%”标记的字符串中，从而可以动态的输出字符串。字符串的格式化语法如下所示：

```
"% s" % str1
"% s % s" % (str1, str2)
```

- 上面的代码中，第 1 行代码使用一个值格式化字符串。第 2 行代码使用多个值格式化字符串，用于替换的值组成一个元组。

【例 6-1】演示字符串的格式化操作。

```
# 格式化字符串
str1 = "version"
num = 1.0
format = "% s" % str1
print format
format = "% s % d" % (str1, num)
print format
```

- 第 4 行代码将变量 str1 的值替换字符串中的% s。
- 第 5 行代码输出结果：version。
- 第 6 行代码，分别用变量 str1、num 的值替换% s 和% d 的值。% d 表示替换的值为整型。

更多资源请访问稀酷客(www.ckook.com)

- 第 7 行代码输出结果：version 1。

注意　如果要格式化多个值，元组中元素的顺序必须和格式化字符串中替代符的顺序一致，否则可能出现类型不匹配的问题。如果将上例中的%s 和%d 调换位置，将抛出如下异常：

```
TypeError: int argument required
```

使用%f 可以格式化浮点数的精度，根据指定的精度做“四舍五入”。示例如下：

```
# 带精度的格式化
print "浮点型数字：%f" % 1.25
print "浮点型数字：%.1f" % 1.25
print "浮点型数字：%.2f" % 1.254
```

- 第 2 行代码格式化浮点数 1.25。默认情况下，将输出小数点后 6 位数字。输出结果：浮点型数字：1.250000。
- 第 3 行代码格式化小数点后 1 位数字，“四舍五入”后的结果为 1.3。输出结果：浮点型数字：1.3。
- 第 4 行代码格式化小数点后 2 位数字。输出结果：浮点型数字：1.25。

此外，Python 还提供了对八进制、十六进制等数字进行格式化的替代符。表 6-1 列出了 Python中格式化字符串的替代符及其含义。

表 6-1　Python 格式化字符串的替代符及其含义

符　号	说　明
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整型
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	根据值的大小决定使用%f 或%e
%G	作用同%g，根据值的大小决定使用%f 或%e
%p	用十六进制数格式化变量的地址

注意　如果要在字符串中输出“%”，则需要使用“%%”。

前面使用了元素格式化多个值，也可以用字典格式化多个值，并指定格式化字符串的名称。下面这段代码说明了字典格式化字符串的用法：

```
# 使用字典格式化字符串
print "%(version)s：%(num).1f" % {"version": "version", "num": 2}
```

- 第 2 行代码，定义了一个匿名字典，该字典中的两个 value 值分别对应字符串中的%（version）s 和%（num）.1f。输出结果：version：2.0。

Python 可以实现字符串的对齐操作，类似 C 语言中的“%［［+/-］n］s”。此外，还提供了用于字符串对齐的函数：

更多资源请访问稀酷客(www.ckook.com)

```
#字符串对齐
word = "version3.0"
print word.center(20)
print word.center(20, "* ")
print word.ljust(0)
print word.rjust(20)
print "%30s" % word
```

- 第3行代码，调用center()输出变量word的值。变量word两侧各输出5个空格。输出结果：

```
     version3.0
```

- 第4行代码，调用center()输出变量word的值，并指定第2个参数的值为“*”。变量word两侧各输出5个“*”。输出结果：

```
* * * * * version3.0* * * * *
```

- 第5行代码，调用ljust()输出变量word的值，ljust()输出结果左对齐。输出结果：

```
version3.0
```

- 第6行代码，调用rjust()输出变量word的值，rjust()输出结果右对齐。参数“20”表示一共输出20个字符，“version3.0”占用10个字符，左边填充10个空格。输出结果：

```
          version3.0
```

- 第7行代码，“%30s”表示先输出20个空格，再输出变量word的值。类似于word.rjust(30)的作用。输出结果：

```
                    version3.0
```

6.1.2　字符串的转义符

计算机中存在可见字符与不可见字符。可见字符是指键盘上的字母、数字和符号。不可见字符是指换行、回车等字符，对于不可见字符可以使用转义字符来表示。Python中转义字符的用法和Java相同，都是使用“\”作为转义字符。

【例6-2】演示转义字符的使用方法。

```
#输出转义字符
path = "hello\tworld\n"
print path
print len(path)
path = r"hello\tworld\n"
print path
print len(path)
```

- 第2行代码，在“hello”和“world”之间输出制表符，在字符串末尾输出换行符。
- 第3行代码输出结果：

```
hello world
```

- 第4行代码输出字符串的长度，其中的“\t”、“\n”各占一个字符。输出结果：12。
- 第5行代码，忽略转义字符的作用，直接输出字符串原始的内容。
- 第6行代码输出结果：hello\tworld\n。
- 第7行代码输出字符串的长度。输出结果：14。

注意　Python的制表符只占1个字符的位置，而不是2个或4个字符的位置。

更多资源请访问稀酷客(www.ckook.com)

Python 支持的转义字符如表 6-2 所示。

表 6-2　Python 的转义字符及其含义

符　号	说　明
\ '	单引号
\ "	双引号
\ a	发出系统响铃声
\ b	退格符
\ n	换行符
\ t	横向制表符
\ v	纵向制表符
\ r	回车符
\ f	换页符
\ o	八进制数代表的字符
\ x	十六进制数代表的字符
\ 000	终止符，\ 000 后的字符串全部忽略

注意　如果要在字符串中输出“ \ ”，需要使用“ \\ ”。

Python 还提供了函数 strip( )、lstrip( )、rstrip( )去掉字符串中的转义符：

```
# strip () 去掉转义字符
word = "\thello world\n"
print "直接输出:", word
print "strip () 后输出:", word.strip ()
print "lstrip () 后输出:", word.lstrip ()
print "rstrip() 后输出:", word.rstrip ()
```

- 第 3 行代码直接输出字符串。输出结果如下所示：

```
直接输出:    hello world
```

- 第 4 行代码调用 strip( )去除转义字符。输出结果：strip( )后输出：hello world。
- 第 5 行代码调用 lstrip( )去除字符串前面的转义字符“ \ t”，字符串末尾的“ \ n”依然存在。输出结果：

```
lstrip () 后输出: hello world
```

- 第 6 行代码调用 rstrip( )去除字符串末尾的转义字符“ \ n”，字符串前面的“ \ t”依然存在。输出结果：

```
rstrip () 后输出:    hello world
```

6.1.3　字符串的合并

与 Java 语言一样，Python 使用“ + ”连接不同的字符串。Python 会根据“ + ”两侧变量的类型，决定执行连接操作或加法运算。如果“ + ”两侧都是字符串类型，则进行连接操作；如果“ + ”两侧都是数字类型，则进行加法运算；如果“ + ”两侧是不同的类型，将抛出异常：

```
TypeError: cannot concatenate 'str' and 'int' objects
```

【例 6-3】演示字符串的连接方法。

更多资源请访问稀酷客(www.ckook.com)

```
#使用"+"连接字符串
str1 = "hello "
str2 = "world "
str3 = "hello "
str4 = "China "
result = str1 + str2 + str3
result + = str4
print result
```

- 第6行代码，把变量str1、str2、str3的值连接起来，并把结果存放在变量result中。
- 第7行代码，使用运算符“+=”连接变量result和str4。
- 第8行代码输出结果：hello world hello China。

可见，使用“+”对多个字符串进行连接稍显繁琐。Python提供了函数join()连接字符串，join()配合列表实现多个字符串的连接十分方便：

```
#使用join()连接字符串
strs = ["hello ", "world ", "hello ", "China "]
result = "".join(strs)
print result
```

- 第2行代码，用列表取代变量，把多个字符串存放在列表中。
- 第3行代码调用join()，依次连接列表中的一个元素。
- 第4行代码输出结果：hello world hello China。

在第5章的5.2.4节学习了reduce()函数，reduce()的作用就是对某个变量进行累计。这里可以对字符串进行累计连接，从而实现多个字符串进行连接的功能：

```
#使用reduce()连接字符串
import operator
strs = ["hello ", "world ", "hello ", "China "]
result = reduce(operator.add, strs, "")
print result
```

- 第2行代码导入模块operator，利用方法add()实现累计连接。
- 第4行代码调用reduce()实现对空字符串“”的累计连接，依次连接列表strs中的一个元素。
- 第5行代码输出结果：hello world hello China。

6.1.4　字符串的截取

字符串的截取是实际应用中经常使用的技术，被截取的部分称之为“子串”。Java中使用函数substr()获取子串，C#使用函数substring()获取子串。Python由于内置了序列，因此可以通过前面介绍的“索引”、“切片”获取子串，也可以使用函数split()来获取。字符串也属于序列，下面这段代码使用序列的索引获取子串：

```
#使用索引截取子串
word = "world"
print word[4]
```

- 第3行代码，访问字符串第5个字符的值。输出结果：d。

通过切片可以实现对字符串有规律的截取。切片的语法格式如下所示：

```
string[start : end : step]
```

更多资源请访问稀酷客(www.ckook.com)

其中，string 表示需要取子串的源字符串变量；［start ： end ： step］表示从 string 的第 start 个索引位置开始到第 end 个索引之间截取子串，截取的步长是 step。即每次截取字符 string［start + step］，直到第 end 个索引。索引从 0 开始计数。

【例 6-4】演示使用切片截取子串的方法。

```
# 特殊切片截取子串
word = "world"
str1 = "hello world"
print word [0:3]
print str1[::2]
print str1[1::2]
```

- 第 4 行代码，截取字符串中第 1 个字符到第 3 个字符之间的部分。输出结果：wor。
- 第 5 行代码，［:: 2］切片省略了开始和结束字符。从字符串的第 1 个字符开始，以 2 为步长逐个截取字符。输出结果：hlowrd。
- 第 6 行代码，切片中的数字“1”表示从字符串的第 2 个字符开始取字符，数字“2”表示以 2 为步长逐个截取字符。输出结果：el ol。

如果要同时截取多个子串，可以使用函数 split( )实现。函数 split( )的声明如下所示：

```
split([char] [,num])
```

- 参数 char 表示用于分割的字符，默认的分割字符是空格。
- 参数 num 表示分割的次数。如果 num 等于“2”，将把源字符串分割为 3 个子串。默认情况下，将根据字符 char 在字符串中出现的个数来分割子串。
- 函数的返回值是由子串组成的列表。

【例 6-5】演示 split( )的使用方法。

```
# 使用 split () 获取子串
sentence = "Bob said: 1, 2, 3, 4"
print "使用空格取子串:", sentence. split()
print "使用逗号取子串:", sentence. split(",")
print "使用两个逗号取子串:", sentence. split(",", 2)
```

- 第 3 行代码，根据空格来获取子串。字符串 sentence 中有 5 个空格，将返回由 6 个子串组成的列表。输出结果：使用空格取子串：['Bob', 'said：', '1，', '2，', '3，', '4']。
- 第 4 行代码，根据逗号来获取子串。字符串 sentence 中有 3 个逗号，将返回由 4 个子串组成的列表。输出结果：使用逗号取子串：['Bob said：1', ' 2', ' 3', ' 4']。
- 第 5 行代码，根据逗号来分割字符串，并把字符串 sentence 分割为 3 个子串。输出结果：使用两个逗号取子串：['Bob said：1', ' 2', '3，4']。

字符串连接后，Python 将分配新的空间给连接后的字符串，源字符串保持不变：

```
str1 = "a"
print id(str1)
print id(str1 + "b")
```

- 第 2 行代码输出 str1 的内部标识。输出结果：12144224。
- 第 3 行代码，进行字符串连接，新的字符串将获得新的标识。输出结果：12486880。

6.1.5　字符串的比较

Java 使用 equals( )比较两个字符串的内容，Python 直接使用“ == ”、“ ! = ”运算符比较两个字符串的内容。如果比较的两个变量的类型不相同，比较的内容也不相同。

更多资源请访问稀酷客(www.ckook.com)

【例 6-6】演示 Python 中字符串的比较。

```
# 字符串的比较
str1 = 1
str2 = "1"
if str1 == str2:
    print "相同"
else:
    print "不相同"
if str(str1) == str2:
    print "相同"
else:
    print "不相同"
```

- 第 2 行代码定义了 1 个数字类型的变量 str1。
- 第 3 行代码定义了 1 个字符串类型的变量 str2。
- 第 4 行代码比较 str1 和 str2 的值。由于 str1 和 str2 的类型不同，所以两者的内容也不相同。输出结果：不相同。
- 第 8 行代码，把数字型的变量 str1 转换为字符串类型，数字 1 被转换为字符串“1”。然后再与 str2 进行比较。输出结果：相同。

如果要比较字符串中的一部分内容，可以先截取子串，再使用“==”运算符进行比较。如果要比较字符串的开头或结尾部分，更方便的方法是使用 startswith( ) 或 endswith( ) 函数。startswith( ) 的声明如下所示：

```
startswith(substring, [,start [,end]])
```

- 参数 substring 是与源字符串开头部分比较的子串。
- 参数 start 表示开始比较的位置。
- 参数 end 表示比较结束的位置，即在“start：end”范围内搜索子串 substring。
- 如果字符串以 substring 开头，则返回“True”，否则返回“False”。

endswith( ) 的参数和返回值类似 startswith( )，不同的是 endswith( ) 从源字符串的尾部开始搜索。

【例 6-7】演示 startswith( ) 和 endswith( ) 的使用方法。

```
# 比较字符串的开始和结束处
word = "hello world"
print "hello" == word[0:5]
print word.startswith("hello")
print word.endswith("ld", 6)
print word.endswith("ld", 6, 10)
print word.endswith("ld", 6, len(word))
```

- 第 3 行代码，先获取子串［0：5］，再与“hello”进行比较。输出结果：True。
- 第 4 行代码调用 startswith( )。比较字符串变量 word 的开头部分“hello”。输出结果：True。
- 第 5 行代码，从字符串变量 word 的结尾到 word［6］之间搜索子串“ld”。输出结果：True。
- 第 6 行代码，从“分片”word［6：10］中搜索子串“ld”。由于搜索的字符不包括位置“10”所在的字符，所以在 word［6：10］中搜索不到子串“ld”。输出结果：False。
- 第 7 行代码，从“分片”word［6：len（word）］中搜索子串“ld”，len（word）的值为“11”。输出结果：True。

注意　startswith( )、endswith( ) 相当于分片［0：n］，n 是源字符串中最后一个索引。startswith( )、endswith( ) 不能用于比较源字符串中任意一部分的子串。

更多资源请访问稀酷客(www.ckook.com)

6.1.6　字符串的反转

字符串反转是指把字符串中最后一个字符移到字符串的第一个位置，按照倒序的方式依次前移。Java 中使用 StringBuffer 类处理字符串，并通过循环对字符串进行反转。

【例 6-8】用 Java 语言实现一个字符串反转的函数。

```
//Java 实现字符串反转
public static String reverse(String s)
{
    int length = s.length();
    StringBuffer result = new StringBuffer(length);
    for(int i = length - 1;i >= 0;i --)
        result.append(s.charAt(i));
    return result.toString();
}
```

Python 没有提供对字符串进行反转的函数，也没有类似 charAt( ) 这样的函数。但是可以使用列表和字符串索引来实现字符串的反转，并通过 range( ) 进行循环：

```
# 循环输出反转的字符串
def reverse(s):
    out = ""
    li = list(s)
    for i in range(len(li), 0, -1):
        out += "".join(li[i-1])
    return out
```

- 第 2 行代码定义了一个函数 reverse( )，参数 s 表示需要反转的字符串。
- 第 3 行代码定义了一个返回变量 out，用于存放字符串反转后的结果。
- 第 4 行代码创建了一个列表 li，字符串 s 中的字符成为列表 li 中的元素。
- 第 5 行到第 6 行代码，从列表中的最后一个元素开始处理，依次连接到变量 out 中。
- 第 7 行代码返回变量 out 的值。

函数 reverse( ) 的调用形式如下所示：

```
print reverse("hello")
```

- 上面的代码输出 reverse( ) 的返回值，输出结果：olleh。

不难发现，Python 的实现代码更简短，而且更容易理解。用户还可以通过列表的 reverse( ) 函数实现字符串的反转，实现的代码将进一步简化：

```
# 使用 list 的 reverse()
def reverse(s):
    li = list(s)
    li.reverse()
    s = "".join(li)
    return s
```

- 第 4 行代码调用列表的 reverse( ) 实现了 for... in... 循环的功能。
- 第 5 行代码调用 join( ) 把反转后列表 li 的元素依次连接到变量 s 中。

Python 的列表是对字符串进行处理的常用方式，灵活使用列表等内置数据结构处理字符串，能够降低编程的复杂度。利用序列的“切片”实现字符串的反转最为简洁，reverse( ) 函数的主体只需要一行代码即可：

```
def reverse(s):
    return s[::-1]
```

更多资源请访问稀酷客(www.ckook.com)

- -1 表示从字符串最后一个索引开始倒序排列。

6.1.7　字符串的查找和替换

Python 提供了一些字符串的查找和替换函数，这些函数的使用方法和 Java 类似，只是函数签名稍有不同而已。

1. 查找字符串

Java 中字符串的查找使用函数 indexOf()，返回源字符串中第 1 次出现子串的索引。如果需要从右往左查找可以使用函数 lastIndexOf()。Python 也提供了类似功能的函数，函数 find()与 Java 中 indexOf()的作用相同，rfind()与 Java 中 lastIndexOf()的作用相同。find()的声明如下所示：

```
find(substring [, start [ ,end]])
```

- 参数 substring 表示待查找的子串。
- 参数 start 表示开始搜索的索引位置。
- 参数 end 表示结束搜索的索引位置，即在分片［start：end］中进行查找。
- 如果找到字符串 substring，则返回 substring 在源字符串中第 1 次出现的索引，否则返回 -1。

rfind()的参数与 find()相同，不同的是 rfind()从字符串的尾部开始查找子串。

【例 6-9】演示 find()、rfind()的使用方法。

```
# 查找字符串
sentence = "This is a apple."
print sentence.find("a")
sentence = "This is a apple."
print sentence.rfind("a")
```

- 第 3 行代码，使用函数 find()，从 sentence 的头部开始查找字符串“a”。输出结果：8。
- 第 5 行代码，使用函数 rfind()，从 sentence 的尾部开始查找字符串“a”。输出结果：10。

2. 替换字符串

Java 使用 replaceFirst()、replaceAll()函数实现字符串的替换。replaceFirst()用于替换源字符串中第 1 次出现的子串，replaceAll()用于替换源字符串中所有出现的子串。这两个函数通过正则表达式来查找子串。而 Python 使用函数 replace()实现字符串的替换，该函数可以指定替换的次数，相当于是 Java 函数 replaceFirst()和 replaceAll()的合并。但是 replace()不支持正则表达式的语法。replace()的声明如下所示：

```
replace(old, new [, max])
```

- 参数 old 表示将被替换的字符串。
- 参数 new 表示替换 old 的字符串。
- 参数 max 表示使用 new 替换 old 的次数。
- 函数返回一个新的字符串。如果子串 old 不在源字符串中，则函数返回源字符串的值。

【例 6-10】演示 replace()的使用方法。

```
# 字符串的替换
centence = "hello world, hello China"
print centence.replace("hello", "hi")
print centence.replace("hello", "hi", 1)
print centence.replace("abc", "hi")
```

- 第 3 行代码，把 centence 中的“hello”替换为“hi”。由于没有给出参数 max 的值，所

更多资源请访问稀酷客(www.ckook.com)

以 centence 中的“hello”都将被“hi”替换。输出结果：hi world，hi China。

- 第4行代码，参数 max 的值为“1”，所以 centence 中第1次出现的“hello”被“hi”替换，后面出现的子串“hello”保持不变。输出结果：hi world，hello China。
- 第5行代码，由于 centence 中没有子串“abc”，所以替换失败。replace( )返回 centence 的值。输出结果：hello world，hello China。

注意　replace( )先创建变量 centence 的拷贝，然后在拷贝中替换字符串，并不会改变变量 centence 的内容。

6.1.8　字符串与日期的转换

在程序开发中，经常把日期类型转换为字符串类型使用。字符串与日期的转换是工作中频繁遇到的问题。Java 提供了 SimpleDateFormat 类实现日期到字符串的转换。Python 提供了 time 模块处理日期和时间。

1. 从时间到字符串的转换

time 模块中的函数 strftime( )可以实现从时间到字符串的转换。strftime( )的声明如下所示：

```
strftime(format[, tuple]) -> string
```

- 参数 format 表示格式化日期的特殊字符。例如，“%Y-%m-%d”相当于 Java 中的“yyyy-MM-dd”。
- 参数 tuple 表示需要转换的时间，用元组存储。元组中的元素分别表示年、月、日、时、分、秒。
- 函数返回一个表示时间的字符串。

参数 format 格式化日期的常用标记如表 6-3 所示。

表 6-3　格式化日期的常用标记

符号	说　明
%a	英文星期的简写
%A	英文星期的完整拼写
%b	英文月份的简写
%B	英文月份的完整拼写
%c	显示本地的日期和时间
%d	日期数，取值在1~31之间
%H	小时数，取值在00~23之间
%I	小时数，取值在01~12之间
%m	月份，取值在01~12之间
%M	分钟数，取值在01~59之间
%j	显示从本年第1天开始到当天的天数
%w	显示今天是星期几，0表示星期天
%W	显示当天属于本年的第几周，星期一作为一周的第一天进行计算
%x	本地的当天日期
%X	本地的当天时间
%y	年份，取值在00~99之间
%Y	年份完整的拼写

更多资源请访问稀酷客(www.ckook.com)

2. 字符串到时间的转换

字符串到时间的转换需要进行两次转换，需要使用time模块和datetime类，转换过程分为如下3个步骤。

（1）调用函数strptime()把字符串转换为一个元组，进行第1次转换。strptime()的声明如下所示：

```
strptime(string, format) -> struct_time
```

- 参数string表示需要转换的字符串。
- 参数format表示日期时间的输出格式。
- 函数返回一个存放时间的元组。

（2）把表示时间的元组赋值给表示年、月、日的3个变量。

（3）把表示年、月、日的3个变量传递给函数datetime()，进行第2次转换。datetime类的datetime()函数如下所示：

```
datetime(year,month,day[,hour[,minute[,second[,microsecond[,tzinfo]]]]])
```

- 参数year、month、day分别表示年、月、日，这3个参数必不可少。
- 函数返回1个datetime类型的变量。

【例6-11】 演示时间到字符串、字符串到时间的转换过程。

```
import time,datetime

# 时间到字符串的转换
print time.strftime("%Y-%m-%d %X", time.localtime())
# 字符串到时间的转换
t = time.strptime("2008-08-08", "%Y-%m-%d")
y, m, d = t[0:3]
print datetime.datetime(y, m, d)
```

- 第4行代码，函数localtime()返回当前的时间（例如，2008-02-14 13：52：11），strftime把当前的时间格式化为字符串类型。输出结果：2008-02-14 13：52：11。
- 第6行代码，把字符串“2008-08-08”转换为一个元组返回。
- 第7行代码，把元组中前3个表示年、月、日的元素赋值给3个变量。
- 第8行代码，调用datetime()返回时间类型。输出结果：2008-08-08 00：00：00。

注意　格式化日期的特殊标记是区分大小写的，%Y与%y并不相同，%Y、%y的含义参考表6-3。

6.2　正则表达式

正则表达式用于搜索、替换和解析字符串。正则表达式的功能强大，使用非常灵活。使用正则表达式需要遵循一定的语法规则，用来编写一些逻辑验证非常方便，例如电子邮件的验证。Python提供了re模块实现正则表达式的验证。

6.2.1　正则表达式简介

正则表达式是用于文本匹配的工具，在源字符串中查找与给定的正则表达式相匹配的部分。一个正则表达式是由字母、数字和特殊字符（括号、星号、问号等）组成的。正则表达式中有许多特殊的字符，这些特殊字符是构成正则表达式的要素。表6-4说明了正则表达式中特殊字符的含义。

更多资源请访问稀酷客(www.ckook.com)

表 6-4　正则表达式中的特殊字符

符　号	说　明
^	正则表达式的开始字符
$	正则表达式的结束字符
\ w	匹配字母、数字、下划线
\ W	匹配不是字母、数字、下划线的字符
\ s	匹配空白字符
\ S	匹配不是空白的字符
\ d	匹配数字
\ D	匹配非数字的字符
\ b	匹配单词的开始和结束
\ B	匹配不是单词开始和结束的位置
.	匹配任意字符，包括汉字
[m]	匹配单个字符串
[m1m2... n]	匹配多个字符串
[m-n]	匹配 m 到 n 区间内的数字、字母
[^m]	匹配除 m 以外的字符串
()	对正则表达式进行分组，一对圆括号表示一组

其中，匹配符“[ ]”可以指定一个匹配范围，例如“[ok]”将匹配包含“o”或“k”的字符。同时“[ ]”可以与 \ w、\ s、\ d 等标记等价。例如，[0 - 9a - zA - Z_] 等价于 \ w，[^0 - 9] 等价于 \ D。

注意　^与 [^m] 中的“^”的含义并不相同，后者的“^”表示“除了...”的意思。

如果要匹配电话号码，需要形如“\ d\ d\ d\ d - \ d\ d\ d\ d\ d\ d\ d”这样的正则表达式。其中出现了 11 次“\ d”，表达方式繁琐。而且某些地区的电话号码是 8 位数，区号也有可能是 3 位或 4 位数字，因此这个正则表达式就不能满足要求了。正则表达式作为一门小型的语言，还提供了对表达式的一部分进行重复处理的功能。例如，“*”可以对正则表达式的某个部分重复匹配多次。这种匹配符号称之为限定符。表 6-5 列出了正则表达式中常用的限定符。

表 6-5　正则表达式中的常用限定符

符　号	说　明
*	匹配零次或多次
+	匹配一次或多次
?	匹配一次或零次
{m}	重复 m 次
{m, n}	重复 m 到 n 次。其中 n 可以省略，表示 m 到任意次

利用 {} 可以控制字符重复的次数。例如，\ d {1, 4} 表示 1 位到 3 位数字。前面提到的电话号码，可以采用如下的正则表达式：

```
\d{3} - \d{8} | \d{4} - \d{7}
```

更多资源请访问稀酷客(www.ckook.com)

❑ 上面的正则表达式匹配区号为3位的8位数电话号码或区号为4位的7位数电话号码。

在书写电话号码时，区号和本地号码之间采用“-”间隔，也有的在区号两侧加圆括号分隔。这个需求也可以使用正则表达式实现：

```
[\(]? \d{3}[\]-)? \d{8}|[\(]? \d{4}[\]-]? \d{7}
```

❑ 上面的“[\ (-]?”表示最多只能取“(”或“-”其中之一，因此该表达式支持“-”或“()”两种分隔区号的方式，同时也支持把区号和本地电话号码连在一起书写。例如：010-12345678、01012345678、(010) 12345678。

注意　“(”和“)”是正则表达式中的特殊字符，如果要把它们作为普通字符处理，需要在前面添加转义字符“\”。

如果要对正则表达式进行嵌套，就需要使用分组“()”。例如，对3位数字重复3次，可以使用如下的正则表达式表示：

```
(\d\d\d){2}
```

❑ 上面的正则表达式如果对字符串“123456789”进行匹配，则匹配到子串“123456”。

如果正则表达式写为如下的形式，将匹配到不同的结果：

```
\d\d\d{2}
```

❑ 该表达式相当于“\d\d\d\d”，匹配的结果为“1234”和“5678”。

正则表达式的每个分组会自动拥有一个组号。从左往右第1个出现的圆括号为第1个分组，表示为“\1”。第2个出现的圆括号为第2个分组，表示为“\2”，以此类推。组号可以用于重复匹配某个分组。例如，对字符串“abc”重复两次可以表示为如下的正则表达式：

```
(abc)\1
```

❑ 上面的正则表达式如果对字符串“abcabcab”进行匹配，则匹配的结果为“abcabc”。

默认情况下，正则表达式将匹配最长的字符串作为结果。可以通过在限定符后面添加“?”的方式，获取最短匹配的结果。例如，对字符“a”到字符“c”之间的字符进行匹配：

```
a.*c
```

如果对字符串“abcabc”进行匹配，则匹配结果为源字符串“abcabc”，即最长匹配。如果把正则表达式改为如下的方式，则将进行最短匹配：

```
a.*?c
```

在“*”后添加了一个“?”，匹配结果为“abc”和“abc”。当匹配到第1个字符“c”后，匹配立刻中断，并返回匹配结果。然后继续进行新的匹配，即返回了第2个“abc”。限定符与“?”组合还有一些其他的用法，如表6-6所示。

表6-6　限定符与“?”的组合

符　号	说　明
*?	匹配零次或多次，且最短匹配
+?	匹配一次或多次，且最短匹配
??	匹配一次或零次，且最短匹配
{m, n}?	重复m次，且最短匹配
(?#...)	正则表达式中的注释

更多资源请访问稀酷客(www.ckook.com)

（续）

符　号	说　明
（？P<name>...）	给分组命名，name 表示分组的名称
（？P=name）	使用名为 name 的分组

注意　表中的（？P<name>...）和（？P=name）是 Python 中的写法，其他的符号在各种编程语言中都是通用的。

6.2.2　使用 sys. re 模块处理正则表达式

Python 的 re 模块具有正则表达式匹配的功能。re 模块提供了一些根据正则表达式进行查找、替换、分隔字符串的函数，这些函数使用一个正则表达式作为第一个参数。re 模块常用的函数如表 6-7 所示。

表 6-7　re 模块的常用函数

函　数	说　明
findall（pattern，string，flags=0）	根据 pattern 在 string 中匹配字符串。如果匹配成功，返回包含匹配结果的列表，否则返回空列表。当 pattern 中有分组时，返回包含多个元组的列表，每个元组对应 1 个分组。flags 表示规则选项，规则选项用于辅助匹配（可用的规则选项参见表 6-8）
sub（pattern，repl，string，count=0）	根据指定的正则表达式替换源字符串中的子串。pattern 是一个正则表达式，repl 是用于替换的字符串，string 是源字符串。如果 count 等于 0，则返回 string 中匹配的所有结果；如果 count 大于 0，则返回前 count 个匹配结果
subn（pattern，repl，string，count=0）	作用和 sub( )相同，返回 1 个 2 元的元组。第 1 个元素是替换结果，第 2 个元素是替换的次数
match（pattern，string，flags=0）	根据 pattern 从 string 的头部开始匹配字符串，只返回第 1 次匹配成功的对象。否则返回“None”
search（pattern，string，flags=0）	根据 pattern 在 string 中匹配字符串，只返回第 1 次匹配成功的对象。如果匹配失败，返回“None”
compile（pattern，flags=0）	编译正则表达式 pattern，返回 1 个 Pattern 对象
split（pattern，string，maxsplit=0）	根据 pattern 分隔 string，maxsplit 表示最大的分割数
escape（pattern）	匹配字符串中的特殊字符，如“*”、“+”、“?”等

注意　函数 match( )必须从字符串的第 0 个索引位置处开始搜索。如果第 0 个索引位置的字符不匹配，match( )的匹配就会失败。

re 模块的一些函数中都有一个 flags 参数，该参数用于设置匹配的附加选项。例如，是否忽略大小写、是否支持多行匹配等。表 6-8 列出了 re 模块的规则选项。

表 6-8　re 模块的规则选项

选　项	说　明
I 或 IGNORECASE	忽略大小写
L 或 LOCALE	字符集本地化，用于多语言环境
M 或 MULTILINE	多行匹配，使^ $ 匹配除了 string 开始结束外，还匹配一行的开始和结束
S 或 DOTALL	使“.”匹配包括“\n”在内的所有字符
X 或 VERBOSE	忽略正则表达式中的空白、换行，以方便添加注释
U 或 UNICODE	“\w”、“\W”、“\b”、“\B”、“\d”、“\D”、“\s”和“\S”都将使用 Unicode

更多资源请访问稀酷客(www.ckook.com)

re模块定义了一些常量来表示这些选项，使用前导符“re.”加选项的简写或名称的方式表示某个常量。例如，re.I或re. IGNORECASE表示忽略大小写。

正则表达式中有3种间隔符号——“^”、“$”和“\b”。“^”匹配字符串首部的子串；“$”匹配结束部分的子串；“\b”用于分隔单词。

【例6-12】演示间隔符在Python中的使用方法。

```
import re
# ^与$的使用方法
s = "HELLO WORLD"
print re.findall(r"^hello", s)
print re.findall(r"^hello", s, re.I)
print re.findall("WORLD$", s)
print re.findall(r"wORld$", s, re.I)
print re.findall(r"\b\w+\b", s)
```

- 第4行代码，匹配以“hello”开始的字符串。由于变量s中的“HELLO”采用的是大写，所以匹配失败。输出结果：[]。
- 第5行代码，添加了辅助参数flags，“re.I”表示匹配时忽略大小写。输出结果：['HELLO']。
- 第6行代码，匹配以“WORLD”结尾的字符串。输出结果：['WORLD']。
- 第7行代码，匹配以“WORLD”结尾的字符串，并忽略大小写。输出结果：['WORLD']。
- 第8行代码，匹配每个英文单词。输出结果：['HELLO', 'WORLD']。

前面介绍了用replace()实现字符串的替换功能，使用re模块的sub()同样可以实现字符串的替换功能。

【例6-13】演示sub()替换字符串的方法。

```
import re

s = "hello world"
print re.sub("hello", "hi", s)
print re.sub("hello", "hi", s[-4:])
print re.sub("world", "China", s[-5:])
```

- 第4行代码的输出结果：hi world。
- 第5行代码，在分片s[-4:]范围内替换“hello”，即在字符串“orld”中替换“hello”。由于没有找到匹配的子串，所以sub()返回s[-4:]。输出结果：orld。
- 第6行代码，在分片s[-5:]范围内替换“world”，即把字符串“world”替换为“China”。输出结果：China。

注意　sub()先创建变量s的拷贝，然后在拷贝中替换字符串，并不会改变变量s的内容。

subn()的功能与sub()相同，但是多返回1个值，即匹配后的替换次数。

【例6-14】演示subn()对字符串的替换以及正则表达式中特殊字符的使用方法。

```
import re
# 特殊字符的使用方法
s = "你好 WORLD2"
print "匹配字母数字:" + re.sub(r"\w", "hi", s)
print "替换次数:" + str(re.subn(r"\w", "hi", s)[1])
print "匹配非字母数字的字符:" + re.sub(r"\W", "hi", s)
print "替换次数:" + str(re.subn(r"\W", "hi", s)[1])
print "匹配空白字符:" + re.sub(r"\s", "* ", s)
```

更多资源请访问稀酷客(www.ckook.com)

```
print "替换次数:" + str(re.subn(r"\s", "* ", s)[1])
print "匹配非空白字符:" + re.sub(r"\S", "hi", s)
print "替换次数:" + str(re.subn(r"\S", "hi", s) [1])
print "匹配数字:" + re.sub(r"\d", "2.0", s)
print "替换次数:" + str(re.subn(r"\d", "2.0", s)[1])
print "匹配非数字:" + re.sub(r"\D", "hi", s)
print "替换次数:" + str(re.subn(r"\D", "hi", s)[1])
print "匹配任意字符:" + re.sub(r".", "hi", s)
print "替换次数:" + str(re.subn(r".", "hi", s)[1])
```

- 第4行代码，“\w”并不能匹配汉字。输出结果：匹配字母数字：你好 hihihihihihi。
- 第5行代码，输出替换次数。替换次数存放在subn()返回元组的第2个元素中。字符串“WORLD2”有6个字符，所以被替换为6个“hi”。输出结果：替换次数：6。
- 第6行代码，替换非字母、数字、下划线的字符。输出结果：匹配非字母数字的字符：hihihihihiWORLD2。
- 第7行代码，汉字“你好”和后面的空格被替换为5个“hi”（每个汉字相当于2个字符）。输出结果：替换次数：5。
- 第8行代码，匹配空白字符（空格、制表符等都属于空白字符）。输出结果：匹配空白字符：你好*WORLD2。
- 第9行代码，由于只有一个空格，所以替换次数为1。输出结果：替换次数：1。
- 第10行代码，替换非空格字符。输出结果：匹配非空白字符：hihihihi hihihihihihi。
- 第11行代码，除空格外，字符串共占用了10个字符的位置。输出结果：替换次数：10。
- 第12行代码，替换数字。输出结果：匹配数字：你好 WORLD2.0。
- 第13行代码，把数字2替换为“2.0”。输出结果：替换次数：1。
- 第14行代码，替换每个英文字符。输出结果：匹配非数字：hihihihihihihihihihi2。
- 第15行代码，替换了10个非数字字符。输出结果：替换次数：10。
- 第16行代码，“.”替换任意字符。输出结果：匹配任意字符：hihihihihihihihihihihi。
- 第17行代码，所有11个字符均被替换。输出结果：替换次数：11。

前面介绍了解析电话号码的正则表达式，下面通过Python程序来实现电话号码的匹配：

```
import re
# 限定符的使用
tel1 = "0791-1234567"
print re.findall(r"\d{3}-\d{8}|\d{4}-\d{7}", tel1)
tel2 = "010-12345678"
print re.findall(r"\d{3}-\d{8}|\d{4}-\d{7}", tel2)
tel3 = "(010)12345678"
print re.findall(r"[\(]? \d{3}[\)-]? \d{8}|[\(]? \d{4}[\)-]? \d{7}", tel3)
```

- 第4行代码，匹配区号为3位的8位数电话号码或区号为4位的7位数电话号码。输出结果：['0791-1234567']。
- 第6行代码，匹配区号为3位的8位数电话号码或区号为4位的7位数电话号码，区号和电话号码之间用“-”连接。输出结果：['010-12345678']。
- 第8行代码，匹配区号为3位的8位数电话号码或区号为4位的7位数电话号码，区号和电话号码之间可以采用3种方式书写。一是直接使用“-”连接，二是区号两侧添加圆括号再连接电话号码，三是区号和电话号码连在一起书写。因此该表达式可以接受tel1、tel2、tel3格式的电话号码。当输入tel3时，输出结果：['(010) 12345678']。

正则表达式的解析非常费时。如果多次使用findall()的方式匹配字符串，搜索效率可能比较

更多资源请访问稀酷客(www.ckook.com)

低。如果要多次使用同一规则匹配字符串，可以使用 compile( ) 函数进行预编译，compile 函数返回 1 个 pattern 对象。该对象拥有一系列方法用于查找、替换或扩展字符串，从而提高字符串的匹配速度。表 6-9 列出了 pattern 对象的属性和方法。

表 6-9　pattern 对象的属性和方法

方　法	说　明
Pattern	获取当前使用的正则表达式
findall（string [，start [，end]]）	查找所有符合 pattern 对象匹配条件的结果，返回 1 个包含匹配结果的列表。参数 string 表示待查找的源字符串，参数 start 表示搜索开始的位置，参数 end 表示搜索结束的位置
finditer（string [，start [，end]]）	返回 1 个包含匹配结果的地址
match（string [，start [，end]]）	用法同 re. match( )
search（string [，start [，end]]）	用法同 re. search( )

【例 6-15】在 1 个字符串中查找多个数字，使用 compile( ) 函数以提高查找的效率。

```
import re
# compile()预编译
s = "1abc23def45"
p = re.compile(r"\d+")
print p.findall(s)
print p.pattern
```

- 第 4 行代码，返回 1 个正则表达式对象 p，匹配变量 s 中的数字。
- 第 5 行代码，调用 p 的 findall( )方法，匹配的结果存放在列表中。输出结果：['1', '23', '45']。
- 第 6 行代码，输出当前使用的正则表达式。输出结果：\ d + 。

函数 compile( )通常与 match( )、search( )、group( )一起使用，对含有分组的正则表达式进行解析。正则表达式的分组从左往右开始计数，第 1 个出现的圆括号标记为第 1 组，依次类推。此外还有 0 号组，0 号组用于存储匹配整个正则表达式的结果。match( ) 和 search( ) 将返回 1 个 match 对象，match 对象提供了一系列的方法和属性来管理匹配的结果。表 6-10 列出了 match 对象的方法和属性。

表 6-10　match 对象的方法和属性

属性和方法	说　明
Pos	搜索的开始位置
Endpos	搜索的结束位置
String	搜索的字符串
Re	当前使用的正则表达式的对象
Lastindex	最后匹配的组索引
Lastgroup	最后匹配的组名
group（index = 0）	某个分组的匹配结果。如果 index 等于 0，表示匹配整个正则表达式
groups( )	所有分组的匹配结果，每个分组的结果组成 1 个列表返回
Groupdict( )	返回组名作为 key，每个分组的匹配结果作为 value 的字典
start（ [group]）	获取组的开始位置
end（ [group]）	获取组的结束位置

更多资源请访问稀酷客(www.ckook.com)

（续）

属性和方法	说　明
span（［group］）	获取组的开始和结束位置
expand（template）	使用组的匹配结果来替换模板 template 中的内容，并把替换后的字符串返回

【例 6-16】演示对正则表达式分组的解析。

```
import re
# 分组
p = re.compile(r"(abc)\1")
m = p.match("abcabcabc")
print m.group(0)
print m.group(1)
print m.group()

p = re.compile(r"(?P<one>abc)(?P=one)")
m = p.search("abcabcabc")
print m.group("one")
print m.groupdict().keys()
print m.groupdict().values()
print m.re.pattern
```

- 第 3 行代码，定义了 1 个分组“（abc）”，在后面使用“\1”再次调用该分组。即 compile()返回 1 个包含 2 个分组的正则表达式对象 p。
- 第 4 行代码，p. match()对字符串“abcabcabc”进行搜索，返回 1 个 match 对象 m。
- 第 5 行代码，调用 match 对象的 group（0）方法，将匹配整个表达式。输出结果：abcabc。
- 第 6 行代码，调用 match 对象的 group（1）方法，返回第 1 次匹配成功的结果。输出结果：abc。
- 第 7 行代码，默认情况下，返回分组 0 的结果。输出结果：abcabc。
- 第 9 行代码，给分组命名，“？P<one>”中的“one”表示分组的名称。“（？P=one）”调用分组“one”，相当于“\1”。
- 第 11 行代码，输出分组“one”的结果。输出结果：abc。
- 第 12 行代码，获取正则表达式中分组的名称。输出结果：['one']。
- 第 13 行代码，获取正则表达式中分组的内容。输出结果：['abc']。
- 第 14 行代码，获取当前使用的正则表达式。输出结果：（？P<one>abc）（？P=one）。

如果使用 match()匹配的源字符串“abcabcabc”改为“bcabcabc”，则 Python 将提示如下错误：

```
AttributeError: 'NoneType' object has no attribute 'group'
```

这种情况可以用 search()替换 match()，search()可以匹配出正确的结果。

6.3　小结

本章讲解了 Python 中字符串的操作，包括字符串的格式化、合并、截取、比较、查找、替换等操作。对比了 Python 与 Java 对字符串操作处理的异同，重点讲解了字符串的几种截取方法、字符串与日期类型之间的转换、正则表达式的语法以及 Python 的 re 模块。正则表达式是文本匹配的工具，通常用于对规则数据的验证，例如电话号码、手机号码等。re 模块提供了 sub()、findall()、search()、complie()等函数对正则表达式进行解析，结合 pattern 对象、match 对象可以更好的处理和控制正则表达式和匹配结果。

更多资源请访问稀酷客(www.ckook.com)

第 7 章　文件的处理

数据的存储可以使用数据库，也可以使用文件。数据库保持了数据的完整性和关联性，而且数据更安全、可靠。使用文件存储数据则非常简单、易用，不必安装数据库管理系统等运行环境。文件通常用于存储应用软件的参数或临时性数据。Python 的文件操作和 Java 的文件操作十分相似，Python 提供了 os、os. path 等模块用于处理文件。本章的知识点：

- 文件的创建、读写和修改
- 文件的拷贝、删除和重命名
- 文件内容的搜索和替换
- 文件的比较
- 配置文件的读写
- 目录的创建和遍历
- 文件和流

7. 1　文件的基本操作

文件通常用于存储数据或应用系统的参数。Python 提供了 os、os. path、shutil 等模块用于处理文件，其中包括打开文件、读写文件、复制和删除文件等函数。

7. 1. 1　文件的打开或创建

文件的打开或创建可以使用内联模块的函数 file( )。该函数可以指定处理模式，设置打开的文件为只读、只写或可读写状态。file( )的声明如下所示：

```
file(name[, mode[, buffering]]) -> file object
```

- 参数 name 是被打开的文件名称。如果文件 name 不存在，file( )将创建名为 name 的文件，然后再打开该文件。
- 参数 mode 是指文件的打开模式。文件的打开模式如表 7-1 所示。
- 参数 buffering 设置缓存模式。0 表示不缓存；1 表示行缓冲；如果大于 1 则表示缓冲区的大小，以字节为单位。
- file( )返回 1 个 file 对象，file 对象可以对文件进行各种操作。

表 7-1　文件的打开模式

参　数	说　明
r	以只读的方式打开文件
r +	以读写的方式打开文件
w	以写入的方式打开文件。先删除文件原有的内容，再重新写入新的内容。如果文件不存在，则创建 1 个新的文件
w +	以读写的方式打开文件。先删除文件原有的内容，再重新写入新的内容。如果文件不存在，则创建 1 个新的文件

更多资源请访问稀酷客(www.ckook.com)

（续）

参　数	说　明
a	以写入的方式打开文件，在文件的末尾追加新的内容。如果文件不存在，则创建 1 个新的文件
a +	以读写的方式打开文件，在文件的末尾追加新的内容。如果文件不存在，则创建 1 个新的文件
b	以二进制模式打开文件。可与 r、w、a、+结合使用
U	支持所有的换行符号。“\ r”、“\ n”、“\ r\ n” 都表示换行

注意　对于图片、视频等文件必须使用“b”的模式读写。

file 类用于文件管理，可以对文件进行创建、打开、读写、关闭等操作。file 类的常用属性和方法如表 7-2 所示。

表 7-2　file 类的常用属性和方法

属性和方法	说　明
Closed	判断文件是否关闭。如果文件被关闭，返回 True
Encoding	显示文件的编码类型
Mode	显示文件的打开模式
Name	显示文件的名称
Newlines	文件使用的换行模式
file（name［, mode［, buffering］］）	以 mode 指定的方式打开文件。如果文件不存在，则先创建文件，再打开该文件。buffering 表示缓存模式。0 表示不缓存；1 表示行缓冲；如果大于 1 则表示缓冲区的大小
flush( )	把缓存区的内容写入磁盘
close( )	关闭文件
read（［size］）	从文件中读取 size 个字节的内容作为字符串返回
readline（［size］）	从文件中读取 1 行作为字符串返回。如果指定 size，表示每行每次读取的字节数，依然要读完整行的内容
Readlines（［size］）	把文件中的每行存储在列表中返回。如果指定 size，表示每次读取的字节数
seek（offset［, whence］）	把文件的指针移动到 1 个新的位置。offset 表示相对于 whence 的位置。whence 用于设置相对位置的起点，0 表示从文件开头开始计算；1 表示从当前位置开始计算；2 表示从文件末尾开始计算。如果 whence 省略，offset 表示相对文件开头的位置
tell( )	返回文件指针当前的位置
next( )	返回下一行的内容，并将文件的指针移到下一行
truncate（［size］）	删除 size 个字节的内容
write（str）	把字符串 str 的内容写入文件
writelines（sequence_ of_ strings）	把字符串序列写入文件

文件的处理一般分为以下 3 个步骤。

（1）创建并打开文件，使用 file( )函数返回 1 个 file 对象。

（2）调用 file 对象的 read( )、write( )等方法处理文件。

更多资源请访问稀酷客(www.ckook.com)

（3）调用 close( )关闭文件，释放 file 对象占用的资源。

注意　close( )方法是必要的。虽然 Python 提供了垃圾回收机制清理不再使用的对象，但是手动释放不再需要的资源是一种良好的习惯。同时也显式地告诉 Python 的垃圾回收器，该对象需要被清除。

【例 7-1】演示文件的创建、写入和关闭操作。

```
# 创建文件
context = '''hello world
hello China
'''
f = file('hello.txt', 'w')          # 打开文件
f.write(context)                    # 把字符串写入文件
f.close ()                          # 关闭文件
```

- 第 5 行代码调用 file( )创建文件 hello. txt，设置文件的访问模式为“w”。file( )返回文件对象 f。
- 第 6 行代码把变量 context 的值写入 hello. txt 文件中。
- 第 7 行代码调用对象 f 的 close( )方法，释放对象 f 占用的资源。

后面讲解的文件读写、删除和拷贝等操作也将遵循这 3 个步骤。

注意　可以使用 open( )替代 file( )，open( )是 file( )的别名。

7.1.2　文件的读取

文件的读取有多种方法，可以使用 readline( )、readlines( )或 read( )函数读取文件。下面将一一介绍这些函数读取文件的实现方法。

1. 按行读取方式 readline( )

readline( )每次读取文件中的一行，需要使用永真表达式循环读取文件。当文件指针移动到文件的末尾，依然使用 readline( )读取文件将出现错误。因此程序中需要添加 1 个判断语句，判断文件指针是否移动到文件的尾部，并且通过该语句中断循环。

【例 7-2】使用【例 7-1】创建的 hello. txt 文件演示 readline( )的使用方法。

```
# 使用 readline () 读文件
f = open("hello.txt")
while True:
    line = f.readline ()
    if line:
        print line,
    else:
        break
f.close ()
```

- 第 3 行代码使用了“Ture”作为循环条件，构成 1 个永真循环。
- 第 4 行代码调用 readline( )，读取 hello. txt 文件的每一行。每次循环依次输出如下结果：

```
hello world
hello China
```

- 第 5 行代码，判断变量 line 是否为真。如果为真，则输出当前行的内容，否则退出循环。

如果把第 4 行代码改为如下语句，读取的方式略有不同，但读取的内容完全相同：

```
line = f.readline(2)
```

更多资源请访问稀酷客(www.ckook.com)

- 该行代码并不表示每行只读取 2 个字节的内容，而是指每行每次读 2 个字节，直到行的末尾。

2. 多行读取方式 readlines( )

使用 readlines( )读取文件，需要通过循环访问 readlines( )返回列表中的元素。函数 readlines( )一次可读取文件中多行数据。

【例 7-3】使用【例 7-1】创建的 hello. txt 文件演示 readlines( )读取文件的方法。

```
# 使用 readlines () 读文件
f = file('hello. txt')
lines = f. readlines ()
for line in lines:
print line,
```

- 第 3 行代码调用 readlines( )，把文件 hello. txt 中所有的内容存储在列表 lines 中。
- 第 4 行代码循环读取列表 lines 中的内容。
- 第 5 行代码输出列表 lines 每个元素的内容，即文件 hello. txt 每行的内容。

3. 一次性读取方式 read( )

读取文件最简单的方法是使用 read( )，read( )将从文件中读出所有内容，并赋值给 1 个字符串变量。

【例 7-4】使用【例 7-1】创建的 hello. txt 文件演示 read( )读取文件的方法。

```
# 使用 read () 读文件
f = open("hello. txt")
context = f. read ()
print context
```

- 第 2 行代码调用 open( )，获取文件 hello. txt 的访问对象 f。
- 第 3 行代码调用对象 f 的 read( )获取文件的内容。
- 第 4 行代码输出文件的内容。

可以通过控制 read( )参数的值，返回指定字节的内容：

```
f = open("hello. txt")
context = f. read(5)
print context
print f. tell ()
context = f. read(5)
print context
print f. tell ()
f. close ()
```

- 第 2 行代码调用 read（5），读取 hello. txt 文件中前 5 个字节的内容，并存储到变量 context 中。此时文件的指针移到第 5 个字节处。
- 第 3 行代码输出变量 context 的结果，输出的结果：hello。
- 第 4 行代码调用 tell( )，输出文件指针当前的位置：5。
- 第 5 行代码再次调用 read（5），读取第 6 个字节到第 10 个字节的内容。
- 第 6 行代码输出结果：worl。
- 第 7 行代码输出文件指针当前的位置：10。

注意　file 对象内部将记录文件指针的位置，以便下次操作。只要 file 对象没有执行 close( )方法，文件指针就不会释放。

更多资源请访问稀酷客(www.ckook.com)

7.1.3　文件的写入

文件写入的实现同样具有多种方法，可以使用 write( )、writelines( )方法写入文件。7.1.1一节使用了 write( )方法把字符串写入文件，writelines( )方法可以把列表中存储的内容写入文件。

【例 7-5】演示如何将列表中的元素写入到文件。

```
#使用 writelines () 写文件
f = file("hello.txt", "w+")
li = ["hello world\n", "hello China\n"]
f.writelines(li)
f.close ()
```

- 第 2 行代码使用“w+”的模式创建并打开文件 hello.txt。
- 第 3 行代码定义了 1 个列表 li。li 中存储了 2 个元素，每个元素代表文件中的 1 行，“\n”用于换行。
- 第 4 行代码调用 writelines( )，把列表 li 的内容写入到文件中。文件中的内容如下所示：

```
hello world
hello China
```

write( )和 writelines( )这两个方法在写入前会清除文件中原有的内容，再重新写入新的内容，相当于“覆盖”的方式。如果需要保留文件中原有的内容，只是需要追加新的内容，可以使用“a+”模式打开文件。

【例 7-6】使用【例 7-5】创建的文件演示文件的修改操作。

```
#追加新的内容到文件
f = file("hello.txt", "a+")
new_context = "goodbye"
f.write(new_context)
f.close ()
```

- 第 2 行代码使用模式“a+”打开文件 hello.txt。
- 第 4 行代码调用 write( )方法，hello.txt 文件的原有内容保持不变，把变量 new_context 的内容写入到 hello.txt 文件的末尾。此时 hello.txt 中的内容如下所示：

```
hello world
hello China
goodbye
```

使用 writelines( )写文件的速度更快。如果需要写入文件的字符串非常多，可以使用 writelines( )方法以提高效率。如果需要写入少量的字符串，使用 write( )方法即可。

7.1.4　文件的删除

文件的删除需要使用 os 模块和 os.path 模块。os 模块提供了对系统环境、文件、目录等操作系统级的接口函数。表 7-3 列出了 os 模块常用的文件处理函数。

表 7-3　os 模块常用的文件处理函数

函　数	说　明
access（path, mode）	按照 mode 指定的权限访问文件
chmod（path, mode）	改变文件的访问权限。mode 用 UNIX 系统中的权限代号表示
open（filename, flag [, mode=0777]）	按照 mode 指定的权限打开文件。默认情况下，给所有用户读、写、执行的权限

更多资源请访问稀酷客(www.ckook.com)

（续）

函　数	说　明
remove（path）	删除 path 指定的文件
rename（old，new）	重命名文件或目录。old 表示原文件或目录，new 表示新文件或目录
stat（path）	返回 path 指定文件的所有属性
fstat（path）	返回打开的文件的所有属性
lseek（fd，pos，how）	设置文件的当前位置，返回当前位置的字节数
startfile（filepath [，operation]）	启动关联程序打开文件。例如，打开的是 1 个 html 文件，将启动 IE 浏览器
tmpfile( )	创建 1 个临时文件，文件创建在操作系统的临时目录中

注意　os 模块的 open( ) 函数与内建的 file( )、open( ) 函数的用法并不相同。

文件的删除需要调用 remove( ) 函数实现。例如，首先创建 1 个文件 hello. txt，然后判断该文件是否存在，如果存在则删除文件 hello. txt。判断文件是否存在需要使用 os. path 模块，os. path 模块用于处理文件和目录的路径。表 7-4 列出了 os. path 模块常用的函数。

表 7-4　os. path 模块常用的函数

函　数	说　明
abspath（path）	返回 path 所在的绝对路径
dirname（p）	返回目录的路径
exists（path）	判断文件是否存在
getatime（filename）	返回文件的最后访问时间
getctime（filename）	返回文件的创建时间
Getmtime（filename）	返回文件最后的修改时间
getsize（filename）	返回文件的大小
isabs（s）	测试路径是否为绝对路径
isdir（path）	判断 path 指定的是否为目录
isfile（path）	判断 path 指定的是否为文件
split（p）	对路径进行分割，并以列表的方式返回
splitext（p）	从路径中分割文件的扩展名
Splitdrive（p）	从路径中分割驱动器的名称
walk（top，func，arg）	遍历目录数，与 os. walk( ) 的功能相同

【例 7-7】演示文件的删除操作。

```
import os

file("hello. txt", "w")
if os. path. exists("hello. txt"):
os. remove("hello. txt")
```

- 第 3 行代码创建文件 hello. txt。
- 第 4 行代码调用 os. path 模块的 exists( ) 判断文件 hello. txt 是否存在。
- 第 5 行代码调用 remove( ) 删除文件 hello. txt。

更多资源请访问稀酷客(www.ckook.com)

7.1.5　文件的复制

file 类没有提供直接复制文件的方法，但是可以使用 read()、write()方法模拟实现文件拷贝的功能。

【例 7-8】把 hello.txt 文件的内容复制给 hello2.txt 文件。

```
#使用 read()、write()实现拷贝
#创建文件 hello.txt
src = file("hello.txt", "w")
li = ["hello world\n", "hello China\n"]
src.writelines(li)
src.close()
#把 hello.txt 拷贝到 hello2.txt
src = file("hello.txt", "r")
dst = file("hello2.txt", "w")
dst.write(src.read())
src.close()
dst.close()
```

- 第 8 行代码以只读的方式打开文件 hello.txt。
- 第 9 行代码以只写的方式打开文件 hello2.txt。
- 第 10 行代码，通过 read()读取 hello.txt 的内容，然后把这些内容写入 hello2.txt。

shutil 模块是另一个文件、目录的管理接口，提供了一些用于复制文件、目录的函数。copyfile()函数可以实现文件的拷贝，copyfile()函数的声明如下所示：

```
copyfile(src, dst)
```

- 参数 src 表示源文件的路径，src 是字符串类型。
- 参数 dst 表示目标文件的路径，dst 是字符串类型。
- 该函数把 src 指向的文件复制到 dst 指向的文件。

文件的剪切可以使用 move()函数模拟，该函数的声明如下所示：

```
move(src, dst)
```

move()的参数和 copyfile()的参数相同，移动 1 个文件或目录到指定的位置，并且可以根据参数 dst 重命名移动后的文件。

【例 7-9】使用 shutil 模块实现文件的拷贝。

```
#shutil 模块实现文件的拷贝
import shutil

shutil.copyfile("hello.txt","hello2.txt")
shutil.move("hello.txt","../")
shutil.move("hello2.txt","hello3.txt")
```

- 第 4 行代码调用 copyfile()，把 hello.txt 的内容复制给 hello2.txt。
- 第 5 行代码调用 move()，把 hello.txt 复制到当前目录的父目录，然后删除 hello.txt。相当于把文件 hello.txt 剪切到父目录。
- 第 6 行代码调用 move()，把 hello2.txt 移动到当前目录并命名为 hello3.txt。hello2.txt 将被删除。

7.1.6　文件的重命名

os 模块的函数 rename()可以对文件或目录进行重命名。

更多资源请访问稀酷客(www.ckook.com)

【例 7-10】演示文件重命名的操作。如果当前目录存在名为 hello. txt 的文件，则重命名为 hi. txt；如果存在 hi. txt 的文件，则重命名为 hello. txt。

```
#修改文件名
import os
li = os.listdir(".")
print li
if "hello.txt" in li:
    os.rename("hello.txt", "hi.txt")
elif "hi.txt" in li:
    os.rename("hi.txt", "hello.txt")
```

- 第 3 行代码调用 listdir( )返回当前目录的文件列表，其中“.”表示当前目录。
- 第 4 行代码输出当前目录包含的文件，rename_ file. py 是本例程序的文件。输出结果如下所示。

```
['rename_file.py', 'hello.txt']
```

- 第 5 行代码判断当前目录是否存在文件 hello. txt。如果存在，则把 hello. txt 重命名为 hi. txt。
- 第 7 行代码判断当前目录是否存在文件 hi. txt。如果存在，则把 hi. txt 重命名为 hello. txt。

在实际应用中，通常需要把某一类文件修改为另一种类型，即修改文件的扩展名。这种需求可以通过函数 rename( )和字符串查找的函数实现。

【例 7-11】把后缀名为“html”的文件修改为以“htm”后缀的文件。

```
#修改后缀名
import os
files = os.listdir(".")
for filename in files:
    pos = filename.find(".")
    if filename[pos + 1:] == "html":
        newname = filename[:pos + 1] + "htm"
        os.rename(filename,newname)
```

- 第 3 行代码调用 listdir( )，返回当前目录的文件列表 files。这样可以对当前目录中的多个文件进行重命名。
- 第 4 行代码循环列表 files，获取当前目录的每个文件名。
- 第 5 行代码调用 find( )查找文件名中“.”所在的位置，并把值赋给变量 pos。
- 第 6 行代码判断文件的后缀名是否为“html”。pos + 1 表示“.”后的位置。
- 第 7 行代码重新组合新的文件名，以“htm”作为后缀名。filename［: pos + 1］表示从 filename 的开头位置到“.”这段分片。
- 第 8 行代码调用 rename 重命名文件。

为获取文件的后缀名，这里先查找“.”所在的位置，然后通过分片 filename［pos + 1:］截取后缀名。这个过程可以使用 os. path 模块的函数 splitext( )实现。splitext( )返回 1 个列表，列表中的第 1 个元素表示文件名，第 2 个元素表示文件的后缀名。修改后的代码如下所示：

```
import os
files = os.listdir(".")
for filename in files:
    li = os.path.splitext(filename)
    if li[1] == ".html":
        newname = li[0] + ".htm"
        os.rename(filename,newname)
```

- 第 4 行代码调用 splitext( )对文件名进行解析，返回文件名和后缀名组成的列表。

更多资源请访问稀酷客(www.ckook.com)

- 第5行代码，通过li［1］判断文件是否以“html”结尾。
- 第6行代码重新组合新的文件名，li［0］表示不带后缀的文件名。

glob模块用于路径的匹配，返回符合给定匹配条件的文件列表。glob模块的主要函数就是glob()，该函数返回符合同一匹配条件的多个文件。上面的例程需要判断文件是否为html后缀，也可以使用glob()直接匹配文件名称：

```
glob.glob("*.html")
```

- 该行代码返回html格式的文件。输出结果如下所示：

```
['hello.html', 'hi.html']
```

glob可以对路径做更多的匹配。例如，下面这段代码可以匹配C盘中以“w”开头的目录中所有的文本文件：

```
glob.glob("c:\\w* \\*.txt")
```

注意　glob()的参数应该满足正则表达式语法的要求。

7.1.7　文件内容的查找和替换

文件内容的搜索和替换可以结合前面学习的字符串查找和替换来实现。

1. 文件内容的查找

【例7-12】从hello.txt文件中查找字符串“hello”，并统计“hello”出现的次数。hello.txt文件如下所示：

```
1 hello world
3 hello hello China
```

从hello.txt中统计字符串“hello”的个数的代码如下：

```
#文件的查找
import re

f1 = file("hello.txt", "r")
count = 0
for s in f1.readlines():
    li = re.findall("hello", s)
    if len(li) > 0:
        count = count + li.count("hello")
print "查找到" + str(count) + "个hello"
f1.close()
```

- 第5行代码定义变量count，用于计算字符串“hello”出现的次数。
- 第6行代码，每次从文件hello.txt中读取1行到变量s。
- 第7行代码，调用re模块的函数findall()查询变量s，把查找的结果存储到列表li中。
- 第8行代码，如果列表中的元素个数大于0，则表示查找到字符串“hello”。
- 第9行代码，调用列表的count()方法，统计当前列表中“hello”出现的次数。
- 第10行代码输出结果：查找到3个hello。

2. 文件内容的替换

【例7-13】把hello.txt中的字符串“hello”全部替换为“hi”，并把结果保存到文件hello2.txt中。

更多资源请访问稀酷客(www.ckook.com)

```
# 文件的替换
f1 = file("hello.txt", "r")
f2 = file("hello2.txt", "w")
for s in f1.readlines ():
    f2.write(s.replace("hello", "hi"))
f1.close ()
f2.close ()
```

- 第 4 行代码从 hello. txt 中读取每行内容到变量 s 中。第 5 行代码先使用 replace( ) 把变量 s 中的“hello”替换为“hi”，然后把结果写入到文件 hello2. txt 中。文件 hello2. txt 的内容如下所示：

```
1 hi world
3 hi hi China
```

7.1.8　文件的比较

Python 提供了模块 difflib 用于实现对序列、文件的比较。如果要比较两个文件，列出两个文件的异同，可以使用 difflib 模块的 SequenceMatcher 类实现。其中的方法 get_opcodes( ) 可以返回两个序列的比较结果。调用方法 get_opcodes( ) 之前，需要先生成 1 个 SequenceMatcher 对象。SequenceMatcher( ) 的声明如下所示：

```
class SequenceMatcher( [isjunk[, a[, b]]])
```

其中，isjunk 表示比较过程中是否匹配指定的字符或字符串，a、b 表示待比较的两个序列。生成序列比较对象后，调用该对象的 get_ opcodes( ) 方法。将返回 1 个元组（tag，i1，i2，j1，j2）。tag 表示序列分片的比较结果。i1、i2 表示序列 a 的索引，j1、j2 表示序列 b 的索引。表 7-5 列出了 get_ opcodes( ) 返回元组（tag，i1，i2，j1，j2）的含义。

表 7-5　get_opcodes( ) 返回元组（tag，i1，i2，j1，j2）的含义

tag 的值	处理过程
Replace	a［i1：i2］被［j1：j2］替换
Delete	a［i1：i2］分片将被删除，此时 j1 等于 j2
Insert	b［j1：j2］插入到 a［i1：i1］位置处，此时的 i1 等于 i2
equal	a［i1：i2］等于 b［j1：j2］

注意　SequenceMatcher（None，a，b）创建序列比较对象，将以 a 作为参考标准进行比较；SequenceMatcher（None，b，a）创建序列比较对象，则把 b 作为参考标准进行比较。

【例 7-14】有两个文本文件 hello. txt 与 hi. txt，想要获知两个文件的异同。hello. txt 文件的内容如下所示：

```
hello world
```

hi. txt 文件的内容如下所示：

```
hi hello
```

【例 7-14】代码以 hello. txt 文件为参照，实现了两个文件的比较，并返回比较结果。

```
import difflib

f1 = file("hello.txt", "r")
```

更多资源请访问稀酷客(www.ckook.com)

```
f2 = file("hi.txt", "r")
src = f1.read()
dst = f2.read()
print src
print dst
s = difflib.SequenceMatcher(lambda x: x == "", src, dst)
for tag, i1, i2, j1, j2 in s.get_opcodes():
    print ("% s src[% d:% d] = % s dst[% d:% d] = % s" % \
    (tag, i1, i2, src[i1:i2], j1, j2, dst[j1:j2]))
```

- 第9行代码生成1个序列匹配的对象s，其中lambda x：x == "" 表示忽略hi. txt中的换行符。如果hi. txt中有多余的换行，并不会作为不同点返回。
- 第10行代码调用方法get_opcodes()获取文件hello. txt与hi. txt的比较结果。
- 第11行、第12行代码输出比较结果。如果hello. txt与hi. txt的内容完全一致，在hello. txt的开始处插入字符串“hi”，并删除hello. txt中的字符串“world”，hello. txt的内容分片［0：5］与hi. txt的内容分片［3：8］相同。比较结果如下所示：

```
insert src[0:0] = dst[0:3] =hi
equal src[0:5] =hello dst[3:8] =hello
delete src[5:11] = world dst[8:8] =
```

7.1.9　配置文件的访问

在应用程序中，通常使用配置文件定义一些参数。例如，数据库配置文件用于记录数据库的字符串连接、主机名、用户名、密码等信息。Windows的ini文件就是一种传统的配置文件，ini文件由多个块组成，每个块由有多个配置项组成。例如，ODBC. ini文件记录了Windows环境下各种数据库存储系统的ODBC驱动信息：

```
[ODBC 32 bit Data Sources]
MS Access Database =Microsoft Access Driver (* .mdb) (32 λ)
Excel Files =Microsoft Excel Driver (* .xls) (32 λ)
dBASE Files =Microsoft dBase Driver (* .dbf) (32 λ)
[MS Access Database]
Driver32 =C: \WINDOWS \system32 \odbcjt32. dll
[Excel Files]
Driver32 =C: \WINDOWS \system32 \odbcjt32. dll
[dBASE Files]
Driver32 =C: \WINDOWS \system32 \odbcjt32. dll
```

其中每个方括号表示一个配置块，配置块下的多个赋值表达式就是配置项。

1. 读取配置文件的内容

Python标准库的ConfigParser模块用于解析配置文件。ConfigParser模块的ConfigParser类可读取ini文件的内容。

【例7-15】从ODBC. ini文件中读取每个配置块、配置项的标题和配置的内容。

```
#读配置文件
import ConfigParser

config = ConfigParser.ConfigParser()
config.read("ODBC.ini")
sections = config.sections()                          #返回所有的配置块的标题
print "配置块:", sections
o = config.options("ODBC 32 bit Data Sources")  #返回所有的配置项的标题
print "配置项:", o
```

更多资源请访问稀酷客(www.ckook.com)

```
v = config.items("ODBC 32 bit Data Sources")
print "内容:", v
#根据配置块和配置项返回内容
access = config.get("ODBC 32 bit Data Sources", "MS Access Database")
print access
excel = config.get("ODBC 32 bit Data Sources", "Excel Files")
print excel
dBASE = config.get("ODBC 32 bit Data Sources", "dBASE Files")
print dBASE
```

- 第 4 行代码创建 1 个 ConfigParser 对象 config。
- 第 5 行代码读取 ODBC.ini 文件。
- 第 6 行代码调用 sections()返回配置块的标题。
- 第 7 行代码输出配置块的标题。输出结果如下所示。

```
配置块: ['ODBC 32 bit Data Sources', 'Excel Files', 'dBASE Files', 'MS Access Database']
```

- 第 8 行代码调用 options()返回“ODBC 32 bit Data Sources”块下各配置项的标题。
- 第 9 行代码输出配置项的标题。输出结果如下所示：

```
配置项: ['ms access database', 'dbase files', 'excel files']
```

- 第 10 行代码调用 items()返回“ODBC 32 bit Data Sources”块下各配置项的内容。
- 第 11 行代码配置项的内容。输出结果如下所示：

```
内容: [('ms access database', 'Microsoft Access Driver (*.mdb) (32 \xce\xbb)'), ('dbase files
', 'Microsoft dBase Driver (*.dbf) (32 \xce\xbb)'), ('excel files', 'Microsoft Excel Driver
(*.xls) (32 \xce\xbb)')]
```

- 第 13 行到第 18 行代码调用 get()方法分别获取“ODBC 32 bit Data Sources”块下每个配置项的内容。

2. 将设置的配置项目写入配置文件

配置文件的写入操作也很简单。首先调用 add_section()方法添加 1 个新的配置块，然后调用 set()方法，设置配置项目，最后写入配置文件 ODBC.ini 即可。

【例 7-16】演示将设置的配置项目写入配置文件的方法。

```
#写配置文件
import ConfigParser
config = ConfigParser.ConfigParser()
config.add_section("ODBC Driver Count")                    #添加新的配置块
config.set("ODBC Driver Count", "count", 2)                #添加新的配置项
f = open("ODBC.ini", "a+")
config.write(f)
f.close()
```

- 第 4 行代码调用 add_section()添加配置块。
- 第 5 行代码调用 set()添加配置项，并设置其内容。
- 第 6 行代码以“a+”模式打开文件 ODBC.ini，追加新的配置。
- 第 7 行代码写配置文件。

3. 修改配置文件

配置文件的修改需要先读取 ODBC.ini 文件，然后调用 set()方法设置指定配置块下某个配置项的值，最后写入配置文件 ODBC.ini。

【例 7-17】演示在 ODBC.ini 文件中添加配置块“ODBC Driver Count”的方法。

更多资源请访问稀酷客(www.ckook.com)

```
#修改配置文件
import ConfigParser
config = ConfigParser.ConfigParser()
config.read("ODBC.ini")
config.set("ODBC Driver Count", "count", 3)          #修改配置项
f = open("ODBC.ini", "r+")
config.write(f)
f.close()
```

- 第4行代码读配置文件ODBC.ini
- 第5行代码调用set()修改“ODBC Driver Count”下的配置项“count”的值为“3”。
- 第6行代码以“r+”的模式打开配置文件ODBC.ini。
- 第7行代码写配置文件。

4. 配置块和配置项的删除

如果要删除某个配置块，调用remove_section()方法，传递需要删除的配置块名作为参数。如果要删除指定配置块下的某个配置项，调用remove_option()方法，传递需要删除的配置块名和配置项名作为参数。

【例7-18】先删除ODBC.ini文件中的配置项“count”，再删除配置块“ODBC Driver Count”。

```
#删除配置文件
import ConfigParser
config = ConfigParser.ConfigParser()
config.read("ODBC.ini")
config.remove_option("ODBC Driver Count", "count")      #删除配置项
config.remove_section("ODBC Driver Count")              #删除配置块
f = open("ODBC.ini", "w+")
config.write(f)
f.close()
```

- 第5行代码调用remove_option()删除“ODBC Driver Count”下的配置项“count”。
- 第6行代码调用remove_section()删除“ODBC Driver Count”配置块
- 第7行代码以“w+”模式写配置文件。

7.2　目录的基本操作

目录是文件存储的集合，对目录的操作涉及目录的创建、删除和目录的遍历等内容。

7.2.1　目录的创建与删除

os模块提供了针对目录进行操作的函数。表7-6列出了os模块中常用的目录处理函数。

表7-6　os模块常用的目录处理函数

函　数	说　明
mkdir（path［，mode=0777］）	创建path指定的1个目录
makedirs（name，mode=511）	创建多级目录，name表示为“path1/path2/...”
rmdir（path）	删除path指定的目录
removedirs（path）	删除path指定的多级目录
listdir（path）	返回path指定目录下所有的文件名
getcwd()	返回当前的工作目录

更多资源请访问稀酷客(www.ckook.com)

（续）

函　数	说　明
chdir（path）	将当前目录改变为 path 指定的目录
walk（top，topdown = True，onerror = None）	遍历目录树

目录的创建和删除可以使用 mkdir( )、makedirs( )、rmdir( )、removedirs( )实现：

```
import os

os.mkdir("hello")
os.rmdir("hello")
os.makedirs("hello/world")
os.removedirs("hello/world")
```

- 第 3 行代码创建 1 个名为“hello”的目录。
- 第 4 行代码删除目录“hello”。
- 第 5 行代码创建多级目录，先创建目录“hello”，再创建子目录“world”。
- 第 6 行代码删除目录“hello”和“world”。

注意　函数 makedirs( )一次可创建多个目录。函数 removedirs( )一次可删除多个目录。函数 mkdir( )一次只能创建一个目录。函数 rmdir( )一次只能删除一个目录。

7.2.2　目录的遍历

目录的遍历有 3 种实现方法——递归函数、os. path. walk( )、os. walk( )。这 3 种方法的实现原理和步骤各不相同。下面以笔者在写作本书第 7 章时的源代码文件为例，来介绍这些方法的使用。第 7 章的源代码文件存储在笔者计算机的“D：\ developer \ python \ example \ 07”中。

1. 递归函数

第 5. 3. 5 节讲解了递归函数，对于目录的遍历可以通过递归函数实现。递归函数可以在函数主体内直接或间接地调用函数本身。

【例 7-19】对笔者写第 7 章的源代码目录进行遍历，输出该目录下所有的文件名称。

```
#递归遍历目录
import os
def VisitDir(path):
    li = os.listdir(path)
    for p in li:
        pathname = os.path.join(path, p)
        if not os.path.isfile(pathname):
            VisitDir(pathname)
        else:
            print pathname

if __name__ == "__main__":
    path = r"D:\developer\python\example\07"
VisitDir(path)
```

- 第 3 行代码定义了名为 VisitDir( )的函数，该函数接受目录路径作为参数。
- 第 4 行代码返回当前路径下所有的目录名和文件名。
- 第 6 行代码调用 os. path 模块的函数 join( )，获取文件的完整路径，并保存到变量 pathname。
- 第 7 行代码判断 pathname 是否为文件。如果 pathname 表示目录，则递归调用 VisitDir( )

更多资源请访问稀酷客(www.ckook.com)

继续遍历底层目录，否则直接输出文件的完整路径。遍历输出结果如下所示：

```
D:\developer\python\example\07\7.1.1\create_file.py
D:\developer\python\example\07\7.1.1\hello.txt
D:\developer\python\example\07\7.1.2\hello.txt
D:\developer\python\example\07\7.1.2\read_file.py
D:\developer\python\example\07\7.1.3\write_file.py
D:\developer\python\example\07\7.1.3\hello.txt
D:\developer\python\example\07\7.1.4\delete_file.py
D:\developer\python\example\07\7.1.5\copy_file.py
D:\developer\python\example\07\7.1.5\hello3.txt
D:\developer\python\example\07\7.1.6\rename_file.py
D:\developer\python\example\07\7.1.6\alter_extension.py
D:\developer\python\example\07\7.1.6\hi.txt
D:\developer\python\example\07\7.1.6\hello.html
D:\developer\python\example\07\7.1.6\hi.html
D:\developer\python\example\07\7.1.7\find_file.py
D:\developer\python\example\07\7.1.7\hello.txt
D:\developer\python\example\07\7.1.7\hello3.txt
D:\developer\python\example\07\7.1.8\difflib.py
D:\developer\python\example\07\7.1.8\hello.txt
D:\developer\python\example\07\7.1.8\hi.txt
D:\developer\python\example\07\7.1.9\config_file.py
D:\developer\python\example\07\7.1.9\ODBC.INI
D:\developer\python\example\07\7.2.1\make_remove_dir.py
D:\developer\python\example\07\7.2.2\visitdir.py
```

- 第14行代码调用VisitDir()作为程序的起点，遍历“D:\developer\python\example\07”下所有的目录。

2. os.path.walk()

在C或C++语言中，必须使用函数递归的方式对目录进行遍历，而Python还可以使用os.path模块中的函数walk()遍历目录。os.path.walk()的函数声明如下所示：

```
walk(top, func, arg)
```

- 参数top表示需要遍历的目录树的路径。
- 参数func表示回调函数，对遍历路径进行处理。所谓回调函数，是作为某个函数的参数使用，当某个事件触发时，程序将调用定义好的回调函数处理某个任务。回调函数必须提供3个参数：第1个参数为walk()的参数arg，第2个参数表示目录列表，第3个参数表示文件列表。
- 参数arg是传递给回调函数func的元组。回调函数的第1个参数必须是arg，为回调函数提供处理参数。参数arg可以为空元组。

【例7-20】使用os.path模块的walk()遍历目录。

```
import os,os.path

def VisitDir(arg, dirname, names):
    for filepath in names:
        print os.path.join(dirname, filepath)

if __name__ == "__main__":
    path = r"D:\developer\python\example\07"
    os.path.walk(path, VisitDir, ())
```

- 第5行代码的输出结果如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
D:\developer\python\example\07\7.1.1
D:\developer\python\example\07\7.1.2
D:\developer\python\example\07\7.1.3
D:\developer\python\example\07\7.1.4
D:\developer\python\example\07\7.1.5
D:\developer\python\example\07\7.1.6
D:\developer\python\example\07\7.1.7
D:\developer\python\example\07\7.1.8
D:\developer\python\example\07\7.1.9
D:\developer\python\example\07\7.2.1
D:\developer\python\example\07\7.2.2
D:\developer\python\example\07\7.1.1\create_file.py
D:\developer\python\example\07\7.1.1\hello.txt
D:\developer\python\example\07\7.1.2\hello.txt
D:\developer\python\example\07\7.1.2\read_file.py
D:\developer\python\example\07\7.1.3\write_file.py
D:\developer\python\example\07\7.1.3\hello.txt
D:\developer\python\example\07\7.1.4\delete_file.py
D:\developer\python\example\07\7.1.5\copy_file.py
D:\developer\python\example\07\7.1.5\hello3.txt
D:\developer\python\example\07\7.1.6\rename_file.py
D:\developer\python\example\07\7.1.6\alter_extension.py
D:\developer\python\example\07\7.1.6\hi.txt
D:\developer\python\example\07\7.1.6\hello.html
D:\developer\python\example\07\7.1.6\hi.html
D:\developer\python\example\07\7.1.7\find_file.py
D:\developer\python\example\07\7.1.7\hello.txt
D:\developer\python\example\07\7.1.7\hello3.txt
D:\developer\python\example\07\7.1.8\difflib.py
D:\developer\python\example\07\7.1.8\hello.txt
D:\developer\python\example\07\7.1.8\hi.txt
D:\developer\python\example\07\7.1.9\config_file.py
D:\developer\python\example\07\7.1.9\ODBC.INI
D:\developer\python\example\07\7.2.1\make_remove_dir.py
D:\developer\python\example\07\7.2.2\visitdir.py
```

3. os.walk()

os 模块也提供了同名函数 walk()，该函数是 Python2.3 开始发布的。os.walk()可用于目录的遍历，功能类似于 os.path 模块中的函数 walk()。os.walk()比 os.path.walk()的效率高，而且不需要回调函数，更容易使用。os.walk()的函数声明如下所示：

```
walk(top, topdown=True, onerror=None)
```

- 参数 top 表示需要遍历的目录树的路径。
- 参数 topdown 的默认值为“True”，表示首先返回目录树下的文件，然后再遍历目录树的子目录。Topdown 的值为“False”，则表示先遍历目录树的子目录，返回子目录中的文件，最后返回根目录下的文件。
- 参数 onerror 的默认值为“None”，表示忽略文件遍历时产生的错误。如果不为空，则提供 1 个自定义函数提示错误信息后继续遍历或抛出异常中止遍历。
- 该函数返回 1 个元组，该元组有 3 个元素。这 3 个元素分别表示每次遍历的路径名、目录列表和文件列表。

【例 7-21】使用 os.walk()遍历目录“D:\developer\python\example\07”。

```
import os
```

更多资源请访问稀酷客(www.ckook.com)

```
def VisitDir(path):
    for root,dirs,files in os.walk(path):
        for filepath in files:
            print os.path.join(root, filepath)

if __name__ == "__main__":
    path = r"D:\developer\python\example\07"
    VisitDir(path)
```

使用 os 模块的函数 walk( ) 只需要提供 1 个参数 path，即待遍历的目录树的路径。os. walk( ) 实现目录遍历的输出结果和递归函数实现目录遍历的输出结果相同。

注意　os. path. walk( ) 与 os. walk ( ) 产生的文件名列表并不相同。读者可以比较一下 os. path. walk( ) 的输出结果与 os. walk( ) 的输出结果。os. path. walk( ) 产生目录树下的目录路径和文件路径，而 os. walk( ) 只产生文件路径。

7.3　文件和流

读写数据的方式有多种，如文件的读写、数据库的读写等。为了有效的表示数据的读写，把文件、外设、网络连接等数据传输抽象的表示为“流”。数据的传输就好像流水一样，从一个容器到另一个容器。程序中数据的传输也是如此。

7.3.1　Python 的流对象

Python 隐藏了流机制，在 Python 的模块中找不到类似 Stream 的类。Python 把文件的处理和流关联在一起，流对象实现了 File 类的所有方法。sys 模块提供了 3 种基本的流对象——stdin、stdout、stderr。这 3 个对象分别表示标准输入、标准输出和错误输出。流对象可以使用 File 类的属性和方法，流对象的处理和文件的处理方式相同。

1. stdin

对象 stdin 表示流的标准输入。下面这段代码就是通过流对象 stdin 读取文件 hello. txt 的内容：

```
import sys

sys.stdin = open("hello.txt", "r")
for line in sys.stdin.readlines():
    print line
```

- 第 3 行代码调用 open( ) 时，在赋值表达式的左侧使用 sys. stdin 替换文件对象即可。可见，流对象的使用方法和文件对象相同。

2. stdout

对象 stdout 表示流的标准输出。前面的程序都是把程序运行后的结果输出到控制台，这里通过 stdout 对象重定向输出，把输出的结果保存到文件中：

```
import sys

sys.stdout = open(r"./hello.txt", "a")
print "goodbye"
sys.stdout.close()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 3 行代码调用 open( ) 以追加模式打开当前目录下的文件 hello. txt，并把 hello. txt 设置为终端输出设备（默认的终端输出设备是系统的控制台）。
- 第 4 行代码，向文件 hello. txt 输出字符串“goodbye”。
- 第 5 行代码关闭对象 sys. stdout。

3. stderr

日志文件通常用于记录应用程序每次操作的执行结果。日志文件的记录便于维护人员了解当前系统的状况，甚至可用于数据的恢复（例如数据库日志文件）。Python 的 stderr 对象用于记录、输出异常信息，通过 stderr 对象可以实现日志文件的功能。例如，当前目录中的文件 hello. txt 内容为空，则在日志文件 error. log 中记录异常信息。如果文件 hello. txt 的内容不为空，则在日志文件 error. log 中记录正确的信息。下面这段代码实现了这部分功能：

```
import sys,time

sys.stderr = open("record.log", "a")
f = open(r"./hello.txt","r")
t = time.strftime("%Y-%m-%d %X", time.localtime())
context = f.read()
if context:
    sys.stderr.write(t + " " + context)
else:
    raise Exception, t + " 异常信息"
```

- 第 3 行代码调用 open( ) 以追加模式打开当前目录下的日志文件 record. log。record. log 成为记录日志的设备。
- 第 4 行代码打开文件 hello. txt。
- 第 5 行代码获取当前的系统时间。
- 第 6 行代码读取文件 hello. txt 的内容，并赋给变量 context。
- 第 7 行代码判断文件 hello. txt 的内容是否为空。
- 第 8 行代码，如果文件的内容不为空，则向 record. log 文件记录当前时间和 hello. txt 的内容：

```
2008-02-24 19:30:09 goodbye
```

- 第 10 行代码，如果文件的内容为空则抛出异常，并在 record. log 文件中记录当前时间和异常信息。输出结果：

```
Traceback (most recent call last):
  File "stderr.py", line 12, in <module>
    raise Exception, t + " 异常信息"
Exception: 2008-02-24 17:56:46 异常信息
```

7.3.2　模拟 Java 的输入、输出流

Java 的 java. io 包提供了一系列处理流的类，根据数据流动的方向分为输入流和输出流。例如，InputStream 类、OutputStream 类。通过第 5 章 5.3.7 节学习的 Generator 函数，可以很方便的构造出流的实现函数。

【例 7-22】模拟 Java 的输入、输出流读写文本文件。首先调用函数 FileInputStream( ) 读取该函数的参数 hello. txt 文件，然后调用函数 FileOutputStream( ) 把 FileInputStream( ) 读取的内容写入到文件 hello2. txt 中。

更多资源请访问稀酷客(www.ckook.com)

```
#文件输入流
def FileInputStream(filename):
    try:
        f = open(filename)
        for line in f:
            for byte in line:
                yield byte
    except StopIteration, e:
        f.close()
        return

#文件输出流
def FileOutputStream(inputStream, filename):
    try:
        f = open(filename, "w")
        while True:
            byte = inputStream.next()
            f.write(byte)
    except StopIteration, e:
        f.close()
        return

if __name__ == "__main__":
    FileOutputStream(FileInputStream("hello.txt"), "hello2.txt")
```

- 第 2 行代码定义了函数 FileInputStream( )。该函数采用 Generator 函数的风格设计，模拟 Java 的 FileInputStream 类的工作方式。
- 第 4 行代码以只读的方式打开 hello. txt 文件。
- 第 5 行代码控制读取文件每行的数据。
- 第 6 行代码控制读取每个字节的数据。
- 第 7 行代码使用 yield 保留字，返回每个字节的数据。
- 第 13 行代码定义了函数 FileOutputStream( )。该函数模拟 Java 的 FileOutputStream 类的工作方式，读取输入流的数据并写入 hello2. txt 文件中。
- 第 15 行代码以写入的方式打开 hello2. txt 文件。
- 第 17 行代码获取 FileInputStream( ) 中 yield 保留字产生的字节。
- 第 18 行代码把每个字节写入到 hello2. txt 文件中。

注意　文件处理中可能遇到各种非正常的情况，使用 try... except 语句捕获异常，可以提高程序的健壮性。有关 try... except 的知识，请参考第 9 章。

7.4　文件处理示例——文件属性浏览程序

本节将运用前面学习的内容，设计一个浏览文件属性的程序。通过给定的目录路径查看文件的名称、大小、创建时间、最后修改日期和最后访问日期。设计一个函数 showFileProperties (path)，path 表示目录的路径，该函数可以查看 path 目录下所有文件的属性。showFileProperties( ) 的实现大致分为如下 3 个步骤。

（1）遍历 path 指定的目录，获取每个子目录的路径。

（2）遍历子目录下的所有文件，并返回文件的属性列表。

（3）分解属性列表，对属性列表的值进行格式化输出。

步骤（1）可以采用 7. 2. 2 节的思路实现目录的遍历。步骤（2）通过 os 模块的函数 stat( )

更多资源请访问稀酷客(www.ckook.com)

返回文件的属性列表，该属性列表包括文件的大小、创建时间、最后修改时间、最后访问时间等信息。步骤（3）通过列表的索引获取文件的各个属性。对于文件的时间属性，调用 time 模块的函数 localtime( )返回时间类型，并结合字符串的格式化输出文件的时间属性。

【例 7-23】设计并调用函数 showFileProperties( )，列出“D：\ developer \ python \ example \ 07 \ 7.2.2”路径下所有文件的属性。

```
def showFileProperties(path):
    "显示文件的属性。包括路径、大小、创建日期、最后修改时间,最后访问时间"
    import time,os
    for root,dirs,files in os.walk(path,True):
        print "位置:" + root
        for filename in files:
            state = os.stat(os.path.join(root, filename))
            info = "文件名:" + filename + " "
            info = info + "大小:" + ("%d" % state[-4]) + " "
            t = time.strftime("%Y-%m-%d %X", time.localtime(state[-1]))
            info = info + "创建时间:" + t + " "
            t = time.strftime("%Y-%m-%d %X", time.localtime(state[-2]))
            info = info + "最后修改时间:" + t + " "
            t = time.strftime("%Y-%m-%d %X", time.localtime(state[-3]))
            info = info + "最后访问时间:" + t + " "
            print info

if __name__ == "__main__":
    path = r"D:\developer\python\example\07\7.2.2"
    showFileProperties(path)
```

- 第 4 行代码，调用函数 os.walk( )对 path 指定的路径进行遍历。
- 第 5 行代码，输出 path 路径下每个子目录的路径。输出结果如下所示：

```
位置:D:\developer\python\example\07\7.2.2
```

- 第 6 行代码，对每个子目录进行遍历，返回文件名称。
- 第 7 行代码，调用 os.stat( )返回文件的属性列表。
- 第 9 行代码，state［-4］表示文件的大小。
- 第 10 行代码，调用 time 模块的 strftime( )把时间类型转换为字符串。其中 state［-1］表示文件的创建时间。
- 第 12 行代码，state［-2］表示文件的最后修改时间。
- 第 14 行代码，state［-3］表示文件的最后访问时间。
- 第 16 行代码输出文件的属性信息。输出的内容如下所示：

```
文件名:visitdir.py 大小:392 创建时间:2008-2-22 10:46:49 最后修改时间:2008-2-22 11:9:26
最后访问时间:2008-2-24 0:0:0
文件名:os_path_walk.py 大小:335 创建时间:2008-2-15 17:1:44 最后修改时间:2008-2-24 12:
32:28 最后访问时间:2008-2-24 0:0:0
文件名:os_walk.py 大小:321 创建时间:2008-2-22 12:45:5 最后修改时间:2008-2-22 14:42:22 最
后访问时间:2008-2-24 0:0:0
```

注意　os.stat( )的参数必须是绝对路径。因此，需要先调用 os.path.join（root，filename）连接文件的路径和文件名。

7.5　小结

本章介绍了 Python 文件编程方面的知识。讲解了文件的打开、读取、写入、删除等操作的

更多资源请访问稀酷客(www.ckook.com)

实现，以及针对目录的编程，包括目录的创建、删除、遍历等。重点介绍了目录的遍历，并采用了递归函数、os. path. walk( )以及 os. walk( )等 3 种方法实现目录的遍历。结合程序开发中经常使用的配置文件，讲解了对配置文件的配置项进行读取、添加、删除、修改等操作。Python 对流机制实现了很好的封装，程序员对文件的操作就是通过流的方式来进行处理。介绍了 Python 中常见的流对象，并实现了流的重定向。最后结合目录的遍历、时间的处理、字符串的格式化等知识，实现了查看文件属性的程序。

更多资源请访问稀酷客(www.ckook.com)

第 8 章　面向对象编程

面向对象的程序设计提供了一种新的思维方式，软件设计的焦点不再是程序的逻辑流程，而是软件或程序中的对象以及对象之间的关系。使用面向对象的思想进行程序设计，能够更好的设计软件架构、维护软件模块、易于框架和组件的重用。随着 Python2.5 版的推出，面向对象编程的机制已经非常完善了，本章将结合面向对象的基本概念和 Python 语法的特性讲解面向对象的编程。本章的知识点：

- 类和对象
- 属性和方法
- 继承和组合
- 类的多态性
- 类的访问权限
- 设计模式的应用

8.1　面向对象编程概述

面向对象是一种方法学。面向对象是一种描述业务问题——设计业务实体和实体之间关系的方法。面向对象技术已经成为当今软件设计和开发领域的主流技术。面向对象主要用于软件开发的分析和设计阶段，通常使用 UML（统一建模语言）进行建模。

统一建模语言并不是软件设计开发的方法，而是一种描述软件开发过程的图形化标记。UML 使用若干种模型图来描述软件开发中的每个重要步骤。

1. 用例图

用例图描述系统使用者与系统之间的交互关系，用例图通常用于系统的分析阶段，分析系统中主要的流程。类图用于描述不同的业务实体以及实体之间的关系。

2. 活动图

活动图是对用例图的补充，用于分析复杂的用例，表示用例中的逻辑行为。活动图类似于传统的数据流程图，可以描述业务的流程、帮助分析人员确定用例中的交互和用例之间的交互关系。

3. 状态图

状态图用于对系统的行为进行建模，用于分析和设计阶段的之间的过渡时期。状态图和活动图有些相似，状态图是对单个对象的描述，强调对象内部状态的变迁过程。

4. 类图

类图包括属性、操作以及访问权限等内容。类图用于系统设计阶段，根据系统用例提炼出系统中的对象。类图是面向对象设计的核心，通过类图可以表示抽象、继承、多态等面向对象特

更多资源请访问稀酷客(www.ckook.com)

性。能够表现系统的架构设计，系统中对象之间的层次关系。

5. 序列图和协助图

序列图和协助图都可以用于描述系统中对象之间的行为，是对系统具体行为的设计。序列图和协助图通常用于类图设计完成后的一个步骤。序列图是对用例的具体实现，通过消息的方式描述系统中对象的生命周期和控制流程。而协助图突出对消息的建模，描述系统对象之间的交互关系。

6. 组件图和部署图

组件图用于描述系统组件之间的交互以及类和组件之间的依赖关系。部署图用于描述系统的部署以及组件之间的配置。

这些图形标记都可以使用 UML 建模工具绘制，例如 visio 和 Rational Rose。使用这些建模工具可以准确、快速的描述系统的行为。visio 是 office 系列的工具之一，使用简单、容易上手。而 Rational Rose 功能强大，能通过模型生成各种语言的代码，以及对代码更新来同步模型等特性。使用 Rational Rose 能更好的管理对象，但是 Rational Rose 软件庞大，必须学习每种图形的绘制方法。建模工具只是辅助系统分析、设计的手段，系统分析人员甚至可以使用手写的方式来描述系统的模型。重要的是能够清晰、准确的分析和设计，而不是软件工具的优劣。

8.2　类和对象

在面向对象的设计中，程序员可以创建任何新的类型，这些类型可以描述每个对象包含的数据和特征，这种类型称之为类。类是一些对象的抽象，隐藏了对象内部复杂的结构和实现。类由变量和函数两部分构成，类中的变量称之为成员变量，类中的函数称之为成员函数。

8.2.1　类和对象的区别

类和对象是面向对象中的两个重要概念，初学者经常把类和对象混为一谈。类是客观世界中事物的抽象，而对象是类实例化后的变量。例如，汽车模型可以制造出不同的汽车，每辆汽车就是一个对象。车牌号可以标识每辆汽车，不同的汽车有不同的颜色和价格，因此颜色、价格是汽车的属性，图 8-1 描述了类和对象的关系。

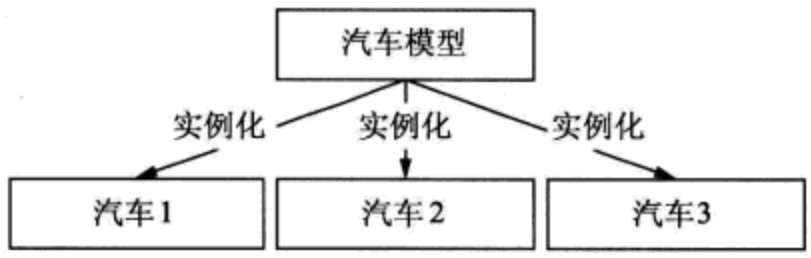


图 8-1　类和对象的关系

汽车模型是对汽车特征和行为的抽象，而汽车是实际存在的事物，是客观世界中实实在在的实体。因此，根据类的定义可以构造出许多对象。现实生活中可以看到很多这样的例子——根据零件模型可以制造出多个零件，根据施工图纸可以建造出多栋楼房。

8.2.2　类的定义

Python 使用 class 保留字定义一个类，类名的首字符一般要大写。当程序员需要创建的类型不能用简单类型来表示时，就需要定义类，然后利用定义的类创建对象。类把需要使用的变量和函数组合在一起，这种方式称之为封装。类定义的结构如下所示：

```
class class_name :
    ...
```

更多资源请访问稀酷客(www.ckook.com)

类必须用保留字 class 定义，保留字 class 后面是类名。类的主体由一系列的属性和方法组成。下面这段代码创建了 1 个简单的水果类，水果具有成长的行为，因此定义了 1 个方法 grow()：

```
# 类的创建
class Fruit:
    def grow(self):
        print "Fruit grow..."
```

- 第 2 行代码定义了一个名为“Fruit”的类。第 3 行代码，在类 Fruit 中创建了 1 个方法 grow()。类的方法至少有 1 个参数 self，self 参数的含义会在后面的内容中介绍。

注意　*类的方法必须有 1 个 self 参数。但是方法被调用时，可以不传递这个参数。*

8.2.3　对象的创建

创建对象的过程称之为实例化，当一个对象被创建后，包含三个方面的特性：对象的句柄、属性和方法。对象的句柄用于区分不同的对象，当对象被创建后，该对象会获取一块存储空间，存储空间的地址即为对象的标识。对象的属性和方法与类的成员变量和成员函数相对应：

```
if __name__ == "__main__":
    fruit = Fruit()
    fruit.grow()
```

- 第 2 行代码对 Fruit 类进行实例化，Python 的实例化与 Java 不同，并不需要 new 保留字。Python 的实例化与函数的调用相似，使用类名加圆括号的方式。第 3 行代码调用 Fruit 类的 grow()方法。输出结果：Fruit grow . . . 。

8.2.4　Python 对象的体系结构

Python2.5 版中的对象分为经典对象（Classic Class）和新型对象（New-style Class）。在 Python2.1 及其之前的版本，只能使用经典对象来实现面向对象的程序设计。由于 Python 中提供了内置类型（例如元组、列表、字典等数据结构），因此从 Python2.2 版开始对新型对象加以支持，这些特殊数据类型继承自统一的新型对象。Python2.5 的新型对象已经十分完善了。Python 中的任何类型都是对象，包括字符串类型、数字类型、内置数据结构等。因此，类和实例之间的界限变得模糊了：类和实例都属于对象。

1. 经典对象

经典对象的定义如 8.2.2 节所示。经典对象的内部由属性和方法组成。属性可以是数据类型，也可以是函数类型。如果属性是函数类型，则该属性就是 1 个方法。Python 的方法很特别，解释器把类的方法也看作是属性。类的内部提供了一些内置的方法，方法名是由前后两个下划线加字母组成。例如，Python 的构造函数__init__。字符串“init”前后都有两个下划线。

2. 新型对象

新型对象内建了 object 对象，该对象是内建类型和新型对象的父类，或称为基类。object 也可以看作是对象。如果要定义新型对象，必须继承 object 对象。语法格式如下所示。

```
class_name(object):
    ...
```

新型对象提供了对类方法和静态方法的支持。类方法可以被类和类的实例化对象调用。classmethod()可以把方法定义为类方法。静态方法可以被类直接调用，可以被所有的实例化对象共享。staticmethod()可以把方法定义为静态方法。使用方法详见 8.3.2 节。

更多资源请访问稀酷客(www.ckook.com)

8.3　属性和方法

类由属性和方法组成。类的属性是对数据的封装，而类的方法则表示对象具有的行为。Python 的构造函数、析构函数、私有属性或方法都是通过名称约定区分的。此外，Python 还提供了一些有用的内置方法，简化了类的实现。

8.3.1　类的属性

类的属性分为私有属性和公有属性。C#、Java 语言的私有属性或方法使用 private 保留字定义，不同的是 Python 并没有提供公有和私有权限的修饰符。类的私有属性不能被该类之外的函数调用，类属性的作用范围完全取决于属性的名称。如果函数、方法或属性的名字以两个下划线开始，则表示私有类型；没有使用两个下划线开始则表示公有属性。类的方法同样遵循这个约定，Python 中有许多语法是通过约定变量名称的方式实现的。

注意　Python 没有保护类型的修饰符。

Python 的属性分为实例属性和静态属性。实例属性是以 self 作为前缀的属性。__init__方法即 Python 类的构造函数，详细用法请参考 8.3.4 节。如果__init__方法中定义的变量没有使用 self 作为前缀声明，则该变量只是普通的局部变量。类中其他方法定义的变量也只是局部变量，而非类的实例属性。

C#、Java 中有一类特殊的属性称为静态变量。静态变量可以被类直接调用，而不被实例化对象调用。当创建新的实例化对象后，静态变量并不会获得新的内存空间，而是使用类创建后静态变量的内存空间。因此，静态变量能够被多个实例化对象共享。在 Python 中静态变量称之为静态属性。

【例 8-1】演示实例属性和类属性的区别。

```
class Fruit:
    price = 0                          # 类属性

    def __init__(self):
        self.color = "red"             # 实例属性
        zone = "China"                 # 局部变量

if __name__ == "__main__":
    print Fruit.price
    apple = Fruit ()
    print apple.color
    Fruit.price = Fruit.price + 10
    print "apple's price:" + str(apple.price)
banana = Fruit ()
print "banana's price:" + str(banana.price)
```

- 第 2 行代码定义了公有的类属性 price，并设置初始值为“0”。
- 第 5 行代码定义了公有的实例属性 color，并设置值为“red”。实例属性前需要使用前缀 self 声明。
- 第 6 行代码定义了局部变量 zone，该变量不能被 Fruit 的实例化对象使用。如果实例化对象直接引用 zone，Python 将抛出错误：AttributeError：Fruit instance has no attribute 'zone'。
- 第 9 行代码输出类属性 price 的值。输出结果：0。
- 第 10 行代码对 Fruit 类实例化，生成对象 apple。

更多资源请访问稀酷客(www.ckook.com)

- 第 11 行代码输出实例属性 color 的值。输出结果：red。
- 第 12 行代码设置类属性 price 的值为“10”。
- 第 13 行代码输出 apple 对象的属性 price 的值。输出结果：apple's price：10。
- 第 14 行代码对 Fruit 类实例化，生成对象 banana。
- 第 15 行代码输出 banana 对象的属性 price 的值。输出结果：banana's price：10。

注意　Python 的类和对象都可以访问类属性，而 Java 中的静态变量只能被类调用。

类的外部不能直接访问私有属性。如果把 color 属性改为私有属性__color，当执行如下语句时，Python 解释器将不能识别属性__color：

```
print fruit.__color
```

Python 解释器将提示如下错误：

```
AttributeError: Fruit instance has no attribute '__color'
```

Python 提供了直接访问私有属性的方式，可用于程序的测试和调试。私有属性访问的格式如下所示：

```
instance._classname__attribute
```

- instance 表示实例化对象；classname 表示类名；attribute 表示私有属性。

【例 8-2】接上例。对 Fruit 类做一些修改，采用直接访问私有属性的方式调用私有属性。

```
# 访问私有属性
class Fruit:
    def __init__(self):
        self.__color = "red"          # 私有属性

if __name__ == "__main__":
    apple = Fruit ()
    print apple._Fruit__color
```

- 第 4 行代码定义了私有属性__color。第 8 行代码输出私有属性__color 的值。输出结果：red。

上述代码的主程序可以直接访问 Fruit 的属性 color，这种直接暴露数据的做法是不提倡的。因为这种方式可以随意的更改实例属性的值，存在程序数据安全方面的问题。这种访问方式只用于开发阶段的测试或调试。通常的做法是定义相关的 get 方法获取实例属性的值。例如，在 Fruit 类中定义 getColor( )方法。下一节将介绍方法的定义和使用。

注意　Python 对类的属性和方法的定义次序没有要求。合理的方式是把类属性定义在类中的最前面，然后再定义私有方法，最后定义公有方法。

类提供了一些内置属性，用于管理类的内部关系。例如，__dict__、__bases__、__doc__等。

【例 8-3】演示常见内置属性的使用方法。

```
class Fruit:
    def __init__(self):
        self.__color = "red"

class Apple(Fruit):                   # Apple 继承了 Fruit
    pass

if __name__ == "__main__":
    fruit = Fruit ()
    apple = Apple ()
    print Apple.__bases__             # 输出基类组成的元组
```

更多资源请访问稀酷客(www.ckook.com)

```
    print apple.__dict__                    # 输出属性组成的字典
    print apple.__module__                  # 输出类所在的模块名
print apple.__doc__                     # 输出 doc 文档
```

- 第 11 行代码，输出 Apple 类的父类组成的元组。由于 Python 支持多重继承，所以可能存在多个父类。输出结果：(<class __main__. Fruit at 0x00BE4AE0 > ,)。
- 第 12 行代码输出 apple 对象中属性组成的字典。输出结果：{'_Fruit__color': 'red'}。
- 第 13 行代码，当前运行的模块名即为“__main__”。输出结果：__main__。
- 第 14 行代码，由于没有定义类的 doc 字符串，所以输出结果为“None”。输出结果：None。

8.3.2　类的方法

类的方法也分为公有方法和私有方法。私有函数不能被该类之外的函数调用，私有的方法也不能被外部的类或函数调用。C#、Java 中的静态方法使用保留字 static 声明，而 Python 使用函数 staticmethod( )或“@ staticmethod”指令的方式把普通的函数转换为静态方法。Python 的静态方法并没有和类的实例进行名称绑定，Python 的静态方法相当于全局函数。

【例 8-4】演示类的方法和静态方法的使用。

```
class Fruit:
    price = 0

    def __init__(self):
        self.__color = "red"

    def getColor(self):
        print self.__color

    @ staticmethod                                  # 静态方法
    def getPrice ():
        print Fruit.price

    def __getPrice ():
        Fruit.price = Fruit.price + 10
        print Fruit.price

    count = staticmethod(__getPrice)                # 静态方法

if __name__ == "__main__":
    apple = Fruit ()
    apple.getColor ()
    Fruit.count ()
    banana = Fruit ()
    Fruit.count ()
    Fruit.getPrice ()
```

- 第 7 行代码定义了公有方法 getColor( )。获取属性__color 的值。
- 第 10 行代码使用“@ staticmethod”指令声明方法 getPrice 为静态方法。
- 第 11 行代码定义 getPrice( )方法，该方法没有 self 参数。
- 第 14 行代码定义了方法__getPrice( )，该方法没有提供参数 self。
- 第 18 行代码调用 staticmethod( )把__getPrice( )转换为静态方法，并赋值给变量 count。count( )即为 Fruit 类的静态方法。
- 第 22 行代码调用 fruit 对象的方法 getColor( )，返回属性__color 的值。
- 第 23 行代码调用静态方法 count( )。由于创建了对象 apple，所以静态属性 price 执行一

更多资源请访问稀酷客(www.ckook.com)

次加“10”的运算。输出结果：10。

- 第 25 行代码，由于创建了对象 banana，因此静态方法 count( )被第 2 次调用。静态属性 price 的值为“20”。输出结果：20。
- 第 26 行代码，调用静态方法 getPrice( )。输出结果：20。

类的外部不能直接访问私有方法。Python 解释器同样不能识别方法__getPrice( )。Python 解释器将提示错误：AttributeError：Fruit instance has no attribute'__getPrice'。

上述代码的 getColor( )方法中有 1 个 self 参数，该参数是区别方法和函数的标志。类的方法至少需要 1 个参数，调用方法时不必给该参数赋值。通常这个特殊的参数被命名为 self，self 参数表示指向对象本身。self 参数用于区分函数和类的方法。self 参数等价于 Java、C#中的 this 保留字。当调用 Fruit 类的 getColor( )方法时，Python 会把函数的调用转换为 grow（fruit)。Python 自动完成了对象 fruit 的传递任务。

Python 中还有一种方法称之为类方法。类方法的作用与静态方法相似，都可以被其他实例对象共享。不同的是类方法必须提供 self 参数。类方法可以使用函数 classmethod( )或“@ classmethod”指令定义。

【例 8-5】接上例，把【例 8-4】程序的静态方法修改为类方法。

```
@ classmethod                                    # 类方法
def getPrice(self):
    print self.price

def __getPrice(self):
    self.price = self.price + 10
    print self.price

count = classmethod(__getPrice)                  # 类方法
```

可见，类方法的使用和静态方法是十分相似的。如果某个方法需要被其他实例共享，同时又需要使用当前实例的属性，则将其定义为类方法。

注意　self 参数的名称可以是任意合法的变量名。建议使用“self”作为参数名，便于程序的阅读和程序的统一。

8.3.3　内部类的使用

Java 可以在类的内部定义类，Python 同样存在这种语法。例如，前面提到的汽车模型。汽车由门、车轮等部件组成，可以设计出汽车、门、车轮等 3 个类。而门、车轮是汽车的一部分，因此把门、车轮表示的类放到汽车的内部。这种在某个类内部定义的类称之为内部类。内部类中的方法可以使用两种方法调用。

第一种方法是直接使用外部类调用内部类，生成内部类的实例，再调用内部类的方法。调用格式如下所示：

```
object_name = outclass_name.inclass_name ()
object_name.method ()
```

其中，outclass_name 表示外部类的名称，inclass_name( )表示内部类的名称，object_name 表示内部类的实例。

第二种方法是先对外部类进行实例化，然后再实例化内部类，最后调用内部类的方法。调用格式如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
out_name = outclass_name ()
in_name = out_name. inclass_name ()
in_name. method ()
```

其中，out_ name ( ) 表示外部类的实例，in_ name 表示内部类的实例。

【例 8-6】演示内部类的使用方法。

```
class Car:
    class Door:                          # 内部类
        def open(self):
            print "open door"
    class Wheel:                         # 内部类
        def run(self):
            print "car run"

if __name__ == "__main__":
    car = Car ()
    backDoor = Car. Door ()                  # 内部类的实例化方法一
    frontDoor = car. Door ()                 # 内部类的实例化方法二
    backDoor. open ()
    frontDoor. open ()
    wheel = Car. Wheel ()
    wheel. run ()
```

- 第 2 行代码定义了内部类 Door。
- 第 3 行代码定义了 Door 类的方法 open( )。

第 5 行代码定义了内部类 Wheel。

- 第 6 行代码定义了 Wheel 类的方法 run( )。
- 第 10 行代码创建 Car 类的实例 car。
- 第 11 行代码创建 Door 类的实例 backDoor。这里使用类名前导的方式创建。
- 第 12 行代码创建 Door 类的实例 frontDoor。这里使用对象名前导的方式创建。
- 第 13 行、第 14 行代码调用 open( )方法，输出结果：open door。
- 第 15 行代码创建 Wheel 类的实例 wheel。
- 第 16 行代码调用 run( )，输出结果：car run。

注意　内部类不适合描述类之间的组合关系，而应把 Door、Wheel 类的对象作为类的属性使用。内部类容易造成程序结构的复杂，不提倡使用。

8.3.4　__init__方法

构造函数用于初始化类的内部状态，为类的属性设置默认值。C#、Java 的构造函数是与类同名的方法，而 Python 的构造函数名为__init__。__init__方法除了用于定义实例变量外，还用于程序的初始化。__init__方法是可选的，如果不提供__init__方法，Python 将会给出 1 个默认的__init__方法。

【例 8-7】演示对 Fruit 类进行初始化。

```
class Fruit:
    def __init__(self, color):          # 初始化属性__color
        self. __color = color
        print self. __color

def getColor(self):
        print self. __color
```

更多资源请访问稀酷客(www.ckook.com)

```
def setColor(self, color):
        self.__color = color
        print self.__color
if __name__ == "__main__":
    color = "red"
    fruit = Fruit(color)                    # 带参数的构造函数
    fruit.getColor ()
    fruit.setColor("blue")
```

- 第 2 行代码，__init__是类 Fruit 的初始化函数，即构造函数。该函数根据参数 color 的值初始化属性__color。
- 第 3 行代码获取参数 color 的值，对 Fruit 类的属性__color 进行赋值，需要前缀 self。否则 Python 解释器将认为__color 是__init__( )中的局部变量。
- 第 4 行代码输出赋值后的属性__color。输出结果：red。
- 第 9 行代码定义方法 setColor( )，设置属性__color 的值。如果初始化后需要改变属性__color的值，则调用 setColor( )方法。
- 第 15 行代码生成 fruit 对象。这里将调用函数__init__( )，并传递变量 color 的值。

第 16 行代码调用方法 getColor( )，返回属性__color 的值。输出结果：red。

第 17 行代码调用方法 setColor( )，重新设置属性__color 的值。输出结果：blue。

Python 可以随时调用__init__( )函数，而 C#、Java 并不能直接调用构造函数：

```
fruit.__init__(color)
```

Java 程序员应该对上面的代码非常熟悉，其中获取数据使用 get 方法，而设置数据使用 set 方法。

注意　__init__方法和 Java、C++ 等开发语言一样不能返回任何值。

8.3.5　__del__方法

析构函数用于释放对象占用的资源。Python 提供了析构函数__del__( )。析构函数可以显式的释放对象占用的资源，是另一种释放资源的方法。析构函数也是可选的。如果程序中不提供析构函数，Python 会在后台提供默认的析构函数。

【例 8-8】演示 Python 的函数__del__( )的使用方法。

```
class Fruit:
    def __init__(self, color):              # 初始化属性__color
        self.__color = color
        print self.__color

    def __del__(self):                      # 析构函数
        self.__color = ""
        print "free..."

    def grow(self):
        print "grow..."

if __name__ == "__main__":
    color = "red"
    fruit = Fruit(color)                    # 带参数的构造函数
    fruit.grow ()
```

- 第 6 行代码定义了析构函数__del__( )。参数 self 对于析构函数同样是不可缺少的。
- 第 16 行代码调用方法 grow( )。当程序执行完方法 grow( )后，对象 fruit 将超出其作用

更多资源请访问稀酷客(www.ckook.com)

域，Python 会结束对象 fruit 的生命周期。输出结果：free . . . 。

如果要显式的调用析构函数，可以使用 del 语句。在程序的末尾添加如下语句：

```
del fruit                                    # 执行析构函数
```

对于C++ 语言，析构函数是必须的，程序员需要为对象分配内存空间，同时也要手动释放内存。使用 Python 编写程序可以不考虑后台的内存管理，直接面对程序的逻辑即可。

8.3.6　垃圾回收机制

Java 中没有提供析构函数，而是采用垃圾回收机制清理不再使用的对象。Python 也采用垃圾回收机制清除对象，Python 提供了 gc 模块释放不再使用的对象。垃圾回收的机制有许多种算法，Python 采用的是引用计数的方式。当某个对象在其作用域内不再被其他对象引用时，Python 就会自动清除该对象。垃圾回收机制很好地避免了内存泄漏的发生。函数 collect( ) 可以一次性收集所有待处理的对象。

【例 8-9】 使用 gc 模块显式的调用垃圾回收器。

```
import gc

class Fruit:
    def __init__(self, name, color):                    # 初始化 name、color 属性
        self.__name = name
        self.__color = color

    def getColor(self):
        return self.__color

    def setColor(self, color):
        self.__color = color

    def getName(self):
        return self.__name

    def setColor(self, name):
        self.__name = name

class FruitShop:                                         # 水果店类
    def __init__(self):
        self.fruits = []

    def addFruit(self, fruit):                           # 添加水果
        fruit.parent = self                              # 把 Fruit 类关联到 FruitShop 类
        self.fruits.append(fruit)

if __name__ == "__main__":
    shop = FruitShop ()
    shop.addFruit(Fruit("apple", "red"))
    shop.addFruit(Fruit("banana", "yellow"))
    print gc.get_referrers(shop)
    del shop
    print gc.collect ()                                  # 显示调用垃圾回收器
```

- 第 4 行代码，在__init__方法中定义了两个属性。__name 表示水果的名称，__color 表示水果的颜色。
- 第 20 行代码定义了 FruitShop 类，表示水果店。
- 第 22 行代码为 FruitShop 类定义了属性 fruits。fruits 是 1 个列表，用于存放水果店中的水果。

更多资源请访问稀酷客(www.ckook.com)

- 第 24 行代码定义了方法 addFruit( )，把对象 fruit 添加到 fruits 列表中。
- 第 25 行代码，设置 fruit 对象的 parent 属性为 self。即把 FruitShop 实例化对象的引用关联到添加的 fruit 对象上。
- 第 30 行、第 31 行代码，向 shop 对象中添加两个 fruit 对象。
- 第 32 行代码，调用函数 get_ referrers( )列出 shop 对象关联的所有对象。输出结果如下所示：

```
    [{'_Fruit__name': 'apple', 'parent': <__main__.FruitShop instance at 0x00BEF0F8>, '_Fruit
__color': 'red'}, {'_Fruit__name': 'banana', 'parent': <__main__.FruitShop instance at
0x00BEF0F8>, '_Fruit__color': 'yellow'}, {'shop': <__main__.FruitShop instance at
0x00BEF0F8>,
    'FruitShop': <class __main__.FruitShop at 0x00BE3AE0>,
    '__builtins__': <module '__builtin__' (built-in)>,
    '__file__': 'gc.py',
    'Fruit': <class __main__.Fruit at 0x00BE3AB0>,
    'gc': <module 'gc' (built-in)>,
    '__name__': '__main__',
    '__doc__': None}]
```

- 第 33 行代码，删除 shop 对象，但是 shop 对象关联的其他对象并没有释放。
- 第 34 行代码，调用函数 collect( )释放 shop 对象关联的其他对象，collect( )返回结果表示释放的对象个数。输出结果：7。

注意　垃圾回收器在后台执行，对象被释放的时间是不能确定的。如果要设置调用垃圾回收器，可以使用 gc 模块的函数实现。

8.3.7　类的内置方法

Python 类定义了一些专用的方法，这些专用方法可以用于不同的应用场合，丰富了程序设计的功能。前面提到的__init__( )、__del__( )就是类的内置方法。表 8-1 列出了类常用的内置方法。

表 8-1　类常用的内置方法

内 置 方 法	说　明
__init__（self,...）	初始化对象，在创建新对象时调用
__del__（self）	释放对象，在对象被删除之前调用
__new__（cls，*args，**kwd）	实例的生成操作
__str__（self）	在使用 print 语句时被调用
__getitem__（self，key）	获取序列的索引 key 对应的值，等价于 seq［key］
__len__（self）	在调用内联函数 len( )时被调用
__cmp__（src，dst）	比较两个对象 src 和 dst
__getattr__（s，name）	获取属性的值
__setattr__（s，name，val）	设置属性的值
__delattr__（s，name）	删除 name 属性
__getattribute__( )	__getattribute __( )的功能与__getattr __( )类似
__gt__（self，other）	判断 self 对象是否大于 other 对象

更多资源请访问稀酷客(www.ckook.com)

（续）

内置方法	说　明
__lt__（self, other）	判断 self 对象是否小于 other 对象
__ge__（self, other）	判断 self 对象是否大于或等于 other 对象
__le__（self, other）	判断 self 对象是否小于或等于 other 对象
__eq__（self, other）	判断 self 对象是否等于 other 对象
__call__（self, * args）	把实例对象作为函数调用

下面对类的常用的内置方法进行说明。

1. __new__()

__new__()在__init__()之前被调用，用于生成实例对象。利用这个方法和类属性的特性可以实现设计模式中的单例模式。单例模式是指创建唯一对象，单例模式设计的类只能实例化1个对象。

【例8-10】　实现单例模式。

```
class Singleton(object):
    __instance = None                                    #定义实例

    def __init__(self):
        pass

    def __new__(cls, * args, * * kwd):                   #在__init__之前调用
        if Singleton.__instance is None:                 #生成唯一实例
            Singleton.__instance = object.__new__(cls, * args, * * kwd)
        return Singleton.__instance
```

- 第2行代码定义了私有的类属性__instance，初始值为“None”。__instance 的值是第1次创建的实例，以后的实例化操作都将共享这个属性。因此，就达到了创建唯一实例的目的。
- 第7行代码重新实现了方法__new__()。
- 第8行代码判断当前的实例是否为“None”，如果为“None”，则调用父类 object 的__new__()方法，生成1个唯一实例。
- 第10行代码返回唯一实例。

2. __getattr__()、__setattr__()和__getattribute__()

当读取对象的某个属性时，Python 会自动调用__getattr__()方法。例如，fruit.color 将转换为 fruit.__getattr__（color）。当使用赋值语句对属性进行设置时，Python 会自动调用__setattr__()方法。__getattribute__()的功能与__getattr__()类似，用于获取属性的值。但是__getattribute__()能提供更好的控制，代码更健壮。

【例8-11】 演示 color、price 属性获取和设置的方法。

```
class Fruit(object):
    def __init__(self, color = "red", price = 0):
        self.__color = color
        self.__price = price

    def __getattribute__(self, name):                    #获取属性的方法
        return object.__getattribute__(self, name)
```

更多资源请访问稀酷客(www.ckook.com)

```
    def __setattr__(self, name, value):                    # 设置属性的方法
        self.__dict__[name] = value

if __name__ == "__main__":
    fruit = Fruit("blue", 10)
    print fruit.__dict__.get("_Fruit__color")                    # 获取 color 属性
    fruit.__dict__["_Fruit__price"] = 5
    print fruit.__dict__.get("_Fruit__price")                  # 获取 price 属性
```

- 第 6 行代码实现了__getattribute__()方法，name 表示属性名。
- 第 9 行代码实现了__setattr__()，设置属性的值。name 表示属性名，value 表示待设置的值。
- 第 10 行代码，__dict__字典是类的内置属性，用于记录类定义的属性。字典中的 key 表示类名，value 表示属性的值。
- 第 14 行代码获取 color 属性的值，因为 color 是私有属性，所以字典的索引表示为“_Fruit__color”：

```
blue
```

- 第 15 行代码设置 price 属性的值。
- 第 16 行代码获取 price 属性的值。输出结果：5。

去掉上述代码中__getattribute__()、__setattr__()的实现代码并不会影响输出结果。但是这些方法可以实现对属性的控制，根据属性名的不同做不同的处理。

注意　Python 中并不存在__setattribute__()方法。

3. __getitem__()

如果类中把某个属性定义为序列，可以使用__getitem __()输出序列属性中的各个元素。假设水果店中销售多种水果，可以通过__getitem__()方法获取水果店中的每种水果。

【例 8-12】 演示__getitem__()的使用方法。

```
class FruitShop:
    def __getitem__(self, i):                    # 获取水果店的水果
        return self.fruits[i]

if __name__ == "__main__":
    shop = FruitShop ()
    shop.fruits = ["apple","banana"]
    print shop[1]
    for item in shop:                          # 输出水果店的水果
        print item,
```

- 第 2 行代码实现了__getitem__()方法，返回序列 fruits 中的每个元素。
- 第 7 行代码给序列属性 fruits 赋值，fruits 属性被自动保存在内建的__dict__字典中。
- 第 8 行代码输出结果如下所示：

```
banana
```

- 第 9 行代码遍历水果店中的水果，输出结果：apple banana。

4. __str__()

__str__()用于表示对象代表的含义，返回 1 个字符串。实现了__str__()方法后，可以直接使用 print 语句输出对象，也可以通过函数 str()触发__str__()的执行。这样就把对象和字符串关联起来，便于某些程序的实现，可以用这个字符串来表示某个类。

更多资源请访问稀酷客(www.ckook.com)

【例8-13】把__doc__字符串的内容作为对象 fruit 描述。

```
class Fruit:
    "Fruit 类"
    def __str__(self):                    # 定义对象的字符串表示
        return self.__doc__

if __name__ == "__main__":
    fruit = Fruit ()
    print str(fruit)                      # 调用__str__
    print fruit
```

- 第2行代码为 Fruit 类定义了文档字符串。
- 第3行代码实现了__str__()，并返回 Fruit 类的文档字符串。
- 第8行代码调用内置函数 str()，即可触发__st__()方法。输出结果：Fruit 类。
- 第9行代码直接输出对象 fruit，返回__str__()方法的值。输出结果：Fruit 类。

注意　__str__()必须使用 return 语句返回。如果__str__()不返回任何值，执行 print 语句将出错。

5. __call__()

在类中实现__call__()方法，可以在对象创建时直接返回__call__()的内容。使用该方法可以模拟静态方法。

【例8-14】演示__call__()的使用方法。

```
class Fruit:
    class Growth:                         # 内部类
        def __call__(self):
            print "grow..."

    grow = Growth ()                      # 返回__call__的内容

if __name__ == '__main__':
    fruit = Fruit ()
fruit.grow ()
Fruit.grow ()
```

- 第2行代码定义了内部类 Growth。
- 第3行代码实现了__call__()方法。
- 第6行代码调用 Growth()，此时将类 Growth 作为函数返回。即为外部类 Fruit 定义方法 grow()，grow()将执行__call__()内的代码。
- 第10行代码调用 grow()，输出结果：grow ...。
- 第11行代码，也可以直接使用 Fruit 类调用 gorw()方法，Fruit 类的 grow()相当于静态方法。输出结果：grow...。

__cmp__()、__gt__()等内置方法用于对象的比较，这部分内容将在8.5节讲解。

8.3.8　方法的动态特性

Python 作为动态脚本语言，编写的程序具有很强的动态性。可以动态添加类的方法，把某个已经定义的函数添加到类中。添加新方法的语法格式如下所示：

```
class_name.method_name = function_name
```

其中，class_name 表示类名，method_name 表示新的方法名，function_name 表示1个已经存在的

更多资源请访问稀酷客(www.ckook.com)

函数，函数的内容即为新的方法的内容。

【例 8-15】演示方法的动态添加。

```
# 动态添加方法
class Fruit:
    pass

def add(self):
    print "grow ..."

if __name__ == "__main__":
    Fruit.grow = add
    fruit = Fruit ()
    fruit.grow ()
```

- 第 2 行代码，定义了 Fruit 类，Fruit 类没有定义任何方法。
- 第 5 行代码定义了函数 add( )。
- 第 9 行代码把函数 add( )添加到 Fruit 类中，方法名为 grow。
- 第 11 行代码输出方法 grow( )的内容。输出结果：grow. . . 。

利用 Python 的动态特性，还可以对已经定义的方法进行修改。修改方法的语法格式如下所示：

```
class_name.method_name = function_name
```

- 其中，class_name 表示类名，method_name 表示已经存在的方法名，function_name 表示 1 个已经存在的函数，该赋值表达式表示将函数的内容更新到方法中。

【例 8-16】演示方法的动态修改。

```
# 动态更新方法
class Fruit ():
    def grow(self):
        print "grow ..."

def update ():
    print "grow ....."

if __name__ == "__main__":
    fruit = Fruit ()
    fruit.grow ()
    fruit.grow = update
    fruit.grow ()
```

- 第 3 行代码，在类 Fruit 中定义了方法 grow( )。
- 第 6 行代码定义了函数 update( )。
- 第 11 行代码调用方法 grow( )，输出结果：grow. . . 。
- 第 12 行代码更新方法 grow( )，把 grow( )更新为函数 update( )，方法名不变。
- 第 13 行代码再次调用方法 grow( )，输出结果：grow. . . . . 。

8.4　继承

继承是面向对象的重要特性之一。继承是相对两个类而言的父子关系，子类继承了父类的所有公有实例变量和方法。继承实现了代码的重用。Python 不提倡过度的包装，所以封装性在 Python 程序中体现的比较弱，实际上继承和多态已经足够好用。

更多资源请访问稀酷客(www.ckook.com)

8.4.1　使用继承

当类设计完成以后，就可以考虑类之间的逻辑关系。类之间存在继承、组合、依赖等关系，可以采用 UML 工具表示类之间的关系。例如，有两个子类 Apple、Banana 继承自父类 Fruit，父类 Fruit 中有 1 个公有的实例变量和 1 个公有的方法。可以用类图来描述这 3 个类之间的关系。图 8-2 表示了 Fruit 类和 Apple、Banana 类之间的继承关系。

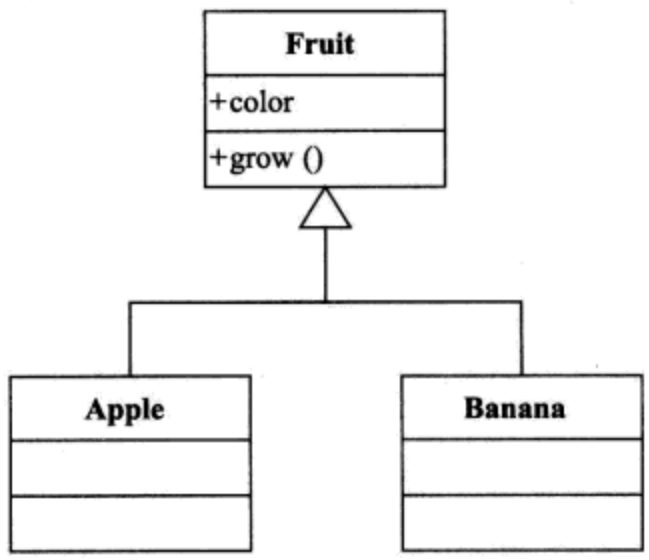


图 8-2　类的继承关系

公有实例变量和方法用“ + ”表示。Apple、Banan 类可以继承 Fruit 类的实例变量 color 和方法 grow( )。

继承可以重用已经存在的数据和行为，减少代码的重复编写。Python 在类名后使用一对括号表示继承的关系，括号中的类即为父类。如果父类定义了__init__方法，子类必须显式调用父类的__init__方法。如果子类需要扩展父类的行为，可以添加__init__方法的参数。

【例 8-17】演示继承的实现。

```
class Fruit:
    def __init__(self, color):
        self.color = color
        print "fruit's color : %s" % self.color

    def grow(self):
        print "grow ..."

class Apple(Fruit):
    def __init__(self, color):
        Fruit.__init__(self, color)
        print "apple's color : %s" % self.color

class Banana(Fruit):
    def __init__(self, color):
        Fruit.__init__(self, color)
        print "banana's color : %s" % self.color

    def grow(self):
        print "banana grow ..."

if __name__ == "__main__":
apple = Apple("red")
apple.grow ()
banana = Banana("yellow")
banana.grow ()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 1 行代码定义了父类 Fruit。
- 第 2 行代码在__init__方法中定义了公有的属性 color。
- 第 6 行代码定义了公有的方法 grow( )。
- 第 9 行代码定义了子类 Apple，Apple 继承了 Fruit。
- 第 11 行代码调用了父类 Fruit 的__init__方法进行初始化。
- 第 12 行代码使用继承父类 Fruit 的属性 color。
- 第 14 行代码定义了子类 Banana，Banana 继承了 Fruit。
- 第 19 行代码定义了和父类 Fruit 同名的方法 grow( )，该方法覆盖了 Fruit 类的 grow( )。
- 第 23 行代码创建了对象 apple。由于 Apple 的__init __方法调用了 Fruit 的__init __方法，所以先输出父类中的信息，再输出子类中的信息。输出结果如下所示：

```
fruit´s color : red
apple´s color : red
```

- 第 24 行代码调用了 apple 对象的 grow( )。由于 Apple 类继承了 Fruit 类，所以直接输出父类方法 grow( )中的信息：

```
grow ...
```

- 第 25 行代码创建了对象 banana。输出结果如下所示：

```
fruit´s color : yellow
banana´s color : yellow
```

- 第 26 行代码，由于 Banana 类的 grow( )覆盖了 Fruit 类的 grow( )，所以这里输出 Banana 类的方法 grow( )中的信息。输出结果：banana grow ...。

还可以使用 super 类的 super( )调用父类的__init__方法。super( )的声明如下所示：

```
super(type, obj)
```

- 其中，type 是某个类，obj 是 type 类的实例化对象。super( )可以绑定 type 类的父类。

【例 8-18】使用 super( )调用父类的__init__方法：

```
#使用 super () 调用父类
class Fruit(object):
    def __init__(self):
        print "parent"

class Apple(Fruit):
    def __init__(self):
        super(Apple, self).__init__ ()
        print "child"

if __name__ == "__main__":
    Apple ()
```

- 第 8 行代码在 Apple 类的__init__( )中调用 super( )，此时的 self 对象是 Apple 类的实例。
- 第 12 行代码创建 Apple 类的实例。输出结果如下所示。

```
parent
child
```

注意　super 类的实现代码继承了 object，因此 Fruit 类必须继承 object。如果不继承 object，使用 super( )将出现错误。

更多资源请访问稀酷客(www.ckook.com)

8.4.2　抽象类的模拟

使用继承之后，子类可以重用父类中的属性和方法，并且可以对继承的方法进行重写。例如，【例8-17】中的Apple、Banana都继承了Fruit类，Apple、Banana类都具有父类的grow()方法。Fruit类是对水果的抽象，不同的水果有不同的培养方法，因此生成的情况也是不同的。Fruit类的grow()是所有水果行为的抽象，并不知道如何生长。因此，grow()方法应该是1个空方法，即抽象方法。

抽象类是对一类事物的特征和行为的抽象，抽象类由抽象方法组成。Python2.5并没有提供抽象类的语法，在Java中可以把Fruit类设计为抽象类，而Python只能用1个普通类来表示Fruit。抽象类的特征是不能被实例化，可以通过Python的NotImplementedError类模拟抽象类。NotImplementedError类继承自Python运行时错误类RuntimeError。当对抽象类进行实例化时，将抛出该异常。

【例8-19】模拟抽象类的实现。

```
def abstract ():
    raise NotImplementedError("abstract")

class Fruit:                                    #抽象类
    def __init__(self):
        if self.__class__ is Fruit:             #如果实例化的类是Fruit,则抛出异常
            abstract ()
        print "Fruit"

class Apple(Fruit):
    def __init__(self):
        Fruit.__init__(self)                    #调用父类的__init__
        print "Apple"

if __name__ == "__main__":
apple = Apple ()
```

- 第1行代码定义了全局函数abstract()，该函数抛出NotImplementedError异常。
- 第6行代码对实例化对象表示的类进行判断。如果是Fruit类，则调用abstract()抛出异常，提示该类不能被实例化。即表示Fruit类是抽象类。
- 第10行代码定义了Apple类，继承了抽象类Fruit。
- 第16行代码对Apple类实例化，输出结果如下所示：

```
Fruit
Apple
```

在上述代码的最后添加如下代码，对Fruit类进行实例化：

```
fruit = Fruit ()
```

程序将抛出如下异常信息：

```
NotImplementedError: abstract
```

同样，Python也没有提供对接口的支持。接口是特殊的抽象类。接口没有数据成员，而是一组未实现的方法的集合。Python对面向对象的语法做了简化，去掉了面向对象中许多复杂的特性。例如，类的属性和方法的限制符——public、private、protected。Python提倡语法的简单、易用，这些访问权限靠程序员的自觉遵守，而不是强制使用。Python作为动态类型语言，程序中并没有“类型”之分，因此抽象类的引入似乎没有任何意义。在最新的Python3000中将会提供抽象类。

更多资源请访问稀酷客(www.ckook.com)

8.4.3　多态性

动态多态是指发出同样的消息被不同类型的对象接收时，有可能导致完全不同的行为。即，在用户不作任何干预的环境下，类的成员函数的行为能根据调用它的对象类型自动作出适应性调整，而且调整是发生在程序运行时。由于 Python 的动态类型，决定了 Python 的多态性。例如，【例 8-17】中的 Apple、Banana 类继承了 Fruit 类，因此 Apple、Banana 具有 Fruit 类的共性。Apple、Banana 的实例可以替代 Fruit 对象，同时又呈现出各自的特性。

【例 8-20】在 FruitShop 类中定义 sellFruit( )方法，该方法提供参数 fruit。SellFruit( )根据不同的水果类型返回不同的结果。代码的 sellFruit( )根据多态性的特性，作出不同的处理。

```
class Fruit:
    def __init__(self, color = None):
        self.color = color

class Apple(Fruit):
    def __init__(self, color = "red"):
        Fruit.__init__(self, color)

class Banana(Fruit):
    def __init__(self, color = "yellow"):
        Fruit.__init__(self, color)

class FruitShop:
    def sellFruit(self, fruit):
        if isinstance(fruit, Apple):                    # 判断参数 fruit 的类型。
            print "sell apple"
        if isinstance(fruit, Banana):
            print "sell banana"
        if isinstance(fruit, Fruit):
            print "sell fruit"
```

- 第 14 行代码，在 FruitShop 类中定义了 sellFruit( )，参数 fruit 接收 Apple、Banana 类的实例。
- 第 15 行到第 20 行代码，分别判断参数 fruit 的类型。如果 fruit 的类型是 Apple，则返回“sell apple”；如果 fruit 的类型是 Banana，则返回“sell banana”；如果 fruit 的类型是 Fruit，则返回“sell fruit”。

【例 8-21】接上例。分别向 sellFruit( )传递 apple、banana 对象，返回一部分相同的结果和一部分不同的结果。

```
if __name__ == "__main__":
    shop = FruitShop ()
    apple = Apple("red")
        banana = Banana("yellow")
        shop.sellFruit(apple)                    # 参数的多态性,传递 apple 对象
        shop.sellFruit(banana)                   # 参数的多态性,传递 banana 对象
```

- 第 5 行代码调用 sellFruit( )，并传递了 apple 对象。输出结果如下所示：

```
sell apple
sell fruit
```

- 第 6 行代码调用 sellFruit( )，并传递了 banana 对象。输出结果如下所示：

```
sell banana
sell fruit
```

Python 的多态性的表达方式比 Java 等强类型语言方便很多。Java 实现 sellFruit( )，需要把其中的参数 fruit 定义为 Fruit 类型。而 Python 不再需要考虑参数的类型了。

更多资源请访问稀酷客(www.ckook.com)

8.4.4　多重继承

Python 支持多重继承，即 1 个类可以继承多个父类。多重继承的语法格式如下所示：

```
class_name(parent_class1, parent_class2...)
```

其中，class_name 是类名，parent_class1 和 parent_class2 是父类名。例如，西瓜是一种经常吃的水果，但是西瓜实质上是一种蔬菜。因此，西瓜既具有水果的特性，又具有蔬菜的特性。水果和蔬菜就可以作为西瓜的父类，西瓜（Watermelon）继承了水果类（Fruit）和蔬菜类（Vegetable）。图 8-3 表示了 Watermelon 类和 Fruit、Vegetable 类之间的多重继承关系。

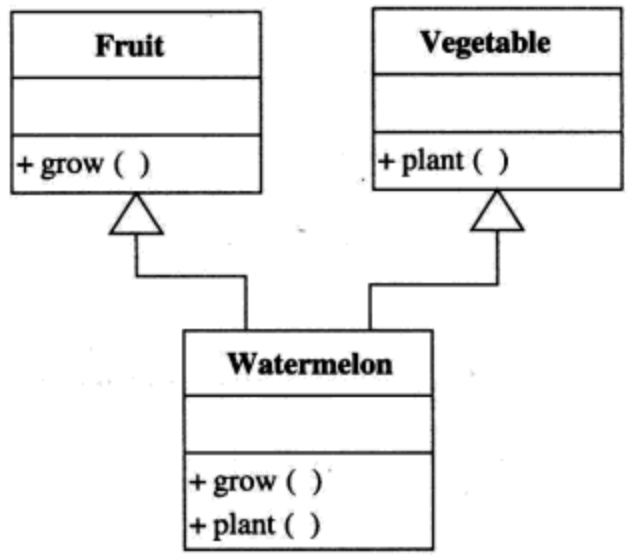


图 8-3　类的多重继承关系

可见 Watermelon 类就同时具有了 Fruit 类的 grow( ) 和 Vegetable 类的 plant( )。

【例 8-22】实现 Watermelon 类的多重继承。

```
class Fruit(object):
    def __init__(self):
        print "initialize Fruit"

    def grow(self):
        print "grow ..."

class Vegetable(object):
    def __init__(self):
        print "initialize Vegetable"

    def plant(self):
        print "plant ..."

class Watermelon(Vegetable, Fruit):                    # 多重继承
    pass

if __name__ == "__main__":
    w = Watermelon ()
    w.grow ()
    w.plant ()
```

- 第 15 行代码定义了 Watermelon 类，该类同时继承了 Vegetable 和 Fruit 类。输出结果：initialize Vegetable。
- 第 20 行代码调用 grow( )，输出结果：grow ...。
- 第 21 行代码调用 plant( )，输出结果：plant ...。

更多资源请访问稀酷客(www.ckook.com)

注意　由于 Watermelon 继承了 Vegetable、Fruit 类，因此 Watermelon 将继承__init__()。但是 Watermelon 只会调用第 1 个被继承的类的__init__，即 Vegetable 类的__init__()。

8.4.5　更好的继承方式

先看一个水果分类的例子。水果根据去果皮的方法可以分为削皮和剥皮两类。因此，可以设计削皮水果类（HuskedFruit）和剥皮水果类（DecorticatedFruit），这两个继承自 Fruit 类。苹果属于削皮水果，因此 Apple 类继承 HuskedFruit 类。而香蕉属于剥皮水果，因此 Banana 类继承 DecorticatedFruit 类。这种设计方式可以用图 8-4 表示。

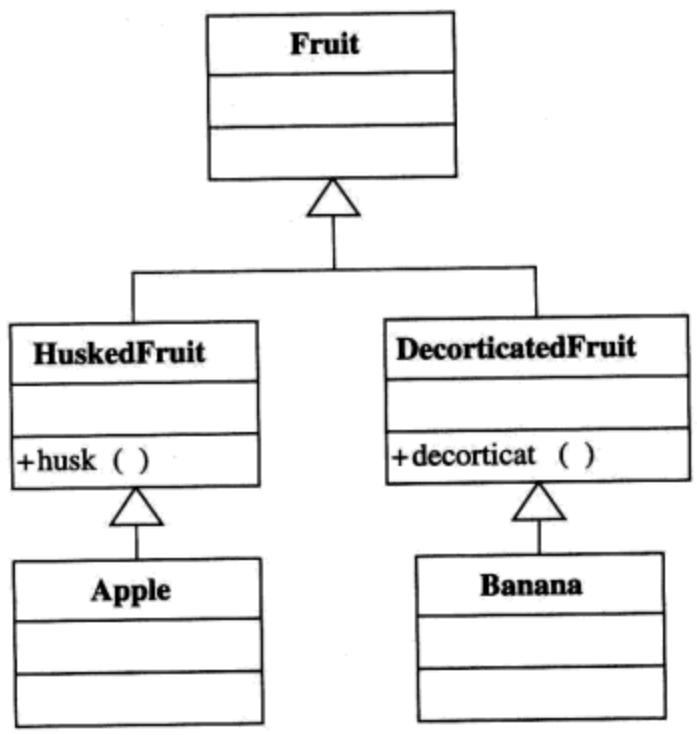


图 8-4　水果分类的设计

【例 8-23】实现图 8-4 所示的继承关系。

```
class Fruit:
    def __init__(self):
        pass

class HuskedFruit(Fruit):                          # 削皮水果
    def __init__(self):
        print "initialize HuskedFruit"

    def husk(self):                                # 削皮方法
        print "husk..."

class DecorticatedFruit(Fruit):                    # 剥皮水果
    def __init__(self):
        print "initialize DecorticatedFruit"

    def decorticat(self):                          # 剥皮方法
        print "decorticat ..."

class Apple(HuskedFruit):
    pass

class Banana(DecorticatedFruit):
    pass
```

如果按照季节对水果进行分类，整个继承结构将发生变化。例如，把水果分为夏季水果和冬

更多资源请访问稀酷客(www.ckook.com)

季水果。这时程序代码将需要做出一些修改。首先要创建新的类型——夏季水果和冬季水果的实现类。然后在继承结构中添加一层，并把它们分别作为削皮水果和剥皮水果的子类。如果对水果再进行新的分类，整个继承结构将增加许多的层次，使代码实现变得十分复杂，而且不易维护。可以采用更灵活的方式改变这个复杂的局面。图 8-5 表示了新的继承结构。

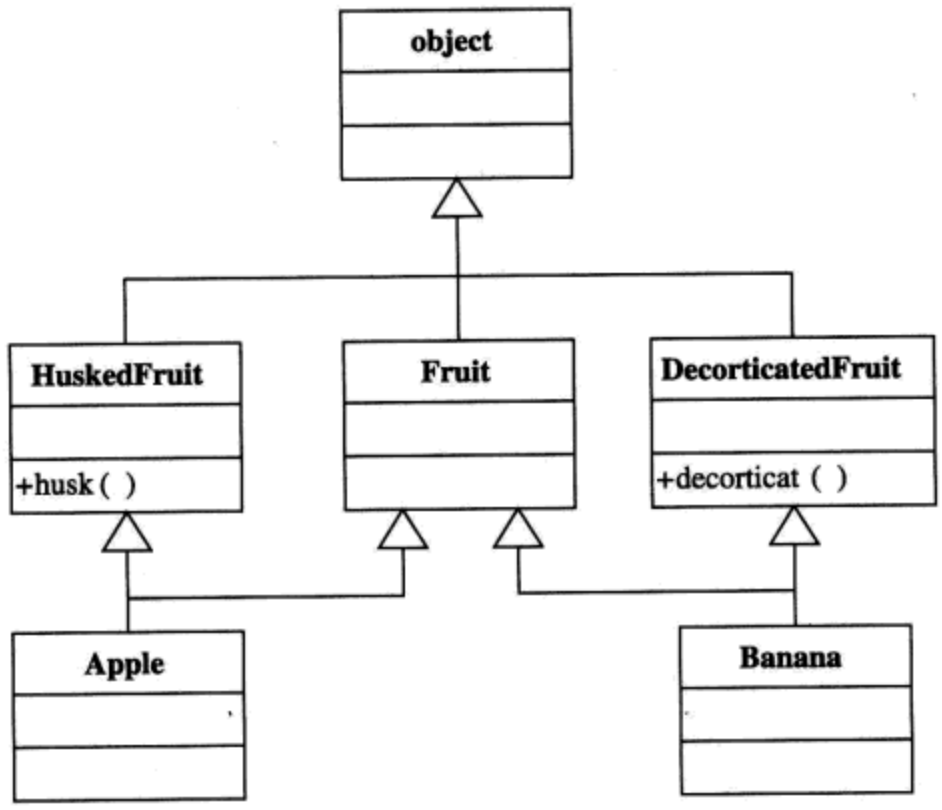


图 8-5　更好的继承结构

可见，把 Fruit 类、HuskdFruit 类、DecorticateFruit 类放在同 1 个层次，具体的水果类使用多重继承的方式继承所属的分类。

【例 8-24】实现图 8-5 所示的继承结构。

```
class Fruit(object):
    def __init__(self):
        pass

class HuskedFruit(object):                          # 削皮水果
    def __init__(self):
        print "initialize HuskedFruit"

    def husk(self):                                 # 削皮方法
        print "husk ..."

class DecorticatedFruit(object):                    # 剥皮水果
    def __init__(self):
        print "initialize DecorticatedFruit"

    def decorticat(self):                           # 剥皮方法
        print "decorticat ..."

class Apple(HuskedFruit, Fruit):
    pass

class Banana(DecorticatedFruit, Fruit):
    pass
```

这里把继承的体系结构分为 3 层：父类 object 层、水果分类层和水果层。减少了继承的层次，同时把依赖关系移到了分类层。如果要对水果进行新的分类，只要在第 2 层中添加新的类

更多资源请访问稀酷客(www.ckook.com)

型，然后在多重继承中添加新的类即可。

8.5　运算符的重载

运算符用于表达式的计算，但对自定义的对象似乎并不能进行计算。运算符的重载可以实现对象之间的运算。Python 把运算符和类的内置方法关联起来，每个运算符都对应 1 个函数。

【例 8-25】　add__()表示运算符“ + ”，__gt__()表示运算符“ > ”。实现对运算符“ + ”、“ > ”的重载。

```
class Fruit:
    def __init__(self, price = 0):
        self.price = price

    def __add__(self, other):                    # 重载加号运算符
        return self.price + other.price

    def __gt__(self, other):                     # 重载大于运算符
        if self.price > other.price:
            flag = True
        else:
            flag = False
        return flag

class Apple(Fruit):
    pass

class Banana(Fruit):
    pass
```

- 第 5 行代码对加号进行重载，使 Fruit 对象的 price 属性进行加法运算。
- 第 8 行代码对大于运算符进行重载，比较 price 属性的大小。如果 self 的 price 属性大于 other 的 price 属性，则返回“True”；否则返回“False”。

【例 8-26】接上例。对 Fruit 实例化对象使用重载后的运算符“ + ”、“ > ”。

```
if __name__ == "__main__":
    apple = Apple(3)
    print "苹果的价格:", apple.price
    banana = Banana(2)
    print "香蕉的价格:", banana.price
    print apple > banana
    total = apple + banana
    print "合计:", total
```

- 第 3 行代码输出 apple 对象的 price 属性。输出结果：苹果的价格：3。
- 第 5 行代码输出 banana 对象的 price 属性。输出结果：香蕉的价格：2。
- 第 6 行代码根据 price 属性的大小，比较 apple 对象和 banana 对象。输出结果：True。
- 第 7 行代码对 apple 对象和 banana 对象执行加法运算。
- 第 8 行代码输出结果：合计：5。

这样就实现了对象的加法运算和对象的比较。对象的加法实际上是对 price 属性相加，而对象的比较也是对 price 属性进行比较。

C++ 支持 << 、>> 等流操作符，Python 可以对流操作符进行重载，实现类似C++ 的流操作。

【例 8-27】实现对运算符 << 的重载。

更多资源请访问稀酷客(www.ckook.com)

```
import sys

class Stream:
    def __init__(self, file):
        self.file = file

    def __lshift__(self, obj):                    # 对运算符 << 进行重载
        self.file.write(str(obj))
        return self

class Fruit(Stream):
    def __init__(self, price = 0, file = None):
        Stream.__init__(self, file)
        self.price = price

class Apple(Fruit):
    pass

class Banana(Fruit):
    pass
```

- 第3行代码定义了Stream类，该类实现了对“<<”运算符的重载。
- 第4行代码Stream类的__init__()定义了属性file，接受流对象。
- 第7行代码对运算符“<<”进行重载，调用file属性的write()把对象obj的内容写入到指定设备。
- 第11行代码定义了Fruit类，并继承Stream类。使Fruit类的对象能够使用运算符“<<”。

【例8-28】接上例。创建apple、banana对象，并使用运算符<<输出price属性的值：

```
if __name__ == "__main__":
    apple = Apple(2, sys.stdout)                  # apple对象可作为流输出
    banana = Banana(3, sys.stdout)
    endl = "\n"
    apple << apple.price << endl
banana << banana.price << endl
```

- 第2行代码，把apple对象作为流输出，输出结果到控制台中。
- 第3行代码，把banana对象作为流输出，输出结果到控制台中。
- 第4行代码定义结束符。
- 第5行代码输出结果：2。
- 第6行代码输出结果：3。

8.6　Python与设计模式

8.6.1　设计模式简介

设计模式（Design Pattern）是面向对象程序设计的解决方案，是复用性程序设计的经验总结。设计模式的概念最早起源于建筑设计大师Christopher Alexander关于城市规划和建筑设计的著作《建筑的永恒方法》，尽管该著作是针对建筑领域，但是其思想也适用于软件开发领域。设计模式是对实际项目中软件设计的不断总结，在经过了一次次修改、重构后再形成的解决方案。因此合理使用设计模式可以设计出优秀的软件。设计模式的目标是形成典型问题的解决方案，设计出可复用的软件结构。设计模式是与语言无关的，任何语言都可以实现设计模式。

设计模式根据使用目的不同而分为创建型模式（Creational Pattern）、结构型模式（Structural

更多资源请访问稀酷客(www.ckook.com)

Pattern）和行为型模式（Behavioral Patterns）。

创建型模式提出了对象创建的解决方案以及数据封装的方法。降低了创建对象时代码实现的复杂度，使对象的创建能满足特定的要求。例如，工厂模式、抽象工厂模式、单例模式、生成器模式等。

结构型模式描述了对象之间的体系结构，通过组合、继承等方式改善体系结构，降低体系结构中组件的依赖性。例如，适配器模式、桥模式、组合模式、装饰器模式、外观模式等。

行为型模式描述了对象之间的交互和各自的职责，有助于实现程序中对象的通讯和流程的控制。例如，迭代器模式、解释器模式、中介者模式、观察者模式等。

Python 是一种简单、灵活的动态语言，Python 简化了面向对象语法的复杂性，但却没有降低面向对象的特性。使用 Python 同样可以实现各种设计模式，而且实现过程比较简单。

8.6.2　设计模式示例——Python 实现工厂方法

在工厂方法模式中，工厂方法用于创建产品，并隐藏了产品对象实例化的过程。工厂方法根据不同的参数生成不同的对象。因此，客户程序只需要知道工厂类和产品的父类，并不需要知道产品的创建过程以及返回的产品类型。例如，定义 Factory 类创建不同的 Fruit 对象。Apple 类和 Banana 类继承自 Fruit 类，Apple 和 Banana 可以看作是具体的产品。根据输入的字符串命令，返回 Fruit 对象。图 8-6 描述了工厂方法的类图。

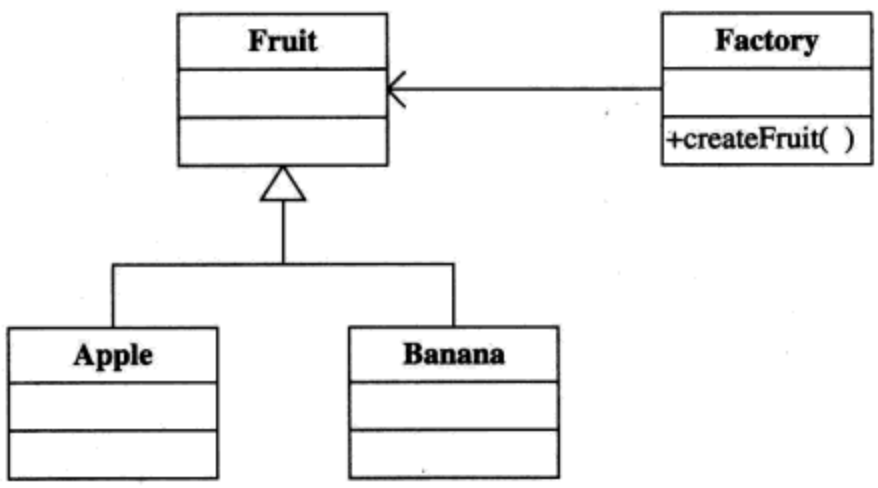


图 8-6　工厂方法的类图

【例 8-29】实现工厂方法。

```
class Factory:                                    # 工厂类
    def createFruit(self, fruit):                     # 工厂方法
        if fruit == "apple":
            return Apple ()
        elif fruit == "banana":
            return Banana ()

class Fruit:
    def __str__(self):
        return "fruit"

class Apple(Fruit):
    def __str__(self):
        return "apple"

class Banana(Fruit):
    def __str__(self):
        return "banana"
```

