目录

[第1章 欢迎来到Python世界 4](#_Toc502860196)

[1.1 什么是Python？ 4](#_Toc502860197)

[1.2 起源 4](#_Toc502860198)

[1.3 特点 4](#_Toc502860199)

[1.4 下载和安装Python 7](#_Toc502860200)

[1.4.1 安装 7](#_Toc502860201)

[1.4.2 自己动手编译Python 7](#_Toc502860202)

[1.5 运行Python 8](#_Toc502860203)

[1.5.1 其它的集成开发环境和执行环境 9](#_Toc502860204)

[1.6 Python文档 9](#_Toc502860205)

[1.7 Python与其他语言的比较 10](#_Toc502860206)

[1.8 其它实现（略） 10](#_Toc502860207)

[1.9 练习 10](#_Toc502860208)

[第2章 Python起步 10](#_Toc502860209)

[2.1 介绍 10](#_Toc502860210)

[2.2 输入/输出 11](#_Toc502860211)

[2.3 注释 13](#_Toc502860212)

[2.4 运算符 14](#_Toc502860213)

[2.5 变量与赋值 15](#_Toc502860214)

[2.6 Python类型 16](#_Toc502860215)

[第17章 网络客户端编程 22](#_Toc502860216)

[第18章 多线程编程 22](#_Toc502860217)

[18.1 Python、线程和全局解释器锁 22](#_Toc502860218)

[18.1.1 全局解释器锁（GIL） 22](#_Toc502860219)

[18.1.2 退出线程 23](#_Toc502860220)

[18.1.3 在 Python 中使用线程 23](#_Toc502860221)

[18.1.4 没有线程支持的情况 23](#_Toc502860222)

[18.1.5 Python 的 threading 模块 25](#_Toc502860223)

[18.2 thread 模块 26](#_Toc502860224)

[18.3 threading 模块 31](#_Toc502860225)

[18.3.1 Thread 类 32](#_Toc502860226)

[18.3.4 斐波那契，阶乘和累加和 39](#_Toc502860227)

[18.3.5 threading 模块中的其它函数 45](#_Toc502860228)

[18.3.6 生产者-消费者问题和 Queue 模块 45](#_Toc502860229)

[18.4 相关模块 49](#_Toc502860230)

[18.5 练习 49](#_Toc502860231)

[第20章 Web编程 52](#_Toc502860232)

[20.1 引言 52](#_Toc502860233)

[20.2 Python 的 Web 应用：简单的 Web 客户端 52](#_Toc502860234)

[20.2.1 URL 53](#_Toc502860235)

[20.2.2 urlparse 模块 54](#_Toc502860236)

[20.2.3 urllib 模块 55](#_Toc502860237)

[20.2.4 urllib2 模块 58](#_Toc502860238)

[20.3 高级的 Web 客户端 59](#_Toc502860239)

[20.4 CGI: 帮助Web服务器处理客户端数据 62](#_Toc502860240)

[20.4.1 CGI 介绍 62](#_Toc502860241)

[20.4.2 CGI 应用程序 64](#_Toc502860242)

[20.4.3 CGI模块 64](#_Toc502860243)

[20.5 创建 CGI 应用程序 64](#_Toc502860244)

[20.5.1 建立 Web 服务器 64](#_Toc502860245)

[20.5.2 建立表单页 65](#_Toc502860246)

[20.5.3 生成结果页 67](#_Toc502860247)

[20.5.4 生成表单和结果页面 69](#_Toc502860248)

[20.5.5 全面交互的 Web 站点 73](#_Toc502860249)

[20.6 在 CGI 中使用 Unicode 编码 85](#_Toc502860250)

[20.7 高级 87](#_Toc502860251)

# 第1章 欢迎来到Python世界

## 1.1 什么是Python？

一门优雅而健壮的编程语言。既有传统编译语言的强大性和通用性，同时也借鉴了简单脚本和解释语言的易用性。

## 1.2 起源

贵铎·范·罗萨姆（Guido van Rossum），1989年底创建了Python，他当时在荷兰的CWI（Centrum voor Wiskunde en Informatica，国家数学和计算机科学研究院）。1991年正式发行第一个公开版。

## 1.3 特点

1.3.1高级

Python语言有高级的数据结构，这样就减少了以前“框架”开发需要的时间。像Python中的列表（大小可变的数组）和字典（哈希表）就是内建于语言本身的。它们的存在大大减少了开发周期和代码量。

1.3.2 面向对象

面向对象编程支持将特定的行为、特性以及和/或功能与它们要处理或所代表的数据结合在一起。Python的面向对象的特性是与生俱来的。然而，Python 绝不想Java或Ruby仅仅是一门面向对象语言，事实上它融汇了多种编程风格。例如，它甚至借鉴了一些像 Lisp 和 Haskell 这样的函数语言的特性。

1.3.3 可升级

Python 提供了基本的开发模块，你可以在它上面开发你的软件，而且当这些需要扩展和增长时，Python的可插入性和模块化架构则能使你的项目生机盎然和易于管理。

1.3.4 可扩展

就算你的项目中有大量的Python代码，你也依旧可以有条不紊地通过将其分离为多个文件或模块加以组织管理。而且你可以从一个模块中选取代码，而从另一个模块中读取属性。更棒的是，对于所有模块，Python的访问语法都是相同的。不管这个模块是Python标准库中的还 是你一分钟之前创造的，哪怕是你用其他语言写的扩展都没问题！借助这些特点，你会感觉自己根据需要“扩展”了这门语言，而且你已经这么做了。

代码中的瓶颈，可能是在性能分析中总排在前面的那些热门或者一些特别强调性能的地方，可以作为 Python 扩展用C重写。需要重申的是，这些接口和纯Python模块的接口是一模一样的，乃至代码和对象的访问方法也是如出一辙的。唯一不同的是，这些代码为性能带来了显著的提升。自然，这全部取决你的应用程序以及它对资源的需求情况。很多时候，使用编译型代码重写程序的瓶颈部分绝对是益处多多的，因为它能明显提升整体性能。

程序设计语言中的这种可扩展性使得工程师能够灵活附加或定制工具，缩短开发周期。虽然像 C、C++乃至 Java 等主流第三代语言（3GL）都拥有该特性，但是这么容易地使用C编写扩展确实是Python的优势。此外，还有像PyRex这样的工具，允许C和Python混合编程， 使编写扩展更加轻而易举，因为它会把所有的代码都转换成C语言代码。

因为Python的标准实现是使用C语言完成的（也就是CPython），所以要使用 C 和 C++ 编写 Python 扩展。Python 的Java实现被称作Jython，要使用Java编写其扩展。最后，还有 IronPython，这是针对 .NET 或 Mono 平台的 C# 实现。你可以使用 C# 或者 VB.Net 扩 展 IronPython。

1.3.5 可移植性

在各种不同的系统上可以看到 Python 的身影，这是由于在今天的计算机领域，Python 取得了持续快速的成长。因为 Python 是用C写的，又由于C的可移植性，使得 Python 可以运行在任何带有ANSI C编译器的平台上。尽管有一些针对不同平台开发的特有模块，但是在任何一个平台上用 Python 开发的通用软件都可以稍事修改或者原封不动的在其他平台上运行。这种可移植性既适用于不同的架构，也适用于不同的操作系统。

1.3.6 易学

Python关键字少、结构简单、语法清晰。这样就使得学习者可以在相对更短的时间内轻松 上手。对初学者而言，可能感觉比较新鲜的东西可能就是 Python 的面向对象特点了。那些还未能全部精通 OOP（Object Oriented Programming, 面向对象的程序设计）的人对径直使用 Python还是有所顾忌的，但是OOP并非必须或者强制的。入门也是很简单的，你可以先稍加涉猎，等到有所准备之后才开始使用。

1.3.7 易读

Python 与其他语言显著的差异是，它没有其他语言通常用来访问变量、定义代码块和进行模式匹配的命令式符号。通常这些符号包括：美元符号（$）、分号（;）、波浪号（~）等等。 没有这些分神的家伙，Python 代码变得更加定义清晰和易于阅读。让很多程序员沮丧（或者欣 慰）的是，不像其他语言，Python 没有给你多少机会使你能够写出晦涩难懂的代码，而是让其他人很快就能理解你写的代码，反之亦然。如前所述，一门语言的可读性让它更易于学习。我们甚至敢冒昧的声称，即使对那些之前连一行 Python 代码都没看过的人来说，那些代码也是相当容易理解的。看看下一章节——“起步”中的例子，然后告诉我们你的进展是多么神速。

1.3.8 易维护

源代码维护是软件开发生命周期的组成部分。只要不被其他软件取代或者被放弃使用，你的软件通常会保持继续的再开发。这通常可比一个程序员在一家公司的在职时间要长得多了。 Python项目的成功很大程度上要归功于其源代码的易于维护，当然这也要视代码长度和复杂度而定。然而，得出这个结论并不难，因为Python本身就是易于学习和阅读的。Python另外一个激动人心的优势就是，当你在阅读自己六个月之前写的脚本程序的时候，不会把自己搞得一头雾水，也不需要借助参考手册才能读懂自己的软件。

1.3.9 健壮性

没有什么能够比允许程序员在错误发生的时候根据出错条件提供处理机制更有效的了。针 对错误，Python 提供了“安全合理”的退出机制，让程序员能掌控局面。一旦你的 Python 由 于错误崩溃，解释程序就会转出一个“堆栈跟踪”，那里面有可用到的全部信息，包括你程序 崩溃的原因以及是那段代码（文件名、行数、行数调用等等）出错了。这些错误被称为异常。 如果在运行时发生这样的错误，Python 使你能够监控这些错误并进行处理。 这些异常处理可以采取相应的措施 例如解决问题 重定向程序流、执行清除或维护步骤、正常关闭应用程序、亦或干脆忽略掉。无论如何，这都可以有效的缩减开发周期中的调试环节。

1.3.10 高效的快速原型开发工具

我们之前已经提到了 Python 是多么的易学易读。但是，你或许要问了， BASIC 也是如此啊， Python 有什么出类拔萃的呢？与那些封闭僵化的语言不同，Python有许多面向其他系统的接口，它的功能足够强大，本身也足够强壮，所以完全可以使用Python开发整个系统的原型。显然，传统的编译型语言也能实现同样的系统建模，但是 Python 工程方面的简洁性让我们可以在同样 的时间内游刃有余的完成相同的工作。此外，大家已经为 Python 开发了为数众多的扩展库，所 以无论你打算开发什么样的应用程序，都可能找到先行的前辈。你所要做的全部事情，就是来 个“即插即用”（当然，也要自行配置一番）！只要你能想得出来，Python 模块和包就能帮你实现。Python 标准库是很完备的，如果你在其中找不到所需，那么第三方模块或包就会为你完 成工作提供可能。

1.3.11 内存管理器

Python解释器负责内存的管理工作，减轻了开发者的工作负担，也降低了程序错误的发生。

1.3.12 解释性和（字节）编译性

Python 是一种解释型语言，这意味着开发过程中没有了编译这个环节。一般来说，由于不 是以本地机器码运行， 纯粹的解释型语言通常比编译型语言运行的慢。然而，类似于 Java， Python 实际上是字节编译的，其结果就是可以生成一种近似机器语言的中间形式。这不仅改善了 Python 的性能，还同时使它保持了解释型语言的优点。

**核心笔记：**

**文件扩展名 Python 源文件通常用.py 扩展名。 当源文件被解释器加载或者显式地进行字节码编译的时 候会被编译成字节码。由于调用解释器的方式不同，源文件会被编译成带有.pyc 或.pyo 扩展 名的文件，你可以在第 12 章“模块”学到更多的关于扩展名的知识。**

## 1.4 下载和安装Python

Python官方网站：<http://python.org>

本书的网站：<http://corepython.com>

### 1.4.1 安装

类Unix系统可能已经安装了Python。最好的查看方法就是通过命令行运行Python，查看它是否在搜索路径上而且运行正常。只需输入：

#python

如果正常安装就会进入Python命令行运行模式。如果没安装会报错。

而对于Windows/DOS系统，从前面提到的两个网站下载python-2.7.10.msi，之后执行该文件安装Python即可。

### 1.4.2 自己动手编译Python

对绝大多数其它平台，下载.tgz文件，解压缩这些文件，然后执行以下操作以编译Python:  
1. ./configure  
2. make  
3. make install

Python通常被安装在固定的位置，所以你很容易就能找到。如今，在系统上安装多种版本的 Python 已经是司空见惯的事情了。虽然容易找到二进制执行文件，你还是要设置好库文件的 安装位置。在Unix中，可执行文件通常会将Python安装到/usr/local/bin 子目录下，而库文件则通常安装在/usr/local/lib/python2.x 子目录下，其中的 2.x 是你正在使用的版本号。MacOS X系统中，Python则安装在/sw/bin以及/或者/usr/local/bin子目录下。而库文件则在/sw/lib, /usr/local/lib,以及/或者  
/Library/Frameworks/Python.framework/Versions子目录下。在Windows中，默认的安装地址是 C:\Python2x。请避免将其安装在 C:\Program Files 目录下。是的，我们知道这是通常安装程序的文件夹。但是 DOS 是不支持“Program Files” 这样的长文件名的，它通常会被用 “Progra~1”这个别名代替。这有可能给程序运行带来一些麻烦，所以最好尽量避免。所以，听我的，将 Python 安装在 C:\Python 目录下，这样标准库文 件就会被安装在 C:\Python\Lib 目录下。

## 1.5 运行Python

3种方式：

（1）交互式解释器

（2）运行Python脚本

（3）调用集成开发环境中的图形用户界面运行Python

1命令行选项

-d 提供调试输出

-O 生成优化的字节码（生成 .pyo 文件）

-S 不导入 site 模块以在启动时查找 Python 路径

-v 冗余输出（导入语句详细追踪）

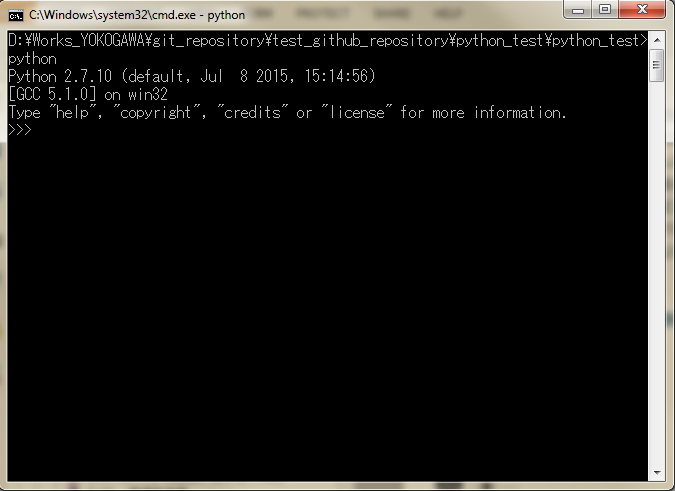
-m mod 将一个模块以脚本形式运行

-Q opt 除法选项（参阅文档）

-c cmd 运行以命令行字符串形式提交的 Python 脚本

file 从给定的文件运行 Python 脚本（参阅后文）

图 1－1 在一个 DOS/命令行 窗口启动 Python



### 1.5.1 其它的集成开发环境和执行环境

很多的软件开发专家事实上会选择在他们喜欢的文本编辑器中编写代码，比如 vi(m) 或者 emacs。除了这些和上面提到到的集成开发环境，还有大量的开源和商业的集成开发环境，下面是个简短的列表：

开源  
IDLE (在 Python 发行版中自带)：<http://python.org/idle/>

PythonWin + Win32 Extensions：<http://starship.python.net/crew/skippy/win32>

IPython (增强的交互式 Python)：<http://ipython.scipy.org>

IDE Studio (IDLE 以及更多)：<http://starship.python.net/crew/mike/Idle>

Eclipse：<http://pydev.sf.net>和 <http://eclipse.org/>

商业

WingWare 开发的 WingIDE Python 集成开发环境：<http://wingware.com/>

ActiveState 开 发 的 Komodo 集 成 开 发 环 境 ：  
<http://activestate.com/Products/Komodo>

通用集成开发环境列表：  
<http://wiki.python.org/moin/IntegratedDevelopmentEnvironments>

**核心提示： 运行本书中的代码实例**

**在本书中，你会发现很多的 Python 脚本样例，可以从本书的网站上下载。但是当你运行它们的时候，请记住这些代码是设计用来从命令行（DOS 命令行窗口或 Unix shell）或者集成开发环境执行的。如果你在使用 Win32 系统，双击 Python 程序会打开DOS窗口，但是在脚本执行完毕后就会关闭，所以你可能看不到输出结果。如果你遇到了这种情况，就直接打开 DOS窗口，从命令行中运行相关的脚本，或者在集成开发环境中执行脚本。另外一种办法，就是在脚本的最后一行后面添加 raw\_input()语句，这样就可以保持窗口开着，直到你按下回车键才关闭。**

## 1.6 Python文档

Python官方网站：<http://python.org>

本书的网站：<http://corepython.com>

## 1.7 Python与其他语言的比较

众说纷纭，无从判断。每门语言都有他的优势和缺点，根据项目实际情况选择合适的开发语言。

## 1.8 其它实现（略）

## 1.9 练习

1-1．安装Python。请检查 Python 是否已经安装到你的系统上，如果没有，请下载并 安装它！

1-2．执行 Python。有多少种运行 Python 的不同方法？你喜欢哪一种？为什么？

**答**：3种。（1）交互式解释器，（2）运行Python脚本，（3）在集成开发环境中的图形界面上执行Python语句。喜欢运行Python脚本。

1-3. Python 标准库。

（a）请找到系统中 Python 执行程序的安装位置和标准库模块的安装位置

（b）看看标准库里的一些文件，比如 string.py。这会帮助你适应阅读 Python 脚本。

（略）

后面习题（略）

# 第2章 Python起步

## 2.1 介绍

本章将对 Python 的主要特性做一个快速介绍，这样你就可以借助以前的编程经验识别出 熟悉的语言结构，并立刻将 Python 付诸使用。虽然细节内容会在后续的章节中逐一讲解，但 是对整体的了解可以让你迅速融入到 Python 中。阅读本章的最好的方法就是在你的电脑上打 开 Python 解释器，尝试书中的示例， 当然你也可以随心所欲的自己做实验。

我们已经在第一章和练习 1–4 中介绍了如何启动 Python 解释器。 在所有的交互示例中， 你会看到 Python 的主提示符( >>> )和次提示符( ... )。主提示符是解释器告诉你它在等待你 输入下一个语句，次提示符告诉你解释器正在等待你输入当前语句的其它部分。

Python 有两种主要的方式来完成你的要求：语句和表达式（函数、算术表达式等）。相信 大部分读者已经了解二者的不同，但是不管怎样，我们还是再来复习一下。语句使用关键字来 组成命令，类似告诉解释器一个命令。你告诉 Python 做什么，它就为你做什么，语句可以有输 出，也可以没有输出。下面我们先用 print 语句完成程序员们老生常谈第一个编程实例，Hello World:

>>> print 'Hello World!'

Hello World!

而表达式没有关键字。它们可以是使用数学运算符构成的算术表达式，也可以是使用括号 调用的函数。它们可以接受用户输入，也可以不接受用户输入，有些会有输出，有些则没有。 （在 Python 中未指定返回值的函数会自动返回 None， 等价于 NULL）下面举一个例子， 函数 abs() 接受一个数值输入，然后输出这个数值的绝对值：  
>>> abs(-4)  
4  
>>> abs(4)  
4

本章中我们将分别介绍语句和表达式。我们先来研究 print 语句。

## 2.2 输入/输出

2.2.1 程序输出，print语句

有些语言， 比如 C, 通过函数输出数据到屏幕，例如函数 printf()。然而在 Python 和大 多数解释执行的脚本语言，使用语句进行输出。很多的 shell 脚本语言使用 echo 命令来输出程 序结果。

**核心笔记：在交互式解释器中显示变量的值**

**通常当你想看变量内容时，你会在代码中使用 print语句输出。不过在交互式解释器中，你可以用print语句显示变量的字符串表示，或者仅使用变量名查看该变量的原始值。在下面的例子中，我们把一个字符串赋值给变量 myString，先用 print来显示变量的内容，之后用变量名称来显示。  
>>> myString = 'Hello World!'  
>>> print myString  
Hello World!  
>>> myString  
'Hello World!'**

**注意：在仅用变量名时，输出的字符串是被用单引号括起来了的。这是为了让非字符串对 象也能以字符串的方式显示在屏幕上－－即它显示的是该对象的字符串表示，而不仅仅是字符 串本身。引号表示你刚刚输入的变量的值是一个字符串。等你对 Python 有了较深入的了解之后，你就知道 print 语句调用 str()函数显示对象， 而交互式解释器则调用 repr()函数来显示对象。**

下划线(\_)在解释器中有特别的含义， 表示最后一个表达式的值。 所以上面的代码执行之后， 下划线变量会包含字符串：  
>>> \_  
Hello World!

