



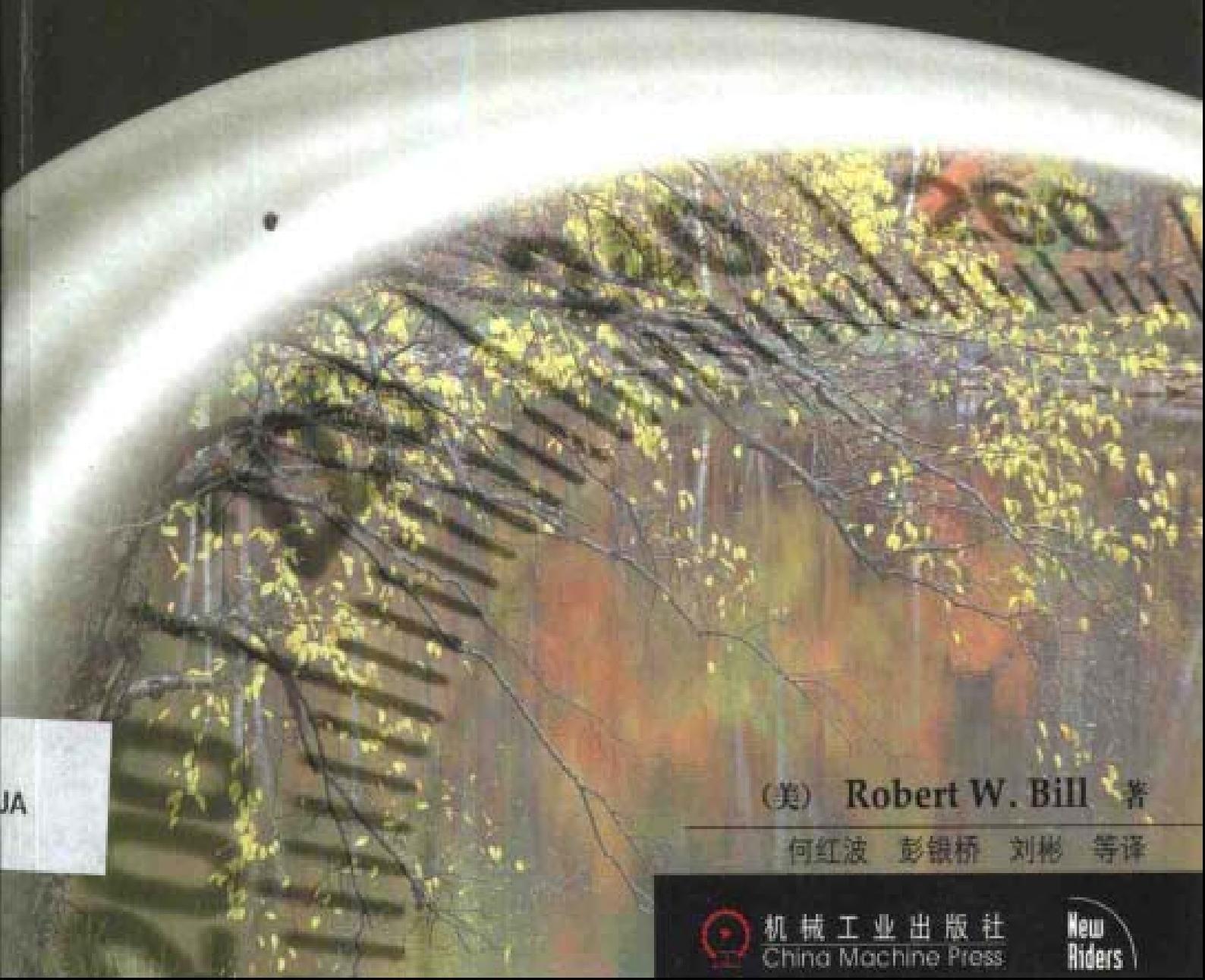


Jython for Java  
Programmers



开发人员专业技术丛书

# Jython 程序设计



(美) Robert W. Bill 著

何红波 彭银桥 刘彬 等译



机械工业出版社  
China Machine Press

New  
Riders

# Jython 程序设计

Java和Python是当前最为流行的编程语言。Java的平台无关性特点是其他语言无法比拟的。它还具有功能强大的集成开发环境，可以快速创建各种Java应用程序；而Python则具有很好的灵活性、开发快速、容易使用。Jython是这两种语言的组合，可以预见这种语言有着广阔前景。本书有助于Java开发者提高开发与部署应用程序的能力，详尽阐述了Java与Jython的异同以及Java与Jython进行组合的强大威力。书中实例丰富，通过大量的实例讲述了Jython的使用方法。本书适合于有一定经验的Java应用开发者。<http://www.newriders.com> 网站提供了本书的全部程序源码下载。

## 本书主要内容涉及：

- 从Python继承过来的Jython的完整语法
- 用Jython建立的函数与函数编程工具
- Jython的整个面向对象编程模型以及它如何实现Java
- 关于编译Jython为Java字节编码的技巧以及Jython的可选静态编译
- 将Jython嵌入Java应用程序的建议
- 用Java扩展Jython
- Jython的自动beam属性
- 用AWT、Swing、Jython进行GUI开发
- 数据库与Jython
- Jython Web开发

- Java 2 高级编程
- Java 2 类库（增补版）
- Java高效编程指南
- Java技术精粹
- Java语言导学（原著第3版）
- 软件工程——Java语言描述
- Java 2 平台安全技术——结构、API设计和实现
- Java 2 认证考试指南（原书第3版）
- Java 程序设计教程（原书第3版）（上册）
- Java程序设计习题与解答（英文版）

适用水平：中、高级

ISBN 7-111-10342-4



9 787111 103424



新华书店



网上购书：[www.china-pub.com](http://www.china-pub.com)

北京市西城区百万庄南街1号 100037

购书热线：(010)68995259, 8006100280 (北京地区)

ISBN 7-111-10342-4/TP · 2451

定价：38.00 元

开发人员专业技术丛书

# Jython 程序设计

(美) Robert W.Bill 著

何红波 彭银桥 刘彬 等译



A1008558



机械工业出版社  
China Machine Press

Jython 语言结合了 Python 的灵活高效与 Java 的强大。本书全面介绍了这门语言，有助于 Java 开发者提高开发与部署应用程序的能力。主要内容包括：Jython 介绍及其详细语法，用 Java 类扩展 Jython，用 Jython 编写各种应用程序等等。本书详尽地阐述了 Java 与 Jython 的异同以及 Java 与 Jython 进行组合的强大威力，书中实例丰富，通过大量的小实例讲述了 Jython 的使用方法。

本书编排独特、讲解透彻，适合于有一定经验的 Java 应用开发者使用。

Robert W. Bill: Jython for Java Programmers.

Authorized translation from the English language edition published by New Riders Publishing, an imprint of Macmillan Computer Publishing U.S.A.

Copyright © 2002 by New Riders Publishing.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2002 by China Machine Press.

本书中文简体字版由美国麦克米兰公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

**本书版权登记号：图字：01-2002-2193**

### **图书在版编目（CIP）数据**

Jython 程序设计 / (美) 比尔 (Bill, R.W.) 著；何红波等译 . - 北京：机械工业出版社，  
2002.6

（开发人员专业技术丛书）

书名原文：Jython for Java Programmers

ISBN 7-111-10342-4

I . J… II . ①比…②何… III . JAVA 语言-程序设计 IV . TP312

中国版本图书馆 CIP 数据核字（2002）第 038859 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：高薇 张鸿斌

北京第二外国语学院印刷厂印刷·新华书店北京发行所发行

2002 年 6 月第 1 版第 1 次印刷

787mm × 1092mm 1/16 · 21.25 印张

印数 0 001 - 4 000 册

定价：38.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

## 译者序

在科学技术高速发展的今天，计算机是重要的辅助和开发工具，而计算机语言又是所有应用程序的万本之源，介绍和引入当今国际流行且高效的计算机语言是计算机学术领域亟待开展的工作，而简明准确的语言将为计算机程序开发者带来事半功倍的效果，这就是我们把本书介绍给读者的重要原因。

Java 和 Python 是当前最为流行的编程语言。Java 的平台无关性特点是其他语言无法比拟的，它还具有功能强大的集成开发环境，可以快速创建各种 Java 应用程序；而 Python 则具有很好的灵活性，开发快速，容易使用。Jython 是这两种语言的组合，可以预见这种语言将有广阔的应用前景。

本书主要由三个大部分组成。第一部分包括前 7 章，详尽阐述了 Jython 语言的组成、命令、规则及具体运用；第二部分包括第 8、9 章，主要介绍了 Jython 的编译和如何在 Java 中嵌入和扩展 Jython；第三部分包括第 10、11、12 章，主要介绍了 Jython 在图形开发、数据库编程及在服务器端 Web 编程方面的应用。

本书是计算机语言高级应用书籍，最适合已熟悉 Java 并对 Jython 有兴趣的人，对于希望增加一门高级语言作为工具的 Java 程序员更是一本难得的好书，而对于已开始用 Python 编程并希望将其扩展到 Java 框架中的程序员也将从中得益，对于那些编程的新手，则至少可以掌握 Java 的基础知识。

本书由中南大学何红波组织翻译，刘彬负责第 1、3、4、5 章的翻译，林立新负责第 9 章翻译，何红波、彭银桥翻译了其他章节。参与翻译工作的还有英宇、谢君英、高健、王翌、徐继伟等人，最后由彭银桥进行全书校对工作。

由于翻译时间紧迫，加之译者水平有限，如有错误与疏漏之处，请读者原谅并提出宝贵意见。我们接收读者意见的邮箱为：yingyu@263.net。

彭银桥  
2002 年 5 月

## 前　　言

Jython 是两种广泛流行语言 Java 和 Python 的组合。Java 已被广泛地接受，这可从很多公司都已配置了基于 Java 的应用得以见证。另外 Java 有庞大的可利用的类库，并有很详尽的文档。Python 则具有很好的灵活性，开发快速，容易使用。在两种语言之间做出选择是很难的事，但有了 Jython，就不再需要做这种选择了。Jython 能够实现在 Java 或 Python 中实现的任何类、算法和模式而不需考虑其他语言，它能在这两种语言之间保持几乎无缝的操作。

无缝一词在本书中十分重要而且不断重现。扩展其他语言像 Perl 或 Python，需要专用的 API 或繁冗的封装器。如果不加改造，任何 C 代码都不可能在 Perl 或 Python 中使用，而任何的 Java 代码却能在 Jython 中运行。由于与 Java 的无缝集成，你可以在 Jython 中引入、使用任何 Java 类并创建派生类。不仅包括那些为某个特定 API 而写的类或与某个特定工具打包的类，而且包括任何 Java 类。另外你还能将 Jython 编译成 Java 的字节代码使其在 Java 的框架内运行。你甚至可以在 Java 中引入、使用任何 Python 类并创建派生类。

在 Jython 和 Java 中也有一些小的差别，正确理解这些差别对理解 Jython 是很有帮助的。Java 是一种类型丰富的语言，而 Jython 则使用动态类型而没有明显的类型定义。Java 有一些包含类的包，而 Jython 有包、模块、类和函数。Java 必须编译，而 Jython 能交互式地运行，解释一个非编译的脚本或编译成字节码。Java 类有像 private 和 protected 的存取符，而 Jython 仅有最小的存取限制而没有明显的像 private 的修饰符。

最有趣的事情是 Jython 和 Java 的差别并不构成很大的麻烦，相反却能互补。Jython 的交互模式提供了一个测试和研究 Java 类的快速方法，而 Java 的接口和抽象类给 Jython 派生类提供了一个指定协议的很好的方法。Jython 的动态类型对快速原型和灵活性有很大的帮助，而 Java 的静态类型也增加了运行时的效率和类型的安全性。这些互补由于其无缝性而显得很好。增加程序的花费来平衡这些特性将对程序的功能有副作用。幸运的是 Jython 使它们简单、易于理解，并且不需要额外的花费。

### 什么是 Jython

要了解 Jython，首先要了解 Python。Python 是用 C 编写的高级的、面向对象的、开放源代码的编程语言。Guido van Rossum 是 Python 的原创者，继而在 Python 的快速发展中产生了一大群高水平的设计者和程序员。使用 Python 的开发人员增长迅速，并一直在持续增长。然而 Sun 的 Java 编程语言也是深入人心的。随着用 Java 实现的项目的数量接近了用 C/C++ 实现的项目，Python 的 Java 实现也变得很有必要。Jython，最初叫做 JPython，就是：Python 语言的 Java 实现。为避免混淆，本书用 CPython 来表示 Python 的 C 语言实现，而用 Jython 来表示 Java 实现，而 Python 表示实现的中性概念和 Python 语言规范的设计特征。

Jython 是一种完整的语言，而不是一个 Java 翻译器或仅仅是一个 Python 编译器，它是一个

Python 语言在 Java 中的完全实现。Jython 也有很多从 CPython 中继承的模块库。最有趣的事情是 Jython 不像 CPython 或其他任何高级语言，它提供了对其实现语言的一切存取。所以 Jython 不仅给你提供了 Python 的库，同时也提供了所有的 Java 类。这使其有一个巨大的资源库。

Jython 和 Python 强调了代码的简明性、方便性和易读性。Jython 使用缩排来对代码块定界以避免使用在 Java 中的大括号。Jython 用新的 - 行来表示一个新的语句的开始，并有几个重要的区别，如允许在每个语句后省略分号。Jython 没有像在 Java 中的 public、private 和 protected 存取符，这样就给程序员提供了快速开发所需要的灵活性，并将注意力集中在程序逻辑上。正像前面所提到的，Jython 不用明显的静态的类型定义，故程序员不需要从程序逻辑转移到类型定义上来。

Jython 的历史要追溯到 Jim Hugunin，他是 Guido van Rossum 在国家研究动力中心 (CNRI) 的同事。Jim Hugunin 认识到 Python 编程语言用 Java 实现的重要性，并实现了最初名为 JPython 的语言。由于要开发 aspectj (<http://aspectj.org/>)，Jim Hugunin 不能继续致力于 JPython 了。所以当 Python 的开发者准备离开 CNRI 时，由当时也在 CNRI 的 Barry Warsaw 继续领导开发。

Python 和 Jython 项目组从 CNRI 离开后，在 Sourceforge 上转变为一种更开放的语言模型。在此期间，一个对 Jython (JPython) 做了主要贡献的人 Finn Bock 领导了 Jython 项目小组。正是由于 Finn Bock 所做的杰出贡献使 Jython 现在成为一个如此有价值的工具。类似 Jython 这样的开放源代码项目与开发和维护它们的人一样杰出，从这个意义上说 Jython 因为有 Finn Bock 的贡献和指导而很幸运。另外一个对 Jython 做了最新有价值贡献的人是 Samuele Pedroni。Samuele 的贡献主要在 Jython 的类装载、导入机制等等。Finn 和 Samuele 目前是 Jython 的两个主要开发者。

## 为什么需要 Jython

Jython 由于继承了 Java 和 Python 二者的特性而显得很独特。本节介绍一下这些特性及其意义。

### 对 Java 类的无缝存取

在 Java 中实现 Python 可以看到有趣的 Java 反射 API 的作用。反射使 Jython 能无缝地使用任何 Java 类。Jython 从 CPython 中继承了很多优点，但 CPython 不像别的专为 Python 所写的一样，在 C 和 Python 之间有一些问题限制了 C 库函数的使用。在 Jython 中真正解决了这个问题，使其编程的效率和生产力得到了很大的提高。

由于与 Java 的无缝集成，Jython 能使任何部署了 Java 应用和框架的公司受益而不需要额外的工作。接受任何一种部门的编程语言，对任何一个公司而言都是不容易的，需要深思熟虑，因为这牵涉到整体结构、服务器和外围的工具。Jython 作为 Java 的一个无缝集成的语言，可以在已存在的 Java 应用上无缝增加而不需要重大抉择。很多公司都花费了很多资金来建立 Java 的应用，这使采用 CPython、Perl、Ruby、PHP 和其他不能透明地集成已有 Java 实现的高级语言的效益降低，吸引力下降。而 Jython 有能力对已存在的 Java 框架进行补充，且二者能无缝地结合。

## 效率

计算编程语言的效率是一个很广泛的课题，它要考虑程序员的时间、总体复杂性、代码的行数、可用性、可维护性和运行效率。当然很多人不同意赋予这些变量的权重，经常是在不同的情况下偏重有所不同。然而本书的前提是 Jython 除运行时的效率外，其他方面都超过其他语言。Jython 的运行速度是可以与其他高级语言相比的，但速度并不是高级语言的目标和特点。区别在于当加速一个需要的应用时，将 Jython 代码翻译成 Java 更有效，这是由于 Jython 与 Java 的无缝集成性。另外对所有有效的 Java 类的直接访问增加了改进已存在的类的可能性。

## 动态类型

在 Jython 中你不必像在 Java 中那样声明类型，因为类型是在运行时决定的。Jython 的列表和映射类型是高级的多态的 Java 类的实例。多态意味着对象能对不同的数据类型工作。例如 Jython 的 `list` 类型可以是一个数字的序列、字符串的序列、字符的序列或它们的组合。动态和多态性的列是对编程的极大的贡献，从很多已放弃显式的静态类型定义的高级语言中可看出它减少了代码的行数，降低了复杂性，提高了程序的效率。

## 内省和动态执行

Jython 有一些允许方便的对象内省和代码的动态执行的内部函数。内省是发现一个对象信息的能力，而动态执行是执行在运行时产生的代码的能力。该功能很大程度上减少了代码的行数并增加了程序的可靠性，使其更加方便维护。这也使数据和程序结构或逻辑更好的集成而不影响重用性，因为所有的东西都是在运行时决定的。

## 第一类函数和函数编程

Jython 与 Python 一样有第一类函数。第一类函数是指能像变量一样的可调用的对象。第一类函数在对事件处理和其他情况下有意义，这导致增加了 Java 内部类的功能。虽然 Java 的内部类与第一类函数类似，但它在方便性与灵活性方面有很大的不足，这是由于 Jython 中的第一类函数减少了 Jython 中的语法开销。

Jython 也包括了所有函数编程所需要的工具。这意味着强制的面向对象的函数编程在 Jython 中得到支持。这显然在教学上很有意义，它使 Jython 程序员能选择最适合于特定问题的编程语言而不是由语言强加。函数化的工具如列表包含、`lambda` 表单、`map`、`filter` 和 `reduce` 也对减少代码的行数、降低复杂性和名字重绑定数（名字重绑定有很大副作用）起到很大的作用。

## 学习周期短

任何 Java 程序员在数日内就能熟悉 Jython。由于有很多内容，关键在于细节的学习，但仅用几天的时间就能拥有 Jython 的快速开发功能确实是很有价值的。对于那些从事测试和技术支持的小组通常并没有很多时间去学习复杂的 Java 代码，但通过对 Jython 的学习能在开销很少的情况下很快提高公司的技术水平和效率。

## 写一次，处处可用

由于 Jython 是用 Java 编写的且由于其可编译成 Java 字节码，因此 Jython 也具有 Java 的“写一次，处处可用”的特点。Jython 能运行在任何可兼容的 Java 1.1 或更高的 Java 虚拟机（JVM）版本的平台上。另外你可将 Jython 应用编译成自足的字节码，它能运行在任何兼容的 JVM 上。在 Linux 上编译的应用能运行在有兼容 JVM 的任何其他平台上。

## Java 安全性

Java 的安全性是特别的而且越来越重要。从沙箱到信号，Jython 有能力使用 Java 的特别的安全框架。

## 代码清晰性

代码的清晰性是 Python 最大的优点，当然也是 Jython 最大的优点。不必要的标点和行都避免了。Jython 代码在可读性和清晰性方面近似于自然语言。这起源于 Python 对代码块和语句的简单描绘的承诺。缩排标记代码块，换行符标记新的语句。在此之上，语法通常支持清晰性。由于 Jython 代码的清晰性，其维护变得更加容易。