更多资源请访问稀酷客(www.ckook.com)

```
if __name__ == "__main__":
    factory = Factory()
    print factory.createFruit("apple")          # 创建 apple 对象
    print factory.createFruit("banana")         # 创建 banana 对象
```

- 第1行代码定义了工厂类 Factory，Factory 类生成 Fruit 对象。
- 第2行代码定义工厂方法 createFruit()，根据字符串类型 fruit 的值，生成不同的 Fruit 对象。如果 fruit 的值为“apple”，则调用 Apple()，返回 Apple 类的实例。如果 fruit 的值为“banana”，则调用 Banana()，返回 Banana 类的实例。
- 第21行代码创建了 Factory 类的实例 factory。
- 第22行代码输出结果：apple。
- 第23行代码输出结果：Banana。

工厂方法包含了前面讲解的面向对象的方法，使用了继承、封装等特性。工厂方法的缺点是在添加新产品时，需要编写新的具体产品类，而且还要修改工厂方法的实现。工厂方法模式的应用非常广泛，通常用于设计对象的生成以及系统的框架。

8.7　小结

本章介绍了 Python 面向对象的编程。首先介绍了面向对象的基本概念以及 UML 建模语言。讲解了 Python 中类的定义、属性和方法的创建以及类的内置方法等知识，比较了 Python 与 Java 等语言面向对象的实现区别。重点讲解了 Python 面向对象中的动态特性、多重继承和灵活的继承方式。Python 再次引入了多重继承机制，而且比C++ 更容易实现和运用。Python 也支持运算符的重载，并通过例子实现了加号和流运算符的重载。最后介绍了 Python 对设计模式的支持，实现了常见的工厂类。

更多资源请访问稀酷客(www.ckook.com)

第 9 章　异常处理与程序调试

本章将介绍 Python 的异常处理。Python 标准库的每个模块都使用了异常，异常在 Python 中除了可以捕获错误，还有一些其他用途。当程序中出现异常或错误时，最后的解决办法就是调试程序。Python 的 IDE 工具提供了不同功能的程序调试。PyDev（Eclipse 的 Python 插件）的程序调试功能比较强大，本章将结合具体的例子讲解程序的调试，包括断点、单步等调试方法。本章的知识点：

- try... except 语句
- raise 语句
- assert 语句
- 自定义异常
- 程序调试

9.1　异常的处理

Python 提供了强大的异常处理机制，程序员通过捕获异常可以提高程序的健壮性。异常处理还具有释放对象、中止循环的运行等作用。

9.1.1　Python 中的异常

异常（Exception）是指程序中的例外、违例情况。异常机制是指当程序出现错误后，程序的处理方法，异常机制提供了程序正常退出的安全通道。当出现错误后，程序执行的流程发生改变，程序的控制权转移到异常处理器。如序列的下标越界、打开不存在的文件、空引用异常等。当异常被引发时，如果没有代码处理该异常，异常将被 Python 接受处理。当异常发生时，Python 解释器将输出一些相关的信息并终止程序的运行。

Python2.5 版中的异常类定义在 exceptions 模块中，并继承自基类 BaseException。BaseException 类是属于 new-style class，是 Python2.5 版推出的类。BaseException 类下有 3 个子类，分别是 Exception、KeyboardInterrupt、SystemExit。在 Python2.5 版之前，KeyboardInterrupt、SystemExit 都是 Exception 下的子类。当按下 Ctrl + C 键中断程序的运行时，将引发 KeyboardInterrupt 异常；当调用 sys. exit( )时，将引发 SystemExit 异常。

Exception 类是最常用的异常类，该类包括 StandardError、StopIteration、GeneratorExit、Warning 等异常类。

StandardError 类是 Python 的错误异常，如果程序中出现逻辑上的错误，将引发该异常。例如除数为 0 的异常。StandardError 类是所有内联异常的基类，放置在默认的命名空间中，因此使用 IOError、EOFError、ImportError 等类，不需要导入 exceptions 模块。StandardError 类中常见的异常如表 9-1 所示。

StopIteration 类判断循环是否执行到尾部，如果循环到达尾部，则抛出该异常。GeneratorExit

更多资源请访问稀酷客(www.ckook.com)

类是由 Generator 函数引发的异常，当调用方法 close( )时引发该异常。Warning 类表示程序中的代码引起的警告。

表 9-1　StandardError 类中常见的异常

异 常 名	说　　明
ZeroDivisionError	除数为 0 引发的异常
AssertionError	assert 语句失败引发的异常
AttributeError	属性引用、分配错误异常
IOError	I/O 操作引发的异常，例如文件的读写
OSError	os 模块的函数引发的错误
ImportError	导入模块时引发的异常
IndexError	索引操作错误引发的异常
KeyError	使用字典中不存在的 key 值而引发的异常
MemoryError	内存耗尽而引发的异常
NameError	变量名不存在而引发的异常
NotImplementedError	方法没有实现而引发的异常
SyntaxError	语法错误引发的异常
IndentationError	代码缩进错误引发的异常
TabError	空格和制表符混合使用引发的异常
TypeError	使用不合适的类型执行运算引发的异常
ValueError	使用不合适的参数值引发的异常

Python 中的异常使用继承结构创建，这种设计方式非常灵活。可以在异常处理程序中捕获基类异常，也可以捕获各种子类异常。Python 使用 try... except 语句捕获异常，异常类型定义在 try 子句的后面。

注意　如果在 except 子句后将异常类型设置为“Exception”，异常处理程序将捕获除程序中断外的所有异常。因为 Exception 类是其他异常类的基类。

9.1.2　try... except 的使用方法

try... except 语句用于处理问题语句，捕获可能出现的异常。try 子句中的代码块放置可能出现异常的语句，except 子句中的代码块处理异常。当异常出现时，Python 会自动生成 1 个异常对象。该对象包括异常的具体信息以及异常的种类和错误位置。例如，当读取 1 个不存在的文件时，解释器将提示异常：

```
file("hello.txt", "r")
```

异常信息如下所示：

```
IOError: [Errno 2] No such file or directory: 'hello.txt'
```

当 hello. txt 文件不存在时，程序出现了例外，解释器提示 IOError 异常。为了使程序更友好，可以添加 try... except 语句捕获 IOError 异常。

【例 9-1】演示用 try... except 语句捕获 IOError 异常。

```
try:
```

更多资源请访问稀酷客(www.ckook.com)

```
    file("hello.txt", "r")
    print "读文件"
except IOError:                          # 捕获 IOError 异常
    print "文件不存在"
except:                                  # 其他异常情况
    print "程序异常"
```

- 第 1 行代码使用 try 语句，file("hello.txt"，"r")是可能出现问题的语句。
- 第 2 行代码，由于文件 hello. txt 不存在，因此 Python 引发异常，程序直接跳转到第 4 行代码。
- 第 3 行代码，此行代码将不会执行。
- 第 4 行代码，文件不存在，触发了 IOError 异常。
- 第 5 行代码输出结果：文件不存在。
- 第 6 行代码，如果 try 语句中出现了其他异常，将跳转到此处。由于引发了 IOError 异常，所以 except 下面的代码块也不会执行：

```
文件不存在
```

try... except 语句后还可以添加 1 个 else 子句。当 try 子句中的代码发生异常时，程序直接跳转到 except 子句；反之，程序将执行 else 子句。

【例 9-2】演示执行除法运算时，如果除数为“0”，将抛出 ZeroDivisionError 异常。

```
try:
    result = 10/0
except ZeroDivisionError:                # 捕获除数为 0 的异常
    print "0 不能做除数"
else:
    print result
```

- 第 2 行代码执行表达式 10/0。
- 第 2 行代码，捕获除数为 0 的异常。
- 第 3 行代码，由于除数为 0，因此输出结果：0 不能被整除。
- 第 5 行代码，else 子句中的代码块将不会被执行。如果第 2 行代码的表达式为 10/2，则异常不会发生，else 子句将被执行。

Python 与 Java 处理异常的模式相似，异常处理语句也可以嵌套：

```
try:
    s = "hello"
    try:                                 # 嵌套异常
        print s[0] + s[1]
        print s[0] - s[1]
    except TypeError:
        print "字符串不支持减法运算"
except:
    print "异常"
```

- 第 3 行代码嵌套了 1 个 try 子句。
- 第 5 行代码，字符串 s 的元素执行减法运算。由于字符串对象不支持减法运算，所以该语句将引发异常。
- 第 6 行代码，捕获类型错误异常，输出结果：字符串不支持减法运算。

注意　如果外层 try 子句中的代码块引发异常，程序将直接跳转到外层 try 对应的 excpet 子句。而内部的 try 子句的代码块将不会被执行。

更多资源请访问稀酷客(www.ckook.com)

try... except 嵌套语句通常用于释放已经创建的系统资源，下一节将进行这方面的介绍。

9.1.3　try... finally 的使用方法

try... except 语句后还可以添加 1 个 finally 子句。finally 子句的作用与 Java 中的 finally 子句类似，无论异常是否发生，finally 子句都会被执行。所有 finally 子句通常用于关闭因异常而不能释放的系统资源。9.1.2 节中【例9-1】的代码捕获了打开不存在文件的异常，但是并没有显式的关闭打开的文件资源。在这段代码中添加 finally 子句，修改后的代码如下所示：

```
# finally 错误的用法
try:
    f = open("hello.txt", "r")
    print "读文件"
except IOError:                                    # 捕获 IOError 异常
    print "文件不存在"
finally:                                           # 其他异常情况
    f.close()
```

- 第 3 行代码，在 try 子句中打开文件 hello. txt，并返回引用 f。变量 f 只在 try 语句内有效，属于局部变量。
- 第 5 行代码捕获到 IOError 异常，输出结果：文件不存在。
- 第 8 行代码，在 finally 子句中关闭打开的资源。该语句中的变量 f 并不是 try 语句中的 f，因此解释器认为变量 f 没有定义。Python 提示异常：NameError: name 'f' is not defined。

因此需要把文件的打开操作置于 try 语句的外层，使变量 f 具有全局性，同时也要捕获文件打开的异常。这种情况就可以使用异常嵌套的语句，每个 try 子句都必须有 1 个 except 子句或 finally 子句与之对应。

【例9-3】接【例9-1】。将【例9-1】的代码修改为如下所示。

```
try:
    f = open("hello.txt", "r")
    try:
        print f.read(5)
    except:
        print "读取文件错误"
    finally:                                       # finally 子句一般用于释放资源
        print "释放资源"
        f.close()
except IOError:
    print "文件不存在"
```

- 第 2 行代码在外层的 try 子句中打开文件 hello. txt。
- 第 4 行代码，读取文件 hello. txt，并放置在内层 try 子句中。
- 第 5 行代码，如果读取文件出现例外，将输出“读取文件错误”。
- 第 7 行代码释放资源。由于变量 f 定义在外层的 try 子句中，因此内层的 finally 子句可以使用变量 f。无论文件读取的异常是否发生，f. close（）语句都将被执行。

注意　由于 Python 动态语言的特殊性，因此，要在某个代码块中使用同一级其他代码块中定义的变量，可以考虑以嵌套的方式或全局变量来实现。

9.1.4　使用 raise 抛出异常

当程序中出现错误，Python 会自动引发异常，也可以通过 raise 语句显式的引发异常。一旦执行了 raise 语句，raise 语句后的代码将不能被执行。

更多资源请访问稀酷客(www.ckook.com)

【例 9-4】演示 raise 语句的使用方法。

```
try:
    s = None
    if s is None:
        print "s 是空对象"
        raise NameError
    print len(s)
except TypeError:
    print "空对象没有长度"
```

- 第 2 行代码创建了变量 s，该变量的值为空。
- 第 3 行代码，判断变量 s 的值是否为空。如果为空，则抛出异常 NameError。
- 第 6 行代码，由于引发了 NameError 异常，所以该行代码以及后面的代码将不会被执行。

如果去掉第 5 行代码，程序执行到第 6 行代码时将引发 TypeError 异常。None 对象不能使用函数 len( )调用。raise 语句通常用于抛出自定义异常。因为自定义异常并不在 Python 的控制范围内，不会被 Python 自动抛出，应使用 raise 语句手工抛出。

9.1.5　自定义异常

Python 允许程序员自定义异常类型，用于描述 Python 异常体系中没有涉及的异常情况。自定义异常必须继承 Exception 类。自定义异常按照命名规范以“Error”结尾，显式的告诉程序员该类是异常类。自定义异常使用 raise 语句引发，而且只能通过手工方式触发。

【例 9-5】演示自定义异常的使用方法。

```
from __future__ import division
class DivisionException(Exception):                    # 自定义异常
    def __init__(self, x, y):
        Exception.__init__(self, x, y)
        self.x = x
        self.y = y

if __name__ == "__main__":
    try:
        x = 3
        y = 2
        if x % y > 0:
            print x / y
            raise DivisionException(x, y)              # 抛出异常
    except DivisionException, div:
        print "DivisionException: x / y = %.2f" % (div.x / div.y)
```

- 第 3 行代码定义了自定义异常 DivisionException，该异常继承自 Exception。
- 第 4 行代码，在构造函数__init__( )中调用基类的__init__( )初始化。
- 第 6 行、第 7 行代码，定义两个属性，x 表示被除数，y 表示除数。
- 第 13 行代码，判断 x 除以 y 的余数是否大于 0。如果大于 0，则表示 x 不能整除 y，将手工抛出自定义异常 DivisionException。
- 第 15 行代码，由于引发了 DivisionException 异常，所以 except 子句也将被执行。
- 第 16 行代码，div 表示 DivisionException 类的实例对象。
- 第 17 行代码，调用 div 的属性 x 除以属性 y。输出结果：DivisionException：x / y = 1.50。

如果将第 15 行代码注释掉，DivisionException 异常将不会被触发。

更多资源请访问稀酷客(www.ckook.com)

9.1.6　assert 语句的使用方法

assert 语句用于检测某个条件表达式是否为真。assert 语句又称为断言语句，即 assert 认为检测的表达式永远为真。if 语句中的条件判断都可以使用 assert 语句检测。例如，检测某个元组中元素的个数是否大于 1。如果 assert 语句断言失败，会引发 AssertionError 异常：

```
t = ("hello",)
assert len(t) >= 1
t = ("hello")
assert len(t) == 1
```

- 第 1 行代码定义了包含元素“hello”的元组。
- 第 2 行代码，检测元组 t 的元素个数是否大于或等于 1。该行代码断言成功。
- 第 3 行代码定义了 1 个序列。由于圆括号中唯一的元素后没有逗号，因此 Python 把该序列作为字符串处理。
- 第 4 行代码，检测序列 t 的元素个数是否等于 1。该行代码断言失败，序列 t 的长度应为 5。assert 引发的异常信息如下所示：

```
Traceback (most recent call last):
  File "assert.py", line 8, in <module>
    assert len(t) == 1
AssertionError
```

assert 语句还可以传递提示信息给 AssertionError 异常。当 assert 语句断言失败时，提示信息将打印到控制台。下面这段代码检测月份的输入范围：

```
# 带 message 的 assert 语句
month = 13
assert 1 <= month <= 12, "month errors"
```

- 第 2 行代码定义变量 month，并赋值为 13。
- 第 3 行代码，当 assert 语句断言失败时，则传递字符串“month errors”给 AssertionError 异常。assert 引发的异常信息如下所示：

```
Traceback (most recent call last):
  File "assert.py", line 12, in <module>
    assert 1 <= month <= 12, "month errors"
AssertionError: month errors
```

注意　Python 支持形如“m <= x <= n”的表达式，等价于表达式“x >= m and x <= n”。这种表达方式更符合日常习惯。

9.1.7　异常信息

当程序出现错误时，Python 都会输出相关的异常信息，并指出错误的行号和错误的程序代码。

【例 9-6】执行除数为 0 的除法运算，将抛出 ZeroDivisionError 异常。

```
def fun():                          # 除法运算
    a = 10
    b = 0
    return a / b

def format():                       # 格式化输出
    print "a / b = " + str(fun())

if __name__ == "__main__":
    format()
```

更多资源请访问稀酷客(www.ckook.com)

上述代码先在主程序中调用 format( )，然后在 format( )中调用 fun( )。这段代码将输出如下所示的异常信息：

```
Traceback (most recent call last):
  File "traceback.py", line 13, in <module>
    format()
  File "traceback.py", line 10, in format
    print "a / b = " + str(fun())
  File "traceback.py", line 7, in fun
    return a / b
ZeroDivisionError: integer division or modulo by zero
```

程序执行时，Python 将产生 traceback 对象，用于记录异常信息和当前程序的状态。在上述代码中，traceback 对象先记录主程序的状态，然后记录 format( )中的状态，最后记录 fun( )中的状态。当 fun( )出现异常时，traceback 对象将输出记录的信息。因此，异常信息应从下往上阅读，最后 1 次出现的行号通常就是错误的发生处。从上述输出信息可以发现第 7 行代码出现了异常。图 9-1 说明了程序的执行顺序和异常的输出顺序。

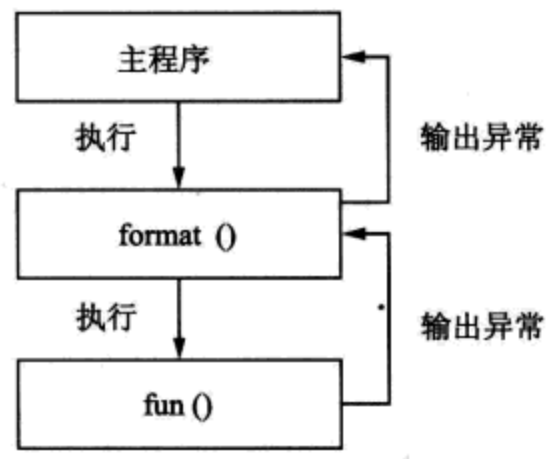


图 9-1　程序的执行顺序和异常的输出顺序

由于程序中没有使用 try... except 捕获异常，因此程序将中断执行。可参考 9.1.2 节介绍的方法捕获 ZeroDivisionError异常。异常捕获后，程序不会输出任何信息。为了便于程序员调试程序，需要输出相关的异常信息。在 Python 中可以直接输出异常实例的内容：

```
import sys

try:
    x = 10 / 0
except Exception, ex:
    print ex
print sys.exc_info()
```

- 第 4 行代码将出现异常，程序跳转到第 6 行代码。
- 第 6 行代码，直接输出 Exception 类的实例 ex：

```
integer division or modulo by zero
```

- 第 7 行代码，调用 sys 模块的 exc_info （） 输出异常的类型、traceback 对象等信息：

```
(<type 'exceptions.ZeroDivisionError'>, ZeroDivisionError('integer division or modulo by
zero',), <traceback object at 0x00BEC328>)
```

9.2　使用 PythonWin 调试程序

PythonWin 是 Python2.5 自带的 IDE 开发工具。PythonWin 具有程序调试的功能，单击【File】 | 【Debug】命令，弹出如图 9-2 所示的子菜单。子菜单中有【Go】（开始调试）、【Step in】（单步）、【Step out】（单步跳出）、【Stop】（停止调试）、【Toggle Breakpoint】（设置断点）等功能。在调试程序时可以使用子菜单右侧对应的快捷键，以提高程序调试的效率。

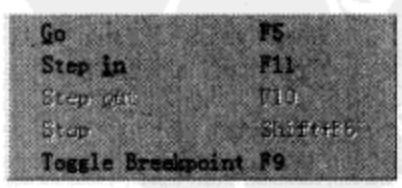


图 9-2　PythonWin 的调试菜单

更多资源请访问稀酷客(www.ckook.com)

下面以 9.1.7 节【例 9-6】的除法程序为例，讲解 PythonWin 的程序调试功能。

首先使用 PythonWin 打开该程序，然后按 F5 键开始调试程序。此时将弹出【Interactive Window】窗口，如图 9-3 所示。

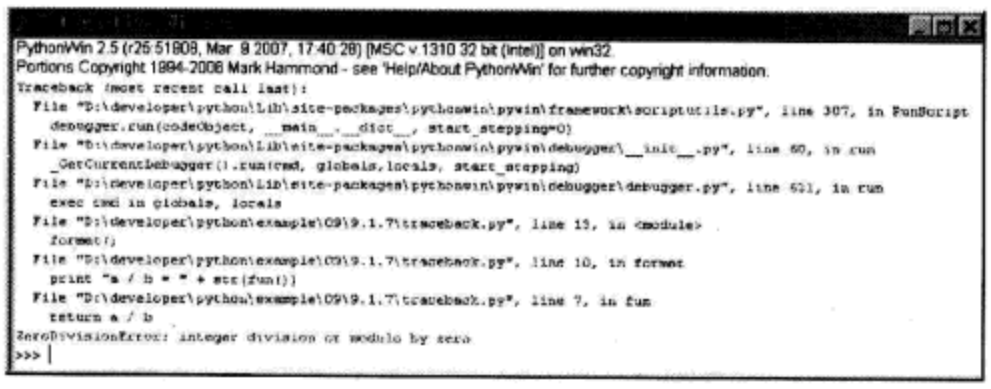


图 9-3　Interactive Window 输出的异常信息

【Interactive Window】窗口中输出了 traceback 对象记录的程序状态，Python 提示程序【例 9-6】中第 7 行代码出现异常。于是转向程序【例 9-6】，按快捷键 Ctrl + G 查找第 7 行代码。此外，还可以使用单步调试的方式找出程序的错误。首先在第 10 行代码处按 F9 键，左侧白色的圆圈即为断点，如图 9-4 所示。

然后按 F5 键启动程序的调试模式，如图 9-4 所示的窗口将发生变化，如图 9-5 所示。调试程序将从第 10 行代码开始执行。

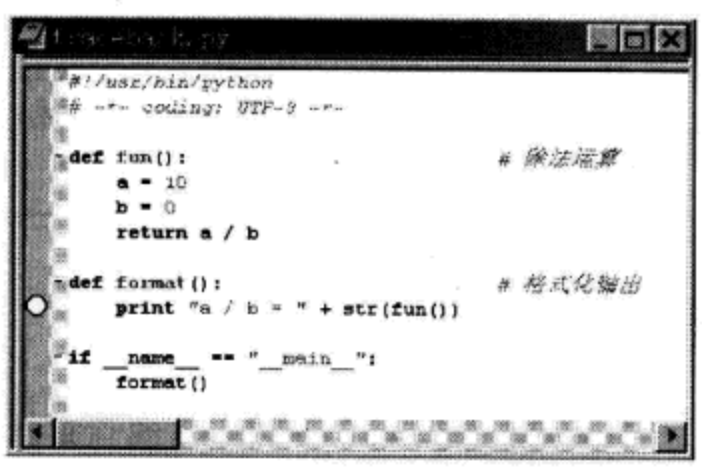


图 9-4　设置断点

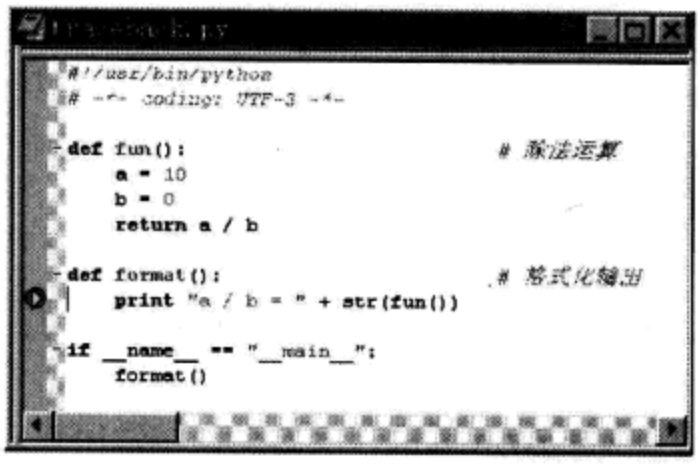


图 9-5　启动程序的调试模式

同时【Interactive Window】窗口的命令行也发生了变化，如图 9-6 所示。

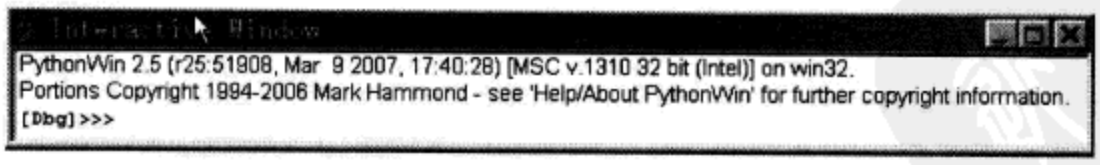


图 9-6　启动程序的调试模式

命令行“[Dbg]>>>”表示 Debug 模式。通过 Debug 命令行可以查看程序中各变量的值。按 F11 键单步调试程序。此时程序运行到第 4 行代码，即图 9-7 所示窗口中黄色箭头所指的位置。

继续按 F11 键单步调试程序。当黄色箭头移动到“return a / b”这行语句时，可以在【Interactive Window】窗口的命令行中分别输入变量“a”、“b”的名字返回其值，并查看表达式“a / b”的值，如图 9-8 所示。

更多资源请访问稀酷客(www.ckook.com)

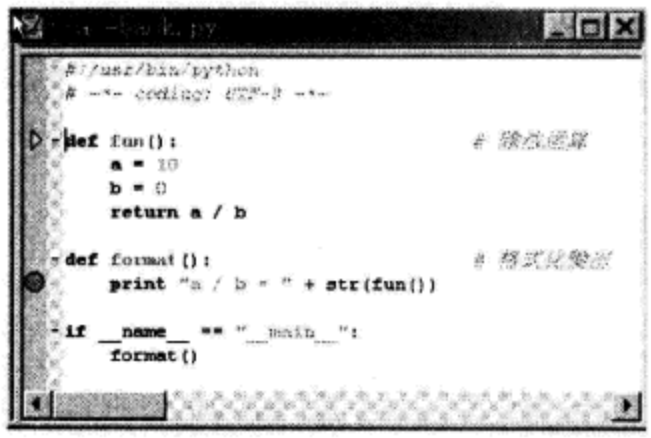


图 9-7　单步调试（Step in）

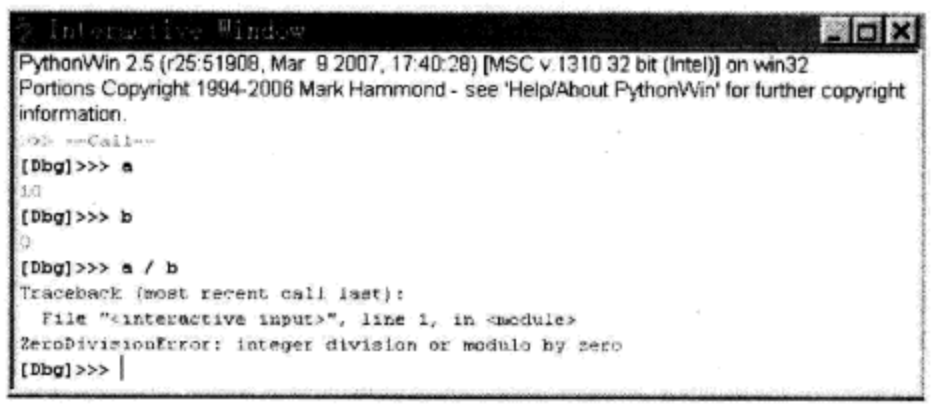


图 9-8　查看变量和表达式的值

表达式“a / b”的运算出现异常，而且变量 b 的值为 0。除数 b 的值为 0，因而出现异常。当执行单步操作运行到函数 fun( )时，也可以按 F10 键跳出程序，并返回到 fun( )的调用处。由于执行表达式“a / b”出现错误，因此【Interactive Window】窗口将输出异常信息，如图 9-9 所示。

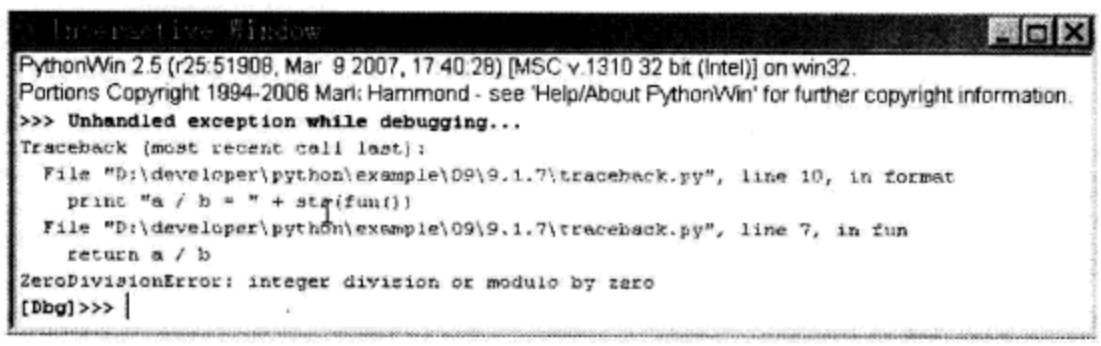


图 9-9　跳出函数（Step out）

注意　当执行 Step out 操作后，调试程序并不会忽略函数 fun( )后面的代码。而是执行完函数 fun( )后再返回。

9.3　使用 Eclipse for Python 调试程序

PythonWin 调试程序的操作与常用开发工具的使用习惯不同。在 Dbg 命令行中才能查看程序中变量的值，使用起来不够方便。Eclipse 作为“万能”开发工具支持多种语言，包括 C、Java、PHP、Python 等语言。可喜的是在 Eclipse 开发环境中可以调试 Python 程序。下面将以前面讲解的程序【例 9-6】为例，说明 Eclipse 中程序的调试功能。

9.3.1　新建工程

Eclipse 安装后第一次运行将提示工作空间的路径设置，工作空间就是存放 Eclipse 新建工程的目录，如图 9-10 所示。笔者在 Eclipse for Python 的根目录下创建了一个目录 workspace，该目录即为 Eclipse 的工程目录。

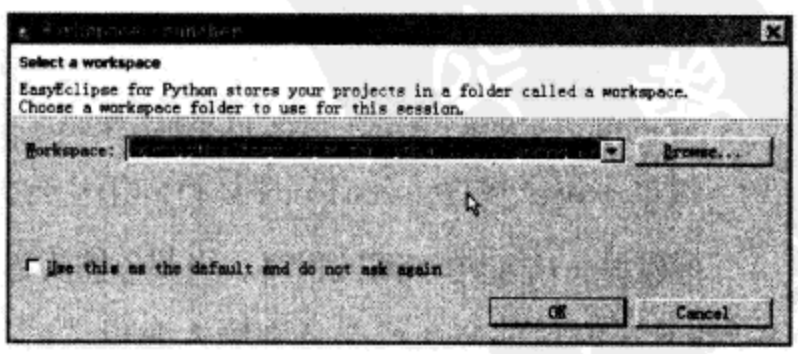


图 9-10　工作空间的路径设置

在调试程序之前需要新建一个工程。单击【File】|【New】|【Project】菜单命令，弹出【New Project】对话框，选中【Pydev】项目中的【Pydev Project】，如图 9-11 所示。然后单击【Next】按钮，显示如图 9-12 所示的对话框。

更多资源请访问稀酷客(www.ckook.com)

该对话框可以设置工程属性，包括工程名、存储位置、使用的 Python 标准库等信息。在【Project Name】文本框中输入工程名称“Debug，Project contents”设置工程的存储路径。取消对【Use default】复选框的选择后，单击【Browse】按钮可以更改工程的存储位置。在【Project type】选项组中选择【python2．5】（请读者根据自己机器中安装的 Python 库的版本进行选择）。勾选【Create default ’src’ folder and add it to the pythonpath?】复选框表示 Debug 工程创建后会生成一个 src 目录，该目录即为 Python 的源代码目录。

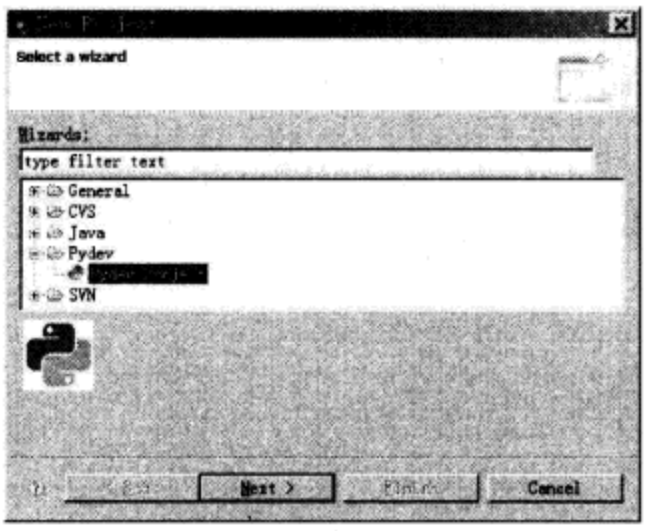


图 9-11　新建 Pydev Project（1）

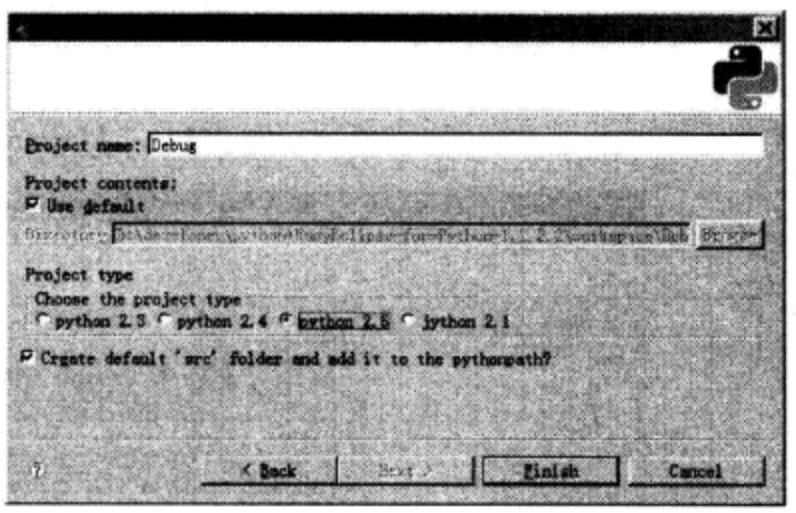


图 9-12　新建 Pydev Project（2）

最后单击【Finish】按钮，完成 Debug 工程的创建。在 src 目录下新建【Pydev Module】，将文件命名为“trace．py”。trace．py 中的代码即为程序【例 9-6】。

9.3.2　配置调试

在调试程序之前，需要设置 Python 解释器的路径，并导入 Python 环境变量下包含的库文件。单击【Window】|【Preferences】命令，弹出如图 9-13 所示的【Preferences】窗口。在该窗口中可以对 Eclipse 的开发环境和各种插件进行设置，其中的节点【Pydev】就是 Python 插件的设置项。

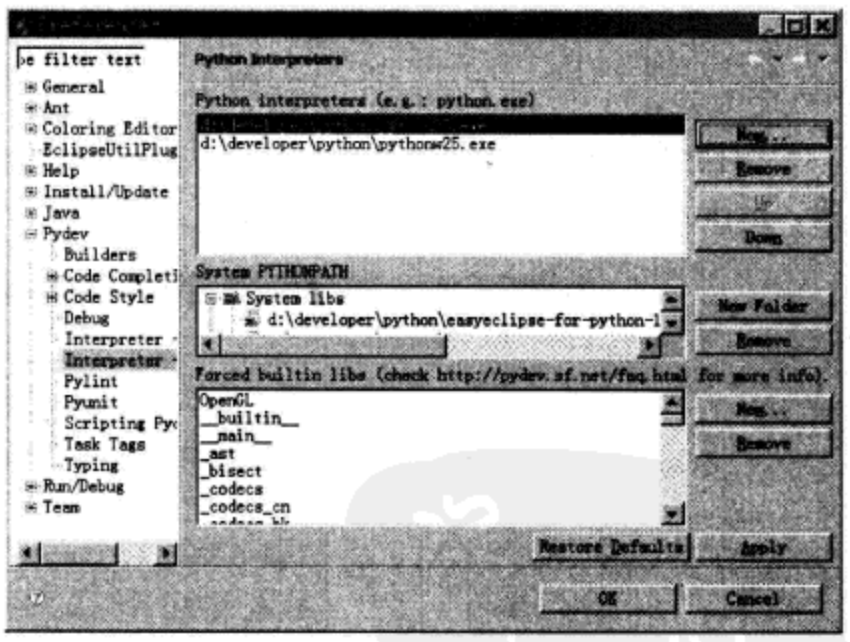


图 9-13　设置 Python 解释器

展开节点【Pydev】后，选中【Interpreter Python】子节点。然后单击【New...】按钮，加入 python25．exe、pythonw25．exe 所在的路径。最后单击【Apply】按钮，Eclipse 将自动加载 Python 环境变量下包含的库文件。

下面设置文件 trace．py 的调试参数。单击【Run】|【Debug...】命令，弹出如图 9-14所示的【Debug】对话框。

选中其中的【Python Run】选项，该选项用于调试 Python 程序。单击该对话框左上角的【新建】按钮，将会在【Python Run】下生成一个新的配置项，如图 9-15 所示。在【Name】文本框中输入调试项的名称，读者可以根据需要自行输入，这里填写为“trace”。【Main】标签页下的【Project】表示需要调试的工程，这里填写前面创建的工程“Debug”。【Main Medule】选项表示需要调试的 Python 文件。当单击【Apply】按钮后，将自动带出环境变量 PYTHONPATH 中的路径。

更多资源请访问稀酷客(www.ckook.com)

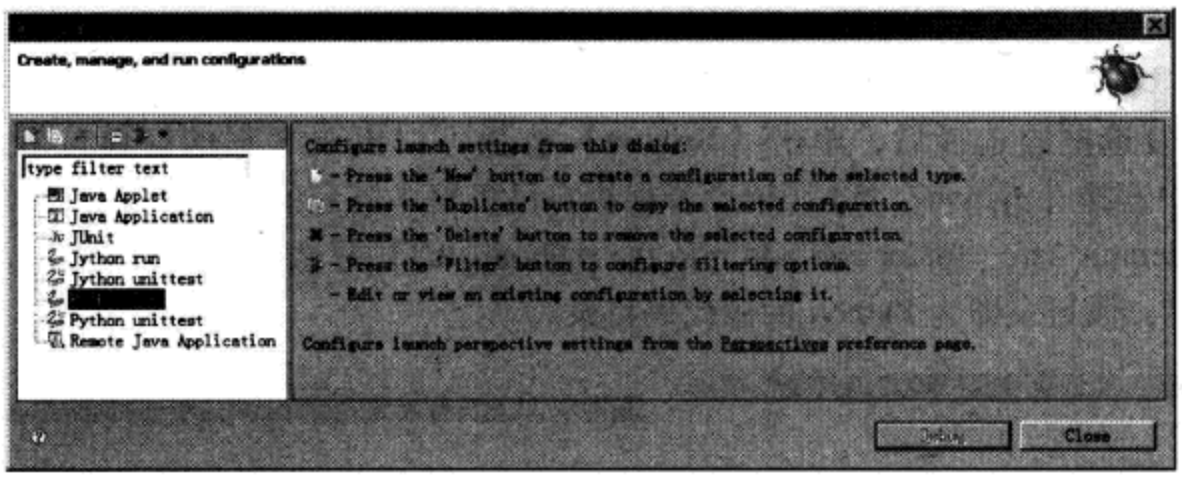


图 9-14　【Debug】对话框

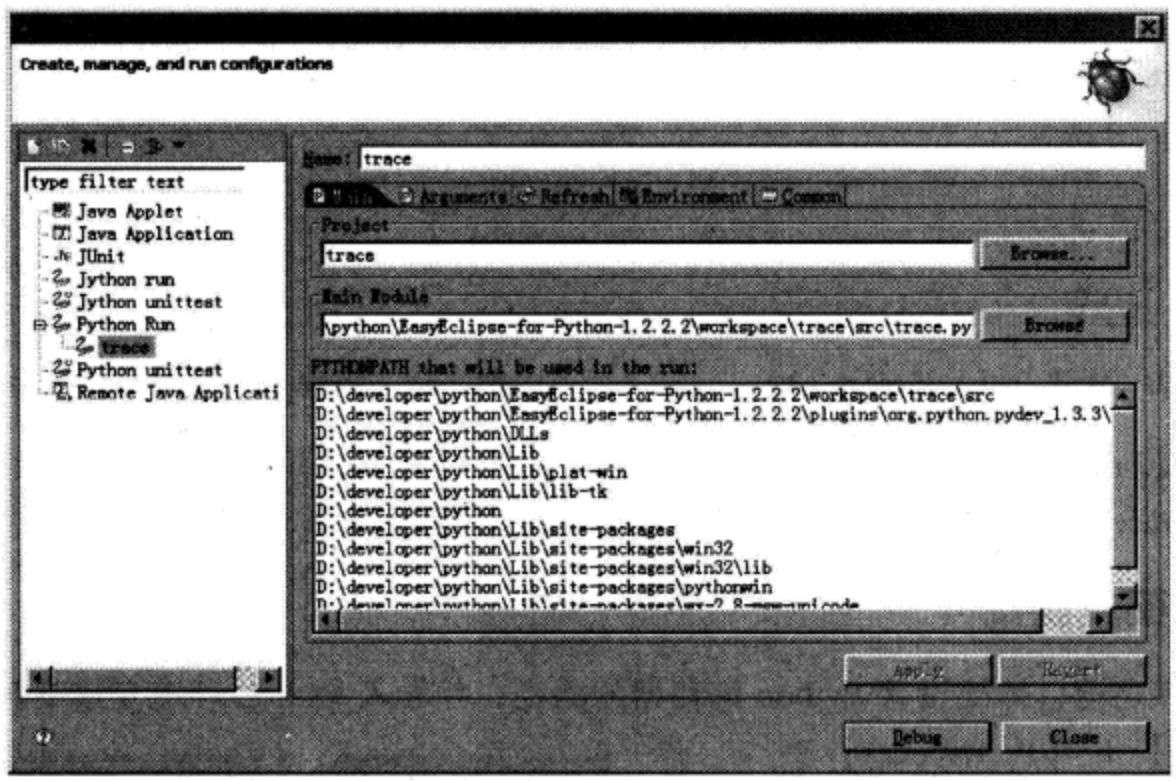


图 9-15　调试程序的配置（1）

切换到【Arguments】标签页，在【Interpreter】选项中选择“python25. exe”的路径作为解释器，pythonw25. exe 用于解释 GUI 程序，如图 9-16 所示。

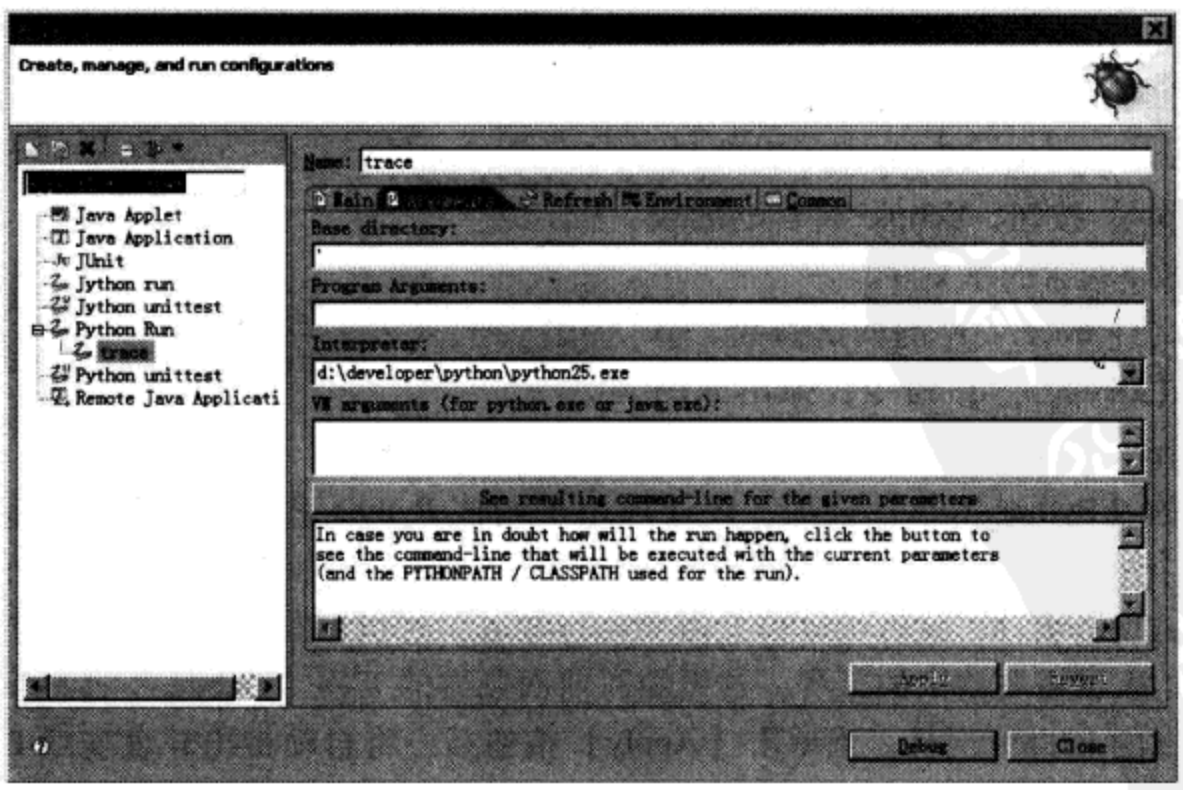


图 9-16　调试程序的配置（2）

更多资源请访问稀酷客(www.ckook.com)

最后单击【Debug】按钮就可以把 trace. py 切换到调试模式下。程序中没有设置任何断点，此时的调试还不能查看程序运行的状态，下面将在程序中添加断点。

9.3.3　设置断点

在文件 trace. py 的第 10 行代码设置断点，然后在【Eclipse】工具栏中，单击【Debug】按钮启动调试程序。trace. py 的调试模式设置成功后，将在【Debug】按钮的下拉菜单中出现【trace】子菜单，如图 9-17 所示。单击【trace】子菜单也可以启动调试程序。

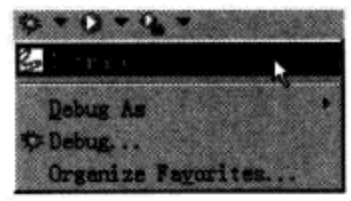


图 9-17　启动调试程序

当程序运行到断点处时，Eclipse 将自动切换到【Debug】窗口下。【Breakpoints】标签页显示了当前程序中的断点信息，如图 9-18 所示。

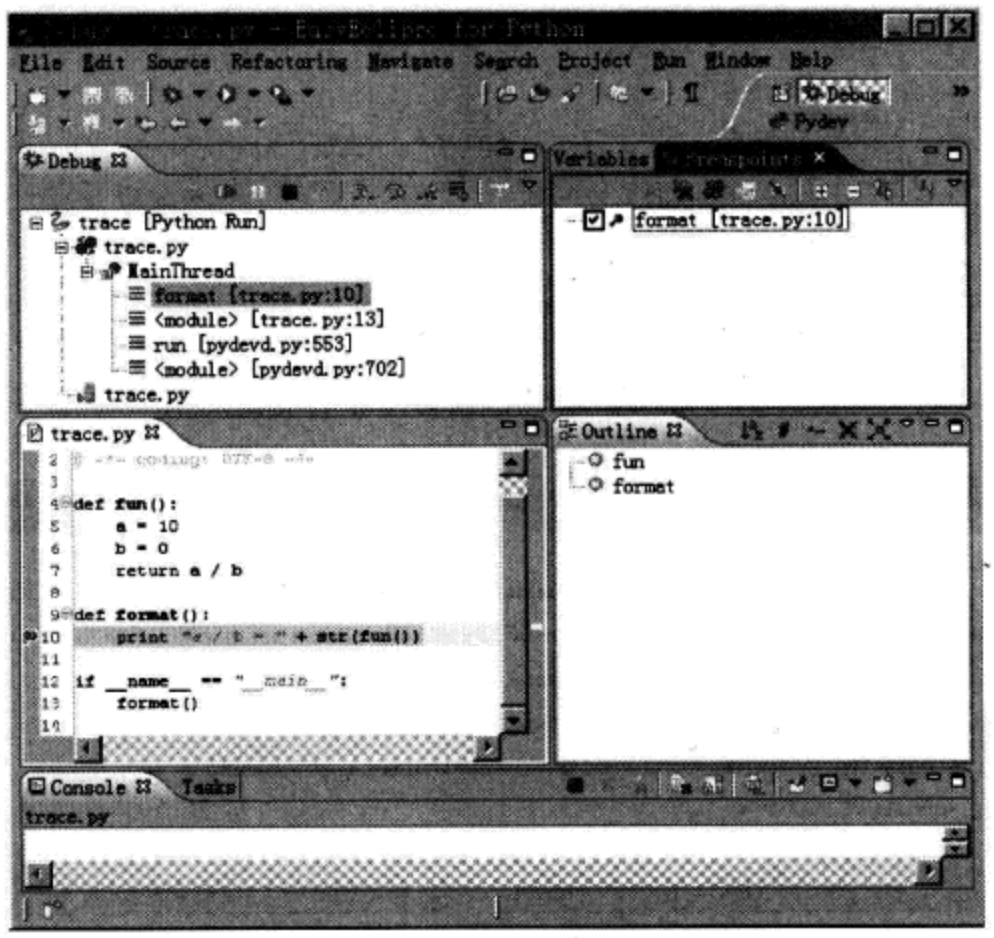


图 9-18　trace. py 的 Debug 调试窗口

其中，【Debug】标签页显示 trace. py 的主线程。【Outline】标签页显示当前程序中定义的函数，【trace. py】标签页中定义了 fun( ) 和 format( ) 这两个函数。【Console】标签页用于显示控制台的输出，例如 print 语句的输出和异常的显示。按 F5 键进行单步调试，函数 format( ) 将调用函数 fun( )。切换到【Variables】标签页可以查看程序中变量的值。当程序单步运行到第 7 行时，可以查看变量 a、b 此刻的值，如图 9-19 所示。

按 F8 键，程序将从函数 fun( ) 返回到函数 format( ) 的 print 语句，此时【Console】标签页并没有任何异常输出。继续按 F5 键进行单步调试，【Console】标签页将输出如图 9-20 所示的异常。

单击【Console】标签页最后一处的链接，Eclipse 将直接定位到异常发生的源点。把变量 b 的值改为非零值，即可修正异常。

更多资源请访问稀酷客(www.ckook.com)

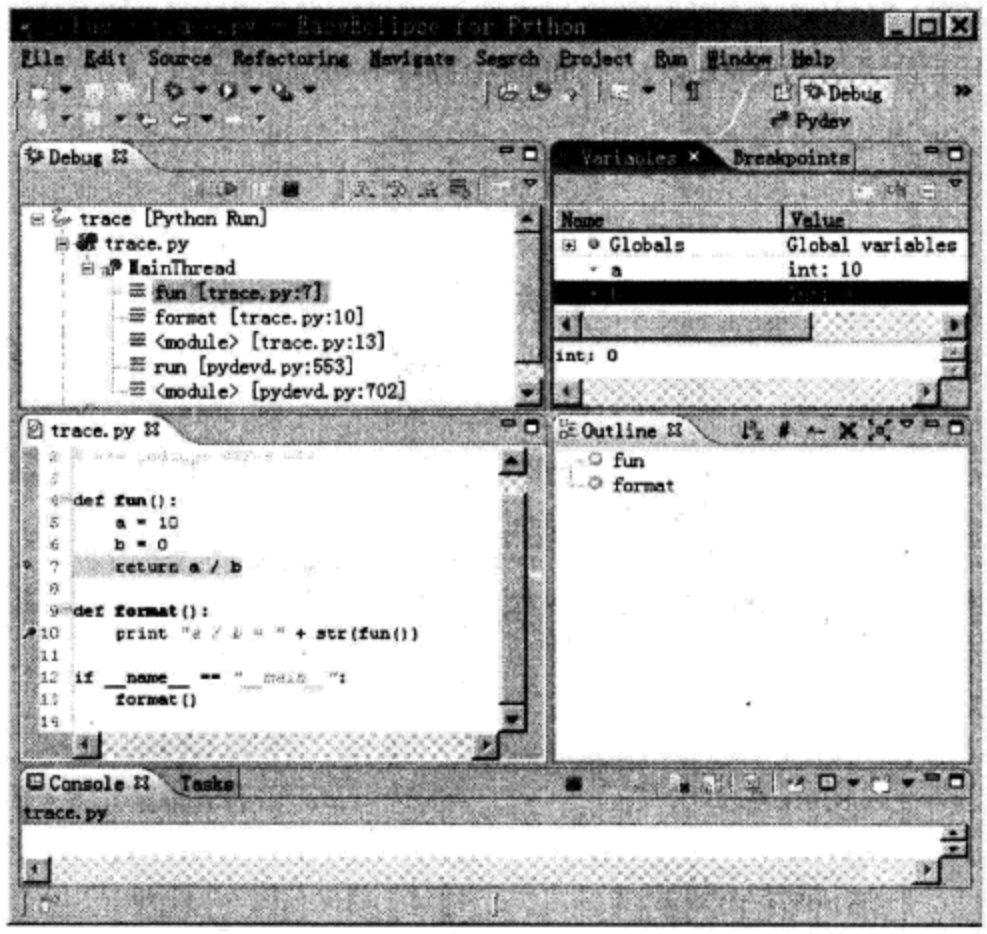


图 9-19　在 Debug 调试窗口中查看变量的值

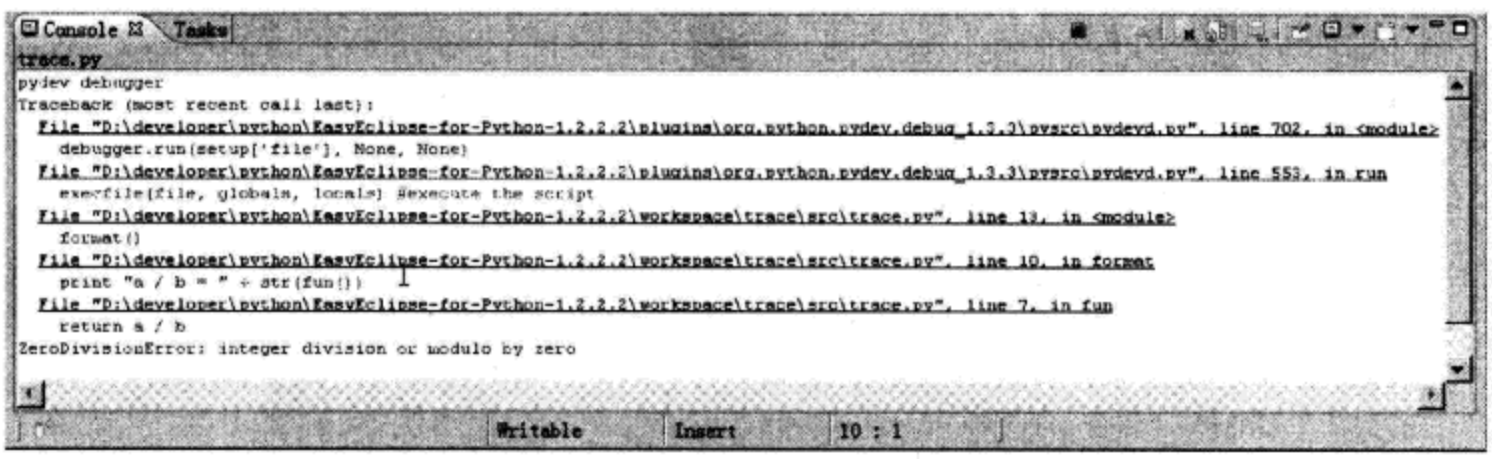


图 9-20　异常信息的输出

9.4　小结

本章介绍了 Python 中异常的处理，重点讲解了 Python2.5 异常的组织结构以及 try... except、try... finally、raise、assert 语句的使用方法。Python 中的 traceback 对象可以记录异常信息和当前程序的状态。当异常发生时，traceback 对象将输出异常信息。异常信息应从下往上阅读。利用输出的异常信息，可以快速的定位错误的行号。最后讲解了程序调试的一般步骤以及断点设置、单步、跳出等调试方法。结合 ZeroDivisionError 异常讲解了 PythonWin、Eclipse 等 IDE 工具的调试功能。当程序出现问题后，调试是解决问题的常用手段。熟练使用 IDE 工具的调试可以提高程序开发的效率。

更多资源请访问稀酷客(www.ckook.com)

第 10 章　Python 的数据库编程

本章将介绍 Python 的数据库编程。日常开发中 90% 的应用都需要使用数据库，数据库为数据提供了安全、可靠、完整的存储方式。Python 提供了多种连接数据库的手段，包括 ODBC、DAO、ADO 以及 Python 的专用模块等方式。Python 的 shelve 模块可以模拟小型数据库，支持以字典的方式访问数据库。SQLite 是一种嵌入式的数据库，通常作为 Python 的 GUI 程序的后台数据库。本章将同时介绍 SQLite 数据库的基本语法和操作方法等方面的内容。本章的知识点：

- ODBC 的连接方式
- DAO 的连接方式
- ADO 的连接方式
- 访问 Oracle 数据库的各种方法
- MySQL 数据库的访问方法
- SQLite 数据库的使用方法

10.1　Python 环境下的数据库编程

Python 提供了多种连接数据库的手段，包括 ODBC、DAO、ADO 以及 Python 的专用模块等方式。Python 提供了连接 Oracle、MySQL 等数据库的专用模块。ADO 和 Python 的专用模块是比较常用的访问方式。

10.1.1　通过 ODBC 访问数据库

ODBC（Open Database Connectivity）是微软建立的一组规范，ODBC 提供了一组对数据库访问的应用程序编程接口。通过这些接口，函数可以完成数据库的各种操作。ODBC 的应用程序不依赖于任何数据库，所有的数据库操作都是由对应的数据库 ODBC 驱动程序完成。ODBC 在大多数的数据库上都可以使用，例如文件系统、Access、SQLServer、Oracle 数据库均可用 ODBC 进行访问。程序员可以直接将 SQL 语句发送给 ODBC 处理，ODBC 统一了数据库的处理方式。下面以连接 Access 数据库为例说明 ODBC 的连接方式。

首先单击【开始】|【程序】命令，找到并打开 Access 数据库。在 Access 中创建表 address，表结构如图 10-1 所示。

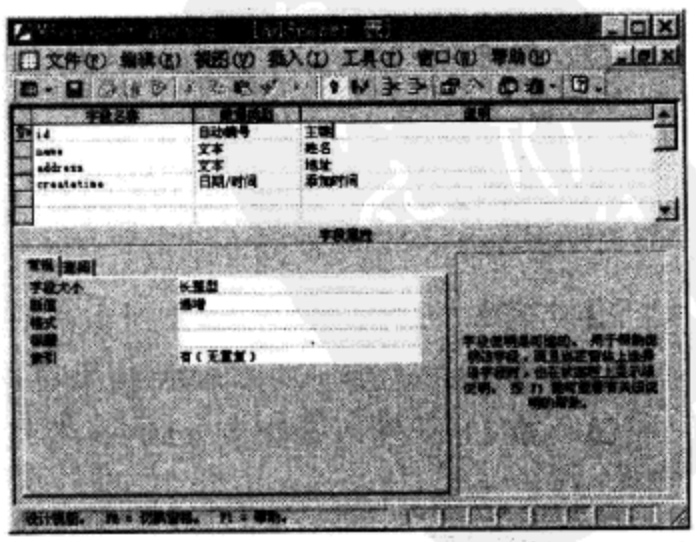


图 10-1　表 address 的结构

在访问 addresses 数据库之前，需要先创建数据源，建立 Access 和 ODBC 的连接。单击【开始】|【设置】|【控制面板】命令，打开【管理工具】|【数据源】窗口，选择【Microsoft Access Driver（*. mdb）】选项，如图 10-2 所示。

更多资源请访问稀酷客(www.ckook.com)

单击【完成】按钮，弹出如图 10-3 所示的对话框。在【数据源名】文本框中输入数据库的名称“addresses”。然后单击【选择...】按钮。弹出【选择数据库】对话框，找到 addresses.mdb 的文件路径，如图 10-4 所示。



图 10-2　创建 Access 数据源

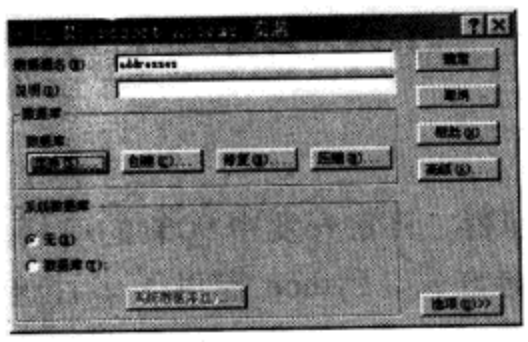


图 10-3　配置 ODBC 数据源

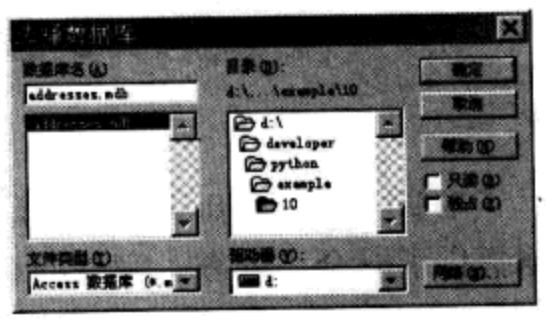


图 10-4　选择 Access 数据库

最后，单击【确定】按钮即完成了 ODBC 连接的配置。这样在程序中就可以正常调用 addresses 数据库。ODBC 连接数据主要是使用 odbc()返回的 curser 对象对数据集合进行操作。

【例 10-1】查询表 address，并返回查询的结果集。

```
import odbc, dbi                                          # 导入 ODBC 模块和驱动程序
import time

db = odbc.odbc('addresses/scott/tiger')                   # 打开数据库连接
curser = db.cursor()                                      # 产生 cursor 游标
curser.execute("select * from address order by id desc")
for col in curser.description:                            # 显示行描述
    print col[0], col[1]
result = curser.fetchall()
for row in result:                                        # 输出各字段的值
    print row
    print row[1], row[2]
    timeTuple = time.localtime(row[3])
    print time.strftime('% Y/% m/% d', timeTuple)
```

- 第 1 行代码导入 odbc、dbi 模块。使程序能找到数据源中设置的 Access 数据库。
- 第 4 行代码连接 Access 数据库 addresses.mdb。其中，addresses 表示数据库名称，scott 表示数据源的用户名，tiger 表示数据源的密码。
- 第 5 行代码返回 Access 数据库的游标对象 curser，该游标对象可以查询、添加、删除和修改数据库。
- 第 6 行代码调用 curser 对象的 execute()方法查询表 address 的内容。
- 第 7 行代码遍历 curser 对象的 description 属性。
- 第 8 行代码输出表 address 的字段名称和字段类型。输出结果如下所示：

```
id NUMBER
name STRING
address STRING
createtime DATE
```

- 第 9 行代码调用 curser 对象的 fetchall()方法，返回结果集对象 result。
- 第 10 行代码遍历 result 结果集。
- 第 11 行代码输出表 address 每行的内容。输出结果如下所示：

```
(3,'\xcd \xf5 \xce \xe5','\xb1 \xb1 \xbe \xa9 \xb6 \xab \xb3 \xc7 \xc7 \xf8', <DbiDate object at 0x00BBB0C0 >)
 (2,'\xc0 \xee \xcb \xc4','\xb1 \xb1 \xbe \xa9 \xb3 \xaf \xd1 \xf4 \xc7 \xf8', <DbiDate object at 0x00BBB0A0 >)
 (1,'\xd5 \xc5 \xc8 \xfd','\xb1 \xb1 \xbe \xa9 \xba \xa3 \xb5 \xed \xc7 \xf8', <DbiDate object at 0x00BBB0D0 >)
```

更多资源请访问稀酷客(www.ckook.com)

- 第 12 行代码输出 name、address 字段的值。输出结果如下所示：

```
王五　北京东城区
李四　北京朝阳区
张三　北京海淀区
```

- 第 13 行代码调用 localtime( )方法把字段 createtime 转换为元组。
- 第 14 行代码调用 strftime( )方法把元组格式化为字符串输出。输出结果如下所示：

```
2008/03/12
2008/02/06
2008/01/09
```

10.1.2　使用 DAO 对象访问数据库

DAO（Data Access Object）是常见的一种连接数据库的方式。DAO 具有面向对象的接口，提供了 Microsoft Jet 数据库引擎，可以直接连接到数据库，操作方式和 ODBC 相似。DAO 适用于小型应用程序或小范围本地使用。DAO 模型是设计关系数据库系统结构的对象类的集合。DAO 提供了管理一个关系型数据库系统所需操作的属性和方法，包括创建数据库、定义表、字段和索引，以及查询、添加、删除、修改等操作。

Python 的扩展模块 win32com. client 实现了 DAO 的连接方式。DAO 的连接方式不需要类似 ODBC 的配置操作，可以在 Python 程序中直接连接数据库。

【例 10-2】通过 DAO 的方式连接 Access 数据库 addresses，并演示 DAO 查询和插入等操作数据库的方法。

```
import win32com.client

engine = win32com.client.Dispatch("DAO.DBEngine.36")          # 实例化数据库引擎
db = engine.OpenDatabase("addresses.mdb")                     # 打开数据库连接
rs = db.OpenRecordset("address")                              # 根据表名返回结果集对象
rs = db.OpenRecordset("select * from address")                # 通过 SQL 语句返回数据集对象
# 插入数据
db.Execute("""
insert into address(name, address, createtime)
values('赵涛', '上海虹口', '2008 -3 -25')
""")
while not rs.EOF:                                             # 输出表 address 中的数据
    print (rs.Fields("address").Value).encode('gb2312')
    rs.MoveNext()
```

- 第 1 行代码导入 win32com. client 模块。该模块调用 win32 的接口访问数据库。
- 第 3 行代码实例化数据库引擎，返回对象 engine。
- 第 4 行代码访问数据库 addresses. mdb，返回数据库连接对象 db。
- 第 5 行代码根据表名返回结果集对象 rs。
- 第 6 行代码根据 SQL 语句返回数据集对象 rs。该 rs 对象的内容和第 5 行代码 rs 对象的内容相同（第 5 行代码和本行代码的作用相同）。
- 第 8 行代码调用 db 对象的 Execute( )方法执行插入操作。
- 第 12 行代码遍历结果集对象 rs，如果 rs 中的记录指针移到表 address 的尾部，则退出循环。
- 第 13 行代码获取字段 address 的值，由于 address 字段中包含中文，因此输出时需要进行编码转换。Python 提供了 encode( )方法进行编码转换，把字符串的字符集转换为“GB2312”即可。输出结果如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
北京海淀区
北京朝阳区
北京东城区
上海虹口
```

- 第 14 行代码调用 MoveNext（）方法，把记录指针移动到下一行。

注意　不同版本的 Access 数据对应的数据库引擎可能不同。如果数据库是 Access97，则使用 DAO. DBEngine. 35 作为引擎。

10. 1. 3　使用 ActiveX Data Object 访问数据库

ActiveX Data Objects（ADO）是微软提供的连接数据库的接口。ADO 的访问速度比 DAO 更快。ADO 可以支持多种数据库，例如 Oracle、SQL Server、Access、MySQL 等。同时 ADO 可以应用于文本文件、Word、Excel 等文件类型的数据源。ADO 的连接方式需要设置连接字符串，连接字符串是由连接数据库的参数组成的。例如，连接 oracle 数据库的连接字符串可以表示为：

```
Provider = OraOLEDB.Oracle;PLSQLRSet = 1;Password = tiger;UserID = scott
```

- 其中，参数 Provider 表示数据源名称，参数 PLSQLRSet 表示返回 PLSQL 的结果集，参数 Password 表示访问数据库的密码，参数 UserID 表示访问数据库的账户。

下面以访问 oracle 数据库示例表 emp、dept 为例说明 ADO 连接方式的使用。表 emp、dept 是 scott 账户下的表。使用 PL/SQL Developer 连接 oracle 数据库，登陆的用户名为 scott，密码为 tiger。要求从表 emp、dept 中查询薪水大于 2 500 元的员工姓名、工作、薪水、所在部门和部门管理者等信息。由于需要列出部门管理者的姓名，所以需要使用自连接，即 from 语句中需要两个表 emp。员工中只有公司老板没有管理者而且薪水大于 2 500 元，所以需要使用外连接才能显示出这条记录。SQL 语句如图 10-5 所示。

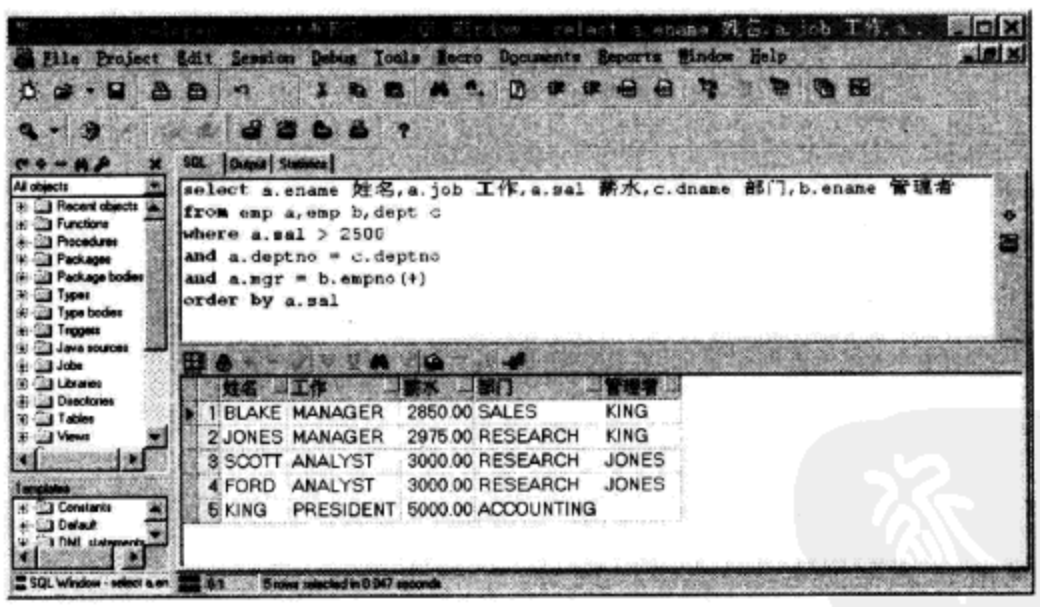


图 10-5　查询薪水大于 2 500 元的员工信息

【例 10-3】通过 ADO 的连接方式实现薪水大于 2 500 元的员工的查询。

```
import win32com.client

conn = win32com.client.Dispatch('ADODB.Connection')          # 实例化数据库连接对象
dsn = 'Provider = OraOLEDB.Oracle;PLSQLRSet = 1;Password = tiger;UserID = scott;
DataSource = ORCL'
conn.Open(dsn)                                               # 打开 oracle 数据库
rs = win32com.client.Dispatch('ADODB.Recordset')
```

更多资源请访问稀酷客(www.ckook.com)

```
sql = """select a.ename 姓名,a.job 工作,a.sal 薪水,c.dname 部门,b.ename 管理者
        from emp a,emp b,dept c
        where a.sal > 2500
        and a.deptno = c.deptno
        and a.mgr = b.empno(+)
        order by a.sal"""
rs.Open(sql, conn)                                          #返回查询的结果集
rs.MoveFirst()
li = list()
while not rs.EOF:
    d = dict()
    for x in range(rs.Fields.Count):                                #把每行数据存储在一个字典中
        key = rs.Fields.Item(x).Name
        value = rs.Fields.Item(x).Value
        d.setdefault(key, value)                                    #建立字段名和字段值的对应关系
    li.append(d)                                                    #把每个字典存储在列表中
    print rs.Fields("薪水").Value, rs.Fields("工作").Value, \
          rs.Fields("部门").Value, rs.Fields("姓名").Value, \
          rs.Fields("管理者").Value
    rs.MoveNext()
for d in li:                                                    #输出列表中的内容
    for key in d.keys():
        print key.encode("gb2312"), d.get(key),
    print
conn.Close()                                                #关闭连接
```

- 第 3 行代码实例化数据库引擎，返回连接对象 conn。
- 第 4 行代码表示 oracle 数据库的连接字符串。
- 第 6 行代码建立 oracle 数据库的连接。
- 第 7 行代码创建结果集对象 rs。
- 第 8 行代码查询工资大于 2 500 元的员工信息。
- 第 14 行代码获取查询的结果集。
- 第 15 行代码把记录指针移动到结果集的第 1 行。
- 第 19 行到第 22 行代码遍历每行数据的字段，建立字段名和字段值的对应关系，并存储到 1 个字典中。
- 第 23 行代码把每行记录对应的字典存储到列表中。
- 第 24 行代码输出结果集的内容，Fields( )方法通过字段名获取字段对象。输出结果如下所示：

```
2850 MANAGER SALES BLAKE KING
2975 MANAGER RESEARCH JONES KING
3000 ANALYST RESEARCH SCOTT JONES
3000 ANALYST RESEARCH FORD JONES
5000 PRESIDENT ACCOUNTING KING None
```

- 第 28 行到第 31 行代码遍历列表中的每个字典，输出列表中的内容。输出结果如下所示：

```
薪水 2850 工作 MANAGER 部门 SALES 姓名 BLAKE 管理者 KING
薪水 2975 工作 MANAGER 部门 RESEARCH 姓名 JONES 管理者 KING
薪水 3000 工作 ANALYST 部门 RESEARCH 姓名 SCOTT 管理者 JONES
薪水 3000 工作 ANALYST 部门 RESEARCH 姓名 FORD 管理者 JONES
薪水 5000 工作 PRESIDENT 部门 ACCOUNTING 姓名 KING 管理者 None
```

- 第 32 行代码关闭数据库连接。

注意　由于 SQL 语句中的别名使用了中文，所以需要调用 encode( )方法转换一下。

更多资源请访问稀酷客(www.ckook.com)

10.1.4　Python 连接数据库的专用模块

Python 提供了连接数据库的专用模块，不同的数据库可以使用相应的专用模块访问数据库。下面将介绍 cx_Oracle、MySQLdb 模块连接 Oracle、MySQL 数据库的使用方法。

1. cx_Oracle 模块

Oracle 是一种适用于大中型和微型计算机的关系数据库管理系统。Oracle 数据库可运行在多种操作系统上，包括 UNIX、Linux、Windwos 等。支持多种网络协议，包括 TCP/IP、DECnet、LU6.2 等。具有良好的可移植性、兼容性、开放性和高吞吐率。支持 Oracle9i、Oracle10g 等不同版本的 Oracle 数据库。

Python 的 cx_Oracle 模块可以访问 Oracle 数据库，cx_Oracle 模块的下载地址为：http://www.python.net/crew/atuining/cx_Oracle/。

【例 10-4】使用 cx_Oracle 模块连接 Oracle9.2 数据库。

```
import cx_Oracle
connection = cx_Oracle.connect("scott", "tiger", "ORCL")          # 连接 oracle 数据库
cursor = connection.cursor()                                      # 获取 cursor 对象操作数据库
sql = """select a.ename 姓名,a.job 工作,a.sal 薪水,c.dname 部门,b.ename 管理者
        from emp a,emp b,dept c
        where a.sal > 2500
        and a.deptno = c.deptno
        and a.mgr = b.empno(+)
        order by a.sal"""
cursor.execute(sql)
for x in cursor.fetchall():
    for value in x:
        print value,
print
cursor.close()
connection.close()
```

- 第 1 行代码导入 oracle 的专用模块 cx_Oracle。
- 第 3 行代码连接 oracle 数据库。
- 第 4 行代码创建游标对象 cursor，该对象可执行 SQL 语句。
- 第 11 行代码执行 SQL 语句，查询工资大于 2 500 元的员工信息。
- 第 12 行到第 15 行代码返回查询的结果集，并输出结果集的内容。输出结果如下所示：

```
BLAKE MANAGER 2850.0 SALES KING
JONES MANAGER 2975.0 RESEARCH KING
SCOTT ANALYST 3000.0 RESEARCH JONES
FORD ANALYST 3000.0 RESEARCH JONES
KING PRESIDENT 5000.0 ACCOUNTING None
```

- 第 16 行代码关闭游标对象。
- 第 17 行代码关闭数据库连接。

2. MySQLdb 模块

MySQL 是一个小型关系型数据库管理系统。MySQL 是一个开源软件，体积小、速度快，广泛地应用在 Internet 上的中小型网站中。MySQL 支持 UNIX、Linux、Mac OS、Windows 等多种操作系统。可支持多种编程语言进行开发。例如，C、C++、Java、Perl、PHP、Python、Ruby 等。提供 TCP/IP、ODBC、JDBC 等多种数据库连接方式。支持 UTF-8、GB2312、Unicode 等多种编码集。

更多资源请访问稀酷客(www.ckook.com)

用 MySQLdb 模块操作 MySQL 数据库非常方便。本书使用的是 MySQL5.0，读者可以到 http://download.mysql.cn/下载所需版本的 MySQL 数据库。在使用 MySQLdb 模块访问数据库之前，先创建 1 个数据库。phpMyAdmin 是 1 个开源的 MySQL 数据库管理软件。读者也可以到 http://download.mysql.cn/下载最新的 phpMyAdmin。下面在 phpMyAdmin 中创建 1 个数据库，如图 10-6 所示。

输入数据库名称，并选择字符集 utf8_unicode_ci，再单击【创建】按钮，数据库 addressbookdb 创建成功。然后在 phpMyAdmin 中创建地址表 address，如图 10-7 所示。

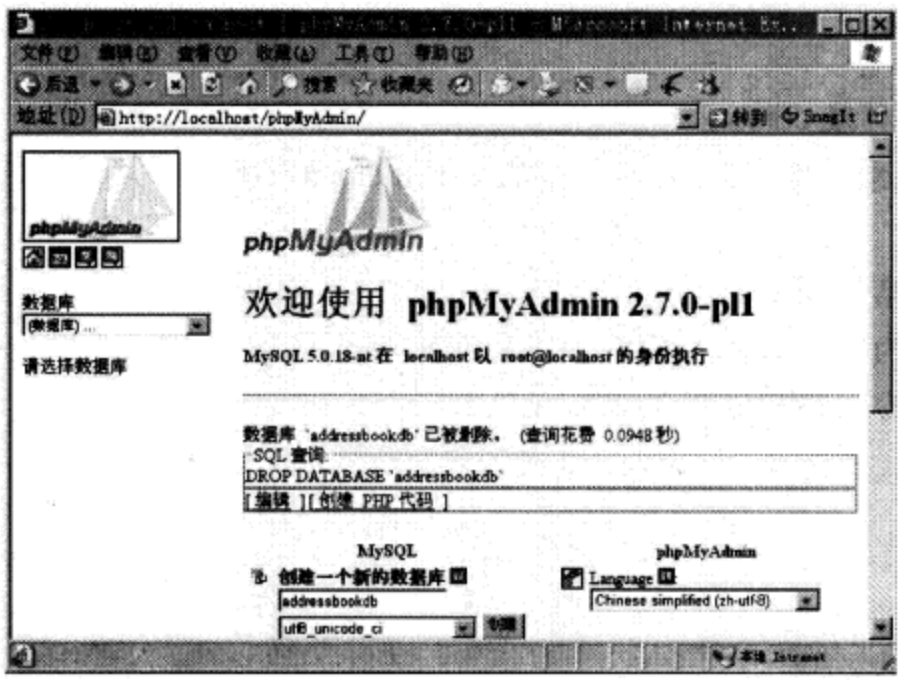


图 10-6　创建数据库 addressbookdb

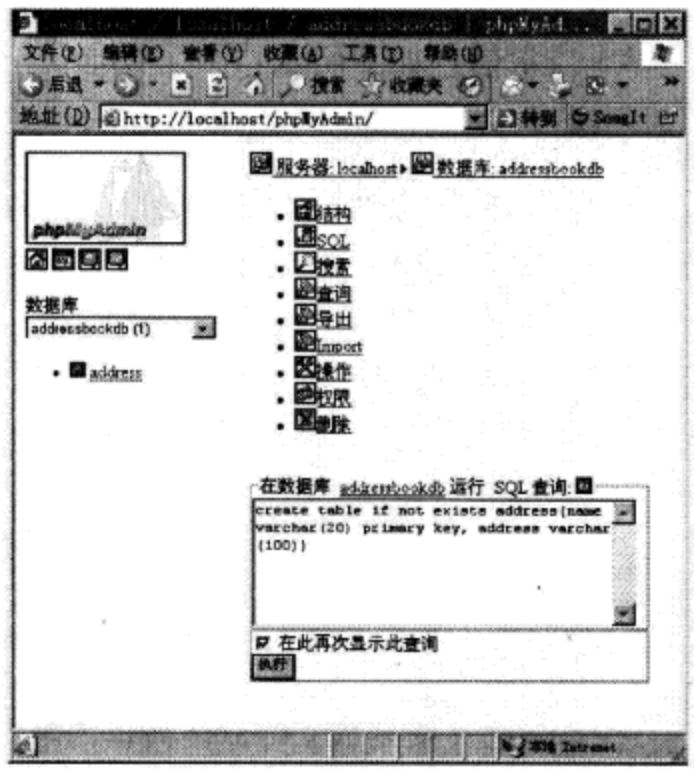


图 10-7　创建表 address

注意　如果 MySQL 数据库中包含中文字符，需要在创建数据库时选择 utf8_ unicode_ ci 字符集，否则 Python 程序执行插入操作时将出现异常。异常信息如下所示：

```
(1406, "Data too long for column 'name' at row 1")
```

【例 10-5】实现 Python 对 MySQL 数据库的插入和查询操作。

```
import os, sys
import MySQLdb
#连接数据库
try:
    conn
MySQLdb.connect(host='localhost',user='root',passwd='',db='ADDRESSBOOKDB')
except Exception, e:
    print e
    sys.exit()
cursor = conn.cursor()
sql = "insert into address(name, address) values (%s, %s)"
values = (("张三", "北京海淀区"), ("李四", "北京海淀区"), ("王五", "北京海淀区"))
try:
    cursor.executemany(sql, values)                    #插入多条数据
except Exception, e:
    print e
sql = "select * from address"
cursor.execute(sql)                                    #查询数据
data = cursor.fetchall()
if data:
    for x in data:
```

更多资源请访问稀酷客(www.ckook.com)

```
        print x[0], x[1]
cursor.close()                          #关闭游标
conn.close()                            #关闭数据库
```

- 第 2 行代码导入 MySQL 的专用模块 MySQLdb。
- 第 5 行代码调用 connect( )方法连接 addressesdb 数据库，用户名为 root，密码为空，主机地址为 localhost。返回数据库连接对象 conn。
- 第 6 行代码，如果数据库不存在，则抛出异常，并调用 sys. exit( )中断程序的执行。异常信息如下所示：

```
(1049, "Unknown database 'addressbookdb'")
```

- 第 9 行代码调用 cursor( )返回游标对象 cursor。
- 第 10 行到第 15 行代码插入多条记录。游标对象 cursor 的 executemany( )方法可以一次性插入多条记录。
- 第 18 行代码查询表 address，返回结果集 data。
- 第 19 行代码判断结果集 data 是否为空。如果不为空，表示查询有返回记录。
- 第 21 行代码输出结果集 data 的内容。输出结果如下所示：

```
李四 北京海淀区
王五 北京海淀区
张三 北京海淀区
```

注意　如果存储到 MySQL 数据库的字符串中含有特殊字符（如单引号），可以使用 MySQLdb 模块的函数 escape_string( )进行转义，然后再存储到数据库中。

10.2　Python 的持久化模块

Python 的标准库提供了几种持久化模块。这些模块可以模拟数据库的操作，会把数据保存在指定的文件中。例如 dbhash、anydbm、shelve 等模块。

10.2.1　用 dbhash 模块读写数据

DBM 是一种文件式数据库，DBM 采用哈希结构进行存储。DBM 是一种简单的数据库，并不具备管理能力，但是比普通文件更稳定、可靠，而且查询速度快。不同的操作系统需要使用不同的 Python 模块实现 DBM 数据库。对于 Windows 操作系统可以使用 dbhash 模块，而 UNIX 操作系统可以使用 gdbm 模块。dbhash 模块主要的方法就是 open（），该方法的声明如下所示：

```
open( filename[, flag[, mode]])
```

其中，filename 表示数据库的名称，flag 表示数据库的打开方式。“w”表示读或写数据库，“r”表示以只读的方式打开数据库，“c”表示创建数据库。默认方式为“r”。mode 用于 UNIX 系统，表示数据库文件的授权方式。

【例 10-6】实现 Windows 环境下 DBM 数据库的读写操作。

```
import dbhash

db = dbhash.open('temp', 'c')                    #创建并打开数据库
db['Tom'] = 'Beijing road'                       #写数据
db['Jerry'] = 'Shanghai road'
for k, v in db.iteritems():                      #遍历 db 对象
```

更多资源请访问稀酷客(www.ckook.com)

```
    print k, v
if db.has_key('Tom'):
    del db['Tom']
print db
db.close()                                    # 关闭数据库
```

- 第 3 行代码打开 temp 数据库，参数“c”表示如果 temp 数据库不存在，则创建该数据库。当前目录下会创建 1 个名为 temp 的文件。open( )方法返回 1 个字典对象 db，该对象具有字典的所有属性和方法。
- 第 4 行、第 5 行代码添加两条记录，db 对象的操作方式和字典相同。
- 第 6 行代码调用 iteritems( )方法，遍历 db 中的 key、value 值。输出结果如下所示：

```
Jerry Shanghai road
Tom Beijing road
```

- 第 8 行代码判断 db 对象中是否存在关键字“Tom”。如果存在则删除对应的数据。
- 第 10 行代码输出对象 db 的内容：

```
{db['Jerry'] = 'Shanghai road'}
```

- 第 11 行代码关闭数据库连接。

dbm 模块的文件系统只支持字符串类型的 value 值。如果给对象 db 赋予数值，程序将抛出异常。例如，在上述程序中添加如下代码：

```
db['Tod'] = 2
```

Python 将提示如下异常信息：

```
TypeError: Data values must be of type string or None.
```

为了统一不同操作系统对 DBM 数据库的要求，Python 的 anydbm 模块提供了操作 DBM 数据库的一般性操作。anydbm 模块的用法和 dbhash 模块类似。

注意　dbhash 模块返回字典的 key、value 值只支持字符串类型。

10.2.2　shelve 模块读写数据

shelve 模块是 Python 的持久化对象模块，该模块与 anydbm 模块的用法相似。但是 shelve 模块返回字典的 value 值支持基本的 Python 类型。例如字符串、数字，也支持元组、列表和字典等特殊类型。shevle 模块的 open( )与 dbhash 模块的 open( )使用方法相似。

【例 10-7】演示 shelve 模块的使用方法。

```
import shelve

addresses = shelve.open('addresses')
addresses['1'] = ['Tom', 'Beijing road', '2008-01-03']
addresses['2'] = ['Jerry', 'Shanghai road', '2008-03-30']
if addresses.has_key('2'):
   del addresses['2']
print addresses
addresses.close()
```

- 第 3 行代码打开 addresses 数据库，如果 addresses 数据库不存在，则创建该数据库。当前目录下会创建 1 个名为 addresses 的文件。open( )方法返回 1 个 shelve 对象 addresses。
- 第 4 行、第 5 行代码添加两条记录，addresses 对象的操作方式和字典相同。

更多资源请访问稀酷客(www.ckook.com)

- 第 6 行代码判断 addresses 对象中是否存在关键字“2”。如果存在，则删除对应的数据。
- 第 8 行代码输出 addresses 对象的内容，输出结果如下所示：

```
{'1': ['Tom', 'Beijing road', '2008 - 01 - 03']}
```

- 第 9 行代码关闭数据库连接。

注意　shelve 模块返回字典的 key 值只支持字符串类型。

10.3　嵌入式数据库 SQLite

SQLite 是一个开源的嵌入式数据库引擎，体积小巧，可应用于 Windows、Linux 等多种操作系统；可以被安装在个人电脑、掌上设备、播放器等不同设备中。SQLite 实现了可配置、事务管理、数据文件的跨平台等特性，已经成为应用最多的一种数据库系统。

10.3.1　sqlite 的命令行工具

SQLite 是非常著名的开源嵌入式数据库。sqlite 可以嵌入到应用程序，并且提供了 SQL 接口访问数据。读者可以从 sqlite 官方网址 http://www.sqlite.org 下载 SQLite 数据库。SQLite 数据库提供了 DOS 命令行工具，该工具也可以在 http://www.sqlite.org 下载。运行命令行工具，在命令行中创建表 test，如图 10-8 所示。



图 10-8　创建表 test

由于之前没有创建数据库，因此创建表 test 后，SQLite 会自动创建 1 个临时文件存储。可以使用 .database 命令查看该文件的路径。如果是 Windows 操作系统，一般在 C:\Documents and Settings\abc\Local Settings\Temp 路径下。表 test 创建成功后可以使用 .schema 命令查看表 test 的结构，如图 10-9 所示。



图 10-9　查看表结构

下面向表 test 中输入两条记录，然后查询表 test 的内容，如图 10-10 所示。



图 10-10　插入数据并查询结果

注意　SQLite 命令行的每行语句以分号结束，对于复杂的 SQL 语句可以换行书写。

更多资源请访问稀酷客(www.ckook.com)

图 10-10 所示记录的显示方式与常见的方式差异很大。一般每条记录会显示在一行中，并且每列数据对齐显示。SQLite3 提供了显示模式的选择，可以使用. mode 命令设置数据的显示模式，如图 10-11 所示。



图 10-11　设置数据的显示模式

注意　表 test 存储在内存中，如果关闭当前的命令行窗口，表 test 所占用的资源也会被释放。再次打开命令行窗口查询表 test，将提示如下所示错误信息：

```
SQL error:no such table:test
```

SQLite 的命令行还提供了许多实用的操作，读者可以通过【.help】命令查看这些具体的命令。命令行工具的功能有限，而且操作不够方便。例如，当某个表的数据量很大时，查看结果集就比较困难。SQLite 有许多第三方的数据库客户端工具，例如 SQLitespy、SQLiteManager。

下面将使用 SQLiteManager 创建存储个人地址簿的数据库，SQLiteManager 的下载地址：http://www.sqlabs.net/sqlitemanager.php。SQLiteManager 可以创建、删除数据库、表、视图等数据库对象，支持查询操作、查询分析，以及修改表结构等功能。

运行 SQLiteManager 后，可以选择打开或创建数据库，如图 10-12 所示。单击【Continue】按钮后，弹出【保存】对话框，指定数据库存储的路径，如图 10-13 所示。

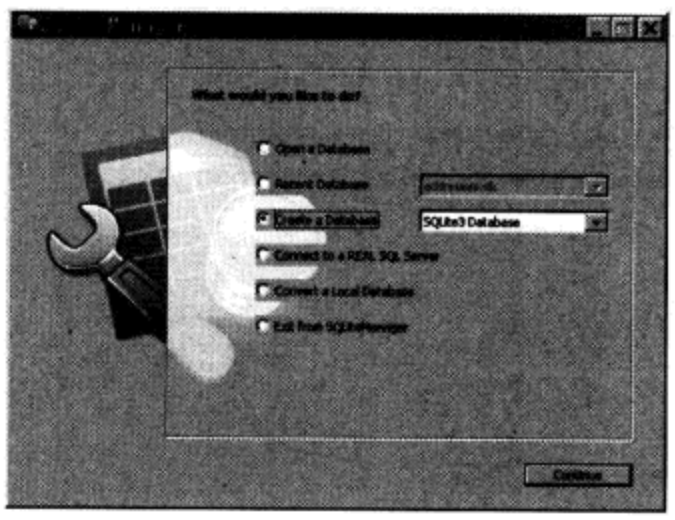


图 10-12　创建数据库



图 10-13　指定数据库的名称和存储路径

完成数据库的创建后，进入 SQLiteManager 的主界面。然后创建表 address，创建后的表结构如图 10-14 所示。单击工具栏中的 SQL 图标，查询表 address 的内容，如图 10-15 所示。

10.3.2　使用 sqlite3 模块访问 sqlite 数据库

python2.5 已经自带了 sqlite3 模块。如果要使用 SQLite 数据库，需要导入 sqlite3 模块。连接 SQLite 数据库大致可以分为以下 6 个步骤。

（1）导入 sqlite3 模块。

（2）调用 connect()创建数据库连接，返回连接对象 conn。

（3）调用 conn.execute()方法创建表结构并插入数据。如果设置了手动提交，需要调用 conn.commit()提交插入或修改的数据。

更多资源请访问稀酷客(www.ckook.com)

（4）调用 conn. cursor( )方法返回游标，通过 cur. execute( )方法查询数据库。

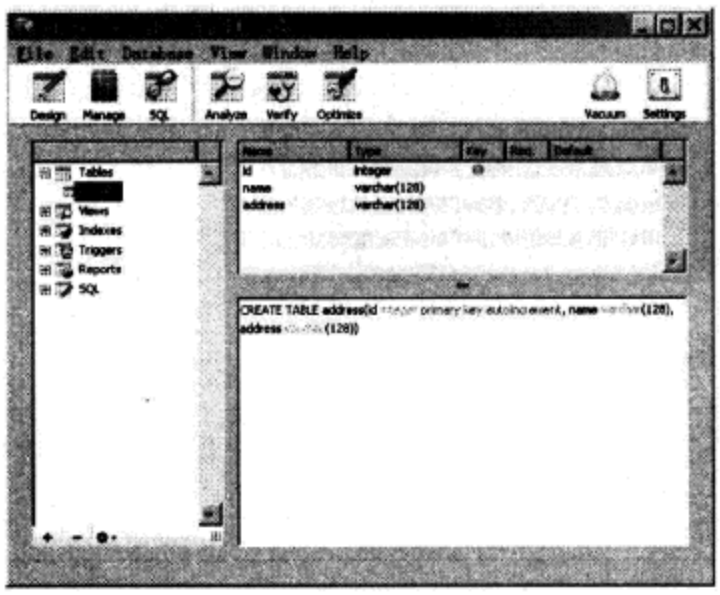


图 10-14　创建表 address

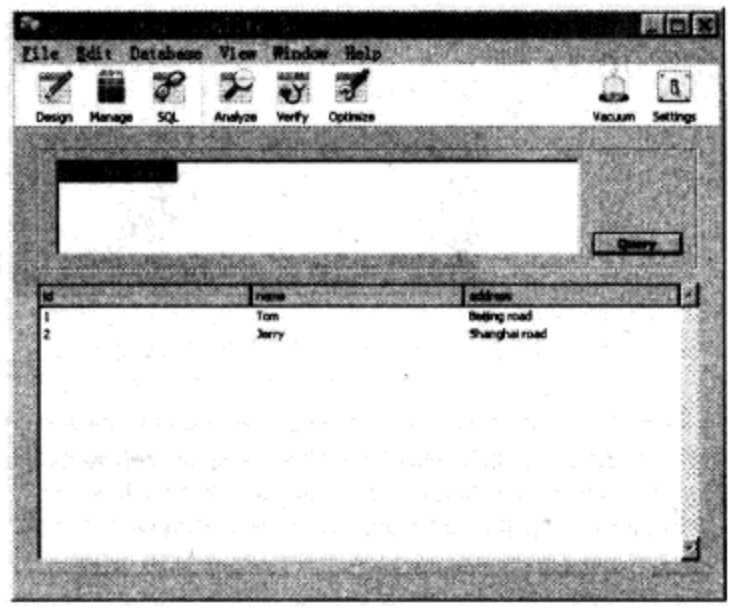


图 10-15　查询表 address 的内容

（5）调用 cur. fetchall( )、cur. fetchmany( )或 cur. fetchone( )返回查询结果。

（6）关闭 cur 和 conn。

【例 10-8】创建数据库 addresses 并创建表 address。然后向表 address 中插入数据并查询数据。

```
import sqlite3

#连接数据库
conn = sqlite3.connect("D:/developer/python/example/10/addresses.db")
#创建表
conn.execute("create table if not exists address(id integer primary key autoincrement,
name varchar(128), address varchar(128))")
#插入数据
conn.execute("insert into address(name, address) values ('Tom', 'Beijing road')")
conn.execute("insert into address(name, address) values ('Jerry', 'Shanghai road')")
#手动提交数据
conn.commit()
#获取游标对象
cur = conn.cursor()
#使用游标查询数据
cur.execute("select * from address")
#获取所有结果
res = cur.fetchall()
print "address: ", res
for line in res:
    for f in line:
        print f,
    print
#关闭连接
cur.close()
conn.close()
```

- 第 1 行代码导入 sqlite3 模块。
- 第 4 行代码连接数据库 addresses. db。如果 addresses. db 不存在，则创建该数据库文件。
- 第 6 行代码，如果表结构 address 不存在，则创建表 address。
- 第 9 行、第 10 行代码向表 address 中插入数据。
- 第 12 行代码手动提交数据。在调用 execute( )方法之前设置数据库的事务级别 conn. isolation_level = " "，这样就可以不用调用 commit （） 方法提交数据。

更多资源请访问稀酷客(www.ckook.com)

- 第 14 行代码调用 cursor( )方法，返回游标对象 cur。对象 cur 用于查询数据。
- 第 16 行代码调用对象 cur 的 execute( )查询表 address。
- 第 18 行代码调用对象 cur 的 fetchall( )返回表 address 的所有数据，并把数据存储在对象 res 中。
- 第 19 行代码输出结果集对象 res。输出结果如下所示：

```
address:  [(1, u'Tom', u'Beijing road'), (2, u'Jerry', u'Shanghai road')]
```

- 第 20 行到第 23 行代码循环输出结果集对象 res：

```
1 Tom Beijing road
2 Jerry Shanghai road
```

- 第 25 行代码关闭游标对象 cur。
- 第 26 行代码关闭数据库连接对象 conn。

10.4　小结

本章介绍了 Python 的数据库编程，重点讲解了 Python 连接数据库的几种方式，包括 ODBC、DAO、ADO 和专用模块。结合各种连接方式，演示了 Python 连接 Access、MySQL、Oracle 等数据库的方法。Python 提供了对持久化的支持，提供了 dbhash、shevle、pickle 等模块。这些模块可以创建 DBM 类型的数据库，并返回字典类型的对象。最后介绍了嵌入式数据库 SQLite，该数据库和 Python2. 5 一起发布，适合于桌面应用程序的数据库。

更多资源请访问稀酷客(www.ckook.com)

更多资源请访问稀酷客(www.ckook.com)

第二篇
Python 的 GUI 程序设计

第 11 章　WxPython 的开发环境

GUI（Graphical User Interface，图形用户界面）开发是一种关系到用户和计算机交互的技术，直接影响到终端用户的使用感受以及 GUI 软件的使用效率。Python 作为一种“胶水性”的语言，提供了众多 GUI 开发库的绑定，适合 GUI 的快速开发。在本章中，将介绍已有的 Python 绑定的多种开发库。在此基础上，具体介绍 WxPython 及其开发环境。本章的知识点：

- 常见的 GUI 开发库 Python 绑定
- WxPython GUI 开发库
- WxPython 的开发环境

11.1　Python 所支持的 GUI 开发库

使用 Python 语言，可以通过多种 GUI 开发库来进行 GUI 开发。这些开发库中有内置在 Python 发行版中的 Tkinter，也包括更庞大和完善的的 PyGtk 开发库等。在使用 WxPython 开发 GUI 程序之前，本节将对 Python 所支持的其他 GUI 开发库进行介绍。

11.1.1　Tkinter

Tkinter 是 Python 发行版本中的标准 GUI 库，通过它可以访问 Tk 的 GUI 操作。这是一种小巧的 GUI 开发库，开发速度快，在小型程序的开发中仍有不少的应用。Tkinter 可以运行在多种系统平台下，包括 Linux、Windows 和 Macintosh 等。从版本 8.0 开始，Tk 在各个系统平台下提供了相同的外观。这种特性使得使用 Tkinter 模块构造的应用程序在不同的系统平台下看起来都是一样的。

在 Tkinter 中有大量的模块，其中 Tk 的接口是在名为_tkinter 的二进制模块中。在此模块中包含对 Tk 的底层访问接口，一般作为共享库或者 DLL 而存在，或者是静态链接到 Python 的解释器中。但是在实际应用程序中，一般情况下不需要使用这些接口。为此，Tkinter 开发库提供了另外一些访问接口。其中最重要的是 Tkinter 和 Tkconstants 模块。实际上，前者将会自动导入后一个模块。因为 Tkinter 模块已经在 Python 的发行版本中内置，所以在使用 Tkinter 开发库的时候，只需要导入 Tkinter 模块即可。下面是两种导入 Tkinter 模块的方式：

更多资源请访问稀酷客(www.ckook.com)

```
In [1]: import Tkinter

In [2]: from Tkinter import *
```

注意　这里使用了 IPython 工具，具体使用方法可参见第 22 章。

- 在 In[1]中直接导入了 Tkinter，所有的 Tkinter 开发库符号可以在此名字空间下使用。
- 在 In[2]中，导入了 Tkinter 模块下的所有符号，从而可以直接使用。

【例 11-1】演示使用 Tkinter 开发库。

```
In [1]: from Tkinter import *

In [2]: root = Tk()

In [3]: word = Label(root, text = "Hello, world!")

In [4]: word.pack()

In [5]: root.mainloop()
```

- 在 In[1]中，导入了 Tkinter 模块中的所有符号，包括使用 Tk 接口所需要的符号等。
- 为了能够初始化 Tkinter，首先需要创建 Tk 的一个根部件。实际上此窗口部件是一个普通的窗口，包括一个标题栏和窗口管理器所提供的窗口装饰部分，如标题栏和最大化按钮等。在一个 Tkinter 开发的应用程序中，只需要创建一个根窗口部件即可。而且此部件的创建必须是在其他的窗口部件创建之前。在 In[2]中，直接使用 Tk 可以创建一个这样的部件。
- 在 In[3]中创建了一个标签部件，并赋值给 word 变量。其中，第一个参数为此部件的父部件，这里为刚刚创建的根窗口。一个标签部件可以显示文本或者图标，甚至是图像。在这里，使用 text 的参数来显示文本“Hello，world!”。
- 在 In[4]中，调用了 word 的 pack 方法。此标签将会根据自身的文本大小来自适应的加入到根窗口部件中。
- 在上面的步骤中，已经创建了一个最简单的 Tkinter 窗口。可以通过调用根窗口部件的 mainloop 方法使 Tkinter 进入其事件循环。在调用此方法后，系统将会显示刚刚创建的窗口。此窗口将会一直显示，直到用户手工将其关闭。实际上，在事件循环中，将会处理 3 类事件：第一类是用户提交的响应，包括键盘按键和鼠标单击等。第二类是窗口系统的事件响应，包括窗口的放大、缩小和重绘等。第三类是 Tkinter 本身的事件响应，包括窗口的排列（通过 pack 方法）等。

图 11-1 所示是【例 11-1】运行后显示的窗口。

从图 11-1 中可以看出，此窗口中只有一个简单的文本标签部件。上方的标题栏中默认是“tk”。并且含有当前系统的窗口装饰，包括最大化、最小化和关闭按钮。用鼠标拖动窗口的边缘可以改变窗口的大小。实际上，图 11-1 在高度上就进行了拉伸。同时可以看到，这里的文本标签并没有相应的改变位置。

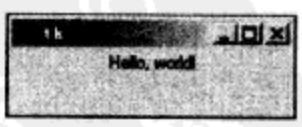


图 11-1　最简单的 Tkinter 窗口

在 Tkinter 开发库中，也提供了适合于大型软件开发的方式。可以使用类来将不同的窗口部件组合起来，从而形成一个复杂的窗口。同时，不同的窗口部件还可以响应不同的事件。

【例 11-2】演示将不同的窗口部件组合起来形成一个复杂的窗口部件，不同的窗口部件可以响应不同的事件。

更多资源请访问稀酷客(www.ckook.com)

```
#filename: tkinter_1.py
from Tkinter import *

class App: #第 4 行,自定义了一个 App 类
    def __init__(self, master): #定义窗体
        frame = Frame(master)
        frame.pack()

        self.hello = Button(frame, text = "Hello",
              command = self.hello)
        self.hello.pack(side = LEFT)

        self.quit = Button(frame, text = "Quit", fg = "red",
              command = frame.quit)
        self.quit.pack(side = RIGHT)

    def hello(self):
        print "Hello,world!"

root = Tk()
root.wm_title("Hello") #设置标题
root.wm_minsize(200, 200) #设置窗口大小

app = App(root)

root.mainloop()
```

- 在代码的第 2 行导入了 Tkinter 中的所有符号。
- 在第 4 行中定义了一个 App 类。在此类的构造函数中，定义了一个自定义的窗口部件。在此窗口部件中有一个 Frame。在每个 Frame 中有两个命令按钮。在每个按钮的创建中，通过 command 参数变量来指定当用户单击时所要执行的代码。在这里，第一个按钮指定了一个自定义的处理函数，也就是打印出特定的字符串。第二个按钮则是调用了系统定义的方法，将退出整个事件循环。然后将这些按钮分别加入到 Frame 中去。在加入的时候，用 side 参数指明窗口部件的加入位置。
- 在创建了一个 App 类后。使用 Tk 方法创建了一个根窗口部件。接下来的两行代码则是分别设置了窗口的标题和大小。
- 在最后两句代码中，首先通过将刚生成的 root 对象传递给 App 类，从而生成了所需的自定义窗口。最后调用 mainloop 进入事件循环。

运行【例 11-2】的代码，可以得到如图 11-2 所示的窗口。

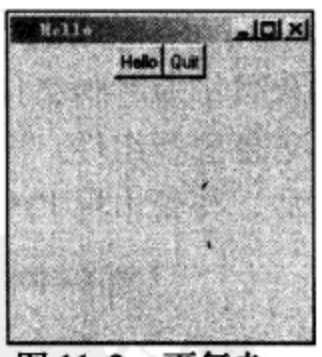


图 11-2　更复杂的 Tkinter 窗口

在图 11-2 所示窗口中，单击【Hello】按钮，在控制台上将会打印出相应的字符串。这是在 App 类的 hello 方法中定义的。单击多次则会打印出多条信息。单击【Quit】按钮，会退出事件循环，控制权将重新回到 SHELL 中：

```
In [7]: run tkinter_1.py
Hello,world!
Hello,world!

In [8]
```

- 实际上，pack 方法还支持更多的参数：

```
In [6]: help(Button.pack)
Help on method pack_configure in module Tkinter:
```

更多资源请访问稀酷客(www.ckook.com)

```
pack_configure(self,cnf = {},** kw) unbound Tkinter.Button method
    Pack a widget in the parent widget. Use as options:
    after = widget-pack it after you have packed widget
    anchor = NSEW (or subset)-position widget according to
    given direction
    before = widget-pack it before you will pack widget
    expand = bool-expand widget if parent size grows
    fill = NONE or X or Y or BOTH-fill widget if widget grows
    in = master-use master to contain this widget
    ipadx = amount-add internal padding in x direction
    ipady = amount-add internal padding in y direction
    padx = amount-add padding in x direction
    pady = amount-add padding in y direction
    side = TOP or BOTTOM or LEFT or RIGHT-where to add this widget.
```

- 在 In[6]中可以看出，pack 方法支持不少用于进行自定义设置的参数。例如可以通过设置 expand 的值来确定是否允许窗体进行扩展等。

在 Tkinter 中，也提供了众多的窗口部件以供选择。同时，由于它内置在 Python 标准发行版中，所以一些 Python 语言的工具也是使用 Tk 来实现的。如在 Python 目录下的 Tools/Scripts 目录中，有一个 redemo. py 脚本文件。使用此工具可以用来对正则表达式进行检测。其图形化的实现方式采用的就是 Tkinter。运行 redemo. py 时的窗口如图 11-3 所示。

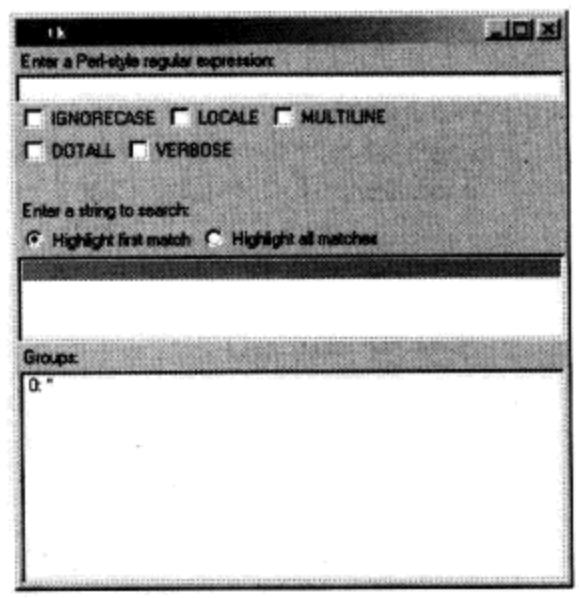


图 11-3　正则表达式分析器

为了能够更好地使用 Tkinter，已有一些封装的库可以使用，如 Pmw 和 Tix 等。通过使用这些高层的封装库，可以更容易地对 Tkinter 进行操作，从而加速 GUI 的开发。

11.1.2　PyGTK

GTK（GIMP ToolKit）是一个使用 C 语言编写的跨平台的图形界面开发库。其名字的来源在于此项目最初是为开发 GIMP 软件而建立的，但随着开发库组件的丰富，现在 GTK 已经成为了功能强大和窗体丰富的 GUI 开发库。著名的 Linux 下图形桌面环境 GNOME 就是基于 GTK 的。PyGTK 模块提供了通过 Python 语言来访问 GTK 图形界面开发库的接口（其项目主页为 http://www. pygtk. org），所以在安装 PyGTK 库之前需要在系统中安装有 GTK 开发环境。项目主页上提供了对各种组件的打包安装程序，包括 GTK 运行库、Python 解释器和 PyGTK 模块。可以根据需要选择组件。图 11-4 所示的窗口中显示了 PyGTK 安装时的组件选择。在选择了 PyGTK 模块后，将会出现 PyGTK 模块的安装提示，如图 11-5 所示。

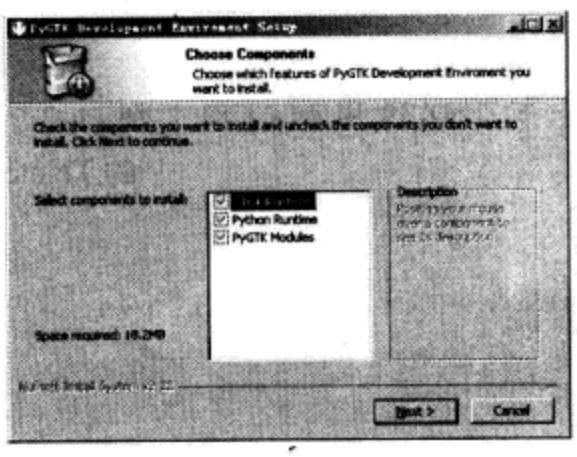


图 11-4　PyGTK 安装时的组件选择

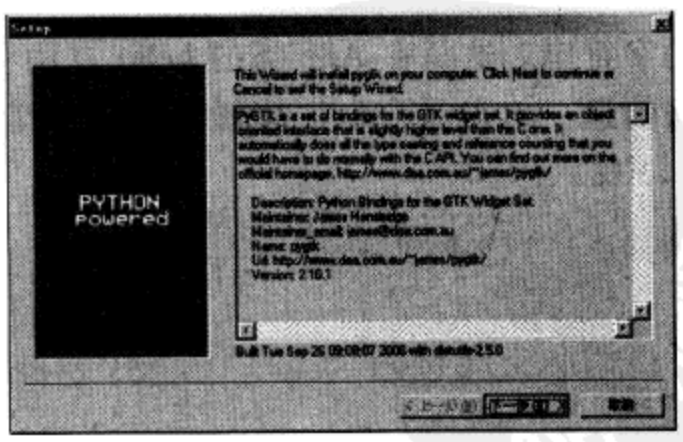


图 11-5　PyGTK 模块的安装

实际上，GTK 也是由多种组件构成的，包括基本数据结构 Glib、使用 C 实现的面向对象开发系统 Gobject、图形绘制库 Cario、字体渲染 Pango 等。其组成层次如图 11-6 所示。

更多资源请访问稀酷客(www.ckook.com)



图 11-6　GTK 的组成层次图

正因为 GTK 是由不同的组件构建而成，所以在安装 PyGTK 的时候，也会安装相关的模块。图 11-7 和图 11-8 显示了 PyCairo 和 PyGObject 的安装过程。

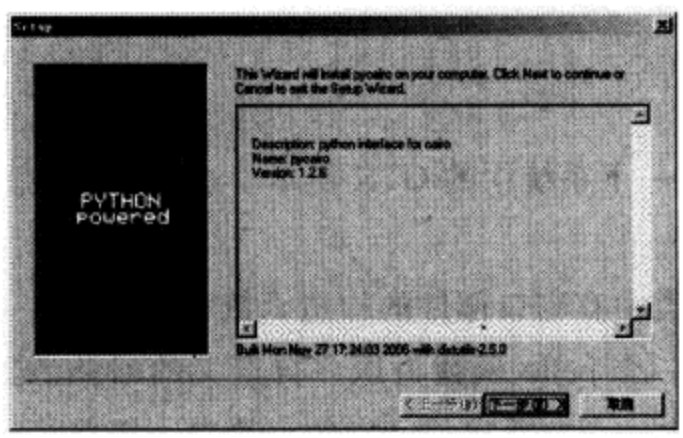


图 11-7　PyCairo 模块的安装

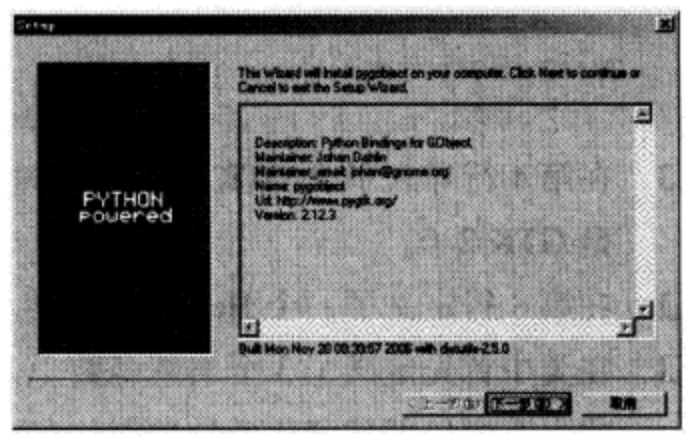


图 11-8　PyGObject 模块的安装

由于 GTK 可以运行在多种不同的系统平台下，所以其显示并不是特别的本地化。在 GTK 中提供了主题选择机制，程序设计者可以根据需要选择不同的窗口部件的主题。

【例 11-3】一个简单的 PyGTK 程序的代码。

```
#filename: gtk_1.py
#encoding = utf-8

import pygtk
pygtk.require('2.0')
import gtk

    class HelloWorld:
        def hello(self, widget, data = None): #自定义的回调函数
            print "Hello World"

        def delete_event(self, widget, event, data = None):
            print "delete event"
            return False

        def destroy(self, widget, data = None):
            gtk.main_quit()

        def __init__(self): #构建特定的窗口部件
            self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.window.connect("delete_event", self.delete_event)
        self.window.connect("destroy", self.destroy)
        self.window.set_border_width(10)
        self.button = gtk.Button("Hello World")
        self.button.connect("clicked", self.hello, None)
        self.button.connect_object("clicked", gtk.Widget.destroy, self.window)
        self.window.add(self.button)
        self.button.show()
        self.window.show()
    def main(self):
        gtk.main()
if __name__ == "__main__":
    hello = HelloWorld()
    hello.main()
```

- 在第 4 行导入了 PyGTK 模块，同时测试了一下系统中的 GTK 版本。这里需要的版本号是 GTK 2.0。
- 在第 8 行定义了一个 HelloWorld 类。在此类中有窗口部件的构造函数和信号等。
- 在类中首先定义了 3 个回调函数。其中第一个 hello 函数为用户自定义的回调函数，参数中有窗口部件和传递的数据。在此函数中，直接输出了特定的字符串。接着是另外两个回调函数，一个用于处理“delete_event”信号，另一个则是窗口销毁的时候被调用。在 delete_event 回调函数中，返回了“False”，这将会触发“destroy”信号，进一步销毁窗口。如果将返回值改为“True”，窗口将不会被销毁。在 destroy 方法中，调用了 main_quit 方法。调用此方法后，GTK 系统将退出事件循环。
- 在 HelloWorld 类的构造函数中，创建了一个特定的窗口。同时通过调用窗口的方法对窗口进行了一定的调整，如设置窗口的边框。然后生成了一个命令按钮，并将其加入到窗口中。要注意的是，在不同的窗口部件中，使用 connect 方法将特定的信号和前面定义的处理函数关联了起来。这样，在窗口遇到特定的事件时，就会调用注册的处理函数。例如，将 button 对象的 clicked 信号和 hello 方法关联起来，这样在单击此命令按钮的时候将会调用 hello 方法。最后调用 show 方法将窗口部件显示出来。
- 在此代码的 main 方法中，调用了 gtk 库的 main 方法。这在每个 PyGTK 应用程序中都是必须的。调用此方法后，系统将控制权交给 PyGTK 核心并等待事件的发生。
- 第 43 行生成了一个 HelloWorld 类的实例，并调用 main 方法使其进入事件循环。

运行【例 11-3】程序，将会显示如图 11-9 所示的窗口。

在图 11-9 所示窗口中，单击中间的命令按钮，将会在控制台上显示“Hello World”，然后退出。这是因为同时将此按钮的单击信号关联到了窗口的销毁信号上。当然，也可以直接单击窗口右上角的关闭按钮，此时将在终端上显示“delete event”的字符串，同时退出。

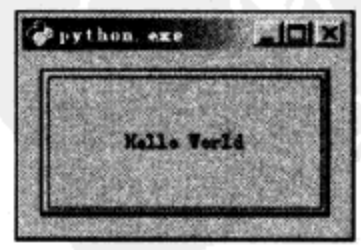


图 11-9　PyGTK 的窗口

更多资源请访问稀酷客(www.ckook.com)

PyGTK 作为一种开源的基于 LGPL 的图形开发库，具有丰富的窗口构件。适合于用来开发大型软件的图形用户界面。

11.1.3　PyQT

QT 作为一种和 GTK 竞争的图形用户界面开发库，同样有丰富的窗口部件类库。QT 是 Trolltech 公司开发的，现在已经扩展为一个跨平台的应用开发框架，其网址为 http://trolltech.com/products。本节将主要关注于其作为图形开发库的部分。Linux 系统平台下另一个桌面环境 KDE 就是基于 QT 的。PyQT 可以使用 Python 来访问 QT 图形开发库接口。可以从 http://www.riverbankcomputing.co.uk/software/pyqt/网址下载对应版本的模块。

可以通过编译源码来安装 PyQT 框架，但是这一般都不是必须的。在 Linux 系统下，可以通过直接从源中安装已经编译好的二进制模块。在 Windows 系统平台下，可以使用已经设置好的安装程序来安装。图 11-10 所示窗口为 PyQT 安装过程的截图。

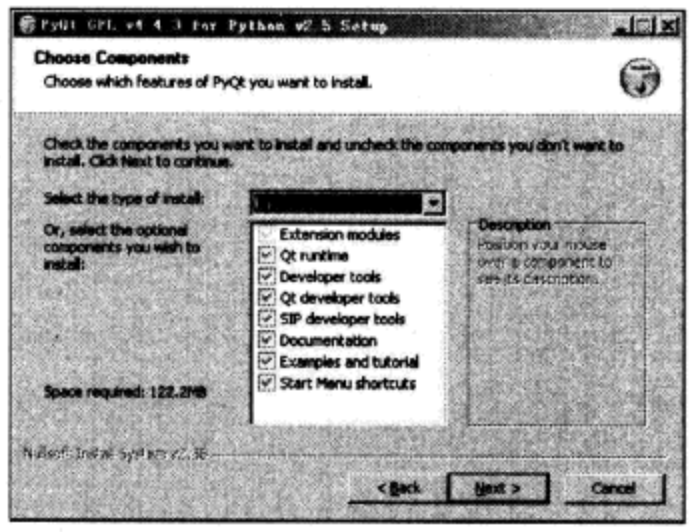


图 11-10　PyQT 的安装截图

从图 11-10 所示窗口中可以看到，PyQT 模块的安装不只是 PyQT 模块，同时还有 QT 的运行库以及一些相关的开发工具。如可以可视化的构建 QT 的图形用户界面程序的 QT Designer 工具。

表 11-1 中介绍了在 PyQT 中所包含的组件。

表 11-1　PyQT 的组件

组件名称	说　明
QtCore	Qt 的实现核心部分，包含事件循环和信号机制等
QtGui	Qt 的 GUI 类库
QtHelp	Qt 的帮助文档实现
QtNetwork	包含 Qt 的网络部分实现
QtOpenGL	Qt 的 OpenGL 支持
QtScript	Qt 的 Javascript 脚本支持
QtSql	Qt 的 Sql 语言支持
QtSvg	可以显示 SVG 类型的文件
QtTest	Qt 的单元测试框架
QtWebkit	Qt 的浏览器引擎实现
QtXml	Qt 的 XML 处理实现，包括 SAX 和 DOM
QtXml Patterns	Qt 的 Xquery 和 Xpath 实现

对于简单的图形界面，一般会用到 QtCore 和 QtGui 组件。但是对于复杂的应用程序，可能会用到【例 11-4】代码中所示的其他组件。

【例 11-4】一个简单的 PyQT 程序的代码。

更多资源请访问稀酷客(www.ckook.com)

```
#filename: qt_1.py
#encoding = utf - 8
import sys
from PyQt4 import QtCore, QtGui

class MyWidget(QtGui.QWidget): #实现自定义的窗口部件
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)

        self.setFixedSize(200, 120)

        self.quit = QtGui.QPushButton("Quit", self)
        self.quit.setGeometry(62, 40, 75, 30)
        self.quit.setFont(QtGui.QFont("Times", 18, QtGui.QFont.Bold))

        #关联信号
        self.connect(self.quit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp, QtCore.SLOT("quit()"))

app = QtGui.QApplication(sys.argv)

widget = MyWidget()
widget.show()

sys.exit(app.exec_())
```

- 在第4行首先从 PyQt4 模块中导入了 QtCore 和 QtGui 模块。也就是说，在这个简单的应用程序中，仅仅只是使用了这两个组件。
- 第6行定义了一个 MyWidget 类，此类是从 Qwidget 类中继承的。在其构造函数中，构造了一个自定义的窗口部件。此窗口中只有一个命令按钮。并通过相关方法设置了字体等属性。第17行调用 connect 方法将信号和处理函数关联了起来。
- 在第20～25行语句中，首先通过调用 Qapplication 类构建了一个 Qt 的应用。将命令行输入 sys.argv 作为了其参数。然后生成了 MyWidget 的一个对象实例，此时将会生成具体的窗口。调用 show 命令将其显示出来。最后调用 exec_方法进入事件循环。

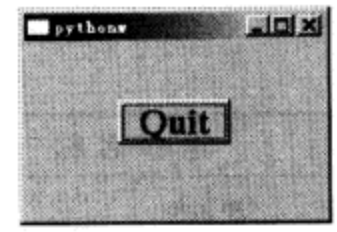


图 11-11　PyQT 的窗口

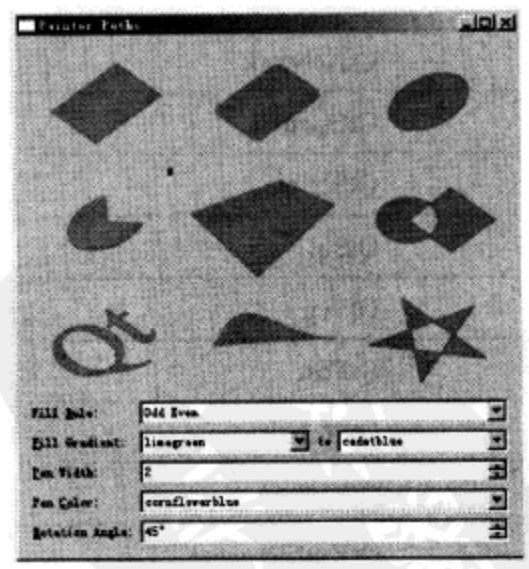


图 11-12　PyQT 中的图形绘制

执行【例 11-4】代码段，可以看到如图 11-11 所示的窗口。虽然图11-11所示只是一个简单的 PyQT 窗口，但是实际上使用 PyQT 可以构建出很复杂的图形用户界面。图 11-12 显示了 PyQT 中各种图形的绘制。

如果需要迅速的构建 GUI 应用程序，PyQT 是一个很好的选择。

11.2　WxPython 介绍

WxPython 作为一种开源的 GUI 图形用户界面开发库，同样具有跨平台的特性。本节将对

更多资源请访问稀酷客(www.ckook.com)

WxPython 和其封装进行介绍。

11.2.1　下载和安装 WxPython

WxPython 是跨平台 GUI 图形界面开发库 wxWidgets 的 Python 语言绑定。其中有丰富的窗体部件类，使得用户可以快速的创建功能强大、界面元素丰富的 GUI 应用程序。同时，WxPython 是跨平台的，也就是说，同一个应用程序不经过修改就可以运行在多种系统平台下，包括 Windows、Linux 和 Macintosh 等。

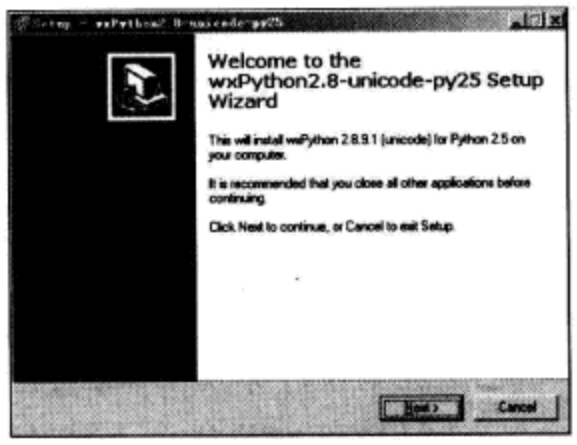


图 11-13　WxPython 的安装截图

WxPython 为使用此开发库的应用程序提供了基于相应系统平台的外观。在 Windows 下，会使用 Windows 的窗体效果；在 Linux 下，会使用桌面环境下的窗体效果。

WxPython 的项目主页是 ttp://wxpython. org/。可以从 http://wxpython. org/download. php 上下载到最新的 WxPython 版本。由于 WxPython 是跨平台的，所以在各个系统平台下都可以直接编译源代码进行安装。不过这样做是比较麻烦的。实际上，可以根据当前系统选择已经编译好的软件包。在 Linux 下，可以根据 Linux 的发行版本选择 DEB 包或者 RPM 包来安装。在 Windows 下，可以选择对应的可执行文件。图 11-13 所示窗口是 Windows 系统平台下 WxPython 的安装截图。

安装完毕后，可以使用下面的操作来测试 WxPython 是否安装成功：

```
In [1]: import wx
In [2]: wx.version()
Out[2]: '2.8.9.1 (msw-unicode)'
```

- 在 In[1]中导入了 wx。所有 WxPython 下的操作符号都定义在这下面。如果此处报错，找不到 wx 模块，则表示 WxPython 的安装没有成功。需要重新安装。另外要注意的是，此语句最好是作为 WxPython 应用程序的开始部分。因为 WxPython 中的子模块如 xrc 等是需要以导入的 wx 模块作为使用前提的。
- 在 In[2]中使用 version 方法来查看当前安装的 WxPython 的版本。在这里当前的版本为“2.8.9.1”，系统平台则是 MS Windows。

11.2.2　第一个 WxPython 程序

使用 WxPython 编写 GUI 应用程序是非常简单的。WxPython 图形开发库提供了大量的窗口部件类来丰富界面元素。首先来看看简单的 WxPython 应用。

【例 11-5】用 WxPython 程序在一个窗口上显示“Hello world!”字符串。

```
#filename: wx_1.py
#encoding = utf-8

#导入 WxPython 包
import wx
class MyFrame(wx.Frame): #创建自定义 Frame

    def __init__(self, parent):
```

更多资源请访问稀酷客(www.ckook.com)

```
        wx.Frame.__init__(self, parent,-1, 'Hello World',
                          size = (300, 300)) #设置窗体

        panel = wx.Panel(self)
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)

        txt = wx.StaticText(panel, -1, 'Hello world! ') #创建文本窗口部件
        sizer.Add(txt, 0, wx.TOP|wx.LEFT, 100)

        self.Centre()

app = wx.App()

frame = MyFrame(None)
frame.Show(True)

app.MainLoop() #进入事件循环
```

- 在第5行导入了 wx 包，这也是所有 WxPython 应用程序中必须的。在以前的代码中，这是在 wxPython 下定义的。
- 第7行定义了一个 MyFrame 类，它继承自 wx 包中的 Frame 类。在 MyFrame 类中，仅仅定义了构造函数，以便进行具体的窗体设置。
- 在构造函数中，首先调用了父类 Frame 的构造函数。在这里还对父窗体进行了设置，包括窗口的标题栏和窗口大小。
- 接着生成了一个 Panel 和 Sizer。这些是 WxPython 所提供的窗口部件，在后面将会进行具体的介绍。这里是容器部件，可以用来放置其他的窗口部件。
- 随后生成了一个静态文本窗口部件，并将其加入到了 Sizer 部件中。在其中，指定了加入的方向和位置等。
- 在此 MyFrame 类的最后，调用了 Centre 方法将窗口放置在桌面环境的中间。
- 随后使用 App 生成了一个 WxPython 的应用实例。然后使用刚刚定义的类生成了一个窗口实例，并调用 Show 方法将此窗口显示。然后调用 MainLoop 使得此应用进入事件循环。最终，所定义的窗口将会显示出来。



图 11-14　简单的 WxPython 窗口

运行【例 11-5】代码，可以得到如图 11-14 所示的窗口。

【例 11-5】仅仅是一个很简单的 WxPython 窗口示例。在接下来的章节中，将对 WxPython 的组成部分进行具体的介绍。

11.2.3　WxPython 的封装：Pythoncard

为了能够更好的使用 WxPython 工具库，不少项目开始对 WxPython 包进行封装。这里要介绍的PythonCard 项目就是为达到封装的目的。此项目为 WxPython 工具库的开发者提供了许多的实用工具和类库来帮助创建跨平台的应用。其项目主页为 http://pythoncard.sourceforge.net/。同其他的 Python 包一样，PythonCard 也提供了源码安装和软件包安装的方式。图 11-15所示为 PythonCard 可执行文件在 Windows 系统下安装的截图。

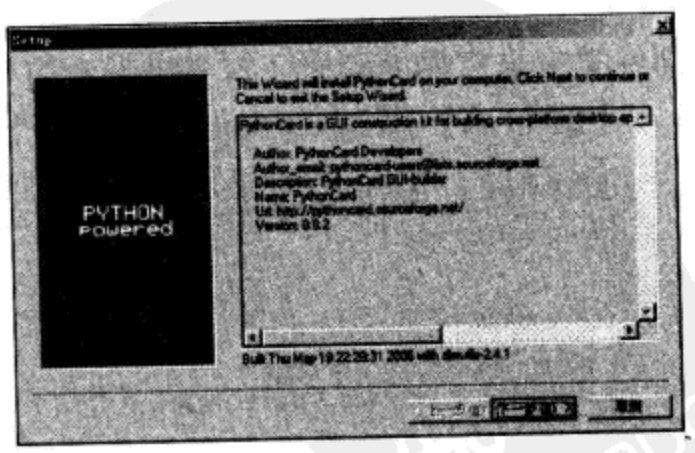


图 11-15　PythonCard 的安装截图

更多资源请访问稀酷客(www.ckook.com)

PythonCard 采用了一种资源文件来对界面进行描述。下面的资源文件描述了一个最简单的窗口。其中仅有一个文本框：

```
#filename: pythoncard_1.rsrc.py
{ 'application':{ 'type':'Application',
            'name':'Minimal',

    'backgrounds':
 [
  { 'type':'Background',
    'name':'bgMin',
    'title':'Minimal',
    'size':( 165, 100 ),

   'components':
   [
    { 'type':'TextField',
      'name':'field1',
      'position':(5, 5),
      'size':(150, -1),
      'text':'Hello World! ' },
   ]
  }
 ]
 }
 }
```

- 这里资源描述文件中的字段都是自解释的。可以从各个字段中看出其含义。

下面是代码文件内容：

```
#filename: pythoncard_1.py
from PythonCard import model

class Minimal(model.Background):
    pass

if __name__ =='__main__':
    app = model.Application(Minimal)
    app.MainLoop()
```

- 从这个代码段来看，使用 PythonCard 构建一个 GUI 应用程序是很方便的。代码段的第 2 行从 PythonCard 包中导入了 model 模块，第 4 行构建了一个 Minimal 类。随后构建了一个应用，并将此 Minimal 类作为参数传递进来。最后调用 MainLoop 方法，即可显示界面。

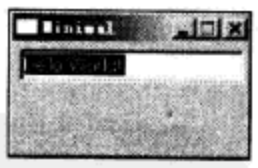


图 11-16　PythonCard 下的简单界面

运行这个代码文件，可以得到如图 11-16 所示的窗口。

值得提及的是，WxPython 中也可以使用资源文件来描述界面。另外，PythonCard 项目已经处于停滞状态，但是这仍然给 GUI 应用的开发提供了另外一种选择。

11.3　WxPython 的开发工具

为了能够加速 WxPython 下的界面开发，可以使用 GUI 设计工具。利用这些工具可以用可视化的方法来设计界面，从而减少编写代码的工作量。本节将介绍一些开源的 WxPython 开发工具。

更多资源请访问稀酷客(www.ckook.com)

11.3.1　使用 wxGlade 开发 WxPython 程序

wxGlade 是一个 GUI 设计工具，其模型来源于知名的 GTK +/GNOME 界面设计工具 Glade，所以 wxGlade 的界面显示和 Glade 的界面显示是相似的，采用了多窗口的显示方式。使用wxGlade 可以生成界面的 Python 代码。其项目主页为 http://wxglade. sourceforge. net/。可以从网站上下载源码或者是对应系统的安装包。图 11-17 所示为 Windows 系统下安装的截图。wxGlade 采用多种窗口的形式，包括程序窗口、构件窗口、属性窗口和窗体可视化窗口等。可以直接单击窗口部件来设计 GUI，如图 11-18 所示。

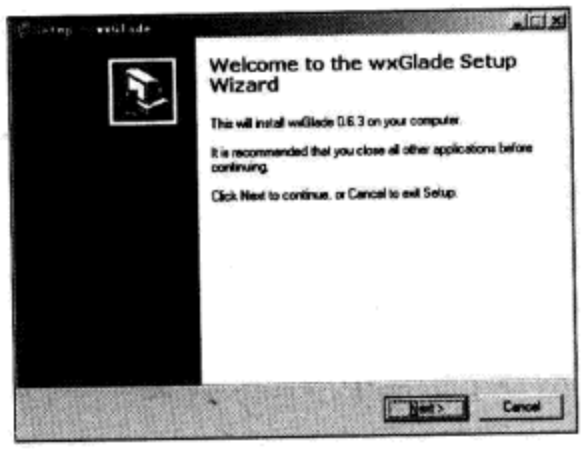


图 11-17　wxGlade 的安装截图

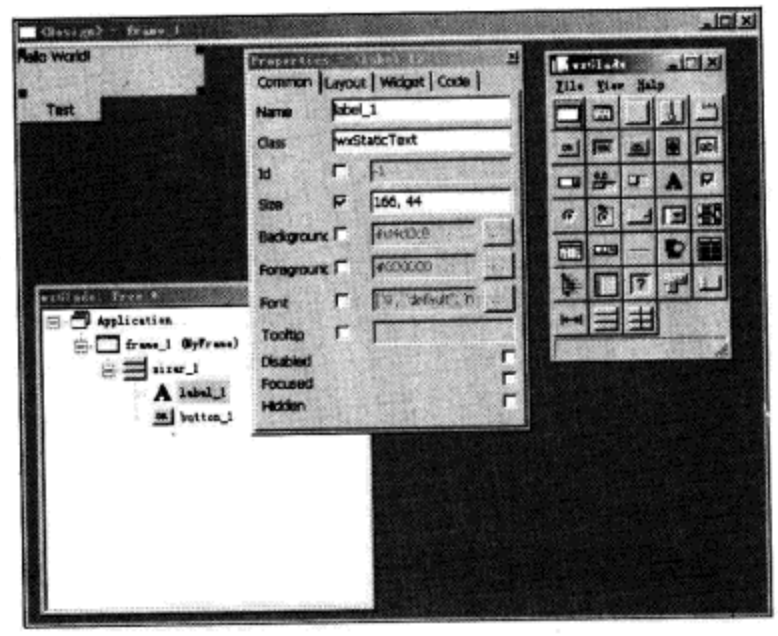


图 11-18　wxGlade 程序截图

11.3.2　使用 wxFormBuilder 开发 WxPython 程序

另外一个开源的 GUI 设计工具是 wxFormBuilder。其项目主页为 http://wxformbuilder. org/。可以从网站上下载源文件或者软件安装包。图 11-19 所示为 Windows 系统下的安装截图。wxFormBuilder 采用了一种集成窗口的形式，将如 wxGlade 中分散的窗口放在了一起。可以通过选择组件面板中的控件来设计 GUI，如图 11-20 所示。

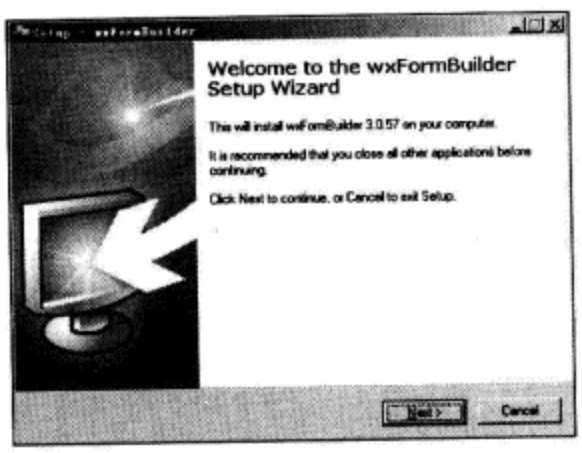


图 11-19　wxFormBuilder 的安装截图

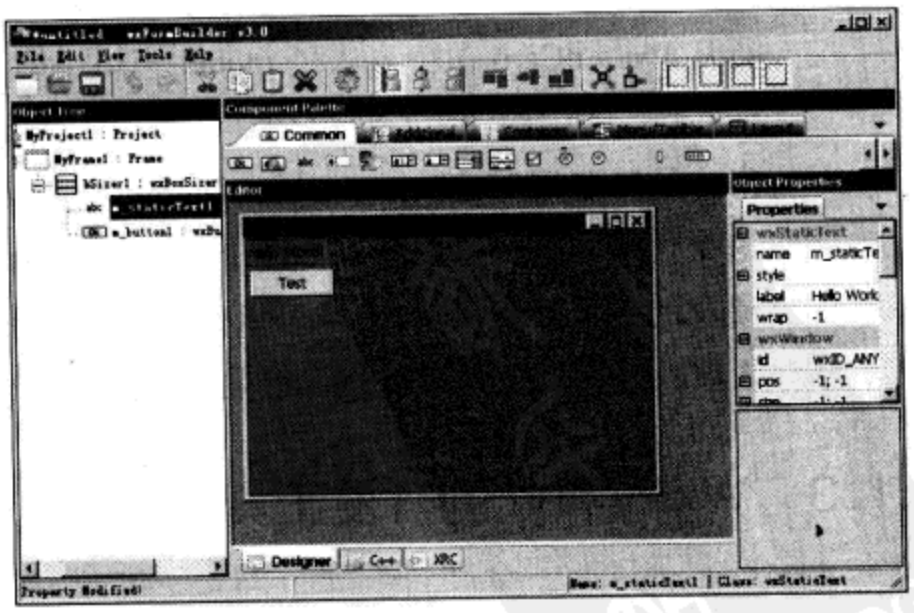


图 11-20　wxFormBuilder 程序截图

wxFormBuilder 可以输出 WxPython 的界面的资源描述文件。

更多资源请访问稀酷客(www.ckook.com)

11.3.3　使用 Boa-constructor 开发 WxPython 程序

Boa-constructor 既是一个 WxPython 的界面设计工具，也是一个强大的 Python 语言 IDE。其项目主页为 http://boa-constructor. sourceforge. net/。可以从该主页下载源文件或者软件安装包。图 11-21 所示为 Windows 系统下的安装截图。用该设计工具，可以通过单击控件来进行界面设计。一个简单的例子如图 11-22 所示。



图 11-21　Boa-constructor 的安装截图

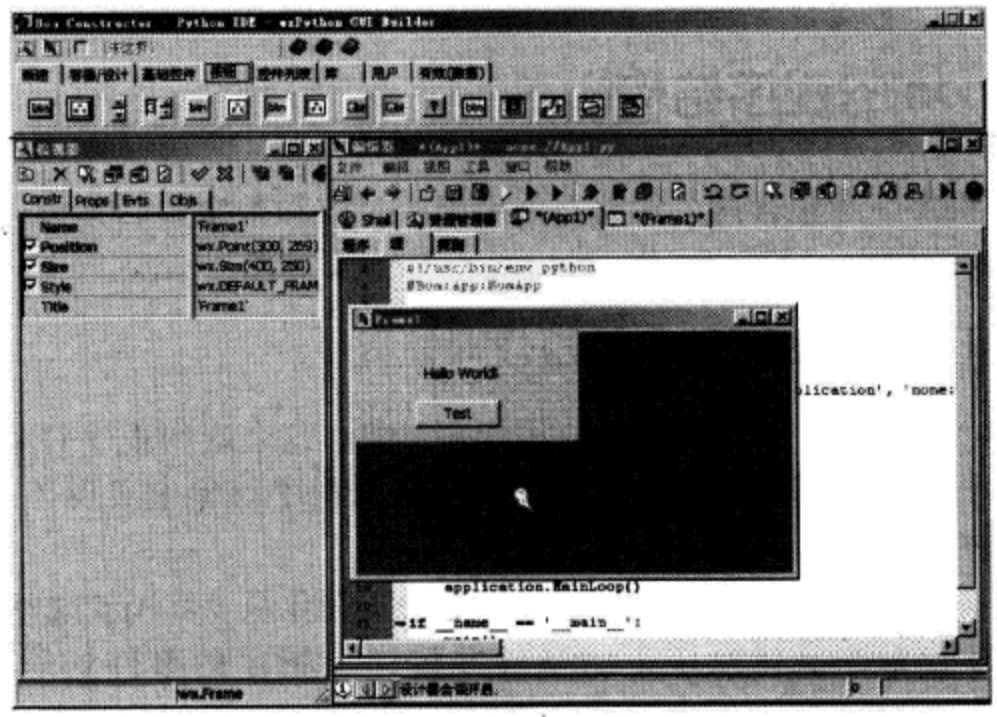


图 11-22　Boa-constructor 程序截图

11.4　小结

本章介绍了 WxPython 下的开发环境。首先介绍了 Python 下已有的图形开发库，包括 Tkinter、PyGTK 和 PyQT 等。接着介绍了当前比较流行的 GUI 开发库——WxPython，包括其安装和基本的使用方法。在此基础上，介绍了对 WxPython 的封装 PythonCard 开发库。最后，介绍了 WxPython 下的 GUI 设计工具，包括 wxGlade、wxFormBuilder 和 Boa-constructor 等。

更多资源请访问稀酷客(www.ckook.com)

第 12 章　WxPython 框架初步

WxPython 框架是一种开源的图形化用户界面(Graphical User Interface，GUI)开发框架。WxPython提供了丰富的类库用来构建 GUI 界面。同时，基于对象的编程方式使得 GUI 开发更加容易。本章的知识点：

- WxPython 的程序结构
- WxPython 框架中的事件
- 对话框
- 工具栏
- 状态栏

12.1　WxPython 的程序结构

一个 WxPython 程序一般包含两个对象：应用程序对象和根窗口。其中，应用程序对象可以通过实例化 wx. App 来实现，根窗口可以通过 wx. Frame 来实现。

12.1.1　WxPython 应用程序的组成

每一个 WxPython 应用程序，都有一个应用程序对象。这个应用程序对象拥有至少一个根窗口，这是 WxPython 程序的必需组成部分。另外，在应用程序对象中实现了一个事件循环处理，将处理窗口和其构件中的事件。

在 WxPython 中，可以通过实例化 wx. App 或者是实例化从 wx. App 继承的子类来生成应用程序对象。此对象将会管理窗口系统中的事件并传递给特定的事件处理函数。在上一章的【例 11-3】中，直接使用了实例化 wx. App 类。其代码部分如下：

```
app = wx.App()

frame = MyFrame(None)
frame.Show(True)

app.MainLoop()  #进入事件循环
```

- 在第 1 行中使用 wx. App 来生成应用程序对象。

【例 12-1】接【例 11-3】。使用从 wx. App 继承子类的方法来实现【例 11-3】程序的功能。

```
#filename: wx_2.py
#encoding = utf-8
import wx

class MyFrame(wx.Frame): #自定义窗体
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, 'Hello World', size = (300, 300))
```

更多资源请访问稀酷客(www.ckook.com)

```
        panel = wx.Panel(self)
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)

        txt = wx.StaticText(panel, -1, 'Hello world! ')
        sizer.Add(txt, 0, wx.TOP|wx.LEFT, 100)
        self.Centre()

class MyApp(wx.App): #自定义应用程序对象

    def OnInit(self):
        self.frame = MyFrame(None)
        self.frame.Show(True)
        return True

app = MyApp() #使用从 wx.App 继承的子类
app.MainLoop()
```

- 【例 12-1】代码的前半部分和【例 11-3】中是一样的。其中的不同在于这里定义了一个 MyApp 类，此类是从 wx. App 类中继承的。
- 第 19 行在此类中定义了一个 OnInit 方法。在此方法中实例化了前面定义的 MyFrame 类，并调用 Show 方法将此窗口显示出来。注意到此方法的最后返回了“True”值，一般情况下这是 WxPython 应用程序所需要的。当此方法返回“False”值的时候，应用程序会立即退出。很多情况下这是因为应用程序所需要的资源缺失而无法继续运行。
- 第 24 行使用自定义的 MyApp 类构造了应用程序对象。而不是原来的 wx. App 类。第 25 行调用 MainLoop 方法运行应用程序。

运行这段代码，同样的将会显示如图 11-14 所示的窗口。

这里的 OnInit 函数将在事件循环处理之前被 WxPython 系统调用。同样地，如果需要在事件循环处理结束之后来执行某些代码，则可以在类中定义 OnExit 函数。在原来的 MyApp 类中，加入 OnExit 函数的代码如下：

```
class MyApp(wx.App): #自定义应用程序对象

    def OnInit(self):
        self.frame = MyFrame(None)
        self.frame.Show(True)
        return True

    def OnExit(self): #自定义的 OnExit 函数
        pass
```

这里除了定义 OnInit 函数之外，还定义了一个 OnExit 类。其中 OnInit 函数是 wx. App 的子类中必需的。OnExit 函数则是在事件循环处理结束之后执行的。在这里并没有进行任何的操作。实际上，在这里可以加入退出时的清理工作。一般说来，这里的操作是和 OnInit 方法中的操作相对应的。例如在 OnInit 方法中执行了数据库的连接操作，那么在 OnExit 方法中可以断开数据库连接。

当 WxPython 程序有输出结果的时候，默认情况下将显示在单独的 WxPython 窗口中。通过在代码中加入调试信息，则可以从诊断窗口中看到具体的程序执行过程。

【例 12-2】接上例。在【例 12-1】代码中加入一些调试信息（其中省略了部分重复的代码）。

```
#filename: wx_3.py
#encoding = utf - 8
import wx
```

更多资源请访问稀酷客(www.ckook.com)

```
class MyFrame(wx.Frame): #自定义窗体
    def __init__(self, parent):
    print "MyFrame __init__"
        wx.Frame.__init__(self, parent, -1, 'Hello World', size = (300, 300))
#省略部分代码

class MyApp(wx.App): #自定义应用程序对象
    #def __init__(self,redirect = True, filename = None):
    #    wx.App.__init__(self, redirect,filename)

    def OnInit(self):
        print "MyApp OnInit"
        self.frame = MyFrame(None)
#省略部分代码

    def OnExit(self):
        print "MyApp OnExit"
        import time
        time.sleep(2)

if __name__ == '__main__':
    print "Main start"
    app = MyApp() #使用从 wx.App 继承的子类
    print "Before MainLoop"
    app.MainLoop()
    print "After MainLoop"
```

- 【例 12-2】主要是在程序代码的各个阶段加入了调试信息。这里主要是输出信息，以便看到 WxPython 执行的过程。在实际编程中可以加入具体的调试信息。
- 在 OnExit 方法中使用了一定时间的休眠，这主要是因为 OnExit 实际上是在事件循环处理结束的时候调用。如果不加入这段休眠代码，信息显示的时间会非常短而看不见。

运行【例 12-2】，将同样得到如图 11-14 所示的窗口。但是，此时还会同时显示一个调试信息窗口，输出在代码中定义的消息，如图 12-1 所示。当关闭所有窗口的时候，将会显示如图 12-2 所示的调试信息窗口。



图 12-1　WxPython 程序的调试信息窗口

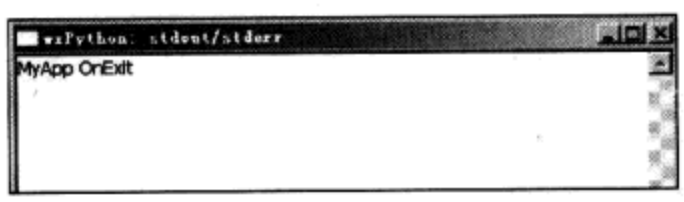


图 12-2　WxPython 程序退出时的调试信息窗口

系统终端的输出信息如下：

```
In [1]: run wx_3.py
Main start
After MainLoop

In [2]:
```

从这里可以看出，只有在实例化生成了 WxPython 的应用程序对象后，输出的信息才会显示在 WxPython 的调试信息窗口中。否则，将显示在系统的终端下。

12.1.2　WxPython 窗口的组成

一个 WxPython 应用程序至少要有一个根窗口。在前面的示例代码中，已经知道可以从 wx. Frame 类中继承而创建窗口。一个应用程序对象和一个根窗口是关联的。在应用程序对象中，

更多资源请访问稀酷客(www.ckook.com)

可以通过调用 SetTopWindow 方法来设置根窗口。在前面的示例代码中并没有使用这个方法。这是因为应用程序对象将会选择窗口列表中的第一个作为根窗口。也就是说，没有必要显示的定义一个根窗口。通过调用 SetTopWindow 方法，还可以更改根窗口。

在 wx. Frame 的构造函数中，可以通过设置其参数来改变默认窗口的样式和行为。这些参数如表 12-1 所示。

表 12-1　wx. Frame 构造函数的参数

参　数	说　明
parent	父窗口
id	窗口的 ID
title	窗口的标题
pos	窗口在屏幕的初始位置
size	窗口的默认大小
style	窗口的类型
name	窗口的内在名字

对于父窗口而言，parent 参数设置为“None”。Frame 将会随着父窗口的销毁而销毁。也就是说，当关闭父窗口的时候，其子窗口同样会被关闭。对于某些平台，将会限制子窗口在父窗口中的位置。pos 参数为窗口在显示屏幕中的初始化位置。默认为(-1， -1)，表示让窗口系统来选择。这里的位置是按照相对于屏幕左上角而言的。size 参数为窗口的初始化大小，默认为(-1， -1)，表示让窗口系统来决定这个尺寸大小。

对于每一个 WxPython 窗口，都有一个唯一的 ID 值。当然，这个唯一性是在一个 Frame 之内，不同的 Frame 之间，其 ID 值是可以重用的。在应用程序中，可以使用 GetId 方法来获取当前 Frame 的 ID 值：

```
class MyApp(wx.App):
    def OnInit(self):

        self.frame = MyFrame(None)
        id = self.frame.GetId()
        print "Frame ID:",id
#省略部分代码
```

在这里定义的 OnInit 方法中，调用了 GetId 方法获取当前 Frame 的 ID 值，并将其输出。

当给 id 参数传递 -1 的时候，WxPython 系统将会自动生成一个合适的 ID。实际上， -1 也是全局常量 wx. ID_ANY 的值。如果没有特殊的需求，可以采用这种方式来生成一个新的 ID。这样可以避免 ID 的冲突。另外一种方式是采用全局的 NewID 函数。这同样可以保证 WxPython 系统生成的 ID 值在当前 Frame 中是唯一的。如下面的代码片段：

```
id = wx.NewId()
frame = wx.Frame.__init__(None, id)
```

这个代码片段中采用了 NewID 函数来生成 ID 值。

还有一种直接的方式是显式的传递一个正整数给 Frame 的构造函数。如果不是必需的情况，最好不要采用这种方式。因为这种方式容易造成 ID 值的冲突。如果一定要用这种方法，则可以通过调用 RegisterID 函数来检查 ID 的唯一性。

除此之外，每个窗口都有一定的样式。这可以通过设置 style 参数来定义窗口部件的样式。一个窗口部件的显示表现在是否可以改变大小、是否可以最大化、最小化和是否可以关闭等。实

更多资源请访问稀酷客(www.ckook.com)

际上，窗口部件的样式是很多样式元素的组合。在 WxPython 中，这些样式元素通过一个常量来标识。而窗口部件的样式就是这些常量的组合。表 12-2 中列出了基本样式元素。

表 12-2　WxPython 窗口部件中的样式元素

样 式 元 素	说　明
CAPTION	具有标题栏
CLOSE_BOX	具有关闭按钮
DEFAULT_FRAME_STYLE	默认窗口样式
FRAME_SHAPED	可以用来创建一个特定形状的窗口
FRAME_TOOL_WINDOW	工具框窗口
MAXIMIZE_BOX	最大化按钮
MINIMIZE_BOX	最小化按钮
RESIZE_BORDER	可以改变窗口大小
SIMPLE_BORDER	不含有窗口边框
SYSTEM_MENU	具有系统菜单

实际上，表 12-2 中有的样式元素是其他样式元素的组合。例如 DEFAULT_FRAME_STYLE 就是下面的样式元素的组合：

```
    wx.MAXIMIZE_BOX | wx.MINIMIZE_BOX | wx.CLOSE_BOX | wx.RESIZE_BORDER | wx.SYSTEM_MENU |
wx.CAPTION
```

也就是说，默认的样式包含最大化按钮、最小化按钮、关闭按钮、标题栏、系统菜单和可变大小的边框。如果需要显示一个不能改变大小的窗口，可以使用下面的样式组合：