Python的print语句，与字符串格式运算符(%)结合使用，可实现字符串替换功能，这一点和 C语言中的printf()函数非常相似：  
>>> print "%s is number %d!" % ("Python", 1)  
Python is number 1!

%s表示由一个字符串来替换，而%d表示由一个整数来替换，另外一个很常用的就是%f，它 表示由一个浮点数来替换。我们会在本章中看到更多类似的例子。Python非常灵活，所以即使你将数字传递 %s，也不会像其他要求严格的语言一样引发严重后果。参阅章节 6.4.1 以了解更多关于字符串格式运算符的信息。Print语句也支持将输出重定向到文件。这个特性是从 Python2.0开始新增的。符号>>用来重定向输出，下面这个例子将输出重定向到标准错误输出：

import sys  
print >> sys.stderr, 'Fatal error: invalid input!'  
import sys  
print >> sys.stderr, 'Fatal error: invalid input!'

下面是一个将输出重定向到日志文件的例子：  
logfile = open('/tmp/mylog.txt', 'a')  
print >> logfile, 'Fatal error: invalid input!'  
logfile.close()

2.2.2 程序输入，raw\_input()内建函数

从用户那里得到数据输入的最容易的方法是使用raw\_input()内建函数。它读取标准输入，并将读取到的数据赋值给指定的变量。你可以使用int()内建函数将用户输入的字符串转换为整数。

>>> user = raw\_input('Enter login name: ')  
Enter login name: root

>>> print 'Your login is:', user  
Your login is: root 上面这个例子只能用于文本输入。 下面是输入一个数值字符串(并将字符串转换为整数） 的例子：  
>>> num = raw\_input('Now enter a number: ')  
Now enter a number: 1024  
>>> print 'Doubling your number: %d' % (int(num) \* 2)  
Doubling your number: 2048 内建函数 int()将数值字符串转换成整数值，这样才可以对它进行数学运算。参阅章节 6.5.3 以了解更多有关内建函数 raw\_input()的知识。

**核心笔记：从交互式解释器中获得帮助**

在学习 Python 的过程中，如果需要得到一个生疏函数的帮助，只需要对它调用内建函数 help()。通过用函数名作为 help()的参数就能得到相应的帮助信息：  
>>> help(raw\_input)  
Help on built-in function raw\_input in module \_\_builtin\_\_:  
raw\_input(...)  
raw\_input([prompt]) -> string

从标准输入读取一个字符串并自动删除串尾的换行字符。如果用入键入了EOF字符（Unix: Ctrl+D, Windows: Ctrl+Z+回车）， 则引发EOFError,在Unix平台，只要可用，就使用 GNU readline库。如果提供提示字符串参数，则显示该字符串并自动删去字符串末尾的换行字符。

核心风格： 一直在函数外做用户交互操作 新手在需要显示信息或得到用户输入时， 很容易想到使用 print 语句和 raw\_input()内 建函数。不过我们在此建议函数应该保持其清晰性， 也就是它只应该接受参数，返回结果。从 用户那里得到需要的数据， 然后调用函数处理， 从函数得到返回值， 然后显示结果给用户。 这 样你就能够在其它地方也可以使用你的函数而不必担心自定义输出的问题。这个规则的一个例外是，如果函数的基本功能就是为了得到用户输出， 或者就是为了输出信息，这时在函数体使 用 print 语句或 raw\_input() 也未尝不可。 更重要的， 将函数分为两大类， 一类只做事， 不 需要返回值（比如与用户交互或设置变量的值）， 另一类则执行一些运算，最后返回结果。如 果输出就是函数的目的，那么在函数体内使用 print 语句也是可以接受的选择。

## 2.3 注释

和大部分脚本及Unix-shell语言一样，Python也使用#符号标示注释，从#开始，直到一行结束的内容都是注释。

>>> **# one comment**  
... print 'Hello World!' **# another comment**  
Hello World!

有一种叫做文档字符串的特别注释。你可以在模块、类或者函数的起始添加一个字符串，起到在线文档的功能，这是Java程序员非常熟悉的一个特性。  
def foo():  
"**This is a doc string.**"  
return True

与普通注释不同，文档字符串可以在运行时访问，也可以用来自动生成文档。

## 2.4 运算符

和其他绝大多数的语言一样，Python 中的标准算术运算符以你熟悉的方式工作

+ - \* / // % \*\*

加、减、乘、除和取余都是标准运算符。Python 有两种除法运算符，单斜杠用作传统除法， 双斜杠用作浮点除法（对结果进行四舍五入）。传统除法是指如果两个操作数都是整数的话， 它将执行是地板除(取比商小的最大整数)（关于“地板除”请参考第5章——译者注），而浮 点除法是真正的除法，不管操作数是什么类型，浮点除法总是执行真正的除法。你可以在第5 章（数字）学到更多有关传统除法、真正的除法及浮点除法的知识。

还有一个乘方运算符， 双星号(\*\*)。尽管我们一直强调这些运算符的算术本质， 但是请注意对于其它数据类型，有些运算符是被重载了，比如字符串和列表。让我们看一个例子：

>>> print -2 \* 4 + 3 \*\* 2 就象你看到的， 运算符的优先级和你想象的一样： + 和 － 优先级最低， \*, /, //， % 优先级较高，单目运算符 + 和 - 优先级更高，乘方的优先级最高。(3 \*\* 2) 首先求值， 然后是 (-2 \* 4)， 然后是对两个结果进行求和。 Python 当然也有标准比较运算符， 比较运算根据表达式的值的真假返回布尔值： < <= > >= == != <> 试一试，看看这些比较运算会得到什么结果：  
>>> 2 < 4  
True  
>>> 2 == 4  
False  
>>> 2 > 4  
False  
>>> 6.2 <= 6  
False  
>>> 6.2 <= 6.2  
True  
>>> 6.2 <= 6.20001  
True Python 目前支持两种“不等于”比较运算符， **!=** 和 <> ， 分别是 C 风格和 ABC/Pascal 风格。目前后者慢慢地被淘汰了， 所以我们推荐使用前者。 Python 也提供了逻辑运算符： **and or not**

使用逻辑运算符可以将任意表达式连接在一起，并得到一个布尔值：  
>>> 2 < 4 and 2 == 4  
False  
>>> 2 > 4 or 2 < 4  
True  
>>> not 6.2 <= 6  
True  
>>> 3 < 4 < 5  
True

最后一个例子在其他语言中通常是不合法的，不过 Python 支持这样的表达式， 既简洁又 优美。它实际上是下面表达式的缩写：

>>> 3 < 4 and 4 < 5

参阅章节 4.5 种可以得到更多有关 Python 运算符的信息。

**核心风格： 合理使用括号增强代码的可读性， 在很多场合使用括号都是一个好主意，而没 用括号的话，会使程序得到错误结果，或使代码可读性降低，引起阅读者困惑。。括号在 Python 语言中不是必须存在的， 不过为了可读性， 使用括号总是值得的。任何维护你代码的人会感 谢你， 在你再次阅读自己的代码时，你也会感谢你自己。**

## 2.5 变量与赋值

Python 中变量名规则与其它大多数高级语言一样， 都是受C语言影响（或者说这门语言本身就是C语言写成的）。变量名仅仅是一些字母开头的标识符－－所谓字母开头－－意指大写或小写字母，另外还包括下划线(\_).其它的字符可以是数字，字母，或下划线。Python变量名是大小写敏感的，也就是说变量”cAsE”与”CaSe”是两个不同的变量。Python是动态类型语言，也就是说不需要预先声明变量的类型。变量的类型和值在赋值那一刻被初始化。变量赋值通过等号来执行。  
>>> counter = 0  
>>> miles = 1000.0  
>>> name = 'Bob'  
>>> counter = counter + 1  
>>> kilometers = 1.609 \* miles  
>>> print '%f miles is the same as %f km' % (miles, kilometers)  
1000.000000 miles is the same as 1609.000000 km

上面是五个变量赋值的例子。第一个是整数赋值，第二个是浮点数赋值，第三个是字符串赋值，第四个是对一个整数增 1， 最后一个是浮点乘法赋值。

Python 也支持增量赋值，也就是运算符和等号合并在一起， 看下面的例子:  
n = n \* 10 将上面的例子改成增量赋值方式就是：  
n \*= 10

**Python 不支持C语言中的自增1和自减1运算符**，这是因为+和－也是单目运算符，Python会将--n解释为-(-n)从而得到n,同样++n的结果也是n.

## 2.6 Python类型

2.6.1 数字

Python 支持五种基本数字类型，其中有三种是整数类型。

int (有符号整数)  
long (长整数)  
bool (布尔值)  
float (浮点值)  
complex (复数)

下面是一些例子： Python 中有两种有趣的类型是Python的长整型和复数类型。请不要将 Python的长整数与C语言的长整数混淆。Python的长整数所能表达的范围远远超过C语言的长整数，事实上，Python长整数仅受限于用户计算机的虚拟内存总数。如果你熟悉Java, Python的长整数类似于Java中的BigInteger类型。

从长远来看， 整型与长整型正在逐步统一为一种整数类型。从Python2.3开始，再也不会报整型溢出错误，结果会自动的被转换为长整数。在未来版本的Python中，两种整数类型将会无缝结合，长整数后缀“L”也会变得可有可无。

布尔值是特殊的整数。尽管布尔值由常量True和False来表示，如果将布尔值放到一个数值上下文环境中（比方将True与一个数字相加），True会被当成整数值1，而False则会被当成整数值0。复数（包括-1的平方根,即所谓的虚数）在其它语言中通常不被直接支持（一般通过类来实现）。

其实还有第六种数字类型，decimal，用于十进制浮点数。不过它并不是内建类型，你必须先导入decimal模块才可以使用这种数值类型。由于需求日渐强烈，Python2.4增加了这种类型。举例来说，由于在二进制表示中有一个无限循环片段，数字1.1无法用二进制浮点数精确表示。因此，数字1.1实际上会被表示成：  
>>> 1.1  
1.1000000000000001  
>>> print decimal.Decimal('1.1')  
1.1

第五章详细将介绍所有的数字类型

2.6.2 字符串

Python 中字符串被定义为引号之间的字符集合。Python支持使用成对的单引号或双引号，三引号（三个连续的单引号或者双引号）可以用来包含特殊字符。使用索引运算符([])和切片运算符([:])可以得到子字符串。

字符串有其特有的索引规则：第一个字符的索引是0，最后一个字符的索引是－1加号(+)用于字符串连接运算，星号(\*)则用于字符串重复。下面是几个例子：  
>>> pystr = 'Python'  
>>> iscool = 'is cool!'  
>>> pystr[0]  
'P'  
>>> pystr[2:5]  
'tho'  
>>> iscool[:2]  
'is'  
>>> iscool[3:]  
'cool!'  
>>> iscool[-1]  
'!'  
>>> pystr + iscool  
'Pythonis cool!'  
>>> pystr + ' ' + iscool  
'Python is cool!'  
>>> pystr \* 2  
'PythonPython'  
>>> '-' \* 20  
'--------------------'  
>>> pystr = '''python  
... is cool'''  
>>> pystr  
'python\nis cool'  
>>> print pystr  
python  
is cool  
>>>

你可以在第六章学到更多有关字符串的知识。

2.6.3 列表和元组

可以将列表和元组当成普通的“数组”，它能保存任意数量任意类型的 Python 对象。和数组一样，通过从 0 开始的数字索引访问元素，但是列表和元组可以存储不同类型的对象。 列表和元组有几处重要的区别。列表元素用中括号([])包裹，元素的个数及元素的值可 以改变。元组元素用小括号(())包裹，不可以更改（尽管他们的内容可以）。元组可以看成是只读的列表。通过切片运算([]和[:])可以得到子集，这一点与字符串的使用方法一样。

>>> aList = [1, 2, 3, 4]  
>>> aList  
[1, 2, 3, 4]  
>>> aList[0]  
1  
>>> aList[2:]  
[3, 4]  
>>> aList[:3]  
[1, 2, 3]  
>>> aList[1] = 5  
>>> aList  
[1, 5, 3, 4] 元组也可以进行切片运算，得到的结果也是元组（不能被修改）：  
>>> aTuple = ('robots', 77, 93, 'try')  
>>> aTuple  
('robots', 77, 93, 'try')  
>>> aTuple[:3]  
('robots', 77, 93)  
>>> aTuple[1] = 5  
Traceback (innermost last):  
File "<stdin>", line 1, in ?  
TypeError: object doesn't support item assignment 你可以在第六章学到更多有关列表、元组以及字符串的知识。

2.6.4 字典

字典是 Python 中的映射数据类型，工作原理类似 Perl 中的关联数组或者哈希表，由键- 值(key-value)对构成。几乎所有类型的 Python 对象都可以用作键，不过一般还是以数字或者 字符串最为常用。 值可以是任意类型的 Python 对象，字典元素用大括号({ })包裹。

>>> aDict = {'host': 'earth'} # create dict  
>>> aDict['port'] = 80 # add to dict  
>>> aDict  
{'host': 'earth', 'port': 80}  
>>> aDict.keys()  
['host', 'port']  
>>> aDict['host']  
'earth'  
>>> for key in aDict:  
... print key, aDict[key]  
...  
host earth  
port 80 在第七章中会详细讲解字典。

2.7 缩进

2.8 循环与条件

2.9 文件

2.10 错误

2.11 函数

2.12 类

2.13 模块

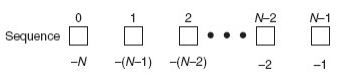
第6章 序列：字符串、列表和元组

接下来我们要研究这样一些Python的类型，它们的成员有序排列的，并且可以通过下标偏移量访问到它的一个或者几个成员，这类Python类型统称为序列，包括下面这些：字符串(普通字符串和unicode字符串)，列表，和元组类型。

因为这些类型其实都是由一些成员共同组成的一个序列整体，所以我们把它们统称为序列，比如说，一个字符串是由一些字符(尽管Python并没有显式的定义字符这个类型)组成的序列，那么“Hello”这个字符串的第一个字符就是“H"，第二个字符就是‘e’……,同样的，列表类型和元组类型就是其他一些Python对象所组成的序列。首先我们来熟悉一下适用于所有序列类型的操作符和内建函数(BIFs)，

6.1 序列

序列类型有着相同的访问模式：它的每一个元素可以通过指定一个偏移量的方式得到。而多个元素可以通过切片操作的方式一次得到，切片操作会在接下来的内容中讲到。下标偏移量是从0开始到总元素数-1结束--之所以要减1是因为我们是从0开始计数的。图 6-1 阐述 了序列的元素是如何存储的。



N == 序列的长度 == len(sequence)

图6-1 有多少可以保存并可以被访问的序列元素

6.1.1 标准类型操作符

标准类型操作符(参见 4.5 节)一般都能适用于所有的序列类型。当然，如果作复合类型的 对象比较的话，这样说可能需要有所保留，不过其他的操作绝对是完全适用的。

6.1.2 序列类型操作符

**成员关系操作符 (in, not in)**

成员关系操作符使用来判断一个元素是否属于一个序列的。比如对字符串类型来说，就是 判断一个字符是否属于这个字符串，对和元组类型来说，就代表了一个对象是否属于该对象序 列。in/not in 操作符的返回值一般来讲就是 True/False，满足成员关系就返回 True，否则返 回 False。该操作符的语法如下：

obj [not] in sequence

表6.1列出了对所有序列类型都适用的操作符。操作符是按照优先级从高到底的顺序排列的。

表6.1序列类型操作符

|  |  |
| --- | --- |
| 序列操作符 | 作用 |
| seq[index] | 获得下标为index的元素 |
| seq[index1:index2] | 获得下标从 ind1 到 ind2 间的元素集合 |
| seq \* expr | 序列重复 expr 次 |
| seq1 + seq2 | 连接序列 seq1 和 seq2 |
| obj in seq | 判断 obj 元素是否包含在 seq 中 |
| obj not in seq | 判断 obj 元素是否不包含在 seq 中 |

**连接操作符( + )**

这个操作符允许我们把一个序列和另一个相同类型的序列做连接。语法如下:

sequence1 + sequence2

该表达式的结果是一个包含 sequence1 和 sequence2 的内容的新序列.注意，这种方式看 起来似乎实现了把两个序列内容合并的概念，但是这个操作不是最快或者说最有效的。对字符串来说，这个操作不如把所有的子字符串放到一个列表或可迭代对象中，然后调用一个join方法来把所有的内容连接在一起节约内存;类似地，对列表来说，我们推荐读者用列表类型的 extend()方法来把两个或者多个列表对象合并。当你需要简单地把两个对象的内容合并，或者说不能依赖于可变对象的那些没有返回值(实际上它返回一个None)的内建方法来完成的时候时，连接操作符还是很方便的一个选择。下面的切片操作可以视作这些情况的例子。

**重复操作符 ( \* )**

当你需要需要一个序列的多份拷贝时，重复操作符非常有用，它的语法如下：

sequence \* copies\_int

copies\_int 必须是一个整数(1.6节里面有讲到，不能是长整数)。像连接操作符一样，该操作符返回一个新的包含多份原对象拷贝的对象。

**切片操作符 ( [], [:], [::] )**

简单地讲，所谓序列类型就是包含一些顺序排列的对象的一个结构。你可以简单的用方括号 加一个下标的方式访问它的每一个元素，或者通过在方括号中用冒号把开始下标和结束下标分开的方式来访问一组连续的元素。下面我们将详细的讲解提到的这两种方式。序列类型是其元素被顺序放置的一种数据结构类型,这种方式允许通过指定下标的方式来获得某一个数据元素，或者通过指定下标范围来获得一组序列的元素。这种访问序列的方式叫做切片，我们通过切片操作符就可以实现我们上面说到的操作。访问某一个数据元素的语法如下：

sequence[index]

sequence 是序列的名字，index 是想要访问的元素对应的偏移量。偏移量可以是正值,范围 从 0 到偏移最大值(比序列长度少1)，用len()函数（下一节会讲），可以得到序列长度，实际的范围是 0 <= inde <= len(sequece)-1。另外，也可以使用负索引，范围是-1到序列的负长度，-len(sequence), -len(sequence) <= index < = -1.正负索引的区别在于正索引以序列的开始为起点，负索引以序列的结束为起点。试图访问一个越界的索引会引发一个如下的异常:

>>> names = ('Faye', 'Leanna', 'Daylen')  
>>> print names[4]  
Traceback (most recent call last): File "<stdin>", line 1, in ?  
IndexError: tuple index out of range

因为Python是面向对象的，所以你可以像下面这样直接访问一个序列的元素(不用先把它赋值给一个变量):

>>> print ('Faye', 'Leanna', 'Daylen')[1]  
Leanna

这个特性在你调用一个返回值是序列类型的函数，并且你只对返回的序列中的一个或某几个元素感兴趣时特别有用。那么我们如何才能一次得到多个元素呢？其实这跟访问某一个单一元素一样简单，只要简单的给出开始和结束的索引值，并且用冒号分隔就可以了，其语法如下:

sequence[starting\_index:ending\_index]

通过这种方式我们可以得到从起始索引到结束索引（**不包括结束索引对应的元素**）之间的一"片"元素。起始索引和结束索引都是可选的，如果没有提供或者用None作为索引值,切片操作会从序列的最开始处开始，或者直到序列的最末尾结束。

**用步长索引来进行扩展的切片操作**

序列的最后一个切片操作是扩展切片操作，它多出来的第3个索引值被用做步长参数。你可以把这个参数看成跟内建函数range()里面的步长参数或者类似于C/C++，Perl，PHP和Java语言里面for语句中的步长参数一样来理解。Python的虚拟机里面其实很早就有了扩展切片操作，只不过以前需要通过扩展的方式来使用。Jython 也支持这个语法(以前叫 JPython)。

以下是几个例子:  
>>> s = 'abcdefgh'  
>>> s[::-1] # 可以视作"翻转"操作  
'hgfedcba'

>>> s[::2] # 隔一个取一个的操作  
'aceg'

**切片索引的更多内容**

切片索引的语法要比简单的单一元素索引灵活的多。开始和结束素引值可以超过字符串的长度。换句话说，起始索引可以小于 0,而对于结束索引，即使索引值为100的元素并不存在也 不会报错，简单地说，即使用100来作为一个长度不到100的序列的结束索引也不会有什么问 题，例子如下：  
>>> ('Faye', 'Leanna', 'Daylen')[-100:100]  
('Faye', 'Leanna', 'Daylen')

有这么一个问题：有一个字符串，我们想通过一个循环按照这样的形式显示它：每次都把位于最后的一个字符砍掉，下面是实现这个要求的一种方法：  
>>> s = 'abcde'  
>>> i = -1  
>>> for i in range(-1, -len(s), -1):  
... print s[:i]  
... abcd  
abc  
ab  
a

可是，该如何在第一次迭代的时候显示整个字符串呢？是否有一种方法可以不用在整个循环之前加入一个额外的print语句呢？我们该如何定义一个索引，来代表整个的序列呢？事实 上在个以负数作为索引的例子里是没有一个真正能解决这个问题的方法的，因为-1已经是“最 小”的索引了。我们不可能用0来作为索引值，因为这会切片到第一个元素之前而什么都不会显示:

>>> s[:0]  
''

我们的方案是使用另一个小技巧:用None作为索引值，这样一来就可以满足你的需要，比如说，在你想用一个变量作为索引来从第一个到遍历最后一个元素的时候：

>>> s = 'abcde'  
>>> for i in [None] + range(-1, -len(s), -1):  
... print s[:i]  
...  
abcde  
abcd  
abc  
ab  
a

现在这个程序符合我们的要求了。在进行下面的内容之前，必须指出，似乎还可以先创建 一个只包含 None 的列表,然后用 extend()函数把 range()的输出添加到这个列表,或者先建立 range()输出组成的列表然后再把 None 插入到这个列表的最前面，然后对这个列表进行遍历， 但是可变对象的内建函数 extend()根本就没有返回值，所以这个方法是行不通的:  
>>> for i in [None].extend(range(-1, -len(s), -1)):  
... print s[:i]  
...  
Traceback (most recent call last):  
File "<stdin>", line 1, in ?  
TypeError: iteration over non-sequence

这个错误发生的原因是[None].extend(...)函数返回 None , None 既不是序列类型也不是 可迭代对象。这种情况下使用上面提到的的列表连接操作来实现是唯一不需要添加额外代码的方法。

6.1.3内建函数（BIFs）

在讲解序列类型的内建函数之前，有一点需要说明，序列本身就内含了迭代的概念，之所以会这样，是因为迭代这个概念就是从序列，迭代器，或者其他支持迭代操作的对象中泛化得来的。由于 Python 的 for 循环可以遍历所有的可迭代类型，在(非纯序列对象上)执行 for 循环时就像在一个纯序列对象上执行一样。而且 Python 的很多原来只支持序列作为参数的内建函数现 在也开始支持迭代器或者或类迭代器了.我们把这些类型统称为"可迭代对象".在这一章里我们 会详细的讨论跟序列关系紧密的内建函数(BIF). 在第八章"条件判断和循环"里面将讨论针对"在循环中迭代"这种情况的内建函数(BIF).