## Unicode 和国际化

Jython 使用了 Java 的 Unicode 实现方法，使得它很容易实现产品的国际化。

## Hotspot 和开发效率

Jython 的速度还不错，但不像纯 Java 那样快。Jython 的主要特点是在开发和维护时间上。然而高级语言的开发中有很强的优先权，这在 Java 的 Hotspot 技术中尤为显著。Hotspot 优化程序中那些最需要优化的部分。它的理论基础是程序的一小部分花费程序的大部分执行时间。只有优化那一部分代码才能很大程度地改变程序的性能。开发代码时这样做效果是明显的。使用 Jython 来编写一个应用，然后将处理器相关的类转换成 Java 是程序的运行性能和程序效率之间理想的结合。Hotspot 类似的技术使采用 Jython 和 Java 的高级语言开发是令人信服的。

## 本书内容

本书是针对那些已经熟悉 Java 或正在学习 Java 的人而编排的。一些 Java 专用术语像 class-path、垃圾回收和接口将不再做解释，这就要求必须知道 Java 的基本知识。尤其是那些希望加快开发速度、嵌入解释器和增加灵活性的 Java 开发者将会对本书最感兴趣。虽然 Jython 是一门完整的编程语言，但由于其与 Java 是互补的而不是 Java 的替代物而显得很独特。

第 1 章“Jython 语法、语句和注释”介绍了语法和语句。Python 的语法是组成 *pythonic* 的很重要的组成部分。*pythonic* 是一个使 Python 和 Jython 代码清晰、简单、独特的那些细节的描述器。Python 用缩排来定界代码块，用换行符来定界语句对很多程序员来说是新颖的方法。第 1 章详细描述了缩排、换行符、语句的规则。另外该章还介绍了交互式解释器，它是一个

Jython的模式，能立即响应你所输入的语句。该章主要是讲一些 Python 通用的内容，并未提到 Jython 独特的内容。

第 2 章“运算符、类型和内置函数”包括 Jython 的数据对象、运算符和内置函数。数据对象或类型都是异常有趣的，因为 Jython 能使用 Python 和 Java 二者的对象。Python 的语言规范定义了 Python 当然也是 Jython 的类型，Jython 专有的类型需要详细说明。详细说明包括 Jython 和 Java 类型的转变。第 2 章也定义了 Jython 的内置函数。内置函数指那些不需要任何引入语句就可工作的那部分函数，这在 Jython 的函数中占有很大的比例。

第 3 章介绍 Jython 的错误、异常和警告，定义了 Jython 的内置异常以及使用 try/except 和 try/finally 语句来处理异常。异常对 Jython 是很重要的，但它也是运行 Java 类和其导致的异常的重要一步。

Java 没有函数，但函数在 Jython 中是很重要的一部分。第 4 章“用户定义的函数和变量的作用域”说明了如何定义和使用 Jython 函数以及如何使用 Jython 的函数编程工具。函数是在类中没有定义的可调用的对象，但函数编程没有一个简洁的定义。函数编程循环寻找表达式的结果。Jython 提供了所有的需要学习和使用函数编程的工具。这些工具已出现在第 2 章。

Jython 有模块，这是另外一个 Java 没有的东西。第 5 章“模块和包”描述了 Jython 的模块。Jython 也像 Java 一样有包，但如第 5 章所说明的，Jython 的包不同于 Java 的对应。在 Jython 的包、模块、类以及 Java 的包、类中，import 语句变得很重要。第 5 章详细说明了模块和包的 import 语句。·

第 6 章“类、实例和继承”介绍了 Jython 类的定义和使用。这包括 Java 类的派生类、Java 接口的使用和 Java 抽象类以及 Java 的访问修饰符如 public、private、protected 的含义。Jython 类与 Java 类有很大的不同，第 6 章详细说明了这一点。

第 7 章“高级类”扩展了 Jython 类信息使其包括 Jython 的特殊类属性。Jython 的特殊属性是那些遵循一定的命名规范并提供特定功能的属性。这些特殊的属性使自定义类行为和创建高级类更加容易。

第 8 章“用 jythonc 编译 Jython”详细介绍了 Jython 的复杂的 jythonc 工具。jythonc 将 Jython 码编译成 Java 字节码。jythonc 使你可采用在 Java 框架内用 Jython 写的类，甚至能使你创建你可在 Java 中引入和使用的类文件。

第 9 章“在 Java 中嵌入和扩展 Jython”描述了如何嵌入一个 Jython 的解释器。将 Jython 编译成 Java 字节码是很重要的，但在 Java 应用中扩展一个 Jython 解释器有很多优点。嵌入使你能有 Jython 系统状态所有的控制权，且能使你在 Java 应用中使用所有的 Jython 特征和模块。很吸引人的地方是嵌入一个 Jython 的解释器非常容易。一个基本的嵌入 Jython 解释器只需要两行 Java 代码，增加额外的配置信息也很容易。一个嵌入的解释器使你在不需要用 jythonc 编译模块的情况下写 Jython 模块来扩展或实现 Java 应用的特征。在我的印象中，Jython 的嵌入特征是其最大的优点。它使得高级语言的开发在充分吸收单个应用中 Java 和 Python 的好处之后能支持快速开发和具有可扩展性。已经有很多项目使用嵌入的 Jython，这个趋势将会继续。

第 10 章“GUI 开发”描述了如何用 Jython 开发图形应用。重点是 Java 的 Abstract Windowing Toolkit (AWT) 和 Swing 应用程序接口。Jython 由于其增加了自动 bean 属性和事件而使图形应

用的开发更加快捷。第 10 章详细介绍了自动 bean 属性和事件，但本质上是由于 Jython 能为这些特征自动搜索组件并为建立属性和事件的句柄建立快捷方式。该方式平衡了 Jython 的语法特征来简化和加速应用。第 10 章也包含了使用 Jython 来创建 Java applet 的信息。

Java 在数据库编程方面是很卓越的，Jython 能在增加其自身的优点时保持所有 Java 的成功之处。第 11 章“数据库编程”包含 Jython 数据库编程的使用。该章也包含一些哈希数据库文件的内容和像 MySQL、PostgreSQL 的关系数据库管理系统（PostgreSQL 为面向对象）。Jython 也能使用 Java 的 JDBC 数据库连接或使用 zxJDBC 实现的 Python 数据库应用程序员接口。第 11 章讨论了这些 API。

第 12 章“服务器端的 Web 编程”描述了使用 Jython 的 Web 编程。对 Jython 而言，这意味着 servlet、Java Server Pages (JSP) 和标记库 taglibs。换句话说，使用 Jython 的服务器端的 Web 编程与普通的（标准的）Java 服务器端的 Web 编程一样。第 12 章解释了使用 jythone 编译的类作为 servlet，也讨论了嵌入 Jython 的 servlet 映射以及 IBM 的组件脚本框架 (BSF)。

## 本书读者对象

本书最适合那些已熟悉 Java 并对 Jython 有兴趣的人，这个范围较大。再具体一些，是那些希望增加一门高级语言作为工具的 Java 程序员，是那些在使用 Java 的公司中的想找一个 Java 的替代物的程序员。另外一类读者将是那些已经开始 Python 编程而希望将其扩展到 Java 框架中的程序员。由于本书假设读者已掌握 Java 的基础知识，所以这类读者还须补上 Java 的基础知识。

第三种读者将是编程的新手，但不是所有新手都适合。本书不能成为一本独立地介绍编程的书。读者至少需要掌握 Java 的基础知识。由于 Jython (及 Python) 是一门理想的编程语言，所以这是可行的。Python 语言具有清晰性和简洁性，它的交互解释器允许实验、立即反馈，快速获取语言特征。Jython 既支持函数编程也支持面向对象编程，使其成为一种很好的教学语言。

## 本书不适合于哪些人

如果你对 Java 没有兴趣，则本书不是写给你的。学习 Jython 需要了解 Java，那些不懂 Java 或不愿意在学习本书的同时研究 Java 的人将不会从本书中获益。Jython 的优势在于其作为 Java 的一个变种能无缝地使用 Java 的各种库。这意味着如果你不想使用以 Java 为中心的工具，那么这本书对你作用不大。

如果你已经是一个 Python 的高级程序员，你也不必读此书。本书中大部分的内容都是 Python 的语法和使用。虽然有很多 Java 专有的例子和信息，但大部分关于语法和使用的章节会使熟练的 Python 程序员觉得是重复的。

## 其他资源

与 Jython 有关的 Internet 资源及其内容如下：

- <http://www.jython.org/>——Jython 的官方网站。

- <http://www.python.org/>——Python 的官方网站。在站点上有大量的 Python 的文档，其对 Jython 也十分有用。
- <http://www.digisprings.com/jython/>——本书相关的网站。在该网站上你可看到一些 Jython 额外的信息，包括本书的勘误表、提示等等。
- <http://sourceforge.net/projects/zxjdbc>——这是 zxJDBC 包的项目主页。该包为 Jython 提供 Python 2.0 数据库功能。

Jython 专用的邮件列表是很有价值的资源。最好从 Jython 相关的主页的链接上订阅这样的邮件列表。有用的邮件列表包括：jython-users、jython-dev 和 jython-announce。jython-users 主要包括一些 Jython 的常见问题和帮助，而 jython-dev 主要包括一些 Jython 自身开发的东西（而不是用 Jython 开发）。announce 是一个小规模的邮件列表，主要来使读者知道最新的版本的一些信息。如果你碰到一些问题在本书的网站：<http://www..newriders.com/> 或在 Jython 的站点上找不到答案，最好咨询邮件列表。

本书英文原版书名：Jython for Java Programmers

本书英文原版书号：ISBN 0-7357-1111-9

# 目 录

译者序

前言

## 第一部分 Jython 简介

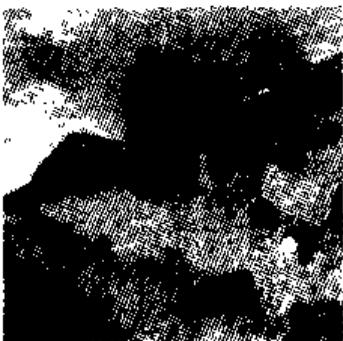
第 1 章 Jython 语法、语句和注释 .....	1
1.1 交互式控制台 .....	1
1.2 行分隔符和块缩进语法 .....	2
1.2.1 分号 .....	2
1.2.2 反斜线 .....	3
1.2.3 开放式分组 .....	3
1.2.4 三重引用 .....	4
1.2.5 代码程序块 .....	4
1.3 注释 .....	5
1.4 文档字符串 .....	6
1.5 语句 .....	6
1.5.1 简单语句 .....	6
1.5.2 复合语句 .....	11
1.6 比较 Jython 和 Java .....	13
第 2 章 运算符、类型和内置函数 .....	15
2.1 标志符 .....	15
2.2 Jython 数据对象 .....	15
2.2.1 数字对象 .....	17
2.2.2 序列对象 .....	18
2.2.3 映射对象 .....	26
2.2.4 PyNone .....	28
2.3 运算符 .....	29
2.3.1 算术运算符 .....	29
2.3.2 移位运算符 .....	29
2.3.3 比较运算符 .....	30
2.3.4 布尔运算符 .....	30
2.3.5 序列运算符 .....	31
2.4 内置功能 .....	32
2.4.1 内省函数 .....	32
2.4.2 数字函数 .....	35
2.4.3 类型转换函数 .....	35

2.4.4 内置文件函数 .....	36
2.4.5 序列函数 .....	36
2.4.6 动态语言函数 .....	37
2.4.7 属性工具 .....	37
2.4.8 函数工具 .....	40
2.5 杂类函数 .....	40
2.6 Jython 数据类型和 Java 数据类型的比较 .....	41
2.7 Java 类型 .....	41
第 3 章 错误和异常 .....	45
3.1 Jython 异常 .....	45
3.2 异常处理 .....	47
3.3 raise 语句 .....	50
3.4 traceback .....	54
3.5 assert 语句和_debug_ 变量 .....	55
3.6 警告框架 .....	55
3.7 比较 Jython 和 Java .....	57
第 4 章 用户定义的函数和变量的作用域 .....	59
4.1 定义函数 .....	59
4.1.1 缩进 .....	60
4.1.2 返回值 .....	61
4.1.3 文档字符串 .....	62
4.1.4 函数属性 .....	62
4.1.5 参数 .....	62
4.2 名空间 .....	64
4.2.1 两个静态范围 .....	64
4.2.2 静态嵌套的范围 .....	66
4.2.3 在用户定义函数中的特殊变量 .....	68
4.3 递归 .....	69
4.4 内置的函数编程工具 .....	69
4.5 同步 .....	76
第 5 章 模块和包 .....	79
5.1 import 语句 .....	79
5.2 Jython 和 Java 的比较 .....	81
5.3 Python 的 package.module 层次 .....	82

5.3.1 sys.path 变量	82	8.1 jythonc 是什么	159
5.3.2 什么是模块	83	8.2 用 jythonc 编译模块	160
5.3.3 特殊的模块变量	84	8.3 路径和经过编译的 Jython	162
5.3.4 什么是包	86	8.3.1 在 JVM 里设置 python.home 属性	162
5.4 Java 的 package.class 层次	88	8.3.2 显式地把目录加到模块里的 sys.path	164
5.5 重载	90	8.3.3 增加 python.path 或 python.prepath	
<b>第6章 类、实例和继承</b>	<b>93</b>	属性	164
6.1 封装、抽象和信息隐藏	93	8.3.4 冻结应用程序	165
6.2 定义 Jython 类	94	8.3.5 写一个定制的 __import__ () 钩子	165
6.3 Jython 类和实例属性	95	8.4 jythonc 选项	167
6.4 构造函数	98	8.5 与 Java 兼容的类	169
6.5 终结器和析构函数	100	8.5.1 一个与 Java 兼容的例子 Jython 类	169
6.6 继承	101	8.5.2 模块全局对象和与 Java 兼容类	173
6.6.1 派生 Jython 类	101	<b>第9章 在 Java 里嵌入和扩展 Jython</b>	<b>177</b>
6.6.2 派生 Java 接口	103	9.1 嵌入 Jython	177
6.6.3 派生 Java 类	104	9.2 嵌入 InteractiveInterpreter	204
6.7 方法重载	108	9.3 嵌入 InteractiveConsole	207
6.8 例子类	110	9.4 扩展 Jython	208
6.8.1 单元素	110	9.4.1 ClassDictInit	209
6.8.2 文件 grep 效用	112	9.4.2 __doc__ 字符串	210
6.8.3 HTTP 报文头	113	9.4.3 异常	210
6.8.4 树	115	9.4.4 参数	210
<b>第7章 高级类</b>	<b>121</b>	9.4.5 在 Java 里引入 Jython 模块	212
7.1 预先存在的类属性	121	9.4.6 使用 PyObject	212
7.2 预先存在的实例属性	124	9.4.7 用 Java 写 Jython 类	214
7.3 一般定制的特殊方法	125	9.4.8 增加 Java 类作为内置 Jython 模块	214
7.4 动态的属性访问	127		
7.5 可调用的钩子: __call__	131	<b>第三部分 用 Jython 编写应用程序</b>	
7.6 特殊的比较方法	132	<b>第10章 GUI 开发</b>	<b>217</b>
7.6.1 少比较方法	133	10.1 比较 Java 和 Jython 的 GUI	217
7.6.2 多比较方法	134	10.2 Bean 属性和事件	220
7.7 对象“真值”	137	10.2.1 Bean 属性	223
7.8 仿真内置数据对象	137	10.2.2 Bean 属性和元组	225
7.8.1 仿真序列	139	10.2.3 Bean 事件	225
7.8.2 仿真映射	153	10.2.4 名字优先权	226
7.8.3 仿真是数字类型	155	10.3 pawt 包	227
<b>第二部分 Jython 内核和用 Java 集成 Jython</b>		10.4 例子	231
<b>第8章 用 jythonc 编译 Jython</b>	<b>159</b>	10.4.1 简单的 AWT 图形	231
		10.4.2 增加事件	232
		10.4.3 图像	234

10.4.4 菜单和菜单事件 .....	235	12.3 GenericServlet 的更多内容 .....	281
10.4.5 拖放 .....	237	12.3.1 Init (ServletConfig) 方法 .....	284
10.4.6 Swing .....	238	12.3.2 service (ServletRequest, ServletResponse) 方法 .....	284
<b>第 11 章 数据库编程 .....</b>	<b>241</b>	12.3.3 destroy () 方法 .....	284
11.1 DBM 文件 .....	241	12.4 HttpServlet .....	284
11.2 序列化 .....	242	12.4.1 HttpServlet 方法 .....	285
11.2.1 marshal 模块 .....	242	12.4.2 HttpServlet 例子 .....	286
11.2.2 pickle 和 cPickle 模块 .....	243	12.4.3 HttpServletRequest 和 HttpServletResponse .....	288
11.2.3 Shelves .....	244		
11.2.4 PythonObjectInputStream .....	245		
11.3 数据库管理系统 .....	246	12.5 PyServlet .....	291
11.3.1 MySQL .....	246	12.5.1 安装 PyServlet .....	292
11.3.2 PostgreSQL .....	248	12.5.2 测试 PyServlet .....	294
11.4 JDBC .....	249	12.6 cookie .....	294
11.5 zxJDBC .....	266	12.7 Session .....	296
11.5.1 连接到数据库 .....	267	12.8 数据库和 Servlet .....	298
11.5.2 游标 .....	268	12.9 JSP .....	300
11.5.3 zxJDBC 和元数据 .....	270	12.9.1 jythonc 编译类和 JSP .....	300
11.5.4 预编译语句 .....	271	12.9.2 在 JSP 中嵌入 PythonInterpreter .....	302
11.5.5 错误和警告 .....	271	12.9.3 一个 Jython Taglib .....	303
<b>第 12 章 服务器端 Web 编程 .....</b>	<b>275</b>	12.9.4 BSF .....	306
12.1 Jython Servlet Container .....	275		
12.2 定义简单的 Servlet 类 .....	276		
12.2.1 一个简单的 Java Servlet .....	276	<b>附录</b>	
12.2.2 一个简单的 Jython Servlet .....	276		
12.2.3 测试 Java 和 Jython Servlet .....	277	<b>附录 A Jython 语句和内置函数快速参考 .....</b>	<b>309</b>

# 第一部分 Jython 简介



## 第 1 章 Jython 语法、语句和注释

本章描述 Jython 语法和语句。Jython 有一套简单明了的语法，能编写出易读的代码，常被认为与 Java、C 和 Perl 等有复杂形式的计算机语言不同。在讲授语法之前，本章介绍 Jython 的交互模式，这种模式可以像使用 shell 控制台一样，每次输入一行代码，并且有助于深入理解 Jython 语法。

### 本章未包括的内容

本章假定你有一个正在运行的 Jython 安装程序。如果没有，那么 New Riders 网站 (<http://www.newriders.com>) 有一个完整的安装程序可下载。另外，“什么是 Jython?”和赞扬 Jython 的内容就不在这里提及了。在开篇介绍和整本书上都将讲述 Jython 的内容和与此有关的问题。

### 1.1 交互式控制台

编程过程中经常遇到一连串的小难题，这些小难题需要在它们被加入到庞大的整体程序之前，被破解、测试并求得验证。那就是交互式控制台的作用——可以交互地开发代码、迅速地测试程序段，并且反复不停地试验直到小难题被解决。交互式控制台也称为交互式翻译机，类似于 shell 程序编译过程，你能交互地输入命令直到它们运行，然后将它们放在一个文件中做一个脚本程序。你逐渐会认识到 Jython 本身有着简单明了、通俗易懂的语法，并且是一种具有完整且丰富的面向对象的程序语言。