```
wx.DEFAULT_FRAME_STYLE ^ (wx.RESIZE_BORDER | wx.MINIMIZE_BOX |wx.MAXIMIZE_BOX)
```

其显示效果如图 12-3 所示。

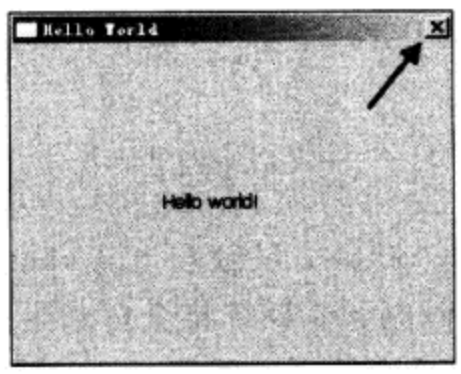


图 12-3　不能改变大小的 WxPython 窗口

实际上还有更多的窗口样式，如有的窗口还会显示帮助按钮等。

12.1.3　WxPython 应用程序的运行过程

通过前面的介绍，可以知道一个 WxPython 应用程序包含一个应用程序对象和一个根窗口。应用程序对象将会处理窗口中的事件响应。从【例 12-2】的输出内容中可以看出 WxPython 应用程序的具体运行步骤。

对于一个 WxPython 的应用程序脚本而言，在生成 WxPython 的应用程序对象之前，和其他的 Python 脚本文件是一样的。在应用程序对象结束之后，也是具有着同样的表现。这也是为什么在【例 12-2】最后的代码段中前后两个输出结果都是在系统终端中的原因。

更多资源请访问稀酷客(www.ckook.com)

在生成 MyApp 类对象的时候，将会首先调用其类中定义的 OnInit 函数。从这里开始，输出的内容将在 WxPython 的调试信息窗口中显示。在调用 OnInit 函数的时候，首先会实例化 MyFrame 类对象，所以这时就会显示在 MyFrame 类的构造函数中的打印信息。通过这样的步骤，WxPython 系统中就生成了一个应用程序对象，并且有了一个根窗口。

随后，调用应用程序对象的 MainLoop 方法，进入事件循环中。注意到这里主片段中的输出信息也显示在了 WxPython 的调试信息窗口中。在此之后，WxPython 系统将处理来自于窗口的事件信息，并调用相应的事件处理程序进行处理。

在关闭所有的 WxPython 窗口之后，OnExit 函数将会被调用。由于这里也有输出信息，所以会重新出现一个调试信息窗口来显示输出。这在程序实际运行中可能会非常短暂而被忽视。另外，还可以调用如 Exit 等函数来退出，但是并不推荐使用这种方法。因为这个时候会跳过程序关闭时的清理函数。

12.2　WxPython 中的事件驱动编程

WxPython 应用采用事件驱动的方式来实现用户界面和 WxPython 系统的交互。通过这种处理方式，系统可以自定义各种事件的处理函数，从而完成相应的功能。在本节中，将对 WxPython 的事件驱动模型及其使用方法进行介绍。

12.2.1　事件驱动的编程模型

事件驱动是 WxPython 应用程序的基本处理机制。这种处理方式通过响应用户对界面的操作事件来完成交互式处理功能。实际上，大部分 GUI 工具库都是采用事件驱动或者信号驱动方式，只不过略有不同。

GUI 中的事件是指用户对界面的各种操作。例如鼠标的移动和单击、键盘按键的按下、窗口大小的改变等，都是在 WxPython 中定义的事件。除了在系统中定义的事件之外，用户还可以自定义事件，从而满足不同的需求。在事件驱动模型中，当发生事件的时候，WxPython 应用程序对象将调用此事件绑定的处理函数。这个调用的过程被称为是事件处理过程。

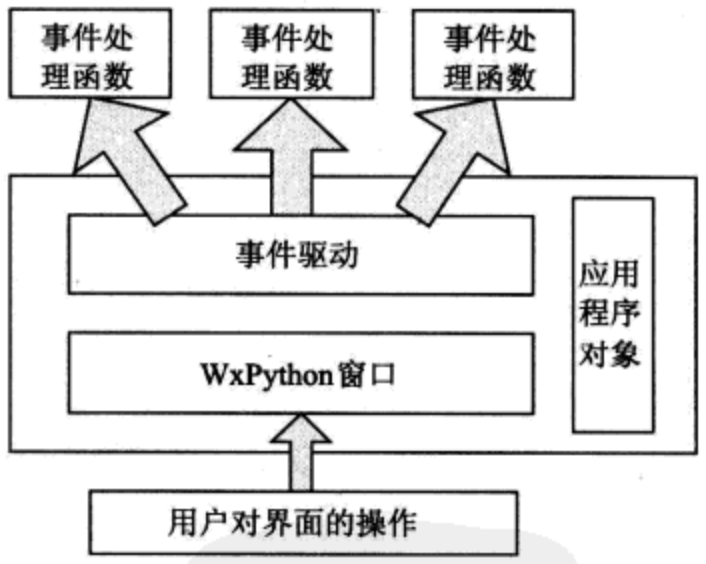


图 12-4　WxPython 的事件驱动模型

一般情况下，对于触发的事件，WxPython 应用程序将会尽快的调用绑定的处理函数来进行处理。但是在某些情况下，如线程之间的通信，并不一定需要立刻处理事件。此时，WxPython 的一种做法是将事件加入到事件队列中，并且在合适的时候触发而调用。一个事件驱动模型如图 12-4 所示。

从图 12-4 中可以看到，当 WxPython 程序调用了 MainLoop 方法后，系统就进入了事件驱动的处理过程中。默认情况下，事件循环过程什么事情也不会做，保持空闲状态。但是当有事件发生的时候，系统将调用相应的事件处理函数。

12.2.2　事件的触发

在 WxPython 应用程序中，一个事件是 wx. Event 类或者其子类的一个实例。如 wx. MouseEvent 表示此事件是使用鼠标的系统事件。通过一个事件的子类是无法区分更细的事件的。一种做法是继续从事件子类中继承而得到更小的事件子类。但是 WxPython 采用了另外一种

更多资源请访问稀酷客(www.ckook.com)

做法，即事件类型。通过事件类型可以识别在同一个事件类下的不同事件类型。如同样属于 wx. MouseEvent 类，可以通过事件类型的值来区分用户是移动了鼠标还是进行了鼠标单击。

实际上，为了能够支持丰富的用户操作和窗口响应，WxPython 中提供了大量的事件类型来满足需要。这些事件类型的名字都是全局性的。

【例 12-3】列出在 WxPython 中所支持的事件类型。

```
In [1]: import wx

In [2]: wx.EVT_【按下 Tab 键】
wx.EVT_ACTIVATE                          wx.EVT_MOUSE_EVENTS
wx.EVT_ACTIVATE_APP                      wx.EVT_MOVE
wx.EVT_BUTTON                            wx.EVT_MOVING
wx.EVT_CALCULATE_LAYOUT                  wx.EVT_NAVIGATION_KEY
wx.EVT_CHAR                              wx.EVT_NC_PAINT
wx.EVT_CHAR_HOOK                         wx.EVT_NOTEBOOK_PAGE_CHANGED
wx.EVT_CHECKBOX                          wx.EVT_NOTEBOOK_PAGE_CHANGING
wx.EVT_CHECKLISTBOX                      wx.EVT_PAINT
wx.EVT_CHILD_FOCUS                       wx.EVT_PALETTE_CHANGED
wx.EVT_CHOICE                            wx.EVT_POWER_RESUME
wx.EVT_CHOICEBOOK_PAGE_CHANGED           wx.EVT_POWER_SUSPENDED
wx.EVT_CHOICEBOOK_PAGE_CHANGING          wx.EVT_POWER_SUSPENDING
wx.EVT_CLOSE                             wx.EVT_POWER_SUSPEND_CANCEL
wx.EVT_COLLAPSIBLEPANE_CHANGED           wx.EVT_QUERY_END_SESSION
wx.EVT_COLOURPICKER_CHANGED              wx.EVT_QUERY_LAYOUT_INFO
wx.EVT_COMBOBOX                          wx.EVT_QUERY_NEW_PALETTE
wx.EVT_COMMAND                           wx.EVT_RADIOBOX
#省略部分代码
wx.EVT_MIDDLE_DOWN                       wx.EVT_UPDATE_UI
wx.EVT_MIDDLE_UP                         wx.EVT_UPDATE_UI_RANGE
wx.EVT_MOTION                            wx.EVT_VLBOX
wx.EVT_MOUSEWHEEL                        wx.EVT_WINDOW_CREATE
wx.EVT_MOUSE_CAPTURE_CHANGED             wx.EVT_WINDOW_DESTROY
wx.EVT_MOUSE_CAPTURE_LOST
```

- 在 In[1]中导入了 wx 包。接着，在 In[2]中输入“wx. EVT_”并按下 Tab 键后，就可以看到在当前 WxPython 系统中定义的事件类型。因为篇幅原因，这里省略了大量的事件类型输出。实际上，WxPython 提供了上百个事件类型。正是这么多的事件类型存在，使得开发者可以控制界面的各种响应。

另外，对于一个特定的事件类，都有一个事件类型集合与其相关联。例如对于 wx. MouseEvent 事件类，有 14 个事件类型与其相关联；对于 wx. CommandEvent 事件类，有 28 个不同的事件类型与其相关联。不同的窗口部件响应不同的事件，而不会发生冲突。

在众多的事件类型中，其中比较重要的类型是鼠标的事件。表 12-3 列出了 wx. MouseEvent 事件类所支持的事件类型。

表 12-3　wx. MouseEvent 事件类支持的事件类型

事件类型	说　明
EVT_LEFT_DOWN	鼠标左键按下
EVT_LEFT_UP	鼠标左键放开
EVT_LEFT_DCLICK	鼠标左键双击
EVT_RIGHT_DOWN	鼠标右键按下
EVT_RIGHT_UP	鼠标右键放开
EVT_RIGHT_DCLICK	鼠标右键双击

更多资源请访问稀酷客(www.ckook.com)

（续）

事件类型	说　明
EVT_MIDDLE_DOWN	鼠标中键按下
EVT_MIDDLE_UP	鼠标中键放开
EVT_MIDDLE_DCLICK	鼠标中键双击
EVT_MOTION	鼠标移动
ENTER_WINDOW	鼠标进入
LEAVE_WINDOW	鼠标离开
EVT_MOUSEWHEEL	鼠标滚轮事件
EVT_MOUSE_EVENTS	一次绑定所有鼠标事件

从表 12-3 中可以看出，对于鼠标事件，WxPython 给出了非常详尽的事件类型。不但包括鼠标的单击事件，还有鼠标的移动变化等。

12.2.3　事件的绑定和处理

通过使用 wx. EvtHandler 类中的 Bind 方法可以将绑定事件的类型、事件对象和事件处理函数关联起来。下面为此方法的函数定义：

```
Bind(event, handler, source = None, id = wx.ID_ANY, id2 = wx.ID_ANY)
```

一般使用的是前面三个参数。其中，第一个参数为事件的类型，第二个参数为绑定此事件的处理函数。第三个参数默认为 None，表示为当前的 Frame。当然，也可以将此事件绑定在特定的 WxPython 的窗口部件中。

【例 12-4】演示事件的绑定和处理过程。

```
#filename: wx_4.py
#encoding = utf - 8
import wx

class MyFrame(wx.Frame):
    def __init__(self, parent):

        wx.Frame.__init__(self, parent,-1, 'Hello World', size = (300, 300))

        panel = wx.Panel(self)
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)

        txt = wx.StaticText(panel, -1, 'Hello world! ')
        sizer.Add(txt, 0, wx.TOP|wx.LEFT, 100)

        button = wx.Button(panel, -1, "Quit")
        sizer.Add(button, 0, wx.TOP|wx.LEFT, 100)
        self.Bind(wx.EVT_BUTTON, self.OnClick, button) #绑定事件

        self.Centre()

    def OnClick(self, event): #自定义事件处理函数
        self.Close(True)

class MyApp(wx.App):
#省略 MyApp 类的代码实现

if __name__ == '__main__':
    app = MyApp()
    app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

- 这个代码段和【例 12-2】的区别在于增加了事件处理部分。在 MyFrame 的构造函数中，加入了一个命令按钮。
- 在代码片段中使用了 Bind 方法将 EVT_BUTTON 事件绑定在 OnClick 函数上，界面响应的部分则是刚刚创建的命令按钮。
- 接着，在 MyFrame 类中还定义了一个 OnClick 函数。在函数实现中调用了 Close 方法来关闭窗口，实际上也同时退出了事件循环。

运行【例 12-4】代码，可以得到如图 12-5 所示的窗口。

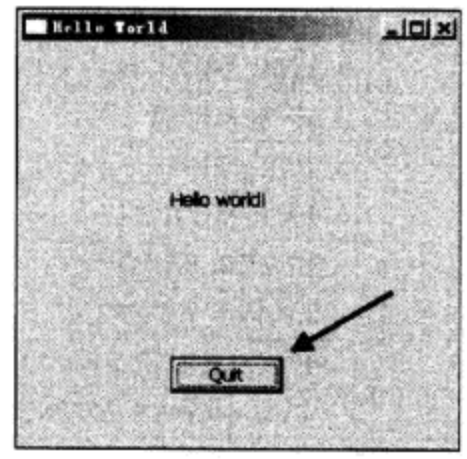


图 12-5　WxPython 应用中的事件驱动

另外，可以对同一个窗口部件绑定多个处理函数。

【例 12-5】为命令按钮绑定多个事件，包括鼠标指针的进入和离开。

```
#省略部分代码
class MyFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent,-1, 'Hello World', size = (300, 300))
#省略部分代码
        self.button = wx.Button(panel, -1, u"退出",size = (100,30))
        sizer.Add(self.button, 0, wx.TOP|wx.LEFT, 90)
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button) #绑定事件
        self.button.Bind(wx.EVT_ENTER_WINDOW, self.OnEnterWindow) #绑定鼠标进入事件
        self.button.Bind(wx.EVT_LEAVE_WINDOW, self.OnLeaveWindow) #绑定鼠标离开事件

    def OnClick(self, event): #自定义鼠标单击处理函数
        self.Close(True)

    def OnEnterWindow(self, event): #自定义鼠标进入处理函数
        self.button.SetLabel(u"退出(鼠标进入)")
        event.Skip()

    def OnLeaveWindow(self, event): #自定义鼠标离开处理函数
        self.button.SetLabel(u"退出(鼠标离开)")
        event.Skip()
#省略部分代码
```

- 【例 12-5】为命令按钮增加了两个事件处理，包括鼠标指针进入和鼠标指针离开的事件处理。在具体的处理函数中，分别将命令按钮的标签设置为不同的值。
- 注意到在这里的事件处理函数中，最后使用了 event 的 Skip 函数。这是因为一般情况下窗口部件对象在处理一个事件完毕后，该事件处理就完结了。如果需要继续处理的话，则可以使用 Skip 方法来改变默认的行为。

运行【例 12-5】代码，程序在鼠标指针进入和鼠标指针离开的时候分别如图 12-6 和图 12-7 所示。

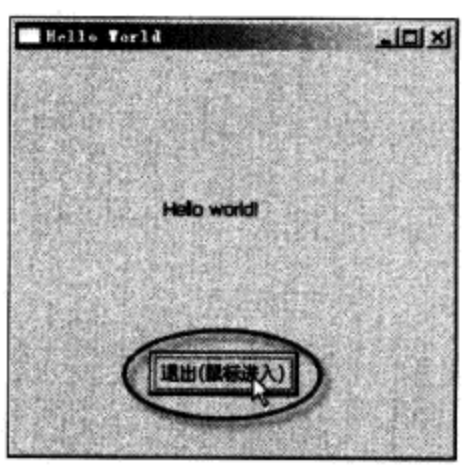


图 12-6　鼠标指针进入时的事件处理

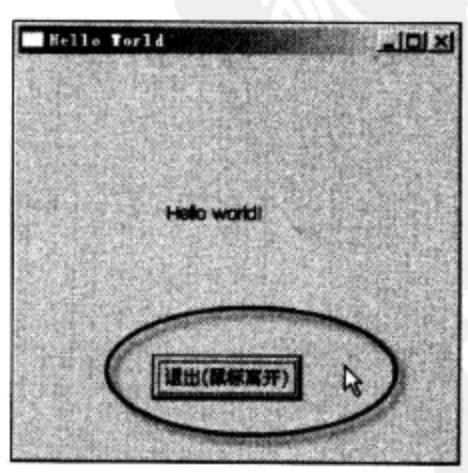


图 12-7　鼠标指针离开时的事件处理

更多资源请访问稀酷客(www.ckook.com)

对于前面介绍的大部分事件类型，实际上很多都有默认的处理函数。一般为不进行任何操作。开发者可以通过设置各种事件的处理函数来实现对窗口部件行为的精细控制。

12.3　WxPython 的常用组件

在 WxPython 的应用程序中，有些组件是会经常见到的。这些常用的组件包括对话框、工具栏和状态栏等。本节将对这些常见组件进行介绍。

12.3.1　对话框

对话框为 WxPython 应用程序提供了与用户交互的信息。在 WxPython 中，已经内置了不少对话框。简单对话框包括消息对话框和文本输入对话框。除了这些简单的对话框之外，在 WxPython 中还有更多的对话框。这将在第 14 章进行介绍，包括对话框的实现。

1. 消息对话框

在消息对话框中可以显示一些简单的信息，并提供用户确认或取消。如在删除文件时的确认对话框等。WxPython 提供了 MessageDialog 类来实现消息对话框。在此类的构造函数中，可以通过设置参数来改变对话框的信息和样式。下面的代码演示了一个确认对话框：

```
dialog = wx.MessageDialog(None,u'确认删除吗？',
                          u'文件删除', wx.YES_NO | wx.ICON_QUESTION)
result = dialog.ShowModal()
dialog.Destroy()
```

第 1 行使用 MessageDialog 类构造了一个消息对话框。这里设置了此消息对话框的显示信息以及标题栏信息。接着调用了 ShowModal 方法，这将使得必须先处理这个对话框。在这里，也保证了在处理确认之前不能做更多的工作。此方法将返回一个值，可以通过此值来判断用户是选择了什么选项，从而进一步进行处理。运行这段代码，得到如图 12-8 所示的对话框。

2. 文本输入对话框

文本输入对话框可以通过使用 TextEntryDialog 来得到。除了提示信息之外，还可以在构造函数中设置此文本输入对话框的默认值。下面的示例代码中演示了文本输入对话框的用法：

```
dialog = wx.TextEntryDialog(None, u"请输入文件名:",
        u'文件名输入', 'test.txt')
if dialog.ShowModal() == wx.ID_OK:
    response = dialog.GetValue()
dialog.Destroy()
```

运行上面的代码，得到如图 12-9 所示的对话框。

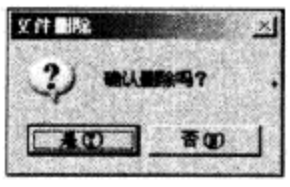


图 12-8　确认对话框

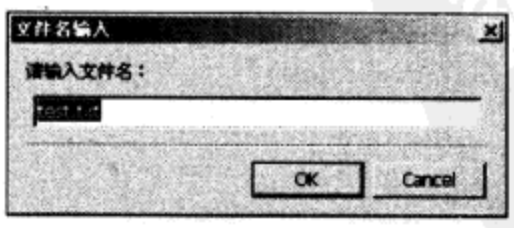


图 12-9　文本输入对话框

12.3.2　工具栏

工具栏提供了一种访问常用命令的方法。在 WxPython 中，可以通过调用 Frame 的 CreateToolBar 方法来生成。

更多资源请访问稀酷客(www.ckook.com)

【例 12-6】演示工具栏的使用方法。

```
#filename: wx_5.py
#encoding = utf-8
import wx

class MyFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, u"简单工具栏", size = (300, 200))

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(wx.ID_EXIT, '', wx.Bitmap('./exit.png'))
        toolbar.Realize()

        self.Bind(wx.EVT_TOOL, self.OnExit, id = wx.ID_EXIT)

        self.Centre()
    def OnExit(self, event):
        self.Close()

app = wx.App()
frame = MyFrame(None)
frame.Show(True)
app.MainLoop()
```

- 在这个根窗口中仅仅有一个工具栏。此工具栏是通过使用 CreateToolBar 方法创建的。为了在工具栏上添加新的按钮，可以使用此工具栏对象的 AddLabelTool 方法，在此方法中指定了用于显示的图片。
- 接着使用 Bind 方法将此工具栏按钮和特定的处理函数绑定。这里，当单击此工具栏按钮的时候，就会执行 OnExit 函数。

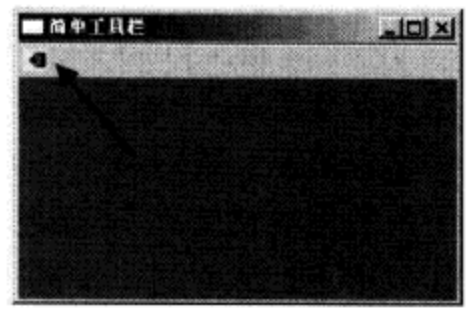


图 12-10　含有工具栏的窗口

运行【例 12-6】代码，显示如图 12-10 所示窗口。当单击此窗口中的工具栏按钮的时候，此窗口将会关闭。这也意味着程序退出。

12.3.3　状态栏

另外一个常用的组件是状态栏。此组件可以用来显示操作的信息等。在 WxPython 中，可以通过使用 CreateStatusBar 方法来创建状态栏。

【例 12-7】演示状态栏的使用方法。

```
#filename: wx_6.py
#encoding = utf-8
import wx

class MyFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, u"简单工具栏和状态栏", size = (300, 200))
#省略部分代码
        self.CreateStatusBar()
        self.SetStatusText(u"状态栏信息")
#省略部分代码

app = wx.App()
frame = MyFrame(None)
frame.Show(True)
app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

这段代码中通过使用 CreateStatusBar 加入了一个状态栏。随后调用了 SetStatusText 方法来设置状态栏的信息。这个信息可以根据实际操作而更改。运行这段代码，可以得到如图 12-11 所示的窗口。

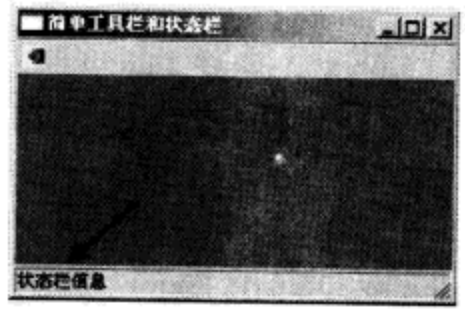


图 12-11　含有状态栏的窗口

12.4　小结

本章对使用 WxPython 来开发用户界面程序进行了初步介绍。首先介绍了一个 WxPython 应用程序所包含的部分：应用程序对象和至少一个根窗口。这两者通过事件驱动关联起来。接下来对 WxPython 中的事件驱动进行了详细的介绍，包括其模型以及具体的使用方法。随后对 WxPython中的几个常用组件进行了讲解，包括简单对话框、工具栏和状态栏等。

更多资源请访问稀酷客(www.ckook.com)

第 13 章　WxPython 库中的基本组件

本章介绍了 WxPython 库中基本组件的使用方法。WxPython 的基本组件包括文本框、按钮、单选框、复选框、列表等，讲解了基本组件的分类、常用属性、方法和事件。WxPython 的列表控件非常丰富，可以定义列表框和下拉列表框，也可以创建可编辑的下拉列表组件。布局管理器统一管理界面中的组件，为组件的排列布局提供了简单的方式，而不用程序设计者显式指定各个组件的排列。WxPython 提供了 Grid、Flex、Box 等布局管理器，根据不同的场合使用相应的布局管理器可以美化应用程序。本章的知识点：

- 基本组件的常用属性和事件
- 各种列表控件的定义和使用方法
- 使用 Sizer 布局管理器的方法

13.1　静态文本框

WxPython 库中的 StaticText 类表示静态文本框，使用 StaticText 类可以设置文本框的大小、颜色、对齐方式等属性。StaticText 类在 wx 包中，使用前需要先导入 wx 包。静态文本框控件的样式可以通过构造函数的参数控制，静态文本框的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String label = EmptyString,
Point pos = DefaultPosition, Size size = DefaultSize,long style = 0,
String name = StaticTextNameStr) -> StaticText
```

- 参数 parent 表示静态文本框控件的父容器。
- 参数 id 用于标识当前的静态文本框控件。
- 参数 label 表示静态文本框中输入的字符串。
- 参数 pos 表示静态文本框左上角的坐标。
- 参数 size 是一个有两个元素的元组，其中的两个元素分别表示静态文本框的长度和宽度。
- 参数 style 表示静态文本框中字符串的对齐方式。
- 参数 name 表示当前控件的名称，可用于控件的查找。

【例 13-1】演示静态文本框基本样式的设置和创建方法。

```
import wx

class TextFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, 'Text', size = (100, 75))
        panel = wx.Panel(self, -1)
        wx.StaticText(panel, -1, "Hello world!", (10, 10), (80, 15))          #创建静态文本框

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
    app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 3 行代码定义了 Frame 类的子类 TextFrame，TextFrame 类用于展示静态文本框控件。
- 第 7 行代码调用 StaticText 的构造函数。文本的内容为“Hello world!”，文本的左上角坐标为（10，10），文本的长度为“80”，宽度为“15”。

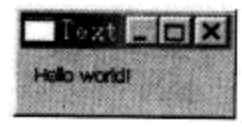


图 13-1　静态文本框控件

【例 13-1】运行后的效果如图 13-1 所示。

【例 13-2】演示 WxPython 的静态文本框可以设置文字的前景色和背景色，以及文字在静态文本框中的对齐方式。

```
import wx

class TextFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Text", size = (100, 75))
        panel = wx.Panel(self, -1)
        text = wx.StaticText(panel, -1, " Hello world!", (10, 10), (80, 15), wx.ALIGN_CENTER)
        text.SetForegroundColour('blue')                                    # 设置前景色
        text.SetBackgroundColour('white')                                   # 设置背景色

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
    app.MainLoop()
```

- 第 7 行代码对 StaticText 类进行实例化，构造函数中增加了对齐方式，wx. ALIGN_CENTER 表示居中对齐。此外，wx. ALIGN_ LEFT 表示左对齐，wx. ALIGN_RIGHT 表示右对齐。
- 第 8 行代码设置前景色，即静态文本框中文字的颜色。
- 第 9 行代码设置背景色，即静态文本框的颜色。



图 13-2　设置静态文本框的颜色

【例 13-2】运行后的效果如图 13-2 所示。

通过 Font 类可以设置静态文本框中文字的字体。Font 类的构造函数如下所示：

```
__init__(self, int pointSize, int family, int style, int weight,
    bool underline = False,String face = EmptyString,
    int encoding = FONTENCODING_DEFAULT) -> Font
```

- 参数 pointSize 用于设置文字的字号。
- 参数 family 用于设置文字的字体类型。
- 参数 style 设置文字的样式，如斜体、粗体等。
- 参数 weight 设置文字的大小。
- 参数 underline 表示是否在文字下方输出下划线，该参数的默认值为不输出下划线。
- 参数 face 表示文字的外观，该参数为可选参数。
- 参数 encoding 设置文字的编码。

【例 13-3】演示静态文本框中字体的设置方法。

更多资源请访问稀酷客(www.ckook.com)

```
import wx

class TextFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None,, -1, 'Text', size = (100, 75))
        panel = wx.Panel(self, -1)
        text = wx.StaticText(panel, -1, "Hello world!", (10, 10), (80, 15))
        font = wx.Font(12, wx.DEFAULT, wx.ITALIC, wx.NORMAL, True)          # 创建字体类型
        text.SetFont(font)                                                  # 设置字体

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码创建 Font 对象。
- 第 9 行代码调用 text 对象的 SetFont( )，给文本框 test 设置 font 对象指定的字体。

【例 13-3】运行后的效果如图 13-3 所示。



图 13-3　静态文本框中的字体

13.2　文本框

文本框控件是应用程序中最常见的一种控件，文本框控件分为单行文本框和多行文本框。

13.2.1　单行文本框

wxPython 提供了 TextCtrl 类创建文本框，TextCtrl 类的构造函数的参数与 StaticText 类似。TextCtrl 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String value = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = 0, Validator validator = DefaultValidator,
        String name = TextCtrlNameStr) -> TextCtrl
```

TextCtrl 类比 StaticText 多提供了 validator 参数，该参数是 wx. Validator 类型。Validator 参数用于过滤数据，使文本框只能输入规定类型的数据。

参数 style 可以定制文本框的对齐方式、显示方式以及文本框的输入方式。表 13-1 列出了 TextCtrl 控件可以使用的样式。

表 13-1　文本框的样式

样　式	说　明
wx. TE_CENTER	控件中的字符串居中显示
wx. TE_LEFT	控件中的字符串左对齐
wx. TE_RIGHT	控件中的字符串右对齐
wx. TE_NOHIDESEL	文本高亮显示
wx. TE_PASSWORD	密码输入框，隐藏输入的文字
wx. TE_PROCESS_ENTER	在控件内按下回车键时，触发相关事件
wx. TE_PROCESS_TAB	当按下 Tab 键切换控件后，触发相关事件
wx. TE_READONLY	设置只读状态的文本控件
wx. TE_MULTILINE	设置多行文本框

更多资源请访问稀酷客(www.ckook.com)

（续）

样　式	说　明
wx. TE_RICH	文本框内的文字可以使用字体、颜色、样式
wx. TE_RICH2	最新版的 TE_RICH
wx. TE_AUTO_URL	如果 TE_RICH 或 TE_RICH2 中存在 URL，该样式可以响应 URL 的单击
wx. TE_LINEWRAP	如果行太长，将以字符为界换行
wx. TE_WORDWRAP	如果行太长，将以单词为界换行

注意　默认情况下，ENTER 键事件和 TAB 键事件由对话框管理。如果要使 TextCtrl 控件响应 ENTER 键事件或 TAB 键事件，需要在 style 中定义相关的样式，然后再绑定事件并编写事件处理函数。

【例 13-4】创建 1 个用户名文本框和 1 个密码文本框。输入用户名和密码后，在密码输入框按回车键，将弹出对话框显示用户名和密码。

```
import wx

class TextCtrlFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'文本框', size=(300, 150))
        panel = wx.Panel(self, -1)
        Label1 = wx.StaticText(panel, -1, u"姓名:", pos = (10, 10))
        self.inputText = wx.TextCtrl(panel, -1, "", pos = (80,10), size=(150, -1))
        # 输入文本框
        self.inputText.SetInsertionPoint(0)                                    # 设置焦点
        Label2 = wx.StaticText(panel, -1, u"密码:", pos = (10, 50))
        # 密码输入框
        self.pwdText = wx.TextCtrl(panel, -1, "", pos = (80,50), size=(150, -1), style =
wx.TE_PASSWORD | wx.TE_PROCESS_ENTER)
        self.Bind(wx.EVT_TEXT_ENTER, self.OnLostFocus, self.pwdText)

    def OnLostFocus(self, evt):
        wx.MessageBox('%s, %s' % (self.inputText.GetValue(), self.pwdText.GetValue()), 'hint')

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextCtrlFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码创建文本框对象 inputText。
- 第 9 行代码，在 inputText 中设置光标。
- 第 12 行代码创建密码文本框，构造函数的 style 参数设置为 wx. TE_PASSWORD。为了处理文本框的回车事件，同时给 sytle 参数设置 wx. TE_PROCESS_ENTER。
- 第 14 行代码绑定 wx. EVT_TEXT_ENTER 事件给 pwdText 控件，OnLostFocus( ) 为事件处理函数，当按下 ENTER 键后会触发 OnLostFocus( ) 函数。
- 第 16 行代码弹出 MessageBox 对话框，显示用户名和密码。

【例 13-4】运行后的效果如图 13-4 所示。

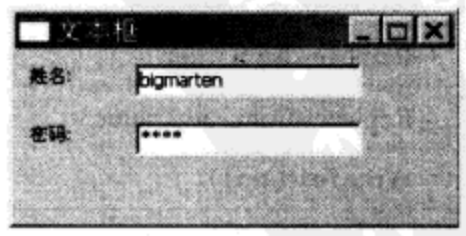


图 13-4　单行文本框和密码文本框

更多资源请访问稀酷客(www.ckook.com)

13.2.2　多行文本框

6 当需要在文本框中输入大量文本时，就需要使用多行文本框控件。通过改变 TextCtrl 控件的样式就可以生成多行文本框控件：

```
import wx

class TextFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'多行文本框', size=(250, 150))
        panel = wx.Panel(self, -1)
        # 创建多行文本框
        multiText = wx.TextCtrl(panel, -1,
                "Python is a good language."
                "wxPython is a GUI API."
                "good job!",
                pos = (10, 10), size = (180, 80), style=wx.TE_MULTILINE | wx.TE_CENTER)
        multiText.SetBackgroundColour("red")
        multiText.SetFocus()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = TextFrame()
    frame.Show()
app.MainLoop()
```

- 第 8 行代码调用 TextCtrl 类创建多行文本框对象 multiText，sytle 参数设置为 wx.TE_MULTILINE，表示多行文本框。同时设置 wx.TE_CENTER，使文本居中显示。
- 第 13 行代码设置 multiText 的背景颜色。
- 第 14 行代码把焦点设置到 multiText 控件。

本段代码运行后的效果如图 13-5 所示。



图 13-5　多行文本框

上面这段代码使用了 TextCtrl 控件的 SetBackgroundColour( )、SetFocus( ) 方法。TextCtrl 控件还提供了许多有用的方法，如表 13-2 所示。

表 13-2　文本框的常用方法

方　法	说　明
AppendText(text)	在控件文本的尾部添加 text 字符串
Clear( )	重置控件中的文本
GetInsertionPoint( )	获取控件中插入点的位置
SetInsertionPoint(pos)	设置控件中插入点的位置
GetRange(from, to)	获取控件中文本从 from 索引到 to 索引的子串
GetValue( )	获取控件中的文本字符串
SetValue(value)	设置控件中的文本字符串
Remove(from, to)	删除从 from 到 to 范围内的字符串
Replace(from, to, value)	使用 value 替换从 from 到 to 范围内的字符串
WriteText(text)	在光标处写入字符串 text
GetLineLength(lineNo)	获取 lineNo 行的长度
GetLineText(lineNo)	获取 lineNo 行的文本

更多资源请访问稀酷客(www.ckook.com)

（续）

方　法	说　明
GetNumberOfLines( )	获取控件中的行数
PositionToXY(pos)	光标所在处的行、列位置，返回行、列组成的元组
XYToPosition(x, y)	根据指定的行、列返回光标所在处的整型位置
IsMultiLine( )	当前控件是否为多行文本框
IsSingleLine( )	当前控件是否为单行文本框

13.3　按钮控件

按钮控件是应用程序中最常用的控件之一。本节将介绍按钮控件的创建以及位图控件的使用方法。

13.3.1　普通按钮

Button 类可以创建按钮控件，按钮控件的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String label = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = 0, Validator validator = DefaultValidator,
        String name = ButtonNameStr) -> Button
```

按钮控件的参数含义与 TextCtrl 控件类似。

【例 13-5】创建 1 个按钮和 1 个文本框，单击按钮后，文本框将显示字符串“Hello world!”（处理单击事件需要添加 wx. EVT_BUTTON 选项，并定义单击事件的处理函数）。

```
1  import wx
2
3  class ButtonFrame(wx.Frame):
4      def __init__(self):
5          wx.Frame.__init__(self, None, -1, u'按钮', size = (300, 75))
6          panel = wx.Panel(self, -1)
7          self.button = wx.Button(panel, -1, u"确定", pos = (10, 10))          # 创建按钮
8          self.Bind(wx.EVT_BUTTON, self.OnClick, self.button)
9          self.button.SetDefault()                                            # 设置默认按钮
10         self.inputText = wx.TextCtrl(panel, -1, "", pos = (100,10), size = (150, -1), style =
wx.TE_READONLY)
11
12     def OnClick(self, event):
13         self.inputText.Value = "Hello world!"
14
15 if __name__ == '__main__':
16     app = wx.PySimpleApp()
17     frame = ButtonFrame()
18     frame.Show()
19     app.MainLoop()
```

- 第 7 行代码调用 Button 类的构造函数创建 button 对象。
- 第 8 行代码设置 button 对象的单击事件，该事件的处理函数为 OnClick( )。
- 第 9 行代码把 button 按钮设置为默认按钮。
- 第 10 行代码创建了只读的文本控件 inputText。
- 第 12 行代码编写函数 OnClick( ) 的处理代码，单击【确定】按钮后，将向 inputText 文本框中输入“Hello world!”。

更多资源请访问稀酷客(www.ckook.com)

【例 13-5】运行后的效果如图 13-6 所示。

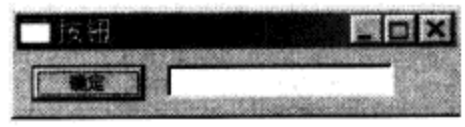


图 13-6　带有按钮的窗口

13.3.2　位图按钮

wxPython 提供了专门的位图控件 BitmapButton 用来创建位图按钮。BitmapButton 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, Bitmap bitmap = wxNullBitmap,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = BU_AUTODRAW, Validator validator = DefaultValidator,
        String name = ButtonNameStr) -> BitmapButton
```

创建位图按钮之前需要定义 Image 类型的对象，并把该对象转换为 Bitmap 类型。然后再创建 BitmapButton 类型的按钮，并关联 Bitmap 的实例化对象。

【例 13-6】使用 PNG 格式的图片创建位图按钮。

```
import wx

class BitmapButtonFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'位图按钮', size = (200, 150))
        panel = wx.Panel(self, -1)
        bmp = wx.Image("button.png", wx.BITMAP_TYPE_PNG).ConvertToBitmap()
        self.button = wx.BitmapButton(panel, -1, bmp, pos = (20, 20), size = (120, 60))
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button)
        self.button.SetDefault()

    def OnClick(self, event):
        wx.MessageBox("Hello world!", u"提示")

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = BitmapButtonFrame()
    frame.Show()
app.MainLoop()
```

- 第 7 行代码创建了 Image 对象，wx.BITMAP_TYPE_PNG 表示图片类型为 PNG 格式。ConvertToBitmap( )把 Iamge 对象转换为 Bitmap 类型的对象。
- 第 8 行代码创建 BitmapButton 类型的按钮，并关联前面创建的 bmp 对象。
- 第 9 行代码设置单击事件，处理函数为 OnClick( )。
- 第 12 行代码编写函数 OnClick( )的处理代码，单击【确定】按钮后，将弹出对话框。对话框的内容为“Hello world!”，标题为“提示”。

【例 13-6】运行后的效果如图 13-7 所示。

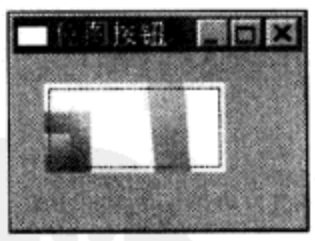


图 13-7　位图按钮

13.4　单选框

在应用程序中，有时需要在一组选项中进行选择，这种应用需要创建单选框控件。当出现多组选项时，通常把同一类选项作为一组使用，这种控件称为单选框分组。本节将介绍单选框的单击事件以及使用循环注册多个事件的方法。

13.4.1　单选框的选择

单选框控件通常成组出现，用户每次操作只能选择其中 1 个单选框控件。RadioButton 类用于构造单选框控件，单选框按钮的构造函数如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
__init__(self, Window parent, int id = -1, String label = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = 0, Validator validator = DefaultValidator,
        String name = RadioButtonNameStr) -> RadioButton
```

RadioButton 类每次只创建 1 个单选框控件。在 wxPython 中还可以使用 RadioBox 类创建单选框，RadioBox 类可以创建一组 RadioButton 类，这些单选框控件将显示在一个矩形中。下一节将介绍 RadioBox 类的使用方法。

【例 13-7】创建 3 个 RadioButton 控件和 1 个 TextCtrl 控件。3 个 RadioButton 控件分别代表“红”、“蓝”、“绿”。当单击某个 RadioButton 控件时，TextCtrl 控件的背景色将改变为对应的颜色。

```
import wx

class RadioFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'单选框', size = (200, 150))
        panel = wx.Panel(self, -1)
        radioRed = wx.RadioButton(panel, -1, u"红", pos = (10, 10))
        radioBlue = wx.RadioButton(panel, -1, u"蓝", pos = (10, 40))
        radioGreen = wx.RadioButton(panel, -1, u"绿", pos = (10, 70))
        self.colors = {"红": wx.RED, "蓝": wx.BLUE, "绿": wx.GREEN}      # 建立颜色和 wx 常量的对应关系
        self.textColor = wx.TextCtrl(panel, -1, "", pos = (80, 10))
        for eachRadio in [radioRed, radioBlue, radioGreen]:              # 注册 Radio 事件
            self.Bind(wx.EVT_RADIOBUTTON, self.OnRadio, eachRadio)

    def OnRadio(self, event):
        radioSelected = event.GetEventObject()                           # 返回选中的 Radio
        str = radioSelected.GetLabel()
        self.textColor.SetBackgroundColour(self.colors[str.encode("UTF-8")])  # 选择 Radio 的背景颜色
        self.textColor.SetFocus()

if __name__ == '__main__':
    app = wx.PySimpleApp()
frame = RadioFrame()
frame.Show()
    app.MainLoop()
```

- 第 7、8、9 行代码分别创建了 3 个 RadioButton 类，并分别设置文本内容为“红”、“蓝”、“绿”。
- 第 10 行代码创建了字典 ccolors，设置了 3 个 RadioButton 控件的文本和颜色常量之间的对应关系。wx. RED 表示红色，wx. BLUE 表示蓝色，wx. GREEN 表示绿色。
- 第 11 行代码创建了文本控件 textColor。
- 第 12 行代码通过 for... in 循环注册 3 个 RadioButton 控件的单击事件，wx. EVT_RADIOBUTTON 表示 RadioButton 控件的单击事件。事件处理函数为 OnRadio( )。
- 第 15 行代码定义了事件处理函数 OnRadio( )，当单击不同的 RadioButton 控件时，文本框的背景会反映不同的颜色。
- 第 16 行代码调用参数 event 的 GetEventObject( )方法，返回当前单击的 RadioButton 对象 radioSelected。
- 第 17 行代码调用 radioSelected 对象的 GetLabel( )方法返回文本值。例如，单击 radioRed 对象返回“红”。
- 第 18 行代码，在自定义字典 colors 中查找 GetLabel( )返回文本所对应的颜色常量，并调用 textColor 的 SetBackgroundColour( )方法设置颜色。

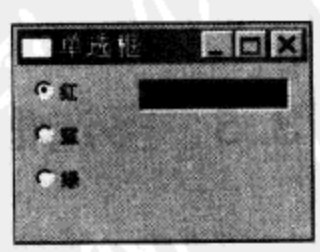


图 13-8　单选框

【例 13-7】运行后的效果如图 13-8 所示。

更多资源请访问稀酷客(www.ckook.com)

注意　本段程序使用了 UTF-8 字符集，字典 colors 使用中文作为 key 值。因此上述第 18 行代码通过 key 值获取字典的 value 值时，需要先调用函数 encode("UTF-8") 对 key 值进行编码。

13.4.2　单选框的分组

当应用程序中出现多组选项时，可以使用 RadioBox 控件进行分组，RadioBox 类可以创建一组 RadioButton。RadioBox 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String label = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        wxArrayString choices = wxPyEmptyStringArray,
        int majorDimension = 0, long style = RA_HORIZONTAL,
        Validator validator = DefaultValidator,
        String name = RadioBoxNameStr) -> RadioBox
```

- 参数 choices 是 1 个字符串列表，表示 RadioBox 中各个 RadioButton 控件的标签名称。
- 参数 majorDimension 是 1 个整型数字，根据参数 style 及此数字来排列 RadioButton 控件。
- 参数 style 设置 RadioButton 控件的排列方式。其中 wx. RA_SPECIFY_COLS 表示按照列进行排列，wx. RA_SPECIFY_ROWS 表示按照行进行排列。

【例 13-8】使用 RadioBox 控件对列表中的文本进行分组，每个文本对应 1 个 Radio 对象。

```
import wx

class RadioBoxFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'单选框', size = (320, 150))
        panel = wx.Panel(self, -1)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        # 生成 RadioBox
        wx.RadioBox(panel, -1, u'颜色', (10, 10), wx.DefaultSize, colorList, 4, wx.RA_SPECIFY_COLS)
        wx.RadioBox(panel, -1, u'颜色', (180, 10), wx.DefaultSize, colorList, 3, wx.RA_SPECIFY_ROWS)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    RadioBoxFrame().Show()
    app.MainLoop()
```

- 第 7 行代码定义了列表 colorList，其中存放了表示 7 种颜色字符串。
- 第 9 行代码定义了 RadioBox 控件，把 Radio 控件排列为 4 列（wx. RA_SPECIFY_COLS 表示按照列进行排列）。
- 第 10 行代码定义了 RadioBox 控件，把 Radio 控件排列为 3 行。wx. RA_SPECIFY_ROWS 表示按照行进行排列。

【例 13-8】运行后的效果如图 13-9 所示。

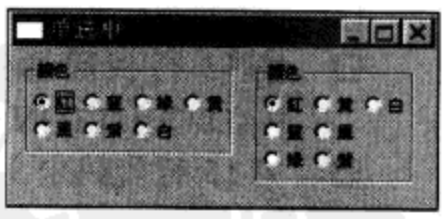


图 13-9　单选框的分组

13.5　多选框

多选框又称复选框，是一个带有文本标签的开关按钮。本节通过“你喜欢的水果”作为调查题目讲述多选框按钮的创建方法和事件响应。

更多资源请访问稀酷客(www.ckook.com)

13.5.1　多选框

多选框通常成组出现，多选框的状态是布尔类型的。用户每次操作可以选择多个多选框控件。CheckBox 类可用于创建多选框控件，CheckBox 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String label = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = 0, Validator validator = DefaultValidator,
        String name = CheckBoxNameStr) -> CheckBox
```

【例 13-9】对“喜欢的水果”进行调查，创建 4 个 CheckBox 多选框，用户可以选择多个选项。

```
import wx

class CheckBoxFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'多选框', size = (120, 180))
        panel = wx.Panel(self, -1)
        wx.StaticText(panel, -1, u"喜欢的水果:", (10, 10), (100, 20))
        wx.CheckBox(panel, -1, u"苹果", (10, 30), (100, 20))
        wx.CheckBox(panel, -1, u"香蕉", (10, 50), (100, 20))
        wx.CheckBox(panel, -1, u"西瓜", (10, 70), (100, 20))
        wx.CheckBox(panel, -1, u"桔子", (10, 90), (100, 20))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    CheckBoxFrame().Show()
    app.MainLoop()
```

❑　第 8 行到第 11 行代码创建了 4 个 CheckBox 控件。

【例 13-9】运行后的效果如图 13-10 所示。



图 13-10　“喜欢的水果”调查

13.5.2　多选框事件

多选框最常用的是 wx.EVT_CHECKBOX 事件。前一节的【例 13-9】创建了调查项目的界面，本节将在【例 13-9】的基础上添加多选框按钮的选择事件。此外添加【全选】复选框和【查看】按钮。单击【全选】，则把所有的多选框都选中。单击【查看】按钮，则查看用户选择的结果。

前面的程序都是直接使用 Bind()方法绑定某个控件的事件处理函数。如果有一组 CheckButton 控件都需要处理 EVT_CHECKBOX 事件，需要写一组 Bind()方法，并分别编写事件处理函数。这种写法十分繁琐，而且代码大量重复。使用 for... in 循环可以解决这个问题，for... in 循环一次可以绑定多个同类控件的事件处理函数。

【例 13-10】接上例，在【例 13-9】的__init__()中添加调查程序中多选框和按钮的单击事件。

更多资源请访问稀酷客(www.ckook.com)

```
import wx

class CheckBoxFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'多选框', size=(120, 200))
        panel = wx.Panel(self, -1)
        wx.StaticText(panel, -1, u"喜欢的水果:", (10, 10), (100, 20))
        self.checkBox1 = wx.CheckBox(panel, -1, u"苹果", (10, 30), (100, 20))
        self.checkBox2 = wx.CheckBox(panel, -1, u"香蕉", (10, 50), (100, 20))
        self.checkBox3 = wx.CheckBox(panel, -1, u"西瓜", (10, 70), (100, 20))
        self.checkBox4 = wx.CheckBox(panel, -1, u"桔子", (10, 90), (100, 20))
        self.allCheckBox = wx.CheckBox(panel, -1, u"全选", (10, 110), (100, 20))
        self.selectFlag = True                                    # 判断全选是否选中
        self.fruit = []                                           # 存放选中的水果
        self.button = wx.Button(panel, -1, u"查看", (10, 140), (100, 20))
        self.Bind(wx.EVT_BUTTON, self.OnClick, self.button)       # 注册 checkbox 的选择事件
        for checkBox in [self.checkBox1, self.checkBox2, self.checkBox3, self.checkBox4]:
            self.Bind(wx.EVT_CHECKBOX, self.OnSelectSingle, checkBox)
        self.Bind(wx.EVT_CHECKBOX, self.OnSelect, self.allCheckBox)
```

- 第 12 行代码创建了标签为【全选】的复选框。
- 第 13 行代码定义了变量 selectFlag，判断【全选】复选框是否选中。
- 第 14 行代码定义了列表 fruit，用于存放选中的水果。
- 第 17 行代码使用 for... in 循环同时注册了 checkBox1、checkBox2、checkBox3、checkBox4 控件的 EVT_CHECKBOX 事件。这 4 个控件的事件处理逻辑相同，都由函数 OnSelectSingle( ) 处理。
- 第 19 行代码注册了【全选】复选框的 EVT_CHECKBOX 事件。由于【全选】复选框的事件处理逻辑与其他 4 个 CheckBox 控件不同，所以需要单独注册。事件处理函数为 OnSelect( )。

【例 13-11】接上例。实现事件处理函数 OnSelectSingle( )。首先获取当前选中的 CheckBox 控件，然后判断 checkBoxSelected 是否选中。如果选中，把该控件的标签名称添加到列表 fruit 中。否则从列表 fruit 中移除该控件的标签名称。最后把变量 selectFlag 置为相反的布尔值。

```
    def OnSelectSingle(self, event):
        checkBoxSelected = event.GetEventObject()
        if checkBoxSelected.IsChecked():
            self.fruit.append(checkBoxSelected.GetLabelText())
        else:
            self.fruit.remove(checkBoxSelected.GetLabelText())
            self.selectFlag = not self.selectFlag
```

【例 13-12】接上例。实现事件处理函数 OnSelect( )。首先清空列表 fruit 中的内容，然后改变 4 个 CheckBox 控件的状态。如果【全选】，则把所有的水果添加到列表 fruit 中，并把其他 4 个 CheckBox 控件置为 Disable；如果【全不选】，则把其他 4 个 CheckBox 控件置为 Enable。最后把变量 selectFlag 置为相反的布尔值。

```
    def OnSelect(self, event):
        self.fruit = []
        for checkBox in [self.checkBox1, self.checkBox2, self.checkBox3, self.checkBox4]:
            checkBox.SetValue(self.selectFlag)
        if self.allCheckBox.IsChecked():
            for checkBox in [self.checkBox1, self.checkBox2, self.checkBox3, self.checkBox4]:
                self.fruit.append(checkBox.GetLabelText())
                checkBox.Disable()
        else:
            for checkBox in [self.checkBox1, self.checkBox2, self.checkBox3, self.checkBox4]:
                checkBox.Enable()
        self.selectFlag = not self.selectFlag
```

更多资源请访问稀酷客(www.ckook.com)

【例 13-13】接上例。实现【查看】按钮的处理函数 OnClick( )。单击【查看】按钮，弹出对话框，该对话框显示出列表 fruit 中存储的水果，即用户选择的水果。

```
def OnClick(self, event):
    str = "".join(self.fruit)                                    # 获取 checkbox 的文本值
    wx.MessageBox(str.encode("cp936"), u"提示")                  # 输出中文对话框
```

【例 13-13】运行后的效果如图 13-11 所示。

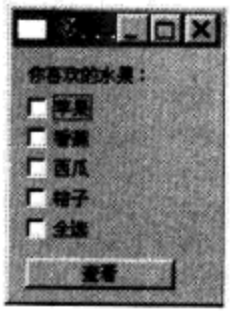


图 13-11　多选框事件窗口

13.6　列表控件

列表控件的作用与单选框和复选框相似，都可以为用户提供选择。列表控件的形式多样，包括列表框、下拉列表等控件。

13.6.1　列表框

列表框把提供给用户选择的选项放到 1 个矩形的区域中，用户可以在列表框中选择 1 个或多个选项。ListBox 类可以创建列表框，ListBox 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, Point pos = DefaultPosition,
        Size size = DefaultSize, wxArrayString choices = wxPyEmptyStringArray,
        long style = 0, Validator validator = DefaultValidator,
        String name = ListBoxNameStr) -> ListBox
```

参数 style 提供了列表框选项的选择方式，如表 13-3 所示。

表 13-3　列表框的样式

样　式	说　明
wx. LB_EXTENDED	使用 shift 键和鼠标连续选择
wx. LB_MULTIPLE	一次可以选择多个选项
wx. LB_SINGLE	一次只能选择一个选项
wx. LB_ALWAYS_SB	列表框始终显示一个垂直滚动条
wx. LB_HSCROLL	列表框显示一个水平滚动条
wx. LB_SORT	列表框中的元素按照字母顺序排列

列表框中还可以提供复选框供用户选择，丰富了用户界面的友好性。CheckListBox 类可以创建带复选框的列表控件。CheckListBox 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, Point pos = DefaultPosition,
        Size size = DefaultSize, wxArrayString choices = wxPyEmptyStringArray,
        long style = 0, Validator validator = DefaultValidator,
        String name = ListBoxNameStr) -> CheckListBox
```

参数 style 也可以使用表 13-3 中所示的样式。

更多资源请访问稀酷客(www.ckook.com)

【例 13-14】创建 1 个列表框控件和 1 个带复选框的列表控件。

```
import wx

class ListBoxFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'列表框', size=(200, 180))
        panel = wx.Panel(self, -1)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        # 关联 colorList 和控件 listBox
        self.listBox = wx.ListBox(panel, -1, (10, 10), (80, 110), colorList, wx.LB_SINGLE)
        self.listBox.SetSelection(0)
        # 关联 colorList 和控件 checklistBox
        self.checkListBox = wx.CheckListBox(panel, -1, (100, 10), (80, 110), colorList, wx.LB_SORT)
        # 绑定单击事件
        self.Bind(wx.EVT_LISTBOX, self.OnSelect, self.listBox)

    def OnSelect(self, event):
        index = self.listBox.GetSelection()
        wx.MessageBox(self.listBox.GetString(index), u"提示")

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ListBoxFrame().Show()
    app.MainLoop()
```

- 第 9 行代码创建了 ListBox 控件，使用户每次只能选择一个选项。
- 第 10 行代码调用 SetSelection( )方法，默认情况下选中第一个选项。
- 第 12 行代码创建了 CheckListBox 控件。
- 第 14 行代码给 ListBox 控件绑定 EVT_LISTBOX 事件，事件处理函数为 OnSelect( )。
- 第 16 行代码实现 OnSelect( ) 函数，每次单击选项后弹出对话框，显示用户所选择的内容。

【例 13-14】运行后的效果如图 13-12 所示。

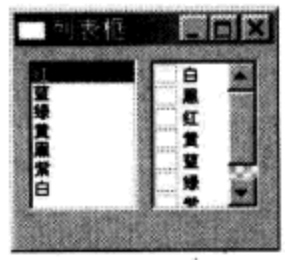


图 13-12　列表框

13.6.2　下拉列表

下拉列表是列表框的另一种表现形式，下拉列表占用空间小，适合存放数据量较少的选项。Choice 类可以用于创建下拉列表。Choice 类的构造函数如下所示：

```
__init__(Window parent, int id, Point pos=DefaultPosition, Size size=DefaultSize,
        List choices=EmptyList, long style=0, Validator validator=DefaultValidator,
        String name=ChoiceNameStr) -> Choice
```

【例 13-15】使用下拉列表框展示 7 种颜色的名称。

```
import wx

class ChoiceFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'下拉列表框', size=(200, 100))
        panel = wx.Panel(self, -1)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        wx.StaticText(panel, -1, u"选择喜欢的颜色:", (10, 10))
        wx.Choice(panel, -1, (120, 10), choices=colorList)          # 把 colorList 绑定到
choice 控件

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ChoiceFrame().Show()
    app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

第 9 行代码创建了 Choice 控件，并把 colorList 绑定到 choice 控件。

【例 13-15】运行后的效果如图 13-13 所示。



图 13-13　下拉列表

13.6.3　可编辑的下拉列表

Choice 类创建的下拉列表是只读状态，用户不能在下拉列表中添加新的项目。如果用户选择的项目不在列表中，应该允许用户自行输入。这种情况下可以使用 ComboBox 控件，ComboBox 控件是文本框和下拉列表的组合。ComboBox 类的构造函数如下所示：

```
__init__(Window parent, int id = -1, String value = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        List choices = EmptyList, long style = 0, Validator validator = DefaultValidator,
        String name = ComboBoxNameStr) -> ComboBox
```

ComboBox 控件有一些常用的显示样式，如表 13-4 所示。

表 13-4　ComboBox 控件的样式

样　式	说　明
wx. CB_DROPDOWN	创建一个带有下拉列表的组合框
wx. CB_SIMPLE	创建一个带有列表框的组合框
wx. CB_READONLY	创建只读的列表框
wx. CB_SORT	创建按字母排序的列表框

【例 13-16】演示使用 ComboBox 控件选择“颜色”的方法。

```
import wx

class ComboBoxFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'可编辑的下拉列表', size = (180, 100))
        panel = wx.Panel(self, -1)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        wx.StaticText(panel, -1, u"喜欢的颜色:", (20, 20))
        # 把 colorList 绑定到 combobox
        wx.ComboBox(panel, -1, u'红', (100, 20), wx.DefaultSize, colorList, wx.CB_DROPDOWN)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    ComboBoxFrame().Show()
    app.MainLoop()
```

【例 13-16】运行后的效果如图 13-14 所示。



图 13-14　可编辑的下拉列表

注意　ComboBox 类是 Choice 的子类，ComboBox 可以使用 Choice 类的方法。

更多资源请访问稀酷客(www.ckook.com)

13.7　使用 Sizers 布局组件

Sizers 布局管理器是管理界面中各种控件的组件，使用 Sizers 组件可以自动解决控件的位置和控件之间的间距问题，提高了 GUI 程序的可控性。

13.7.1　Sizers 布局管理器

前面的程序都是通过设置构造函数的 pos、size 属性调整控件之间的位置。WxPython 提供了 Sizers 布局管理器自动设置控件之间的位置，不再用手动指定每个控件的布局。Sizers 管理器可以帮助程序员设置控件的位置和间距，并负责将每个控件输出到屏幕上。Sizers 布局管理器的使用一般分为以下步骤。

（1）创建 Sizers 布局管理器。布局管理器分为 grid sizer、flex grid sizer、gird bag sizer、box sizer，根据布局的需要调用相关的构造函数。WxPython 的布局管理器都继承自父类 Sizer。

（2）调用 SetSizer( )将布局管理器添加到容器中，这里所指的容器一般是窗口、Panel 等组件。这样就建立了布局管理器和窗口之间的联系。SetSizer( )方法的声明如下所示（其中的 sizer 表示布局管理器）：

```
SetSizer(self, Sizer sizer, bool deleteOld = True)
```

（3）调用容器的 Add( )方法将各个组件添加到布局管理器中，add( )方法的声明如下所示：

```
Add(self, item, int proportion = 0, int flag = 0, int border = 0, PyObject userData = None) ->
wx.SizerItem
```

其中 item 是需要添加到布局管理器的组件。proportion 表示当窗体的大小发生变化时，控件之间的比例。Proportion 是一个比例值。flag 表示窗体风格、对齐方式、边框等信息。例如 flag 指定了边框，则参数 border 表示边框的大小。userData 参数可以被用来传递额外的数据。

（4）调用容器的 Fit( )方法计算布局管理器和容器窗口的大小：

```
Fit(self)
```

13.7.2　Grid Sizer 布局

grid sizer 布局采用表格的形式分配各种控件，不需要设置控件在容器中的位置，直接添加到 grid sizer 布局管理器中即可。

【例 13-17】创建 3 行 3 列的布局，然后根据列表中的元素创建 7 个 Button 控件，并添加到 3 行 3 列的布局中。

```
import wx

class GridSizerFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"布局管理器 - GridSizer", size = (300, 150))
        panel = wx.Panel(self, -1)
        sizer = wx.GridSizer(rows = 3, cols = 3, hgap = 5, vgap = 5)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        for color in colorList:
            btn = wx.Button(panel, -1, color)
            sizer.Add(btn, 0, 0)
        panel.SetSizer(sizer)
```

更多资源请访问稀酷客(www.ckook.com)

```
        panel.Fit()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = GridSizerFrame()
    frame.Show()
    app.MainLoop()
```

- 第 7 行代码创建了 GridSizer 布局。整个布局分为 3 行 3 列，组件之间的水平距离和垂直距离均为 5。
- 第 9 行代码通过循环创建了 7 个 Button 控件。
- 第 11 行代码把 7 个 Button 控件添加到布局管理器中。
- 第 12 行代码调用 SetSizer( )方法，把 GridSizer 布局关联到 Panel 组件中。
- 第 13 行代码调用 Fit( )方法计算 GridSizer 布局和 Panel 组件的位置。

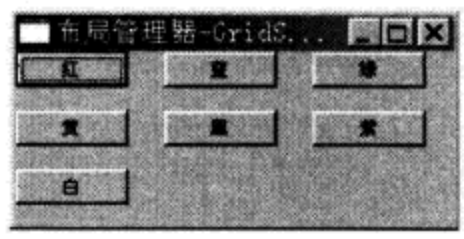


图 13-15　Grid 布局管理器

【例 13-17】运行后的效果如图 13-15 所示。

13.7.3　Flex Grid Sizer 布局

flex grid sizer 布局管理器是对 grid sizer 的改进，flex grid sizer 布局管理器用于存放大小可改变的组件。

【例 13-18】创建大小各不相同的 Button 控件，然后把这些控件添加到 flex grid sizer 布局管理器中。

```
import wx

class GridSizerFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"布局管理器-gridSizer", size = (300, 150))
        panel = wx.Panel(self, -1)
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑']
        sizer = wx.FlexGridSizer(rows = 3, cols =3, hgap =5, vgap =5)
        for color in colorList:
            btn = wx.Button(panel, -1, color)
            sizer.Add(btn, 0, 0)
        btn = wx.Button(panel, -1, u'紫')
        btn.SetMinSize((100,40))
        sizer.Add(btn, 0, 0)
        btn = wx.Button(panel, -1, u'白')
        btn.SetMinSize((50,30))
        sizer.Add(btn, 0, 0)
        panel.SetSizer(sizer)
        panel.Fit()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = GridSizerFrame()
    frame.Show()
app.MainLoop()
```

- 第 8 行代码创建 FlexGridSizer 类型的布局，整个布局分为 3 行 3 列，组件之间的水平距离和垂直距离均为 5。
- 第 9 行代码通过 for... in 循环创建 5 个 Button 控件。
- 第 11 行代码把这 5 个控件添加到布局管理器中。

更多资源请访问稀酷客(www.ckook.com)

- 第 12 行代码创建标签名为“紫”的 Button 控件。
- 第 13 行代码设置控件的大小，使该控件的“尺寸”大于前面的 5 个 Button 控件。

【例 13-18】运行后的效果如图 13-16 所示。

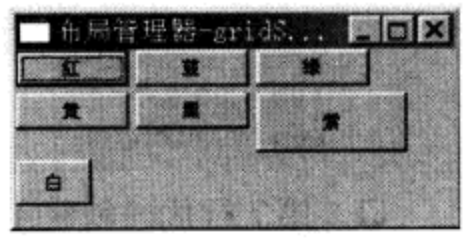


图 13-16　Flex Grid 布局管理器

13.7.4　Grid Bag sizer 布局

grid bag sizer 布局提供了更强大的布局能力。grid bag sizer 能够将某个控件添加到特定的单元格，并能使该控件跨行、列显示，类似于 HTML 中的 span 标签。

【例 13-19】演示 grid bag sizer 布局管理器的使用方法。

```
import wx

class BagSizerFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"布局管理器-bagSizer", size = (350, 120))
        panel = wx.Panel(self, -1)
        sizer = wx.GridBagSizer(hgap=5, vgap=5)
        colorList = [u'红', u'蓝', u'绿']
        col = 0
        for color in colorList:
            btn = wx.Button(panel, -1, color)
            sizer.Add(btn, pos=(0, col))
            col = col + 1
        btn = wx.Button(panel, -1, u'紫')
        sizer.Add(btn, pos=(1, 0), span=(1, 3), flag=wx.EXPAND)
        btn = wx.Button(panel, -1, u'白')
        sizer.Add(btn, pos=(0, col + 1), span=(2, 1), flag=wx.EXPAND)
        btn = wx.Button(panel, -1, u'黄')
        sizer.Add(btn, pos=(2, 0), span=(1, 2), flag=wx.EXPAND)
        btn = wx.Button(panel, -1, u'黑')
        sizer.Add(btn, pos=(2, 2), span=(1, 1), flag=wx.EXPAND)
        panel.SetSizer(sizer)
        panel.Fit()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = BagSizerFrame()
    frame.Show()
    app.MainLoop()
```

- 第 7 行代码创建 GridBagSizer 类型的布局，组件之间的水平距离和垂直距离均为 5。
- 第 9 行到第 13 行代码通过循环水平添加“红”、“蓝”、“绿”3 个按钮。
- 第 14 行代码，跨“红”、“蓝”、“绿”3 列添加按钮“紫”。wx.EXPAND 表示使控件按一定方向增量排列。
- 第 16 行代码，跨“红”、“紫”所在的 2 行添加按钮“白”。
- 第 18 行代码，跨“红”、“蓝”2 列添加按钮“黄”。
- 第 20 行代码，在“绿”所在的列添加按钮“黑”。

【例 13-19】运行后的效果如图 13-17 所示。



图 13-17　Bag 布局管理器

注意　如果后添加到布局管理器中的控件占用了已添加到布局管理器中的控件位置，将编译出错。例如，如果标签为“黑”的控件占用了其他控件的位置，布局将会失败。

更多资源请访问稀酷客(www.ckook.com)

13.7.5　Box Sizer 布局

box sizer 是布局管理器中最简单、最灵活的一种布局，box sizer 可以水平或垂直排列，使各个组件从左至右或从上到下排列在一条线上。box sizer 可以嵌套使用，水平排列的布局可以嵌套水平或垂直排列的布局，这样使得应用程序的布局具有更多的灵活性。

【例 13-20】创建水平排列的 BoxSizer。

```
import wx

class GridSizerFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"布局管理器-boxSizer", size = (500, 80))
        panel = wx.Panel(self, -1)
        sizer = wx.BoxSizer(wx.HORIZONTAL)                          # 水平对齐的 BoxSizer
        colorList = [u'红', u'蓝', u'绿', u'黄', u'黑', u'紫', u'白']
        for color in colorList:
            btn = wx.Button(panel, -1, color)
            sizer.Add(btn, 0, 0)                                    # 把控件添加到布局管理器中
        panel.SetSizer(sizer)
        panel.Fit()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = GridSizerFrame()
    frame.Show()
    app.MainLoop()
```

第 7 行代码创建了水平排列的布局 BoxSizer。第 11 行代码把 7 个按钮水平添加到布局中。

【例 13-20】运行后的效果如图 13-18 所示。



图 13-18　Box 布局管理器

13.8　小结

本章讲解了 wxPython 库中的基本组件，包括静态文本框、文本框、按钮、单选框、复选框、列表等常用的控件。通过具体的例子说明了各种控件的样式设置和事件响应的处理。最后介绍了 wxPython 的布局管理器，重点说明了 grid、grid bag、box、flex grid 等布局管理器的使用。

更多资源请访问稀酷客(www.ckook.com)

第 14 章　WxPython 库中的菜单、窗口与对话框控件

本章将介绍 wxPython 库中的菜单、窗口与对话框控件的使用方法。菜单是系统的功能列表，其形式灵活多样，包括多级菜单、位图菜单以及上下文菜单等。窗口是应用程序的容器组件，窗口中可以放置各种 wxPython 控件。wxPython 窗口提供了各种常见的窗口类型，包括 MDI（多文档界面 Multiple Document Interface）窗口、MiniFrame（迷你）窗口以及 HTML 窗口。对话框用于应用程序的提示、录入、数据验证等操作。本章的知识点：

- 菜单的创建
- 多级菜单
- 位图菜单
- MDI 窗口的创建
- MiniFrame 窗口的创建
- 对话框的创建
- 模态对话框和非模态对话框

14.1　菜单

菜单是桌面应用程序中最常用的控件之一。菜单展现了应用程序包含的各种功能。本节将介绍菜单的事件、快捷键、位图菜单、多级菜单、上下文菜单等方面的内容。

14.1.1　创建菜单

菜单的创建一般分为如下几个步骤。

（1）创建菜单栏。调用 wx. MenuBar 类创建菜单栏。以下代码创建一个名为 menuBar 的菜单栏：

```
menuBar = wx.MenuBar()
```

（2）创建父菜单。调用 wx. Menu 类创建父菜单。以下代码创建一个名为 menu 的菜单：

```
menu = wx.Menu()
```

（3）把父菜单添加到菜单栏中。调用 MenuBar 类的 Append( )方法。例如，下面这行代码添加了“文件”菜单。menu 是菜单对象：

```
menuBar.Append(menu, u"文件")
```

（4）添加子菜单。调用 Menu 类的 Append( )方法添加子菜单。例如，下面这行代码添加了“消息框”子菜单。“1000”表示子菜单的编号，菜单的编号是唯一的：

```
menu.Append(1000, u"消息框")
```

（5）添加菜单事件，菜单事件为 wx. EVT_MENU。

MenuBar 类的 Append( )方法的声明如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
Append(self, Menu menu, String title) -> bool
```

其中，menu 是添加到菜单栏的菜单项，title 表示菜单的名称。

Menu 类的 Append( )方法的使用与 MenuBar 类的 Append( )类似，不同的是前者直接创建子菜单，并把子菜单添加到父菜单中。

【例 14-1】演示菜单的创建方法。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"菜单", size=(300, 200))
        p = wx.Panel(self)
        menuBar = wx.MenuBar()                                    # 添加菜单栏
        menu = wx.Menu()
        menuBar.Append(menu, u"文件")                             # 添加菜单
        menu.Append(1000, u"消息框")                              # 添加子菜单
        menu.AppendSeparator()
        menu.Append(1001, u"退出")
        self.Bind(wx.EVT_MENU, self.OnHello, id=1000)                    # 添加菜单事件
        self.Bind(wx.EVT_MENU, self.OnExit, id=1001)
        self.SetMenuBar(menuBar)

    def OnHello(self, event):
        wx.MessageBox(u"你好!", u"提示")

    def OnExit(self, event):
        self.Close(True)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 9 行代码把父菜单 menu 添加到菜单栏中，并把菜单命名为“文件”。
- 第 10 行代码在“文件”菜单下添加子菜单“消息框”。
- 第 11 行代码添加菜单分隔线。
- 第 12 行代码在“文件”菜单下添加子菜单“退出”。
- 第 13 行代码为“消息框”菜单添加事件处理函数 OnHello( )。
- 第 14 行代码为“退出”菜单添加事件处理函数 OnExit( )。

【例 14-1】运行后的效果如图 14-1 所示。

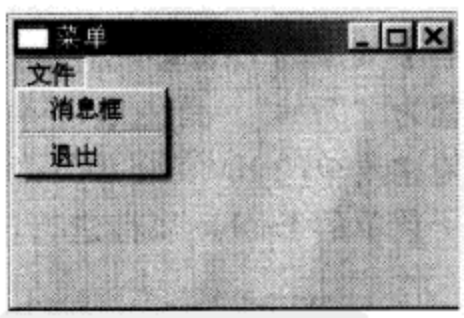


图 14-1　菜单的创建

MenuBar 类还提供了一些有用的方法，如表 14-1 所示。

表 14-1　MenuBar 类的常用方法

方　法	说　明
Append（self, Menu menu, String title）	添加菜单对象，并设置菜单的名称
Enable（self, int id, bool enable）	设置 id 所对应菜单的 Enable 属性。enable 为 Ture，则表示菜单可用
FindItemById（self, int id）	返回 id 所对应的菜单项，返回类型为 MenuItem
FindMenu（self, String title）	根据 title 返回菜单栏给定的整数索引
GetLabel（self, int id）	根据菜单的 id 返回菜单的文本标签
GetMenu（self, size_t pos）	返回 pos 索引处的菜单对象，pos 从 0 开始计算

更多资源请访问稀酷客(www.ckook.com)

（续）

方　法	说　明
GetMenuCount（self）	返回菜单栏中的菜单数量
GetMenuLabel（self，size_t pos）	返回 pos 索引处的菜单名称
Insert（self，size_t pos，Menu menu，String title）	将菜单 menu 插入到 pos 索引处
IsEnabled（self，int id）	判断 id 所对应菜单的 enable 属性
Remove（self，size_t pos）	删除 pos 索引处的菜单，返回被删除的菜单对象
Replace（self，size_t pos，Menu menu，String title）	使用菜单 menu 替换 pos 索引处的菜单，返回被替换的菜单对象
SetLabel（self，int id，String label）	设置 id 对应的菜单标签
SetMenuLabel（self，size_t pos，String label）	设置 pos 索引处的菜单标签

14.1.2　菜单的事件

菜单的事件是对菜单处理的响应方式。菜单的事件包括单击、选择等操作。前面一节使用了 wx. EVT_MENU 响应菜单事件，菜单对象还可以处理表 14-2 所示的事件。

表 14-2　菜单的响应事件

事　件	说　明
EVT_MENU_CLOSE	关闭菜单时触发的事件
EVT_MENU_HIGHLIGHT	菜单高亮显示时触发的事件
EVT_MENU_HIGHLIGHT_ALL	任何菜单高亮显示时都执行同一个操作
EVT_MENU_OPEN	打开菜单时触发的事件

每个菜单的事件响应都对应一个事件处理函数。当遇到特定的操作时，将触发相应的事件，然后调用事件处理函数处理。

【例 14-2】实现菜单的 EVT_MENU_HIGHLIGHT 事件，并对子菜单的打开、屏蔽进行控制。当子菜单的 Eanble 属性为“True”时，设置为“False”；子菜单 Enable 属性为“False”时，设置为“True”。首先通过 MenuBar 对象的 IsEnabled( )方法获取某个子菜单当前的 Enable 属性，传递菜单的 ID 值给 IsEnabled( )即可返回相应的布尔值。然后调用 Menu 对象的 Enable( )方法对子菜单的 Enable 属性进行设置。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"菜单的属性", size=(300, 200))
        p = wx.Panel(self)
        self.mainMenu = wx.Menu()
        self.mainMenu.Append(1000, u"退出")
        setMenu = wx.Menu()
        subSetMenu = setMenu.Append(1001, u"打开/屏蔽菜单")
        self.Bind(wx.EVT_MENU, self.OnExit, id = 1000)
        self.Bind(wx.EVT_MENU_HIGHLIGHT, self.OnItemSelected, id = 1000)
        self.Bind(wx.EVT_MENU, self.OnEnable, subSetMenu)
        menuBar = wx.MenuBar()
        menuBar.Append(self.mainMenu, u"主菜单")
        menuBar.Append(setMenu, u"设置")
        self.SetMenuBar(menuBar)
```

更多资源请访问稀酷客(www.ckook.com)

```
    def OnEnable(self, event):
        menubar = self.GetMenuBar()
        enabled = menubar.IsEnabled(1000)                    # 获取菜单的 enable 属性
        self.mainMenu.Enable(1000, not enabled)              # 设置菜单的 enable 属性

    def OnExit(self, event):
        self.Close()

    def OnItemSelected(self, event):                         # 当菜单高亮时返回菜单的名称
        item = self.GetMenuBar().FindItemById(event.GetId())  # 获取菜单项
        wx.MessageBox(u"菜单:%s" % item.GetText())

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 12 行代码对菜单 ID 为 1000 的子菜单设置 EVT_MENU_HIGHLIGHT 事件。事件处理函数为 OnItemSelected( )。
- 第 13 行代码，当单击菜单对象 subSetMenu 时，对 ID 为 1000 的子菜单的 Enable 属性进行设置。处理函数为 OnEnable( )。
- 第 21 行代码调用 menubar.IsEnabled（1000）返回 ID 为 1000 的子菜单的 Enable 属性值，并存入变量 enabled 中。
- 第 22 行代码调用设置 ID 为 1000 的子菜单的 Enable 属性为当前相反的布尔值。
- 第 28 行代码，首先调用 Event 的 GetId( )方法返回事件源的 ID，然后调用 MenuBar 对象的 FindItemById( )方法返回 ID 对应的子菜单对象 item。
- 第 29 行代码返回子菜单对象 item 的 Text 属性。

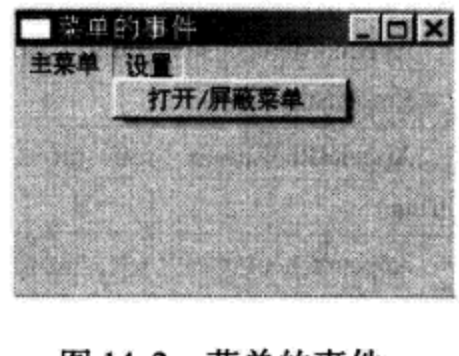


图 14-2　菜单的事件

【例 14-2】运行后的效果如图 14-2 所示。

14.1.3　多级菜单

Menu 类支持嵌套操作，即可以定义多级菜单。新创建的 Menu 对象可以添加到已经创建的 Menu 对象中。这种操作可以使用 Menu 对象的 AppendMenu( )方法实现。前面使用了 Append( )方法动态添加菜单，也可以使用 AppendCheckItem( )、AppendRadioItem( )方法添加带多选标记、单选标记的菜单。

【例 14-3】演示定义多级菜单的方法。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"多级菜单", size=(300, 100))
        p = wx.Panel(self)
        menu = wx.Menu()
        submenu = wx.Menu()
        submenu.AppendCheckItem(-1, u"菜单 1")               # 多选标记
        submenu.AppendRadioItem(-1, u"菜单 2")               # 单选标记
        menu.AppendMenu(-1, u"子菜单", submenu)              # 添加子菜单
        menuBar = wx.MenuBar()
        menuBar.Append(menu, u"菜单")
        self.SetMenuBar(menuBar)
```

更多资源请访问稀酷客(www.ckook.com)

```
if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

【例 14-3】运行后的效果如图 14-3 所示。



图 14-3　多级菜单

前面使用了一些 Menu 类的操作方法，表 14-3 列出了 Menu 类的一些常用方法。

表 14-3　Menu 类的常用方法

方法	说明
Append（self, int id, String text = EmptyString, String help = EmptyString, int kind = ITEM_ NORMAL）	添加菜单
AppendItem（self, MenuItem item）	添加菜单项
AppendMenu（self, int id, String text, Menu submenu, String help = EmptyString）	在某个菜单对象上添加子菜单
AppendRadioItem（self, int id, String text, String help = EmptyString）	添加带单选标记的菜单项
AppendCheckItem（self, int id, String text, String help = EmptyString）	添加带多选标记的菜单项
AppendSeparator（self）	添加菜单分隔线
Delete（self, int id）	删除指定 id 的菜单
DeleteItem（self, MenuItem item）	删除指定的菜单项
Enable（self, int id, bool enable）	设置 id 对应菜单的 enable 属性
FindItem（self, String item）	返回菜单项的索引
FindItemById（self, int id）	根据 id 返回对应的菜单项
GetLabel（self, int id）	根据菜单的 id 返回菜单的标签
GetMenuItems（self）	返回菜单中菜单项列表
GetMenuItemCount（self）	返回菜单中菜单项的数量
InsertItem（self, size_t pos, MenuItem item）	将新的菜单添加到已有的菜单中
SetLabel（self, int id, String label）	设置 id 对应的菜单标签
XSetMenuLabel（self, size_t pos, String label）	设置 pos 索引处的菜单标签

14.1.4　菜单的快捷键

定义菜单的快捷键通常采用助记符的方式。助记符作为菜单名称的一部分，用 1 个带有下划线的字母表示。用户按下 Alt 键后可以激活窗口中的菜单，然后按下菜单中的助记符就可以访问相应菜单的内容。通常使用菜单英文名称的首字母作为助记符。如果菜单的名称中需要包含“&”符号，必须使用“&&”来表示 &字符。菜单快捷键的添加可以在菜单创建的同时完成。例如下面这行代码添加“退出”菜单，并同时设置快捷键：

```
exit = menu.Append(-1, u"&x退出")
```

更多资源请访问稀酷客(www.ckook.com)

【例 14-4】演示在菜单中添加快捷键的方法。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"菜单快捷键", size=(300, 200))
        p = wx.Panel(self)
        menu = wx.Menu()
        exit = menu.Append(-1, u"&x 退出")
        self.Bind(wx.EVT_MENU, self.OnExit, exit)
        menuBar = wx.MenuBar()
        menuBar.Append(menu, u"&m 菜单")
        self.SetMenuBar(menuBar)

    def OnExit(self, event):
        self.Close()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码调用 menu 对象的 Append( )添加菜单，菜单名称中使用了“&x”作为助记符。
- 第 11 行代码给主菜单定义了助记符“&m”。当用户按下 Alt 键，然后按下 m 键即可展开菜单。最后按下 x 键即可退出程序。

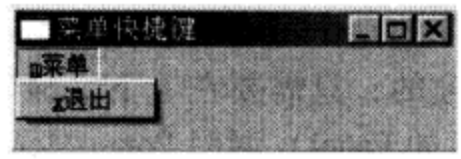


图 14-4　菜单快捷键

【例 14-4】运行后的效果如图 14-4 所示。

注意　助记符在同一个菜单中是唯一的，在整个菜单栏中可以重复。

14.1.5　位图菜单

传统菜单的展现方式比较单一，位图菜单很好的解决了这个问题。位图菜单可以在菜单项的左侧添加 bmp、png 格式的图片，丰富了应用程序的界面。位图菜单的创建和普通菜单创建的过程大致相同，菜单项使用 MenuItem 对象创建，菜单项添加前需要指定 Bitmap 对象，最后把 Bitmap 对象添加到 MenuItem 中。

【例 14-5】演示实现带位图的菜单。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"位图菜单", size=(300, 100))
        p = wx.Panel(self)
        menu = wx.Menu()
        bmp = wx.Bitmap("previous.png", wx.BITMAP_TYPE_PNG)
        item = wx.MenuItem(menu, -1, u"位图 1")
        item.SetBitmap(bmp)                          # 给菜单增加位图
        menu.AppendItem(item)
        bmp = wx.Bitmap("next.png", wx.BITMAP_TYPE_PNG)
        item = wx.MenuItem(menu, -1, u"位图 2")
        item.SetBitmap(bmp)                          # 给菜单增加位图
        menu.AppendItem(item)
        menuBar = wx.MenuBar()
```

更多资源请访问稀酷客(www.ckook.com)

```
        menuBar.Append(menu, u"菜单")
        self.SetMenuBar(menuBar)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码定义了 png 格式的位图对象 bmp，该位图对象指向图片 previous.png。
- 第 9 行代码创建了 MenuItem 对象 item。
- 第 10 行代码给对象 item 添加位图 bmp。
- 第 11 行代码把菜单对象 item 添加到菜单 menu 中。
- “位图 2”菜单的创建方式与“位图 1”菜单的创建方法类似。

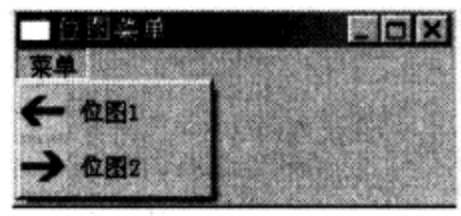


图 14-5　位图菜单

【例 14-5】运行后的效果如图 14-5 所示。

14.1.6　上下文菜单

上下文菜单又称弹出式菜单，即单击鼠标右键后弹出的菜单。上下文菜单丰富了桌面应用程序的操作，用户可以快速的定位到上下文菜单中指定的某个功能。上下文菜单也是由 Menu 类创建的，只需要在上下文显示的容器中绑定 wx.EVT_CONTEXT_MENU 事件，然后由容器组件（如 Panel）调用 PopupMenu（）方法弹出上下文菜单即可。

【例 14-6】在 TextCtrl 控件中添加上下文菜单，并把用户选择的菜单文本复制到 TextCtrl 创建的文本框中。

```
import wx

class PopupFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"弹出式菜单", size=(200, 75))
        self.panel = wx.Panel(self)
        self.textCtrl = wx.TextCtrl(self.panel, -1, pos=(10, 10))
        menuBar = wx.MenuBar()
        self.SetMenuBar(menuBar)
        self.popupmenu = wx.Menu()
        menuList = [u"菜单1", u"菜单2", u"菜单3"]
        for menuitem in menuList:
            item = self.popupmenu.Append(-1, menuitem)
            self.Bind(wx.EVT_MENU, self.OnPopupItemSelected, item)    # 菜单项选择事件
            self.textCtrl.Bind(wx.EVT_CONTEXT_MENU, self.OnPopup)     # 在文本框中绑定弹出式菜单

    def OnPopup(self, event):
        pos = self.panel.ScreenToClient(event.GetPosition())          # 获取鼠标的坐标
        self.panel.PopupMenu(self.popupmenu, pos)

    def OnPopupItemSelected(self, event):
        item = self.popupmenu.FindItemById(event.GetId())
        self.textCtrl.SetLabel(item.GetText())

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = PopupFrame()
    frame.Show()
    app.MainLoop()
```

- 第 10 行代码定义了菜单 popupmenu，后面将会把该菜单设置为上下文菜单。

更多资源请访问稀酷客(www.ckook.com)

- 第 11 行代码定义了菜单的菜单项，使用列表 menuList 存储。
- 第 12 行代码通过 for... in 循环把菜单列表 menuList 中的菜单项添加到菜单 popupmenu 中，并返回 MenuItem 对象 item。
- 第 14 行代码定义对象 item 的选择事件，处理函数为 OnPopupItemSelected( )。
- 第 15 行代码在文本框控件中设置上下文菜单，处理函数为 OnPopup( )。该函数用于绑定文本框控件 textCtrl 和菜单 popupmenu。
- 第 18 行代码获取鼠标右键单击时的坐标。
- 第 19 行代码调用 PopupMenu( )方法在 pos 坐标处弹出上下文菜单。
- 第 23 行代码把菜单项的文本值复制给 textCtrl 的 Label 属性。

【例 14-6】运行后的效果如图 14-6 所示。



图 14-6　上下文菜单

14.2　窗口

在 wxPython 中使用 wx. Frame 类表示窗口。窗口是一个包含多个组件的矩形区域，窗口是程序运行后的容器。本节将介绍常见的 MDI（多文档界面）窗口、工具栏窗口的创建和使用方法。

14.2.1　Frame 窗口

所有的用户操作都是在窗口中完成的，wxPython 使用 wx. Frame 类创建窗口。Frame 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String title = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = DEFAULT_FRAME_STYLE, String name = FrameNameStr)
```

- 参数 parent 表示当前窗口的父窗口。
- 参数 id 表示窗口的编号。
- 参数 title 表示窗口的标题。
- 参数 pos 表示窗口显示时的位置。
- 参数 size 表示窗口显示时的大小。
- 参数 sytle 用于定制窗口的外观。
- 参数 name 表示窗口的名称。

wxPython 应用程序需要首先创建 wx. Frame 的子类，然后在程序的入口处对子类进行实例化。这样就可以创建 Frame 窗口。

【例 14-7】创建 1 个最简单的 Frame 窗口。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'自定义窗口', size = (300, 100))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    myFrame = MyFrame()
    myFrame.Show()
    app.MainLoop()
```

【例 14-7】运行后的效果如图 14-7 所示。

更多资源请访问稀酷客(www.ckook.com)



图 14-7　简单的 Frame 窗口

14.2.2　带工具栏和状态栏的 Frame 窗口

一个完整的应用程序窗口通常包含菜单、工具栏、状态栏等要素。菜单的创建前面已经介绍过了，工具栏和状态栏可以分别使用 Frame 类的 CreateToolBar( )、CreateStatusBar( )方法创建。

【例 14-8】演示带工具栏和状态栏的窗口的创建过程。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u'自定义窗口', size=(300, 200))
        png_save = wx.Image('./icons/save_page.png', wx.BITMAP_TYPE_PNG).ConvertToBitmap()
        png_home = wx.Image('./icons/go_home.png', wx.BITMAP_TYPE_PNG).ConvertToBitmap()
        png_cfg = wx.Image('./icons/settings.png', wx.BITMAP_TYPE_PNG).ConvertToBitmap()
        png_forward = wx.Image('./icons/go_forward.png', wx.BITMAP_TYPE_PNG).ConvertToBitmap
()
        png_back = wx.Image('./icons/go_back.png', wx.BITMAP_TYPE_PNG).ConvertToBitmap()
        toolbar = self.CreateToolBar(wx.TB_HORIZONTAL | wx.TB_TEXT)      # 创建工具栏
        toolbar.AddSimpleTool(100, png_save, "Save page")
        toolbar.AddSimpleTool(200, png_home, "Go home")
        toolbar.AddSimpleTool(220, png_back, "Go back")
        toolbar.AddSimpleTool(230, png_forward, "Go Forward")
        toolbar.AddSimpleTool(400, png_cfg, "Settings")
        toolbar.Realize()
        self.CreateStatusBar()                                    # 创建状态栏

if __name__ == '__main__':
    app = wx.PySimpleApp()
    myFrame = MyFrame()
    myFrame.Show()
    app.MainLoop()
```

- 第 6 行到第 10 行代码创建了 5 个位图对象，这个 5 个位图对象将添加到工具栏中。
- 第 11 行代码调用 CreateToolBar( )创建水平放置的工具栏。
- 第 12 行到第 16 行代码，在工具栏中添加 5 个位图按钮。
- 第 17 行代码调用 ToolBar 类的 Realize( )调整窗口中的工具栏。
- 第 18 行代码调用 CreatteStatusBar( )创建状态栏。

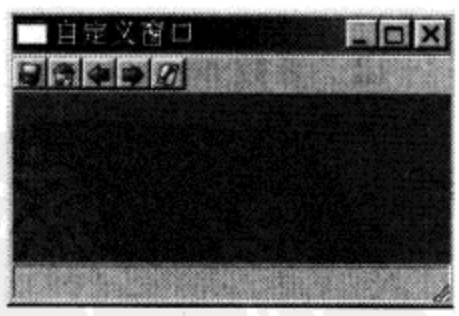


图 14-8　带工具栏和状态栏的 Frame 窗口

【例 14-8】运行后的效果如图 14-8 所示。

应用程序中如果同时包含菜单和工具栏组件，而且需要使某个菜单项和某个工具栏按钮打开同一个模块，只要使工具栏按钮取对应菜单的 ID 即可。例如添加子菜单 Settings 与上述程序中的工具栏按钮 Settings 对应：

```
menu = wx.Menu()
menu.Append(400, "Settings")
toolbar.AddSimpleTool(400, png_cfg, "Settings")
```

更多资源请访问稀酷客(www.ckook.com)

注意　上面图片的访问使用了相对路径，访问 icons 子目录需要在路径前面添加“.”，“.”表示当前目录。

14.2.3　MDI 窗口

MDI（多文档界面）窗口由多个窗口或其他组件组成。MDI 窗口程序可以创建子窗口，子窗口由 MDI 窗口管理和控制。在 wxPython 中使用 MDIParentFrame 类创建 MDI 窗口，MDIParentFrame 类的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String title = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = wxDEFAULT_FRAME_STYLE |wxVSCROLL |wxHSCROLL,
        String name = FrameNameStr) -> MDIParentFrame
```

【例 14-9】演示 1 个 MDI 窗口的创建过程。默认的 MDI 窗口会自带 1 个 Window 菜单，该菜单可以对 MDI 窗口中的子窗口进行排列操作。

```
import wx

class MDIFrame(wx.MDIParentFrame):                                    # 创建 MDI 窗口
    def __init__(self):
        wx.MDIParentFrame.__init__(self, None, -1, u"MDI 窗口", size = (600,400))
        menubar = wx.MenuBar()                                          # 添加菜单栏
        self.SetMenuBar(menubar)                                       # 提供带有 windows 的默认菜单

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MDIFrame()
    frame.Show()
    app.MainLoop()
```

- 第 3 行代码定义了 MDIFrame 类，该类继承自 wx.MDIParentFrame 类，即实现了 MDI 窗口的创建。
- 第 5 行代码调用父类 MDIParentFrame 的构造函数，初始化窗口的标题名称和大小。

【例 14-9】运行后的效果如图 14-9 所示。

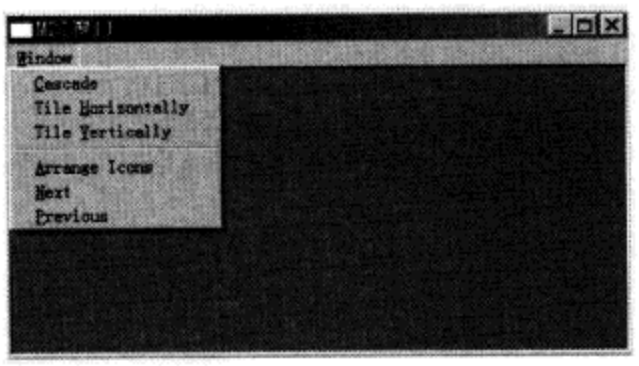


图 14-9　MDI 窗口

【例 14-10】接上例。给 MDI 窗口添加菜单，并能够在 MDI 窗口中创建子窗口。

```
import wx

class MDIFrame(wx.MDIParentFrame):
def __init__(self):
        wx.MDIParentFrame.__init__(self, None, -1, u"MDI 窗口", size = (300,200))
        menubar = wx.MenuBar()                                          # 添加菜单栏
        menu = wx.Menu()                                               # 添加菜单
        menu.Append(5000, u"新建(&N)")                                  # 添加子菜单
        menu.Append(5001, u"退出(&X)")
        menubar.Append(menu, u"文件(&F)")
        self.Bind(wx.EVT_MENU, self.OnNewWindow, id = 5000)            # 绑定菜单项的事件
        self.Bind(wx.EVT_MENU, self.OnExit, id = 5001)
        self.SetMenuBar(menubar)

    def OnExit(self, evt):
        self.Close(True)
```

更多资源请访问稀酷客(www.ckook.com)

```
    def OnNewWindow(self, evt):
        win = wx.MDIChildFrame(self, -1, u"子窗口")                    # 新建子窗口
        win.Show(True)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MDIFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码创建“新建”子菜单。单击该菜单将创建子窗口。
- 第 11 行代码给“新建”菜单绑定单击事件，处理函数为 OnNewWindow()。
- 第 19 行代码使用 wx. MDIChildFrame 类创建 MDI 子窗口。
- 第 20 行代码调用 show()方法显示子窗口。

【例 14-10】运行后的效果如图 14-10 所示。

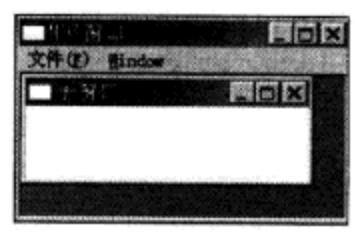


图 14-10　带菜单栏的 MDI 窗口

14.2.4　MiniFrame 窗口

MiniFrame 创建的窗口不在任务栏中显示，可以完成应用程序中一些特殊的要求，MiniFrame 通常作为 MDI 窗口中的工具栏窗口使用。MiniFrame 的构造函数如下所示：

```
__init__(self, Window parent, int id = -1, String title = EmptyString,
        Point pos = DefaultPosition, Size size = DefaultSize,
        long style = DEFAULT_MINIFRAME_STYLE, String name = FrameNameStr)
```

其中 style 属性可以定制窗口标题栏的按钮。窗口标题栏的右侧一般都有最小化、最大化、关闭等 3 个按钮。对于一些特定的应用程序通常需要定制，例如屏蔽最小化和最大化按钮。

【例 14-11】使用 wx. MiniFrame 定制窗口标题栏的按钮。

```
import wx

class MiniFrame(wx.MiniFrame):
    def __init__(self):
        wx.MiniFrame.__init__(self, None, -1, u'不能最小化和最大化的窗口', pos = (100, 100), size
= (300, 200),
                              style = wx.DEFAULT_FRAME_STYLE | wx.CLOSE_BOX)
        panel = wx.Panel(self, -1, size = (300, 200))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MiniFrame()
    frame.Show()
    app.MainLoop()
```

- 第 3 行代码创建 MiniFrame 类，该类继承自 wx. MiniFrame 类。
- 第 5 行代码初始化 MiniFrame 类，并设置窗口的 style 为 wx. DEFAULT_FRAME_STYLE | wx. CLOSE_BOX。也可以把 style 设置为 wx. DEFAULT_FRAME_STYLE^（wx. MINIMIZE_BOX | wx. MAXIMIZE_BOX））。

【例 14-11】运行后的效果如图 14-11 所示。



图 14-11　MiniFrame 窗口

14.3　对话框

对话框分为模态对话框、提示对话框、文本对话框、打开文件对话框等多种类型。在

更多资源请访问稀酷客(www.ckook.com)

wxPython中对话框都继承自 wx. Dialog 类。下面将一一介绍这些对话框的应用方法。

14. 3. 1　对话框的创建

对话框的创建和使用与 Frame 相似，不同的是对话框表示一次信息交换的活动。当交换完成后单击对话框的按钮将关闭该对话框，对话框只是应用程序生命周期的一部分。

对话框由主程序的窗口创建。继承自 wx. Dialog 的对话框分为模态对话框和非模态对话框。如果使用 Show( )方法显示的对话框称为非模态对话框，该对话框的窗口可以失去焦点，对主程序窗口进行其他操作。如果使用 ShowModal( )方法显示对话框，则此时的对话框称为模态对话框。用户必须关闭对话框后，才能对主程序窗口进行操作。桌面应用程序中通常都包含“关于”对话框，该对话框用于介绍应用程序的发布信息等内容。

【例 14-12】 创建“帮助” -> “关于”菜单，单击“关于”菜单后将创建并显示“关于”对话框。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"自定义对话框", size=(300, 200))
        MenuHelp = wx.Menu()
        aboutMenu = MenuHelp.Append(-1, u"关于(&A)")
        menuBar = wx.MenuBar()
        menuBar.Append(MenuHelp, u"帮助(&H)")
        self.Bind(wx.EVT_TOOL, self.ShowAboutDlg, aboutMenu)
        self.SetMenuBar(menuBar)
    def ShowAboutDlg(self, event):
        pos = self.GetPosition()
        dialog = MyDialog(self, -1, u"关于")
        dialog.SetPosition((pos[0] + 100, pos[1] + 60))

class MyDialog(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(100,100))
        self.panel = wx.Panel(self)
        self.OkBtn = wx.Button(self, 10, u"确定", pos=(10,20),size=(80,30))
        self.Bind(wx.EVT_BUTTON, self.CloseDlg, self.OkBtn)
  self.Show()

    def CloseDlg(self, event):
        self.Close()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
app.MainLoop()
```

- 第 3 行代码定义了 MyFrame 窗口类，该类包含了 aboutMenu 子菜单。单击 aboutMenu 子菜单将创建 MyDialog 对话框类。
- 第 13 行代码定义了函数 ShowAboutDlg( )，该函数用于创建并显示 aboutMenu 菜单。
- 第 15 行代码创建对话框 dialog。
- 第 16 行代码设置对话框 dialog 显示时相对于 MyFrame 窗口的位置。
- 第 18 行代码创建了 MyDialog 对话框类，该类继承自 wx. Dialog。MyDialog 类中创建了 1 个“确定”按钮，用户单击“确定”按钮后关闭打开的对话框。
- 第 24 行代码调用 wx. Dialog 类的 Show( )方法显示对话框。

更多资源请访问稀酷客(www.ckook.com)

【例 14-12】运行后的效果如图 14-12 所示。

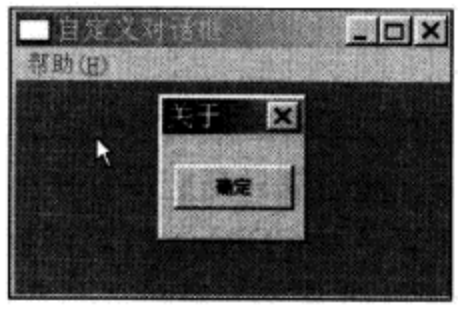


图 14-12　【关于】对话框

注意　由于模态对话框的约束性，因此模态对话框通常作为获取数据的窗口。只有从模态对话框中返回数据后，才能关闭对话框。

14.3.2　提示对话框

提示对话框通常用于帮助用户进行交互式操作。例如，退出应用程序时显示退出提示，删除数据时显示删除提示。这些提示功能有助于用户体验的完善。提示对话框由 wx. MessageDialog 类创建。wx. MessageDialog 类的构造函数如下所示：

```
__init__(self, Window parent, String message, String caption = MessageBoxCaptionStr,
        long style = wxOK |wxCANCEL |wxCENTRE,
        Point pos = DefaultPosition) -> MessageDialog
```

- 参数 parent 表示父窗口。
- 参数 message 表示对话框的内容。
- 参数 caption 表示对话框的标题。
- 参数 style 表示对话框的风格及包含的按钮。
- 参数 pos 表示对话框显示的位置。

【例 14-13】在主程序窗口中创建“退出”按钮，单击“退出”按钮显示对话框，提示是否退出应用程序。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"对话框", size = (300, 100))
        button = wx.Button(self, wx.ID_OK, u"退出", pos = (10, 10))
        button.SetDefault()
        self.Bind(wx.EVT_BUTTON, self.OnClick, button)

    def OnClick(self, event):
        # 创建选择对话框
        dlg = wx.MessageDialog(None, u"是否退出?", u"退出", wx.YES_NO | wx.ICON_QUESTION)
        # 判断用户选择的返回值
        if (dlg.ShowModal() == wx.ID_YES):
            frame.Close()
        dlg.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 12 行代码创建 MessageDialog 对话框，根据参数 wx. YES_NO，对话框生成“是”、“否”两个按钮。
- 第 14 行代码调用 ShowModal( )，并判断返回值是否为 wx. ID_YES。即用户是否单击了“是”按钮。
- 第 15 行代码，如果用户单击了“是”按钮，则关闭主程序。
- 第 16 行代码调用 Destroy( )方法释放对话框对象。

【例 14-13】运行后的效果如图 14-13 所示。

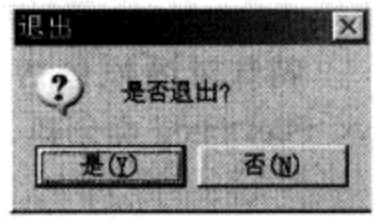


图 14-13　提示对话框

14.3.3　文本输入对话框

文本输入对话框用于返回输入字符串的值，文本输入对话框通常由 1 个静态文本框、1 个文本输入框和“OK”、“Cancel”按钮组成。文本输入对话框可由 wx. TextEntryDialog 类创建。wx. TextEntryDialog 类的构造函数如下所示：

```
__init__(self, Window parent, String message, String caption = GetTextFromUserPromptStr,
String defaultValue = EmptyString, long style = TextEntryDialogStyle, Point pos = DefaultPosition)
 -> TextEntryDialog
```

【例 14-14】创建 1 个 TextCtrl 控件，当用户在 TextCtrl 控件中按下回车键后，将显示 TextEntryDialog 对话框。用户输入文本后，单击“OK”按钮将把输入的值返回给 TextCtrl 控件。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "TextEntryDialog", size = (300, 100))
        panel = wx.Panel(self, -1)
        self.textCtrl = wx.TextCtrl(panel, -1, "", pos = (10,10), style = wx.TE_PROCESS_ENTER)
        self.textCtrl.Bind(wx.EVT_TEXT_ENTER, self.OnClick, self.textCtrl)

    def OnClick(self, evt):
        # 创建文本对话框
        self.dialog = wx.TextEntryDialog(None, u"输入文本", u"文本对话框", "", style = wx.OK |
wx.CANCEL)
        if self.dialog.ShowModal() == wx.ID_OK:
            self.textCtrl.SetLabel(self.dialog.GetValue())          # 获取输入文本的内容

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 7 行代码创建了 TextCtrl 的实例 textCtrl，并设置该控件的 style 为 wx. TE_PROCESS_ENTER，处理回车事件。
- 第 8 行代码绑定 textCtrl 的回车事件，处理函数为 OnClick( )。
- 第 12 行代码创建了 TextEntryDialog 的实例 dialog，文本输入对话框包含“OK”和“Cancel”2 个按钮。
- 第 13 行代码判断用户单击的按钮，如果单击了“OK”按钮，则返回用户输入的文本。
- 第 14 行代码调用 dialog 的 GetValue( )方法获取用户输入的文本，然后调用 textCtrl 的 SetLabel( )方法把文本填入 textCtrl 控件的文本框。

【例 14-14】运行后的效果如图 14-14 所示。

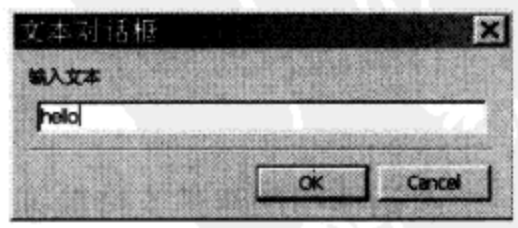


图 14-14　文本输入对话框

更多资源请访问稀酷客(www.ckook.com)

14.3.4　特殊对话框

特殊对话框包括打开文件对话框、字体对话框、颜色选择对话框。打开文件对话框由 wx. FileDialog 类创建，该类的构造函数如下所示：

```
__init__(self, Window parent, String message = FileSelectorPromptStr,
        String defaultDir = EmptyString, String defaultFile = EmptyString,
        String wildcard = FileSelectorDefaultWildcardStr,
        long style = FD_DEFAULT_STYLE,
        Point pos = DefaultPosition) -> FileDialog
```

其中 wildcard 表示文件类型的过滤字符串。该字符串由文件类型名、文件后缀名两部分组成。例如 Python source（*.py）| *.py，表示文件类型为 Python 源文件，文件后缀名为 py。如果需要打开多种类型的文件，需要在多种过滤字符串之间用“|”分隔。例如 Python source（*.py）| *.py | All files（*.*）| *.*，表示打开 Python 源文件和所有文件。

【例 14-15】演示“打开”文件对话框的创建和使用方法。

```
import wx
import os

def ShowFileDialog():
    filterFile = "Python source (*.py)|*.py|All files (*.*)|*.*"          # 过滤文件
    dialog = wx.FileDialog(None, u"选择文件", os.getcwd(), "", filterFile, wx.OPEN)
    dialog.ShowModal()
    dialog.Destroy()

if __name__ == "__main__":
    app = wx.PySimpleApp()
    ShowFileDialog()
```

- 第 4 行代码创建了“打开”文件对话框，该对话框将打开显示当前目录中的文件，并对可打开的文件类型进行过滤。
- 第 5 行代码对文件类型进行过滤。文件类型的下拉列表中提供“py”为后缀的文件和所有文件两种类型。
- 第 6 行代码创建了“打开”文件对话框 dialog。os. getcwd（）获取当前文件所在的路径。
- 第 7 行代码以模式对话框的形式显示 dialog 对象。

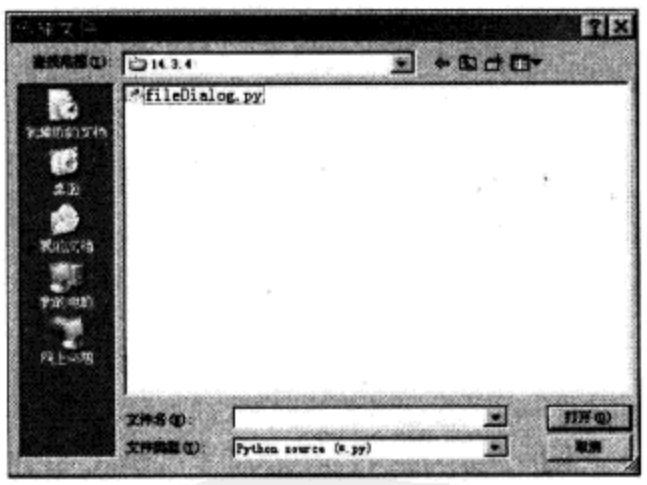


图 14-15　特殊对话框

【例 14-15】运行后的效果如图 14-15 所示。

此外，wx. ColourDialog 类可以用于创建颜色对话框，wx. FontDialog 类可以用于创建字体对话框。读者可以参考相关 API 自己尝试创建这些类型的对话框。

14.3.5　窗口与对话框的交互

对话框中的组件支持验证能力，而窗口不支持组件内容的验证。wx. PyValidator 类实现了组件的验证，在组件的 validator 选项中指定继承 wx. PyValidator 类的子类就实现了组件的验证功能。wx. PyValidator 类的 TransferToWindow( )和 TransferFromWindow( )这两个方法可以实现窗口和对话框的交互。TransferToWindow( )方法在对话框打开时执行，TransferFromWindow( )方法在对话框关闭时调用。通过这两个方法可以实现窗口和对话框的数据交换。

更多资源请访问稀酷客(www.ckook.com)

【例 14-16】创建 Frame 窗口，该窗口用于计算两个数字的和。单击“确定”按钮后把数字存放在字典中，并分别传递给对话框中的文本框。

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"对话框的验证", size=(300, 60))
        panel = wx.Panel(self, -1)
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        self.addTextCtrl = wx.TextCtrl(panel, -1, "", pos=(10,10))
        addStaticText = wx.StaticText(panel, -1, " + ")
        self.addTextCtrl2 = wx.TextCtrl(panel, -1, "")
        btn = wx.Button(panel, -1, u"计算")
        btn.Bind(wx.EVT_BUTTON, self.OnClick, btn)
        sizer.Add(self.addTextCtrl)
        sizer.Add(addStaticText)
        sizer.Add(self.addTextCtrl2)
        sizer.Add(btn)
        panel.SetSizer(sizer)
        panel.Fit()

    def OnClick(self, event):
        data = {0:self.addTextCtrl.GetValue(), 1:self.addTextCtrl2.GetValue()}
        dlg = MyDialog(data)
        dlg.ShowModal()
        dlg.Destroy()
```

- 第 12 行代码定义了“确定”按钮的处理函数 OnClick()。
- 第 20 行代码定义了字典 data，把两个文本框中的数字存放到字典中。

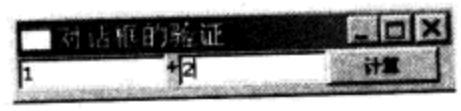


图 14-16　Frame 窗口

【例 14-16】运行后如图 14-16 所示。

【例 14-17】演示设计执行计算的对话框类，该对话框接收窗口定义的字典数据，然后进行计算。

```
class MyDialog(wx.Dialog):
    def __init__(self, data):
        wx.Dialog.__init__(self, None, -1, u"验证器")
        addStaticText  = wx.StaticText(self, -1, u"数字 1:")
        addStaticText2 = wx.StaticText(self, -1, u"数字 2:")
        self.addTextCtrl  = wx.TextCtrl(self, validator=DataValidator(data, 0))  # 添加验证
        self.addTextCtrl2 = wx.TextCtrl(self, validator=DataValidator(data, 1))
        btn = wx.Button(self, wx.ID_OK, u"确定")
        btn.Bind(wx.EVT_BUTTON, self.OnClick, btn)
        btn.SetDefault()
        sizer = wx.BoxSizer(wx.VERTICAL)
        gridSizer = wx.FlexGridSizer(2, 2, 5, 5)
        gridSizer.Add(addStaticText, 0, wx.ALIGN_LEFT)
        gridSizer.Add(self.addTextCtrl, 0, wx.EXPAND)
        gridSizer.Add(addStaticText2, 0, wx.ALIGN_LEFT)
        gridSizer.Add(self.addTextCtrl2, 0, wx.EXPAND)
        sizer.Add(gridSizer, 0, wx.EXPAND|wx.ALL, 5)
        sizer.Add(btn, 0, 5)
        self.SetSizer(sizer)
        sizer.Fit(self)

    def OnClick(self, event):
        result = float(self.addTextCtrl.GetValue()) + float(self.addTextCtrl2.GetValue())
        wx.MessageBox(str(result),u"结果")
        self.Close()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 6 行、第 7 行代码在两个文本框中添加验证功能，验证由 DataValidator 类处理，该类将接收字典 data。
- 第 9 行代码创建计算两个文本框的结果按钮。

【例 14-17】运行后，对话框的显示如图 14-17 所示。



图 14-17　计算验证的对话框

【例 14-18】用 DataValidator 类实现创建数据的接收。

```
class DataValidator(wx.PyValidator):                    # 对话框的验证器
    def __init__(self, data, key):
        wx.PyValidator.__init__(self)
        self.data = data
        self.key = key

    def Clone(self):                                    #Clone()方法必须实现
        return DataValidator(self.data, self.key)

    def Validate(self, w):                              # 验证数据
        return True

    def TransferToWindow(self):                         # 对话框打开时调用
        textCtrl = self.GetWindow()
        textCtrl.SetValue(self.data.get(self.key, ""))
        return True

    def TransferFromWindow(self):                       # 对话框关闭时调用
        return True
```

- 第 2 行代码定义了构造函数，用于接收 data 字典和相应的索引，并把参数的值赋给 DataValidator 的成员变量 data 和 key。
- 第 7 行代码覆盖了 Clone( )方法，该方法必须提供。Clone( )返回验证器的拷贝。
- 第 10 行代码用于验证提供 validator 选项的对话框组件。
- 第 14 行代码调用 GetWindow( )方法获得当前组件。
- 第 15 行代码通过索引 key 查找当前组件的值，即把窗口文本框的内容传递给对话框的文本框。
- 第 18 行代码 TransferFromWindow( )在对话框关闭时调用，这里返回“True”即可。

在 DataValidator 类的 validate( )方法可以实现对组件数据的验证，读者可以自己编写一些相关的验证规则来丰富对话框的验证功能。

14.4　小结

本章讲解了 wxPython 的菜单、窗口和对话框的相关知识。介绍了菜单、多级菜单、位图菜单和上下文菜单的创建，以及菜单事件的处理、快捷键的响应方式等内容。随后介绍了 wxPython 组件中的 Frame 窗口。Frame 窗口分为 MDIFrame、MiniFrame 等。最后介绍了对话框程序的编写，讲解了几种常见的对话框，包括模态对话框、非模态对话框、提示对话框、文本输入对话框、特殊对话框等。

更多资源请访问稀酷客(www.ckook.com)

第 15 章　WxPython 库中的高级控件

本章将介绍 wxPython 库的表格、列表、树、树表等可视化控件的使用方法。表格控件用于展示二维数据；列表控件提供了图标模式和详细列表模式来显示数据；树型控件用于展示存在父子关系的数据；树表控件结合了树型控件和列表控件的优点，用于显示更加复杂的数据。另外，还将介绍非可视化控件定时器，该控件用于定时控制程序的运行。本章介绍的控件比较复杂，但是 wxPython 提供了非常简单的 API 方法实现这些控件。本章的知识点：

- 表格控件
- 列表控件
- 树型控件
- 树表控件
- 定时器控件

15.1　表格控件

表格控件（Grid）是 wxPython 中最复杂的控件之一，表格控件通常用于浏览数据。本节将介绍表格控件的创建、单元格的设置和 PyGridTable 对象的使用方法等知识。

15.1.1　表格控件的创建

wxPython 的表格控件功能非常强大，提供了数据展示的方式。表格控件可以显示二维形式的数据。wxPython 提供了 wx. grid 包，wx. grid 包提供了创建和控制表格的类和方法。表格控件的创建一般分为如下几个步骤。

（1）创建 Grid 类的实例 grid。Grid 类的构造函数的参数与 wx. Window 的构造函数类似：

```
grid = wx.grid.Grid (self)
```

（2）创建表格控件，rowNum 表示表格控件的行数，colNum 表示表格控件的列数：

```
grid.CreateGrid (rowNum, colNum)
```

（3）设置行标题和列标题。rowData 表示行数据，colData 表示列数据：

```
grid.SetRowLabelValue (row, rowData)
grid.SetColLabelValue (row, colData)
```

（4）设置单元格的值。row 表示单元格的行号，col 表示单元格的列号，value 表示需要设置的值：

```
grid.SetCellValue (row, col, value)
```

【例 15-1】使用 Grid 类创建一个二维表格。

```
import wx
import wx.grid
```

更多资源请访问稀酷客(www.ckook.com)

```
class MyFrame(wx.Frame):
    def __init__(self):
        rowTitles = [u"第 1 行", u"第 2 行", u"第 3 行", u"第 4 行"]
        colTitles = [u"第 1 列", u"第 2 列", u"第 3 列", u"第 4 列"]
        wx.Frame.__init__(self, None, title=u"表格", size=(450,160))
        grid = wx.grid.Grid(self)                                    # 定义表格控件
        grid.CreateGrid(4, 4)
        for row in range(4):
            grid.SetRowLabelValue(row, rowTitles[row])              # 设置行标题
            grid.SetColLabelValue(row, colTitles[row])               # 设置列标题
            for col in range(4):
                grid.SetCellValue(row, col, str(col))                # 设置单元格的值

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
app.MainLoop()
```

- 第 6 行代码定义行标题，行标题存放在 rowTitles 列表中。
- 第 7 行代码定义列标题，列标题存放在 colTitles 列表中。
- 第 10 行代码调用 CreateGrid( )方法创建了 4 行 4 列的表格。

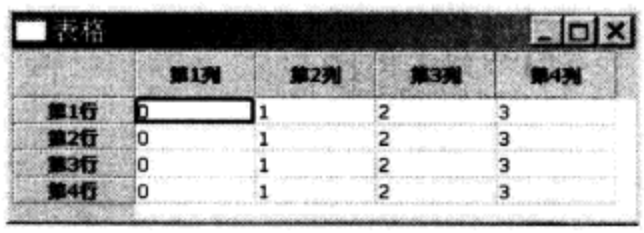


图 15-1　表格控件的创建

【例 15-1】运行后的效果如图 15-1 所示。

15.1.2　通过 PyGridTable 创建表格

表格控件还可以通过 PyGridTable 来创建，PyGridTable 可以定制更复杂的表格。通过继承 PyGridTableBase 类可以实现对表格控件更复杂的设置，继承 PyGridTableBase 类必须实现 GetAttr( )、GetNumberRows( )、GetNumberCols( )、GetValue( )和 SetValue( )这些方法。设置表格的奇数行和偶数行的颜色便于数据的浏览，是程序界面设计中经常需要实现的功能。

【例 15-2】演示 PyGridTable 创建区分奇数行和偶数行的表格控件。

```
import wx
import wx.grid as grid

class OddEvenTable(grid.PyGridTableBase):
    def __init__(self):
        grid.PyGridTableBase.__init__(self)
        self.odd = grid.GridCellAttr()                              # 获取奇数行
        self.odd.SetBackgroundColour("yellow")
        self.even = grid.GridCellAttr()                             # 获取偶数行

    def GetAttr(self, row, col, kind):                              # 对每个单元格进行控制
        attr = [self.even, self.odd][row % 2]
        attr.IncRef()
        return attr

    def GetNumberRows(self):
        return 10

    def GetNumberCols(self):
        return 10

    def GetValue(self, row, col):          # Grid 类的 SetTable()被调用时,此方法被调用
```

更多资源请访问稀酷客(www.ckook.com)

```
        return str(col)

    def SetValue(self, row, col, value):          #当对单元格设置值时,此方法被调用
        print (row, col, value)
```

- 第 4 行代码定义 OddEvenTable 类，继承了 PyGridTableBase 类。
- 第 8 行代码设置偶数行的背景为黄色。
- 第 11 行代码对奇、偶行进行过滤。
- 第 16 行代码 GetNumberRows( )方法返回表格的行数。
- 第 19 行代码 GetNumberCols( )方法返回表格的列数。
- 第 22 行代码 GetValue( )方法返回第 row 行第 col 列的值。
- 第 25 行代码 SetValue( )方法设置第 row 行第 col 列的值。当对单元格进行设置时，此方法将被调用。

【例 15-3】接上例。在主程序 MyFrame 类中调用 OddEvenTable 类设置表格的样式和内容。

```
class MyFrame(wx.Frame):
        def __init__(self):
            wx.Frame.__init__(self, None, -1, u"表格的方法", size=(900,250))
            gd = grid.Grid(self)
            table = OddEvenTable()
            gd.SetTable(table, True)

    if __name__=='__main__':
        app = wx.PySimpleApp()
        frame = MyFrame()
        frame.Show()
        app.MainLoop()
```

当 SetTable( )方法被调用时，将调用 OddEvenTable 类的 GetValue( )方法设置表格单元的内容。

【例 15-3】运行后的效果如图 15-2 所示。

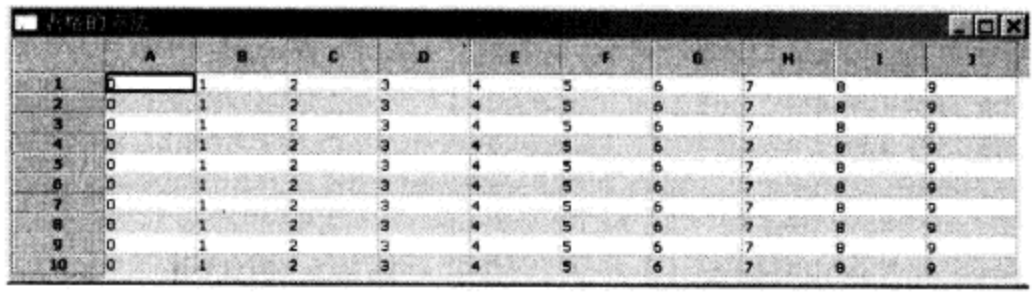


图 15-2　设置奇、偶行的表格控件

15.1.3　表格控件的单元格

表格控件的单元格可以根据不同的数据需求进行设置。例如，可以把某个单元格设置为下拉列表、复选框、数字选择等控件。wx. grid 包提供了一系列的 Editor 编辑器，这些编辑器可以在单元格中嵌套下拉列表、复选框、数字选择等控件。此外，单元格可以对数据的字体、颜色、格式进行设置。

【例 15-4】演示 wx. gird 包中提供的各种 Editor 编辑器的使用以及单元格样式的设置方法。

```
import wx
import wx.grid as grid

class SimpleGrid(grid.Grid):
    def __init__(self, parent):
```

更多资源请访问稀酷客(www.ckook.com)

```
grid.Grid.__init__(self, parent, -1)
self.CreateGrid(8, 4)
self.EnableEditing(True)
self.SetColLabelValue(0, u"第 1 列")                                    # 设置列标题
self.SetColLabelAlignment(wx.ALIGN_LEFT, wx.ALIGN_BOTTOM)
self.SetRowLabelValue(0, u"第 1 行")                                    # 设置行标题
self.SetRowLabelAlignment(wx.ALIGN_RIGHT, wx.ALIGN_BOTTOM)
self.SetCellValue(0, 0, "Hello")
self.SetCellTextColour(0, 0, wx.RED)                                    # 设置单元格的字体
self.SetCellFont(0, 0, wx.Font(10, wx.ROMAN, wx.ITALIC, wx.NORMAL))
self.SetCellValue(1, 0, "Read Only")
self.SetReadOnly(1, 0, True)                                            # 设置只读单元格
self.SetCellBackgroundColour(1, 0, wx.RED)          # 设置单元格的背景颜色
self.SetCellEditor(2, 0, grid.GridCellNumberEditor(1, 10))  # 嵌入数字选择控件
self.SetCellValue(2, 0, "1")
self.SetCellEditor(2, 1, grid.GridCellFloatEditor(0, 1))  # 嵌入浮点数编辑控件
self.SetCellValue(2, 1, "10.19")
attr = grid.GridCellAttr()
attr.SetTextColour(wx.BLUE)
self.SetColAttr(1, attr)                                # 设置一列的单元格属性
self.SetCellAlignment(5, 1, wx.ALIGN_CENTRE, wx.ALIGN_BOTTOM)
self.SetCellValue(5, 1, u"跨行、列的单元格")
self.SetCellSize(5, 1, 2, 2)                                            # 设置跨行、列的单元格
self.SetCellEditor(6, 0, grid.GridCellChoiceEditor(["one", "two", "three"]))
                                                                        # 嵌入下拉列表
self.SetCellValue(6, 0, "one")
self.SetCellEditor(7, 0, grid.GridCellBoolEditor())     # 嵌入复选列表
self.SetCellValue(7, 0, "True")

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"单元格的设置", size=(500,250))
        self.grid = SimpleGrid(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码调用 EnableEditing( )方法，使单元格可以被编辑。
- 第 9 行代码到第 11 行代码设置表格第 1 列的标题，并设置标题的对齐方式为左下对齐。
- 第 11 行代码到第 13 行代码设置表格第 1 行的标题，并设置标题的对齐方式为右下对齐。
- 第 14 行代码设置第 1 行第 1 列的单元格字体为红色。
- 第 16 行代码设置第 1 行第 1 列的单元格为罗马字体，并设置斜体。
- 第 19 行代码设置第 2 行第 1 列单元格的状态为“只读”。
- 第 19 行代码设置第 2 行第 1 列单元格的背景颜色为红色。
- 第 20 行代码，在第 3 行第 1 列使用数字选择编辑器 GridCellNumberEditor（1，10），创建可从 1 到 10 范围内选择的编辑器。
- 第 22 行代码，在第 3 行第 2 列使用浮点数字编辑器 GridCellFloatEditor（0，1），浮点小数精度为 1。当编辑数字 10.19 时，编辑器会自动进行“四舍五入”，把数字改为 10.2。
- 第 24 行到第 26 行代码获取属性对象 attr，并调用 attr 的 SetTextColour( )方法设置字体的颜色为蓝色，最后调用 SetColAttr（1，attr）设置第 2 列的字体颜色为蓝色。

更多资源请访问稀酷客(www.ckook.com)

- 第 28 行到第 29 行代码设置从第 6 行第 2 列开始的跨两行、两列的单元格。
- 第 30 行到第 31 行代码，在第 7 行第 1 列设置 GridCellChoiceEditor 编辑器，该编辑器包括“one”、“two”、“three” 3 个选项。下拉列表的默认项为“one”。当编辑该单元格时，将显示下拉列表。
- 第 32 行到第 33 行代码，在第 8 行第 1 列设置 GridCellBoolEditor 编辑器，默认值为“True”。当编辑该单元格时，将显示复选框控件。



图 15-3　单元格的 Editor 编辑器

【例 15-4】运行后的效果如图 15-3 所示。

15.1.4　处理单元格的事件

表格控件支持大部分的鼠标和键盘响应事件，这些事件可以对某个单元格进行操作。

【例 15-5】实现鼠标的单击、右击、单元格选择和键盘输入等事件。单元格的事件处理原理和其他控件的处理方法相同，首先绑定单元格的事件，指定事件处理函数，然后实现事件处理函数。

```
import wx
import wx.grid as grid
import pprint

class MyGrid(grid.Grid):
    def __init__(self, parent):
        grid.Grid.__init__(self, parent, -1)
        self.CreateGrid(5, 5)
        # 表格的事件
        self.Bind(grid.EVT_GRID_CELL_LEFT_CLICK, self.OnCellLeftClick)          # 当单击鼠标左键时触发
        self.Bind(grid.EVT_GRID_CELL_RIGHT_CLICK, self.OnCellRightClick)        # 当单击鼠标右键时触发
        self.Bind(grid.EVT_GRID_SELECT_CELL, self.OnSelectCell)                 # 当选择单元格时触发
        self.Bind(grid.EVT_GRID_RANGE_SELECT, self.OnRangeSelect)               # 当选择多个单元格时触发
        self.Bind(grid.EVT_GRID_CELL_CHANGE, self.OnCellChange)                 # 当单元格内容改变时触发
        # 当编辑单元格时触发的事件
        self.Bind(grid.EVT_GRID_EDITOR_SHOWN, self.OnEditorShown)
        self.Bind(grid.EVT_GRID_EDITOR_HIDDEN, self.OnEditorHidden)
        self.Bind(grid.EVT_GRID_EDITOR_CREATED, self.OnEditorCreated)
        # 响应键盘输入
        self.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)

    def OnCellLeftClick(self, evt):
        print "(%d, %d)" % (evt.GetRow(), evt.GetCol())
        evt.Skip()

    def OnCellRightClick(self, evt):
        wx.MessageBox("%s" % (evt.GetPosition()), u"提示")
        evt.Skip()

    def OnRangeSelect(self, evt):
        if evt.Selecting():
            print "(%s, %s)" % (evt.GetTopLeftCoords(), evt.GetBottomRightCoords())
        evt.Skip()

    def OnCellChange(self, evt):
        value = self.GetCellValue(evt.GetRow(), evt.GetCol())
        print value

    def OnSelectCell(self, evt):
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.SetCellBackgroundColour(evt.GetRow(), evt.GetCol(), wx.RED)
        evt.Skip()

    def OnEditorShown(self, evt):
        print "show"
        evt.Skip()

    def OnEditorHidden(self, evt):
        print "hidden"
        evt.Skip()

    def OnEditorCreated(self, evt):
        print evt.GetControl()
        evt.Skip()

    def OnKeyDown(self, evt):
        value = evt.GetPosition()
        pp = pprint.PrettyPrinter()
        pp.pprint(value)
        print "KeyCode:" + str(evt.GetKeyCode())
        evt.Skip()

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"表格的事件", size=(500,180))
        self.grid = MyGrid(self)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 10 行代码绑定单元格的单击事件 EVT_GRID_CELL_LEFT_CLICK。当鼠标左键单击时触发该事件。
- 第 11 行代码绑定单元格的右击事件 EVT_GRID_CELL_RIGHT_CLICK。当鼠标右键单击时触发该事件。
- 第 12 行代码绑定单元格的选择事件 EVT_GRID_SELECT_CELL。当选择某个单元格时触发该事件。
- 第 13 行代码绑定单元格的多选事件 EVT_GRID_RANGE_SELECT。当选择多个单元格时触发该事件。
- 第 14 行代码绑定单元格内容的改变事件 EVT_GRID_CELL_CHANGE。当编辑单元格的内容时触发该事件。
- 第 16 行到第 18 行代码绑定编辑器相关的事件 EVT_GRID_EDITOR_SHOWN、EVT_GRID_EDITOR_HIDDEN、EVT_GRID_EDITOR_CREATED。当显示单元格编辑器时，将触发 EVT_GRID_EDITOR_SHOWN 事件。当隐藏单元格编辑器时，将触发 EVT_GRID_EDITOR_HIDDEN 事件。当某种类型的单元格编辑器被第一次调用时，将触发 EVT_GRID_EDITOR_CREATED 事件。
- 第 20 行代码绑定单元格的 EVT_KEY_DOWN 事件，当键盘按下时触发该事件。

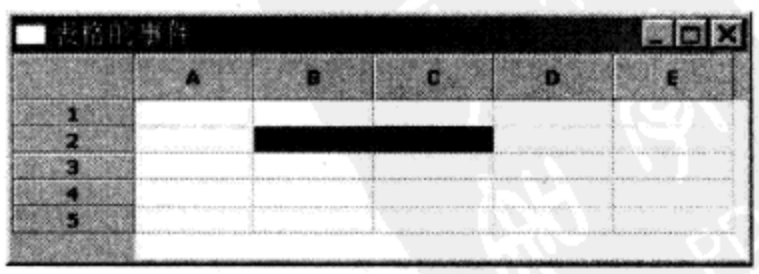


图 15-4　表格的事件

【例 15-5】运行后的效果如图 15-4 所示。

更多资源请访问稀酷客(www.ckook.com)

15.2　高级列表控件

高级列表控件（ListCtrl）是数据显示的另一种方式。ListCtrl 控件可以提供详细列表、图标列表等显示方式。ListCtrl 控件的列对齐、排序等辅助功能可以通过继承 wx. lib. mixins. listctrl 包的 mixin 类实现。

15.2.1　列表控件的创建

列表控件 ListCtrl 的作用类似于第 13 章中介绍的 ListBox 控件。ListCtrl 控件采用列表的方式显示信息，ListCtrl 的功能比 ListBox 控件更强大。ListCtrl 控件可以对列表项进行排列，并能够定制列表的显示样式。例如详细列表模式、图标模式等。列表控件的创建有两种方法。第一种方法是直接实例化 wx. ListCtrl 类，然后关联列表数据。第二种方法是自定义列表控件类，继承 wx. ListCtrl 或 wx. lib. mixins. listctrl 包中的 mixin 类。这些 mixin 类实现了一些辅助性的功能，例如列表列的对齐、列表列的排序等。

【例 15-6】使用上面所介绍的第二种方法实现类似 Windows 系统中详细列表样式的列表。这里可以继承 wx. ListCtrl 类和 wx. lib. mixins. listctrl. ListCtrlAutoWidthMixin 类来实现详细列表的功能，其中 ListCtrlAutoWidthMixin 类实现了对列表列的自动对齐功能。

```
import sys
import wx
import wx.lib.mixins.listctrl as listmix

data = {1 : ("1", "listctrl", u"列表控件"),
        2 : ("2", "grid", u"表格控件"),
        3 : ("3", "tree", u"树控件"),
        4 : ("4", "timer", u"定时器控件") }

class MyListCtrl(wx.ListCtrl, listmix.ListCtrlAutoWidthMixin):
    def __init__(self, parent, ID, pos=wx.DefaultPosition, size=wx.DefaultSize, style=0):
        wx.ListCtrl.__init__(self, parent, ID, pos, size, style)
        listmix.ListCtrlAutoWidthMixin.__init__(self)   # 使列的宽度与 ListCtrl 的宽度自动对齐
        self.setColumns()

    def setColumns(self):
        # 定义列头
        self.InsertColumn(0, u"第 1 列")
        self.InsertColumn(1, u"第 2 列")
        self.InsertColumn(2, u"第 3 列")
        # 设置各列中的值
        items = data.items()
        for key, values in items:
            index = self.InsertStringItem(sys.maxint, values[0])
            for i in range(len(values)):
                self.SetStringItem(index, i, values[i])
        # 设置列的宽度
        self.SetColumnWidth(0, 100)
        self.SetColumnWidth(1, 100)
        # 第 3 列的宽度自动对齐
        self.SetColumnWidth(2, wx.LIST_AUTOSIZE)
```

- 第 5 行代码定义了字典 data 作为列表控件的数据。
- 第 10 行代码定义了 MyListCtrl 类，该类继承了 wx. ListCtrl 和 listmix. ListCtrlAutoWidthMixin类。

更多资源请访问稀酷客(www.ckook.com)

- 第 14 行代码调用自定义的 setColumns( )函数设置列表数据。
- 第 18 行到第 20 行代码调用 InsertColumn( )设置列表的列头。
- 第 24 行代码调用 InsertStringItem( )方法在列表中添加一行，返回行索引 index。
- 第 26 行代码调用 SetStringItem( )方法设置第 index 行第 i 列单元格的内容。
- 第 28 行、第 29 行代码设置第 1 列和第 2 列的列宽为 100。
- 第 31 行代码自动调整第 3 列的宽度为列表控件剩余的宽度。

注意　自定义的 MyListCtrl 类继承了 ListCtrlAutoWidthMixin 类，SetColumnWidth( )方法中的第 2 个参数设置为 wx. LIST_AUTOSIZE 才有效。

【例 15-7】接上例。在 MyFrame 类中实例化 MyListCtrl 类，设置 ListCtrl 类的 style 为 wx. LC_REPORT，wx. LC_REPORT 表示详细信息列表。

```
class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title=u"列表控件", size=(400, 200))
        sizer = wx.BoxSizer(wx.VERTICAL)
        self.list = MyListCtrl(self, -1, style=wx.LC_REPORT)
        sizer.Add(self.list, 1, wx.EXPAND)
        self.SetSizer(sizer)
```

【例 15-7】运行后的效果如图 15-5 所示。

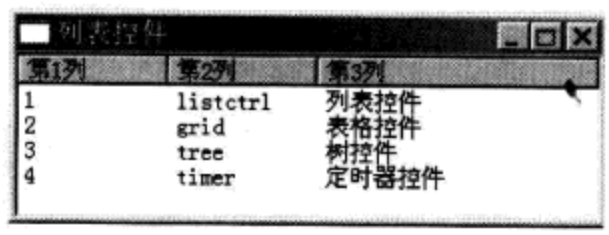


图 15-5　详细列表

15.2.2　列表控件的排序

Windows 操作系统的详细列表可以对列进行排序，ListCtrl 控件也可以添加排序功能，当单击列表中的列标题时将对当前列的数据进行排序。如果要对 ListCtrl 控件进行排序，需要在 MyFrame 类中继承 ColumnSorterMixin 类，而不是在 MyListCtrl 类中继承 ColumnSorterMixin 类。因为排序是需要对列表控件的每列进行排序，所以是在 MyListCtrl 控件的容器类 MyFrame 中添加排序功能。

【例 15-8】接上例。将【例 15-7】中的 MyFrame 类修改为如下所示。

```
class MyFrame(wx.Frame, listmix.ColumnSorterMixin):
    def __init__(self):
        wx.Frame.__init__(self, None, title=u"列表控件", size=(400, 200))
        sizer = wx.BoxSizer(wx.VERTICAL)
self.list = MyListCtrl(self, -1, style=wx.LC_REPORT
                               |wx.BORDER_NONE
                               |wx.LC_SORT_ASCENDING)
        sizer.Add(self.list, 1, wx.EXPAND)
        self.itemDataMap = data                                    # 用于排序的字典数据
        listmix.ColumnSorterMixin.__init__(self, len(data))
        self.SetSizer(sizer)
        self.SetAutoLayout(True)

    def GetListCtrl(self):
        return self.list
```

- 第 5 行代码实例化 MyListCtrl 类，设置列表控件为详细列表，并按照升序排序。
- 第 9 行代码把字典 data 赋值给 ListCtrl 对象的 itemDataMap 属性。itemDataMap 属性为字

更多资源请访问稀酷客(www.ckook.com)

典类型，用于对列表数据的排序。

- 第 10 行代码调用父类 ColumnSorterMixin 类的构造函数，给列表添加排序功能。ColumnSorterMixin 类的构造函数的调用必须放在 MyListCtrl 类的构造函数和 itemDataMap 属性的赋值之后。只有对 itemDataMap 属性赋值并添加了列表数据以后，才能初始化排序类。
- 第 14 行代码实现了 GetListCtrl( )方法，该方法用于返回创建的列表。

修改 MyFrame 类后的【例 15-8】运行后的效果如图 15-6 所示。

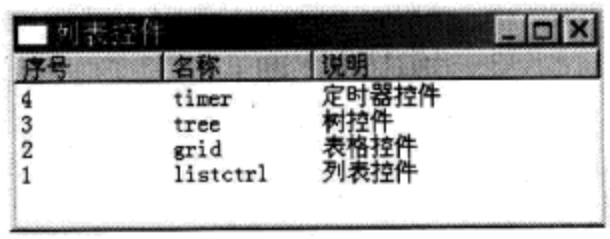


图 15-6　排序后的详细列表

注意　MyFrame 类继承了 ColumnSorterMixin 类后，必须实现 GetListCtrl( )方法返回列表对象。此外必须给 ItemDataMap 属性赋值。

15.2.3　带位图的列表控件

带位图的列表控件类似于 Windows 操作系统中的图标模式。位图列表的创建分为如下几个步骤。

（1）创建 ImageList 对象存储图标资源 wx. Bitmap 对象。

（2）创建 ListCtrl 控件，并设置 sytle 为 wx. LC_ ICON。

（3）调用 AssignImageList( )方法关联 ImageList 对象。

（4）调用 InsertImageStringItem( )方法插入列表项，并设置图标的说明字符串。

【例 15-9】使用 15.2.1 节中所介绍的创建 ListCtrl 控件的第一种方法创建位图列表。

```
import wx
import sys, glob

data = {0: "prior", 1: "next", 2: "house", 3: "save", 4: "setting"}

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, u"带位图的列表", size=(200,180))
        il = wx.ImageList(16, 16, True)                          # 创建图像列表
        for name in glob.glob("image/icon?.png"):                # 搜索 image 子目录中的图标
            bmp = wx.Bitmap(name, wx.BITMAP_TYPE_PNG)
            il.Add(bmp)
        self.list = wx.ListCtrl(self, -1, style=wx.LC_ICON | wx.LC_AUTOARRANGE)
        self.list.AssignImageList(il, wx.IMAGE_LIST_NORMAL)      # 把图像列表关联到列表控件中
        for x in range(5):
            self.list.InsertImageStringItem(x, data[x], x)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 4 行代码定义了列表数据 data，data 为字典类型，存放了列表图标的说明字符串。
- 第 9 行代码定义 ImageList 类的实例 il，用于存放 16 * 16 像素的图片。
- 第 10 行代码调用 glob( )方法搜索 image 子目录中 PNG 格式的图片，glob( )方法支持字符串的匹配，“icon? . png”表示匹配以“icon”字符串开始的 PGN 格式的文件名。
- 第 11 行代码根据匹配的文件名创建 wx. Bitmap 对象。
- 第 12 行代码把 wx. Bitmap 对象添加到 ImageList 列表中。

更多资源请访问稀酷客(www.ckook.com)

- 第 13 行代码创建 ListCtrl 控件，wx. LC_ICON 表示位图列表，wx. LC_AUTOARRANGE 表示列表项根据 Frame 窗口的大小自动排列。
- 第 14 行代码把 ImageList 关联到列表控件中。
- 第 16 行代码调用 InsertImageStringItem( )，把 data 字典的内容添加到图标下面。

【例 15-9】运行后的效果如图 15-7 所示。



图 15-7　位图列表

15.3　树型控件

树型控件用于展示有父子关系的数据。本节将介绍树型控件的创建、事件的处理以及树表控件的创建和使用方法。

15.3.1　树型控件的创建

树型控件采用分层的方式显示复杂数据，数据呈现父子结构的关系。例如 Windows 操作系统中的资源管理器的左侧就是一个目录树。树型控件展示了数据的层次关系，是桌面应用程序经常使用的控件之一。树型控件的创建一般分为如下几个步骤。

（1）创建 TreeCtrl 类的实例。下面这行代码创建了树型对象 tree：

```
tree = wx.TreeCtrl(self)
```

（2）调用 TreeCtrl 类的 AddRoot( )添加根节点。下面这行代码添加了根节点 root，根节点的名称为“root”：

```
root = tree.AddRoot("root")
```

（3）自定义递归函数，递归添加子节点。调用 AppendItem（parent，node）添加子节点。parent 表示父节点，node 表示子节点。

【例 15-10】遍历以二叉树形式组成的列表 data，创建一个多级树型控件。二叉树的每个节点最多包含两个子节点，因此 data 列表中第二层子列表也包含两个元素。其中第一个元素表示父元素，第二个元素表示父元素所包含的子元素。

```
import wx

data = ["1",
       ["2", ["2-1", "2-2", ["2-3", ["2-3-1", "2-3-2"]]]],
       ["3", ["3-1", "3-2"]],
       ["4", ["4-1", "4-2", "4-3"]],
        "5"]

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title=u"树型控件", size=(200, 300))
        self.tree = wx.TreeCtrl(self)                                    # 创建树型控件
        root = self.tree.AddRoot("root")                                 # 添加根节点
        self.AddTreeNodes(root, data)                                    # 添加子节点
        self.tree.Expand(root)                                           # 展开根节点

    def AddTreeNodes(self, parent, tree):
        for node in tree:
            if type(node) == str:                                        # 当前节点为子节点时直接添加
                self.tree.AppendItem(parent, node)
            else:
                item = self.tree.AppendItem(parent, node[0])
```

更多资源请访问稀酷客(www.ckook.com)

```
                self.AddTreeNodes(item, node[1])                    # 递归添加子节点

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = TestFrame()
    frame.Show()
    app.MainLoop()
```

- 第 3 行代码定义了多维列表 data，data 列表采用多层结构的形式组织。data 列表作为树型控件的数据。
- 第 15 行代码展开树型控件的 root 根节点。
- 第 17 行代码自定义 AddTreeNodes( )递归函数，用于在树型控件中添加子节点。遍历 data 列表中的数据项，如果当前节点的类型为字符串类型，则直接向父节点中添加。如果当前节点的类型为列表类型，则调用 AppendItem（parent，node［0］），添加列表的第 1 个元素，即父元素。然后递归调用 AddTreeNodes（item，node［1］）继续遍历。

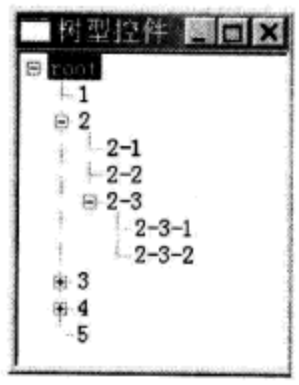


图 15-8　树型控件

【例 15-10】运行后的效果如图 15-8 所示。

15.3.2　树型控件的事件

树型控件的事件主要有节点的选择事件、展开事件、合并事件等。当树型控件创建完成后，直接绑定相关的事件类型，然后实现相应的处理函数即可。

【例 15-11】实现树型控件的常用事件。

```
import wx

data = ["1",
        ["2", ["2-1", "2-2", ["2-3", ["2-3-1", "2-3-2"]]]],
        ["3", ["3-1", "3-2"]],
        ["4", ["4-1", "4-2", "4-3"]],
         "5"]

class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title=u"树型控件", size=(300, 200))
        self.tree = wx.TreeCtrl(self)
        root = self.tree.AddRoot("root")
        self.AddTreeNodes(root, data)
        # 事件响应
        self.Bind(wx.EVT_TREE_ITEM_EXPANDED, self.OnItemExpanded, self.tree)
        self.Bind(wx.EVT_TREE_ITEM_COLLAPSED, self.OnItemCollapsed, self.tree)
        self.Bind(wx.EVT_TREE_SEL_CHANGED, self.OnSelChanged, self.tree)
        self.tree.Expand(root)

    def AddTreeNodes(self, parent, tree):
        for node in tree:
            if type(node) == str:                                   # 当前节点为子节点时直接添加
                self.tree.AppendItem(parent, node)
            else:
                item = self.tree.AppendItem(parent, node[0])
                self.AddTreeNodes(item, node[1])                    # 递归添加子节点

    def OnItemExpanded(self, evt):
        wx.MessageBox("OnItemExpanded:
    " + self.tree.GetItemText(evt.GetItem()), u"提示")
```

更多资源请访问稀酷客(www.ckook.com)

```
        def OnItemCollapsed(self, evt):
wx.MessageBox("OnItemCollapsed:" + self.tree.GetItemText(evt.GetItem()), u"提示")

    def OnSelChanged(self, evt):
        wx.MessageBox("OnSelChanged:" + self.tree.GetItemText(evt.GetItem()), u"提示")

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = TestFrame()
    frame.Show()
    app.MainLoop()
```

- 第 16 行代码绑定节点的展开事件 wx.EVT_TREE_ITEM_EXPANDED，处理函数为 OnItemExpanded()。
- 第 17 行代码绑定节点的合并事件 wx.EVT_TREE_ITEM_COLLAPSED。处理函数为 OnItemCollapsed()。
- 第 18 行代码绑定节点的选择事件 wx.EVT_TREE_SEL_CHANGED。处理函数为 OnSelChanged()。
- 第 30 行代码，当展开节点时，调用 OnItemExpanded()函数返回操作节点的文本内容。首先调用 Event 对象的 GetItem()方法返回当前操作的节点对象，然后调用 GetItemText()方法获取节点对象的文本内容。

15.3.3　树表控件

树表控件用于展示既存在父子关系，又需要按列显示的数据。树表控件是 TreeCtrl 控件和 wx.LC_ REPORT 风格的 ListCtrl 控件混合体。树表控件中的树型结构存放在列表的第 1 列中。树表控件同时具备 TreeCtrl 控件和 ListCtrl 控件的属性和方法。wx.gizmos 包中的 TreeListCtrl 类可以创建树表控件，树表控件的创建分为以下几个步骤。

（1）创建 TreeListCtrl 类的实例。下面这行代码创建了 tree 对象：

```
tree = wx.gizmos.TreeListCtrl()
```

（2）调用 SetImageList()方法设置树表控件中节点的图标。下面这行代码中的 il 表示 ImageList 图片列表：

```
tree.SetImageList(il)
```

（3）调用 AddColomn()方法添加树表的列。

（4）调用 AddRoot()方法添加树表的根节点，即在第 1 行第 1 列单元格中添加根节点。

（5）调用 SetItemText()方法设置树表的列标题或单元格的内容。下面这行代码设置第 1 行第 2 列单元格的内容为空：

```
tree.SetItemText(self.root, "", 1)
```

（6）调用 AppendItem()方法在根节点下添加子节点。下面这行代码在根节点 root 下添加子节点 value：

```
tree.AppendItem(self.root, value)
```

（7）调用 SetItemText()方法在子节点下添加子孙节点。调用 SetItemImage()方法设置树结构的展开、合并连线图标。下面这行代码在第 1 列 child 节点中添加字符串为 value 的子节点：

```
tree.SetItemText(child, value, 0)
```

更多资源请访问稀酷客(www.ckook.com)

【例 15-12】创建树表控件。

```
import wx
import wx.gizmos as gizmos

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title=u"树表控件", size=(400, 400))
        # 创建树表控件
        self.tree = gizmos.TreeListCtrl(self, -1, style = wx.TR_DEFAULT_STYLE
                                        |wx.TR_FULL_ROW_HIGHLIGHT)
        self.il = wx.ImageList(16, 16, True)
        # 给树表添加图标
        self.il.Add(wx.ArtProvider_GetBitmap(wx.ART_FOLDER, wx.ART_OTHER, (16,16)))
        self.il.Add(wx.ArtProvider_GetBitmap(wx.ART_FILE_OPEN, wx.ART_OTHER, (16,16)))
        self.il.Add(wx.ArtProvider_GetBitmap(wx.ART_NORMAL_FILE, wx.ART_OTHER, (16,16)))
        self.tree.SetImageList(self.il)
        # 添加树表的列
        self.tree.AddColumn(u"第 1 列")
        self.tree.AddColumn(u"第 2 列")
        self.tree.AddColumn(u"第 3 列")
        self.tree.SetColumnWidth(0, 186)
        self.root = self.tree.AddRoot("root")
        self.tree.SetItemText(self.root, "", 1)                    # 设置第 2 列第 1 行单元格的文本
        self.tree.SetItemText(self.root, "", 2)                    # 设置第 3 列第 1 行单元格的文本
        for x in range(5):                                        # 设置子节点的文本和图标
            child = self.tree.AppendItem(self.root, str(x))
            self.tree.SetItemText(child, str(x), 0)
            self.tree.SetItemText(child, str(x), 1)
            self.tree.SetItemText(child, str(x), 2)
            self.tree.SetItemImage(child, 0, which = wx.TreeItemIcon_Normal)
            self.tree.SetItemImage(child, 1, which = wx.TreeItemIcon_Expanded)
            for y in range(5):
                last = self.tree.AppendItem(child, str(y))
                self.tree.SetItemText(last, str(x) + "-" + str(y), 0)
                self.tree.SetItemText(last, str(x) + "-" + str(y), 1)
                self.tree.SetItemText(last, str(x) + "-" + str(y), 2)
                self.tree.SetItemImage(last, 0, which = wx.TreeItemIcon_Normal)
                self.tree.SetItemImage(last, 1, which = wx.TreeItemIcon_Expanded)
        self.tree.Expand(self.root)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

- 第 8 行代码创建 TreeListCtrl 类，wx.TR_FULL_ROW_HIGHLIGHT 表示当单击树表的某个节点时，则选中节点所在的行。
- 第 12 行到第 14 行代码把 wxPython 自带的位图类 wx.ArtProvider_GetBitmap 添加到 ImageList 图片列表中。
- 第 15 行代码把位图添加到树表控件中。
- 第 17 行到第 19 行代码为树表添加列。
- 第 24 行到第 37 行代码为树表的单元格添加内容。

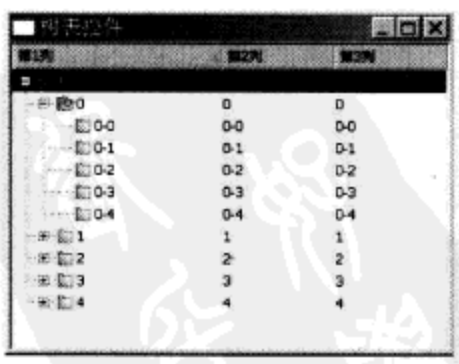


图 15-9　数表控件

【例 15-12】运行后的效果如图 15-9 所示。

15.4　定时器控件

定时器控件是非可视化的控件，用于定时控制程序的运行。许多程序语言都提供了对定时器

更多资源请访问稀酷客(www.ckook.com)

控件的支持，wxPython 可以使用 wx. PyTimer、wx. Timer 和 wx. CallLater 类创建定时器控件。定时器类提供了 3 个主要的方法。Start（mil）方法用于启动定时器，参数 mil 表示毫秒。Stop( ) 方法用于停止定时器。Restart( )方法用于重新启动定时器程序，进行新一轮的处理。wx. Timer 用于被其他类继承，并通过覆盖特定的方法实现对时间的控制。wx. CalLater 类可以向处理程序传递参数，并返回处理程序的计算结果。

【例 15-13】 使用 wx. PyTimer 和 wx. CallLater 类创建定时器控件。通过 wx. PyTimer 类的延时功能在任务栏中显示计时器，通过 wx. CallLater 类创建数字计数器，并传递一些参数给处理程序。

```
import wx
import time

class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, size=(320, 100))
        panel = wx.Panel(self, -1)
        sb = self.CreateStatusBar(2)
        sb.SetStatusWidths([100, 220])
        self.count = 0
        self.timer = wx.PyTimer(self.Notify)                          # 创建定时器
        self.timer.Start(1000)                                        # 设置间隔时间
        self.inputText = wx.TextCtrl(panel, -1, "", pos=(10,10), size=(50, -1))
        self.inputText2 = wx.TextCtrl(panel, -1, "", pos=(10,10), size=(50, -1))
        btn = wx.Button(panel, -1, u"带参数的定时器")
        btn2 = wx.Button(panel, -1, u"停止")
        self.Bind(wx.EVT_BUTTON, self.OnStart, btn)
        self.Bind(wx.EVT_BUTTON, self.OnStop, btn2)
        sizer = wx.FlexGridSizer(cols=4, hgap=10, vgap=10)
        sizer.Add(self.inputText)
        sizer.Add(self.inputText2)
        sizer.Add(btn)
        sizer.Add(btn2)
        panel.SetSizer(sizer)
        panel.Fit()

    def OnStart(self, evt):
        self.timer2 = wx.CallLater(1000, self.OnCallTimer, 1, 2, 3)       # 带参数的定时器

    def OnStop(self, evt):
        self.timer2.Stop()                                                # 停止定时器
        self.inputText2.Value = str(self.timer2.GetResult())              # 返回参数的计算结果
        del self.timer2

    def OnCallTimer(self, *args, **kw):
        self.count = self.count + 1
        self.inputText.Value = str(self.count)
        tup = args
        total = 0
        for x in tup:
            total = total + int(x)
        self.timer2.Restart(1000, total)                                  # 重新启动定时器
        return total

    def Notify(self):
        t = time.localtime(time.time())
        self.SetStatusText(u"定时器", 0)
        self.SetStatusText(time.strftime("%d-%B-%Y %I:%M:%S", t), 1)