类型转换

内建函数 list(),str()和tuple()被用做在各种序列类型之间转换。你可以把它们理解成其他语言里面的类型转换，但是并没有进行任何的转换。这些转换实际上是工厂函数（在第4章介绍），将对象作为参数，并将其内容（浅）拷贝到新生成的对象中。表 6.2 列出了适用于序列类型的转换函数。

表 6.2 序列类型转换工厂函数

|  |  |
| --- | --- |
| 函数 | 含义 |
| list(iter) | 把可迭代对象转换为列表 |
| str(obj) | 把 obj 对象转换成字符串(对象的字符串表示法) |
| unicode(obj) | 把对象转换成 Unicode 字符串(使用默认编码) |
| basestring() | 抽象工厂函数,其作用仅仅是为 str 和 unicode 函数提供父类，所以不能被 实例化，也不能被调用(详见第 6.2 节) |
| tuple(iter) | 把一个可迭代对象转换成一个元组对象 |

我们又用了一次“转换”这个词。不过，为什么Python里面不简单地把一个对象转换成另一个对象呢？回过头看一下第4章就会知道，一旦一个 Python 的对象被建立，我们就不能更改其身份或类型了。如果你把一个列表对象传给list（）函数，便会创建这个对象的一个浅拷贝，然后将其插入新的列表中。同样地，在做连接操作和重复操作时，我们也会这样处理。 所谓浅拷贝就是只拷贝了对对象的索引，而不是重新建立了一个对象！如果你想完全的拷 贝一个对象(包括递归，如果你的对象是一个包含在容器中的容器),你需要用到深拷贝，关于浅 拷贝和深拷贝的更多信息会在本章的末尾讲到。 str()函数在需要把一个对象的可打印信息输出时特别有用，不仅仅是对序列类型，对其他 类型的对象同样如此.Unicode()是 str()函数的 unicode 版本， 它跟 str()函数基本一样.list()和 tuple()函数在列表类型和元组类型的互换时非常有用.不过， 虽然这些函数也适用于 string 类型(因为 string 类型也是序列的一种),但是在 string 类型上应用 tuple()和 list()函数却得 不到我们通常希望的结果。

Operational Python 为序列类型提供以下可操作 BIFs(见表 6.3).注意，len(),reversed()和 sum()函 数只能接受序列类型对象作为参数， 而剩下的则还可以接受可迭代对象做为参数， 另外， max() 和 min()函数也可以接受一个参数列表.

表 6.3 序列类型可用的内建函数 函数名 功能 enumerate(iter) a 接受一个可迭代对象作为参数，返回一个 enumerate 对象(同 时也是一个迭代器),该对象生成由 iter 每个元素的 index 值 和 item 值组成的元组(PEP 279) len(seq) 返回 seq 的长度 max(iter,key=None) or max(arg0,arg1...,key=None) b 返回 iter 或(arg0,arg1,...)中的最大值， 如果指定了 key, 这个 key 必须是一个可以传给 sort()方法的,用于比较的回 调函数. min(iter, key=None) or min(arg0, arg1.... key=None) b 返回 iter 里面的最小值;或者返回(arg0,arg2,...)里面的最小值;如果指定了 key,这个 key 必须是一个可以传给 sort()方法的,用于比较的回调函数.

reversed(seq) c 接受一个序列作为参数,返回一个以逆序访问的迭代器(PEP 322) sorted(iter, func=None, key=None, reverse=False) c 接受一个可迭代对象作为参数，返回 一个有序的列表;可选参数 func,key 和 reverse 的含义跟 list.sort()内建函数的参数含义一 样. sum(seq, init=0) a 返 回 seq 和 可 选 参 数 init 的 总 和 , 其 效 果 等 同 于 reduce(operator.add,seq,init) zip([it0, it1,... itN]) d 返回一个列表，其第一个元素是 it0,it1,...这些元素的第 一个元素组成的一个元组，第二个...,类推. a. Python2.3 新增 b. 从 Python2.5 开始支持关键字参数 c. Python2.4 开始支持 d. Python2.0 加入，Python2.4 加强 我们将分别在每个序列的章节里面提供使用这些函数的例子.  
6.2 字符串

字符串类型是 Python 里面最常见的类型.我们可以简单地通过在引号间包含字符的方式 创建它.Python 里面单引号和双引号的作用是相同的，这一点 Python 不同于其他类 Shell 的脚 本语言，在这些脚本语言中，通常转义字符仅仅在双引号字符串中起作用，在单一号括起的字 符串中不起作用。Python 用"原始字符串"操作符来创建直接量字符串,所以再做区分就没什么 意义了。其他的语言，比如 C 语言里面用单引号来标示字符，双引号标示字符串,而在 Python 里面没有字符这个类型.这可能是双引号和单引号在 Python 里面被视作一样的的另一个原因。几乎所有的 Python 应用程序都会某种方式用到字符串类型.字符串是一种直接量或者说是一种 标量,这意味着 Python 解释器在处理字符串时是把它作为单一值并且不会包含其他 Python 类型 的。字符串是不可变类型,就是说改变一个字符串的元素需要新建一个新的字符串.字符串是由 独立的字符组成的，并且这些字符可以通过切片操作顺序地访问。

根据在 2.2 章节里面对类型和类的概念进行的统一，Python 实际上有 3 类字符串.通常意 义的字符串 (str)和 Unicode 字符串(unicode)实际上都是抽象类 basestring 的子类.这个 basestring 是不能实例化的，如果你试图实例化一个 basestring 类,你会得到以下报错信息:  
>>> basestring('foo')  
Traceback (most recent call last): File "<stdin>", line 1, in <module>  
TypeError: The basestring type cannot be instantiated

第16章 网络编程-练习

1 套接字。面向连接和无连接有什么区别？

2 客户/服务器架构。用你自己的语言描述这个架构，并给出几个例子。

3 套接字。TCP和UDP中，哪一种服务器在接受连接后，把连接交给不同的套接字处理与客户的通讯。

4 客户。修改TCP（tsTclnt.py）和UDP（tsTclnt.py）客户端，让服务器的名字不要在代码里写死，要允许用户指定一个主机名和端口，只有在两个值都没有输入的时候，才使用默认值。

5 Intenet网络和套接字。找到Python Libaray Reference中7.2.2节的示例，TCP客户/服务器程序，实现它并让它运行起来。先运行服务器，然后是客户端。源代码的一个在线版本可以在这里找到：

<http://www.python.org/doc/current/lib/Socket_Example.html>

你认为这个服务器太无聊了，决定要修改服务器，让它能识别以下命令：

date 服务器将返回它的当前时间，i.e.，time.ctime(time.time())os得到操作系统的信息（os.name）

ls 得到当前目录的文件列表 (提示: os.listdir()可以得到目录列表，os.curdir 能得到当前目录) 附加题: 要能接受“ls dir”指令，并返回 dir 目录的文件列表。

做这个作业的时候，你不一定要有网络——你的机器可以跟自己通讯。注：在服务器退出后，要清除绑定后才能再次运行。否则，有可能得碰到“端口已经被使用”（“port already bound”）的错误信息。操作系统一般会在 5 分钟内清除绑定。所以，请耐心等待。

6. 日期时间服务。使用 socket.getservbyname()函数得到 UDP 协议中， “daytime”服务所 对 应 的 端 口 。 请 参 考 getservbyname() 函 数 的 文 档 ， 查 阅 详 细 的 语 法 。（ 即 ：socket.getservbyname.\_\_doc\_\_）。现在，写一个程序发送一个随便什么数据过去，等待回答。一旦你收到了服务器的信息，显示到屏幕上。

7. 半双工聊天。创建一个简单的，半双工的聊天程序。“半双工”的意思是当创建一个连接，服务启动的时候，只有一个人可以打字，另一个人只有在等到有消息通知他输入消息时，才能说话。一旦消息发送出去后，要等到有回复了才能发送下一条消息。一个人是服务端，另一个人是客户端。

8. 全双工聊天。修改你刚才的程序，改成全双工，即两个人可以独立地发送和接收消息。

9. 多用户全双工聊天。再次修改你的程序，把聊天服务改成支持多用户版本。

10. 多用户，多房间全双工聊天。现在把聊天服务改成支持多个用户，多个房间。

11. 网页客户。写一个 TCP 客户，连到你最喜欢的网站的 80 端口（去掉“http://”和其它的后缀信息，只用主机名）。一旦创建了一个连接，发送 HTTP 命令字符串“GET /\n”，把服务器返回的所有数据写到一个文件中。（GET 命令用于得到网页， “/”表示要得到的文件， “\n”把命令发送到服务器）。检查得到的文件的内容，它是什么？怎么检查你得到的数据是否正确？（注：你可能要在命令后加一个或是两个回车，一般来说，一个就可以了。 ）

12. 休眠服务器。创建一个“休眠”服务器，客户可以要求要“休眠”几秒钟。服务器就去做休眠的操作。休眠结束后，返回一个消息给客户，表示结束。客户在收到消息的时候，应该刚好等待了指定的时间。这就是一个简单的“远程过程调用”（“remote procedure call”），即客户发送一个指令，网络另一边的远程的机器执行这个命令。

13. 名字服务器。设计并实现一个名字服务器。这个服务器负责维护一个主机名－端口对的数据库，以及一个描述这个服务器提供的服务的字符串。 选择一个或几个服务器到你的名字服务器上进程“注册”。 （注意，这时，这些服务器是名字服务器的客户）。每一个客户在启动的时候，都不知道它们想要找的服务器的信息。名字服务器的客户也是这样。这些客户应该发送一个请求到名字服务器，说明它们想要得到什么服务。名字服务器返回一个主机名－端口对给客户，客户这时就可以连到合适的服务器来处理它的请求。

附加题：

(1) 在名字服务器中，加入对常用请求的缓冲。

(2) 在名字服务器中，加入日志功能，记录下哪个服务器注册了，客户在请求哪一个服务。

(3) 名字服务器应该周期性地“ping”这些注册了的服务器的对应端口，以确定这些服务器还在运行中。在连续数次 ping 失败后，就把这个服务器从列表中删除。你可以实现一些真实的服务，来注册到你的名字服务器上，或者也可以使用一些哑服务（它们根本不对请求做应答）。

14. 错误检查和优美的退出。 本章中， 我们所有客户和服务器的例子代码都没有做错误检查。我们没有检查用户是否按下了^C 来退出服务，或^D 来结束客户输入，也没有检查 raw\_input()函数得到的输入的合法性，也没有检查网络错误。由于这些弱点，我们很可能会在退出程序的时候，没有关闭套接字，也有可能会丢失数据。选择一对客户/服务器例子，加入足够的错误检查，让程序能正常的退出。比方说会关闭网络连接。

15. 异步和 SocketServer。选取 TCP 服务器例子，使用某一个混合类(mix-in)，让你的程序成为一个异步服务器。测试你的服务器，创建并同时运行多个客户，在服务器的输出里查看你的服务器是否在同时响应多个请求。

16. \*扩展 SocketServer 类。在 SocketServer TCP 服务器代码中，我们不能使用原来的 TCP客户，要做修改。这是因为 SocketServer 类在多个请求之间不保持连接。(a) 从 TCPServer 和StreamRequestHandler 中派生出新的类，重新设计服务器的架构，让服务器能为每个客户只使用一个连接。（而不是每个请求一个连接）(b) 把你之前的问题的解决方案应用到(a)部分，让多个客户的请求可以被并行地处理。

# 第17章 网络客户端编程

# 第18章 多线程编程

## 18.1 Python、线程和全局解释器锁

### 18.1.1 全局解释器锁（GIL）

Python 代码的执行由 Python 虚拟机(也叫解释器主循环)来控制。Python 在设计之初就考虑到要在主循环中，同时只有一个线程在执行，就像单 CPU 的系统中运行多个进程那样，内存中可以存放多个程序，但任意时刻，只有一个程序在 CPU 中运行。同样地，虽然 Python 解释器中可以“运行”多个线程，但在任意时刻，只有一个线程在解释器中运行。

对 Python 虚拟机的访问由全局解释器锁（GIL）来控制，正是这个锁能保证同一时刻只有一个线程在运行。在多线程环境中，Python 虚拟机按以下方式执行：

1. 设置 GIL

2. 切换到一个线程去运行

3. 运行：

a. 指定数量的字节码指令，或者

b. 线程主动让出控制（可以调用 time.sleep(0)）

4. 把线程设置为睡眠状态

5. 解锁 GIL

6. 再次重复以上所有步骤

在调用外部代码（如 C/C++扩展函数）的时候，GIL 将会被锁定，直到这个函数结束为止（由于在这期间没有 Python 的字节码被运行， 所以不会做线程切换）。 编写扩展的程序员可以主动解锁 GIL。

不过，Python 的开发人员则不用担心在这些情况下你的 Python 代码会被锁住。

例如，对所有面向 I/O 的(会调用内建的操作系统 C 代码的)程序来说，GIL 会在这个 I/O 调用之前被释放，以允许其它的线程在这个线程等待 I/O的时候运行。 如果某线程并未使用很多 I/O 操作，它会在自己的时间片内一直占用处理器（和GIL）。也就是说，I/O 密集型的 Python 程序比计算密集型的程序更能充分利用多线程环境的好处。

对源代码，解释器主循环和 GIL 感兴趣的人，可以看看 Python/ceval.c 文件。

### 18.1.2 退出线程

当一个线程结束计算，它就退出了。线程可以调用 thread.exit()之类的退出函数，也可以使用Python 退出进程的标准方法，如 sys.exit()或抛出一个 SystemExit 异常等。不过，你不可以直接“杀掉”("kill")一个线程。

在下面一节中，我们将要讨论两个跟线程有关的模块。这两个模块中，我们不建议使用 thread模块。这样做有很多原因，很明显的一个原因是，当主线程退出的时候，所有其它线程没有被清除就退出了。但另一个模块 threading 就能确保所有“重要的”子线程都退出后，进程才会结束。（我们等一会会详细说明什么叫“重要的”，请参阅守护线程的核心提示）。

主线程应该是一个好的管理者，它要了解每个线程都要做些什么事，线程都需要什么数据和什么参数，以及在线程结束的时候，它们都提供了什么结果。这样，主线程就可以把各个线程的结果组合成一个有意义的最后结果。

### 18.1.3 在 Python 中使用线程

在 Win32 和 Linux, Solaris, MacOS, \*BSD 等大多数类 Unix 系统上运行时，Python 支持多线程编程。Python 使用 POSIX 兼容的线程，即 pthreads。

默认情况下，从源代码编译的(2.0 及以上版本的)Python 以及 Win32 的安装包里，线程支持是打开的。想要从解释器里判断线程是否可用，只要简单的在交互式解释器里尝试导入 thread 模块就行了，只要没出现错误就表示线程可用。

import thread

如果你的 Python 解释器在编译时，没有打开线程支持，导入模块会失败：

Traceback (innermost last): File "<stdin>", line 1, in ?

ImportError: No module named thread

这种情况下，你就要重新编译你的 Python 解释器才能使用线程。 你可以在运行配置脚本的时候，加上“--with-thread”参数。参考你的发布版的 README 文件，以获取如何编译支持线程的 Python的相关信息。

### 18.1.4 没有线程支持的情况

第一个例子中，我们会使用 time.sleep()函数来演示线程是怎样工作的。time.sleep()需要一个浮点型的参数，来指定“睡眠”的时间（单位秒）。这就意味着，程序的运行会被挂起指定的时间。

我们要创建两个“计时循环”。一个睡眠 4 秒种，一个睡眠 2 秒种，分别是 loop0()和 loop1()。（我们命名为“loop0”和“loop1”表示我们将有一个循环的序列）。 如果我们像例 18.1 的 onethr.py中那样，在一个进程或一个线程中，顺序地执行 loop0()和 loop1()，那运行的总时间为 6 秒。在启动 loop0()，loop1()，和其它的代码时，也要花去一些时间，所以，我们看到的总时间也有可能会是 7 秒钟。

例 18.1 单线程中运行的循环 (one\_thread.py)

在单线程中顺序执行两个循环。一定要一个循环结束后，另一个才能开始。总时间是各个循环运行时间之和。

#!/usr/bin/env python

from time import sleep, ctime

def loop0():

print 'start loop 0 at:', ctime()

sleep(4)

print 'loop 0 done at:', ctime()

def loop1():

print 'start loop 1 at:', ctime()

sleep(2)

print 'loop 1 done at:', ctime()

def main():

print 'starting at:', ctime()

loop0()

loop1()

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()

我们可以通过运行 onethr.py 来验证这一点，下面是运行的输出：

$ python one\_thread.py

starting at: Tue Dec 12 10:23:33 2017

start loop 0 at: Tue Dec 12 10:23:33 2017

loop 0 done at: Tue Dec 12 10:23:37 2017

start loop 1 at: Tue Dec 12 10:23:37 2017

loop 1 done at: Tue Dec 12 10:23:39 2017

all DONE at: Tue Dec 12 10:23:39 2017

假定 loop0()和 loop1()里做的不是睡眠，而是各自独立的，不相关的运算，各自的运算结果到最后将会汇总成一个最终的结果。这时，如果能让这些计算并行执行的话，那不是可以减少总的运行时间吗？这就是我们现在要介绍的多线程编程的前提条件。

### 18.1.5 Python 的 threading 模块

Python 提供了几个用于多线程编程的模块，包括 thread, threading 和 Queue 等。thread 和threading 模块允许程序员创建和管理线程。 thread 模块提供了基本的线程和锁的支持， 而 threading提供了更高级别，功能更强的线程管理的功能。Queue 模块允许用户创建一个可以用于多个线程之间共享数据的队列数据结构。我们将分别介绍这几个模块，并给出一些例子和中等大小的应用。

**核心提示：避免使用 thread 模块**

**出于以下几点考虑，我们不建议您使用 thread 模块。 首先，更高级别的 threading 模块更为先进， 对线程的支持更为完善， 而且使用 thread 模块里的属性有可能会与 threading 出现冲突。 其次，低级别的 thread 模块的同步原语很少（实际上只有一个），而 threading 模块则有很多。**

**不过，出于对学习 Python 和线程的兴趣，我们将给出一点使用 thread 模块的例子。这些代码只用于学习目的，让你对为什么应该避免使用 thread 模块有更深的认识，以及让你了解在把代码改为使用 threading 和 Queue 模块时，我们能获得多大的便利。**

**另一个不要使用 thread 原因是，对于你的进程什么时候应该结束完全没有控制，当主线程结束时，所有的线程都会被强制结束掉，没有警告也不会有正常的清除工作。我们之前说过，至少threading 模块能确保重要的子线程退出后进程才退出。**