Jython 可以在没有任何参数或是添加了 -i 命令行开关的情况下开始。在开始运行 Jython 以后，你可以看见如下内容：

```
Jython 2.0 on java1.2.2 (JIT: symcjit)
Type "copyright", "credits" or "license" for more information.
>>>_
```

开始 Jython 后出现的第一行将包括 Jython 和 Java 的版本信息，也包括你正在使用的 JIT 的信息。如果你设置 JAVA\_COMPILER 环境变量为 NONE，Jython 将在 JIT 里有如下变化：

```
Jython 2.0 on java1.2.2 (JIT: NONE)
Type "copyright", "credits" or "license" for more information.
>>>
```

在第三行中的 > > > 是交互式解释器的提示符。在提示符下，你可以键入 Jython 代码，当每行命令输入完成后，代码会自动换行。这里又会出现一个提示符，是三个句点…，当有多行语句时会出现这种情况。

这种在交互模式中的测试、试验和探索不仅在 Jython 中使用，而且也是测试 Java 代码的一种有力工具。对于这本书来说，交互模式是 Java 的重点。这本书第一部分设计的例子很容易在交互模式中用于试验、检查以及探索。在本书中，提示符标记 >>> 和 ... 都被用来表明交互模式。

## 1.2 行分隔符和块缩进语法

简单的 Jython 语句包含在逻辑行中。你可能问：逻辑行是什么？Java 和许多其他语言使用标点，以便代码更清楚。分号是在 Java 中的行分隔符。简单的 Java 打印语句如下所示：

```
System.out.println("Hello world");
```

注意分号。而更简单的 Jython 打印语句如下：

```
print "Hello world"
```

其中没有分号标记逻辑行的结束。分号是用在许多语言中作为行终止的一个标志，但是在 Jython 中，一个换行符足够表示逻辑行之间的边界，但是也有一些例外。如果 Jython print 语句被输入交互解释器，按回车键将使解释器执行逻辑行的内容。这对于一个 Jython 文件也一样。在一个文件中，独立的各行 Jython 代码作为独立的逻辑行被解释，除非它们属于列在下面的例外之一。

在使用下面的方法时，简单语句的逻辑行与物理行是不一样的：

- 分号。
- 反斜线。
- 开放式分组。
- 三重引号。

### 当前交互限制

一项有关交互解释器的否认声明：尽管语法是合法的，当在交互模式下使用反斜线、开放式封装和三重引用时，Jython 的当前实现将产生一个错误，这个语法在 Jython 文件中所起作用就像在 CPython 交互式解释器中一样（在 C 中执行的 Jython 对应物）。等你读到这里，它也能在交互模式中工作，但如果不能，这些带有扩展逻辑行例子应该被放在一个文件中并作为一个相对子 Jython 解释器的自变量运行（例如 Jython example.py）。

#### 1.2.1 分号

第一个例子是分号。这是惟一一个与缩短逻辑行有关的例子。在同一个物理行中，加入两

条或更多的物理行，用分号将语句分开。即使在同一命令行中没有多重的语句，分号也不会影响任何东西。因此，如果 Java 编程习惯在语句的结尾加上分号，是不会有什么害处的。在选择这种语法以确保没有为紧密性而牺牲可读性的时候，记得注意代码的清晰无误。

程序清单 1-1 在一个物理的行上有多重打印语句。在两个打印语句之间，分号作为逻辑行的分隔符。然而，第二条语句没有一个分号，因为在它以后的换行符起着分隔符的作用。

#### 程序清单 1-1 分号的使用

```
>>># it's tradition, what can I say
>>>print "Hello "; print "world"
Hello
world
```

自动通知打印语句插入一个换行符。这是 Jython 打印语句的行为部分。如果这没被要求，你需添加一个逗号在打印语句后，为第二语句预留一些空间。

```
>>>print "hello ",; print "world"
hello world
```

#### 1.2.2 反斜线

反斜线 (\) 特性是一个行延续性的字符。多重的行能用反斜线 \ 形成一个逻辑行。将一个变量赋值，就是一个简单的 Jython 语句，但是如果你需要增加很多的值以至超出了物理行，那该怎么办？程序清单 1-2 显示了如何使用 \ 插入多重的行来解决这个问题。程序清单 1-2 不管语法的正确与否，都不会进行交互地工作。测试程序清单 1-2，把代码放在一个文件（在例子中使用了 backslash.py）并且在命令行中与 python backslash.py 一起运行。

#### 程序清单 1-2 用反斜线进行行扩展

```
# file: backslash.py
var = 10 + 9 + 8 + 7 + 6 +
5 + 4 + 3 + 2 + 1
print var
```

在命令行运行 python backslash.py 的结果是：

55

#### 1.2.3 开放式分组

开放式分组——意味着成对的方括号 []、花括号 {} 和括号 {} ——在一命令行中有它们开放的（左边的）字符，和在随后一行中表示完成的另外一边字符。逻辑行将会延伸至表示结束这些分组的另一半出现。程序清单 1-3 使用开放式分组在多重物理行上描写一条 print 语句。尽管是合法的语法，程序清单 1-3 也不会进行交互式工作（版本 2.1a1）。

#### 程序清单 1-3 使用括号引出多重命令行

```
# file: parens.py
```

---

```
print ("this, " +
      "that, " +
      "the other thing.")
```

---

运行 `python parens.py` 的结果是：

```
this, that, the other thing
```

在程序清单 1-3 第二、三行上的缩进是为了阅读方便——这与逻辑行的继续延长物无关。但在 Jython 的缩进例子中，用 \ 或开放式封装相联系的命令行，与随后行中的缩进没有关系。这是第一行缩进所做的。缩进规则将在随后与复合语句一同谈到。

#### 1.2.4 三重引用

在 Jython 里，你能有选择的使用三个匹配引用标记字符串的开始和结束，匹配的任何一个 """ 或 '''. 在三重引用字符串中，除了另外的三重引用外换行符和包含的引用都被保存起来。程序清单 1-4 注意它是怎样保存内部引用加换行符字符的。程序清单 1-4 是合法的语法，但也不交互地工作。

##### 程序清单 1-4 三重引用的字符串

---

```
# file: triplequote.py
print """This is a triple-quoted string.
Newlines and quotation marks such
as ' and ' are preserved in the string."""
```

---

运行 `python triplequote.py` 的结果是：

```
This is a triple-quoted string.
Newlines and quotation marks such
as ' and ' are preserved in the string.
```

#### 1.2.5 代码程序块

有了起到逻辑行分隔符作用的换行符，Jython 要比 Java 显然地用到少得多的标点。能进一步减少行干扰的是在复合叙述中 Jython 代码块的符号。这些分组，或代码的“程序块”，是由 Java 中括号 {} 来指定的。

这里有一个重要的区别——Jython 不使用花括号组织语句，但是更趋向使用 “:” 字符和缩进空白的组合。Java 的花括号用符号的多重包围来处理分组的多重级别，或者就用符号套用符号的形式 {{}}。而 Jython 却使用多重水平的缩进——一个缩进表示的第一个分组，两个缩进表示第二个分组，依此类推。程序清单 1-5 使用 Jython 的 if 语句显示出复合语句是怎样使用 : 和缩进来定义代码分组的。

##### 程序清单 1-5 基于缩进的分组

---

```
>>>if 1:
...     print 'Outer'
...     if 1:
```

```

...     print "Inner"
...     print "Outer- again"
...
Outer
Inner
Outer- again

```

注意显示出第六行的换行符键入后，在该行最左边的空白处，复合语句是怎样执行的。在最左边的这个表目通知解释器缩排了的分组已完成，因此复合陈述也已完成。

### 什么是缩进层？

缩进层是关系一致的任何内容。最普遍的是每个缩进层就是一个制表键或 4 个空格。这将意味着第二缩进层将是两个制表键或 8 个空格。因此将制表键和空格键混合工作，你能把第一层缩进设置成 4 个空格，把第二层缩进设置成一个制表键，第三层缩进设置成一个标签和 4 个空格等等。一致性很重要。知道你的编辑器正在使用什么空白也是重要的，你能因此在转换编辑器时消除潜在的混乱。什么是合法的，并且什么是被推荐的不同。明智的建议是使用 4 个空格缩排并且不混合制表键和空格键。

## 1.3 注释

回到本章前面的程序清单 1-1，我们看见另外一个重要特征。# 符号。这指明一个注释。所有在 # 字符和物理行结尾之间的代码都被解释器所忽略。它的出现并没被限制为行的开始；它能像这样出现在语句以后：

```
print "Hello world" # It's tradition, what can I do
```

换行符仍然终止逻辑行，但是在 # 和换行符之间的空格被当作一种注释并且被解释器忽略。

注释代码的另外一种方法是匿名方式，并且不影响包含着它的名空间。这使得字符串像注释一般有用，特别是 triple-quoted 字符串，因为他们允许注释跨越多行。程序清单 1-6 显示出不同的注释。程序清单 1-6 不像前面所提到的，当前能在交互式解释器中工作。

### 程序清单 1-6 注释

```

# file: comments.py
# This is a comment
"This works for a comment"
"""This comment can span
multiple lines. Code that you
do not want executed can go
in triple quotes, and it is safe
to use other quotation marks within
the triple quotes, such as:
print 'hello' """

```

在运行 jython comments.py 时没有输出。

## 1.4 文档字符串

在介绍交互模式时，被提及到的探索和发现取决于内省，这是在遮光盖下观察的能力。当在看内部对象时，一个既有帮助又十分重要的 Jython 工具是 Jython 的自我文档，习惯地把它称为 doc 字符串。这些是匿名的字符串文字，它们出现在函数、方法以及类的定义中。

程序清单 1-7 定义了一个称为 foo 的函数。这个例子的第二行是一个匿名字符串文字和一个用于定位的 doc 字符串。注意这个字符串使用 triple-quote 标志，但这并没被要求。只要文档延长超过一行时，任何合法字符串引用工作，triple-quotes 就使它更容易。

程序清单 1-7 文档字符串 doc

---

```
>>> def foo():
...     """This is an example doc string"""
...     pass
...
>>> dir(foo)
['__dict__', '__doc__', '__name__', 'func_closure', 'func_code',
'func_defaults', 'func_doc', 'func_globals', 'func_name']
>>> foo.__doc__
'This is an example doc string'
>>>
```

---

函数 dir() 是一个内置函数，它可以在名空间里看名字绑定。程序清单 1-7 使用 dir 看函数 foo，并且返回在 foo 内被定义了的名字列表。在这个列表中，它是 \_\_doc\_\_。看一看 foo.\_\_doc\_\_ 的内容。给出了在函数定义里提供的绑定到字符串的名字。调用 dir( foo ) 的实际结果在 Jython 2.0 和 Jython 2.1a1 版本之间是不同的。如果你的输出稍微不同，不必太担心。

## 1.5 语句

本节包括 Jython 语句的定义及其使用的例子。语句被分类成简单语句、带有单个从句的、带有复合语句的、那些带有多重复句的或是带有一个关联程序代码块的。

### 1.5.1 简单语句

简单的语句是有一个从句并包含一个逻辑行的那些 Jython 语句。这些语句经常与内置函数相混淆；然而，也有区别。语句是与定义在一个名空间的函数或数据相对立的语法的一部分。

以下章节列出 Jython 的简单语句和他们的语法，并且提供一个例子用法。

#### 1. assert

assert 语句测试一个表达式是否是真，如果它不是真，就产生一个异常。如果两个表达式被提供，那么第二个表达式将被用作生成的 AssertionError 的增量。设置 \_debug\_ = 0，和可能在以后释放的 -O 命令行开关，使 asserts 无效。

语法：

`"assert" expression [, expression]`

例子：

```
>>>a=21
>>>assert a<10
Traceback (innermost last):
  File '<console>', line 1, in ?
AssertionError:
```

### 2. assignment

一个赋值就是把一个对象绑定在一个标志符上。Python 有简单的赋值（=）和增量赋值（+=， -=， \*=， /=， \*\*=， %=， <=>=， &=， |=， ^=）。

语法：

```
variable assignment-operator value
```

例子：

```
>>>var=3
>>>print var
3
>>>var += 2
>>>print var
5
```

### 3. break

break 语句终止一个闭合循环的执行，并继续循环块后的执行。这意味着如果它成立，就将跳过 else 块。

语法：

```
break
```

例子：

```
>>>for x in (1,2,3,4):
...     if x==3:
...         break
...     print x
...
1
2
>>>
```

### 4. continue

continue 语句终止现在循环块的执行，并在它的下一个迭代中又开始闭合循环。

语法：

```
continue
```

例子：

```
>>>for x in (1,2,3,4):
...     if x==3:
...         continue
...     print x
...
```

```

1
2
4
>>>

```

### 5. del

del 语句删除一个变量。变量可以是名空间或特定列表或字典值中的一种。

语法：

```
'del' identifier
```

例子：

```

>>>a='foo'
>>>print a
foo
>>>del a
>>>print a
Traceback (innermost last):
  File "<console>", line 1, in ?
NameError: a>>>
>>> a = [1,2,3]
>>> del a[1]
>>> print a
[1, 3)

```

### 6. exec

exec 语句执行 Jython 代码。exec 语句需要一个表达式，它代表需要执行的代码。一个字符串、打开的文件对象或代码对象都能提供给这个表达式，做 exec () 的第一参数。如果两个参数被提供，第二参数作为代码在其被执行的全局字典中被使用。如果三个参数被提供，第一个是作为被全局名空间使用了的字典，第二个是在代码被执行的本地名空间中。

语法：

```
'exec' expression ["in" expression [",," expression]]
```

例子：

```

>>>exec 'print "The exec method is used to print this"'
The exec method is used to print this
>>>

```

### 7. global

global 语句告诉解析器使用为在整个当前程序代码块中列出的标志符的全局名字绑定。为什么？因为在局部程序代码块的一项赋值标志了所有对作为局部的赋值变量的引用。这会从下列两个代码小应用程序的比较中看出：

```

>>>var = 10
>>>def test():
...     print var # try and print the global identifier 'var'
...
>>>test()
10
>>>

```

我们看到 var 被找到并打印出来。但如果你在 print 语句后把 var 赋上某值，那将会发生什么？

```
>>>var = 10
>>>def test():
...     print var # try and print the global identifier 'var'
...     var = 20 # assign to 'var' in local namespace
...
>>>test()
Traceback (innermost last):
  File "<console>", line 1, in ?
    File "<console>", line 2, in test
NameError: local: 'var'
>>>
```

因为在代码块以内赋值，标志符 var 被指定为局部，这样打印语句是一个错误，因为 var 在局部名空间里不存在。这是要采用 global 的原因。

语法：

```
global identifier [, ' identifier]*
```

例子：

```
>>>var = 10
>>>def test():
...     global var # must designate var global first.
...     print var # try and print the global identifier 'var'
...     var = 20 # assign to 'var' in local namespace
...
>>>test()
10
>>>print var # check global 'var'
20
>>>
```

### 8. import

import 语句定位和初始化被引入的，并把它绑定到 import 被调用范围内的变量上。你能选择性地更改变量名称，用 as new-name 作为引入语句的后缀把 imports 绑定到该变量上。

语法：

```
import module-name
OR
from module-name import names
OR
import module-name as new-name
OR
from module-name import name as new-name
```

例子：

```
>>>import sys
>>>from java import util
>>>import os as myOS
>>>from sys import PackageManager as pm
```

### 9. pass

“do thing” 语句。这个语句是个占位符。

语法：

```
pass
```

例子：

```
>>>for x in (1,2,3,4):
...     pass
...
>>>
```

10. print

print 语句求一个表达式的值，如果需要，就可把结果变换为一个字符串，并且把字符串写入 sys.stdout 或用 >> 语法指向其他像文件一样的对象。一个像文件的对象是一个具有定义的 write 方法。

语法：

```
print [expression]
```

或者

```
print >> fileLikeObject, [expression]
```

例子：

```
>>>print "Hello world"
Hello world
```

11. raise

raise 语句求用它提供的任何表达式的值，并因此产生一个异常。

语法：

```
raise [expression [, expression [, traceback]]]
```

例子：

```
>>>raise ValueError, "No value provided"
Traceback (innermost last):
  File "<console>", line 1, in ?
ValueError: No value provided
```

12. return

return 语句终止这种方法或函数的执行，在求任何用作返回值的表达式的值后，它在该方法或函数里被调用。如果表达式没有被提供，就会返回 None 值。

语法：

```
return [expression]
```

例子：

```
>>>def someFunction():
...     return 'This string is the return value'
```

### 1.5.2 复合语句

复合语句是有一组或是与它们相关的代码块的那些语句。象 if、for 和 while 这样的流动控制语句都是复合语句。不提及 Jython 的块缩进语法，是很难介绍这些的。代码块或语句分组，在 Java 中是被包含在花括号 {} 中。在嵌套的括号 {{}} 中的多重分组。而在 Jython 中并不是这样的。Jython 使用冒号 : 以及标志程序代码块的缩进。有了这些知识，我们就能定义 Jython 的复合语句。以下章节解释 Jython 的复合语句和它们的定义。

#### 1. Class

class 语句被用来定义一个类。一个类语句的求值在当前范围定义一个类。如果定义了一个类，调用一个类就是调用它的构造函数，并且返回它的一个实例。

语法：

```
"class" name[(base-class-name)]:
```

例子：

```
>>>class test: # no base class
...     pass      # place holder
...
>>>t = test() # Calls class statement to make an instance
```

#### 2. def

def 语句是指函数和方法是怎样被定义的。

语法：

```
"def" name([parameters]):
    statements
```

例子：

```
>>>def hello(person):
...     print "Hello ", person
...
>>>hello("world") # calls the hello function
Hello world
```

#### 3. for

for 语句是一个流控制语句，它会通过循环对序列的每个成员迭代。for 语句也能包括一个可选 else 子句，它可在序终止后被执行。

语法：

```
"for" variable "in" expression:""
    statements
    ["else:"]
        statements
```

提供的表达式必须要对一个列表求值。

例子：

```
>>>for x in (1,2,3):
...     print x,
...else:
...     print "in else clause"
...
1 2 3 in else clause
>>>
```

## 4. if

if 语句是当一个表达式求值为真时，有条件地执行一个代码块。

语法：

```
if expression:
    statements
elif expression:
    statements
else:
    statements
```

如果第一个表达式求值为假时，如果 elif 表达式被提供，解释器将继续求 elif 表达式的值。如果所有的 elif 条件是假，语句的 else 组将被执行。

例子：

```
>>>if a==b:
...     print "variable a equals variable b"
...elif a>b:
...     print "variable a is greater than b"
...else:
...     print "variable a is less than b"
```

## 5. try

try 语句执行一个代码块，直到结束或遇到一个错误。

语法：

```
"try:"
    statements
"except" ["," expression ["," variable]]:""
    statements
["else:"
    statements]
OR
["finally:"
    statements]
```

else 或 finally 子句中只有一个子句能被使用，不能同时使用。如果代码的 try 程序块运行没有误差，它继续执行 finally 里的程序代码块。如果 try 程序块有一个错误，它就停止那个块的执行并且继续执行 except 子句。

例子：

```
>>>try:
...     1/0
...except ZeroDivisionError, e:
...     print "You cannot divide by zero: ", e
```

```

...
You cannot divide by zero: integer division or modulo
>>>

```

## 6. while

while 语句执行一个代码块，只要提供的表达式求值为真。

语法：

```
'while' expression ":"
```

例子：

```

>>>x = 10
>>>while x>0:
...     print x,
...     x -= 1
...
10 9 8 7 6 5 4 3 2 1