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

更多资源请访问稀酷客(www.ckook.com)

- 第 8 行代码创建了任务栏，并把任务栏分为 2 栏。
- 第 9 行代码调用 SetStatusWidths（[100, 220]），设置任务栏两栏的宽度分别为“100”、“220”。
- 第 11 行代码调用 wx. PyTimer（self. Notify），Notify( )为定时器的处理函数，实现了对程序的定时控制。
- 第 12 行代码调用 Start（1000）启动定时器，并设置时间间隔为 1 秒。
- 第 17 行代码为按钮 btn 绑定处理函数 OnStart( )，该函数创建了 wx. CallLater 定时器。
- 第 18 行代码为按钮 btn2 绑定处理函数 OnStop( )，该函数用于停止定时器。
- 第 28 行代码启动 wx. CallLater 定时器，时间间隔为 1 秒。定时器把处理过程委托给 OnCallTimer( )函数实现，并向该函数传递 3 个参数 1、2、3。
- 第 32 行代码调用 wx. CallLater 定时器的 GetResult( )方法返回 OnCallTimer( )的计算结果。
- 第 33 行代码删除 wx. CallLater 定时器。
- 第 35 行代码的 OnCallTimer( )处理函数主要实现了三个功能。第一，通过 self. count 变量进行计数，每秒对变量加 1。第二，对传递的 3 个参数执行加法计算，并把结果返回给定时器。第三，调用 Restart( )方法重新启动定时器，进行下一轮对 OnCallTimer( )的调用。
- 第 45 行代码的 Notify( )处理函数获取当前的时间，并把时间转换为字符串类型，最后输出到任务栏中。

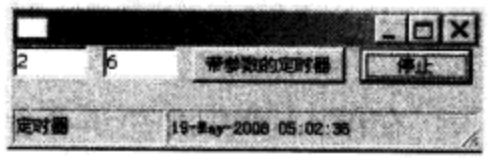


图 15-10　定时器

【例 15-13】运行后的效果如图 15-10 所示。

15.5　小结

本章讲解了 wxPython 中高级控件的创建、事件和方法。介绍了表格控件的两种创建方法：使用 Grid 类的 CreateGrid( )方法直接创建和通过 PyGridTable 系列类创建。介绍了两种风格的列表控件：详细信息列表和图标模式的列表，并实现了对详细信息列表的排序操作。重点介绍了树型控件和树表控件的创建，树表控件是一种功能强大的组件，实现了列表和树结合的方式展现数据。最后介绍了定时器控件，讲解了 wx. PyTimer、wx. CallLater 类创建定时器的方法。

更多资源请访问稀酷客(www.ckook.com)

第 16 章　WxPython 库中的高级功能

除了在前面介绍过的各种图形用户界面控件外，WxPython 还提供了一些高级的功能，包括内置的 HTML 显示、XRC 文件和对系统剪贴板的使用等。本章的知识点：

- WxPython 下的 HTML 显示
- XRC 文件的使用
- 系统剪贴板的操作

16.1　WxPython 下的 HTML 显示

在第 13 章中介绍了在窗口中加入文本控件，但是那些文本控件只能支持有限的几种文本格式。当需要用更多的格式显示文本时，文本控件就无能为力了。WxPython 提供了用来处理 HTML 格式的模块 wx. html，可以在 WxPython 中处理 HTML 格式的文档。

16.1.1　HTML 的显示

在 wx. html 中有一个 HtmlWindow 类，可以用来创建包含 HTML 内容的窗口。通过调用 SetPage 方法，可以指定需要显示的 HTML 内容。

【例 16-1】在 WxPython 应用程序中显示一个简单的 HTML 文档内容。

```
#filename: wx_html_1.py
#encoding = utf - 8
import wx
import wx.html as html

class MyFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, u"简单 HTML 窗口")
        htmlwin = html.HtmlWindow(self)

        htmlwin.SetPage(u"""<h1>一级标题</h1>
                        <b>这是 HTML 加粗文本</b>
                        <br></br>
                        <i>这是 HTML 斜体文本</i>""")

app = wx.PySimpleApp()
frame = MyFrame(None)
frame.Show()
app.MainLoop()
```

- 第 4 行导入了 wx. html 模块。后面的代码就可以使用此模块中的相关类和方法。这里将主要使用 HtmlWindow。
- 在 MyFrame 的构造函数中，调用 wx. html 模块中的 HtmlWindow 类生成了一个 HTML 窗口。接着设置了此 HTML 窗口中的内容。

更多资源请访问稀酷客(www.ckook.com)

【例 16-1】运行后的结果如图 16-1 所示。

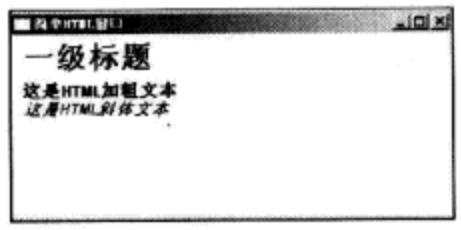


图 16-1　简单的 HTML 窗口

由于 HtmlWindow 是 ScrolledWindow 的子类，所以在默认情况下，当 HtmlWindow 中的内容超出了显示区域的时候，将会自动开启滚动条。HTML 窗口中的内容支持大部分的 HTML 标签。

【例 16-2】显示一个更复杂的 HTML 窗口。

```
#filename: wx_html_2.py
#encoding = utf - 8
import wx
import wx.html as html

class MyFrame(wx.Frame):
    def __init__(self, parent): #生成窗口
        wx.Frame.__init__(self, parent, -1, u"统计窗口", size = (300, 300))
        panel = wx.Panel(self, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        htmlwin = html.HtmlWindow(panel, -1, style = wx.NO_BORDER)
        htmlwin.SetPage(page)

        vbox.Add((-1, 10), 0)
        vbox.Add(htmlwin, 1, wx.EXPAND | wx.ALL, 9)
        bitmap = wx.StaticBitmap(panel, -1, wx.Bitmap('./wxPython.jpg'))
        hbox.Add(bitmap, 1, wx.LEFT | wx.BOTTOM | wx.TOP, 10)
        buttonOk = wx.Button(panel, wx.ID_CLOSE, u'确定')
        self.Bind(wx.EVT_BUTTON, self.OnClose, id = wx.ID_CLOSE)
        hbox.Add((100, -1), 1, wx.EXPAND | wx.ALIGN_RIGHT)
        hbox.Add(buttonOk, flag = wx.TOP | wx.BOTTOM | wx.RIGHT, border = 10)
        vbox.Add(hbox, 0, wx.EXPAND)
        panel.SetSizer(vbox)
        self.Centre()

    def OnClose(self, event):
        self.Close()

app = wx.App()
frame = MyFrame(None)
frame.Show(True)
app.MainLoop()
```

【例 16-2】的代码和【例 16-1】的实际区别并不大，只是这里的窗口稍微复杂一点。另外，这段代码也是在 MyFrame 类的构造函数中生成了一个 HtmlWindow 窗口。然后使用了 SetPage 方法来设置 HTML 内容。此函数中的 page 变量的值如下所示：

```
page = u'<html > <body bgcolor = "#8e8e95" > <table cellspacing = "5" border = "0" width = "250" > \
<tr width = "200" align = "left" > \
<td bgcolor = "#e7e7e7" >  特征量 </td > \
<td bgcolor = "#aaaaaa" >   <b > 数值 </b > </td > \
</tr > \
<tr width = "200" align = "left" > \
<td bgcolor = "#e7e7e7" >  最大值 </td > \
```

更多资源请访问稀酷客(www.ckook.com)

```
<td bgcolor = "#aaaaaa" >   <b >9000 </b > </td > \
</tr > \
<tr align = "left" > \
<td bgcolor = "#e7e7e7" >  最小值 </td > \
<td bgcolor = "#aaaaaa" >   <b >3000 </b > </td > \
</tr > \
<tr align = "left" > \
<td bgcolor = "#e7e7e7" >  中间值 </td > \
<td bgcolor = "#aaaaaa" >   <b >5000 </b > </td > \
</tr > \
<tr align = "left" > \
<td bgcolor = "#e7e7e7" >  平均值 </td > \
<td bgcolor = "#aaaaaa" >   <b >5666 </b > </td > \
</tr > \
</body > </table > </html >'
```

运行【例 16-2】，可以得到如图 16-2 所示的窗口。



图 16-2　更复杂的 HTML 窗口

16.1.2　基于 HtmlWindow 的简单浏览器

实际上，可以通过使用 HtmlWindow 来实现一个简单的浏览器。在上一节的两个示例代码中，使用了 SetPage 方法来加载 HTML 的文档内容。另外，还可以通过 AppendToPage 方法来将 HTML 内容加到已创建好的 HtmlWindow 窗口中。上面的方法只是能够加载已经是 HTML 格式的内容。在 WxPython 中，提供了另外两个函数可以用来加载本地的 HTML 文档和远程的 URL。这两个函数是 LoadFile 和 LoadPage。通过这两个函数可以实现一个简单浏览器的功能。

【例 16-3】演示 LoadPage 函数的使用方法。

```
#filename: wx_html_3.py
#encoding = utf - 8
import wx
import wx.html as html

class MyFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, u"简单 HTML 浏览器")
        htmlwin   = html.HtmlWindow(self)

        htmlwin.LoadPage("http://www.google.com")

app = wx.PySimpleApp()
frame = MyFrame(None)
frame.Show()
app.MainLoop()
```

【例 16-3】代码段和前面两个示例的主要区别在于使用了 LoadPage 方法。这里加载了 http：//www. google. com 网站的主页。运行【例 16-3】，将得到如图 16-3 所示的窗口。

从图 16-3 中可以看出，这个简单浏览器窗口中显示了大部分的页面元素。但是更进一步观

更多资源请访问稀酷客(www.ckook.com)

察，会发现它与一般的浏览器窗口还是有区别的。所以这个 HtmlWindow 的浏览器版本只是实现了网页中一些比较简单的功能。



图 16-3　简单的浏览器窗口

16.2　WxPython 中的 XRC

在图形用户界面应用程序的开发中，实际上大部分的工作都是在进行界面设计。使用 XRC（XML Resource）可以将界面设计的工作从应用程序的开发中分离开来。也就是说，将界面的设计使用一种特定格式的 XML 文件来指定，这样将使得应用程序更加清晰可读，同时也分离了界面和逻辑设计。本节将对 WxPython 中 XRC 的使用方法进行介绍。

16.2.1　使用 XRC

XRC 实现了逻辑处理和界面设计的分离。这种思想在很多设计中都是有先例的，最著名的是在网页设计中，将页面的显示使用 CSS 样式表。在 WxPython 的界面设计中也是如此，可以将界面代码从 WxPython 应用程序中的逻辑处理中分离开来。尤其是当界面比较复杂的时候，这种分离带来的好处更加明显。这种处理方式带来的另外一个好处是可以通过设计不同的 XRC 文件来变换界面，因为 XRC 中只有对界面元素的描述。

当使用 XRC 文件来描述的时候，源程序可以变得比较简单和清晰。

【例 16-4】使用 XRC 文件来生成界面。

```
#filename: xrc_1.py
#encoding = utf - 8
import wx
from wx import xrc

class MyApp(wx.App):
    def OnInit(self):
        self.res = xrc.XmlResource('clipboard.xrc') #从 XRC 文件中读取界面描述
        self.init_frame()
        return True

    def init_frame(self):
        self.frame = self.res.LoadFrame(None, 'MyFrame2')
        self.panel = xrc.XRCCTRL(self.frame, 'm_panel1')
        self.text1 = xrc.XRCCTRL(self.panel, 'm_textCtrl4')
        self.text2 = xrc.XRCCTRL(self.panel, 'm_textCtrl5')
        self.frame.Bind(wx.EVT_BUTTON, self.OnCopy, id = xrc.XRCID('m_button6')) #绑定事件
        self.frame.Bind(wx.EVT_BUTTON, self.OnPaste, id = xrc.XRCID('m_button7')) #绑定事件
        self.frame.Bind(wx.EVT_BUTTON, self.OnQuit, id = xrc.XRCID('m_button8')) #绑定事件
        self.frame.Show()

    def OnQuit(self, event):
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.frame.Close(True)

    def OnCopy(self, event):
        pass

    def OnPaste(self, event):
        pass

if __name__ == '__main__':
    app = MyApp(False)
    app.MainLoop()
```

- 在第 4 行从 wx 中导入了 xrc 模块。这使得可以读取 XRC 文件。
- 在 MyApp 类的 OnInit 方法中，首先使用了 XmlResource 类从指定的文件中读取界面描述，然后接下来调用了 init_frame 方法。
- 在 init_frame 方法中，从 XRC 文件中获取了特定窗口空间名称的对象。其中有些读取出来的对象这个代码段中并没有使用。接着对窗口中的按钮进行了事件绑定。
- 接下来在 MyApp 类中定义了 3 个事件处理函数。其中 OnQuit 被调用的时候，窗口将退出。而另外两个函数则暂时为空函数。

16.2.2　XRC 文件

在【例 16-4】的代码中加载了 clipboard.xrc 文件，此文件中描述了窗口界面元素。下面是此界面资源的 XML 文件：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<resource xmlns="http://www.wxwindows.org/wxxrc" version="2.3.0.1">
    <object class="wxFrame" name="MyFrame2">
        <style>wxDEFAULT_FRAME_STYLE|wxTAB_TRAVERSAL</style>
        <size>420,300</size>
        <title></title>
        <object class="wxPanel" name="m_Panel1">
            <object class="wxFlexGridSizer">
                <rows>2</rows>
                <cols>2</cols>
                <vgap>0</vgap>
                <hgap>0</hgap>
                <growablecols></growablecols>
                <growablerows></growablerows>
                <object class="sizeritem">
                    <option>0</option>
                    <flag>wxALL</flag>
                    <border>5</border>
                    <object class="wxTextCtrl" name="m_textCtrl4">
                        <size>300,100</size>
                        <value>Hello, World!</value>
                        <maxlength>0</maxlength>
                    </object>
                </object>
                <object class="sizeritem">
                    <option>0</option>
                    <flag>wxALL</flag>
                    <border>5</border>
                    <object class="wxButton" name="m_button6">
                        <label>复制</label>
                        <default>0</default>
                    </object>
                </object>
```

更多资源请访问稀酷客(www.ckook.com)

```
            <object class = "sizeritem" >
                <option >0 < /option >
                <flag >wxALL < /flag >
                <border >5 < /border >
                <object class = "wxTextCtrl" name = "m_textCtrl5" >
                    <size >300,100 < /size >
                    <value > < /value >
                    <maxlength >0 < /maxlength >
                < /object >
            < /object >
            <object class = "sizeritem" >
                <option >0 < /option >
                <flag >wxALL < /flag >
                <border >5 < /border >
                <object class = "wxButton" name = "m_button7" >
                    <label >粘贴 < /label >
                    <default >0 < /default >
                < /object >
            < /object >
            <object class = "sizeritem" >
                <option >0 < /option >
                <flag >wxALL < /flag >
                <border >5 < /border >
                <object class = "wxButton" name = "m_button8" >
                    <label >退出 < /label >
                    <default >0 < /default >
                < /object >
            < /object >
        < /object >
    < /object >
< /object >
< /resource >
```

从上面的文件中可以看到，此 XRC 文件采用了 XML 的表示方法。其中一个 < object > 节点表示一个窗口控件，而下面的子节点则为此窗口控件的属性。上面的 XRC 文件可以按照其文件格式手工生成。但通常可以采用 XRC 界面工具来生成。例如，在第 11 章中介绍的界面生成工具就可以完成这样的工作。图 16-4 显示了使用 wxFormBuilder 工具来生成 GUI 界面。

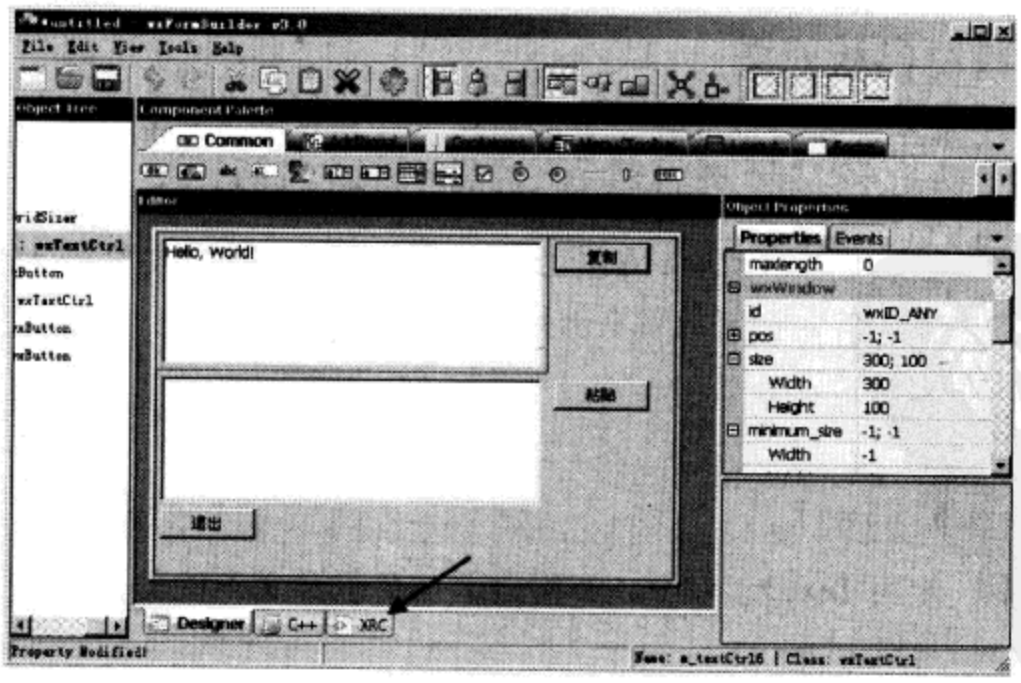


图 16-4　wxFromBuilder 中的 XRC 生成

在图 16-4 中显示了 GUI 界面的设计。单击图中箭头所示的“XRC”标签，就可以得到这个窗口的 XRC 描述内容。

更多资源请访问稀酷客(www.ckook.com)

16.3　剪贴板的操作

剪贴板是在各个应用程序之间交换数据的一种方式。一个应用程序可以将数据保存在剪贴板中，另外一个应用程序可以从剪贴板中获取数据。保存在剪贴板中的数据可以是文本、图像或者是其他的文件对象等。本节将对 WxPython 如何操作系统剪贴板进行介绍。

16.3.1　WxPython 中的剪贴板对象

在大部分的应用程序中，都提供了如剪切、复制和粘贴的操作。这些操作实际上就是在和系统的剪贴板打交道。剪切是将源对象放置在剪贴板中，而源对象消失。复制同剪切类似，但是源对象并不消失。粘贴是指将剪贴板中的对象放置在目标对象中。从这里可以看出，对于和系统剪贴板相关的操作，可以包含三个操作元素：源对象、目标对象和剪贴板。这里的源对象和目标对象不一定是处于同一个应用程序中。

在 WxPython 中，保存在剪贴板中的数据为 wx. DataObject 对象。实际上在此对象中包含用来描述所包含数据的格式元信息。此信息使得应用程序可以知道如何处理剪贴板数据。例如在一个文本处理程序中，可以让用户选择粘贴文本的格式。在 WxPython 中内置有 3 个 wx. DataObject 对象：wx. TextDataObject、wx. BitmapDataObject 和 wx. FileDataObject。分别用来表示文本对象、图像对象和文件对象。实际上，这也是系统剪贴板中最常见的对象。

由于操作系统中的剪贴板是全局性的，即对于不同的应用程序都是相同的，所以在 WxPython 应用程序中剪贴板对象 wx. TheClipboard 是一个全局对象。在操作剪贴板之前，需要通过调用 Open 方法打开此对象。只有在此方法返回了“True”之后才能够继续使用剪贴板。这是因为剪贴板是全局性的对象，当其正在被另外一个应用程序操作的时候，对此剪贴板的操作有可能失败。此时 Open 方法将会返回“False”。所以在操作剪贴板之前，需要检查 Open 方法的返回值。当然，也可以使用剪贴板对象的 IsOpened 方法来判断。在操作剪贴板结束之后，需要调用 Close 方法将其关闭。因为使用剪贴板将会阻止其他应用程序对剪贴板的访问，所以对剪贴板的操作时间应该尽可能的短。

16.3.2　处理剪贴板对象

通过使用 wx. DataObject 得到了剪贴板对象之后，就可以使用它来在不同的应用程序之间交互数据了。全局对象 wx. TheClipboard 提供了可以用来访问其保存对象的方法。SetData 方法可以用来将源对象放置在剪贴板中，GetData 方法可以将剪贴板中的对象放置到目的对象中去。

【例 16-5】演示如何将剪贴对象放置在剪贴板中。

```
text = wx.TextDataObject("Hello world!")
if wx.TheClipboard.Open():
    wx.TheClipboard.SetData(text)
    wx.TheClipboard.Close()
```

- 在第 1 行中，使用 TextDataObject 类构建了一个包含文本的剪贴对象，其中的文本内容为“Hello world!”。
- 在第 2 行中，调用 Open 方法来判断剪贴对象是否可以打开。如果剪贴板对象当前可用，则调用 SetData 方法将前面的剪贴对象放置在剪贴板中，并接着马上调用 Close 方法断开和剪贴板对象的连接。这里的代码并没有处理当剪贴板对象正在使用的情况。

运行【例 16-5】后，系统剪贴板中就有一段文本数据了。

更多资源请访问稀酷客(www.ckook.com)

当系统剪贴板中有可以剪贴的数据时，应用程序使用 GetData 方法可以获取其中的数据。此方法的参数为所需要放置剪贴板数据的对象，为 wx. DataObject 类或其子类的实例化对象。通过此方法的返回值可以知道系统的剪贴板中是否含有和参数格式一致的数据。当返回值为“True”的时候，剪贴板中的数据将会放置在参数所表示的对象中。

【例 16-6】演示如何从剪贴板中获取数据。

```
text = wx.TextDataObject()
success = False
if wx.Open():
    success = wx.TheClipboard.GetData(text)
    wx.TheClipboard.Close()
if success:
    print text.GetText()
```

- 在第 1 行中首先使用 TextDataObject 类生成了一个可以容纳文本数据的剪贴板对象。
- 在第 2 行中设置了一个标志位，用来指示从系统剪贴板中获取数据是否成功。
- 在第 3 行中使用 Open 方法检查剪贴板是否可以使用。当可以访问的时候，则调用 GetData 方法来从系统剪贴板中获取文本内容的数据。当此方法返回“True”的时候，可以调用文本内容剪贴板对象的 GetText 方法来获取其文本数据。

如果将【例 16-6】的代码接在【例 16-5】的代码后面执行，将会输出“Hello world!”。

需要注意的是，上面代码的运行必须是在应用程序对象初始化之后，否则 wx. TheClipboard 对象将为“None”，从而得不到预想的结果。wx. TheClipboard 对象除了上面介绍的几个方法以外，还可以使用 Clear 方法来清除剪贴板中的数据。另外，当应用程序将数据保存在剪贴板中的时候，只有在剪贴板对象调用了 Flush 方法后，数据才能够在此应用程序退出后继续保存在剪贴板中。否则，该剪贴板对象会在应用程序退出时销毁。

当剪贴数据为图像的时候，可以采用 wx. BitmapDataObject 类的对象。类中有 GetBitmap 和 SetBitmap 方法用来获取和设置对象。需要注意的是，这里和剪贴板交互的数据需要是 wx. Bitmap 类型。另外在 WxPython 中定义的剪贴板数据类型是 wx. FileDataObject 类。可以通过使用 GetFilenames 方法来从此对象中获取文件名。

16.3.3　剪贴板操作实例

【例 16-7】接【例 16-4】。完善【例 16-4】代码中的 OnCopy 和 OnPaste 方法。

```
#filename: clipboard_1.py
#encoding = utf - 8
import wx
from wx import xrc

class MyApp(wx.App):
#省略部分代码
    def OnCopy(self, event): #复制数据到剪贴板中
        data = wx.TextDataObject()
        data.SetText(self.text1.GetValue())
        if wx.TheClipboard.Open():
            wx.TheClipboard.SetData(data)
            wx.TheClipboard.Close()
        else:
            wx.MessageBox("不能打开剪贴板", "错误")

    def OnPaste(self, event): #从剪贴板中读取数据
        data = wx.TextDataObject()
        success = False
        if wx.TheClipboard.Open():
```

更多资源请访问稀酷客(www.ckook.com)

```
            success = wx.TheClipboard.GetData(data)
            wx.TheClipboard.Close()
        else:
            wx.MessageBox("不能打开剪贴板", "错误")

        if success:
            self.text2.SetValue(data.GetText())
        else:
            wx.MessageBox("格式不匹配", "错误")

if __name__ == '__main__':
    app = MyApp(False)
    app.MainLoop()
```

- 以上代码中主要完善了 OnCopy 和 OnPaste 方法。其中 OnCopy 方法中首先生成了一个 TextDataObject 对象，并复制为 text1 文本控件中的值。接着调用 wx.TheClipboard 的 SetData 方法将此对象放置在剪贴板对象中。
- 在 OnPaste 方法中，同样的也是生成了一个 TextDataObject 对象，然后使用 GetData 方法从剪贴板中获取了文本数据。当格式匹配的时候，将 text2 文本控件的内容设置为此值。

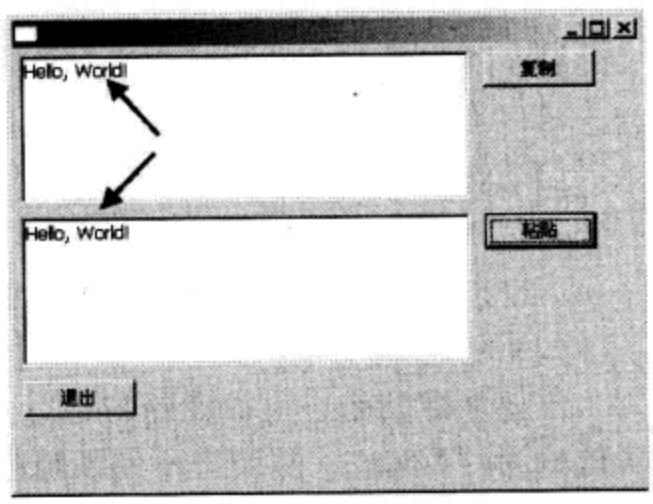


图 16-5　剪贴板对象的使用

运行【例 16-7】，将得到如图 16-5 所示的界面。

在窗口刚生成的时候，仅仅只是上半部的文本控件中才有特定的字符串。当单击了【复制】按钮后，此字符串将会复制到系统的剪贴板中。此时，如果在其他应用程序（如记事本）中使用粘贴命令，则此字符串将会传送到该应用程序中。在此窗口中，单击【粘贴】按钮，系统剪贴板中的数据将会出现在窗口下半部的文本控件中。

16.4　小结

本章介绍了 WxPython 下的高级功能。首先对 WxPython 下的 HTML 显示进行了介绍，包括基本 HTML 内容的显示，以及使用 wx.HtmlWindow 构建了 WxPython 中的一个简单浏览器。接着介绍了将应用程序中的界面设计和逻辑处理分离的 XRC，包括其文件格式和使用方法。最后对系统剪贴板的操作进行了讲解，主要包括剪贴板对象 wx. TheClipboard 中数据的写入和读取。

更多资源请访问稀酷客(www.ckook.com)

第三篇
Python 的 Web 开发

第 17 章　Python 的 HTML 应用

随着互联网的发展和 HTTP 协议的普及，HTML 已成为一种最热门和最重要的语言之一。熟练地掌握 HTML 语言，才能够对现在的各种网页结构有个清晰的认识。Python 的标准库中提供了众多处理 HTML 的模块。本章的知识点：

- HTML 的标签
- URL 的处理，包括解析、拼合和分解等
- URL 的编码与解码
- urllib/urllib2 模块的使用方法
- 使用 httplib 模块获取资源
- HTMLParser 模块的使用方法
- sgmllib. py 的使用方法
- htmllib 高级模块的使用方法

17.1　HTML 介绍

HTML（Hyper Text Markup Language，超文本标记语言）是一种标记语言。这种语言提供了在互联网资源之间自由访问的方法。

17.1.1　HTML 的历史

HTML 最初是由 Tim Berners-Lee 在欧洲量子物理实验室创建的。1982 年，Tim Berners-Lee 为了使世界各地的物理学家能够方便的进行合作研究，建立了 HTML。这时候 HTML 以纯文本格式为基础，可以使用任何文字编辑器处理。最初仅有少量标记，简单且易于掌握运用。随着 HTML 使用率的增加和 Internet 的发展，人们对于 HTML 规范标准的需求也在增加。

HTML 是由国际组织万维网联盟（W3C）来维护的。HTML 4.0 加入了很多特定浏览器的元素和属性，同时也开始清理其标准，把一些元素和属性标记为过时的，建议不再使用它们。HTML 的未来和 CSS 结合会更好。版本 4.0 是现在使用最多的 HTML 版本。Python 中的模块支持也主要是以版本 4.0 为基础。

HTML 版本 5 目前仍为草案，已被 W3C 接纳。它主要提供了一些新的元素和属性。例如，

更多资源请访问稀酷客(www.ckook.com)

增加的标签 < nav > 和 < footer > 将有利于搜索引擎的索引整理。另外还为其他浏览要素通过一个标准接口提供新的功能，如 < audio > 和 < video > 等标记。在 HTML5 中，将有着新的解析顺序和新的通用属性等。这是一项长期的工作，仍需要多年的努力。

读者如果需要了解 HTML 更详细的信息，访问 http：//www. w3c. org 网站可以了解最新的 HTML 规范。

17. 1. 2　SGML、HTML 和 XHTML 三者的关系

在网络发展的早期，为了能够解决网络环境下文档共享的问题，一种称作标准通用标记语言（SGML，Standard Generalized Markup Language）的专用语言被发明出来，于 1986 年被 ISO 组织发布为国际标准。SGML 将重点放在一个文档中的组成部分上，从而使得信息的接收者可以不受信息发送者的约束。实际上这是一种通用的文档结构描述标记语言，主要用来定义文献模型的逻辑和物理类结构。

一个 SGML 语言程序由 3 部分组成，包括语法定义、文件类型定义和文件实例。语法定义中定义了文件类型和文件实例的语法结构。文件类型定义中定义了文件实例的结构和组成结构的元素类型。而文件实例是 SGML 语言程序的主体部分。

虽然早期的 HTML 和 SGML 之间并没有直接关系，但是随着复杂度的增加，HTML 已经成为了 SGML 的一种派生语言。因为在 SGML 的实际使用中，每一个特定的 DTD 都定义了一类文件。其用途很广，包括科研、金融等行业都可以使用。而将用于万维网的语言称为 HTML 语言。SGML 是标记语言的标准，也就是说所有标记语言都是依照 SGML 制定的，当然包括HTML。在 Python 中，处理 HTML 的库有的是基于处理 SGML 库的，也是因为这个原因。

HTML 的特点是能够处理结构化文档结构、字形字体、版面布局、链接等超文本文档结构，可以使浏览器阅读和重新格式化 HTML 页面。因为互联网的发展，所以使其成为了一种最有名的 SGML 格式。由于其快速的发展，现在的 HTML 语言规范非常的臃肿，内容和格式表现都放在一起，不利于文档的分析操作。为此，XHTML 被提出来了。

XHTML（Extensible Hypertext Markup Language）是 HTML 和 XML 的结合。XML 将在第 18 章中进行介绍。简单地说，由于 SGML 的过度复杂，在实际应用中很多的特性都用不到，XML 则是 SGML 的简化版，省略了其中复杂和不常用的部分。

XHTML 是一个基于 XML 的标记语言，看起来与 HTML 基本相似，只有一些小的但重要的区别。这是一个比 HTML 更加严格的语言，其中使用了许多的 XML 规则。本质上说，XHTML 是一种过渡技术，结合了 XML 的强大功能及 HTML 的简单特性。虽然现在 XHTML 给 HTML 设计者带来了一些困难，但这是以后网页编制的标准。在 Python 中处理 HTML 语言时，如果使用 XHTML，将会使得运行更加流畅。

17. 1. 3　HTML 的标签

HTML 是由许多元素构成的，元素是一个 HTML 文档结构的基本组成部分。元素包括页头（head）、表格（table）、段落（p）和列表（li）等。浏览器将输出 HTML 组成的各个元素。元素中可以包含文本、链接、图片或者其他元素。

HTML 文档使用标签来表示各种元素。标签由一个左尖括号（ < ）和一个右尖括号（ > ）包围的标签名组成。标签一般是成对出现的，用来指出标签作用的范围。结束标签和起始标签相似，只是在结束标签括号中的标签名中以斜杠（/）开头。在 HTML 的规范中，有些标签并没有结束标签。实际上，在浏览器解析的时候，会自动将其加上结束标签后再进行处理。虽然如此，还是推荐使用符合 W3C 规范的网页书写方式。

更多资源请访问稀酷客(www.ckook.com)

HTML 的标签大部分都含有许多属性，这些属性主要用来指定标签元素的相关信息。例如指定超链接的 URL 地址。属性都是放在开始标签中的。每个标签都有自己特定的属性。由于 HTML 标准的发展，各个浏览器对于标签和属性的支持也有很大的差异。在书写 HTML 文档的时候，为了能够适应特定的浏览器，需要查询特定浏览器的标签和属性支持情况。

HTML 规范中的标签众多，其中有部分已经被推荐不再继续使用。总的说来，现在的规范中主要包含这样几种类型的标签：结构性标签、呈现性标签、超文本链接标签和框架页面标签。除此之外，HTML 文档中可以使用“<！—这是注释-->”的方式来加入注释信息。

17.1.4　HTML 的框架组成

HTML 文件可以使用任何的文本编辑器来进行编辑。虽然可以在浏览器中看到 HTML 中的图片或者不同格式的文本，但是实际上 HTML 文档本身只是很普通的文本文件。当然使用“所见即所得”的网页编辑器会有助于加速网页开发，但这不是必需的。

【例 17-1】一个比较简单的 HTML 文件。

```
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <title>标题栏:第一个 HTML 页面</title>
</head>
  <body>
<h2>第一个 HTML 页面</h2>
Hello, <b>World</b>
<! ---这是注释-->
<p>
可以从
<a href="http://www.w3c.org">W3C</a>
网站上找到 HTML 的语言规范
<br>&copy;2009
  </body>
  </html>
```

在任一文本编辑器中输入上面的 HTML 源文件，并以 first.html 为文件名保存。确保文件是使用 .html 格式来保存的，而不是其他格式（如 Windows 下的 doc 格式等）。打开 IE 浏览器，在地址栏中输入 HTML 文件的路径及其名称，显示如图 17-1 所示。

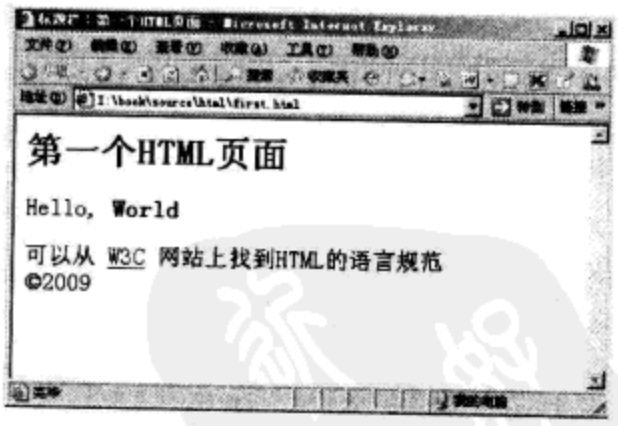


图 17-1　简单的 HTML 页面

HTML 文件开始于 <html> 并结束于 </html> 标签。所有的其他 HTML 内容都将会在此 HTML 文件之间。虽然 HTML 和 XHTML 标准要求每个 HTML 文档都需要在外部有一个 <html> 标签，但是现在大部分的浏览器在没有此标签的情况下，也能够正常显示。

除了框架网页外，所有的 HTML 和 XHTML 文件包含 head 和 body 这两个部分。可以将文档的一些相关信息放在 head 段中，而将文档的内容放在 body 段中。body 段的内容将会在浏览器中得到显示，所以网页编制者大部分工作都是在处理 body 段中的内容。

在 head 段中有几个不同的标签，用来指示一个具体的文档信息。例如在【例 17-1】中，使用了 <title> 标签，用来显示文档的标题“第一个 HTML 页面”。每种浏览器都有一定的 head 段标签支持，有些标签甚至还可以使得 HTML 文档产生动画效果。

在【例 17-1】的 HTML 文档中出现的标签还有 <h2>、<b>、<a>、<p> 和
 等。在这个简单的 HTML 文档中，除了没有出现框架标签外，其他的如结构性标签、呈现性标签、

更多资源请访问稀酷客(www.ckook.com)

超链接标签和注释都出现了。对于每种标签，浏览器都会做相应的解释。例如对 < h2 > 标签，浏览器使用了加大一号字体进行显示。而对于 < p > 标签，则在页面上进行了换行。

HTML 文档的最大创新和核心是超文本。超文本使得用户可以通过单击超链接来转到另一个相关的页面。通过这种跳转方式，用户可以获取感兴趣的信息，包括 HTML 文档或者其他的多媒体资源。

17.2　URL 的处理

统一资源定位符是 Internet 上用来定位资源的基础。本节首先描述其概念，然后进一步对在 Python 中如何解析 URL 和获取 URL 资源进行介绍。和本节内容有关系的主要是 urlparse 和 urllib 这两个模块。它们分别用作 URL 的解析、拼合和其资源的获取。

17.2.1　统一资源定位符 URL

HTML 利用统一资源定位符（Universal Resource Locator，URL）来定位 Internet 上的 HTML 文档信息。在 Internet 上的每个文档和资源都有着唯一的一个资源定位符。URL 的语法定义如下：

```
协议://授权/路径？查询
```

这里，协议部分给出了信息的交换方式，比较常用的协议如表 17-1 所示。

表 17-1　常用的协议

协议名称	协议含义
http	超文本传送协议
https	使用安全套接层的超文本传送协议
ftp	文件传输协议
mailto	电子邮件
file	本地电脑上的文件
telnet	Telnet 协议

当前 Internet 上用得最多的还是 http 协议。各种协议的使用频率依照表 17-1 中由上至下的顺序依次递减。请注意，file 协议主要是用于显示本地电脑上的文件，如前面的【例 17-1】文件。

授权部分包括服务器的名称或者其 IP 地址。如果还有端口号的话，则需要使用冒号分隔开。路径部分包含 HTML 的具体路径信息，一般来说使用“/”符号进行分隔。查询部分一般是传递给服务器上的数据库进行动态查询所需要的参数。

一种包含授权的统一资源定位符的语法如下：

```
协议://用户名@ 密码:子域名．域名．顶级域名:端口号/目录/文件名．文件后缀？参数 = 值#标志
```

上面是一个比较完整的统一资源定位符。一般的 URL 都是在上面的基础上的一些简化。下面是一些 URL 示例。

（1）http://www. w3. org/2005/10/Process-20051014/activities. html

（2）ftp://ftp. test. com/pub/index. txt

（3）first. html

（4）../images/logo. jpg

在上面的示例 URL 中，前面两个使用的是绝对地址，这是因为这两个地址都是上面 URL 语

更多资源请访问稀酷客(www.ckook.com)

法的一个简化。两个 URL 都包含协议、服务器地址等信息。对于这样的 URL，浏览器知道使用什么样的协议和去哪里获取相关的信息。在后面的两个 URL 中没有指定访问协议，甚至没有指定服务器地址。这样的 URL 称为相对 URL。当浏览器遇到这样的 URL 时，会将协议默认为 http，而将服务器地址设置为当前地址。

统一资源定位符一般来说是区分大小写的，但是服务器的管理员可以设置是否对此加以区分。有些服务器在收到不同大小写的查询时给出的回复是相同的。不过，在实际操作和网页编写中，最好还是对大小写进行区分。

在 HTML 文档中，通过 < a > 标签来定义一个超链接，在其 href 属性中可以定义一个目的 URL。而在 < a > 标签中的内容将会作为单击的对象。对于一般的浏览器，都会将这些超链接显示为一定的格式。通常的做法是在下面加上下划线并使用不同的颜色。

17.2.2　URL 的解析

Python 中用来对 URL 字符串进行解析的模块是 urlparse。这是在 Python 版本 2.2 中开始引入的。在该模块中除了 urlparse 方法外，其他主要使用的方法还包括 urljoin、urlsplit 和urlunsplit等。

在 Python 语言中，urlparse 对于 URL 的定义是采用的六元组，如下所示：

```
scheme://netloc/path;parameters? query#fragment
```

此元组中的每个值都是字符串。在 URL 中，不存在的元组为空值。这里，有些组成部分没有进一步区分，如网络地址部分，并没有进一步区分服务器地址和端口地址，而仅仅都是作为一个字符串来表示的。

在解析 URL 的时候，所有的“%”转义符都不会被处理。另外，除了在路径当中的第一个起始斜线（如果有的话）外，分隔符将会被去掉。

表 17-2 描述了使用 urlparse 方法返回的对象的属性。

表 17-2　urlparse 方法返回对象中的属性

属性	索引值	值含义	默认值
scheme	0	协议	空字符串
netloc	1	服务器地址	空字符串
path	2	路径	空字符串
params	3	参数	空字符串
query	4	查询部分	空字符串
fragment	5	分片部分	空字符串
username		用户名	None
password		密码	None
hostname		主机名	None
port		端口	None

注意　在表 17-2 中，netloc 属性值实际上包含了此表中的最后 4 个属性值。

urlparse 方法还有两个可选的参数：default_scheme 和 allow_fragments。

default_scheme 主要是用来为不包含协议部分的 URL 指定默认协议。此参数的默认值为空字符串。allow_fragments 则用来指示是否可以对地址进行分片。此参数的默认值为“True”。

更多资源请访问稀酷客(www.ckook.com)

【例 17-2】使用 urlparse 解析 URL（注意到其中使用了 IPython 工具。该工具的使用方法将在第 22 章进行介绍）。

```
In [1]: import urlparse

In [2]: r = #此处因为排版而断行
   urlparse.urlparse('http://alice:secret@ www.hostname.com:80/%7Ealice/python.cgi?query=
text#sample')

In [3]: r
Out[3]:
('http',
'alice:secret@ www.hostname.com:80',
'/%7Ealice/python.cgi',
'',
'query=text',
'sample')

In [4]: r.scheme
Out[4]: 'http'

In [5]: r.netloc
Out[5]: 'alice:secret@ www.hostname.com:80'

In [6]: r.path
Out[6]: '/%7Ealice/python.cgi'

In [7]: r.params
Out[7]: ''

In [8]: r.query
Out[8]: 'query=text'

In [9]: r.fragment
Out[9]: 'sample'

In [10]: r.username
Out[10]: 'alice'

In [11]: r.password
Out[11]: 'secret'

In [12]: r.hostname
Out[12]: 'www.hostname.com'

In [13]: r.port
Out[13]: 80

In [14]: r.geturl()
Out[14]:
'http://alice:secret@ www.hostname.com:80/%7Ealice/python.cgi?query=text#sample'

In [15]: r2 = urlparse.urlparse("www.python.org/about","http")

In 16]: r2
Out[16]: ('http', '', 'www.python.org/about', '', '', '')
```

- 在 In［1］中，使用 impot 语句导入了 urlparse 模块，这是所有使用其模块中的方法时所必须要做的一个步骤。
- 在 In［2］中，采用此模块中的 urlparse 方法来对指定的 URL 进行解析，并赋值给 r 变量。
- 在 In［3］中打印了 r 的原始值。可以看到，这里出现了一个六元组。具体的属性值的

更多资源请访问稀酷客(www.ckook.com)

含义可以参见表 17-2。

- 在 In［4-13］中分别输出了各个 URL 组成部分的信息。注意，网络地址中包括用户名、密码、主机名和端口等四部分。
- In［14］中提供了一个从 URL 六元组得到其 URL 信息的方法。
- 在 In［15］和 In［16］中，使用了 default_scheme 参数变量。可以看到，当 URL 中不包含协议的时候，将会自动使用此参数。

17.2.3　URL 的拼合

由于 URL 可以分成多个部分，所以这一小节介绍如何对 URL 进行拼合。当提供了 URL 中的绝对地址和相对地址的时候，可以使用 urlparse 模块中的 urljoin 方法将其拼合起来。urljoin 方法有两个参数，一个是绝对地址，另外一个是相对地址。直接调用此函数，将会拼合产生一个 URL 字符串：

```
#在下面的代码片段中省略了导入 urlparse 模块
In [2]: r = urlparse.urljoin("http://www.zeroc.com","ice.html")

In [3]: r
Out[3]: 'http://www.zeroc.com/ice.html'
```

在 In［2］的输入参数中包含绝对 URL 和相对 URL。返回值在 In［3］中输出，可以看到已经将两者拼合起来了。

通过上面的代码可以看到，urljoin 方法的使用是很简单的，仅仅是将两个类型的 URL 组成一个完整的 URL。但是实际上，由于对输入的参数并没有限制，所以还有如下需要注意的地方。

（1）当输入的参数都是空字符串的时候，返回的拼合 URL 也是空字符串：

```
In [3]: r = urlparse.urljoin("","")

In [4]: r
Out[4]: ''
```

在 In［3］中，urljoin 的两个参数都是空字符串。可以看到其得到的 URL 输出也是一个空字符串。

（2）在相对 URL 中如果有协议字段，则优先使用相对 URL 中的协议。否则使用绝对 URL 中的协议字段：

```
In [5]: r = urlparse.urljoin("http://www.python.org","ftp://www.python.org/faq")

In [6]: r
Out[6]: 'ftp://www.python.org/faq'

In [7]: r = urlparse.urljoin("http://www.python.org","www.python.org/faq")

In [8]: r
Out[8]: 'http://www.python.org/www.python.org/faq'
```

- 在 In［5］的 urljoin 方法中，两个参数中都包含绝对地址。而两者的协议字段并不相同，一个是 htpp，另一个是 ftp。在 In［6］的输出结果中，可以看到生成 URL 中的协议采用了相对地址中的协议值。
- 在 In［7，8］中可以看到，当相对字符串中不包含协议的时候，就会将所有的字符串认为是一个路径信息，从而将字符串的值和绝对 URL 的值结合了起来。

（3）当绝对 URL 和相对 URL 中都含有服务器地址且不相同的时候，将采用相对 URL 地址的服务器地址和路径：

更多资源请访问稀酷客(www.ckook.com)

```
In [9]: r = urlparse.urljoin("http://www.python.org","http://www.python.com/faq")

In [10]: r
Out[10]: 'http://www.python.com/faq'
```

- 在上面的代码中，需要拼合的两个字符串中的服务器地址是不同的。在输出结果中可以看到，拼合后的 URL 采用了相对 URL 中的服务器地址。

17.2.4　URL 的分解

和 urlparse 方法类似，urlsplit 方法可以用来对 URL 进行分解。其函数原型和 urlparse 类似，也是接受一个字符串，然后给出一个五元组。和 urlparse 方法的结果相比，这里的结果少了 param参数，从而从六元组减少成了五元组。下面是一个简单的例子：

```
In [11]: r = urlparse.urlsplit("http://www.python.org:80/faq.cgi? src=fie")

In [12]: r
Out[12]: ('http', 'www.python.org:80', '/faq.cgi', 'src=fie', '')
```

这里演示了 urlsplit 的用法，可以看到和 urlparse 基本类似，只是最后生成的是一个五元组，去除了 param 输出值。这也反映了网络的变化情况。

而 urlunsplit 方法则是 urlsplit 的反向方法，是将通过 urlsplit 生成的五元组组合成一个 URL 字符串。这两个方法组合在一起使得可以有效的格式化 URL，特殊字符可以在这个过程中得到转换：

```
In [20]: r = urlparse.urlunsplit(("http","www.python.org","faq","",""))

In [21]: r
Out[21]: 'http://www.python.org/faq'

In [22]: r = urlparse.urlunsplit(
   ....:         urlparse.urlsplit("http://www.python.org/faq?")
   ....: )

In [23]: r
Out[23]: 'http://www.python.org/faq'
```

- 在 In［20，21］中主要演示了 urlunsplit 方法的使用。注意到，urlunsplit 方法中的参数是一个五元组。在 Out［21］中显示了最后得到的结果。这个过程基本上是 urlsplit 的反向过程。
- 在 In［22，23］中可以看到，Out［23］中的输出结果相对于原来的输入，区别在于对 URL 进行了格式化。这里的变化主要是删除了后面的“?”字符。

【例 17-3】 演示使用 urlparse 中的 urljoin、urlsplit、urlunsplit 这 3 个方法。这个代码片段的目的是将部分绝对地址和相对地址组成 URL，并通过判断 URL 的协议类型来进一步处理。对于 file 类型的文件，处理将忽略。而对于 ftp 类型的文件，则将其改成 http 协议。

```
#filename urlparse_1.py
import urlparse

abs_urls = ["http://www.python.org","ftp://www.linux.org","http://www.gtk.org","file://"]
rel_url = "faq.html"

for abs_url in abs_urls:
    url = urlparse.urljoin(abs_url, rel_url) #拼合 URL
    expected_url = url
    scheme, netloc, path, query, fragment = urlparse.urlsplit(url) #分解 URL

    if scheme or scheme == "file":
       print url," ==== > None"
       continue
```

更多资源请访问稀酷客(www.ckook.com)

```
    if scheme is not "ftp":
        expected_url = urlparse.urlunsplit ( ('http', netloc, path, query, fragment))
print url," ====>", expected_url
```

- 使用 urlparse 模块中的函数都需要在文件开始处导入 urlparse 模块。
- 第 4 行和第 5 行中定义了一个绝对地址的集合和一个相对地址，在后面将对其进行处理。
- 在 for 循环中，对绝对地址集合中的每个绝对地址都进行处理。首先是使用 urljoin 将特定的绝对地址和相对地址拼合成一个 URL 字符串。接着是使用 urlsplit 方法将此字符串 URL 分解成为一个五元组。
- 分解成一个五元组后，后面有两个 if 判断。第一个判断是判断协议字段是否存在或者是否为 file 类型协议，如果是的话则将此忽略。这种判断在实际中常发生在不需要处理本地文件的时候。第二个判断主要是判断协议是否为 ftp 协议，如果是的话，则使用 http 协议来代替。在后面的处理中，使用 urlunsplit 来将特定的五元组组成一个 URL。
- 最后将此转换输出。

【例 17-3】运行的结果如下所示：

```
In [1]: run urlparse_1.py
http://www.python.org/faq.html ====> http://www.python.org/faq.html
ftp://www.linux.org/faq.html ====> http://www.linux.org/faq.html
http://www.gtk.org/faq.html ====> http://www.gtk.org/faq.html
file:///faq.html ====> None
```

从输出结果中可以看到，代码片段很好的完成了预期的工作：当原来的协议为 http 的时候，就按照拼合好的 URL 输出；如果是 ftp，则转换成 http 协议；如果是 file 协议，则输出“None”，表示 file 类型的协议是被忽略的。

17.2.5　URL 的编码与解码

在 URL 中使用的是 ASCII 字符集中的字符。当使用不在 ASCII 字符集中的字符时，就需要对此字符进行编码。特别是对于东亚地区的字符，包括中文。ASCII 字符集中的编码规则是这样的：在百分号后面接上两个十六进制数字，和其在 ASCII 字符表中的对应值相同。

即使是 ASCII 字符集中的字符，有些也不能直接使用。常见的一种情况是不能在 URL 中使用空格字符。如果使用的话，也将会出错。这时候，可以将空格符编码成“%20”。例如下面的 URL：

```
http://www.python.org/advanced%20search.html
```

在上面的 URL 中，可以看到用“%20”替代了空格符。实际上，这个 URL 将从主机 www.python.org 上获取“advanced search.html”页面。

ASCII 字符集中有些字符可能会使得 URL 非法，这些字符被称为保留字符。保留字符是那种不能在 URL 中出现的字符。例如斜线字符将会用来分隔路径。如果需要使用斜线字符而不是将其作为路径的分隔符，则需要对其进行转义。表 17-3 中列出了 URL 编码中保留的字符。

表 17-3　URL 编码中保留的字符

保留字符	URL 编码
;	%3B
/	%2F
?	%3F
:	%3A

更多资源请访问稀酷客(www.ckook.com)

（续）

保 留 字 符	URL 编码
@	%40
=	%3D
&	%26
（空格）	%20

ASCII 字符集中有些字符会导致上下文歧义，这些字符被称为不安全字符。不安全的字符是指那些虽然在 URL 中没有特殊的意义，但是在 URL 的上下文中可能有特殊的含义。例如双引号在标签中是用来分隔属性和其值的，如果在 URL 中有双引号，则有可能会使得浏览器解析的时候发生错误。此时，可以通过使用“%22”来编码双引号，从而解决这种冲突。表 17-4 中列出了不安全的字符。

表 17-4　URL 编码中不安全的字符

不安全字符	URL 编码
<	%3C
>	%3E
“	%22
#	%23
%	%25
{	%7B
}	%7D
|	%7C
\	%5C
^	%5E
~	%7E
[	%5B
]	%5D
‘	%60

从表 17-3 和表 17-4 中可以看出，大部分的标点符号都需要编码，特别是那些保留字符。对于如中文字符的输入，URL 也需要进行编码。一般地，对于不在 ASCII 字符集中的字符，如果不知道是否需要编码，则最好都进行一次编码。

即使是对于 ASCII 字符集合中的字符进行编码也是没有问题的。但是需要注意的是，当字符具有特定的含义的时候，此时不应进行编码。例如在 HTTP 协议中对协议字段上的斜线进行编码是不对的。这会阻止浏览器对于 URL 的正确访问。

在 urllib 中有一套可以对 URL 进行编码和解码的方法，表 17-5 列出了这些方法。

表 17-5　urllib 模块中 URL 编码和解码的方法

方　　法	功　　能
quote	对 URL 进行编码
quote_safe	同 quote 方法，进一步将空格表示成“ + ”符号
unquote	对 URL 进行解码
unquote_safe	同 unquote 方法，进一步将“ + ”符号变成空格

更多资源请访问稀酷客(www.ckook.com)

注意到在表 17-5 中，URL 的编码和解码都有着两个类似的函数。对于 URL 编码而言，主要的是 quote 方法，其参数为需要编码的字符串，而可选参数则是不需要编码的字符集合。对于 quote_safe 方法，其具体的用法和 quote 类似，还有一个功能是可以将空格转换成加号。

【例 17-4】演示解码和编码的过程是互逆的，两个方法的功能也是一一逆对应的。

```
In [1]: import urllib

In [2]: r1 = urllib.quote("/~test/")

In [3]: r1
Out[3]: '/%7Etest/'

In [4]: r2 = urllib.quote("/~test/public html")

In [5]: r2
Out[5]: '/%7Etest/public%20html'

In [6]: r3 = urllib.quote_plus (" /~test/public html")

In [7]: r3
Out [7]: '%2F%7Etest%2Fpublic+html'

In [8]: urllib.unquote_plus (r3)
Out [8]: '/~test/public html'

In [9]: urllib.unquote (r2)
Out [9]: '/~test/public html'

In [10]: r = urllib.quote (" /~test/", " ~/")

In [11]: r
Out [11]: '/~test/'
```

- 使用 URL 编码和解码的方法都需要导入 urllib 模块，如 In［1］中所示。
- 在 In［2，3］中，演示了 quote 方法的常规使用。注意，在输出中，“~”字符编码被替换成了“%7E”。
- 在 In［4-7］中，分别对一个包含空格字符的字符串进行了 URL 编码。在 Out［5，7］中，可以看到 quote 与 quote_plus 两者的具体区别。在使用 quote 的时候，“/”字符并没有转换，而空格符号变成了“%20”。同时，在使用 quote_plus 的时候，将“/”字符替换成了“%2F”，并同时将空格字符替换成了“+”符号。这也是两者的主要区别。
- 在 In［8，9］中，对已经得到的编码 URL 进行解码。分别采用了两种方法：unquote_plus 和 unquote。两种方法得到的结果是一样的，这是因为其输入参数也是和输出相对应的。
- 在 In［11］中，演示了编码方法中第二个参数的使用方法。可以注意到，当将“~”字符加入到忽略字符列表后，再次输出的时候就不用对“~”进行编码了。

17.2.6　中文的编码与解码

由于很多浏览器都不支持在 URL 中输入中文，所以当 URL 中包含中文的时候，需要对 URL 进行编码。例如在百度搜索中，可以通过使用“http：//www. baidu. com/s? wd = keyword”的 URL 来搜索包含 keyword 的网页。当 keyword 为中文的时候，浏览器对其进行了 URL 编码。图 17-2 中显示了使用“URL 编码”时的浏览器 URL 信息。

更多资源请访问稀酷客(www.ckook.com)

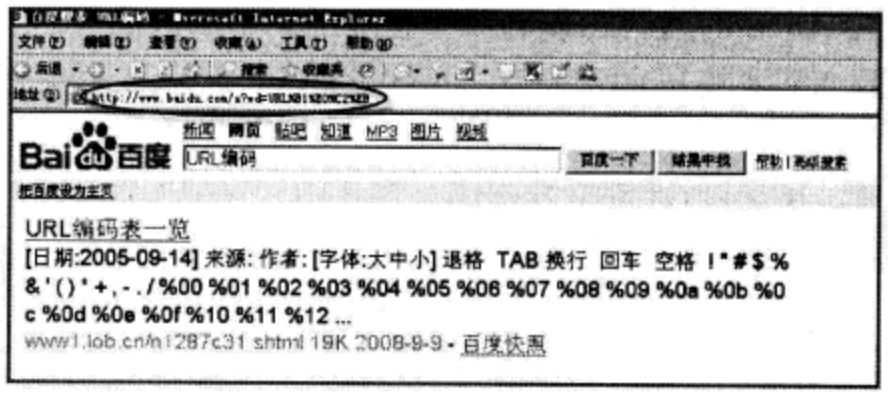


图 17-2　中文 URL 编码

【例 17-5】演示对“URL 编码”字符串进行编码的过程。

```
In [3]: quote("URL 编码")
Out[3]: 'URL% B1% E0% C2% EB'

In [4]: r = quote("URL 编码")

In [5]: unquote(r)
Out[5]: 'URL \xb1 \xe0 \xc2 \xeb'

In [6]: print unquote(r)
URL 编码
```

- 在 In［3］中，对“URL 编码”字符串进行了 URL 编码，得到的结果和图 17-2 地址栏中显示的是一样的。
- 在 In［4］中得到 URL 编码后，在 In［5，6］中对其进行了解码。可以看到，解码后的结果和“URL 编码”字符串是一样的。

虽然现在有些浏览器已经支持在地址栏中直接显示中文，但是对于中文字符使用 URL 编码总是安全的。在一定的时候，这种编码还可以起到混淆信息的作用。如果得到一个这样的字符串：“% D5% E2% CA% C7% D2% BB% B6% CE% C3% D8% C3% DC% B5% C4% CF% FB% CF% A2”，在不处理之前是不知道这个字符串的意思的。使用 URL 解码就可以知道编码之前的信息：

```
In [8]: unquote("% D5% E2% CA% C7% D2% BB% B6% CE% C3% D8% C3% DC% B5% C4% CF% FB% CF% A2 ")
Out[8]: '这是一段秘密的消息'
```

当将编码信息和未编码信息混合在一起的时候，这种信息的混淆度将会更大。

17.2.7　查询参数的编码

在 HTTP 中有两种提交数据的方式：get 和 post。在这两种方式中都需要提交查询参数和其值，特别是在 get 方法中还会将这种值对显示在地址栏中（参见图 17-2）。由于在 HTTP 中经常遇到这种情况，所以在 urllib 模块中提供了对查询参数进行编码的方法：urlencode。

urlencode 方法的作用就是将查询的参数值对返回成 URL 编码的形式。参数值对可以是一系列的（keyword，value）参数值对，也可以是一个字典数据：

```
#省略导入 urllib 模块语句
In [3]: urllib.urlencode([('keyword1','value1'),('keyword2','value2')])
Out[3]: 'keyword1 = value1&keyword2 = value2'

In [4]: urllib.urlencode()
Out[4]: 'keyword2 = value2&keyword1 = value1'

In [5]: urllib.urlencode({'keyword2':'value2','keyword1':'value1'})
Out[5]: 'keyword2 = value2&keyword1 = value1'
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［3］中，查询参数为一个（keyword，value）的列表，这里有两个参数值对。Out［3］的输出结果中，urlencode 方法将此参数值对列表编码成了一个 URL 字符串，其中查询参数的顺序和在列表中的顺序是一致的。
- 在 In［4，5］中，分别对两个使用字典结构的查询数据进行了编码。可以看到，urlencode方法也将这些查询数据编码成了 URL 字符串。
- 需要注意的是，在 In［4，5］中，虽然输出的 URL 字符串是一样的，但是两者所接受的字典结构数据输入的时候并不一样。这是因为字典内部的无序性造成的。从下面代码片段的测试中可以看出来：

```
In [14]: a = {'keyword1':'value1','keyword2':'value2'}

In [15]: b = {'keyword2':'value2','keyword1':'value1'}

In [16]: a is b
Out[16]: False

In [17]: a == b
Out[17]: True
```

由于字典的这种内部无序性，所以在对查询数据进行编码时，如果需要对编码后的 URL 字符串输出进行控制，则应该采用在 In［3］中的列表形式。

urlencode 方法还可以接受一个可选的参数，用来对输入查询参数中的数据进行控制。默认值为 False，即当查询数据（keyword，value）列表中的 value 也为列表的时候，将其整个使用 quote_plus 方法进行编码，并作为查询参数的值。而当其值为 True 的时候，对于上述的这种情况，会将 value 列表中的每个值都和 keyword 组成一个查询参数值对：

```
In [18]: urllib.urlencode([('keyword',('value1', 'value2', 'value3'))])
Out[18]: 'keyword=%28%27value1%27%2C+%27value2%27%2C+%27value3%27%29'

In [19]: urllib.urlencode([('keyword',('value1', 'value2', 'value3'))], True)
Out[19]: 'keyword=value1&keyword=value2&keyword=value3'
```

- 由于可选参数的默认值是 False，当需要此方法的默认行为的时候，可以将其省略。在 In［18］中，对 value 为列表的查询数据进行了编码。从输出结果可以看到，此方法将其作为了一个整体来看待，这里只有一个 keyword 查询值。后面的值是经过了 quote_plus 方法编码后的数据。可以使用 unquote_plus 来对其进行解码：

```
In [20]: r = urllib.urlencode([('keyword',('value1', 'value2', 'value3'))])

In [21]: urllib.unquote_plus (r)
Out [21]: " keyword= ('value1', 'value2', 'value3')"
```

- 从 Output［21］的结果来看，和 In［18］中输入的 value 值是一样的，这也进一步表明了 urlencode 方法在可选参数为 False 的时候的内部操作。
- 在 In［19］中，urlencode 的可选参数为 True。此时会将 value 为列表的值对扩展成多个 keyword-value 值对，输出结果如 Out［19］所示。

当可选参数为 False（也是默认值）的时候，由于会首先使用 str 方法对 value 值进行格式化，从而可能得到非预期的结果，所以当 value 为列表的时候，需要谨慎选择 urlencode 方法的可选参数。

更多资源请访问稀酷客(www.ckook.com)

17.3　CGI 的使用

公共网关接口（Common Gateway Interface，CGI）是外部应用程序和 HTTP 服务器之间交互的一个通用接口标准。CGI 本身不是一种语言，也不是一种网络协议，仅仅定义了 HTTP 服务器和程序之间的交换信息规范。CGI 可以使用任何语言来书写，当然可以使用 Python 语言。在本节中，将详细介绍 CGI 的相关知识及其使用方法。

17.3.1　CGI 介绍

CGI 是一种动态网页构建技术，虽然其他竞争技术的出现使得 CGI 使用的机会变少，但由于使用简便，仍不失为一种快速构建交互环境的方法。CGI 虽然需要由服务器来提供，但是和服务器是相互独立的。只要 HTTP 服务器支持 CGI，即可运行相应的脚本。幸运的是，现在几乎绝大多数流行的 HTTP 服务器都是支持 CGI 的。

虽然 CGI 在服务器上的设置很容易，但是存在两个比较大的问题。一个是性能问题，由于每个 CGI 脚本都会调用外部程序来生成 HTTP 输出，当请求较多的时候将使得服务器的负载过重。另外一个问题是安全性问题，因为通过 CGI 脚本可以直接调用系统中的程序，并执行相应的操作和访问系统文件。在这种情况下，系统的一些漏洞可能会通过 CGI 脚本的执行而暴露出来。

在性能方面，现在已经有一些改进的技术来解决这个问题。如通过 mod_python 模块可以有效的提高 Apache 服务器对于 Python 脚本文件的响应速度。而在安全性方面，则大多数的 HTTP 服务器将 CGI 脚本文件限制在一个特定的文件夹中，如 cgi-bin 目录。这样就阻止了在不经意的情况下将 CGI 脚本暴露给外界。只有经过确认的 CGI 脚本才可以放入特定的文件夹中，从而提高了安全性。

在 Apache 服务器中，默认已经开启了对于 CGI 的支持。保存 CGI 脚本文件的目录为 cgi-bin。在使用 Python 来编写 CGI 脚本文件的时候，需要在第一行中指定 Python 可执行文件的位置。只有这样，Apache 才能够找到特定的 Python 脚本文件并运行。

CGI 脚本文件通过环境变量、命令行参数和标准输入输出与 HTTP 服务器进行通信，传递有关参数并进行处理。当方法为“GET”的时候，CGI 程序通过环境变量来获取客户端提交的数据。而当方法为“POST”的时候，CGI 程序将通过标准输入流和环境变量来获取客户端的数据。在 CGI 程序返回处理结果给客户端的时候，则是通过标准输出流将数据输出到服务器进程中。

当客户端请求一个 CGI 程序的时候，CGI 需要输出信息来声明请求的 MIME 类型，并通过服务器来传递给客户端。CGI 的环境变量 HTTP_ACCEPT 提供了可以被客户端和服务器端接受的 MIME 类型列表。当含有多个 MIME 类型的时候，使用逗号分隔开来。例如 image/gif、text/html 和 */* 等。表 17-6 中列出了现有 HTTP 所支持的 MIME 类型。

表 17-6　HTTP 支持的 MIME 类型

类　型	子类型	说　明
Application	Octet-stream	MIME 编码的二进制数据
Audio	Basic	音频文件
Image	Gif、Jpeg	图像文件
Message	External-body	对于信息的封装
Multipart	Mixed digest	混合类型，可以支持文件上传等操作
Text	Html、Plain	HTML 文档，纯文本文档
Video	mpeg	视频文件

在本节中，主要讨论的 MIME 类型是 text/html，也就是 HTML 文档。

CGI 脚本输出的内容包括两部分。一部分是文件头，其中有指示下面的数据是什么 MIME

更多资源请访问稀酷客(www.ckook.com)

类型的信息。在 Python 中，输出 text/html 类型文件的时候将使用如下的头文件信息：

```
print "Content-Type: text/html"
print
```

在上面的两个语句中，第一个语句主要用来指示下面将要输出的内容为 HTML 文档。当客户端收到这样的信息时，就可以使用特定的渲染方法来显示文档。第二个语句是打印出一个空行，这里主要用来表示文件头的结束。当然，也可以将这两个语句合并成下面的一个语句：

```
print "Content-Type: text/html \n \n"
```

之所以需要输入两个换行符，也是因为需要在这个部分的后面输出一个完整的空行来表示文件头的结束。

输出的第二部分是具体的文件信息。例如要在客户端显示 first. html 文档，则可以将此文档的 HTML 内容通过标准输出完整的输出。最简单的方法是使用 print 语句将文档的内容一行一行输出：

```
print '<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd" >'
print '<html >'
print '<head >'
#省略部分代码
print '<br >&copy;2009'
print '</body >'
print '</html >'
```

把上面的文件头和文件内容结合为一个文件，并保存在 CGI 脚本目录下，一般为 cgi-bin。当客户端请求此脚本文件的时候，系统将会返回上述的 HTML 文档。而此时在客户端则可以通过 MIME 类型来渲染下面的文档内容。在这里，浏览器一般是会显示如图 17-1 所示的效果。

在 Python 标准库中的 cgi 模块可以用来处理 CGI 脚本。此模块提供了丰富的方法用来处理各种输入情况，同时也有一些方法用来输出相关的信息。在下面的小节中，将对使用 cgi 模块中的方法进行详细介绍。

17.3.2　获取 CGI 环境信息

Internet 上原来大部分的 HTML 文档都是静态的。当使用了 CGI 脚本后，可以根据请求的不同来输出动态的信息。如下面的代码中，将输出当前服务器的时间：

```
#! E:/Python25/python.exe
#encoding = utf-8

print "Content-Type: text/plain \n \n"

import datetime
print datetime.datetime.now()
```

- 第 1 行代码指示了执行此 CGI 脚本文件的程序。由于这里的是 Python 代码，所以指明了 Python 可执行程序的目录。
- 在输出部分，首先是输出了文件头信息。这里的文件头是 text/plain 类型，表示下面输出的内容可以按照文本内容来处理。注意到这里后面的两个换行符，将会输出一个空行，从而用来指示当前文件头的输出结束。
- 在导入了 datetime 模块后，直接调用此模块的 now 方法，输出了当前的服务器时间。在这里，用户可以根据需要编写代码。



图 17-3　CGI 脚本文件的输出

将此代码保存为 mydate. cgi 文件，并放在 cgi-bin 目录中。其中 cgi-bin 目录为专门执行 CGI 脚本代码的位置。启动浏览器，调用 CGI 脚本文件，可以得到如图 17-3

更多资源请访问稀酷客(www.ckook.com)

所示的结果。

CGI 环境准备完毕后，就有一些环境变量存在。可以通过这些环境变量来对请求做一些简单的判断。例如环境变量 REMOTE_ADDR 中包含客户端的 IP 地址，其值可以通过 os. environ 得到。

【例 17-6】通过判断 REMOTE_ADDR 的值可以知道是否为本地访问。然后分别打印出不同的信息，从而给用户不同的访问体验。

```
#! E:/Python25/python.exe
#encoding = utf-8

import os
print "Content-Type: text/html \n \n"

remote_addr = os.environ['REMOTE_ADDR']

if remote_addr == '127.0.0.1':
    print "来自本地的访问"
else:
    print "来自外部的访问"
```

- 这里获取了 CGI 环境中的 REMOTE_ADDR 的值，并赋值给 remote_addr 变量。
- 通过这个值和“127.0.0.1”的比较，来判断是否为来自本地的访问。然后根据比较的结果显示给用户不同的信息。可以在这里加入具体的处理代码。

除了 REMOTE_ADDR 环境变量以外，还有不少其他的 CGI 环境变量。在 cgi 模块中，提供了一个实用函数 print_environ，可以显示在 CGI 环境中所有可用的变量及其值的信息。

【例 17-7】显示当前服务器中所有支持的 CGI 变量。

```
#! E:/Python25/python.exe
#encoding = utf-8

print "Content-Type: text/html \n \n"

import cgi

cgi.print_environ()
```

- 这段代码前面的内容和前面介绍过的部分是一样的，主要是输出文件头信息。需要注意的是，这里使用的 MIME 类型是 text/html。这是因为后面的方法中输出的是 HTML 文档。如果使用 text/plain 的话，则会将 HTML 文档原样输出，而不会被浏览器解析。
- 在导入了 cgi 模块后，代码直接调用了 cgi 模块的 print_environ 方法，输出当前的 CGI 环境中的所有变量。

下面是【例 17-7】代码的输出结果。

```
Shell Environment:
COMSPEC
C:\WINDOWS\system32\cmd.exe
DOCUMENT_ROOT
I:/server/htdocs
GATEWAY_INTERFACE
CGI/1.1
HTTP_ACCEPT
*/*
HTTP_ACCEPT_ENCODING
gzip, deflate
HTTP_ACCEPT_LANGUAGE
zh-cn
HTTP_CONNECTION
Keep-Alive
```

更多资源请访问稀酷客(www.ckook.com)

```
    HTTP_HOST
    localhost
    HTTP_USER_AGENT
    Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR
3.0.04506.648; .NET CLR 3.5.21022; InfoPath.2; CIBA)
    PATH
    C: \WINDOWS\system32; C: \WINDOWS; C: \WINDOWS\System32\Wbem;
    PATHEXT
    .COM; .EXE; .BAT; .CMD; .VBS; .VBE; .JS; .JSE; .WSF; .WSH
    QUERY_STRING
    REMOTE_ADDR
    127.0.0.1
    REMOTE_PORT
    12325
    REQUEST_METHOD
    GET
    REQUEST_URI
    /cgi-bin/print_envrion.cgi
    SCRIPT_FILENAME
    I: /server/Apache/cgi-bin/print_envrion.cgi
    SCRIPT_NAME
    /cgi-bin/print_envrion.cgi
    SERVER_ADDR
    127.0.0.1
    SERVER_ADMIN
    admin@ localhost
    SERVER_NAME
    localhost
    SERVER_PORT
    80
    SERVER_PROTOCOL
    HTTP/1.1
    SERVER_SIGNATURE
    <address>Apache/2.0.63 (Win32) PHP/5.2.6 mod_python/3.3.1 Python/2.5.2 Server at localhost
Port 80</address>
    SERVER_SOFTWARE
    Apache/2.0.63 (Win32) PHP/5.2.6 mod_python/3.3.1 Python/2.5.2
    SYSTEMROOT
    C: \WINDOWS
    WINDIR
    C: \WINDOWS
```

- 从输出结果中可以看出，当访问 CGI 脚本的时候，许多的信息可以从 CGI 的环境变量中获取，这也是 CGI 和客户端进行交互的一个重要部分。

表 17-7 总结了一些 CGI 环境中重要的环境变量。

表 17-7　CGI 中的部分环境变量

环境变量	含　义
REMOTE_ADDR	服务器的地址
PATH_INFO	路径信息
REQUEST_METHOD	客户端请求的方法，可以为 GET 或 POST
SERVER_NAME	服务器名字
SERVER_PORT	服务器端口
SCRIPT_NAME	服务器执行脚本的名字
HTTP_USER_AGENT	客户端软件
SERVER_PROTOCOL	服务器支持的协议
SERVER_SOFTWARE	服务器端的软件信息
QUERY_STRING	查询字符串

更多资源请访问稀酷客(www.ckook.com)

注意　这些获取的值都是原始的字符串，不包含格式化信息。如果需要从其中获取相应的信息，可以利用 cgi 中提供的方法。

17.3.3　解析用户的输入

CGI 技术的一个重要作用是可以获取用户的输入，从而做出相应的处理。虽然请求的方法有 GET 和 POST，但是 cgi 模块使用了一种统一的方式来处理用户的输入。这就是 FieldStorage 类。

FieldStorage 类的使用方式很简单，直接不带参数的初始化一次即可。其对象将会自动根据请求方法的类型通过标准输入或者环境变量来获取这些值。

可以将 FieldStorage 类的对象看成是 Python 语言中的字典数据结构，支持字典访问的部分方法，包括 has_key( ) 和 key( )。当然，内置的方法 len( ) 也是支持的。当其值为空字符串的时候，FieldStorage 类的对象默认将不会进行存储。如果需要存储这样的空字符串，可以在初始化 FieldStorage 的时候加入可选参数 keep_blank_values。

【例 17-8】显示 CGI 环境中所有的用户输入。

```
#! E:/Python25/python.exe
#encoding = utf-8

print "Content-Type: text/html \n \n"

import cgi
form = cgi.FieldStorage()

for key in form.keys():
    print key," == > ", form[key].value
    print " < br / > "
```

- 在导入了 cig 模块后，实例化了一个不加初始化参数的 FieldStorage 对象 from。此时，用户访问的信息都保存在了 from 对象中。
- 下面的循环中，通过遍历 form 对象中的所有键值，并显示相应的值信息。这里的值是通过其 value 属性得到的。这和其对象的 getvalue 方法得到的结果是一致的。注意，在最后面加入了“ < br / > ”标签。这是由于这里输出的是 HTML 文档，可以通过使用 < br / > 来实现换行。而通过输入换行符是不能实现换行的。

图 17-4 中显示了这段代码的使用方法。这里，使用了 GET 的访问方式。直接在 URL 中加入了查询的参数“value1 = str&value2 = 10”。可以看出上面的代码正确的显示了输入的参数。

上面的代码有一个问题，那就是当某个字段含有多个输入的时候，不能得到正确的处理，如图 17-5 所示。

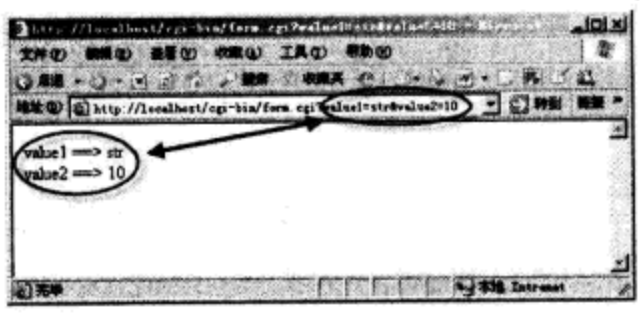


图 17-4　用户输入的获取和显示



图 17-5　getvalue 方法不能正确处理多输入的情况

在上面的测试中，在输入中使用了两个 value1 的值。这样在输出结果的时候，将得不到正确的值，甚至是连一个值也不能得到。

更多资源请访问稀酷客(www.ckook.com)

为了解决这个问题，FieldStorage 类对象提供了 getlist 方法，可以用来获取一个字段的多个输入值。这些输入值组成了一个列表。可以将【例 17-8】代码改成下面的形式：

```
#! E:/Python25/python.exe
#encoding = utf-8

print "Content-Type: text/html \n \n"

import cgi
form = cgi.FieldStorage()

for key in form.keys():
    print key," == > ", form.getlist(key)
    print " < br/ > "
```

- 这里主要改变的地方在最后的循环中，使用了 getlist 方法来获取对应字段的值。这样在遇到有多个输入的时候，可以正确的处理。

这段代码对于上面多输入情况的处理如图 17-6 所示。

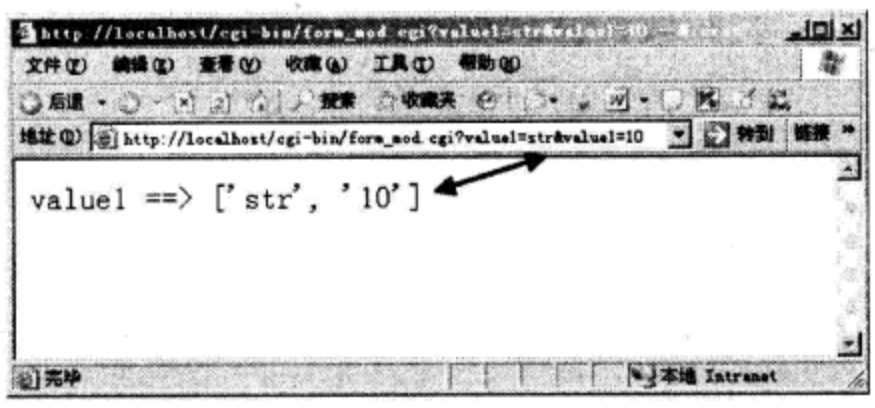


图 17-6　getlist 方法可以正确处理多输入的情况

- 在图 17-6 所示的输出结果中可以看到，CGI 脚本现在能正确处理多输入的情况了，并保存在了列表中。用户可以使用这个结果做进一步的处理。

在上面处理用户输入的过程中，都没有考虑到用户输入字符的特殊性。实际上，用户输入的字符可能是含有如 HTML 标签一类的字符。这时候，就需要对这样的特殊字符进行转义，否则会使最终显示结果出现偏差。例如在图 17-7 中，特定的输入影响了最后的显示。

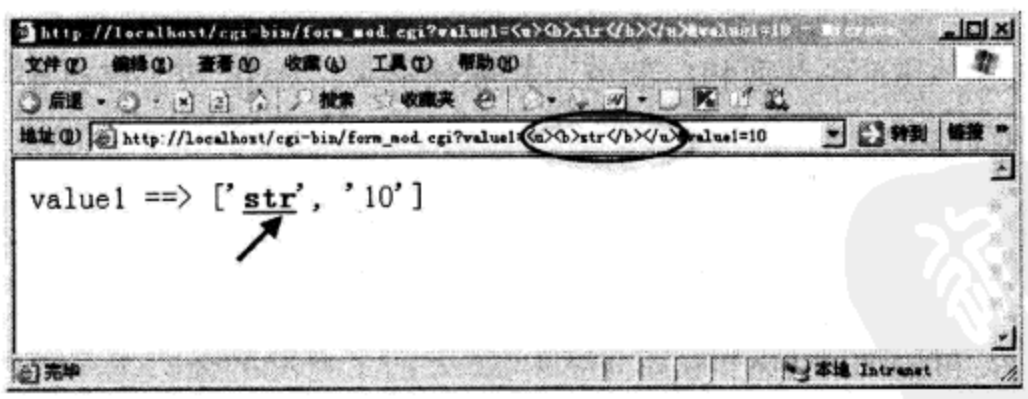


图 17-7　没有对输入数据进行转义的情况

在图 17-7 所示的输出结果中可以看到，并没有将 value1 的值正确显示出来。实际上，由于其输入值包含 HTML 标签，因此在直接回显中对这些标签进行了解析。最终显示的结果就使得 str 加粗和有下划线。这并不是所获得的原始数据。

在 cgi 模块中提供了 escape 方法来解决这个问题。此方法将对“&”、“ < ”和“ > ”等字符进行转义，从而在显示的时候可以正确的显示出来。上面代码的修正如下：

更多资源请访问稀酷客(www.ckook.com)

```
#! E:/Python25/python.exe
#encoding = utf-8

print "Content-Type: text/html \n \n"

import cgi
form = cgi.FieldStorage()

for key in form.keys():
    for value in form.getlist(key):
        print key," == > ", cgi.escape(value)
        print " <br/ > "
```

在上面的代码片段中，主要的修改在于最后对于每个值都调用了 escape 方法对其进行转义。从而使得浏览器可以正常的显示。需要注意的是，这里仅仅只是需要在 HTML 输出的时候才对其进行转义。其值还是原来的值，而不是经过转义后的值。

现在这段代码就可以正确的处理这种含有特殊字符的输入数据了，如图 17-8 所示。

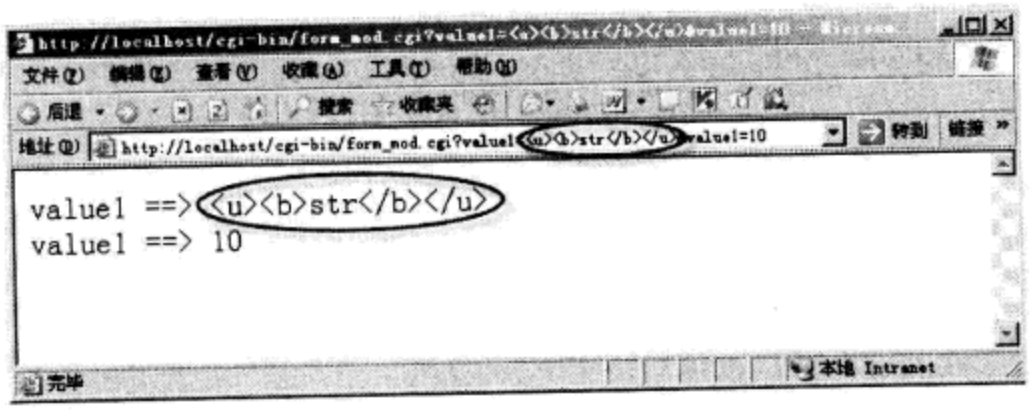


图 17-8　对输入数据进行转义的情况

从输出结果中可以看到，输出的 value1 值和输入的是一样的。

在上面的实例中，都是采用的 GET 请求方法。实际上，在需要传递大量数据的时候，还是使用 POST 方法的居多。虽然两者有些不同，但是上面介绍的获取用户输入的方法，对于请求方法为 POST 的情况同样也是适用的。

17.4　获取 HTML 资源

使用浏览器可以从 URL 中获取相应的资源并展示出来，如 IE 等。但是很多时候只是需要对获取的资源进行特定的处理，这个步骤的第一步就是要能够获取特定的 URL 资源。本节将对这个过程进行介绍。主要涉及的模块包括 urllib、urllib2 和 httplib。

17.4.1　使用 urlopen 和 urlretrieve 获取 HTTP 资源

在 urllib 模块中，可以使用 urlopen 和 urlretrieve 方法来获取网络资源。和 open 内置方法可以读取文件一样，urlopen 可以用来读取 URL 资源。和 open 方法不同的是，这里的 urlopen 并不能对获取的数据进行 seek 操作。返回值中有一个可以读的文件 handler，从而可以实现对数据的读取。其使用方法是直接在参数中输入 URL 地址，即可使用 urlopen 方法打开资源，然后就可以使用文件的读取来对数据进行操作了。

【例 17-9】演示使用 urlopen 方法打开资源，然后使用文件的读取函数来对数据进行操作。

```
#省略导入 urllib 模块语句
In [3]: r = urllib.urlopen("http://www.python.org")
```

更多资源请访问稀酷客(www.ckook.com)

```
In [4]: r.fp.read()
Out[4]: '<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/
/www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" > \n\n\n<html xmlns = "http://www
.w3.org/1999/xhtml" xml:lang = "en" lang = "en" > \n\n<head > \n   <meta http-equiv = "cont
ent-type" content = "text/html; charset = utf-8" / > \n   <title > Python Programming Lan
guage -- Official Website </title > \n   <meta name = "keywords" content = "python progr
amming language object oriented web free source" / > \n   <meta name = "description"
content = "       Home page for Python, an interpreted, interactive, object-oriente
d, extensible \n       programming language. It provides an extraordinary combinat
ion of clarity and \n       versatility, and is free and comprehensively ported."
#省略部分输出结果
```

- 在 In［3］中，使用 urlopen 方法直接打开了 URL 为 http：//www. python. org 的 Python 官网，并将返回值赋值给变量 r。这里确保可以正常的访问 Internet 资源。需要注意的是，由于读取网络数据需要一定的时间，所以这里将会处于阻塞模式，需要等待一段时间。
- 在 In［4］中使用 r. fp. read（）来读取 URL 资源。在 Out［4］中可以看到 http：//www. python. org 的 HTML 源码输出结果。

在【例 17-9】的代码中，直接调用了 read（）而并没有提供参数。也可以针对数据来实现相应的文件操作，包括部分读取等。

【例 17-10】演示对数据实现相应的文件操作，包括部分读取等操作。

```
#filename: urllib_1.py
import urllib
fp = urllib. urlopen (" http: //www. python. org")

op = open (" python. html", " wb")
n = 0
while True:
    s = fp. read (1024)
    if not s: #遇到了 EOF
        break
    op. write (s)
    n = n + len (s)

fp. close ()
op. close ()

print " retrieved", n, " bytes from", fp. url
```

- 这里的 urlopen 和前面的使用方法类似，直接在参数中输入 URL 字符串即可。
- 使用 open 方法打开一个文件名为 python. html 的文件，作为接收 URL 资源的文件。注意到这里使用了二进制写模式打开，在网络中应用写的时候，最好是使用这种模式，以便最大限度的保持获取数据的完整性。
- 在 while 循环中，直到遇到 EOF 才跳出循环（EOF 是网络数据流结束的标记）。在 fp 上使用 read 方法每次读取 1024 个字节，并将这些数据写入 python. html 文件中。
- 最后将所有的文件描述符关闭，并打印出从网络中读取或者说是写入文件的字节数。

【例 17-10】的运行结果如下：

```
In [5]: run urllib_1.py
retrieved 16578 bytes from http: //www. python. org
```

在运行结果中可以看到，从 Python 的官方网站上获取了 16 578 个字节的资源，这和 python. html文件的大小是一致的，如图 17-9 所示。

更多资源请访问稀酷客(www.ckook.com)

在图 17-9 中可以看到，文件的实际大小和获取的数据大小是相同的。使用文本编辑器打开此 HTML 文件可以看到，其结果和【例 17-9】Out［4］中输出的结果是一样的。这表明 urlopen 很好的完成了获取特定 URL 资源的任务。

当得到保存的文件后，可以使用浏览器将其打开，如图 17-10 所示。为了对比，笔者在图 17-10 中嵌入了 Python 官方网站主页在 IE 上的显示。

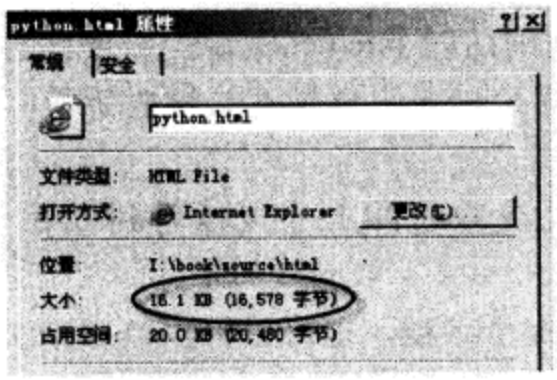


图 17-9　获取 URL 资源后保存的文件

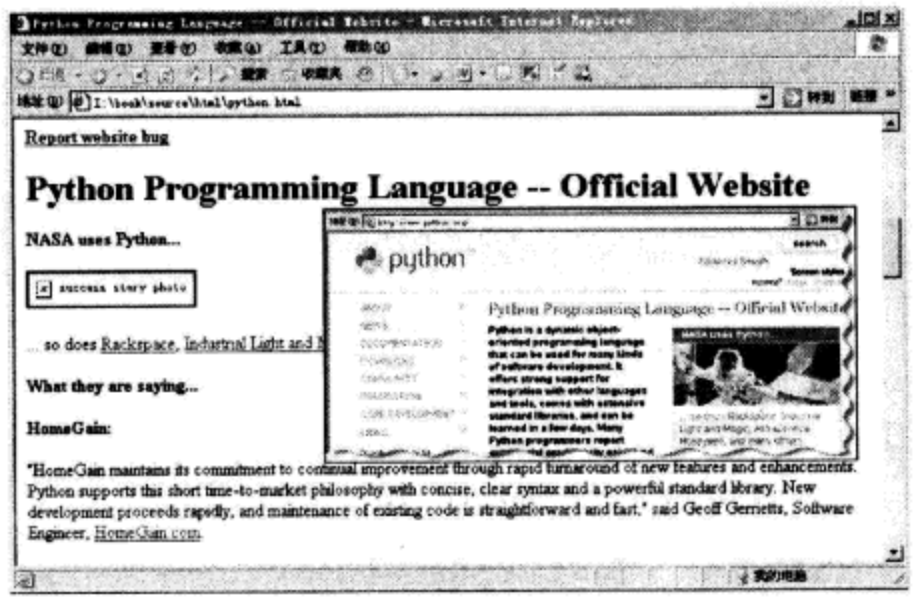


图 17-10　本地文件 python. html 和 Python 官网的区别

从图 17-10 中可以看出，两者的文字内容是相同的，但是在显示上两者还是有着不小的区别。首先表现在图片的缺失，其次是 HTML 文档布局的不同。这是因为原 HTML 文档中的图片和 CSS 的 URL 使用的是相对 URL，当 HTML 文档下载到本地后，而图片文件以及 CSS 文件并没有同时下载下来，造成了图 17-10 所示页面中的图片不能显示出来。

一般情况下，urlopen 的可选参数 data 为空的时候，支持多种协议，包括本地文件、HTTP 协议等。当为 HTTP 数据的时候，urlopen 将使用 GET 方法来获取相关的资源。但是当可选参数为非空的时候，则前面的协议字段必须为 HTTP。此时 urlopen 将使用 POST 方法来获取 URL 资源，同时，data 参数将作为 POST 方法的数据一并传递给远端服务器。注意，这里的 data 需要使用 urlencode 编码过后的查询数据。

1. 使用代理

urlopen 支持使用代理来获取 URL 资源。可以使用两种方式来影响 urlopen 获取资源时的代理选择。第一种是通过使用环境变量。这些环境变量包括 http_proxy 和 ftp_proxy，分别支持 HTTP 和 FTP 协议的代理。在 Windows 下，可以通过写入此环境变量来影响代理设置。在 Linux 下可以使用下面的命令来设置：

```
http_proxiy = http: //www. proxy. com: 3128
```

需要注意的是，在启动 Python 的交互式解析器之前，需要首先设置此环境变量。此后，代理设置才会在 urlopen 调用的时候起作用。

第二种方法是直接将代理写入到 urlopen 方法的可选参数 proxies 中去。这样，可以实现不同的时间使用不同的代理的功能。如下面的代理设置：

```
In [6]: proxies = {'http': 'http://www. proxy. com:3128'}

In [7]: urllib. urlopen(url, proxies = proxies)
```

- 在 In［6］中，设置了一个代理。这里的代理设置指明了在使用 HTTP 协议的时候将使

更多资源请访问稀酷客(www.ckook.com)

用 http：//www. proxy. com 代理服务器。

- 在 In［7］中，将在上面设置的代理服务器指定给了 urlopen 中的 proxies 参数，从而实现了代理的设置。在这里，urlopen 将通过设置的代理来获取指定的 URL 资源。

如果不需要使用任何代理，可以将 proxies 参数指定为一个空列表，从而也就清空了代理的设置：

```
In [8]: urllib.urlopen(url, proxies = {})
```

- 这里将 proxies 设置为空列表是必须的，从而明确了不使用任何代理。如果将列表去掉，urlopen 将使用环境变量中设置的代理服务器。

2. 直接保存 URL 资源

除了 urlopen 方法以外，urllib 模块中另外一个可以使用的方法是 urlretrieve。此方法和 urlopen 类似，但是也有些不同之处。最大的不同是此方法将获取的资源直接保存在文件中，而不是返回一个文件对象。

此方法也可以接受一个 URL 字符串作为输入，但是其返回的是一个指向保存文件的路径和一个 HTTPMessage 实例对象。对此 HTTPMessage 实例对象将在下一节中进行介绍。

对于返回的文件名，可以通过输入参数中的 filename 来指定。如果不指定文件名（这也是默认情况），系统将生成一个临时文件来保存获取的资源，并将此临时文件的路径返回。另外一个参数 data 的含义和 urlopen 方法中的相同：

```
In [3]: filename, m = urllib.urlretrieve("http://www.google.com")

In [4]: filename
Out[4]: 'c:\\docume~1\\administrator\\locals~1\\temp\\tmpjreye7'

In [5]: filename, m = urllib.urlretrieve("http://www.google.com", filename = "I:/book/source/
html/google.html")

In [6]: filename
Out[6]: 'I:/book/source/html/google.html'
```

- 在 In［3］的输入参数中没有指定 filename 参数，所以在返回值中的文件名是一个临时文件，这从 Out［4］的输出结果中可以看出来。
- 在 In［5］中，输入参数中有 filename 参数。使用了 urlretrieve 方法后，返回的文件名和输入的文件名参数是一样的。同时，在文件管理器中可以看到，在指定的目录下已经生成了 google. html 文件。

当然，通过此方法获取的 HTML 文档也和前面使用 urlopen 方法获取的 HTML 文档类似，仅仅只是包含 HTML 内容，而其他的资源是不会同时下载下来的。

在 urlretrieve 方法中还有一个 reporthook 参数。此参数可以用来将 urlretrieve 方法执行的过程通过图形化的方式显示给用户。其值是一个函数，会在每次获取资源块的时候被调用。此函数有 3 个参数，分别为文件块的个数、文件块的大小和文件的大小。当文件大小的值为 -1 的时候，表示此时无法获得整个文件的大小，特别是对于 ftp 流数据协议而言。

【例 17-11】使用 reporthook 参数得到获取资源时候的实时信息。

```
In [1]: import urllib

In [2]: def reporthook(block_count, block_size, file_size):
   ...:     if file_size == -1:
   ...:         print " retrieved data", block_count* block_size
   ...:     else:
```

更多资源请访问稀酷客(www.ckook.com)

```
    ...:         print "retrieve data", block_count* block_size," /", file_size
    ...:

   In
   urllib.urlretrieve (" file: ///I: /book/source/html/python. html", filename = "", reporthook
=reporthook)
   retrieve data 0 / 16578
   retrieve data 8192 / 16578
   retrieve data 16384 / 16578
   retrieve data 24576 / 16578
   Out [3]:
   ('c: \ \docume ~1 \ \sea \ \locals ~1 \ \temp \ \tmpup0_0x. html',
    <mimetools. Message instance at 0x01535620 >)

   In [4]: urllib.urlretrieve (" http: //www. python. org", filename = "", reporthook = reporthook)
   retrieve data 0 / 16578
   retrieve data 8192 / 16578
   retrieve data 16384 / 16578
   retrieve data 24576 / 16578
   Out [4]:
   ('c: \ \docume ~1 \ \sea \ \locals ~1 \ \temp \ \tmp5adz1d',
    <httplib. HTTPMessage instance at 0x01509A58 >)

   In [5]: urllib.urlretrieve (" http: //www. google. com", filename = "", reporthook = reporthook)
   retrieved data 0
   retrieved data 8192
   Out [5]:
   ('c: \ \docume ~1 \ \sea \ \locals ~1 \ \temp \ \tmpwfexxn',
    <httplib. HTTPMessage instance at 0x01495FA8 >)
```

- 在 In［2］中，定义了一个 reporthook 函数，其中的 3 个参数的含义在前面已经做了描述。函数体仅仅有打印语句，主要是根据文件大小的值来打印获取到的数据。
- 在 In［3］中，使用了在前面通过 urlopen 获取资源保存的文件 python. html。由于是本地文件，所以文件的长度是可以获知的，这里是 16 578 个字节。可以看到，每获取8192个字节后，urlretrieve 方法将会调用 reporthook 定义的函数打印出相应的信息。注意，最后一次显示的字节数超过了整个文件的大小。这是因为每次读取的时候都是使用固定块大小读取的。另外，这里使用了临时文件（在 Out［3］的输出结果中可以看出）。
- In［4］的输出结果和 In［3］的基本类似，除了临时文件不同外，在执行 In［4］语句的时候，用户可以明显感觉到打印时的停顿。这主要是因为 In［4］是通过 HTTP 协议来获取 URL 资源的，获取网络资源比获取本地资源要慢得多。当网络资源很大的时候，例如一个下载文件，使用此方法可以让用户看到获取资源的具体情况。
- 可以看到，在 In［5］中没有输出文件的大小。这表示当前文件的大小是不知道的。这种情况在 ftp 中比较常见，在 http 中也是会遇到的。

结合 urlretrieve 方法和 reporthook 参数，可以构成一个简单的下载器，并能显示资源获取的具体进度。下面是此代码片段：

```
def download(url,filename = ""):
    def reporthook(block_count, block_size, file_size):
        if file_size == -1:
            print " Can't determine the file size, now retrieved", block_count* block_size
        else:
            percentage = int ( (block_count* block_size* 100.0) /file_size)
            if percentage > 100:
                print " 100% "
            else:
                print "% d% % " % (percentage)
```

更多资源请访问稀酷客(www.ckook.com)

```
filehandler, m = urllib.urlretrieve(url,filename,reporthook=reporthook)
print "Done"
return filehandler
```

使用方法是直接输入需要获取的 URL 资源，文件名参数是可选的。当此段代码运行的时候，将会使用百分比的形式打印出所获取资源的具体信息。

17.4.2　分析返回资源的相关信息

urlopen 方法返回的是一个类似文件对象的可读的返回值，且支持如 read、readline、readlines 和 close 等方法。这些相关的方法已经在上一节中进行了演示。从返回值中可以读取相应 URL 的 HTML 文档内容。除此之外，还有一些信息可以从返回值中获取到。此文件对象还支持 geturl 和 info 这两个方法用来获取这些相关的信息。

geturl 方法将返回此文件对象所对应的 URL。需要注意的是，这个 URL 并不一定是在 urlopen方法中输入的 URL。这主要是有两个方面的原因。一个是有可能会对 URL 字符串进行格式化，第二个原因是在 HTTP 中有重定向功能，可以将 URL 资源定向到其他资源上去。可以通过调用此方法和输入的 URL 比较来了解系统资源是否发生了变动：

```
In [2]: r = urllib.urlopen("http://google.com")

In [3]: r.geturl()
'http://www.google.com/'

In [4]: r.url
'http://www.google.com/'
```

- 在 In［2］中，使用 urlopen 打开了一个 URL 资源，并将其返回值赋值给变量 r。现在变量 r 是一个资源可读取的文件对象。
- 在 In［3］中使用了此文件对象的 geturl 方法，从而得到现在文件对象所对应的 URL 字符串。从输出结果中可以看到，值和输入的 URL 相比发生了变化。这是因为在请求 http：//google.com的资源的时候，将会自动定向到“http：//www.google.com/”资源上去。
- 在 In［4］中，使用了返回值的 url 属性来显示其 URL 信息。可以看到其输出值和geturl 方法的输出结果是一样的。实际上，在 geturl 的实现中，就是直接返回 url 属性的。不过在使用的时候，还是推荐使用 geturl 方法。

文件对象支持的另一个方法是 info。此方法返回一个 HTTPMessage 类的实例，它继承于 mimetoos 模块中的 Message 类，其中包含此文件对象头部的所有元信息，包括编码、MIME 类型等。例如可以通过对象的 gettype、getmaintype 和 getsubtype 等方法来获取各种 MIME 类型：

```
In [2]: r = urllib.urlopen("http://www.google.com")

In [3]: m = r.info()

In [4]: m.gettype()
Out[4]: 'text/html'

In [5]: m.getmaintype()
Out[5]: 'text'

In [6]: m.getsubtype()
Out[6]: 'html'

In [7]: m
Out[7]: <httplib.HTTPMessage instance at 0x020E6210>
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［2］中使用 urlopen 获取了一个特定 URL 的资源。在 In［3］中使用 info 方法将其文件对象的头文件元信息复制给变量 m。
- 在 In［4、5、6］中，使用此对象的 gettype 等方法来获取相关的 MIME 信息。从输出信息中可以看出，MIME 类型为 text/html，这也是大部分 HTML 文档的 MIME 类型。
- 在 In［7］中可以看到此对象的具体信息，进一步验证了这是一个 HTTPMessage 类对象。

除了这些信息之外，还可以通过直接使用 items 方法来查看文件头中的相关信息。这些信息还包括连接日期、超时等信息：

```
In [8]: for k, v in r.info().items():
            print k, " = ", v

set-cookie  =  PREF = ID = 660a9d1e73af260a: NW = 1: TM = 1226505364: LM = 1226505364: S =
89yaIItvyI-azS-3; expires = Fri, 12-Nov-2010 15:56:04 GMT; path = /; domain = .google.com
expires = -1
server = gws
connection = Close
cache-control = private, max-age = 0
date = Wed, 12 Nov 2008 15:56:04 GMT
content-type = text/html; charset = ISO-8859-1

In [9]: r.headers
Out[9]: <httplib.HTTPMessage instance at 0x020E6210 >
```

- 在 In［8］中，使用循环打印文件头中已有的所有信息。在输出结果中可以看到，主要的参数包括“set-cookie”、“expires”、“server”等。
- 在 In［9］中输出了文件对象的 headers 属性，可以看到其输出值和 Out［7］是相同的。实际上，info 方法在实现上就是直接返回 headers 成员的。虽然如此，在实际使用的时候，还是建议使用 info 方法来进行处理。

17.4.3　自定义获取资源方式

在 urllib 模块中，还有 URLopener 及其继承类 FancyURLopener，利用它们可以实现对 URL 资源的读取。实际上，urlopen 方法就是一个 FancyURLopener 类的实例，并通过调用其实例的 open 方法来实现的。在这两个类中，一般来说使用 FancyURLopener 类就够了。URLopener类主要是针对协议不是 http、ftp、file 等协议的情况。在本节中，将主要讨论 HTTP 协议的情况。

通过继承 FancyURLopener 类，可以重载其中的多个方法，从而自定义实现对于 URL 资源获取的方式。主要有进行简单认证、自定义特定状态处理函数等。

1. 访问需要简单认证的 URL 资源

现在 Internet 上有不少资源为了保密，需要输入用户名和密码才可以访问。如果直接使用 urlopen 方法来访问这类资源，将不能得到期望的资源。在 FancyURLopener 类中有 prompt_user_passwd方法用来处理用户名和密码。当访问的资源需要使用简单认证进行访问的时候，将会调用此方法并应用得到的用户名和密码。如果不重载此函数的话，默认情况下将会从控制台上读取。见下面的源码片段：

```
def prompt_user_passwd (self, host, realm):
    """ Override this in a GUI environment!"""
    import getpass
    try:
        user = raw_input (" Enter username for % s at % s: " % (realm,
                                                    host))
```

更多资源请访问稀酷客(www.ckook.com)

```
        passwd = getpass.getpass("Enter password for %s in %s at %s: " %
            (user, realm, host))
        return user, passwd
    except KeyboardInterrupt:
        print
        return None, None
```

- 在上面的代码片段中可以看到，prompt_user_passwd 函数使用 raw_input 和 getpass 方法分别从终端中得到用户名和密码，并将此值返回。如果遇到了键盘中断，则返回空值。

访问需要简单认证的 URL 资源时，可以通过重载上面的函数来实现。见下面的代码片段：

```
class myURLOpener(urllib.FancyURLopener):

    def setAuth(self, user, passwd):
        self.user = user
        self.passwd = passwd

    def prompt_user_passwd (self, host, realm):
        return self.user, self.passwd

myurlopener = myURLOpener ()
myurlopener.setAuth (" user", " passwd")

op = myurlopener.open (" http: //www.secret.com")
```

- 这里自定义了一个 URL 资源访问类 myURLOpener。其中定义了两个方法，其中的 setAuth 方法是自定义的，而 prompt_user_passwd 方法则是重载的。对于 setAuth 方法，设置了用户名和密码。而在重载的方法中，这里仅仅是直接返回了用户名和密码。
- myurlopener 是 myURLOpener 类的一个实例，并通过调用 setAuth 方法设置在 prompt_user_passwd 方法中返回的用户名和密码。
- 最后，调用了其 open 方法来访问一个需要简单认证的 URL 资源。这将会自动使用 prompt_user_passwd 方法返回值中的用户名和密码。这里，将会返回一个文件描述符，和使用 urlopen 方法返回的文件描述符是一致的。通过此对象，使用前面介绍过的方法，同样可以获取相关的信息，如文件头元信息等。

2. 自定义文件头元信息

可以直接使用 FancyURLopener 类的实例化来得到一个实例。此实例对象支持多种方法用来添加文件头元信息，从而对访问 URL 资源进行自定义处理。在下面的代码片段中，设置了一些文件头的元信息：

```
In [2]: opener = urllib.FancyURLopener()

In [3]: opener.addheader('User-Agent','Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)')

In [4]: opener.addheader('Accept', 'text/html')

In [5]: opener.addheader('Connection', 'close')

In [6]: opener.open( http://www.baidu.com')
Out[6]: <addinfourl at 22103464 whose fp = <socket._fileobject object at 0x014B1830 > >
```

- 在 In［2］中，直接使用 FancyURLopener 的构造函数生成了一个其类的对象。此对象可以用来预先设置一些文件头的元信息。
- 在 In［3，4，5］中，分别设置了 3 个不同的文件头元信息，包括“User- Agent”、

更多资源请访问稀酷客(www.ckook.com)

"Accept"等。通过这样的设置，可以有效的避开某些 URL 资源对于访问的限制。例如通过设置 User-agent 的值可以用来模拟不同的浏览器。

- 在 In［6］中，使用了其对象的 open 方法，上面设置的文件头元信息将会起作用。同时，返回值和前面使用 urlopen 方法得到的是一样的。

此对象还提供了如 addheaders 这样整体添加的方式，使用方法和前面的类似。只是通过使用列表的方式简化了操作。

另外，对象中还包含 version 属性，此属性主要是用来定义在 urllib 访问 URL 资源时的 User-Agent。这在前面的自定义处理中也遇到了，也可以直接通过改变此值来改变访问 URL 资源时的 User-Agent：

```
In [7]: op.version
Out[7]: 'Python-urllib/1.17'
```

3. 自定义 HTTP 状态处理

FancyURLopener 类中已经提供了部分 HTTP 响应状态码的默认处理程序。这些状态码包括 301、302、303、307 和 401 等，如表 17-8 所示。

表 17-8　HTTP 部分响应状态码

HTTP 状态码	说明
301	请求数据被删除
302	资源临时重定向
303	建议客户端访问其他 URL
307	请求的资源临时性删除
401	错误的请求
404	访问资源不存在

当响应状态码为 401 的时候，可能会需要进行简单的认证，这在前面已经进行了介绍。对于其他的状态码，在默认情况下，将会调用 http_error_default 方法来处理错误的情况。

可以通过重载如 http_error_30x（如 http_error_301 和 http_error_302 等）的函数来自定义相应的处理方式。这里以 http_error_302 方法为例。302 的状态码表示此资源被临时重定向了，同时，在 HTTP 应答中包含此 URL 资源的新地址。处理程序可以使用此地址重新发起请求。在 FancyURLopener 类中已经有了对于此种状态的默认处理方式：

```
def http_error_302(self, url, fp, errcode, errmsg, headers, data=None):
    """Error 302 -- relocated (temporarily)."""
    self.tries += 1
    if self.maxtries and self.tries >= self.maxtries:
        if hasattr(self, "http_error_500"):
            meth = self.http_error_500
        else:
            meth = self.http_error_default
        self.tries = 0
        return meth(url, fp, 500,
                    "Internal Server Error: Redirect Recursion", headers)
    result = self.redirect_internal(url, fp, errcode, errmsg, headers,
                                    data)
    self.tries = 0
    return result
```

- 这里的处理方式主要是在可以重试的次数之内，对新地址进行访问尝试。

可以对此方法进行重载，从而实现特定的 HTTP 状态处理：

更多资源请访问稀酷客(www.ckook.com)

```
def http_error_302(self, url, fp, errcode, errmsg, headers,data=None):
    if headers.has_key('location'):
newurl = headers['location']
    elif headers.has_key('uri'):
        newurl = headers['uri']
    else:
        return
    print url,"==>",new_url
    return urllib.FancyURLopener.http_error_302 (self, url, fp, errcode, errmsg, headers, data)
```

- 在上面的代码片段中，通过重载 http_error_302 方法，可以使得在获取需要重定向 URL 资源的时候调用此方法。在前面的多个判断中，主要是得到当前的新 URL，这是从文件头元信息中的 location 和 uri 键值来获得的。然后打印此新旧 URL 信息。最后，调用 urllib 模块中的默认 302 状态码处理函数。

17.4.4　使用 urllib2 模块获取资源

urllib 模块可以完成的工作都可以使用 urllib2 来完成。一般说来，如果只是需要访问 URL 资源或者是进行简单的认证，使用 urllib 模块也就足够了。但是，当需要以比较灵活的方式访问 URL 资源的时候，就可以使用 urllib2 模块来实现。

1. urllib2 模块的基本使用方法

urllib2 模块的基本使用方法和 urllib 模块是相似的。也有比较相似的函数访问接口，如 urlopen方法等：

```
In [1]: import urllib2

In [2]: fp = urllib2.urlopen("http://www.python.org")

In [3]: fp.read()
Out[3]: '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html xmlns="http://www.
w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n    <meta http-equiv="conte
nt-type" content="text/html; charset=utf-8" />\n    <title>Python Programming Lang
uage -- Official Website</title>\n    <meta name="keywords" content="python progra
mming language object oriented web free source" />\n    <meta name="description" c
ontent="        Home page for Python, an interpreted, interactive, object-oriented
#省略部分输出结果

In [4]: fp.geturl()
Out[4]: 'http://www.python.org'

In [5]: fp.info().items()
Out[5]:
[('content-length', '16578'),
 ('accept-ranges', 'bytes'),
 ('server',
  'Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_ssl/2.2.3 OpenSSL/0.9.8c mod_wsgi/2.0 Python/2.4.4'),
 ('last-modified', 'Thu, 13 Nov 2008 09:29:34 GMT'),
 ('connection', 'close'),
 ('etag', '"6008a-40c2-c0255b80"'),
 ('date', 'Thu, 13 Nov 2008 12:21:49 GMT'),
 ('content-type', 'text/html')]
```

- 在 In［1］中导入了 urllib2 模块，这样 urllib2 模块中的方法才可以在后面被使用。
- 在 In［2，3］中可以看到，urllib2 模块和 urllib 模块一样支持使用 urlopen 方法来打开 URL 资源，并返回一个可读的文件对象。使用 read 方法可以从此文件对象中读取资源。

更多资源请访问稀酷客(www.ckook.com)

从输出结果中可以看到和以前获得的结果是一样的。

- In［4，5］中演示了 geturl 和 info 方法的使用。可以看到，输出值和 urllib 模块中的输出结果也是一样的。

虽然看起来两者的使用方法是相同的，但是在 urllib2 模块中的方法更加灵活。这里所接受的参数除了包括 URL 字符串以外，还可以包括 Request 对象：

```
In [1]: import urllib2

In [2]: request = urllib2.Request("http://www.python.org")

In [3]: fp = urllib2.urlopen(request)

In [4]: fp.read()
Out[4]: '<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" > \n\n\n<html xmlns = "http://www.
w3.org/1999/xhtml" xml:lang = "en" lang = "en" > \n\n<head > \n   <meta http-equiv = "conte
#省略部分输出结果
```

这里使用 Request 类来生成 request 对象，然后通过使用 urlopen 方法来打开此对象，从而同样的实现了前面的功能。

同样地，可选参数 data 为“None”的时候，将会使用 GET 方法来获取 URL 资源。当 data 不为空的时候，则使用 POST 方法将数据传递给 URL 资源：

```
import urllib2
request = urllib2.Request(url = 'http://www.example.com/login.cgi', data = 'data')
fp = urllib2.urlopen(req)
fp.read()
```

这里的 data 数据将发送给 http：//example.com/test.cgi 进行处理。

2. ur2lib2 中的 Handler

在 urllib2 模块中，提供了许多的 Handler，其中，以前通过继承 urllib 模块中类的做法都被 Handler 的做法替代。这些 Handler 包括 ProxyHandler、HTTPBasicAtuhHandler 等众多处理模块。同时，可以通过 build_opener 方法来构造并安装这些处理程序。

【例 17-12】通过一个需要代理服务器来访问一个需要简单认证的页面。

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.proxy.com:3128/'})
proxy_auth_handler = urllib2.HTTPBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'user', 'passwd')

opener = build_opener(proxy_handler, proxy_auth_handler)
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.cgi')
```

- 第 1 行中使用 ProxyHandler 生成了一个代理服务器的 handler，使得在后面访问 URL 资源的时候，将会使用这里的代理服务器。
- 第 2 行中使用 HTTPBasicAuthHandler 建立了一个简单认证的 handler，并在第 3 行中设置了用户名和密码。
- 第 5 行使用 build_opener 方法构建了一个自定义资源获取对象，并在第 6 行中对此调用 install_opener 进行安装。
- 第 7 行中使用 urllib2 模块中的 urlopen 方法来访问指定的 URL 资源。此时，将会通过指定的代理服务器进行访问。当遇到需要简单认证的页面时，则可以输入对应的用户名和密码来通过认证，从而获取资源。

更多资源请访问稀酷客(www.ckook.com)

【例 17-12】中的代码片段还有如下一种写法：

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.proxy.com:3128/'})
proxy_auth_handler = urllib2.HTTPBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'user', 'passwd')

opener = build_opener(proxy_handler, proxy_auth_handler)
opener.open('http://www.example.com/login.cgi')
```

- 和【例 17-12】的主要区别在于最后一行。这里并没有对此资源获取的对象进行安装，而是直接调用了其 open 方法来打开 URL 资源。

在 urllib2 模块中，定义了众多的 handler 处理程序。甚至还包括用来访问 HTTPS 协议的相关类。可以说，urllib2 模块能够处理非常复杂的 URL 资源获取情况。

17.4.5　使用 httplib 模块获取资源

httplib 模块实现了 HTTP 和 HTTPS 协议的客户端部分。一般情况下，这个模块并不直接使用，而是作为 urllib 的基础模块。但是实际上，此模块还是比较适用于在访问 HTTP 资源的时候使用，包括 GET 和 POST 方法等。

在 httplib 模块中提供了 HTTPConnection（HTTPSConnection）和 HTTPResponse 类，用来处理具体的 URL 资源。在 HTTPConnection 类的构造函数中，其输入参数为需要访问的 HTTP 服务器。当省略端口的时候，默认使用 80 端口。下面是 3 种 HTTPConnection 实例对象的构造方式：

```
In [2]: conn1 = httplib.HTTPConnection("www.python.org")

In [3]: conn2 = httplib.HTTPConnection("www.python.org:80")

In [4]: conn3 = httplib.HTTPConnection("www.python.org",80)
```

- In［2］中是默认的构造方式，此时将会默认使用 80 端口。注意，这里的访问都是通过 HTTP 来访问的。
- In［3］中的输入参数使用了冒号来分隔 URL 地址和端口。
- 在 In［4］中使用了第二个参数 port，并将 80 端口值赋值给这个变量。

生成了连接对象后，就可以使用其中的方法来获取对应的资源了。主要可以使用的方法是 request。该方法接受两个参数，一个是访问的方法，可以为“GET”和“POST”，另外一个是需要获取的 URL 资源：

```
In [5]: conn1.request("GET","/index.html")

In [6]: r1 = conn1.getresponse()

In [7]: print r1.status, r1.reason
200 OK

In [8]: r1.read(256)
Out[8]: '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html xmlns="http://www.
w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n    <meta http-equiv="conte
nt-type" content="text/html; ch'

In [9]: conn1.close()
In [12]: conn2.request("GET","/test.html")
```

更多资源请访问稀酷客(www.ckook.com)

```
In [13]: r2 = conn2.getresponse()

In [14]: print r2.status, r2.reason
404 Not Found

In [15]: data2 = r2.read()

In [16]: data2
Out[16]: '<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/
/www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" > \n\n\n <html xmlns = "http://www
.w3.org/1999/xhtml" xml:lang = "en" lang = "en" > \n\n <head > \n    <meta http-equiv = "cont
#省略部分输出结果
k Links -- > \n   </div > \n\n   <div id = "content-body" > \n      <div id = "body-main" > \n
     <div id = "content" > \n          \n          <div id = "breadcrumb" > \n              \
n            Page Not Found \n          </div > \n\n\n\n          <! --utf-8-- > <! --0.4
.1-- > <h1 class = "title" > Error 404: File Not Found </h1 > \n <p > The URL you requested
was not found on this server. </p > \n <p > Try our <a class = "reference" href = "/" > home
#省略部分输出结果

In [17]: "Error 404: File Not Found" in data2
Out[17]: True

In [18]: conn2.close()
```

- 在 In［5］中，调用 request 方法来获取指定服务器的 index.html 文档资源。当请求资源完毕后，可以使用 getresponse 方法来得到返回值。这是 HTTPResponse 类的一个对象。需要注意的是，当再次使用 request 来获取指定的 URL 资源的时候，需要先将流中的数据读取完毕。否则会发生错误。
- 此 HTTPResponse 实例对象含有一些属性，用来指示此访问的一些信息。例如，status 属性表示服务器返回的状态码，reason 属性表示服务器返回状态的原因。在 In［7］中，显示了此访问请求的状态码和原因。从输出结果中可以看到，状态码为 200，表示请求成功。
- 在 In［8］中读取了一定字节的资源。如果在后面需要利用此连接对象来继续请求 URL 资源，则需要将这里缓冲的数据读取完毕。
- In［9］中通过调用 close 方法可以将此连接对象关闭。
- 在下面的代码片段中，访问了一个并不存在的 URL 资源。这在 In［14］的输出结果中可以看到，其服务器返回的状态码为 404，表示找不到请求的 HTML 文档。需要注意的是，服务器也返回了一个页面，其中给出了信息说明请求的文件找不到。通过 In［17］可以看到，在输出结果中有“Error 404：File Not Found”的字样。

如果需要自定义请求，在 httplib 中可以使用 putrequest、putheader 方法来实现。如下面的代码片段：

```
In [25]: conn3.putrequest("GET", "/index.html")

In [26]: conn3.putheader("User-Agent","Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)")

In [27]: conn3.putheader("Accept", "* /* ")

In [28]: conn3.endheaders()

In [29]: r3 = conn3.getresponse()

In [30]: print r3.status, r3.reason
```

更多资源请访问稀酷客(www.ckook.com)

```
200 OK"

In [31]: data3 = r3.read(256)

In [32]: data3
Out[32]: '<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/
/www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" > \n\n\n<html xmlns = "http://www
.w3.org/1999/xhtml" xml:lang = "en" lang = "en" > \n\n<head > \n   <meta http-equiv = "cont
ent-type" content = "text/html; ch'
```

- 在这个代码片段中，没有使用 request 方法来访问 URL 资源，而是通过自定义的方式访问的。这里，首先使用 putrequest 方法来访问特定的 URL 资源。
- 在 In［26，27，28］中，设置了一些文件头的元信息。
- 通过 getresponse 方法获取相应的 URL 资源。通过查看其 status 和 reason 属性，可以看到其和 conn1 得到的输出结果是一样的。

17.5　HTML 文档的解析

在获取资源后，一般需要对 HTML 文档进行进一步处理。Python 中提供了 HTMLParser模块来解析 HTML 文档。另外，由于 HTML 语言是属于 SGML 语言家族的一部分，所以也可以使用 sgmllib 来处理 HTML 文档，为 HTML 文档的进一步处理做初步的分析。htmpllib 是建立在 sgmllib 模块上的 HTTP 文档高级处理模块，可以对 HTML 文档进行更细的处理。

17.5.1　使用 HTMLParser 模块

HTMLParser 是一个用来解析 HTML 文档的模块，具有小巧、快速和使用简便的特点。利用它可以分析 HTML 文档中的标签和数据等。HTMLParser 采用了一种事件驱动的方式，使得模块在找到一个特定的对象（如标签或者数据）的时候，可以调用用户定义的函数来进行处理。这里，用户定义的函数名称都是以 handle_开头的，是 HTMLParser 的成员函数。

在具体使用的时候，可以从 HTMLParser 中派生出新的类，然后重新定义这些以 handle_开头的函数。当此派生类生成的对象遇到特定的对象的时候，就会调用对应的处理函数。

在 HTMLParser 中定义的 handle_函数如表 17-9 所示。

表 17-9　HTMLParser 中的 handle_函数

函数名称	函数用途
handle_starttag	处理开始标签和结束标签在一起的，如 < br / >
handle_startendtag	处理开始标签
handle_endtag	处理结束标签
handle_data	处理数据
handle_charref	处理字符实体
handle_entityref	处理实体参考
handle_comment	处理注释
handle_decl	处理定义
handle_pi	处理“处理指令”

更多资源请访问稀酷客(www.ckook.com)

在默认情况下，表 17-9 中的这些处理函数都不会做任何的处理。当需要处理特定的元素的时候，可以重载对应的函数，并加入自定义的处理部分。

例如在前面的示例中，所要获取的资源有可能不存在。此时，服务器一般会返回状态码为 404 的错误，从而指示所请求的资源不存在。另外，服务器还会返回包含特定字符串的页面，用来显式的通知用户。如果这些页面中有“404 Not Found”或者“Error 404”等类似的字符串，则认为当前所要获取 URL 资源不存在。下面的代码片段对一组 URL 来判断是否含有相应的 URL 资源。判断的方法还是通过前面说的这种字符串匹配方式：

```
#filename: htmlparser_1.py
import urllib
import urlparse
import HTMLParser

class CheckHTML (HTMLParser.HTMLParser): #从 HTMLParser 类中继承
    available = True
    def handle_data (self, data): #定义处理数据的方法
        if " 404 Not Found" in data or " Error 404" in data: #当含有特定字符串的时候
            self.available = False

check_urls = [" index"," test"," help"," news", " faq"," download"] #需要检查的 URL

for url in check_urls:
    new_url = urlparse.urljoin (" http: //www.python.org/", url) #拼合 URL
    fp = urllib.urlopen (new_url) #打开 URL 资源
    data = fp.read () #读取 URL 资源
    fp.close ()

    p = CheckHTML () #生成一个 CheckHTML 类对象
    p.feed (data) #解析上面获得的数据
    p.close ()

    #判断 URL 是否存在
    if p.available:
        print new_url, " == > OK"
    else:
        print new_url, " == > Not Found"
```

- 在代码片段的最前面，导入了必需的几个模块，包括 urllib、urlparse 和 HTMLParser 等。
- 接下来定义了一个 CheckHTML 类。此类继承自 HTMLParser 类。在类的实现中，仅仅是重载了 handle_data 函数。当获取到数据的时候，则判断是否其中含有“404 Not Found”或者是“Error 404”字符串。如果有的话，则认为此 URL 资源不存在。
- 在下面的循环中，对于需要检查的每个 URL 资源都执行同样的操作。首先是将 URL 拼合起来，然后使用 urlopen 方法来获取相应的 URL 资源，并将获取的资源使用 CheckHTML 对象的 feed 方法判断变量的值，来输出“OK”或者是“Not Found”。

上面代码片段的运行结果如下：

```
In [3]: % run htmlparser_1.py
http: //www.python.org/index == >OK
http: //www.python.org/test == >Not Found
http: //www.python.org/help == >OK
http: //www.python.org/news == > OK
http: //www.python.org/faq == > Not Found
http: //www.python.org/download == > OK
```

- 从运行结果来看，以上代码可以有效的用来判断 URL 资源的有效性。要注意的是，这里的判断方法并不是很可靠。如果在正常的文档中出现了如“Error 404”这样的字符

更多资源请访问稀酷客(www.ckook.com)

串，也会认为此 URL 资源不存在，所以需要根据情况来调整判断标准。

17.5.2　sgmllib 的 HTML 文档处理

虽然 sgmllib 模块是用来处理 SGML 文档的，但该模块实际上并没有实现一个完整的 SGML 文档解析器。更确切地讲，sgmllib 仅仅只是实现了 HTML 文档中使用的一部分。另外一个原因是这个模块是作为 htmllib 模块的基类。

sgmllib. py 文件位于 Python 安装目录的 Lib 目录下，其中包含 sgmllib 模块的实现。另外，此文件本身已经提供了简单的 HTML 文档处理功能。当直接使用此文件的时候，将会对参数中的文件进行解析。如果不提供参数，则默认处理 test. html 文件。下面是此 sgmllib. py 文件针对 first. html 文档的处理结果：

```
In [1]: % run sgmllib.py first.html
start tag: <html>
data: '\n'
start tag: <head>
#省略部分输出结果
data: '\n'
end tag: </head>
data: '\n'
start tag: <body>
data: '\n'
start tag: <h2>
data: '\xb5\xda\xd2\xbb\xb8\xf6HTML\xd2\xb3\xc3\xe6'
end tag: </h2>
data: '\nHello,'
start tag: <b>
data: 'World'
end tag: </b>
data: '\n'
comment: ' \xd5\xe2\xca\xc7\xd7\xa2\xca\xcd'
data: '\n'
#省略部分输出结果
data: '\n\xcd\xf8\xd5\xbe\xc9\xcf\xd5\xd2\xb5\xbdHTML\xb5\xc4\xd3\xef\xd1\xd4'
data: '\xb9\xe6\xb7\xb6\n'
#省略部分输出结果
end tag: </html>
data: '\n\n'
```

- 通过运行此 sgmllib. py 文件，可以看到它对所有的标签都进行了识别和输出。主要包括开始标签、结束标签、注释和数据等。另外，其中还包含“\n”的数据。虽然这些数据在 first. html 文档中是不可见的，但是实际上在 HTML 文档处理的时候，还是作为文档的一部分数据来处理的。另外可以注意到，这里的中文并没有正确的显示，这主要是编码的原因。

17.5.3　HTML 文档元素的处理

在上面的使用中出现相应结果的原因是在 sgmllib. py 中定义了一个测试类 TestSGMLParser，可以输出部分的标签信息。下面是其 TestSGMLParser 类的代码：

```
class TestSGMLParser(SGMLParser):

    def __init__(self, verbose=0): #初始化函数,实例化的时候调用
        self.testdata = ""
        SGMLParser.__init__(self, verbose)
```

更多资源请访问稀酷客(www.ckook.com)

```
    def handle_data (self, data): #处理数据
        self.testdata = self.testdata + data
        if len (repr (self.testdata)) >= 70:
            self.flush ()

    def flush (self): #对于数据的具体处理
        data = self.testdata
        if data:
            self.testdata = ""
            print 'data: ', repr (data)

    def handle_comment (self, data): #处理注释信息
        self.flush ()
        r = repr (data)
        if len (r) > 68:
            r = r [: 32] + '...' + r [-32:]
        print 'comment: ', r

    def unknown_starttag (self, tag, attrs): #处理开始标签
        self.flush ()
        if not attrs:
            print 'start tag: <' + tag + '>'
        else:
            print 'start tag: <' + tag,
            for name, value in attrs:
                print name + '=' + '"' + value + '" ',
            print '>'

    def unknown_endtag (self, tag): #处理结束标签
        self.flush ()
        print 'end tag: </' + tag + '>'

    def unknown_entityref (self, ref): #处理实体参考
        self.flush ()
        print '* * * unknown entity ref: &' + ref + '; '

    def unknown_charref (self, ref): #处理字符实体参考
        self.flush ()
        print '* * * unknown char ref: &#' + ref + '; '

    def unknown_decl (self, data): #处理声明
        self.flush ()
        print '* * * unknown decl: [' + data + '] '

    def close (self): #关闭处理过程
        SGMLParser.close (self)
        self.flush ()
```

- TestSGMLParser 类继承自 SGMLParser 类。
- 类中重载了多个方法，从而达到了处理 HTML 文档的目的。这些方法包括 handle_data、handle_comment 和 unknown_starttag 等。
- 在上面的代码中，很多方法都对数据进行了限定和相应的处理，这也是在处理 HTML 文档时候的一个好习惯。

除了可以重载在 SGMLParser 类中已有的方法外，还可以使用 start_tag、do_tag 和 end_tag 的方式来处理特定的 tag。

【例 17-13】将一个 HTML 文档中的所有超链接都提取出来。

更多资源请访问稀酷客(www.ckook.com)

```
#filename: sgmllib_1.py
#导入必须的模块
import sgmllib
import urllib

class LinkDemo(sgmllib.SGMLParser):#从 SGMLParser 类中继承
    def ___init___(self): #构造函数
        sgmllib.SGMLParser.___init___(self)
        self.links = [] #初始化时设置 links 为空

    def start_a(self, attributes): #处理 <a> 标签
        for link in attributes:
            tag, attr = link[:2]
            if tag == "href": #当取得了 URL 的时候将其加入到 links 中
                self.links.append(attr)

f = urllib.urlopen("http://www.baidu.com") #打开指定的 URL 地址
data = f.read()
f.close()

ld = LinkDemo() #实例化一个 LinkDemo 对象
ld.feed(data)

for i,link in enumerate(ld.links): #打印相关的信息
    print i," == > ",link
```

- 这里的 LinkDemo 类主要需要注意的地方是定义了一个 start_a 的方法。此方法将在遇到 <a> 标签的时候被调用。其方法内部的实现主要是将连接的 URL 提取出来，并将其放在一个内部变量 links 中。
- 通过 17.4 节中介绍的方法，获取了特定 URL 的资源。
- 接着生成了一个 LinkDemo 类的实例，并将得到的 URL 资源数据送入处理。
- 最后打印出获得的超链接信息。

上面代码片段的运行结果如下：

```
In [1]: run sgmllib_1.py
0 == > http://passport.baidu.com/? login&tpl=mn
1 == > http://news.baidu.com
2 == > http://tieba.baidu.com
3 == > http://zhidao.baidu.com
4 == > http://mp3.baidu.com
5 == > http://image.baidu.com
#省略部分输出结果
12 == > http://utility.baidu.com/traf/click.php? id=215&url=http://www.baidu.com
13 == > http://jingjia.baidu.com
14 == > http://top.baidu.com
15 == > /home.html
16 == > http://ir.baidu.com
17 == > http://www.baidu.com/duty/
18 == > http://www.miibeian.gov.cn
```

- 从输出结果中可以看到，当前 URL 资源 http://www.baidu.com 中共有 19 个超链接。

通过重载 SGMLParser 中的方法，以及定义特定标签的处理函数，可以实现灵活处理 HTML 文档的目的。

17.5.4　使用 htmllib 处理 HTML 文档

htmllib 模块提供了一种标签驱动的 HTML 文档解析方式。解析后的数据将送给 formatter 对象进行处理。在此模块中包含 HTMLParser 类。通过继承 HTMLParser 类可以有效地对HTML文

更多资源请访问稀酷客(www.ckook.com)

档进行处理。需要注意的是,此类中没有包含数据资源的获取过程,所以在使用的时候,需要由文件对象获取数据资源。

此模块是基于 sgmllib 模块的,所以在 sgmllib 模块中的方法在这里都是适用的。例如在【例 17-13】的 LinkDemo 类中,将 SGMLParser 换成 HTMLParser 也是可以的。

在 HTMLParser 类中,已经支持在 XHTML 1.0 规范中所定义的各种实体名称。同时,对于大部分元素也有了相应的处理函数。如果需要自定义处理 HTML 文档,只需要重载相应的处理函数,加入自己的处理部分就可以了。

【例 17-14】对【例 17-13】中的代码进行改写——将 HTML 文档中的超链接找出来,并输出更加详细的信息。

```
#filename: htmllib_1.py
import htmllib
import urllib
import formatter
import string

class LinkDemo(htmllib.HTMLParser): #从 HTMLParser 类中继承

    def __init__(self, verbose=0): #初始化的时候调用,将 links 设置为空。这里的 links 为字典结构
        self.links = {}
        f = formatter.NullFormatter()
        htmllib.HTMLParser.__init__(self, f, verbose)

    def anchor_bgn(self, href, name, type): #锚点标签开始的时候处理
        self.save_bgn()
        self.link = href

    def anchor_end(self): #锚点标签结束的时候处理
        text = string.strip(self.save_end())
        if self.link and text:
            self.links[text] = self.links.get(text, []) + [self.link]

fp = urllib.urlopen("http://www.baidu.com") #打开指定的 URL
data = fp.read()
fp.close()

linkdemo = LinkDemo() #实例化一个 LinkDemo 对象
linkdemo.feed(data) #处理输入
linkdemo.close()

for href, link in linkdemo.links.items(): #打印相关的信息
    print href, "=>", link
```

- 这个代码片段中定义的类是继承自 HTMLParser 类。其中对构造函数进行了自定义。另外还重载了 HTMLParser 类中的 anchor_bgn 和 anchor_end 方法。这两个方法主要是用来对遇到的超链接开始标签和结束标签进行处理。
- 在构造函数中,定义了一个变量为 links 的字典数据结构,主要用来保存已经处理过的超链接信息。然后定义了一个 formatter 类型的 NullFormatter,这个格式化类型的 NullFormatter 将对传过来的数据默认不进行处理。在 formatter 模块中,还可以自定义各种数据格式化类。
- 第 23 行至第 25 行代码主要是通过 urlopen 来获取指定 URL 的资源。在这里,将所有的数据都读取出来赋值给 data,供以后使用。
- 第 27 行至第 29 行代码首先生成了一个 LinkDemo 类的实例对象。通过调用此实例对象

更多资源请访问稀酷客(www.ckook.com)

的 feed 方法，对获取的 URL 资源进行处理。处理的过程中如果遇到超链接，会自动调用在 LinkDemo 类中定义的函数来进行处理。

- 最后，将这些超链接信息输出。

【例 17-14】的输出结果如下所示：

```
In [1]: run htmllib_1.py
企业推广 => ['http://jingjia.baidu.com']
关于百度 => ['/home.html']
新闻 => ['http://news.baidu.com']
使用百度前必读 => ['http://www.baidu.com/duty/']
登录 => ['http://passport.baidu.com/?login&tpl=mn']
京 ICP 证 030173 号 => ['http://www.miibeian.gov.cn']
视频 => ['http://video.baidu.com']
空间 => ['http://hi.baidu.com']
About Baidu => ['http://ir.baidu.com']
知道 => ['http://zhidao.baidu.com']
贴吧 => ['http://tieba.baidu.com']
更多>> => ['/more/']
MP3 => ['http://mp3.baidu.com']
图片 => ['http://image.baidu.com']
搜索风云榜 => ['http://top.baidu.com']
把百度设为主页 => ['http://utility.baidu.com/traf/click.php?id=215&url=http://
www.baidu.com']
高级 => ['/gaoji/advanced.html']
hao123 => ['http://www.hao123.com']
帮助 => ['/search/jiqiao.html']
```

- 在输出结果中可以看到，与【例 17-13】相比，不仅显示了超链接，而且显示了其文字信息。

除了 NullFormatter 类外，formatter 模块中还提供了 AbstractFormatter 类来对数据进行格式化处理。在构造此类的实例对象时，需要提供一个 writer 对象。这里的写对象对应着多种格式化事件。在 formatter 模块中，已经提供了一个类 AbstractWriter。这个类实现的功能只是简单的打印每个函数调用的信息：

```
In [1]: import htmllib

In [2]: import formatter

In [3]: w=formatter.AbstractWriter()

In [4]: f=formatter.AbstractFormatter(w)

In [5]: import urllib

In [6]: fp=urllib.urlopen("http://www.baidu.com")

In [7]: data=fp.read()

In [8]: fp.close()

In [9]: p = htmllib.HTMLParser(f)

In [10]: p.feed(data)

    send_flowing_data('body{margin:4px
0}p{margin:0;padding:0}img{border:0}td,p,#u{f
    ont-size:12px}#b,#u,#l                td,a{font-family:arial}#kw{font:16px
Verdana;height:1.78e
    m;padding-top:2px}#b{height:30px;padding-top:4px}#b,#b
a{color:#77c}#u{padding-r
```

更多资源请访问稀酷客(www.ckook.com)

```
ight:10px;line-height:19px;text-align:right;margin:0  03px ! important;margin:0 0
#省略部分输出结果
send_flowing_data('[2]')
new_font((None, None, 1, None))
send_flowing_data('\xcd\xf8')
send_flowing_data('[7]')
send_flowing_data(' var w=document. f. wd;w. focus();function s(o){if(w. value. lengt
h>0){var h=o. href;var q=encodeURIComponent(w. value);if(h. indexOf("q=")!  =-1){o. hr
ef=h. replace(new RegExp("q=[^')
#省略部分输出结果
send_flowing_data('[19]')
send_flowing_data(' (image)')

In [11]: p.close()
```

- In［1，2，5］分别导入了代码片段运行所需要的 htmllib、formatter 和 urllib 模块。
- 在 In［3，4］中，定义了一个数据格式化处理函数。这里的主要处理方式是直接输出处理数据时候的函数调用信息。
- In［6，7，8］中通过 urlopen 获取了相应的 URL 资源。
- 在 In［9］中构造了一个 HTMLParser 文档处理实例，将前面生成的格式化处理函数作为了参数。然后调用此对象的 feed 方法。可以看到，在输出中打印出了数据处理的流程。最后，调用 close 方法将其关闭。

通过对 writer 对象进行自定义，可以实现 URL 资源数据的特定输出。如将所有的标签去掉、针对加粗标签进行特殊处理等。

一般说来，如果只是需要对 HTML 文档进行解析，则使用 sgmllib 模块就足够了。但是如果还需要对数据进行重新展示，则需要利用 htmllib 来完成。

17.6　小结

本章讲解了 Python 语言在 HTML 方面的应用。在介绍了 HTML 语言相关的概念后，对 URL 的处理（包括解析、拼合和分解）进行了详细地介绍。同时，对 URL 和查询参数的编码与解码也一并进行了介绍。然后介绍了 CGI 编程，包括 CGI 环境和对用户输入的处理。接着介绍了通过使用 urllib、urllib2 和 httplib 模块中的多种方式来获取 URL 资源。最后介绍了 HTML 文档的解析，包括使用 HTMLParser 模块、sgmllib 模块和 htmllib 模块等。

更多资源请访问稀酷客(www.ckook.com)

第 18 章　Python 和 XML

XML（eXtensible Markup Language）技术已经日趋成为当前许多新生技术的核心，在许多领域都有着不同的应用。Python 语言中已经包含了对 XML 的支持。本章将介绍如何使用Python来分析 XML 文件。本章的知识点：

- XML 的良构性和有效性
- XML 的文档结构
- XML 简单 API 的原理
- xml. sax 模块的使用方法
- 文档对象模型（DOM）原理
- xml. dom 模块的使用方法

18. 1　XML 介绍

XML 并不是一种完全创新的标记语言，而是 Web 发展到一定阶段的必然产物。这种语言既具有 SGML 的核心特征，又有着 HTML 的简单特性，另外还具有定义严格、语法明确和结构良好等许多新的特性。

18. 1. 1　XML 的演进历史

XML 是 W3C 规范的一种文档标记语言。XML 使用一种简单可读的标签和通用的语法来标记文档数据，为计算机处理数据提供了一种统一的格式。这种格式具有足够的灵活性，使得其可以用在电子数据交换、矢量图形、对象序列化等领域的系统中。

由于 HTML 的巨大成功，因此其语言标准在迅速演进的同时，也加入了比最初设计要复杂得多的功能。加入这些功能的目的是使得 HTML 能够满足多领域包括科学研究和商业应用的任务要求。但是最终结果是 HTML 失去了最初的简单性，而且在使用上也出现了随意和不规范的问题。另外，由于 HTML 的扩展性不强，用户不能自由地定义新标签并进行有效性验证，从而导致了行业之间数据交换的不便。

随着互联网的进一步发展，为了解决 HTML 语言的应用局限性，W3C 开始尝试在 Web 上引入 SGML 语言。SGML 语言虽然复杂，但是也有 HTML 语言所不具有的优势，那就是可扩展性、文档的结构化和灵活性。最终，在 1998 年 2 月，W3C 发布了 XML 版本 1. 0 规范。

在 1. 0 规范的摘要中是这样定义 XML 的：“可扩展标记语言（XML）是标准通用标记语言（SGML）的一个子集。定义 XML 规范的目的是要使得在 Web 上能以现有超文本标记语言（HTML）的使用方式，接收和处理通用的 SGML 的方式成为可能。XML 语言的设计中需要考虑到实现的方便性，同时也要注意到不同语言之间的互操作性。”

更多资源请访问稀酷客(www.ckook.com)

18.1.2　XML 的优点和限制

XML 是一种元标记语言，在 XML 文档中的数据都是以字符串的形式保存下来的。需要注意的是，XML 仅仅包含数据，而对于数据的处理并不在 XML 的文档中。XML 文档的基本组成单元是元素，在 XML 规范中就对元素的定义进行了规定。范围包括：如何分隔标签，如何放置数据，如何放置属性等。从表面上看，XML 文档的标签和 HTML 文档中的标签没有什么不同，但是实际上两者有着重大的区别。

在 XML 这个缩写词中，其中的 X 代表可扩展性，也就是说语言可以根据不同的需求进行扩展。XML 语言的“元标记性”使得其运行用户创建自己的专用标签，用来适应不同应用的需要。这和 HTML 这种预设置标记语言不同，在 HTML 中只能使用规范中已经定义的各种标签，不能随意添加标签。而 XML 则允许用户根据需要自己定义标签。如数学工作者可以根据自己工作的需要定义各种公式标签等，化学工作者可以使用元素来描述分子、原子等。尽管 XML 在允许应用的元素方面非常灵活，但同时 XML 还是保持着非常严格的语法检查。XML 中提供了一系列规范，用来规定在 XML 文档中标签应该如何放置、什么样的元素名称是合法的、元素有哪些属性等。这些规定使得 XML 解析器可以有效的处理 XML 文档。满足 XML 规范的 XML 文档被称为是良构（Well-form）的。XML 中存在任何一个语法错误都是非良构的，并且也不会通过 XML 解析器的处理。

由于互操作性的原因，组织或者个人可能仅仅只是使用某一部分标签，并对其进行规范，这些标签的集合就被称为 XML 应用。一个 XML 应用并不是说使用 XML 的应用软件，而是 XML 语言在某一应用领域的使用。XML 描述了文档的结构和语义，但是没有对其数据显示进行规定。如一个元素可能是日期或者人名等，但是不会规定元素显示的时候是使用加粗还是斜体。也就是说，XML 是一种结构化的语义标记语言，而不是一种数据表示语言。

可以将 XML 允许应用的标记规范都放在一个文件中，这被称为 XML 模式（XML Schema）。一个 XML 文档可以和某个模式比较。如果匹配的话，则表示此文档是合法（valid）的，而不匹配的时候则为无效的。XML 文档的是否有效只是和 XML 模式的比较有关系，所以不是所有的 XML 文档都是有效的，在许多情况下，只需要文档是良构的就可以了。

现在已经有很多 XML 模式语言，其中使用最广泛的是在 XML 规范中规定的 DTD（Document Type Definition,文档类型定义），所以 XML 文档是自描述的，可供计算机处理，同时数据也可以重用。除此之外，XML 还具有简单性，使得其易学易用且容易实现。

由于 XML 语言的众多优点，因此成为了很多配置文件的首选。另外，XML 和网络结合起来，也形成了一种互相通信的标准方法，包括简单对象访问协议（SOAP）和 XML-RPC 等。但是由于对 XML 的无限制使用，现在的 XML 文档也是非常的复杂。

特别要注意的是，XML 仅是一种标记语言，不是无所不能的。XML 文档本身只是作为数据描述，而不能做更多的操作。现在的项目中使用 XML 作为配置文件，但是最终还是应用程序来处理 XML 文件，而不是其本身。另外，XML 也不是一种网络传输协议。XML 不会向网络发送数据，而仅仅只是有可能使用 HTTP、FTP 等传递的数据会采用 XML 编码。这些数据还是需要另外的工具来处理，而不是 XML 文档。

最后需要说明的是，XML 不是数据库，不会取代如 Oracle 或者 MySQL 等数据库。数据库中可以包含 XML 数据，但是数据库本身并不是 XML 文档。同样地，可以保存 XML 数据或者读取 XML 数据，但是这些工作都需要用另外的工具来实现。XML 可以作为一个很好的数据交换方法，但是本身不具备数据处理的能力。

更多资源请访问稀酷客(www.ckook.com)

18.1.3　XML 技术的 Python 支持

由于 XML 语言的迅速发展，现在已成为包含 DTD、XML Schema、XPath、XSLT 等许多相关技术的一个集合了。另外，由于 XML 对于多种语言的支持，现有的支持多种 XML 处理技术和语言的库都是比较复杂的。幸运的是，Python 语言的标准库中提供了对于 XML 文档的基本处理。在保留最基本和最重要操作的同时，简化了各种复杂的操作，降低了处理开销。除此之外，Internet 上已经有了许多用来处理 XML 文档的相关库，它们是标准库的很好的补充。

XML 的处理方式有两大类：SAX（Simple API for XML，XML 的简单 API）和 DOM（Document Object Model，文档对象模型）。虽然 DOM 技术在 SAX 技术后面出现，并且是作为 SAX 替代者的形象出现的，但是实际上这两类处理方式都有各自的优缺点，设计者可以根据需要选用。

SAX 采用的是事件驱动方式来处理 XML 文档。这使得当其遇到一个特定标签的时候，将会调用指定的方法，从而实现对特定标签的处理。这种处理方式的另一个特点是采用流数据方式来处理 XML 文档。这使得应用程序在解析的时候不需要将整个 XML 文档都读取完毕后才能进行处理——可以在读取了部分数据后就进行处理，而且可以在满足某个条件的时候终止对于 XML 文档的解析。这样的处理方式使得处理速度相对来说是比较快的。

在 Python 库中主要实现 SAX 处理方式的是 xml. sax 模块。此模块中提供了 4 类数据的处理器定义，分别是内容处理器、DTD 处理器、实体解析处理器和错误处理器。通过实现每个处理器中定义的特定的函数，可以实现对特定元素的处理。除此之外，Python 库中还提供了 xml. parsers. expat 模块。这是一个轻量级的基于 SAX 处理方式的 XML 解析模块。需要注意的是，这里的处理将不会对 XML 文档的良构性进行保证，而只是在遇到出现处理错误的时候才报告出现的问题。

DOM 是一种通过树结构和数据来表示的信息集合。其实这是一种与系统平台和语言无关的表示和处理 XML 文档的 W3C 标准。使用 DOM 方式处理 XML 文档的时候，会将整个 XML 文档转换为一个由其数据组成的树结构。这种处理方式使得可以很容易访问各个元素。另外，添加、修改和删除特定的元素也是比较容易的，所以这种处理方式一般用于需要对 XML 文档频繁更改的情况下。显然，这种处理方式将消耗大量的内存和处理器时间，对处理机器的要求也比较高。但是由于其设计精良，因此对于 XML 的访问非常简便。

在 Python 库中，xml. dom 模块实现了 DOM 访问 XML 文档的处理方式。模块中提供了对树中节点以及元素访问的多种方法。另外，还提供了 xml. dom. minidom 模块来简化使用 DOM 处理方式，实现了 DOM 中的大部分功能。除此之外，还有一个在 Python 语言版本 2. 5 中引入的 ElementTree 模块。此模块提供了一种对于 XML 文档的快速处理方式。

以上提到的模块在后面将会进一步介绍，包括其处理 XML 文档的详细实现。除了这些标准库中自带的 XML 处理模块以外，还有一些第三方提供的优秀 XML 处理组件，其中比较突出的是 lxml，其网址是 http：//codespeak. net/lxml/。实际上这是一个 XML 库的 Python 语言绑定。此模块的使用方式和 ElementTree 模块很相似。在后面，也会利用到该库中的一些功能。

18.2　XML 文档概览和验证

XML 的标签远比 HTML 的标签使用灵活，但是另一方面，XML 的标签使用要比 HTML 严格得多。这主要是要求 XML 文档必须是良构的。相对于 HTML 而言，所有的开始标签必须要有结束标签，所有的属性值必须使用引号等。虽然这些规则使得书写 XML 文档的时候会要困难一些，但是这有助于 XML 文档的规范化。

更多资源请访问稀酷客(www.ckook.com)

18.2.1　XML 文档的基本概念

由于 XML 文档使用文本字符串来存储数据，所以可以使用任何的文本编辑器来编辑 XML 文档。当然，现在也有不少 XML 特定应用的编辑器，为 XML 编辑提供了很多特色支持。

【例 18-1】编辑一个比较简单的 XML 文档（文件名：xml_1. xml）。

```
<? xml version = '1. 0' encoding = 'utf-8' standalone = 'no' ? >
<goods >
    <shirt >
        <name >Helen < /name >
        <size >170 < /size >
        <quantity >10 < /quantity >
        <color >black < /color >
    < /shirt >
    <shirt >
        <name >First < /name >
        <size >175 < /size >
        <quantity >9 < /quantity >
        <color >white < /color >
    < /shirt >
<goods >
```

从【例 18-1】的 XML 文档中可以看到，即使没有其他文档的介绍，此文档也有着很好的自描述性。可以使用很多工具来处理 XML 文档，如 Internet Explorer（IE），如图 18-1 所示。

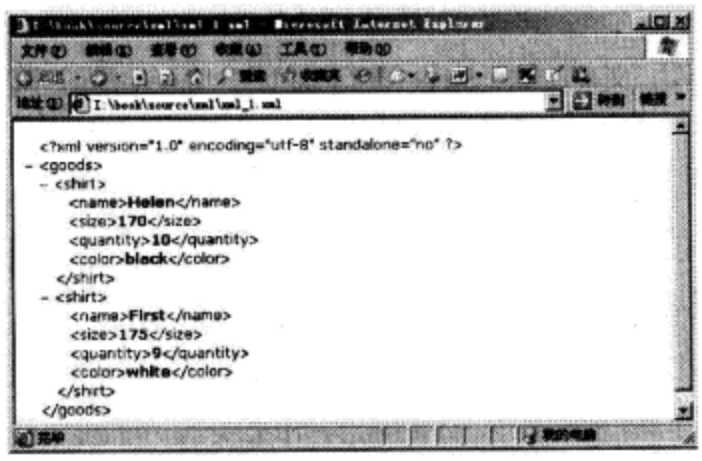


图 18-1　修改 path 环境变量

在如图 18-1 所示窗口中，IE 使用了一种树状的形式将 XML 表现了出来。可以看到在一些元素的前面有可以折叠的标记，这是因为其包含子元素。虽然这里使用的是简单的数据，但是实际上，XML 中可以保存任何类型的数据。

虽然 XML 文档是使用文本格式来描述，但是对文件的保存格式没有什么具体要求。当 Web 服务器提供 XML 文档的时候，其 MIME 类型可能为 application/xml 或者 text/xml。但在实际应用中，前者还是要优于后者的。

18.2.2　XML 文档的结构良好性验证

只有结构良好的 XML 文档才能够被 XML 解释器处理。为了能够得到结构良好的 XML 文档，需要遵守下面这样一些规则。

（1）所有开始标签必须要有对应的结束标签。

（2）元素可以嵌套，但是不能重叠。

（3）有且只能有一个根元素。

（4）属性值必须使用引号包含起来。

更多资源请访问稀酷客(www.ckook.com)

（5）一个元素不能有两个同样属性名字的属性。

（6）注释不能出现在标签内部。

（7）没有转义的“ < ”或者“ & ”不能出现在元素和属性的字符中。

上面讲到的这些规则在后面都会看到具体的实例。

由于一个良构的 XML 文档需要满足上面所讲的众多条件，所以检查一个 XML 文档是否为良构的，是一个相当复杂的任务，需要对 XML 文档检查每个条件是否满足。实际上，由于不良结构的 XML 文档就不能通过 XML 解析器的处理，所以可以采用一种简单的方法来判断 XML 文档是否为良构的。那就是在判断了其他的条件后，使用最简单的代码来处理 XML 文档，看是否可以通过 XML 解释器的处理过程。如果可以通过，则 XML 文档肯定是良构的。如果不能够通过，虽然不能判断 XML 一定不是良构的，但是可以根据输出的错误信息来查看是否是 XML 文档处理的问题。

【例 18-2】以 ElementTree 模块为例，直接使用其提供的 parse 方法。如果 parse 方法不报错，则表示此 XML 文档是良构的。一个简单的判断方式如下。

```
#filename: et_1.py
#encoding = utf-8

import xml.etree.ElementTree as ET

try:
    ET.parse("xml_1.xml")
    print u"这是一个良构的 XML 文档"
except Exception, e:
    print u"这可能是一个非良构文档"
    print u"出错信息：",e
```

- 在代码的开始处导入了 ElementTree 模块，这是一个处于 xml.etree 中的模块。并对其使用 ET 的别名。
- 接下来使用了一个 try-except 结构。处于 try 中的语句调用了 ET 的 parse 方法。该方法接受一个文件名或者文件对象。这里是在【例 18-1】中已经说明的 xml_ 1 文件。如果 XML 文档是良构的，会继续执行下面的语句，输出相关的信息。否则将转到 except 部分，输出对应的 XML 文档是否为良构的信息和其出错信息。

【例 18-2】代码运行的结果如下：

```
In [2]: run et_1.py
这是一个良构的 XML 文档
```

可以看到，在【例 18-1】中的 XML 文档（xml_ 1）是良构的。

当将【例 18-1】中的 xml 文档修改为不满足在上面描述的任何一条规则的时候，XML 将不满足良构条件。从而导致解析出错。

如将【例 18-1】XML 文档的最后一行中的“ </goods > ”改为“ < goods > ”，将会出现下面的结果：