**只建议那些有经验的专家在想访问线程的底层结构的时候，才使用 thread 模块。而使用线程的新手们则应该看看我们是如何把线程应用到我们的第一个程序，从而增加代码的可读性，以及第一段例子如何进化到我们本章的主要的代码的。如果可以的话，你的第一个多线程程序应该尽可能地使用 threading 等高级别的线程模块。**

## 18.2 thread 模块

我们先看看 thread 模块都提供了些什么。除了产生线程外，thread 模块也提供了基本的同步数据结构锁对象(lock object，也叫原语锁，简单锁，互斥锁，互斥量，二值信号量)。如之前所说，同步原语与线程的管理是密不可分的。

表 18.1 中所列的是常用的线程函数以及 LockType 类型的锁对象的方法。

start\_new\_thread()函数是 thread 模块的一个关键函数，它的语法与内建的 apply()函数完全一样，其参数为：函数，函数的参数以及可选的关键字参数。不同的是，函数不是在主线程里运行，而是产生一个新的线程来运行这个函数。

现在，把线程加入到我们的 one\_thread.py 例子中。稍微改变一下 loop\*()函数的调用方法，我们得到了例 18.2 的 mtsleep1.py。

表 18.1 thread 模块和锁对象

thread 模块函数

start\_new\_thread(function, args, kwargs=None)

产生一个新的线程，在新线程中用指定的参数和可选的kwargs 来调用这个函数。

allocate\_lock() 分配一个 LockType 类型的锁对象

exit() 让线程退出

LockType 类型锁对象方法

acquire(wait=None) 尝试获取锁对象

locked() 如果获取了锁对象返回 True，否则返回 False

release() 释放锁

例 18.2 使用 thread 模块 (mtsleep1.py)

这儿执行的是和 one\_thread.py 中一样的循环，不同的是，这一次我们使用的是 thread 模块提供的简单的多线程的机制。两个循环并发地被执行（显然，短的那个先结束）。总的运行时间为最慢的那个线程的运行时间，而不是所有的线程的运行时间之和。

#!/usr/bin/env python

import thread

from time import sleep, ctime

def loop0():

print 'start loop 0 at:', ctime()

sleep(4)

print 'loop 0 done at:', ctime()

def loop1():

print 'start loop 1 at:', ctime()

sleep(2)

print 'loop 1 done at:', ctime()

def main():

print 'starting at:', ctime()

thread.start\_new\_thread(loop0, ())

thread.start\_new\_thread(loop1, ())

sleep(6)

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()

start\_new\_thread()要求一定要有前两个参数。所以，就算我们想要运行的函数不要参数，我们也要传一个空的元组。

这个程序的输出与之前的输出大不相同，之前是运行了 6，7 秒，而现在则是 4 秒，是最长的循环的运行时间与其它的代码的时间总和。

$ python mtsleep1.py

starting at: Tue Dec 12 10:38:47 2017

start loop 0 at: Tue Dec 12 10:38:47 2017

start loop 1 at: Tue Dec 12 10:38:47 2017

loop 1 done at: Tue Dec 12 10:38:49 2017

loop 0 done at: Tue Dec 12 10:38:51 2017

all DONE at: Tue Dec 12 10:38:53 2017

睡眠 4 秒和 2 秒的代码现在是并发执行的。这样，就使得总的运行时间被缩短了。你可以看到，loop1 甚至在 loop0 前面就结束了。程序的一大不同之处就是多了一个“sleep(6)”的函数调用。为什么要加上这一句呢？因为，如果我们没有让主线程停下来，那主线程就会运行下一条语句，显示“all done”，然后就关闭运行着 loop0()和 loop1()的两个线程，退出了。

我们没有写让主线程停下来等所有子线程结束之后再继续运行的代码。这就是我们之前说线程需要同步的原因。在这里，我们使用了 sleep()函数做为我们的同步机制。我们使用 6 秒是因为我们已经知道，两个线程（你知道，一个要 4 秒，一个要 2 秒）在主线程等待 6 秒后应该已经结束了。

你也许在想，应该有什么好的管理线程的方法，而不是在主线程里做一个额外的延时 6 秒的操作。因为这样一来，我们的总的运行时间并不比单线程的版本来得少。而且，像这样使用 sleep()函数做线程的同步操作是不可靠的。如果我们的循环的执行时间不能事先确定的话，那怎么办呢？这可能造成主线程过早或过晚退出。这就是锁的用武之地了。

上一次修改程序， 我们去掉了 loop 函数， 现在， 我们要再一次修改程序为例 18.3 的 mtsleep2.py，引入锁的概念。运行它，我们看到，其输出与 mtsleep1.py 很相似，唯一的区别是我们不用为线程什么时候结束再做额外的等待。使用了锁，我们就可以在两个线程都退出后，马上退出。

$ python mtsleep2.py

starting at: Tue Dec 12 10:53:01 2017

start loop 1 at: Tue Dec 12 10:53:01 2017

start loop 0 at: Tue Dec 12 10:53:01 2017

loop 1 done at: Tue Dec 12 10:53:03 2017

loop 0 done at: Tue Dec 12 10:53:05 2017

all DONE at: Tue Dec 12 10:53:05 2017

我们是怎么通过锁来完成任务的呢？先看一看代码吧。

例 18.3 使用线程和锁 (mtsleep2.py)

这里，使用锁比 mtsleep1.py 那里在主线程中使用 sleep()函数更合理。

#!/usr/bin/env python

import thread

from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec, lock):

print 'start loop', nloop, 'at:', ctime()

sleep(nsec)

print 'loop', nloop, 'done at:', ctime()

lock.release()

def main():

print 'starting at:', ctime()

locks = []

nloops = range(len(loops))

for i in nloops:

lock = thread.allocate\_lock()

lock.acquire()

locks.append(lock)

for i in nloops:

thread.start\_new\_thread(loop,(i, loops[i], locks[i]))

for i in nloops:

while locks[i].locked(): pass

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()逐行解释

**1-6 行**

在 Unix 启动信息行后面，我们导入了 thread 模块和 time 模块里我们早已熟悉的几个函数。我们不再在函数里写死要等 4 秒和 2 秒，而是使用一个 loop()函数，把这些常量放在一个列表 loops里。

**8-12 行**

loop()函数替换了我们之前的那几个 loop\*()函数。在 loop()函数里，增加了一些锁的操作。一个很明显的改变是，我们现在要在函数中记录下循环的号码和要睡眠的时间。最后一个不一样的地方就是那个锁了。每个线程都会被分配一个事先已经获得的锁，在 sleep()的时间到了之后就释放相应的锁以通知主线程，这个线程已经结束了。

**14-34 行**

主要的工作在包含三个循环的 main()函数中完成。我们先调用 thread.allocate\_lock()函数创建一个锁的列表，并分别调用各个锁的 acquire()函数获得锁。获得锁表示“把锁锁上”。锁上后，我们就把锁放到锁列表 locks 中。下一个循环创建线程，每个线程都用各自的循环号，睡眠时间和锁为参数去调用 loop()函数。 为什么我们不在创建锁的循环里创建线程呢？有以下几个原因： (1) 我们想到实现线程的同步，所以要让“所有的马同时冲出栅栏”。(2) 获取锁要花一些时间，如果你的线程退出得“太快”，可能会导致还没有获得锁，线程就已经结束了的情况。

在线程结束的时候，线程要自己去做解锁操作。最后一个循环只是坐在那一直等（达到暂停主线程的目的），直到两个锁都被解锁为止才继续运行。由于我们顺序检查每一个锁，所以我们可能会要长时间地等待运行时间长且放在前面的线程，当这些线程的锁释放之后，后面的锁可能早就释放了（表示对应的线程已经运行完了）。结果主线程只能毫不停歇地完成对后面这些锁的检查。最后两行代码的意思你应该已经知道了，就是只有在我们直接运行这个脚本时，才运行 main()函数

在核心笔记中我们就已经说过，使用 thread 模块只是为了给读者演示如何进行多线程编程。你的多线程程序应该使用更高级别的模块，如 threading 等。现在我们就开始讨论它。

## 18.3 threading 模块

接下来，我们要介绍的是更高级别的 threading 模块，它不仅提供了 Thread 类，还提供了各种非常好用的同步机制。表 18.2 列出了 threading 模块里所有的对象。

在这一节中，我们会演示如何使用threading类来实现多线程。之前已经介绍过锁的基本概念，这里我们将不会提到锁原语。而 Thread 类也有某种同步机制，所以，没有必要详细介绍锁原语。

表 18.2 threading 模块对象

Thread 表示一个线程的执行的对象

Lock 锁原语对象（跟 thread 模块里的锁对象相同）

RLock 可重入锁对象。使单线程可以再次获得已经获得了的锁（递归锁定）。

Condition 条件变量对象能让一个线程停下来， 等待其它线程满足了某个“条件”。如，状态的改变或值的改变。

Event 通用的条件变量。多个线程可以等待某个事件的发生，在事件发生后，所有的线程都会被激活。

Semaphore 为等待锁的线程提供一个类似“等候室”的结构

BoundedSemaphore 与 Semaphore 类似，只是它不允许超过初始值

Timer 与 Thread 相似，只是，它要等待一段时间后才开始运行。

核心提示：守护线程

另一个避免使用 thread 模块的原因是，它不支持守护线程。当主线程退出时，所有的子线程不论它们是否还在工作，都会被强行退出。有时，我们并不期望这种行为，这时，就引入了守护线程的概念。

threading 模块支持守护线程， 它们是这样工作的： 守护线程一般是一个等待客户请求的服务器，如果没有客户提出请求，它就在那等着。如果你设定一个线程为守护线程，就表示你在说这个线程是不重要的，在进程退出的时候，不用等待这个线程退出。 就像你在第 16 章网络编程看到的，服务器线程运行在一个无限循环中，一般不会退出。

如果你的主线程要退出的时候，不用等待那些子线程完成，那就设定这些线程的 daemon 属性。即，在线程开始（调用 thread.start()）之前， 调用 setDaemon()函数设定线程的 daemon 标志（thread.setDaemon(True)） 就表示这个线程“不重要”

如果你想要等待子线程完成再退出，那就什么都不用做，或者显式地调用thread.setDaemon(False)以保证其 daemon 标志为 False。 你可以调用 thread.isDaemon()函数来判断其 daemon 标志的值。新的子线程会继承其父线程的 daemon 标志。整个 Python 会在所有的非守护线程退出后才会结束,即进程中没有非守护线程存在的时候才结束。

### 18.3.1 Thread 类

threading 的 Thread 类是你主要的运行对象。 它有很多 thread 模块里没有的函数， 详见表 18.3。用 Thread 类，你可以用多种方法来创建线程。我们在这里介绍三种比较相像的方法。你可以任选一种你喜欢的，或最适合你的程序以及最能满足程序可扩展性的（我们一般比较喜欢最后一个选择）：

（1）创建一个 Thread 的实例，传给它一个函数

（2）创建一个 Thread 的实例，传给它一个可调用的类对象

（3）从 Thread 派生出一个子类，创建一个这个子类的实例

表 18.3 Thread 对象的函数

start() 开始线程的执行

run() 定义线程的功能的函数（一般会被子类重写）

join(timeout=None) 程序挂起，直到线程结束；如果给了 timeout，则最多阻塞 timeout 秒

getName() 返回线程的名字

setName(name) 设置线程的名字

isAlive() 布尔标志，表示这个线程是否还在运行中

isDaemon() 返回线程的 daemon 标志

setDaemon(daemonic) 把线程的 daemon 标志设为 daemonic（一定要在调用 start()函数前调用）

**创建一个 Thread 的实例，传给它一个函数**

第一个例子的中，我们将初始化一个 Thread 对象，把函数（及其参数）像上一个例子那样传进去。 在线程开始执行的时候， 这个函数会被执行。 把 mtsleep2.py 脚本拿过来， 做一些调整加入 Thread对象的使用，就成了例 18.4 中的 mtsleep3.py。

运行的输出跟之前很相似：

$ python mtsleep3.py

starting at: Tue Dec 12 11:28:40 2017

start loop 0 at: Tue Dec 12 11:28:40 2017

start loop 1 at: Tue Dec 12 11:28:40 2017

loop 1 done at: Tue Dec 12 11:28:42 2017

loop 0 done at: Tue Dec 12 11:28:44 2017

all DONE at: Tue Dec 12 11:28:44 2017

那么， 都做了些什么修改呢？在使用 thread 模块时使用的锁没有了。 新加了一些 Thread 对象。在实例化每个 Thread 对象的时候， 我们把函数（target） 和参数（args） 传进去， 得到返回的 Thread实例。实例化一个 Thread（调用 Thread()）与调用 thread.start\_new\_thread()之间最大的区别就是，新的线程不会立即开始。在你创建线程对象，但不想马上开始运行线程的时候，这是一个很有用的同步特性。

例 18.4 使用 threading 模块 (mtsleep3.py)

threading 模块的 Thread 类有一个 join()函数，允许主线程等待线程的结束。

#!/usr/bin/env python

import threading

from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec):

print 'start loop', nloop, 'at:', ctime()

sleep(nsec)

print 'loop', nloop, 'done at:', ctime()

def main():

print 'starting at:', ctime()

threads = []

nloops = range(len(loops))

for i in nloops:

t = threading.Thread(target=loop, args=(i, loops[i]))

threads.append(t)

for i in nloops: # start threads

threads[i].start()

for i in nloops: # wait for all

threads[i].join() # threads to finish

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()

所有的线程都创建了之后，再一起调用 start()函数启动，而不是创建一个启动一个。而且，不用再管理一堆锁（分配锁， 获得锁， 释放锁， 检查锁的状态等）， 只要简单地对每个线程调用 join()函数就可以了。

join()会等到线程结束，或者在给了 timeout 参数的时候，等到超时为止。使用 join()看上去会比使用一个等待锁释放的无限循环清楚一些(这种锁也被称为"spinlock")。

join()的另一个比较重要的方面是它可以完全不用调用。一旦线程启动后，就会一直运行，直到线程的函数结束，退出为止。如果你的主线程除了等线程结束外，还有其它的事情要做（如处理或等待其它的客户请求）， 那就不用调用 join()， 只有在你要等待线程结束的时候才要调用 join()。

**创建一个 Thread 的实例，传给它一个可调用的类对象**

与传一个函数很相似的另一个方法是在创建线程的时候，传一个可调用的类的实例供线程启动的时候执行——这是多线程编程的一个更为面向对象的方法。相对于一个或几个函数来说，由于类对象里可以使用类的强大的功能，可以保存更多的信息，这种方法更为灵活。

把 ThreadFunc 类加入到 mtsleep3.py 代码中，并做一些其它的小修改后，就得到了例 18.5 中的 mtsleep4.py。运行它，就会得到如下的输出：

$ python mtsleep4.py

starting at: Tue Dec 12 11:52:17 2017

start loop 0 at: Tue Dec 12 11:52:17 2017

start loop 1 at: Tue Dec 12 11:52:17 2017

loop 1 done at: Tue Dec 12 11:52:19 2017

loop 0 done at: Tue Dec 12 11:52:21 2017

all DONE at: Tue Dec 12 11:52:21 2017

那么，这次又改了些什么呢？主要是增加了 ThreadFunc 类和创建 Thread 对象时会实例化一个可调用类 ThreadFunc 的类对象。 也就是说， 我们实例化了两个对象。 下面， 来仔细地看一看 ThreadFunc类吧。

我们想让这个类在调用什么函数方面尽量地通用，并不局限于那个 loop()函数。所以，我们加了一些修改，如，这个类保存了函数的参数，函数本身以及函数的名字字符串。构造函数\_\_init\_\_()里做了这些值的赋值工作。

创建新线程的时候， Thread 对象会调用我们的 ThreadFunc 对象， 这时会用到一个特殊函数\_\_call\_\_()。由于我们已经有了要用的参数，所以就不用再传到 Thread()的构造函数中。由于我们有一个参数的元组， 这时要在代码中使用 apply()函数。 如果你使用的是 Python1.6 或是更高版本，

你可以使用 11.6.3 节中所说的新的调用语法，而不用像第 16 行那样使用 apply()函数：

self.res = self.func(\*self.args)

例 18.5 使用可调用的类 (mtsleep4.py)

此例中，我们传了一个可调用的类(的实例)，而不是仅传一个函数。相对 mtsleep3.py 中的方法来说，这样做更具面向对象的概念。

#!/usr/bin/env python

import threading

from time import sleep, ctime

loops = [4,2]

class ThreadFunc(object):

def \_\_init\_\_(self, func, args, name=''):

self.name = name

self.func = func

self.args = args

def \_\_call\_\_(self):

apply(self.func, self.args)

def loop(nloop, nsec):

print 'start loop', nloop, 'at:', ctime()

sleep(nsec)

print 'loop', nloop, 'done at:', ctime()

def main():

print 'starting at:', ctime()

threads = []

nloops = range(len(loops))

for i in nloops:

#print loop.\_\_name\_\_

t = threading.Thread(target=ThreadFunc(loop,(i, loops[i]), loop.\_\_name\_\_))

threads.append(t)

for i in nloops: # start threads

threads[i].start()

for i in nloops: # wait for all

threads[i].join() # threads to finish

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()

**从 Thread 派生出一个子类，创建一个这个子类的实例**

最后一个例子介绍如何子类化 Thread 类，这与上一个例子中的创建一个可调用的类非常像。使用子类化创建线程（第 29-30 行）使代码看上去更清晰明了。我们将在例 18.6 中给出 mtsleep5.py的代码，以及代码运行的输出。比较 mtsleep5.py 和 mtsleep4.py 的任务则留给读者做为练习。

下面是 mtsleep5.py 的输出，同样，跟我们的期望一致：

$ python mtsleep5.py

starting at: Tue Dec 12 13:46:40 2017

start loop 0 at: Tue Dec 12 13:46:40 2017

start loop 1 at: Tue Dec 12 13:46:40 2017

loop 1 done at: Tue Dec 12 13:46:42 2017

loop 0 done at: Tue Dec 12 13:46:44 2017

all DONE at: Tue Dec 12 13:46:44 2017

在读者比较 mtsleep4 和 mtsleep5 两个模块的代码之前，我们想指出最重要的两点改变：(1)我们的 MyThread 子类的构造函数一定要先调用基类的构造函数(第 9 行)，(2)之前的特殊函数\_\_call\_\_()在子类中，名字要改为 run()。

现在，在 MyThread 类中，加入一些用于调试的输出信息，把代码保存到 myThread 模块中(见例18.7)，并在下面的例子中，导入这个类。除了简单地使用 apply()函数来运行这些函数之外，我们还把结果保存到实现的 self.res 属性中，并创建一个新的函数 getResult()来得到结果。

### 18.3.4 斐波那契，阶乘和累加和

例 18.8 中的 mtfacfib.py 脚本比较了递归求斐波那契，阶乘和累加和函数的运行。脚本先在单线程中运行这三个函数，然后在多线程中做同样的事，以说明多线程的好处。

例 18.6 子类化 Thread (mtsleep5.py)

我们现在要子类化 Thread 类， 而不是创建它的实例。 这样做可以更灵活地定制我们的线程对象，而且在创建线程的时候也更简单。

#!/usr/bin/env python

import threading

from time import sleep, ctime

loops = [4,2]

class MyThread(threading.Thread):

def \_\_init\_\_(self, func, args, name=''):

threading.Thread.\_\_init\_\_(self)

self.name = name

self.func = func

self.args = args

def run(self): # When a ThreadFunc object is called by a Thread object, two methods are all accepted.