```

## 1.6 比较 Jython 和 Java

表 1-1 列举在 Jython 使用的语句，并且把它们与 Java 的实现进行比较。

表 1-1 Jython 与 Java 的语句比较

语句	比 较
assert	assert 没有在 Java 版本 1.4 前的 Java 实现里直接对应的语句
assignment	Jython 和 Java 在赋值方面很相近，虽然在这本书后面会谈到一些关于运算符和作用域的不同
break	Jython 和 Java break 语句作用相同
continue	Jython 和 Java continue 语句作用相同
del	Jython 的 del 语句没有一个 Java 对应物
exec	这与 Java 的 Runtime 类中 exec 方法相混淆。然而不是这样。Jython exec 是动态地执行 Jython 代码，而 Java Runtime 类的 exec 却为运行外部的系统处理而存在——而不是动态地执行 Java 代码
global	Java 没有直接对应 Jython global 的语句
import	Java 的 import 语句不允许像 Jython 的 import module as newName 语句把加载的类绑定到一个任意的名字。Java 也不使用 from module import name 语句。类似的 Jython 和 Java import 语句的简单比较表明这些语法差别：
	Java: import javax.servlet.*; Jython: from javax.servlet import *
pass	因为在分组代码块中使用的标点，Java 不要求像 pass 语句的“do thing”占位符，Java 就只用一套空的花括号—— {} 或是一个终止分号的空字符串使一个代码块不具任何函数。Jython 稀少的标点创造了要用占位符来使语法有意义的状况
print	print 在 Jython 里是语句。但在 Java 中，print 是另外对象的一个方法
raise	Jython 的 raise 语句是与 Java 的 throw 语句相对应的
return	Jython 和 Java 的 return 语句作用相同
class	Jython 和 Java 的类语句作用类似，但语法不同。一个 Jython 类包括它在类名字以后括号中的基类，而 Java 使用扩展 (class this extends that) 语法
def	Java 的方法特征符使用了很多 Jython 不使用的信息。像 public int getID (String name) 一样的特征符翻译成 Jython 的 def getID (name)。如此简洁是因为在 Jython 中任何事情是 public，并且类型是不被显式声明的

(续)

语句	比较
for	主要的差别是 Python 的 for 语句要求一个序列。它不使用在 Java 使用的（变量，测试，步长）。循环匹配的第一子句在 Python 和 Java 中看起来像这样： Java: for (int x = 10; x > 0; x++) Python: for x in (10, 9, 8, 7, 6, 5, 4, 3, 2, 1); 恰好，Python 有 range() 和 xrange() 函数为 Python 的 for 语句生成序列： Python: for x in range(10, 0, -1): Java 也缺乏存在于 Python 的 for 语句中的可选的 else 子句
if	Python 的 if 语句与 Java 的类似，除了用 Java 写的 else if 语句就是 Python 的 elif
try	在 Python 中的 try/except/else/finally 与 Java 的 try/catch/finally 相对应。它们在各自的语言里处理异常。然而，在 Python 中，try/except/else 和 try/finally 是不能被混淆的 try 语句的两种独立的形式
while	Python 和 Java while 语句是类似的，但是 Python 当条件求值为假时，有一个被执行的附加 else 子句



## 第 2 章 运算符、类型和内置函数

本章解释了 Jython 的标志符、数据对象、运算符和内置函数。这一章讨论的几乎所有项在没有用户支持的语法下都是可用的——就像 Java 程序在没有任何引入语句下在 `java.lang` 包里获得类一样。异常是 Java 对象。Jython 在 Jython 语法里使用 Java 对象，但是与 Java 对象的相互作用需要一个引入语句。在使用对象之前，知道对象绑定的名字（标志符）是否合法是很重要的，因此，这一章以合法的 Jython 标志符的定义开始。

### 2.1 标志符

用户定义的标志符由字母、数字和下划线组成，但必须以字母或下划线开始。空格是不允许的。标志符的长度不限。标志符不能使用以下单词，Jython 保留的单词列表如下：

and	def	finally	in	print
as	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	

一些合法的标志符举例如下：

`abcABC123`

`A1b2C3`

`_1a2`

`_a_1`

一些不合法的标志符举例如下，并说明了不合法的理由：

`1abc` # 标志符不能以数字开始。

`this variable` # 不允许有空格。

`cost&basis` # 仅允许字母、数字和“—”。

合法的标志符并不一定是好的标志符。人们选择使用 Jython 还是 Python，是有许多理由的，但是阐明是主要的。通过有对标志符深思的选择来作有益的补充，并在需要的地方注释说明它们。

### 2.2 Jython 数据对象

Jython 的数据对象是从 Python 语言描述演化而来的，就像 Jython 本身一样。Python 的类型

名有整形、长整形、浮点型、复数、字符串、元组、列表和字典。

然而，Jython 没有任何传统意义上的类型。这种似是而非仅在语义上。基本类型是数据描述，不是类或实例。然而，在 Jython 里，所有数据通过 org.python.core 包里的 Java 类来表示。Jython 没有任何基本类型。

在这本书里，单词“类型”的使用仅指 Jython 的数据类的简便用法。考虑到 CPython 类并不是所有对象，每一个数据对象类的使用对 Jython 是唯一的。在表示相应的 Python 类的 Java 类名之前加前缀“Py”，因此它们是 PyInteger、PyFloat、PyComplex、PyString、PyTuple、PyList、PyDictionary 等等。每个类名和它的 Py \* 联合类名在整个书中可以交换使用（例如，tuple 和 PyTuple）。

关于像 PyFloat 和 PyList 的 Py \* 类的介绍在本书中还是一个技术上不成熟的执行细节，但把他们包含在这儿讲的原因就是在使用 type() 内置函数时经常会遇到他们。

Jython 的数据类型是动态的——它们是在运行时决定的。它意指在运行时赋给标志符一个值，该值决定哪个数据类与显式的类型声明不一致。绑定名字的对象的 Jython 代码像这样：

```
>>>aString = "I'm a string"
>>>aInteger = 5
>>>aLong = 2500000000L
```

这段代码绑定 PyString 对象到名字 aString，绑定 PyInteger 对象到名字 aInteger，绑定 PyLong 对象到名字 aLong。字符串类型通过引号来表示，整型类型通过数字 5 表示，而长整型类型通过加后缀 L 的整形数表示。PyString 或 PyInteger 像它在 Java 里一样没有任何变量类是通过在他们前面加前缀而被决定的。像 PyLong 对象的 L 后缀这样的显式标注在 Jython 是很少的。

为了确定一个对象是哪个类型，你可以使用内置函数 type():

```
>>>type("I'm a string")
<jclass org.python.core.PyString at 1777024>
>>>s = 'Another string'
>>>type(s)
<jclass org.python.core.PyString at 177024>
>>>aInteger = 5
>>>type(aInteger)
<jclass org.python.core.PyInteger at 7033304>
>>>aLong = 2500000000L
>>>type(aLong)
<jclass org.python.core.PyLong at 871578>
```

注意数的表示可以不同于你最初输入的——不是值，仅是表示（例如，.1 = 0.1）。

在 org.python.core 包里有许多 Py 前缀类。这些类有许多公有方法，这些方法中的大多数是被称做特殊类方法，或者是与 Java 的 Py \* 类一起运行的方法。toString() 方法就是一个这样的例子。对 Java 程序员来说，它是常见的，也可能是希望的，但是对 Python 的 Java 实现来说它是独特的。仅运行在 Java 中的那些需要知道它。这一节定义基本的 Jython 数据对象，把特殊的类方法和 Java 特殊方法放到后面的章节。

这里讨论的类并不是所有用 Py \* 类表示的。在这些讨论的类中，有四个数字对象、三个序列对象、两个映射对象和 Jython 的 None 对象。

### 2.2.1 数字对象

PyInteger、PyLong、PyFloat 和 PyComplex 是表示 Jython 的数字对象。数字类型变量产生赋给相匹配类型一个数字值。这就意味着如果你绑定一个变量到复数，那就使用 PyComplex 对象。为了阐明这些，看表 2-1 中数字对象的定义和例子。

表 2-1 数字对象

Py类	定    义	例    子
Py- Inte- ger	整个数字大小限制在 Java 基本整型类能表示的范围内 (32 位或 +/-2147483647)	x = 28 x = -6
Py- Long	数字后接 “L” 或 “L”。它在大小上仅受可用的存储器的 限制	x = 2147483648L x = 6L
Py- Float	浮点数大小限制在 Java 基本双精度类型能表示的范围内 (64 位)	x = 02.7182818284 x = 0.577216
Py- Complex	一个复数表示两个浮点数相加。它的第一个数表示实部， 它的第二个数表示虚部，字母 “j” 加到第二个数后表示虚部	x = 003.4E9 x = 2.0 + 3.0 x = 9.32345 + 4j

从表 2-1 中注意到 PyInteger 类的值有一个固定的范围。引起一个 PyInteger 对象超过它的最大极限的操作会产生一个如下的溢出异常：

```
>>> int = 2147483647
>>> int += 1
Traceback (innermost last):
File "<console>", line 1, in ?
OverflowError: integer addition: 2147483647 + 1
```

如果一个整数对象有超出这个极限的危险，它应该通过加 L 后缀变成 PyLong 对象。尽管 PyLong 能通过大写 L 和小写 l 中的任何一个构成，但是不鼓励使用小写 l，因为它与数字 1 相似。

四个数字对象中，仅 PyComplex 类的实例有一个公有的实例方法和一个通常用于 Jython 的实例变量，实例变量是复数的实部和虚部，用 x.real 和 x.imag 存取复数变量 x。实例方法是 .conjugate.。复数的 conjugate 如下：

```
conjugate(a + bi) = a - bi
```

存取复数对象的实部和虚部的例子或调用 conjugate 实例方法使用象 Java 对象一样的点表示法 (instance.var)。这样调用 conjugate 方法：

```
>>>z = 1.5+2.0j
>>>z.conjugate()
(1.5-2.0j)
>>>z + z.conjugate()
(3+0j)
```

将这样存取 PyComplex 实例的实部和虚部：

```
>>>aComplexNumber = 1+4.5j
>>>aComplexNumber.real
1.0
>>>aComplexNumber.imag
```

```
4.5
>>># Note that .real and .imag are PyFloat types
>>>type(aComplexNumber.imag)
<jclass org.python.core.PyFloat at 3581654>
>>>type(aComplexNumber.real)
<jclass org.python.core.PyFloat at 3581654>
```

数字对象的另一个特性是它们都是不可改变的——数字对象的值不能改变。假如你需要增加一个叫做 x 的整型对象。那么，你必须使用赋值操作，改变对象 id，而不是改变值。要看对象的 id。你可以使用内置函数 id()。

```
>>> x = 10          # 创建不可变变量
>>> id (x)         # 得到 x 的 id -
2284055
>>> x = x + 1      # 一个赋值需要加 1
>>> id (x)         # 检查它的新 id
4456558
```

x 的值似乎用赋值运算符增加了，但是因为赋值，x 是一个用不同的 id 标注的不同的对象。

在 Boolean (true/false) 表达式中，当数字对象的值等于 0 时，它们都是 false，否则都是 true。这通过使用 Jython 的 if-else 语句来示范：

```
>>>if (0 or 0L or 0.0 or 0.0+0.0j):
...     print 'True'
...else:
...     print 'False'
...
False
```

这四个类是 Jython 的数字对象，但并不是所有的数字对象都能在 Jython 中使用。Java 的数字类在 Jython 里不用修改就能使用。为了能这样，以 import java 开始，然后实例化你所需要的类。

```
>>>import java
>>>l = java.lang.Long(1000)
>>>print l
1000
>>>l.floatValue() # invoke Java instance method
1000.0
```

## 2.2.2 序列对象

PyString、PyTuple 和 PyList 是用做 Jython 序列的类。因为序列化了，这些是有序的数据集。其明显的特征是它的数据和它的可变性。Python 类仅支持字符数据，而 PyTuple 和 PyList 支持任何类型的对象。字符串和元组是不可变的，也就是说它们是不能被改变，但列表是可变的。不像数字对象，序列有一个长度。内置函数 len() 返回序列的长度：

```
>>> len("This is a string sequence")
25
```

序列共享一些通用的语法，即索引和分片语法。

虽然创建不同的序列有不同的语法，但是他们都使用方括号来引用数据集里的元素。首先，检验每一个序列的创建：

```
>>> S = "abc"      # This is a string
>>> T = ("a", "b", "c") # This is a tuple
>>> L = ["a", "b", "c"] # This is a list
```

你会看到它们都有惟一的封装标记——引号、圆括号和方括号。然而，如果你想访问一个序列的第二个元素（“b”），它们都使用相同的语法：

```
>>> S[1]
'b'
>>> T[1]
'b'
>>> L[1]
'b'
```

在方括号里的数字是序列里元素的索引号，从 0 开始。像这样访问带有索引的序列元素返回那个索引的元素，从序列（左边）开始。索引数也可以是负数，它表示从序列的末端（右边）的元素的个数。序列的最后一个元素的索引为 -1，倒数第二个为 -2，依次类推：

```
>>> S = "beadg"
>>> S[-1]
'g'
>>> S[-3]
'a'
```

序列也共享同样的分片符号。一个分片返回一个初始序列的一个子集。在方括号里的分号 ‘:’ 用来表示一个分片。出现在这个分号的两边的数字表示分片的范围。正数（从左边记数）、负数（从右边记数）和没有数字（表示各自的结束点）可用来表示范围。因此，一个像 S [3: 6] 这样的分片符号应该读作“序列 S 的分片从索引号 3 开始，一直到索引号 6，但不包括索引号 6”。这个分片的代码像这样：

```
>>> S = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> S[3: 6]      # 分片从 3 到 6，但不包括 6
[4, 5, 6]
```

一个像 S [3:] 这样的分片应该读作“序列 S 的分片从索引号 3 开始，一直到它的结束点”。这个分片的代码像这样：

```
>>> S = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> S[3:]
[4, 5, 6, 7, 8, 9]
>>
>>> # 如果我们颠倒分片数
>>> S[:3]
[1, 2, 3]
```

一个像 S [4: -2] 这样的分片应该读作“序列 S 的分片从索引号 4 开始，一直到从末端数起的第二个索引号，但不包括该索引号”。这个分片的代码像这样：

```
>>> S = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> S[4:-2]
[5, 6, 7]
>>>
>>> # 负数也可以在左边
>>> S[-4:-1]
[6, 7, 8]
```

Jython 序列在分片语法里也支持第三个整数来表示步长。第三个整数被方括号里的第二个分号隔开，它指一个像 `S[2:8:3]` 这样的分片应该读作“序列 S 的分片从索引号 2 开始，包括每隔三个的索引号，一直到索引号 8”。增加这个第三个数让你使分片像这样：

```
>>> S = 'zahbexlwvo2 fwzo4rsimd'
>>> S[2::2] # 这个分片从索引号 2 开始到结束点，步长为 2
'hello world'
>>>
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[-1:3:-1] # 最后索引号到第三个索引号，索引步长为 -1
[9, 8, 7, 6, 5]
```

序列类中的两个是不可变的——`PyString` 和 `PyTuple`。表 2-1 示范了可变性和当改变一个不可变类型（元组和字符串）时产生的异常。另外，在程序清单 2-1 创建一个 `PyList` 对象和改变它来示范可变性。

#### 程序清单 2-1 序列对象实例

```
>>> S = "This is a string" # Make a PyString instance
>>> type(S) # Confirm object type
<jclass org.python.core.PyString at 8057966>
>>> S[2] = "u" # Try to change the string
Traceback (innermost last):
File '<console>', line 1, in ?
TypeError: can't assign to immutable object
>>>
>>> T = (1, 2.1, "string") # make a PyTuple with varied data
>>> type(T) # Confirm object type
<jclass org.python.core.PyTuple at 4573025>
>>> T[2] = "new string" # Try to change the tuple
Traceback (innermost last):
File '<console>', line 1, in ?
TypeError: can't assign to immutable object
>>>
>>> L = [1, 2.1, "string"] # Make a List with varied data
>>> type(L) # Confirm object type
<jclass org.python.core.PyList at 5117945>
>>> L[2] = "new string" # Try to change the list
>>> L[2]
'new string'
```

在布尔 (Boolean) 表达式中，空序列赋值 `false`，所有其他序列赋值 `true`。

序列多于数字对象，每个序列都有它们自己的一节。下面三节详细地说明了 Jython 的序列

特性。每一节都介绍了对象的语法、对象的特性以及与它相关的方法。

### 1. PyString —— 字符串对象

Jython 的字符串类继承了 Java 的字符串特性，它们根据 Unicode 2.0 标准用两字节字符集构造。这是 Jython 里仅有的字符串类型，从 Unicode 3.0 字符集中的 8 位和 16 位字符类型中的 Cpython 分离出来。Cpython 实现有一个字符串语法，在 Unicode 字符加前缀字母 u。Jython 支持这种语法的兼容性，但仅在对特殊字符 \u 和 \N 的解释时，字符串发生变化。

字符串类型或 PyString 类的实例可通过把一个变量赋值给字符串来创建。字符串用匹配的引号来写。或者是每个匹配的单引号 (')、双引号 ("") 或 Jython 约定的三引号 ("""或'''') 才能起作用。不同的标记允许像如下的其他引号的嵌入：

```
"single-quotes ' in double-quotes"
'double-quotes " in single-quotes'
"""Triple quotes let you use 'anything' inside
except the matching triple quote, and
let you span multiple lines"""
'''This is also a triple quoted string
just in case you need """ inside.'''
```

在 Jython 中文字字符串连接是一个隐含操作。在文字字符串之间留空格来连接它们。

```
>>> print "Adding" "strings" "together is" "this simple."
Adding string together is this simple.
```

这是一个方便的方法，用来保护字符串里不同的引号，如下所示：

```
>>> print "I'm able to" "mix" "different" "quotes this way."
I'm able to mix "different" quotes this way.
```

注意这仅在字符串文字里起作用。你不能用对象的值来连接字符串文字。连接运算符 + 需要在以后的章节讨论。下面代码示范了使用隐含连接非字符串文字产生异常：

```
>>> S = "String value held by object 's'"
>>> print "String literal" S
Traceback (innermost last):
(no code object) at line 0
File "<console>", line 1
"literal" S
^
SyntaxError: invalid syntax
```

有一个另外的引号规则，用来允许数据的字符串表示。这是一个反向引用，包含在反向引用里的表达式结果被转换成数据的字符串表示。下面的代码显示了一个 PyInteger 对象在反向引用整数文字作为表达式。注意表达式结果是被转换成的字符串，而不是反向引用里文字的内容：

```
>>> n = 5
>>> s = `n + 2`
>>> s
'7'
>>> type(s)
<type 'org.python.core.PyString' at 5471111>
```

反向函数像 repr() 函数一样，repr() 函数返回数据的字符表示，在重构那些数据时该数据是有用的。不同于内置 str() 函数，它返回对象的字符描述。时常 repr(object) == str(object)，

但不能保证两个相等，因为这两个函数有不同的目标。当讨论 Jython 对象和它们的特殊 `_str_` 和 `_repr_` 方法时，我们再重新回顾。

当绑定一个字符串文字值给这样的名字时，`PyString` 实例被生成：

```
>>>S = "A string"
>>>type(S)
<class 'org.python.core.PyString' at 177024>
```

Python 类型对象支持许多方法。这些方法用类似 Java 的点标注法称作 `object.method`。`split` 方法返回被限界符指定的子字符串列表，例如：

```
>>>S = '1,2,3,4,5'
>>>S.split(',')
['1', '2', '3', '4', '5']
```

如果没有专门的限界符，它按空格分开：

```
>>>S = "A string with spaces."
>>>S.split() #注意，没有专门的限界符
['A', 'string', 'with', 'spaces.']}
```