```
In [3]: run et_1.py
这可能是一个非良构 XML 文档
出错信息：  no element found: line 15, column 7
```

- 从输出信息可以看出，这里的 XML 文档解析发生了错误。具体的错误信息也给出来了，主要是元素没有闭合。出错信息中还给出了具体的出错位置，这里给出的位置不一定总是正确的，仅仅可以作为参考。

对 XML 文档的最后两个标签进行交换，如下所示：

更多资源请访问稀酷客(www.ckook.com)

```
#省略部分文档
        <quantity>9</quantity>
        <color>white</color>
        <goods>
</shirt>
```

则其代码输出结果如下：

```
In [4]: run et_1.py
这可能是一个非良构 XML 文档
出错信息：  mismatched tag: line 15, column 2
```

如在 XML 文档的最后面再加上一个根元素“<goods></goods>”，则其代码的输出结果如下：

```
In [5]: run et_1.py
这可能是一个非良构 XML 文档
出错信息：  junk after document element: line 16, column 0
```

如将 XML 文档的第一行中变为如下的形式：

```
<? xml version=1.0 encoding='utf-8' standalone='no' ? >
```

则运行此段代码将会给出如下的信息：

```
In [6]: run et_1.py
这可能是一个非良构 XML 文档
出错信息：  XML declaration not well-formed: line 1, column 14
```

在上面给出了几种违反 XML 文档良构规则的情况，并显示了 ElementTree 模块对于这些情况的处理。可见对 XML 文档的结构要求还是很高的。还有一些其他的情况，这里虽然没有给出具体的处理方式，但是只要是不符合要求，在 ElementTree 模块中都是可以识别出来的。

【例 18-3】演示使用 lxml 模块，通过异常的类别来确定是否为非良构文档。其代码如下：

```
#filename: lxml_1.py
#encoding=utf-8

import lxml.etree as ET

try:
    ET.parse("xml_1.xml")
    print u"这是一个良构的 XML 文档"
except SyntaxError, e:
    print u"这是一个非良构 XML 文档"
    print u"出错信息：",e
```

- 代码中首先导入了 lxml 模块。可以看到，此段代码和【例 18-2】中的代码是很相似的。这是因为 lxml 设计的时候就尽量做到和 ElementTree 模块兼容。这里有一个不同的地方是，原来 except 部分中的 Exception 成了这里的 SyntaxError。从而通过此异常类型可以判断出测试的 XML 文档不是良构的。

当 XML 文档为良构的时候，运行此段代码和【例 18-2】没有区别。而当 XML 不是良构的时候，如上面 In［3］中的情形，将最后的结束标签改为开始标签。下面是其输出结果：

```
In [13]: run lxml_1.py
这是一个非良构 XML 文档
出错信息：  Premature end of data in tag goods line 15, line 16, column 1
```

可以看到也能够判断出是否为良构文档。同时给出了出错的信息。注意到这里输出的信息要比【例 18-2】中输出的信息要详细一些。

更多资源请访问稀酷客(www.ckook.com)

18.2.3　XML 文档的有效性验证

要解析 XML 文档，文档的结构良好性是必须满足的要求。而有效性则不一定。但是，由于 XML 的语法非常灵活，对标签元素又没有做特殊的规定，这使得 XML 文档的标签使用非常随意。从而引入了 XML 文档的有效性验证。只有满足了一定条件的 XML 文档才是有效的。

为了规范 XML 文档中的元素及其使用方式，XML 规范中通过使用 DTD 文件来对其进行描述。在 DTD 文件中说明了 XML 文档的元素、元素类型以及内容形式，并且为这些组成结构及其之间的关系定义了相应的规则。DTD 是 XML 版本 1.0 中的定义语言，但其本身并不是由 XML 文本组成的。为此，W3C 采用了 Microsoft 公司提出的 XML Schema 方案。这是一种使用 XML 文本来对 XML 文档进行定义的模式。

在 Python 语言的标准库中没有包含对 XML 文档的有效性验证功能。下面介绍的方法都是基于 lxml 模块的。

1. DTD 验证

在 lxml 模块中进行 DTD 验证的框架如下：

```
import lxml.etree as ET
dtd = ET.DTD('dtd_file.dtd')
f = file('xml_file.xml')
xml = ET.XML(f.read())
dtd.validate(xml)
```

- 首先从 lxml 中导入 etree 模块，这是后面各种类和方法的基础。
- 使用 etree 模块中的 DTD 类，按照给定的 DTD 构建了一个 DTD 对象，其中将会包含此 DTD 文件中的各种元素定义等。将其赋值给 dtd 变量。
- 使用 file 构建一个文件对象，并赋值给变量 f。
- 使用 etree 模块中的 XML 类构建了一个 XML 实例，其构造函数的参数为 XML 文档内容，并将此对象赋值给 xml 变量。
- 通过调用 dtd 对象中的 validate 方法，来验证此 XML 文件的有效性。

【例 18-4】演示 DTD 验证的使用方法。

```
In [2]: import lxml.etree as ET

In [3]: from StringIO import StringIO

In [4]: f = StringIO("<! ELEMENT empty EMPTY>")

In [5]: dtd = ET.DTD(f)

In [6]: xml = ET.XML("<empty />")
In [7]: dtd.validate(xml)
Out[7]: True

In [8]: xml = ET.XML("<empty>content</empty>")

In [9]: dtd.validate(xml)
Out[9]: False

In [10]: dtd.error_log.filter_from_errors()[0]
Out[10]: <string>:1:0:ERROR:VALID:DTD_NOT_EMPTY: Element empty was declared EMPTY this
one has content
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［2，3］中导入了后面代码运行所必需的模块。
- In［4］使用 StringIO 生成了一个文件对象。其字符串为一个 DTD 定义，这里的含义指明了 empty 元素的类型，这里其值必须为空。元素的类型包括 4 类：EMPTY（空元素）、ANY（任意）、Mixed（混合元素）和 Children（子元素）。通过此文件对象，在 In［5］中，通过使用 etree 中的 DTD 类构造了一个 DTD 类的对象。
- 在 In［6］中，使用了一个由 empty 空标签组成的元素。然后接着使用上面生成的 dtd 对象来对其进行验证。从输出结果来看，这里的 XML 文档是满足 DTD 定义的。
- 在接下来的两行中，使用了一个非空的 empty 元素，并对其调用 dtd 对象的 validate 方法进行验证。从输出结果可以看出，它不满足 DTD 定义，从而也不具有有效性。最后在 In［10］中，通过使用 error_log 得到了具体的错误信息。从错误信息中可以看出，empty 元素被定义成了空元素，而这里却有内容。

2. XML Schema 验证

XML Schema 模式给 XML 文档的书写带来了更大的灵活性，加速了 XML 技术在相关领域的发展。这种技术可以使 XML 文档的某一部分有效。XML Schema 技术的规范非常复杂，这里仅仅采用了最简单的方式来作为介绍。

XML Schema 的验证方式和 DTD 的验证方式是类似的，大部分几乎都是一样的，仅仅只是将构造 DTD 类对象改为构造 XML Schema 类对象。由于 XML Schema 类是用 XML 语言描述的，所以在构造的时候并不是输入文件对象，而是通过 parse 方法解析过的对象。

【例 18-5】演示 XML Schema 验证的使用方法。

```
#省略导入 lxml 和 StringIO 模块
In [4]: f = StringIO('''
   ...: <xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema" >
   ...: <xsd:element name = "a" type = "AType"/ >
   ...:    <xsd:complexType name = "AType" >
   ...:      <xsd:sequence >
   ...:        <xsd:element name = "b" type = "xsd:string" / >
   ...:      </xsd:sequence >
   ...:    </xsd:complexType >
   ...: </xsd:schema >
   ...: ''')

    In [5]: xmlschema_doc = ET.parse(f)
    In [6]: xmlschema = ET.XMLSchema(xmlschema_doc)

    In [7]: valid_str = StringIO('<a><b></b></a>')

    In [8]: xml1 = ET.parse(valid_str)

    In [9]: xmlschema.validate(xml1)
    Out[9]: True

    In [10]: invalid_str = StringIO('<a><c></c></a>')

    In [11]: xml2 = ET.parse(invalid_str)

    In [12]: xmlschema.validate(xml2)
    Out[12]: False

    In [13]: print xmlschema.error_log
     <string>:1:0:ERROR:SCHEMASV:SCHEMAV_ELEMENT_CONTENT: Element 'c': This element is
not expected. Expected is ( b ).
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［4］中，定义了一个 XML Schema 的字符串，并将其转化为文件对象，再将其赋值给变量 f。在这个 XML Schema 定义中，定义了各个元素的要求和相互之间的关系（这里并不对 XML Schema 的语法进行详细的介绍）。
- 由于 XML Schema 本身也是使用 XML 语言书写的，所以首先需要通过 parse 方法将其转换，以便于后面的处理。
- In［6］中构造了一个 XMLSchema 类对象，其输入参数为 In［1］的 XML Schema 定义调用 parse 函数后的值。
- In［7，8，9］和 In［10，11，12］中分别构造了一个合法的 XML 文件对象和非法的 XML 文件对象。接着是解析成一个 xml 对象。然后调用 XMLSchema 对象的 validate 方法，从而得到相应的结果。同样地，可以通过 XMLSchema 对象的 error_log 来查看错误信息。

通常，在使用 XML 文档的时候，都需要对 XML 的良构性和有效性进行验证后才可以继续使用。可以利用上面两小节中介绍的方法来实现这些要求。

18.3　XML 文档的结构

在本节中，将对 XML 文档的大部分构成元素进行介绍。XML 文档是由文本标签构成的，可能看起来和 HTML 中的标签一样，但是实际上还是有区别的。在 HTML 中，只能使用已经定义好的标签，而在 XML 文档中可以根据需要创建标签。除了元素以外，还有很多其他的文档结构。在本节中，将使用 ElementTree 模块来对 XML 文档的结构进行说明。

18.3.1　XML 的元素和标签

XML 文档是由 XML 元素构成的，元素是通过标签来限定的。而在开始标签和结束标签之间的任何字符都是此元素的内容，包括空白字符。

1. 元素中的标签

在【例 18-1】中，< color > black </color > 就是此 XML 文档中的一个元素。这里，< color > 和 </color > 是此元素的标签，而 black 是此元素的内容。可以看到，XML 的标签和 HTML 的标签是很相似的。开始标签以“ < ”开始，结束标签以“ </ ”开始，后面是此元素的名字和结束符“ > ”。这里，元素的名字一般反映着所包含元素的内容。在 ElementTree 模块中，Element 类可以用来构建标签：

```
In [1]: import xml.etree.ElementTree as ET

In [2]: root = ET.Element("color")

In [3]: root.text = " black "

In [4]: tree = ET.ElementTree(root)

In [5]: import sys

In [6]: tree.write(sys.stdout)
<color > black </color >
```

- 在 In［2］中使用了 Element 类，其构造函数的输入参数为标签的名字。
- In［3］中设置了此元素的内容，使用的是 text 属性。
- 在 In［4］中使用模块中的 ElementTree 方法构建了一个树对象。其构造函数的输入参数为根结点对象，并赋值给 tree 对象。

更多资源请访问稀酷客(www.ckook.com)

- 最后调用 tree 对象的 write 方法，将构建的 XML 文档输出。这里，将输出的信息发给了标准输出，也就是终端。当然，也可以将输出的信息写入特定的文件。

没有任何内容的元素被称为空元素。对于 XML 的空元素有一个特殊的语法：可以直接在元素名字前后加上“ < ”和“ / > ”构成一个空元素。在 XHTML 中，换行符是“ < br / > ”，而不是 HTML 规范中的“ < br > ”。这其实和“ < br > < /br > ”是一样的。这两种方式使用哪一种都是可以的，这取决于 XML 文档编写者。但是，在 XML 中不能写成“ < br > ”，因为这是非良构的元素，在 XML 解析的时候将会发生错误：

```
In [8]: ET.ElementTree(ET.Element("br")).write(sys.stdout)
<br />
```

也就是说，对于没有内容的元素，就可以表示成空元素的形式。

另外需要注意的是，在 XML 文档中是区分大小写的。也就是说， < color > 和 < COLOR > 是不同的。如果是使用 < color > 开始标签的话，则不能使用 < /COLOR > 标签将其闭合。使用大写还是小写并不重要，但是一定要保持前后的一致性。如下所示：

```
In [9]: color = ET.Element("color")

In [10]: COLOR = ET.Element("COLOR")

In [11]: color.tag == COLOR.tag
Out[11]: False

In [12]: color_clone = ET.Element("color")

In [13]: color.tag == color_clone.tag
Out[13]: True
```

- 从这里的输出结果可以看出，元素是区别大小写的。这里的 color 和 COLOR 两个元素的标签不相同。而 color 和 color_clone 的则相同。

2. XML 树结构

在【例 18-1】中，描述了一个货物（goods）信息。可以看到，在 goods 元素中包含 shirt 元素。而在 shirt 元素中，还有如 size、color 等其他元素。在这里，称 shirt 为 goods 的子元素，同时又为 size 等的父元素。而如 size 和 color 这样的同级元素则互称为兄弟。在这里，name、size、quantity 和 color 均为兄弟元素。

在 XML 文档中，所有的元素最多都只有一个父元素。

另外，如果一个元素的开始标签在另一个元素内，则其结束标签也一定要在这个元素内。所以，重叠的标签在 XML 规范中是不被支持的。如“ < size > < color > Only for test < /size > < /color > ”就不被 XML 支持。因为这里 color 元素的开始标签在 size 元素内部，而其结束标签在 size 元素外部。

每个 XML 文档都有一个没有父元素的元素，并且所有其他元素都是包含在此元素之中。这个元素被称为根元素。一个结构良好的 XML 文档都含有这样一个元素。由于 XML 有根元素，并且其中的元素范围都是不会重叠的，所以 XML 文档可以形成一个树结构，如图 18-2 所示。

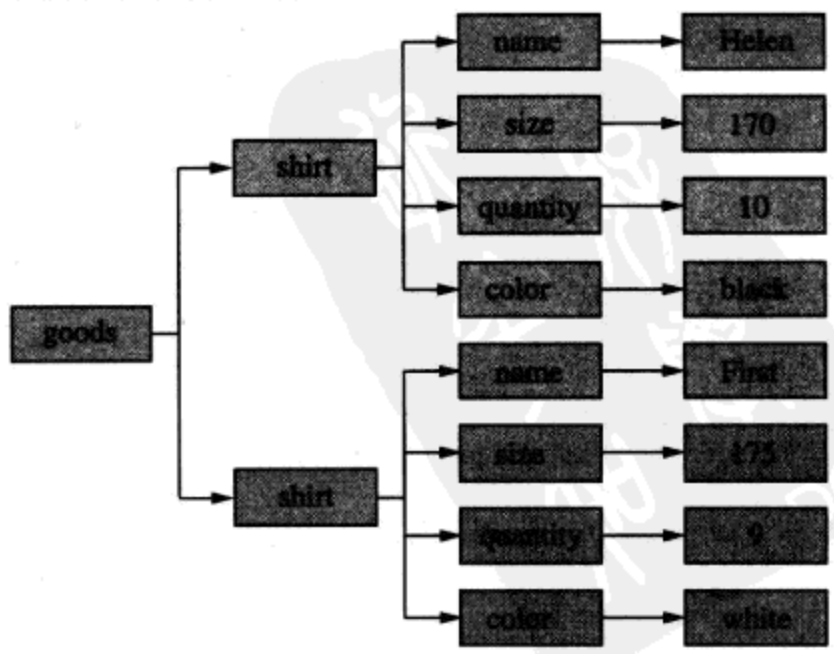


图 18-2　XML 树结构

在图 18-2 中，浅色的标记为元素标签，

更多资源请访问稀酷客(www.ckook.com)

而深色的标记为元素内容。在这里，所有的元素内容都只是字符串，并没有包含任何标签。实际上，在 shirt 和 name 之间还是可以输入内容的，这样就构成了元素的混合内容。另外，在 shirt 和 name 元素之间是有空白字符等内容的，只是这些内容会被大多数 XML 处理程序忽略。包含混合内容的 XML 文档具有实际的用途，可以用在各个领域。

【例 18-6】演示如何构建子元素。

```
In [12]: root = ET.Element("goods")

In [13]: shirt = ET.SubElement(root,"shirt")

In [15]: name = ET.SubElement(shirt,"name")

In [16]: size = ET.SubElement(shirt,"size")

In [17]: tree = ET.ElementTree(root)

In [18]: tree.write(sys.stdout)
<goods><shirt><name /><size /></shirt></goods>
#XML 树结构
<goods>
<shirt>
    <name />
    <size />
</shirt>
  </goods>
```

- 在 In［12］中用 Element 类创建了一个 root 对象。
- 在 In［13］中用 SubElement 类构建了一个 shirt 对象，其中，shirt 对象为 root 对象的子元素，而 root 对象为 shirt 对象的父元素。
- 在 In［15，16］中用 SubElement 对象构建了 shirt 的另外两个子对象。这两个子对象之间的关系是兄弟节点关系。
- 在 In［18］中将构建的 XML 树进行了输出。在下面还将此 XML 树的样式形象的表示了出来。当然实际中的输出还是上面的那一行数据。
- 这里的 < goods > 节点为根元素节点。

这里并没有为各个节点添加内容，通过下面的方法可以使得个元素节点具有相应的内容：

```
In [19]: name.text = "Helen"

In [20]: size.text = "170"

In [21]: ET.dump(tree)
<goods><shirt><name>Helen</name><size>170</size></shirt></goods>
```

- 在 In［19，20］中分别为 name 和 size 节点增加了内容。
- 在 In［21］中，调用 ElementTree 的 dump 方法将此 xml 树结构打印出来。这里的输出使用的是最普通的 XML 格式。

18.3.2　元素的属性

XML 文档中的元素可以拥有属性。属性是放在元素开始标签中的名字和值的对。名字和值通过等号分开，中间的空格是可选的。值需要使用单引号或者双引号括起来。例如，在原来的 shirt 中可以加上质量的属性，下面的两种方法都可以实现：

更多资源请访问稀酷客(www.ckook.com)

```
<goods>
    <shirt quality="A">
        <name>Helen</name>
        <size>170</size>
        <quantity>10</quantity>
        <color>black</color>
    </shirt>
    <shirt quality = 'B'>
        <name>First</name>
        <size>175</size>
        <quantity>9</quantity>
        <color>white</color>
    </shirt>
</goods>
```

在第 1 个 shirt 元素中，使用了双引号且没有空格的形式，在第 2 个 shirt 元素中，使用了单引号加上空格的形式。采用哪种形式书写取决于 XML 文档的编写者。另外，如果有多个属性，属性之间的顺序是没有影响的。

在 ElementTree 模块中，元素的属性实际上是作为字典数据结构来保存的。可以通过 attrib 属性来访问。下面将会生成一个具有属性的元素：

```
In [22]: shirt = ET.Element("shirt")

In [23]: shirt.attrib["quality"] = "A"

In [24]: ET.dump(shirt)
<shirt quality="A" />
```

- 在 In［22］中构建了一个 shirt 的元素。然后直接使用 attrib 属性生成了此元素的一个属性 quality，并赋值为“A”。
- 在 In［24］中输出了此节点的相关信息，可以看到，shirt 元素已经具有了 quality 属性，并且其值为 A。

当然，也可以在元素构建的时候将属性放在构造函数中：

```
In [25]: shirt = ET.Element("shirt", quality="A")

In [26]: ET.dump(shirt)
<shirt quality="A" />
```

- 在 In［25］中将 quality 的属性和值放在了 shirt 元素的构造函数中。

如果将上面的子元素的数据放在属性中，就成为了下面的形式：

```
<goods>
    <shirt quality="A" size="170">
        <name>Helen</name>
        <quantity>10</quantity>
        <color>black</color>
    </shirt>
    <shirt quality = 'B' size='175'>
        <name>First</name>
        <quantity>9</quantity>
        <color>white</color>
    </shirt>
</goods>
```

在上面的这个例子中，将 size 子元素的值作为属性放入到了 shirt 元素中。当然还可以将其他的子元素作为属性放在 shirt 元素中。

因为在属性中也可以保存数据，这里就有一个是将数据保存在属性中还是保存在子元素中的

更多资源请访问稀酷客(www.ckook.com)

问题。这个问题并没有一个严格的规则可以遵循，而且现在仍然是一个热门的讨论话题。一般说来，倾向于将元信息放在属性中而将信息本身放在子元素中。但是，很多时候很难将两者区别开来。

由于子元素和属性的性质不同，所以还是有一些使用时可以参考的地方。在有 DTD 的情况下，子元素是严格有序的，而属性则是无序的。所以在需要保存无序数据的情况下，还是尽量使用属性。而当数据可能需要以后进行扩展的时候，则最好使用子元素结构，因为属性不具有可扩展性。而当需要保存一两个不需要扩展的非结构化信息的时候，使用属性要方便一些，也更直观一些。总的说来，属性主要是用来保存元素的元信息，相对于子元素来说不便于处理，对于比较简单、比较固定的信息，使用属性较为合适。

18.3.3　XML 的文字

在 XML 文档中，有元素名字、属性名字等，这些都被称为 XML 名字。XML 的文档规范非常严格，在 XML 名字空间的规范上也是统一的。可以用作 XML 名字的字符包括字母表中的字母、数字以及其他 Unicode 字符，可以使用的标点符号包括下划线、连接符和句号。但不能使用引号、$ 号、% 号和分号等这些标点符号。在 XML 的名字中不能有任何的空白字符，包括空白符、换行符等。另外，以 XML 开头的 XML 名字（不区分大小写）是被 W3C 的 XML 规范保留的。

XML 名字只能以字母、下画线以及 Unicode 字符开始，以其他可选字符开始的 XML 名字是不合法的。这些 XML 名字的字符使用限制如表 18-1 所示。

表 18-1　XML 名字使用的限制

字　　符	是否可用	是否可以作为开始字符
字母	可以，包括 A ~ Z 和 a ~ z	可以
数字	可以，包括 0 ~ 9	不可以
Unicode 字符（非标点）	可以	可以
_	可以	可以
-	可以	不可以
.	可以	不可以
‘	不可以	不可以
“	不可以	不可以
$	不可以	不可以
%	不可以	不可以
:	不可以（可以用在 XML 名字空间中）	不可以
XML（不区分大小写）	可以	不可以（W3C 保留）

对于使用 Unicode 字符的元素，可以在原来的文件中加入一个元素，其组成如下：

```
<牛仔裤 质量 = '一等品'>
    <品牌 > 西部传奇 < /品牌 >
    <大小 >175 < /大小 >
    <数量 >20 < /数量 >
    <颜色 > 棕蓝色 < /颜色 >
< /牛仔裤 >
```

将上面的文件保存为 xml_2. xml 文件，在 IE 浏览器中将其打开，如图 18-3 所示。

更多资源请访问稀酷客(www.ckook.com)

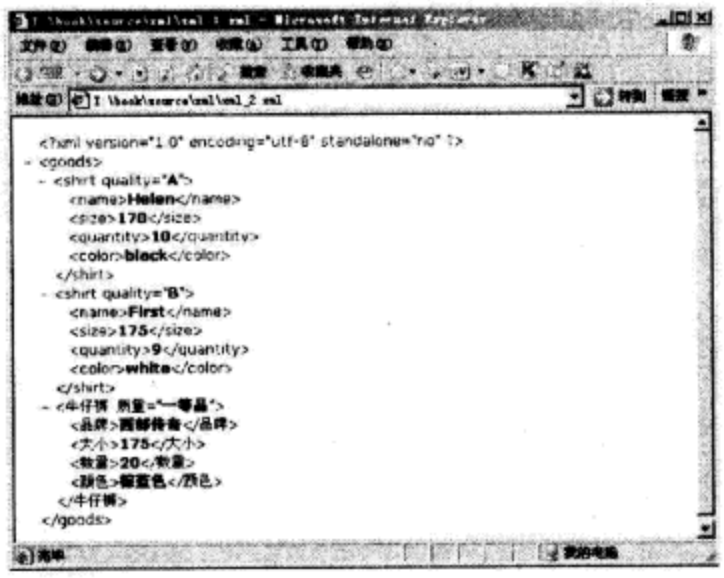


图 18-3　含有 Unicode 字符的 XML 文件显示

一些 XML 名字的合法性如表 18-2 所示。

表 18-2　部分 XML 名字的合法性

XML 名字	合　法　性
color	合法
day-of-year	合法
_10	合法
téléphone	合法
my. name	合法
-10	不合法，不能以-开始
my name	不合法，不能有空格
day/year	不合法，不能使用/
color' type	不合法，不能使用'
XML_Type	合法，但不推荐使用

表 18-2 中给出的 XML 名字的合法性规则只会在解析的时候发生错误。在 XML 文档的构造过程中并不会有问题。

【例 18-7】 演示在 ElementTree 模块中，非法的 XML 名字的元素对象是可以构建的，但是在解析的时候将会发生错误。

```
In [27]: invalid = ET.Element("% my name")

In [28]: invalid.text = "content"

In [29]: ET.dump(invalid)
<% my name> content </% my name>

In [30]: from StringIO import *

In [31]: ET.parse(StringIO(ET.dump(invalid)))
<% my name>content </% my name>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ExpatError                                Traceback (most recent call last)
#省略部分输出结果
-> 1254          self._parser.Parse ("", 1) # end of data
   1255          tree = self._target.close ()
   1256          del self._target, self._parser # get rid of circular references

ExpatError: syntax error: line 1, column 0
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［27］中可以看出，使用了“% my name”的 XML 名字构建元素。并将其内容设置为“content”。从 In［29］的输出结果可以看出，在构建的时候 ElementTree 模块并没有对输入数据进行 XML 名字的合法性检查。
- 在 In［32］中，对得到的数据调用 parse 方法，这时可以看到，将报错“syntax error”，表示此 XML 名字是不合法的。

18.3.4　字符实体

在元素内容中有“<”字符时不能直接输入，因为“<”总是被解释为开始标签的开始。这时，可以通过输入“&dt;”来完成。使用字符实体可以在 XML 文档中输入特定的字符。主要在下面这些情况下应该使用字符实体。

（1）输入的字符可能会被解释为其他意思。

（2）因输入设备的限制而不能输入的某些字符，如版权符号。

所有字符实体都是以“&”开始和以“;”作为结束。对于 Unicode 字符，有十进制和十六进制这两种表示方法。其中，十进制为“&#value;”，十六进制为“&#xvalue;”。例如，要输入 Euro 符号，则可以使用“€”或者“€”。

在 XML 规范中已经预先定义了 5 个字符实体，如表 18-3 所示。

表 18-3　XML 规范中预定义的字符实体

字 符 实 体	符　　号
&dt;	<
>	>
&	&
'	‘
"	“

在表 18-3 中，“&dt;”和“&”必须在元素内容中用来代替原义。“"”和“'”常用在属性中。

【例 18-8】演示在 ElementTree 模块中直接解析输入的字符实体。如字符实体非法，在解析的时候报错。

```
In [32]: entity_ref = ET.XML("<a>&lt;</a>")

In [33]: ET.dump(entity_ref)
<a>&lt;</a>

In [34]: entity_ref.text
Out[34]: '<'

In [35]: entity_ref = ET.XML("<a>&entity;</a>")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ExpatError                                Traceback (most recent call last)
#省略部分输出结果
   1244     def feed(self, data):
-> 1245         self._parser.Parse(data, 0)
   1246
   1247     ##

ExpatError: undefined entity: line 1, column 3
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［32］中使用 XML 类生成了一个指定 XML 字符串的对象，然后使用 dump 方法输出了此元素。由于 dump 方法输出的是最原始的 XML 文档，所以这里的输出结果和输入的字符串是相同的。而在 In［34］的输出结果中，其字符经过了转义，成了“<”。
- 在 In［35］中不存在“&entity;”这样的字符实体，所以字解析的时候就出现了错误：“undefined entity”。可以看到这里的字符实体是不存在的。

18.3.5　CDATA 段

在 XML 中，CDATA 段用来声明不应被解析为标签的文本。在其中可以使用特殊字符，包括小于号、大于号和双引号等，而不必使用其字符实体。

CDATA 段通过使用“<![CDATA［”和“]]>”来设置。所有在这中间的内容都会被当作原始字符，而不会进行字符转义或者是当作标签。这在输入源码的时候特别有用。

如需要保存下面的一段 Python 代码数据，其 XML 文件可能会写成这样（文件保存为 xml_3.xml）：

```
<? xml version = '1.0' encoding = 'utf-8' standalone = 'no' ? >
<source >
    <function >
        <min >
        def min(x, y):
            if x < y: return x
            else: return y
        </min >
    </function >
</source >
```

此文件是不能通过 XML 解析的，因为在语句“if x < y”中，没有对“<”使用字符实体，从而导致将“<”解析为开始标签，进一步导致解析出错。在 IE 中显示出错如图 18-4 所示。

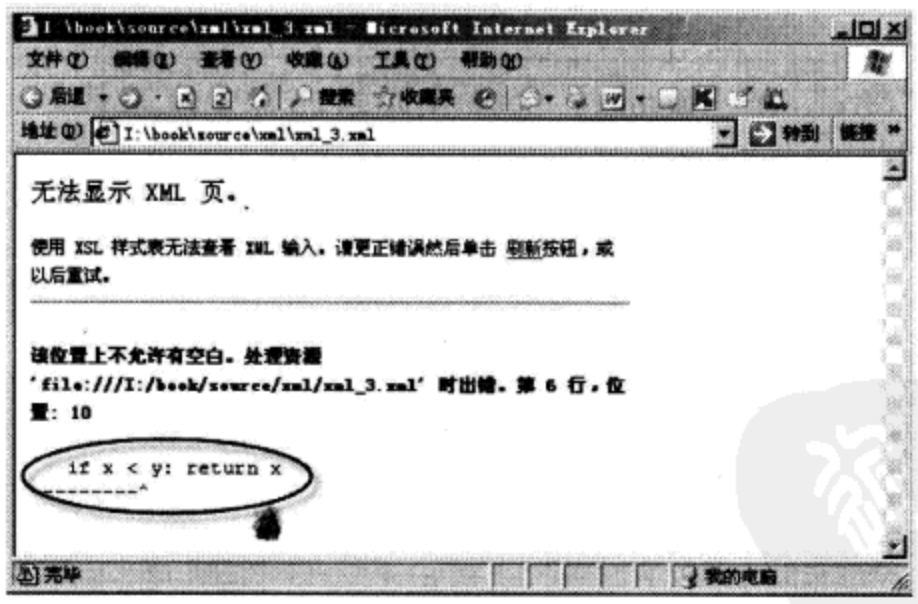


图 18-4　含有“<”的出错 XML 文档

直接使用 ElementTree 模块的 parse 方法也可以确认这个 XML 文档不是良构的，从而在解析的时候会出错：

```
In [36]: ET.parse("xml_3.xml")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ExpatError                                Traceback (most recent call last)

I:\book\source\xml\<ipython console> in <module>()
```

更多资源请访问稀酷客(www.ckook.com)

```
#省略部分输出结果
  1244      def feed(self, data):
-> 1245          self._parser.Parse(data, 0)
  1246
  1247      ##

ExpatError: not well-formed (invalid token): line 6, column 9
```

可以使用字符实体或者 CDATA 段来解决这个问题。

【例 18-9】演示使用字符实体的方法。

```
<? xml version = '1.0' encoding = 'utf-8' standalone = 'no' ? >
<source >
    <function >
        <min >
        def min(x, y):
            if x &lt; y: return x
            else: return y
        </min >
    </function >
</source >
```

【例 18-10】演示使用 CDATA 段的方法。

```
<? xml version = '1.0' encoding = 'utf-8' standalone = 'no' ? >
<source >
    <function >
        <min >
        <! [CDATA[
        def min(x, y):
            if x < y: return x
            else: return y
        ]] >
        </min >
    </function >
</source >
```

【例 18-9】和【例 18-10】在 IE 中的显示分别如图 18-5 和图 18-6 所示。

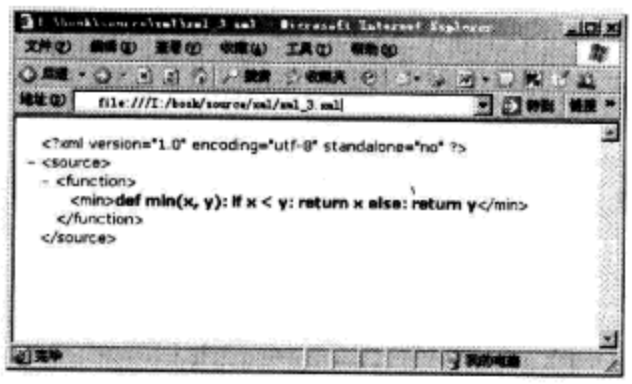


图 18-5　使用字符实体的 XML 文档

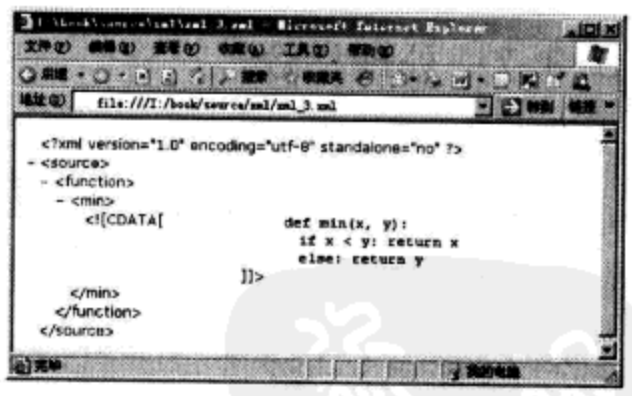


图 18-6　使用 CDATA 段的 XML 文档

在图 18-5 和图 18-6 中可以看到，IE 对元素内容中的空白字符进行了压缩。但是当这些空白字符是处于 CDATA 段中的时候，则会全部保留。

另外需要注意的是，唯一不能在 CDATA 段中出现的是其结束界定符“]] >”。CDATA 段主要是方便用户阅读，而不是针对 XML 处理程序的。XML 解析器不需要知道数据是从CDATA段还是从字符实体中来。也就是说，在访问这些数据的时候，其差别将会忽略。

上面介绍的是在 XML 文档中存储保留字符的方法。要注意这种内部存储和外部表示之间的关系。在外部显示的时候可以对这些相关的字符进行处理。而在内部存储的时候

更多资源请访问稀酷客(www.ckook.com)

则是必须的。

18.3.6　注释

在 XML 文档中，也可以对文档结构和元素内容作出解释。这些注释不属于 XML 文档的内容，XML 的解释器也不会去解析这些数据。XML 的注释为“<！—”和“-->”之间的部分。

如下面的 XML 注释：

```
<！—这些货物信息还需要进一步核实-->
```

因为注释不是文档内容的一部分，所以可以出现在文档的任何地方。也可以出现在根元素的前面或后面。不过，注释不能出现在标签中和嵌套注释。

一般的 XML 解释器都不会解析这些注释，而只是简单的丢弃，所以不要写依赖于注释的 XML 文件。注释仅仅是为了增加可读性而使用，而不是为了方便计算机处理。

【例 18-11】演示 ElementTree 模块在处理 XML 文档的时候会将注释去掉。

```
In [37]: xml_str = """<test>
   ....: <! -- Comment -- >
   ....: content
   ....: </test>
   ....: """

In [38]: tree = ET.XML(xml_str)

In [39]: ET.dump(tree)
<test>
content
</test>
```

- 从 In［37］和 In［39］的结果对比来看，两者的区别在于在后面的输出结果中省略掉了注释信息。注释信息只是在 XML 文档中具有解释文档的作用。而对于 XML 文档的解析不是必须的。

18.3.7　处理指令

XML 文档通过处理指令（Processing Instruction，PI）可以包含用于特定应用的指令。一个处理指令通过“<?”和“? >”来界定。紧接在“<?”后面的是处理指令名，接着是属性值对。如在 XHTML 文档中可能会有下面的语句：

```
<? robots index = "yes" follow = "no"? >
```

这里的 robots 元信息是指示搜索引擎是否检索以及如何检索一个页面的指示。这里的处理指令名字是 robots。这条特定的处理指令有两个属性，分别为 index 和 follow，其值分别为 yes 和 no。从名字可以看出，index 的值指示搜索引擎是否可以检索一个页面，而 follow 的值则指示是否可以跟踪此文档的链接。通过这样的处理指令，XML 文档用来为应用程序提供一些操作特性，从而使得 XML 文档不仅仅只是一个保存数据的载体。

从形式上来说，<? xml version = “1.0” encoding = “ASCII” standalone = “yes”? > 也是一个处理指令，其作用是对 XML 文档进行标识。这在绝大部分 XML 文档中都是常见的。

另外，在 XML 文档中还常可以看到下面的处理指令：

```
<? xml-stylesheet type = "text/xsl" href = "xsl1.xsl"? >
```

更多资源请访问稀酷客(www.ckook.com)

上面的处理指令是使用指定的 XSL 文件对 XML 文档进行格式转换。

需要注意的是，使用 XML（不区分大小写）开始的处理指令名是为 XML 规范所保留的。具体来说，处理指令名字的取值和 XML 名字的取值是一致的。

18.3.8　XML 定义

XML 文档是使用文本字符串来保存数据的，而且可以为每个 XML 文档指定所使用的编码。默认情况下，XML 使用 UTF-8 编码。这是 ASCII 编码的超集，因此一个纯的 ASCII 文本文件也是一个 UTF-8 文件。尽管如此，大部分的 XML 解释器还是可以解析许多种字符编码的。在 XML 的定义中可以通过 encoding 属性来指定字符编码。如有些 XML 文档使用的是 ISO-8859-1 字符编码，从而可以让许多非英语的欧洲国家的字符显示出来。

一个 XML 最好有一个 XML 定义，这样可以检查 XML 文档是否为有效的，虽然这并不是必须的。下面是一个包含 XML 定义的 XML 文档：

```
<? xml version = "1.0" encoding = "ASCII" standalone = "yes"? >
<name >
   Alice
</name >
```

XML 文档不一定需要有 XML 定义，但是如果有 XML 定义，必须将其放在文档的第一行。在此定义之前不应该有任何的注释、定义或者空格等。这是因为很多的 XML 解释器将会使用“<? xml”这 5 个字符来对 XML 文档进行一些猜测，包括文件编码等。

encoding 在 XML 定义中是可选的，如果省略的话，则默认采用 Unicode 字符集。如果元信息和编码定义有冲突，会优先采用元信息。

另外，standalone 属性用来定义外部定义的 DTD 的存在性。取值可为“yes”和“no”，默认情况下为“no”。当为“no”的时候表示此 XML 文档不是独立的，而是取决于外部定义的 DTD。而“yes”值则表示此 XML 文档是自包含的。这也是一个可选的属性。

关于 XML 定义的详细说明可以参看相关的文档。可以使用文档定义来决定一个 XML 文档的有效性，具体的操作方法请参见 18.2.3 节。

18.4　使用 SAX 处理 XML 文档

SAX 是一种流行的解析和处理 XML 文档的接口。在本节中将首先介绍 SAX 的使用背景以及此接口的主要组成部分。然后介绍 Python 标准库中支持 SAX 接口的模块 xml.sax，并演示在实际应用中的实现方法。

18.4.1　SAX 介绍

SAX 是一种基于事件的 XML 文档解析方式。这项技术最早是由 XML 的 JAVA 语言开发者推动的，到现在已经成为了一种通用的 XML 文档处理接口。SAX 通过数据流的方式来读取文档。当读取部分内容后，就会对文档进行解析，并产生相应的事件。程序编写者可以针对特定的事件来书写自己的逻辑代码。通过这种序列文档的方法处理文档，可以使得这项技术对于内存的要求较小。当前的最新接口标准是 SAX2，其中包含了对于 XML 名字空间的支持。

作为 XML 文档处理的轻量级接口集合，在 SAX 中定义了多种用来处理 XML 文档的接口。SAX2 在 Python 标准库中的实现是 xml.sax 模块。此模块中实现了 4 种类型的接口，如表 18-4 所示。

更多资源请访问稀酷客(www.ckook.com)

表 18-4　xml. sax 模块中支持的 SAX2 接口

接　口	说　明
ContentHandler	内容处理接口
DTDHandler	DTD 定义处理接口
EntityResolver	实体解析器处理接口
ErrorHandler	错误信息处理接口

在实际应用中，SAX2 通过各种接口来实现对 XML 文档的处理。这种基于事件的处理方式可以使得应用程序在解析到某个特定的内容的时候，做出实时的反应。而不像 DOM 技术那样需要对整个文档进行解析。这种处理方式的过程是首先对 XML 文档进行分块，形成一段一段的数据流。然后按照流数据的方式对每个 XML 数据块进行解析。当解析到有特定的内容包括元素和 DTD 定义等的时候，将调用已经注册的各种内容处理器。当内容处理器处理结束后，应用程序将根据情况继续处理后面的流数据或者抛出异常而终止。具体的 SAX 处理流程如图 18-7 所示。

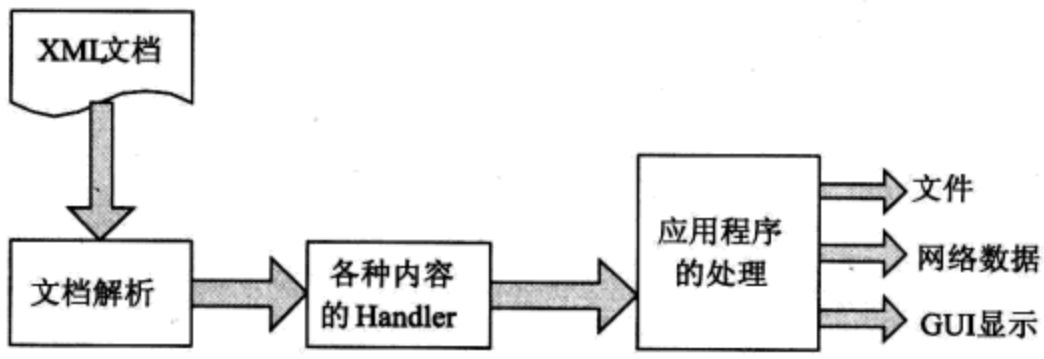


图 18-7　SAX 的处理方式

在 SAX 技术中，将 XML 文档的解析对象称为 reader。在 xml. sax 模块中，提供了xmlreader子模块用来为处理器提供数据。该对象将从源文件中读取内容，并为注册好的处理器提供相关的数据。源文件不一定是从 XML 文档中读取，只需要满足解析对象一定的要求就可以了。但是实际上，一般都是从 XML 文档中读取的。应用程序将通过处理器来处理 xmlreader 模块读取的数据。

18.4.2　SAX 处理的组成部分

xml. sax 是由 4 种主要的处理器接口组成的。每种接口都会在特定的时候被解析器触发，如 ContentHandler 将会在系统读取文档特定内容的时候触发等。应用程序只需要实现自身感兴趣的 handler 就可以了。这些处理器可以在一个对象中实现，也可以分布在多个对象中。另外需要注意的是，这些对象的实现都需要继承自原始的 xml. sax. handler 模块中的类，使得每个 XML 相关对象有默认的处理方式。下面将具体的介绍这 4 类处理器和 XMLReader 对象的使用方法。

1. ContentHandler 接口

ContentHandler 接口是在使用 SAX 技术的时候用得最多的处理器对象。此类中也提供了丰富的方法，用来处理 XML 文档数据。并且有不同的事件类型供在不同的时候调用。在ContentHandler基类中主要包含的常用方法如表 18-5 所示。

表 18-5　ContentHandler 中包含的常用方法

方　法	说　明
startDocument	文档处理开始的时候调用
endDocument	文档处理结束的时候调用
startElement	元素开始的时候调用

更多资源请访问稀酷客(www.ckook.com)

（续）

方　法	说　明
endElement	元素结束的时候调用
startElementNS	含有名字空间的元素开始的时候调用
endElementNS	含有名字空间的元素结束的时候调用
characters	字符数据处理的时候调用
ignorableWhitespace	可以忽略空白字符处理的时候调用
processingInstruction	收到处理指令的时候调用
skippedEntity	收到可以忽略的实体参考的时候调用

通过在表 18-5 中介绍的多种方法，使得 SAX 技术在处理 XML 文档的时候能够做到快速和完整。在这里，startElement 方法是 XML 应用程序编写者最感兴趣的一个方法。因为在数据解析中遇到的每个元素，这个方法所注册的函数都会被调用。此方法有两个参数，分别是所遇到元素的名字和属性值。通过判断名字和属性等信息，应用程序可以实现自己的业务逻辑，从而实现对 XML 文档的处理。

【例 18-12】演示 startElement 方法的使用。

```
#filename: sax_1.py
#encoding = utf-8

from xml.sax import *

class MyHandler(ContentHandler):
    def startDocument(self) :#开始处理文档的时候调用
        print "开始处理 XML 文档"
        print " ==============="
        print "name \tquality"
        print " - - - - - - - - - - - - - - - -"

    def endDocument(self) :#结束处理文档的时候调用
    print " ==============="
        print "结束处理 XML 文档"

    def startElement( self, name, attrs) :#处理元素
        if name == "shirt":#打印出所有 shirt 标签的相关属性
          print "% s \t% s"% (attrs['name'], attrs['quality'])

parser = make_parser() #生成解析对象
parser.setContentHandler(MyHandler())

#需要分析的 XML 数据
data = """ <goods >
            <shirt name = "Helen" quality = "A" />
            <shirt name = "Fayer" quality = "A +" />
            <coat name = "Dayie" quality = "B +" />
            <shirt name = "CaC" quality = "A-" />
            </goods >
"""
import StringIO
parser.parse(StringIO.StringIO(data))
```

- 在代码片段的最前面导入了 xml.sax 模块中所需的方法和对象。这在每个需要使用 SAX 技术处理文档的时候是必须的。
- 接着，代码片段定义了一个 MyHandler 类，并且继承自 ContentHandler 类，从而使得没

更多资源请访问稀酷客(www.ckook.com)

有重载的相关方法有了一个默认的处理方式。

- 在 MyHandler 类中实现（重载）了 3 个方法，分别是 startDocument、endDocument 和 startElement。在表 18-5 中的其他方法也可以根据需要重载并书写相关的业务逻辑。在 startDocument 和 endDocument 方法中，主要是输出一些信息，用来对最后的结果进行排版处理。需要注意的是，这两个方法都只会调用一次。并且 startDocument 方法将会在本接口中的任何方法调用之前执行，而 endDocument 方法则会在所有的方法结束之后被调用。
- startElement 方法中有两个参数，一个是 name，表示解析的 XML 名字。另外一个是 attrs，表示解析 XML 对象的属性。在函数体中，通过判断 XML 名字是否为“shirt”，从而将处理 XML 元素的范围限制在一定的范围之内。当元素名为 shirt 的时候，则函数主体将打印此元素属性中的 name 和 quality 的值。
- 通过调用 xml. sax 模块下的 make_parser 方法，生成了一个 XML 的解析对象。接下来的语句设置了此解析对象的 ContentHandler 处理器。从而将自定义的处理逻辑和 XML 文档处理联系了起来。
- 最后，调用了 parse 方法对 XML 数据进行分析。通过 xml. sax 模块，在解析遇到元素的时候，将会调用已经设置好的处理函数。

【例 18-12】的运行结果如下所示：

```
In [2]: run sax_1.py
开始处理 XML 文档
==============
name    quality
--------------------
Helen   A
Fayer   A+
CaC     A-
==============
结束处理 XML 文档
```

- 从上面的输出中可以看出，对于此 XML 数据分析结果的正确性得到了验证。在前面出现的 name 中，仅仅只有 shirt 元素的属性成员。

通过【例 18-12】可以看到，使用 SAX 处理 XML 数据还是比较方便的。只需要继承自指定的类，并且设置好相关的处理逻辑，即可实现对于 XML 数据的查询等操作。但是也可以看到，由于 SAX 采用的是事件触发的方式，所以当某个特定的事件触发的时候，传递给注册函数的数据仅仅只是这个事件所获取的值。这种方式对于简单的查询处理是适用的。但是却难于实现对其他位置元素的访问等类似功能。如果要在 SAX 中完成这样的操作，则需要自己保存相关的数据状态。所以，在处理 XML 文档的时候，选择合适的 ContentHandler 类是很重要的。

2. DTDHandler 接口

如果需要处理有关 DTD 定义的内容，可以使用 DTDHandler 对象，但是实际上，此类中仅仅只是提供了 notationDecl 和 unparseEntityDecl 这两个方法。其中，前一个方法主要用来处理 DTD 定义中的符号定义。而后一个方法则是用来对实体定义进行解析，这里仅仅只是会处理不能被解析的实体定义。解析对象可以通过 setDTDHandler 方法来加入处理函数。一个处理 DTD 定义的框架如下所示：

```
class MyHandler(DTDHandler):
    pass

parser = make_parser()
parser.setDTDHandler(MyHandler())
```

更多资源请访问稀酷客(www.ckook.com)

在这个代码框架中，首先生成了一个继承自 DTDHandler 的类，然后调用解析对象的setDTDHandler方法设置此 DTD 处理对象。通过这样的步骤，在 XML 数据解析的时候，遇到特定的 DTD 内容就会调用特定的处理函数了。

3. EntityResolver 接口

EntityResolver 接口是为实体参考解析所设置的接口。当解析器遇到外部实体参考的时候，就会调用此对象中的 resolveEntity 方法来处理。通过重载这个方法的实现，应用程序可以实现对外部实体的解析。在默认情况下会返回系统标识符。一个处理外部实体参考的框架如下所示：

```
class MyHandler(EntityResolver):
    pass

parser = make_parser()
parser.setEntityResolver(MyHandler())
```

在上面的代码中，首先生成了一个 MyHandler 类，此类继承自 EntityResolver，然后调用parser解析对象的 setEntityResolver 方法设置此外部实体参考处理对象。通过这样的步骤，在 XML 数据解析的时候，遇到特定的外部实体参考内容就会调用特定的处理函数了。

从上面的介绍中可以看到，DTDHandler 和 EntityResolver 两个对象并没有提供丰富的接口来处理 XML 数据。实际上，现在已提供的方法对于处理 XML 文档也是不够的。这主要是因为 SAX 是一种简单的 XML 处理接口，更加复杂的功能相对于此接口来说是非必需的。很多使用 SAX 来处理 XML 数据的应用程序都不关心这些方面的内容。如果需要处理这些特殊情况，可以使用其他 XML 处理机制。

4. ErrorHandler 接口

ErrorHandler 接口可以使得应用程序在处理 XML 文档的时候能够实现对于错误的处理。可以通过 setErrorHandler 方法来将此接口中的方法注册到解析对象上，从而在解析 XML 数据的时候实现实时的错误处理。一个处理错误信息的框架如下所示：

```
class MyHandler(ErrorHandler):
    pass

parser = make_parser()
parser.setErrorHandler(MyHandler())
```

在上面的代码中，首先定义了继承自 ErrorHandler 的 MyHandler 类。可以在此类中重载原来 ErrorHandler 类中的相关方法。在使用 make_parser 方法生成一个解析对象后，可以使用setErrorHandler方法来设置错误处理函数。

在 ErrorHandler 接口中有 warning、error 和 fatalError 等 3 种方法，可以根据错误的严重程度来调用相应的错误处理方法。

（1）warning 方法主要是处理那些在解析 XML 数据时的微小错误。当遇到这类错误的时候，默认情况下将会打印出错误信息，并且继续处理后面的数据流。如果在此处理函数中触发异常，将会导致解析过程停止。

（2）error 方法主要是处理比较严重但是可以恢复的错误。重载此方法的时候，如果不抛出异常，将继续解析后面的数据。需要注意的是，在遇到 error 的消息后，并不能保证后面的数据解析一定是正确的。

（3）fatalError 方法是处理那些非常严重且不能恢复的错误，在遇到这种事件后，解析应该停止。

这 3 个方法都只接受一个参数，那就是 SAXException 异常类的一个实例。此实例中提供了

更多资源请访问稀酷客(www.ckook.com)

丰富的方法用来获取出错的相关信息，包括出错的位置和上下文信息。如果在处理中需要终止对 XML 数据的解析，则直接触发 SAXException 异常即可。

5. XMLReader 接口

要使用以上的 4 个接口，需要将这些实现的接口注册在一个 SAX 的解析对象上。在前面的示例中已经见过，通过使用 xml. sax 的 make_parser 方法可以生成一个 XMLReader 对象。该对象支持上面所介绍的 4 个处理器接口，可以通过如 setErrorHandler 的方法来设置相应的接口。如果需要从当前的解析对象中获取已经注册的处理器接口，则将上述方法中的 set 换成 get 即可。如下面的代码所示：

```
handler = parser.getErrorHandler ( )
```

在 XMLReader 接口中，还有一个 setLocale 方法，可以允许应用程序来设置警告和错误信息的本地化。这在 XML 数据的处理中并不是必须的。除此之外，SAX 还提供了特性（features）和属性（properties）的概念。其中，特性是指在解析的时候可以打开或者关闭的功能，而属性则是和 XML 文档解析的状态相关联的一个属性值对。根据特定的 SAX 实现，这些值的读写需求并不一样，有些可能只能读，而另外一些只能写。在现在的应用程序中，包括特性和属性的使用都是比较少的。

在此接口中最重要的方法是 parse。调用此方法将开始解析 XML 数据并产生相应的事件信息，包括内容信息、DTD 信息和错误信息等。该方法接受一个参数，参数可以是文件名、URL 或文件对象，甚至是 InputSource 对象。当 parse 方法返回的时候，输入处理的过程也结束了。具体的使用方法参见前面的示例。

表 18-6 给出了在 XMLReader 接口中常用的方法。

表 18-6　XMLReader 中包含的常用方法

方　法	说　明
Parse	解析 XML 数据并产生 SAX 事件信息
getContentHandler	获取 XML 数据内容的处理函数
setContentHandler	设置 XML 数据内容的处理函数
getDTDHandler	获取 XML 数据的 DTD 定义的处理函数
setDTDHandler	设置 XML 数据的 DTD 定义的处理函数
getEntityResolver	获取 XML 数据的外部实体参考的处理函数
setEntityResolver	设置 XML 数据的外部实体参考的处理函数
getErrorHandler	获取 XML 数据的错误处理函数
setErrorHandler	设置 XML 数据的错误处理函数
setLocale	设置警告和错误信息的本地化
getFeature	获取特性数据
setFeature	设置特性数据
getProperty	获取属性数据
setProperty	设置属性数据

通过设置合适的 Handler，可以有效地处理 XML 文档数据。

18.5　使用 DOM 处理 XML 文档

DOM（Document Object Model，文档对象模型）是另外一种流行的解析和处理 XML 文档的接口规范，可以用来访问 XML 文档中的数据。在本节中将介绍 DOM 接口规范，包括 DOM 内核、对 DOM 节点的增删和修改操作以及其他相关的接口。同时通过实例来介绍这些接口的使用方法。

更多资源请访问稀酷客(www.ckook.com)

18.5.1　DOM 介绍

DOM 技术最初是为解析 HTML 语言而开发的，后来被 W3C 组织规范成为解析类 XML 语言的接口标准。在 DOM 中提供了丰富的接口用来操作 XML 数据，包括查询、增加、删除等。现在已经成为处理 XML 文档的主要方式。

DOM 在处理 XML 数据的时候，会将所有的 XML 数据在内存中形成一个树结构，所以当文件比较大的时候，可能会消耗计算机比较大的内存空间。也正因为如此，DOM 技术使得应用程序可以很方便的在不同的 DOM 节点之间进行访问，并进行相关的操作。另外，DOM 作为一项 W3C 标准，已经被大部分的浏览器支持。图 18-8 中显示了在 Firefox 浏览器中的 DOM 检查器的使用。

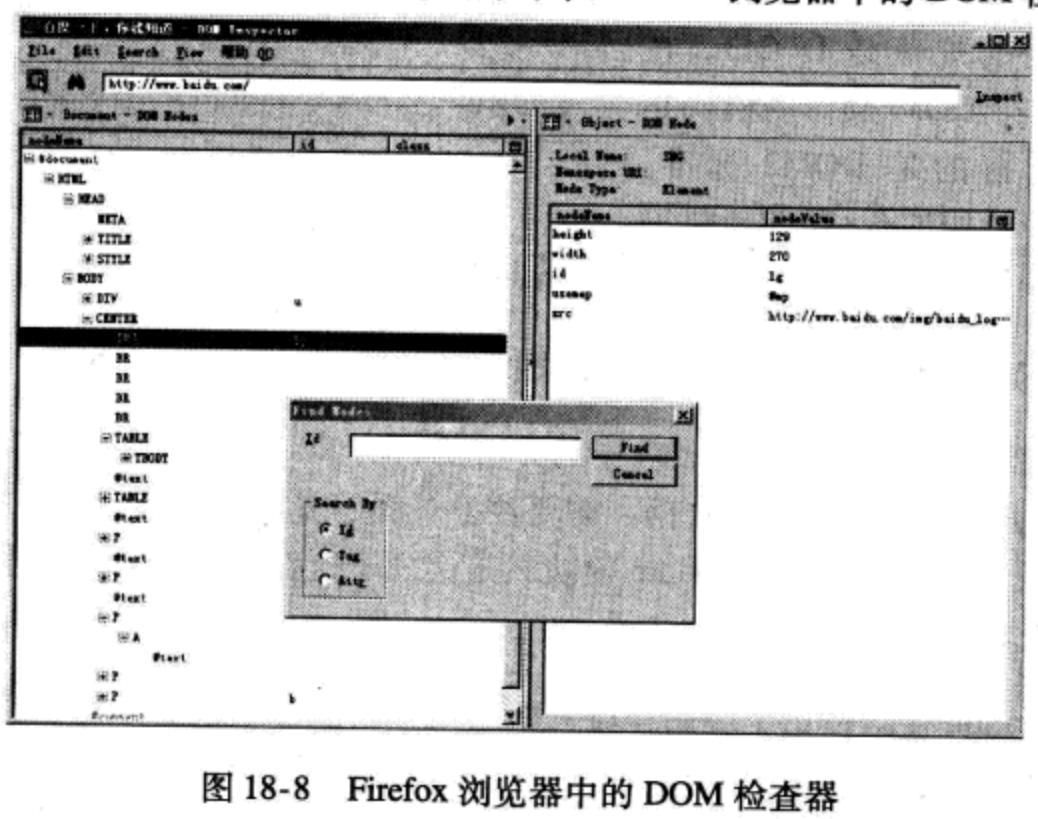


图 18-8　Firefox 浏览器中的 DOM 检查器

从图 18-8 中可以看到，DOM 技术在处理 XML 文档的时候，会将数据处理成树结构。虽然如此，实际上在 DOM 技术的发展过程中，有过新模块的加入和旧模块的废除。表 18-7 中显示了在 DOM 技术中支持的主要特性。这些特性也被应用程序所采用。

表 18-7　DOM 技术中包含的特性

特性	说　明
Core	此特性中包含处理良构的 XML 文档所需的基础结构。其中并不包含任何的 DTD 定义的相关信息。特别地，其中并没有包含外部实体参考、处理指令等相关信息。在这个特性中，提供了众多的接口来处理文档数据（这也是在后面将要详细讨论的）
XML	此特性中包含 XML 相关的一些信息，包括在上面没有规范的实体参考和定义等。另外，还包括 CDATA 段和处理指令等
Events	此特性的各个实现之间的差别比较大，而且已经分成了很多的小特性。各种实现虽然千差万别，但是都会支持最基本的 Events 特性，而其他的则只需要实现部分。这些事件中，包括面向用户的操作接口和对于文档的修改操作接口等
Range	此特性提供了一种为无法转换为一系列 DOM 节点的文档数据处理的接口。这特别适合于需要将某些数据高亮显示给用户的时候
Traversal	此特性支持应用程序在文档节点之间进行移动。当生成树结构后，可以很方便地实现对父节点、子节点和兄弟节点的遍历。另外，节点是可以被过滤的。这样，就只需要访问满足条件的节点
Views	为一个文档数据提供了多种不同类型的显示。在实际中并不常用
CSS	此特性为文档数据提供了对 CSS 版本 2 规范的访问接口。这个特性一般用在 XML 数据的浏览器显示

更多资源请访问稀酷客(www.ckook.com)

由于 DOM 的技术规范是由一系列规范组成的，所以其内容也很丰富。从 DOM 的操作上来说，DOM 提供接口将 XML 文档显示成 DOM 节点对象的分层树结构。在形成的这种文档结构中，有些节点下面可以有不同类型的节点，而另外一些则是叶子节点。其中，DOM 的文档节点和根结点是对应的。根元素节点下面可以包含子元素节点，并包括属性节点。而子元素节点则可以继续包含子元素节点，或者是文本节点。一个 DOM 树结构如图 18-9 所示。

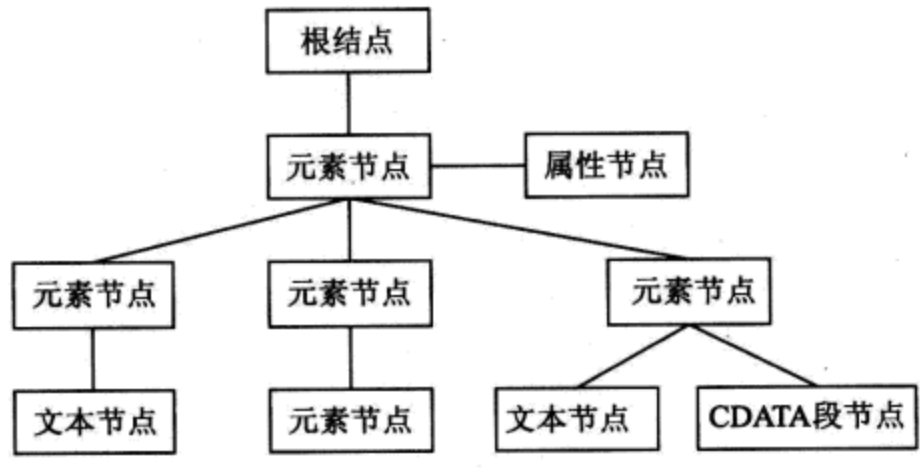


图 18-9　一个 DOM 树结构

由于 DOM 的规范过程是分阶段的，所以形成了 DOM 的层次。现在广泛使用的是 DOM2，最新的规范是 DOM3。在 Python 标准库中支持 DOM 操作的模块是 xml. dom，主要支持的是 DOM2 标准。而对 DOM3 规范中的如加载/存储规范等并不支持。在模块中提供了getDOMImplementation方法来查看在标准库中是否支持某特性。

18. 5. 2　xml. dom 模块中的接口操作

当 XML 文档数据通过 xml. dom 模块中的方法解析为 DOM 树结构后，就可以调用 DOM 规范中定义的方法和属性来访问 XML 数据。需要注意的是，这里的方法是在 DOM 中规定的，而不是在 xml. dom 模块中规定的。xml. dom 模块只是实现了这些操作 XML 数据的接口。

1. DOMImplemetation 接口

DOMImplemetation 接口提供了应用程序判断 DOM 实现是否支持某些特性的方法。其中包含 hasFeature 方法用来测试特性是否存在。此方法有两个参数，一个是 DOM 规范所支持的特性，在表 18-7 中有介绍。而第二个则是 DOM 规范的层次：

```
In [1]: from xml.dom.minidom import DOMImplementation

In [2]: implementation = DOMImplementation()

In [3]: implementation.hasFeature("Core","2.0")
Out[3]: True

In [4]: implementation.hasFeature("Events","2.0")
Out[4]: False

In [5]: implementation._features
Out[5]:
[('core', '1.0'),
 ('core', '2.0'),
 ('core', '3.0'),
 ('core', None),
 ('xml', '1.0'),
 ('xml', '2.0'),
 ('xml', '3.0'),
 ('xml', None),
 ('ls-load', '3.0'),
 ('ls-load', None)]
```

- 在 In［1］中，从 xml. dom. minidom 中导入了 DOMImplementation 接口。其中，minidom是 DOM 接口的一个轻量级实现。
- In［2］中创建了一个 implementation 对象。

更多资源请访问稀酷客(www.ckook.com)

- 在 In［3，4］中，通过调用此 implementation 对象的 hasFeature 方法，可以知道特定的层次特性是否被现在的 DOM 实现所支持。从输出结果来看，这里的 Core 版本 2.0 是支持的，而 Events 版本 2.0 则不被支持。
- 实际上，在此对象的内部变量_features 中，记录了当前 DOM 接口实现所支持的所有特性。这从 In［5］的输出结果中可以看出来。

另外，也可以使用此对象中的 createDocument 和 createDocumentType 方法来生成 Document 接口对象，如下所示：

```
In [6]: doctype = implementation.createDocumentType("Goods","","Goods.dtd")

In [7]: document = implementation.createDocument("","goods",doctype)

In [8]: document
Out[8]: <xml.dom.minidom.Document instance at 0x00BDA3F0>
```

- 在 In［6］中通过调用 createDocumentType 方法生成了一个文档定义对象，并作为后面 createDocument 方法的一个参数。
- In［7］中调用对象的 createDocument 方法生成了一个 Document 对象。此对象的具体使用方法接下来将进行介绍。

2. Document 接口

Document 接口可以用来表示整个 XML 文档。实际上，这是 XML 文档树结构的根，同时提供了对数据的初始访问入口。在上面的代码片段中所生成的 document 对象就是一个 xml.dom.minidom.Document 接口的实例。

除了能够使用上述的方法来生成 Document 接口的实例外，还可以通过解析 XML 文档数据来得到。在 minidom 模块中有 parse 和 parseString 方法。其中，前一个方法的输入参数为文件或者文件对象。而后一个方法的输入参数则是字符串：

```
In [2]: from xml.dom.minidom import parse, parseString

In [3]: dom1 = parse("xml_1.xml")

In [4]: dom1
Out [4]: <xml.dom.minidom.Document instance at 0x01527F80>

In [5]: fp =  open (" xml_1.xml"," r")

In [6]: dom2 = parse (fp)

In [7]: dom2
Out [7]: <xml.dom.minidom.Document instance at 0x01541120>

In [8]: fp.close ()

In [9]: data = """
   ...: <goods>
   ...:       <shirt quality=" A" >
   ...:           <size>170</size>
   ...:       </shirt>
   ...: </goods>
   ...: """

In [10]: dom3 = parseString (data)

In [11]: dom3
Out [11]: <xml.dom.minidom.Document instance at 0x015B16E8>
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［2］中从 xml. dom. minidom 导入了 parse 和 parseString 方法。
- In［3，4］中演示了通过文件名来调用 parse 方法的过程。在 Out［4］中可以看出已经生成了一个 Document 对象。
- In［5，6，7，8］中演示了通过文件对象来调用 parse 方法的过程。可以看到生成了 Document 对象。
- 在 In［8，9，10］中可以看到，当需要解析的 XML 数据为字符串输入的时候，可以使用 parseString 方法来实现解析。结果也是和上面同样的生成了 Document 对象。

由于 Document 接口是整个文档结构的根，所以其输出即为 XML 文档的输出。通过调用 toxml 方法可以将整个 XML 文档输出：

```
In [12]: dom3.toxml()
Out[12]: u'<? xml version = "1.0" ? > <goods > \n     <shirt quality = "A" > \n          <size >
170 </size > \n     </shirt > \n </goods >'

In [13]: print dom3.toxml()
<? xml version = "1.0" ? > <goods >
    <shirt quality = "A" >
        <size >170 </size >
    </shirt >
</goods >
```

- 通过调用 toxml 方法，可以将此文档对象输出。另外可以看到，在输出中添加了“<? xml version = ‘1.0’ ? >”部分。

在 Document 对象中，还提供了多种生成节点的方法。如通过 createElement 生成元素节点，通过 createTextNode 生成文本节点，通过 createAttribute 生成属性节点等。通过生成这些节点，并利用后面部分介绍的节点操作方法，就可以组成一个完整的 XML 树结构了：

```
In [14]: elem = dom3.createElement("element")

In [15]: elem.toxml()
Out[15]: '<element/ >'

In [16]: text = dom3.createTextNode("Text date")

In [17]: text.toxml()
Out[17]: 'Text date'

In [18]: attrib = dom3.createAttribute("quality")

In [19]: attrib.name
Out[19]: 'quality'
```

- 在 In［14］中生成了一个元素节点，在 In［15］中调用 toxml 方法将其结果输出，输出采用的是标签的形式。
- 在 In［16］中生成了一个文本节点，在 In［17］中调用 toxml 方法将其结果输出，其输出为字符串的形式。
- 在 In［18］中生成了一个属性节点，由于它不是 Node 接口对象，所以不能调用 toxml 方法。直接读取其 name 属性，可以看到前面刚刚设置的值。

3. Node 接口

Node 接口是文档对象模型的基本数据类型，用来表示所构建文档树结构的每个单个节点。这是 DOM 中最常用的接口之一，而且是其他如 Document 接口的基础。也就是说，Document 对象只是 Node 对象的一个特例，Node 接口中支持的操作在 Document 对象中都可以使用。在 DOM

更多资源请访问稀酷客(www.ckook.com)

中的结构都是 Node 对象，每个节点的类型通过 nodeType 来区别。不同的节点对象有着不同的节点类型，包括元素节点、属性节点和文本节点等：

```
In [2]: from xml.dom.minidom import *

In [3]: dom1 = parse("xml_1.xml")

In [4]: root = dom1.documentElement

In [5]: root
Out[5]: <DOM Element: goods at 0x1533468>

In [6]: root.nodeType
Out[6]: 1

In [7]: root.ELEMENT_NODE
Out[7]: 1
```

- In［2，3］中的操作和前面的相同。
- 在得到 dom1 对象后，可以使用 Document 接口中的 documentElement 属性来获得根元素节点，并赋值给 root 变量。
- 在 In［5］中，可以看到这个节点是一个元素节点。在 In［6］中通过其 nodeType 属性来查看其节点类型。这里得到的值为“1”，这也是元素节点的数值表示，从 In［7］中可以看到这一点。在 Node 接口中还定义了其他的节点类型，如表 18-8 所示。

表 18-8　Node 接口中支持的节点类型

节点类型	数值表示
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

由于 XML 文档可以解析成为树结构对象，所以这里的节点也具有树的特性。节点可能有父节点、子节点或者是兄弟节点等。可以通过 parentNode 值来获取父节点，通过 childNodes 获取子节点集合。而 firstChild 和 lastChild 则分别是第一个和最后一个子节点。另外，通过使用 previousSibling和 nextSibling 可以获得属于同一个父节点下的前一个和后一个兄弟节点：

```
In [8]: childs = root.childNodes

In [9]: childs
Out[9]:
[<DOM Text node "
```

更多资源请访问稀酷客(www.ckook.com)

```
    " >,
 <DOM Element: shirt at 0x1533508 >,
 <DOM Text node "
    " >,
 <DOM Element: shirt at 0x1533850 >,
 <DOM Text node "
" >]

In [10]: shirt = root.childNodes[1]

In [11]: shirt
Out[11]: <DOM Element: shirt at 0x1533508 >

In [12]: shirt.parentNode
Out[12]: <DOM Element: goods at 0x1533468 >

In [13]: parent = shirt.parentNode

In [14]: parent == root
Out[14]: True
```

- 在 In［8］中，通过 childNodes 属性获得了根结点下的所有节点对象。从 Out［9］中可以看出其中有 3 个文本节点和两个元素节点。这里之所以出现了文本节点，是因为在 XML 的解析中，对空格字符也会作为文本来对待。
- 在 In［10］中将此列表的第二项赋值给了 shirt 变量。接下来，从 Out［11］中可以看出这是一个元素节点。
- 在 In［12，13，14］中测试了此 shirt 元素节点的父节点是否和根结点是同一对象。从 Out［14］的结果可以看到，两者确实是同一对象：

```
In [15]: root.firstChild
Out[15]:
<DOM Text node "
    " >

In [16]: root.lastChild
Out[16]:
<DOM Text node "
" >

In [17]: shirt.previousSibling
Out[17]:
<DOM Text node "
    " >

In [18]: shirt.nextSibling
Out[18]:
<DOM Text node "
    " >
```

- 在 In［15，16］中，通过 firstChild 和 lastChild 分别得到了 root 根元素节点的第一个子节点和最后一个子节点。
- 在 In［17，18］中，通过 previousSibling 和 nextSibling 分别得到了 shirt 元素节点的前一个兄弟节点和后一个兄弟节点。

另外，可以通过 atributes 来获取 Node 对象的所有属性值。此属性只对于元素节点起作用，而对于其他节点则返回 None：

更多资源请访问稀酷客(www.ckook.com)

```
In [19]: attrib = shirt.attributes

In [20]: attrib.items()
Out[20]: [(u'quality', u'A')]
```

- 在 In［19］中通过 attributes 获取了 shirt 元素的属性，这是一个 NamedNodeMap 对象。接下来，通过调用 items 方法，可以得到此元素的所有属性值对。

除了上面的属性可以获取 Node 对象的信息以外，还提供了操作节点的方法。其中 appendChild方法可以用来创建一个子节点，removeChild 方法用来删除一个节点。另外，replaceChild方法可以将一个已有的节点使用另外一个节点对象替代。这些方法将在 18.5.3 节中进行介绍。

4. Element 接口

Element 类也是 Node 类的子类，所以 Node 中支持的属性和方法在 Element 类中都可以使用。此接口主要用来实现对元素节点的操作。

节点的元素名字可以通过 tagName 来获得：

```
In [21]: shirt.tagName
Out[21]: u'shirt'
```

在上面得到的 shirt 对象就是一个 Element 对象。可以通过其 tagName 属性来获得此元素节点的名字。

在 Element 接口中提供了 getAttribute 和 setAttribute 方法用来对元素节点的属性进行操作。这两个方法分别用来获取元素的属性和设置元素的属性：

```
In [22]: shirt.getAttribute("quality")
Out[22]: u'A'

In [23]: print shirt.toxml()
<shirt quality="A">
        <name>Helen</name>
        <size>170</size>
        <quantity>10</quantity>
        <color>black</color>
    </shirt>

In [24]: shirt.setAttribute("quality","B")

In [25]: print shirt.toxml()
<shirt quality="B">
        <name>Helen</name>
        <size>170</size>
        <quantity>10</quantity>
        <color>black</color>
    </shirt>
```

- 在 In［22］中通过调用 getAttribute 方法获得了属性名为“quality”的值。这与 In［23］中的输出结果也是一致的。
- 在 In［24］中通过调用 setAttribute 方法设置了属性名为“quality”的值为“B”。这从 In［25］的输出结果中也可以看出来。这里，由于需要设置的属性名是存在的，所以就修改了原来的值。而当属性名在元素中是不存在的时候，则将新建一个属性值对。

更多资源请访问稀酷客(www.ckook.com)

5. Text 接口

Text 也是实现了 Node 接口，但是文本节点不能再有子节点。这是 XML 文档中主要用来保存数据的部分。其对象中有一个 data 属性，用来表示当前节点的数据：

```
In [26]: name = shirt.childNodes[1]

In [27]: name.childNodes
Out[27]: [<DOM Text node "Helen">]

In [28]: name_text = name.childNodes[0]

In [29]: name_text.data
Out[29]: u'Helen'
```

- 在 In［26］中得到了 shirt 元素下的 name 子节点，接着在 In［28］中得到了 name 元素节点下的文本节点。
- 在 In［29］中通过其对象的 data 属性得到了文本节点的内容。

6. 支持的 DOM 接口小结

除了上面介绍的一些 DOM 接口外，在 Python 的 XML 处理标准库中还有其他一些接口。在 xml.dom 模块中所支持的接口如表 18-9 所示。

表 18-9　xml.dom 模块中包含的接口

接　口	说　明	常用属性和方法
DOMImplemetation	访问底层实现的接口	hasFeature：检测模块是否实现了特定的特性
		createDocument：返回一个新的 DOM 对象
Node	文档节点的基本访问接口	nodeType：只读属性，指示节点的类型，包括 ELEMENT_NODE,ATTRIBUTE_NODE，TEXT_NODE，CDATA_SECTION_NODE，ENTITY_NODE，PROCESSING_INSTRUCTION_NODE，COMMENT_NODE，DOCUMENT_NODE，DOCUMENT_TYPE_NODE，NOTATION_NODE
		parentNode：当前节点的父节点
		childNodes：当前节点的子节点
		firstChild：如果有的话，返回第一个子节点
		lastChild：如果有的话，返回最后一个子节点
		hasChildNodes：测试是否有子节点
		appendChild：添加节点
		removeChild：删除节点
		replaceChild：替换节点
Nodelist	访问一系列节点的接口	item：返回序列中的对象
		length：节点集合中包含的节点个数
DocumentType	包含文档定义等信息	name：在 DOCTYPE 定义的根元素名字
		entities：外部实体参考的定义
		notations：符号定义

更多资源请访问稀酷客(www.ckook.com)

（续）

接　口	说　明	常用属性和方法
Document	表示着访问整个 XML 文档	documentElement：XML 文档的根元素
		createElement：创建一个元素
		createTextNode：创建一个文本节点
		createAttribute：创建属性节点
		createComment：创建注释
		getElementsByTagName：通过标签名字获取元素
Element	DOM 树中的元素节点	tagName：元素的名字
		hasAttribute：在元素中查看是否有特定的属性
		getAttribute：获取特定的属性
		setAttribute：设置特定的属性
Attr	元素节点的属性值	name：属性名字
		localName：名字空间内容
		prefix：名字空间前缀，没有则为空
Comment	源文档中的注释	data：注释的数据
Text	XML 文档中的文本数据	data：文本内容的数据
ProcessingInstruction	访问处理指令的接口	target：处理指令中的目标，为第一个空格前的字符串
		data：处理指令中的内容，为第一个空格后的字符串

在 DOM 的树结构中，各个类型的节点父接口都是 Node，有不少其他接口都是继承自 Node，如介绍过的 Element、Text 以及 Attr 和 Comment 等。

18.5.3　对 XML 文档的操作

DOM 技术是将整个 XML 文档转换成一个树结构并放在内存中。这种处理方式相对于 SAX 的一个好处就是可以很方便地对 XML 文档中的数据进行操作，包括查询、增加和修改等。在本小节中，将具体介绍使用 DOM 接口对 XML 数据的操作方法。

1. 对 XML 文档的遍历

通过使用 childNodes 属性，可以实现对一个节点对象的遍历。在下面的代码片段中，利用 childNodes 属性，并使用递归实现此功能：

```
In [30]: def traversal(node):
   ....:     if not node.childNodes: #递归结束条件
   ....:         return
   ....:     for child in node.childNodes:
   ....:         print child
   ....:         traversal(child) #递归调用
   ....:

In [31]: traversal(root)
<DOM Text node "
    ">
<DOM Element: shirt at 0x15420d0>
<DOM Text node "
        ">
<DOM Element: name at 0x1586df0>
```

更多资源请访问稀酷客(www.ckook.com)

```
<DOM Text node "Helen">
<DOM Text node "
        ">
#省略部分输出结果
<DOM Element: color at 0x149e4e0>
<DOM Text node "black">
<DOM Text node "
    ">
<DOM Text node "
    ">
#省略部分输出结果
<DOM Element: color at 0x1588148>
<DOM Text node "white">
<DOM Text node "
    ">
<DOM Text node "
">
```

- 在 In［30］中，定义了一个 travarsal 函数。此函数的功能是遍历一个指定的节点对象。此函数是一个递归函数。函数主体分为两部分，前一部分是递归终止条件，后一部分是递归过程。在前一部分中，当节点不再含有子节点的时候，则针对此子节点的递归过程结束。在后一部分中，对节点对象中的每个子节点，都需要调用 traversal 函数来处理。同时，在递归过程中，打印了节点对象。
- 上述函数遍历的结果在 In［31］的输出结果中可以看到。需要注意的是，在结果中出现了很多没有实际内容的文本节点，这些文本节点的内容都是硬回车所致。在实际的处理中，可以将这些节点忽略。

实际上，遍历节点不仅仅只是可以通过 childNodes 来实现。在节点对象的属性中，还可以通过其他的方法来实现。如结合 firstChild 和 nextSibling 来实现。而如果结合 lastChild 和 previousSibling，则可以用来实现对于一个节点对象的遍历。下面的代码片段中演示了如何使用 firstChild 和 nextSibling 结合来实现对于一个节点的遍历：

```
def another_traversal(node):
    if not node.childNodes:
        return

    firstChild = node.firstChild
    next = firstChild
    while next:
        print next
        another_traversal(next)
        next = next.nextSibling
```

- 这里定义了另外一个用来遍历节点的函数 another_traversal。主要利用了 firstChild 和 nextSibling 两个节点的属性来实现。在函数主体的第一部分，递归结束条件中和traversal 函数中是一样的，也就是说，当节点没有子节点的时候结束。接下来，获取此节点的第一个子节点，并将此值赋值给变量 next。然后对此节点进行遍历，同时使用 nextSibling 属性来求得子节点的下一个兄弟节点，直到将所有的兄弟节点都访问到为止。
- 此函数对于 root 对象的运行结果和 traversal 函数是一样的，这里省略了输出结果。

2. 对 XML 文档的查询

对于 XML 数据中的查询，一种可行的方式是利用前面的遍历函数，并加入判断条件来确定元素。

更多资源请访问稀酷客(www.ckook.com)

【例 18-13】寻找元素名字为“name”的元素节点。

```
def search_by_traversal (node, name):
    if not node.childNodes:
        return
    for child in node.childNodes:
        if child.nodeType == Node.ELEMENT_NODE and child.tagName == name:
            print child.toxml()
        traversal(child)
```

- 代码中定义了 search_by_traversal 函数，有两个参数，第一个为需要查询的节点，第二个为需要查询的元素名字。
- 函数主体部分和 traversal 函数是类似的，只是在输出部分做了一些修改。只有当节点对象为 ELEMENT_NODE 元素节点且其元素名字为指定的名字的时候，才打印出此元素。这样，可以找到指定节点下的所有的元素名字为“name”的元素节点。

【例 18-13】运行结果如下：

```
In [35]: search_by_traversal (root, "color")
<color>black</color>
<color>white</color>
```

- 在 In［35］中，在根元素节点下寻找元素名字为“color”的元素节点。可以看到，最终找到了两个节点，并打印出来。如果运行的时候出现 Node 或者 ELEMENT_NODE 这类没有定义的错误，则需要在之前从 xml.dom.minidom 中将其导入。

另外一种做法是使用 Python 标准库中提供的方法 getElementsByTagName。此方法作用在元素节点上。通过指定元素名字来返回当前节点树下所有指定了属性名字的节点。下面的代码实现了上面 In［35］中同样的功能：

```
In [36]: root.getElementsByTagName("color")
Out[36]: [<DOM Element: color at 0x149e4e0>, <DOM Element: color at 0x1588148>]
```

- 在 In［36］中，通过在 root 对象上调用 getElementsByTagName 方法，实现了对此节点的元素查询功能。Out［36］中给出了查找到的两个符合条件的对象。通过调用 toxml 方法可以知道这两个对象和 In［35］中的输出结果是一样的：

```
In [37]: colors = root.getElementsByTagName("color")

In [38]: for color in colors:
   ....:     print color.toxml()
   ....:

<color>black</color>
<color>white</color>
```

- 从上面的输出结果中可以看到确实和前面一样实现了查询功能。

只要是搜索到的结果还是元素节点，就可以使用 getElementsByTagName 继续搜索。如在下面的代码片段中，可以首先查找 shirt 元素，然后对找到的每个 shirt 元素来查找 color 元素，同样的可以实现前面的查询功能：

```
In [39]: shirts = root.getElementsByTagName("shirt")

In [40]: shirt1 = shirts[0]

In [41]: color1 = shirt1.getElementsByTagName("color")[0]

In [42]: color1.toxml()
```

更多资源请访问稀酷客(www.ckook.com)

```
Out[42]: u'<color>black</color>'

In [43]: shirt2 = shirts[1]

In [44]: color2 = shirt2.getElementsByTagName("color")[0]

In [45]: color2.toxml()
Out[45]: u'<color>white</color>'
```

- 在 In［39］中，获得了根元素节点下元素名字为“shirt”的所有元素节点。这里，由于知道 XML 的内容，所以没有对其进行判断，而直接将返回元素列表的两个值分别赋值给了 shirt1 和 shirt2。但是在实际操作中，还是需要对返回值进行判断的，否则有可能会触发异常。同样地，后面对“color”元素名字的查找也没有进行判断。
- 在 In［41，42］中，在 shirt1 元素中查找 color 元素。而在 In［44，45］中，则是在 shirt2 中查找 color 元素。从输出结果来看，同样的完成了查找 XML 文档中所有元素名字为“color”的元素节点的任务。

从上面可以看出，getElementsByTagName 查找的时候实际上是递归的，也就是在指定节点下的所有符合要求的元素节点都会被找到。

3. 对 XML 文档的增、删、改

DOM 技术一个重要的特性，就是能够实现对 XML 文档数据的修改。这一点是 SAX 基于事件的处理方式不能实现的。在 DOM 接口中，提供了丰富的接口用来对 XML 文档数据进行处理。其中，在 Document 接口中提供了产生各种类型节点的方法，这在 18.5.2 节中的 Document 接口部分已经介绍过了。例如通过 createElement 创建一个元素节点等：

```
In [46]: goods = dom1.createElement("goods")

In [47]: goods.toxml()
Out[47]: '<goods/>'

In [48]: test = dom1.createElement("test")
```

- 在 In［46］中，通过使用 Document 接口中的 createElement 方法生成了一个元素节点。在 In［47］的输出结果中可以看到生成了一个空元素。在 In［48］中生成了一个测试用的元素节点，将在后面使用这个对象。

同样地，可以生成其他类型的节点，如文本节点：

```
In [49]: text = dom1.createTextNode("This is a text node")

In [50]: text
Out[50]: <DOM Text node "This is a..." >
```

- 在 In［49］中生成了一个文本节点，其文本内容为“This is a text node”。在 In［50］的输出结果中可以看到这点。

在 Node 接口中提供了 appendChild 方法，可以为指定的节点创建一个子节点。方法的参数为需要增加的子节点：

```
In [51]: test.appendChild(text)
Out[51]: <DOM Text node "This is a..." >

In [52]: goods.appendChild(test)
Out[52]: <DOM Element: test at 0x15883c8 >

In [53]: goods
```

更多资源请访问稀酷客(www.ckook.com)

```
Out[53]: <DOM Element: goods at 0x1587f58>

In [54]: goods.toxml()
Out[54]: '<goods><test>This is a text node</test></goods>'
```

- 在 In［51］中调用了 test 对象的 appendChild 方法，其参数为 text。而在 In［52］中则调用了 goods 对象的 appendChild 方法，其参数为 test。操作的结果将使得 text 作为 test 的子节点，而 test 作为 goods 的子节点。在 In［54］中可以看到现在 goods 元素节点已经不是空元素了，而是一个包含子节点的元素节点。

同样地，接口中也提供了 removeChild 方法来删除特定的子节点。此方法的参数必须为指定节点的子节点，否则将触发异常：

```
In [55]: goods.removeChild(test)
Out[55]: <DOM Element: test at 0x15883c8>

In [56]: goods
Out[56]: <DOM Element: goods at 0x1587f58>

In [57]: goods.toxml()
Out[57]: '<goods/>'
```

- 在 In［55］中，通过调用 goods 的 removeChild 方法将 test 元素从 goods 的 XML 树结构中删除。从 Out［57］的结果可以看出，现在 goods 对象又是一个空元素了。另外，从 Out［56］和 Out［53］的输出结果来看，goods 对象的地址始终是相同的。

除此之外，Node 接口中还提供了 replaceChild 方法用来替换已有的节点。两个参数分别是替换的节点和待替换的节点：

```
In [58]: goods.appendChild(test)
Out[58]: <DOM Element: test at 0x15883c8>

In [59]: goods.toxml()
Out[59]: '<goods><test>This is a text node</test></goods>'

In [60]: test2 = dom1.createElement("test2")

In [61]: goods.replaceChild(test2,test)
Out[61]: <DOM Element: test at 0x15883c8>

In [62]: goods.toxml()
Out[62]: '<goods><test2/></goods>'
```

- 在 In［58，59］中，通过使用 appendChild 方法重新将 test 元素节点作为其子节点。从 Out［59］中可以看到现在 goods 对象的 XML 数据。
- 在 In［60］中重新创建一个元素节点 test2。
- 在 In［61］中，使用 replaceChild 方法将 test 元素节点使用 test2 元素节点替换。Out［62］的输出结果可以看到这次替换节点后的结果。

结合上面介绍的操作，可以很容易地创建 XML 文档，或者对 XML 文档的数据进行修改。

【例 18-14】演示生成 xml_1.xml 中的部分内容。

```
In [63]: goods.removeChild(test2)
Out[63]: <DOM Element: test2 at 0x158ca80>

In [64]: shirt = dom1.createElement("shirt")
```

更多资源请访问稀酷客(www.ckook.com)

```
In [65]: goods.appendChild(shirt)
Out[65]: <DOM Element: shirt at 0x15f0260>

In [66]: shirt.setAttribute("quality","A")

In [67]: name = dom1.createElement("name")

In [68]: text = dom1.createTextNode("Helen")

In [69]: name.appendChild(text)
Out[69]: <DOM Text node "Helen">

In [70]: shirt.appendChild(name)
Out[70]: <DOM Element: name at 0x1436828>

In [71]: size = dom1.createElement("size")

In [72]: text = dom1.createTextNode("170")

In [73]: size.appendChild(text)
Out[73]: <DOM Text node "170">

In [74]: shirt.appendChild(size)
Out[74]: <DOM Element: size at 0x158fa30>

In [75]: goods.toxml()
Out[75]: '<goods><shirt quality="A"><name>Helen</name><size>170</size></shirt>
</goods>'
```

- In［63］中调用 removeChild 将 test2 从 goods 的子节点中删除。现在 goods 对象又成为了一个空元素节点。
- 在 In［64，65］中新生成了一个 shirt 元素节点，并将其作为 goods 对象的子节点。
- 在 In［66］中通过调用 setAttribute 设置了一个 shirt 的元素属性。其属性名为“quality”，而其值为“A”。
- 在 In［67 - 70］生成了一个 name 元素节点，并将其作为 shirt 的一个子节点。其中 name 的文本内容为“Helen”。
- 同样地，在 In［71 - 74］中生成了一个 size 元素节点，并将其也作为 shirt 的一个子节点。
- 在 In［75］中，将 goods 对象的 XML 数据输出。从输出结果中可以看到，这里的内容和 xml_1.xml 文件中的部分内容是完全一致的。

当使用 DOM 技术的时候，要灵活地使用 DOM 接口中所提供的处理 XML 文档的增、删、改操作方法。这对于需要频繁改变的 XML 文档也是适用的。

18.6　小结

本章主要讲解了 Python 语言在 XML 领域的使用。首先对 XML 的基本概念进行了介绍并解释了 XML 文档的良构性和有效性。接下来通过对于一个 XML 文档的剖析介绍了 XML 文档中的重要组成部分，包括元素、属性和注释等。最后，介绍了现在的两种处理 XML 的方式：SAX 和 DOM，通过实例演示了两种技术的具体用法。

更多资源请访问稀酷客(www.ckook.com)

第 19 章　Python 的 Web 开发——Django 框架的应用

随着各种网络应用都使用浏览器作为客户端，Web 开发已经成为非常重要的一个领域。Python语言提供了多种 Web 开发框架供开发者选择。其中，Django 因为易用和功能的平衡而得到了广泛的使用。本章的知识点：

- 常见的 Web 开发框架
- MVC 模式的原理及使用方法
- Django 中的 MVC 架构
- Django 开发环境的搭建
- Django 框架的视图和模板系统
- Django 框架中的 URL 设计
- Django 框架的高级应用

19.1　常见的 Web 开发框架

随着 Internet 的迅速发展，越来越多的应用开始具有网络功能。这些应用的加入，使得应用的业务逻辑变得非常复杂。为了使 Web 开发人员更加关注于应用的业务逻辑而不是对于基础 HTML 的处理，出现了 Web 开发框架。框架已经定义了一系列的处理方式和模板，Web 开发人员只需要开发属于应用的业务逻辑部分即可。除此之外，大部分框架还具有分层的作用，使得业务逻辑可以细化到不同的逻辑层次，从而实现了组件化。通过使用 Web 开发框架，可以迅速地开发出具有类似模板但针对不同业务逻辑的系统。

在 Python 语言中，已经有了多种 Web 开发框架。其中不但有老牌的 Zope，还有新生的 Django；不但有模块化构建的 Turbogears，也有小巧简单的 web. py 等。在已有十几种框架的情况下，仍然还有不同的 Web 框架出现。本节将介绍使用比较多的几种框架，并对各种 Web 框架的优劣做初步的比较。

19.1.1　Zope

Zope 是 Python 世界中 Web 开发的“杀手级”框架，其中有 Web 开发中所需要的绝大部分内容。Zope 是主要使用 Python 语言开发的开放源代码 Web 应用服务器。可以运行在包括 Linux、Windows 等多种系统平台上。其使用组件模式开发，可以很方便的加入所需的模块。强大的功能包括多种数据库适配器、内置的 Web 服务器、多协议支持、索引和查找、内置安全模型和集群等。这些功能使得其成为众多 Web 开发框架中最具商业价值的框架。

Zope 框架主要是由 Zope 公司领导开发并提供技术支持。由于其开放源码的特性，来自世界各地的开发者都为其作出了重要贡献。Zope 框架的主页地址为 http://www. zope. org/。现在的 Zope 框架有版本 2（Zope2）和版本 3（Zope3）这两种版本。由于 Zope3 和 Zope2 的设计目标并不完全一致，从而导致两者有着比较大的区别。现在，Zope2 和 Zope3 在两个分支上并行的

更多资源请访问稀酷客(www.ckook.com)

发展。

Zope2 的最初设计目标是能够创建动态生成的网站并且通过 Web 来对其继续管理，即最初是作为一个网页服务器来设计的，但是在实现的版本中，Zope2 的功能非常强大，具备了对开发人员很好的支持，例如强大的内容管理系统、具有面向对象的数据库和 Python 的良好支持等。这使得许多的开发人员将其作为一个 Web 应用开发平台，而和原来的开发动态网站的初衷没有什么关系了。

为了将 Zope2 框架中的这部分 Web 应用开发功能抽取出来，Zope3 被开发出来。其设计初衷就是作为 Web 应用开发平台。如果在选型的时候需要选择一个网络开发平台，则可以使用 Zope3。由于 Zope3 的改变相对比较大，使用大量的设计模式，框架的可重用性也比较高。也正因为如此，原来在 Zope2 平台开发的代码并不能不加修改的用在 Zope3 平台上，这也进一步阻止了 Zope3 框架的普及。不过，Zope3 优良的设计将会使得 Web 应用向其转移——虽然这个过程比较漫长。

在 Zope 中，一些主要的强大功能如下。

（1）支持多平台系统。由于 Zope 是使用 Python 语言开发的，这使 Zope 具有了 Python 的跨平台特性。现在，除了 Linux、Windows 系统平台以外，Zope 框架还支持 Solaris、MacOS X 和 FreeBSD 等系统平台。

（2）内置的面向对象数据库。因为已经内置了一个完全面向对象且易用的数据库，所以使用 Zope 不需要安装数据库。在 Zope 中创建的每个对象，包括文档、图片和文件夹等，都可以保存在这个数据库中。

（3）丰富的数据库适配器。当不想使用 Zope 所提供的数据库时，可以通过 Zope 支持的数据库适配器来支持自己所选择的其他数据库。这些数据库系统支持包括 Oracle、MySQL、PostgreSQL和 Microsoft SQL 等。除此之外，还包含并不是传统数据库的对象，如 LDAP 和 IMAP 等。

（4）内置的 Web 服务器。Zope 有一个内置的 Web 服务器，具有快速多线程处理的能力。这是通过 Twisted 框架来支持的。一般情况下，这个 Web 服务器足够用了。

（5）多协议支持。在 Zope 中，现在已经存在的许多 Internet 协议都得到了很好的支持。包括 HTTP、FTP 和 Telnet 等。另外，对于其他 Web 相关的技术如 DOM、XML、SOAP 和WebDAV 都有良好的支持。

（6）内置安全模型。在 Zope 中内置有动态的安全模型，可以提供强大的安全选项和能力。这使得开发者可以通过编辑权限列表为整个网站设置权限，也可以通过设置对象属性来为每个特定的对象设置安全特性。这是商业应用的一个基础。

（7）集群和负载均衡。在 Zope 产品中，包含 Zope 企业选项（Zope Enterprise Options，ZEO）来为 Zope 提供集群和负载均衡功能。这是商业使用的另一个基础。

（8）开源和可扩展性。Zope 是开放源码的，这使得所有的开发人员都可以看到源码并进行修改。在这个过程中，Zope 公司对主分支的源码修改进行控制。同时，Zope 架构具有着很好的可扩展性。可以通过自己编写组件来扩展 Zope 的能力。

在 Zope 框架的基础上，形成了内容管理框架（Content Management Framework，CMF）。使用 CMF 可以很方便的构建内容管理应用。Plone 则是其中的佼佼者。Plone 是一个开放源码的内容管理系统。它具有用户友好和功能强大的特点，拥有很多重量级的客户。现在，Plone 已经从公认最优秀的内容管理系统开始发展成为一个强大的应用开发平台。

19.1.2　TurboGears

TurboGears 框架的目的是将现在各个层次中最优秀的组件组合起来，形成一个简单易用、对

更多资源请访问稀酷客(www.ckook.com)

用户友好的框架。通过组合方式，使得 Web 开发者可以使用 Python 中各个层次被推荐的模块。当然，如果愿意的话，也可以通过自己的修改来对部分组件进行替换。这种整合方式，使得TurboGears框架提供了 Web 开发从前端到后端的整合。

TurboGears 框架的项目地址为 http://www.turbogears.org。可以从项目主页上面下载TurboGears的软件包。当然，也可以直接在 Python 的命令行中使用 easy_install 命令来安装：

```
In [1]: ! easy_install turbogears
#输出结果经过修改
Searching for turbogears
Downloading http://files.turbogears.org/eggs/TurboGears-1.1b1-py2.5.egg
Adding TurboGears 1.1b1 to easy-install.pth file
Searching for tgMochiKit
Downloading http://pypi.python.org/packages/2.5/t/tgMochiKit/tgMochiKit-0.2beta2-py2.5.egg
Adding tgMochiKit 0.2beta2 to easy-install.pth file
Searching for TurboJson>=1.2.1
Downloading http://pypi.python.org/packages/2.5/T/TurboJson/TurboJson-1.2.1-py2.5.egg
Adding TurboJson 1.2.1 to easy-install.pth file
Searching for TurboCheetah>=1.0
Best match: TurboCheetah 1.0
Downloading http://files.turbogears.org/eggs/TurboCheetah-1.0-py2.5.egg
Processing TurboCheetah-1.0-py2.5.egg
Adding TurboCheetah 1.0 to easy-install.pth file
Searching for simplejson>=1.9.1
Best match: simplejson 2.0.4
Downloading http://files.turbogears.org/eggs/simplejson-2.0.4-py2.5-win32.egg
Processing simplejson-2.0.4-py2.5-win32.egg
Adding simplejson 2.0.4 to easy-install.pth file
#省略部分输出结果
Finished processing dependencies for turbogears
```

在 In［1］中，调用 easy_install 方法来安装 TurboGears 框架。从其输出结果来看，除了安装 TurboGears 基本组件外，还同时安装了许多相关的模块。使用 easy_install 的安装方法，可以减少需要下载多个软件安装包的困难。

在 TurboGears 的基本开发框架中，系统将分为如图 19-1 所示的层次，并使用了相应的组件。在 ORM 层可以使用 SQLObject；在 Web 的控制核心使用的是 CherryPy；在模板系统则可以是 Kid 模板系统；在 Web 的 AJAX 库中使用的是 Mochikit。

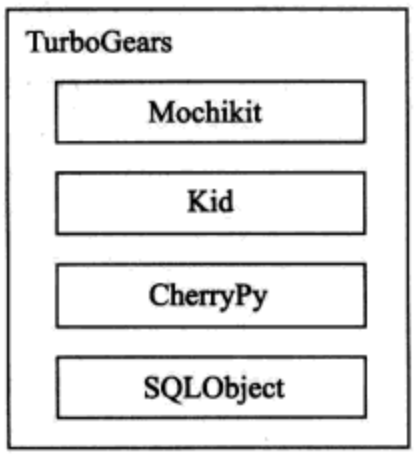


图 19-1　TurboGears 框架的基本组成

由于 TurboGears 框架中组件的可替代性，从而可以使用熟悉的模块来构建 Web 应用。除了上面提到的基本组成以外，TurboGears 还提供了丰富的组件。这些组件包括网页认证组件、注册组件和信息管理组件等。这种组件式的框架使用方式也为人所喜欢。

19.1.3　Django

相对于 TurboGears 的组件集合的形式，Django 是一个完整的快速 Web 开发框架。使用Django框架来开发 Web 应用，可以将精力集中关注在其最关键性的问题上。这样可以在一个相对比较短的时间里完成一个具有丰富功能的 Web 应用。Django 框架是从实际项目中诞生出来的，这使得其框架中提供的功能特别适合动态网站的建设，特别是管理接口。项目最初是由 Lawrence Journal-World 报纸的两位程序员 Adrian Holovaty 和 Simon Willison 为了能够更好的发布新闻而创建的项目。自从 2005 年发布为开源项目之后，全世界的开发工作者都可以为此贡献代码。

更多资源请访问稀酷客(www.ckook.com)

在 Django 框架中包含开发 Web 网络应用所需的组件。这些组件包括数据库的对象关系映射、动态内容管理的模板系统和丰富的管理界面。在 Django 框架中没有使用像 TurboGears 框架中那样的第三方 Python 模块。在 Django 框架中，可以使用管理脚本文件 manage. py 来构建简单的开发服务器。当然，同样也可以使用 mob_python 或者 FastCGI 模块来构建。

Django 框架作为一种快速的网络框架，还有下面的一些特点。

（1）组件的合理集成。在 Django 框架中，已经有一套集成在一起的组件。这些组件都是 Django 项目组开发的，并为开源界所修改和使用。Django 框架中组件的设计目的是可以实现重用性，并具有易用性。

（2）对象关系映射和多数据库支持。Django 框架的数据库组件——对象关系映射（Object-Relation Mapper,ORM）提供了数据模块和数据引擎之间的接口。支持的数据库包括 PostgreSQL、MySQL 和 Sqlite 等。这种设计可以使得在切换数据库的时候只需要修改配置文件即可。这为应用开发者在设计数据库时提供了很好的灵活性。

（3）简洁的 URL 设计。Django 框架中的 URL 系统设计非常强大和灵活。可以在 Web 应用中为 URL 设计匹配模式，并用 Python 函数处理。这种设计使得 Web 应用的开发者可以创建与用户友好的 URL，同时对搜索引擎也是友好的。

（4）自动化的管理界面。在 Django 框架中已经提供了一个易用的管理界面。这个界面可以很方便的管理用户数据，具有高度的灵活性和可配置性。

（5）强大的开发环境。在 Django 中提供了强大的 Web 开发环境。其中提供有一个可供开发和测试使用的轻量级 Web 服务器。当启用调试模式后，Django 会显示大量的调试信息。这将会使得消除 BUG 非常容易。

Django 框架的发展非常迅速，已经成为中小网站迅速构建的首选框架。关于 Django 框架的更多内容，将在本章的 19. 3 节至 19. 5 节进行详细介绍。

19. 1. 4　其他的 Web 开发框架

除了上面介绍的一些 Web 开发框架外，还有一些其他的 Web 开发框架。虽然使用这些开发框架的人数不如使用前面介绍的那几种的人多，但是也有着各自的特点。

1. Pylons

Pylons 是一个比较新的结构化和灵活的轻量级 Web 应用开发框架。Pylons 不像 TurboGears 框架那样由各个 Python 模块组成，也不像 Django 框架那样由集成的组件构成。在设计 Pylons 框架的时候，设计者借鉴了来自于 Ruby、Python 和 Perl 世界的先进理念，试图能够组成 Web 应用的最佳组成方式。在使用 Pylons 框架的时候，Web 应用开发者可以集中精力专注于应用的逻辑处理，而不是将精力花费在如何应用框架组件上面。借助于 Python 语言强大的功能，Pylons 具有强大的开发特性，并在易用性上取得平衡。同时 Pylons 并不隐藏框架后台运行的细节，这使它具有足够的灵活性。同时，Pylons 是一个开源项目，可以在需要的时候得到来自于开源世界的支持。其框架项目主页是 http://pylonshq. com/。

2. Karrigell

Karrigell 也是一个灵活的 Web 开发框架。Karrigell 具有简单和灵活的语法，独立于任何数据库、ORM 和模板引擎，这使得开发者可以选择特定的编码样式。在 Karrigell 框架的开发包中有内置的 Web 服务器，并且有一个纯 Python 的数据库引擎。当然，也可以使用外部的 Web 服务器和数据库。Karrigell 框架可以灵活的使用 Python 和 HTML 来构建 Web 应用。其框架项目主页是 http://karrigell. sourceforge. net/。

更多资源请访问稀酷客(www.ckook.com)

3. web. py

web. py 框架是一种超轻量级的 Web 开发框架，相当的小巧。虽然 web. py 框架比较小巧，其功能特性一点都没有损失，而且使用起来更加简单和直接。在使用 web. py 网络框架的时候，可以很好的观察到 Web 应用的底层通信。这可以使有经验的 Web 开发者能够比较快速的发现网络故障。web. py 框架项目主页是 http://webpy. org/。

4. Uliweb

Uliweb 是由 Limodou 发起并创建的新的 Web 开发框架。这是一个国人开发的优秀的 Web 开发框架。该框架充分借鉴了已有 Web 开发框架的优点，同时使得该框架尽可能的简单，能够轻松上手和使用。Uliweb 框架并不是完全重新开发的，利用了已有的一些 Python 库。同时，对于现有框架中不足的部分进行了重写。Uliweb 框架项目主页是 http://uliwebproject. appspot. com/。

还有更多的 Python Web 网络框架没有在这里介绍，包括 CherryPy、Quixote、web2py 和 Paste 等。具体的使用方法可以参考项目自带的帮助文档。

19. 1. 5　各种开发框架的选择

在上面介绍的众多 Web 开发框架中，都有着各自的特性和适用范围。开发者需要根据项目的需要和以后可能的扩展来选择 Web 开发框架。

如果只是想简单迅速的开发 Web 应用，可以选择比较轻量级的 Web 开发框架，如 web. py 或者 karrigell 等。通过使用这些网络框架，可以使用简单的语法迅速的完成简单的网络应用。大部分的网络框架都内置了 Web 服务器，使得开发者可以迅速的开发和测试生成的网络应用，但是当需求增加的时候，这种开发框架的特性就受限了。因为其中很多的功能组件都需要开发者自己来实现，这给使用这种类型网络框架的 Web 应用的扩展带来了一些困难。

如果网络应用所需的功能特别复杂，可以尝试使用 Zope。因为 Zope 提供了 Web 开发所需的大部分功能组件。丰富的功能组件使得 Zope 可以满足 Web 应用中的大部分需求，但是，正是由于它能够满足大部分的功能需求，从而导致这种重量级的网络框架难于掌握。由于其功能强大，内容繁芜，学习和掌握都需要大量的时间和长期的投入。

在轻量级和重量级的网络框架之间取得平衡的是如 TurboGears 和 Django 这样的框架。TurboGears使用组合的模块来构建网络应用，而 Django 则使用集成的模块来构建网络应用。两者都提供了相对比较丰富的功能组件，包括 ORM、模板和自动化功能等。这些开发框架能够相对快速的构建功能比较复杂的网站。相对来说，Django 框架要比 TurboGears 框架流行一些。这主要得益于 Django 框架的良好整体设计。包括自动化的管理接口、出色的模板系统和良好的 URL 设计等。

虽然前面已经给出了一些选择 Web 开发框架使用的准则，但是只有根据项目的需求选择 Web 开发框架，才是最合适的。

19. 2　MVC 模式

MVC（Model-view-controller）是一种在软件工程中广泛使用的设计模式。特别适合于 GUI 设计和 Web 应用设计，可以方便的修改应用的表示层而不影响业务逻辑，或者是修改底层业务逻辑而不影响其他部分。现在新的 Web 开发框架很多都是采用 MVC 模式设计。

更多资源请访问稀酷客(www.ckook.com)

19.2.1　MVC 模式介绍

MVC 模式将一个应用分为 3 个层次：模型层次、视图层次和控制层次。通过这种分层的方式，设计模式可以将应用的输入处理、界面显示以及控制流程分开。这种处理方式可以使得开发者自由的修改应用的输入处理而不用担心界面显示的不同。也就是说，MVC 模式可以有效地将应用分为不同的模块，从而实现应用的松耦合。

1. 模型层次

在模型层次上，包含了业务逻辑流程和状态。这些模型包含如何处理数据的方法。业务模型的设计是 MVC 模式的设计核心。在实际处理中，业务模型的流程对其他层来说是透明的。模型将接受从视图层次传递过来的数据，并返回最终的处理结果给表示层。很多情况下，模型层次中还包含一个数据持久层。该层包含了对数据库的访问对象和接口，用来屏蔽底层数据库操作方式的不同。可以通过数据抽象对数据进行建模和抽象，在其中只是规定了需要管理的数据模型，而并没有提供如何具体实现的模型。使用这种方式可以提高模型的重用性。

2. 视图层次

视图层次包含着最终用户的操作界面，包括输入数据和输出显示等。对于具体的应用，可以有多个视图显示。如对一个 Web 应用来说，可以显示为 HTML 界面，同时也可以显示为XHTML 和 XML 界面，甚至是 TXT 或者 PDF 输出等。这种设计的好处是可以有效地减少 Web 应用的复杂性，从而可以快速的构建多个不同的视图。在视图层次上只是规定了业务数据的获取和输出，而不会在这个层次上来做这些数据的具体处理业务。具体的处理将会交给模型层次来处理。

3. 控制层次

控制层次可以实现对应用的具体控制。控制层次可以将模型层次和视图层次结合起来，共同完成特定的应用请求。控制层次的作用可以看作是一个分发器，只是将具体的数据转发给特定的业务逻辑处理，并将处理后的数据输出。Web 应用的开发者可以选择合适的业务模型和合适的视图。实际上，在控制层次上并不会做任何的数据处理操作。例如，Web 用户选择一个 URL 后，当控制层收到请求，将会转发给合适的模型来处理。在特定的模型处理完毕后，可以通过合适的视图显示出来。由于有控制层次的存在，所以一个模型可以对应多个视图。同样地，一个视图也可以对应多个模型。

Web 应用中常见的 MVC 设计如图 19-2 所示。

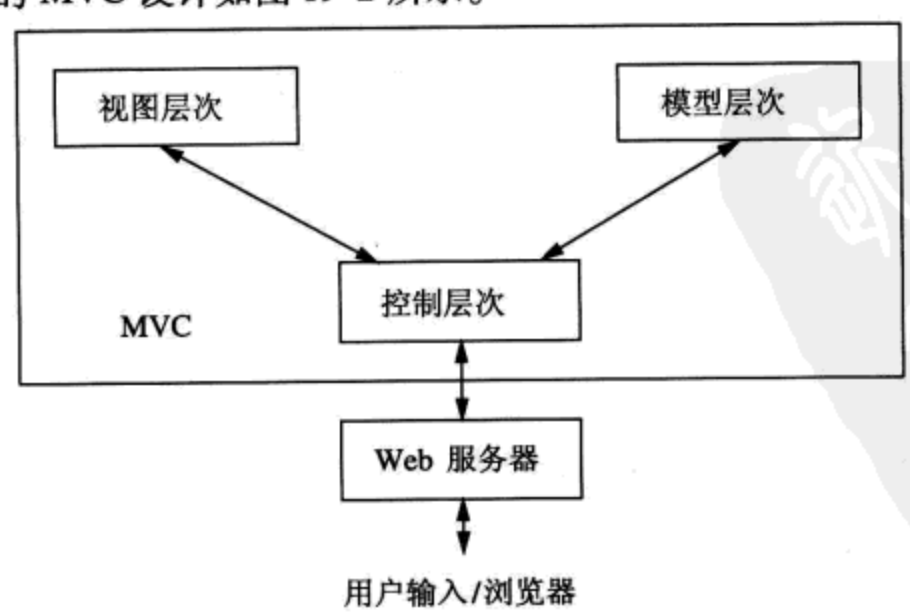


图 19-2　Web 应用的 MVC 模式

更多资源请访问稀酷客(www.ckook.com)

19.2.2　MVC 模式的优点和缺点

MVC 的分层方法可以将业务逻辑和视图显示分开，从而实现简化处理加速开发的目的。这样的设计使得最终的应用结构清晰，扩展性强。

使用 MVC 模式可以有效的构建 Web 应用框架。由于这种模块化的构成，可以使得多个视图对应到一个模型。这种做法的一个最大的好处是可以迅速实现用户的需求。例如在数据模型中可能有很多相似类型的数据，可能需要输出为 HTML 显示，或者是 XML 显示，但是对于数据的处理则可能是类似的。如果遵循 MVC 模式设计的话，则可以使用一个数据模型来实现。通过这种设计，可以减少代码的维护量。当页面输出变化的时候，不需要修改数据模型。

这种分层的模式可以使得其中一个层次的改变只是需要在本层次修改就可以了。例如当需要修改业务的逻辑处理的时候，只需要在模型层次上修改即可。另外，由于在模型层次上输出的数据并不包含任何显示格式，这可以直接通过接口来作用于视图层次。

控制层次可以将不同的模型和不同的视图组合在一起来完成业务逻辑处理。例如，在 Web 应用中，一般都需要处理用户请求。控制层次可以将不同的用户请求传递到特定的数据模型中，最后输出为不同的视图。另外，MVC 模式有利于代码的管理。可以将代码分成多个部分来单独完成，只需要在各自的模块中定义好接口就可以了。

虽然 MVC 模式有许多优点，但是在实际应用中，因为要对模型和视图进行合理的划分不是一件容易的事情。同时，严格的分离会使得应用的调试比较困难，所以还需要进行比较详细的设计。最重要的是，在应用 MVC 模式进行设计的时候，需要有这种将业务逻辑和显示分离的思想。在实际设计的时候，考虑到模块组合的复杂性，可以使得应用更加具有扩展性。

19.2.3　Django 框架中的 MVC

Django 作为一个流行的基于 Python 的 Web 开发框架，也是支持 MVC 模式的。在 Django 框架中，当 URL 被请求的时候，将会调用指定的 Python 方法。通过业务逻辑处理后，将会通过模板来呈现页面。在 Django 的设计小组中，将这种实现方式称为 MVT（Model-view-template）框架。原始 MVC 模式中的控制层次内置在 Django 中来实现。

Django 在 django. db. model. Model 中实现了在网站设计中需要使用的数据模型。在数据模型中定义了保存在数据库中的各种对象属性。通过继承自 Model 类生成的对象，同时可以通过添加 Field 来为特定的数据增加方法。在 Django 的数据模型中，提供了丰富的访问数据对象接口。数据模型中的数据将会同步到后台的数据库。而 Django 则提供了一个良好的 ORM 来使得开发者可以从视图和模板中访问数据库中的数据。

在视图层次上，Django 框架实现了良好的 URL 设计和处理。当收到 URL 请求的时候，Django 将会使用一组已经预定义好的 URL 模式，匹配到合适的处理器。实际上，URL 的设计也是网站视图层次上的设计。这决定了 Web 应用如何读取 URL 请求以及如何显示网页。对于每个特定的 URL，Django 都会有一个特定的视图函数来进行处理。可以看到，Django 框架的视图处理是分成多个步骤的。其框架在收到 URL 请求后，通过页面函数来处理，最后将页面响应返回给浏览器显示。

在 Django 框架中，还提供了强大的模板解析功能，通过页面函数处理来输出页面响应。模板系统使得 Web 应用的开发者集中在需要展现的数据上。这种开发方式可以使页面设计者特别关注于输出网页的构成。

Django 框架良好的 MVC 模式设计使得 Django 成为一个流行的 Web 开发框架。接下来，将具体的介绍 Django 的使用方法。

更多资源请访问稀酷客(www.ckook.com)

19.3　Django 开发环境的搭建

在使用 Django 框架之前，首先要搭建 Django 开发环境。开发环境的搭建，首先需要下载 Django 的软件包并安装。在安装完毕后还需要设置数据库环境。

19.3.1　Django 框架的安装

由于 Python 语言可以跨平台，因此 Django 可以很方便的安装在包括 Windows 和 Linux 等系统平台上。因为 Django 框架是使用 Python 语言开发的，且框架中已经实现了 Web 开发所需的组件，所以安装 Django 框架的基本条件是系统中已安装了 Python。

Django 项目的主页为 http://www.djangoproject.com/。当前最新的版本号为 1.0.2。最常见的安装方式是在其主页下载源码文件并安装。这种方式对于 Windows 和 Linux 平台都是适合的。要注意的是，在安装 Django 之前需要保证系统中已经安装了 Python。

（1）从 http://www.djangoproject.com/download 上下载 Django 源码包。

（2）将下载的源码包解压：

```
tar zxvf Django-1.0.2.tar.gz
```

（3）进入到刚刚解压后的 Django 目录：

```
cd Django-1.0.2
```

（4）安装 Django。如果安装在 Linux 系统下，则需要具有安装的权限：

```
python setup.py install
```

对于特定的系统平台，可以安装针对特定平台的软件包。如在 Ubuntu 和 Debian 等发行版的 Linux 中，可以使用 apt 程序来安装：

```
apt-get install python-django
```

当前的 Django 版本已经可以用于实际的项目中，但是，由于现在 Django 项目还处于持续的开发中，如果需要使用一些新的特性，则需要安装 Django 的开发版本。可以使用如下的方式来获取开发版本，并按照上面源码的安装方式安装：

```
svn co http://code.djangoproject.com/svn/django/trunk/
```

其中，svn 为版本管理工具 Subversion 的命令工具。后面的 URL 地址为其开发版本的下载地址。可以从其主页地址 http://subversion.tigris.org/上下载。

安装完 Django 框架后，可以通过如下的方式来测试安装是否成功：

```
In [1]: import django

In [2]: print django.VERSION
(1, 0, 2, 'final', 0)
```

- 在 In[1]中导入了 django 模块。如果 Django 软件包安装成功，则此语句将运行成功，否则表示软件包安装不成功。
- 在 In[2]中输出了 Django 框架中的 VERSION，表示当前框架的版本号。

19.3.2　数据库的配置

Django 框架的唯一需求是 Python，所以数据库在 Django 的 Web 开发中并不是必需的，但是

更多资源请访问稀酷客(www.ckook.com)

在实际的网站设计中，大部分的数据还是保存在数据库之中。而 Django 已经支持多种数据库，包括 MySQL、PostgreSQL 和 Sqlite 等。Django 有着一个设计良好的 ORM，这种方式使得其可以有效的屏蔽底层数据库的不同。

对于每个 Django 应用，其目录中都有一个 setttings. py 文件，用来实现对数据库的配置。setting. py文件是一个 Python 脚本文件，在其中可以设置 Django 项目的属性。每个 Django 项目都有特定的配置文件。

在 setting. py 文件中，可以通过设置下面的属性值来设置 Django 对数据库的访问。

（1）DATABASE_ENGINE：此值用来设置数据库引擎的类型，可以设置的类型有 sqlite3、mysql、postgresql 和 ado_mssql 等。

（2）DATABASE_NAME：此值用来设置数据库的名字。如果数据库引擎使用的是 SQLite，则需要指定全路径。

（3）DATABASE_USER：此值用来指定连接数据库时候的用户名。当数据库引擎使用 SQLite 的时候，不需要设置此值。

（4）DATABASE_PASSWORD：此值用来指定用户 DATABASE_USER 的密码。当数据库引擎使用 SQLite 的时候，不需要设置此值。

（5）DATABASE_HOST：此值用来指定数据库所在的主机。当此值为空的时候，表示数据将保存在本机中。当数据库引擎使用 SQLite 的时候，不需要设置此值。

（6）DATABASE_PORT：此值用来设置连接数据库时使用的端口号。当留空的时候将使用默认端口。当数据库引擎使用 SQLite 的时候，不需要设置此值。

19.4　Django 框架的应用

Django 的开发环境配置完毕后，就可以开发 Web 应用了。Django 框架提供了一种迅速的方法来创建功能丰富的 Web 应用。本节将描述一个 Web 应用创建的完整过程。

19.4.1　Web 应用的创建

Django 框架中的一个网站可以包含多个 Django 项目。而一个 Django 项目则包含一组特定的对象。这些对象包括 URL 的设计、数据库的设计以及其他的一些选项设置。Django 框架提供了实用工具用来对 Web 应用进行管理，这就是 django-admin. py。当 Django 软件包安装完毕后，在 scripts 目录中将会包含 django-admin. py 文件。另外，如果是在 Linux 下使用安装包的方式安装，将会创建 django-admin 的链接。

django-admin. py 中有许多的命令选项，可以通过这些选项来操作项目。使用 help 选项可以查看 django-admin. py 命令中支持的命令选项：

```
In [1]: ! django-admin.py help
Usage: django-admin.py subcommand [options] [args]
Options:
#省略部分输出结果
  --traceback           Print traceback on exception
  --version             show program's version number and exit
  -h, --help            show this help message and exit
Type 'django-admin.py help <subcommand>' for help on a specific subcommand.
Available subcommands:
  cleanup
  compilemessages
  createcachetable
  dbshell
```

更多资源请访问稀酷客(www.ckook.com)

```
  diffsettings
  dumpdata
  flush
#省略部分输出结果
  startapp
  startproject
  syncdb
  test
  testserver
  validate
```

从输出结果中可以看出，django-admin. py 命令有大量的命令选项用来管理 Django 项目。如果需要查看特定命令选项的功能，可以使用“django-admin. py help command”来完成：

```
In [2]: ! django-admin. py help startproject
Usage: django-admin. py startproject [options] [projectname]

Creates a Django project directory structure for the given project name in the
current directory.
#省略部分输出结果
```

可以通过 startproject 来迅速的创建项目。在下面的操作中，将生成一个 blog 的最初应用：

```
In [3]: ! django-admin. py startproject blog
```

在 In［3］中，Django 框架将在当前目录下，使用 startproject 命令选项生成一个项目名为 blog 的 Web 应用，如图 19-3 所示。

从图 19-3 中可以看出，在使用了 startproject 命令选项后，Django 框架生成了一个 blog 的目录。其中有一个基本 Web 应用所需要的文件集合。下面是这些文件的介绍。

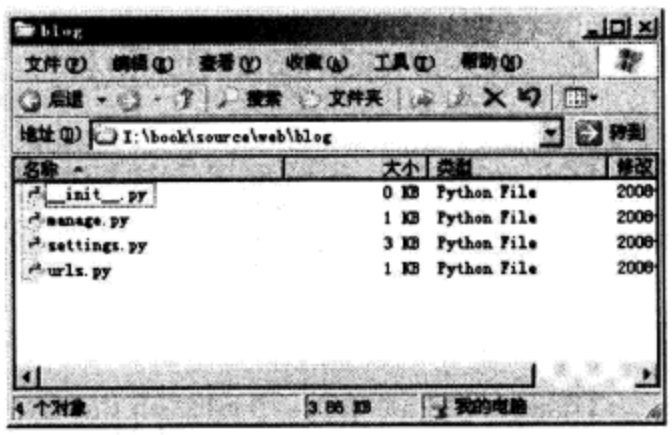


图 19-3　Django 框架生成的项目

（1）__init __. py：这是一个空文件，其用途主要用来指示 Python 语言将此网站目录当作是 Python 的包。

（2）manage. py：此文件中主要可以使得网站管理员来管理 Django 项目。

（3）settings. py：这是此 Django 项目的配置文件。

（4）urls. py：此文件中包含 URL 的配置文件，这也是用户访问 Django 应用的方式。

这些文件中仅仅只是包含一个最简单的 Web 应用所需的代码。当应用变得复杂的时候，将会对这些代码进行扩充。

需要注意的是，由于 Django 的项目是作为 Python 的包来处理的，所以在项目命名的时候尽量不要和已有的 Python 模块中的名字冲突。否则在实际使用的时候有可能出错。另外，尽量不要将网站的代码放在 Web 服务器的根目录下。这可能会带来安全问题。

19. 4. 2　Django 中的开发服务器

在 Django 框架中包含了一个轻量级的 Web 应用服务器，可以在开发的时候使用。在 Web 应用形成产品之前，不需要对其进行配置，如针对 Apache 的设置等。该内置的服务器可以在代码修改的时候自动加载，从而实现网站的迅速开发。

切换到在 In［4］中生成的 blog 文件夹，可以使用下面的方法来启动内置的服务器：

更多资源请访问稀酷客(www.ckook.com)

```
In [4]: ! python manage.py runserver
Validating models...
0 errors found

Django version 1.0.2 final, using settings 'blog.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

在启动内置的 Web 应用服务器的时候，Django 会检查配置的正确性。如果配置正确，将使用 setting.py 文件中的配置启动此开发服务器。

启动浏览器，连接此 Web 服务器，可以显示 Django 项目的初始化页面。如图 19-4 所示。

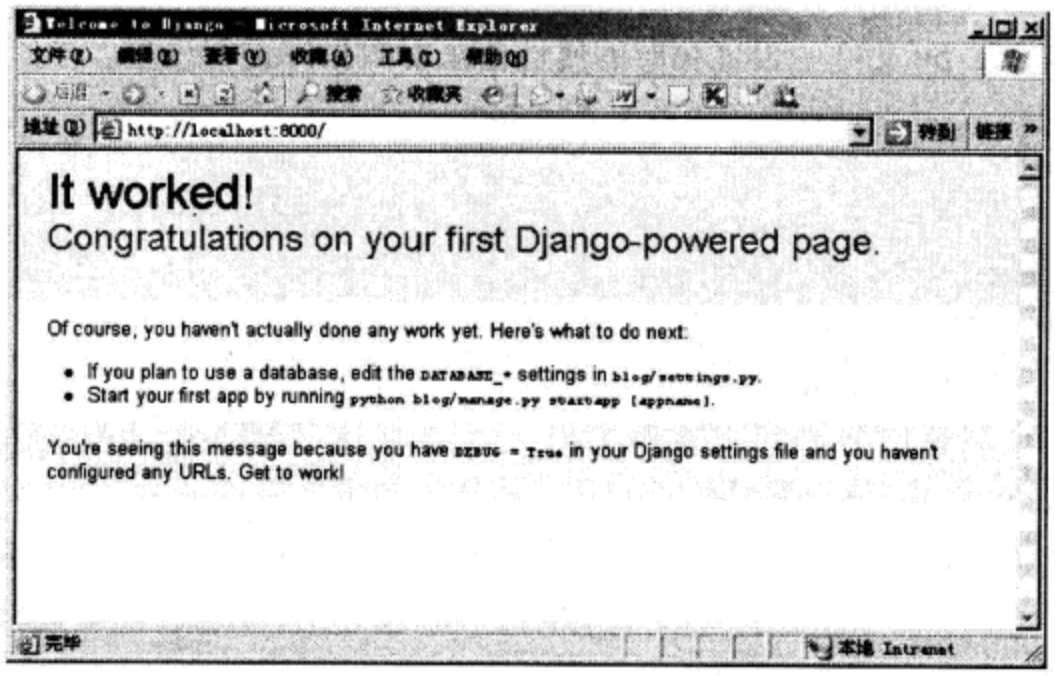


图 19-4　Django 项目的初始化页面

从图 19-4 中可以看出，Django 已经正确安装并且已经生成了一个项目。在这个指示性页面中，还介绍了下一步的操作。在连接此服务器的时候，在控制台下还会显示如下的信息：

```
[05/Dec/2008 13:57:12] "GET / HTTP/1.1" 200 2049
```

- 这个输出信息中显示了连接的时间以及响应信息。在输出响应中，显示出 HTTP 的状态码为 200，表示此连接已经成功。

在默认情况下，使用“python manage.py runserver”将会默认在本机的 8000 端口监听。当 8000 端口被占用而需要使用其他端口的时候，可以使用下面的命令来监听其他端口。

```
In [5]: ! python manage.py runserver 8001
```

- In［5］中将对本机的 8001 端口进行监听。

在上面的命令中，都只是在本机进行监听。也就是说，Django 只是接受来自本机的连接。在多人开发 Django 项目的情况下，可能需要从其他主机来访问 Web 服务器。此时，可以使用下面的命令来接受来自其他主机的请求：

```
In [6]: ! python manage.py runserver 0.0.0.0:8000
```

- In［6］中将对本机的所有网络接口监听 8000 端口。这样可以满足多人合作开发和测试 Django 项目的需求。这样就可以在其他主机来访问此 Web 服务器了。

如果要中断此服务器，按 Ctrl + C 或者 Ctrl + Break 键即可。

更多资源请访问稀酷客(www.ckook.com)

19.4.3　创建数据库

在 19.3.2 小节中已经介绍了如何在 Django 项目中配置和使用数据库。对于网站的开发者来说，可以选取适合的数据库。SQLite 作为一种轻量级嵌入型的数据库引擎，有着其他数据库所不具备的优点。虽然这只是一个轻量级的数据库引擎，但是已经支持大多数的 SQL 语法。SQLite 由于去除了数据库设计中的复杂部分，这使得其具有高效性。SQLite 利用高效的内存组织，将数据库中的数据保存在文件中。这种处理方式使得最后数据的尺寸比其他数据库系统都要小。由于 SQLite 使用方便和无需配置的特性，所以在本章下面的讲述过程中，将使用 SQLite 数据库引擎。

可以通过在 setting. py 文件中设置相应的属性值来对数据库进行设置：

```
    DATABASE_ENGINE = 'sqlite3'           #'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or
'oracle'.
    DATABASE_NAME = '. /blog. db3'       # Or path to database file if using sqlite3.
    DATABASE_USER = ''                  # Not used with sqlite3.
    DATABASE_PASSWORD = ''             # Not used with sqlite3.
    DATABASE_HOST = ''                  # Set to empty string for localhost. Not used with sqlite3.
    DATABASE_PORT = ''                  # Set to empty string for default. Not used with sqlite3.
```

- 设置 SQLite 数据库只需要设置两个值。一个是 DATABASE_ENGINE，用来指定使用 sqlite3，另一个是 DATABASE_NAME，这里为数据库文件 blog. db3。

在上面只是设置了数据库，但是并没有生成数据库。这可以通过使用 mange. py 中的子命令 syncdb 来实现：

```
In [7]: ! python manage.py syncdb
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): y
Please enter either "yes" or "no": yes
Username: admin
E-mail address: admin@ server. com
Password:
Password (again):
Superuser created successfully.
Installing index for auth. Permission model
Installing index for auth. Message model
```

- 在 In［7］中使用了 manage. py 的 syncdb 子命令，其实这也可以通过 django-admin. py来实现。此子命令用来创建配置文件 INSTALLED_APPS 域中还没有创建的表。在输出结果的前半部分，在数据库中创建了特定的应用。这从配置文件中可以看出来：

```
INSTALLED_APPS = (
    'django. contrib. auth',
    'django. contrib. contenttypes',
    'django. contrib. sessions',
    'django. contrib. sites',
)
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［7］输出结果的后半部分，Django 为项目生成了一个管理用户。在接下来的命令行中提示了此管理用户的相关信息。

在执行了这个子命令之后，在 blog. db3 文件中将保存生成的数据库表。使用 SQLite 工具可以看到这个结果，如图 19-5 所示。

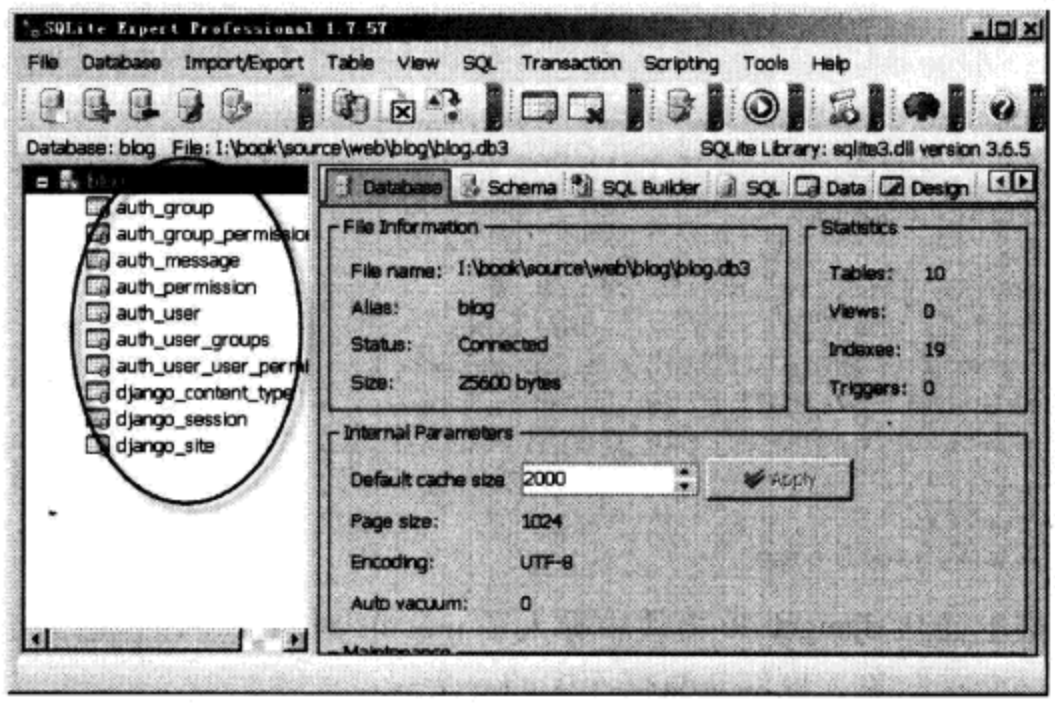


图 19-5　Django 中创建的数据库

19.4.4　生成 Django 应用

一个 Django 的网站可能会有多个 Django 应用。可以使用 manage. py 的 startapp 子命令来生成 Django 应用。一个应用中可以包含一个数据模型以及相关的处理逻辑：

```
[8]: ! python manage.py startapp Account
```

在使用了 startapp 子命令后，将会生成如图 19-6 所示的目录结构。

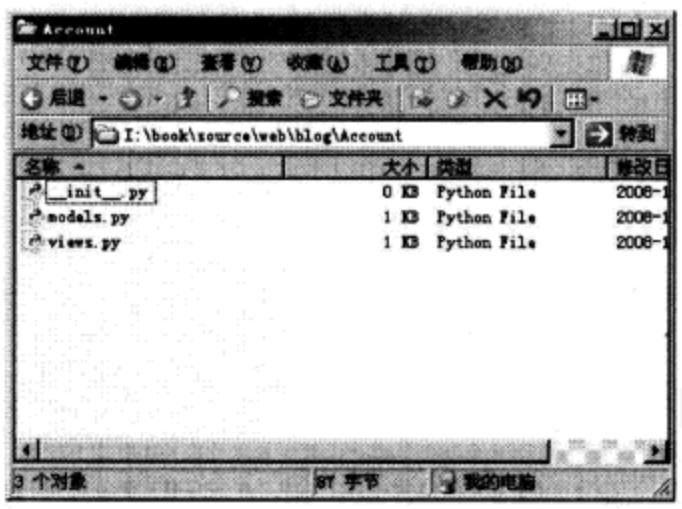


图 19-6　Django 框架创建的应用

从图 19-6 中可以看出，在 blog 目录下生成了一个 Account 目录。此目录中的文件信息定义了应用的数据模型信息以及处理方式。其中有如下的 3 个文件。

（1）__init __. py：这是一个空文件，在这里是必需的。用来将整个应用作为一个 Python 模块加载。

（2）models. py：此文件中定义了数据模型相关的信息。

（3）views. py：此文件中包含此模型的视图相关操作。

更多资源请访问稀酷客(www.ckook.com)

19.4.5　创建数据模型

创建 Django 的应用后，需要定义保存在数据库中的数据。实际上，数据模型就是一组相关对象的定义，包括类、属性和对象之间的关系等。为了创建数据模型，可以通过修改 models.py 文件来实现。此文件是一个 Python 脚本文件。其中定义了将要保存到数据库中的表。在下面的代码中，定义了一个 Account 表：

```
#encoding = utf-8

from django.db import models

class Account(models.Model):#生成一个 Account 类
    username = models.CharField('用户名', max_length = 30)
    password = models.CharField('密码', max_length = 30)
    email = models.EmailField('电子邮箱', blank = True)
    desc = models.TextField('描述', max_length = 500, blank = True)

    def __str__(self):
        return '% s' % (self.name)
```

- 在代码的最上面从 django.db 模块中导入了 models 对象。可以在后面定义多个类，每个类都表示一个类对象，也就是数据库中的一个表。
- 接下来，定义了一个 Account 类。此类继承自 models 中的 Model 类。在 Account 类的主体部分，定义了 4 个域用来描述个人的相关信息。这里有个人账号的名字、密码、邮箱以及描述。在函数主体的第 1 行是名字的数据模型信息。可以看到，这里使用了 models 中的 Charfield 域，表示此对象为字符域。其构造函数中使用了两个参数，包括在数据库中保存的域名和最大长度限制。接下来的 3 个域也是类似的。只是其中采用了不同的数据模型，如 EmailField 和 TextField 等。更多的模型可以参看 Django 的文档。
- 在其后使用了 __str__方法用来描述类。这里使用了其 name 属性。

在创建了数据模型后，可以在 setting.py 文件中加入此应用：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'blog.Account',
)
```

- 在 INSTALLED_APPS 的最后面加入了 blog.Account 值，用来将刚刚生成的应用加入到整个 Django 项目中。

将此应用加入到项目中后，可以继续使用 syncdb 在数据库中生成未创建的数据模型：

```
In [9]: ! python manage.py syncdb
Creating table Account_account
```

- 在 In［9］中可以看到，这里生成了一个名为 Account_account 的表。通过 SQLite 工具查看 blog.db3 文件，可以看到在其中已经生成了相应的数据表，并且有了对应的域。

19.4.6　URL 设计

在 Django 项目中，URL 的设计是定义在配置文件的 ROOT_URLCONF 属性值中。在此属性

更多资源请访问稀酷客(www.ckook.com)

值中定义了收到 URL 后的处理方式。实际上，URL 的配置文件也是一个 Python 脚本文件，可以在此文件中设置访问的 URL。

当 Django 服务器启动后，对于收到的每个 URL 请求，Django 会将此 URL 请求分解，得到相关的 URL 部分，并将此 URL 结果和 URL 配置文件中的设计进行匹配。在每次请求的时候，Django 的开发服务器将会打印出此请求的相关信息。其中第二部分为请求的 URL 信息。例如在下面的 URL 请求中，所请求的 URL 地址为“/url/sub-url/”。Django 将会在 URL 的配置文件中匹配此请求 URL，并调用相应的方法：

```
[05/Dec/2008 19:43:17] "GET /url/sub-url/ HTTP/1.1" 404 1946
```

在 ROOT_URLCONF 属性值中可以设置此 URL 的信息。在这里的值是“blog. urls”，也就是说 URL 的信息定义在根目录下的 urls. py 文件中：

```
ROOT_URLCONF = 'blog.urls'
```

有两种方式来定义 URL。一种是直接定义特定 URL 的处理方式，另外一种是递归使用子目录中的 URL 配置文件。下面的例子演示了这两种方法：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'blog.views.index'),
    (r'^account/', include('blog.Account.urls')),
)
```

- 在 URL 的配置文件中，一般在最上面使用语句“from django. conf. urls. defaults import *”。此语句提供了 URL 配置的情况。如 patterns 用来指定访问前缀模式。
- 在接下来有两个 URL 配置信息。每一项都有两个部分，第一部分为 URL 的正则表达式匹配设置，第二部分为 URL 的处理函数。
- 这两个 URL 设计，其中一个是直接使用的 URL 前缀信息。“^$”表示在访问 Web 服务器根目录的时候，接下来的“blog. views. index”则表示此处理函数为根目录下 views. py 文件中的 index 函数。在第二个 URL 匹配中，表示如果 URL 是以“account/”作为前缀，将调用后面的处理函数。这个处理函数使用了 include 对象，用来表示具体的 URL 配置信息可以从 Account 目录下的 urls. py 中查找。

下面是在 Account 目录下的 urls. py 文件信息：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'Account.views.index'),
    (r'^random_number/$', 'Account.views.random_number'),
)
```

这里的 URL 配置文件和上面那个配置文件是类似的。只是这里的 URL 处理将会去掉 account 前缀。

当 Django 收到请求的 URL 后，会将处理后的 URL 针对其配置文件依次匹配。如果有匹配，则调用相应的页面处理函数。如果找不到，则会调用相应的错误信息。如访问前面提到过的“/url/sub-url/”地址，将会出现如图 19-7 所示的结果。

从图 19-7 中可以看出，此页面显示了具体的 URL 匹配信息。

更多资源请访问稀酷客(www.ckook.com)

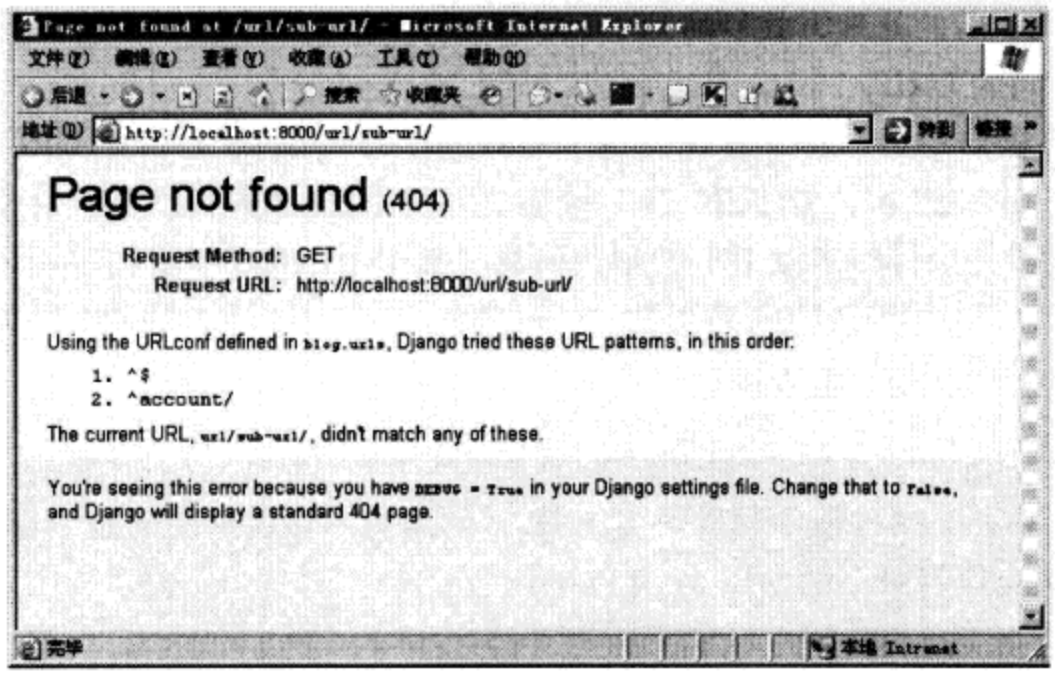


图 19-7　Django 框架中的 404 页面

19.4.7　创建视图

配置了 URL 之后，可以在视图文件中书写特定的视图函数，或者称页面函数。当 Django 收到特定的 URL 后，将会调用特定的页面函数。此页面函数只是 Python 文件中一个特定的函数。下面是根目录下的 urls. py 文件代码：

```
#coding = utf-8

from django. http import HttpResponse

def index(request):
    html = """ <html >
                <title >Main < /title >
                <body >
                    <h1 >Main Page < /h1 > < hr >
                < /body >
            < /html > """
    return HttpResponse(html)
```

- 在第 3 行从“django. http”导入了 HttpResponse 类。后面可以使用此类来构建一个 HTTP的响应。
- 接着定义了一个 index 方法。当访问主页的时候会调用此函数来生成 HTTP 响应。其参数 request 为连接请求的对象，其中包含与请求相关的其他信息，如请求参数等。例如，request. POST 可以用来表示使用 POST 方法请求的参数信息。
- 函数主体中有两个语句，第一个语句中定义了一个 html 变量，其中是 HTML 文档的内容。接下来，使用 HttpResponse 类的构造函数生成了一个 HTTP 响应，其参数即为前面定义的 HTML 文档内容。

在配置 URL 和定义了页面函数后，即可以在浏览器中浏览此 URL 页面，如图 19-8 所示。

在图 19-8 所示页面中可以看出，此页面中显示了在 index 函数中的 HTML 文档内容。

在上面的例子中，仅仅只是显示了静态的页面信息。当然也可以显示动态信息。在 Account 目录下的 views. py 文件中，random_number 函数可以显示多个不同的随机数：

更多资源请访问稀酷客(www.ckook.com)

```
import random
def random_number(request):
    body1 = "Random: %f <br />" % random.random()
    body2 = "Random: %f <br />" % random.random()
    body3 = "Random: %f <br />" % random.random()

    html = "<html><body>" + body1 + body2 + body3 + "</body></html>"
    return HttpResponse(html)
```

此函数定义在 views.py 文件中，首先导入了 random 模块，然后在其 HTML 文档中生成了 3 个随机数。每次请求的时候，输出的内容都不一样。图 19-9 中显示了一种输出结果。

从图 19-9 所示页面中可以看到，输出了不同的内容。

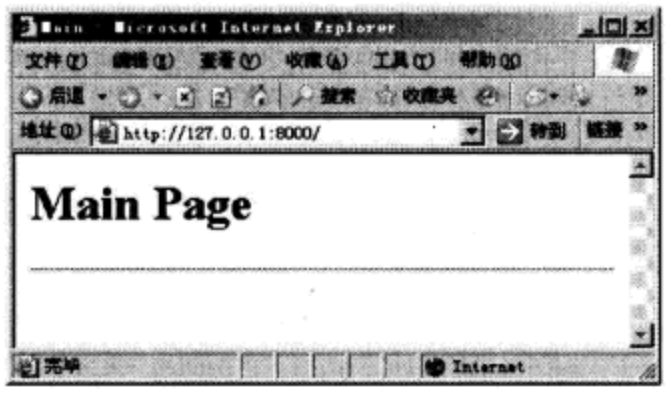


图 19-8　Django 中的页面

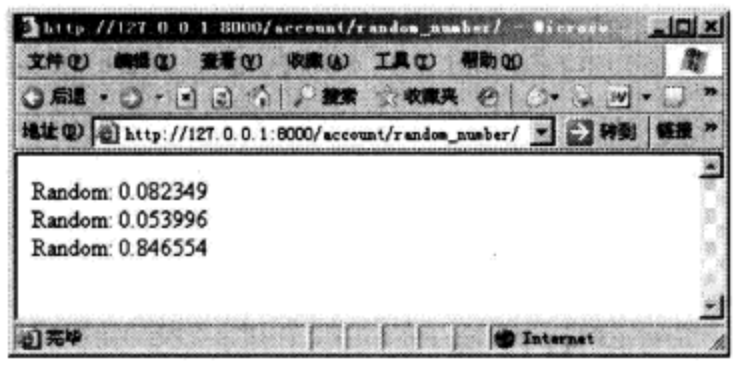


图 19-9　在页面中显示动态内容

19.4.8　模板系统

在已经介绍的视图函数设计中，页面显示和数据没有分离。利用 Django 框架中提供的模板系统，可以有效地分离页面显示和数据。另外，这种模板系统的文件可以重用，从而减少了系统设计的复杂性。

在配置文件 settings.py 中，可以在TEMPLATE_DIRS属性中设置相应的值：

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
'./templates',
)
```

在上面的设置中，将当前目录的 templates 作为模板文件保存。需要注意的是，即使是在Windows系统平台下，在路径中也是需要使用斜杠的。可以将显示分为两部分，一部分是显示，一部分是数据。其中，可以在 templates 目录中加入 list_index.html 文件：

```
<html>
<title>Main</title>

<body>
<h1>Main Page</h1>
<hr/>

{% for blog in entries %}
    <b>{{ blog.title }}</b><br />
    {{ blog.content }}<p />
{% endfor %}

</body>
</html>
```

这里的文件改写了原来的 HTML 文档的内容信息。除了大致框架类似之外，其中还使用了模板标签。

更多资源请访问稀酷客(www.ckook.com)

这里使用的模板标签是 for，这将会遍历集合中的所有对象，随后显示了此对象的title和 content 属性。

修改根目录下的 views. py 文件，如下所示：

```
#coding = utf-8

from django. shortcuts import render_to_response

entries = [
    {'title':'Python Portability', 'content':'Python runs on Windows, Linux/UNIX, Mac OS X, OS/2,
Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to the Java and .NET
virtual machines.'},
    {'title':'Python Programming Language', 'content':'Python is a dynamic object-oriented
programming language that can be used for many kinds of software development. It offers strong
support for integration with other languages and tools, comes with extensive standard libraries, and
can be learned in a few days. Many Python programmers report substantial productivity gains and feel
the language encourages the development of higher quality, more maintainable code.'}
]

def index(request):
    return render_to_response('list_index. html', {'entries': entries})
```

- 在代码的最前面从 django. shortcuts 模块中导入了 render_to_response 方法。这将在后面生成 HTTP 响应。
- 接下来定义了 entries 变量，这是一个列表。列表中的每项都是一个字典数据。这里包含文档的标题和内容。
- 在改写的 index 函数中，使用了 render_to_response 函数。这里有两个参数，其中第一个是模板的文件，第二个是字典数据信息，用来实例化模板文件中的占位符。

通过使用模板文件，显示了如图 19-10 所示的页面。

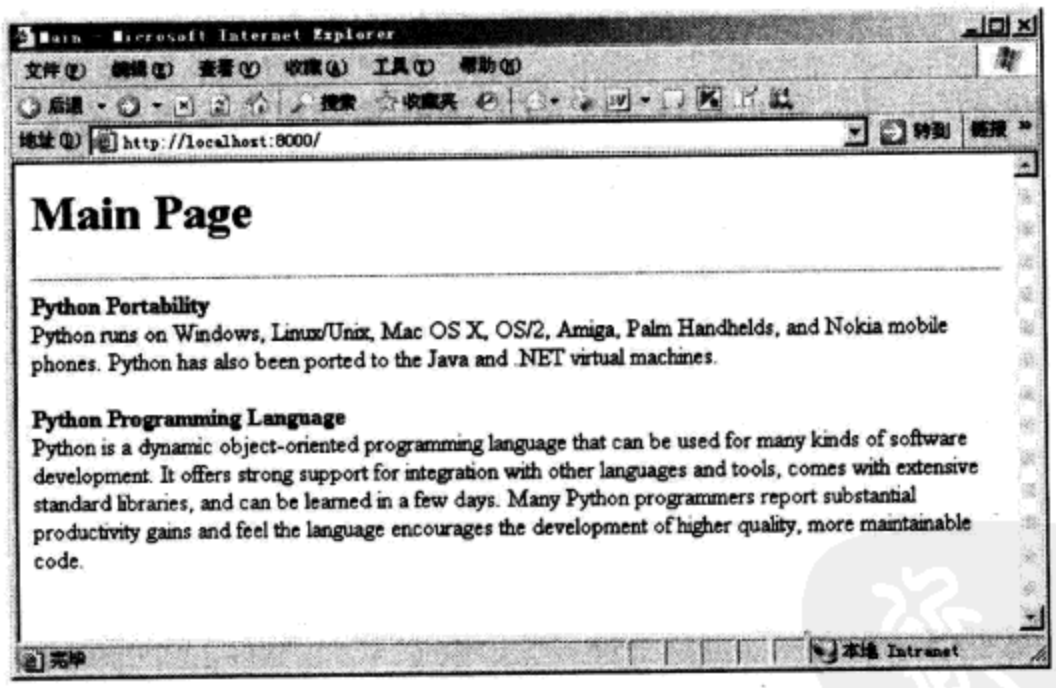


图 19-10　模板文件的显示

19. 4. 9　发布 Django 项目

虽然 Django 已经内置了开发服务器用来测试和快速开发，但是不适合于产品环境。因为其内置的服务器并不是一个全功能的 Web 服务器。本小节将主要介绍如何在 Apache 服务器中使用 mod_python 来发布 Django 应用。

Mod_python 是 Apache 的一个模块，可以使服务器能解析 Python 文件。mod_python 模块可以使用 Python 来编写基于 Web 的应用，其速度也要快于传统的 CGI 处理。mod_python 模块还具

更多资源请访问稀酷客(www.ckook.com)

有其他的高级特性，包括访问 Apache 内部数据等。其主页地址为 http://www.modpython.org/，可以到该网站下载源码安装包。

在 Django 项目的配置文件中，原来的模板目录使用的是相对目录。但是这只能在 Django 自带的测试服务器中使用。另外，Django 项目所在的目录也不应该是在 Web 服务器的根目录下，所以这里需要将其目录改为绝对目录：

```
MEDIA_ROOT = 'I:/book/source/web/blog/media

TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    'I:/book/source/web/blog/template',
)
```

- 这里将原来的相对目录改成了绝对目录。这样处理后，Apache 服务器就可以找到相应的文件了。具体的有 MEDIA_ROOT 和 TEMPLATE_DIRS。

接着，需要 Apache 服务器中的 httpd.conf 配置文件，增加下面的配置信息：

```
LoadModule python_module modules/mod_python.so
```

- 通过这样的处理，Apache 服务器就可以处理 Python 脚本文件了。

最后在 Apache 服务器的配置文件中加入下面的 Django 项目信息：

```
<Location "/">
    SetHandler python-program
    PythonPath "['I:/book/source/web/blog/'] + sys.path"
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE blog.settings
    PythonAutoReload Off
    PythonDebug On
</Location>
```

这样，当 Apache 服务器启动的时候，就可以通过浏览器来访问了。

19.5　Django 框架的高级应用

上一节已经介绍了 Django 框架的基本使用方法，但那只是构建了比较简单的 Web 应用。Django 框架还提供了更多的功能，用来构建丰富的 Web 应用。

19.5.1　管理界面

Django 框架的最大特性之一就是内置有一个很好的管理界面。可以在此管理界面中对项目的数据进行管理，包括新建和删除等功能。这个管理界面的调出只需要对原项目经过一定的改造即可。下面来演示这种过程。

（1）在项目的配置文件 INSTALLED_APPS 属性中加入“django.contrib.admin”：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'blog.Account',
)
```

更多资源请访问稀酷客(www.ckook.com)

（2）调用 manage. py 的 syncdb 子命令。此子命令将会在数据库中创建还未生成的表。如下所示。

```
In [13]: ! python manage. py syncdb
Creating table django_admin_log
Installing index for admin. LogEntry model
```

- 从输出信息中可以看到生成了管理界面所需的表。如果以前没有生成管理员用户的话，则在这里可能需要输入管理员用户的用户名和密码。由于在前面创建表的时候已经创建了用户名和密码，所以这里没有提示输入。

（3）在项目根目录下的 urls. py 文件中，修改必要的部分来增加对管理界面的支持。下面是修改后的 urls. py 文件：

```
from django. conf. urls. defaults import *

# Uncomment the next two lines to enable the admin:
# 启用管理界面修改的第一部分
from django. contrib import admin
admin. autodiscover()

urlpatterns = patterns('',
    (r'^$', 'blog. views. index'),
    (r'^account/', include('blog. Account. urls')),

    # Uncomment the admin/doc line below and add 'django. contrib. admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django. contrib. admindocs. urls')),

# Uncomment the next line to enable the admin:
# 启用管理界面修改的第二部分
    (r'^admin/(. * )', admin. site. root),
)
```

- 这里和原来的 urls. py 文件有两个部分不同。第一个部分从 django. contrib 导入了 admin，并调用了其 autodiscover 方法。在第二部分中，设置了 URL 的匹配设计。
- 在调用 manage. py 的 startproject 命令生成项目的时候，将会在生成的 urls. py 中包含这些代码，只是暂时注释掉了而已。

（4）在 Account 目录下生成一个 admin. py 文件。其内容如下：

```
from django. contrib import admin
from models import Account

admin. site. register(Account)
```

使用了 admin. site 的 register 方法，将此数据模型关联到了管理界面中。

在浏览器中输入“http://127. 0. 0. 1:8000/admin/”，将会出现如图 19-11 所示的页面。在这里可以使用前面创建的管理员用户名和密码来登录。登录后的界面如图 19-12 所示。



图 19-11　Django 的管理界面登录

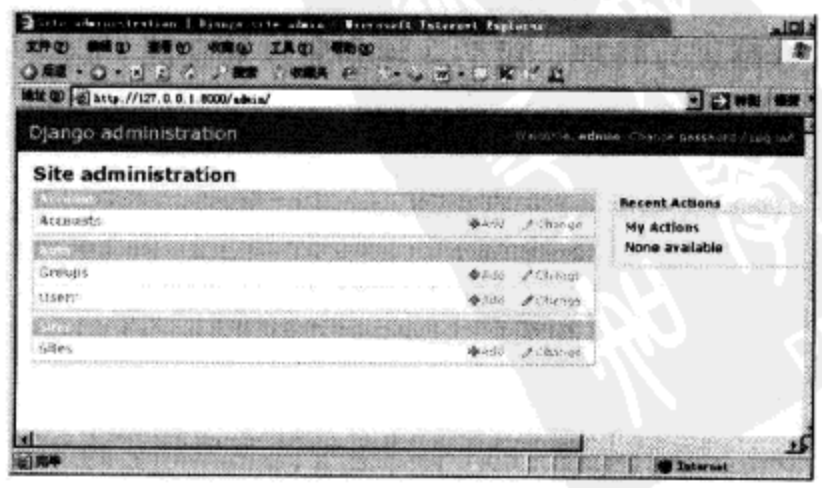


图 19-12　Django 的管理界面内容

更多资源请访问稀酷客(www.ckook.com)

从图 19-12 中可以看出，其中包含前面定义的数据模型。可以通过此管理界面来增加和删除数据。另外还可以看到，这里还有 Django 框架内置的几个数据模型。Django 框架中的管理员用户和密码就是保存在 Users 表中的。

19.5.2　生成数据库数据

在上面一小节中看到，可以通过 Django 框架的管理界面来对数据模型中的数据进行增加和修改，图 19-13 中演示了如何使用 Django 管理界面来增加数据。

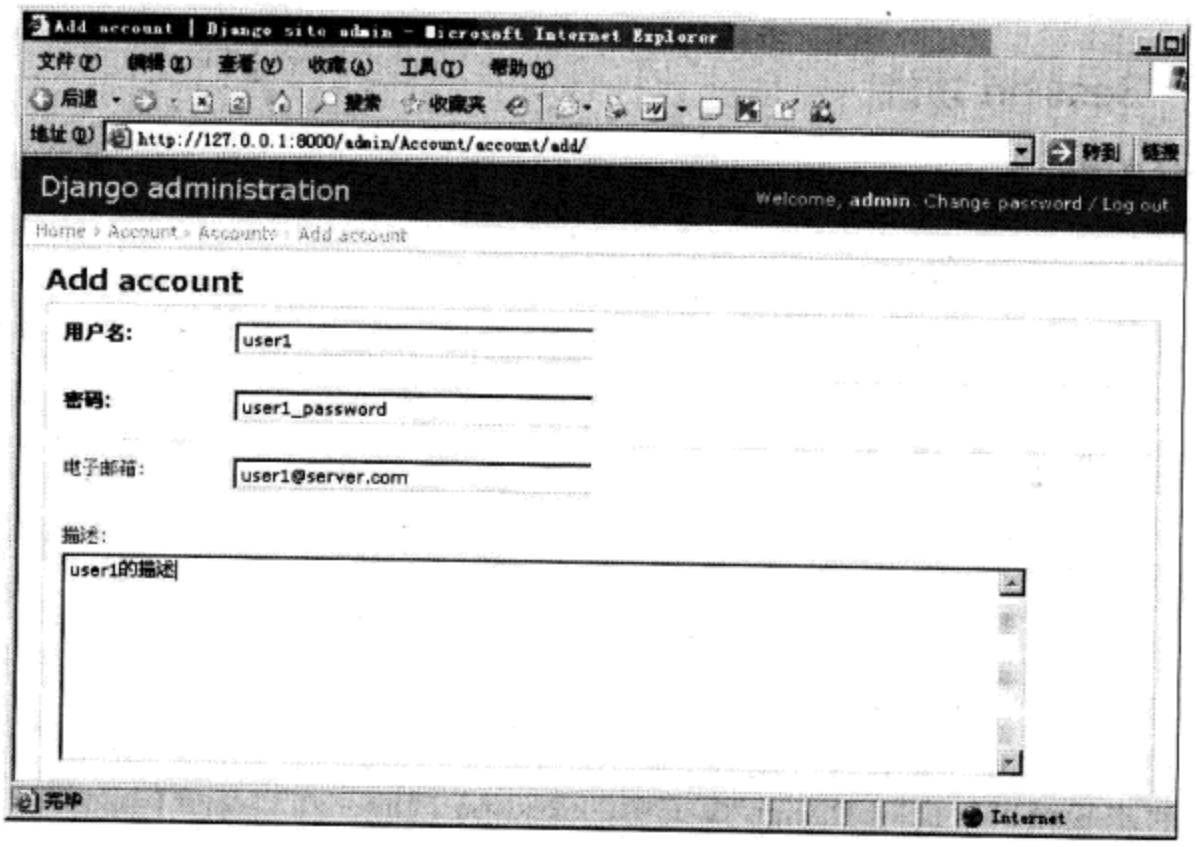


图 19-13　通过 Django 框架的管理界面增加数据

在如图 19-13 所示的窗口中，可以看到 Django 框架提供了生成数据模型数据的一种方法。这种生成方式比较简单和直观。也可以使用 Django 提供的 API 来生成数据。可以直接调用模型数据的构造函数生成一个对象，然后使用其 save 方法将生成的对象保存在数据库中。使用类的 objects 对象的 all 方法可以得到此模型数据中的所有数据。这里，使用了 manage. py 的 shell 子命令生成了一个控制端。当然，在原来的 SHELL 中操作也是可以的，但是需要进行适当的设置：

```
In [15]: ! python manage.py shell
Python 2.5.2 (r252:60911, Feb 21 2008, 13:11:45) [MSC v.1310 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ? object also works, ?? prints more.