#apply(self.func, self.args)

self.res = self.func(\*self.args)

def loop(nloop, nsec):

print 'start loop', nloop, 'at:', ctime()

sleep(nsec)

print 'loop', nloop, 'done at:', ctime()

def main():

print 'starting at:', ctime()

threads = []

nloops = range(len(loops))

for i in nloops:

#print loop.\_\_name\_\_

t = MyThread(loop, (i, loops[i]), loop.\_\_name\_\_)

threads.append(t)

for i in nloops: # start threads

threads[i].start()

for i in nloops: # wait for all

threads[i].join() # threads to finish

print 'all DONE at:', ctime()

if \_\_name\_\_ == '\_\_main\_\_':

main()

例 18.7 MyThread 子类化 Thread (myThread.py)

为了让 mtsleep5.py 中，Thread 的子类更为通用，我们把子类单独放在一个模块中，并加上一个getResult()函数用以返回函数的运行结果。

#!/usr/bin/env python

import threading

from time import sleep, ctime

class MyThread(threading.Thread):

def \_\_init\_\_(self, func, args, name=''):

threading.Thread.\_\_init\_\_(self)

self.name = name

self.func = func

self.args = args

def getResult(self):

return self.res

def run(self):

#apply(self.func, self.args)

print 'starting', self.name, 'at:', ctime()

self.res = self.func(\*self.args)

print self.name, 'finished at:', ctime()

在单线程中运行只要简单地逐个调用这些函数，在函数结束后，显示对应的结果。在多线程中，我们不马上显示结果。 由于我们想让 MyThread 类尽可能地通用(能同时适应有输出和没输出的函数)，我们会等到要结束时才会调用 getResult()函数，并在最后显示每个函数的结果。

由于这些函数运行得很快（斐波那契函数会慢一些），你会看到，我们得在每个函数中加上一个 sleep()函数，让函数慢下来，以便于我们能方便地看到多线程能在多大程度上加速程序的运行。不过实际工作中，你一般不会想在程序中加上sleep()函数的。下面是程序的输出：

$ python mtfacfib.py

\*\*\* SINGLE THREAD

starting fib at: Tue Dec 12 14:08:21 2017

233

fib finished at: Tue Dec 12 14:08:23 2017

starting fac at: Tue Dec 12 14:08:23 2017

479001600

fac finished at: Tue Dec 12 14:08:25 2017

starting sum at: Tue Dec 12 14:08:25 2017

78

sum finished at: Tue Dec 12 14:08:26 2017

\*\*\* MULTIPLE THREADS

starting fib at: Tue Dec 12 14:08:26 2017

starting fac at: Tue Dec 12 14:08:26 2017

starting sum at: Tue Dec 12 14:08:26 2017

fac finished at: Tue Dec 12 14:08:27 2017

sum finished at: Tue Dec 12 14:08:27 2017

fib finished at: Tue Dec 12 14:08:28 2017

233

479001600

78

all DONE

例 18.8 斐波那契，阶乘和累加和 (mtfacfib.py)

在这个多线程程序中，我们会分别在单线程和多线程环境中，运行三个递归函数。

#!/usr/bin/env python

from mythread import MyThread

from time import ctime, sleep

def fib(x):

sleep(0.005)

if x < 2: return 1

return (fib(x-2) + fib(x-1))

def fac(x):

sleep(0.1)

if x < 2: return 1

return (x \* fac(x-1))

def sum(x):

sleep(0.1)

if x < 2: return 1

return (x + sum(x-1))

funcs = [fib, fac, sum]

n = 12

def main():

nfuncs = range(len(funcs))

print '\*\*\* SINGLE THREAD'

for i in nfuncs:

print 'starting', funcs[i].\_\_name\_\_, 'at:', ctime()

print funcs[i](n)

print funcs[i].\_\_name\_\_, 'finished at:', ctime()

print '\n\*\*\* MULTIPLE THREADS'

threads = []

for i in nfuncs:

t = MyThread(funcs[i], (n,), funcs[i].\_\_name\_\_)

threads.append(t)

for i in nfuncs:

threads[i].start()

for i in nfuncs:

threads[i].join()

print threads[i].getResult()

print 'all DONE'

if \_\_name\_\_ == '\_\_main\_\_':

main()

### 18.3.5 threading 模块中的其它函数

除了各种同步对象和线程对象外，threading 模块还提供了一些函数。见表 18.4。

表 18.4 threading 模块的函数

activeCount() 当前活动的线程对象的数量

currentThread() 返回当前线程对象

enumerate() 返回当前活动线程的列表

settrace(func) 为所有线程设置一个跟踪函数

setprofile(func) 为所有线程设置一个profile 函数

### 18.3.6 生产者-消费者问题和 Queue 模块

最后一个例子演示了生产者和消费者的场景。生产者生产货物，然后把货物放到一个队列之类的数据结构中，生产货物所要花费的时间无法预先确定。消费者消耗生产者生产的货物的时间也是不确定的。

表 18.5 常用的 Queue 模块的属性

Queue 模块函数

queue(size) 创建一个大小为 size 的 Queue 对象

Queue 对象函数

qsize() 返回队列的大小（由于在返回的时候，队列可能会被其它线程修改，所以这个值是近似值）

empty() 如果队列为空返回 True，否则返回 False

full() 如果队列已满返回 True，否则返回 False

put(item,block=0) 把 item 放到队列中，如果给了 block（不为 0），函数会一直阻塞到队列中有空间为止get(block=0) 从队列中取一个对象，如果给了 block（不为 0），函数会一直阻塞到队列中有对象为止。

Queue 模块可以用来进行线程间通讯，让各个线程之间共享数据。现在，我们创建一个队列，让生产者（线程）把新生产的货物放进去供消费者（线程）使用。要达到这个目的，我们要使用到 Queue模块的以下属性（见表 18.5）。

很容易地，我们就能写出例 18.9 的 prodcons.py 的代码。

例 18.9 生产者－消费者问题 (prodcons.py)

这个实现中使用了 Queue 对象和随机地生产（和消耗）货物的方式。生产者和消费者相互独立并且并发地运行。

#!/usr/bin/env python

from random import randint

from time import sleep

from Queue import Queue

from mythread import MyThread

def writeQ(queue):

print 'producing object for Q...',

queue.put('xxx', 1)

print "size now", queue.qsize()

def readQ(queue):

val = queue.get(1)

print 'consumed object from Q... size now', \

queue.qsize()

def writer(queue, loops):

for i in range(loops):

writeQ(queue)

sleep(randint(1, 3))

def reader(queue, loops):

for i in range(loops):

readQ(queue)

sleep(randint(3, 5))

funcs = [writer, reader]

nfuncs = range(len(funcs))

def main():

nloops = randint(2, 5)

q = Queue(32)

threads = []

for i in nfuncs:

t = MyThread(funcs[i], (q, nloops), funcs[i].\_\_name\_\_)

threads.append(t)

for i in nfuncs:

threads[i].start()

for i in nfuncs:

threads[i].join()

print 'all DONE'

if \_\_name\_\_ == '\_\_main\_\_':

main()

逐行解释

**1-6 行**

在这个模块中，我们要使用Queue.Queue对象以及我们在例18.7中给出的线程类mythread.MyThread。我们将使用 random.randint()函数来随机的进行生产和消耗。并从 time 模块中导入了常用的属性。

**8-16行**

writeQ()和 readQ()函数分别用来把对象放入队列和消耗队列中的一个对象。在这里我们使用字符串'xxx'来表示队列中的对象。

**18-26 行**

writer()函数只做一件事，就是一次往队列中放入一个对象，等待一会，然后再做同样的事，一共做指定的次数，这个次数是由脚本运行时随机生成的。reader()函数做的事比较类似，只是它是用来消耗对象的。

你会注意到，writer 睡眠的时间一般会比 reader 睡眠的时间短。这可以减少 reader 尝试从空队列中取数据的机会。writer 的睡眠时间短，那 reader 在想要数据的时候总是能拿到数据。

**28-29 行**

设置有多少个线程要被运行。

31-47 行

最后，就到了 main()函数，它与之前的所有脚本的 main()函数都很像。先是创建所有的线程，然后运行它们，最后，等两个线程都结束后，得到最后的运行结果。

从本例中，我们可以了解到，一个要完成多项任务的程序，可以考虑每个任务使用一个线程。这样的程序在设计上相对于单线程做所有事的程序来说，更为清晰明了。

本章中，我们看到了单线程的程序在程序性能上的限制。尤其在有相互独立的，运行时间不确定的多个任务的程序里，把多个任务分隔成多个线程同时运行会比顺序运行速度更快。由于 Python解释器是单线程的，所以不是所有的程序都能从多线程中得到好处。不过，你已经对 Python 下的多线程有所了解，在适当的时候，可以利用它来改善程序的性能。

## 18.4 相关模块

下表列出了一些多线程编程中可能用得到的模块：

表 18.6 多线程相关的标准库模块

thread 基本的，底级别的线程模块

threading 高级别的线程和同步对象

Queue 供多线程使用的同步先进先出（FIFO）队列

mutex 互斥对象

SocketServer 具有线程控制的 TCP 和 UDP 管理器

## 18.5 练习

**1. 进程与线程。线程与进程的区别是什么？**

答：（1）进程，（有时候也成为重量级进程），是程序的一次执行。每个进程都有自己独立的地址空间，内存，堆栈以及其它记录其运行轨迹的辅助数据。操作系统管理在其上运行的所有进程，并为这些进程公平的分配时间。各个进程使用进程间通讯，而不能直接共享信息。

（2）线程，有时候也被称为轻量级线程，跟进程有些相似，不同的是，所有的线程运行在同一个进程中，共享相同的运行环境。它们可以想象成是在主进程或“主线程”中并行运行的“迷你进程”。

线程之间可以比进程之间更方便地共享数据以及相互通信。

**2. Python 的线程。在 Python 中，哪一种多线程的程序表现得更好，I/O 密集型的还是计算密集型的？**

答：I/O密集型的线程性能表现更好，因为Python解析器是单线程执行的，所以计算密集型的线程只能轮流占用CPU使用时间。

**3. 线程。你认为，多CPU 的系统与一般的系统有什么大的不同？多线程的程序在这种系统上的表现会怎么样？**

答：线程一般是并发执行的，正是由于这种并发执行的机制使得多个任务的成为可能。实际上，在单个CPU的系统上，真正的并发是不能的，每个线程会被安排成每次只运行一小会，然后就把CPU让出来，让其他的线程去运行。多线程的程序在这种系统上性能表现更好。

4. 线程和文件。把练习 9-19 的答案做一些改进。我们要得到一个字节值，一个文件名然后显示在文件中那个字节出现了多少次。假设这个文件非常的大。文件是可以有多个读者的，那我们就可以创建多个线程，每个线程负责文件的一部分。最后，把所有的线程的结果相加。使用 timeit()对单线程和多线程分别进行计时，对性能的改进进行讨论。

5. 线程，文件和正则表达式。你有一个非常大的 mailbox 文件——如果没有的话，你可以把你所有的 e-mail 的原始信息放到一个文本文件中。你现在要做的是，使用你在 15 章写的识别 e-mail 地址和网页 URL 的正则表达式，分析出这个大文件里的所有的 e-mail 地址和 URL，把这些链接写到一个.html(或.htm)文件中。在这个文件生成时，会自动显示一个浏览器，打开这个文件，显示所有的链接。使用多线程来分隔处理大文件和把结果写到一个新的.html 文件的操作。在浏览器中测试一下你的结果，确保那些链接都能正常工作。

6. 线程和网络。你在之前做的聊天服务器程序（练习 16-7 到 16-10）也许会用到重量级线程或者说进程，把那个代码改成多线程的。

7. \*线程和 Web 编程。练习 19.1 中的爬虫，是一个单线程的网页下载程序， 但可以利用多线程提高性能。修改 crawl.py（你可以叫它 mtcrawl.py），让它可以使用多个不相关的线程来下载网页。注意要使用某种锁的机制以确保不会在访问链接队列的时候出现访问冲突。

**8. 线程池。修改例 18.9 的代码，不再是一个生产者和一个消费者，而是可以有任意个消费者线程（一个线程池），每个线程可以在任意时刻处理或消耗任意多个产品。**

答：

1 Threadpool模块详解

原文地址：https://chrisarndt.de/projects/threadpool/

|  |  |
| --- | --- |
| **Title:** | Easy to use object-oriented thread pool framework |
| **Author:** | Christopher Arndt |
| **Version:** | 1.3.2 |
| **Date:** | 2015-11-29 |
| **License:** | MIT License |

警告:

本模块已经被废弃，只在PyPI上提供下载，以支持仍然在使用的旧项目。如果是新项目，请不要再使用。使用标准库例的multiprocessing模块或者异步处理的asyncio模块。

描述

线程池是一个对象，它维护一个工作线程池来并行执行耗时的操作。它通过将作业放入工作请求队列中将作业分配给这些线程，在那里它们被下一个可用线程拾取。然后在后台执行请求的操作，并将结果放入另一个队列中。

线程池对象可以在所有线程都可用或者在所有线程完成工作之后立即从该队列中的所有线程收集结果。 然后可以定义回调来处理每个结果。

这个模块被认为是一个扩展的例子，而不是成品。随意调整它也是你的需求。

基本的使用方法：

>>> pool = ThreadPool(poolsize)

>>> requests = makeRequests(some\_callable, list\_of\_args, callback)

>>> [pool.putRequest(req) for req in requests]

>>> pool.wait()

查看模块源代码的结尾，获取更长的带注释的使用示例。

说明文档：

下载：

你可以通过下面的链接下载该模块的最新版本：

（1）[下载目录](https://chrisarndt.de/projects/threadpool/download/)

或者直接看源代码：

（2）[threadpool.py](https://chrisarndt.de/projects/threadpool/threadpool.py.html)

你还可以通过pip进行安装：

（3）[sudo] pip install threadpool

还或者你从github仓库中检出：

（4）git clone <https://github.com/SpotlightKid/threadpool.git>

9. 文件。创建一些线程来计算一些（可能很大量的）文件中一共有多少行。你可以选择要使用多少个线程。比较单线程与多线程的性能差异。提示：回顾一下第 9 章（文件和输入输出）的练习。

10. 把你之前的解决方案应用到你选择的几个任务中，如，处理一些 e-mail，下载一些网页，处理一些 RSS 和 Atom feeds，聊天时的消息处理，解一个迷题等。

# 第20章 Web编程

## 20.1 引言

Web 应用：客户端/服务器计算。

## 20.2 Python 的 Web 应用：简单的 Web 客户端

有一点需要记清楚，浏览器只是 Web 客户端的一种。任何一个通过向服务器端发送请求来获得数据的应用程序都被认为是“客户端”。 当然，也可以建立其他的客户端从而在因特网上检索出文档和数据。这样做的一个重要原因就是浏览器的能力有限，也就是说，它主要用于查看并同其他 Web站点交互。另一方面，一个客户端程序，有能力做得更多—它不仅可以下载数据，同时也可以存储、操作数据，甚或可以将其传送到另外一个地方或者传给另外一个应用。

一个使用 urllib 模块下载或者访问 Web 上的信息的应用程序[使用 urllib.urlopen() 或者urllib.urlre- trieve()]可以被认为是简单的 Web 客户端。你所要做的就是提供一个有效的 Web 地址。

### 20.2.1 URL

简单的 Web 应用包扩使用被称为 URL（统一资源定位器）的 Web 地址。这个地址用来在 Web 上定位一个文档，或者调用一个 CGI 程序来为你的客户端产生一个文档。URL 是大型标识符 URI（统一资源标识）的一部分。这个超集是建立在已有的命名惯例基础上的。一个 URL 是一个简单的 URI，使用已存在的协议或规划（也就是 http，ftp 等）作为地址的一部分。为了进一步描绘这些，我们将会引入 non-URL 的 URI，有时这些被成为 URN（统一资源名称），但是在今天我们唯一使用的一种 URI  
是 URL，至于 URI 和 URN 你也许没有听到太多，这或许已被保存成 XML 标识符了。如街道地址一样，Web 地址也有一些结构。美国的街道地址通常是这种格式“号码 街道名称”，例如 123 主大街。这个和其他国家不同，他们有自己的规则。URL 使用这种格式：

prot\_sch://net\_loc/path;params?query#frag

URL部件 描述  
prot\_sch 网络协议或者下载规划  
net\_loc 服务器位置(或许也有用户信息)  
path 斜杠( / )限定文件或者 CGI 应用程序的路径。  
Params 可选参数  
query 连接符( & )连接键值对  
frag 拆分文档中的特殊锚

net\_loc 可以进一步拆分成多个部件，有些是必备的，其他的是可选部件，net\_loc 字符串如下：

user:passwd@host:port

表 20.2 中描述了这些部件。

部件 描述  
user 登录名  
password 用户的密码  
host Web 服务器运行的机器  
port 端口号（默认 80）  
在这四个当中，host 主机名是最重要的。端口号只有在 Web 服务器运行其他非默认端口上时才会被使用。（如果你不确定所使用的端口号，可以参到第十六章）。

用户名和密码部分只有在使用 FTP 连接时候才有可能用到，因为即使是使用 FTP，大多数的连接都是使用匿名这时是不需要用户名和密码的。

Python 支持两种不同的模块，分别以不同的功能和兼容性来处理 URL。一种是 urlparse，一种是 urllib。这里我们将会简单的介绍下它们的功能。

### 20.2.2 urlparse 模块

（1）urlparse.urlparse()

urlparse 模块提供了操作 URL 字符串的基本功能。这些功能包括 urlparse(), urlunparse()和 urljoin()。urlparse()将 URL 字符串拆分成如上所描述的一些主要部件。语法结构如下：

urlparse(urlstr, defProtSch=None, allowFrag=None)

urlparse()将 urlstr 解析成一个 6-元组(prot\_sch, net\_loc,path, params, query, frag)。这里的每个部件在上边已经描述过了。如果 urlstr 中没有提供默认的网络协议或下载规划时可以使用 defProtSch。allowFrag 标识一个 URL 是否允许使用零部件。下边是一个给定 URL 经 urlparse() 后的输出：

>>>urlparse.urlparse('http://www.python.org/doc/FAQ.html')

('http', 'www.python.org', '/doc/FAQ.html', '', '', '')

（2）urlparse.urlunparse()

urlunparse()的功能与 urlpase()完全相反—它拼合一个 6-元组(prot\_sch, net\_loc, path,params, query, frag)- urltup,它可能是一个 URL 经 urlparse()后的输出返回值。于是，我们可以用如下方式表示：

urlunparse(urlparse(urlstr)) = urlstr

你或许已经猜到了 urlunpase()的语法：  
urlunparse(urltup)

（3）urlparse.urljoin()  
在需要多个相关的 URL 时我们就需要使用 urljoin()的功能了，如，在一个 Web 页中生成的一系列页面的 URL。Urljoin()的语法是：

urljoin(baseurl, newurl, allowFrag=None)

urljoin()取得 baseurl， 并将其基路径(net\_loc 附加一个完整的路径， 但是不包括终端的文件)与 newurl 连接起来。例如：

>>> urlparse.urljoin('http://www.python.org/doc/FAQ.html', \ 'current/lib/lib.htm')

'http://www.python.org/doc/current/lib/lib.html'

### 20.2.3 urllib 模块

**核心模块：urllib**

**urllib 模块提供了所有你需要的功能，除非你计划写一个更加低层的网络客户端。urllib 提供了了一个高级的 Web 交流库，支持 Web 协议， HTTP， FTP 和 Gopher 协议，同时也支持对本地文件的访问。urllib 模块的特殊功能是利用上述协议下载数据(从因特网、局域网、主机上下载)。使用这个模块可以避免使用 httplib， ftplib 和 gopherlib 这些模块，除非你想用更低层的功能。在那些情况下这些模块都是可选择的(注意：大多数以\*lib 命名的模块用于客户端相关协议开发。并不是所有情况都是这样的，或许 urllib 应该被命名为“internetlib”或者其他什么相似的名字)。**

Urllib 模块提供了在给定的 URL 地址下载数据的功能，同时也可以通过字符串的编码、解码来确保它们是有效 URL 字符串的一部分。我们接下来要谈的功能包括 urlopen(), urlretrieve(),quote(),unquote(), quote\_plus(), unquote\_plus(), 和 urlencode() 。 我 们 可 以 使 用urlopen()方法返回文件类型对象。你会觉得这些方法不陌生，因为在第九章我们已经涉及到了文件方面的内容。

（1）urllib.urlopen()  
urlopen() 打开一个给定 URL 字符串与 Web 连接，并返回了文件类的对象。语法结构如下：

urlopen(urlstr, postQueryData=None)

urlopen()打开 urlstr 所指向的 URL。 如果没有给定协议或者下载规划， 或者文件规划早已传入，urlopen()则会打开一个本地的文件。

对于所有的 HTTP 请求，常见的请求类型是“GET”。在这些情况中，向 Web 服务器发送的请求字符串(编码键值或引用，如 urlencode()函数的字符串输出[如下])应该是 urlstr 的一部分。

如果要求使用“POST”方法，请求的字符串（编码的）应该被放到 postQueryData 变量中。 （要了解更多关于“GET”和“POST”方法的信息，请查看 CGI 应用编程部分的普通文档或者文本，这些我们在下边也会讨论）。GET 和 POST 请求是向 Web 服务器上传数据的两种方法。

一旦连接成功，urlopen() 将会返回一个文件类型对象，就像在目标路径下打开了一个可读文件。例如，如果我们的文件对象是 f，那么我们的“句柄”将会支持可读方法如：f.read(),  
f.readline(), f.readlines(), f.close(),和 f.fileno().