字符串处理是 Jython 能力之一，`PyString` 类有许多方法。表 2-2 列出了 `PyString` 方法。字符串方法的一个很重要特性就是它们总是返回字符串的拷贝，而从不修改字符串。

表 2-2 PyString 方法

方 法	描 述
<code>capitalize</code>	<code>s.capitalize()</code> 返回 <code>s</code> 的字符串匹配值，第一个字母大写： <code>&gt;&gt;&gt; s = "monday"</code> <code>&gt;&gt;&gt; s.capitalize()</code> <code>Monday</code>
<code>center</code>	<code>s.center(width)</code> 返回位于指定宽度的字符串中央的 <code>s</code> 的值。该宽度表示希望返回的字符串的总数量， <code>s</code> 的值用空格来填充以满足宽度： <code>&gt;&gt;&gt; s = "Title"</code> <code>&gt;&gt;&gt; s.center(40)</code> <code>'                               Title'</code>
<code>Count</code>	<code>s.count(substring [, start [, end]])</code> 返回一个整数，该数表示一个子字符串在 <code>s</code> 的字符串值中被找到的次数。start 和 end 参数是可以选择的，并定义在搜寻的那部分字符串： <code>&gt;&gt;&gt; s = "BacBb is the music-cryptic spelling of Bach."</code> <code>&gt;&gt;&gt; s.count("c")</code> <code>4</code> <code>&gt;&gt;&gt; s.count("c", 18)</code> <code>3</code> <code>&gt;&gt;&gt;&gt; s.count("c", 18, 28)</code> <code>2</code>
<code>Encode</code>	<code>s.encode([encoding] [, error-handling])</code> 返回 <code>s</code> 的字符串值， <code>s</code> 是带有指定的字符编码的字符串或如果没有指定则是缺省的字符编码。error-handling 参数表示编码转换处理怎么处理错误。Error-handling 可设定为严格的、可忽略的或替换： <code>&gt;&gt;&gt; s = "Rene Descartes"</code> <code>&gt;&gt;&gt; s.encode("iso8859_9", "replace")</code> <code>'Ren? Descartes' # 注意替换字符 "?"</code>

(续)

方 法	描 述
endswith	如果 s 以 suffix 结尾, s.endswith(suffix [, start [, end]]) 返回 1, 否则返回 0。选项 start 和 end 参数表示当检查时要考虑的那部分字符串: <pre>&gt;&gt;&gt; s = "jython.jar" &gt;&gt;&gt; s.endswith(".jar") 1 &gt;&gt;&gt; s.endswith("n", 5, 6) 1</pre>
expandtabs	s.expandtabs([tabsize]) 返回带有在 tabsize 里用指定的空格数替换的 tab 字符的字符串, 如果没有指定任何 tabsize, 则返回 s。tab 字符在字符串里表示为 \t <pre>&gt;&gt;&gt; s = "\tA tab-indented line" &gt;&gt;&gt; s.expandtabs(4) ' A tab-indented line'</pre>
find	s.find(substring [, start [, end]]) 返回字符串 s 中子字符串第一次出现的索引号。start 和 end 索引号是可选择的, 并限制在这些提供的范围内搜寻。如果子字符串没找到, 则返回-1 <pre>&gt;&gt;&gt; s = "abcdedcba" &gt;&gt;&gt; s.find("c") 2 &gt;&gt;&gt; s.find("c", 5) 6</pre>
index	s.index(substring [, start [, end]]) 与 find 一样, 但如果子字符串没找到则产生一个 ValueError 异常
isalnum	s.isalnum() 返回 0 或 1, 依赖于 s 字符串值是否都是文字数字 <pre>&gt;&gt;&gt; s = "123abc" &gt;&gt;&gt; s.isalnum() 1 &gt;&gt;&gt; s = "1.2.3.a.b.c" &gt;&gt;&gt; s.isalnum() 0</pre>
isalpha	除了在这个方法前字符串必须都是字符时才返回 1, 其他方面, s.isalpha() 与 isalnum 一样
isdigit	除了在这个方法前字符串必须都是数字时才返回 1, 其他方面, s.isdigit() 与 isalnum 一样
islower	如果字符串中至少有一个字符, 并且所有字符都是小写, s.islower() 返回 1; 如果字符串中没有任何字符或它们中有一个是大写, 则返回 0
isspace	s.isspace() 测试字符串是否都是空格, 如果是, 则返回 1; 如果字符串有非空格字符, 则返回 0
istitle	“Title casing” 是每个单词的第一个字母被大写的地方, 没有任何非第一个字符是大写; 如果字符串符合这个规则, s.istitle() 返回 1 <pre>&gt;&gt;&gt; s = "Jython For Java Programmers" &gt;&gt;&gt; s.istitle() 1</pre>
isupper	如果字符串中所有字符都是大写, s.isupper() 返回 1; 如果字符串中有一个小写或没有任何字符, 则返回 0
join	s.join(sequence) 连接 list、tuple、java.util.Vector 或任何其他的用字符串值 s 作为分隔的 sequence <pre>&gt;&gt;&gt; s = ", and "</pre>

(续)

方 法	描 述
	>>> s.join ( ["This", "that", "the other thing"]) This , and that, and the other thing
ljust	s.ljust (width) 用空格填充字符串的右边，以得到特定宽度的字符串
lower	s.lower () 返回把字符串的所有字符变成小写的拷贝
lstrip	s.lstrip (width) 用空格填充字符串的左边，以得到特定宽度的字符串
replace	s.replace (old, new, [ , maxsplits]) 返回用 “new” 字符串代替 “old” 字符串的字符串拷贝。有一个可选参数 maxsplits，它是一个旧字符串被替换的最大次数的数值。因此，s.replace (“this”, “that”, 5) 将仅返回被 “that” 替换的 “this”的最开始五个实例
rfind	s.rfind (substring [, start [, end]]) 与 find一样，除了它返回字符串 s 里最高的索引或子字符串最右边的出现，且子字符串在该字符串 s 里能被找到外
rindex	s.rindex (substring, [, start [, end]]) 与 rfind一样，除了如果子字符串没找到则产生一个 ValueError 异常外
rjust	s.rjust (width) 用空格来填充字符串的左边以得到指定宽度的字符串
rstrip	s.rstrip () 返回截去尾部空白的字符串
split	s.split ([separator, [, maxsplits]]) 返回一列 s 的字符串部分，该 s 由提供的 separator 或空格分开，或没有提供任何 separator, maxsplits 参数，如果被提供了，则限制被执行的 splits 的数量，同时从左边记数，因此列表里最后的字符串包含所有剩下的
splitline	s.splitlines ([keepends]) 返回一列字符串，该列字符串由划分的 s 在每一个它包含的换行符创建
startswith	s.startswith (prefix [, start [, end]]) 比较 prefix 和入口字符串的开始，或者被可选的 start 和 end 参数指定的字符串部分。如果 prefix 匹配被比较的字符串的开始，则它返回 1，否则返回 0
strip	s.strip () 返回字符串 s 的拷贝，该字符串前面和后面空格被删去
swapcase	s.swapcase () 返回字符串 s 的拷贝，该字符串具有被转换的首字母的大写
title	s.title () 返回 s 标题格式的拷贝。标题格式是每个单词的第一个字母大写，而其他字母不是大写
translate	s.translate (table [, deletechars]) 返回 s 被转换的拷贝。转换包含两件事情：字符（被 table 指定的）的再映射和一列要删除的字符（被 deletechars 指定的）。你想删除的字符被包含在作为可选参数的字符串里，因此要删除 a, b 和 c，使用：
	>>> s.translate (table, "abc") 转换表是一个 256 字符的字符串，该字符串通常是用模块的 maketrans 方法生成的。在这里引入和使用模块是有点过早了，但这是相当直观的：
	>>> import string # this loads the string module >>> table = string.maketrans ("abc", "def") >>> s = "abcdefg" >>> s.translate (table, "g") # translate "abc" and delete "g" "defdef"
upper	s.upper () 返回字符串 s 的拷贝，该字符串的所有字母大写

变量 s 被用来表示 PyString 的实例。

## 2. PyTuple

PyTuple 类是不可变的序列，它包含任何类型对象的引用。有时候可能有些冲突，因为元组是不可变的，但它可以包含可变的对象的引用。如果它的内容改变了，元组怎么能是不可变的？因为元组包含了对象的引用——对象的 id，所以它可以。当一个可变元组对象的值改变

时，它的 id 仍保持不变。因此，元组本身是不变的。

元组的语法是特别的，一个逗号分隔表赋值给元组，但元组是用圆括号表示的。这就意味着你能在圆括号里用逗号分隔表创建一个元组，或者用空圆括号创建一个空元组。

```
>>>t = (1,2,3)
>>>type(t)
<class 'org.python.core.PyTuple' at 6879429>
>>>t = 1,2,3
>>>type(t)
<class 'org.python.core.PyTuple' at 6879429>
>>> type((1,2,3))
<class 'org.python.core.PyTuple' at 6879329>
>>>myTuple = (1, 'string', 11.5) # tuples can contain any object type
```

构建一个单元素元组仍需要一个逗号：

```
>>>t1 = ('element one',)
>>>t2 = "element one",
>>>type(t1)
<class 'org.python.core.PyTuple' at 4229391>
>>>type(t2)
<class 'org.python.core.PyTuple' at 4229391>
>>>print t1
('element one',)
>>>print t2
('element one',)
```

现在，改变元组来证实它是不可变的：

```
>>>T = ("string", 1, 2.4)
>>>T[0] = "another string"
Traceback (innermost last):
File "<console>", line 1, in ?
TypeError: can't assign to immutable object
```

与 PyString 对象相比，PyTuple 对象没有任何相关方法。

### 3. PyList

PyList 与 PyTuple 相似，但 PyList 是可变的，其值能够被改变。这也使得 PyList 类在序列类中是惟一的。一个列表可在方括号 [1, 2, 3] 里通过封装逗号分隔的对象来表示，或者仅是方括号来创建空列 list []。包含在列表里的对象可以是任何对象。列表包含各种对象，如 integer、float、string、complex，如下：

```
>>>myList = [1,'a string', 2.4 , 3.0+4j]
>>>type(myList)
<class 'org.python.core.PyList' at 250438>
```

PyList 对象有九种相关方法，这些方法在表 2-3 中有所描述。

表 2-3 PyList 方法

方法	描述
append	L.append (object) 把对象加到列表 L 的末尾

(续)

方 法	描 述
count	L.count (value) 返回一个表示 “value” 出现在列表中的次数的整数
extend	L.extend (list) 把 “list” 的元素加到列表 L 的末尾
index	L.index (value) 返回一个整数，该整数表示 value 在列表中第一次出现的索引号
insert	L.insert (index, object) 表示在指定的 index 处把 object 插入到列表 L 中
pop	L.pop ([index]) 从列表中移出对象并返回。如果没有提供 index，则使用列表的最后一个元素
remove	L.remove (value) 从列表 L 中移出第一次出现的指定的 value
reverse	L.reverse () 在适当位置颠倒列表的顺序。没有返回值，因为是在适当位置对列表进行操作
sort	L.sort ([comfunc]) 在适当位置对列表进行排序。如果提供 comfunc，它比较两个值 (x, y)，如果 x 小于 y，则返回负数；如果 x 等于 y，则返回 0；如果 x 大于 y，则返回正数

字母 L 通常用来表示 PyList 的实例。

像数字类一样，Jython 在不做任何修改的情况下能使用所有 Java 序列。PyString、PyTuple 和 PyList 是特定的 Jython 对象，但你可以像使用 java.util.Vector 一样容易使用。注意部分符号和其他的 Jython 特定的列表属性可能在你正在使用的 Java 序列对象中不起作用。

### 2.2.3 映射对象

Jython 有两个映射类或散列 (hash) 类型，它们被引用做字典 (dictionaries)。“映射”是指在不可变键和另一个对象——值之间进行连接。Jython 的两个映射类的差别是它们使用的键类型。

字典里的值可通过方括号里提供的键或通过 PyDictionary 方法进行存取。键的符号如 D[key]，像下面例子一样使用：

```
>>> D = {"a": "alpha", "b": "beta"}
>>> D["a"]
'alpha'
>>> D["b"]
'beta'
```

映射也有长度——一个表示映射含有多少个键：值对的数字。内置函数 len () 检索这个值：

```
>>> D = {1: "a", 2: "b", 3: "c"}
>>> len(D)
3
```

#### 1. PyDictionary

PyDictionary 类表示用户创建的字典，且与在 Cpython 能找到的单一字典类型是并行的。大括号用来创建一个字典，冒号用来分隔键—值对。创建一个限定为名字 D 的 PyDictionary 像这样：

```
>>> D = {"String type key": "String", 1: "int", 2.0: "float"}
>>> type(D)
<jclass org.python.core.PyDictionary at 1272917>
```

在上述的字典里的键是不同的类型，但所有类型是不可变的。按值比较的可变类型不能是 PyDictionary 键，由它所产生的异常可知：

```
>>> L = [1, 2, 3, 4] # 创建一个列表（可变序列）
>>> D = {L: "My key is a list"} # 使 L 为键值
Traceback (innermost last):
File "<console>", line 1, in ?
TypeError: unhashable type
```

另一个对键非常注意的是：用作键的数字值按照它们的值怎样比较来进行散列。PyInteger 1 与 PyFloat 1.0 是不同的对象，但它们进行比较是相等的，因此在字典里函数的键是相同的。

```
>>> A = 1
>>> B = 1
>>> C = 1.0 + 0j
>>> eval ("A==B==C") # 注意：整数 1 表示真
1
```

上述的三个数字值在值上是相等的。在这种情况下也可能意味着它们散列成相同的值：

```
>>> hash (A) == hash (B) == hash (C)
1 # 这表示真 1
```

这意味着使用这些值作为字典键将被解释成下列键：

```
>>> D = {1: "integer key", 1.0: "float key", 1+0j: "complex key"}
>>> D # 转储 D 的值以证实真正只有一个键
{1: 'complex key'}
```

类型 PyDictionary 对象有九个与它们相关的方法，并在表 2-4 中列出。

表 2-4 PyDictionary 方法

方 法	描 述
clear	D.clear () 从字典 D 中删除所有的键：值对
copy	D.copy () 返回字典 D 的浅复写
get	D.get (key [, defaultVal]) 如果与指定键相关的值存在，则返回它。如果字典没有指定键，则返回一个可选的缺省值
has_key	D.has_key (key) 如果字典 D 有指定的键，返回 1；否则返回 0
items	D.items () 返回字典 D 里所有的键：值对的复写
keys	D.keys () 返回字典 D 里所有键的列表，该列表是字典 D 的复写，因此改变这列表并不影响初始字典里的键
setdefault	D.setdefault (key [, defaultVal]) 返回与指定键相关的值，如果该键不存在，且提供了一个可选的 defaultVal，那么增加 defaultVal 作为指定的键的相关值，并返回 defaultVal
update	D.update (dict) 用在 dict 里找到的所有键：值对修改字典 D 里所有的键：值对。增加字典 D 里不存在的键，并用 dict 参数里的值修改字典 D 里存在的键
values	D.values () 返回字典 D 里所有值的列表，这个列表是字典 D 值的复写，因此改变这列表并不影响初始字典里的值

字典 D 用来表示 PyDictionary 实例。

## 2. PyStringMap

Jython 的 PyStringMap 类是一个仅使用字符串作为键的字典。这与可使用任何不可变对象作为键的 PyDictionary 形成对照。这不是 Jython 程序员正常创建的对象，但它是许多内置函数返回的对象。它为这样受限的键所提供的执行优势而存在。

PyStringMap 通常用作名空间，在名空间里它使字符串键有意义。因此返回名空间信息的内置函数返回一个 PyStringMap 对象。这些函数是 locals()、globals() 和 vars()。另外，在命名为 \_dict\_ 的 PyStringMap 里，Jython 对象保持对象特性。在交互式的解释器中检验这些来看 PyStringMap 使用：

```
>>># Fill up namespace so there's something to look at
>>>functional = ['Erlang', 'Haskell']
>>>imperative = ['C++', 'Java']
>>>
>>>globalStringMap = globals()
>>>type(globalStringMap)
<jclass org.python.core.PyStringMap at 6965863>
>>> # Now let's look at the global namespace contents with a "print"
>>> # Your global namespace may differ significantly if you have
>>> # been doing other exercises in the same interpreter session.
>>>print globalStringMap
{'globalStringMap': {...}, '__name__': '__main__', '__doc__': None,
 'imperative': ['C++', 'Java'], 'functional': ['Erlang', 'Haskell']}
>>>
>>>class test:
...     pass
...
>>>testNames = vars(test)
>>>type(testNames)
<jclass org.python.core.PyStringMap at 6965863>
>>>print testNames
{ '__module__': '__main__', '__doc__': None}
>>>
>>>type(test.__dict__)
<jclass org.python.core.PyStringMap at 6965863>
```

PyStringMap 和 PyDictionary 是特定的 Jython 映射类。然而，所有的 Java 映射类在 Jython 里也是可用的。

### 2.2.4 PyNone

没有值可用内置对象 None 来表示。None 在 Jython 里的使用就象 null 在 Java 里的使用一样。PyNone 类是表示这种类型的对象。PyNone 是一个称做 PySingleton 内部对象的派生类。这也意味着仅有一个单一的 PyNone 类型的对象。在条件语句里 PyNone 实例赋 false 值，没有长度。

```
>>>type(None)
<jclass org.python.core.PyNone at 3482695>
>>>a = None
>>>len(a)
Traceback (innermost last):
File "<console>", line 1, in ?
```

```

TypeError: len() of unsized object
>>> if a:
...     print "This will not print because a evaluates to false."
...
>>>

```

## 2.3 运算符

运算符分成五类：算术运算符、移位运算符、比较运算符、布尔运算符和序列运算符。按照与它运行的特定的类或类型，与运算符一起使用的对象类型被包括在它的描述里例如“数字”适合于 PyInteger、PyLong、PyFloat 和 PyComplex，“序列”适合于 PyList、PyTuple 和 PyString，当然“字典”适合于 PyDictionary。

### 2.3.1 算术运算符

开始三个数字运算符是一元的，因此它们的描述是基于在单变量  $x$  前加前缀。后六个运算符是二元的，它们是按照表达式“ $x$  运算符  $y$ ”来描述的。算术运算符列表于表 2-5。

表 2-5 算术运算符

运算符	描述
$-( -x)$	一元减运算符，求出数字对象的负数，并仅用于数字对象
$+( +x)$	一元加运算符，使数字对象 $x$ 不变，并只能应用于数字对象
$~( ~x)$	一元求反运算符，求出 $x$ 的逐位求反（代数相当于 $- (x + 1)$ ）。它仅能用在 PyInteger 和 PyLong 对象中
$+(x + y)$	加运算符，求出两个数字对象的和。注意加应用于数字对象，但 + 符号也是用来连接两个在序列运算符说明的序列对象
$-(x - y)$	减运算符，求出两个数字对象的差
$*(x * y)$	乘运算符，求出 $x$ 和 $y$ 相乘的积。它应用于数字对象，但 * 运算符也用于序列
$**(x ** y)$	幂运算符，求出 $y$ 的 $x$ 次方的值，并能用于所有的数字对象
$/(x / y)$	除运算符，求出 $x$ 除以 $y$ 的商，除运算符能用于除了 $y = 0$ 外的所有数字对象，当然，如果 $y = 0$ ，则会产生一个 ZeroDivisionError 异常
$%(x \% y)$	取模运算符求出 $x$ 除以 $y$ 的余数。取模运算符能用于除了 $y = 0$ 外的所有数字对象，当然，如果 $y = 0$ ，则会产生一个 ZeroDivisionError 异常