In [1]: from Account.models import Account

In [2]: ac = Account(name="user2",password="user2_password",
   ...: email="user2@server.com",desc="description for user2")

In [3]: ac.save()

In [4]: ac_list = Account.objects.all()

In [5]: print ac_list
[<Account: user1>, <Account: user2>]
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In[15]中使用 manage. py 中的 shell 子命令生成了一个新的控制台。接下来的输入起始是从 In[1]开始的。
- In[1]从模型文件中导入了 Account 类。在 In[2]中使用其构造函数生成了一个 Account 对象，其中的参数即为各个属性的内容。
- 在 In[3]中调用 sava 方法将生成的 ac 对象保存到数据库中。在 In[4]中使用 all 方法得到了一个数据库中的数据列表。在 In[5]中输出了此列表信息。可以看到，现在数据库中已经有了两条数据。其中一条是使用管理界面创建的数据，而另外一条则是刚刚使用 API 生成的。

19.5.3　Session 功能

因为 HTTP 是无状态的协议，所以当需要在多个连接之间保持状态时，可以使用 session。Session 的实现可以根据服务器来决定，一般是保存在 cookie 中。通过此值可以用来在不同的连接中交换数据。在 Django 框架中，session 将保存在 request 对象的 session 值中。此值是一个字典对象，可以通过字典的相关操作来改变 HTTP 的 session 值。

如果需要在 Django 项目中开启 Session 功能，可以修改 settings. py 中属性 MIDDLEWARE_CLASSES 的值，加入下面的语句：

```
'django. contrib. sessions. middleware. SessionMiddleware',
```

然后在此配置文件中的 INSTALLED_APPS 加入下面的语句：

```
'django. contrib. sessions'
```

如果此前没有生成相关的数据库，还需要调用 manage. py 的 syncdb 子命令来创建相应的表。

通过上面的步骤就可以启用 Django 框架中的 Session 功能。在实际使用的时候，还需要在浏览器中开启 cookie 功能。

【例 19-1】演示在登录过程中 Session 的用途。当正确登录的时候，Django 将会在 Session 中生成 username 的内容。在退出的时候，将会删除此 Session 内容。

首先在 template 目录下生成一个 login. html 文件：

```
<html>
<title>Login Page</title>
<body>
{% if not username %}
<form method="post" action="/login/">
    Username: <input type="text" name="username" value="" ><br/>
    Password: <input type="password" name="password" value="" >
    <input type="submit" value="Login" >
</form>
{% else %}
{{ username }} is log out! <br/>
<form method="post" action="/logout/" >
    <input type="submit" value="Logout" >
</form>
{% endif %}
</body>
</html>
```

- 在这个页面中使用了 Django 的 if 模板标签，用来根据具体的条件显示不同的内容。在这里，即可以根据是否含有 username 来显示登录界面或者登出界面。
- 其他的 HTML 内容和普通的登录界面内容是一样的。这里不做过多的描述。

【例 19-2】接上例。在 views. py 文件中加入下面的内容，用来处理登录和登出操作。

更多资源请访问稀酷客(www.ckook.com)

```
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect
from Account.models import Account

def login(request): #登录操作
    username = request.POST.get('username', None)
password = request.POST.get('password', None)

    if username:
        ac_list = Account.objects.all() #获取数据库中的数据
        for ac in ac_list:
            if ac.username == username and ac.password == password:
request.session['username'] = username
                return render_to_response('login.html', {'username':username})

    return render_to_response('login.html')

def logout(request): #登出操作
    try:
        del request.session['username']
    except KeyError:
        pass
    return HttpResponseRedirect("/login/")
```

- 此段代码中定义了两个函数，分别是登录处理函数和登出处理函数。在登录处理函数中，首先通过从 POST 的数据得到 username 和 password 的值。
- 其后，判断传递过来的数据是否有 username。如果有 username，则从数据库 Account 表中获取所有的数据。对于数据集合中的每个数据，判断数据库中的用户名和密码是否和从 HTTP 请求中传递过来的数据相等。如果两者相等的话，则返回登录的页面。否则，返回登出的界面。
- 在登出操作函数中，将删除 Session 中的 username 值。接着返回到登录页面。

【例 19-3】接上例。修改 urls.py 文件，加入下面的 URL 前缀信息。

```
urlpatterns = patterns('',

    (r'^$', 'blog.views.index'),
    (r'^login/$', 'blog.views.login'),
    (r'^logout/$', 'blog.views.logout'),
    (r'^account/', include('blog.Account.urls')),
    (r'^admin/(.*)', admin.site.root),
)
```

这里设置了 login 和 logout 的页面处理函数。图 19-14 中显示了登录初始化页面。在此页面中输入用户名和密码后，单击【Login】按钮，将触发定义在 views.py 中的 login 函数，并将 username 和 password 的值传入 Django 项目。在处理函数中，Django 将会根据 POST 的值进行处理。当 username 为空或者用户名密码不匹配的时候，则返回到此页面。如果匹配，将会显示如图 19-15 所示的页面。

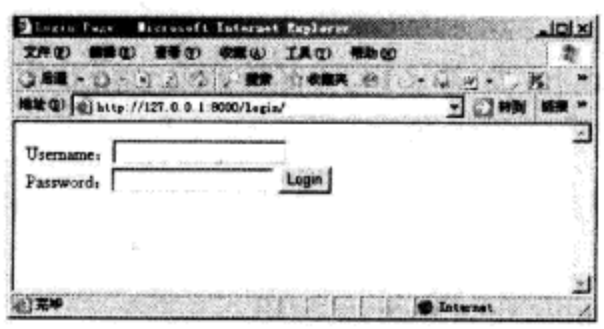


图 19-14　Session 实例中的登录初始化页面

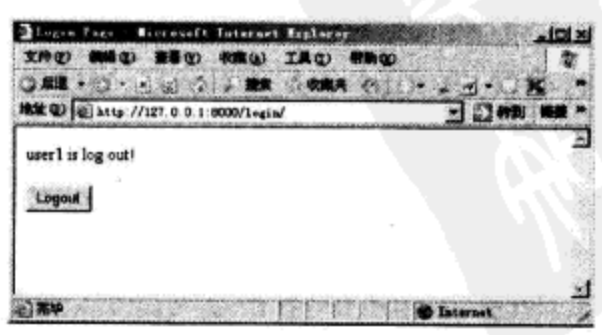


图 19-15　Session 实例中的登录页面

更多资源请访问稀酷客(www.ckook.com)

在图 19-15 中可以看到，页面中显示了登录的相关信息。这时，单击页面中的【Logout】按钮，将会回到图 19-14 所示的界面。

19.5.4　国际化

Django 框架中提供了完整的国际化支持，包括 Python 代码文档和模板文档。通过使用国际化功能，可以使得网站可以显示为多种语言。对 Django 项目的国际化包括如下的 3 个步骤。

（1）首先是定义在 Python 代码文件和模板文件中需要国际化的字符串。

（2）其后可以设置在不同语言下的字符串的文件。

（3）配置网站，显示特定的语言信息。

需要国际化的字符串包括两种，一种是在 Python 源码文件之中的，另外一种是在模板文件之中的。这两种的国际化方式并不相同。在 Python 源文件中，可以使用 django. utils. translation 中的 gettext 函数。其具体用法的演示如下：

```
from django. utils. translation import gettext
def index(request):
    content = 'Main Page'
    trans_text = gettext(content)
    ...
```

这里仅仅只是使用了文件中的部分代码。首先导入了 gettext 方法，可以使用此方法作用于需要国际化的字符串上。当有大量的字符串需要国际化的时候，使用 gettext 可能会占据大量的输入。习惯的做法是将 gettext 方法别名为“_”。如下所示：

```
from django. utils. translation import gettext as _
def index(request):
    content = 'Main Page'
    trans_text = _(content)
    ...
```

这里和上面一个代码片段的区别在于这里将 gettext 别名了_ 函数。同样地，可以国际化模板中的字符串。为了达到这一目的，需要在模板文件中的开始处加载 i18n 模板文件。如下所示：

```
{% load i18n % }
```

在模板文件中国际化字符串的最简单方法是使用 trans 模板标签。trans 模板标签的使用方法和 gettext 函数是类似的。此模板标签接受一个字符串作为参数，以此来标记此字符串是需要国际化的。如模板文件中下面的语句：

```
{% trans " Main Page" % }
```

这里将模板文件中的“Main Page”字符串标记为可以国际化。

接下来是生成国际化文件。可以使用 django-admin. py 的 makemessages 命令来生成 po 文件。具体命令如下：

```
make-messages. py -l zh_CN
```

这时会生成 locale/zh_CN/LC_MESSAGES/django. po 文件。可以直接翻译此 po 文件，也可以使用工具来修改。修改的方式很简单——在对应的字符串上翻译即可。

最后是设置语言。这种配置既可以是整个项目层次的，也可以是针对每个 Session 的。Django将在如下的 4 个地方来查找语言配置。

（1）首先是从当前用户的 Session 中看是否有 django_language 字符串。

（2）查看用户请求中是否包含 django_language 的 cookie。

更多资源请访问稀酷客(www.ckook.com)

（3）如果在 cookie 中找不到相应的内容，则查看此请求中的 HTTP_ACCEPT_LANGUAGE 头信息。Django 将寻找其支持的语言配置。

（4）最后是在 setting. py 文件中查找 LANGUAGE_CODE 的设置。

如果需要页面显示中文的相关信息，则可以将 LANGUAGE_CODE 设置为“zh_CN”：

```
LANGUAGE_CODE = 'zh_CN'
```

图 19-16 显示了在中文环境下的管理界面。

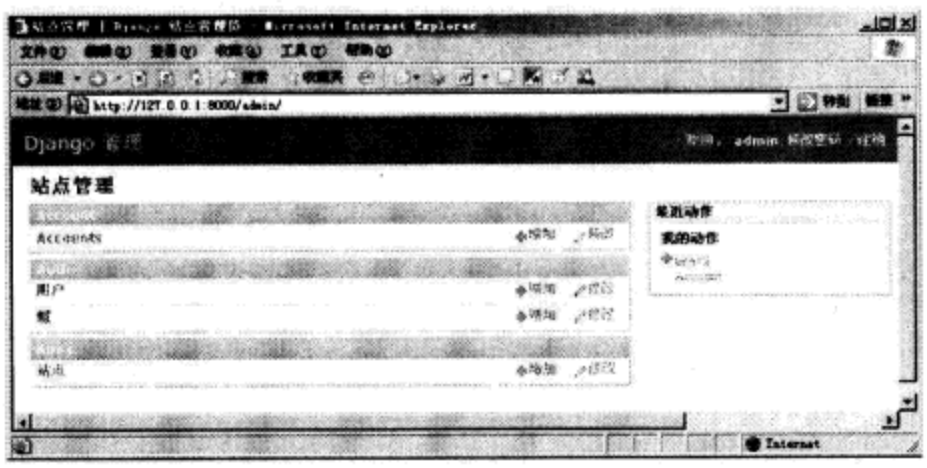


图 19-16　中文界面下的管理界面

19.6　小结

本章讲解了 Python 下的 Web 开发，主要介绍了 Django 框架的使用。首先介绍了 Python 语言中常见的 Web 开发框架，包括 Zope、TurboGears 和 Django 等，以及 Web 开发框架的选择。随后，介绍了在 Web 开发框架中常用的 MVC 模式。接下来的 3 个小节集中介绍了 Django 框架的应用，包括数据模型，页面处理和模板系统等，还有 HTTP 会话和国际化等内容。

更多资源请访问稀酷客(www.ckook.com)

第 20 章　敏捷方法学在 Python 中的应用——测试驱动开发

敏捷方法学是一种新兴的软件开发方法学，其核心为进化设计。在敏捷开发中，系统的设计是随着用户的需求而变化的，所以此时测试驱动开发尤其显得重要。这种开发方法在实现具体的功能之前需要完成测试部分。只有满足测试条件的代码才符合要求。在 Python 中的代码测试包含两种类型：unittest 和 doctest。本章的知识点：

- 测试驱动开发的特点和优势
- 测试驱动开发的应用步骤和注意事项
- 测试驱动开发环境的建立
- Python 下的单元测试
- Python 下的 doctest 应用

20.1　测试驱动开发

测试驱动开发（Test Driven Development，TDD）是敏捷开发中的一个重要组成部分。其基本的思想是测试先行。也就是说在开发具体的功能代码之前，需要首先编写此功能的测试代码。只有通过了测试代码，才能够加入到代码仓库中。在本节中，将对测试驱动开发这种开发模式进行详细的介绍，包括其优势和应用步骤等。

20.1.1　测试驱动开发模式（TDD）

随着软件规模的增大，软件开发人员的增多，在软件开发过程中出现了指导软件开发的开发模式。这规定了在软件开发的过程中各种文档和功能的规范标准。在软件开发的生命周期中，包含需求、设计、编码、测试和维护等阶段。通过采用不同的开发方法，可以使得不同类型的开发人员参与其中，从而实现最终的软件开发目标。对于这些不同的开发阶段，不同的开发模式虽然有着同样的阶段和共同的开发目标，但是不同的阶段所承担的任务和完成的时间是不一样的。

传统的软件开发模型有瀑布模型、演进模型、螺旋模型、喷泉模型和智能模型等。虽然在传统软件开发模型中提出了很多的开发模型，但是到现在还在被广泛使用的软件开发模型是瀑布模型。这种软件开发模型是 Winston Royce 于 1970 年提出来的，将一个软件开发的生命周期划分成了 6 个阶段：软件计划、需求分析和定义、系统设计、编码实现、软件测试和软件维护。在这 6 个阶段中，每个阶段有自己的工作内容，并且需要有相应的文档支持。其关系是一种自上向下加上部分反馈的运作模式。图 20-1 描述了这种开发模型的应用过程。

更多资源请访问稀酷客(www.ckook.com)

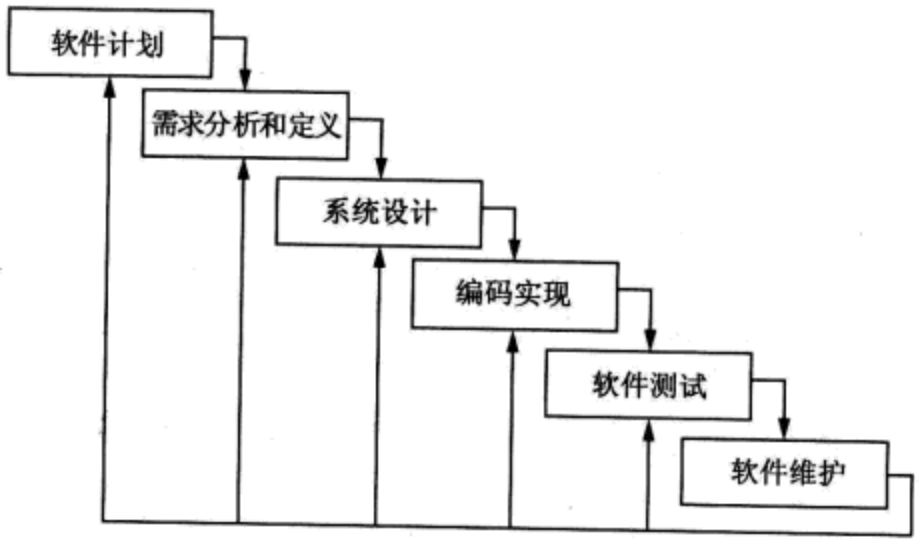


图 20-1　瀑布模型的应用过程

虽然瀑布模型为软件开发提供了按照阶段划分的检查点，但是这种开发模型已经越来越不适应现代化的软件开发。这种开发模式最大的问题在于只能够在软件生命周期的后期才能够看到最终结果，无法适应多变的用户需求。由于缺少应变能力，因此项目的开发具有比较大的风险。

在传统的软件开发模型中，测试阶段是放在软件生命周期的后期来完成的。这种处理方式使得软件编码中的问题只有在后期才能够被发现，从而带来严重的后果。一般情况下，在这种开发模式下测试阶段所占的时间比例都要占整个开发所用时间的一半左右。为了解决传统开发模式下的这些问题，敏捷方法学在快速开发的目标上强调基于测试的开发过程。

测试驱动开发的核心思想是强调测试在整个开发过程中的作用。在开始编写功能代码之前，需要首先编写符合需求的测试代码。对于一个特定的功能，需要完成此功能的测试代码，包括测试用例。在整个开发过程中，需要对每个阶段都进行测试驱动。也就是说，在实现某种功能之前，需要思考该功能的测试和验证过程，在编写了相关的测试代码之后，才开始进一步的编码工作。通过这样的方式，将各个功能完善，从而形成最终的目标软件。

比较流行的测试驱动开发模型有 V 测试模型和 X 测试模型。在图 20-2 中给出了 V 测试模型的示意图。

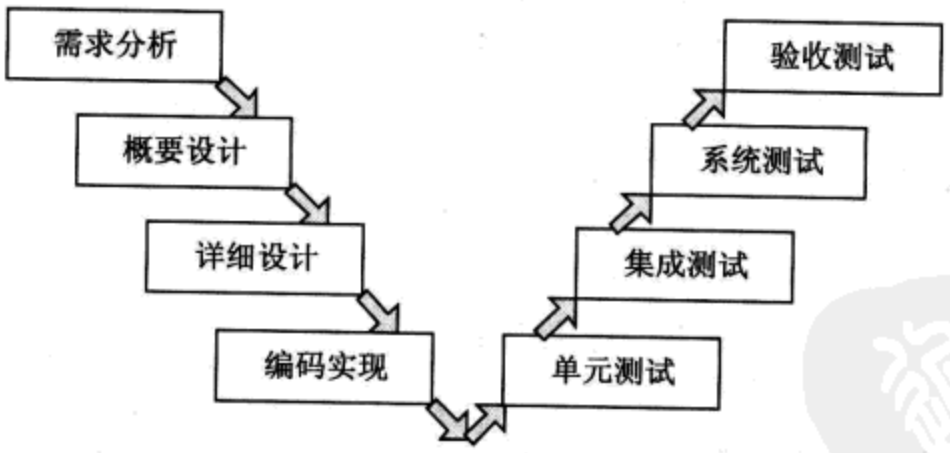


图 20-2　V 测试模型示意图

在图 20-2 中可以看到，在软件开发周期中的每个阶段，都需要考虑相应的测试工作。这些阶段包括需求分析阶段、概要设计阶段、详细设计阶段和编码实现阶段。相应的测试过程则被称为单元测试、集成测试、系统测试和验收测试。在软件的实际开发过程中，这些测试阶段可以根据需要来进行调整。测试代码也最好是有相关的测试文档，但是和强调文档的传统开发模型不同，这里的文档也是围绕测试这个需求而完成的。这种开发模型使得项目和其质量保证同时展开。

更多资源请访问稀酷客(www.ckook.com)

20.1.2　TDD 的优势

TDD（Test-driven development，测试驱动开发）采用的是以测试为核心的开发模式，所以具有传统开发模型所不具备的一些优点。一个最大的好处就是可以在短时间内构建出一个软件的原型出来。这使得在真正开始软件开发的时候能够对软件的实现规划有个好的了解，从而规避风险。在实际的开发过程中，这种以测试为核心的开发方法也可以使得在改正程序错误方面有着独特的优势。设想在写了一段代码后，开始进行单元测试，这个时候发现的问题就是在新写的这部分代码中。但是在传统的开发模型中，在整体编码完毕后再进行测试，这时候出现的问题就很难定位了。

在软件开发结束的时候，最终产品是伴随着详细的测试套件一起发布的。这从另一个侧面加强了最终产品的可靠性，同时为软件产品的升级和维护提供了便利。在测试驱动开发中，项目主管可以很清楚地看到已经完成了哪些软件需求，从而有助于把握项目进度。但传统的方式只能够定义一些检查点，而不能更细的划分功能需求点。

在传统的基于文档的开发模式中，开发人员和文档编写人员是分开的。这样一方面加重了沟通成本，另一方面使得开发人员对修订文档并不关心。而在测试驱动开发中，测试过程中所使用的测试用例代码可以作为代码的注释。也就是说，在编写代码的时候实际上也是在编写代码文档。在 Python 标准库中，提供了一种 doctest 模块，可以直接在代码的注释中书写测试用例。

测试驱动开发模式作为敏捷开发中的核心组成部分，同样也具有敏捷方法学的共同目标。这就是适应快速多变的用户需求。在传统的开发模型中，当前面的需求设计阶段确定下来后，后面如果需要再度更改需求是一件不太可能的事情。因为这意味着项目需要从瀑布的顶端开始运作。而对于测试驱动开发来说，总是通过编写测试用例，优先考虑实现代码的使用需求，包括功能、过程和接口等。这种方式使得最终代码符合后期开发的需求，能够提供代码的内聚性和复用性。

20.1.3　TDD 的使用步骤

TDD 开发模式在实现软件开发目标的同时，另外一个目标是整洁可用的代码。在明确了当前软件开发的需求后，TDD 中新添加一个功能的基本过程包括如下步骤。

（1）明确当前代码需要完成的功能，必要的时候书写相关的接口等。

（2）快速新增一个测试。

（3）运行所有的测试，看是否可以通过。若是测试通过则转到（6）。

（4）对功能代码进行细微的改动。

（5）重新运行所有的测试用例，保证全部通过。

（6）对代码进行重构，消除重复设计，优化代码结构。

上面的过程是对一个功能点的添加过程。将需求分解成多个不同的功能点，循环将其加入到软件中，即可完成一个软件实现的目标。在实际的操作中，还有如下一些注意事项。

在书写功能代码之前需要编写测试代码。书写功能代码的目的是为了能够通过测试代码的测试。这里可以看到，测试实例越丰富，表示功能代码书写得越完备。当有部分测试用例没有通过的时候，就要对功能代码进行修改，这样重复编码和测试的过程，直到通过测试为止。

对于那些没有通过测试的代码是不能够放入产品代码之中的。产品中的每行代码都应该是通过了充分测试的。只有这样，才能够保证后续加入的代码不会受前面加入的错误代码的影响。最后，在实际测试的时候，只需要对实现功能代码的部分进行测试就可以了。这样可以保证测试用例的简单。

在多个测试阶段中，最核心的是单元测试。在 Python 的标准库中，有两类代码测试模块。

更多资源请访问稀酷客(www.ckook.com)

分别为 unittest 和 doctest。两者都可以实现单元测试的功能。unittest 可以用来书写 PyUnit 的测试代码。和其他语言的单元测试工具一样，支持对软件代码的自动化测试。doctest 是一个特殊的测试模块。此模块将测试用例内置在了函数的文档字符串中，从而达到了文档和测试代码的统一。接下来将具体介绍如何使用 Python 标准库所提供的模块进行测试驱动开发。

20.2　unittest 测试框架

在 Python 标准库中，用来实现单元测试的模块是 unittest。通过此模块可以实现对软件的测试驱动开发过程。本节将详细介绍 unittest 模块的使用方法，包括如何创建测试用例、如何创建固件和如何创建测试套件等。

20.2.1　unittest 模块介绍

在其他的编程语言中，都有着自己的一套测试驱动开发工具。这些工具大多数都是属于 xUnit 家族。如 Java 语言的 JUnit 工具，C++ 语言中的 CppUnit 工具，还有 NUnit、RubyUnit 和 vbUnit 工具等。Python 语言也有这样的测试驱动开发工具——在 Python 版本 2.1 后被标准库引入的 PyUnit，别名为 unittest。unittest 模块提供了一种规范的方法来构造单元测试用例。如果不使用这个模块提供的方法而采用手工构造的方法，则需要编写大量的辅助测试代码而得不偿失。

和其他自动化测试框架一样，unittest 框架有着和其他自动化测试框架类似的接口。特别要提到的是，PyUnit 可以看作是 JUnit 工具的 Python 语言版本。读者如果对 JUnit 的 API 熟悉的话，可以看到 PyUnit 和其是非常相似的。

unittest 模块支持测试的自动化处理。更多的功能包括可以共享代码测试的初始化和结束代码、将测试用例封装成一个测试套件以及测试的多元化显示等。在 unittest 模块中提供了相关的类和方法，使得开发者可以很容易处理这些测试工作。

unittest 模块支持单元测试的各种重要概念。其中最重要的概念是测试用例。这是单元测试的最小组成部分。在其中仅仅只是查看特定的功能实现，检查在特定输入下的响应情况。unittest 模块提供了 TestCase 基类，用来创建新的测试用例。

在一组测试用例中，包含共同需要处理的代码，被称为测试固件。这些代码可能是在测试之前需要进行的初始化工作，或者是测试结束后所做的代码清理工作等。例如在测试之前，可能需要建立某些文件夹或者是开启某些服务。在构建测试用例的时候，可以使用 setUp 和 tearDown 来执行初始化和结束工作。在测试执行的时候，setUp 将首先被执行，而且只会执行一次。当此方法通过后，不管后面的测试是否通过，都会执行 tearDown 函数。

随着测试用例的增多，一个一个对测试用例管理显然是非常低效的。unittest 模块中提供了测试套件来解决这个问题。测试套件将一组测试用例集合起来作为一个测试对象。需要注意的是，测试套件是可以嵌套的，也就是说，可以在测试套件中包含测试套件。此功能是在 TestSuite 类中提供的。在其中的测试用例或者是测试套件都将会依次运行。

另外，在 unittest 模块中还有一个测试运行器，用来运行这些测试并为用户提供输出结果。这些运行器可能是一个图形化的接口或者是文本的接口，甚至只是一个简单的值来指示测试运行的结果。unittest 测试框架中使用 TestRunner 类来为测试的运行提供环境。这些类对象中提供了一个 run 方法，其中接受 TestCase 或者 TestSuite 参数，并返回运行结果。最常用的是 TextTestRunner运行器，默认情况下将向终端输出测试运行结果。

对于一个基于测试的软件来说，测试是软件产品中的一部分。实际上，在 Python 的标准安装中，就有对于标准模块的测试代码。这可以从其安装目录 Lib/test 中看到。

更多资源请访问稀酷客(www.ckook.com)

20.2.2　构建测试用例

软件测试的基本组成单元是测试用例，也可以说，单元测试是通过一些测试用例构建而成的。在测试用例中，包含对特定功能的测试。在 unittest 模块中，可以通过继承 TestCase 类来构建单元测试用例。通过覆盖 TestCase 类中的 runTest 函数可以实现功能的具体测试。

【例 20-1】演示构建单元测试用例。

```
#filename: tdd_1.py
#encoding = utf-8

import unittest
import string

class StringReplaceTestCase1(unittest.TestCase):
    """测试空字符替换"""
    def runTest(self):
        source = "HELLO"
        expect = "HELLO"
        result = string.replace(source, "", "")
        self.assertEqual(expect, result)

class StringReplaceTestCase2(unittest.TestCase):
    """测试空字符替换成常规字符"""
    def runTest(self):
        source = "HELLO"
        expect = "* H* E* L* L* O* "
        result = string.replace(source, "", "* ")
        self.assertEqual(expect, result)

class StringReplaceTestCase3(unittest.TestCase):
    """测试常规字符替换为空字符"""
    def runTest(self):
        source = "HELLO"
        expect = "HEO"
        result = string.replace(source, "LL", "")
        self.assertEqual(expect, result)

class StringReplaceTestCase4(unittest.TestCase):
    """测试常规字符换"""
    def runTest(self):
        source = "HELLO"
        expect = "HEMMO"
        result = string.replace(source, "LL", "MM")
        self.assertEqual(expect, result)
```

- 在【例 20-1】的示例代码中，有 4 段类似的代码。其中每段代码都在测试 string 模块中的 replace 方法的功能实现。具体来说，4 个测试用例分别是测试空字符替换成空字符、空字符替换成其他字符、其他字符替换成空字符和常规字符之间的替换。
- 在第 1 段代码中，首先定义了一个 StringReplaceTestCase1 测试类，此类继承了 unittest 模块中的 TestCase 基类。通过这种继承方式，StringReplaceTestCase1 类已经成为了 unittest自动化测试框架中的一个单元测试实例。对于具体需要测试的功能代码，则可以通过重载 runTest 方法来实现。可以看到，在 StringReplaceTestCase1 中也仅仅只是有一个函数名为 runTest 的方法。其中就实现了字符串替换功能中的空字符边界条件的检查。在 runTest 函数主体中有 4 个语句。其中前面两个字符串定义了源字符串和期望的字符串。第 3 个语句则是使用了 replace 函数来对字符串进行空字符的替换。在最后一个语句中，使用 assertEqual 方法来检查函数返回结果是否和期望结果一致。

更多资源请访问稀酷客(www.ckook.com)

- 接下来的 3 段代码和第 1 段代码是类似的。只是由于测试功能不一样，所以具体的测试单元实例实现的也不一样。

单元测试的最后步骤都是使用断言判断测试是否通过。在 Python 语言中，可以用内置的 assert语句来实现测试用例运行时候的断言。当测试用例运行时，如果断言为假，将会触发 AssertionError异常。同时，自动化测试框架会认为此测试用例测试失败。由于 Python 语言中的 assert 语句可能会在某些情况下被优化掉，所以在 TestCase 类中提供了 assert_方法，使得其在测试用例中不会因为优化而去掉。实际上，assert_方法是 assertTrue 和 failUnless 的别名。在unittest 模块中，提供了丰富的方法用来对测试结果进行判断。表 20-1 给出了所提供的测试方法。

表 20-1　unittest 模块中的测试方法

方　法	说　明
assertEqual	当两者相等的时候测试通过
assertEquals	
failUnlessEqual	
assertNotEqual	当两者不相等的时候测试通过
assertNotEquals	
failIfEqual	
assertAlmostEqual	当两者几乎相等的时候测试通过
assertAlmostEquals	
failUnlessAlmostEqual	
assertNotAlmostEqual	当两者几乎不相等的时候测试通过
assertNotAlmostEquals	
failIfAlmostEqual	
assertRaises	当触发特定异常的时候测试通过
failUnlessRaises	
assert_	当表达式为真的时候测试通过
assertTrue	
failUnless	
assertFalse	当表达式为假的时候测试通过
failIf	

在表 20-1 提供的方法中，除了测试是否触发异常的方法外，其他方法都有一个可选的信息参数。此参数可以在测试不通过的时候显示出来。

可以通过直接使用这些类的构造函数来生成一个测试用例的实例。如下面的操作：

```
In [1]: from tdd_1 import *

In [2]: testcase1 = StringReplaceTestCase1()
```

具体的测试运行方法将在后面进行介绍。

20.2.3　构建测试固件

随着单元测试实例的增多，在各个测试代码中会有很多相似的操作。如在【例 20-1】中，每个测试用例都需要在测试代码的前面加入对源字符串的定义。当然该实例还是比较简单的，在实际工作中会有很多复杂而琐碎的工作，如建立数据库的连接或启动某些服务

更多资源请访问稀酷客(www.ckook.com)

等。为了减少这种重复，testCase 类提供了 setUp 方法来使得单元测试实例在执行的时候都执行设置工作。

【例 20-2】接上例。在【例 20-1】的代码中加上测试固件。

```
#filename: tdd_2.py
#encoding = utf-8

import unittest
import string

class SimpleStringReplaceTestCase(unittest.TestCase):
    """ 准备测试的源字符串 """
    def setUp(self):
        self.source = "HELLO"

class StringReplaceTestCase1(SimpleStringReplaceTestCase):
    """测试空字符替换"""
    def runTest(self):
        expect = "HELLO"
        result = string.replace(self.source, "", "")
        self.assertEqual(expect, result)

class StringReplaceTestCase2(SimpleStringReplaceTestCase):
    """测试空字符替换成常规字符"""
    def runTest(self):
        expect = "* H* E* L* L* O* "
        result = string.replace(self.source, "", "* ")
        self.assertEqual(expect, result)

class StringReplaceTestCase3(SimpleStringReplaceTestCase):
    """测试常规字符替换为空字符"""
    def runTest(self):
        expect = "HEO"
        result = string.replace(self.source, "LL", "")
        self.assertEqual(expect, result)

class StringReplaceTestCase4(SimpleStringReplaceTestCase):
    """测试常规字符换"""
    def runTest(self):
        expect = "HEMMO",
        result = string.replace(self.source, "LL", "MM")
        self.assertEqual(expect, result)
```

- 本例和【例 20-1】的最大不同是这里使用了测试固件，将原来在各个单元测试用例中需要实现的代码统一到了一起。
- 在第 1 段代码中，定义了 SimpleStringReplaceTestCase 类，该类继承自 unittest.TestCase 类。随后 4 段代码中的类则继承自 SimpleStringReplaceTestCase 类。这种处理方式使得继承的各个类都拥有同样的初始化设置功能。
- 在 SimpleStringReplaceTestCase 类中，定义了一个 setUp 方法。在此方法中可以放置单元测试用例中的公共工作。方法主体中仅有一条语句，就是设置了源字符串。
- 在后面定义的 4 个类的 runTest 方法中，唯一的区别在于将原来定义在各个测试用例中的源字符串改成了使用 setUp 方法中设置的字符串。这种处理方式使得可以在大量的单元测试用例中共享数据。

在测试运行过程中，如果在 setUp 中定义的函数出错，测试框架会认为此单元测试用例出错，不会执行 runTest 方法中定义的测试代码。

同样地，如果需要在结束的时候执行某些代码，TestCase 类中提供了 tearDown 方法用来执行在运行 runTest 之后的清理工作。setUp 和 tearDown 是对应的。也就是说，如果 setUp 方法执行

更多资源请访问稀酷客(www.ckook.com)

成功的话，则不管 runTest 方法是否执行成功，tearDown 方法都将会执行。

20.2.4　组织多个测试用例

当有多个测试用例的时候，【例 20-2】中使用了测试固件，但是，所有的测试用例还是单独作为一个类存在的。这种处理方式实际上是非常耗时的，所以并不推荐这样使用。unittest 模块提供了另一种更方便的方式来组织多个测试用例。

【例 20-3】演示将多个测试用例组织起来。

```
#filename: tdd_3.py
#encoding = utf-8

import unittest
import string

class StringReplaceTestCase(unittest.TestCase):
    def setUp(self):
        self.source = "HELLO"

    """测试空字符替换"""
    def testBlank(self):
        expect = "HELLO"
        result = string.replace(self.source, "", "")
        self.assertEqual(expect, result)

    """测试空字符替换成常规字符"""
    def testBlankOrd(self):
        expect = "* H* E* L* L* O* "
        result = string.replace(self.source, "", "* ")
        self.assertEqual(expect, result)

    """测试常规字符替换为空字符"""
    def testOrdBlank(self):
        expect = "HEO"
        result = string.replace(self.source, "LL", "")
        self.assertEqual(expect, result)

    """测试常规字符换"""
    def testOrd(self):
        expect = "HEMMO"
        result = string.replace(self.source, "LL", "MM")
        self.assertEqual(expect, result)
```

- 在【例 20-3】中只有一个 StringReplaceTestCase 类，此类是从 unittest.TestCase 中继承的。
- 在类中首先定义了前面的 setUp 方法，用来实现对测试过程中的初始化设置。
- 接着定义了 4 个函数。可以看到，这里并没有重载 runTest 方法，而是使用了 4 个测试函数。注意到这里的函数都是以“test”开始的，这在单元测试框架中是一种惯例。各个函数中的内容和前面在类中 runTest 方法的实现是一样的。

可以使用带参数的测试类构造函数来生成测试用例的实例。其中在 StringReplaceTestCase 有 4 个测试方法。

```
In [1]: from tdd_3 import *

In [2]: testBlank = StringReplaceTestCase("testBlank ")

In [3]: testOrd = StringReplaceTestCase("testOrd ")
```

更多资源请访问稀酷客(www.ckook.com)

- 在这里，生成了两个测试用例的实例：testBlank 和 testOrd。

20.2.5　构建测试套件

当有多个测试用例的时候，可以根据测试的用途和特性将其组合。unittest 模块中提供了 TestSuite 类来生成测试套件。直接使用此类的构造函数可以生成一个测试套件的实例。在此类中提供了一个方法 addTest 来将单元测试用例加入到测试套件中：

```
StringReplaceTestSuite = unittest.TestSuite()

StringReplaceTestSuite.addTest(StringReplaceTestCase("testBlank"))
StringReplaceTestSuite.addTest(StringReplaceTestCase("testOrd"))
```

- 在上面的3句代码中，首先通过调用 TestSuite 的构造函数生成了一个测试套件实例。接下来的两句使用该实例的 addTest 方法将 StringReplaceTestCase 类中的两个功能测试加入到了测试套件中。

在实际操作中，会在测试模块中返回已经构建好的测试套件。如下面的函数：

```
def suite():
    StringReplaceTestSuite = unittest.TestSuite()

    StringReplaceTestSuite.addTest(StringReplaceTestCase ("testBlank "))
    StringReplaceTestSuite.addTest(StringReplaceTestCase ("testOrd "))

    return StringReplaceTestSuite
```

- 这里定义了一个 suite 函数，用来返回已经创建好的测试套件实例。函数主体中的部分和前面的内容是相同的。这种方式会使测试更加便捷。

同样，可以在 TestSuite 类的构造函数中生成测试套件实例：

```
def suite():

    tests = ['testBlank', 'testOrd']
StringReplaceTestSuite = unittest.TestSuite(
                         map(StringReplaceTestCase, tests))

    return StringReplaceTestSuite
```

- 这里使用了一种在 TestSuite 构造函数中生成测试套件实例的方法。

由于一般需要加入多个单元测试用例，因此在上面的两种方法中，都是需要将每个测试用例加入其中。这种处理方式使得在含有大量单元测试用例的时候会很繁琐。由于经常需要加入所有的测试用例，所以在 unittest 模块中提供了一个实用方法 makeSuite，用来创建一个由测试用例类内所有的测试用例所组成的测试套件实例：

```
def suite():

StringReplaceTestSuite = unittest.makeSuite(StringReplaceTestCase,'test')

    return StringReplaceTestSuite
```

- 这里的代码主要是使用了 makeSuite 方法来为 StringReplaceTestCase 类中的所有单元测试用例生成测试套件。

实际上测试套件是可以嵌套的。也就是说，可以将测试套件加入到另一个测试套件中，从而将多个测试套件组合在一起。这种处理方式和将多个测试用例加入到测试套件中是一样的。

更多资源请访问稀酷客(www.ckook.com)

【例 20-4】演示加入另外一个测试用例类 StringStripTestCase。

```
#filename: tdd_4.py
#encoding = utf-8

import unittest
import string

class StringStripTestCase(unittest.TestCase):
    def testBlank(self):
        expect = "HELLO"
        result = string.strip("HELLO      ")
        self.assertEqual(expect, result)

    def testStr(self):
        expect = "HELLO"
        result = string.strip("xxHELLOxx", "xx")
        self.assertEqual(expect, result)

class StringReplaceTestCase(unittest.TestCase):
#省略 StringReplaceTestCase 实现代码

def suite():
    StringStripTestSuite = unittest.makeSuite(StringStripTestCase,'test')
    StringReplaceTestSuite = unittest.makeSuite(StringReplaceTestCase,'test')

    alltests = unittest.TestSuite((StringStripTestSuite, StringReplaceTestSuite))
    return alltests
```

- 在【例 20-4】代码中，在定义 StringReplaceTestCase 类的基础上，定义了 StringStripTestCase 的另外一个测试类。此类的创建方式和 StringReplaceTestCase 类是类似的。在此测试类中，有 testBlank 和 testStr 这两个单元测试用例。StringStripTestCase 测试类的用途是用来测试 string 模块中的 strip 方法。
- 在函数的后面，定义了一个 suite 方法用来返回生成的测试套件实例。在函数主体中，使用 unittest 模块中的 makeSuite 方法从 StringStripTestCase 测试类和 StringReplaceTestCase测试类中构建测试套件实例。最后将这两个测试套件实例组合成了一个测试套件实例并返回。

加入到测试套件中的测试实例，其运行的顺序是通过使用 Python 内置函数 cmp 对测试方法名排序而得到的。

20.2.6　重构代码

在测试驱动开发模式下，由于需要进行迅速的开发和测试，所以在测试的过程中，需要对已经完成的代码进行重构。重构可以使软件产品维持一种相对简单和可读的特性。这种特性在后续维护中是很重要的。

在测试类中，需要测试的测试用例可能是很多的。在这种情况下，如果对于每个测试用例都生成如【例 20-1】的一个测试用例，会使得代码非常冗长。为此，可以将例子中的具体代码部分实现重构，使得可以更好的关注测试用例。

【例 20-5】演示代码部分重构的方法。

```
class StringReplaceTestCase(unittest.TestCase):

    def setUp(self):
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.source = "HELLO"

    def checkequal(self, result, object, methodname, * args):
        realresult = getattr(object, methodname)(* args)
        self.assertEqual(
            result,
            realresult
        )

    """测试空字符替换"""
    def testBlank(self):
        self.checkequal("HELLO", "", "replace", "", "")

    """测试空字符替换成常规字符"""
    def testBlankOrd(self):
        self.checkequal("* H* E* L* L* O* ", self.source, "replace", "", "* ")

    """测试常规字符替换为空字符"""
    def testOrdBlank(self):
        self.checkequal("HE* O", self.source, "replace", "LL", "* ")

    """测试常规字符换"""
    def testOrd(self):
        self.checkequal("HEMMO", self.source, "replace", "LL", "MM")
```

- 在定义的 StringReplaceTestCase 类中，对原来的测试用例代码进行了重构处理。经过这样的处理，可以在每个功能的测试中加入更多的测试用例。
- 对测试类中的各个测试用例进行了重构，将重复的功能放在了一起。这样的处理使代码变得简单和可读。
- 重构的函数为 checkequal 方法。其参数中包含期望得到的结果、源字符串、方法名以及参数。从这里可以看出，此重构函数不但可以用在 replace 方法的测试中，而且可以用在其他方法的测试中，包括前面的 strip 方法测试等。

重构代码是伴随着测试进行的。在对代码进行修改的时候，最好同时也对代码进行重构，从而保持代码的整洁。

20.2.7　执行测试

在测试用例和测试套件都完成之后，可以通过测试来检查软件产品代码是否编写正确。有多种测试方法可以用来测试代码。

1. 交互式执行测试

在 unittest 测试框架中提供了 TestRunner 类来为测试的运行提供环境。在实际使用中，最常见的 TestRunner 类是 TextTestRunner 类。此类的实现使用一种文字化的运行方式来报告最后的测试结果。在默认情况下，此类将会把输出结果发送到 sys. stderr 上。

【例 20-6】演示使用 TextTestRunner 类执行测试的方法。

```
#filename: tdd_5.py
#encoding = utf-8

import unittest
import string

class StringStripTestCase(unittest.TestCase):
```

更多资源请访问稀酷客(www.ckook.com)

```
#省略 StringStripTestCase 实现代码

class StringReplaceTestCase(unittest.TestCase):
#省略 StringReplaceTestCase 实现代码

def suite():
    StringStripTestSuite = unittest.makeSuite(StringStripTestCase,'test')
    StringReplaceTestSuite = unittest.makeSuite(StringReplaceTestCase,'test')

    alltests = unittest.TestSuite((StringStripTestSuite, StringReplaceTestSuite))
    return alltests

if __name__ == "__main__":
    runner = unittest.TextTestRunner()

    runner.run(suite())
```

- 本例和【例 20-4】文件不同的是，最后加入了执行测试的代码。使用 TextTestRunner 类构建了一个运行器对象。此对象提供了一个 run 方法，开始执行测试。run 方法所接受的参数为本例中定义的测试套件 suite()。这样，测试框架将会自动运行测试套件中的测试用例。

实际上，使用 TextTestRunner 类可以很容易在交互式终端环境下进行测试。如在 Python 命令行中进行测试：

```
In [1]: import unittest

In [2]: import tdd_5

In [3]: runner = unittest.TextTestRunner()

In [4]: runner.run(tdd_5.StringReplaceTestCase("testBlank "))
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
Out[4]: <unittest._TextTestResult run=1 errors=0 failures=0>
```

- 在 In[1,2]中导入了所需要的模块，这包括待测试用例所在的文件。
- 在 In[3]中使用 TextTestRunner 类的构造函数生成了一个运行器实例。
- 在 In[4]中调用其对象的 run 方法，对 StringReplaceTestCase 测试类中的 testBlank 测试实例进行了测试。从输出结果可以看出，测试成功。
- 从 Out[4]中可以看出，返回了一个 TestResult 方法。当测试不通过的时候，可以查看其对象的 errors 和 failures 属性，得到产生错误的测试对象。

2. 命令行运行测试

在 unittest 模块中有一个全局方法 main，利用它可以方便的测试已经构建好的测试模块。

【例 20-7】接上例。使用 main 方法修改【例 20-6】的代码。

```
#省略部分代码

def suite():
    StringStripTestSuite = unittest.makeSuite(StringStripTestCase,'test')
    StringReplaceTestSuite = unittest.makeSuite(StringReplaceTestCase,'test')

    alltests = unittest.TestSuite((StringStripTestSuite, StringReplaceTestSuite))
```

更多资源请访问稀酷客(www.ckook.com)

```
    return alltests

if __name__ == "__main__":
    #runner = unittest.TextTestRunner()
    #runner.run(suite())

    unittest.main()
```

本例与【例 20-6】的主要区别在于使用了 unittest.main 方法。实际上，main 方法将使用 unittest.TestLoader 类来自动查找和加载测试类中的测试用例。在执行【例 20-7】的时候，测试模块中的所有单元测试用例都将会被执行。下面的操作演示了运行结果：

```
In [5]: run tdd_5.py
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

- 从输出结果中可以看到，测试模块中所包含的 6 个单元测试用例都被正确执行。

可以通过加上“-v”参数来显示详细的信息。更多的参数可以使用“-h”来查看：

```
In [6]: run tdd_5.py -v
testBlank (__main__.StringReplaceTestCase) ... ok
testBlankOrd (__main__.StringReplaceTestCase) ... ok
testOrd (__main__.StringReplaceTestCase) ... ok
testOrdBlank (__main__.StringReplaceTestCase) ... ok
testBlank (__main__.StringStripTestCase) ... ok
testStr (__main__.StringStripTestCase) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.015s

OK
```

- 这里显示了具体测试用例运行的过程。可以看到，这里的顺序是根据类名和方法名的字母顺序来排定的。

修改【例 20-5】中 StringReplaceTestCase 测试类的 testOrdBlank 实现代码，如将“HE * O”改为“HEO”。这时候，再次运行测试的时候将出错：

```
In [7]: run tdd_5.py -v
testBlank (__main__.StringReplaceTestCase) ... ok
testBlankOrd (__main__.StringReplaceTestCase) ... ok
testOrd (__main__.StringReplaceTestCase) ... ok
testOrdBlank (__main__.StringReplaceTestCase) ... FAIL
testBlank (__main__.StringStripTestCase) ... ok
testStr (__main__.StringStripTestCase) ... ok

======================================================================
FAIL: testOrdBlank (__main__.StringReplaceTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tdd_5.py", line 50, in testOrdBlank
    self.checkequal("HEO", self.source, "replace", "LL", "* ")
  File "tdd_5.py", line 36, in checkequal
    realresult
AssertionError: 'HEO' != 'HE* O'

----------------------------------------------------------------------
Ran 6 tests in 0.016s
```

更多资源请访问稀酷客(www.ckook.com)

```
FAILED (failures=1)
----------------------------------------------------------------------
SystemExit                                 Traceback (most recent call last)
#省略部分输出结果

SystemExit: True
WARNING: Failure executing file: <tdd_5.py>
```

- 在输出结果中可以看到，这里的 6 个单元测试用例成功了 5 个，失败了 1 个。同时，给出了此失败测试用例的具体信息："AssertionError: 'HEO' ! = 'HE * O'"。当然，这里并不是功能代码实现的问题，而是期待的测试结果不正确。在实际操作中，一般是需要对功能代码进行修改的。

实际上，也可以将 unittest 模块作为脚本运行，将需要执行的测试套件中的测试用例名来作为参数传递。如下面的操作示例：

```
In [8]: run unittest.py  tdd_5.StringReplaceTestCase.testBlank
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

- 这里同样实现了对 testBlank 测试实例的测试。

3. 图形界面下的测试

在 PyUnit 中，还有一个图形化的测试界面用来实现测试的图形化显示。这是使用 Tkinter 编写的。可以使用下面的命令来执行测试：

```
In [9]: run unittestgui.py tdd_5.suite
```

图 20-3 所示窗口为测试模块中含有测试用例失败情况的显示。当测试执行完毕后，图中显示了总共完成的测试用例和失败的测试用例。这里的进度条中使用红色明确的指示了测试模块含有未通过的测试。这些信息在下面的对话框中显示了出来。单击下面的失败测试用例，可以看到更详细的错误信息。

在修正了源代码中的错误后，重新执行此测试，可以看到如图 20-4 所示的结果。

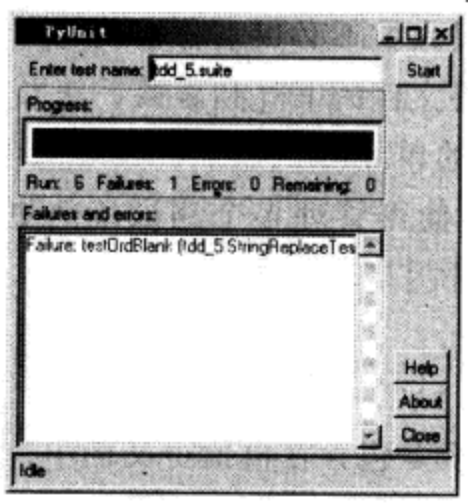


图 20-3　测试模块中含有失败测试用例的情况

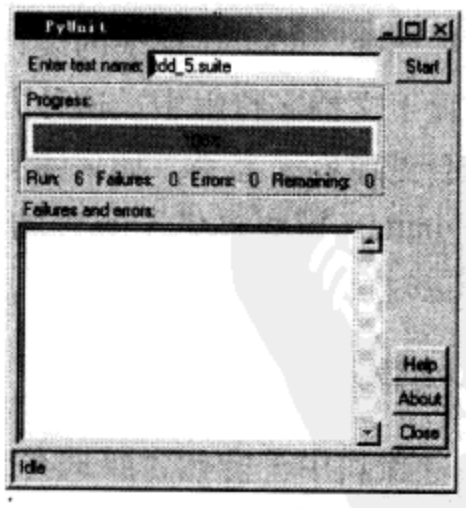


图 20-4　测试模块全部通过测试的情况

在图 20-4 所示窗口中可以看到进度条为绿色。表明测试模块中的所有单元测试用例都通过了测试。其实，这也是测试驱动开发希望达到的结果。只有在这种测试用例都通过的前提下，才能够继续添加新的功能和代码。

更多资源请访问稀酷客(www.ckook.com)

20.3　使用 doctest 模块进行测试

在 Python 语言中还支持另外一种测试框架，这就是 doctest 模块。使用此模块可以将代码中的 docstring 作为测试用例来运行，从而判断函数执行的正确性，形成运行结果。本节将介绍如何使用 doctest 来执行测试。

20.3.1　doctest 模块介绍

doctest 模块作为一种新的单元测试框架，可以有效的利用代码注释中的文档内容。这种处理方式使得文档可以作为测试代码来执行。

在支持 doctest 模块的代码中，docstring 由普通注释部分和代码执行部分两部分构成。其中，普通注释部分和原来的注释形式是相同的。对于可执行部分则有一定的格式要求。其中可执行部分通过使用“ > > > ”和“…”来和普通注释部分区分。注意，这里的提示符均为Python标准 Shell 中的提示符。对于可执行部分，开发人员可以预先编写测试用例。这些测试用例的书写方式和 unittest 模块中是类似的，只不过这里采用了在 Python 终端下运行的方式。在可执行部分中包含输入和输出两个部分。在实际运行过程中，doctest 模块将搜索代码中 docstring 的可执行部分，并实际执行这些代码。然后比较运行的结果和期望值，作为一次测试结果。

doctest 作为一种测试框架，可以实现文档和代码的同步，并可以作为书写教程的一个好工具。另外，可以将文档字符串变成可执行文档，从而完成对代码的测试。与 unittest 模块比较，doctest 测试框架有一些特点。首先是使用方法简单，只需要拷贝和粘贴 Python 终端下的交互式输入和输出即可。这种处理方式使得开发人员在项目初期可以更快的入手。另外，可以使用命令行参数来灵活的控制测试代码的运行，并通过增加选项来增强文本的输出显示。同时，还可以灵活的选择需要测试的 docstring。

但是，正是由于 doctest 模块的这种简单性，因此不适合于某些领域。doctest 是通过终端获取用户的输出结果来进行比较的，这就要求测试的机器所输出的结果都是一致的。另外，在 doctest 模块中没有提供测试固件，这在有多个单元测试用例的时候会比较繁琐。

虽然 doctest 不是一种很严格的测试驱动开发框架，但是作为一种新的测试框架，也给开发者提供了另外一种选择。doctest 模块的设计初衷是开发人员不必再书写相关的测试代码，而仅仅是将这种测试内置于文档字符串中。这种方式可以完善文档的书写和审核。如果是刚开始编写测试，文档测试将会更加容易上手。如果需要编写大量的单元测试用例，则可以考虑使用 unittest 测试框架。因为 unittest 模块提供了完善的类和方法，使得管理多个测试用例更加方便。同时，unittest 框架还可以对测试用例进行自定义，从而有助于提高软件测试的可靠性。

20.3.2　构建可执行文档

因为在 doctest 模块中没有包含测试类或者方法，都是包含在可执行文档中，所以doctest测试框架的核心是构建可执行的文档部分。

以 Python 帮助文档中介绍的例子为基础来演示 doctest 模块的使用。下面就是 Python 中一个简单的可执行文档：

```
    > > > factorial(5)
    120
    """
```

更多资源请访问稀酷客(www.ckook.com)

这个代码片段中的 factorial 为函数名。其功能为得到一个数的阶乘值。在这里，使用了“5”作为参数，而“120”是期望得到的结果。注意到，这里使用了 Python 的提示符“ > > > ”,看起来这就像是在 Python 标准 Shell 中输入的一样。下面代码中有完整的阶乘实现和其 docstring。

【例 20-8】演示阶乘实现的 docstring 代码。

```
#filename: doctest_1.py
#encoding = utf-8

def factorial(n):
    """Return the factorial of n, an exact integer > = 0.
    If the result is small enough to fit in an int, return an int. Else return a long.

    这里包含了多个测试例,用来测试 factorial 方法的返回值
    > > > [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    > > > [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    > > > factorial(30)
    265252859812191058636308480000000L
    > > > factorial(30L)
    265252859812191058636308480000000L
    > > > factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be > = 0

    Factorials of floats are OK, but the float must be an exact integer:
    检测浮点数是否为整数
    > > > factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    > > > factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    溢出检查
    > > > factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n > = 0:
        raise ValueError("n must be > = 0")
    if math.floor(n) ! = n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor < = n:
        result * = factor
        factor + = 1
    return result
```

- 这里的代码虽然比较长，但是实际上和前面介绍的可执行文档部分并没有本质的区别。

更多资源请访问稀酷客(www.ckook.com)

只是这里的测试用例更多一些。

- 由于这种测试方法将测试放在了代码注释中，所以可以很容易的为测试过程书写代码。在对 factorial 方法实现的测试中，有三个方面的测试。第一个方面是测试阶乘的返回值，包括小于 0 的数值。第二个方面是对输入进行检查，看是否为整数，如果为浮点数也需要能够转换为整数。第三个方面是对阶乘是否溢出进行检查。
- 最后是 factorial 函数的实现，其中就有上面测试中各种情况的处理。如输入小于 0 的值、输入浮点数和输出结果太大的情况等。

在 doctest 模块的实际使用中，可以直接从 Python 标准终端进行拷贝。

20.3.3　执行 doctest 测试

为了能够对含有可执行文档的 docstring 执行测试，可以在【例 20-8】需要测试的代码后面加上下面的部分。通过这样的处理，此模块就具有了 doctest 测试的功能：

```
def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

这里实际上是定义了一个_test 函数。此函数首先导入 doctest 模块，然后调用了其中的 testmod 方法。testmod 方法将会搜索整个模块中的__doc__来寻找文档字符串中的可执行部分，并进行 doctest 测试。直接运行此文件，即可执行 doctest 测试。具体操作如下：

```
In [1]: run doctest_1.py
```

从运行结果来看，并没有任何的输出结果。这是因为现在所有的测试用例都通过了测试。当对文档字符串中可执行部分做一些修改，doctest 测试失败。则再次运行的时候，显示如下：

```
In [2]: run doctest_1.py
**********************************************************************
File "doctest_1.py", line 17, in __main__.factorial
Failed example:
    [factorial(n) for n in range(6)]
Expected:
    [1, 1, 3, 6, 24, 120]
Got:
    [1, 1, 2, 6, 24, 120]
**********************************************************************
1 items had failures:
   1 of   8 in __main__.factorial
***Test Failed*** 1 failures.
```

上面的输出结果给出了很详细的出错信息。其中有出错的单元实例以及最终输出结果的对比。当测试用例全部通过测试的情况下，可以加上“-v”参数来显示具体的测试过程：

```
In [3]: run doctest_1.py -v
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
```

更多资源请访问稀酷客(www.ckook.com)

```
ok
#省略部分输出结果
Trying:
    factorial(30L)
Expecting:
    265252859812191058636308480000000L
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
ok
Trying:
    factorial(30.1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
ok
#省略部分输出结果
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items had no tests:
    __main__
    __main__._test
1 items passed all tests:
   8 tests in __main__.factorial
8 tests in 3 items.
8 passed and 0 failed.
Test passed.
```

上面的输出结果给出了 doctest 模块详细的测试过程。doctest 模块将执行文档字符串中的可执行部分，并与期望值进行比较，从而形成测试结果。最后给出了此次 doctest 测试的综合结果信息。

另外，在 doctest 模块中还有一个 testfile 方法，用来读取指定文件中包含可执行文档的 docstring。如果将文档字符串保存在 factorial_docstring.txt 文件中，则使用下面的代码同样可以实现 doctest 测试。这种处理方法的好处是可以将 docstring 集中处理：

```
def _test():
    import doctest
    doctest.testfile("factorial_docstring.txt")

if __name__ == "__main__":
    _test()
```

这里的 factorial_docstring.txt 文件中保存有针对 factorial 方法的 docstring 文档。

上面介绍了 doctest 测试的应用。从示例代码中可以看出，这种测试方法还是比较简单的。在实际操作中，可以根据具体情况，将 doctest 模块和 unittest 模块结合起来使用，以便发挥测试驱动开发模式的最大优势。

更多资源请访问稀酷客(www.ckook.com)

20.4　小结

本章以测试驱动开发模式为基础，讲解了敏捷方法学在 Python 中的应用。首先详细地介绍了测试驱动开发模式的含义、优势及应用步骤，并介绍了 Python 语言中对测试驱动开发的支持。接下来，详细地介绍了 Python 语言中的两种测试框架。一种是 unittest 模块，这是单元测试工具 xUnit 在 Python 语言中的实现。详细地介绍了如何使用 unittest 模块来构建测试用例、组成测试套件并进行测试的过程。另一个是 doctest 模块。这是一种新的测试框架，直接使用文档字符串中的可执行部分来进行测试，很好的结合了代码和文档。

更多资源请访问稀酷客(www.ckook.com)

第四篇
Python 的其他应用

第 21 章　Python 中的进程和线程

进程和线程是操作系统中的基本运行单元。在 Python 的标准安装中，对两者有很好的支持。掌握进程和线程编程，将有助于理解系统的运行过程，提高程序性能。本章的知识点：

- 进程、线程的概念
- 程序运行环境
- 使用 system、abort 和 exec 家族函数
- subprocess 模块介绍
- signal 模块和信号
- 多线程的生成和终止
- 多线程的数据同步

21.1　进程和线程

进程和线程作为操作系统中的两个重要概念，既有相似之处，也有不同之处。本节首先介绍两者的概念，然后对 Python 中的实现方式进行简单介绍，最后介绍标准库中与进程和线程相关的函数和模块。

21.1.1　进程和线程的概念

对于一般的程序，可能会有若干个进程，同时每一个进程又可能有多个同时执行的线程。进程是资源管理的最小单位，而线程则是程序执行的最小单位。

1. 进程

直观来说，进程就是正在执行的程序，是多任务操作系统中执行任务的基本单元，是包含了程序指令和相关资源的集合。在 Windows 操作系统下，打开【任务管理器】，在【进程】选项卡中就可以看到计算机中当前正在运行的进程，如图 21-1 所示。

操作系统隔离各个进程可以访问的地址空间。如果进程间需要传递信息，可以使用进程间通信（Inter-Process Communication）或者其他方式，如文件或者数据库等。在进程调度中，进程进行切换所需要的时间是比较多的。为了能够更好地支持信息共享和减小切换开销，从进程中演化出了线程。

更多资源请访问稀酷客(www.ckook.com)



图 21-1　Windows 下的进程

2. 线程

线程是进程的执行单元。对于大多数程序来说，可能只有一个主线程，但是为了能够提高效率，有些程序会采用多线程，在系统中所有的线程看起来都是同时执行的。例如，现在的多线程网络下载程序中，就使用了这种线程并发的特性：程序可以将欲下载的文件分成多个部分，然后同时进行下载，从而加快下载速度。虽然线程不是一个容易掌握和使用的概念，但是，从网络下载的示例可以看到，如果运用得当，还是可以获得很不错的性能的。

3. 进程和线程的对比

明确进程和线程的区别，这对于使用 Python 编程是非常重要的。一般地，进程是重量级的。具体包括进程映像的结构、执行细节以及进程间切换的方法。在进程中，需要处理的问题包括进程间通信、临界区管理和进程调度等。这些特性使得新生成一个进程的开销比较大。线程是轻量级的。线程之间共享许多资源，容易进行通信，生成一个线程的开销较小，但是使用线程会遇到如死锁、数据同步和实现复杂等问题，需要妥善解决。

由于 Python 语言使用了全局解释器锁（Global Interpretor Lock，GIL）和队列模块，所以在线程实现上的复杂度相对于其他语言来说要低得多。需要注意的是，由于 GIL 的存在，所以 Python 解释器不是线程安全的。因为当前线程必须持有这个全局解释器锁，从而可以安全的访问 Python 对象。虽然使用 GIL 使得 Python 不能很好的利用多 CPU 的优势，但是现在还没有比较好的办法来代替，因为去掉 GIL 会带来许多问题。

针对 I/O 受限的程序（如网络下载类），可以使用多线程来提高程序性能。而对于 CPU 受限的程序（如科学计算类），使用多线程并不会带来效率的提升。这个时候，建议使用进程或者混合进程和线程的方法来实现。

21.1.2　Python 中对于进程和线程处理的支持

在前面提到，Python 对于进程和线程处理都有着很好的支持。本小节将介绍在 Python 语言的标准库中有哪些相关的模块和函数。表 21-1 和表 21-2 中描述了这些模块和函数，具体的使用方法将在后面的章节中进行介绍。

更多资源请访问稀酷客(www.ckook.com)

表 21-1　与进程和线程相关的 Python 模块

模块名称	说　明
os/sys	包含基本进程管理函数
subprocess	Python 基本库中多进程相关模块
signal	Python 基本库中信号相关模块
threading	Python 基本库中线程相关模块

表 21-2　os/sys 模块中与进程相关的函数

函数名称	说　明
popen	生成新的进程
system	直接生成字符串所代表的进程
abort/exit	终止进程
exec 家族	在现有进程环境下生成新进程

21.2　Python 下的进程编程

进程是程序运行的实例，本节将介绍 Python 下的进程编程。首先介绍进程的运行环境，然后对 os 模块中与进程相关的函数进行说明，从而可以用来生成和终止进程。在此基础上，将更进一步讨论 subprocess 和 signal 模块的进程高级应用。在掌握了这些相关模块的使用方法后，就可以很方便地管理系统中的进程了。

21.2.1　进程的运行环境

在具体的介绍进程管理之前，需要了解进程的运行环境。对于每个运行的进程，系统都会提供一个相关的运行环境。一般地，可以将进程的运行环境看作是环境变量的集合。例如，Windows 下的环境变量如图 21-2 所示。

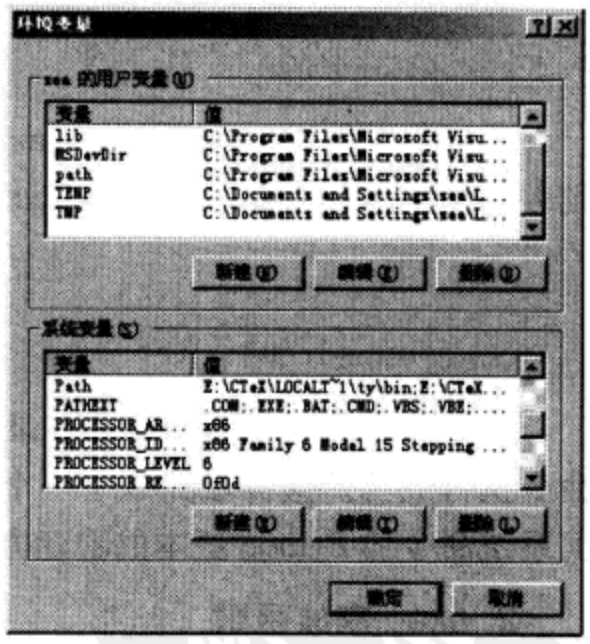


图 21-2　Windows 下的环境变量

当进程启动的时候，环境变量和其值也就确定下来了，只有当前进程能够修其环境变量。而无论是此进程的父进程还是子进程，都没有这种权利。在创建进程的时候，子进程将会得到当前父进程运行环境的一个副本。当子进程创建完毕后，对于父进程环境变量的修改就不会影响到子进程了。

在 Python 中，os 模块提供了 environ 属性，用来记录当前进程的运行环境。这是一个字典型的数据结构，其中，键（key）为环境变量的变量名，值（value）为环境变量的值。按照惯例，环境变量的变量名一般都全部用大写字母。environ 支持字典类型的所有相关操作。如果要获取当前环境变量中的 PATH 值，可以使用下面的代码：

```
import os

path = os.environ.get('PATH')
print path
```

输出将会打印系统中路径环境变量的值。当然，如果需要获取当前进程中所有环境变量，可以使用下面的代码：

更多资源请访问稀酷客(www.ckook.com)

```
import os

for key in os.environ.keys():
    print key, '\t', os.environ[key]
```

同样，可以对环境变量值进行设置。可以修改已有的环境变量，或者新生成一个环境变量：

```
import os

os.environ['key'] = 'value'
```

当 key 在环境变量中已经存在的时候，将会被改写。而当不存在的时候，则会新生成。在模块导入的时候，PYTHONPATH 环境变量就会影响到模块导入的查找路径。需要强调的是，这里修改或者新生成的环境变量仅仅对当前的进程是有效的。如果希望环境变量能够起全局作用，则需要将这些环境变量写入到系统配置中。

21.2.2　创建进程

创建进程是系统管理的重要组成部分。Python 语言提供了多种方式来创建进程，除了在 os 模块中有丰富的创建进程函数外，还有其他专门的模块如 subprocess 等来管理进程。在本节中，主要对 os 模块中的进程创建函数进行说明，对其他模块则在本章的其他节中进行说明。

在 os 模块中主要包括的 system 和 exec 家族函数，能够适应不同的创建进程需求，具体的将在下面进行介绍。同时，还有一些其他的创建进程方法，如 os.popen * 等函数，读者可以参看相关的文档，在这里就不具体介绍了。

1. system 函数

system 函数是用来创建进程的最快捷方式，其函数原型如下：

```
system(command)
```

此函数在新进程中执行 command 字符串命令。如果返回值为“0”，表示命令执行成功，否则表示失败。现在，可以通过 Python 来执行系统命令。如下面的代码：

```
import os

print os.system("dir")
```

结果将输出这段代码所在目录的文件和文件夹。注意到程序最后还打印出“0”，表示这次命令执行成功。这样就完成了一个 dir 系统命令的 Python 版本，在第 22 章中还可以看到针对任何系统命令的 Python 版本的构建方法。

2. exec 家族函数

exec 家族中有 8 个类似的函数（execl、execle、execlp、execlpe、execv、execve、execvp、execvpe）。虽然用它们都可以创建进程，但是和 system 函数还是有些不同。system函数实际上是调用系统内置的命令行程序来执行系统命令，所以在命令结束之后会将控制权返回给 Python 进程。但是，所有的 exec 函数在执行命令之后，将会接管 Python 进程，而不会将控制权返回。换句话说，Python 进程会在调用 exec 函数后终止。新生成的进程将会替换调用进程。这些函数都没有返回值，如果发生错误，将会触发OSError异常。

下面是一个简单的例子，演示了 execl(path, arg0, arg1, ...)函数的使用方法：

更多资源请访问稀酷客(www.ckook.com)

```
import os

notepad = 'c:\\windows\\not_notepad.exe'
os.execl(notepad)
notepad = 'c:\\windows\\notepad.exe'
os.execl(notepad)
```

脚本执行的结果如图 21-3 所示。

从图 21-3 所示的结果中可以看到，第一次调用 execl 函数时，由于命令不正确，因此Python触发了 OSError 异常。第二次调用 execl 函数时，打开了记事本程序，同时原来的Python解释器退出，这也是 exec 家族函数和 system 函数不同的地方。

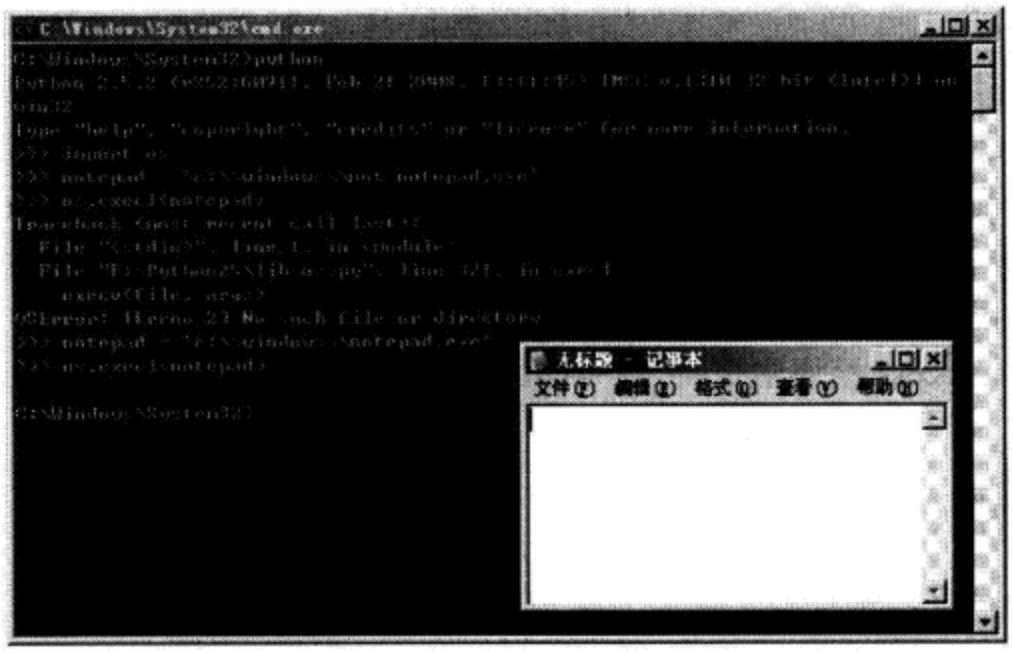


图 21-3　execl 函数示例程序

21.2.3　终止进程

同创建进程的过程一样，终止进程的过程也是系统管理的重要组成部分。Python 同样提供了多种不同的方式来终止进程。其实在前面就遇到过一种方法，那就是使用 return 语句。当 Python 脚本遇到最外层的 return 语句而退出的时候，这个进程也就终止了。除此之外，还有两种方法终止进程：sys. exit 和 os. abort。

1. sys. exit 函数

exit 函数是一种“温和”的终止进程方式，在程序退出之前会执行一些清理操作，同时将返回值给调用进程（一般是操作系统）。使用此返回值，系统可以判断程序是否是正常退出或者运行出了异常。下面是一个程序框架，用来读取用户给定的参数：

```
import sys

try:
filename = sys.argv[1]
print filename
except:
print "Usage:", sys.argv[0], "filename"
sys.exit(1)
return 0
```

当用户提供了文件名参数的时候，系统将打印文件名并返回“0”；否则将触发异常，并使用 exit 函数返回“1”。调用的函数通过检查这个值就可以知道用户是否提供了参数。

更多资源请访问稀酷客(www.ckook.com)

2. os. abort 函数

和 exit 函数不同的是，abort 函数是一种“暴力”的退出方式，将会直接给进程发送终止信号（SIGABORT 信号）。在默认情况下，这将会终止进程，同时不会做相关的清理操作。需要注意的是，可以使用 signal. signal( )来为 SIGABORT 信号注册不同的信号处理函数，从而改变其默认行为。

一般地，在使用 abort 函数的地方都可以使用 exit 函数来代替，这样的终止程序方式更加优雅，但是在 exit 不能终止程序或者运行时间过长的情况下，可以尝试使用 abort 函数来解决。

21.3　使用 subprocess 模块管理进程

在 21.2.2 和 21.2.3 小节中，对进程的创建和终止进行了说明。但那些只是基本的进程管理函数，无法满足更复杂的需求。为此，Python 中提供了 subprocess 模块进行高级进程管理。

subprocess 模块是作为进程管理的高级模块在 2.4 版本被引入的。subprocess 可以调用外部的系统命令创建新的子进程，同时连接到子进程的 input/output/error 管道上，并得到子进程的返回值。这个模块可以用来替代一些旧模块的方法，如 os. system、os. spawn*、os. popen*、popen2. *、commands. *等。subprocess 模块中提供了一个类和两个实用函数来管理进程，下面将分别进行介绍。

21.3.1　使用 Popen 类管理进程

subprocess 模块中高级进程的管理能力都是来自于 Popen 类的灵活使用。这源于该类中函数原型的丰富参数：

```
class Popen( args, bufsize = 0, executable = None, stdin = None, stdout = None, stderr = None,
preexec_fn = None, close_fds = False, shell = False, cwd = None, env = None, universal_newlines =
False, startupinfo = None, creationflags = 0)
```

其中，args 参数为要执行的外部程序。其值可以是字符串或者序列。除此之外，其他的类参数都有默认值，可以根据需要进行修改。

【例 21-1】演示使用 Popen 类的方法，并逐步解释其高级用法。这个例子的目的是向 www. sina. com. cn 发送 4 个探测 ping 报文，并获取相应的输出留待后面进行处理。这里采用的方式是直接调用系统中的 ping 命令。

Linux 下的程序代码如下：

```
import subprocess

pingP = subprocess. Popen(args = 'ping  - c 4 www. sina. com. cn', shell = True) #生成 ping 进程
print pingP. pid #打印进程 ID
print pingP. returncode #打印进程返回值
```

Windows 下的程序代码如下：

```
import subprocess

pingP = subprocess. Popen(args = 'ping - n 4 www. sina. com. cn', shell = True) #生成 ping 进程
pirnt pingP. pid #打印进程 ID
print pingP. returncode #打印进程返回值
```

说明　由于 Linux 和 Windows 下的 ping 命令格式不一样，所以这里为每个系统给出了相关代码，在后面的示例中，将主要以 Windows 下的代码为例进行介绍。

更多资源请访问稀酷客(www.ckook.com)

在上面的代码中，Popen 的第二个类参数为 shell 的值。在 Linux 下，当 shell 为“False”时，Popen 将调用 os. execvp 执行对应的程序。而 shell 为“True”时，如果命令为字符串，Popen 直接调用系统 shell 来执行指定的程序。

如果命令为一个序列，则其第一项是定义命令字符串，其他项为命令的附加参数。在 Windows下，无论 shell 为何值，Popen 都首先调用 CreateProcess 来执行指定的外部程序；若参数为序列，则需要先用 list2cmdline 转化为字符串。在这里为了方便起见，都是使用的字符串作为类的命令参数。代码的最后两行分别打印出所生成子进程的进程 pid 和返回值。

将上面的代码保存为 pingP_1. py 文件，在 Windows 下代码的一种输出结果如下所示（Linux 下的输出结果会稍有不同）：

```
C:\ >python pingP_1.py
2956
None

C:\ >
Pinging jupiter.sina.com.cn [121.194.0.210] with 32 bytes of data:

Reply from 121.194.0.210: bytes =32 time =26ms TTL =52
Reply from 121.194.0.210: bytes =32 time =26ms TTL =52
Reply from 121.194.0.210: bytes =32 time =30ms TTL =52
Reply from 121.194.0.210: bytes =32 time =30ms TTL =52

Ping statistics for 121.194.0.210:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 26ms, Maximum = 30ms, Average = 28ms
C:\ >
```

在这段代码输出结果中，上面的输出部分“2956”和“None”是子进程的进程 ID 和返回值。下面则是外部 ping 程序的输出结果。多次执行这段代码，进程 ID 将会不断变化。而在子进程的返回值中，输出“None”表示此子进程还没有终止。

从输出结果中可以看到，代码生成一个子进程并执行 args 中指定的命令，然后继续执行下面的语句。由于网络应用的延时，这就使得在打印出了进程 ID 和返回值后才输出外部命令的结果。

由于外部程序是在一个新生成的子进程中执行的，所以如果不加以限制，则有可能会将原进程和子进程的输出结果混淆。如果需要等待该子进程的结束，可以使用 Popen 类中的 wait( )函数。

【例 21-2】演示使用 Popen 类中的 wait( )函数等待子进程结束的方法。

```
#filename: pingP_2.py
import subprocess

pingP = subprocess.Popen(args ='ping  - n 4 www.sina.com.cn', shell = True) #生成 ping 进程
pingP.wait() #等待进程完成
pirnt pingP.pid #打印进程 ID
print pingP.returncode #打印进程返回值
```

wait( )函数等待子进程的完成，将会返回子进程的返回值。下面是代码运行的输出结果：

```
C:\ >python pingP_2.py
Pinging jupiter.sina.com.cn [121.194.0.209] with 32 bytes of data:

Reply from 121.194.0.209: bytes =32 time =26ms TTL =52
Reply from 121.194.0.209: bytes =32 time =28ms TTL =52
Reply from 121.194.0.209: bytes =32 time =29ms TTL =52
Reply from 121.194.0.209: bytes =32 time =26ms TTL =52
```

更多资源请访问稀酷客(www.ckook.com)

```
Ping statistics for 121.194.0.209:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 26ms, Maximum = 29ms, Average = 27ms
2820
0
C:\ >
```

从上面的输出结果可以看到，现在子进程的进程 ID 和返回值已经在子进程输出结果的后面了。同时，子进程的返回值已经变为“0”，表示子进程已经顺利退出。

从【例 21-1】和【例 21-2】中可以看到，子进程被创建后，其标准输入、标准输出和标准错误处理都和原进程没有关系。如果需要管理子进程的输入和输出，可以改变 Popen 类中的 stdin、stdout 和 stderr 等类的参数。这是非常方便的，如果使用以前的进程创建方法，则需要将输入和输出重定向。

【例 21-3】 演示收集子进程的输出结果。

```
#filename pingP_3.py
import subprocess

pingP = subprocess.Popen(args = 'ping  - n 4 www.sina.com.cn', shell = True, stdout = sub-
process.PIPE)
pingP.wait() #等待进程完成
print pingP.stdout.read() #读取进程的输出信息
print pingP.pid
print pingP.returncode
```

- 在 Popen 的类参数中，stdin、stdout、stderr 分别用来指定程序标准输入、标准输出和标准错误的处理器，其值可以为“PIPE”、文件描述符和“None”等。默认值都为“None”。
- 在获取输出结果后，pingP. stdout（<open file ‘<fdopen>’, mode ‘rb’ >）成为一个可读的文件对象，可以使用相应的文件操作函数来读取。

从【例 21-3】输出结果来看，和【例 21-2】的结果是一样的，但是两者是完全不同的。在【例 21-2】中，子进程的输出结果没有得到控制。但在【例 21-3】中，其子进程的输出结果被收集起来了。如果将脚本中的“print pingP. stdout. read( )”这句代码注释掉，程序的输出结果如下：

```
C:\ >python pingP_3.py
3528
0
C:\ >
```

另外一种方式是采用 Popen 类提供的 communicate 方法。为了能够演示 communicate 方法的使用，将外部命令改为了 cat（注意：这是在 Linux 下的系统命令），这个命令在不提供参数的情况下，从标准输入中读取数据并将其原样输出。下面是代码示例：

```
#filename: catP.py
import subprocess

pingP = subprocess.Popen(args = 'cat', shell = True, stdin = subprocess.PIPE)
pingPout, pingPerr = pingP.communicate(input = 'Hello Python')
print pingPout.read() #读取进程的输出信息
```

communicate( ) 方法返回的是一个（stdout, sterr）。需要注意的是，因为数据都是缓存在内存中，所以当数据量很大的时候不要使用这个方法。

更多资源请访问稀酷客(www.ckook.com)

通过灵活的设置类参数和使用类中的方法，可以有效地管理进程。除了上面介绍的外，还有其他类参数和类方法，读者可以参考相关的文档。

21.3.2 调用外部系统命令

subprocess模块还提供了两个实用函数来直接调用外部系统命令：call( ) 和check_all( )。两者是对上面Popen类构造函数使用方法的一种简化。其参数列表和Popen构造函数的参数列表是一样的。call函数会直接调用命令生成子进程，并且等待子进程结束，然后返回子进程的返回值。check_call函数和call函数的主要区别在于：如果返回值不为“0”，则触发CallProcessError异常。返回值保存在这个异常对象的returncode属性中。下面是一个简单的例子：

```
import subprocess

retcode = subprocess.call(["ls", "-l"]) #调用 ls  -l 命令
```

在下面还可以看到，这个实用函数可以用来替代如os.system( )等函数，而且已经比较好的实现了os.system( )函数的功能。

21.3.3 替代其他进程创建函数

subprocess作为进程管理的高级模块，可以替代原有的Python中的系统命令调用方法。在前面介绍过os.system( )函数，一种可能的替代方式如下：

```
sts = os.system("cmd")
 === >
p = subprocess.Popen("cmd", shell = True)
sts = os.wait(p.pid, 0)
```

在上面的处理中，并没有考虑到命令执行异常的情况，一种更加容错的方法如下：

```
try:
returncode = subprocess.call("cmd", shell = True)
if returncode < 0:
    print "Child was terminated by signal ", -returncode #子进程被信号中断
else:
    print "Child returned with code ", returncode #子进程正常返回
except OSError, e:
    print "Execution failed:", e          #发生了异常
```

这里的处理考虑了多种可能的情况，包括子进程被信号中断的情况。当子进程被信号中断的时候，将返回信号的负值。在上面代码的第4行中，取代码的负值则得到了相关的信号值。相对于os.system( )函数来说，后面这种处理方法有如下一些好处。

（1）没有使用C标准函数system( )，从而避免了其使用的一些限制。

（2）没有像os.system( )一样隐式的调用系统shell。

（3）可以使用参数列表，不需要对命令进行转义。

（4）可以更好地处理返回值，如可以识别进程的信号中断。

现在还是有不少人使用os.system( )，除了其方法是最早出现的之外，主要是简单方便。而subprocess的call( )方法也是同样简单易用的。除了能够替代os.system( )函数外，其他很多进程创建函数也是可以被subprocess模块中的类和方法替代的，实现方法类似，此处不再赘述。

21.4 进程间的信号机制

信号处理也是进程间通信的一种方式。合理有效地利用信号，可以使进程的处理更加

更多资源请访问稀酷客(www.ckook.com)

灵活便捷。信号是操作系统的一种软件中断，采用异步方式传递给应用程序相关消息。例如，终端用户按下中断键，会通过信号机制生成一个信号。应用程序针对特定的信号可以采取相应的措施。在 Python 中，对每个信号都有一个默认的信号处理程序。如在 21.2.3 节介绍过的 os.abort 函数，其实质也是向应用程序发送特定的信号。应用程序收到这样的信号后，可以使用自定义的信号处理程序。当没有自定义信号处理程序时，则会采用默认的信号处理程序。

信号机制最早由 UNIX 操作系统引进，现在已经在各个操作系统中都有类似的实现，但是由于历史原因，信号机制的实现并不完全相同。也就是说，每个系统都实现了自己特有的信号集合。signal 模块并不是使用 Python 语言来实现的，而是直接依赖于具体的系统平台，所以 signal 模块仅仅只包含系统中定义的信号，对于其他信号则是忽略的，这一点在处理信号的时候是需要注意的。

21.4.1　信号的处理

在 signal 模块中提供了管理信号的处理方法。其核心函数是 signal.signal( ) 函数，作用是为中断信号注册指定的信号处理函数。当程序收到了在其中注册的信号后，就会执行指定的信号处理函数。此方法有两个参数，分别是需要注册的信号和对应的信号处理函数。在此方法的第二个参数中，可以是系统定义的某个信号处理函数，也可以是在 signal 模块中已经预设的一些信号处理函数。现在已经有两个预设的信号处理函数，一个是 SIG_DFL，这是信号的默认处理函数；另一个是 SIG_IGN，将简单的忽略信号。

程序在运行的时候，一般可以通过按下 Ctrl + C 键来终止进程。如果希望能够在程序退出的时候做一些维护工作。此时就可以使用信号来完成。还是以【例 21-1】为例来说明，原来是发送 4 个 ping 探测报文，但是现在只需要发送两个探测报文。在不修改原来代码的情况下，可以通过按下 Ctrl + C 键发送 SIGINT 信号来完成。在现有的情况下，这种方式会触发键盘中断异常，从而终止 Python 程序的运行。程序输出结果如下所示：

```
C:\ >python pingP_2.py
Pinging jupiter.sina.com.cn [121.194.0.206] with 32 bytes of data:

Reply from 121.194.0.206: bytes=32 time=26ms TTL=52
Reply from 121.194.0.206: bytes=32 time=26ms TTL=52

Ping statistics for 121.194.0.206:
    Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 26ms, Maximum = 26ms, Average = 26ms
Control-C
Traceback (most recent call last):
  File "pingP_2.py", line 5, in <module>
    pingP.wait()
  File "C:\Python25\lib\subprocess.py", line 834, in wait
    obj = WaitForSingleObject(self._handle, INFINITE)
KeyboardInterrupt
```

其中的“Control-C”是控制按键，这里为按下 Ctrl + C 键时候的输出字符。由于子进程的 ping 程序也是用于处理 SIGINT 信号的，所以这里可以在收到这个信号的时候打印统计信息。而对于调用进程而言，对 SIGINT 信号的默认处理程序就是触发键盘中断异常。

这种中断程序运行的方式过于简单粗暴。如果在代码的后面还需要进行其他处理，也不会得到执行。此时，可以采用信号机制。

更多资源请访问稀酷客(www.ckook.com)

【例 21-4】演示信号机制的使用方法。

```
#filename: pingP_signal.py
import subprocess
import signal

def sigint_handler(signum, frame): #SIGINT 信号处理函数
    print "In signal SIGINT handler"
signal.signal( signal.SIGINT, sigint_handler) #设置 SIGINT 信号处理函数

pingP = subprocess.Popen(args = 'ping  - n 4 www.sina.com.cn', shell = True)
pingP.wait() #等待子进程完成,后面在这里会被中断
pirnt pingP.pid
print pingP.returncode
```

- 与【例 21-1】相比，代码主体部分并没有改动，主要是加入了信号处理部分。即在代码段中的第 5 行～第 7 行实现并注册了信号处理函数。
- 第 5 行～第 6 行是信号处理函数的定义部分。函数有两个参数，第一个是信号编号，第二个参数包含当前的堆栈帧。信号编号是识别信号的唯一标识，在同一个操作系统中是唯一定义的，而在不同的操作系统中可能相同。由于仅仅是示例代码，所以只是打印了一些信息。在实际代码中，信号的处理过程要复杂得多。
- 第 7 行代码将 SIGINT 信号和 sigint_handler 信号处理函数联系起来，使得程序在收到相关信号的时候可以找到正确的注册函数入口。在 signal 方法中的第一个参数使用的是 SIGINT，而不是普通的数值，这可以提高代码的可移植性。

【例 21-4】代码运行后的输出结果如下：

```
C:\ >python pingP_signal.py

Pinging jupiter.sina.com.cn [121.194.0.207] with 32 bytes of data:

Reply from 121.194.0.207: bytes =32 time =26ms TTL =52
Reply from 121.194.0.207: bytes =32 time =26ms TTL =52

Ping statistics for 121.194.0.207:
    Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 26ms, Maximum = 26ms, Average = 26ms
Control-C
In signal SIGINT handler
3344
-1073741510
```

- 和原来没有信号处理输出结果的【例 21-1】相比，主要变化在后面的 3 行中。当按下 Ctrl + C 键的时候，系统向程序发出了键盘中断消息，但是程序并没有就此结束，而是进入已经关联好的信号处理函数。这里将会打印信号处理函数中的内容。
- 后面还输出了子进程的进程 ID 和返回值，可以看到程序在中断后继续执行。程序会在收到信号的时候将当前程序挂起，去处理信号中断函数。当信号处理完毕后，程序会从中断位置继续执行后面的代码。当然，有些代码在中断后，其后面的代码是不能继续执行的（例如 Linux 中的系统调用），在编写信号代码的时候需要注意这一点。注意到后面的返回值也和以前的不一样，这是因为子进程也被系统中断了。

signal 模块中提供了一个实用函数 getsignal，用来查询特定信号值所关联的信号处理函数。返回值是一个可调用的 Python 对象，或者是 SIG_IGN、SIG_DFL、None 中的一个。例如，在上

更多资源请访问稀酷客(www.ckook.com)

面针对 SIGINT 信号使用此函数，可以得到如“ < function sigint_handler　at 0x00B4A930 > ”这样的值。

21.4.2　信号的使用规则

信号的使用是很方便的，但是在使用信号的时候需要注意如下事项。

（1）信号的支持和实现在不同的系统上是不一样的，所以有些程序在不同的系统上表现可能会不一样。

（2）除了 SIGCHLD 信号（如果信号集支持的话）以外，信号处理函数一旦被设置后就不会改变，除非显式地重新设置。

（3）没有办法在临界区临时的屏蔽掉信号。

（4）尽管信号是一种异步的信息传递机制，但是实际上在进行长时间计算的时候使用信号，可能会产生一定的延时。

（5）当程序在执行 I/O 操作的时候收到信号中断，有可能使得在信号处理函数执行完毕后触发异常，或者直接触发异常。如下面的代码：

```
#filename: io_signal.py
import signal

def sigint_handler(signum, frame): #SIGINT 信号处理函数
    print "In signal SIGINT handler"
signal.signal( signal.SIGINT, sigint_handler) #注册 SIGINT 信号处理函数

while True:
ret = raw_input('Prompt >'> #I/O 处理,将在这里发生中断
print "Hello, ", ret
```

以上代码的输出结果如下所示：

```
C:\ >python io_signal.py

H:\switch\04.sources\zhouwei >python test_hh.py
Prompt > Python
Hello,  Python
Prompt > Traceback (most recent call last):
  File " io_signal.py", line 8, in <module >
    ret = raw_input("Prompt > ")
KeyboardInterrupt
```

从上面的输出结果可以看出，在收到信号后，程序还是处于 I/O 中，所以直接触发了键盘中断异常。在设计 I/O 类程序的时候需要注意这一点。

（6）Python 语言中的信号是使用 C 语言实现的，而 C 的信号处理函数总是会返回的，所以没有必要去关注同步错误，如 SIGFPE 和 SIGSEGV。

（7）Python 语言已经为部分信号注册了处理函数，如在前面的 SIGINT 信号，默认情况下，就会转化为 KeyboardInterrupt 异常。

（8）当信号和线程同时使用的时候，必须要小心。如果使用不当，可能会出现意想不到的问题。在同时使用信号和线程的时候，特别要记住的是：总是在主线程中执行 signal( ) 函数，所以不能使用线程作为线程间的通信方式。

21.5　多线程概述

使用多线程处理技术，可以有效地实现程序并发，优化处理能力。同时，功能的更小划分，

更多资源请访问稀酷客(www.ckook.com)

可以使代码的可重用性更好。本节首先介绍多线程的概念，然后对 Python 中对于线程的支持进行介绍，主要是 threading 类。最后将对多线程中的常见问题——同步进行介绍。

21.5.1　什么是多线程

多线程使得系统可以在单独的进程中执行并发任务。虽然进程也可以在独立的内存空间并发执行，但生成一个新进程必须为其分配独立的地址空间，并维护其代码段、堆栈段和数据段等，这种开销是很昂贵的。另外，进程间的通信实现也不方便。在程序功能日益复杂后，需要有更好的系统模型来满足要求。由此产生了线程。

线程是“轻量级”的，一个进程中的线程使用同样的地址空间，且共享许多资源。启动线程的时间远远小于启动进程的时间和空间，同时，线程间的切换也要比进程间切换快得多。由于使用同样的地址空间，所以在线程之间的数据通信比较方便。一个进程下的线程之间可以直接使用彼此的数据。当然，这种方便性也会带来一些问题，特别是同步问题。

多线程对于那些 I/O 受限的程序特别适用。其实多线程使用的一个重要目的，就是最大化 CPU 的资源利用。当某一线程在等待 I/O 的时候，另外一个线程可以占用 CPU 资源。如最简单的 GUI 程序，一般需要有一个任务支持前台界面的交互，同时还要有一个任务支持后台的处理。这时候，就适合采用线程模型，因为前台界面一般是在等待用户的输入操作，包括键盘输入和鼠标单击等。除此之外，多线程在网络领域和嵌入式领域应用也是比较多的。

21.5.2　线程的状态

在一个线程的生命周期内，会在不同的状态之间转换。在任何一个时刻，线程总是处于某种线程状态中。虽然不同的操作系统可以实现不同的线程模型，定义不同的线程状态，但是总的说来，一个线程模型中下面的状态是通用的。

.（1）就绪状态：线程已经获得了除 CPU 外的其他资源，正在参与调度，等待被执行。当被调度选中后，将立即执行。

（2）运行状态：占用 CPU 资源，正在系统中运行。

（3）休眠状态：暂时不参与调度，等待特定的事件发生，如 I/O 事件。

（4）中止状态：线程已经运行结束，等待系统回收线程资源。

Python 使用全局解释器锁（GIL）来保证在解释器中仅仅只有一个线程，并在各个线程之间切换。当 GIL 可用的时候，处于就绪状态的线程在获取 GIL 后就可以运行了。线程将在指定的间隔时间内运行。当时间到期后，正在执行的线程将重新进入就绪状态并排队。GIL 重新可用并且为就绪状态的线程获取。当然，特定的事件也有可能导致中断正在运行的线程。具体的线程状态转移将在下面进行详细介绍。

21.5.3　Python 中的线程支持

现在，Python 语言中已经为各种平台提供了多线程处理能力，包括 Windows、Linux 等系统平台。在具体的库上，提供了两种不同的方式。一种是低级的线程处理模块 thread，仅仅提供了一个最小的线程处理功能集，在实际的代码中最好是不要直接使用该模块。另外一种是高级的线程处理模块 threading，现在大部分应用的线程实现都是基于此模块的。threading 模块是基于 thread 模块的，部分实现思想也来自于 Java 的 threads 类。

多线程设计的最大问题是如何协调多个线程，因此，在 threading 模块中，提供了多种数据同步的方法。为了能够更好的实现线程同步，Python 中提供了 Queue 模块，用来同步线程。在 Queue 模块中，有一个同步的 FIFO 队列类型，特别适合线程之间的数据通信和

更多资源请访问稀酷客(www.ckook.com)

同步。

由于大部分程序不需要有多线程处理的能力，所以在 Python 启动的时候并不支持多线程。也就是说，Python 中支持多线程所需要的各种数据结构特别是 GIL 还没有创建。当 Python 虚拟机启动的时候，多线程处理并没有打开，而仅仅只支持单线程。这样做的好处是使得系统处理更加高效。只有当程序中使用了如 thread. start_new_thread 等方法的时候，Python 才知道需要有多线程处理的支持。这时，Python 虚拟机才会自动创建多线程处理所需要的数据结构和 GIL。

21.6　生成和终止线程

生成和终止线程是线程管理的关键步骤。在前面说过，只有生成了线程后，Python 虚拟机才会开启对于多线程处理的支持。由于 Python 对于线程的支持既有低级方法，也有高级方法，所以生成和终止线程的方式也是多样的。本节将详细介绍这些内容。

21.6.1　使用 thread 模块

thread 模块作为低级模块，虽然不被推荐直接使用，但是因为其用法非常简单，在某些简单场合也是可以试试的。其中的核心是 start_new_thread 方法，可以用来生成线程。此方法接受一个函数对象作为参数，同时还有此函数的参数列表，另外还有一个可选的字典参数。此方法将会返回生成新线程的标识符。

【例 21-5】一个简单的使用线程的例子。

```
#filename: thread_1.py
import thread
import time

def worker(index,create_time): #具体的线程
print (time.time()-create_time)," \t \t",index
print "Thread %d exit..." %  (index)

for index in range(5):
thread.start_new_thread(worker, (index,time.time())) #启动线程

print "Main thread exit..."
```

- 【例 21-5】代码片段将会生成 5 个子线程。一般说来，在进程启动的时候，会生成一个默认的线程，这个线程叫做主线程。这里的主线程就是 Python 解释器。由主线程衍生出来的线程称为子线程，它们也有自己的程序入口等。在大部分线程模型中，除了主线程比较特殊以外，其他线程之间并没有明显的从属关系或者层次关系。
- 在第 9 行和第 10 行中，在 for 循环中调用了 thread 模块的 start_new_thread 方法 5 次，从而生成了 5 个子线程。在 start_new_thread 函数中，后面有两个参数。其中一个是 worker 函数，这是在代码中定义的函数。另外一个是这个 worker 函数的参数，注意到这里的参数是一个元组。在 worker 的两个参数中，第一个参数是 index 值，而第二个是线程创建的时间。
- 在 worker 函数中，为了简单起见，仅仅是打印出了本线程运行的时间和其 index 值。其中，运行时间是通过将打印时的时间和创建的时间进行相减而得到的。

下面是程序的两次运行结果：

更多资源请访问稀酷客(www.ckook.com)

```
#第一次运行结果
0.0        0
Thread 0 exit...
0.0        1
Thread 1 exit...
Main thread exit...
0.0        2
Thread 2 exit...
0.0        3
Thread 3 exit...
0.0160000324249          4
Thread 4 exit...
#第二次运行结果
Main thread exit...
0.0        0
Thread 0 exit...
0.0        1
Thread 1 exit...
0.0159997940063          2
Thread 2 exit...
0.0159997940063          3
Thread 3 exit...
0.0159997940063          4
Thread 4 exit...
```

- 从两次运行的输出结果来看，结果并不一样，这也是多线程程序的一个特点。因为线程之间的调度是很难预知的。
- 其中有些线程经过的时间为“0.0”，这主要是因为线程从创建至打印经过的时间太短，已经不在时间的精度范围之内，从而显示为“0.0”。这并不代表线程的创建不需要时间，仅仅是这个时间很短罢了。
- 从输出结果来看，打印出来的 index 值还是有序的。这同样表明这些线程之间具有并发的特性。其主要原因是因为这里的时间太短，基本上线程创建出来就被调度而执行了打印语句。当然，如果执行次数足够多的话，有可能会出现乱序的情况，只是这种可能性比较小。

【例 21-6】接上例。对【例 21-5】的代码稍加修改，使得线程的并发性得到体现。

```
#filename: thread_2.py
import thread
import time

def worker(index,create_time):
    time.sleep(1) #休眠 1 秒钟
print (time.time()-create_time)," \t \t",index
print "Thread %d exit..." %  (index)

for index in range(5):
thread.start_new_thread(worker, (index,time.time())) #启动 5 个线程

print "Main thread exit..."
```

- 对【例 21-5】的修改主要是在线程中加入了一个休眠：time.sleep（1），使得每个线程在调度之前都休眠 1 秒钟的时间。

现在再来看看两次执行【例 21-6】代码片段的结果：

```
#第一次运行结果
Main thread exit...
1.0        0
Thread 0 exit...
```

更多资源请访问稀酷客(www.ckook.com)

```
1.0      4
Thread 4 exit...
1.0150001049     2
Thread 2 exit...
1.0150001049     1
Thread 1 exit...
1.0150001049     3
Thread 3 exit...
#第二次运行结果
Main thread exit...
1.01600003242    1
Thread 1 exit...
1.01600003242    0
1.01600003242    2
Thread 2 exit...
1.01600003242    4
Thread 4 exit...
1.01600003242    3
Thread 3 exit...
Thread 0 exit...
```

- 多次运行【例 21-6】，index 的值会出现不同的排序。另外，注意到第二次运行结果，连前面的时间也是无序的，这并不表明结束时间早的反而在后面打印，而这是与如线程调度、输出缓冲有关。从这里，可以看出线程的并发性。

在上面的代码中没有出现退出线程的语句或者函数。实际上，当线程函数执行结束的时候，线程就已经默认终止了。当然，也可以使用 thread 模块中的方法显式退出方法 exit，这将会触发 SystemExit 异常。如果此异常没有被捕获的话，线程将会安静终止。注意，这里的线程终止并不会影响到其他线程的运行。

表 21-3 列出了 thread 模块中的常用方法。

表 21-3　thread 模块中的常用方法

方法名称	说　明
start_new_thread	生成一个新线程并且返回其标识符的值
exit	退出线程，触发一个 SystemExit 异常
get_ident	获取当前线程的标识符
allocate_lock	返回一个锁对象
interrupt_main	在主线程中触发一个 KeyboardInterrupt 异常
stack_size	返回线程堆栈的大小

21.6.2　使用 threading. Thread 类

使用 threading 模块来创建线程是很方便的。简单地说，只要继承 threading. Thread，然后在 __init __方法中调用 threading. Thread 类中的__init __方法，重写类的 run 方法就可以了。

【例 21-7】演示一个使用 threading 的多线程程序框架。

```
import threading

class ThreadSkeleton(threading. Thread):

    def __init __(self): #线程构造函数
        threading. Thread. __init __(self)
```

更多资源请访问稀酷客(www.ckook.com)

```
    def run(self):
        pass

thread = ThreadSkeleton ()
thread.start()
```

- 程序的主要步骤是构建一个继承自 threading. Thread 的类，然后实例化此对象，并调用相应的方法来生成线程。
- 在类的构造函数__init __中，调用了 threading. Thread 类的构造函数。这里使用的都是 Thread 类构造函数中的默认值。当然，也可以传递特定的参数以改变其默认行为。【例 21-7】仅是为了演示，因此构造函数并没有实际作用，即使将其去掉也不影响生成多线程程序框架。在实际代码中，可以将线程运行之前的一些准备工作放在这里。
- run 方法中的代码是每个线程将要执行的部分。这里只是一个多线程程序框架，并没有实际代码。
- 在代码片段的最后两行中创建了一个 ThreadSkeleton 对象，并调用 start 方法执行类在 run 方法中的代码，主线程继续执行后面的代码。在使用 start 方法的时候需要注意，此方法对于一个线程最多只能调用一次。

【例 21-8】接【例 21-5】。将【例 21-5】改为使用 threading 模块。

```
#filename: threading_1.py
import threading
import time

class ThreadDemo(threading.Thread):
    def __init__(self, index, create_time): #线程构造函数
        threading.Thread.__init__(self)
        self.index = index
        self.create_time = create_time
    def run(self): #具体的线程运行代码
        time.sleep(1) #休眠 1 秒钟
        print (time.time()-self.create_time),"\t",self.index
        print "Thread % d exit..." % (self.index)

for index in range(5):
    thread = ThreadDemo (index, time.time())
    thread.start() #启动线程
print "Main thread exit..."
```

- 从【例 21-8】中可以看到，这里的代码结构是在【例 21-5】的多线程程序框架的基础上扩充的，包括类中的构造函数和 run 方法等。
- 第 15 行 ~ 第 17 行代码生成了 5 个独立的线程，并传递了参数值。

本段代码的一种运行结果如下：

```
Main thread exit...
1.0      0
Thread 0 exit...
1.01600003242    2
Thread 2 exit...
1.01600003242    1
Thread 1 exit...
1.01600003242    3
Thread 3 exit...
1.01600003242    4
Thread 4 exit...
```

代码运行的结果和【例 21-5】的输出结果是类似的，多次运行结果将使得这种线程的并发性表现得更加明显。

更多资源请访问稀酷客(www.ckook.com)

threading. Thread 也没有显式的线程终止方法，当 run 方法运行结束时，这个线程就默认中止了。这和 thread 模块的处理是一样的。

表 21-4 列出了 threading 模块中 Thread 类的常用方法。

表 21-4　threading 模块中 Thread 类的常用方法

方法名称	介　绍
start	开始运行生成的线程实例
run	重载此方法，作为线程的运行部分
join	等待线程的结束
getName	返回线程的名字
setName	设置线程的名字
isAlive	查看线程是否还是活动的
isDaemon	返回线程的是否后台运行标志
setDaemon	设置线程的后台运行标志

- 线程的名字不但可以通过表 21-4 中的方法来设置，也可以在 Thread 类的 __init __构造函数中设置。如果不设置线程的名字，系统将使用如“Thread-N”的名字。

【例 21-9】演示使用构造函数来设置线程名字的方法。

```
class ThreadDemo (threading.Thread):
    def __init__(self, index, create_time, thread_name):
        threading.Thread.__init__(self, name = thread_name)
        self.index = index
        self.create_time = create_time
```

21.7　管理线程

在线程生成和线程终止之间，就是线程的运行时间。这段时间可以对指定的线程进行管理，从而更好的利用线程的并发特性。

21.7.1　线程状态转移

当线程生成后，线程就在不同的线程状态之间转移。在前面已经给出的大部分线程模型中基本上都共有 4 种状态：就绪状态、运行状态、睡眠状态和中止状态。图 21-4 给出了在这 4 种线程状态之间的线程转移示意图。

需要说明的是，图 21-4 中的休眠状态并不仅仅是通过 sleep 后得到的状态，而是包括如等待资源等其他状态。这里的睡眠状态仅仅是一个资源等待示意状态（睡眠也可以看做是等待 CPU 资源）。图中仅仅表示了以调用 sleep 作为到达睡眠状态的方法。

图 21-4 是真实线程转移的一个简化，从中可以看到各个状态之间的转移条件和方法。以【例 21-8】来说，thread 调用 start 方法后将生成线程，线程状态变为就绪状态。在就绪状态中，如果此线程获取了 GIL，将转变为运行状态，执行在 run 方法中定义的代码。在执行过程中，遇到 sleep 函数，线程将进入睡眠状态。当过了一定时间后，系统将唤醒

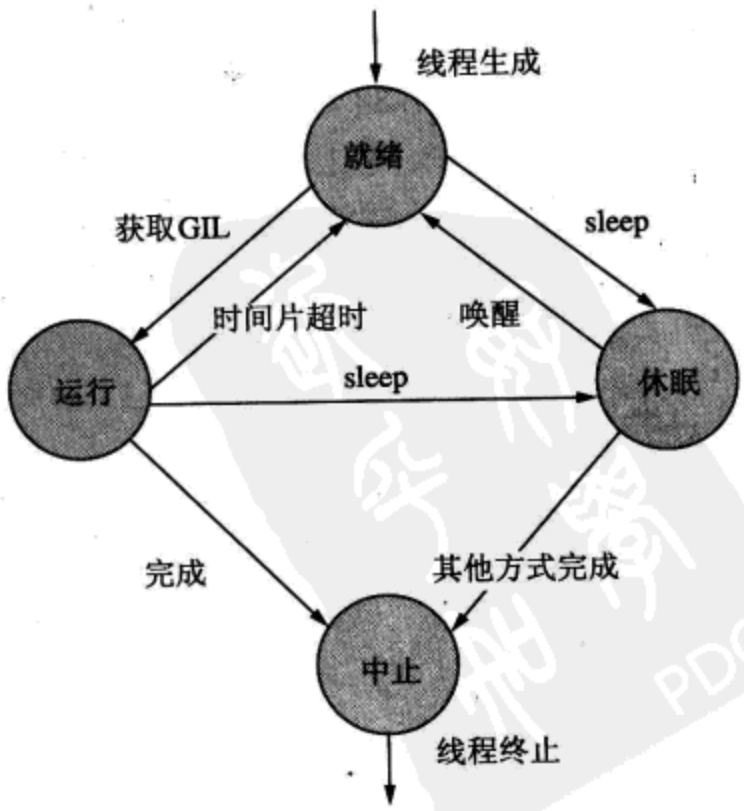


图 21-4　线程状态转移图

更多资源请访问稀酷客(www.ckook.com)

线程，并重新进入就绪状态。当再度获取 GIL 将所有的代码运行完毕后，线程将进入线程中止状态，然后视情况释放线程资源。

21.7.2　主线程对子线程的控制

在上面的所有多线程示例程序中可以看到，打印信息“Main thread exit...”一般出现在打印信息（如“Thread 4 exit...”）之前。也就是说，当主线程生成子线程之后，主线程将会继续执行，而不会等待子线程的结束，但是很多时候，可能需要主线程等待所有子线程的完成。就【例 21-8】来说，也就是要在所有的子线程输出信息打印完毕后才执行主线程后面的代码。这可以通过使用 Thread 类中的 join 方法来实现。

【例 21-10】接【例 21-9】。演示主线程等待所有子线程的完成，修改的部分如下。

```
threads = []
for index in range(5):
    thread = ThreadSkeleton(index, time.time())
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join() #等待线程完成

print "Main thread exit..."
```

代码片段中使用 threads 来保存已经生成的线程。在后面对每个线程都调用了 join 方法，让主线程等待子线程的完成。

【例 21-10】代码的一种输出结果如下：

```
1.0      2
Thread 2 exit...
1.0      1
Thread 1 exit...
1.0      0
Thread 0 exit...
1.0150001049     4
Thread 4 exit...
1.0150001049     3
Thread 3 exit...
Main thread exit...
```

多次运行【例 21-10】代码后可以看到，主线程的输出信息已经放到了最后。这表明主线程在所有子线程都结束后才继续执行。

join 方法还有一个可选的 timeout 超时参数。如果线程没有正常退出或者通过某个异常退出，且超时的情况下，主线程就不再等待子线程了。由于 join 的返回值始终是“None”，所以在 join 方法中有超时参数的情况下，join 返回后无法判断子线程是否已经结束。这时，必须使用 Thread 类中的 isAlive 方法来判断是否发生了超时。

使用 join 方法的时候，需要注意以下事项。

（1）在超时参数不存在的情况下，join 操作将会一直阻塞，直到线程终止。

（2）一个线程可以多次使用 join 方法。

（3）线程不能在自己的运行代码中调用 join 方法，否则会造成死锁。

（4）在线程调用 start 方法之前使用 join 方法，将会出现错误。

21.7.3　线程中的局部变量

有时候需要在每个线程中使用各自独立的变量，一个显而易见的方法就是每个线程都使用自己

更多资源请访问稀酷客(www.ckook.com)

的私有变量。为了方便，Python 中提供了一种简单的机制来解决这个问题。这就是 threading. local。其使用方法也很简单，其成员变量在每个线程中都是不同的。

【例 21-11】演示 threading. local 的使用方法。

```
import threading
import random, time

class ThreadLocal():
    def __init__(self):
        self.local = threading.local() #生成 local 数据对象

    def run(self):
        time.sleep(random.random()) #随机休眠时间
        self.local.number = []
        for i in range(10):
            self.local.number.append(random.choice(range(10)))
        print threading.currentThread(), self.local.number

threadLocal = ThreadLocal()
threads = []
for i in range(5):
    t = threading.Thread(target = threadLocal.run)
    t.start() #启动线程
    threads.append(t)
for i in range(5):
    threads[i].join #等待线程完成
```

- 在 ThreadLocal 类中使用 threading. local( )生成了类的局部变量。此局部变量将在不同的线程中保存为不同的值。
- ThreadLocal 类中的 run 方法主要是将 10 个随机数放到前面生成的局部变量中并打印出来。

下面是【例 21-11】代码执行的一种结果：

```
<Thread(Thread-5, started) > [4, 4, 4, 1, 7, 8, 3, 4, 0, 7]
<Thread(Thread-1, started) > [7, 5, 8, 4, 2, 9, 0, 6, 4, 3]
<Thread(Thread-4, started) > [7, 6, 5, 0, 6, 0, 2, 5, 0, 4]
<Thread(Thread-2, started) > [1, 4, 4, 9, 9, 1, 2, 3, 7, 6]
<Thread(Thread-3, started) > [3, 1, 3, 3, 1, 0, 2, 1, 6, 4]
```

从上面的输出结果中可以看到，每个线程都有着自己的不同的值。

21.8　线程之间的同步

由于同一进程中的所有线程都是共享数据的，如果对线程中数据的并发访问不加以限制，结果将不可预期。在严重的情况下，还会产生死锁。为了解决这个问题，需要允许线程独占性的访问共享数据，这就是线程同步。本节将详细介绍同步的概念和 Python 下具体的实现。需要注意的是，这些问题在进程中也是存在的，只是在多线程环境下更加常见而已。在后面介绍的解决线程之间数据同步问题的方法，对进程也是适用的。在本节中，将具体介绍 4 种线程同步机制，包括锁机制、条件变量、信号量和同步队列。每种同步机制都有自己的优缺点，读者可以根据需要从中选择。

21.8.1　临界资源和临界区

临界资源是指一次只允许一个线程访问的资源，包括如打印机一类的硬件资源和互斥变量一

更多资源请访问稀酷客(www.ckook.com)

类的软件资源。对临界资源的共享只能够采用互斥的方式。也就是说，在一个线程访问的时候，其他线程必须等待。此时，线程之间不能交替使用该资源，否则就会导致执行结果的不可预期和不一致性。一般地，线程中访问临界资源的代码部分被称为临界区。

简单地说，访问临界区的代码不能同时执行。在线程进入临界区之前，首先要检查是否已经有线程在访问临界资源。如果临界资源空闲，才可以进入临界区执行，并设置访问标志，使得其他线程不能再进入临界区。如果临界资源忙，该线程需要等待，直到临界资源被释放。

在前面的多线程示例代码中，由于不存在共享变量这种软件临界资源，所以不存在数据同步的问题。

【例 21-12】演示引入一个全局的计数器的方法。

```
#filename: threading_2.py
import threading
import time

class Counter: #计数器类
    def __init__(self):
        self.value = 0
    def increment(self):
        self.value = self.value + 1 #将 value 值加 1
        value = self.value #并返回这个 value 值
        return value

counter = Counter()

class ThreadDemo(threading.Thread):
#省略了__init__构造函数
    def run(self):
        time.sleep(1)
value = counter.increment()
        print (time.time()-self.create_time),"\t",self.index, "\tvalue: ", value

for index in range(100): #将生成 100 个线程
    thread = ThreadDemo (index, time.time())
    thread.start() #启动线程
```

- 第 5 行 ~ 第 11 行代码构建了一个计数类。其中，在类的构造函数中将计数器的值初始化为“0”。类的实例可以调用 increment 方法来增加内部计数，同时返回这个值。
- run 方法中调用了 Counter 类中的 increment 方法，将计数器的值加“1”。

由于采用了全局变量，所以每个线程都会将计数器加 1。对于产生的 100 个线程而言，计数器最终将到达 100，但是实际上并非如此。下面是【例 21-12】代码的一种输出结果：

```
0       value: 1
2       value: 2
3       value: 4
1       value: 3
5       value: 5
#省略了部分输出结果
97      value: 90
98      value: 91
99      value: 92
92      value: 84
```

将【例 21-12】代码运行多次，虽然结果不一定如上所示，但是有一点是共同的—— Count 类的 value 值基本上没有达到 100。这就是因为没有对临界区进行互斥访问造成的。为了能够对临界

更多资源请访问稀酷客(www.ckook.com)

区加以区别对待，需要在原来的代码中加入临界区的进入部分和离开部分，如图 21-5 所示。

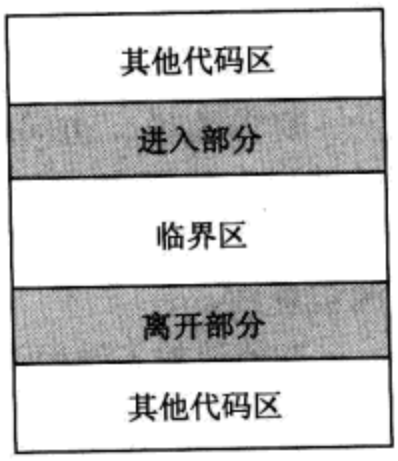


图 21-5　加入临界区的进入部分和离开部分

对于临界区的访问必须加以限制，重点就是要如何实现这些进入部分和离开部分。针对这个问题已经有多种实现方法。在实际操作中，对于临界区的访问需要遵循如下一些访问原则。

（1）空闲让进：当临界资源处于空闲状态的时候，应该允许申请进入临界区的线程。

（2）忙则等待：当临界资源正在被某个线程访问的时候，如果有其他线程使用临界资源，则必须等待。这可以保证对于临界区的互斥访问，每次最多只会有一个线程使用临界资源。

（3）有限等待：对于要求进入临界区的线程，应该保证在有限的时间内能够使用临界资源，而不是无终止的等待。

（4）让权等待：当线程不能够进入临界区使用临界资源的时候，表示此时有线程在使用临界资源。此时该线程应该将自己阻塞，并释放 CPU 资源。

21.8.2　锁机制

一种数据之间同步的简单方法就是使用锁机制。在低级 thread 模块和高级 threading 模块中都提供了锁机制。当然，threading 模块中的锁机制也是在 thread 模块的基础上实现的。在 Python 中，这是最低层次的数据同步原语。一个锁有总是处于“已锁”和“未锁”这两种状态之中。为此提供了两种操作：“加锁”和“解锁”，分别用来改变锁的状态。参照图 21-5，对于一个锁来说，如果是未锁的状态，线程在进入部分将使用加锁操作将锁的状态变为已锁。当临界区代码执行完毕，可以使用解锁操作将锁状态改为未锁。如果需要使用临界资源的线程发现锁的状态为已锁的时候，则必须阻塞等待，直到锁状态改变为未锁。

在【例 21-12】代码中，执行结果没有像预期的那样（即 Count 的值变为 100），也是因为对临界区的不正确访问造成的。对现有的代码加入锁机制，首先需要找到临界区。经过分析，可以看到线程对 Count 中 increment 方法的访问不同步。具体说来，临界区包括 increment 方法中的如下代码：

```
        self.value = self.value + 1
        value = self.value
```

为了解决这个问题，需要对这段代码使用锁机制。加入了 threading 模块中锁机制的代码如下。

低层次 thread 锁机制版本：

```
class Counter: #计数器类
    def __init__(self):
        self.value = 0
        self.lock = thread.allocate_lock()
def increment(self):
        self.lock.acquire() #获取锁,进入临界区
        self.value = self.value + 1
        value = self.value
        self.lock.release() #释放锁,离开临界区
        return value
高层次 threading 锁机制版本:
class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()
def increment(self):
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.lock.acquire() #获取锁,进入临界区
        self.value = self.value + 1
        value = self.value
        self.lock.release() #释放锁,离开临界区
        return value
```

- 由于 threading 中的 Lock 类是简单构建在 thread 的锁机制之上的，所以在这里的两者看起来非常相似。事实上，thread 模块中的锁机制也是此模块中唯一被推荐为可以直接使用的对象。
- 锁机制的实现机制也非常简单，就是加锁和解锁。在代码中可以看到，适当的将临界区使用锁包裹起来就实现了线程之间的数据同步。
- 在__init __构造函数中，使用 thread. allocate_lock( )或者 threading. Lock （） 生成了一个互斥锁。当锁生成的时候，其初始化状态是未锁的。
- 两个模块都提供了两个基本的方法，分别对应加锁和解锁。当状态为未锁的时候，acquire方法将改为锁定并立即返回。如果状态为已锁，将阻塞直到有另外一个线程调用 release 将锁释放成未锁状态。release 方法只能在已锁的状态下调用。它将锁的状态改变为未锁状态就立即返回。
- 当有多个线程阻塞在 acquire 方法的时候，只有一个线程可以改变锁的状态。而选择等待的哪个线程则和具体的实现有关。
- 这里锁机制的方法都是原子操作的。所谓原子操作，就是不能被更高等级中断抢夺优先的操作。也就是说，这段代码在执行的时候不可能被优先级更高的线程中断。这保证了临界区的进入部分和离开部分不会因为其他线程的中断而产生问题。

运行这段代码可以看到，虽然输出的线程 index 值依然是无序的，但是后面的 value 值确实是依次递增的，直到 100。如下面的输出结果所示：

```
0       value: 1
2       value: 2
1       value: 3
#省略了部分输出结果
44      value: 97
60      value: 98
78      value: 99
95      value: 100
```

从上面的输出结果可以看出，简单的锁机制已经可以解决比较简单的线程数据同步问题了。

21.8.3　条件变量

虽然锁机制可以解决一些数据同步问题，但这只是最低层次的同步。当线程变多，且其关系复杂的时候，就需要更加高级的同步机制了。这就是这里要介绍的“条件变量”，使用这种机制可以使得只有在特定的条件下才对临界区进行访问。条件变量通过允许线程阻塞和等待线程发送信号的方式弥补了锁机制中的锁状态不足的问题。

在条件变量同步机制中，线程可以使用条件变量来读一个对象的状态进行监视，或者用其发出事件通知。当某个线程的条件变量被改变的时候，相应的条件变量将会唤醒一个或者多个被此条件变量阻塞的线程。然后这些线程将重新测试条件是否满足，从而完成线程之间数据的同步。

Python 中的 Condition 类提供了这种机制的实现。由于条件变量同样可以用于锁机制，所以其中也提供了 acquire 方法和 release 方法。除此之外，此同步机制还提供了 wait，notify和 notify-All 等 3 个常用方法。调用 wait 方法将使线程处于阻塞状态。它有一个可选的参数，用来指定超

更多资源请访问稀酷客(www.ckook.com)

时时间。如果不指定超时时间，线程将一直阻塞，直到被 notify 或者notifyAll唤醒。notify 方法用来唤醒等待此条件变量满足的线程。notifyAll 方法和notify类似，但是它是唤醒所有等待此条件的线程。

生产者-消费者问题（Producer-consumer problem）是一个非常著名和经典的同步问题。在这个问题中有两个线程：生产者线程和消费者线程。生产者线程生产产品，同时，消费者线程消耗产品。当消费者消费产品的时候，如果没有产品，消费者线程将阻塞，直到生产者线程生产出产品。为了简单起见，这里的生产者的缓冲区是无限的。

【例 21-13】演示生产者-消费者问题的一个实现。

```
#filename: producer_consumer_1.py
from threading import Thread, Condition, currentThread
import time

class Goods: #产品类
    def __init__(self): #初始化函数
        self.count = 0
    def produce(self,num = 1): #产品增长
        self.count + = num
    def consume(self): #产品减少
        if self.count:
            self.count - = 1
    def isEmpty(self): #判断产品是否为空
        return not self.count

class Producer(Thread): #生产者类
    def __init__(self,condition,goods,sleeptime = 1):
        Thread.__init__(self)
        self.cond = condition
        self.goods = goods
        self.sleeptime = sleeptime

    def run(self):
        cond = self.cond
        goods = self.goods
        while 1:
            cond.acquire()
            goods.produce()
            print "Goods Count: ", goods.count, "Producer thread produced "
            cond.notifyAll() #通知满足此条件变量的线程
            cond.release()
            time.sleep(self.sleeptime)

class Consumer(Thread): #消费者类
    def __init__(self,index, condition,goods,sleeptime = 4):
        Thread.__init__(self, name = str(index))
        self.cond = condition
        self.goods = goods
        self.sleeptime = sleeptime
    def run(self):
        cond = self.cond
        goods = self.goods
        while 1:
            time.sleep(self.sleeptime)
            cond.acquire()
            while goods.isEmpty():
                cond.wait() #如果为空,则等待
            goods.consume()
            print "Goods Count: ", goods.count, "Consumer thread",currentThread().
getName(),"consumed "
```

更多资源请访问稀酷客(www.ckook.com)

```
            cond.release()

goods = Goods()
cond = Condition()

producer = Producer(cond,goods)
producer.start() #启动生产者线程
producer.join()  #等待生产者线程完成
for i in range(5):
    consumer = Consumer(i,cond,goods)
    consumer.start() #启动 5 个消费者线程
    consumer.join()
```

- Goods 类只是一个简单的产品类，提供了产品数量的增加和减少。注意，这里只在产品减少的 consume 方法中进行了判断，而在产品增加 produce 方法中没有作此判断，所以在后面设计生产者和消费者线程的时候，消费者线程消费的速度应该不小于生产者线程生产产品的速度。
- 在消费者 Consumer 类中，当没有产品的时候，调用了条件变量的 wait 方法（cond. wait ()），此线程将进入阻塞状态，直到被 notify 或者 notify 唤醒。
- 在生产者 Producer 类中，生产者生成一个产品后，就使用 notifyAll 方法通知等待的线程（cond. notifyAll( )）。
- 注意到在消费者类和生产者类中，在调用 wait 方法或者 notify 方法的时候，都是在获取了互斥锁的前提下进行的。
- 在最后面的几行代码中，生成了 5 个消费者线程和 1 个生产者线程，同时，生产者生产产品的速度是消费者消费的 4 倍，所以不会出现产品无限增多的情况。

下面是【例 21-13】代码的一种可能的输出结果：

```
Goods Count:  1 Producer thread produced One Item
Goods Count:  2 Producer thread produced One Item
Goods Count:  3 Producer thread produced One Item
Goods Count:  2 Consumer thread 1 consumed One Item
Goods Count:  1 Consumer thread 0 consumed One Item
Goods Count:  0 Consumer thread 2 consumed One Item
Goods Count:  1 Producer thread produced One Item
Goods Count:  0 Consumer thread 3 consumed One Item
Goods Count:  1 Producer thread produced One Item
Goods Count:  0 Consumer thread 4 consumed One Item
Goods Count:  1 Producer thread produced One Item
Goods Count:  0 Consumer thread 1 consumed One Item
Goods Count:  1 Producer thread produced One Item
Goods Count:  0 Consumer thread 0 consumed One Item
Goods Count:  1 Producer thread produced One Item
Goods Count:  0 Consumer thread 1 consumed One Item
#省略部分输出结果
```

- 如果人工不加以干预，本代码将会一直执行下去。从输出结果来看，除了最初有部分剩余产品外，后面只要生成产品就被消费了。这也是可以解释的，因为消费者的消费速度要快于生产者生产产品的速度。

特别需要注意的是，要对锁或者条件变量仔细检查，防止死锁产生。对于没有使用超时参数的 wait 方法，一定要有一个对应的 notify 方法调用，或者采用 notifyAll 方法。

21.8.4　信号量

信号量是一种比较古老和有效的数据同步机制。这是荷兰的计算机科学家 Dijkstra 发

更多资源请访问稀酷客(www.ckook.com)

明的。信号量主要用在需要对有限的资源进行同步的时候。信号量内部维护了对于资源的一个计数器，用来表示还可用的资源数。这个计数器是不会小于 0 的。尽管在学术界经常称用在信号量上的操作为 P、V 操作，但是实际上还是与 acquire 和 release 是一样的。

在 Python 中，Semaphore 类提供了这种同步机制的实现。类中提供了 acquire 和release这两个方法。调用 acquire 方法的时候，如果内部计数器大于 0，则将其减 1 并返回。如果内部计数器等于 0，则阻塞此线程，直到有线程使用 release 方法将内部计数器更新到大于 1。而 release 方法则比较简单，将内部计数器加 1 即可。同时，如果有线程在等待资源，则将其唤醒。

举个简单的例子，假设资源的数目为 5，则可以使用如下语句来构造信号量：

```
max_resource = 5
res_sema = Semaphore( value = max_resource)
```

当有多个线程需要使用此资源的时候，则在每次使用资源的前后加上对信号量的操作。下面是代码片段：

```
res_sema.acquire()
#使用此资源
res_sema. release()
```

在这里，当信号量的内部计数器的值大于 0 的时候，此段代码是一直可以执行的，但是当计数器的值为 0 的时候，就需要等待其他线程释放资源才能继续执行了。

21.8.5　同步队列

线程 threading 模块提供了上面这些同步机制，但是在实践中，最容易处理的是同步队列 Queue。这是一个专门为多线程访问所设计的数据结构，能够有效地实现线程对资源的访问。程序可以通过此结构在线程间安全有效的传递数据。

Queue 模块中有一个 Queue 的类。其构造函数中可以指定一个 maxsize 值。当 maxszie 值小于或等于 0 的时候，表示对队列的长度没有限制。当大于 0 的时候，则指定了队列的长度。当队列到达最大长度而又有新的线程要加入队列的时候，则需要等待。Queue 类中有不少方法，但是最重要的是 put 和 get 方法。put 方法将需要完成的任务放入队列。而 get 则是从队列中获取任务。需要注意的是，在这些方法中，有些方法由于多线程的原因，返回值并不一定是准确的，如 qsize、empty 等。

【例 21-14】演示使用简单的同步队列。

```
#filename: queue_1.py
import threading, Queue
import time, random

class Worker(threading. Thread): #工作类
    def __init__(self, index, queue): #构造函数
        threading.Thread. __init__(self)
        self.index = index
        self.queue = queue
    def run(self):
        while 1:
            time.sleep(random. random())
            item = self. queue. get() #从同步队列中获取对象
            if item is None: #循环终止条件
                break
            print "index:",self.index, "task", item, "finished"
            self.queue.task_done()
```

更多资源请访问稀酷客(www.ckook.com)

```
queue = Queue.Queue(0) #生成一个不限制长度的同步队列
for i in range(2):
    Worker(i, queue).start() #生成两个线程
for i in range(10):
    queue.put(i) #向同步队列中加入对象
for i in range(2):
    queue.put(None)
```

- 在最后面的代码中，生成了一个不限制队列长度的同步队列。同时生成了两个线程。
- 向队列中放入 10 个整数，线程的作用就是将这些值打印出来（表示任务完成）。注意，在这里使用了 None 来作为队列的结束标志。

下面是【例 21-14】的一种运行结果：

```
index: 1 task 0 finished
index: 1 task 1 finished
index: 0 task 2 finished
index: 1 task 3 finished
index: 0 task 4 finished
index: 0 task 5 finished
index: 1 task 6 finished
index: 1 task 7 finished
index: 0 task 8 finished
index: 0 task 9 finished
```

从输出结果来看，同步队列很好地完成了工作。

21.8.6　线程同步小结

上面介绍了 4 种线程的数据同步机制，包括锁机制、条件变量、信号量和同步队列。这 4 种同步机制各有优劣，读者可以根据需要选用。需要说明的是，采用数据同步机制将会大大降低线程的性能，因为这些管理机制都有着不小的开销，所以正确地识别是否有临界区并准确的找到，对于系统性能的影响是很大的。表 21-5 中列出了 4 种线程数据同步机制中支持的方法。

表 21-5　线程数据同步的常用方法

同步机制	方　法	说　明
锁机制	使用互斥锁来实现数据同步	
	acquire	获取锁
	release	释放锁
条件变量	同样使用互斥锁，可以实现带有条件的临界区访问	
	acquire	获取锁
	release	释放锁
	wait	阻塞线程，直到有事件来唤醒
	notify	唤醒等待此条件变量的线程
	notifyAll	同 notify，但是将会唤醒所有等待此条件的线程
信号量	可以用来保护多个资源的访问	
	acquire	获取资源，将内部计数器减 1；如果内部计数器为 0，则等待
	release	释放资源，将内部计数器加 1
同步队列	使用同步队列的入队和出队来有效地实现数据同步	

更多资源请访问稀酷客(www.ckook.com)

（续）

同步机制	方　法	说　明
	qsize	队列的长度（线程的原因，不一定准确）
	empty	队列是否为空（线程的原因，不一定准确）
	full	队列是否已满（线程的原因，不一定准确）
	put	将任务入队
	put_nowait	将任务入队，如果队列已满则抛出 Full 异常
	get	将任务出队
	get_nowait	将任务出队，如果队列为空则抛出 Empty 异常
	task_done	指示上一个入队任务是否完成操作
	join	等待队列中任务完成操作

多线程的数据同步管理非常复杂，特别是在进程较多和关系交错的情况下，所以在管理比较多的线程的时候，可以采用一些已有的一些框架，如 Twisted 等。在实现多线程的时候，最好能够衡量一下由此带来的性能提升和复杂性所带来的维护开销。

21.9　小结

本章讲解了 Python 中进程和线程的相关知识。首先介绍了进程和线程的概念，并介绍了 Python 语言中针对进程和线程处理的支持。随后讲述了 Python 下的进程编程，主要介绍了进程运行环境、进程的创建和终止、subprocess 模块以及信号机制等知识。最后讲解了多线程，包括多线程的状态和状态转移、生成和终止线程的方法、线程之间的同步等内容。

更多资源请访问稀酷客(www.ckook.com)

第 22 章　基于 Python 的系统管理

由于 Python 语言的“胶水性”，系统管理是其大显身手的地方。越来越多的系统管理员已经将 Python 语言作为首选语言。这种现象并不仅仅限于类 Linux 系统，即使在 Windows 下，也可以使用 Python 进行系统管理。本章的知识点：

- IPython 的使用方法
- IPython 和相关的 Shell 命令
- 对文件和目录进行比较
- 文件的归档和压缩
- 使用 Python 定期执行任务

22.1　增强的交互式环境 IPython

Python 的一个强大之处就是自带有交互式解析环境，可以有效的测试语言特性、书写代码和测试代码。在交互式解释器中执行代码可以马上得到结果，这可以让人迅速的根据结果对代码进行修改。IPython 由 Fernando 设计，是一个用于构建针对解决特定问题的交互式环境，同时具有高可配置性。功能当然也比原始的 Python 解释器要强大得多。

22.1.1　IPython 介绍

许多语言要测试语言的功能必须要生成程序文件。虽然 Python 语言的解释器可以在不生成文件的前提下对语言的功能进行测试，但是 Python 语言中自带的解释器不能满足更加高级的应用，特别是不能满足系统管理员的需要。因为系统管理员经常做的事情，就是小型代码的书写并运行，而 IPython 正好可以满足这种需要。

IPython 是一种增强的交互式 Python 解释器，设计精巧并同时具有良好的扩展性。IPython设计的目的是为了能够创造一个交互式的完整计算环境。为了能够支持这一点，IPython有两个很具特色的组件，一个是非常强大的 IPython 解释器，另外一个是交互式的并行计算环境。在一般系统管理员的使用中，仅仅会用到其强大的解释器部分。

IPython 解释器的强大功能使其不仅仅可以作为 Python 的解释器，甚至可以直接作为系统管理员的工作环境。它具有如下一些功能。

（1）magic 函数：内置了很多函数用来实现各种功能。
（2）Tab 补全：可以有效的补齐 Python 语言的模块、方法和类等的名称。
（3）源码编辑：可以直接修改源码并运行。
（4）宏：可以将一段代码定义为一个宏，便于以后运行。
（5）历史记录：提供了强大的历史记录功能。
（6）对象自省：有着强大的对象自省功能。
（7）执行系统命令：可以直接在交互式 Shell 中执行系统命令。

更多资源请访问稀酷客(www.ckook.com)

上面的这些功能，将会在下面进行详细的介绍和示例。另外，IPython 还有很多其他有用的功能。具体的可以参看 IPython 的文档，上面有详细的介绍。

22.1.2　IPython 的安装

IPython 的主页是 http：//ipython. scipy. org/。上面有关于 IPython 的官方资源，包括文档、下载和常见问题的解答等。为了能够安装 IPython，首先需要下载相应的软件包。可以从 http：//ipython. scipy. org/dist/上根据自己所用的操作系统下载对应的软件包。本书使用的是 IPython0. 9. 1 版本（可能这已经不是 IPython 的最新版本，读者可以根据需要去下载最新版本）。

对于 Ipython 的安装，有使用源码安装、使用安装包安装或者直接运行等 3 种方式。分别说明如下。

1. 使用源码安装 IPython

最重要、最广泛也最基础的方式当然是通过源码来安装。首先从 IPython 的网站上下载相应的源码包。即从 http：//ipython. scipy. org/dist/下载源码包ipython-0. 9. 1. tar. gz。将此源码包解压到一个目录中，可以看到目录中有一个 setup. up 文件。使用 python setup. py install来安装。这步操作将会把 IPython 的库安装在 site-packages 目录下，同时在 scripts 目录下生成 ipython 的可执行脚本。

在类 Linux 等系统下，这还会同时在 Python 的可执行目录下生成可执行脚本。在Windows系统下，为了能够直接使用 IPython，需要将 scripts 的目录加入到环境变量中。

在 Linux 下的安装方法如下：

```
root@ Linux:/root $ tar zxf ipython-0. 9. 1. tar. gz
root@ Linux:/root $ cd ipython-0. 9. 1/
root@ Linux:/root/ipython-0. 9. 1 $ python setup. py install
```

然后直接使用 ipython 命令就可以启动这个增强交互式 Python 解释器了。

在 Windows 下，可能没有 tar 命令，这就需要找到合适的解压软件将软件包解压，然后再进行安装。安装完成之后，如果打开 IPython 的时候提示没有此命令，还需要修改环境变量，在 path 环境变量中加入 scripts 目录。下面是具体的步骤。

（1）打开【控制面板】，双击【系统】图标，选择【高级】选项卡中的【环境变量】选项，对环境变量进行修改。

（2）如图 22-1 所示，可以看到对话框分为了两个部分。其中，上面部分设置的环境变量仅仅对当前用户起作用，下面部分设置的环境变量可以对整个系统起作用。如果只是需要在本用户下使用IPython，修改上面部分的环境变量即可。如果要让每个登录用户都可以使用 IPython，必须要修改下面部分的环境变量。

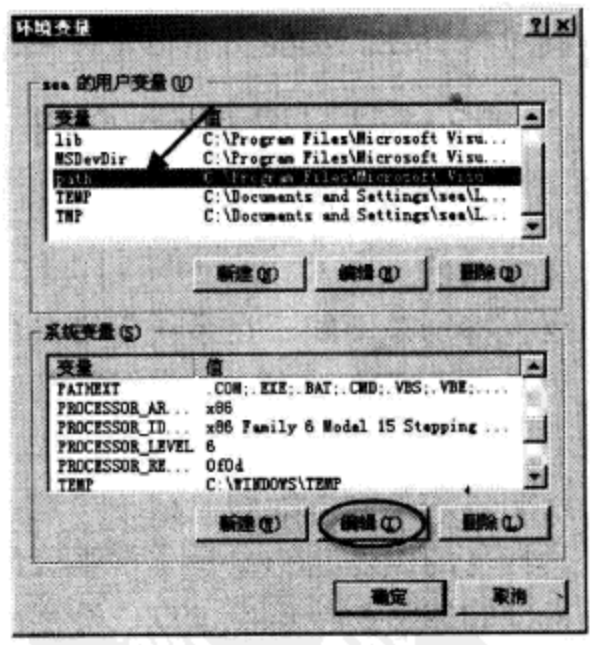


图 22-1　修改 path 环境变量

（3）找到 path 环境变量后，在如图 22-1 所示对话框中，单击下方的【编辑】按钮，在打开的【编辑用户变量】对话框中添加 scripts 的所在目录，并使用分号与其他目录分开，如图 22-2所示。

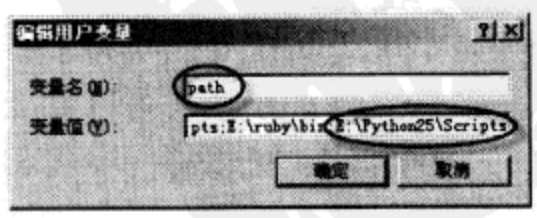


图 22-2　将 scripts 目录加入 path 变量

更多资源请访问稀酷客(www.ckook.com)

2. 使用安装包安装 IPython

如果觉得安装源码包比较繁琐，可以采用制作者预先已经打包好的 IPython 包。在Windows下，可以直接下载安装程序。在 Linux 下，可以使用包管理器来进行安装。如在 Debian 和 Ubuntu 系统中，可以直接输入命令 apt-get install ipython 来安装。系统会自动下载软件包并配置好。其他的 Linux 发行版可以参看其软件安装文档。

在 Windows 下，可以直接下载已经配置好的可执行程序。双击安装程序，将会自动找到系统已经安装的 Python 目录并进行安装，如图 22-3 所示。

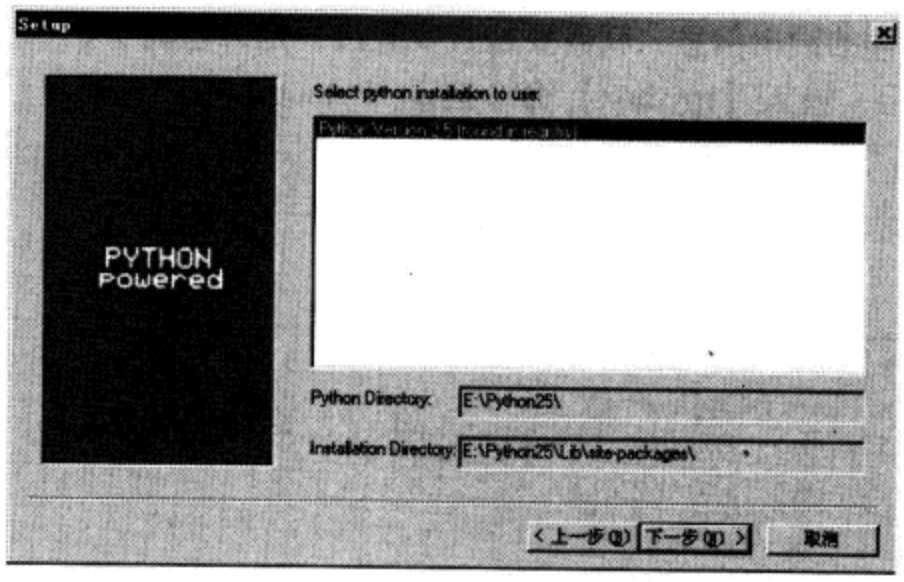


图 22-3　IPython 的 Windows 安装界面

说明　这里推荐同时安装 PyReadline，以便为 IPython 提供一些比较好的功能，如带格式拷贝等。具体的下载网址和 IPython 的网址是一样的。其当前最新版本为 1.5。

3. 直接运行 IPython

第三种安装方法是“不安装”。其实，这个意思是说只是在需要的时候直接使用，并不将 IPython 安装到系统中。当将 IPython 的源码包解压以后，可以在目录下看到 ipython. py。直接运行此代码就可以得到 IPython 的一个运行实例。这适合于某些比较特殊的场合。例如，只是需要暂时使用一下，而并不需要安装或者没有安装的权限。

当然，这些方法中，直接使用已经制作好的安装包来安装是最方便的，但是有些时候可能需要对源码进行一些修改后再安装，这就必须要使用第一种源码的安装方式了。

22.1.3　IPython 的启动

正确安装 IPython 后，就可以直接输入【ipython】命令来启动 IPython 解释器了。在第一次使用的时候，IPython 将会配置环境。下面是一种可能的输出结果：

```
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:

C: \Documents and Settings \ <name > \_ipython

Successful installation!
Please read the sections 'Initial Configuration'and 'Quick Tips'in the
IPython manual (there are both HTML and PDF versions supplied with the
distribution) to make sure that your system environment is properly configured to take ad-
vantage of IPython's features.
```

更多资源请访问稀酷客(www.ckook.com)

```
Important note: the configuration system has changed! The old system is
still in place, but its setting may be partly overridden by the settings in
" ~/_ipython/ipy_user_conf.py" config file. Please take a look at the file
if some of the new settings bother you.

Please press <RETURN> to start IPython.
```

上面的信息主要是告诉用户，IPython 已经安装且配置成功了，并告诉了用户配置文件所在的目录。同时，这里还提示用户可以先看看其帮助文件。特别要注意的是，这里有“Successful installation”的字样，如果没有看到这样的输出信息，就需要检查安装过程是否正确了。

IPython 配置完成后，输入【ipython】命令启动 IPython，可以看到下面的信息：

```
Python 2.5.2 (r252: 60911, Feb 21 2008, 13: 11: 45) [MSC v.1310 32 bit (Intel)]
Type " copyright", " credits" or " license" for more information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ? object also works, ?? prints more.