此外，f.info()方法可以返回 MIME（Multipurpose Internet Mail Extension，多目标因特网邮件扩展）头文件。这个头文件通知浏览器返回的文件类型可以用哪类应用程序打开。例如，浏览器本身可以查看 HTML（HyperText Markup Language，超文本标记语言），纯文本文件，生成（指由数据显示图像——译者注）PNG（Portable Network Graohics）文件，JPEG（Joint Photographic Experts Group）或者 GIF（Graphics Interchange Format）文件。其他的如多媒体文件，特殊类型文件需要通过扩展的应用程序才能打开。

Table 20.4 urllib.urlopen() File-like Object Methods  
urlopen()对象方法 描述  
f.read([bytes]) 从 f 中读出所有或 bytes 个字节  
f.readline() 从 f 中读出一行  
f.readlines() 从 f 中读出所有行并返回一个列表  
f.close() 关闭 f 的 URL 的连接  
f.fileno() 返回 f 的文件句柄  
f.info() 获得 f 的 MIME 头文件  
f.geturl() 返回 f 所打开的真正的 URL

最后， geturl()方法在考虑了所有可能发生的间接导向后， 从最终打开的文件中获得真实的 URL，这些文件类型对象的方法在表 20.4 中有描述。如果你打算访问更加复杂的 URL 或者想要处理更复杂的情况如基于数字的权限验证，重定位，coockie 等问题，我们建议你使用 urllib2 模块，这个在 1.6 版本中有介绍（多数是试验模块）。它同时还有一个 urlopen()函数，但也提供了其他的可以打开各种 URL 的函数和类。关于 urllib2 的更多信息，将会在本章的下一部分介绍。

（2）urllib.urlretrieve()

如果你对整个 URL 文档的工作感兴趣，urlretrieve()可以帮你快速的处理一些繁重的工作。下边是 urlretrieve()的语法：

urlretrieve(urlstr, localfile=None, downloadStatusHook=None)

除了像 urlopen()这样从 URL 中读取内容，urlretrieve()可以方便地将 urlstr 定位到的整个HTML 文件下载到你本地的硬盘上。你可以将下载后的数据存成一个本地文件或者一个临时文件。如果该文件已经被复制到本地或者已经是一个本地文件，后续的下载动作将不会发生。

如果可能，downloadStatusHook 这个函数将会在每块数据下载或传输完成后被调用。调用时使用下边三个参数：目前读入的块数，块的字节数和文件的总字节数。如果你正在用文本的或图表的视图向用户演示“下载状态”信息，这个函数将会是非常有用的。  
urlretrieve()返回一个 2-元组，(filename, mime\_hdrs)。filename 是包含下载数据的本地文  
件名，mime\_hdrs 是对 Web 服务器响应后返回的一系列 MIME 文件头。要获得更多的信息，可以看mimetools 的 Message 类。对本地文件来说 mime\_hdrs 是空的。

关于 urlretrieve()的简单应用，可以看 11.4(grabweb.py)中的例子。在本章的 20.2 中将会介绍 urlretrieve()更深层的应用。

**（3）urllib.quote()** and **urllib.quote\_plus()**  
quote\*()函数获取 URL 数据，并将其编码，从而适用于 URL 字符串中。尤其是一些不能被打印的或者不被 Web 服务器作为有效 URL 接收的特殊字符串必须被转换。这就是 quote\*()函数的功能。

quote\*()函数的语法如下：

quote(urldata, safe='/')

逗号，下划线，句号，斜线和字母数字这类符号是不需要转化。其他的则均需要转换。另外，那些不被允许的字符前边会被加上百分号(%)同时转换成 16 进制,例如： “%xx”，“xx”代表这个字母的 ASCII 码的十六进制值。当调用 quote\*()时，urldata 字符串被转换成了一个可在 URL 字符串中使用的等价值。safe 字符串可以包含一系列的不能被转换的字符。默认的是斜线（/）.

quote\_plus() 与 quote()很像，另外它还可以将空格编码成+号。下边是一个使用 quote()和和quote\_plus()的例子：

>>> name = 'joe mama'  
>>> number = 6  
>>> base = 'http://www/~foo/cgi-bin/s.py'  
>>> final = '%s?name=%s&num=%d' % (base, name, number)  
>>> final  
'http://www/~foo/cgi-bin/s.py?name=joe mama&num=6'  
>>>  
>>> urllib.quote(final)  
'http:%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe%20mama%26num%3d6'  
>>>  
>>> urllib.quote\_plus(final)  
'http%3a//www/%7efoo/cgi-bin/  
s.py%3fname%3djoe+mama%26num%3d6'

（4）urllib.unquote() 和 urllib.unquote\_plus()  
也许和你猜到的一样， unquote\*()函数与 quote\*()函数的功能安全相反—它将所有编码为“%xx”式的字母都转换成它们的 ASCII 码值。Unquote\*()的语法如下：

unquote\*(urldata)

调用 unquote()函数将会把 urldata 中所有的 URL-编码字母都解码，并返回字符串。Unquote\_plus()函数会将加号转换成空格符。

（5）urllib.urlencode()

在 1.5.2 版的 Python 中，urlopen()函数接收字典的键-值对，并将其编译成 CGI 请求的 URL 字符串的一部分。键值对的格式是“键=值”,以连接符(&)划分。更进一步，键和它们的值被传到quote\_plus()函数中进行适当的编码。下边是 urlencode()输出的一个例子：

>>> aDict = { 'name': 'Georgina Garcia', 'hmdir': '~ggarcia' }  
>>> urllib.urlencode(aDict)  
'name=Georgina+Garcia&hmdir=%7eggarcia'  
urllib 和 urlparse 还有一些其他的功能，在这里我们就不一一概述了。阅读相关文档可以获得更多信息。  
（6）安全套接字层支持  
在 1.6 版中 urllib 模块通过安全套接字层(SSL)支持开放的 HTTP 连接.socket 模块的核心变化是增加并实现了 SSL。 随后，urllib 和 httplib 模块被上传用于支持 URL 在“https”连接规划中的应用。除了那两个模块以外，其他的含有 SSL 的模块还有： imaplib, poplib 和 smtplib。

### 20.2.4 urllib2 模块

正如前面所提到的，urllib2 可以处理更复杂 URL 的打开问题。一个例子就是有基本认证（登录名和密码）需求的 Web 站点。最简单的“获得已验证参数”的方法是使用前边章节中描述的 URL 部件 net\_loc，也就是说：http://user:passwd@www.python.org.这种解决方案的问题是不具有可编程性。然而使用 urllib2，我们可以通过两种不同的方式来解决这个问题。  
我们可以建立一个基础认证处理器(urllib2.HTTPBasicAuthHandler),同时在基本 URL 或域上注册一个登录密码，这就意味着我们在 Web 站点上定义了个安全区域。 （关于域的更多信息可以查看RFC2617[HTTP 认证：基本数字认证]）。一旦完成这些，你可以安装 URL 打开器，通过这个处理器打开所有的 URL。

另一个可选的办法就是当浏览器提示的时候，输入用户名和密码 ，这样就发送了一个带有适当用户请求的认证头。在 20.1 的例子中，我们可以很容易的区分出这两种方法。

逐行解释

逐行解释  
1–7 行

普通的初始化过程，外加几个为后续脚本使用的常量。

9–15 行

代码的“handler”版本分配了一个前面提到的基本处理器类，并添加了认证信息。之后该处理器被用于建立一个 URL-opener，并安装它以便所有已打开的 URL 能用到这些认证信息。这段代码和urllib2 模块的 Python 官方文档是兼容的。

## 20.3 高级的 Web 客户端

Web 浏览器是基本的 Web 客户端。主要用来在 Web 上查询或者下载文件。而 Web 的高级客户端并不只是从因特网上下载文档。

高级 Web 客户端的一个例子就是网络爬虫（aka 蜘蛛和机器人）。这些程序可以基于不同目的在因特网上探索和下载页面，其中包括：

* 为 Google 和 Yahoo 这类大型的搜索引擎建索引
* 脱机浏览—将文档下载到本地，重新设定超链接，为本地浏览器创建镜像。
* 下载并保存历史记录或框架
* Web 页的缓存，节省再次访问 Web 站点的下载时间。

我们下边介绍网络爬虫:crawl.py,抓取 Web 的开始页面地址（URL），下载该页面和其它后续链 接页面，但是仅限于那些与开始页面有着相同域名的页面。如果没有这个限制的话，你的硬盘将会 被耗尽！crwal.py 的代码在例子 20.2 中展示。

例20.2 crwal.py代码可以在[github](https://github.com/tupelo-shen/python_test/blob/master/code/sources/chapter20/crawl.py)上看到。这个爬虫程序包括两个类，一个管理真个 crawling 进程（Crawler）,一个检索并解析每一个下 载的 Web 页面（Retriever）。

逐行解释（一个类一个类的）

1–11 行

该脚本的开始部分包括 Python 在 Unix 上标准的初始化行以及一些模块属性的导入，它们都会 在本应用程序中用到。

13–49 行

Retriever 类的责任是从 Web 下载页面，解析每个文档中的链接并在必要的时候把它们加入 “to-do”队列。我们为每个从网上下载的页面都创建一个 Retriever 类的实例。Retriever 中的方 法展现了它的功能：构造器（\_\_init\_\_()）、filename()、download()、和 parseAndGetLinks()。

filename()方法使用给定的 URL 找出安全、有效的相关文件名并存储在本地。大体上说，它会 去掉 URL 的“http://”前缀，使用剩余的部分作为文件名，并创建必要的文件夹路径。那些没有 文件名前缀的 URL 则会被赋予一个默认的文件名“index.htm”。（可以在调用 filename()时重新指定 这个名字。 ）

构造器实例化了一个 Retriever 对象，并把 URL 和通过 filename()获得的相应文件名都作为本地属性保存起来。

正如你想象的， download()方法实际会连上网络去下载给定链接的页面。它使用 URL 调用 urllib.urlretrieve()函数并把结果保存在 filename 中（该值由 filename()返回）。 如果下载成功， parse()方法会被调用来解析刚从网络拷贝下来的页面；否则会返回一个错误字符串。

如果 Crawler 判定没有错误发生，它会调用 parseAndGetLinks()方法来解析新下载的页面并决 定该页面中每个链接的后续动作。

51-98 行 Crawler 类是这次演示中的“明星”，掌管在一个 Web 站点上的整个抓爬过程。如果我们为应用程序添加线程，就可以为每个待抓爬的站点分别创建实例。Crawler 的构造器在初始化过程中存储了 3 样东西，第一个是 q，一个待下载链接的队列。这个队列在运行过程中会有涨落，有页面处理完毕 它就变短，在下载的页面中发现新的链接则会让它变长。

Crawler 包含的另两个数值是 seen，一个所有“我们已看过”（已下载）的链接的列表，和 dom， 我们把主链接的域名存储在这里，并用这个值来判定后续链接是否是该域的一部分。

Crawler 还有一个静态数据成员 count。这个计数器只是用来保存我们已经从网上下载的对象数 目。每有一个页面成功下载它就会增加。

除了构造器 Crawler 还有其他两个方法，getPage()和 go()。go()只是简单的启动 Crawler，它 在代码的主体部分被调用。 go()中有一个循环， 只有队列中还有待下载的新链接它就会不停的执行。 然而这个的真正工作者，却是 getPage()方法。

getPage()初始化了一个 Retriever 对象，并把第一个链接赋给它然后让它执行。如果页面下载 成功，计数器会增加并且链接会被加到“已看”列表。它会反复地检查每个已下载页面中的所有链 接并判决是否有链接要被加入待下载队列。go()中的主循环会不停的推进处理过程直到队列为空， 这时便大功告成。

属于其他域的链接、已经下载过的链接、已在队列中待处理的链接、以及“mailto:”类型的 链接在扩充队列时都会被忽略掉。

100-114 行 main()是程序运行的起点，它在该脚本被直接调用时执行。其他导入 crawl.py 的模块则需要调 用 main()来启动处理过程。main()需要一个 URL 来启动处理，如果在命令行指定了一个（例如这个 脚本被直接调用时），它就会使用这个指定的。否则，脚本进入交互模式，提示用户输入起始 URL。 一旦有了起始链接，Crawler 就会被实例化并启动开来。

运行结果：

Enter starting URL:**www.baidu.com/index.html**

('www.baidu.com/index', '.html')

( 1 )

URL: http://www.baidu.com/index.html

FILE: www.baidu.com/index.html

\* http://www.baidu.com/

……（省略）

( 2 )

URL: http://www.baidu.com/duty/

FILE: www.baidu.com/duty/index.html

... new, added to Q

……（省略）

( 3 )

URL: http://www.baidu.com/duty/index.html

FILE: www.baidu.com/duty/index.html

... discarded, already in Q

……（省略）

( 4 )

URL: http://www.baidu.com/duty/baozhang.html

FILE: www.baidu.com/duty/baozhang.html

... discarded, already in Q

……（省略）

( 5 )

URL: http://www.baidu.com/duty/yinsiquan.html

FILE: www.baidu.com/duty/yinsiquan.html

... discarded, already in Q

... discarded, already in Q

... discarded, not in domain

\* http://www.baidu.com/duty/copyright.html

……（省略）

( 6 )

URL: https://www.baidu.com/more/

FILE: \*\*\* ERROR: invalid URL "https://www.baidu.com/more/"

## 20.4 CGI: 帮助Web服务器处理客户端数据

### 20.4.1 CGI 介绍

Web 开发的最初目的是在全球范围内对文档进行存储和归档（大多是教学和科研目的的）。这些零碎的信息通常产生于静态的文本或者 HTML。

HTML 是一个文本格式而算不上是一种语言，它包括改变字体的类型、大小、风格。HTML 的主要 特性在于它对超文本的兼容性，文本以一种或者是高亮的形式指向另外一个相关文档。可以通过鼠 标点击或者其他用户的选择机制来访问这类文档。这些静态的 HTML 文档在 Web 服务器上，当有请求 时，将被送到客户端。

随着因特网和 Web 服务器的形成， 产生了处理用户输入的需求。 在线零售商需要能够单独订货， 网上银行和搜索引擎需要为用户分别建立帐号。因此发明了这种执行模式，并成为了 Web 站点可以从用户那里获得特苏信息的唯一形式（在 Java applets 出现之前）。反过来，在客户提交了特定数 据后，就要求立即生成 HTML 页面。

现在 Web 服务器仅有一点做的很不错，获取用户对文件的请求，并将这个文件（也就是说 HTML 文件）返回给客户端。它们现在还不具有处理字段类特殊数据的机制。将这些请求送到可以生成动 态 HTML 页面的扩展应用程序中并返回给客户端，这些还没有成为 Web 服务器的职责。

这整个过程开始于 Web 服务器从客户端接到了请求（GET 或者 POST），并调用合适的程序。然后开始等待 HTML 页面—与此同时，客户端也在等待。一旦程序完成，会将生成的动态 HTML 页面返回 到服务器端，然后服务器端再将这个最终结果返回给用户。服务器接到表单反馈，与外部应用程序交互，收到并返回新生成的 HTML 页面都发生在一个叫做 Web 服务器 CGI（Common Gateway Interface） 的接口上.图 20-3 描述了 CGI 的工作原理，逐步展示了一个用户从提交表单到返回最终结果 Web 页 面的整个执行过程和数据流。

客户端输入给 Web 服务器端的表单可能包括处理过程和一些存储在后台数据库中的表单。需要记住的是，在任何时候都可能有任何一个用户去填写这个字段，或者点击提交按钮或者图片，这更 像激活了某种 CGI 活动。

创建 HTML 的 CGI 应用程序通常是用高级编程语言来实现的，可以接受、处理数据，向服务器端返回HTML 页面。目前使用的编程语言有 Perl， PHP， C/C++,或者 Python。

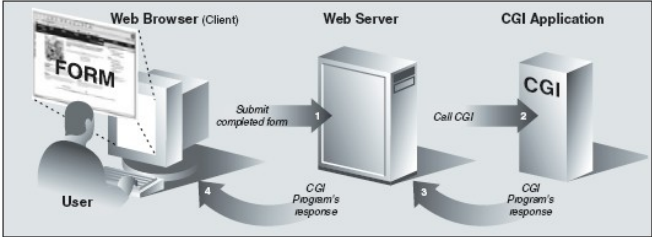


图 20-3 CGI 工作概要图。CGI 代表了在一个 Web 服务器和能够处理用户表单、生成并返回动态 HTML 页的应用程序间的交互。

在我们研究 CGI 之前，我们必须告诉你典型的 Web 应用产品已经不再使用 CGI 了。

由于它词义的局限性和允许 Web 服务器处理大量模拟客户端数据能力的局限性，CGI 几乎绝迹。 Web 服务的关键使命依赖于遵循像 C/C++这样语言的规范。如今的 Web 服务器典型的部件有 Aphache 和集成的数据库部件（MySQL 或者 PostgreSQL），Java（Tomcat），PHP 和各种 Perl 模块，Python 模块，以及 SSL/security。然而，如果你工作在私人小型的或者小组织的 Web 网站上的话就没有必要使用这种强大而复杂的 Web 服务器， CGI 是一个适用于小型 Web 网站开发的工具。

更进一步来说， 有很多 Web 应用程序开发框架和内容管理系统， 这些都弥补了过去 CGI 的不足。 然而，在这些浓缩和升华下，它们仍旧遵循这 CGI 最初提供的模式，可以允许用户输入，根据输入 执行拷贝，并提供了一个有效的 HTML 做为最终的客户端输出。因此，为了开发更加高效的 Web 服务有必要理解 CGI 实现的基本原理。

### 20.4.2 CGI 应用程序

CGI 应用程序和典型的应用程序有些不同。主要的区别在于输入、输出以及用户和计算机交互方面。当一个 CGI 脚本开始执行时，它需要检索用户-支持表单，但这些数据必须要从 Web 的客户端才可以获得，而不是从服务器或者硬盘上获得。

这些不同于标准输出的输出将会返回到连接的Web客户端，而不是返回到屏幕、CUI 窗口或者硬盘上。这些返回来的数据必须是具有一系列有效头文件的HTML。否则，如果浏览器是 Web 的客户端，由于浏览器只能识别有效的 HTTP 数据（也就是 MIME和 HTML），那么返回的也只能是个错误消息（具体的就是因特网服务器错误）。

最后，可能和你想象的一样，用户不能与脚本进行交互。所有的交互都将发生在 Web 客户端（用户的行为），Web 服务器端和 CGI 应用程序间。

### 20.4.3 CGI模块

在 cgi 模块中有个主要类：FieldStorage 类，它完成了所有的工作。在 Python CGI 脚本开始时 这个类将会被实例化，它会从 Web 客户端（具有 Web 服务器）读出有关的用户信息。一旦这个对象 被实例化，它将会包含一个类似字典的对象，具有一系列的键-值对，键就是通过表单传入的表单条 目的名字，而值则包含相应的数据。

这些值本身可以是以下三种对象之一。它们既可以是 FieldStorage 对象（实例）也可以是另一 个类似的名为 MiniFieldStorage 类的实例，后者用在没有文件上传或 mulitple-part 格式数据的情 况。MiniFieldStorage 实例只包含名字和数据的键-值对。最后，它们还可以是这些对象的列表。这发生在表单中的某个域有多个输入值的情况下。

对于简单的 Web 表单，你将会经常发现所有的 MiniFieldStorage 实例。下边包含的所有的例子 都仅针对这种情况。

## 20.5 创建 CGI 应用程序

### 20.5.1 建立 Web 服务器

为了可以用 Python 进行 CGI 开发，你首先需要安装一个 Web 服务器， 将其配置成可以处理 Python CGI 请求的模式，然后让你的 Web 服务器访问 CGI 脚本。其中有些操作你也许需要获得系统管理员的帮助。

如果你需要一个真正的 Web 服务器，可以下载并安装 Apache。Aphache 的插件或模块可以处理 Python CGI，但这在我们的例子里并不是必要的。如果你准备把自己的服务"带入真实世界",也许会想安装这些软件。尽管它们似乎过于强大。

为了学习的目的或者是建立小型的 Web 站点，使用 Python 自身带的 Web 服务器就已经足够了。 在第 20.8 节，你将会实际的学习如何建立和配置简单的基于 Python 的 Web 服务器。如果你想在本阶段获得更多知识，你也可以现在提前阅读那部分。然而，这并不是本章的焦点。

如果你只是想建立一个基于 Web 的服务器，你可以直接执行下边的 Python 语句。

$ **python -m CGIHTTPServer**

-m 选项是在 2.4 中新引进的，如果你使用的是比这旧的 Python 版本，或者想看下它执行的不同方式，请看 14.4.3. 无论如何，最终它需要工作起来...