变量 D 用来表示 PyDictionary 实例。

### 2.3.2 移位运算符

移位运算符应用于数字对象。表 2-6 列出了移位运算符。

表 2-6 移位运算符

运算符	描述
$x << y$	左移运算符，将 $x$ 的位向左移，移动 $y$ 次
$x >> y$	右移运算符，将 $x$ 的位向右移，移动 $y$ 次
$x \& y$	逻辑与 (AND) 运算符，将 $x$ 和 $y$ 两个操作数的对应位作逻辑与运算

(续)

运算符	描述
$x y$	逻辑或 (OR) 运算符，将 $x$ 和 $y$ 两个操作数的对应位作逻辑或运算
$x^y$	逻辑异或运算符，将 $x$ 和 $y$ 两个操作数的对应位作逻辑异或运算

### 2.3.3 比较运算符

比较运算需要理解什么是真 (true) 和什么是假 (false)。在 Jython 里，false 就是数字 0，None 是空列表、空元组或空字典，其他的都是 true。比较运算符（见表 2-7）以整数 1 为 true 和 0 为 false。它们可在任何类型的 Jython 对象——数字、序列或字典中使用。

表 2-7 比较运算符

运算符	描述
<	小于
>	大于
$=$	等于
$\geq$	大于或等于
$\leq$	小于或等于
$\neq$	不等于
$\diamond$	不等于的另一种写法，这种写法是过时了的，不鼓励大家使用
is object	如果对象 $x$ 和对象 $y$ 相同，则 “ $x$ is $y$ ” 取 1。“id” 是一个返回对象 id 的内置函数，因此 “is” 是 $id(x) == id(y)$ 的一种简写
identity match	
is not object	如果对象 $x$ 的 id 和对象 $y$ 的 id 不相同，则 “ $x$ is not $y$ ” 取 1(true)
identity mismatch	
in sequence	如果 $x$ 是序列 $L$ 的成员，则 “ $x$ in $L$ ” 取 1(true)
membership	
not in sequence	如果 $x$ 不是序列 $L$ 的成员，则 “ $x$ not in $L$ ” 取 1(true)
exclusion	

### 2.3.4 布尔运算符

布尔运算符（见表 2-8）和真值条件。

表 2-8 布尔运算符

运算符	描述
not	如果自变量是 false，则是一个返回 1 的布尔运算符
or	如果第一个自变量的值是真，则是一个返回第一个自变量的值的布尔运算符；如果第一个自变量的值不是真，则是一个返回第二个自变量的值的布尔运算符
and	如果第一个自变量的值是假，则是一个返回第一个自变量的值的布尔运算符；如果第一个自变量的值是真，则是一个返回第二个自变量的值的布尔运算符

### 2.3.5 序列运算符

表 2-9 显示了序列对象的运算符。程序清单 2-2 显示了如何串接序列。

表 2-9 序列运算符

运算符	描述
+	用作串接序列，程序清单 2-2 显示了与每一种 Jython 序列对象的串接与序列使用时的重复运算，语法是：
*	序列 * 乘子，乘子是序列重复的次数，因此它必须是整数。这也就是说它只能是 PyInteger 或 PyLong 对象。像下面的例子一样使用： <pre>&gt;&gt;&gt; L = [1]*3 &gt;&gt;&gt; L [1, 1, 1]</pre>

### 程序清单 2-2 串接序列

```
>>> var1 = 'This, '
>>> var2 = 'that, '
>>> var1 + var2 + "and the other thing."
'This, that, and the other thing.
>>>
>>> L = ['b', 'e', 'a']
>>> L + ['d', 'g']
['b', 'e', 'a', 'd', 'g']
>>>
>>> T1 = (1, 1.1, 1.0+1.1j)
>>> T2 = (1L, 'one', ['o', 'n', 'e']) #notice list within the tuple
>>> T1 + T2
(1, 1.1, (1+1.1j)), 1L, 'one', ['o', 'n', 'e'])
```

尽管不是运算符，字符串格式字符对 Jython 的字符串处理能力是非常重要的。Jython 也支持用在 C 的 printf 函数类型的字符串格式字符。字符串里有一些占位符，它由 % 符和一个字母组成。例如，%s 是一个占位符，该占位符可被外部的字符串值代替。当字符串后接 % 和元组或包含需要的值的字典时，这些字符可用值来代替。表 2-10 列出了可用的格式字符列表和它们支持的数据类型。

表 2-10 字符串格式字符

字符	Jython 类型
%c	字符
%d	整数（在 C 的 printf 中最初是表示十进制整数）
%e	浮点数，在科学标记法中用小写 “e”
%f	浮点数
%g	浮点数，根据输出时占宽度较小的一种自动选择 %e 或 %g 格式
%i	整数
%o	无符号八进制整数
%r	提供对象的字符串表示，该表示由内置函数 repr() 的使用来决定

(续)

字符	Jython 类型
%s	字符串
%u	无符号整数（在 C 的 printf 中最初是表示十进制无符号整数）
%x	用小写字母表示无符号十六进制整数
%E	浮点数，PyFloat，使用大写字母“E”转换成科学标记法
%G	浮点数，根据输出时占宽度较小的一种自动选择%E 或%g 格式
%X	用大写字母表示无符号十六进制整数
%%	文字%

在转换时提供给格式化字符串的值可以在字典里或元组里。如果使用元组，按照它们出现的顺序插入有元组值的格式字符。该语法如下：

```
>>>"Meet the %s at %i o'clock" % ("train", 1)
"Meet the train at 1 o'clock"
```

随着格式字符串变得较大时，顺序转换就难于使用。为了处理这些，Jython 允许字典提供值来格式化字符串。然而，字典的键必须在字符串的圆括号里提供。该语法的例子如下：

```
>>>D = {"what": "train", "when": 1}
>>>"Meet the %(what)s at %(when)i o'clock" % D
"Meet the train at 1 o'clock"
```

当在长字符串里陷入错误或冲突时，键名是有用的线索。

## 2.4 内置功能

Jython 的内置功能 (built-ins) 是在没有任何用户定义对象时 Jython 里的可用的异常、函数和变量。这一节仅包括函数和变量，而异常留到第 3 章的“错误和异常”中讲述。

Jython 的内置函数是一个丰富的工具集，它把内省、类型操纵和动态执行形成流水线操作。本节也是按照这些功能进行组织的。

### 2.4.1 内省函数

内省 (introspection) 是一种语言提供关于它本身的信息。Java 为此有映象 API。Jython 通过它的内置函数提供内省。内省函数粗略地分为检查对象和显示定义了什么对象。

Jython 里每个对象都有一个类型和 id，它们能分别从内置函数 type () 和 id () 中检索到。这些函数定义在表 2-11 中。

表 2-11 对象检查函数

函数	描述
type (object)	返回指定对象的类型作为 org.python.core 包的一个类名
id (object)	返回一个整数，它表示指定对象的唯一 id 号

假如有一个叫 S 的字符串对象，像这样检索 S 的类型和 id：

```
>>>S = "String object"
>>>type(S)
<jclass org.python.core.PyString at 1272917>
>>>id(S)
2703160
>>>S2 = S
>>>id(S2) # to prove S2 is a reference to the same object
2703160
```

检查内置函数，而不是字符串，用同样的方法进行。看看内置函数类型：

```
>>>type(type)
<jclass org.python.core.PyReflectedFunction at 2530155>
>>>id(type)
2804756
```

PyReflectedFunction 是一个本章没有讨论的 org.python.core 的 Py\* 类。它显然是一个函数类型，考虑到函数是类型名的一部分，它应该是可调用的。首先，实例调用它来检索它自己的类型，但知道一个对象是可调用的并不总是这样。幸运的是 Jython 有内置函数 callable(object)。根据讨论的对象是否是可调用的对象，callable(object) 返回 1 或 0(true 或 false)。

```
>>> callable(type) # 内置函数"type"是可调用的吗?
1
>>> callable(S) # 字符串"S"是可调用的吗?
0
```

但是内置函数 callable() 对 Java 对象的判断并不总是正确的，并不总是可信的。

type、id 和 callable 内置函数检查对象的属性。然而你并不总是知道哪个函数被定义了。有些内置函数仅检查存在什么名字绑定。这些函数是 dir()、vars()、globals() 和 locals()。要完全理解这些，需要理解好本章提到的 Jython 的命名空间。第 4 章“用户自定义函数和变量的作用域”详细说明了 Jython 的命名空间。

**dir([object])** 返回在指定的对象里已定义的名字列表，如果没有指定对象，则返回当前域内的名字列表。

**vars([object])** 返回指定对象里约束名的字典。这实际上是自己内部的 \_dict\_ 对象，这将在后面与 Jython 类定义进行讨论。vars 函数仅对有 \_dict\_ 属性的对象起作用。如果没有指定对象，它与 locals() 一样。

**globals()** 返回如同字典一样的对象，该对象表示在全局命名空间内定义的变量。

**locals()** 返回如同字典一样的对象，该对象表示在局部命名空间内定义的变量。

程序清单 2-3 包括函数定义和类定义，因此我们能使用 locals() 和 vars()。注意当测试程序清单 2-3 时，你得到的准确的输出可能不同。输出的差异可能由以前的交互实验或其他的不合理原因产生的。

### 程序清单 2-3 探究命名空间

---

```
>>>def test1():
...     "A function that prints its own local namespace"
...     avariable = "Inside test function"
...     print locals()
```

```

...
>>> class test2:
...     "An empty class"
...     pass
...
>>> dir()      # look at list of global names
['__doc__', '__name__', 'test1', 'test2']
>>>
>>> globals()  # look at global names and values
{'test2': <class '__main__.test2' at 5150801>, '__name__': '__main__',
 '__doc__': None, 'test1': <function test1 at 7469777>}
>>>
>>> dir(test1) # look at names in the test function
['__doc__', '__name__', 'func_code', 'func_defaults', 'func_doc',
 'func_globals', 'func_name']
>>>
>>> dir(test2) # look at list of names in the class
['__doc__', '__module__']
>>>
>>> test1()    # invoke method that prints locals()
{'avariable': 'Something to populate the local namespace'}
>>>
>>> vars(test2) # user-defined classes have a __dict__, so vars works
{'__module__': '__main__', '__doc__': None}

```

属性检查不是仅局限于 Jython 对象。Jython 的特别之处在于它平等地应用于 Java 对象。语句 import java 把可用的 java 包引入 Jython 里。所有 Java 都可用同样的方式检查。程序清单 2-4 检查了 java.util.ListIterator。这是一个常见的接口，但可以假定它是你想探究的最近下载的包。你可用 dir(package) 遍历包每层的内容，看看它们里面是什么。在 Jython 里，如程序清单 2-4 所示范一样，这些都可交互实现。程序清单 2-4 的实际输出根据 Java 的版本不同而有所不同。

#### 程序清单 2-4 探究 Java 包

```

>>> import java
>>> dir(java)
['__name__', 'applet', 'awt', 'beans', 'io', 'lang', 'math', 'net', 'rmi',
 'security', 'sql', 'text', 'util']
>>> dir(java.util)
['AbstractCollection', 'AbstractList', 'AbstractMap',
 'AbstractSequentialList', 'AbstractSet', 'ArrayList', 'Arrays', 'BitSet',
 'Calendar', 'Collection', 'Collections', 'Comparator',
 'ConcurrentModificationException', 'Date', 'Dictionary',
 'EmptyStackException', 'Enumeration', 'EventListener', 'EventObject',
 'GregorianCalendar', 'HashMap', 'HashSet', 'Hashtable', 'Iterator',
 'LinkedList', 'List', 'ListIterator', 'ListResourceBundle', 'Locale', 'Map',
 'MissingResourceException', 'NoSuchElementException', 'Observable',
 'Observer', 'Properties', 'PropertyPermission', 'PropertyResourceBundle',
 'Random', ' ResourceBundle', 'Set',
 'SimpleTimeZone', 'SortedMap', 'SortedSet', 'Stack', 'StringTokenizer',
 'TimeZone', 'Timer', 'TimerTask', 'TooManyListenersException', 'TreeMap',
 'TreeSet', 'Vector', 'WeakHashMap', '__name__', 'jar', 'zip']
>>> dir(java.util.ListIterator)
['', 'add', 'hasPrevious', 'nextIndex', 'previous', 'previousIndex', 'set']

```

```
>>> test = java.util.ListIterator() # try calling it anyway
Traceback (innermost last):
File "<console>", line 1, in ?
TypeError: can't instantiate interface (java.util.ListIterator)
```

## 2.4.2 数字函数

内置数字函数对通常的数学运算来说是很方便的。表 2-12 列出了六个函数。

表 2-12 内置数字函数

函 数	描 述
abs (x)	返回数 x 的绝对值
divmod (x, y)	返回 x 除以 y 的除法和模（例如，2/3, 2%3）
hex (x)	返回整数 x 的十六进制表示作为字符串
oct (x)	返回整数 x 的八进制表示作为字符串
pow (x, y [, z])	返回 $x^{**}y$ 。如果提供 z，则返回 $x^{**}y \% z$
round (x [, ndigits])	返回 x 靠近 0 的 PyFloat 数，如果提供 ndigits，x 从数字上接近 ndigits。负数 ndigits 表示数的左边，正数 ndigits 表示数的右边。当前在 Jython 2.0 中 ndigits 变量必须是整数。这与 Cpython 实现稍微有点不同，在 Cpython 里 ndigits 可以是任何能转换成整数的数

## 2.4.3 类型转换函数

Java 类型转换需要类型 casts——在 Java 程序里通常是普遍的。在 Jython 里，这些类型转换用内置函数处理。这些函数按照返回的对象命名。这种简明性使得定义没有必要——下面的例子充分阐明了它们的使用。程序清单 2-5 显示了每种类型转换函数。

程序清单 2-5 类型转换

```
>>>int(3.5)
3
>>>long(22)
22L
>>>float(8)
8.0
>>>complex(2.321)
(2.321+0j)
>>>tuple("abcdefg")
('a', 'b', 'c', 'd', 'e', 'f', 'g')
>>>list("abcdefg")
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

coerce (object1, object2) 也是一个内置函数。coerce 函数用两个对象作为参数，测试这些对象是否能用通常的对象表示，并在同类型的元组里返回两个值。如果它不可能把值表示成通常的类型，coerce 返回 None 对象。浮点数和整数都能用浮点数表示：

```
>>>coerce(3.1415, 6)
(3.1415, 6.0)
```

然而，字符串和浮点数不能表示成这样的类型：

```
>>>results = coerce("a", 2.3)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: number coercion failed
```

#### 2.4.4 内置文件函数

仅有—个内置文件函数。该函数是 open。open 函数返回一个可写可读的文件对象。open 函数的语法如下：

```
open(filepath [, mode [, buffer]])
```

filepath 是到打开文件的指定的平台路径和文件名。

提供给 open 函数的 mode 符号决定文件打开是为读、写、追加还是读和写。这个模式也说明了它是文本文件还是二进制文件。文本模式意指换行转换平台，二进制模式仅指二进制数据，因此没有使用换行转换，而仅使用 Unicode 字符的低序字节。为了在 Jython 里写二进制数据，二进制模式必须明白地说明，否则 Unicode 字符用缺省 java codec 来读写文件。注意在 open 函数里模式设置是可选择的，当没有指定模式时，使用读文本的缺省模式。在模式变量中使用的符号如下：

r = 读

w = 写

a = 当加到“r”、“w”或“a”后（例如“r+”、“w+”或“a+”）表示读和写

b = 二进制模式

给 open 函数的可选 buffer 变量现在忽略。

#### 2.4.5 序列函数

除了字符串特定的函数外，这些内置函数也需要列表类型作为参数或返回一个列表。

这些给 PyString 序列的特殊函数是 intern、chr、unichr 和 unicode，它们在表 2-13 中进行了描述。

表 2-13 特殊的 PyString 函数

函 数	描 述
intern (string)	把特殊的字符串放到生存期长的字符串 PyStringMap 里
chr (integer)	对 integer <= 65535，chr 返回有一定整数长度的字符（长度为 1 的 PyString）
ord (character)	这个函数与 chr (integer) 相反（见上面的列表项）。对字符 c，chr (ord (c)) == c
unichr (integer)	与 chr 一样，chr 和 unichr 这两个方法与 Cpython 是可兼容的
unicode (string [, encoding [, errors]])	这个函数使用指定的编码创建一个新的 PyString 对象。在编码文件夹的 Jython Lib 目录可找到可用的编码

对任何 Jython 序列对象起作用的函数描述在表 2-14 中。

表 2-14 内置序列函数

函数	描述
max (sequence)	返回序列里最大的数
min (sequence)	返回序列里最小的数
slice ([start, ] stop[, step])	Slice 函数是序列分片语法的同义词，主要被扩展名使用
range([start, ] stop[, step])	range 和 xrange 函数用来生成列表。这些函数形式是一样的，但它们的实现是不同的。range 函数构建并返回整数列表，而 xrange 返回提供所需要的整数的 xrange 对象。对大范围来说，xrange 是有优点的。作用域内的所有变量必须是整数 (PyInteger 或 PyLong)。在指定的 start 数或 0 处开始记数，直到指定的 stop 整数，但不包括 stop 整数。如果提供了可选的 step 变量，它就作为增量数。下面的例子最好地阐述了 range 和 xrange:
	<pre>&gt;&gt;&gt; range (4) [0, 1, 2, 3] &gt;&gt;&gt; range (3, 6) [3, 4, 5] &gt;&gt;&gt; range (2, 10, 2) [2, 4, 6, 8] &gt;&gt;&gt; xrange (4) (0, 1, 2, 3) &gt;&gt;&gt; type (xrange (4)) # 检查从 xrange 返回的类型 &lt;jclass org.python.core.PyXRange at 7991538&gt; &gt;&gt;&gt; type (range (4)) # 检查从 xrange 返回的类型 &gt;&gt;&gt; type (range (4)) &lt;jclass org.python.core.PyList at 7837392&gt; &gt;&gt;&gt; range (10, 0, -1) # 使用 range 向后记数 backwards [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]</pre>

#### 2.4.6 动态语言函数

Jython 的动态语言函数允许用象 compile、apply、execfile 和 eval 这样的函数进行动态代码生成、编译和执行。它们在这里没说明，因为如果本书在这点上做简单的处理，就有可能引起混淆。

#### 2.4.7 属性工具

Jython 有丰富的内置函数与 Jython 类和它们的属性一起作用。特别是对属性来说，有 hasattr、delattr、getattr 和 setattr 函数。hasattr、delattr、getattr 函数的参数都是对象、属性（例如 hasattr (object, attribute)）。setattr 需要另外两个（第三个）参数，它指定属性将设置的值。

Jython 类属性一般用象 Java 一样的点标记法来访问。如果对象 A 包含属性 b，访问属性 b 的语法是 A.b：

```
>>> class A:
...     b = 10
...
>>> A.b
10
```

Jython 的属性工具作为与对象属性一起作用的替代的方式存在，但它们比传统的点标记法相比，明显有较少的共性。本节粗略地讲述了这些内置函数关于两个对象的使用。对象中的第

一个是空 Jython 类，但第二个是一个除了一个公有实例变量——var1 以外什么也没有的 Java 类。用 Java 类捕获就是它们不像 Jython 类一样灵活。专用属性不能从 Java 类中增加或删除。

从创建和空 Jython 类开始：

```
>>>class JythonClass:  
...     pass
```

其次，构造新类的一个实例，因而我们有一个对象来使用 hasattr、setattr 和 getattr：

```
>>>jyInstance = JythonClass()
```

hasattr 函数检查对象里是否含有属性。对象和属性名也是 hasattr 函数的参数。为了检查 jyInstance 实例里的属性 var1，使用以下语句：