In [1]:
```

在输出的信息中，可以看到 IPython 对几种帮助方式都做了详细的介绍。浏览一下就可以使得初次使用的人也可以很快的上手。最后注意到 IPython 输出了“In [1]:”，这可能是和标准的 Python 解释器提示符的最大不同了。在标准的 Python 解释器中，采用的是“>>>”的方式，而这里采用了数字提示符的方式。在以后可以看到，这种数字的提示符将会起到很大的作用。

22.1.4 IPython 的环境配置

为了构建一个合适的 IPython 环境，可以对具体的环境进行最优化调整。在系统第一次运行的时候，将会在信息中显示 IPython 配置文件所保存的路径。一般情况下，在 Linux 系统下，配置文件在主目录下的 .ipython 目录中。在 Windows 系统下，则为 C：\Documents and Settings\ <name> 下的_ipython。具体的配置文件为 ipy_user_conf.py。

将 ipy_user_conf.py 文件打开可以看到，此文件仅仅是一个使用 Python 语法的配置文件。在默认的配置文件中，有许多的配置选项是没有打开的，用户可以根据自己的环境需要来打开这些选项。另外，配置文件中还提供了几个实用函数用来帮助配置，如 import_all 函数。在配置文件中使用 import_all（“os sys”）将使得在 IPython 的命令行中不必再导入 os 和 sys 模块。

除此之外，在配置目录中还有其他的一些已经预置好的配置文件，包括 ipythonrc-math、ipythonrc-numeric、ipythonrc-physics 等。如果需要使用这些特定的配置文件启动IPython，可以使用【ipython-p】命令来启动。

22.1.5 IPython 的退出

退出 IPython 的方式有多种，同样可以采用 Python 标准 Shell 中的退出方式，不过在这里还需要进行确认。另外，也可以使用【Exit】或者【Quit】命令不经过确认而退出。

22.2 和 IPython 的简单交互

当提示符出现的时候，就可以在提示符下输入任何想测试的代码了。IPython 除了支持

更多资源请访问稀酷客(www.ckook.com)

Python语言的众多特性外，还支持直接在提示符下执行系统命令。同时，IPython 还提供了许多的 magic 函数，用来处理特殊的对象，或者完成其他工作。

22.2.1　IPython 中的输入与输出

相对于标准的 Python 交互式环境而言，IPython 对输入和输出的功能进行了增强。

【例 22-1】演示在 IPython 下输入和输出的如下 3 种情况。

```
In [1]: for i in range(5):
   ...:     print i
   ...:
0
1
2
3
4
In [2]:
```

【例 22-1】是直接测试 Python 相关代码。简单地看，可能和标准的 Python 解释器并没有什么区别，但是实际上，两者除了在提示符方面有区别外，在字符串显示方面也是不一样的：

```
In [2]: ! cd E:\Python25\

In [3]: ! dir
驱动器 E 中的卷是 PROG
 卷的序列号是 C413-1C2C

 E:\Python25 的目录

2008-12-24  02:16    <DIR>          .
2008-12-24  02:16    <DIR>          ..
2008-09-22  20:02    <DIR>          DLLs
2008-09-22  20:02    <DIR>          Doc
2008-09-22  20:02    <DIR>          include
#省略部分输出结果
In [4]:
```

In［2］是在 IPython 解释器中直接执行外部系统命令【！ cd E：\ Python25 \ 】的情况，注意到前面有“!”前缀。虽然执行该命令没有输出结果，但是实际上系统的目录已经到了“E：\ Python25 \ ”。这可以从 In［3］的系统命令【！ dir】的输出结果中看出。

另外，可以执行 IPython 中的 magic 函数：

```
In [4]: % cd E:\Python25\
E:\Python25
In [5]:
```

带有“%”前缀的命令表示是 IPython 中的 magic 函数，这些函数具有特殊的功能。虽然这里的命令也是【cd】，但和前面的实现是不一样的。从输出结果中也可以看出两者的不同之处。前者仅仅是调用外部系统命令，本身没有输出信息，而后面的 magic 函数则给出了当前所在的目录。

22.2.2　输出提示符的区别

在 Python 语言特性的测试上，从上面与 IPython 的交互情况来看，和标准的 Python 解释器好像并无区别，两者都是将字符串输出并显示在屏幕上。其实不然。再来多看几个交互的例子：

更多资源请访问稀酷客(www.ckook.com)

```
In [1]: a = 0

In [2]: b = "IPython"

In [3]: c = [i for i in range(5)]

In [4]:
```

这里在交互式 Shell 中定义了 3 个变量（a，b，c），其中分别为数字、字符串和列表。由于这里仅仅是赋值，所以也没有输出结果。现在将这几个变量的值打印出来：

```
In [4]: print a
0

In [5]: print b
IPython

In [6]: print c
[0, 1, 2, 3, 4]

In [7]:
```

这里的目的是将各个变量的值输出。交互式 Shell 将各个变量的值打印了出来，这好像没有什么特殊的地方。再来看看下面的输出结果：

```
In [7]: a
Out[7]: 0

In [8]: b
Out[8]: 'IPython'

In [9]: c
Out[9]: [0, 1, 2, 3, 4]

In [10]:
```

注意到输出结果中出现了类似“Out［7］:”的输出提示符，和输入提示符是一一对应的。这里的目的也是输出各个变量的值，可是和上面的结果却不一样。如上面的输出差异在标准的 Python 交互式环境中是看不见的，如下面的输出结果：

```
> > > c = [i for i in range(5)]
> > > print c
[0, 1, 2, 3, 4]
> > > c
[0, 1, 2, 3, 4]
> > >
```

22.2.3　输出提示符不同的原因

之所以出现不同的输出提示符，其原因是对于对象的__str __和__repr __调用的区别。两者都是输出变量的值，但是前者是转化为字符串输出，而后者是一种对于对象的表示。虽然在很多情况下两者的显示结果都是一样的，但是有些时候却是不同的。

【例 22-2】演示出现不同的输出提示符。

```
In [1]: class StrReprDemo(object):
   ...:     def __str __(self):
   ...:         return "StrReprDemo: call the __str __ function"
   ...:     def __repr __(self):
```

更多资源请访问稀酷客(www.ckook.com)

```
    ...:         return "StrReprDemo: call the __repr__ function"
    ...:
In [2]: srd = StrReprDemo()

In [3]: print srd
StrReprDemo: call the __str__ function

In [4]: srd
Out[4]: StrReprDemo: call the __repr__ function

In [5]:
```

在【例 22-2】中，定义了一个继承自 object 的类 StrReprDemo 和两个特殊函数（__str__和__repr__）。然后在类定义的基础上生成了一个实例 srd。最后按照先前的做法，首先是使用 print 方法输出此值，然后是直接使用 Python 对象输出此值。可以看到，两者的输出结果是不一样的。在“print srd”语句的输出中，代码输出了在__str__函数中返回的内容。而在直接输出 srd 的值的时候，代码输出了在__repr__函数中返回的内容。也就是说，使用 print 语句来输出函数的值的时候，实际上返回的是系统中的特殊函数__str__中的值。而对于简单的在解释器中输出对象的值，则是调用__repr__这个特殊函数的返回值。从输出结果中可以看出两者的区别。一般说来，调用 str（obj）的时候，Python 会调用__str__函数。在使用 repr（obj）的时候，会调用__repr__函数。如果使用字符串输出，则可以分别使用“%s”和“%r”。

由于现在 Python 中所有的对象最终都是继承自 object 的，所以当对象定义中没有包含这些特殊函数的时候，就会调用 object 的特殊函数：

```
In [5]: class Test():
   ....:     pass
   ....:

In [6]: test = Test()

In [7]: print test
<__main__.Test object at 0x0149BE50>

In [8]: test
Out[8]: <__main__.Test object at 0x0149BE50>
```

在 In［5］中定义了一个最简单的测试类 Test。这里并没有显式的写明，但是实际上该类还是继承自 object 的。随后通过两种方法输出了对象的值，一种是字符串的表示，另外一种是对象的表示。在输出结果中可以看到这个对象的相关属性，包括地址等。

类的这种实现对于那些最基本的类型也是一样的，包括数字、字符串和列表等。这也就解释了为什么在代码中仅仅只是定义了一些基本类型的变量，而使用相关的语句则可以显示出变量的值。当然，对于不同类型的对象，默认的输出结果是不一样的。如对于字符串对象和类对象的输出结果就可以看出明显的不同。

其实在标准的 Python 解释器中，两种不同的输出结果也是可以看到的：

```
#省略 StrReprDemo 类定义
>>> srd = StrReprDemo()
>>> print srd
StrReprDemo: call the __str__ function
>>> srd
StrReprDemo: call the __repr__ function
>>>
```

从上面的输出结果中可以看到，Python 的标准交互式环境也是区别__str__和__repr__的。只是

更多资源请访问稀酷客(www.ckook.com)

当对象的这两种表示都一样的时候，无法区分罢了。而在 IPython 中，之所以能够在显示对象的值的时候显示 Out［n］的标记，主要还是因为 print 语句的原因。在 IPython 中，print 语句会将代码输出打印在终端上，并不会返回任何值，所以当 IPython 遇到返回值不为“None”的时候，就会在其前面加上 Out［n］的提示符，供以后使用。

22.2.4　IPython 中的 In 和 Out

【例 22-3】为了能够更好地理解 IPython 中的提示符，来看看其具体的类型和其中的值。

```
In [1]: a = [i for i in range(5)]

In [2]: a
Out[2]: [0, 1, 2, 3, 4]

In [3]: print In
['\n', u'a = [i for i in range(5)]\n', u'a\n', u'print In\n']

In [4]: print Out
{2: [0, 1, 2, 3, 4]}

In [5]: type(In)
Out[5]: <class 'IPython.iplib.InputList'>

In [6]: type(Out)
Out[6]: <type 'dict'>

In [7]:
```

- 从 Out［5］和 Out［6］中可以看到，In 和 Out 对象的类型分别为 IPython. iplib. InputList 和 dict。前者实际上是一个列表。
- In［3］和 In［4］的输出结果显示了 In 和 Out 的值。从输出信息中可以看出，在 In 中记录了所有的输入，即使是一个简单的回车也会被记录下来。而对于 Out，则会记录所有输出不为“None”的值，其中字典的键为相应 In 中的数字。这也是为什么 In 可以使用列表来记录，而 Out 则需要使用字典来记录的原因。

In 和 Out 的这种设计可以使 IPython 充分利用列表和字典的操作优势，完成复杂的工作，比如分片。在 22.4.5 小节中，将会利用这种特性来实现区段编辑等功能。

22.3　IPython 中的 magic 函数

当输入是以“%”开始的时候，IPython 会将其视为一个 magic 函数。这种 magic 函数为 IPython 提供了许多功能，将要在下面介绍的许多功能就是通过 magic 函数来实现的，包括源码直接编辑、历史记录功能等。

22.3.1　Magic 函数的使用和构造

这是在上面已经出现过的一个例子：

```
In [4]: % cd E:\Python25\
E:\Python25
In [5]:
```

“cd”这个 magic 函数的作用就是将当前目录切换到指定的目录（此目录需要存在）。

对于 magic 函数，一般需要在其前面加上“%”，但是当 IPython 的配置变量 automagic 打开

更多资源请访问稀酷客(www.ckook.com)

的时候，则不必在 magic 命令前加“%”而可以直接使用。在 IPython 进行解析的时候，magic 函数的解析优先级是比较低的。也就是说，如果输入可以解析为 Python 语言中的对象，则会优先解析为 Python 中的对象。下面是在 IPython 文档中示意的例子：

```
In [1]: cd E:\Python25\
E:\Python25

In [2]: cd = None

In [3]: cd ..
------------------------------------------------------------
   File "<ipython console>", line 1
     cd..
       ^
SyntaxError: invalid syntax

In [4]: % cd ..
E:\

In [5]: del cd

In [6]: cd Python25
E:\Python25
```

在上面可以看到，给 cd 赋值之后，在 automagic 的情况下，就覆盖掉了 magic 函数中的值。如果这时直接去调用 cd 函数，就会出错。当然，使用“%”前缀的 magic 函数总是不会出错的。只有在删除了这个 cd 变量后，才可以正常识别 cd 的 magic 函数功能。

当然，也可以定义自己的 magic 函数：

```
In [1]: import IPython.ipapi

In [2]: def doHello(self, arg):
   ...:     ip = self.api
   ...:     ip.ex("print 'Hello, ', %s" % (repr(arg)))
   ...:
   ...:

In [3]: ip = IPython.ipapi.get()

In [4]: ip.expose_magic("hello", doHello)

In [5]: % hello IPython
Hello,  IPython
```

在上面的代码中定义了一个“hello”命令，功能很简单，就是将后面的参数输出。根据这个例子，可以实现更加复杂的 magic 函数。

在 IPython 中常用的 magic 函数见表 22-1。

表 22-1　IPython 中常用的 magic 函数

magic 函数	说　明
alias	定义一个别名
bg	将 bg 后面的函数放在后台执行
bookmark	定义一个书签目录
cd	切换目录
dhist	显示目录历史
edit	编辑一个文件并执行

更多资源请访问稀酷客(www.ckook.com)

（续）

magic 函数	说　明
env	显示环境变量
hisotry	显示历史记录
lsmagic	显示所有的 magic 函数
macro	定义一个宏
magic	显示当前的 magic 系统帮助
pdef	显示函数的定义
pdoc	显示对象的文档字符串
pinfo	显示对象的详细信息
psearch	查找 Python 对象
psource	输出对象的代码
pwd	显示当前目录
pycat	使用语法高亮来显示一个 Python 文件
r	重复执行上次的命令
run	执行文件
save	将代码保存为文件
time	计算一段代码的执行时间
timeit	计算一段代码的运行时间，系统将自动选择循环次数
who	打印所有的 Python 变量
who_ls	返回所有 Python 变量的列表
whos	和 who 类似，但是会输出更详细的信息

如果想查看系统中都有哪些 magic 函数，可以使用“% lsmagic”命令进行查看。如果需要查看关于 magic 函数完整的帮助，直接输入“% magic”，就会显示所有可用的 magic 函数及其使用方法。

下面两小节将介绍 magic 函数中的一些常用命令。另外，还有一些将在其他章节中进行介绍。

22.3.2　目录管理

在 Python 的标准 Shell 中，虽然可以使用 os. chdir（）来改变目录和使用 os. getcwd（）来获取当前目录，但是使用起来不大直观。

在 IPython 中，有几个 magic 函数和目录管理相关，利用它们可以很方便的管理目录。这些命令包括 cd、pwd、bookmark 和 dhist 等。

1. 使用 cd 切换目录

在前面和 IPython 的简单交互中，曾经看到过 cd 的使用方法。这和在外部 Shell 中的 cd 使用方法是一样的。最常见的用法是“cd directory”，这将使得当前目录变为 directory。如果 cd 后面不接任何参数，则表示切换到当前用户的主目录。如果使用连接符（“cd -”），则会切换到上次的目录中。一个简单的示例如下：

```
In [1]: cd E:\Python25
E:\Python25
```

更多资源请访问稀酷客(www.ckook.com)

```
In [2]: cd
C:\Documents and Settings\sea

In [3]: cd -
E:\Python25
```

注意到每次切换目录的时候，都会将切换后的目录输出。cd 命令提供了一个参数-q，使得 IPython 不用打印此信息。

```
In [4]: pwd
Out[4]: 'E:\\Python25'

In [5]: cd -q ..

In [6]: pwd
Out[6]: 'E:\\'

In [7]: cd Python25
E:\Python25
```

上面演示了在 cd 命令中使用-q 参数的效果。

在上面的演示中，还用到了一个 magic 函数 pwd。其作用是打印出当前的目录信息。虽然简单，但还是很有用的。

2. 使用 bookmark 管理书签目录

IPython 支持将目录加入书签。这是通过 bookmark 命令来完成的。书签一旦建立，除非显式删除，否则在 IPython 中总是可用的。换句话说，即使是 IPython 退出了，书签在下次启动的时候也是可用的。有两种方式来加入目录书签。一种是在当前目录上直接使用 bookmark 命令，第二种是使用 bookmark 命令直接定义一个目录书签。

```
In [1]: cd E:\Python25\
E:\Python25

In [2]: bookmark py

In [3]: bookmark script E:\\Python25\\Scripts
```

- 在 In［1］中，使用 cd 命令切换到了一个特定的目录。在 In［2］中，可以直接使用 bookmark 后加上书签名字来定义一个目录书签。
- 在 In［3］中，采用了第二种目录书签的定义方式。后面的参数分别是书签名字和所标记的书签。这里需要注意的是，在 Windows 下目录的分隔符需要使用转义，或者直接采用如“E：/Python25/Scripts”的格式。

定义书签后，可以使用 bookmark 命令的-l 选项来查看现在 IPython 中定义的所有书签。需要再次提到的是，这里的书签是全局性的。下面是重新打开一个 IPython 终端后的结果：

```
In [1]: % bookmark -l
Current bookmarks:
py      -> E:\Python25
script -> E:\Python25\Scripts
```

在 cd 中可以使用这些书签，以便加快对目录的处理。在 cd 命令中可以通过-b 选项来切换到书签目录中去。

更多资源请访问稀酷客(www.ckook.com)

```
In [4]: pwd
Out[4]: 'E:\\Python25'

In [5]: cd -b script
(bookmark:script) -> E:\Python25\Scripts
E:\Python25\Scripts

In [6]: pwd
Out[6]: 'E:\\Python25\\Scripts'
```

- 在 In［4］中，首先使用 pwd 命令查看了当前所在的目录，在 Out［4］中输出当前的目录为“E：\\Python25”。在 In［5］中使用 cd 命令进入到了书签名字为 script 的书签目录。从输出信息中可以看到已经切换到了这个书签目录中。而 In［6］的检查也验证了这一点。

在使用 cd -b 的时候，可以使用自动补全信息来获取相关的书签名字。在-b 选项的后面，直接按 TAB 键，可以看到 IPython 中已有的书签名字（关于 Tab 补全的内容，将在下一小节进行介绍）：

```
In [7]: cd  -b <TAB>
py      script

In [7]: cd -b
```

书签生成以后，可以通过-d 选项来将其删除，或者使用-r 来删除所有已定义的书签。这个过程同样也是对全局起作用的：

```
In [7]: bookmark tool E:\\Python25\\Tools

In [8]: bookmark -l
Current bookmarks:
py      -> E:\Python25
script -> E:\Python25\Scripts
tool    -> E:\Python25\Tools

In [9]: bookmark -d tool

In [10]: bookmark -l
Current bookmarks:
py      -> E:\Python25
script -> E:\Python25\Scripts

In [11]: bookmark -r

In [12]: bookmark -l
Current bookmarks:

In [13]:
```

- 在 In［7］中，使用 bookmark 命令定义了一个新的书签目录“E：\ Python25 \ Tools”。这从 In［8］的输出结果可以看出来。
- 在 In［9］中，使用-d 选项命令将 tool 书签目录删除了。这样，再使用 bookmark －l 的时候，此书签目录就不存在了。
- 在 In［11］中，使用 bookmark 命令的-r 选项将所有的书签目录都删除了。使用此命令的时候要慎重，因为这些操作都是不可恢复的。

3. 使用 dhist 查看目录历史

使用书签目录的前提是需要定义书签，但是有的时候可能仅仅只是需要在已经输入的目录中进行操作。这时，IPython 提供了 dhist 命令，可以输出曾经在 IPython 会话中切换过的目录：

更多资源请访问稀酷客(www.ckook.com)

```
In [14]: dhist
Directory history (kept in _dh)
0: E:\Python25
1: E:\Python25
2: E:\Python25
3: C:\Documents and Settings\sea
4: E:\Python25
5: E:\
6: E:\Python25
7: E:\Python25
8: E:\Python25\Scripts
9: E:\Python25
10: E:\Python25\Scripts
```

- 从 In［14］的输出结果中可以看到，在 IPython 中曾经切换过的目录都在这里。从输出信息的最上面还可以看到，这个值其实是放在_dh 变量中的。

如果输出结果过多，可以在命令后面加入限制：如果命令后面加上一个数字 n，表示是输出最后的 n 个目录；如果命令后面有两个数字（n 和 m），表示输出 n 和 m 之间的目录（其中不包括 m 数字所代表的目录）：

```
In [15]: dhist 4
Directory history (kept in _dh)
7: E:\Python25
8: E:\Python25\Scripts
9: E:\Python25
10: E:\Python25\Scripts

In [16]: dhist 4 8
Directory history (kept in _dh)
4: E:\Python25
5: E:\
6: E:\Python25
7: E:\Python25
```

使用 cd - n 可以直接进入切换目录历史记录中出现过的目录。在 cd - 的后面还可以按 TAB 键来获取现在 IPython 中已有的目录历史：

```
In [17]: cd -<TAB>
-00 [E:\Python25]                          -06 [E:\Python25]
-01 [E:\Python25]                          -07 [E:\Python25]
-02 [E:\Python25]                          -08 [E:\Python25\Scripts]
-03 [C:\Documents and Settings\sea]        -09 [E:\Python25]
-04 [E:\Python25]                          -10 [E:\Python25\Scripts]
-05 [E:\]

In [17]: cd -08
E:\Python25\Scripts
```

- 在第一个 In［17］中，在“-”后按 TAB 键可以输出现在已有的目录历史。当在第二个 In［17］中输入 cd -08 后，IPython 将会切换到目录历史中 08 序号所指的目录。

除了上面的这些目录管理方法外，cd 命令还提供了对目录历史的搜索。cd 命令的选项“—”后加上需要搜索的字符串，IPython 将会在目录历史中按照从最新到最旧的顺序来搜索是否有出现指定字符串的目录。如果有，则切换到第一个找到的目录，否则输出找不到目录的相关信息：

```
In [18]: cd --Python
E:\Python25

In [19]: cd --test
No matching entry in directory history
```

更多资源请访问稀酷客(www.ckook.com)

通过这种方法，可以很容易的对目录进行管理。

22.3.3　对象信息的收集

在 IPython 中，不但可以对系统进行管理，而且还提供了多种方法来对 Python 对象的信息进行查看和收集。这些 magic 函数主要包括许多以“p”开头的命令，其中有 page、pdef、pdoc、pfile、pinfo、psource 和 psearch 等。

1. 查看环境变量信息

在介绍这些命令之前，先来看看如何查看当前的系统环境配置。这个命令是 env 命令。其作用是显示当前的环境配置情况：

```
In [1]: env
Out[1]:
{'ALLUSERSPROFILE': 'C:\\Documents and Settings\\All Users',
 'APPDATA': 'C:\\Documents and Settings\\sea\\Application Data',
 'CCHZPATH': 'E:\\CTeX\\LOCALT~1\\cct\\fonts',
 'CCPKPATH': 'E:\\CTeX\\LOCALT~1\\fonts\\pk\\modeless\\cct\\dpi$d',
 'CLIENTNAME': 'Console',
 'COMMONPROGRAMFILES': 'C:\\Program Files\\Common Files',
 'COMPUTERNAME': '1FCB1F2C310A4D2',
 'COMSPEC': 'C:\\WINDOWS\\system32\\cmd.exe',
#省略部分输出结果
```

从上面的输出结果中可以看到现在系统中的环境配置变量。

2. 使用 page 处理输出信息

当执行外部命令的时候，输出的结果可能过长，这时可以使用 page 来对输出信息进行处理，从而获得一个更好的显示。当不带有参数的时候，将会对上次的输出信息进行处理。这个处理过程有点像外部系统命令 more 的功能。如在上面的输出结果中的长度超过了一页，可以使用 page 命令对其进行处理。

```
In [2]: page
{'ALLUSERSPROFILE': 'C:\\Documents and Settings\\All Users',
 'APPDATA': 'C:\\Documents and Settings\\sea\\Application Data',
 'CCHZPATH': 'E:\\CTeX\\LOCALT~1\\cct\\fonts',
 'CCPKPATH': 'E:\\CTeX\\LOCALT~1\\fonts\\pk\\modeless\\cct\\dpi$d',
 'CLIENTNAME': 'Console',
 'COMMONPROGRAMFILES': 'C:\\Program Files\\Common Files',
#省略部分输出结果
'PROCESSOR_REVISION': '0f0d',
 'PROGRAMFILES': 'C:\\Program Files',
---Return to continue, q to quit---
```

从最后的输出信息中可以看到，当输出一页后，IPython 将会等待用户操作。

3. 使用 pfile 命令

当含有 Python 源文件的时候，可以使用 pfile 来查看其相关的信息。如果在系统中只是有 .pyc 文件，则不会有显示信息（例如当系统中含有 os 模块的源文件的时候）。

【例 22-4】演示使用 pfile 命令的方法。

```
In [1]: import os

In [2]: pfile os
r"""OS routines for Mac, NT, or Posix depending on what system we're on.
```

更多资源请访问稀酷客(www.ckook.com)

```
This exports:
  - all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.
  - os.path is one of the modules posixpath, ntpath, or macpath
  - os.name is 'posix', 'nt', 'os2', 'mac', 'ce' or 'riscos'
  - os.curdir is a string representing the current directory ('.' or ':')
#省略部分输出结果
```

可以看到，输出信息中包含了 os 模块的相关信息。

4. 使用 pdef、pdoc 和 pinfo 查看对象信息

对于一个 Python 对象，可以使用 pdef、pdoc、pinfo 查看其函数定义等相关信息。pinfo 命令能够得到更详细的信息。

```
In [1]: def add(x, y):
   ...:     """This is a simple addition"""
   ...:     return (x+y)
   ...:

In [2]: pdef add
 add(x, y)

In [3]: pdoc add
Class Docstring:
    This is a simple addition
Calling Docstring:
    x.__call__(...) <==> x(...)

In [4]: pinfo add
Type:           function
Base Class:     <type 'function'>
String Form:    <function add at 0x01444DB0>
Namespace:      Interactive
File:           c:\documents and settings\sea\ipython\<ipython console>
Definition:     add(x, y)
Docstring:
    This is a simple addition
```

- 在 In［1］中，定义了一个函数 add，实现了最简单的加法。其中还包含文档字符串。
- 在 In［2］中，使用 pdef 命令来查看 add 函数的原型。
- 在 In［3］中，使用 pdoc 命令来查看 add 函数的文档字符串。
- 在 In［4］中，使用 pinfo 命令可以查看此函数的相关信息。pinfo 命令除了能够查看函数信息外，还可以查看任何 Python 对象的信息。

5. 使用 psearch 查找对象信息

使用 psearch 命令可以搜索当前名字空间中已有的 Python 对象：

```
In [5]: psearch a*
abs
add
all
any
apply
```

这里，使用 psearch 命令搜索以“a”开头的 Python 对象。可以看到，显示了刚刚在前面定义的 add 函数。除此之外，还有如 abs 等内置的函数。如果需要从输出结果中去掉这些内置对象的信息，可以使用 psearch 的-e 选项先将内置函数去掉：

更多资源请访问稀酷客(www.ckook.com)

```
In [5]: psearch -e builtin a*
add
```

另外，psource 可以用来显示一个对象的代码，这在拷贝代码的时候比较有用。还有一个 magic 函数是 pycat，其作用是高亮显示 Python 对象的源码。由于印刷原因，所以这里没有给出示例代码，读者可以自行在 IPython 中进行测试。

22.4　IPython 适合于系统管理的特点

在上一节中，通过与 IPython 简单交互的示例，已经看到 IPython 交互式命令行的一些特性，但是那些仅仅是 IPython 中许多特性中的一小部分。在本节中，将详细介绍 IPython 的其他功能。正是这些功能使得其成为众多 Python 开发人员的首选 Shell。

22.4.1　Tab 补全

在 Linux 和 Windows 的命令行工具中，都有 Tab 补全的功能。该功能可以加速键盘的键入，但是在 Python 的标准命令行中，默认情况下是不存在这种功能的（如果有 rlcompleter 和 readline 库支持的话，也有有限的 Tab 支持）。但在 IPython 中具有 Tab 补全的功能，并且对这个功能进行了加强。

1. Tab 补全的使用方法

对于命令行中输入的命令，IPython 会检查每个命令是否有 magic 命令。如果有此类命令，则表示这些命令是 IPython 中自定义的方法，由 IPython 进行处理。否则将命令交给标准的 Python 解释器去处理。使用 Tab 补全功能可以在 import 之后补齐系统中已有的模块名称：

```
In [1]: import sy<TAB>
symtable sys       symbol

In [1]: import di<TAB>
dis        difflib    distutils dircache
```

- 上面中的 <TAB> 表示是键入 TAB。

同样也可以补齐模块中的方法和成员变量名称：

```
In [1]: import sys

In [2]: sys.plat<TAB>
In [2]: sys.platform
Out[2]: 'win32'

In [3]: sys.p<TAB>
sys.path                    sys.path_importer_cache sys.prefix
sys.path_hooks              sys.platform
```

- 当需要补全的时候，如果只有一个可选项，IPython 将会直接补全，如在上面的 In［2］中，在 sys 模块中只有一个以 plat 开头的成员，所以就直接补全了。

除此之外，最重要的功能就是补齐代码中的变量名称：

```
In [1]: import sys

In [2]: platform = sys.platform

In [3]: print plat<TAB>
In [3]: print platform
win32
```

更多资源请访问稀酷客(www.ckook.com)

通过TAB补全可以辅助输入，同时减少错误的发生。

2. Tab 补全的两种方式

在 IPython 中有两种补全方式：常规补全和菜单补全。两者的不同之处在于，当有多于一个可选项的时候，常规补全是首先尝试最大化匹配展开，如果还有多余一个的选择，则列出所有的匹配结果。菜单补全则不会显示可能的匹配列表，而是在可能的匹配中进行轮换。在IPython中，默认是使用常规补全方式，从上面的示例代码中也可以看出来。

除了能够补全 Python 中的对象以外，还可以对 IPython 所带有的 magic 函数进行补齐：

```
In [4]: % r<TAB>
% r        % rehash  % rehashx % rep     % reset    % run      % runlog
```

可以看到，在% 后面是 IPython 的 magic 函数。输入 r 后，按TAB键，可以显示出 magic 函数中所有以 r 开头的 magic 函数。如果需要显示所有的 magic 函数，则在输入% 后直接按TAB键，即可显示所有的 magic 函数：

```
In [4]: %
% Exit            % ed              % pdoc            % run
% Pprint          % edit            % pfile           % runlog
% Quit            % env             % pinfo           % save
% alias           % exit            % popd            % sc
% autocall        % hist            % profile         % store
% autoindent      % history         % prun            % sx
% automagic       % logoff          % psearch         % system_verbose
% bg              % logon           % psource         % time
% bookmark        % logstart        % pushd           % timeit
% cd              % logstate        % pwd             % unalias
% clear           % logstop         % pycat           % upgrade
% color_info      % lsmagic         % quickref        % who
% colors          % macro           % quit            % who_ls
% cpaste          % magic           % r               % whos
% debug           % p               % rehash          % xmode
% dhist           % page            % rehashx
% dirs            % pdb             % rep
% doctest_mode    % pdef            % reset
```

在上面可以看到，有不少在表 22-1 中出现的 magic 函数。

最后要注意的是，IPython 在补全的时候，总是根据上下文的信息来给出补全的信息。这是 IPython 的一种特性，而不是一个限制。如在上面的介绍中，在 import 语句后总是给出系统中存在的模块，而在“sys.”后则仅仅给出在 sys 模块中存在的成员。可以说，IPython 中的 TAB 补全是一种智能补全方式，这使得补全的结果总是想要的，从而改善命令行中的交互体验。

22.4.2　历史记录功能

IPython 会记录下每次输入的信息，以便更好的利用这些信息，提高输入效率。可以使用 magic 函数中的 history 命令来获取已有的历史记录。

1. hisotry 命令的使用方法

【例 22-5】演示使用 history 命令来获取已有的历史记录的方法。

```
In [1]: a = 0

In [2]: b = "IPython"

In [3]: c = [i for i in range(5)]
```

更多资源请访问稀酷客(www.ckook.com)

```
In [4]: cd C:\
C:\

In [5]: history
1: a = 0
2: b = "IPython"
3: c = [i for i in range(5)]
4: _ip.magic(r"cd C:\_"[:-1])
5: _ip.magic("history ")
```

从 history 的输出信息中可以看到，所有的输入信息都被完整的记录了下来。需要注意的是，后面的两项并没有按照输入的样子输出，这是因为 cd 和 history 都是 magic 函数。从这里也可以看出 IPython 对于 magic 函数的处理方式，实际上是调用 IPython. ipapi 中的magic方法来实现的。如果希望显示最初输入的原始格式，则可以使用“history-r”命令：

```
In [6]: history -r
1: a =0
2: b = "IPython"
3: c = [i for i in range(5)]
4: cd C:\
5: history
6: history -r
```

在以上输出结果中可以看到，所有的输入都是按照原样输出的，没有做任何修改。注意到，现在的输出中每个命令输入的前面都有一个数字指示。当需要将代码复制到其他编辑器的时候，如果需要去掉前面的数字，则可以使用“history-n”命令：

```
In [7]: history -n
a =0
b = "IPython"
c = [i for i in range(5)]
_ip.magic(r"cd C:\_"[:-1])
_ip.magic("history ")
_ip.magic("history -r")
_ip.magic("history -n")
```

可以看到，这里的输出结果中没有了前面的数字指示。同样地，由于没有-r 选项，IPython对结果进行了转换。如果需要得到一个原始的不带数字指示的输出，可以同时使用-r 和-n 选项。

2. 在历史记录中查找命令

需要注意的是，history 命令提供了-g 选项，用来输出有指定模式的历史记录：

```
In [8]: history -g history
046: history
0134: test.magic("history")
0135: history -r
0136: history -n
0140: history -g history
===
shadow history ends, fetch by %rep <number> (must start with 0)
=== start of normal history ===
5: _ip.magic("history ")
6: _ip.magic("history -r")
7: _ip.magic("history -n")
8: _ip.magic("history -g history")
```

在前面输出的是“影子历史记录（shadow history）”。即：只要是曾经在 IPython 交互式命令行中出现过的含有搜索模式的命令，都有可能在这里出现。所以这个结果可能是很长的。在后

更多资源请访问稀酷客(www.ckook.com)

面，则输出了当前会话下的包含搜索模式的历史记录。

上面是一种在历史记录中搜索的方式。除此之外，如果有 readline 支持的话，还可以使用下面的两种搜索方式。

（1）直接输入，然后按 Ctrl + p 或者 Ctrl + n 键在历史记录中寻找满足已经输入的字符的命令。如果在没有输入的情况下按下这两个键，就和光标移动上下键的功能是一样的。

（2）按 Ctrl + r 键查找，将会在命令历史中寻找包含输入字符的命令，并尽可能的补全。

3. 使用“_”符号访问输出结果

在 IPython 和标准 Python 的命令行中，不仅可以访问输入的命令，而且还可以访问输出结果。具体的方法是使用“_”字符。这表示最后的输出。

【例 22-6】演示使用“_”字符访问输出结果的方法。

```
In [1]: test = "IPython"

In [2]: _
Out[2]: ''
In [3]: test
Out[3]: 'IPython'

In [4]: _
Out[4]: 'IPython'

In [5]: test = _

In [6]: test
Out[6]: 'IPython'
```

- 在 In［1］中定义了一个 test 变量，这并没有输出结果，或者说是返回结果是“None”。当在 In［2］中使用“_”的时候，可以看到为空字符串。
- 在 In［3］中输出了 test 字符串，结果为 test 变量的值。此时，在 In［4］中引用“_”，就变成了 test 变量的值。
- 在 In［5］中再度进行了赋值，可以看到同样输出了 test 的值。

在标准 Python 命令行中，也具有这样的功能。两者的一个区别是，当系统返回为“None”的时候，在标准 Python 命令行中将触发 NameError 异常：

```
>>> test = "Python"
>>> _
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> test
'Python'
>>> _
'Python'
```

可以看到，在第一次调用“_”的时候出现了 NameError 异常。

在 IPython 的历史记录功能中，实际上还是对其中 In 和 Out 的相关操作。因为在 In 和 Out 中存储了输入和输出的信息，所以可以很方便的将这些信息输出。在输出结果的同时可以根据 history 命令的选项做一些调整。这在前面和 IPython 的交互过程中所打印的 In 和 Out 的值可以看出来。对于 In 的列表类型，可以直接使用 In［:］得到原来列表的一个复制。同样地，在 Out 字典中的值可以存储为任何类型。当其类型为列表的时候，可以使用列表的相关操作来对其进行操作。这也为操作外部系统命令的输出提供了一种方法。

更多资源请访问稀酷客(www.ckook.com)

22.4.3　执行外部的系统命令和运行脚本文件

在 IPython 中，可以很容易的执行外部系统命令和运行文件。

1. 使用“!”执行外部系统命令

IPython 使用“!”符号来执行外部系统命令，也可以用“$”符号将 Python 变量转化为 Shell 变量。通过这两个符号的有机结合，可以有效的执行 IPython 与外部系统命令之间的交互以及外部文件，进一步完成复杂的工作：

```
In [1]: cd E:\Python25
E:\Python25

In [2]: ! dir
 驱动器 E 中的卷是 PROG
 卷的序列号是 C413-1C2C

 E:\Python25 的目录

2008-10-24  02:16    <DIR>          .
2008-10-24  02:16    <DIR>          ..
2008-06-22  20:02    <DIR>          DLLs
2008-06-22  20:02    <DIR>          Doc
2008-06-22  20:02    <DIR>          include
2008-10-24  00:03            79,167 ipython-wininst.log
2008-10-28  16:06    <DIR>          Lib
2008-06-22  20:02    <DIR>          libs
#省略部分输出结果
```

在上面，首先将目录定位到 Python 的根目录下，然后调用外部系统命令 dir 将目录中的文件输出。注意到在前面添加了“!”，表示后面的命令将由外部 Shell 来解析。

```
In [3]: pattern = "*.exe"

In [4]: ! dir $pattern
 驱动器 E 中的卷是 PROG
 卷的序列号是 C413-1C2C

 E:\Python25 的目录

2008-02-21  13:11            24,064 python.exe
2008-02-21  13:12            24,576 pythonw.exe
2008-10-24  00:03            61,440 Removeipython.exe
2008-10-24  00:03            61,440 Removepyreadline.exe
2008-02-21  13:11             4,608 w9xpopen.exe
               5 个文件        176,128 字节
               0 个目录 25,324,122,112 可用字节
```

在 In［3］中，定义了一个 Python 的变量，这个变量是一种搜索模式“*.exe”。在前面加上“$”符号就可以将此 Python 变量转化为可以在外部 Shell 中使用的变量。这从 In［4］的输入命令可以看出来。直接使用 $ pattern 放在 dir 命令的后面，使得只有满足此搜索模式的文件或者文件夹才会显示出来。这里，所有的以“exe”结尾的文件名都被输出。

2. 使用“$”操作符需要注意的问题

需要注意的是，在“$”符号后面只能是简单的变量名，而不会对其做 Python 运算：

更多资源请访问稀酷客(www.ckook.com)

```
In [5]: ! dir $(pattern[:-3] + "dll")
 驱动器 E 中的卷是 PROG
 卷的序列号是 C413-1C2C

 E:\Python25 的目录

找不到文件

In [6]: new_pattern = pattern[:-3] + "dll"

In [7]: ! dir $new_pattern
 驱动器 E 中的卷是 PROG
 卷的序列号是 C413-1C2C

 E:\Python25 的目录

2004-12-16  18:22           499,712 msvcp71.dll
2004-12-16  18:22           348,160 msvcr71.dll
2004-06-10  18:21           219,136 unicows.dll
               3 个文件      1,067,008 字节
               0 个目录 25,324,122,112 可用字节
```

- In [5] 中，后面的“$（pattern[:-3] +"dll")”运算是试图将搜索模式变为“*.dll”，但是实际上，外部系统命令并没有找到任何文件。换句话说，这里没有对这个运算进行展开，以得到最终的值。在这里，使用（）后得到的值为“None”，所以最终输出为“找不到文件”。
- In [6] 和 In [7] 中，首先对“$（pattern[:-3] +"dll")”进行了计算，并赋值给 new_pattern 变量，然后再次调用 dir 外部命令，这个时候就得到了所期望的结果。

在调用外部系统命令的时候，如果 Python 变量不存在，则会触发 NameError 异常：

```
In [8]: ! dir $null
ERROR: An unexpected error occurred while tokenizing input
The following traceback may be corrupted or invalid
The error message is: ('EOF in multi-line statement', (479, 0))

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
#省略部分输出结果
```

- In [8] 中，试图解析一个不存在的 Python 变量，触发了 NameError 异常，同时也中断了外部系统命令的输出。

需要注意的是，在外部系统命令（包括参数）中，如果有“$”，则需要使用“$”将其转义。也就是说，两个“$”符号才能真正表示在外部 Shell 中的 $ 符号。

【例 22-7】假设在当前目录下有个 $ test. txt 文件，将其内容打印出来。

```
In [9]: ! type $test.txt
ERROR: An unexpected error occurred while tokenizing input
The following traceback may be corrupted or invalid
The error message is: ('EOF in multi-line statement', (479, 0))

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
#省略部分输出结果
NameError: name 'test' is not defined
```

更多资源请访问稀酷客(www.ckook.com)

```
In [10]: ! type $ $test.txt
This is a file for testing.
In [11]:
```

- 在 In［9］中，试图打印出 $ test. txt 的值。这时系统会对 $ test 进行解析。当发现 test 的 Python 变量不存在的时候，触发了 NameError 异常。
- 在 In［10］中，使用了“ $ $ test. txt”方式，从而使得外部系统命令可以正常的解析为“type $ test. txt”，得到了期望的结果。

3. 将系统命令的输出结果赋值给 Python 变量

实际上，对于使用“!”符号的外部系统命令来说，可以将输出结果复制给某个 Python 变量，留待以后使用：

```
In [11]: result = ! dir $pattern

In [12]: result
Out[12]: SList (.p,.n, .l, .s, .grep(), .fields(), sort() available):
0:  驱动器 E 中的卷是 PROG
1:  卷的序列号是 C413-1C2C
2:
3:  E:\Python25 的目录
4:
5: 2008-02-21  13:11            24,064 python.exe
6: 2008-02-21  13:12            24,576 pythonw.exe
7: 2008-10-24  00:03            61,440 Removeipython.exe
8: 2008-10-24  00:03            61,440 Removepyreadline.exe
9: 2008-02-21  13:11             4,608 w9xpopen.exe
10:              5 个文件        176,128 字节
11:              0 个目录 25,324,122,112 可用字节
```

- 在 In［11］中，将特定搜索模式下的文件输出结果赋值给了 Python 的变量 result。
- 在 In［12］中输出了 result 的值。可以看到，这个输出结果和前面直接使用外部系统命令输出的结果还是有一些区别的。前面的输出结果仅仅只是输出到终端而已，而这里保存下来的命令输出结果则是一个列表结构，可以对其做进一步处理。

4. “!!”操作符的使用方法及其和“!”操作符的区别

替代“!”符号的一种选择是“!!”符号。两者的区别主要在于后者的输出结果直接就是一个列表结构，并且输出结果不能赋值给某个 Python 变量：

```
In [13]: !! dir $pattern
Out[13]: SList (.p,.n,.l,.s,.grep(),.fields(), sort() available):
0:  驱动器 E 中的卷是 PROG
1:  卷的序列号是 C413-1C2C
2:
3:  E:\Python25 的目录
4:
5: 2008-02-21  13:11            24,064 python.exe
6: 2008-02-21  13:12            24,576 pythonw.exe
7: 2008-10-24  00:03            61,440 Removeipython.exe
8: 2008-10-24  00:03            61,440 Removepyreadline.exe
9: 2008-02-21  13:11             4,608 w9xpopen.exe
10:              5 个文件        176,128 字节
11:              0 个目录 25,324,122,112 可用字节

In [14]: result = !! dir
```

更多资源请访问稀酷客(www.ckook.com)

```
'! dir' 不是内部或外部命令,也不是可运行的程序
或批处理文件。
```

- 在 In［13］中演示了“!!”符号的用法。可以看到，它和 Out［12］输出的结果是一样的。
- 当试图将输出结果赋值给 result 变量的时候，IPython 将报错。赋值给 Python 变量的功能仅仅适用于使用“!”符号的系统命令。

当然，由于“!!”符号命令的输出结果是列表结构，所以可以直接访问 Out 来获得输出的值。如果仅仅是只需要取得最后的输出值，还可以使用“_”符号。

在 IPython 的实际处理中，使用”!”符号其实仅仅只是简单的调用 system 函数（参见第 21 章中的 21.22 节）。而针对“!!”符号的处理实际上是使用了 magic 函数 sx，有下面的关系：

```
!! command
== >
% sx command
```

其中 sx 命令也是一种调用外部系统命令的方式。

5. 运行外部的脚本文件

除了可以直接在 IPython 解释器中交互运行代码外，还可以直接执行外部的脚本文件。一种简单的方式是使用“!”符号调用外部命令 python 来执行脚本。另外一种方法是通过 magic 函数中的 run 命令来实现。使用方法是直接在“run”命令后面输入文件名即可。注意，如果在当前目录下有一个文件 test.py，则该文件名也是可以补全的：

```
#filename: test.py
def hello(arg):
print "Hello,", arg

hello("IPython")
```

这里的代码段很简单，就是打印出字符串。

当需要在 IPython 下执行 test.py 文件的时候，有下面的两种运行方式：

```
In [15]: ! python test.py
Hello, IPython

In [16]: % run test.py
Hello, IPython
```

从上面的输出结果可以看到，两种方式得到了同样的结果。

22.4.4　对象的查看和自省

Python 中的任何数据都是对象。可以使用代码查看内存中通过对象而存在的其他模块或者函数，获取其信息，并对其进行操作。这种功能就是自省。在 Python 中，其实就已经具有了这种强大的功能了，如最简单的 dir 函数：

```
In [1]: import sys

In [2]: print dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__st
din__', '__stdout__', '_current_frames', '_getframe', 'api_version', 'argv', 'bu
iltin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'dis
playhook', 'dllhandle', 'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec
```

更多资源请访问稀酷客(www.ckook.com)

```
_prefix', 'executable', 'exit', 'exitfunc', 'getcheckinterval', 'getdefaultencod
ing', 'getfilesystemencoding', 'getrecursionlimit', 'getrefcount', 'getwindowsve
rsion', 'hexversion', 'ipcompleter', 'maxint', 'maxunicode', 'meta_path', 'modul
es', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'setchec
kinterval', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 's
tdout', 'subversion', 'version', 'version_info', 'warnoptions', 'winver']
```

在 In［1］中，首先导入了 sys 模块。然后在 In［2］中，使用 dir（sys）得到了此模块中支持的所有方法和成员。

1. “?”和“??”操作符的使用方法

使用上面的 dir（）函数已经很方便了，但 IPython 的两个操作符可以使得这种功能更加强大。这两个操作符是“?”和“??”。两种操作的区别在于前者会截断输出的长字符串，而后者不会。同时，如果有源码的话，“??”操作符还会高亮输出源代码。这同时还可以做帮助用：

```
In [1]: import os

In [2]: os.system?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function system>
Namespace:      Interactive
Docstring:
    system(command) -> exit_status

Execute the command (a string) in a subshell.

In [3]: os.system??
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function system>
Namespace:      Interactive
Docstring [source file open failed]:
    system(command) -> exit_status

Execute the command (a string) in a subshell.
```

上面演示了“?”和“??”操作符的使用方法。注意到，两者给出的信息都非常详细，包括函数方法的类型、基类等。需要注意的是，在两种输出结果中，可以看到后者在 Docstring 的后面有“source file open failed”的信息。这是因为找不到源文件，所以没有给出源代码。另外，由于印刷的原因，实际上的彩色显示这里也是没有办法看到的。

使用“?”和“??”操作符不仅可以对 Python 的对象进行查询，还可以查看 magic 函数的方法甚至是源码：

```
In [29]: %sx??
Type:           Magic function
Base Class:     <type 'instancemethod'>
String Form:    <bound method InteractiveShell.magic_sx of <IPython.iplib.Intera
ctiveShell object at 0x00B49D70>>
Namespace:      IPython internal
File:           e:\python25\lib\site-packages\ipython\magic.py
Definition:     %sx(self, parameter_s='')
Source:
    def magic_sx(self, parameter_s=''):
        """Shell execute - run a shell command and capture its output.
```

更多资源请访问稀酷客(www.ckook.com)

```
        % sx command

        IPython will run the given command using commands.getoutput(), and
        return the result formatted as a list (split on '\\n').  Since the
#省略部分输出结果
```

在上面可以看到，当系统中有 Python 对象的源码的时候，将会对源码中的成员函数进行高亮显示。

2. 使用 who、who_ls 和 whos 查看对象信息

IPython 还提供了一组 magic 函数来对已有的对象进行查看，主要包括 who、who_ls 和 whos 等。Who 函数的使用方法很简单，将直接返回 IPython 中已有的对象。同时，who 还可以对对象进行过滤：

```
In [1]: a = 0

In [2]: b = "IPython"

In [3]: c = [i for i in range(5)]

In [4]: who
a       b       c       i
In [5]: who int
a       i

In [6]: who str
b

In [7]: who list
c
```

- 在前面的 3 个输入中，分别给 3 个变量赋值为整数、字符串和列表。
- 在 In［4］中使用了 who 函数，可以看到在 IPython 交互式环境中定义的对象。需要注意的是，在 c 的定义中，使用了变量 i，这也出现在了 who 的输出结果中。
- 后面三个输入、输出主要是查看 who 的结果过滤功能。可以看到，针对不同的过滤，who 输出了不同的结果。

who 函数仅仅是输出相关的类型信息，但是并没有返回值。也就是说，不能对其输出结果进行后续的操作。为了解决这个问题，IPython 提供了 who_ls 函数，用来将输出返回：

```
In [8]: who_ls
Out[8]: ['a', 'b', 'c', 'i']

In [9]: who_ls int
Out[9]: ['a', 'i']
```

这样可以使用“_”来获取上次输出的值：

```
In [10]: print _
['a', 'i']
```

如果需要获取对象的更多信息，可以使用 whos 函数。该函数的用法和 who 相同，who 的过滤也是可以用在 whos 命令上的：

```
In [11]: whos
Variable   Type    Data/Info
-----------------------------
```

更多资源请访问稀酷客(www.ckook.com)

```
a       int     0
b       str     IPython
c       list    [0, 1, 2, 3, 4]
i       int     4
```

每种对象有着各自详细的信息输出。如对于字符串，将会输出字符串。对于列表和元组等，则会打印出其长度。如果输出长度过长，将会截断中间的输出。

22.4.5　直接编辑代码

虽然在命令行中编辑代码是可行的，但是很多时候还是希望能够在编辑器中来编辑代码。另外，为了每次都生成一个代码文件也是不必要的。在 IPython 中，通过 magic 函数中的 edit 命令，可以在设置的编辑器中编辑代码，同时又不会生成文件。

1. edit 命令的使用方法

edit 命令会根据环境变量 EDITOR 的设置来调用相应的编辑器。如果环境变量中没有设置编辑器，则调用 vi（Linux 操作系统下）或者记事本（Windows 操作系统下）。也可以在配置文件中进行设置。退出编辑器则自动回到 IPython 环境下。同时，编辑器中输入的代码将会在当前名字空间下被 Python 解释器执行。直接输入 edit 命令将会打开编辑器，如图 22-4 所示。

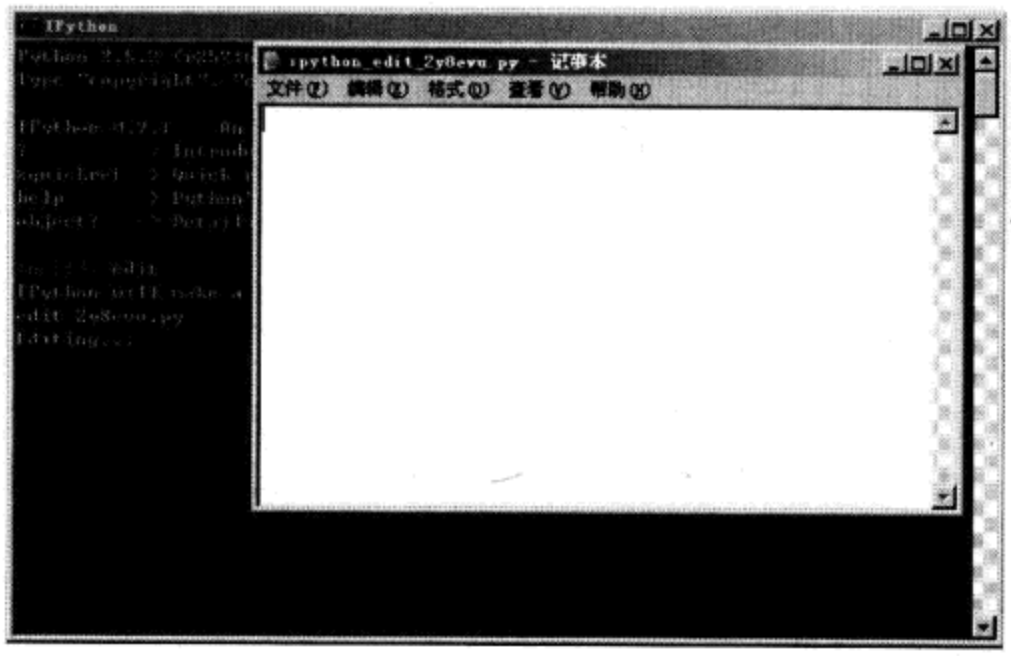


图 22-4　使用 IPython 中的 edit 命令

从图 22-4 所示窗口中可以看到，当输入 edit 命令的时候，IPython 将会调用外部的编辑器，图 22-4 中调用的是记事本程序。注意到，实际上是生成了一个临时文件。在此临时文件中可以直接输入程序代码。

【例 22-8】直接在编辑器中编辑一个打印出素数的函数。

```
def prime_numbers(number):
    if number < = 3:
        return range(2,number)

    primes = [2] +  range(3,number,2)

    index = 1
    max = number * * 0.5
    while 1:
        i = primes[index]
        if i > max:
            break
        index + =1
```

更多资源请访问稀酷客(www.ckook.com)

```
        primes = [x for x in primes if (x % i) or (x == i)]
    return primes
```

输入完代码后，关闭编辑器，IPython 会立即执行这一段代码。如下所示：

```
In [1]: edit
IPython will make a temporary file named: c:\docume~1\sea\locals~1\temp\ipython_
edit_2y8evu.py
Editing...done.Executing edited code...
Out[1]: 'def prime_numbers(number):\n    if number <= 3:\n        return range(2
,number)\n\n    primes = [2] + range(3,number,2)\n\n    index = 1\n    max = nu
mber**0.5\n    while 1:\n        i = primes[index]\n        if i>max:\n
    break\n        index += 1\n        primes = [x for x in primes if (x % i) o
r (x == i)]\n    return primes'
```

从输出的信息中可以看到 IPython 详细的处理过程。

另外，从上面的 Out［1］中可以看到，IPython 将会把输入的内容作为 Out 输出。当 IPython 解析了这一段代码后，在当前的作用范围内就生成了一个 prime_numbers 函数。可以直接使用这个函数，来得到不同的素数表：

```
In [2]: prime_numbers(34)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

2. 对上一次代码进行修改

很多时候，可能是需要对上一次修改后的代码片段继续进行修改，这时就可以使用 edit - p 命令。在图 22-5 中清晰的显示了这一点。

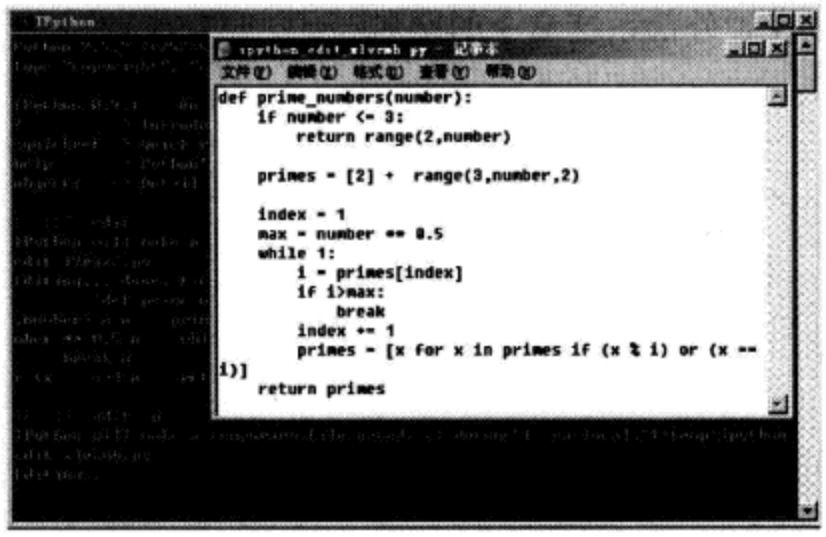


图 22-5　IPython 中的 edit -p 使用

从图 22-5 中可以看出，当使用 edit - p 命令时，IPython 将调用编辑器，并同时在其中填入上一次的所输入的代码。

3. 修改而不执行代码

当只是修改代码而并不希望立即对代码进行解析的时候，可以使用 edit -x 命令：

```
In [3]: edit -x
IPython will make a temporary file named: c:\docume~1\sea\locals~1\temp\ipython_
edit_blfqbt.py
Editing...
Out[3]: 'def prime_numbers(number):\n    if number <= 3:\n        return range(2
,number)\n\n    primes = [2] + range(3,number,2)\n\n    index = 1\n    max = nu
mber**0.5\n    while 1:\n        i = primes[index]\n        if i>max:\n
    break\n        index += 1\n        primes = [x for x in primes if (x % i) o
r (x == i)]\n    return primes'
```

更多资源请访问稀酷客(www.ckook.com)

注意到这里的输出信息中为“Editing…”，而不是“Editing... done. Executing edited code…”。这表示仅仅只是将这些输入保存在了 Out［3］中，并没有对代码进行执行。如果这时对代码进行修改，则不能够反映到 IPython 中来。



图 22-6　IPython 中的历史记录分片源码编辑

IPython 中源码编辑的一个强大功能是可以对历史记录中的代码片段进行编辑。命令格式为“edit n：m”，其中 n 和 m 都为历史记录的数字。其中的输入为包括 n 但是不大于 m 的历史记录。当 n 大于 m 或者 n 和 m 都为 0 的时候，此命令和 edit 命令的效果是相同的。一个简单的示例代码如图 22-6 所示。

在图 22-6 所示窗口中可以看到，这里使用的历史记录分片是“1：4”，所以将会在编辑器里出现输入历史记录为 1、2、3 所输入的内容。最终显示结果和预期的是一致的。

22.4.6　设置别名和宏

为了能够加速命令的键入，可以对比较复杂的命令设置一个别名。如果代码过长，可以使用宏来代替。使用别名或者宏就像使用原来的命令或者代码一样。

1. 设置别名

使用 alias 命令，可以设置外部系统命令的别名：

```
In [1]: alias ns netstat -an

In [2]: ns

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1110           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3389           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:19780          0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1029         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:27086        127.0.0.1:27087        ESTABLISHED
#省略部分输出结果
```

- 在 In［1］中定义了一个别名 ns，用来表示系统命令 netstat -an，在 In［2］中，当使用 ns 命令的时候，就好像使用！netstat -an 命令一样。

可以使用%l 标识符来表示使用别名的时候输入的参数：

```
In [3]: alias myecho echo "Input: <% l>"

In [4]: myecho test
"Input: <test>"
```

可以使用“%s”来针对每个输入参数进行更详细的设置：

```
In [5]: alias myecho2 echo first % s second % s

In [6]: myecho2 arg1 arg2
first arg1 second arg2
```

更多资源请访问稀酷客(www.ckook.com)

```
In[7]: myecho2 arg1
ERROR: Alias <myecho2> requires 2 arguments, 1 given.
#省略部分输出结果
```

- 在 In［5］中，定义了一个别名 myecho2。在其中用了两个%s 来表示在别名使用时候的参数，它们都是作为字符串来输入的。
- 在 In［6］和 In［7］中，演示了 myecho2 的用法。首先是在使用的时候输入了两个参数，这个时候别名得到了正确的输出结果。而第二次则只是输入了一个参数，此时别名运行就报错了，显示此别名需要两个参数，而现在只提供了一个参数。

当在 alias 后不使用任何参数的时候，将输出当前的 alias 对应表。

2. 设置宏

在 Python 的语言标准中没有宏的概念，但是 IPython 做了扩展，使得宏的功能成为交互式环境中的一个重要特性。这是通过 magic 中的 macro 命令实现的。宏允许用户为一段代码定义一个名字，从而可以使用此名字来反复的运行此段代码。其语法为在 macro 命令后面是所定义宏的名字和其代表的代码段。其中，代码段是历史记录的分片：

```
In [1]: import datetime

In [2]: datetime.date.today() + datetime.timedelta(days=1)
Out[2]: datetime.date(2008, 11, 11)

In [3]: macro tomorrow 2
Macro 'tomorrow' created.To execute, type its name (without quotes).
Macro contents:
datetime.date.today() + datetime.timedelta(days=1)

In [4]: tomorrow
- - - - - - -> tomorrow()
Out[5]: datetime.date(2008, 11, 11)

In [6]: macro
Out[6]: ['tomorrow']
```

- 在前面的两个输入中，输入了可以求出明天时间的代码。在 In［3］中，使用 macro 命令定义了一个名字为“tomorrow”的宏。接着在 In［4］中，可以看到宏的使用方法。使用宏的目的相当于调用其所代表的代码。需要注意的是，如果在一个全新的 IPython 环境中调用此宏，则需要再度导入 datetime 模块，否则将会报错。因为宏仅仅只是一个代码的缩写而已。
- 如果不加任何参数，则和 alias 命令一样，也是输出现在 IPython 中已有的宏定义。

当然，可以编辑宏并将其写入文件，从而长期保存下来供以后使用。

22.5　使用 Python 进行文件管理

在第 7 章中已经介绍了 Python 对文件的部分处理方式。由于对于一个系统的管理最终将会归结到系统的信息的管理上，所以本节将继续对 Python 的文件管理功能进行介绍。这些高级部分包括对文件进行比较、归档、压缩等操作。

22.5.1　文件的比较

对于一个系统管理员来说，比较文件是会经常遇到的。在具有版本管理的系统中，需要使用

更多资源请访问稀酷客(www.ckook.com)

这种功能来判断哪些文件经过了改动，哪些文件是相同的。当然，一种可能的办法是使用在第 7 章中介绍的方法：对于需要比较的文档，依次读取每个数据进行比较。虽然这种方法是可行的，但是效率却非常低。所以有必要寻找其他更好的方法。

在 Python 的标准模块中，filecmp 模块提供了文件的比较功能。filecmp 模块提供了 cmp 和 cmpfiles 这两个方法，分别针对文件和目录进行比较。

【例 22-9】在当前目录下生成 3 个文件：text1. txt、text2. txt 和 text3. txt。其中，前两个文件没有任何内容，在 text3. txt 文件中写入随意的数据。使用 cmp 方法对两个文件进行比较。

```
In [1]: import filecmp

In [2]: filecmp.cmp("text1.txt","text2.txt")
Out[2]: True

In [3]: filecmp.cmp("text1.txt","text3.txt")
Out[3]: False
```

- cmp 方法的使用是直接在后面输入两个需要比较的文件名即可。当文件内容相同的时候，将会显示出“True”，否则显示“False”，表示文件不相等。

【例 22-10】接上例。为了演示 cmpfiles 的使用方法，在当前目录下再新建两个目录：test1 和 test2。将【例 22-9】生成的 3 个文件都拷贝到 test1 文件夹中，同时将 text3. txt 的内容清空。再将 text1. txt 和 text3. txt 拷贝到 test2 文件夹中。现在，目录结构如下所示：

```
In [4]: ! tree /F
卷 BOOK 的文件夹 PATH 列表
卷序列号码为 7C935A65 18F8:F57A
I:.
|   text1.txt
|   text2.txt
|   text3.txt
|
├─test1
|       text1.txt
|       text2.txt
|       text3.txt
|
└─test2
        text1.txt
        text3.txt
```

然后使用 cmpfiles 方法来比较 test1 和 test2 这两个目录：

```
In [5]:
filecmp.cmpfiles("test1","test2",["text1.txt","text2.txt","text3.txt"])
Out[5]: (['text1.txt'], ['text2.txt'], ['text3.txt'])
```

- cmpfiles 方法有 3 个参数，其中前面两个参数为需要比较的目录，第 3 个为需要比较的文件列表。
- cmpfiles 方法返回 3 个列表，分别表示匹配的文件、不匹配的文件和比较出错的文件。这里出错的可能性包括文件缺失、权限不足等。从这里的输出结果可以看到，此方法正确的给出了目录中指定文件的比较情况。

注意，cmpfiles 方法需要提供比较的文件列表。当文件比较多的时候，这个方法就显得繁琐了。在 filecmp 模块中，还有一个 dircmp 类，提供了比较目录的许多高级方法。其构造函数很简单，直接输入两个目录即可。另外，还有两个可选的参数，分别是 ignore 和 hide。ignore表示忽略的目录，hide 表示需要隐藏的名字。构造完毕后，可以使用类的方法 report 来查看目录比较的结果：

更多资源请访问稀酷客(www.ckook.com)

```
In [6]: result = filecmp.dircmp("test1","test2")

In [7]: result.report()
diff test1 test2
Only in test1 : ['text2.txt']
Only in test2 : ['text3.txt']
Identical files : ['text1.txt']
```

更方便的是，dircmp 类中提供了很多的成员，可以很方便的求得特定的信息，如只在左边出现的文件等。表 22-2 中显示了 dircmp 类所提供的成员列表。

表 22-2　dircmp 类中所支持的成员属性

成员属性	说明（所比较的两个目录分别为 a 和 b）
left_list	在 a 中被忽略或者隐藏的文件和文件夹
right_list	在 b 中被忽略或者隐藏的文件和文件夹
common	在 a 和 b 中都出现的文件和文件夹
left_only	只在 a 中出现的文件和文件夹
right_only	只在 b 中出现的文件和文件夹
common_dirs	在 a 和 b 中都出现的文件夹
common_files	在 a 和 b 中都出现的文件
common_funny	在 a 和 b 中都出现但是比较出错的名字
same_files	在 a 和 b 中都相同的文件
diff_files	在 a 和 b 中都出现且不相同的文件
funny_files	在 a 和 b 中都出现，但是不能比较的文件
subdirs	一个字典，其中有 common_ dir 中的目录到 dircmp 对象的映射

当然，为了简单快速的比较文件，可以比较文件的 md5 值。如果 md5 的值不相同，则文件是不相等的。

22.5.2　文件的归档

当系统中有很多的文件时，为了能够有效地进行管理，一种方法是将这些文件进行归档。系统管理员经常会使用如 tar 类的命令来管理这些文件。在 Python 中的 tarfile 模块可以有效地对归档文件进行管理，包括生成、检测和解开等。

tarfile 模块提供了一个 open 方法可以用来打开 tar 文件，并返回一个 TarFile 对象。在 TarFile 类中，提供了丰富的方法用于对文件进行操作：

```
In[1]: import tarfile

In[2]: tar = tarfile.open("textfile.tar","w")

In[3]: tar.add("text1.txt")
In[4]: tar.add("text2.txt")
In[5]: tar.add("text3.txt")

In [6]: tar.add("test1")
In [7]: tar.add("test2")

In [8]: tar.close()
```

- 在 In［1］中导入了 tarfile 模块。

更多资源请访问稀酷客(www.ckook.com)

- In［2］中使用 tarfile 的 open 方法，在写模式下打开了一个 textfile. tar 文件，供后面进一步操作。
- 在 In［3-7］中加入了 3 个文本文件和两个目录。注意到添加目录的时候，其中的文件都会加入进来。最后调用 close 方法关闭了 TarFile 对象。

这时查看当前目录，可以看到生成了一个 textfile. tar 文件，并且其中有 text1. txt 等 3 个文本文件和两个目录。这可以使用 TarFile 类中的 list 方法来看到：

```
In [9]: tar = tarfile.open("textfile.tar","r")

In [10]: tar.list()
-rw-rw-rw- user/group        22 2008-11-11 12:06:35 text2.txt
-rw-rw-rw- user/group         2 2008-11-11 11:20:31 text3.txt
-rw-rw-rw- user/group        22 2008-11-11 12:06:30 text1.txt
drwxrwxrwx user/group         0 2008-11-11 11:14:06 test1/
-rw-rw-rw- user/group         0 2008-11-11 11:07:13 test1/text1.txt
-rw-rw-rw- user/group         0 2008-11-11 11:07:13 test1/text2.txt
drwxrwxrwx user/group         0 2008-11-11 11:23:44 test2/
-rw-rw-rw- user/group         0 2008-11-11 11:17:13 test2/text1.txt
-rw-rw-rw- user/group         2 2008-11-11 11:20:31 test2/text3.txt
In [11]: tar.close()
```

- 在 In［9］中，注意到这里是使用读模式将文件打开的，这就保证了在对 TarFile 类的操作过程中不会破坏源文件。如果使用写模式打开，会将已归档文件中的所有源文件删除。
- 通过 In［10］中的 list 方法，可以看到加入了上面的 3 个文本文件。
- 最后要记得调用 close 方法将其关闭。

使用 list 方法是将这些信息打印出来，如果需要返回这些获取的文件信息，可以使用 getnames方法。同样地，name 和 members 成员属性也可以提供相关的一些信息：

```
#省略文件的打开和关闭过程
In [16]: tar.getnames()
Out[16]:
['text2.txt',
 'text3.txt',
 'text1.txt',
 'test1/',
 'test1/text1.txt',
 'test1/text2.txt',
 'test2/',
 'test2/text1.txt',
 'test2/text3.txt']

In [17]: tar.name
Out[17]: 'I:\\book\\source\\sysadmin\\textfile.tar'

In [18]: tar.members
Out[18]:
[<TarInfo 'text2.txt' at 0x149c630>,
 <TarInfo 'text3.txt' at 0x149c890>,
 <TarInfo 'text1.txt' at 0x149c990>,
 <TarInfo 'test1/' at 0x15edc70>,
 <TarInfo 'test1/text1.txt' at 0x15edd10>,
 <TarInfo 'test1/text2.txt' at 0x15edcb0>,
 <TarInfo 'test2/' at 0x149c6f0>,
 <TarInfo 'test2/text1.txt' at 0x149cc30>,
 <TarInfo 'test2/text3.txt' at 0x149cb50>]
```

- 在 In［16］中，使用 getnames 方法得到了一个归档文件中的所有文件的列表，可以使

更多资源请访问稀酷客(www.ckook.com)

用“_”符号来获取这个值，以便做出进一步处理。

- In［17］中的 name 成员属性显示了此归档文件的全路径。
- 在 In［18］中显示了此归档文件所包含的所有成员，每个都是一个 TarFile 对象。

如果需要解开归档文件，只需调用 TarFile 类中的 extractall 方法即可。这个方法可以加上两个可选的参数。一个是需要解开的路径（默认值为当前工作目录）。另外一个是需要解开的成员（默认值为解开所有的成员）：

```
#省略了文件的打开和关闭过程
In [23]: tar.extractall("tmp")

In [24]: ! tree /F
卷 BOOK 的文件夹 PATH 列表
卷序列号码为 7C935A65 18F8:F57A
I:.
|   #省略了部分输出结果
|   textfile.tar
|
└─tmp
    |   text1.txt
    |   text2.txt
    |   text3.txt
    |
    ├─test1
    |       text1.txt
    |       text2.txt
    |
    └─test2
            text1.txt
            text3.txt
```

- 这里省略了文件打开和关闭的过程。需要注意的是，只有在适合的模式下打开文件才可以调用 extractall 方法。
- 在 In［23］中，使用 extractall 方法将所有的成员都解开到 tmp 目录下。通过 In［24］的输出结果可以看到，tmp 目录完整的保持了归档文件的目录形式。

22.5.3　文件的压缩

在上一小节中已经实现了文件的归档，但有时还需要对文件进行压缩。特别是在有大量的文本文件的时候，如各种日志，在归档后因为占用磁盘空间巨大，所以都是需要压缩后才进行保存的。

由于压缩算法和压缩格式的不同，在 Python 的标准库中有多种与压缩相关的模块。这些模块主要有 zlib、gzip、bz2 和 zipfile 等。其实，在上面使用 tarfile 模块操作归档文件的时候就可以在其中压缩，一步实现归档和压缩操作。

为了能够查看压缩的效果，首先使用下面的代码来生成一个大数据文件：

```
In [2]: f = open("large.txt","w")
In [3]: for i in range(200000):
   ...:     f.write("IPython is an enhanced interactive Python shell") #写入文件
   ...:

In [5]: f.close()
```

- 在 In［2］中，首先使用写模式打开了一个 large.txt 文件。接着在第二个输入语句中将 200 000 个字符串写入到该文件中。

可以看到该文件的大小大概为 335MB 左右：

更多资源请访问稀酷客(www.ckook.com)

```
In [12]: ls
 驱动器 I 中的卷是 BOOK
 卷的序列号是 18F8-F57A

 I:\book\source\sysadmin\tmp 的目录

2008-11-03  03:22     <DIR>          .
2008-11-03  03:22     <DIR>          ..
2008-11-03  03:18       351,990,050 large.txt
               1 个文件    351,990,050 字节
               2 个目录  8,658,436,096 可用字节
```

按照上一小节中创建归档文件的方式来创建这个压缩文件：

```
In [14]: tar = tarfile.open("large.tar.gz","w:gz")

In [15]: tar.add("large.txt")

In [16]: tar.close()

In [17]: ls
 驱动器 I 中的卷是 BOOK
 卷的序列号是 18F8-F57A

 I:\book\source\sysadmin\tmp 的目录

2008-11-11  13:24     <DIR>          .
2008-11-11  13:24     <DIR>          ..
2008-11-11  13:25         1,024,531 large.tar.gz
2008-11-11  13:18       351,990,050 large.txt
               2 个文件    353,014,581 字节
               2 个目录  8,657,408,000 可用字节
```

- 在 In［14］中，使用 w：gz 方式打开了归档文件。这个打开模式表示在写入的同时打开 gz 的压缩。后面的操作和归档文件的操作是一样的。因为压缩操作一般是比较慢的，所以在添加大数据文件的时候，可能会耗时比较长。
- 在 In［17］中，比较了一下文件压缩前后的大小。可以看到，对于这种规则的文本文件，压缩比例是很高的（压缩前为 335MB，压缩后为 1MB）。

在 tarfile 模块中，还支持使用 bz2 的方式压缩：

```
In [18]: tar = tarfile.open("large.tar.bz","w:bz2")

In [19]: tar.add("large.txt")

In [20]: tar.close()

In [21]: ls
 驱动器 I 中的卷是 BOOK
 卷的序列号是 18F8-F57A

 I:\book\source\sysadmin\tmp 的目录

2008-11-11  11:31     <DIR>          .
2008-11-11  11:31     <DIR>          ..
2008-11-11  11:25         1,024,531 large.tar
2008-11-11  11:52            72,739 large.tar.bz
2008-11-11  11:18       351,990,050 large.txt
               3 个文件    353,087,320 字节
               2 个目录  8,656,932,864 可用字节
```

更多资源请访问稀酷客(www.ckook.com)

- 这里和前面的操作没有什么区别，唯一不同的地方是在打开模式中指定使用的是 bz2 方式进行压缩。
- 在添加文件的时候，这里所耗费的时间比前面的要长得多。这是因为，bz2 使用了一种压缩率更高的算法，从而消耗的时间也要长很多。在后面的文件大小比较中也可以看出两种方法的区别：使用 bz2 方式压缩后的文件竟然只有 70 多 KB 大小。

对于文件的压缩，还有很多第三方提供的模块。由于它们采用的是不同的压缩算法和压缩率，因此所需的时间也是不同的。可以根据需要来选择压缩的模块。

22.6　使用 Python 定时执行任务

在系统管理中，经常需要定时执行某个任务，如将任务放在计划任务中。使用 Python 可以很简单的实现这一点。Python 标准模块中的相关模块主要有 sched 等。

22.6.1　使用休眠功能

如果需要定期的执行某个命令，一个很简单的方法是在需要等待的时间内使得程序进入休眠状态。time.sleep（）可以很好的完成这个任务。

【例 22-11】每过 60 秒钟在终端上显示当前计算机的网络连接情况。

```
#filename: sleeping_1.py
import time
import os

def main (cmd, inc = 60):
    while True:
        os.system (cmd)
        time.sleep (inc)

main (" netstat -an", 60)
```

- 在最前面导入了 time 模块，其中包含许多和时间相关的方法，sleep 就是其中一个。
- 这里的 main 函数有两个参数，一个是需要执行的命令，另外一个是等待的时间。函数的结构很简单，就是一个无休止的 while 循环。在执行命令后，将 sleep 休眠指定的时间。然后再重新执行命令，如此反复。

函数在调用 time.sleep( ) 的时候，将释放 CPU 资源。这种方法虽然简单，但是由于缺乏灵活性，因此只能用来实现比较简单的任务。

22.6.2　使用 sched 来定时执行任务

另外一个更高级的定时执行任务方式是使用 sched 模块。

【例 22-12】演示使用 sched 模块定时执行任务。

```
#filename: sched_1.py
import time, os
import sched

schedule = sched.scheduler (time.time, time.sleep)

def execute_ command (cmd, inc):
    os.system (cmd)
```

更多资源请访问稀酷客(www.ckook.com)

```
    schedule.enter(inc, 0, execute_command, (cmd, inc))

def main(cmd, inc=60):
    schedule.enter(0, 0, execute_command, (cmd, inc))
    schedule.run()

main("netstat -an", 60)
```

- 在第 5 行中定义了一个 schedule 实例。其中的两个参数分别为 time.time 和 time.sleep。一般的定时都是使用这两个值。
- 在 execute_command 函数中，首先执行指定的命令，然后调用 schedule 的 enter 方法继续进入下一个定时周期。
- 在 main 函数中，将 execute_command 函数放入需要定时执行的队列中。并调用 run 方法来启动调度机制。

22.7　小结

本章讲解了 Python 在系统管理方面的知识，包括 Windows 和 Linux 两种操作系统。详细地介绍了 IPython 的使用方法以及高级应用，并给出了常用的 Shell 命令。然后介绍了Python对文件操作的支持。最后介绍了如何使用 Python 来自动执行任务，以便减少手工操作的工作量，提高工作效率。

更多资源请访问稀酷客(www.ckook.com)

第 23 章　Python 和网络编程

网络编程是实现各类网络应用的基础，现在的大部分网络程序都是基于 Socket（套接字）的。如何有效的构建具有丰富功能的客户端和服务器端一直是使用 Python 进行网络编程的目标。Python 标准库中提供了对网络编程的良好支持。本章的知识点：

- 开放式系统互连参考模型
- TCP/IP 网络互连模型
- Socket 基础知识
- TCP 的服务器端创建
- TCP 的客户端创建
- 异步通信方式
- Twisted 框架的使用

23.1　网络模型介绍

随着网络的迅速发展，网络上面的应用越来越丰富。在前面的章节中，已经介绍了如 HTML 网页这样的网络应用。在本节中，将抛开网络应用的实现细节，具体讨论现在存在的两种网络模型：开放式系统互连参考模型和 TCP/IP 模型。

23.1.1　OSI 简介

1983 年，国际标准化组织（International Organization for Standardization，ISO）发布了著名的 ISO/IEC 7498 标准，也就是开放式系统互连参考模型（Open System Interconnection Reference Model，OSI）。这个标准定义了网络的七层框架，试图使得计算机在整个世界范围内实现互联。在 OSI 中，网络体系结构被分成下面的七层。

（1）物理层：定义了通信设备的传输规范，规定了激活、维持和关闭通信节点之间的机械特性、电气特性和功能特性等。此层为上层协议提供了一个传输数据的物理媒介。

（2）数据链路层：定义了数据封装以及传送的方式。这个层次的数据单位称为“帧”。数据链路层包括两个重要的子层：逻辑链路控制层（Logic Link Control，LLC）和介质访问控制层（Media Access Control，MAC）。LLC 用来对节点间的通信链路进行初始化，并防止链路中断，确保系统的可靠通信。而 MAC 则用来检测包含在数据帧中的地址信息。这里的地址是链路地址或物理地址，是在设备制造的时候设置的。网络上的两种设备不能有相同的物理地址，否则会造成网络信息传送失败。

（3）网络层：定义了数据的寻址和路由方式。这一层负责对子网间的数据选择路由，并实现网络互连等功能。

（4）传输层：为数据提供端到端传输。这是比网络层更高的层次，是主机到主机的层次。传输层将对上层的数据进行分段并进行端到端传输。另外，还提供差错控制和流量控制机制。

更多资源请访问稀酷客(www.ckook.com)

（5）会话层：用来为通信的双方制定通信方式，包括建立和拆除会话。另外，此层将会在数据中插入校验点来实现数据同步。

（6）表示层：为不同的用户提供数据和信息的转换。同时还提供解压缩和加解密服务。这一层保证了两个主机的信息可以互相理解。

（7）应用层：控制着用户绝大多数对于网络应用程序的访问，提供了访问网络服务的接口。在第 17 章表 17.1 中介绍的协议都是属于应用层的。

七个层次的关系如图 23-1 所示。

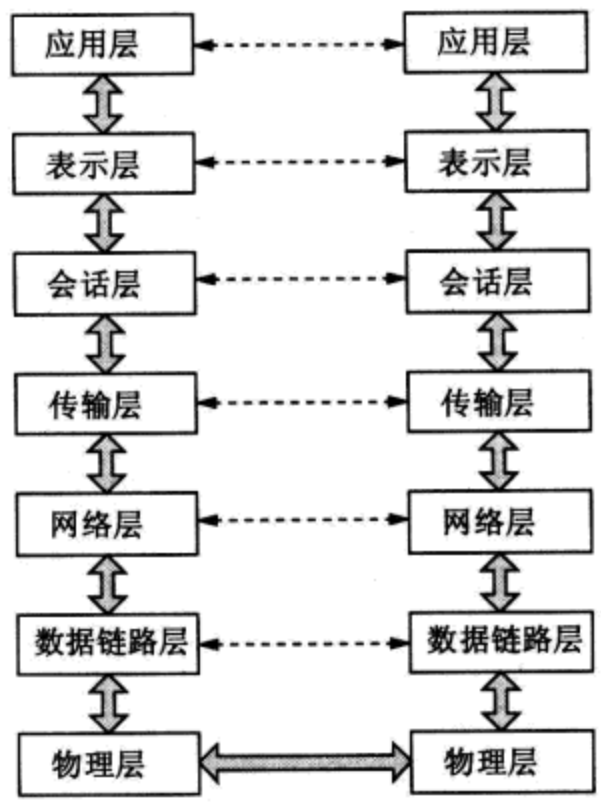


图 23-1　OSI 模型

在图 23-1 中，数据信息的实际流程是用实线的箭头标记的，层次的关系是用虚线箭头标记的。在 OSI 的七层模型中，数据访问只会在上下的两层之间进行。

OSI 模型是一个通用的网络系统模型，并不是一个协议定义，所以实际上 OSI 模型从来没有被真正的实现过。但是，由于其模型的广泛指导性，现在的网络协议都已经纳入了 OSI 模型的范围之内。在其模型中，从下至上层次依次增加，其中物理层为第一层，数据链路层为第二层，依此类推，应用层为第七层。在称呼 OSI 协议中的模型的时候，可以直接使用本来的名字，也可以直接使用数字层次。例如，前面介绍过的 HTTP 属于应用层的协议，则可以称 HTTP 协议为第七层协议。又如路由器主要是用来路由 IP 报文的，则称其为三层设备。

23.1.2　TCP/IP 简介

实际上在 OSI 模型出现之前，就已经有了 TCP/IP 的研究和实现。时间最早可以追溯到 20 世纪 70 年代，为互联网的最早的通信协议。TCP 为传输层的协议，而 IP 则为网络层的协议。两个层次中有代表性的协议组合代表了一系列的协议族，还有 ARP、ICMP 和 UDP 协议等。由于 TCP/IP 协议出现的比 OSI 早，所以并不符合 OSI 模型，其对应关系如图 23-2 所示。

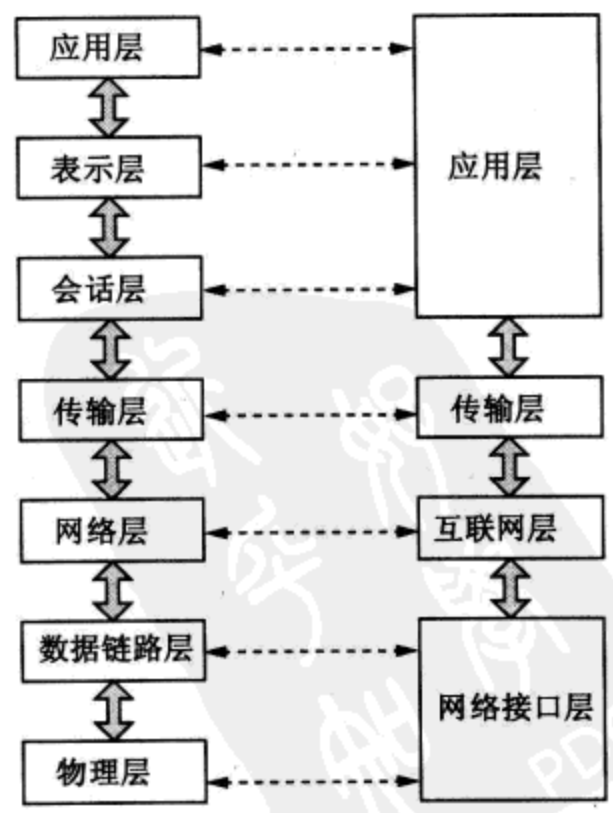


图 23-2　TCP/IP 模型和 OSI 模型的对应关系

在图 23-2 中，图的左边为 OSI 模型，右边为 TCP/IP 模型。从图中可以看到，TCP/IP 模型并不关心互联网层以下的组成，而是将数据输出统一成了网络接口层。这样，互联网层只需要将数据发往网络接口层就可以了，不需要关心下层的具体操作。在 OSI 模型中，则将这些功能分成了数据链路层和物理层，而且还进行了进一步的划分。在传输层和网络层大部分还是一致的。而对于 OSI 中的上面三层，在 TCP/IP 模型中则将其合并成了应用层。

在现在的 Internet 中，主要采用的基本上都是 TCP/IP 协议。这已经成为了互联网上通信的事实标准。现在，TCP/IP 协议已经可以运行在各种信道和底层协议之上。

在 TCP/IP 模型中，最主要的两个协议 TCP/IP 分别属于传输层和互联网层。在互联网层中，标志主机的方法是使

更多资源请访问稀酷客(www.ckook.com)

用 IP 地址，如 192. 168. 0. 1 就是一个内网主机的 IP 地址。通过对 IP 地址的类别划分，可以将整个 Internet 网络划分成不同的子网构成。而在传输层中，标志一个应用的方法是通过端口号来标志的，这些不同的端口号表示是不同的应用。例如，80 端口一般来说是 HTTP 协议，23 号端口是 Telnet 协议等。这样，在 TCP/IP 模型中，标志一个主机上的应用即可以通过地址-端口对来表示。

23. 2　Socket 应用

随着 TCP/IP 协议的流行，其中的 Socket 编程已经成为实现网络应用程序的基础。在本节中，将对 Socket 的产生和 Socket 的使用做详细地介绍。

23. 2. 1　Socket 基础知识

随着 TCP/IP 协议的使用，套接字（Socket）也越来越多的被使用在网络应用程序的构建中。实际上，Socket 编程也已经成为网络中传送和接收数据的首选方法。套接字最早是由伯克利在 BSD 中推出的一种进程间通信方案和网络互联的基本机制。到现在，已经有多种相关的套接字实现，但是大部分还是遵循着最初的设计要求。

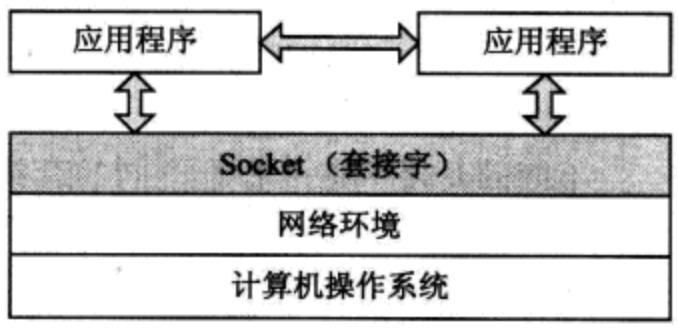


图 23-3　套接字的使用示意图

套接字相当于是应用程序访问下层网络服务的接口。使用套接字，可以使得不同的主机之间进行通信，从而实现数据交换。图 23-3 是 Socket 使用的图示。

一个正在被使用的套接字有着和其匹配的通信类型和特定的相关进程信息，并且会和另外一个套接字交换数据。当前 Socket 规范支持两种类型的套接字，即流套接字和数据报套接字。其中，流套接字提供了双向有序且不重复的数据服务。而数据报套接字虽然支持双向的数据流，但是对报文的可靠性和有序性并不保证。换句话说，通过数据报套接字接口获取的数据有可能是重复的，这需要上层的应用程序来进行区别。其实，还有更多的套接字类型，如原始套接字类型等，这和系统中的实现是相关的。

23. 2. 2　Socket 的工作方式

套接字在工作的时候将连接的对端分成服务器端和客户端，这也是 CS（Client-Server，客户端——服务器端）模式的由来。从这里也可以看出，套接字的对端通信中是不对等的。根据不同的通信方式，通信协议可以分为对称协议和非对称协议。

一般说来，服务器程序将在一个众所周知的端口上监听服务请求。换句话说，就是服务进程始终是存在的，直到有客户端的访问请求唤醒服务器进程。服务器进程然后会和客户端进程之间进行通信，交换数据。这个请求和响应的过程如图 23-4 所示。

Python 标准库中支持套接字接口的模块是 socket。该模块中有生成套接字、等待连接、建立连接和传输数据的方法。任何应用程序要使用套接字，都必须调用 socket 方法生成一个套接字对象。对于服务器端而言，首先需要调用 bind 方法绑定一个套接字接口地址，接着使用 listen 方法开始监听客户端请求。当有客户端请求过来的时候，将通过 accept 方法来生成一个连接对象。然后就可以通过此连接对象发送和接收数据了。数据传输完毕后可以调用 close 方法将生成的连接关闭。服务器端通信过程的方法调用如图 23-5 所示。

更多资源请访问稀酷客(www.ckook.com)

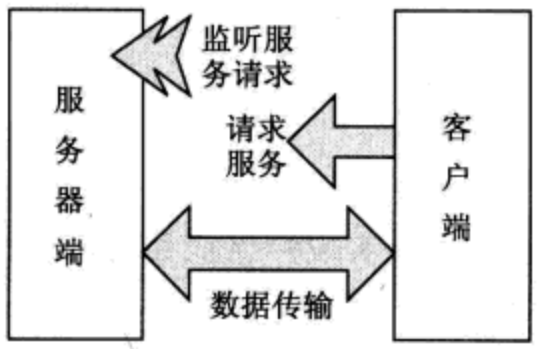


图 23-4　服务器端和客户端的通信过程

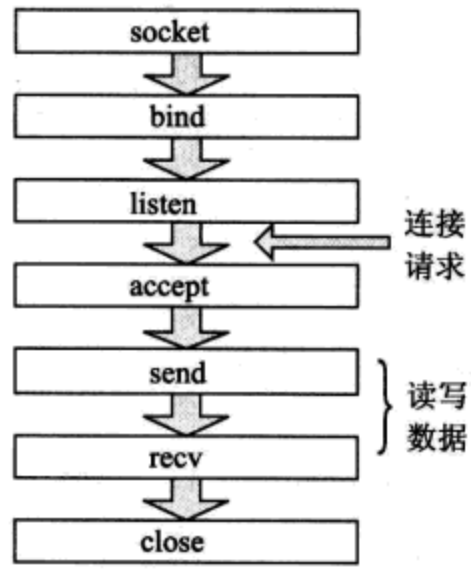


图 23-5　服务器端的通信过程

客户端应用程序在生成了套接字对象后，可以调用 bind 方法来绑定自己的请求套接字接口地址，然后调用 connect 方法来连接服务器端进程。当连接建立后，就可以使用 send 和 recv 来传输数据了。最后也是需要使用 close 方法将端口关闭。具体的通信过程如图 23-6 所示。

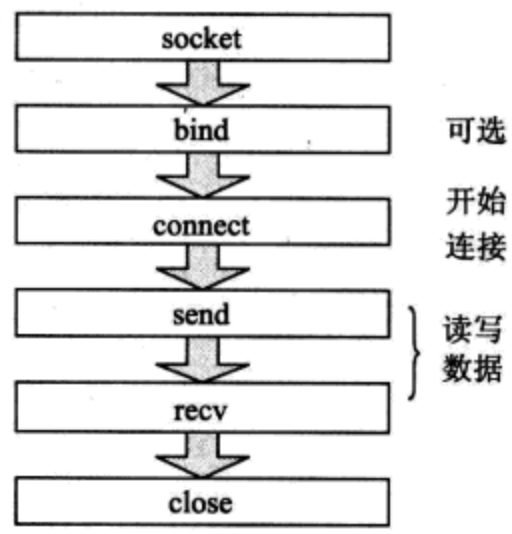


图 23-6　客户端的通信过程

不同的套接字类型有着不同的套接字地址。在 AF_UNIX 地址族中使用的是一个简单的字符串；在 AF_INET 地址族中使用的是 host 和 port 地址对，其中 host 为主机地址名、IP 地址或者是 Internet 上的 URI，而 port 则是一个整数；在 AF_INET6 地址族中，是用（host，port，flowinfo，scopeid）四元组来表示的，其中的 host 和 port 与 AF_INET 中是相同的。

23.3　服务器端和客户端通信

服务器端和客户端的通信是构建现在网络通信的基础。包括在第 17 章中介绍的 HTTP 访问，也是一种客户端和服务器端的通信，即浏览器从 HTTP 服务器上获取 HTML 数据并显示出来。在本节中，将具体介绍服务器端的构建和客户端的访问方式。

23.3.1　服务器端的构建

在 23.2.2 小节中已经介绍了服务器端通信的基本框架，可以用其中所介绍的方法来构建服务器端应用程序。在 Python 中，即是按照图 23-5 所示的过程来完成。需要注意的是，当服务器端或者是客户端关闭了生成的连接后，服务程序将会重新等待下一个连接的到来。

1. 一个简单的服务程序

【例 23-1】通过调用前面介绍的方法创建一个简单的服务器端应用程序。

```
#filename: simple_server.py
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #生成 socket 对象

host = socket.gethostname()
port = 1234
s.bind((host, port)) #绑定 socket 地址
```

更多资源请访问稀酷客(www.ckook.com)

```
s.listen(10) #开始监听

while True:
    c, addr = s.accept() #接受一个连接

    print 'Get connection from', addr
    c.send('This is a simple server') #发送数据

    c.close() #关闭连接
```

- 在代码的第 2 行，导入了 socket 模块。这是任何使用套接字接口都需要导入的模块。其中提供了套接字的大部分功能实现。
- 接着调用了模块中的 socket 方法来生成 socket 对象。该方法有 3 个参数，分别为地址族、套接字类型和协议。现在地址族支持 3 种类型，包括 AF_INET、AF_INET6 和 AF_UNIX，其中默认值为 AF_INET。套接字类型包括 SOCK_STREAM 和 SOCK_DGRAM，或者是其他的 SOCK_常量，其中的默认值为 SOCK_STREAM。协议值一般为 0。可以看出，代码片段中的 socket 调用也可以简写成下面的形式：

```
s = socket.socket()
```

- 在生成了一个套接字对象后，可以通过调用 bind 方法来绑定一个套接字地址。这里，AF_INET 的套接字地址为地址端口对。首先调用 gethostname 方法得到了当前机器的主机地址，并赋值给 host 变量。在 socket 中，还提供了许多这样类似的实用方法。同时将 port 赋值为 1234。需要注意的是，小于 1024 的端口地址在部分系统中被认为是特权端口，绑定这样的端口需要有一定的权限。这里使用了一个高于 1024 数值的端口号。在 bind 方法中，注意到它只有一个参数，这个参数是一个地址端口对。
- 接着，就可以使用 listen 方法来使得服务器进入侦听过程了。listen 方法有一个参数，用来设置连接队列的长度。

然后使用了 while 循环，其条件 True 将使得服务器端进入死循环。也就是说，服务器端将一直处于侦听状态。

- 使用 accept 方法可以接受客户端的一个连接。此对象返回一个元组，该元组有两个值。其中第一个值为生成的连接对象，可以使用此对象来发送和接收数据。而第二个值为建立 socket 连接的对端地址。
- 接着使用 send 方法来向客户端发送一个字符串数据。最后，调用 close 方法关闭此连接，从而释放此 socket 连接所占用的资源。

在一个终端中运行这个代码片段，将会阻塞并等待客户端的连接。此时打开另一个终端连接到此服务应用程序，服务器端和客户端的输出分别如下所示。

服务器端的输出：

```
In [2]: run simple_server.py
Get connection from ('127.0.0.1', 26720)
```

客户端的输出（另外一个终端）：

```
In [3]: ! telnet locahost 1234
This is a simple server
```

- 从上面的输出结果中可以看到，服务器端和客户端是结合在一起的。在客户端通过 telnet 的方法发起一个连接，服务器端接收到此连接后，将首先打印一条信息，并向客户端发送一个消息。从服务器的输出结果可以看出，连接对端为（'127.0.0.1', 26720），

更多资源请访问稀酷客(www.ckook.com)

表示此连接的客户端 IP 地址为 127.0.0.1，也就是本机地址。端口号为 26720，这是客户端自动生成的。而客户端则将是收到的消息打印了出来。

2. 通用的时间服务程序

上面的程序仅仅只是能够运行在 IPv4 地址下，而且端口地址也是指定的。在下面的代码片段中，客户端可以从 IPv4 地址或者 IPv6 地址来访问服务程序。另外，端口号也可以在代码片段启动的时候由用户指定：

```
#filename: time_server.py
import socket
import datetime
import sys

DEFAULT_PORT = 1234 #默认端口

def timeServer(port):
    host = '' #使用本机地址
    s = None
    for res in socket.getaddrinfo(host, port, socket.AF_UNSPEC, socket.SOCK_STREAM, 0,
socket.AI_PASSIVE): #在本机的所有地址监听
        af, socktype, proto, canonname, sa = res
        try:
                s = socket.socket(af, socktype, proto)
        except socket.error, msg:
            s = None
            continue
        try:
            s.bind(sa) #绑定 socket 地址
            s.listen(10) #开始监听
        except socket.error, msg:
            s.close()
            s = None
            continue
        break
    if s is None: #生成 socket 出错
        print 'could not open socket'
        return 1
    while True:
        c, addr = s.accept()
        print 'Get connection from', addr
        c.send(str(datetime.datetime.now())) #发送当前时间
        c.close()
if __name__ == '__main__':
    port = DEFAULT_PORT
    if len(sys.argv) > 1: #判断用户的输入
        try:
            port = int(sys.argv[1])
            if port<0 or port>=65536: #端口的范围判断
                port = DEFAULT_PORT
        except Exception, e:
            port = DEFAULT_PORT
    timeServer(port) #调用 timeServer 函数生成服务进程
```

- 前面定义了一个 timeServer 函数，用来实现此时间服务器功能。其中，host 为空串，在实际使用的时候会转化为本机地址。
- 接着，使用 getaddrinfo 方法来获取当前主机中的地址信息。该方法有 6 个参数，前两个为必需的主机地址和端口号。接着为可选的地址族、套接字类型、协议和标志位。这里的地址族使用了 AF_UNSPEC，使得在得到的结果中可以包括 IPv4 和 IPv6 地址。此方法将返回一个五元组的列表。其中五元组的结构为（family，socktype，proto，canon-

更多资源请访问稀酷客(www.ckook.com)

name，sockaddr），其中 sockaddr 可以用在 bind 方法中。

- 对于生成的每一个 socket 地址信息，都会使用 bind 和 listen 方法来绑定地址并监听。当遇到错误的时候，将会退出。
- 下面收到连接的过程与【例 23-1】基本类似，只是在发送数据的时候，将会发送服务器当前的时间信息。
- 在最后，将会判断用户执行脚本的输入，看是否有更多的参数。如果有更多的参数，将其作为端口号。需要注意的是，端口号必须在 0 和 65536 之间。如果不指定端口号，将会使用默认端口 1234。

这个代码片段的运行方式和【例 23-1】类似。只是在客户端的显示中将会返回连接时候服务器端的时间信息。

3. 使用 SocketServer 模块

从上面的几个例子来看，使用最原始的套接字接口生成一个服务进程还是比较复杂的。为了能够简化服务器端的编程，Python 标准库中提供了 SocketServer 模块。在此模块中有 5 个服务类，其关系如图 23-7 所示。

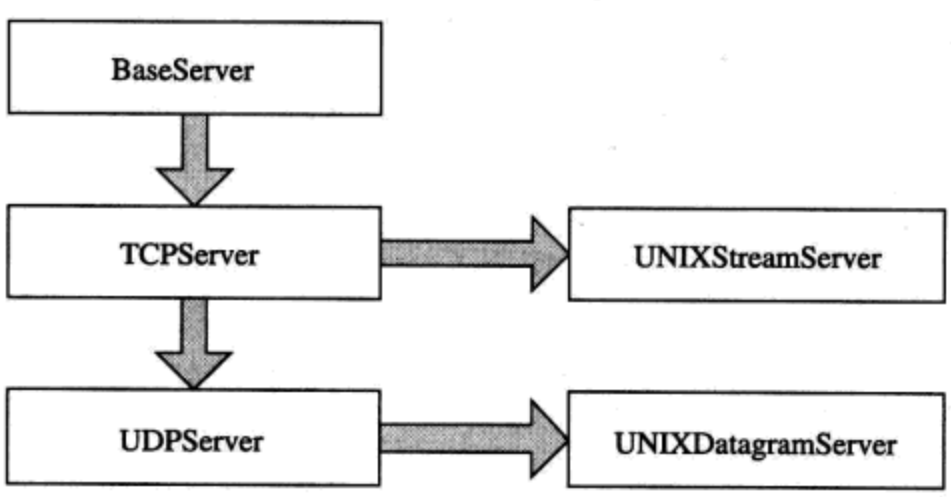


图 23-7　SocketServer 框架

一般说来，BaseServer 类不会被实际使用，这样实际上在 SocketServer 模块中就有 4 个类。其中最重要的类是 TCPServer 类。此类中有简单 TCP 协议实现的服务器端接口，为应用程序提供了可靠的流数据传输。接着是 UDPServer 类，此类中有数据报服务的接口。此类也同样提供数据传输服务，但是并不保证数据的可靠性和有序性。

另外两个使用比较少的是 UNIXStreamServer 和 UNIXDatagramServer 类。这两个类使用的都是 UNIX 域套接字地址，所以这些类不能使用在非 UNIX 平台上。实际上，IP 和 UNIX 的服务器端仅仅只是在套接字地址上不同而已。

除此之外，还有更多模块中的类是从 TCPServer 类中继承的，包括 BaseHTTPServer、SimpleHTTPServer、CGIHTTPServer、SimpleXMLRPCServer 和 DocXMLRPCServer 类等。通过对 SocketServer 模块中类的继承，可以实现更多功能的服务器端应用程序接口。

通过使用 SocketServer 模块来构建服务器端应用程序，可以将主要的实现代码放在处理器方法中。这是一些可以重载的函数，默认情况下并不会做更多的事情。当服务器端收到一个请求的时候，一个请求响应处理就被实例化，并开始具体的处理过程。具体哪个重载的方法被调用是根据所继承的类来决定的。如在 BaseRequestHandler 类中，将所有的处理都放在了 handle 方法中，在服务器被访问的时候调用。在这里，服务器可以通过 request 属性来获取连接对端的 socket 信息。而对于 TCPServer 而言，使用的是 StreamRequestHandler 类，其中有两个属性：self. rfile 属性用于读取，self. wfile 属性用于输出。可以通过文件对象的操作来和客户端通信。

更多资源请访问稀酷客(www.ckook.com)

【例 23-2】使用 SocketServer 模块实现服务器端。

```
#filename: tcp_socketServer.py
from SocketServer import TCPServer, StreamRequestHandler

class MyHandler(StreamRequestHandler):

def handle(self): #重载处理方法
        addr = self.request.getpeername() #获取连接对端地址
        print 'Get connection from', addr
        self.wfile.write('This is a tcp socket server') #发送数据
host = ''
port = 1234
server = TCPServer((host, port), MyHandler) #生成 TCP 服务器

server.serve_forever() #开始监听并处理连接
```

- 第 1 行代码从 SocketServer 模块中导入了 TCPServer 和 StreamRequestHandler。这些将在后面被用到。
- 第 4 行代码定义了一个连接处理类 MyHandler，这是从 StreamRequestHandler 类中继承的，StreamRequestHandler 类是专门用于 TCP 流服务的。
- 在 MyHandler 类中，仅仅是重载了 handler 方法，从而改变了其默认的处理方式。在此方法中有 3 条语句，其中通过 request 的 getpeername 方法可以得到连接对端的地址，并在第二句中打印出来。第三个语句中使用了 wfile 属性，这是一个文件对象，可以将文件对象的操作作用在上面。这里，使用了 write 方法，向客户端写入了一行信息。
- 接下来，通过调用 TCPServer 类的构造函数生成了一个服务程序对象。构造函数中有两个参数，其中，第一个是套接字地址，第二个为连接的处理类。可以看到，这里的套接字地址和前面例子中的是一样的。
- 最后调用此服务程序对象的 serve_forever 方法，开始监听并处理连接请求。调用这个方法可以使得其在被人工终止之前始终可以处理连接。实际上，这个方法只是简单的将此对象的 handle_ request 方法放在了一个无限循环中。

对于自定义的连接处理类，除了可以重载 handle 方法以外，还可以重载 setup 和 finish 方法。其中，前者是在 handle 方法之前做一些初始化工作，默认情况下不做任何处理。而 finish 方法则是在 handle 方法调用之后做一些连接的清理工作，默认情况下也是不做任何处理。当在 setup 或者 handle 方法中触发异常的时候，finish 方法将不会被调用。

另外，当连接到来的时候，handle 方法将会被调用。这时候，有些属性值将可以读取。例如在前面例子中用到的 request 属性，这个值是一个连接请求对象。其值在流服务和数据报服务中是不一样的。在流服务中，其值为一个 socket 对象。而在数据报服务中，其值仅仅只是一个字符串。除了 request 属性外，连接中还会提供其他的属性。例如通过 client_address 来表示客户端地址，通过 server 来表示一个服务器的实例。

对于每种具体的服务端接口，相应的类都提供了特定的方法来完成服务。通过重载某些方法，可以很快的实现不同的应用服务器程序。

23.3.2　客户端的构建

前面的示例代码在生成了服务器应用程序之后，是通过使用系统自带的应用来访问服务程序的。实际上，也可以使用 Python 来生成客户端来访问服务程序。

相对于建立服务程序而言，创建一个客户端是比较简单的。在图 23-6 中已经介绍了客户端创建的基本流程。

更多资源请访问稀酷客(www.ckook.com)

【例 23-3】使用图 23-6 中所示的流程创建一个简单的客户端程序。

```
#filename: simple_client.py
import socket

s = socket.socket() #生成一个 socket 对象

server = socket.gethostname()
port = 1234
s.connect((server, port)) #连接服务器

print s.recv(1024) #读取数据

s.close() #关闭连接
```

- 对于一个客户端程序而言，首先也是需要将 socket 模块导入。这样，后面才可以使用创建客户端程序所必需的函数。
- 接下来的部分其功能为连接服务器。首先是获取服务器的地址和服务端口。这里使用了 gethostname 方法来获取服务器地址，这是因为服务应用程序就是处于当前主机上。通过 connect 方法可以连上服务程序。
- 当客户端连上服务器后，一直等待的服务进程会被唤醒，并处理此连接。在【例 23-1】中，其处理方式是直接发送一个字符串。在这里的客户端处理中，直接调用 recv 方法获取服务器端发送过来的数据。
- 最后调用 close 方法将连接关闭。虽然这一步在这里并不是必须的，但在 socket 对象使用完毕后将其关闭是一个良好的习惯。

启动【例 23-1】的服务程序，然后使用此客户端连接服务器，可以看到和 telnet 应用有着同样的输出结果。

通过这种方式，可以用来构建一个简单的 HTTP 客户端。这个客户端并不具有浏览器的功能，只是获取指定 URL 的 HTML 文档内容。具体的代码片段如下：

```
#filename: simple_http_client.py
import socket
import urlparse
import sys

def httpget(url):
    up = urlparse.urlparse(url) #解析 URL
    host = up[1]
    page = up[2]
    s = socket.socket() #生成 socket 对象

    port = 80 #使用 80 端口号
    s.connect((host, port)) 连接服务器
    cmd = "get " + page + "\n"
    s.send(cmd) #发送 HTTP 命令

    print s.recv(1024) #获取内容

    s.close() #关闭连接

if __name__ == "__main__":
    httpget(sys.argv[1]) #调用 httpget 函数
```

- 这里的程序和【例 23-3】中的基本框架是类似的。程序的开始定义了一个 httpget 函数，其参数为要获取内容的 URL。
- 通过使用 urlparse 模块中的 urlparse 方法，得到了此 URL 的网络地址和页面地址。需要

更多资源请访问稀酷客(www.ckook.com)

注意的是，这里采用了一种简化的方法，并不支持包含查询参数的 URL。

- 在生成了 socket 对象后，使用 connect 开始连接服务器。在套接字地址中的端口号中，使用了 HTTP 常用的 80 端口。
- 接下来，使用 HTTP 的 get 命令获取特定的 HTTP 页面，并使用 recv 来获取返回的 HTML 文档内容。在 recv 方法中的参数 1024 使得这里只能最多读取 1024 个字节。可以根据需要来调节该参数的大小，以获取所有的文档内容，并做进一步处理。
- 最后，调用 close 方法关闭此连接。

运行此代码，一种输出结果如下：

```
In [2]: run simple_http_client.py http://www.google.com/intl/en/holidaylogos.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.or
    g/TR/html4/loose.dtd">
     <html>
     <head>
     <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>More Google: Holiday Logos</title>
     <linkrel="stylesheet" type="text/css"
href="http://www.google.com/css/gcs.css">

<link rel="stylesheet" type="text/css" href="css/holidaylogos.css">
</head>
<body>
#省略部分输出结果
```

- 可以看到这里输出了指定的 URL 内容。实际上还有很多的情况都没有考虑到，如错误处理等。但是，这毕竟已经实现了一种简单的获取内容的方法。其他具有此功能的库（如 urllib）也是构建在这个基础上的。

23.4　异步通信方式

在上面的示例中，所有的服务器端实现都是同步的。也就是说，服务程序只有处理完一个连接后，才能处理另外一个连接。如果要使服务器端应用程序能够同时处理多个连接，则需要使用异步通信方式。在 Python 标准库中有 3 种处理方式：Fork 方式、线程方式和异步 I/O 方式。本节将详细介绍这 3 种处理方式。

23.4.1　使用 Fork 方式

当有多个连接同时到达服务器端的时候，可以通过 fork 的方式进行处理。对于接收到的每个连接，主进程都生成一个子进程专门用来处理此连接，而主进程则依旧保持在侦听状态。这样，对于每个连接，都有一个对应的子进程来处理。由于生成的子进程和主进程是同时运行的，所以不会阻塞新的连接。这种方式的好处是处理比较简单有效。但是由于生成进程消耗的资源比较大，这种处理方式在有许多连接的时候可能会带来性能问题。

实际上，这种处理方式在 SocketServer 中已经实现了。在前面的例子中，TCPServer 类生成的实例使用的还是同步通信机制，但是在模块中也提供了 Fork 处理的类，这就是 ForkingMixin。

【例 23-4】接【例 23-2】，将【例 23-2】中的代码修改为 Fork 处理方式。

```
#filename: forking_tcp_socketServer.py
from SocketServer import TCPServer, ForkingMixIn, StreamRequestHandler
import time
class Server(ForkingMixIn, TCPServer): #自定义 Server 类
```

更多资源请访问稀酷客(www.ckook.com)

```
    pass

class MyHandler(StreamRequestHandler):

    def handle(self): #重载 handle 函数
        addr = self.request.getpeername()
        print 'Get connection from', addr #打印客户端地址
        time.sleep(5) #休眠 5 秒钟
        self.wfile.write('This is a ForkingMixIn tcp socket server') #发送信息

host = ''
port = 1234
server = Server((host, port), MyHandler)

server.serve_forever() #开始侦听并处理连接
```

- 这里和【例 23-2】主要有两个地方不同：一个是定义了 Server 类，另一个是使用此类生成了一个对象。
- 在前面 Server 类的定义中，可以看到 Server 类是从 ForkingMixIn 和 TCPServer 类中继承的。这使此类具有了这两个类的特点。也就是说，既提供流数据传输服务，又提供多连接处理功能。其中 ForkingMixIn 是需要第一个继承的类。同样地，也可以结合 ForkingMixIn 和 UDPServer 生成一个可以同时处理多个连接的数据报服务器端应用程序。
- 在最后生成 Server 类对象的地方，将原来版本中的 TCPServer 换成了 Server 类，也就是这里自定义的服务器类。这样新生成的对象将具有处理多连接的功能。
- 除了上面的两点差别之外，还有一点是在 MyHandler 类中的 handle 方法中加了一个休眠过程。这主要也是为了在演示的时候可以看到异步通信功能。

运行上面的服务器端应用程序，并打开多个不同的命令行窗口，然后同时连接此服务器，将可以看到在等待 5 秒钟后，都会显示服务器端写入的字符串。可见，这种处理方式很好的实现了异步通信。只是在有大量连接的时候，消耗的资源会比较大。

注意　这种方式仅适用于 Linux 系统。在 Windows 下由于没有 fork 方法，将会报错。具体的出错信息如下：

```
In [2]: run forking_tcp_socketServer.py
----------------------------------------------------------------
Exception happened during processing of request from ('127.0.0.1', 31493)
Traceback (most recent call last):
  File "E:\Python25\lib\SocketServer.py", line 222, in handle_request
    self.process_request(request, client_address)
  File "E:\Python25\lib\SocketServer.py", line 429, in process_request
    pid = os.fork()
AttributeError: 'module' object has no attribute 'fork'
----------------------------------------------------------------
```

- 从最后一条信息可以看出，这里的 os 模块中没有 fork 方法。

23.4.2　使用线程方式

由于线程是一种轻量级的进程，具有进程所没有的优势，所以在上面的 Fork 处理方式中，当有大量连接而消耗资源太多的时候，则可以使用线程的方式来进行处理。线程实现的方式和 Fork 的处理方式是类似的。当有连接到来的时候，主线程将生成一个子线程来处理连接，从而在子线程处理连接的时候，主线程还是保持在侦听状态，而不会阻塞连接。

由于生成的子线程和主线程之间具有同样的地址空间，所以这种处理方式的效率是比较高的。

更多资源请访问稀酷客(www.ckook.com)

但是，这种方式最大的一个问题是大量的使用线程会带来线程之间的数据同步问题。如果处理得不好，则有可能会使得服务程序失去响应。在对资源不是很敏感的时候，还是推荐使用 Fork 的方式来处理。当然，也可以采取其他方式来实现。如 Stackless Python，这是一个 Python 的增强版本，能够有效的利用线程而避免传统线程中的性能和复杂度问题，其网址为 http：//stackless. com。

在 SocketServer 类框架中也提供了使用线程处理多连接的接口——ThreadingMixIn。

【例 23-5】接上例，将【例 23-4】的代码修改为线程处理方式。

```
#filename:threading_tcp_socketServer.py
from SocketServer import TCPServer, ThreadingMixIn, StreamRequestHandler
import time

class Server(ForkingMixIn, TCPServer): #自定义 Server 类
    pass

class MyHandler(StreamRequestHandler):

    def handle(self): #重载 handle 函数
        addr = self.request.getpeername()
        print 'Get connection from', addr #打印客户端地址
        time.sleep(5) #休眠 5 秒钟
        self.wfile.write('This is a ThreadingMixIn tcp socket server')

host = ''
port = 1234
server = Server((host, port), MyHandler)

server.serve_forever()#开始侦听并处理连接
```

- 这里的大部分示例代码和【例 23-4】中是相同的。其中，唯一的不同在于生成 Server 类的时候采用的是 ThreadingMixIn 类。这样生成的 Server 类实例在处理多连接的时候将采用线程的方式来处理。

23.4.3　异步 IO 方式

当同时会有多个连接的时候，可以采用 Fork 和线程的方式，但是对那种持续时间长且数据突发的多连接，这些处理方式所占用的资源太多。一种改进的方式是采用专门的异步 IO 通信方式，即在一定的时间段内查看已有的连接并处理。处理的过程包括读取数据和发送数据。

在 Python 标准库中，有 asyncore 和 asynchat 模块用来实现这种处理方式。这些框架的实现方式依赖于 select 和 poll 方法，这两个方法都定义在在 select 模块中。虽然 poll 方法更具有可扩展性，但是这种方法不能用在 Windows 系统下。

1. select 方法

select 的使用方法是对指定的文件描述符进行监视，并在文件描述符集改变的时候做出响应。在 Python 标准库中，具体的实现是 select 模块中的 select 方法。这实际上也是 select 系统调用的一个接口。

select 函数有 3 个必须的参数和 1 个可选的时间参数。其中前 3 个参数都为文件描述符列表，分别用来表示等待输入、输出和错误的文件描述符。空的列表也是允许的，但是如果 3 个参数都是空的列表，则是平台相关的——在 Linux 系统平台下是被允许的，但在 Windows 系统平台下这是不会被接受的。可选的时间参数为一个浮点数，用来指定系统监视文件描述符集改变的超时时间。当此参数被忽略的时候，函数将会阻塞到至少有一个文件描述符准备好的情况下才返回。而当设置超时时间为 0 的时候，则表示调用的时候从不阻塞。

更多资源请访问稀酷客(www.ckook.com)

在文件描述符集合中可以被接受的对象类型包括 Python 中的文件描述符，如 sys. stdin，或者是通过 open 和 popen 方法得到的对象。还包括通过 socket. socket（）方法返回的 socket 对象。这些可接受对象的一个共同特征是都可以通过 fileno（）方法来获得具体的描述符，一般使用整数来表示。在 Windows 系统平台下，文件描述符是不被接受的。这是因为 Windows 中实现 TCP/IP 协议的 WinSock 不能处理文件描述符。

此方法的返回值是一个有 3 个值的元组，元组中的 3 个值即为在 select 方法中的前 3 个参数中已经准备好的文件描述符。当等待时间超时且没有任何已经准备好的文件描述符的时候，则返回由 3 个空的列表组成的元组。

【例 23-6】演示 select 的使用方法，使其在收到客户端发送的报文后将收到的信息打印出来。

```
#filename: select_ex.py

import socket, select

s = socket.socket() #生成 socket 对象

host = socket.gethostname()
port = 1234
s.bind((host, port)) #绑定套接字接口地址

s.listen(5) #开始服务器端监听

inputs = [s]
while True:
    rs, ws, es = select.select(inputs, [], []) #使用 select 方法
    for r in rs:
        if r is s:
            c, addr = s.accept() #处理连接
            print 'Get connection from', addr
            inputs.append(c)
        else:
            try:
                data = r.recv(1024) #接收数据
                disconnected = not data
            except socket.error:
                disconnected = True

            if disconnected:
                print r.getpeername(), 'disconnected'
                inputs.remove(r)
            else:
                print data #打印接收到的数据
```

- 在代码片段的最前面导入了 socket 和 select 模块。select 方法包含在 select 模块中。
- 接下来的 3 个代码段和前面的示例代码中是一样的。包括生成 socket 对象、绑定套接字接口地址和开始服务器端的监听。
- 设置了一个 inputs 列表变量，用来记录需要处理输入的 socket 对象。
- 在 while 循环中，首先调用了 select 方法。select 方法中的 3 个参数分别为 inputs 和两个空列表。由于没有超时时间参数，所以这里的调用将会一直阻塞到前 3 个集合中有至少一个文件描述符准备好。并将返回值分别赋值给了 3 个变量，分别表示准备好的输入、输出和错误的文件描述符。可以看到，这里的 select 现在仅仅只是用于套接字的连接。实际上，如果加入对 sys. stdin 的监听，还可以实现对命令行输入数据的处理。
- 本例的功能仅仅只是将收到的数据打印出来，所以这里只是考察了用于输入的文件描述

更多资源请访问稀酷客(www.ckook.com)

符。对于已经准备好输入的每个文件描述符，这里也仅只是考察监听的输入情况。当确定好之后，代码调用了 accept 方法来获得 socket 对象和连接对端地址。然后将此生成的 socket 对象也放在 inputs 列表中。

- 对于非监听端口的准备输入的文件描述符，则通过调用 recv 方法来接收数据。当数据传输结束也就是没有数据的时候或者是发生 socket 错误的时候，设置连接断开标志。当连接断开的时候，打印相关的信息并将其连接对象从 inputs 列表中删除。
- 最后打印出收到的数据信息。

运行【例 23-6】，同时开启多个终端连接此服务器端，可以看到此代码很好的处理了多连接情况。同时可以看到，当有多个连接的时候，在服务器端的输出结果将会错乱。

2. poll 方法

除了 Windows 平台外，poll 方法在其他系统下应用很广泛。这是一种比 select 方法更容易扩展和更好的方法。在需要同时为很多连接服务的时候比较有用。其中的一个原因是，select 方法采用的是一种位图索引（Bitmap Index）的方式来处理文件描述符，而 poll 方法则仅仅只需要处理感兴趣的文件描述符。最终的结果就是 select 方法和最大的文件描述符是一致的，而 poll 方法则和文件描述符的个数是一致的。poll 方法的这种特点可以有效的降低服务器的处理负担。

poll 方法同样是在 select 模块中实现的。当调用 poll 方法的时候，将得到一个 Polling 类对象。注意，poll 方法没有任何输入参数。在生成的 Polling 类对象中，有 register、unregister 和 poll 等 3 个方法。

在调用 Polling 对象的 poll 方法之前，需要使用 register 方法将待监视的文件描述符进行注册。注册的文件描述符可以是对象使用 fileno 方法得到的值。多次注册一个文件描述符也不会发生错误，这和注册一次的效果是一样的。同时，在后面可以调用 unregister 方法将注册的文件描述符取消。当注册了某些对象（如 socket 对象）后，就可以调用其对象的 poll 方法了。此方法有一个可选的超时参数，用来指定获取对象返回之前的等待时间。如果此值被省略、为负数或 0，则调用此方法将阻塞到至少有一个事件到达。

调用此方法后，将返回（fd，event）对的列表。其中，fd 为文件描述符，event 用来指示发生的事件。event 是一个位掩码，通过一个整数的位来对应特定的事件信息。表 23-1 是定义在 select 模块中的事件信息。

表 23-1　用于 poll 方法的事件信息

事件信息	说　明
POLLIN	含有可读的数据
POLLPRI	含有紧急可读的数据
POLLOUT	含有需要写出的数据
POLLERR	发生了错误
POLLHUP	连接断开
POLLVAL	错误的请求

如果需要知道特定的事件是否发生，可以使用“&”操作符，如下所示：

```
if event & select.POLLIN:
    #处理 POLLIN 事件
```

更多资源请访问稀酷客(www.ckook.com)

【例 23-7】接上例，重写【例 23-6】中的代码，演示 poll 具体的使用方法。

```
#filename: poll_ex.py

import socket, select #生成 socket 对象

s = socket.socket() #

host = socket.gethostname()
port = 1234
s.bind((host, port)) #绑定套接字接口地址

fd_dict = {s.fileno(): s}

s.listen(5) #开始服务器端监听

p = select.poll() #生成 Polling 对象
p.register(s) #注册 socket 对象

while True:
events = p.poll() #获取准备好的文件对象

    for fd, event in events:
        if fd in fd_dict:
            c, addr = s.accept() #处理连接
            print 'Got connection from', addr
            p.register(c)
            fd_dict[c.fileno()] = c #加入连接 socket

        elif event & select.POLLIN:
            data = fd_dict[fd].recv(1024) #接收时间
            if not data:
                print fd_dict[fd].getpeername(), 'disconnected'
                p.unregister(fd) #取消注册
                del fd_dict[fd]
        else:
            print data #打印数据
```

- 代码片段的前面三个部分和【例 23-6】中是一致的。
- 定义了一个字典变量 fd_dict，用来保存需要监视的对象。
- 在接下来的代码段中，使用 select 模块中的 poll 方法生成了一个 Polling 对象，并调用 register 方法注册了前面的 socket 对象。
- 在 while 循环中，包含了对多连接的处理。在 while 循环中首先调用了 Polling 对象的 poll 方法。这里并没有加上任何超时参数，也就是说当代码运行到这里的时候，将阻塞直到有一个事件发生。
- 判断返回的 fd 是否在 fd_dict 字典中，如果存在，则表示此为监听连接的 socket。调用 accept 方法来获得客户端连接和客户端地址。将客户端地址输出，并注册客户端连接和将其加入 fd_dict 字典变量中。
- 接下来的代码段中，通过 event 和 POLLIN 标志位来判断是否为需要接收数据的事件。如果是需要接收数据的事件，则开始接收数据。当连接断开的时候，则调用 unregister 方法取消此连接的注册信息，并将其从字典中删除。
- 最后是打印接收到的数据。

按照【例 23-6】同样的操作来运行，可以看到最终的效果是一样的。

23.4.4　使用 asyncore 模块

使用 asyncore 模块同样可以实现异步通信方式。实际上，模块中提供了用来构建异步通信方

更多资源请访问稀酷客(www.ckook.com)

式的客户端和服务器端的基础架构。特别适用于聊天类的服务器端和协议的实现。其基本思想是创建一个或者多个网络信道，而实际上网络信道是 socket 对象的一个封装。当信道创建后，通过调用 loop 方法来激活网络信道的服务，直到最后一个网络信道关闭。

在 asyncore 模块中，主要用于网络事件循环检测的 loop 方法是其核心。在 loop 方法中将会通过 select 方法来检测特定的网络信道。当 select 方法返回有事件的 socket 对象后，loop 方法检查此事件和套接字状态并创建一个高层次的事件信息。然后针对此高层次的事件信息调用相应的方法。asyncore 提供了底层的 API 用来创建服务器。

同时，在模块中还有一个 dispatcher 类，这是一个对 socket 对象的轻量级封装，用于处理网络交互事件。其中的方法是在异步 loop 方法中调用的，或者可以直接当成一个普通的非阻塞 socket 对象。其框架如图 23-8 所示。

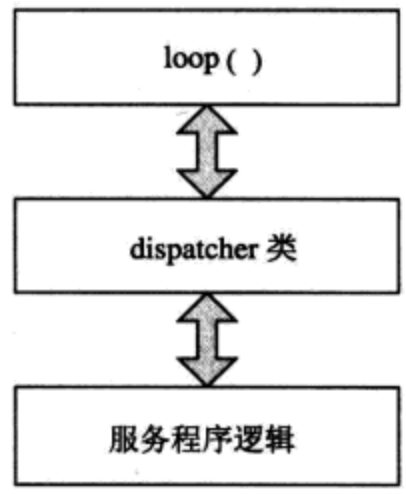


图 23-8　asyncore 模块框架

在 dispatcher 类中，当在特定的时间或连接状态的条件下，会触发一些高层次事件。在继承类中可以通过重载这些方法来处理特定的事件。表 23-2 中给出了默认的事件。

表 23-2　dispatcher 类中定义的方法

方 法 名	说　明
handle_connect	连接时候的访问接口
handle_close	没有数据时的接口
handle_accept	从监听端口上获取数据

在进行异步处理的过程中，通过网络信道的 readable 和 writable 方法来对事件进行控制。通过这些方法能够判断是否需要使用 select 或者 poll 方法来读取事件。在 asyncore 模块中，readable 和 writable 方法默认是不做任何判断，而直接返回“True”。可以通过重载这些方法来判断需要检查的连接，从而控制流程以及网络状态。接着，就可以使用 handle_read 和 handle_write 方法来读写网络数据，完成对数据的接收和发送。

【例 23-8】下面是 Python 帮助手册中的代码，演示了 asyncore 模块的使用方法。

```
#filename: asyncore_ex.py

import asyncore, socket

class HttpClient (asyncore.dispatcher): #定义了一个 HttpClient 类

    def __init__(self, host, path): #类的构造函数
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM) #创建 socket 对象
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path
    def handle_connect(self): #连接调用接口
        pass
    def handle_close(self): #接口关闭函数
        self.close()
    def handle_read(self): #读取数据
        print self.recv(1024)
    def handle_write(self): #写入数据
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]
    def writable(self): #判断是否写入数据
```

更多资源请访问稀酷客(www.ckook.com)

```
        return (len(self.buffer) > 0)
if __name__ == '__main__':
c = HttpClient('www.python.org', '/')

asyncore.loop() #开始异步通信处理方式
```

- 在代码的最上面导入了 asyncore 和 socket 模块，为本代码片段的实现提供了基础。
- 接下来，定义了一个 HttpClient 类，这是从 dispatcher 类中继承的。通过适当的重载 dispatcher 中的处理函数，HttpClient 对象将能够有效地处理网络事件。
- 在接下里的代码段中定义了 HttpClient 类的构造函数。其构造函数首先调用父类 dispatcher 的构造函数。接着调用 create_ socket 函数来创建 socket 对象。该函数封装了 socket 模块中的 socket 方法。在调用了 socket 方法之后，还使用 setblocking 设置其阻塞方式为非阻塞，并获取了套接字的文件描述符。最后通过调用 add_channel 方法将其文件描述符加入。具体的实现如下所示：

```
#asyncore.py 中 dispatcher 类的 create_socket 函数实现
def create_socket(self, family, type):
    self.family_and_type = family, type
    self.socket = socket.socket(family, type)
    self.socket.setblocking(0)
    self._fileno = self.socket.fileno()
    self.add_channel()
```

- 在构造函数中，使用 connect 方法连接特定服务器的 80 端口，这也是 HTTP 协议的默认端口。并设置类变量 buffer 为一个 HTTP 获取命令，此处构造的报文将在后面合适的时候被发送，用来获取 HTML 内容。
- 接下来定义了 4 个事件处理函数，分别在不同的事件发生的时候被调用。
- handle_connect 方法将在 HTTP 连接的时候被调用。这里的处理方法是直接被 pass，也就是说不做任何操作处理。
- 在 handle_close 方法中，直接对 socket 对象调用 close 方法关闭连接。此方法将在 HTTP 关闭的时候被调用。
- 在 handle_read 方法中，调用了 recv 方法来获取 HTTP 数据。此方法将会在获取数据的时候被调用。另外，recv 方法中的参数为一次最大读取的字节数。需要注意的是，缓冲区的大小最好选择为 2 的乘幂，如 1024 或者 4096 等。
- handle_write 函数用来处理发送下的情况。这里首先调用 send 方法发送数据，其返回值为已经发送成功的数据。然后设置 buffer 为未发送的数据。之所以要这样处理，是因为在异步通信过程中，不一定能够保证每次发送都全部发送成功。
- writable 函数用来判断在什么时候发送数据。在函数主体中只是判断了需要发送数据的缓冲区是否为空。如果不为空，则返回“True”，表示需要发送数据。而当缓冲区为空的时候，则不需要继续发送数据。
- 在代码的最后面，使用 HttpClient 类生成一个实例，最后调用 asyncore 模块的 loop 方法。当运行的时候，将会获取特定的 URL 数据。
- 在上面的代码中，可以看到其中并没有对 readable 函数的重载，所以 asyncore 模块将会去轮询看是否需要接收数据。

运行【例 23-8】，可以得到下面的结果：

```
In [2]: run asyncore_ex.py
HTTP/1.1 200 OK
Date: Fri, 28 Nov 2008 05:32:43 GMT
Server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_ssl/2.2.3 OpenSSL/0.9.8c mod_w
```

更多资源请访问稀酷客(www.ckook.com)

```
sgi/2.3 Python/2.4.4
Last-Modified: Thu, 27 Nov 2008 15:22:08 GMT
ETag: "105800d-4146-4ecd0800"
Accept-Ranges: bytes
Content-Length: 16710
Connection: close
Content-Type: text/html; charset=UTF-8

<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.or
g/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Python Programming Language -- Official Website</title>
  <meta name="keywords" content="python programming language object oriented web
 free source" />
#省略部分输出结果
```

- 从这里的输出结果来看，并不仅仅只是输出了 HTML 文档的内容，而且输出了 HTTP 响应的内容。在输出结果的第 1 行中，表明此次连接的获取是正常的。后面给出了此文档的相关元信息。例如最后一次修改的时间、文档长度和文档类型等。接着 HTTP 响应后给出的才是具体的 HTML 文档内容。

23.5　Twisted 网络框架

asyncore 为编写异步通信的程序提供了一个框架，但是在使用时还是比较麻烦的。如果需要更好的解决这个问题，可以使用 Twisted 框架。

Twisted 是一个面向对象基于事件驱动的顶级通信框架，可以完成大部分的网络应用任务。同时，Twisted 框架具有很好的网络性能，提供了异步通信机制。在本节中，将在介绍 Twisted 网络框架的基础上，对如何使用它来构建网络服务器端进行介绍。

23.5.1　认识 Twisted 框架

Twisted 是使用 Python 语言实现的强大的网络框架，已经应用于多个领域。Twisted 是 ZOPE 中 HTTP 服务器的实现部分，可以和大名鼎鼎的 ACE（Adaptive Communication Environment，自适应网络通信环境）网络框架媲美。它特别适合于用来编写服务器端的应用程序，对于其中的很多细节，Twisted 都有比较完美的实现。

Twisted 框架的下载网址是 http：//www.twistedmatrix.com/，当前的最新版本为 8.1.0。在 Twisted 框架中，已经提供了许多可重用的协议和接口。这些协议包括 SSH2、FTP、POP3 和 SMTP 等，甚至还有对 MSN 等即时通信协议的支持。安装完毕后，在 site-packages 目录下将会生成一个 twisted 的目录。在其 protocols 目录下有这些协议的实现。

在使用 Fork 和线程方式来进行异步通信传输的时候，实质上还是采用了一种轮询的方式来处理连接。而对于 Twisted 框架（包括前面的 asyncore 模块）来说，采用的是一种事件驱动的方式。也就是说，只需要在事件发生的点构建相应的代码就可以了。例如构建一个服务器端应用程序，只需要在部分事件的接口上书写代码即可，包括新连接到来的时候、新数据到来的时候和连接关闭的时候等。Twisted 框架做了进一步封装。例如不仅仅只是针对新数据到来时候的处理，而是将这样的事件分解成更基本的事件，包括换行之前数据的到来事件等。

Twisted 框架是由模块化的组件组成的。这些模块化元素包括协议、工厂、反应器（reactor）

更多资源请访问稀酷客(www.ckook.com)

和 Deferred 对象等。工厂用来产生一个新的实例，一种实例可以产生一个类型的协议。这些协议定义了如何和服务器交换数据。在运行的时候，每次连接都会产生一个协议实例。其中，用来管理事件信息循环的反应器是整个 Twisted 应用服务器的核心。另外，Deferred 对象则是一种处理延时调度器处理的方法。使用这种方法，可以使得不必等待运行结果而返回。

23.5.2　Twisted 框架下服务器端的实现

【例 23-9】重新实现【例 23-4】和【例 23-5】文件中的功能。

```
#filename: twisted_server_1.py
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory

class SimpleServer(Protocol):

    def connectionMade(self): #连接建立的时候
        print 'Get connection from', self.transport.client

    def connectionLost(self, reason):连接断开的时候
        print self.transport.client, 'disconnected'

    def dataReceived(self, data):#接收数据的时候
        print data

factory = Factory()
factory.protocol = SimpleServer

port = 1234
reactor.listenTCP(port, factory)
reactor.run() #进入循环
```

- 【例 23-9】给出了一个简单的服务器实现，其功能为将收到的数据打印出来，同时在连接开始和连接断开的时候给出信息提示。这里采用了非常典型的事件驱动编程方式，只是重载了特定事件的代码实现。
- 在代码的最上面从 twisted.internet 中导入了 reactor，并从 protocol 中导入了 Protocol 和 Factory 类。这些基本概念在前面都已经做过简单的介绍。其中，reactor 是整个 Twisted 应用的核心，后两个类为协议实现接口。
- 接着定义了一个 SimpleServer 类，此类是从 Protocol 类中继承的。此类中实现了通信协议的基本框架，并定义了相关的通信接口。这些接口将在通信时的特定事件中被触发。例如连接建立的时候或者是连接断开的时候。
- 当连接建立的时候，也就是通信协议中最开始协商的时候，将会调用 connectionMade 方法。这里的实现仅仅只是输出了连接信息。其中 self.transport 中保存了当前的连接对象，可以通过其中的 client 属性来获取连接的客户端。
- 当连接断开的时候，将会调用 connectionLost 方法。这里的实现也是打印相关的信息，表明此连接已经断开。
- 当收到数据的时候，将会调用 dataReceived 方法。此方法这里的实现是将收到的数据完全照原样输出来。
- 在定义了 SimpleServer 类后，调用了 Factory 类的构造函数并构建了一个工厂方法实例。其后设置了其协议为前面已经定义好的 SimpleServer 类对象。
- 最后的代码段中，首先调用了反应器的 listenTCP 方法，其中设置了监听的端口号和上面生成的工厂对象。最后使用反应器的 run 方法进入循环。在此期间程序将会监听 1234

更多资源请访问稀酷客(www.ckook.com)

端口，当收到连接请求的时候，将会调用 SimpleServer 类中定义的处理函数来处理特定的事件，如连接建立的时候调用 connectionMade 方法等。

运行【例 23-9】的代码，将得到类似如【例 23-4】的输出结果。区别在于服务器端将输出在客户端输入的任何数据，包括换行符。而且这里的连接是由客户端来控制的。只有当客户端断开连接的时候，服务器端才会调用 connectionLost 方法。另外，这里对客户端的数目没有限制，只要服务器端的资源允许，客户端就可以连接服务器。

23.5.3　Twisted 框架下服务器端的其他处理

【例 23-9】给出了一个简单的 Twisted 服务器端应用程序。实际上，Twisted 框架的扩展能力是很强的，下面将具体介绍一些对【例 23-9】演示的服务器端的改进方法。

在【例 23-9】的输出结果中，由于服务器端缓冲区设置的原因，可能会将收到的数据一个字符一个字符的输出。实际上对客户端来说，合理的处理方式是在收到一个换行符后进行处理。因为在实际操作中这种需求很多，所以在 twisted. protocols. basic 中预设置了一些处理方式，其中有 LineReceiver 用来实现上面的需求。此接口实现了 dataReceived 接口且在收到换行符的时候被调用。

【例 23-10】接上例，演示 LineReceiver 接口的使用方法。

```
#filename: twisted_server_2.py
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory
from twisted.protocols.basic import LineReceiver

class SimpleServer(Protocol):

    def connectionMade(self): #连接建立的时候
        print 'Get connection from', self.transport.client

    def connectionLost(self, reason):连接断开的时候
        print self.transport.client, 'disconnected'

    def lineReceived(self, line): #当收到一行数据的时候
        print line

factory = Factory()
factory.protocol = SimpleServer

port = 1234
reactor.listenTCP(port, factory)
reactor.run() #进入循环
```

- 【例 23-10】代码和【例 23-9】的区别主要在 SimpleServer 类中定义了 lineReceived 方法。此方法是在服务器端收到一行数据的时候被调用。

运行【例 23-10】可以发现，在客户端输入数据的时候，服务器端并不会将数据输出。而只是在收到换行符时，才调用 lineReceived 方法，并将接收到的数据输出。

【例 23-11】在前面的实现中，连接的断开是由客户端来控制的，服务器端并没有控制权。下面实现服务器端的控制功能。具体的实现是在收到连接后打印信息并关闭连接。

```
#filename: twisted_server_3.py
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory

class SimpleServer(Protocol):

    def connectionMade(self): #连接建立的时候
        print 'Get connection from', self.transport.client
```

更多资源请访问稀酷客(www.ckook.com)

```
        self.transport.loseConnection()

factory = Factory()
factory.protocol = SimpleServer

port = 1234
reactor.listenTCP(port, factory)
reactor.run() #进入循环
```

- 在这段代码的 SimpleServer 类中，仅仅只是定义了一个 connectionMade 接口方法。其实现是在打印客户端的相关信息之后，直接调用了其 transport 对象的 loseConnection 方法。这样将由服务器端来中断连接。

运行【例 23-11】的时候，客户端将在连接后直接断开，而只是在服务器端打印出所连接的客户端的相关信息。

在前面的服务实现中，只是由服务器端来获取客户端数据，而没有将服务器端的数据发送给客户端。其实在 Twisted 框架中也已经有这样的处理方法，请看【例 23-12】。

【例 23-12】实现一个 ECHO 服务器，将接收到的所有数据都原样发送给客户端。

```
#filename: twisted_server_4.py
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory
from twisted.protocols.basic import LineReceiver

class EchoServer(Protocol):

    def connectionMade(self): #连接建立的时候
        print 'Get connection from', self.transport.client

    def lineReceived(self, line): #将收到的数据返回给客户端
        self.transport.write(line)
factory = Factory()
factory.protocol = EchoServer

port = 1234
reactor.listenTCP(port, factory)
reactor.run() #进入循环
```

- 【例 23-12】代码中定义了一个 EchoServer 类。和【例 23-10】中定义的 SimpleServer 的主要区别在于这里定义的 dataReceived 的实现部分。其函数主体中的程序只有一句，主要是调用了连接对象 transport 的 write 方法，并将收到的数据发送到客户端。

将【例 23-12】代码运行在服务器上，并使用客户端连接此应用服务。当在客户端上输入一行数据的时候，服务器端会将收到的数据发送回客户端。这样从客户端看来，是显示了当前输入的数据，从而实现了客户端的回显。

在上面介绍的代码片段中，并没有过多的业务逻辑。当上面的服务器应用程序运行的时候，只要服务器的资源允许，都可以接受客户端连接。在特定的时候，由于服务器资源所限，只能够接受一定的客户端连接。下面的代码实现了这种功能：

```
#filename: twisted_server_5.py
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, Factory
from twisted.protocols.basic import LineReceiver

class EchoServer(Protocol):

def connectionMade(self): #连接建立的时候
```

更多资源请访问稀酷客(www.ckook.com)

```
        print 'Get connection from', self.transport.client
        self.factory.numProtocols = self.factory.numProtocols +1
        if self.factory.numProtocols > 5: #当连接超过5个的时候,断开连接
            self.transport.write("Too many connections, try later")
            self.transport.loseConnection()
        print 'Get connection from', self.transport.client

    def connectionLost(self, reason): #断开连接
        self.factory.numProtocols = self.factory.numProtocols-1

    def lineReceived(self, line): #将收到的数据返回给客户端
        self.transport.write(line)

factory = Factory()
factory.protocol = EchoServer

port = 1234
reactor.listenTCP(port, factory)
reactor.run() #进入循环
```

- 这里定义的 EchoServer 类和在【例 23-12】文件中定义的类是不同的。差别主要在 connectionMade 和 connectionLost 两个接口方法。
- 在 connectionMade 方法中，实现了对客户端数目的限制。在 factory 实例中的 numProtocols 属性保存了现在已有的连接数目，在每次连接的时候将其加“1”。然后判断此值和设置值的大小，如果超过了所设置的值，则向客户端发送一个“客户端太多，建议重试”的消息，并调用 transport 对象的 loseConnection 方法中断此连接。
- 在连接断开的时候，将会调用 connectionLost 方法接口。具体的实现中，将 numProtocols 属性值减“1”，从而表示现在连接的数目少了一个。

运行这段代码，当超过 5 个客户端连接的时候，将会被中断。只有在有的客户端退出连接后，才能够重新接入。

上面的几个代码段中介绍了如何将一个简单的服务器端改变成功能丰富的服务器端的办法。上面也只是介绍了部分功能的实现。实际上，在 Twisted 框架的基础上，可以非常方便的实现很多复杂的网络功能。具体的可以参见 Twisted 相关文档。

23.6　小结

本章讲解了 Python 语言在网络编程上的使用。首先介绍了现在分析比较多的网络模型，包括开放式系统互连参考模型和 TCP/IP 网络互连模型。接着对 socket 进行了介绍，并在此基础上实现了服务器端和客户端，包括使用 SocketServer 模块等。接下来对异步通信方式进行了介绍，包括 Fork 方式、线程方式、异步 I/O 方式、asyncore 模块。最后着重介绍了 Twisted 框架及其使用方法。

更多资源请访问稀酷客(www.ckook.com)

第 24 章　常见的 Python 网络应用

互联网中存在着许多类型的网络应用。在本章中，将具体介绍传输文件、收发邮件、远程登录和网络管理等方面的功能。另外，还对在网络中的抓包分析进行介绍。本章的知识点：

- ftplib 模块的使用方法
- 使用 pop3lib 和 smtplib 模块收发邮件
- 使用 Telnet 协议远程登录
- 使用 SNMP 管理网络
- 在网络中抓包分析

24.1　使用 FTP 传输文件

文件传送协议（File Transfer Protocol，FTP），是一个用于将数据文件从一台主机传送到另外一台主机上的传输协议。这是 Internet 上最早的应用协议之一。

24.1.1　FTP 的工作原理和 Python 库的支持

FTP 协议的工作原理是将 FTP 的客户端连接到 FTP 的服务器端，并给出用户名和密码进行认证。这一步结束后客户端可以浏览服务器端上的文件，执行一些交互操作。这个通道为命令通道，表示此连接主要是用来传输 FTP 命令的。当客户端需要从服务器端下载文件的时候，FTP 客户端将会再和远程主机中的 FTP 服务器端建立一个连接。在服务器端认证了客户端后，就可以在服务器端下载文件了。客户端需要上载文件的时候，操作步骤和此类似。此连接为数据通道，主要用来传输数据。FTP 协议是运行在 TCP 之上的，从而保证了 FTP 传输数据的准确有序。图 24-1 描述了 FTP 协议的工作方式。

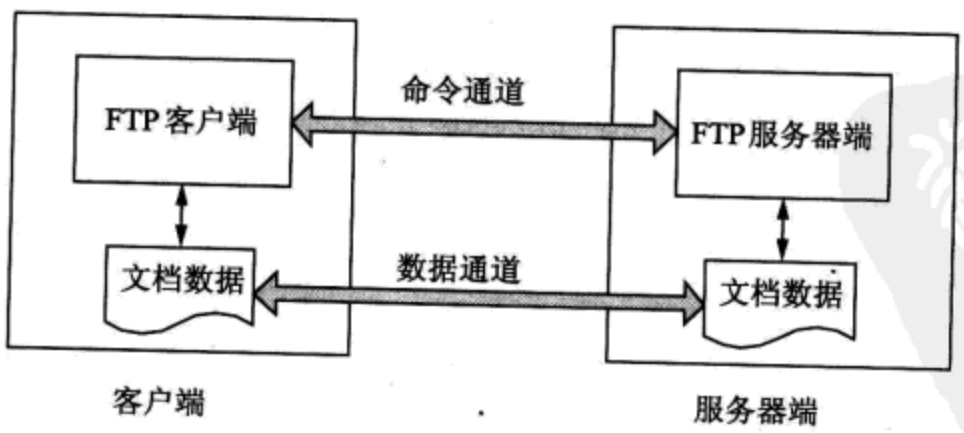


图 24-1　FTP 协议的工作方式

在 Python 标准库中，ftplib 模块提供了对 FTP 客户端实现的支持。由于 FTP 服务器的实现比较复杂，因此在标准库中没有包含此实现。如果需要开发 FTP 服务器端，可以使用第 23 章中介绍的 Twisted 框架。在后面介绍的网络应用中，也只是介绍了客户端的实现，而没有介绍网络服

更多资源请访问稀酷客(www.ckook.com)

务器端的实现。这一节中介绍的协议的服务器端实现，都可以在 Twisted 框架中找到相关的实现。

FTP 协议和 HTTP 协议有点相似，都可以获取文档数据，但是两者也有着本质的区别，使得其成为不同类型的协议。最大的区别在于前者是将命令和数据分开传输的，这被称为带外数据。而在 HTTP 协议中，控制信息和数据信息都是放在一起的，这被称为带内数据。由于两者的相似，所以都可以采用 urllib 等模块来获取文档资源。这只需要将 URL 中的协议值设置为“FTP”即可。实际上，在 ftplib 模块中，对 URL 的处理还是调用 urllib 模块中的函数来处理的。同时，也由于两者的不同，在实现 FTP 客户端的时候，还是需要使用 ftplib 模块。

24.1.2　FTP 的登录和退出

在 ftplib 模块中有一个 FTP 类。使用此类可以构建一个 FTP 客户端的对象。使用此对象的方法即可以完成大部分 FTP 客户端的操作：

```
In [1]: import ftplib

In [2]: ftp = ftplib.FTP()

In [3]: ftp
Out[3]: <ftplib.FTP instance at 0x00BDA9B8>
```

- 在 In［1］中导入了 ftplib 模块，将其中的符号导入到环境中。接着在 In［2］中使用 ftplib 模块中的 FTP 类构造了一个实例。这在 Out［3］的输出结果中也可以看出来。

在 FTP 类的构造函数中有几个可选的参数，主要有需要连接的主机地址和需要认证的用户名以及密码等。在此构造函数中提供主机地址和认证信息与使用 FTP 对象的方法的效果是相同的。这从 FTP 类构造函数的源代码可以看出来：

```
def __init__(self, host='', user='', passwd='', acct=''):
    if host:
        self.connect(host)
        if user: self.login(user, passwd, acct)
```

- 在这个代码段中，首先判断是否提供了主机地址。如果提供了主机地址，则可以使用其对象的 connect 方法连接主机。然后判断是否提供了 user 信息，如果提供了 user 信息，则使用其对象的 login 方法来进行认证。

【例 24-1】演示在 FTP 构造函数中及使用 FTP 对象方法的登录方式。

```
In [4]: ftp = ftplib.FTP("192.168.3.1","admin","admin")

In [5]: ftp.quit()
Out[5]: '221 Goodbye.'

In [6]: ftp = ftplib.FTP()

In [7]: ftp.connect("192.168.3.1")
Out[7]: '220 FTP Server@ IF ready'

In [8]: ftp.login("admin","admin")
Out[8]: '230 User admin logged in.'
In [9]: ftp.quit()
Out[9]: '221 Goodbye.'
```

- 在 In［4，5］ 中，演示了在 FTP 构造函数中提供相关连接信息的登录方式。在 In［6-9］中，演示了使用 FTP 对象方法的登录方式。

更多资源请访问稀酷客(www.ckook.com)

- 在 In［5，9］中都调用了 FTP 对象的 quit 方法。这将发送“QUIT”命令给 FTP 服务器，从而断开连接。
- 两种登录方式最终的效果是一样的，都是和 FTP 的服务器端建立了一个命令通道。然后可以通过该通道发送 FTP 命令来传输数据等。而退出连接命令“QUIT”也是通过该通道来传输的。
- 另外的一个不同是在使用 FTP 的对象方法的时候，会有一些数据输出。例如，在使用 connect 方法时，连接成功后显示了服务器端的欢迎信息。在用户登录成功后，会显示用户登录成功的消息。其实 ftplib 模块也提供了查看这些信息的方法：

```
In [11]: ftp.getwelcome()
Out[11]: '220 FTP Server@ IF ready'
```

- 这里的 getwelcome 方法显示了初次连接 FTP 服务器的信息。这些信息中一般会显示服务器的相关信息或者是帮助信息等。

由于很多 FTP 服务器支持匿名登录，因此 ftplib 模块中的方法也支持这一点。默认情况下，passwd 和 acct 参数为空字符串。当没有提供 user 参数时，默认为“anonymous”。当 user 参数为“anonymous”时，passwd 默认为“anonymous@”。login 方法在连接建立后，只能调用一次。在认证之后，就不应该再使用此方法提供认证。

ftplib 模块已经封装了 FTP 协议中的部分常用命令，如 QUIT 等，但是对 FTP 协议中的很多命令，ftplib 模块并没有进行封装。具体的 FTP 命令可以从 RFC95 中查看到。发送命令也可以通过 sendcmd 方法来实现：

```
In [12]: ftp.sendcmd("PASV")
Out[12]: '227 Entering Passive Mode (192,168,3,1,249,215).'
```

- 在 In［12］中向 FTP 的服务器端发送了一个 PASV 命令，进入被动模式。当然，此功能也可以使用 set_ pasv 方法来实现。也可以使用 sendcmd 方法来发送其他 FTP 命令。

24.1.3　FTP 的数据传输

FTP 协议最大的用途就是进行文件传输。在下面的代码片段中演示了 ftplib 模块中的 FTP 协议数据传输功能：

```
In [15]: ftp.pwd()
Out[15]: '/home/admin'

In [16]: ftp.mkd("python_book")
Out[16]: '/home/admin/python_book'

In [17]: ftp.cwd("python_book")
Out[17]: '250 CWD command successful'

In [18]: fp = open('simple_server.py','rb')

In [19]: ftp.storbinary("STOR simple_server.py",fp)
Out[19]: '226 Transfer complete.'

In [20]: ftp.retrlines("LIST")
-rw-r--r--   1 admin  users          331 Nov 29 06:35 simple_server.py
Out[20]: '226 Transfer complete.'
```

- 在 In［15］中使用 pwd 方法得到了当前所在的目录，这个位置是服务器端的当前位置，在传输数据之前一般都要使用此命令查看一下。

更多资源请访问稀酷客(www.ckook.com)

- 在 In［16］中使用了 mkd 方法来新建一个“python_ book”目录。接着在 In［17］中使用 cwd 方法切换到这个目录。在 Out［17］中可以看出当前的命令执行成功。在上面的三个方法中，其方法和 FTP 命令的名字是一样的。
- 在 In［18］中打开了一个文件对象，并在随后使用 storbinary 方法将数据存储到了服务器中。这个方法有 3 个参数。其中前面两个参数是必须的，分别用来表示存储命令和文件对象。这里的存储命令为“STOR simple_ server. py”，具体的过程中将会把 fp 所指向的文件对象数据保存到服务器端的 simple_ server. py 中。在 Out［19］的输出结果中，显示了此命令执行成功，传输数据完成。在 FTP 协议的数据传输中，可以分为文本传输和二进制传输两类，所以在 FTP 对象所支持的方法中，有些命令有两类。其中，后缀为“lines”的命令供文本传输使用，后缀为“binary” 的命令供二进制传输使用。
- 在 In［20］中使用 retrlines 方法，并传递了“LIST”命令。可以看到，在输出结果中显示了当前目录下的文件信息。

在使用 ftplib 模块进行 FTP 客户端程序编写的时候，可以使用 set_ debuglevel 方法来设置显示调试信息。默认情况下调试信息是不会显示的。将参数设置为“2”可以显示连接过程中的所有信息：

```
In [22]: ftp.sendcmd("UMASK")
* cmd* 'UMASK'
* put* 'UMASK\r\n'
* get* '500 UMASK not understood\r\n'
* resp* '500 UMASK not understood'
---------------------------------------------------------------------
error_perm                                Traceback (most recent call last)
#省略部分输出结果

error_perm: 500 UMASK not understood
```

```
In [23]: ftp.sendcmd("QUIT")
* cmd* 'QUIT'
* put* 'QUIT\r\n'
* get* '221 Goodbye.\r\n'
* resp* '221 Goodbye.'
Out[23]: '221 Goodbye.'
```

- 在 In［22］中发送了一个非 FTP 协议中所规范的命令。从调试信息中可以看出客户端向服务器端发送了 UMASK 命令。而服务器端响应此命令“500 UMASK not understood”，表示 FTP 协议不支持此命令。
- 在 In［23］中演示了一个正确命令的调试信息。同时，使用这种方法还关闭了 FTP 连接。

ftplib 模块中提供了对 FTP 客户端实现的基本支持，但是使用 ftplib 模块中的类和方法，也可以实现 FTP 服务器端的镜像等高级功能。

24.2　使用 POP3 获取邮件

电子邮件应用是 Internet 上最古老的应用之一。即使现在，电子邮件应用也只是仅次于 HTTP 服务的第二大应用。电子邮件包括两个步骤：接收邮件和发送邮件。本节将介绍接收邮件的功能实现。而将发送邮件的功能放在下一节中介绍。

24.2.1　POP3 协议介绍

接收邮件用得最多的协议是 POP 协议（邮局协议，Post Office Protocol），默认使用 TCP 的

更多资源请访问稀酷客(www.ckook.com)

110 端口。由于现在使用的最多的是第三版，所以被简称为 POP3 协议。虽然已经有了功能更加丰富的邮件获取协议如 IMAP4 等，但是由于 POP3 协议比较简单，仍得到了广泛应用。在本小节中只关注于 POP3 协议客户端功能的实现。

POP3 采用的仍然是服务器端-客户端工作模式。在这种工作模式下，当前主机一般为客户端，而服务器端为 POP3 的邮件服务器。在 Python 标准库中支持 POP3 协议的实现是 poplib 模块。该模块也只是实现了客户端的功能。

一个典型的邮件客户端获取邮件的步骤如下。

（1）首先使用 DNS 协议来解析 POP3 服务器的 IP 地址。

（2）使用 TCP 协议来连接邮件服务器的 110 端口（默认情况下）。

（3）当客户端连上 POP3 服务器后，将通过 USER 命令来传送邮箱账号，通过 PASS 命令来传送邮箱密码。提供这些信息来完成用户认证。

（4）当完成用户认证后，使用 STAT 命令来获取 POP3 服务器中此账号的邮箱统计资料，包括邮件的总数和邮件大小等。

（5）同时可以使用 LIST 命令列出 POP3 服务器中的已有邮件数量。

（6）邮件客户端可以通过 RETR 命令来获取邮件，并使用 DELE 命令来设置删除标志。

（7）所有的邮件获取完毕后，可以使用 QUIT 命令中断与 POP3 邮件服务器的连接。同时，服务器会删除已设置为删除标志的邮件。

在 poplib 模块中包含 POP3 和 POP3_ SSL 这两个类。其中，POP3_ SSL 类为 POP3 类的子类，实现了基于 SSL 的邮件传输。POP3 类对象的大部分方法实现了上面所述典型客户端实现中的各个步骤。表 24-1 中描述了 POP3 类中定义的部分方法。

表 24-1　POP3 类对象中定义的方法

方 法 名	说　明
set_debuglevel	设置调试信息级别
getwelcome	获取服务器的欢迎信息
user	发送邮箱名
pass_	发送邮箱密码，锁定邮箱
stat	获取邮箱的统计信息
list	获取邮箱的邮件列表信息
retr	获取特定的邮件
dele	设置邮件的删除标志
rset	删除所有邮件的删除标志
noop	不做任何事情，保持连接的时候使用
quit	中断连接，解锁邮箱

24.2.2　poplib 模块的使用方法

下面的代码片段演示了如何使用 poplib 模块来获取邮件：

```
In [1]: import poplib

In [2]: pop = poplib.POP3("pop3.server.com")

In [3]: pop.getwelcome()
```

更多资源请访问稀酷客(www.ckook.com)

```
Out[3]: '+OK POP3 ready'

In [4]: pop.user("admin@ pop3.server.com")
Out[4]: '+OK'

In [5]: pop.pass_("admin")
Out[5]: '+OK authorization succeeded (eyou mta)'
```

- 在 In［1］中导入了 poplib 模块，这提供了实现 POP3 客户端功能的方法。
- 在 In［2］中使用 POP3 类的构造函数构造了一个对象，并赋值给 pop 变量。其中，构造函数有两个参数。必须的参数是 POP3 服务器的地址。第二个可选的参数为 POP3 服务器的端口，省略的时候默认为 110 端口。
- 在 In［3］中调用 pop 对象的 getwelcome 方法来获取服务器的输出。在 Out［3］中可以看到，已经连接上了 POP3 服务器。
- 在 In［4，5］中分别输入了邮件服务器的邮箱名和密码，并且通过了服务器的认证：

```
In [6]: pop.stat()
Out[6]: (254, 116747443)

In [7]: resp, items, octets = pop.list()

In [8]: import string, random

In [9]: id, size = string.split(random.choice(items))

In [10]: resp, text, octets = pop.retr(id)
```