这将会在当前机器的当前目录下建立一个端口号为 8000 的 Web 服务器。然后可以在启动这个服务器的目录下建立一个cgi-bin， 将 Python CGI 脚本放到那里。 将一些 HTML 文件放到那个目录下，或许有些.py CGI 脚本在cgi-bin/中，然后就可以在地址栏中输入这些地址来访问 Web 站点啦。

http://localhost:8000/friends.html

http://localhost:8000/cgi-bin/friends2.py

### 20.5.2 建立表单页

在例 20.3 中，我们写了一个简单的 Web 表单，friends.html。

<HTML><HEAD><TITLE>Friends CGI Demo (static screen)</TITLE></HEAD>

<BODY><H3>Friends list for: <I>NEW USER</I></H3>

<FORM ACTION="/cgi-bin/friends1.py">

<B>Enter your Name:</B>

<INPUT TYPE=text NAME=person VALUE="NEW USER" SIZE=15>

<P><B>How many friends do you have?</B>

<INPUT TYPE=radio NAME=howmany VALUE="0" CHECKED> 0

<INPUT TYPE=radio NAME=howmany VALUE="10"> 10

<INPUT TYPE=radio NAME=howmany VALUE="25"> 25

<INPUT TYPE=radio NAME=howmany VALUE="50"> 50

<INPUT TYPE=radio NAME=howmany VALUE="100"> 100

<P><INPUT TYPE=submit>

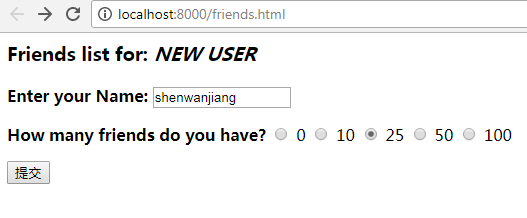
</FORM>

</BODY>

</HTML>

正如你可以在代码中看到的一样，这个表单包括两个输入变量：person和howmany，这两个值将会被传到我们的CGI脚本friends1.py中。 你会注意到在例子中我们将CGI脚本初始化到主机默认的cgi-bin目录下（“Action” 连接）。（如果这个信息与你开发环境不一样的话，在测试 Web页面和CGI之前请更新你的表单事件）。同时由于表单事件中缺少METHOD 子标签，所有的请求将会采用默认的GET方法。选择GET方法是因为我们的表单没有太多的字段，同时我们希望我们的请求字段可以在“位置”（aka“Address”， “Go To”） 条中显示，以便你可以看到被送到服务器端的 URL。

如图所示：



上图是在chrome浏览器的查看结果。

### 20.5.3 生成结果页

这些输入是由用户完成的，然后按下了“Submit”按钮（可选的，用户也可以在该文本字段中按下回车键获得相同的效果。）当这些发生后，在例20.4中的脚本， friends1.py 将会随 CGI 一起被执行。

这个脚本包含了所有的编程功能，读出并处理表单的输入，同时向用户返回结果HTML页面。所有的这些“实际”的工作仅是通过四行Python代码来实现的（14-17 行）。

表单的变量是 FieldStorage 的实例，包含 person 和 howmanyh 字段的值。我们把这些值本分别 存入Python的who和howmany变量中。变量reshtml包含需要返回的HTML文本的正文，还有一些动态填好的字段，这些数据都是从表单中读入的。

#! /usr/bin/env python

# -\*- coding: UTF-8 -\*-

'''

Results Screen CGI code

'''

import cgi

reshtml = '''Content-Type: text/html \n

<HTML>

<HEAD>

<TITLE>

Friends CGI Demo (dynamic screen)

</TITLE>

</HEAD>

<BODY>

<H3>

Friends list for: <I>%s</I>

</H3>

Your name is: <B>%s</B><P>

You have <B>%s</B> friends

</BODY>

</HTML>'''

form = cgi.FieldStorage()

who = form['person'].value

howmany = form['howmany'].value

print reshtml % (who, who, howmany)

**核心提示: HTML 头文件是从 HTML 中分离出来的。**

**有一点需要向 CGI 初学者指明的是，在向 CGI 脚本返回结果时，须先返回一个适当的 HTTP 头文 件后才会返回结果 HTML 页面。 进一步说， 为了区分这些头文件和结果 HTML 页面， 需要在 friends1.py 的第五行中插入几个换行符。在本章的后边的代码中也是这样的处理。**

图 20-6 是可能出现的屏幕，假设用户输入的名字为“erick allen”,单击“10 friends”单选 按钮。

如果你是一个Web 站点的生产商，你也许会想，“如果这个人忘记了的话，我能自动的将这个人的名字大写，会不会更好些？”这个通过 Python 的 CGI 可以很容易就实现（我们很快就会进行试验！）。

注意 GET 请求是如何将表单中的变量和值加载在URL地址条中的。你是否观察到了 friends.html页面的标题有个“static”， 而 friends.py脚本输出到屏幕上的则是“ dynamic”？我们这样做的一个原因就是：指明 friends.html 文件是一个静态的文本，而结果页面却是动态生成的。换句话说，结果页面的 HTML 不是以文本文件的形式存在硬盘上的，而是由我们的CGI脚本生成的，并且将其以本地文件的形式返回。



图20-6 Chrome浏览器上的显示结果

在下边的例子中，我们将会更新我们的 CGI 脚本，使其变得更灵活些，从而完全绕过静态文件。

### 20.5.4 生成表单和结果页面

我们删除fiends.html文件并将其合并到friends2.py中。这个脚本现在将会同时生成表单页和结果页面。但是我们如何控制生成哪个页面呢？好吧，如果有表单数据被发送，那就意味着我们需要建立一个结果页面。如果我们没有获得任何的信息，这就说明我们需要生成一个用户可以输入数据的表单页面。

例子 20.5 展示的就是我们的新脚本 friends2.py 那么我们改变了哪些脚本呢？让我们一起看下这个脚本的代码块。

#! /usr/bin/env python

# -\*- coding: UTF-8 -\*-

import cgi

header = 'Content-Type: text/html \n\n'

formhtml = '''<HTML><HEAD><TITLE>

Friends CGI Demo</TITLE></HEAD>

<BODY><H3> Friends list for: <I>NEW USER</I> </H3>

<FORM ACTION="/cgi-bin/friends2.py">

<B>Enter your Name:</B>

<INPUT TYPE=hidden NAME=action VALUE=edit>

<INPUT TYPE=text NAME=person VALUE="NEW USER" SIZE=15>

<P><B>How many friends do you have?</B>

%s

<P><INPUT TYPE=submit></FORM></BODY></HTML>'''

fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'

def showForm():

friends = ''

for i in [0, 10, 25, 50, 100]:

checked = ''

if i == 0:

checked = 'CHECKED'

friends = friends + fradio % (str(i), checked, str(i))

print header + formhtml % (friends)

reshtml = '''<HTML><HEAD><TITLE>Friends CGI Demo</TITLE></HEAD>

<BODY><H3>Friends list for: <I>%s</I></H3>

Your name is: <B>%s</B><P>

You have <B>%s</B> friends

</BODY>

</HTML>'''

def doResults(who, howmany):

print header + reshtml % (who, who, howmany)

def process():

form = cgi.FieldStorage()

if form.has\_key('person'):

who = form['person'].value

else:

who = 'NEW USER'

if form.has\_key('howmany'):

howmany = form['howmany'].value

else:

howmany = 0

if form.has\_key('action'):

doResults(who, howmany)

else:

showForm()

if \_\_name\_\_ == '\_\_main\_\_':

process()

例 20.5生成表单和结果页面（friends2.py) 将 friends.html 和 friends1.py 合并成 friends2.py。得到的脚本可以同时显示表单和动态生成的HTML结果页面，同时可以巧妙的知道应该输出哪个页面。

逐行解释

1-5 行

除了通常的起始、和模块导入行，我们还把 HTTP MIMI 头从后面的HTML正文部分分离出来，放在了这里。因为我们将在返回的两种页面（表单页面和结果页面）中都使用它，而又不想重复写文本。当需要输出时，我们将把这个头字串加在相应的HTML正文中。

7-28行

所有这些代码都是为了整合 CGI 脚本里的 friends.html表单页面。我们对表单页面的文本使用一个变量formhtml，还有一个用来创建单选按钮的字符串变量fradio。我们从 friends.html复制了这个单选按钮HTML文本，但我们意在展示如何使用Python来生成更多的动态输出—见 22-26行的for循环。

showForm()函数负责对用户输入生成表单页。它为单选按钮创建了一个文字集，并把这些 HTML文本行合并到了formhtml主体中，然后给表单加上头信息，最后通过把整个字符串输出到标准输出的方式给客户端返回了整块数据。

这段代码中有两件有趣的事值得注意。第一点是表单中12行action处的“hidden”变量，这里的值为“edit”。我们决定显示哪个页面（表单页面或是结果页面）的唯一途径是通过这个字段。我们将在第 53-56 行看到这个字段如何起作用。

还有，请注意我们在生成所有按钮的循环里把单选按钮0设置为默认按钮。这表明我们可以在一行代码里（第18行）更新单选按钮的布局和/或它们的值，而不用再写多行文字。这也同时提供了更多的灵活性，可以用逻辑来判断哪个单选按钮被选中—见我们脚本的下一个升级版，后面的 friends3.py。现在你或许会想“既然我也可以选择 person 或 howmany 是否出现， 那为什么我们要用一个 action 变量呢？”这是一个很好的问题，因为在这种情况下你当然可以只用 person 或 hwomany。 然而，action 变量代表了一种更明显的出现，不光是它的名字还有它的作用—其代码很容易理解。person 和 howmany 变量都是对其值起作用，而 action 变量则被用作一个标志。

创立action的另一个原因是我们将会再一次使用它来帮助我们决定生成哪一页。具体来说，我们需要在person变量出现时会显示一个表单（而不是生成结果页面） --如果在这里仅依赖person变量,你的代码运行将失败。

31-39行

显示结果页的代码实际上和friends1.py中的一样。

41-56 行

因为这个脚本可以产生出不同的页面，所以我们创建了一个包括一切的process()函数来获得表单数据并决定采用何种动作。看起来process()的主体部分也和friends1.py中主体部分的代码相似。

然而它们有两个主要的不同。因为这个脚本也许可以、也许不能取得所期待的字段（例如，第一次运行脚本时生成一个表单页，这样的话就不会给服务器传递任何字段），我们需要用 if 语句把从表单项取得的值“括起来”， 并检查它们此时是否有效。还有我们上面提到的 action 字段，它可以帮助我们判定应生成哪一个页面。第50-53行作了这种判定。 在图20-7和图20-8中，你会先看到脚本生成的表单页面（已经输入了一个名字并选择了一个单选按钮），然后是结果页面，也是这个脚本生成的。

### 20.5.5 全面交互的 Web 站点

我们最后一个例子将会完成这个循环。如在前面中，用户在表单页中输入他/她的信息，然后我们处理这些数据，并输出一个结果页面。现在我们将会在结果页面上加个链接允许返回到表单页面，但是我们返回的是含有用户输入信息的页面而不是一个空白页面。我们页加上了一些错误处理程序，来展示它是如何实现的。

现在在例子 20.6 中我们展示我们最后的更新，friends3.py。

#! /usr/bin/env python

# -\*- coding: UTF-8 -\*-

import cgi

from urllib import quote\_plus

from string import capwords

header = 'Content-Type: text/html \n\n'

url = '/cgi-bin/friends3.py'

errhtml = '''

<HTML><HEAD><TITLE>Friends CGI Demo</TITLE></HEAD>

<BODY><H3> ERROR </H3>

<B>%s</B><P>

<FORM><INPUT TYPE=button VALUE=Back ONCLICK=" window.history.back()"</FORM>

</BODY>

</HTML>'''

def showError(error\_str):

print header + errhtml % (error\_str)

pass

formhtml = '''<HTML><HEAD><TITLE>

Friends CGI Demo</TITLE></HEAD>

<BODY><H3> Friends list for: <I>%s</I></H3>

<FORM ACTION="%s">

<B>Enter your Name:</B>

<INPUT TYPE=hidden NAME=action VALUE=edit>

<INPUT TYPE=text NAME=person VALUE="%s" SIZE=15>

<P><B>How many friends do you have?</B>

%s

<P><INPUT TYPE=submit></FORM></BODY></HTML>'''

fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'

def showForm(who, howmany):

friends = ''

for i in [0, 10, 25, 50, 100]:

checked = ''

if i == howmany:

checked = 'CHECKED'

friends = friends + fradio % (str(i), checked, str(i))

print header + formhtml % (who, url, who, friends)

reshtml = '''

<HTML><HEAD><TITLE>Friends CGI Demo</TITLE></HEAD>

<BODY><H3>Friends list for: <I>%s</I></H3>

Your name is: <B>%s</B><P>

You have <B>%s</B> friends

<P>Click <A HREF="%s">here</A> to edit your data again.

</BODY>

</HTML>'''

def doResults(who, howmany):

newurl = url+'?action=reedit&person=%s&howmany=%s' % (quote\_plus(who), howmany)

print header + reshtml % (who, who, howmany, newurl)

def process():

error = ''

form = cgi.FieldStorage()

if form.has\_key('person'):

who = form['person'].value

else:

who = 'NEW USER'

if form.has\_key('howmany'):

howmany = form['howmany'].value

else:

if form.has\_key('action') and form['action'].value == 'edit':

error = 'Please select number of friends.'

else:

howmany = 0

if not error:

if form.has\_key('action') and form['action'].value != 'reedit':

doResults(who, howmany)

else:

showForm(who, howmany)

else:

showError(error)

if \_\_name\_\_ == '\_\_main\_\_':

process()

friends3.py 和 friends2.py 没有太大的不同。我们请读者比较不同处；这里我们简要的介绍了主要的不同点。

简略的逐行解释

第10行

我们把 URL 从表单中抽出来是因为现在有2个地方需要它，结果页面是它的新顾客。

10-19， 69-71， 75-82 行 所有这些行都用来处理新特性—错误页面。如果用户没有选择单选按钮，指明朋友数量，那么 howmany 字段就不会传送给服务器，在这种情况下，showError()函数会返回一个错误页面给客户。

错误页面的显示使用了 JavaScript 的“后退”按钮。因为按钮都是输入类型的，所以需要一个 表单，但不需要有动作因为我们只是简单的后退到浏览器历史中的上一个页面。尽管我们的脚本目 前只支持（或者说探测、测试）一种类型的错误， 但我们仍然使用了一个通用的 error 变量，这是为了以后还可以继续开发这个脚本，给它增加更多的错误检测。

例 20.6 全用于交互和错误处理(friends3.py)

通过加上返回输入信息的表单页面的连接，我们实现了整个循环，给了用户一次完整的 Web 应用体验。我们的应用程序现在也进行了一些简单的错误验证，在用户没有选择任何单选按钮时，可以通知用户。

#! /usr/bin/env python

# -\*- coding: UTF-8 -\*-

import cgi

from urllib import quote\_plus

from string import capwords

header = 'Content-Type: text/html \n\n'

url = '/cgi-bin/friends3.py'

errhtml = '''

<HTML><HEAD><TITLE>Friends CGI Demo</TITLE></HEAD>

<BODY><H3> ERROR </H3>

<B>%s</B><P>

<FORM><INPUT TYPE=button VALUE=Back ONCLICK="window.history.back()"</FORM>

</BODY>

</HTML>'''

def showError(error\_str):

print header + errhtml % (error\_str)

pass

formhtml = '''<HTML><HEAD><TITLE>

Friends CGI Demo</TITLE></HEAD>

<BODY><H3> Friends list for: <I>%s</I></H3>

<FORM ACTION="%s">

<B>Enter your Name:</B>

<INPUT TYPE=hidden NAME=action VALUE=edit>

<INPUT TYPE=text NAME=person VALUE="%s" SIZE=15>

<P><B>How many friends do you have?</B>

%s

<P><INPUT TYPE=submit></FORM></BODY></HTML>'''

fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'

def showForm(who, howmany):

friends = ''

for i in [0, 10, 25, 50, 100]:

checked = ''

if str(i) == howmany:

checked = 'CHECKED'

friends = friends + fradio % (str(i), checked, str(i))

print header + formhtml % (who, url, who, friends)

reshtml = '''

<HTML><HEAD><TITLE>Friends CGI Demo</TITLE></HEAD>

<BODY><H3>Friends list for: <I>%s</I></H3>

Your name is: <B>%s</B><P>

You have <B>%s</B> friends

<P>Click <A HREF="%s">here</A> to edit your data again.

</BODY>

</HTML>'''

def doResults(who, howmany):

newurl = url+'?action=reedit&person=%s&howmany=%s' % (quote\_plus(who), howmany)

print header + reshtml % (who, who, howmany, newurl)

def process():

error = ''

form = cgi.FieldStorage()

if form.has\_key('person'):

who = capwords(form['person'].value)

else:

who = 'NEW USER'

if form.has\_key('howmany'):

howmany = form['howmany'].value

else:

if form.has\_key('action') and form['action'].value == 'edit':

error = 'Please select number of friends.'

else:

howmany = 0

if not error:

if form.has\_key('action') and form['action'].value != 'reedit':

doResults(who, howmany)

else:

showForm(who, howmany)

else:

showError(error)

if \_\_name\_\_ == '\_\_main\_\_':

process()

31，42-44，52，56-59 行

这个脚本的一个目的是创建一个有意义的链接，以便从结果页面返回表单页面。当有错误发生时，用户可以使用这个链接返回表单页面去更新他/她填写的数据。新的表单页面只有当它包含了用户先前输入的信息时才有意义。（如果让用户重复输入这些信息会很令人沮丧！ ）

为了实现这一点，我们需要把当前值嵌入到更新过的表单中。在第31行，我们给 name 新增了一个值。这个值如果给出的话，会被插入到 name 字段。显然地，在初始表单页面上它将是空值。第42-44行，我们根据当前选定的朋友数目设置了单选按钮。最后，通过第52行和56-59行更新了的 doResults()函数，我们创建了这个包含已有信息的链接，它会让用户“返回”到我们更改后的表单页面。

**62 行**

最后我们从美学角度上加了一个简单的特性。在 friends1.py 和 friends2.py 的截屏中，可以看到返回结果和用户的输入一字不差。在上述的截屏中，如果用户的名字没有大写这将影响返回的页面。我们加了一个对 string.capwords()函数的调用从而自动的将用户名置成大写。capwords() 函数可以将传进来的每个单词的第一个字母置成大写的。这或许不是必要的特性，但是我们还是愿意一起分享它，以便你知道这个功能的存在。

下边我们将会展示四个截屏，表明用户和 CGI 表单及脚本的交互过程。 在第一个截屏图20－9中， 我们调用 friends3.py 生成了一个熟悉的新表单页面。 输入“fool bar”，同时故意忘记检查单选按钮。单击 Submit 按钮后将会返回错误页面，请看第二个截屏图 20-10。

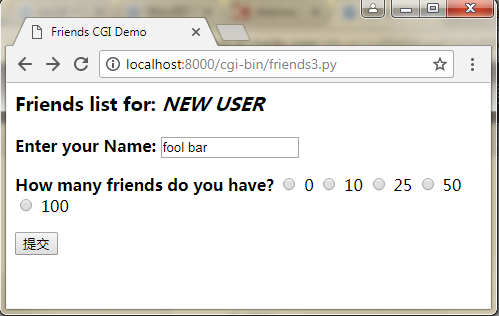


图20-9

图 20-9 friends的初始表单页面在Windows7 Pro操作系统 Chrome 浏览器上的显示（friends3.py）

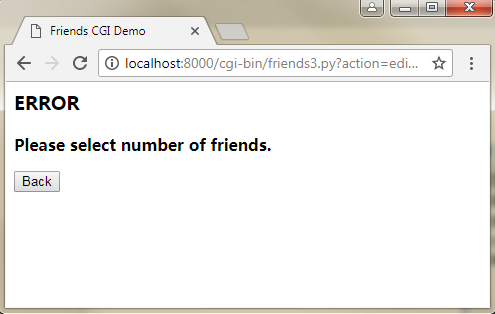


图 20-10 Friends 的错误页面（无效的用户输入）在Chrome浏览器上的显示（Friends3.py）

我们单击“后退”按钮，选择“50”单选按钮，重新提交表单。结果页面如图 20-11，也非常像， 但是现在在页面底部有个额外的连接。这个连接将会把我们带到表单页面。新表单页面和最初的页 面的唯一区别是所有用户输入的数据都被设置成了“默认值”， 这意味着这些值在表单中已经存在了。 我们可以看图 20-12。

这时用户可以更改任何一个字段或者重新提交表单。

毫无疑问你会开始注意到我们的表单和数据已经变得复杂多了，生成的 HTML 页面是这样，结果页面更是复杂。如果你有HTML文本和应用程序的接入点的话，你可能会考虑与Python 的HTMLgen模块的连接，HTMLgen 是Python的一个扩展模块，专用于生成 HTML 页面。

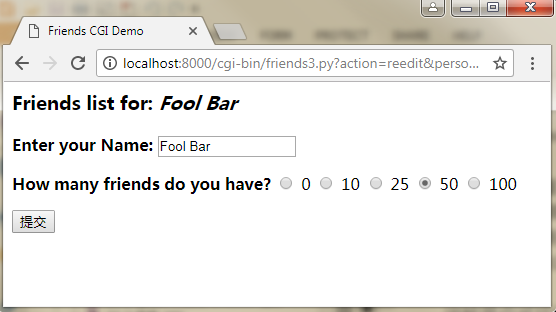


图 20-11 带有当前信息的更新后的 friends 表单页面

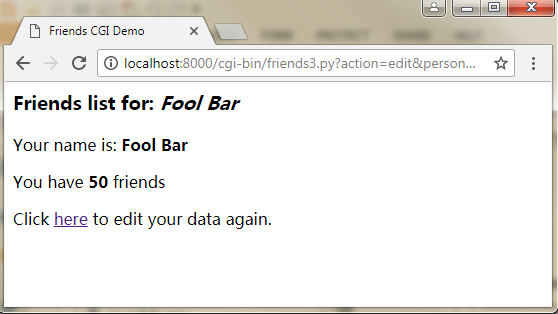


图 20-12 friends 结果页面（有效输入）（friends3.py）

## 20.6 在 CGI 中使用 Unicode 编码

在第六章“序列”中，我们介绍了 Unicode 字符串的使用。在 6.8.5 部分，我们给了个简单的 例子脚本： 取得 Unicode 字符串， 写入一个文件， 并重新读出来。 在这里， 我们将演示一个具有 Unicode 输出的简单 CGI 脚本，并给浏览器足够的提示，从而可以正确的生成这些字符。

唯一的要求是你的计算机必须装有对应的东亚字体以便浏览器可以显示它们。

为了看到 Unicode 的作用，我们将会用 CGI 脚本生成一个多语言功能的 Web 页面。首先我们用 Unicode 字符串定义一些消息。我们假设你的编辑器只能输入ASCII 编码。因此，非 ASCII 编码的字符使用\u转义符输入。实际上从文件或数据库中也能读取这些消息。

# Greeting in English, Spanish,  
# Chinese and Japanese. UNICODE\_HELLO = u""" Hello!  
\u00A1Hola!  
\u4F60\u597D!  
\u3053\u3093\u306B\u3061\u306F!  
"""

例 20.7 简单 Unicode CGI 示例（unicode\_cgi.py）。这个脚本输出到你 Web 浏览器端的是 Unicode 字符串。

#!/usr/bin/env python

CODEC = 'UTF-8'

UNICODE\_HELLO = u'''

Hello!