```
>>>hasattr(jyInstance, "var1")  
0
```

你得到的结果是 0 或假。var1 不是当前定义的。

现在假定我们想定义或“设置”它。当分派任务来处理时，也有 setattr 函数。setattr 函数需要三个参数：对象、属性名和绑定到属性名的值。如下设置 var1 的 jyInstance 属性：

```
>>>setattr(jyInstance, "var1", "A new value for var1")
```

现在用 hasattr 来检查 jyInstance 里的“var1”：

```
>>>hasattr(jyInstance, "var1")  
1
```

结果为 1 或真，证实了有 var1。

既然 jyInstance 有属性，我们可以用 getattr 函数来检索它的值。getattr 函数需要对象和属性名作为参数：

```
>>>getattr(jyInstance, "var1")  
'A new value for var1'
```

getattr() 函数也为类里的访问成员和模块里的对象工作：

```
>>> import os  
>>> getattr(os, "pathsep") # get a variable from a module  
';'  
>>> getattr(os, "getcwd")() # call a function within a module  
'C:\\WINDOWS\\Desktop'
```

最后，属性可用 delattr 函数删除：

```
>>>delattr(jyInstance, "var1")  
>>>hasattr(jyInstance, "var1")  
0
```

然而 Java 类不同于 Jython 类。忘掉上述的 Jython 例子，而用 Java 类延展一个新的例子。假定如下定义 Java 类：

```
public class JavaClass {
```

```

public String var1;
public void JavaClass() {}
}

```

也假定 JavaClass 用 javac 编译并且它是在你启动 Jython 的当前目录。

你可用 JavaClass 来显示 Jython 的内置属性函数是如何与 Java 对象一起作用的。通过引入 JavaClass 和创建一个实例来启动：

```

>>>import JavaClass
>>>javaInstance = JavaClass()

```

现在用 hasattr 来测试 JavaClass 里的属性 var1 的存在：

```

>>>hasattr(javaInstance, "var1")
1

```

既然 var1 被证实存在，用 setattr 多次改变它，并用 getattr 测试它的新值：

```

>>>setattr(javaInstance, "var1", "A new value for var1")
>>>getattr(javaInstance, "var1") # check out the new value
'A new value for var1'
>>>setattr(javaInstance, "var1", "Yet another value")
>>>getattr(javaInstance, "var1")
'Yet another value'

```

到目前为止，Java 对象像 Jython 对象一样运作。当你试着删除一个属性或增加一个属性时，差别出现了。首先，你能删除 var1 吗？

```

>>>delattr(javaInstance, "var1")
Traceback (innermost last):
File "<console>", line 1, in ?
TypeError: can't delete attr from java instance: var1

```

结果是不能。你能增加一个专用属性吗？

```

>>># Can we add another arbitrary attribute
>>>setattr(ja, "var2", "Value for var2")
Traceback (innermost last):
File "<console>", line 1, in ?
TypeError: can't set arbitrary attribute in java instance: var2

```

还是不能。

用一不同的 Java 类：java.util.Vector 继续检测。首先，如下导入和创建 Vector 的一个实例：

```

>>>import java
>>>v = java.util.Vector() # import a well-known java class

```

为属性 toString 检查 Vector 实例：

```

>>>hasattr(v, "toString") # in case you forgot
1

```

对象检查包括测试一个实例是不是指定类的一部分。内置 isinstance 函数为被提供作为变量的实例和类决定实例是不是指定类的一部分。

```
>>> isinstance(v, java.util.Vector)
1
```

如果结果是 0，就说明 v 不是 java.util.Vector 的实例。

对于实例是不是所提供的类的一个实例，从程序清单 2-6 知道内置函数 `isinstance` 返回 1 或 0 (true 或 false)。语法是 `isinstance (instance, class)`。

## 2.4.8 函数工具

函数程序设计的元素显示在 Jython 的内置函数中。对列表的函数操作是 `filter`、`map`、`reduce` 和 `zip`。所有这些内置函数需要另一个函数作为它的参数，所以这些讨论留到 Jython 函数那章。

## 2.5 杂类函数

同样有用的一些内置函数不像其他函数那样容易归类。因为它们没有特殊的关系，最好仅提供它们的函数列表和定义，也包括有利于说明的例子，如表 2-15 所示。

表 2-15 杂类函数

函数	描述
<code>cmp (x, y)</code>	比较函数 <code>cmp</code> ，需要两个参数来比较。根据是不是 <code>x &lt; y</code> 、 <code>x == y</code> 或 <code>x &gt; y</code> 分别返回 -1、0 或 1
<code>len (object)</code>	<code>len</code> 返回一个序列或映射对象的长度。如果它不在映射对象上运作，那它包括序列函数
<code>repr (object)</code>	<code>repr</code> 返回对重建对象有用的对象的字符串表示，假如你需要一个 <code>java.lang.Vector</code> 对象的字符串表示， <code>repr</code> 函数提供下面简单例子所显示的： <pre>&gt;&gt;&gt;&gt;&gt; import java &gt;&gt;&gt; v = java.util.Vector() &gt;&gt;&gt; repr(v) [1] &gt;&gt;&gt; v.add('A string') 1 &gt;&gt;&gt; v.add(4) 1 &gt;&gt;&gt; v.add([1,2,3,4,5]) 1 &gt;&gt;&gt; repr(v) ['A string', 4, [1, 2, 3, 4, 5]]'</pre>
<code>str (object)</code>	<code>Str</code> 返回描述指定的 <code>object</code> 的字符串。这与 <code>repr</code> 不同，这个字符串是一个描述或标记，而不是 <code>repr ()</code> 所希望的数据的严格表示
<code>reload (module)</code>	这个函数是自明的，但它的用途不是马上就明显的。这个函数允许你用交互模式运行，把一个编辑改变成你所需要的模块，返回到交互模式来重载已改变的模块
<code>raw_input ([prompt])</code>	<code>raw_input</code> 函数从标准输入读取字符串，截去尾部换行 <pre>&gt;&gt;&gt;&gt;&gt; name = raw_input("Enter your name: ") Enter your name: Robert &gt;&gt;&gt; print name Robert</pre>
<code>input ([prompt])</code>	除了给出 <code>prompt</code> 的内容外，其他的与 <code>raw_input</code> 相同 <pre>&gt;&gt;&gt; input("Enter a Jython expression: ") Enter a Jython expression: 2+3 5 &lt;-- the evaluated result of 2+3</pre>
<code>hash (object)</code>	相同值的两个对象有相同的散列值。这个散列值用内置 <code>hash</code> 函数检索

## 2.6 Jython 数据类型和 Java 数据类型的比较

Jython 数据类型和 Java 数据类型的区别是什么？Java 是严格的类型语言，其数据存储在有确定类型的对象里。下面的 Java 语句清晰地创建一个具有自知值 “I'm a string”的 java.lang.String 类型的 myString 名的对象：

```
String myString = "I'm a string";
```

Jython 也是一种强类型的语言，数据通过特定类型的对象来表示。较大的区别是 Jython 是动态地被类型化的。不像前述的小 Java snippet 程序，Jython 在它的代码里不使用显式的类型信息。因此在编译时信息是不可用的。相反，Jython 根据在运行时的赋值来决定类型。

在类型化里这个区别意味着什么？警惕过于简化类型系统是有益的，但是 Java 显式的类型化意味着 Java 编译器不能在编译时对变量进行某种假设。这要虑及许多优化和编译时的类型安全检查。对 Jython 的动态类型来说，它意味着灵活、高级数据对象，该数据对象要考虑简洁代码、增加编程逻辑的清晰并增加程序员的生产率。简而言之，Jython 数据对象以牺牲编译时的优化来换取更大的灵活性。

## 2.7 Java 类型

Jython 的最主要优点就是不用修改就能使用 Java 类。这也产生了从 Java 类型转换成 Jython 类型的问题。Java 有许多类型。方法、构造函数和变量使用类型。另一方面，Jython 是没有基本类型的，有不是 Java 里本原数据对象，并动态决定的类型。这也意味着 Jython 的类型需要转换成适合于方法变量的 Java 类型，并且从适合于方法的 Java 类型转换返回值。Jython 按照表 2-16 自动转换类型。

表 2-16 Java 和 Jython 之间的类型转换

Java	Jython
Char	PyString，长度为 1
Byte	PyInteger
Short	PyInteger
Int	PyInteger
Long	PyInteger
Float	PyFloat
Double	PyFloat
java.lang.String	PyString
byte []	PyString
char []	PyString
PyObject	PyObject
A java class	PyJavaClass，表示给定的 Java 类
An instance of a Java class	标志在 instance._class_ 里的 Java 类的 PyInstance，例如： <pre>&gt;&gt;&gt; from java.lang import Double &gt;&gt;&gt; d = Double(2.0) &gt;&gt;&gt; d._class_ &lt;jclass java.lang.Double at 3291150&gt;</pre>

(续)

Java	Jython
Boolean	PyInteger (当从 Jython PyInteger 转换成 Java Boolean 时, 0 == false 和 nonzero == true。当从 Java Boolean 转换成 Jython 类型时, true == 1 和 false == 0。)
java.lang.Object	java.lang.Object

这些类型相容地转换。当特定 Jython 对象的相应 Java 类出现在方法特征符时, 特定 Jython 对象就起作用, 并且返回的 Java 类型转换成它相应的 Jython 类。

Java 方法参数或返回值可以是一个 Java 类或 Java 类数组的实例。如果一个方法指定 Java 类 A 作为参数类型, 你就可以实例化和提供 A 的一个实例、A 的 Java 派生类的一个实例或 A 的一个 Jython 派生类来满足需要。如果 A 的实例是 Java 方法的返回类型, Jython 则把它转换成 A 或返回 A 的任何派生类的 PyInstance。

如果在 Java 方法特征符里希望有 Java 对象数组, 你可提供包含从特定对象的类或派生类创建的对象数组。你可使用 jarray 模块来创建 Java 数组。jarray 模块包含两个函数: array 和 zeros。array 函数需要两个变量: 第一个是序列, 第二个是表示数组类型的代码字母。数组的长度等于提供的序列的长度。序列可以是字符串、列表或合适的元组。为了创建一个字符数组(在 Java 中是 char []), 首先引入 jarray 模块, 再提供一个字符序列作为第一个参数, 然后键入代码 c 作为第二个参数, 如下:

```
>>> import jarray
>>> myJavaArray = jarray.array("abcdefg", "c")
>>> myJavaArray
array(['a', 'b', 'c', 'd', 'e', 'f', 'g'], char)
>>> myJavaArray = jarray.array(['a','b','c','d'], "c")
>>> myJavaArray
array(['a', 'b', 'c', 'd'], char)
```

数组类型的 Java “类”也可以用来在 jarray.array 函数的第二个变量里设计数组的类型。对 Java 基本数据类型来说, 这就是指封装了基本类型的 java.lang 类。为了从 Jython PyList 获得 int [], 可使用如下语句:

```
>>> import jarray
>>> import java # Required to see the java.lang.Integer class
>>> myIntArray = jarray.array([1,2,3,4,5,6,7,8,9], java.lang.Integer)
array([1, 2, 3, 4, 5, 6, 7, 8, 9], java.lang.Integer)
>>> i = java.lang.Integer(1)
```

为了创建一个特定长度和类型的数组, 仅 0 和 null 值使用 jarray.zeros 函数。jarray.zeros 函数也需要两个变量: 第一个变量是数组的长度, 第二个是用来设置数组类型的类型代码或类。下面是使用 jarray.zeros 函数来创建长度为十的 java byte []:

```
>>> import jarray
>>> byteArray = jarray.zeros(10, "b")
>>> byteArray
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], byte)
```

所有类型代码、与它们相关的 Java 类型以及这些类型的 java.lang 包的类列于表 2-17 中。

表 2-17 数组类型代码、类型和类

类型代码	Java 类型	类型封装器类
z	Boolean	java.lang.Boolean
c	char	java.lang.Char
b	byte	java.lang.Byte
h	short	java.lang.Short
i	int	java.lang.Integer
l	long	java.lang.long
f	float	java.lang.float
d	double	java.lang.double

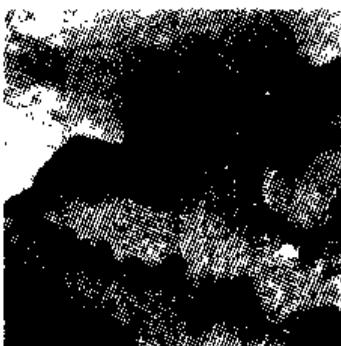
对没有类型代码的类的数组，可使用类来设置数组类型。例如，可用下列语句来创建类 java.util.Date 的 Data []：

```
>>> import java
>>> import jarray
>>> myDateArray = jarray.zeros(5, java.util.Date)
>>> myDateArray
array([None, None, None, None, None], java.util.Date)
```

当因为类型转换的限制在调用特定的 Java 方法特征符和重载方法遇到困难时，最好的解决方法是创建 Java 类型的对象封装器的实例。如果 PyInteger 不能适当地转换来调用需要 short 类型的方法特征符，则通过先创建 short 类型的对象封装器 java.lang.Short 来强制正确的识别。假如方法 A 用一个特征符 A (short s)、另一个 A (int i) 和调用 short 特征符来重载是不能运行的。下面的例子说明怎样来改正：

```
>>> import java
>>> myPyInt = 5
>>> myJavaShort = java.lang.Short(myPyInt)
>>> ## Here you could call the hypothetical method "A(myJavaShort)"
```





## 第 3 章 错误和异常

本章详细地说明了 Jython 的内置异常和相关的机制。异常对象主题也引入了 try/except 语句、raise 语句、traceback 对象、assert 语句和 \_debug\_ 变量的最近研究，以及最近增加的警告框架。

### 3.1 Jython 异常

Jython 的异常定义在类 org.python.core.exceptions 里。这个类的源代码包含了图 3-1 所示的异常层次。每个异常的相应消息字符串被包含在本图的类名里。

你不能用一个数除以 0，不能访问一个不存在的名字，也不能使用比你机器所具有的更大的存储。无论什么时候破坏这些规则，Jython 解释器通过产生异常来强制执行这些规则。Jython version 2.1 用灰色来显示增加的警告。你在图 3-1 看到的警告是异常，但由于警告框架介绍了一些独特的东西，所以将在本章稍后讨论。

研究 Jython 异常的最简单的方法是在 Jython 的交互解释时设法创造一个异常的环境。下面是因为错误而产生一些错误和异常的例子。你不能访问没有定义的变量：

```
>>> print x
Traceback (innermost last):
  File "<console>", line 1, in ?
NameError: x
```

你不能访问超过列表范围的列表索引：

```
>>> L = []
>>> print L[1]
Traceback (innermost last):
  File "<console>", line 1, in ?
IndexError: index out of range: 1
```

你不能调用一个不存在的方法：

```
>>> L.somemethod() # Calling a non-existent method name is an error
Traceback (innermost last):
  File "<console>", line 1, in ?
AttributeError: 'list' object has no attribute 'somemethod'
```

在 Jython 缩排文件，如下面的循环所示：

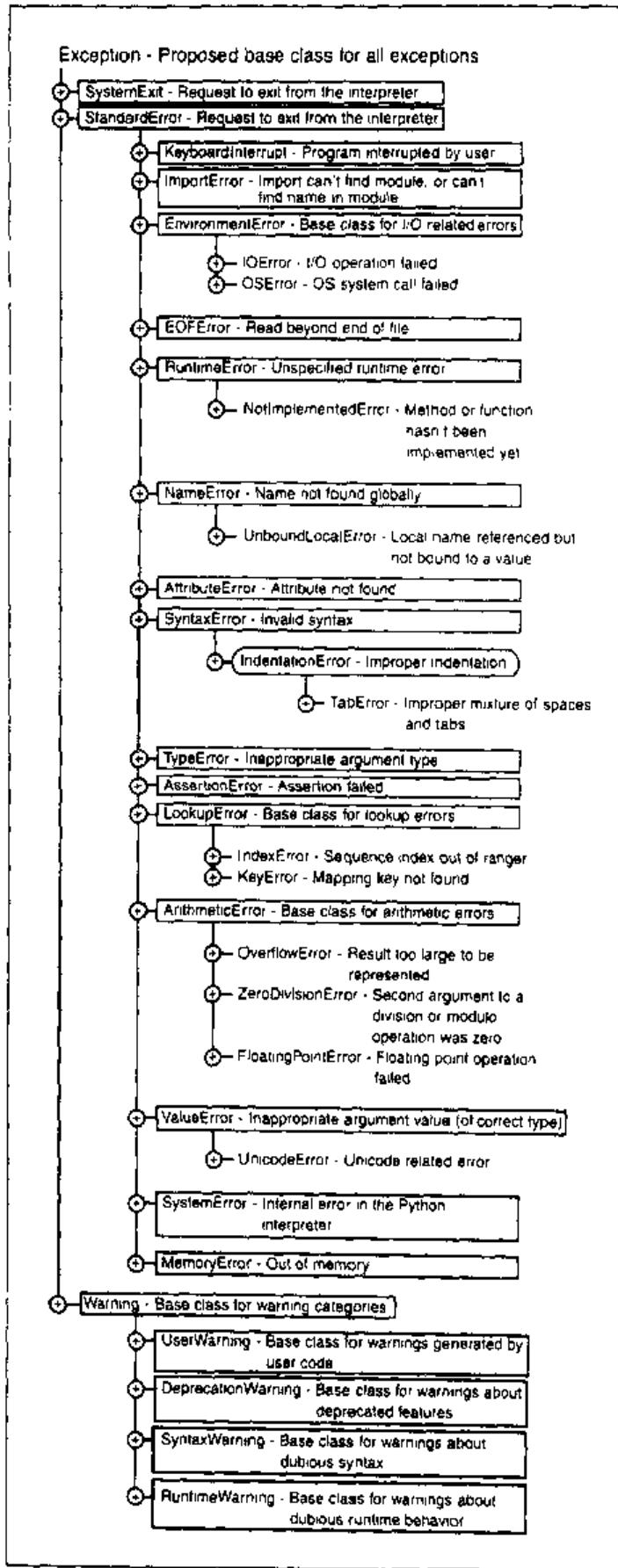


图 3-1 Python 的异常层次

```
>>>for x in range(10):
...     i = x    # This line is indented one tab
...     print i # This line is indented 5 spaces
Traceback (innermost last):
  (no code object) at line 0
  File "<console>", line 3
      print i
          ^
SyntaxError: inconsistent dedent
```

当一个引入的 Java 类在 Java 代码里抛出异常时，Java 异常也可能在 Jython 里产生。在 Jython 里异常像 Java 异常一样进行了说明。程序清单 3-1 显示了一个简单的例子：

程序清单 3-1 Jython 里的 Java 异常

```
//file: Except.java
public class Except {
    public Except() {}
    public void test() throws IllegalAccessException {
        throw new IllegalAccessException("Just testing");
    }
}

>>>#Using the above Java class in Jython
>>> import Except
>>> e = Except()
>>> e.test()
Traceback (innermost last):
  File "<console>", line 1, in ?
  java.lang.IllegalAccessException: just testing
      at Except.test(Except.java:4)
...
java.lang.IllegalAccessException: java.lang.IllegalAccessException:
Just testing
```

## 3.2 异常处理

在 Java 术语里，当错误发生时，异常被抛出。抛出的异常一直传播直到某些东西“捕获”它。在 Jython 行话里，异常是“被产生”，而不是“被抛出”。当 Jython 异常被产生时，程序流转去搜寻异常处理器。搜寻不仅仅包括异常发生的作用域，还通过堆栈层传播检索合适的 except 子句。如果合适的 except 子句不存在，就中断程序，并显示带有 traceback 信息的异常。

### Jython 2.1 和 sys.excepthook

在 Jython 2.1 里新特性是 sys.excepthook，你可设计 sys.excepthook 给可调用对象，虽然这个特征在写这本书时还不在 Jython 里，但当你读这本书时它可能已存在了。这就意味着当异常没有找到任何适当的 except 字句来处理它时，它在终止程序前被传递给 sys.excepthook。那样，无论你赋任何值，在程序终止前 sys.excepthook 能利用异常做你想做的事。这在程序终止前关闭和刷新资源是有价值的。

在 Jython 里 try 语句有两种格式。第一种是 try/except/else，它有如下语法：