- 在 In［6］中使用 stat 方法获取了当前账户的邮件信息。此方法返回一个由当前账户下的邮件数量和邮件的大小这两个元素组成的列表。从 Out［6］的输出结果中可以看出，这个邮箱账号的邮件数量为 254，文件大小为 116747443 字节。
- 在 In［7］中使用 list 方法来获取当前账户的邮件列表。这个方法的返回值是一个三元组，其中第二个成员为一个所有邮件信息的列表。邮件信息包括邮件的序列号和邮件的大小。其他两个成员则分别为响应对象和字节数。
- 在 In［8，9］中选择了返回邮件列表中的一个随机邮件，并使用 split 方法将得到的邮件信息分解成了邮件 ID 和大小信息。
- 在 In［10］中调用了 retr 方法来获取指定的邮件，其返回值和 list 方法的返回值类似。只是这里的第二个成员为单行邮件数据所组成的列表：

```
In [11]: text = string.join(text,"\n")

In [12]: import StringIO, rfc822

In [13]: fp = StringIO.StringIO(text)

In [14]: message = rfc822.Message(fp)

In [15]: for key, value in message.items():
   ....:     print key, "=", value
   ....:
received = from sender by localhost with local (Exim 4.69)
 (envelope-from <sender@ pop3.server.com>)
 id 1KP8H7-0002Jw-5D
 for admin@ pop3.server.com; Sat, 02 Aug 2008 11:59:25 +0800
from = Sender <sender@ pop3.server.com>
content-disposition = inline
```

更多资源请访问稀酷客(www.ckook.com)

```
content-transfer-encoding = 8bit
mime-version = 1.0
user-agent = Mutt/1.5.18 (2008-05-17)
to = Admin <admin@ pop3.server.com>
#省略部分输出结果
date = Sat, 2 Aug 2008 11:59:25 +0800
x-eyoumail-smtpauth = sender@ pop3.server.com
message-id = <E1KP8H7-0002Jw-5D@ localhost>
content-type = text/plain; charset=gbk
x-gpg-fingerprint = 53D2 DDCE AB5C 8DC6 188B  1CB1 F766 DA34 8D8B 1C6D
subject = Re: hi, =? gbk? B? wvNC0zrBy8wo=? =

In [16]: print message.fp.read()
On Sat, Aug 02, 2008 at 11:14:37AM +0800, Admin wrote:
> 测试一下邮件服务器的可用性?
好的

In [17]: pop.noop()
Out[17]: '+OK'

In [18]: pop.quit()
Out[18]: '+OK'
```

- 在 In［11］中将得到的邮件数据列表连成了一个字符串。在 In［13］中使用 StringIO 生成了一个文件对象。
- 在 In［14］中使用 rfc822 模块中的 Message 类构建了一个邮件报文对象，其构造函数的参数为刚刚得到的文件对象。
- 在 In［15］中，从 message 类的字典对象中获取了相关的文件头信息。从输出结果中可以看到，此邮件的文件头有大量的信息，包括发送邮件服务器、发送日期时间和邮件主题等。注意到，这里的邮件头信息是经过编码的。
- 接着使用文件对象的 read 方法读取了文件的主体内容，这从 In［16］的输出结果可以看到。其中，内容使用的是 GBK 编码。
- 在 In［17］中使用了 noop 方法。这个函数不做任何事情，只是为了能够保持连接。因为邮件服务器都会有超时设置，当连接超过了一定的时间，服务器端会主动断开连接。如果此时再发送获取邮件命令，会发生如下面这样的错误：

```
error_proto: -ERR timeout (eyou mta)
```

- 在 In［18］中调用了 quit 方法退出，并删除所有已经设置了删除标志的邮件。

通过调用 set_ debuglevel 方法设置调试信息级别，就可以看到每次输入命令时的具体操作信息。通过这些信息可以查看错误的产生原因。

24.3　使用 SMTP 发送邮件

在上面一节中，介绍了获取邮件的功能实现。在本节中，将介绍使用 SMTP 协议来发送邮件。简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）能够可靠高效的传输邮件。

24.3.1　SMTP 协议介绍

SMTP 的服务器端并不一定需要是发送邮件的接收端。这使得 SMTP 协议可以采取一种相当于“接力”的方式在服务器之间来传输邮件。SMTP 协议的原理如图 24-2 所示。

在 Python 标准库中支持 SMTP 协议的是 smtplib 模块。该模块中的 SMTP 类实现了 SMTP 协

更多资源请访问稀酷客(www.ckook.com)

议的客户端功能。一个典型的 SMTP 客户端和 POP3 客户端的操作过程是类似的。只是在最后操作的时候不是获取邮件，而是发送邮件。

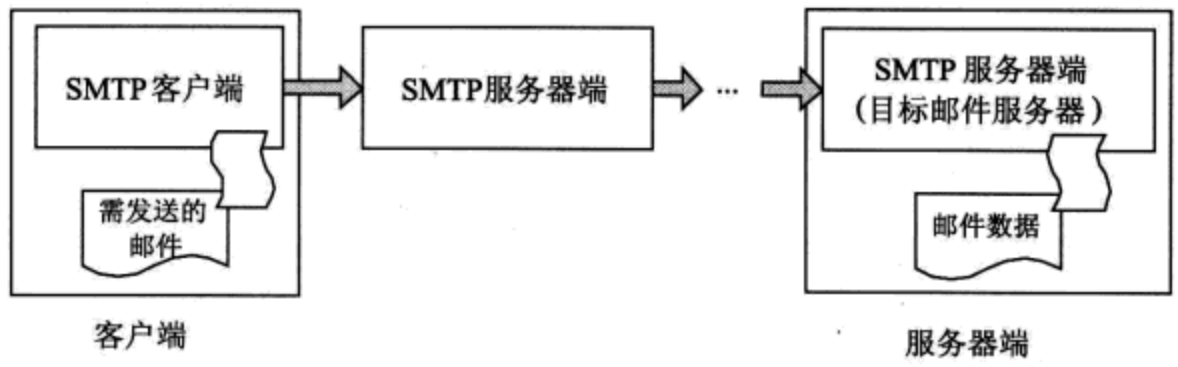


图 24-2　SMTP 协议的原理图

24.3.2　smtplib 模块的使用

下面是一个使用 smtplib 模块发送邮件的示例代码片段：

```
In [1]: import smtplib

In [2]: smtp = smtplib.SMTP("smtp.server.com")

In [3]: smtp.login("admin@server.com", "server")
Out[3]: (235, 'go ahead (eyou mta) (eyou.net_80)')

In [4]: smtp.noop()
Out[4]: (250, 'OK (eyou mta)')
```

- 在 In［1］中导入了 smtplib 模块。这为后面实现 SMTP 协议的客户端功能提供了方法。
- 在 In［2］中使用 SMTP 类构建了一个对象，并连接了 SMTP 服务器。这里采用了直接在构造函数中提供参数的方式。实际上也可以构建一个对象后调用其 connect 方法来连接服务器。其中的两个参数是 SMTP 服务器的地址和端口。端口号在省略的情况下默认为 25。如果将这里的端口地址附加在主机后通过“:”分开，也可以解析出来。
- 在 In［3］中通过调用 login 方法来进行 SMTP 认证。有些 SMTP 服务器不需要认证即可发送邮件，在这种情况下就不需要调用这个方法实现认证，但是这样将会通过这些 SMTP 服务器发送大量的垃圾邮件。因此，现在的大部分 SMTP 邮件服务器都是需要进行认证的。
- 在 In［4］中调用了 noop 方法。这里并没有做任何操作，而只是为了保持连接。每个 SMTP 服务器都有一定的超时时间设置。当超过一定的时间后，服务器端会自动断开连接，所以可以使用此方法来维持连接：

```
In [5]: FROM = 'admin@server.com'

In [6]: TO = 'receiver@server.com'

In [7]: SUBJECT = "Hello"

In [8]: BODY = "Hello, Python."

In [9]: import string

In [10]: body = string.join((
   ....: "From: %s" % FROM,
   ....: "To: %s" % TO,
   ....: "Subject: %s" % SUBJECT,
   ....: "",
   ....: BODY), "\r\n")
```

更多资源请访问稀酷客(www.ckook.com)

```
In [11]: print body
From: admin@ server. com
To: receiver@ server. com
Subject: Hello

Hello, Python.
```

- 这里构造了一个邮件的正文数据。其中，In［5，6］中定义了发件人和收件人的信息。接下来定义了邮件的标题和内容。最后在 In［10］中构造了一个邮件的主体内容。从 In［11］的输出结果中可以看到具体的内容。

```
In [12]: smtp. sendmail(FROM, [TO], body)
Out[12]: {}

In [13]: smtp. quit()
```

- 在 In［12］中调用此 SMTP 对象的 sendmail 方法来发送邮件。其中的 3 个参数分别为发件人、收件人和邮件主体。需要注意的是，发件人只是一个邮件地址；收件人则是一个列表，表示可以为多个收件人；这里的邮件主体是前面构造出来的邮件内容。此调用在发生错误的情况下，将会触发异常。如果没有异常抛出，将返回字典对象，每一项的内容为没有成功发送的邮件地址以及服务器返回的错误信息。在这里，Out［12］中返回了一个空的字典结果，表示所有的发件人都发送成功。
- 最后调用此对象的 quit 方法中断和 SMTP 服务器的连接。

也可以通过调用 set_debuglevel 方法设置调试信息级别，来查看每次和 SMTP 服务器连接的通信细节。

24.4　使用 Telnet 协议远程登录

Telnet 协议也是 TCP/IP 协议族中的成员，提供了远程终端登录应用。Telnet 协议规范在 RFC854 中被定义，在其中还定义了一种通用字符终端，被称为网络虚拟终端（Network Virtual Terminal，NVT）。Telnet 协议可以工作在任何操作系统和任何终端之间。

24.4.1　Telnet 协议介绍和 Python 库的支持

Telnet 协议采用的是带内数据形式，也就是将信令和数据放在一起传输。在 Telnet 协议中，字节 0xff 被作为命令开始的指示。在 Telnet 协议中支持的命令如表 24-2 所示。

表 24-2　Telnet 协议中支持的命令

命 令 名 称	十进制代码	说　　明
ABORT	238	进程异常终止
AO	245	异常终止传输
AYT	246	检测对端是否运行
BRK	243	中断
DM	242	数据标记
DO	253	选项协商
DONT	254	选项协商
EC	247	转义字符

更多资源请访问稀酷客(www.ckook.com)

（续）

命 令 名 称	十进制代码	说　　明
EL	248	删除行
EOF	236	文件结束符
EOR	239	记录结束符
GA	249	继续进行
IAC	255	命令解释标志
IP	244	中断进程
NOP	241	无操作
SB	250	子选项开始
SE	240	子选项结束
SUSP	237	挂起当前进程
WILL	251	选项协商
WONT	252	选项协商

在表 24-2 中有 4 个“选项协商”请求，其中，DO 表示发送端想让接收端激活选项；DONT 表示发送端让接收端禁止选项；WILL 表示发送端自身将激活选项；WONT 表示发送端自身禁止选项。选项协商是双向的，通信的对端可以发送选项协商请求，也可以接受或者拒绝此请求。

在 Python 标准库中，telnetlib 模块实现了 Telnet 协议的客户端功能。该模块中包含 Telnet 类，其中有实现 Telnet 协议客户端功能的方法。在 Telnet 类的构造函数中有两个可选的参数。如果生成对象的时候不包含参数，则此对象将不会连接任何的 Telnet 服务器。其中的两个参数分别为 Telnet 服务器的主机地址和端口。如果不提供端口号，在默认的情况下为 23 号端口。

由于此协议用于交互式通信，因此类中提供了许多用于发送和接收数据的方法。特别是提供了许多类型的读取数据方法用于读取数据。例如，使用 read_until 方法获取数据直到某个特定的字符串；使用 read_all 方法获取所有的数据，直到遇到 EOF 标志等。还可以使用 set_option_negotiation_callback 方法来进行选项协商，并调用指定的回调函数。

24.4.2　telnetlib 模块的使用方法

下面是一个使用 telnetlib 模块进行远程交互的示例代码片段：

```
In [1]: import telnetlib

In [2]: telnet = telnetlib.Telnet()

In [3]: telnet.open("192.168.3.1")

In [4]: telnet.read_until("login:")
Out[4]: '\r\nLinux 2.6.18-1-386 (192.168.3.2) (pts/1)\r\n\nServer login:'

In [5]: telnet.write("user\n")

In [6]: telnet.read_until("Password:")
Out[6]: ' \x01Password:'

In [7]: telnet.write("password\n")

In [8]: telnet.write("ls\n")
```

更多资源请访问稀酷客(www.ckook.com)

```
In [9]: telnet.write("exit\n")

In [10]: telnet.read_all()
Out[10]: '\r\nLast login: Sun Nov 30 16:19:15 2008 from 192.168.3.2 on pts/0\r
\nLinux ipv6 2.61.8-1-386 #1 Mon Sep 13 23:29:55 EDT 2004 i686 GNU/Linux\r\n\r\nW
elcome!  \r\n\r\nThis is a server of China\r\nrunning Debian Linux!  \r\n\:
~                                                                          $
ls\r\n20062229343265196-1.pdf\r\n8634-2.6.15-172-include.tgz\r\nAbstract.doc
\r\nAbstract.odt\r\nAbstract.pdf\r\n
#省略部分输出结果
```

- 在 In［1］中导入了 telnetlib 模块。这为后面实现 Telnet 协议的客户端功能提供了方法。
- In［2］中使用 telnetlib 模块中的 Telnet 类构建了一个对象。其中没有包含参数，所以实例化的时候不会做更多的操作。
- 在 In［3］中调用了 open 方法。此方法将会连接指定的 Telnet 服务器端。
- 在 In［4-7］中输入了用户名和密码来进行认证。其中，在 In［4］中使用 read_until 方法来获取数据。这里的参数为“login:”，表示在连接 Telnet 服务器端之后，客户端等待此字符串的出现。此方法还有一个 timeout 的超时参数。此参数指定了等待数据的超时时间。当超过此时间之后，如果还没有读取到特定的字符串，将触发异常。在 Out［4］中输出了获取此字符串之前的数据。然后通过 write 方法向服务器端发送数据，这里将用户名发送到了服务器端。接着通过同样的步骤发送了密码。
- 在 In［8］中使用 write 方法输入了“ls”命令。这里输入的命令是在 Telnet 服务器端系统上所支持的命令。
- 在 In［9］中同样调用 write 方法向 Telnet 服务器端发送了“exit”命令。当服务器端的系统收到此命令后，将会断开连接。
- 在 In［10］中使用 read_all 方法获取所有在缓冲区中的数据，直到遇到“EOF”标志为止。

与上面介绍的邮件处理相关协议实现类似，telnetlib 模块也可以通过调用 set_debuglevel 方法设置调试信息级别，来查看每次和 Telnet 服务器连接的通信细节。

24.5　使用 SNMP 管理网络

随着网络规模的扩大，需要使用一种有效的协议来管理网络。这个时候，简单网络管理协议（Simple Network Management Protocol，SNMP）被提了出来。SNMP 协议提供了几种简单的操作集合，使得网络管理员可以很方便有效的来管理网络设备，包括路由器、交换机、服务器和打印机等。

24.5.1　SNMP 协议的组成

通过 SNMP 协议管理的信息非常丰富，包括接口的状态和信息等，甚至可以是交换机的内部温度等。虽然 SNMP 协议定义中号称是要尽量简单，但是实际上，SNMP 协议的组织并不简单。

SNMP 协议采用的也是客户端-服务器端架构，但是和其他协议的组织并不一样。在一般情况下，都是客户端连接服务器端。但是在 SNMP 协议中，采取的是服务代理（Agent）和管理站（Network Management Station，NMS）模型。在每个需要管理的网络设备上，都有一个 SNMP 代理。而对网络设备的管理和维护则是通过管理站来和 SNMP 代理之间进行交互的。在 SNMP 协议中，每个 SNMP 代理将会返回代理中 MIB 定义的各种信息。图 24-3 中显示了 NMS 和网络代理之间的关系。

更多资源请访问稀酷客(www.ckook.com)

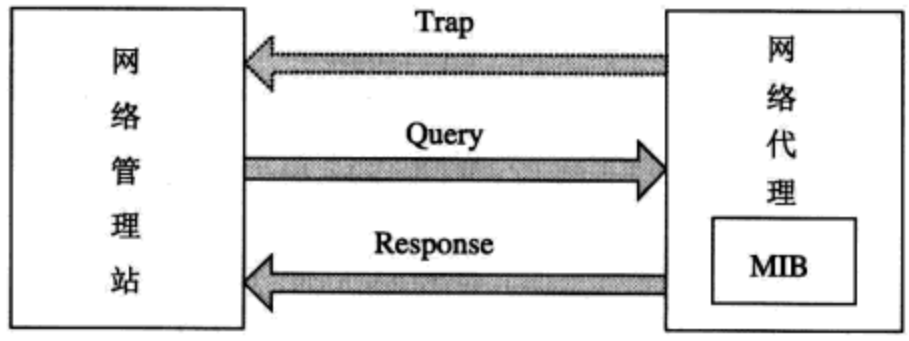


图 24-3　NMS 和网络代理之间的关系

在 NMS 和网络代理之间通过 SNMP 标准消息来通信。SNMP 使用的是 UDP 传输协议。每个消息之间都有一个单独的数据报文。报文有两个部分，分别是 SNMP 的报文头和报文数据。图 24-4 中显示了具体的 SNMP 报文信息。

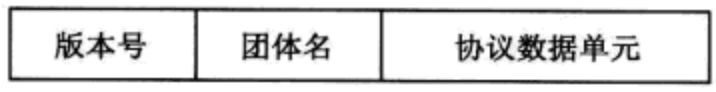


图 24-4　SNMP 报文消息的组成

由于 SNMP 协议有好几个版本，所以在每个报文数据的最前面是版本号。现在 SNMP 有 3 个版本，而且在网络设备中都有使用。现在用得比较多的是第二版本和第三版本。对于不支持的协议版本，网络代理可以直接忽略。团体名是 SNMP 代理用来对 SNMP 管理站进行认证的数据。当失败的时候，将从代理向管理站发送一个认证失败的 Trap 消息。在协议数据单元中包含 SNMP 的消息类型和具体的内容。

24.5.2　PySNMP 框架介绍及其使用方法

在 Python 标准库中没有包含对于 SNMP 协议的实现。但是现在已经有不少第三方使用 Python 语言的 SNMP 协议实现。这里主要介绍 PySNMP 框架，其项目主页地址为 http：//pysnmp.sourceforge.net。其稳定版 2.x 版本提供了 SNMP v1/v2c 的功能，而开发版 4.1x 版本则全面支持 SNMP 三个版本和 MIB 查找等新功能。用户可以根据所管理的网络设备情况和系统需求来选择不同的框架版本。在安装了 PySNMP 软件包后，在 scripts 目录下会有一套用于 SNMP 管理的工具，其中包括 pysnmpget、pysnmpset、pysnmpwalk 和 pysnmpbulkwalk 等。这些工具的用途从其名字也可以看出来。例如 pysnmpget 主要是使用 SNMP 协议的 get 方法来获取网络设备数据。

下面的代码片段使用了 PySNMP 框架来设置和获取网络设备的相关信息：

```
In [1]: from pysnmp.entity.rfc3413.oneliner import cmdgen

In [2]: from pysnmp.proto import rfc1902

In [3]: errorIndication, errorStatus, errorIndex, varBinds = cmdgen.CommandGenerator().
setCmd(
   ...: cmdgen.CommunityData('public','public',1),
   ...: cmdgen.UdpTransportTarget(('192.168.3.253',161)),
   ...: ((1,3,6,1,2,1,1,1,0), rfc1902.OctetString(Server Description))
   ...: )

In [4]: print errorIndication
None

In [5]: print errorStatus
6

In [6]: print errorStatus.prettyPrint()
noAccess(6)
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［1，2］中导入了相关的对象，这将在下面的 SNMP 应用中使用到。
- 在 In［3］中使用 CommandGenerator 类对象的 setCmd 方法来对网络设备设置数据。其中，团体名为 public。需要设置的网络主机地址和端口在 UdpTransportTarget 方法的参数中指定。端口号 161 是 SNMP 服务的默认端口号。接下来，这里的对象标识符“(1，3，6，1，2，1，1，1，0)”表示网络设备的描述。最后面的数据为待设置的数据。此方法返回一个四元组，为 errorIndication，errorStatus，errorIndex 和 varBinds。其中，非空的 errorIndication 表示发生了 SNMP 系统错误；errorStatus 和 errorIndex 一起用来指示协议报文发生错误的位置，当 errorStatus 被解释为 True 的时候，则 errorIndex 指示发生错误的位置；varBinds 是一组管理对象的元组，其中包含请求的数据。
- 在 In［4，5］中查看了 errorIndication 和 errorStatus 的信息。可以看到 errorStatus 并不为 0，则在 In［6］中可以使用 errorStatus 的 prettyPrint 方法对错误信息进行直观的显示。从输出结果来看，这里出错的原因是因为没有权限（“noAccess”）：

```
In [7]: errorIndication, errorStatus, errorIndex, varBinds = cmdgen.CommandGenerator().
getCmd(
   ...: cmdgen.CommunityData('public','public',1),
   ...: cmdgen.UdpTransportTarget(('192.168.3.253',161)),
   ...: ((1,3,6,1,2,1,1,1,0))
   ...: )

In [8]: print errorIndication
None

In [9]: print errorIndex
0

In [10]: print varBinds
[(ObjectName('1.3.6.1.2.1.1.1.0'), OctetString("'Cisco Internetwork Operating Sy
stem Software \\r\\nIOS (tm) C2900XL Software (C2900XL-C3H2S-M), Version 12.0(5.
2)XU, MAINTENANCE INTERIM SOFTWARE\\r\\nCopyright (c) 1986-2000 by cisco Systems
, Inc.\\r\\nCompiled Mon 17-Jul-00 17:35 by ayounes'"))]
```

- 在 In［7］中使用了 getCwd 方法，即使用了 SNMP 的 get 命令来获取网络设备的数据。这里的参数和前面 In［3］中 setCwd 方法的参数类似。区别在于这里没有需要设置的数据信息。
- 在 In［8，9］中输出了错误指示信息。从 In［9］的输出结果中可以看到，这里的 SNMP GET 命令运行没有发生错误。
- 在 In［10］中输出了 varBinds 的值。在输出结果中，输出了 IP 地址为“192.168.3.253”的网络设备的系统信息。从具体的输出内容来看，此网络设备为 Cisco 设备。

关于使用 SNMP 协议来操作网络设备的更多内容，可以参看相关文档。

24.6　网络分析

通过网络分析可以有效地知道网络协议的运行过程，排除网络运行中出现的故障。在网络应用中，通过抓包来分析网络是网络管理员必备的一种技术。

24.6.1　网络分析概述

在 Python 中使用与网络相关的模块时，可能会因为网络断开等各种原因而造成运行错误。当出现这些错误的时候，可以通过特定协议模块所提供的调试方法来判断错误发生的原因。如通

更多资源请访问稀酷客(www.ckook.com)

过 poplib 模块中的 set_debuglevel 方法，可以知道获取邮件的时候命令的发送以及响应信息。通过这些信息可以大致判断出问题出现的原因。在实际使用中，这种方法有两个缺陷：第一，不是所有的网络相关模块都提供了调试所用的方法；第二，不能知道每个连接通信的细节，所以这时候，就需要通过对数据报文进行直接抓取来判断问题所在。

对通信报文进行直接抓取不但可以知道在网络连接中出现的问题，还可以通过这些数据来对通信协议进行分析。在开源世界中最常用的网络分析工具是 Wireshark 软件。图 24-5 所示是 Wireshark 工具软件在分析一个 HTTP 会话时候的截图。

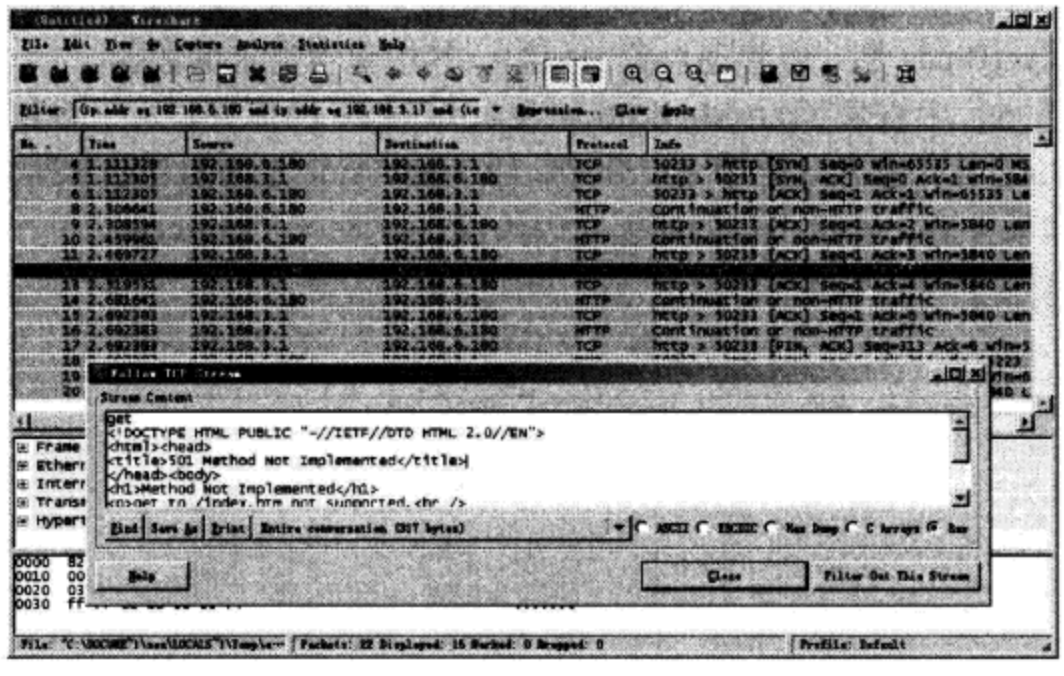


图 24-5　Wireshark 分析 HTTP 会话的截图

从图 24-5 所示的截图中可以看出，这里给出了在 HTTP 会话中通信的每个报文。包括最初客户端发送到 HTTP 服务器上的 SYN 报文等连接建立时候的报文。在截图中还使用了 Filter 功能，从而在这里仅仅显示了 HTTP 会话的通信数据。实际上，在不设置 Filter 的情况下，将会收到在此监听期间的其他类型报文，如 ARP 报文数据等。

24.6.2　使用 Scapy 在网络中抓包分析

各种网络分析软件大部分都是基于抓包库 libpcap 的，包括 Wireshark 工具。同样地，有多种 libpcap 的 Python 绑定接口。但是，这些封装的使用都比较麻烦。Scapy 是一个使用 Python 语言开发的交互式网络分析工具。项目的主页地址为 http：//www. secdev. org/projects/scapy/。可以通过 Scapy 来监听、管理和发送网络数据，同时还可以测试设备和探测以及发现网络等。

现在 Scapy 仅仅有一个 scapy. py 文件，既可以直接使用它进入交互式环境，也可以作为模块导入来使用。在使用网络抓包功能的时候，因为需要对网络接口做特权操作，所以需要有管理员权限。

【例 24-2】演示 Scapy 的使用方法。

```
In [1]: from scapy import *

In [2]: ls()
ARP         : ARP
ASN1_Packet : None
BOOTP       : BOOTP
CookedLinux : cooked linux
DHCP        : DHCP options
DNS         : DNS
DNSQR       : DNS Question Record
```

更多资源请访问稀酷客(www.ckook.com)

```
DNSRR       : DNS Resource Record
Dot11       : 802.11
#省略部分输出结果
UDP         : UDP
UDPerror    : UDP in ICMP
X509Cert    : None
X509RDN     : None
X509v3Ext   : None
_IPv6OptionHeader : IPv6 not implemented here
```

- 在 In［1］中使用导入模块的方法将 Scapy 中的符号导入。
- 在 In［2］中调用了 ls 方法，通过使用此方法可以获取所有的层。由于输出结果过多，所以这里省略了部分输出结果。不过从结果可以看出来，在 Scapy 中实现了许多的层次。

Scapy 可以通过 sr1 来发送报文数据：

```
In [3]: sr1(IP(dst = "192.168.3.1")/TCP())
Begin emission:
Finished to send 1 packets.
.....*
Received 7 packets, got 1 answers, remaining 0 packets
Out[3]: <IP  version = 4L ihl = 5L tos = 0x0 len = 44 id = 0 flags = DF frag = 0L ttl = 60 proto
 = tcp chksum = 0x1117 src = 192.168.3.1 dst = 192.168.3.180 options = '' |<TCP  sport = htt
p dport = ftp_data seq = 1869503955 ack = 1 dataofs = 6L reserved = 0L flags = SA window = 584
0 chksum = 0x85ea urgptr = 0 options = [('MSS', 1460)] |<Padding  load = '\x00\x00' |>>>>
```

- 在 In［3］中使用 sr1 方法来发送一个 TCP 数据到特定的主机。在 Out［3］中可以看到主机对此 TCP 消息的应答。其中有详细的各层数据信息。例如其中 IP 的版本号为 4，TTL 值为 60 等。

可以使用 sniff 方法来获取指定网络接口上的数据：

```
In [4]: sniff(prn = lambda x: x.show(), count = 100)
###[ Ethernet ]###
  dst = 00:15:c5:e1:c8:95
  src = 00:1e:ec:61:cb:f3
  type = 0x800
###[ IP ]###
     version = 4L
     ihl = 5L
     tos = 0x0
     len = 48
     id = 11032
     flags =
     frag = 0L
     ttl = 128
     proto = udp
     chksum = 0x6a05
     src = 192.168.3.180
     dst = 123.155.17.129
     options = ''
###[ UDP ]###
        sport = 26214
        dport = 8631
        len = 28
        chksum = 0x340d
###[ Raw ]###
           load = 'd(^\xdd\r:p\xd50tGP7\x00,\xa6O$2G'
###[ Ethernet ]###
  dst = ff:ff:ff:ff:ff:ff
  src = 00:16:d3:c8:db:25
```

更多资源请访问稀酷客(www.ckook.com)

```
  type = 0x800
###[ IP ]###
     version = 4L
     ihl = 5L
     tos = 0x0
#省略部分输出结果
```

- 在 In［4］中使用 sniff 方法来捕获 100 个报文，并打印出具体的数据。从这里可以看出，Scapy 输出了各种层次的网络数据。如果不需要详细的数据信息，可以使用 summary 方法。这样仅仅只是会显示统计信息：

```
In [5]: sniff(prn = lambda x: x.summary())
 802.3 00:0f:e2:07:f2:e0 > 01:80:c2:00:00:00 / LLC / STP / Raw
Ether / ARP who has 192.168.3.2 says 192.168.3.1 / Padding
802.3 00:14:69:d2:6b:82 > 01:00:0c:cc:cc:cd / LLC / SNAP / STP / Raw
Ether / ARP who has 192.168.3.140 says 192.168.3.12 / Padding
00:12:44:36:80:00 > 33:33:00:00:00:01 (0x86dd) / Raw
Ether / ARP who has 192.168.3.20 says 192.168.3.24 / Padding
#省略部分输出结果
```

- 在 In［5］中使用 summary 方法输出了统计信息。

除了这些功能之外，Scapy 还提供了众多方法用来测试和诊断网络。例如可以使用 traceroute 方法来获取路由路径等。通过获取具体的网络数据，可以有效的分析通信协议。

24.7　小结

本章讲解了常见的 Python 网络应用。首先介绍了 FTP 协议并使用 ftplib 模块进行文件传输，接着通过 POP3 和 SMTP 协议来收发邮件。然后介绍了使用 telnetlib 模块来进行远程登录操作。在介绍了网络管理的基础知识后，对使用 PySNMP 框架进行了说明。最后，介绍了网络分析以及 Scapy 模块的使用方法。

更多资源请访问稀酷客(www.ckook.com)

第 25 章　图像处理和游戏开发

随着信息内容的丰富，更多的数据将会使用多媒体的形式存储。能够快速的处理图像已经成为一种常见的任务，包括调整图像大小、对图像进行增强处理等。另外，游戏功能是多媒体开发中的一个重点。本章的知识点：

- 图像处理的基本概念
- 变换等图像的基本处理方法
- 对图像进行增强处理
- PyGame 模块介绍
- 使用 PyGame 开发游戏

25.1　图像处理的基本概念

在 Python 的标准库中，虽然没有用于对图像处理的模块，但是有大量的第三方模块可用来完成对图像的处理工作。这些处理工作包括从简单的图像裁剪到复杂的人像识别等。在本节中，将介绍 Python 下的图像处理包，并具体介绍 PIL 模块对图像处理的支持。

25.1.1　Python 下的图像处理包

Python 标准库中没有包含图像处理的相关模块。但是实际上，模块中还是有与图像处理相关的模块的。由于这种模块如 jpeg 等只能用在 IRIX 操作系统下，所以严格地说在 Python 标准库中没有一种通用的图像处理模块。由于 jpeg 等模块是基于 IRIX 的图形库的，因此在使用的时候也是需要有 IRIX 图形库的支持。基于这种处理模块的局限性，在这里不做更多的讨论。可以在 Python 语言的标准库文档中查看其使用方法。

在 Python 中，有许多第三方模块提供了图像处理功能。其中一种是 PythonMagick。这是一个 ImageMagick 软件包的 Python 绑定。ImageMagick 是一款可以运行在多种系统平台下的用来创建、编辑和合成位图的软件库，可以将其看作是图像处理中的“瑞士军刀”。支持的图像格式包括 DPX、EXR、GIF、JPEG、PNG 和 SVG 等。使用 ImageMagick 图像处理套件可以转换、旋转、缩放、裁剪图像，也可以对图像进行色彩处理等操作。除此之外，还可以在图像上实现一些特殊的效果，包括在位图上写字和画上一些简单的图形。PythonMagick 作为 ImageMagick 套件的 Python 语言绑定，实现了 ImageMagick 套件的功能。可以通过使用 PythonMagick 来实现对图形图像的处理，如裁剪等操作。

另外一款强大的图像处理套件是 OpenCV（Open Source Computer Vision）库。这是一个用于计算机视觉处理方面的图像库。这个库和 ImageMagick 图像处理套件的目的不一样，它更加关注于对图像进行高级处理的功能，包括人脸识别、动作识别和轨迹建模等。此库的功能很强大，但是由于需要比较多的图像学知识，所以一般是用在比较特定的领域中。OpenCV 库有 Python 语言的绑定，可以通过 Python 来访问 OpenCV 所提供的功能。

更多资源请访问稀酷客(www.ckook.com)

虽然上面的图像处理库都能够实现对图像处理的功能模块，但是实际上大部分都只是原来已有的工具的 Python 语言绑定。

PIL(Python Imaging Library) 是一个支持图像处理的 Python 语言库，由 PythonWare 提供。可以从 http://www.pythonware.com/products/pil/主页上下载 PIL 的安装包。当前的最高版本为 1.1.6。和其他 Python 模块一样，可以使用 PIL 的源文件来安装：

```
$ tar zxf pil-1.1.6.tar.gz

$ cd pil-1.1.6/

$ python setup.py install
```

❑ 这里省略了前面的命令行提示符。

在 Linux 系统平台下，可以通过使用软件包的方式来安装。如在 Ubuntu 系统下可以使用 apt-get 命令来获取和安装 PIL：

```
$ apt-get install python-pil
```

❑ 通过使用此命令，系统将自动寻找源并安装对应的软件包。

在 Windows 系统下，可以从项目的主页上下载已经设置好的可执行文件。在下载的时候要注意和当前系统中安装的 Python 版本相对应。PIL 模块在 Windows 系统下的安装过程如图 25-1 所示。

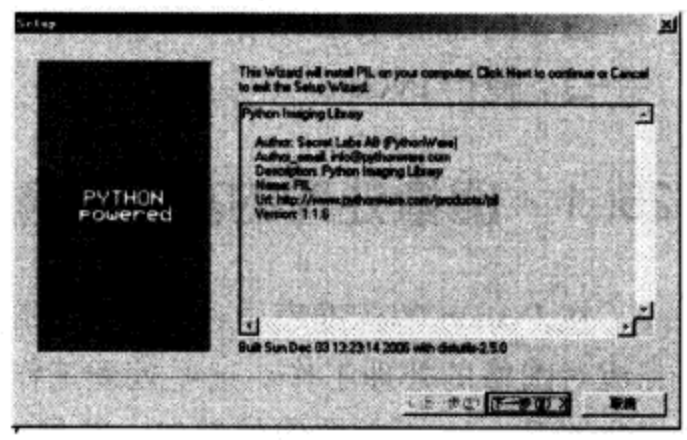


图 25-1　PIL 在 Windows 系统下的安装截图

PIL 模块提供了对于位图的数字处理功能，包括查看图片信息、对图片进行剪切和旋转以及对图像进行进一步的增强功能。虽然在 PIL 中没有如 OpenCV 那样强大的功能，但是所提供的功能已经足够丰富，支持的文件格式也足够多，所以在一般的图像处理中，使用 PIL 是一个很好的选择。

25.1.2　PIL 支持的图像文件格式

在计算机绘图中有两类图像：一类是矢量图，另外一类是点阵图，一般称为位图。

矢量图是一种基于计算机数字对象的绘图，其图形的构成元素包括点、线、多边形等这样的几何图像。在实际显示的时候，一般都是通过数学公式计算得到的。正因为矢量图形的这种实现方式，使得其产生的文件比较小，而且在对图像进行缩放和旋转等操作后，图像不会出现失真。这种图像与分辨率无关，在输出结果的时候不会影响到图像的清晰度。现在 Internet 上流行的 Flash 使用的就是矢量图。矢量图的一个最大的问题是无法表现颜色的细节变化。由于这种类型图像的特点，所以在 PIL 中没有包含对这类图像处理的模块。在介绍 PIL 处理图像的部分，也不支持矢量图像。

位图的基本组成单元为像素。通过这些像素的不同颜色的排列，就构成了颜色丰富的图像。当放大位图的时候，可以看到组成图像的各个像素单元，当放大到一定程度的时候会造成图像失真。由于这种图像需要保存每个像素的内容，所以其文件比较大，而且其大小是随着像素的增加而增加的。但是由于采用了像素这种微小图像单元来显示色彩，这使得位图可以表现丰富的颜色，从而得到逼真的图像效果。这样在处理位图的时候，就需要考虑分辨率的影响。这对图像最后的输出是很重要的。

更多资源请访问稀酷客(www.ckook.com)

由于位图保存时候的文件比较大，这使得位图有多种不同的存储方式被开发出来。下面是对常见的位图图像文件格式的介绍。

（1）BMP 格式。BMP 格式是 Windows 系统平台下的标准图像文件格式，已经被广泛的支持。在 BMP 图像中，所包含的图像信息相对比较丰富，很多图像处理软件都将其作为中间格式使用。由于 BMP 没有进行压缩，所以其文件占用空间比较大。其扩展名为 bmp。

（2）JPEG 格式。JPEG 是联合图片专家组（Joint Photographic Expert Group）开发并命名的一种图像格式。这种图像文件采用了一种先进的压缩技术，对图像中的视觉不敏感部分进行了有损压缩，保持了其中主要的图像特征。这使得 JPEG 格式的文件相对比较小，从而在 Internet 上推广开来。后来，JPEG 组织更是推出了 JPEG2000 格式，具有了更高的压缩比和更多的图像特性。如 JPEG2000 可以实现渐进传输，先传递轮廓，再传递细节数据。其扩展名为 jpeg 或 jpg。

（3）GIF 格式。GIF（Graphics Interchange Format）格式是 CompuServe 机构针对网络传输带宽的限制而开发出来的一种图片格式。在 GIF89a 标准中，图像中可以包含透明区域，同时可以存储多幅静态图片而形成连续的动画。这种格式压缩比高，产生的文件比较小。GIF 格式的这些特点使得其在 Internet 上得到了广泛的应用，但是 GIF 格式只能存储不超过 256 色的图像。其扩展名为 gif。

（4）PNG 格式。PNG（Portable Network Graphics）是一种新兴的网络图像格式。最初是为了打破 GIF 格式的专利垄断而设计的。PNG 在设计的时候考虑到了文件大小和图像质量的关系，采用了一种无损压缩方式来对图像文件进行压缩。这使得 PNG 格式的图像在保持图像质量的同时，减小了文件大小。同时，PNG 还支持设置透明区域，这使其在网络设计中也具有一定的优势。现在大部分的图像处理软件都已经支持 PNG 格式。其扩展名为 png。

在位图中还有大量的其他图像文件格式，如 PSD、TIFF 和 PPM 等。这些图像格式都各有特点，各有不同的用途。实际上，PIL 模块支持将近 30 多种不同的图像文件格式，其中就包括上面介绍的这几种。由于不同文件格式的读取和保存方式不一样，所以 PIL 对于部分特殊的图像文件格式只有读取或保存功能。当然，对于上面介绍的这 4 种最常见的图像文件格式，PIL 都提供了良好的读取和保存功能。

由于 PIL 模块采用了一种插件的方式来实现对图像文件的支持，所以对于 PIL 不支持的图像文件格式，用户还可以自己书写特定格式的解码插件。这种使用插件的方式不需要对原来的 PIL 库进行修改。这些图像文件格式的解码插件一般为 XxxImagePlugin. py，其中 Xxx 为一种特定的图像文件格式。

25. 1. 3　图像处理中的其他概念

由于位图中是采用像素来保存颜色，所以图像的大小就是指其水平方向和垂直方向上的像素个数。如 1600 × 1200 表示长为 1600 像素、宽为 1200 像素的图像大小。在 PIL 模块中，可以通过 size 属性来获取图像的大小。这是一个有两个元素的元组，其中的含义分别为长和高。

为了能够定位图像上的一个点，在 PIL 中采用的是笛卡儿坐标系，其中（0，0）位于屏幕的左上角。通过一个定位点加上长和高就能够定位图像中的矩形了。如（0，0，800，600）表示图像中左上角 800 × 600 大小的一个矩形。

另外一个重要的概念是颜色模式。颜色模式决定了图像如何描述和重现图像中的色彩。在显示和打印输出的时候将会用到此概念。PIL 所支持的常见颜色模式有 RGB、CMYK 和灰度模式等。这些不同的颜色模式专注于不同的色彩表达方式，用在不同的领域。

RGB 颜色模式采用的是将色彩分成 3 个颜色通道：红色通道、蓝色通道和绿色通道。每个通道使用 8 位来表示颜色信息，从 0 到 255 分别用来表示该种颜色的多少。最终色彩的显示就是

更多资源请访问稀酷客(www.ckook.com)

通过这三种颜色的叠加来实现的。RGB 颜色模式可以产生大量肉眼可识别的颜色，适合于显示器的输出。在计算机上做图像处理的时候，大部分情况下都是使用这种颜色模式。

CMYK 颜色模式是采用青（Cyan）、品红（Magenta）、黄（Yellow）和黑（blacK）4 种油墨颜色来生成色彩，所以这种颜色模式一般是用于印刷领域。CMYK 虽然也可以产生 RGB 颜色模式下的颜色，但是两者产生的原理是不一样的。其中，RGB 产生颜色的方法称为加色法，因为是叠加各个颜色通道来产生颜色的。CMYK 模式则被称为减色法，通过色素合成后吸收光线来产生不同的颜色。CMYK 模式的图像文件尺寸比较大，所以一般只是在需要将图像印刷的时候才将图像转化为 CMYK 颜色模式。

另外，灰色颜色模式只包含灰度信息，而不包括彩色信息。这种灰度图像可以看作是只由一种颜色通道组成的。一般是使用 8 比特来表示颜色信息。

在 PIL 中，图像的颜色模式是通过 mode 来定义的。现在 PIL 支持的模式有 1、L、P、RGB、RGBA、CMYK、YcbCr、I 和 F 等。另外，对于其他几种特殊的模式也有部分的支持，包括 LA、RGBX 和 RGBa。

PIL 模块中支持的插值算法包括如下 4 种。

（1）NEAREST。这是一种最近邻插值方法，将会选择输入图像中最近的像素来处理。

（2）BILINEAR。这是一种双线性插值方法。在操作的时候将会对输入的图像选择 2 ×2 的区域来进行线性插值。

（3）BICUBIC。这是一种双立方插值方法。在操作的时候将会对输入的图像选择 4 ×4 的区域来进行立方插值。

（4）ANTIALIAS。这是一种通过使用高质量的采样器作用在所有输入图像的像素上，从而得到输出图像的方法。

在 PIL 中，当需要将一个图像缩小的时候，ANTIALIAS 是唯一的插值方法。而 BILINEAR 和 BICUBIC 更适合于等比例变换或者是图像放大等操作。

25.2　图像的基本处理

使用数字图像，首先要根据需求对其进行基本的处理。这些功能需求包括图像文件的操作和图像文件的变换等。其中，对于图像文件的操作有读取图像的相关信息、读取和写入特定的文件和在不同的图像文件格式之间转换。而对于图像的变换操作包括剪切图像区域、缩放图像和旋转图像等。本节将介绍这些功能的实现方法。

25.2.1　图像的读写操作

在 PIL 中最重要和最常见的是 Image 模块中的 Image 类。其中包括对图像的大部分处理操作。可以通过几种方法来生成一个图像的对象。最常见的是从文件中读取图像，还可以通过处理其他图像而得到，或者只是建立一个空的图像文件，再在后面进行修改：

```
In [2]: import PIL

In [3]: from PIL import Image

In [4]: im1 = Image.open("Lenna.png")

In [5]: im2 = Image.Image()

In [6]: im1,im2
Out[6]:
(<PIL.PngImagePlugin.PngImageFile instance at 0x0107A530>,
 <PIL.Image.Image instance at 0x010F6148>)
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［2］中导入了 PIL 模块，这使得后面可以使用该模块中所定义的类和方法。
- 在 In［3］中，从 PIL 中导入了 Image 模块。实际上，在 PIL 库中有许多模块，Image 模块是其中最重要的一个。
- 在 In［4］中使用 Image 模块所提供的 open 方法，从指定的图像文件“Lenna. png”读取了一个 Image 对象。当文件打开出错的时候，将会触发 IOError 异常。
- 在 In［5］中，使用 Image 模块中 Image 类的构造函数定义了一个空的图像对象。当然，这可以在以后通过图像操作来修改。一个更加友好的方式是调用 Image 类中的 new 方法，在其中可以指定图像的颜色模式和大小。
- 在 In［6］中打印出前面创建的两个图像对象。从输出结果中可以看到，两者都是 Image 对象的实例。同时，前面一个通过读取文件生成的图像对象已经识别出这是 PNG 格式的图像文件，并使用了相应的图像插件来处理。

生成了图像实例后，后面针对图像的操作都是在此基础上完成的。Image 类中提供了大量的方法来处理数字图像。首先，可以使用 show 方法将原始的图片显示出来：

```
In [7]: im1.show()
```

- 当 show 函数被调用的时候，控制台将会挂起，并显示图像。



图 25-2　原始图像显示

图 25-2 所示是 Windows 系统下 Lenna. png 文件的显示效果。

图 25-2 中所用的图像是在图形学处理中常用的 Lenna 头像。在 Windows 系统平台下，当 Image 对象的 show 函数被调用的时候，PIL 将会首先生成一个临时文件，然后再使用 Windows 系统中注册的图像显示工具来显示。这种处理方式的效率很低，所以一般只是用于调试。

将图像对象保存起来也很容易，使用 save 函数即可。如果只是需要将源图像保存，直接在参数中加上需要保存的文件名即可：

```
In [8]: im1.save("Lenna_2.png")
```

- 在 In［8］中调用 im1 对象的 save 方法后，将会在源目录中生成一个名为“Lenna_2. png”的文件。

save 方法还可以包含一个可选的文件格式选项，可以通过此选项来实现不同图像文件格式之间的转换。这将在 25. 2. 3 小节中进行介绍。

25. 2. 2　获取图像信息

当生成了 Image 对象后，就可以通过此 Image 对象来获取关于此图像文件的信息。这些信息可以包括在 25. 1. 2 小节和 25. 1. 3 小节中介绍的图形处理基本概念，具体包括分辨率、文件格式、图像文件的大小、颜色模式、文件名和读写属性等。对于仅仅是通过 Image 类的构造函数生成的空文件对象，下面所介绍的大部分操作都是不允许的，因为其图像的文件格式为“None-Type'”。在进行下面的步骤之前，最好是保证图像对象包含有图像数据。

【例 25-1】演示获取“Lenna. png”图像文件信息的操作步骤。

```
In [9]: im1.filename
Out[9]: 'Lenna.png'

In [10]: im1.size
```

更多资源请访问稀酷客(www.ckook.com)

```
Out[10]: (512, 512)
In [11]: im1.mode
Out[11]: 'RGB'

In [12]: im1.getbands()
Out[12]: ('R', 'G', 'B')

In [13]: im1.format
Out[13]: 'PNG'

In [14]: im1.info
Out[14]: {'gamma': 0.45455000000000001}

In [15]: im1.readonly
Out[15]: 0
```

- 在 In［9］中使用 filename 属性输出了此图像对象的文件名。这在需要对图像对象进行进一步操作的时候比较有用。
- 在 In［10］中输出了图像文件的大小。可以看到，这里输出的是一个有两个元素的元组，分别表示图像文件的长和高。这里，Lenna. png 文件大小为 512 × 512。
- 在 In［11］中输出了此图像对象的颜色模式。此图像文件的颜色模式为 RGB。在 PIL 中，常见的颜色模式有 3 种。在灰度图像中，一般会输出“L”模式。对于真彩色的图像，则常为“RGB”模式。对于那些准备印刷的图像，则有可能是“CMYK”模式。在 In［12］中，进一步输出了图像文件颜色模式的通道。这里，RGB 颜色模式包含 R、G 和 B 三个通道。
- 在 In［13］中输出了图像文件的文件格式。此值将会决定此文件的处理插件。这里图像为 PNG 格式，而处理插件为“PngImagePlugin”。需要注意的是，在 PIL 处理图像文件的时候，其扩展名只是作为一种判断手段，所以，即使将此 PNG 图像文件改为 gif 后缀，PIL 仍然可以正常识别出来。因为 PIL 会根据图像文件的文件头来推测相关信息。
- In［14］中输出了特定文件对象的附加信息。其中，info 属性是一个字典对象，使用键值对的方式保存了基本信息外的更多特定文件信息。在本例中，PNG 文件格式包含 gamma 和 transparency 信息。其中，transparency 键值在图像文件不包含透明区域的时候被省略，所以在 Out［14］中只看到了 gamma 的值。
- 在 In［15］中输出了此图像对象的读写属性。当值为 1 的时候表示此图像对象是只读的。此时如果要对此对象进行操作，就需要先对其进行拷贝。

由于不同的图像文件格式之间的差别比较大，所以在这里只是介绍了所有格式通用的一些属性。实际上，每种格式都会有自己特有的一些属性，如 GIF 中可以有 background、duration、transparency 和 version 等属性。读者如果需要了解每种图像文件格式所支持的更多属性，可以参看 PIL 文档中对于文件支持的具体描述。

25.2.3　图像文件格式的转换

由于不同的需要，经常需要将一种图像文件格式转换为另一种图像文件格式。例如，用于网络传输的图片使用 BMP 格式是不合适的。此时可以将其转换为 JPEG 或者 PNG 格式，从而加速网络传输。

在 PIL 模块中可以很方便的实现不同图像文件格式之间的转换。实现的方法就是利用前面已经介绍过的 save 函数。在保存的时候，PIL 将会根据文件名来决定所要保存文件的格式。在这里需要注意到 PIL 对于这些文件的读写支持。对于不支持写入的图像文件格式，使用 save 方法将

更多资源请访问稀酷客(www.ckook.com)

会出错。

【例 25-2】将源图像转换为 JPEG 格式。

```
In [16]: im1.save("Lenna.jpeg")

In [17]: im3 = Image.open("Lenna.jpeg")

In [18]: im3.format
Out[18]: 'JPEG'

In [19]: im3.mode
Out[19]: 'RGB'
```

- 在 In［16］中使用 save 方法将 im1 图像对象保存为了“Lenna.jpeg”文件。这时在文件系统中将会生成一个图像文件。为了验证此图像文件是否为 JPEG 格式的。在 In［17，18，19］中打开了此图像文件并输出了相关的图像信息。从输出结果可以看到，生成的“Lenna.jpeg”文件已经是 JPEG 格式的了。

通过使用这种方法，可以实现 PNG 格式到 JPEG 格式的转换。如果遇到图像文件的其他格式，可以按照这种方法进行操作。在 save 方法中，还有两个可选的参数，其中一个为图像文件的格式信息。PIL 库也会通过此值来获取图像的文件格式信息。当文件名不是标准后缀名的时候，必须使用此值来指定文件格式。

【例 25-3】演示如何输出一个没有扩展名的图像文件。

```
In [20]: im1.save("Lenna","JPEG")

In [21]: im4 = Image.open("Lenna")

In [22]: im4.format
Out[22]: 'JPEG'
```

- 在 In[20]中，使用了 save 方法来保存文件对象，但是现在的文件名为“Lenna”，没有包含任何的文件格式信息，所以需要在第二个参数中指定图像文件的格式。从 In[21,22]的结果来看，这种操作同样很好的实现了不同格式之间的转换。实际上，这里生成的“Lenna.jpeg”和“Lenna”两个文件的数据是相同的。

save 方法中的另一个可选参数是对应图像文件格式的其他信息。如在保存为 JPEG 格式文件的时候，可以设置 quality 的值用来指示 JPEG 的图像质量。图像文件格式的读取信息和写入信息不一定是相同的。

对于图像的不同颜色模式也是可以转换的。Image 类中的 convert 方法实现了这个功能。

【例 25-4】将原图转换成灰度图像。

```
In [23]: im5 = im1.convert("L")

In [24]: im5.show()
```

- 在 In［23］中，使用了 convert 方法，并将其参数设置为“L”。这是一种 8 位的灰度图像。在 In［24］中调用了 show 方法来显示此图像。图 25-3 中给出了此灰度图像的显示。从图 25-3 中可以看出，灰度图像保留了原图除颜色外的大部分图像细节。

也可以将此图像转换成模式为“1”的灰度图像：

```
In [25]: im6 = im1.convert("1")

In [26]: im6.show()
```

图 25-4 中显示了此图像。

更多资源请访问稀酷客(www.ckook.com)



图 25-3　L 模式灰度图像显示



图 25-4　1 模式灰度图像显示

PIL 支持图像在不同的颜色模式之间进行转换。因为有些颜色模式之间不支持直接转换，所以在颜色模式转换的时候，可能需要使用中间颜色模式。

25.2.4　图像的裁剪和合成

在 PIL 模块中有图像的裁剪和合成功能。使用 Image 类中的 crop 方法可以从图像中获取一个矩形空间。crop 方法的参数为一个有 4 个元素的元组，其中 4 个元素分别表示左、上、右和下的坐标。crop 采用的裁剪方式是延时的，也就是说，只有在实际操作的时候才会从图像中获取对应区域的数据，所以对于源图像的修改不一定会影响到剪切的图像。如果需要获得一份剪切区域的特定拷贝，可以对剪切区域调用 load 方法。

【例 25-5】演示剪裁图像的方法。

```
In [27]: rec = (140,100,400,400)

In [28]: region = im1.crop(rec)

In [29]: region.show()
```

- 在 In［27］中使用含有四个元素的元组定义了一个矩形区域，这四个元素分别表示左上右下的坐标。从这里可以看出，此矩形的长和高为 260 和 300 像素。
- 在 In［28］中使用 crop 方法从 im1 图像对象中裁剪了一个矩形区域，该区域就是在 In［27］中定义的矩形。
- In［29］中调用 show 方法显示这个裁剪出来的图像，如图 25-5 所示。



图 25-5　裁剪后的图像

剪切图像之后，可以将此图像对象和其他图像对象进行合成，使之成为新的数字图像。在合成图像时，可以先对需要合成的源图像进行拷贝，这样可以避免对源图像的修改。在 PIL 模块中，可以通过调用 copy 方法来实现：

```
In [30]: im_copy = region.copy()
```

更多资源请访问稀酷客(www.ckook.com)

- 在 In［30］中实现了对图像对象的复制。

PIL 中提供了 paste 方法用来实现两个不同图像对象之间的合成。此方法有两个参数，分别为待粘贴的图像对象和其矩形区域。具体的函数操作如下：

```
In [31]: im7 = Image.open("scene.jpg")

In [32]: im7.paste(region, rec)

In [33]: im7.show()
```

- 在 In［31］中使用 Image 类的 open 方法从文件系统中加载了一个名为“scene.jpg”的文件，该文件在后面合成图像时将作为底图。
- 在 In［32］中调用了生成图像对象的 paste 方法。其中第一个参数为从 im1 图像对象中裁剪出来的部分图像，第二个参数指定了具体的矩形区域坐标信息。
- 在 In［33］中使用 show 方法显示此图像。图 25-6 中显示了合成后的图像。



图 25-6　合成后的图像

当源图像与待粘贴图像的颜色模式不一致的时候，会自动进行颜色模式的转换。在 paste 方法中，还可以有一个可选的透明遮罩参数。当透明遮罩层的参数值为 255 的时候，表示粘贴的图像是没有透明度的。当其数值为 0 的时候，表示粘贴的图像完全透明。通过修改此值，可以改变合成图像的显示。

【例 25-6】演示可选的透明遮罩参数的使用方法。

```
In [34]: mask = region.point(lambda i: i <100 and 255)

In [35]: mask = mask.convert("1")

In [36]: im8 = Image.open("scene.png")

In [37]: im8.paste(im = region,box = rec,mask = mask)

In [38]: im8.show()
```

- 在 In［34］中使用 Image 类中的 point 方法得到了满足参数所指定函数的像素点集，并将其赋值给 mask 变量。
- 对于 mask 层的图像对象，其颜色模式可以为“1”、“L”或“RGBA”等三种颜色模式。在 In［35］中使用了“1”模式。
- 由于前面合成图像的底图已经改变，所以在 In［36］中重新打开了此底图文件。
- 在 In［37］中调用了 paste 方法，其中前两个参数在前面已经介绍过。第三个参数则是在 In［35］中定义的 mask 变量。
- 在 In［38］中使用了 show 方法显示此图像。

图 25-7 中显示了使用 mask 后的合成图像。



图 25-7　使用 mask 后合成的图像

另外，图像对象还可以将图像分解成多个颜色通道来进行单独处理。PIL 中提供了 split 方法来实现此功能。同时，还提供了 merge 方法来合成所提供的颜色通道：

更多资源请访问稀酷客(www.ckook.com)

```
In [39]: r, g, b = region.split()

In [40]: merge_im = Image.merge("RGB",(g,r,b))

In [41]: merge_im.show()
```

- 在 In［39］中使用了 Image 类对象的 split 方法，将指定图像对象分成了不同的颜色通道显示。这对于不同的颜色模式是不一样的。在这里，将 region 变量分成了三个通道，分别为 R、G 和 B 通道。返回结果中的每个颜色通道都是一个 Image 对象，可以使用 Image 类中包含的各种方法。
- 在 In［40］中没有对各个颜色通道进行任何操作，而是直接使用了 Image 类的 merge 方法来将各个颜色通道合并成一个图像对象。这里的颜色模式为“RGB”，而其具体的颜色通道为(g, r, b)。需要注意的是，这里对 R 颜色通道和 G 颜色通道进行了交换。
- 在 In［41］中调用 show 方法显示了此图像。

图 25-8 中显示了此多颜色通道合成后的图像。

比较图 25-8 和图25-5两幅图像，可以看出两者有很明显的不同。



图 25-8　多颜色通道合成后的图像

25.2.5　图像的变换

在 PIL 中提供了丰富的函数来操作图像的变换，包括图像的缩放和旋转等。

1. 图像的缩放

可以使用 resize 方法对图像进行缩放。在其参数中直接指定需要转换的图像大小。

【例 25-7】演示图像的缩放操作。

```
In [42]: im8 = im1.resize((200,200))

In [43]: im8.show()
```

- 在 In［42］中调用 Image 对象的 resize 方法对图像进行缩放。其参数为一个有两个元素的元组，两个元素分别表示图像的长和高。
- 在 In［43］中调用 show 方法显示此图像。

图 25-9 中是源图像缩放成为 200 ×200 大小的显示。

由于图像的变换涉及不同像素的输入输出，因此不同的插值算法将得到不同的图像效果。在 PIL 库中支持的插值算法在 25.1.3 小节中进行了介绍。在 resize 以及后面要介绍的 thumbnail、rotate 和 transpose 方法的参数中，都有一个可选的 Filter 参数。可以根据图像质量的需要来选择合适的插值算法。如果省略此参数，将会被指定为使用 NEAREST 算法。



图 25-9　缩放后的图像

在实际应用中，经常需要得到一个图像的缩略图。PIL 中提供了一个实用函数 thumbnail 来完成这项功能。thumbnail 方法总是会得到一个不超过图像自身尺寸大小的缩略图。其参数值和 resize 方法中是一样的。下面是 thumbnail 的操作示例：

```
In [44]: im1.thumbnail((200,200))
```

需要注意的是，thumbnail 方法操作的是源图像。也就是说，此方法的操作对象是原始图像，将原始图像变换成缩略图大小。如果不希望在原来图像上修改，则可以使用 copy 方法先获取原

更多资源请访问稀酷客(www.ckook.com)

始图像的一个拷贝。

2. 图像的旋转

PIL 库中提供了 rotate 方法来实现对图像的旋转。其参数为需要旋转的角度。正数为逆时针方向，负数为顺时针方向。

```
In [45]: im1 = Image.open("Lenna.png")

In [46]: im9 = im1.rotate(-45)

In [47]: im9.show()
```

- 由于在 In［44］中 im1 图像对象被改变，所以在 In［45］中重新加载了此图像文件。
- 在 In［46］中使用 Image 类的 rotate 方法将源图像顺时针旋转了 45°。
- 在 In［47］中调用 show 方法显示此图像。

图 25-10 所示是源图像顺时针方向旋转 45°后的效果。



图 25-10　旋转后的图像

在图 25-10 中可以看到，对于没有图像数据的部分，PIL 使用了黑色来补全。另外，由于原图像的大小不能满足旋转后的要求，所以对图像进行了裁剪。在 rotate 方法中提供了一个可选的参数 expand，用来决定旋转后的图像大小。当该参数的值为“真”的时候，将会扩展图像尺寸用来容纳整个图像。在默认情况下，此值为“假”，表示保持源图像的大小，如图 25-10 中显示的那样。

因为将图像旋转 90°的倍数的情况是最多的，所以 PIL 库中提供了 transpose 方法来突出这种功能。其参数为一种常用的动作。除了旋转功能之外，transpose 方法还支持图像的翻转操作。表 25-1 中列出了 transpose 方法所支持的动作类型。

表 25-1　transpose 方法支持的动作类型

动作类型	说　明
FLIP_LEFT_RIGHT	左右翻转
FLIP_TOP_BOTTOM	上下翻转
ROTATE_90	逆时针旋转 90°
ROTATE_180	逆时针旋转 180°
ROTATE_270	逆时针旋转 270°

【例 25-8】演示对前面裁剪的图像进行左右翻转和旋转 180°的操作。

```
In [48]: im9 = im1.transpose(Image.FLIP_LEFT_RIGHT)

In [49]: im9.show()

In [50]: im10 = im1.transpose(Image.ROTATE_180)

In [51]: im10.show ()
```

- 这里的操作和前面其他图像变换的操作方法类似。在 In［48，50］中对原图使用了左右翻转的操作和旋转 180°的操作。并在随后通过 show 方法显示图像。

图 25-11 和图 25-12 中显示了变换后的图像。

更多资源请访问稀酷客(www.ckook.com)



图 25-11　左右翻转后的图像



图 25-12　旋转 180°后的图像

一种更一般的图像变换方法是使用 transform 函数。此函数中提供了丰富的操作方法和参数来支持图像的不同变换操作。

25.3　图像处理的高级应用

Image 类是 PIL 库所提供的最基本的图像操作类。实际上，在 PIL 中还有很多针对图像的高级处理类，如 ImageChops 类、ImageEnhance 类和 ImageFilter 类等。本节将介绍这些类中的方法如何作用于图像，从而形成特殊的效果。

25.3.1　图像的通道操作

本小节中介绍的操作都是定义在 PIL 库的 ImageChops 模块中。此模块中包含对通道的相关操作。这些方法有多种用途，如实现特殊的效果、图像复合等。在 PIL 当前的发行版中，针对通道的操作还只是支持 8 位图像，如“L”和“RGB”颜色模式。ImageChops 模块中的大部分操作都使用一个或两个图像对象作为参数，并返回一个图像对象。

在 ImageChops 模块提供的方法中，有些和 Image 类中方法的功能是一样的。如 ImageChops 类中的 duplicate 方法和 Image 类中的 copy 方法功能是相同的，都是返回一个给定图像的拷贝。另外一个是 ImageChops 类中的 composite 方法和 Image 类中 composite 方法，都可以实现复合图像的功能。除此之外，ImageChops 类中提供了一些实用函数来处理图像。

通过使用 constant 方法可以返回一个填满特定像素的图像：

```
In [52]: import PIL.ImageChops as IC

In [53]: im11 = Image.new("RGB",(300,300))

In [54]: im12 = IC.constant(im11, 100)
```

- 在 In［52］中从 PIL 库中导入了 ImageChops 模块，并使用别名 IC。
- 在 In［53］中使用 Image 的 new 方法生成了一个长和宽都为 300 像素的 RGB 图像 im11。
- 在 In［54］中调用 ImageChops 模块中的 constant 方法，把 im11 填满了像素值 100 而生成了一个单一像素的图像。

图 25-13 中显示了填满单一像素的图像。

ImageChops 模块中提供了 invert 方法用来实现对图像的反色处理。这种处理后的图像类似于胶卷相片中的底片效果。如下面的操作示例：

更多资源请访问稀酷客(www.ckook.com)

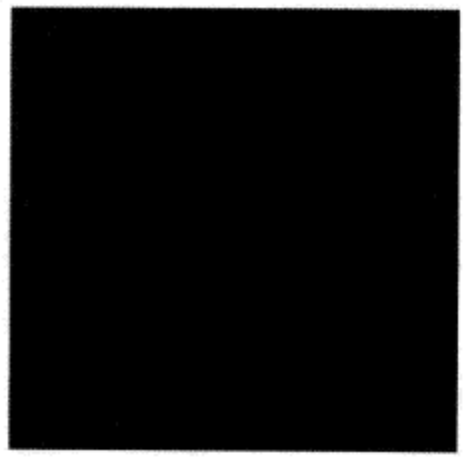


图 25-13　填满单一像素的图像

```
In [55]: im13 = IC.invert(im1)

In [56]: im13.show()
```

- In［55］中调用 invert 函数对 im1 图像对象进行了反色处理。随后调用 show 方法将其显示出来。

图 25-14 中显示了处理后的反色图像。



图 25-14　反色后的图像

ImageChops 模块中提供了大量的图像混合模式。使用适当的图像混合模式可以实现各种特殊的效果。这些混合模式方法都有两个需要混合的图像对象，并将生成的图像对象返回。下面是一些常见的混合模式。

（1）lighter。亮化模式，返回两个图像对象中更亮的像素点。

（2）darker。暗化模式，返回两个图像对象中更暗的像素点。

（3）difference。差值模式，返回两个图像的差值对象。两个对象的像素点之间的明暗度越相近，混合后的颜色越暗；两个对象的颜色差别越大，生成的图像越明亮。

（4）multiply。正片叠底模式。在图像混合的时候，如果混合的图像为黑色，最终的输出结果也为黑色；如果和白色混合，则图像不受影响。

（5）screen。屏幕模式。这种模式的操作过程与正片叠底模式相反。最终的混合结果是，图像中白色的部分还是为白色，黑色的部分则不会对源图像有影响。

（6）add。相加模式，将原始图像和混合图像的对应像素相加。

（7）subtract。相减模式，将原始图像和混合图像的对应像素相减。

【例 25-9】 演示如何使用图像的混合模式。

```
In [57]: im14 = Image.open("scene.png")

In [58]: IC.lighter(im1,im14).show()

In [59]: IC.darker(im1,im14).show()
```

更多资源请访问稀酷客(www.ckook.com)

```
In [60]: IC.difference (im1, im14).show ()

In [61]: IC.multiply (im1, im14).show ()

In [62]: IC.screen (im1, im14).show ()

In [63]: IC.add (im1, im14).show ()

In [64]: IC.subtract (im1, im14).show ()
```

- 在 In［57］中使用 Image 的 open 方法加载了一幅图像作为混合使用的对象。
- 在 In［58-64］中，分别使用上面所介绍的 7 种混合模式对源图像和混合图像进行了混合。在各个语句中，直接使用了 show 方法来显示图像。

图 25-15 中显示了使用这几种混合模式处理后的图像。



a)亮化模式



b)暗化模式



c)差值模式



d)正片叠底模式



e)屏幕模式



f)相加模式



g)相减模式

图 25-15　不同混合模式处理后的图像

更多资源请访问稀酷客(www.ckook.com)

在实际操作中，混合图像一般是具有某种特征的遮罩层。这样在进行混合处理的时候，就可以实现对原图的特殊效果。

25.3.2　图像的增强处理

PIL 库中提供了 ImageEnhance 模块用来实现对图像的增强处理。包括调整图像的色彩平衡、明亮度、对比度和锐度等。

在 ImageEnhance 模块中的增强类中都有一个公用接口，这就是 enhance 方法。其中有一个参数 factor。一般地，当其值为 1.0 的时候，表示保持源图像的数据；当其值比较小的时候，表示更少的色彩，如明亮度或饱和度；当其值比较大的时候，表示更多的色彩。

1. 调整图像的色彩平衡

通过使用 Color 类可以调整图像的色彩平衡，类似于彩色电视的色彩控制。当因子为 1.0 的时候为源图像。当因子为 0.0 的时候，表示图像为黑白图像。

【例 25-10】演示调整图像色彩平衡的方法。

```
In [65]: from PIL import ImageEnhance as IE

In [66]: en = IE.Color(im1)

In [67]: en.enhance(0.0).show()

In [68]: en.enhance(0.4).show()

In [69]: en.enhance(0.8).show()
```

- 在 In［65］中，从 PIL 中导入了 ImageEnhance 模块，并将其别名为 IE。
- 在 In［66］中使用了 Color 类，并将 im1 作为其参数。
- 在 In［67，68，69］中，分别显示了因子为“0.0”、“0.4”和“0.8”的图像。

图 25-16 中显示了这 3 幅不同色彩平衡下的图像。



a) 因子值：0.0　　b) 因子值：0.4　　c) 因子值:0.8

图 25-16　不同色彩平衡下的图像

2. 调整图像的明亮度

通过使用 Brightness 类可以调整图像的明亮度。当因子为 1.0 的时候为源图像，因子为 0.0 的时候表示图像为黑色图像。

【例 25-11】演示调整图像的明亮度的方法。

```
In [70]: en = IE.Brightness(im1)

In [71]: en.enhance(0.1).show()
```

更多资源请访问稀酷客(www.ckook.com)

```
In [72]: en.enhance (0.4) .show ()

In [73]: en.enhance (0.7) .show ()
```

- 【例 25-11】的代码操作和【例 25-10】的不同仅仅在于使用了 Brightness 类。其明亮度因子分别使用了“0.1”、“0.4”和“0.7”。

图 25-17 中显示了这 3 幅不同明亮度下的图像。



a) 因子值:0.1　　b) 因子值：0.4　　c) 因子值:0.7

图 25-17　不同明亮度下的图像

3. 调整图像的对比度

通过使用 Contrast 类可以调整图像的对比度。当因子为 1.0 的时候为源图像，当因子为 0.0 的时候表示图像为白色图像。

【例 25-12】演示调整图像的对比度的方法。

```
In [74]: en = IE.Contrast (im1)

In [75]: en.enhance(0.1).show()

In [76]: en.enhance(0.4).show()

In [77]: en.enhance(0.7).show()
```

- 【例 25-12】和【例 25-10】的 Color 类操作中的不同在于使用了 Contrast 类。这里对比度因子同样使用了“0.1”、“0.4”和“0.7”。

图 25-18 中显示了这 3 幅不同对比度下的图像。



a) 因子值:0.1　　b) 因子值：0.4　　c) 因子值：0.7

图 25-18　不同对比度下的图像

4. 调整图像的锐度

通过使用 Sharpness 类可以调整图像的锐度。当因子为 1.0 的时候为源图像；当因子为 0.0

更多资源请访问稀酷客(www.ckook.com)

的时候为模糊的图像；当为 2.0 的时候为锐化的图像。

【例 25-13】演示调整图像锐度的方法。

```
In [78]: en = IE. Sharpness (im1)

In [79]: en. enhance(-30.0). show()

In [80]: en. enhance(0). show()

In [81]: en. enhance(30.0). show()
```

- 这里的代码操作和【例 25-10】的 Color 类操作中的不同在于使用了 Sharpness 类。为了强调对比，这里的锐度因子使用了“ - 30”、“0” 和 “30”。

图 25-19 中显示了这 3 幅不同锐度的图像。



a) 因子值:-30.0　　b) 因子值：0.0　　c) 因子值：30.0

图 25-19　不同锐度的图像

25.3.3　PIL 中的内置滤镜

在 PIL 库中内置了不少滤镜，可以用来实现各种特殊的效果。其中比较常用的滤镜有模糊、轮廓、浮雕和边缘增强等。这些都是定义在 ImageFilter 模块之中的。在实际使用的时候，使用 Image 类中的 filter 方法可以实现滤镜功能。

【例 25-14】演示滤镜的使用方法。

```
In [82]: from PIL import ImageFilter as IF

In [83]: im15 = im1. filter(IF. BLUR)

In [84]: im15. show()

In [85]: im16 = im1. filter(IF. CONTOUR)

In [86]: im16. show()

In [87]: im17 = im1. filter(IF. DETAIL)

In [88]: = im17. show()

In [89]: im18 = im1. filter(IF. EDGE_ENHANCE)

In [90]: im18. show()

In [91]: im19 = im1. filter(IF. EMBOSS)

In [92]: im19. show()

In [93]: im20 = im1. filter(IF. SHARPEN)
```

更多资源请访问稀酷客(www.ckook.com)

```
In [94]: im20.show()
```

- 在 In［82］中，从 PIL 库中导入了 ImageFilter 模块，并将其别名为 IF。
- 在 In［83-94］中使用了 ImageFilter 模块中的 6 个常用滤镜，并在随后调用 show 方法显示。

图 25-20 中显示了使用滤镜后的图像。



a) BLUR　　b) CONTOUR　　c) DETAIL



d) EDGE_ENHANCE　　e) EMBOSS　　f) SHARPEN

图 25-20　PIL 中滤镜的使用效果

在 PIL 库中，还有更多的模块用来处理图像。例如可以通过使用 ImageDraw 模块在图像上作图或者写字。使用 PIL 库不但可以实现对于图像的常规处理，还可以通过使用滤镜和混合模式等功能来使得图像具有特殊的效果。

25.4　使用 PyGame 进行游戏开发

随着多媒体的发展，越来越多的游戏被开发出来。不但可以利用 Python 简单易用的特点快速开发游戏，而且 Python 中有大量绑定的游戏开发库可以辅助开发。本节将介绍 Python 语言中优秀的游戏开发包 PyGame。

25.4.1　Python 游戏开发包：PyGame

PyGame 是一个基于 SDL（Simple Direct Media Layer，简易直控媒体层）的游戏开发环境。SDL 是一个可以运行在多个系统平台下的非常成熟的图形和多媒体库。由于构建 Pygame 的两大基石 Python 和 SDL 都具有良好的跨平台性，所以 Pygame 也可以运行在多个系统平台下。通过使用 Pygame 库，可以有效的处理用户的输入输出和控制多媒体资源。

虽然使用脚本语言来构建游戏被很多人感到不解，认为这样的实现效率会很低，但是使用

更多资源请访问稀酷客(www.ckook.com)

Pygame 软件包的好处在于可以在速度和效率之间找到一个好的平衡。除开那些特大型的游戏，一般规模的游戏开发使用 Pygame 不会存在效率问题。在 Pygame 的项目主页上有很多使用 Pygame 开发的游戏，有些游戏的性能还是不错的。

Pygame 的项目主页是 http：//www. pygame. org，可以从项目主页中下载源文件来安装。但是由于此软件包的依赖性，所以还是推荐安装预制好的安装包。在 Ubuntu 或者 Debian 等操作系统下，可以使用 apt-get 来安装。在 Windows 操作系统下，推荐下载可执行软件包。图 25-21 中显示了可执行文件安装时候的一个截图。

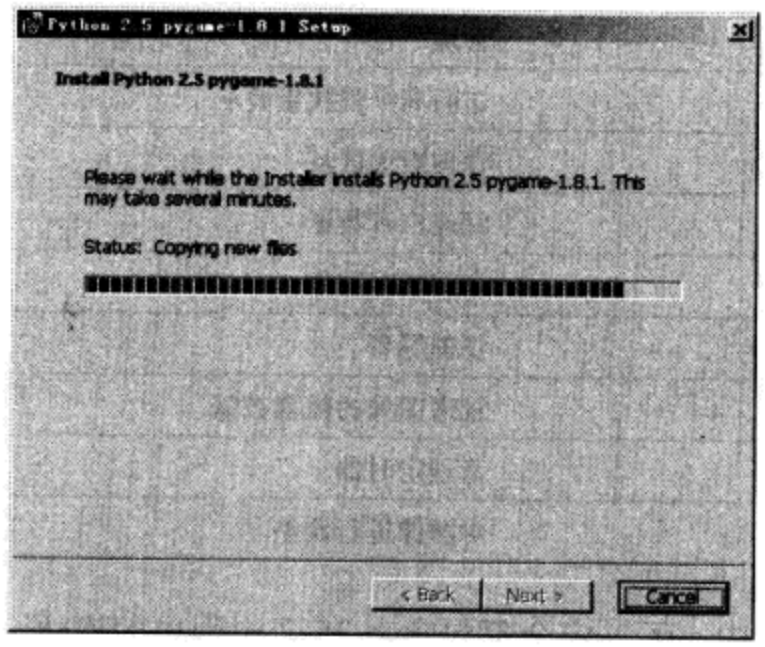


图 25-21　Pygame 软件包的安装

安装完毕后，可以通过下面的操作来检测 Pygame 软件包是否已经正确安装：

```
In [1]: import pygame

In [2]: pygame.ver
Out[2]: '1.8.1release'
```

- 在 In［1］中导入了 pygame 模块。如果安装不成功，导入过程将会出错。
- 在 In［2］中输出了 Pygame 的版本号。当前最新的版本号为“1. 8. 1release”。

实际上，Pygame 软件包中有多个模块，它们都统一放在 pygame 的名字空间下。这些模块中，有些模块在 pygame 中是很基本和常用的。例如由于每个游戏中都会包含一些界面，因此会用到 display 模块。而如果需要处理用户的输入，则需要使用 keyboard、joystick 或者 mouse 模块。其他的一些模块虽然使用的机会比较少，但是也是必不可少的一部分。表 25-2 中列出了 pygame 软件包中的模块。

表 25-2　pygame 软件包中的模块

模块名称	说　明
cdrom	控制 CD 驱动器
cursors	加载鼠标指针文件
display	窗口，用户的访问界面
draw	画图，包括点线等形状
event	管理事件
font	使用系统字体
image	加载和保存图像

更多资源请访问稀酷客(www.ckook.com)

（续）

模块名称	说　明
joystick	使用 joystick 或者类似的设备
key	从键盘中读取键值
mixer	加载和播放声音
mouse	管理鼠标
movie	播放视频文件
music	管理声音
overlay	访问高级视频重叠层
rect	管理矩形区域
sndarray	管理声音数据
sprite	管理动态图像
surface	管理屏幕
surfarray	管理图像的像素数据
time	管理定时器
transform	对图像进行调整

在 pygame 的安装套件中，不一定全部包含表 25-2 中列出的模块，可以使用下面的方法来判断是否有特定的模块：

```
if pygame.joystick is None:
    print "joystick module is not available"
else:
    print "joystick module is available"
```

25.4.2　创建基本的 Pygame 窗口

使用 pygame 创建的游戏由一定的框架构成，具体的方法如【例 25-15】所示。

【例 25-15】演示一个最基本的 Pygame 游戏的构成，并包含最简单的事件处理。

```
#filename: pygame_1.py
#encoding = utf-8

import pygame
from pygame.locals import *
from sys import exit

background_image_filename = 'background.png'
mouse_image_filename = 'mouse.png'

#Pygame 初始化
pygame.init()

screen = pygame.display.set_mode((512, 512), 0, 32)
pygame.display.set_caption("Hello, World!")

#加载图片
background = pygame.image.load(background_image_filename).convert()
mouse_cursor = pygame.image.load(mouse_image_filename).convert_alpha()

#进入 Pygame 循环
```

更多资源请访问稀酷客(www.ckook.com)

```
while True:

    for event in pygame.event.get():
        if event.type == QUIT:
            exit()

    screen.blit(background, (0,0))

    x, y = pygame.mouse.get_pos()

    x-= mouse_cursor.get_width() / 2
    y-= mouse_cursor.get_height() / 2

    screen.blit(mouse_cursor, (x, y))

    pygame.display.update()
```

- 在最上面的代码中，首先导入了 pygame。这使得可以访问 pygame 名字空间下的所有子模块，包括 display、event 等。接下来的语句从 pygame 模块中的 locals 中导入了所有的符号。在 locals 中定义了用在 pygame 中的各种常量。
- 接下来赋值了两个变量，它们分别表示背景图片和鼠标指针图片。
- 接下来调用了 pygame 的 init 方法。此方法将会对所有的子模块进行初始化。pygame 库将会检查包括驱动器在内的设备是否已经准备好。为了加快加载的速度，可以只使用子模块的初始化方法，如 pygame.key.init()。但是，由于初始化只需要做一次，而这种分别进行初始化的方式也不会加快速度，所以还是推荐使用对 pygame 进行全局初始化。
- 接下来使用 display 模块的 set_mode 方法生成了一个 512×512 的区域。并将其作为游戏中绘制的窗口。随后调用 set_capion 方法设置了窗口的标题为“Hello，World!”。set_mode 方法中的第二个参数为显示的标志。这里设置为“0”，表示不需要对屏幕进行任何设置。实际上，此参数支持如表 25-3 所示的标志。

表 25-3　display 所支持的标志

标　志	说　明
FULLSCREEN	全屏窗口
DOUBLEBUF	双缓冲的显示
HWSURFACE	硬件加速的显示
OPENGL	使用 OpenGL 渲染的显示
RESIZABLE	使用可调整区域的显示
NOFRAME	使用无框架的显示

- 接下来的两个语句调用了 Image 模块中的 load 方法，对两幅图片进行了加载。
- 接下来进入了 Pygame 的事件循环。这里将对事件信息进行判断。如果为 QUIT 的话，则调用 exit 退出整个程序。
- 接下来使用了 screen 中的 blit 方法，将背景图片放置在窗口中。
- 接下来的 4 个语句是设置鼠标指针图片所在的位置。由于需要将鼠标指针放在鼠标图片的正中间，所以做了这样的处理。
- 最后调用了 display 模块的 update 方法。这可以使得鼠标指针在移动的时候，用户不会觉得指针图片非常地抖动。

运行这段代码，将显示如图 25-22 的界面。

更多资源请访问稀酷客(www.ckook.com)

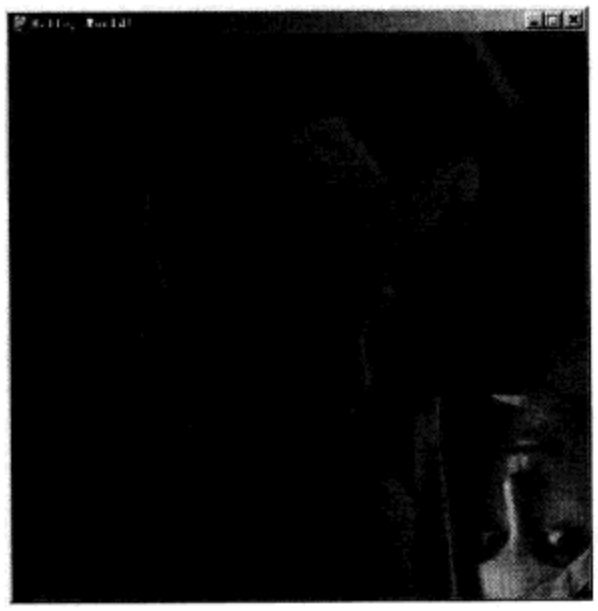


图 25-22　Pygame 的基本窗口显示

在图 25-22 所示窗口中，使用了前面图像处理的部分图像。其中右下角的为鼠标指针图片。

25.4.3　事件系统

在上面的实例代码中，并没有包含更多对事件的处理。事件的来源可能来自于键盘、鼠标或者是其他的外部设备。

【例 25-16】接上例。中在【例 25-15】代码加入了键盘的事件处理。

```
#filename: pygame_1.py
#encoding = utf-8

import pygame
from pygame.locals import *
from sys import exit

background_image_filename = 'background.png'
mouse_image_filename = 'mouse.png'

#Pygame 初始化
pygame.init()

#省略部分代码

#进入 Pygame 循环
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            exit()
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                exit()

#省略部分代码

    pygame.display.update()
```

- 【例 25-16】代码和【例 25-15】的主要区别是，这里的事件判断中加入了对键盘事件的判断。当事件类型为 KEYDOWN（也就是当键盘按下）的时候，再判断按下的键值，如果按下的为 ESC 键，则退出程序。

在 Pygame 软件包中，支持表 25-4 中的事件类型。

更多资源请访问稀酷客(www.ckook.com)

表 25-4　Pygame 支持的事件类型

事件类型	说　明
QUIT	用户单击了关闭按钮
ACTIVEEVENT	Pygame 的窗口显示或者隐藏
KEYDOWN	键被按下
KEYUP	键被释放
MOUSEMOTION	鼠标移动
MOUSEBUTTONDOWN	鼠标按下
MOUSEBUTTONUP	鼠标被释放
JOYAXISMOTION	joystick 移动
JOYBALLMOTION	joystick 球移动
JOYHATMOTION	joystick 帽移动
JOYBUTTONDOWN	joystick 被按下
JOYBUTTONUP	joystick 被释放
VIDEORESIZE	窗口调整
VIDEOEXPOSE	窗口重绘
USEREVENT	用户自定义事件

在【例 25-16】的循环中加入 print 语句，打印出所有的事件信息。可能的运行结果如下：

```
In [1]: run pygame_1.py
<Event(17-VideoExpose {})>
<Event(1-ActiveEvent {'state': 1, 'gain': 0})>
<Event(1-ActiveEvent {'state': 1, 'gain': 1})>
<Event(4- MouseMotion {'buttons': (0, 0, 0), 'pos': (412, 0), 'rel': (412, 0)})>
<Event(1-ActiveEvent {'state': 1, 'gain': 0})>
<Event(2-KeyDown {'scancode': 1, 'key': 27, 'unicode': u'\x1b', 'mod': 0})>
```

在上面的输出结果中可以看到 Pygame 框架接收了多少事件信息，包括其类型。对于有些事件类型，还有更多的参数，如在 MouseMotion 事件类型中就包含鼠标的位置。而在 KeyDown 事件类型中，则有被按下的键值。

25.5　小结

本章介绍了 Python 下的图像处理和游戏开发功能。在图像处理部分重点介绍了 PIL 模块，通过实例讲解了 PIL 在基本图像处理和高级处理上的应用。这些处理包括对图像的变换以及对图像的增强。最后介绍了游戏开发软件包 Pygame。内容包括如何构建一个游戏界面，并使用事件来实现对游戏的控制。

更多资源请访问稀酷客(www.ckook.com)

第 26 章　Python 语言的扩展与嵌入

虽然 Python 语言可以完成很多的任务，但是对于某些领域还是有一些不足，此时需要在 Python 中调用其他语言中已有的代码。另外，有些应用程序希望在其中增加对 Python 脚本语言的支持，从而达到使用插件的目的。这两种情况下都需要对原 Python 进行扩展。这种不同语言的融合可以充分发挥各自的特长。本章的知识点：

- 为 Python 编写 C 扩展模块
- Python 语言的嵌入
- 在 C 中使用 Python 模块

26.1　Python 语言的扩展

Python 语言良好的扩展性使得其可以使用其他语言书写的模块。这种扩展性也是 Python 语言流行的一个原因。这种扩展方式使得其他语言的开发人员可以使用自己喜欢的语言来开发可供 Python 语言使用的模块。

26.1.1　Python 扩展简介

在为 Python 语言书写模块的时候，Python 开发人员可以使用 Python 语言，也可以使用熟悉的其他语言。例如，使用 C 或者C++ 这些编译语言来书写 Python 模块；Java 的开发者可以使用 Jython；C#语言的开发者可以使用 IronPython。在本节中，将仅仅考虑 C 和C++ 的扩展。

Python 是一种脚本语言，却有着和其他编译语言一样的强大功能。其中的一个原因就是 Python 的设计使得开发者可以使用 C 或者C++ 对其进行扩展。使用 C/C++ 来书写 Python 模块，一方面可以利用 Python 语言的灵活性和丰富功能，另一方面可以利用 C/C++ 的运行性能。这种处理方式使得 Python 语言也具备了类似 C 和C++ 的运行速度，从而解决了脚本语言运行速度缓慢的问题。经过这样的处理后，Python 语言的使用范围进一步得到了扩展。

在 Python 中的部分模块就是使用 C 语言实现的，如 math 模块。这样做的目的也是考虑到了运行效率。而标准库中的绝大部分库，都是使用 Python 语言实现的。这些文件可以在安装目录下的 Lib 目录下看到。由于 Python 语言本身是使用 C 编写的，所以有类似 C 的堆栈段。这也使得可以直接使用 C 或者是具有“extern C”声明的C++ 语言。对于采用兼容 C 语言的 Python 扩展，一般有两类。一类是 C 模块，这种处理方式和 Python 模块是类似的。另一种是 C 类型，其处理方式就像 Python 中内置的类型一样，包括列表等。

除了能够为 Python 提供其核心语言所不具有的功能和提高系统性能外，从另一个角度说还可以对源码进行保护。这在 Python 的商业化使用中是非常必要的。因为 Python 作为解释型的脚本语言，本身对于源代码的保护能力不强。将核心代码使用其他语言来书写，并通过一定的方式链接到 Python 中，这有助于私有代码的保护。

在 Python 中使用 C/C++ 加入内置的模块是非常简单的。在 Python 的应用编程接口（Appli-

更多资源请访问稀酷客(www.ckook.com)

cation Programmers Interface，API）中定义了一组函数、宏以及变量，用来访问 Python 语言的运行环境。这些 Python 的 API 接口被组合到了一个 C 源文件中，并且通过 Python. h 文件来引用。

26. 1. 2　一个 C 扩展的例子

在 Python 语言中编写 C 扩展一般包括下面几个部分。

（1）通过 C 语言实现一个 Python 接口。

（2）注册此访问表。

（3）加入模块初始化代码。

（4）对代码进行编译和测试。

在这几个部分中，最后一个步骤是单独的，而前面三个部分都是可以在代码中体现出来的。

【例 26-1】演示 Python 扩展接口部分的使用方法。

```
/* fielname: extend. c* /
#include < Python. h >

static PyObject *
extend_system(PyObject * self, PyObject * args)
{
    const char * command;
    int sts;

    if (! PyArg_ParseTuple(args, "s", &command))
        return NULL;

sts = system(command);
    return Py_BuildValue("i", sts);
}
```

- 在代码片段的最前面，通过 include 处理指令将“Python. h”文件包含了进来。一种推荐的方法是将此导入语句放在所有其他包含的头文件之前。这样可以在某些系统下避免可能的冲突。
- 所有包含在 Python. h 文件中的符号都是以“Py”或者“PY”开头的，除了那些在标准头文件中定义的符号。这些符号包括函数、宏和变量等。为了使用方便，在 Python. h 文件中实际上已经有一些常用的头文件。例如 < stdio. h > 、< string. h > 、< errno. h > 和 < stdlib. h > 等。
- 在函数的主体部分实现了一个 extend_system 方法。注意到，此方法被定义成了 static（静态的）。此函数的返回值为一个 PyObject 对象指针，此值将在后面被调用。此方法有两个参数，一个为 self，另一个为 args。当被实现为一个函数的时候，此时的 self 值永远都会是 NULL，这样输出的参数都将会传输到 args 变量中来。
- args 变量实际上是一个指向所有输入参数的元组。这表明输入的参数将不能被修改。由于现在传递进来的数据都还是 Python 对象，为了能够在 C 语言中使用，必须对其进行相应的数据类型转换。
- 在定义了两个变量后，Python API 中的函数 PyArg_ ParseTuple 用来检查输入参数并将其转换为 C 语言中的值。此函数将使用一个模板字符串来判断所需的类型，以及所要保存的 C 对象的类型。这里的 s 表示字符串类型。PyArg_ParseTuple 方法在所有要求都满足的情况下将返回 True。这些要求包括参数都有正确的类型，并且其值被保存到了合适的 C 语言变量中。而当其中任何一个条件不满足的时候（例如有非法的参数传递进

更多资源请访问稀酷客(www.ckook.com)

来），此方法都会返回 False。在后面失败的情况下，也会触发一个异常，同时此函数马上返回 NULL 值。

- 接下来的语句是一个 system 的系统函数调用。此方法中输入的值即是前面从 Python 输入参数中解析出来的值。更具体地说是通过 PyArg_ParseTuple 函数来实现的。此函数得到了一个返回值，赋值给 sts 变量。
- 由于此 extend_system 函数的返回值为 PyObject 对象，所以不能将 sts 变量直接返回。这里，通过使用 Py_BuildValue 方法，可以将 sts 值转换为 PyObject 类型的变量。从功能上来看，Py_BuildValue 函数和 PyArg_ParseTuple 函数是对应的，但是两者的处理方式却几乎一样，也采用了格式化的字符串来实现数据的转换。在这里，其格式化字符串为 i，表示这里将返回整型数据。需要注意的是，这里的整型是指 Python 语言中的整型。

如果 C 语言中的函数不需要任何的返回值（也就是 void 的函数），在这里同样的需要返回一个值。在 Python 核心语言中，与 C 语言中的 void 相对应的是 None，所以需要返回一个值为 None 的 Python 对象。使用下面的方法可以达到这个目的：

```
Py_INCREF(Py_None);
return Py_None;
```

这里实际上是通过 Py_RETURN_NONE 宏来实现的。

Py_None 是 C 语言中一个用来表示 Python 中 None 的符号。这和 C 语言中的 NULL 指针是不一样的。很多情况下，NULL 表示一种错误。而 None 则是一个合法的值。

26.1.3　模块方法表和初始化函数

在定义了 Python 扩展的 C 语言接口后，再加上 Python 的模块方法表和初始化函数即可成为一个完整的 Python 扩展。

【例 26-2】接上例。为【例 26-1】定义模块方法表。

```
static PyMethodDef ExampleMethods[] = {
    ...
    {"system",  example_system, METH_VARARGS,
     "Execute a shell command."},
    ...
{NULL, NULL, 0, NULL}

};
```

- 这里定义了一个 ExampleMethods 变量。这实际上也是 C 语言扩展中提供的可以在 Python 终端中访问的方法。注意到此变量也定义成了 static（静态）。
- 在这个变量中定义了多个成员。其中每个成员有 4 个元素。变量通过辨别成员是否使用 NULL 和 0 的元素来作为结束条件。
- 在这 4 个元素中，第 1 个元素为可以在 Python 中调用的方法，第 2 个元素为当前 C 语言扩展中的函数名称。最后一个元素为此方法接口的文档描述。
- 第 3 个元素用来告诉 Python 解释器在 C 函数中的调用惯例。一般来说，这个值应该为“METH_VARARGS”或者是“METH_VARARGS | METH_KEYWORDS”。后一种写法表示这两个标志位都被设置。如果此值为 0，表示使用了已经被废除的 PyArg_ParseTuple 旧版本中的方法。
- 当标志位中只有“METH_VARARGS”的时候，从 Python 解释器中传递过来的参数将会被 PyArg_ParseTuple 函数处理。而 METH_KEYWORDS 是用来设置字典参数的。在这种情况下，C 函数应该可以接受一个附加的“PyObject *”参数，其中有一个字典类

更多资源请访问稀酷客(www.ckook.com)

型。在这个时候，可以使用 PyArg_ParseTupleAndKeywords 方法来解析参数。

上面定义的模块方法表将在模块初始化函数中被传递给解释器。模块初始化函数必须要使用 initNAME( ) 函数的形式，其中 NAME 为模块的名字。这应该是模块文件中唯一不是 static 的方法。

【例 26-3】接上例。演示将【例 26-1】的模块初始化。

```
PyMODINIT_FUNC
initextend (void)
{
    (void) Py_InitModule("extend", ExtendMethods);
}
```

- 在上面可以看到，此函数是此模块文件中唯一没有设置为 static 的方法。
- 在定义的 initextend 方法中，使用了 PyMODINIT_FUNC 方法。为此函数定义了 void 的返回值。这种处理方法可以使得其在其他语言下也能够正确的使用。例如在C++ 环境中就会加上“extern ‘c’”的代码。

26.1.4　编译和测试

在前面已经介绍了对 Python 进行扩展的前三个部分：C 接口、模块方法表和初始化函数。

【例 26-4】接上例。显示【例 26-1】Python 扩展的完整代码：

```
#C 接口
static PyObject *
extend_system(PyObject * self, PyObject * args)
{
    const char * command;
    int sts;

    if (! PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}

#模块方法表
static PyMethodDef ExampleMethods[] = {
    ...
    {"system",  example_system, METH_VARARGS,
     "Execute a shell command."},
{NULL, NULL, 0, NULL}
};

#初始化函数
PyMODINIT_FUNC
initextend (void){
    (void) Py_InitModule("extend", ExtendMethods);
}
```

上面分别介绍了实现 Python 扩展的几个组成部分。其中的每个部分都有各自的用途和特点，缺一不可。

为了能够对其进行测试，还需要将其编译成 Python 模块并运行。

编译 Python 模块，首先需要构建 Python 开发环境。在 Linux 系统下，直接安装如python2.5-dev 的软件包即可。在 Windows 系统环境下则相对比较麻烦，需要设置正确的编译器。为了简单起见，下面对 Python 扩展模块的测试是在 Linux 下完成的。将此 extend.c 文件拷贝到 Linux 系统下的某个目录中。

更多资源请访问稀酷客(www.ckook.com)

1. 用编译器直接编译

一种可行的方式是直接使用编译器将其编译成相应的模块。在当前目录中新建一个 Makefile 文件。其文件内容如下：

```
PYI = /usr/include/python2.5

extend.so: extend.c
	gcc extend.c -g -I$(PYI) -fpic -shared -o extend.so

clean:
	rm -f extend.so core
```

- 在此 Makefile 文件中，首先定义一个变量 PYI，指定了当前 Python 开发头文件所在的目录。
- 随后，在文件中指定了两个目标，第一个是 extend.so，其依赖的文件为 extend.c。具体的执行步骤是下面的这条命令。第二个是一个清理目标，将生成的文件删除。

使用 make 命令，即可以对指定的目标进行编译：

```
book@6[extern]$ make
gcc extend.c -g -I/usr/include/python2.5 -fpic -shared -o extend.so
```

上面的输出结果显示了具体的命令执行过程。这样，将在当前目录下生成 extend.so 文件。此文件可以被 Python 解析器加载。

2. 用 distutils 模块生成

另外一种方式是 Python 所提供的——使用 distutils 模块来生成。在当前目录下创建一个 setup.py 文件，其文件内容如下：

```
#! /usr/bin/python

from distutils.core import setup, Extension

module_extend = Extension('extend',[extend.c'])

setup ( name = 'Extend', version = '1.0',
description = "This is a Python extend Demo",
ext_modules = [module_extend])
```

- 这是 Python 为其扩展所设计的一套完整的方案。这里，首先从 distutils.core 中导入了 setup 方法和 Extension 类。
- 接着，使用 Extension 类的构造函数生成了一个扩展模块。这里的模块名为 extend，具体的源文件为 extend.c。
- 最后调用 setup 方法。其中的几个参数分别指定了名字、版本和描述等。最重要的是在 ext_modules 属性中设置了刚刚生成的扩展模块。

通过上面的设置，此模块的编译环境就完成了。

运行下面的命令，即可以生成相应的 so 模块：

```
python setup.py build
```

运行完毕后，在当前目录的 build/temp.linux-i686-2.5/目录下会看到生成的extend.pyd 文件。此模块可以被 Python 的解释器调用。

在当前目录中启动 IPython 解释器，即可以完成对此 Python 扩展的测试：

更多资源请访问稀酷客(www.ckook.com)

```
book@ 6[extern] $ ipython
Python 2.5.2 (r252:60911, Nov 14 2008, 19:46:32)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ? object also works, ?? prints more.

In [1]: import extend

In [2]: extend.system('ls')
build  extend.c  extend.so  Makefile  setup.py

In [3]: extend.system('mkdir test')

In [4]: extend.system('ls')
build  extend.c  extend.so  Makefile  setup.py  test
```

- 在 In［1］中导入了 extend 模块。实际上，这里使用的是 extend.so 文件。当首次导入 extend 模块的时候，初始化函数将会被调用。
- 在 In［2］中调用了其中的 system 函数。其中传入了命令“ls”。通过在 C 函数中执行此命令，将会打印当前的文件夹。在输出结果中可以看到具体的信息。
- 在 In［3，4］中进一步对 system 的功能进行了测试。这里使用“mkdir test”命令生成一个 test 目录，然后调用 ls 命令显示了当前目录下的文件。可以看到，system 的命令得到了正确的执行。

26.2　Python 语言的嵌入

随着 Python 功能的越来越强大，很多的应用程序希望能够加入 Python 语言作为其插件的实现。包括 GIMP 等软件在内的应用都可以使用 Python 来书写特定的插件。本节将介绍如何将 Python 加入到其他语言的应用程序中。

26.2.1　Python 嵌入简介

在上一节中，介绍了如何使用其他的语言书写可供 Python 使用的模块，来对 Python 进行扩展。当然，同样可以使用 Python 来为 C 或者C++ 等语言提供支持。这种嵌入方式使得如 C 或者 C++ 等语言可以获得 Python 强大功能的支持，同时也不失 C 本身的效率。从这一点来说，Python 的嵌入和 Python 的扩展并没有本质的区别。

这种嵌入方式有很多用途。例如一个应用程序使用了其他的静态语言，但同时又希望能够书写一些 Python 脚本文件来实现一些实用工具。这种设计方式使得可以将核心功能使用如 C 或者 C++ 的语言来实现，但是一些外围的工作可以采用 Python 来实现。同时，这样还可以给用户提供 Python 接口。用户可以使用这些接口来完成原来应用程序并不提供的功能。如上面提到的 GIMP 软件，某个用户如果实现了一种图像处理算法，则可以首先使用 Python 语言的插件来对其进行验证。

在实际的实现过程中，虽然和 Python 的扩展过程很类似，但是还是有些不同。最大的不同在于使用 Python 扩展的时候，主要的执行程序是使用 Python 语言编写的。而对 Python 嵌入的方式来说，可能主程序和 Python 并没有什么关系，只有在使用到 Python 的时候才会起作用。

当将 Python 嵌入的时候，需要提供函数的主程序代码。在程序的主函数中，至少要完成的

更多资源请访问稀酷客(www.ckook.com)

工作包括初始化 Python 的解释器。这实际上是通过调用函数 Py_Initialize 来完成的。当然，其中还有一些可选的参数用来传递给 Python 的解释器。在初始化完成之后，应用程序中就可以调用此解释器对 Python 代码进行解析了。

在向 Python 的解释器传递参数的时候，可以有多种方式。最常见的方式是使用字符串，这可以通过 PyRun_SimpleString 方法来完成。或者是传递文件对象或者文件名，这可以通过调用 PyRun_SimpleFile 方法来完成。当然，实际上这里也可以使用在 Python 扩展中所介绍的方法来创建和使用 Python 对象。

26.2.2　一个 Python 嵌入的例子

将 Python 嵌入到其他应用程序中的最简单方式是使用高层的接口。这种接口可以在其他类型的文件中执行 Python 脚本。不过这种处理是不能和外界进行交互的。这种处理方式比较适合于在文件中执行某些简单的操作。

【例 26-5】演示高级接口的使用方法。

```
/* emb_1.c* /
#include <Python.h>

int main(int argc, char * argv[])
{
    Py_Initialize ();

    #解释执行参数中的语句
    PyRun_SimpleString("from time import sleep\n"
                       "print 'sleep 1s' \n"
                       "sleep(1) \n");

    Py_Finalize();

    return 0;
}
```

- 从【例 26-5】中可以看到，在最上面还是使用 include 处理指令加载了“Python.h”头文件。其中包含嵌入 Python 所需要用到的函数和变量。
- 在 main 函数中，首先调用 Py_Initialize 方法，对 Python 解释器进行初始化，初始化之后，即可以使用 Python 解释器来对 Python 代码进行解释。
- 在第 9 行中使用了 PyRun_SimpleString 方法。此方法主要是对简单的字符串进行解释处理执行。在这里，将要解释 Python 的 3 个语句。第一个语句是从 time 模块中导入 sleep 方法；第二个 Python 语句是向终端输出一个提示信息；最后一个 Python 语句用来休眠 1 秒钟。
- 在上面的代码执行完毕后，需要调用 Py_ Finalize 方法来销毁当前的 Python 解释环境。

将当前目录下的 Makefile 文件修改为如下的内容：

```
PYI = /usr/include/python2.5

emb_1: emb_1.c
        gcc emb_1.c -g -I $(PYI) -fpic -shared -o emb_1

clean:
        rm -f emb_1 core
```

- 和修改之前内容的不同主要在于这里用 emb_1.c 代替了 extend.c，并用目标 emb_1 代替了 extend.so。

更多资源请访问稀酷客(www.ckook.com)

执行这个生成的 emb_ 1，可以看到如下的结果：

```
book@ 6[extern] $ chmod +x emb_1

book@ 6[extern] $ ./emb_1

sleep 1s

book @ 6[extern] $
```

- 这里首先使用 chmod 命令，将生成的可执行程序加上可执行属性。
- 执行此文件，可以看到输出结果。另外，在输出后面一个提示符之前，等待了 1 秒钟。通过这些输出结果可以看到，此 C 文件正确地执行了 Python 代码片段。

26.2.3　更好的嵌入

在上面的代码中，仅仅只是能直接执行嵌入在程序中的 Python 代码。这在很多时候是不能满足需要的。在继承方式中，下面的代码可以实现解析外部提供的 Python 代码：

```
#filename: emb_2.c
#include <Python.h>

int main(int argc, char * argv[])
{
    PyObject * pName, * pModule, * pDict, * pFunc;
    PyObject * pArgs, * pValue;
    int i;

    #判断参数个素
    if (argc < 3) {
        fprintf(stderr,"Usage: call python_file function_name [args] \n");
        return 1;
    }
```

- 上面的代码功能主要是对输入的参数进行检查。输入的参数如果小于 3 个，则打印出提示信息：

```
    #初始化 Python 解释环境
    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /*  Error checking of pName left out * /

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);
```

- 首先使用 Py_Initialize 方法初始化了 Python 执行环境。
- 随后使用 PyString_FromString 从参数中获取模块的名字，并将其转化为 Python 中的 String 对象。
- 接着调用了 PyImport_Import 将此模块加载。
- Py_DECREF 的作用是手工对参考计数进行管理，从而实现了 Python 解释器对 Python 对象的内存回收。

```
if (pModule ! = NULL) {
#获取函数
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /*  pFunc is a new reference * /
```

更多资源请访问稀酷客(www.ckook.com)

```
        if (pFunc && PyCallable_Check(pFunc)) {
#对参数进行处理
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (! pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument \n");
                    return 1;
                }
                /* pValue reference stolen here: * /
                PyTuple_SetItem(pArgs, i, pValue);
            }
#调用 Python 中的函数
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue ! = NULL) {
                printf("Result of call: % ld\n", PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }
```

- 在这段代码中，首先获取了 Python 模块中的函数。随后调用 PyCallable_Check 方法对其进行检测，看是否可以被调用。
- 接着是对输入的参数进行判断，并对其进行类型转换。
- 最后，调用 PyObject_CallObject 方法来执行 Python 规范，并得到 Python 结果。

```
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n"); #调用函数失败
                return 1;
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"% s\"\n", argv[2]); #函数查找失败
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"% s\"\n", argv[1]); #Python 模块加载失败
        return 1;
    }
```

- 上面的代码主要是针对出错信息进行处理。

```
    Py_Finalize();
    return 0;
}
```

- 调用 Py_Finalize 方法销毁 Python 运行环境。

按照 emb_1 的方式修改 Makefile 文件，可以得到 emb_2 可执行文件。此代码即可以实现调用不同的 Python 函数。例如有下面的一个代码文件：

```
#filename: ari.py
```

更多资源请访问稀酷客(www.ckook.com)

```
#encoding = utf-8

def add(x,y):
    print "add a, b"
    return x + y

def mul(x,y):
    print "mul a, b"
    return x* y
```

- 这是一个很简单的 Python 代码文件，其中定义了 add 和 mul 这两个方法，实现了加法和乘法。

通过 emb_2 可执行文件可以对这段 Python 代码进行测试：

```
book @ 6[extern] $. /emb_2 ari add 4,5
add 4, 5
Result of call: 9

book @ 6[extern] $. /emb_2 ari mul 4,5
mul 4, 5
Result of call: 20
```

- 在对 emb_2 的测试中可以看到，此可执行文件已经可以解释 ari. py 文件中的 Python 代码了，并得到了正确的结果。

26.3　小结

本章讲解 Python 语言的扩展和嵌入。为了能够增强 Python 的扩展性，Python 语言既支持使用其他语言来书写 Python 模块，也支持将 Python 的解释器功能嵌入到其他的应用程序中去。在本章中，详细的介绍了这两种方式，从而可以使得在使用 Python 的时候更加得心应手。

更多资源请访问稀酷客(www.ckook.com)

第 27 章　Windows 下的 Python

Python 现在已经成为一个跨多平台的语言，可以在多个系统中使用。在 Windows 系统下，除了 Python 标准库之外，还有 Python 语言的 Windows 扩展模块。通过 COM 技术还可以访问 Windows 中的对象。本章的知识点：

- COM 技术介绍
- 使用 COM 对象
- 全局唯一标识符
- 使用 win32com 访问 Windows 应用

27.1　组件对象模型

组件对象模型（Component Object Model，COM）定义了不同组件之间相互通信的方式，提供了一种在不同的应用程序之间共享功能的规范。此模型允许多个不同的组件之间进行通信，而不用关心具体的实现语言和底层运行的计算机系统平台。在本节中将对组件对象模型进行介绍。

27.1.1　组件对象模型介绍

组件对象模型是 Microsoft 提出的一种创建组件和使用组件在应用程序中的通信规范。现在，Windows 下的大部分应用程序都支持 COM 技术。任何支持 COM 技术的系统都能够可靠地使用 COM 组件，而不用关心对象是如何实现的。在 COM 架构基础上，用户可以使用单一功能的组件，按照特定的需求将其组合起来，从而形成一个复杂的实际应用。

这种组件技术的由来也是和软件工业的发展相关的。当软件规模变大后，如果仍采用那种静态的组成方式，则在产品完成后对其修改会非常困难。为了解决这个问题，一种解决方案就是将应用程序的功能拆分为多个独立简单功能的部分，这就是组件的含义。使用组件的最大好处是在后面的维护过程中，可以使用新的组件来替代旧的组件。

需要注意的是，这种组件技术也是一种发展的过程，包括 COM。实际上，COM 技术也是对象链接与嵌入技术（Object Linking and Embedding）的发展。在 COM 技术被提出之后，又有其增强技术包括 DCOM（Distributed COM，分布式 COM 技术）和 COM + （DCOM + 管理）被相继提出。在 COM 的技术基础之上，还建立了更多其他相关技术，如 ActiveX 技术甚至更高层的 DirectX 技术等。随着组件技术的进一步发展，Microsoft 公司推出了 .NET 框架，其中就有用于替代 COM 的核心技术 CLR（Common Language Runtime，公共语言运行库）。即使如此，COM 技术还是被广泛用在 Windows 系统中。图 27-1 显示了 Microsoft 组件技术发展之间的关系。

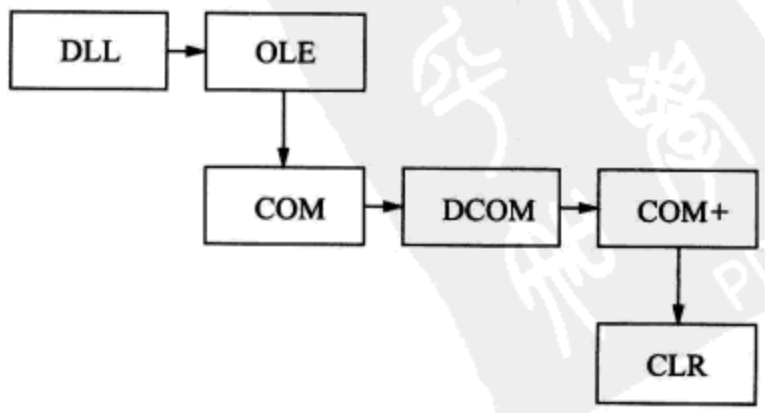


图 27-1　组件技术发展之间的关系

更多资源请访问稀酷客(www.ckook.com)

使用组件技术还可以实现应用程序的快速开发。虽然应用程序会有特定的组件实现，但是常规的组件都已经可以找到。这样避免了重复开发。另外，由于组件的透明性，所以可以很容易地实现应用程序的网络化功能（通过 DCOM）。可以说，COM 组件是一种为其他应用程序提供服务的很好方法。使用 COM 技术的发布方案，COM 客户可以动态的找到其所需的组件。

虽然 COM 技术只是一个规范标准，但实际上在 Windows 下也有具体的实现。特别地，有一个 COM 库可以供所有客户使用。其中包含一些 API 函数，用来创建 COM 程序。这个 COM 库保证了所有使用此组件的应用程序都有着同样的运行方式。COM 还提供其他一些功能。这也是 COM 技术能够得到迅速发展的一个原因。

27.1.2　COM 结构

COM 技术是基于组件概念的。对于一个基本的组件，至少需要有两方面的功能。一是需要能够查询组件中提供了什么接口。第二是需要组件能够进行自我生命周期管理，具体是通过参考计数（Reference Counting）来实现的。这两个功能需要在 IUnknown 接口中的几个方法中实现。这些方法包括 QueryInterface、AddRef 和 Release 等。所有 COM 组件都需要实现 IUnknown 接口。通过这种方式来保证所有 COM 组件的相同处理能力。

COM 的结构如图 27-2 所示。

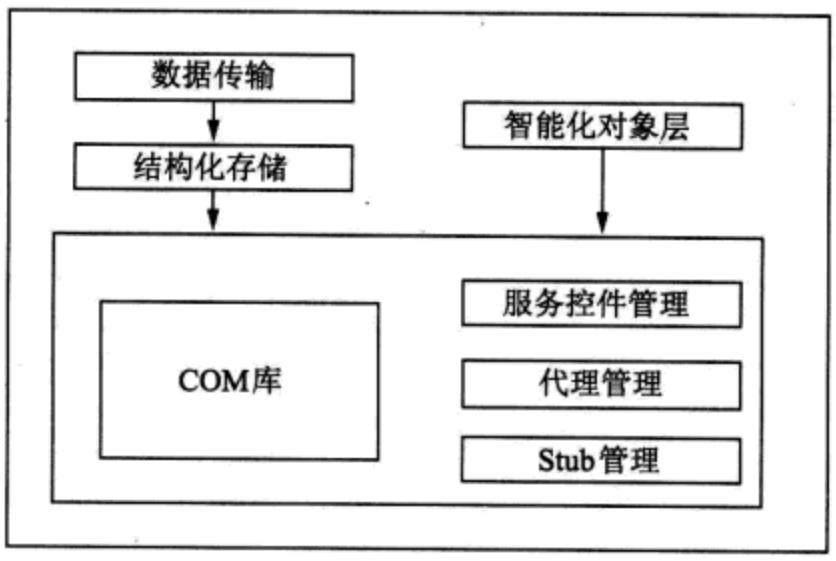


图 27-2　COM 的结构

在图 27-2 中最重要的部分是 COM 组件库，它支持对象的各种操作，包括定义、创建、删除和调度等。另外，都实现了上面介绍的基本功能，包括查询对象接口以及遍历对象功能。同时，COM 还提供了接口定义语言（Interface Definition Language，IDL）用来定义 COM 技术上的接口，实际上也是对于对象的一组逻辑相关的操作方法。

COM 技术采用了 IDL 来描述组件接口。包含标准接口和自定义接口。在 COM 技术中为所有的组件定义了一个共同的父接口 IUnknown。正如前面所介绍的，其中包含组件的基本方法。在组件模型中，除了基本规范和 COM 实现外，在 COM 中还包含永久存储、智能命名标记和数据统一转移等核心的系统组件。

虽然在 COM 组件中定义了许多的接口，但是实际上并没有实现。一个经常使用的接口是 IDispatch，它允许在脚本环境下访问 COM 对象，这些脚本环境就包括 Python 语言。虽然 COM 技术定义了 IDispatch 的接口，但是 COM 对象则负责实现具体的接口。而具体的实现则还依赖于组件所提供的功能。实现了 IDispatch 接口的组件被称为自动化组件。

27.1.3　COM 对象的交互

GUID（Global Unique Identifier，全局唯一标识符）是一个 16 字节的二进制值。它一般是通过网卡地址和 CPU 时钟联合生成的。由于这两者在全局范围内都是不同的，所以任何两个计算机系统都不会产生相同的 GUID 值。在需要使用全局唯一的标识符的时候，例如多个节点协同工作的情况下，可以使用 GUID。在组件中的类标志 GUID 被称为 CLSID（Class Identifier，类唯一标识符）。可以根据注册表，从 CLSID 中确定组件的位置。

在 COM 模型中，可以将类唯一标识符传递给 COM 核心并获取实例化的对象，这被称为

更多资源请访问稀酷客(www.ckook.com)

COM 客户端。在实际操作中，可以通过调用 CoCreateInstance 或者 CoGetClassObject 接口来生成。图 27-3 中显示了 COM 客户端和 COM 组件之间的交互过程。

在图 27-3 中，首先是 COM 客户端调用 CoCreateInstance 方法来访问 COM 库。在收到这个请求后，COM 库将定位并实例化服务器端的组件对象。接着，COM 库将可以使用的接口指针返回给 COM 客户端。这样，COM 客户端可以利用这个返回值来调用这个接口方法，实现特定的功能。在服务器端，将使用 COM 中的接口方法来生成对象。在客户端中，可以使用类工厂方法来创建对象。图 27-4 中显示了 COM 客户端的创建过程。

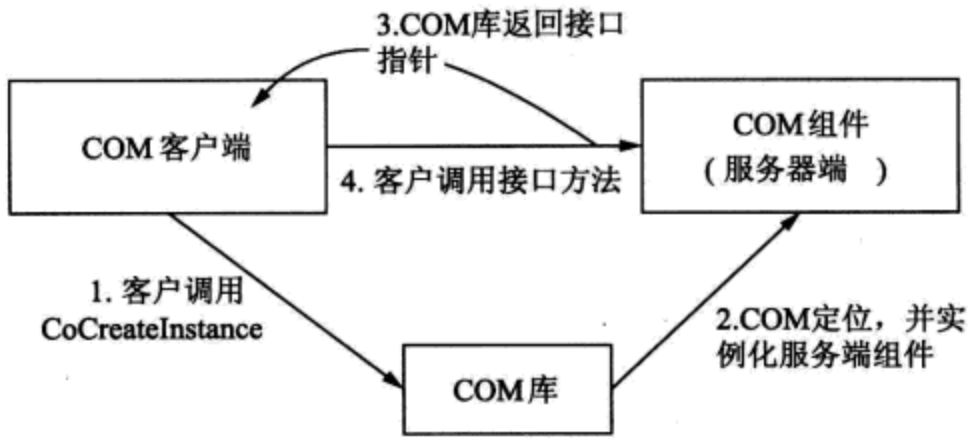


图 27-3　COM 客户端和 COM 组件的交互过程

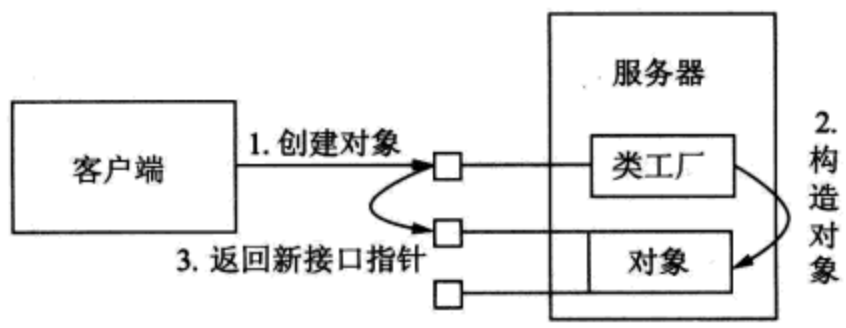


图 27-4　COM 客户端的创建

27.2　Python 对 COM 技术的支持

在 Python 语言中包含对 COM 技术的支持。这是通过 Python 的 Windows 扩展包来实现的。在本节中，将介绍如何通过此扩展包实现对 COM 组件的访问。

27.2.1　Python 中的 Windows 扩展：PyWin32

Python 的开发人员对是否将 Python 的 Windows 扩展支持加入 Python 的发行版本中曾经有过争议，但是最终还是没有获得通过。一个最重要的原因就是希望 Python 语言能够尽可能地作为一个跨平台的语言。加入过多的特定系统平台下的标准模块将会使得这种特性变弱。

在 Windows 下支持通过 Python 访问 Windows 对象的软件包是 PyWin32。其项目主页地址为 http://python.net/crew/mhammond/win32/。现在项目组已经将项目开始迁移到 Sourceforge 平台下。新的项目主页地址为 http://pywin32.sourceforge.net/projects/。可以从项目主页上下载特定 Python 版本的软件安装包进行安装。由于此软件包仅仅只是能够在 Windows 系统平台下使用，所以这里推荐使用可执行文件来安装软件包。图 27-5 所示为 PyWin32 软件包安装中的截图。在安装完毕后可以使用如下的操作来查看 PyWin32 软件包是否安装成功：

更多资源请访问稀酷客(www.ckook.com)

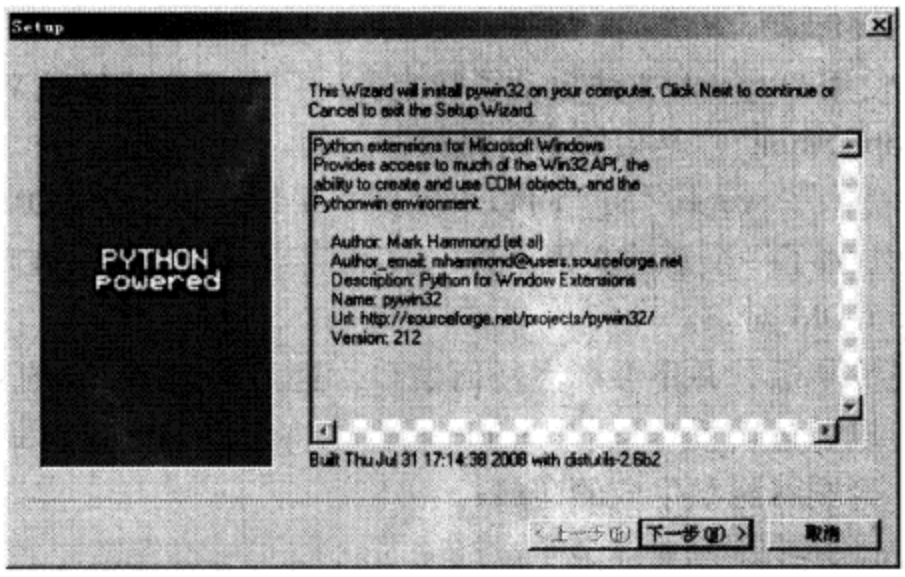


图 27-5　PyWin32 的安装截图

```
In [1]: import win32com
```

如果没有报错，说明已经安装成功。实际上，在安装了 PyWin32 软件包后，并不只是包含 win32com 模块，还有其他的操作 Windows API 的模块。在下面将介绍如何通过使用 win32com 模块来操作 COM 对象。

27.2.2　客户端 COM 组件

客户端主要关注的 COM 对象是自动化组件对象。这些对象为其他编程环境提供了可以使用的接口，使得用户直接通过接口来操作 COM 对象。这里可以使用 Python 的 Windows 扩展来操作自动化组件对象。这些组件对象包括 Microsoft Office、Internet Explorer 或者是支持 COM 组件的其他程序。

COM 对象的信息是保存在 Windows 的注册表中的，其中有每个组件的详细信息。当需要新生成一个组件的时候，系统将在注册表中查找此组件并使用找到的类。需要注意的是，这里的类并不是指 Python 语言中的类，而是一个 COM 类。其中，类中定义了具体的实现，而对象只是类的一个实例。类是通过前面介绍过的一个 CLSID 或者是 ProgID 来在注册表中注册。其中，CLSID 是一个 GUID，这在全局都是唯一的。而 ProgID 则可以显示一个友好的字符串，但是并不保证唯一。一般地，ProgID 都会是 COM 对象描述的字符串。如对 Microsoft Word 程序可以使用“Word. Applicatio”；对 Microsoft Excel 可以使用“Excel. Application”。

可以通过使用 win32com 模块中的 client. Dispatch 方法来创建一个 COM 对象。此方法可以接受一个 ProgID 或者是 CLSID 作为参数。这里的 ProgID 或者 CLSID 将作为此 COM 组件的标志。

【例 27-1】演示使用 client. Dispatch 方法生成一个 Word 对象。

```
In [1]: import win32com

In [2]: from win32com import client

In [3]: word = client.Dispatch("Word.Application")

In [4]: word
Out[4]: <COMObject Word.Application>

In [5]: doc = word.Documents.Add()

In [6]: doc
Out[6]: <COMObject Add>
```

- 在 In［1，2］中，导入了 win32com 中的 client 模块。此模块中包含客户端 COM 组件编

更多资源请访问稀酷客(www.ckook.com)

程所需要的相关库。

- 在 In［3］中使用 Dispatch 方法启动了特定的应用程序，这里是 Word，所以其中的参数为“Word. Application”。然后将运行结果赋值给 word 变量。
- 在 In［4］中显示了 word 对象，可以看到此变量现在为一个 COM 对象。
- 在 In［5］中增加了一个新的文档，并将其赋值给 doc 变量。在 In［6］中可以看到此变量也是一个 COM 对象。

当上面的操作执行完毕后，并不会产生任何窗口。这是因为 Word 的 COM 对象在生成的时候默认是不可见的。可以通过设置对象的 Visible 属性来将窗口显示或隐藏。当其值为 True 的时候，表示显示此窗口，否则将隐藏生成的窗口：

```
In [7]: word.Visible = True
```

在 In［7］中设置了 word 对象的 Visible 值为 True。这将会使得生成的 Word 实例显示出来。对于具体的 COM 对象有着不同的属性和方法。当此语句执行完毕后，将打开一个 Word 窗口，如图 27-6 所示。

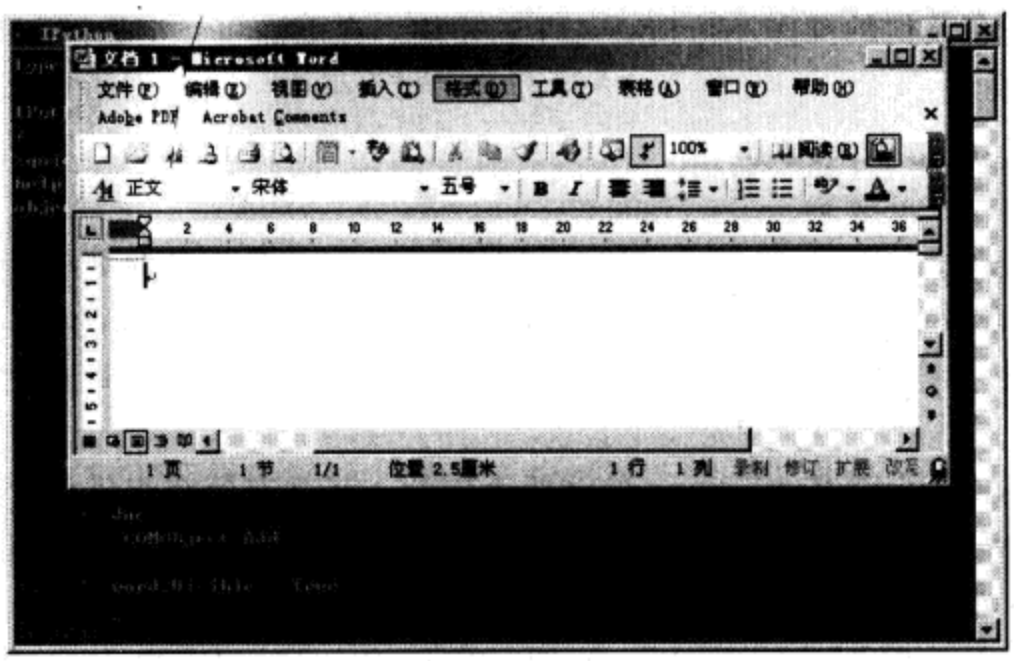


图 27-6　使用 win32com 操作 Word

当生成了 Word 的 COM 对象后，就可以使用此自动化组件所提供的方法来对 word 对象进行操作了。例如可以在 Word 中通过 COM 接口加入文字。当然，也可以调用应用程序提供的接口来关闭文档。实际上，Python 也会管理 COM 对象的生成周期。当发现变量不再被使用的时候，Python 会自动关闭应用程序。一种方便的方法是将变量赋值为其他值。如下面的操作中将其赋值为“None”：

```
In [8]: word = None
```

这种方式同样可以关闭此 COM 客户端。

27.2.3　实现 COM 组件

由于 Python 在字符串处理方面具有很大的优势，所以有些时候其他语言需要调用 Python 来处理字符串。其他语言可以通过扩展来访问 Python 中的实现，但是这对于每种语言都需要一个相应的扩展。另外一种实现方式就是使用 COM 组件。其好处就是可以使得其在实现的时候不必关心客户端 COM 组件的语言。在 PyWin32 包中包含实现 COM 组件的方法。

在 Python 中实现 COM 对象将有下面的步骤。

（1）首先定义一个 Python 类，此类将作为 COM 类。

更多资源请访问稀酷客(www.ckook.com)

（2）定义将要提供给外界程序所使用的接口和属性。
（3）设置 ProgID 和 CLSID 的值。
（4）实现提供给外部使用的方法。
（5）调用 win32com 中的方法来注册 COM 服务组件。

【例 27-2】演示注册一个 Python 的 COM 组件。

```
#filename: com_1.py
#encoding = utf-8
class PythonUtilities(object):
    _public_methods_ = [ 'SplitString' ] #提供的接口名

    _reg_progid_ = "Python.Utilities" #注册的 ProgID

    _reg_clsid_ = "{71A4461C-6EDC-4cda-AF97-6744234C8D38}"#注册的 CLSID

    def SplitString(self, val, sep = None):
        import string
        if sep!  = None: sep  = str(sep)
        return string.split(str(val), sep)

if __name__ =='__main__':
    #注册 COM 服务组件
    print "Registering COM server..."
    import win32com.server.register
    win32com.server.register.UseCommandLine(PythonUtilities)
```

- 【例 27-2】代码片段有两个部分。其中前面定义了一个 PythonUtilities 类，这是一个 COM 类。后面的部分对上面定义的类进行注册。
- 在定义的 PythonUtilities 类中，有 3 个特殊的属性。其中，_public_methods_属性用来指定提供给外部程序所使用的接口，这里仅仅提供了一个方法 SplitString。_reg_progid_属性用来指定 ProgID，COM 客户端将可以利用此值来访问 COM 对象，并使用提供的接口。此值最好是简单且易记。第 3 个属性_reg_clsid_定义了此 COM 组件的 GUID 值。此值不要手工指定，一般是通过工具生成的。实际上，Windows 系统提供了一个实用工具 guidgen 来生成 GUID 值。图 27-7 显示了此工具的使用方法。
- 实际上，也可以使用下面的代码来生成 GUID 值：

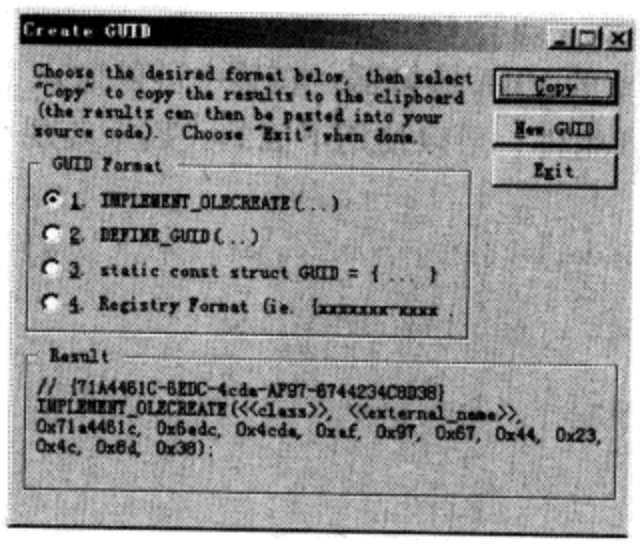


图 27-7　guidgen 工具的使用

更多资源请访问稀酷客(www.ckook.com)

```
In [1]: import pythoncom

In [2]: pythoncom.CreateGuid()
Out[2]: IID('{A22616D4-C261-4B19-8D91-1160278EB7C5}')
```

- 在 PythonUtilities 中还定义了一个 SplitString 函数。此函数即为提供给外部使用的具体实现。在函数主体中可以看到，此函数除去参数判断部分，实际上就是简单的调用了 Python标准库 string 模块中的 split 方法。
- 后面的代码放入了“if __name__ =='__main__':”代码中，这使得下面的代码只会在文件被调用的时候执行。这里的代码首先从 win32com.server 模块中导入了 register，然后使用其中的 UseCommandLine 方法来注册 PythonUtilities 这个 COM 组件。

直接运行此代码，可以实现对定义的 COM 组件的注册：

```
In [9]: run com_1.py
Registering COM server...
Registered: Python.Utilities
```

从输出信息中可以看到，系统已经注册了这个 COM 组件。

这时，就可以在其他应用程序中使用这个 COM 组件了。作为一个简单的实例，可以在 VBA 中来实现对已注册的 COM 组件的访问。

（1）首先打开 Word 中的 VBA 编辑器。

（2）单击菜单栏中的【工具】|【宏】|【宏】命令，或按 Alt + F8 键，打开如图 27-8 所示的【宏】对话框。

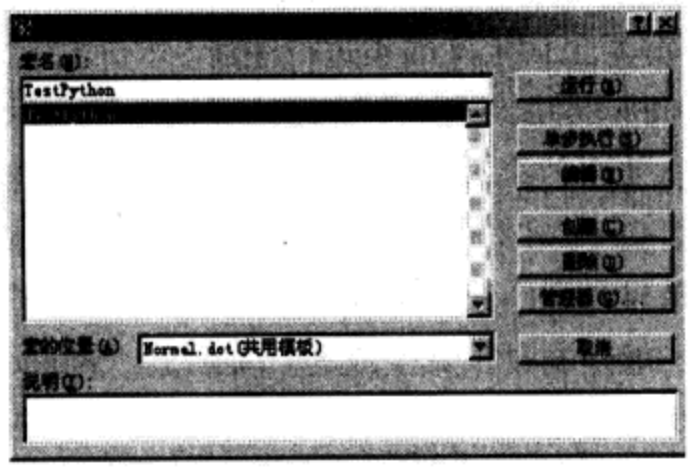


图 27-8　【宏】对话框

（3）在宏名中输入一个名称（如“TestPython”），然后单击右侧的【创建】按钮。

（4）这时，将会打开 Word 的 VBA 编辑器。在其中可以输入 VB 代码。

为了访问刚刚注册的 COM 组件，可以使用下面的代码来实现：

```
Sub TestPython()
    Set PythonUtils = CreateObject("Python.Utilities")

    response = PythonUtils.SplitString("Hello World")

    For Each Item In response
    MsgBox Item
    Next
End Sub
```

- 在这个 VB 方法中，首先使用 CreateObject 建立了和所定义的 COM 组件的关联。
- 在第二个语句中调用 PythonUtils 所提供的接口 SplitString 来对指定的字符串进行分拆，并将返回值赋值给 response 变量。

更多资源请访问稀酷客(www.ckook.com)

❑ 后面的三个语句则是使用对话框显示了字符串分拆后的各个部分。

在 VBA 编辑器中，单击【运行】|【运行子过程】/【用户窗体】命令（或按 F5 键）运行此宏。运行结果如图 27-9 所示。

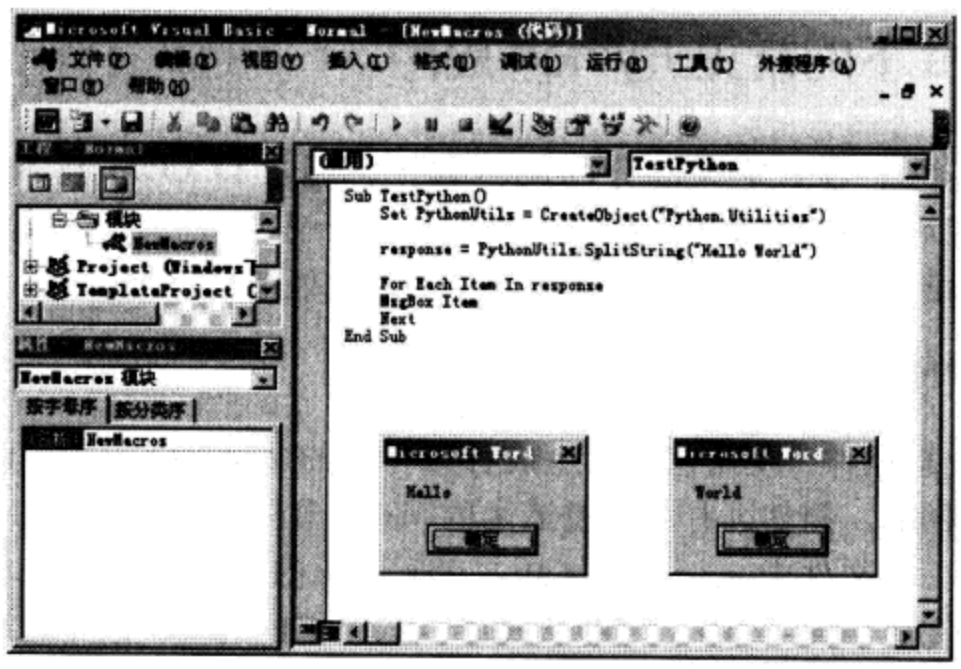


图 27-9　在 VBA 中调用注册的 COM 组件

笔者对图 27-9 做了一些处理，将代码界面和运行的结果放在了一幅图中。从图 27-9 中的两个输出对话框可以看出，通过调用 PythonUtils 组件中的 SplitString 接口，将指定的字符串分解成了多个部分。

为了不影响到注册表，一个好的习惯是当此 COM 组件不使用的时候将其从注册表中删除。这也是很容易实现的。在运行文件的时候，在后面加上“--unregister”即可：

```
In [10]: run com_1.py --unregister
Registering COM server...
Unregistered: Python.Utilities
```

现在如果再度运行 VBA 中的宏，将会出现如图 27-10 所示的错误。

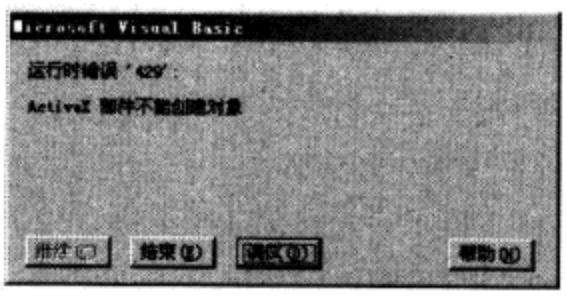


图 27-10　找不到 COM 组件的错误显示

在图 27-10 所示的对话框中，单击【调试】按钮还可以找到出错的 VBA 代码。本例将定位到 TestPython 方法中的第 1 行，表明现在找不到相应的 COM 组件。

当然，在 Python 中也是可以测试此 COM 组件的：

```
In [11]: run com_1.py
Registering COM server...
Registered: Python.Utilities

In [12]: s = win32com.client.Dispatch("Python.Utilities")

In [3]: s.SplitString("Hello World")
Out[13]: (u'Hello', u'World')
```

❑ 在 In［12］中使用 Dispatch 方法访问了前面定义的 ProgID 所指代的 COM 组件，并生成

更多资源请访问稀酷客(www.ckook.com)

了一个 COM 对象。

- In［13］中调用了 SplitString 方法。可以看到，这里同样将指定的字符串分解成了两个字符串。

27.3　Windows 下的常见 Python 应用

由于 Windows 系统平台下绝大部分的应用程序都是支持 COM 技术的，所以可以通过 win32com 模块来实现对这些应用程序的自动访问。这对于需要经常做一些重复性工作的用户是很有用的。本节将介绍一些 Windows 系统下应用的自动化处理。

27.3.1　对 Word 的自动访问

Word 是一款字处理软件，使用它可以生成各式各样的文档。对 Word 实现自动化处理，对于商务办公是很有帮助的。例如，有一个邀请函的 Word 模板，其中需要加上被邀请人的名字。而这些名字放在另外一个文件中。这可以打开一个邀请函的模板文件，输入一个名字，然后保存。如此反复，最终完成任务。但是，当需要邀请的人很多的时候，这种处理方式的效率会非常低。为了提高工作效率，可以用 win32com 模块来实现自动化处理。

一个 COM 客户端的典型实现是生成一个 COM 对象，然后调用 COM 对象所提供的方法进行处理，最后关闭此对象。对于处理邀请函的任务，这种处理方式是很合适的。

【例 27-3】演示使用 win32com 自动操作 Word 的过程。

```
#filename: word.py
#encoding = utf-8

from time import sleep
import win32com

def word(name):

    #生成了一个 COM 对象
    word = win32com.client.Dispatch('Word.Application')
    doc = word.Documents.Add()
    word.Visible = True
    sleep(1)

     #对 Word 内容的操作
    rng = doc.Range(0,0)
    rng.InsertAfter(u'尊敬的%s:\n'% name)
    rng.InsertAfter(u'    诚邀您参加于下周五晚六点在公司举行的晚会。')
    sleep(1)

    #保存文件
    filename = name + ".doc"
    doc.SaveAs(filename)

    #关闭程序
    doc.Close(False)
    word.Application.Quit()

if __name__ =='__main__':
    names = ['Alice', 'Bob', 'Eve']
    for name in names:
        word(name)
```

- 在 word 函数这段代码中有 4 个部分。其中的第 1 部分在前面已经做过介绍。注意到，这里使用了 sleep 函数，主要的功能是可以看到操作 word 文档的过程。

更多资源请访问稀酷客(www.ckook.com)

- 在第 2 部分中实现了对 Word 内容的操作。其中有一个可变的内容部分——名字。这可以使得每个人得到的邀请函都是不一样的。
- 在第 3 部分的代码中，使用 SaveAs 方法保存了文件。文件名就是被邀请人的名字。
- 在最后，首先调用了 doc 的 Close 方法将文档关闭。这时，Word 中就没有了任何的可编辑文档。而在其后，调用 Quit 方法退出程序。
- 在测试的过程中只是使用了 3 个人的名字。实际上，这个部分的内容可以从数据文件或者是数据库中读取。

运行此代码文件，会在【我的文档】文件夹中生成 3 个以被邀请人的名字分别命名的 Word 文件。将这 3 个文件打开后，可以发现每个文件都对应着不同的被邀请人。其中一个文件的内容如图 27-11 所示。

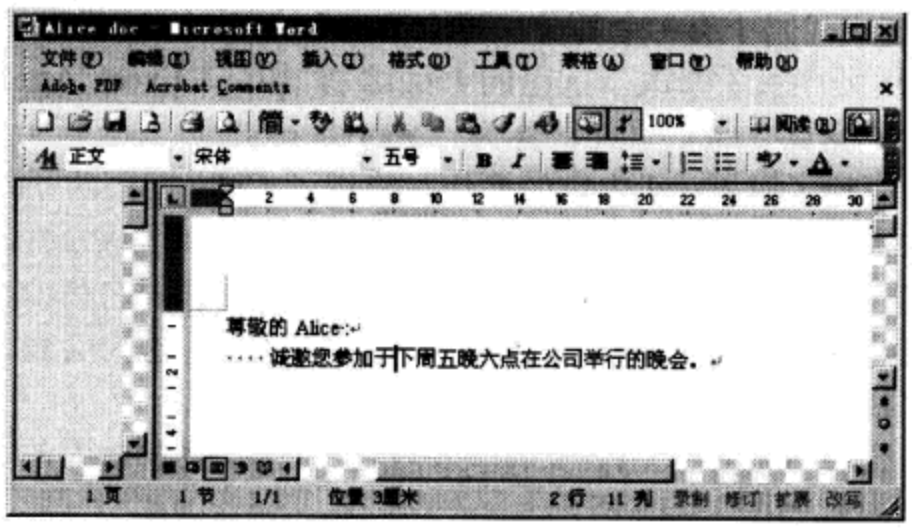


图 27-11　自动化生成的 Word 文档

当然，这段代码还是很粗糙的，还需要进一步的改进。在生成多个文档的时候，没有必要对每个名字都重新打开应用程序，而只是需要新建和关闭文档即可。

27.3.2　对 Excel 的自动访问

Excel 是一款强大的表格软件。下面实现的内容和在 Word 中实现的类似，只是这里是在不同的单元格中显示内容。

【例 27-4】演示使用 win32com 自动操作 Excel 的过程。

```
#filename: excel.py
#encoding = utf-8

from time import sleep
import win32com

def excel(name):

    #生成了一个 COM 对象
    ex = win32com.client.Dispatch('Excel.Application')
    wk = ex.Workbooks.Add()
    nwk = wk.ActiveSheet
    ex.Visible = True
    sleep(1)

     #对 Excel 内容的操作

    nwk.Cells(1,1).value = u'尊敬的%s:\n'% name
```

更多资源请访问稀酷客(www.ckook.com)

```
    nwk.Cells(2,1).value = u'    诚邀您参加于下周五晚六点在公司举行的晚会。'

    #保存文件
    filename = name + ".xlsx"
    wk.SaveAs(filename)

    #关闭程序
    wk.Close(False)
    ex.Application.Quit()

if __name__=='__main__':
    names = ['Alice', 'Bob', 'Eve']
    for name in names:
        excel(name)
```

- 在 excel 函数这段代码中有 4 个部分。每个部分和前面 Word 操作中的都基本类似。只是针对 Excel 应用程序做了修改。这里的 ProgID 改为了“Excel. Application”。而其所包含的对象也发生了变化，每个页面被称为“Workbooks”。
- 在第 2 部分中实现了对 Excel 内容的操作。使用了 Cells 方法的 value 属性值来设置单元格的内容。这里对第 1 行第 1 列和第 2 行第 1 列的两个单元格的内容进行了更改。
- 在第 3 部分代码中，使用 SaveAs 方法保存了文件。文件的名称即为被邀请人的名字。
- 在最后，首先调用了 wk 对象的 Close 方法将文档关闭。这个时候，Excel 中就没有了任何的可编辑表格。其后，调用 Quit 方法退出 Excel 程序。

运行此代码文件，会在【我的文档】文件夹中生成 3 个以被邀请人的名字分别命名的 Excel 文件。其中一个文件的内容如图 27-12 所示。

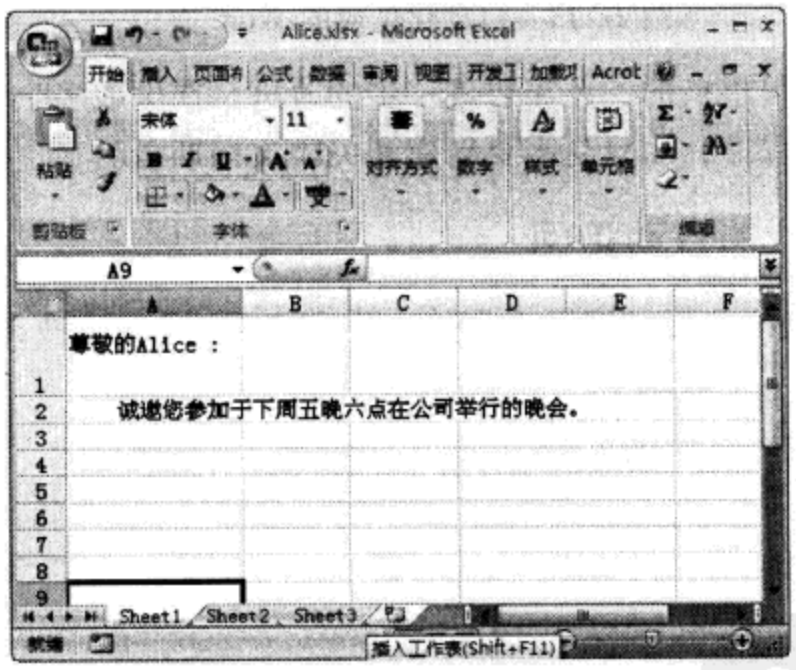


图 27-12　自动化生成的 Excel 文档

从上面可以看出，COM 组件并不关心所生成对象的应用程序版本，所以即使这里的 Word 和Excel两者的版本并不一致，也不影响两个应用程序具有着同样的访问接口和操作方式。

27.3.3　对 Powerpoint 的自动访问

PowerPoint 是一款知名的演示文档制作工具。这里依然以处理邀请函的任务为例，在 Power Point中实现自动化功能。

【例 27-5】演示使用 win32com 自动操作 PowerPoint 的过程。

```
#filename: powerpoint.py
#encoding = utf-8
```

更多资源请访问稀酷客(www.ckook.com)

```
from time import sleep
import win32com

def powerpoint(name):

    #生成了一个 COM 对象
    ppt = win32com.client.Dispatch('PowerPoint.Application')
    pres = ppt.Presentations.Add()
    ppt.Visible = True
    sleep(1)

    #对 Powerpoint 内容的操作
    s1 = pres.Slides.Add(1,1)
    s1_0 = s1.Shapes[0].TextFrame.TextRange
    s1_0.Text = u'尊敬的%s \n'% name

    s1_1 = s1.Shapes[1].TextFrame.TextRange
    s1_1.InsertAfter(u'    诚邀您参加于下周五晚六点在公司举行的晚会。')

    #保存文件
    filename = name + ".pptx"
    pres.SaveAs(filename)

    #关闭程序
    pres.Close()
    ppt.Application.Quit()

if __name__=='__main__':
    names = ['Alice', 'Bob', 'Eve']
    for name in names:
        powerpoint(name)
```

- 在 powerpoint 函数这段代码中有 4 个部分。每个部分和前面 Word 操作中的基本类似，但是针对 PowerPoint 应用程序做了修改。这里的 ProgID 改为了“PowerPoint.Application”。而其所包含的对象也发生了变化，每个演示文稿被称为“Presentations”。
- 在第 2 部分中实现了对 PowerPoint 内容的操作。调用 Slides.Add 新建一个演示页面，并分别在不同的页面布局部分输入了相应的内容。
- 在第 3 部分代码中，使用 SaveAs 方法保存了文件。文件名即为被邀请人的名字。
- 在最后，首先调用 wk 对象的 Close 方法将演示文稿关闭。这个时候，PowerPoint 中就没有了任何可以编辑的演示文稿。其后，调用 Quit 方法退出 PowerPoint 程序。

运行此代码文件，会在【我的文档】文件夹中生成 3 个以被邀请人的名字分别命名的 PowerPoint 文件。其中一个文件的内容如图 27-13 所示。



图 27-13　自动化生成的 PowerPoint 文档

更多资源请访问稀酷客(www.ckook.com)

27.3.4　对 Outlook 的自动访问

Outlook 是 Windows 系统下出色的电子邮件管理软件。下面以实例说明对 Outlook 的自动访问。

【例 27-6】演示使用 win32com 自动操作实现用 Outlook 自动发送邮件。

```
#filename: outlook.py
#encoding = utf-8

from time import sleep
import win32com

def outlook(name):

    #生成了一个 COM 对象
outlook = win32com.client.Dispatch('Outlook.Application')

    mail = outlook.createItem(0)
    mail.Recipients.Add('%s@server.com'% name)
mail.Subject = u"邀请函"

    body = ""u'尊敬的% s \n'% name
    body.append(u'    诚邀您参加于下周五晚六点在公司举行的晚会。')

    mail.Body = body

    mail.Send() #发送邮件

    outlook.Quit()

if __name__ =='__main__':
    names = ['Alice', 'Bob', 'Eve']
    for name in names:
        outlook(name)
```

- 这里的代码和前面的有一些区别。因为前面的应用中都是完成文档，而这里的应用则是发送电子邮件。在 Outlook 函数的最前面，使用 Dispatch 方法生成了一个 COM 对象。其中，ProgID 为“Outlook.Application”。
- 接着调用其 createItem 方法，生成了一个邮件的实例。
- 接下来为邮件的各个域设置合适的值。其中最关键的域有 3 个：邮件接收人、邮件主体和邮件正文内容。
- 当邮件构造完毕后，就可以使用其对象的 Send 方法将此邮件发送出去。
- 最后调用 Quit 方法退出应用程序。

实际上，这种处理方式和手工打开 Outlook 来发送邮件的效果是一样的。运行此段代码，将可以在 Outlook 的【已发送邮件夹】中可以看到刚刚发送的这 3 封邮件。

27.4　小结

本章讲解了 Python 语言在 Windows 系统下的扩展。首先介绍了组件对象模型，分析了 COM 的结构以及各个组件之间的交互处理。随后介绍了 Python 中支持 Windows 系统的扩展软件包 PyWin32，并使用其中的 win32com 模块作为客户端组件来控制 Windows 的应用程序。接着介绍了如何使用 win32com 模块构建 COM 组件，并演示了如何在 VBA 中访问自定义的 COM 组件服务。最后对 Windows 系统下的常见应用做了详细的介绍，主要针对 Office 中的软件自动化处理，包括 Word、Excel、PowerPoint 和 Outlook 等。

更多资源请访问稀酷客(www.ckook.com)

[General Information]
书名=Python 开发技术详解
作者=周伟著
页数=532
SS号=12400701
出版日期=2009.08
出版社=北京市：机械工业出版社
SSLIB-JPG=http://image5.5read.com/image/ss2jpg.dll?did=b48&pid=AC5E950A27423271F058C8BD40AD4200E35F2F2B1E2FF2D1BB4F17C08279BF167BF6822907BB53B472D19A73F99B8F5C5C4062F5235DC2C9988E6138C6364B139DDB47A732B5E56350A1DA2F862617F6700CEEF03482A9C65B8E5EFBD0876D7F95DB893F4954256B7AB1B80665 3D15855A01543&jid=/

更多资源请访问稀酷客(www.ckook.com)