\u00A1Hola!

\u4F60\u597D!

\u3053\u3093\u306B\u3061\u306F!

'''

print 'Content-Type: text/html; charset=%s\r' % CODEC

print '\r'

print '<HTML><HEAD><TITLE>Unicode CGI Demo</TITLE></HEAD>'

print '<BODY>'

print UNICODE\_HELLO.encode(CODEC)

print '</BODY></HTML>'

CGI 产生的第一个头信息指出内容类型（content-type）是 HTTP。此处还声明了消息是以 UTF-8 编码进行传输的，这点很重要，这样浏览器才可以正确的翻译它。

print 'Content-type: text/html; charset=UTF-8\r'  
print '\r'

然后输出真正的消息。事先用 string 类的 encode()方法先将这个字符串转换成 UTF-8 序列。

如果你在你的浏览器中运行这个 CGI，你将会获得如图 20-13 所示的输出。

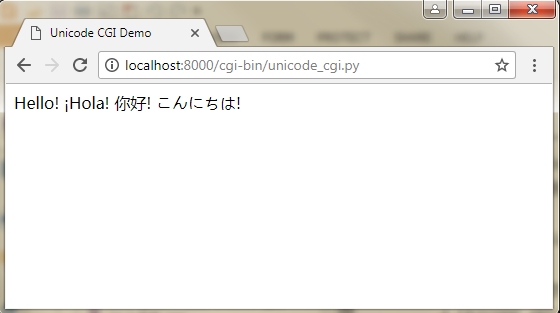


图 20-13 简单的 CGI Unicode 编码在Chrome上的输出（unicode\_cgi.py）

## 20.7 高级

CGI现在我们来看看 CGI编程的高级方面。这包括 cookie 的使用（保存在客户端的缓存数据），同一个CGI字段的多重值，和用multipart表单实现的文件上传。为了节省空间，我们将会在同一个程序中向你展示这三个特性。首先让我们看下多次提交问题。

20.7.1 Mulitipart 表单提交和文件的上传

目前，CGI特别指出只允许两种表单编码，“application/x-www-form-urlencoded”和 “multipart/form-dat”。 由于前者是默认的， 就没有必要像下边那样在 FORM 标签里声明编码方式。

<FORM enctype="application/x-www-form-urlencoded" ...>

但是对于 multipart 表单，你需要像这样明确给出编码：

<FORM enctype="multipart/form-data" ...>

在表单提交时你可以使用任一种编码，但在目前上传的文件仅能表现为multipart编码。 Multipart 编码是由网景在早期开发的，但是已经被微软（开始于 IE4 版）和其他的浏览器采用。

通过使用输入文件类型完成文件上传：

<INPUT type=file name=...>

这个指令表现为一个空的文本字段，同时旁边有个按钮，可以让你浏览文件目录系统，找到要上传的文件。在使用 multipart 编码时，你客户端提交到服务器端的表单看起来会很像带有附件的email。同时还需要有一个单独的编码，因为它还没有聪明到“通过URL编码”的程度，尤其是对一个二进制文件。这些信息仍然会到达服务器，只是以一种不同的“封装”形式而已。

不论你使用的是默认编码还是 multipart 编码，cgi 模块都会以同样的方式来处理它们，在表单提交时提供键和相应的值。你还可以像以前那样通过 FieldStorage 实例来访问数据。

20.7.2 多值字段

除了上传文件，我们将会展示如何处理具有多值的字段。最常见的情况就是你有一系列的复选框允许用户有多个选择。每个复选框都会标上相同的字段名，但是为了区分它们，会有不同的值与 特定的复选框关联。

正如你所知道的，在表单提交时，数据从用户端以键-值对形式发送到服务器端。当提交不止一 个复选框时，就会有多个值对应同一个键。在这种情况下，cgi 模块将会建立一个这类实例的列表， 你可以遍历获得所有的值，而不是为你的数据指定一个 MiniFielStorage 实例。总的来说不是很痛苦。

20.7.3 cookie

最后，我们会在例子中使用 cookie。如果你对 cookie 还不太熟悉的话，可以把它们看成是 Web 站点服务器要求保存在客户端（例如浏览器）上的二进制数据。

由于 HTTP 是一个“无状态信息”的协议，如你在本章最开始看到的截图一样，是通过 GET 请 求中的键值对来完成信息从一个页面到另一个页面的传递。实现这个功能的另外一种方法如我们以 前看到的一样，是使用隐藏的表单字段，如在后期 friends.py 脚本中对 action 变量的处理。这些 信息必须被嵌入新生成的页面中并返回给客户端，所以这些变量和值由服务器来管理。

还有一种可以保持对多个页面浏览连续性的方法就是在客户端保存这些数据。这就是引进 cookie 的原因。服务器可以向客户端发送一个请求来保存 cookie，而不必用在返回的 Web 页面中嵌入数据的方法来保持数据。Cookie 连接到最初的服务器的主域上（这样一个服务器就不能设置或者覆盖其他服务器上的cookie）,并且有一定的生存期限（因此你的浏览器不会堆满 cookie）。

这两个属性是通过有关数据条目的键-值对和cookie 联系在一起的。cookie 还有一些其他的属性，如域子路径，cookie 安全传输请求。

有了 coockies，我们不再需要为了跟踪用户而将数据从一页传到另一页了。虽然这在隐私问题上也引发了大量的争论，多数Web 站点还是合理地使用了cookie。为了准备代码，在客户端获得请求文件前，Web 服务器向客户端发送“SetCookie”头文件要求客户端存储 cookie

一旦在客户端建立了cookie，HTTP\_COOKIE 环境变量会将那些 cookie 自动放到请求中发送给服务器。cookie 是以分号分隔的键值对存在的。要访问这些数据，你的应用程序就要多次拆分这些字 符串（也就是说，使用 str.split()或者手动解析）。cookie 以分号（； ）分隔，每个键-值对中间都 由等号（=）分开。

和 multipart 编码一样，cookie 同样起源于网景，他们实现了 cookie 并制定出第一个规范并沿用至今，在下边的 Web 站点中你可以接触这些文档:

http://www.netscape.com/newsref/std/cookie\_spec.html

一旦 cookie 标准化以后，这些文档最终都被废除了，你可以从评论请求文档（RFCs）中获得更 多现在的信息。现今发布的最新的 cookie 的文件是 RFC2109。

20.7.4 使用高级 CGI

现在我们来展示CGI应用程序，adv\_cgi.py，它的代码功能和本章前部分讲到的friends3.py的差别不是很大。默认的第一页是用户填写的表单，它由四个主要部分组成：用户设置cookie字符串，姓名字段，编程语言复选框列表，文件提交框。在图20-14中可以看到示图。

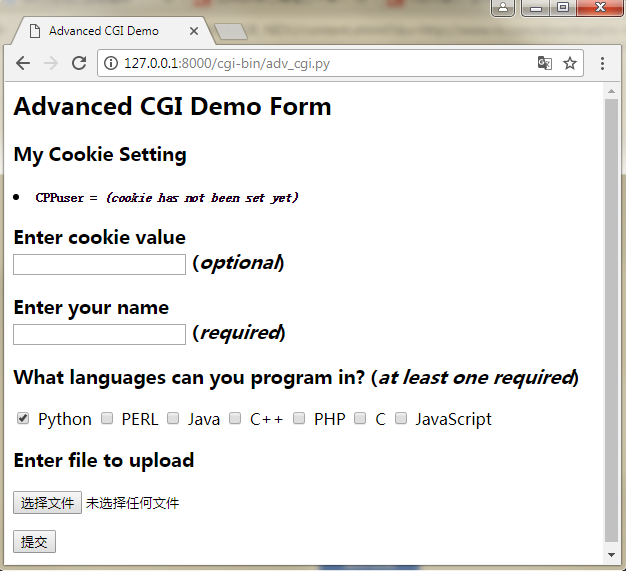


图20-14

注意查找文件的按钮在不同的浏览器中显示的文字是不同的，如，“Browse...”, “Choose”, “...”等。

这些数据以 mutipart编码提交到服务器端，在服务器端以同样的方式用 FieldStorage实例获取。唯一不同的就是对上传文件的检索。在我们的应用程序中，我们选择的是逐行读取，遍历文件。如果你不介意文件的大小的话，也可以一次读入整个文件。

由于这是服务器端第一次接到数据，这时，当我们向客户端返回结果页面时 ，我们使用 “SetCookie:”头文件来捕获浏览器端的cookie。

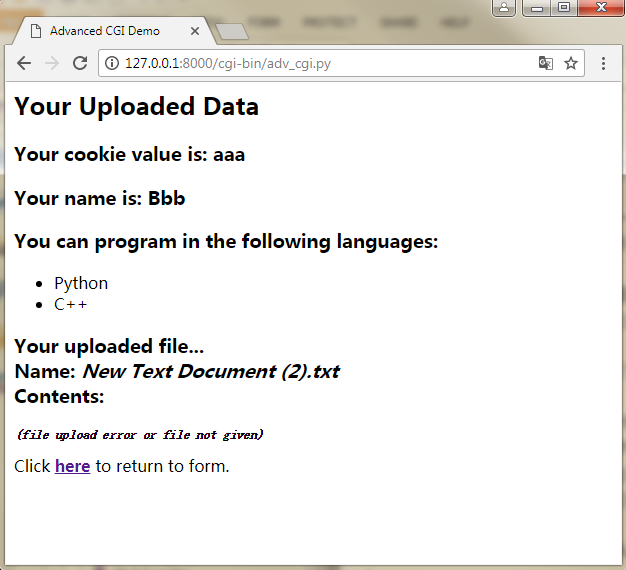
****

图20-15

在图20-15中，你可以看到数据提交后的结果展示。用户输入的所有数据都可以在页面中显示出来。在最后对话框中指定的文件也被上传到了服务器端，并显示出来。 你也会注意到在结果页面下方的那个链接，它使用相同的CGI脚本，可以帮我们返回表单页。如果我们单击下方的那个链接，没有任何表单数据提交给我们的脚本，因此会显示一个表单页面。然而，如你在图 20-15中看到的一样，所有的东西都可以显示出来，并非是一个空的表单！我们前边输入的信息都被显示出来了！在没有表单数据的情况下我们是怎样做到这一点的呢（将其隐藏或者作为 URL 中的请求参数）？实际上秘密是这些数据都被保存在客户端的 cookie 中了。

用户的cookie将用户输入表单中的值都保存了起来，用户名，使用的语言，上传文件的信息都会存储在cookie中。当脚本检测到表单没有数据时，它会返回一个表单页面，但是在表单页面建立前，它们从客户端的cookie中抓取了数据（当用户在单击了那个链接的时候将会自动传入）并且相应的将其填入表单中。因此当表单最终显示出来时，先前的输入便会魔术般的显示在用户面前。

我们相信你现在已经迫不及待的想看下这个程序了，详见例子 20.8.

adv\_cgi.py和我们本章前部分提到的CGI脚本friends3.py相当的像。它有表单页、结果页、 错误页可以返回。新的脚本中除了有所有的高级 CGI 特性外，我们还在脚本中增加了更多的面向对象特征：用类和方法代替了一系列的函数。我们页面的HTML文本对我们的类来说都是静态的了，这就意味着它们在实例中都是以常量出现的—虽然我们这里仅有一个实例。

Example 20.8 Advanced CGI Application (advcgi.py) 这个脚本有一个处理所有事情的主函数，AdvCGI， 它有方法显示表单、错误或结果页面，同时 也可以从客户端（Web 浏览器）读写 cookie。

#!/usr/bin/env python

# -\*- coding: UTF-8 -\*-

from cgi import FieldStorage

from os import environ

from cStringIO import StringIO

from urllib import quote, unquote

from string import capwords, strip, split, join

class AdvCGI(object):

"""docstring for AdvCGI"""

header = 'Content-Type: text/html\n\n'

url = '/cgi-bin/adv\_cgi.py'

formhtml = '''<HTML><HEAD><TITLE>

Advanced CGI Demo</TITLE></HEAD>

<BODY><H2>Advanced CGI Demo Form</H2>

<FORM METHOD=post ACTION="%s" ENCTYPE="multipart/form-data">

<H3>My Cookie Setting</H3>

<LI> <CODE><B>CPPuser = %s</B></CODE>

<H3>Enter cookie value<BR>

<INPUT NAME=cookie value="%s"> (<I>optional</I>)</H3>

<H3>Enter your name<BR>

<INPUT NAME=person VALUE="%s"> (<I>required</I>)</H3>

<H3>What languages can you program in?

(<I>at least one required</I>)</H3>

%s

<H3>Enter file to upload</H3>

<INPUT TYPE=file NAME=upfile VALUE="%s" SIZE=45>

<P><INPUT TYPE=submit>

</FORM></BODY></HTML>'''

langSet = ('Python', 'PERL', 'Java', 'C++', 'PHP','C', 'JavaScript')

langItem = '<INPUT TYPE=checkbox NAME=lang VALUE="%s"%s> %s\n'

def getCPPCookies(self): # read cookies from client

if environ.has\_key('HTTP\_COOKIE'):

for eachCookie in map(strip, split(environ['HTTP\_COOKIE'], ';')):

if len(eachCookie) > 6 and eachCookie[:3] == 'CPP':

tag = eachCookie[3:7]

try:

self.cookies[tag] = eval(unquote(eachCookie[8:]))

except(NameError, SyntaxError):

self.cookies[tag] = unquote(eachCookie[8:])

else:

self.cookies['info'] = self.cookies['user'] = ''

if self.cookies['info'] != '':

self.who, langStr, self.fn = split(self.cookies['info'], ':')

self.langs = split(langStr, ',')

else:

self.who = self.fn = ' '

self.langs = ['Python']

def showForm(self): # show fill-out form

self.getCPPCookies()

langStr = ''

print self.langs

for eachLang in AdvCGI.langSet:

if eachLang in self.langs:

langStr += AdvCGI.langItem % (eachLang, ' CHECKED', eachLang)

else:

langStr += AdvCGI.langItem % (eachLang, '', eachLang)

if not self.cookies.has\_key('user') or \

self.cookies['user'] == '':

cookStatus = '<I>(cookie has not been set yet)</I>'

userCook = ''

else:

userCook = cookStatus = self.cookies['user']

print AdvCGI.header + AdvCGI.formhtml % (AdvCGI.url,cookStatus, userCook, self.who, langStr, self.fn)

errhtml = '''<HTML><HEAD><TITLE>

Advanced CGI Demo</TITLE></HEAD>

<BODY><H3>ERROR</H3>

<B>%s</B><P>

<FORM><INPUT TYPE=button VALUE=Back

ONCLICK="window.history.back()"></FORM>

</BODY></HTML>'''

def showError(self):

print AdvCGI.header + AdvCGI.errhtml % (self.error)

reshtml = '''<HTML><HEAD><TITLE>

Advanced CGI Demo</TITLE></HEAD>

<BODY><H2>Your Uploaded Data</H2>

<H3>Your cookie value is: <B>%s</B></H3>

<H3>Your name is: <B>%s</B></H3>

<H3>You can program in the following languages:</H3>

<UL>%s</UL>

<H3>Your uploaded file...<BR>

Name: <I>%s</I><BR>

Contents:</H3>

<PRE>%s</PRE>

Click <A HREF="%s"><B>here</B></A> to return to form.

</BODY></HTML>'''

def setCPPCookies(self):# tell client to store cookies

for eachCookie in self.cookies.keys():

print 'Set-Cookie: CPP%s=%s; path=/' % (eachCookie, quote(self.cookies[eachCookie]))

def doResults(self):

'''display results page'''

MAXBYTES = 1024

langlist = ''

for eachLang in self.langs:

langlist = langlist + '<LI>%s<BR>' % eachLang

filedata = ''

while len(filedata) < MAXBYTES:# read file chunks

data = self.fp.readline()

if data == '':

break

filedata += data

else: # truncate if too long

filedata += '... <B><I>(file truncated due to size)</I></B>'

self.fp.close()

if filedata == '':

filedata = '<B><I>(file upload error or file not given)</I></B>'

filename = self.fn

if not self.cookies.has\_key('user') or self.cookies['user'] == '':

cookStatus = '<I>(cookie has not been set yet)</I>'

userCook = ''

else:

userCook = cookStatus = self.cookies['user']

self.cookies['info'] = join([self.who, join(self.langs, ','), filename], ':')

self.setCPPCookies()

print AdvCGI.header + AdvCGI.reshtml % (cookStatus, self.who, langlist, filename, filedata, AdvCGI.url)

def go(self): # determine which page to return

self.cookies = {}

self.error = ''

form = FieldStorage()

if form.keys() == []:

self.showForm()

return

if form.has\_key('person'):

self.who = capwords(strip(form['person'].value))

if self.who == '':

self.error = 'Your name is required. (blank)'

else:

self.error = 'Your name is required. (missing)'

if form.has\_key('cookie'):

self.cookies['user'] = unquote(strip(form['cookie'].value))

else:

self.cookies['user'] = ''

self.langs = []

if form.has\_key('lang'):

langdata = form['lang']

if type(langdata) == type([]):

for eachLang in langdata:

self.langs.append(eachLang.value)

else:

self.langs.append(langdata.value)

else:

self.error = 'At least one language required.'

if form.has\_key('upfile'):

upfile = form["upfile"]

self.fn = upfile.filename or ''

if upfile.file:

self.fp = upfile.file

else:

self.fp = StringIO('(no data)')

else:

self.fp = StringIO('(no file)')

self.fn = ''

if not self.error:

self.doResults()

else:

self.showError()

if \_\_name\_\_ == '\_\_main\_\_':

page = AdvCGI()

page.go()

逐行解释（以块划分）

**1-8**行

普通的起始、和模块导入行出现在这里。唯一你可能不太熟悉的模块是 cStringIO，我们曾在第10章简单讲解过它并在例20.1中用过。cStringIO.StingIO()会在字符串上创建一个类似文件的对象，所以访问这个字符串与打开一个文件并使用文件句柄去访问数据很相似。

**9-13** 行 在声明 AdvCGI 类之后，header 和 url（静态）变量被创建出来，在显示所有不同页面的方法中会用到这些变量。

**14-80** 行 所有这个块中的代码都是用来创建、显示表单页面的。那些数据属性都是不言自明的。getCPPcookie()取得Web客户端发来的 cookie 信息，而 showForm()校对所有这些信息并把表单页面返回给客户端。

73-82 行

这个代码块负责错误页面。

83-144 行

结果页面的生成使用了本块代码。setCPPcookie()方法要求客户端为我们的应用程序存储 cookie，而 doResults()方法聚集所有数据并把输出发回客户端。

20.8 Web（HTTP）服务器0

20.8 Web（HTTP）服务器

20.8.1 用 Python 建立 Web 服务器

20．9 相关模块0

20.10 练习