```
"try:" code-block
"except" [expression [, "target"]]: "code-block"
["else:" code-block]
```

程序清单 3-2 显示了这种语法的一个例子。

#### 程序清单 3-2 try/except/else 语法

```
>>> try:
...     print a # a is not defined, so is a NameError
... except SyntaxError:
...     print "I caught a SyntaxError"
... except NameError:
...     print "I caught a NameError"
... except:
...     print "A plain except catches ANY type of exception"
... else:
...     print "no exception was raised in the try block
...
I caught a NameError
```

程序清单 3-2 中的执行流从 try 里的代码程序块开始。如果异常被产生，except 程序块被搜寻以处理所产生的特定类型的异常，或没有特定异常类型的 except（它是普通的，能处理任何异常）。如果在 try 程序块里没有任何异常产生，继续执行 else 程序块。Try/except 的 except 部分在代码程序块前有两个可选择的元素。第一个与其相关的代码程序块是要处理的异常类，程序清单 3-2 为除了最后一个 except 子句以外的所有子句指定了异常类。程序清单 3-2 没有使用第二个可选元素——target 元素。如果异常的参数值被提供的话，target 元素可获得它。看一看 raise 语句显示的异常的“参数值”是什么：

```
>>> try:
...     raise SyntaxError, "Bad Syntax"
... except SyntaxError, target:
...     print target
...
Bad Syntax
```

字符串 Bad Syntax 是当产生异常时给 SyntaxError 类的参数。因此这也是变量 target 被限制到了 except 子句。

图 3-1 不仅显示了 Jython 的异常和它们的消息字符串，而且也显示了异常的层次。为了处理 OverflowError、ZeroDivisionError 或 FloatingPointError，你仅需要处理它们的超类——Arithmeti-Error。处理 StandardError 异常会覆盖大多数错误，处理基类 Exception 将处理除了用户自定义的没有从 Exception 继承的以外所有异常。程序清单 3-3 显示了异常层次怎样允许你通过处理一组异常的公共基类来处理一组异常。

#### 程序清单 3-3 Except 子句里的基类

```
>>> try:
...     1/0
... except ArithmeticError, e:
```

```

...     print "Handled ArithmeticError-", e
...
Handled ArithmeticError- integer division or modulo
>>> L = [1,2,3]
>>> try:
...     print L[10] # this index does not exist
... except LookupError, e:
...     print "Handling LookupError-", e
...
Handling LookupError- index out of range: 10

```

第二种格式是 try/finally。执行 try 代码程序块，并且不管是否产生异常，finally 代码程序块会被执行。它的语法如下：

```

"try:" code-block
"finally:" code-block

```

程序清单 3-4 显示了一个不管 try 程序块发生了什么，用 finally 程序块来关闭文件对象的例子。

#### 程序清单 3-4 使用 try/finally 来确保文件被关闭

```

>>> fileA = open("datafile", "w")
>>> try:
...     fileB = open("olddata", "r")
...     print >>fileA, fileB.read()
... finally:
...     fileA.close()
...     if "fileB" in vars():
...         fileB.close()
...

```

在程序清单 3-4 中另一个有用的技巧是在设法关闭文件之前测试文件是否在 vars() 中。如果 open() 函数失败，就没有 fileB 要关闭，因此测试存在 vars() 就会通过关闭一个不存在的文件对象阻止 finally 程序块里的第二个异常产生。

程序清单 3-5 示范了嵌套 try/except 语句。在此程序清单里，只有最外层的 try 语句有合适的 except 表达式，因此该外层是处理异常的。对于程序清单 3-5 另一个要注意的是使用了 sys.exit() 函数。你在使用这个函数之前必须先引入 sys 模块，实际上所做的是产生 SystemExit 异常。这也就是说如果你在 except 子句里捕获了 SystemExit 异常，系统就不退出。

#### 程序清单 3-5 嵌套 try 语句

```

>>> import sys
>>>
>>> try:
...     try:
...         try:
...             sys.exit() # This raises the 'SystemExit' exception
...         except ValueError:
...

```

```

...
    pass
...     except SyntaxError:
...
    pass
... except SystemExit:
...
    print "SystemExit exception caught- not exiting."
...
SystemExit exception caught- not exiting.

```

程序清单 3-1 给出了抛出 `java.lang.IllegalAccessException` 异常的 Java 程序。这产生了在 Jython 里特定的 Java 异常是怎么处理（捕获）的问题。程序清单 3-6 是程序清单 3-1 列出的 Jython 代码的修订版本。Try/except 已增加了一个 except 子句，特定用来处理 Java 类中抛出的 `java.lang.IllegalAccessException` 异常。为了处理 Java 异常，你也可以使用一个空的 except 子句以便它是通用的、可以处理任何异常的，或者在异常前使用 `import java`，并在用来处理异常和它的派生类的 except 子句里提供特定的异常包和类。

程序清单 3-6 在 Jython 里处理的 Java 异常

```

//file: Except.java
public class Except {
    public Except() {}
    public void test() throws IllegalAccessException {
        throw new IllegalAccessException("Just testing");
    }
}
>>> #Using the above Java class in Jython
>>> import Except # import the exception-throwing Java class
>>> import java
>>> try:
...     e = Except()
...     e.test()
... except java.lang.IllegalAccessException:
...     print "Caught IllegalAccessException"
...
Caught IllegalAccessException

```

### 3.3 raise 语句

在 Jython 语言里，我们有时候可能希望明确地得到在通常情况下不会出现的异常。为了达到这个目的，可使用 `raise` 语句。例如，用 `raise` 语句来检测参数类型是非常有效的。Jython 在 2.1 或早期版本中并没有为检测参数类型提供编译时间。这就意味着如果参数是一个确定的类型，代码必须在运行时测试它，并对不匹配的类型使用 `raise` 语句。程序清单 3-7 正是那样做的。函数 `listComplement` 需要两个列表或元组作为参数。在任何其他处理之前进行参数测试，并且如果任何一个参数不是列表或元组就产生 `TypeError` 异常。

#### 接口和 Jython

Jython 2.1 和早期版本没有参数、协议、接口或类型检测。Python 的“类型”特

别兴趣组提出了与这些主题相关的建议。后来的思想、研究和艰苦的工作产生了一些 Python 提高建议，这些提高建议可能在将来新推出的版本里改变上述现象，可以在 Jython 的 WWW 站点或与这本书相关的 WWW 站点查看。

程序清单 3-7 确定了两个列表的补集。与列表 2(L2) 相关的列表 1(L1) 的补集是所有在 L2 而不在 L1 的元素。最佳解释是列表 1(L1) 被转换成字典关键字以便使用快速的 has\_key 函数来代替序列成员测试 in 函数。在程序清单 3-7 中，引入了类型模块比较参数类型与 ListComplement 函数 (TupleType 和 ListType) 所需要的类型。

程序清单 3-7 Jython 的 raise 语句

---

```
# file sets.py

# import canonical types to compare with parameter types
from types import ListType, TupleType

def listComplement(S1, S2):
    if not (type(S1) == ListType or type(S1) == TupleType and
            type(S2) == ListType or type(S2) == TupleType):
        raise TypeError, "Only lists and tuples are supported."
    D = {}
    for x in S1:
        D[x] = 1 # Convert S1 to dictionary keys.
    complement = [] # An empty list to hold results.
    for item in S2:
        if not D.has_key(item):
            complement.append(item)
    return complement

# Test the function with lists
list1 = range(0, 100, 3)
list2 = range(0, 100, 7)
resultSet = listComplement(list1, list2)
print 'Complement of list1 relative to list2: ', resultSet

# Now Test the function with a list and a numeric type
notalist = 5

resultSet = listComplement(notalist, list2)
```

---

在命令行运行 jython sets.py 的结果：

```
Complement of list1 relative to list2: [7, 14, 28, 35, 49, 56, 70, 77, 81,
98]
Traceback (innermost last):
  File "sets.py", line 26, in ?
  File "sets.py", line 8, in listComplement
TypeError: Only lists and tuples are supported.
```

程序清单 3-7 比平均交互式例子长一些，最好把它的内容存到用命令行 jython sets.py 能运行的文件里。运行 sets.py 脚本的结果显示了 listComplement 函数因不匹配的参数产生 TypeError。

在程序清单 3-7 里 raise 语句使用两个表达式，但在 raise 语句的语法里实际上有三个可选的表达式。raise 语句的语法如下：

```
"raise" [expression ["," expression ["," expression]]]
```

替换 expression 所表示的名字，读起来就如下所示：

```
"raise" [type ["," value ["," traceback]]]
```

type 可以是异常、字符串或产生的异常类的实例的实类。程序清单 3-7 使用了实例 TypeError 类来产生错误。然而，它也可能使用 TypeError 的实例来产生与程序清单 3-8 所列出的相同的异常。在 raise 语句里使用字符串作为类型产生基于字符串的异常。基于字符串的异常是合法的，但不鼓励支持类或实例异常。

### 程序清单 3-8 指定产生的异常类型的不同方法

```
>>> # The first way is the actual class.
>>> # Use the built-in function "type()" to confirm it is a class.
>>> type(TypeError)
<jclass org.python.core.PyClass at 7907289>
>>> raise TypeError
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError:
>>>
>>> # The second way is a string.
>>> raise "A String-based exception"
Traceback (innermost last):
  File "<console>", line 1, in ?
A String-based exception
>>>
>>> # The third way is an instance.
>>> Err = TypeError()    # Create the instance.
>>> type(Err)           # Confirm it is an instance.
<jclass org.python.core.PyInstance at 2735779>
>>> raise Err          # Raise the instance
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError:
```

Raise 语句的值或第二个表达式是异常类的构造函数参数。回顾程序清单 3-7，我们看到类 TypeError 是用最合适的 lists 和 tuples 支持的值实例化的。这就意味着下列行：

```
>>> Err = TypeError("Only lists and tuples are supported")
>>> raise Err
```

与这行相同：

```
>>> raise TypeError, "Only lists and tuples are supported"
```

当 type (raise 语句里的第一个表达式) 是一个实例时属于一个特殊情形。value 表达式是构造函数参数，如果类已经被构造 (一个实例)，值已经无关紧要了。既然这样，规则就是如果

异常的类型是实例，则值表达式必须是 None。

异常的构造函数得到的参数的个数依赖于值表达式是否是元组。列表、字典、数字或除了元组以外任何其他的仅作为一个参数被传递。另一方面，元组作为一个参数列表被传递。这就意味着有 len(T) 作为元组 T 的参数。程序清单 3-9 导入只打印出传递到它的构造函数的参数个数的 MyException 类。程序清单 3-9 包含了一个新的引入语句：from MyException import exceptionA。从模块 MyException 引入 exceptionA 类。因有这些引入语法，exceptionA 成为局限于命名空间的名字，它是用在程序清单 3-9 的异常类。

### 程序清单 3-9 异常参数

---

```
>>> from MyException import exceptionA
>>> raise exceptionA, (1,2,3,4,5,6,7,8,9)
Number of parameters = 9    <-- 9 parameters, all PyInteger objects.
Traceback (innermost last):
  File "<console>", line 1, in ?
exceptionA: <MyException.exceptionA instance at 1689053>
>>>
>>> # Now try a list instead of a tuple
>>> raise exceptionA, [1,2,3,4,5,6,7,8,9]
Number of parameters = 1    <-- Only one parameter, a PyList object.
Traceback (innermost last):
  File "<console>", line 1, in ?
exceptionA: <MyException.exceptionA instance at 1075173>

# MyException.py source

class exceptionA(Exception):
    """An exception class for testing constructor parameters"""

    def __init__(self, *args): # '*'args' catches all remaining arguments
        print "Number of parameters =", len(args)
```

---

在程序清单 3-9 里的 `__init__` 方法有一个虚参：`*args`。星号允许这个方法捕获所有剩下的自变量，这是一种允许参数个数检测能正确运行的机制。`*` 参数语法的详细讨论放在第 4 章“用户定义的函数和变量的作用域”中。

如果可选的第三个表达式在 `raise` 语句里，它必须是一个 `traceback` 对象。看下一节关于 `traceback` 更多的信息。

如果没有提供任何参数给 `raise` 语句该怎么办？这产生最后在同样作用域里产生的异常并在程序清单 3-10 中证明。

### 程序清单 3-10 没有任何表达式的 raise 语句

---

```
>>> # First, define a function so the two exception can
>>> # be within the same local scope
>>> def test():
...     try:
...         raise SyntaxError, "Bad Syntax"
...     except:
...         pass
```

---

```

...     print "passed first raise"
...     raise
...
>>> # Call the function to see how the second 'raise' works
>>> test()
passed first raise
Traceback (innermost last):
  File "<console>", line 1, in ?
  File "<console>", line 3, in test
SyntaxError: Bad Syntax

```

---

### 3.4 traceback

当异常出现，你可看见的消息以 traceback 开始。同样，当产生异常时，可选的第三个表达式必须赋一个 traceback 对象。traceback 对象有助于分步进入产生异常的点。当你在异常处理器 (try 后的 except 或 finally) 时，traceback 对象可用 sys.exc\_info() 访问。从 sys.exc\_info() 返回三项，它们可解释成 type、value 和 traceback。这与 raise 语句后的表达式顺序相似。traceback 模块也存在并在与 tracebacks 一起运行时有帮助。

程序清单 3-11 通过产生异常把这些项联系起来，使用 sys.exc\_info() 方法得到 traceback 信息，于是使用 traceback 模块的 exc\_lineno 和 extract\_tb 函数来打印异常产生的行数，并打印整个 traceback。函数 sys.exc\_info() 实际上返回长度为三的元组。Jython 允许你检索有值的元组，因此该行提供捕获每个元组元素的三个标志符。

程序清单 3-11 使用 traceback

---

```

# file: raise.py
import sys
import traceback

try:
    raise SyntaxError, 'Bad syntax'
except:
    exc_type, exc_value, exc_tb = sys.exc_info()
    print "The type of exception is:", exc_type
    print "The value supplied to the exception was", exc_value
    print "The type of exc_tb is:", type(exc_tb)
    print "The line where the exception was raised is",
    print traceback.tb_lineno(exc_tb)
    print "The (filename, line number, function, statement, value) is:"
    print traceback.extract_tb(exc_tb)

The results from running 'jython raise.py' are:
The type of exception is: exceptions.SyntaxError
The value supplied to the exception was Bad syntax
The type of exc_tb is: org.python.core.PyTraceback
The line where the exception was raised is 6
The (filename, line number, function, statement, value) is:
[('raise.py', 6, '?', 'raise SyntaxError, "Bad syntax"')]

```

---

### 3.5 assert语句和\_debug\_变量

产生异常的另一个方法是用 assert 语句。Jython 的 assert 语句适合于在用户指定的某一条件下通过测试来帮助调试程序，并相应地产生一个 AssertionError 异常。assert 语句的语法如下：

```
"assert" expression [", " expression]
```

如果第一个表达式赋值为真，则为空操作；如果为假，则产生一个 AssertionError 异常。第二个可选表达式是传递给 AssertionError 的值。程序清单 3-12 要求用户输入，然后使用 assert 语句来确保用户真正键入了数据。

程序清单 3-12 用 assert 语句测试用户输入

---

```
>>> def getName():
...     name = raw_input("Enter your name: ")
...     assert len(name), "No name entered! program halted"
...     print "You entered the name %s" % name
...
>>> getName()
Enter your name: Robert
You entered the name Robert
>>> getName()
Enter your name:
Traceback (innermost last):
  File "<console>", line 1, in ?
    File "<console>", line 3, in getName
AssertionError: No name entered! program halted
>>>
>>> # assert's behavior changes when __debug__ = 0
>>> __debug__ = 0
>>> # With __debug__ off, the AssertionError is never raised
>>> getName()
Enter your name:
You entered the name
>>>
```

---

由于 assert 语句是被作为一个调试工具来设计的，当内部变量 \_\_debug\_\_ 的值为 1 时，assert 语句才能被求值。在当前已推出的 Jython 版本中，\_\_debug\_\_ 变量的处境有点尴尬。为了达到最优化，在 Jython 启动时本应将 \_\_debug\_\_ 变量设置为只读，并具有一个零开关，这样就能使 \_\_debug\_\_ 变量的值为零，然而，在当前的版本中却没有这样的开关。而为使 assert 语句在所有调试代码中失效，就需要你将 \_\_debug\_\_ 变量赋值为零（此时 \_\_debug\_\_ 变量也就不可能是只读型）。或许这一情况在本书出版时已得到改善，如果真如此，您能在本书相关的网站中获得有关信息。程序清单 3-12 示范了如何给 \_\_debug\_\_ 变量赋值，及当 \_\_debug\_\_ 变量值为 0 时，assert 语句是如何不被赋值的。

### 3.6 警告框架

警告允许 Python 逐步地发展，而不是一旦你升级到 Jython 的一个较新的版本就突然地而临旧代码的异常，警告对于至少一年以上的版本才发出，以保证任何不支持或更改行为不会令人惊奇。在 Jython 2.1 里警告框架是新的。

异常和警告之间的区别是什么？警告不是致命的。你可以收到无数个警告而程序没有被终止。警告中的特定内置函数或语句与异常中的一样不存在。要使警告起作用，必须引入警告模块。虽然在把警告转换成异常的警告模块里有一个警告过滤器，但是警告并不“传播”或需要像异常一样处理。

在警告模块里，有一个 `warn` 函数。这个 `warn` 函数是相对于警告的，而 `raise` 是相对于异常的。`warnings.warn` 函数就是可以发布警告的。程序清单 3-13 定义称做 `oldFunction` 的函数，我们假定这个 `oldFunction` 是不被支持的。为了通报用户这些变化，`oldFunction` 发布一个警告。

### 程序清单 3-13 警告本身不支持的函数

```
# file: warn.py
def oldFunction():
    import warnings
    warnings.warn('"oldFunction" is deprecated. Use "newFunction"')
```

在程序清单 3-13 里的 Jython 代码是称做 `warn.py` 的内容。程序清单 3-14 也使用这个文件。为了测试 `warn.py` 里的警告，从包含文件 `warn.py` 的同一目录启动 Jython 解释器，并键入：

```
>>> import warn
>>> warn.oldFunction()
/home/rbill/warn.py:3: UserWarning: "oldFunction" is deprecated. Use
"new Function"
    warnings.warn('"oldFunction" is deprecated. Use "newFunction"')
```

像这样显示警告，但文件继续执行而不中断。

程序清单 3-13 提供了一个参数给 `warn` 函数——显示的消息。也有第二个可选的参数，它是 `warning` 类型（常常称做 `warning` 类）。这个类应是异常类 `warning` 的派生类，对这个可选“类”参数有四个选择。当然这个通用类是 `warning`——其他 `warnings` 的超类。其他的是 `UserWarning`、`SyntaxWarning` 和 `RuntimeWarning`。当没有提供的类参数时，缺省类是 `UserWarning`。

对 `warnings.warn` 函数也有第三个可选参数，它是 `stacklevel`。`stacklevel` 就像一行多米诺骨牌，在第一个推倒后所有的都连续倾倒。继续模拟，最后倒下的一张多米诺骨牌就等于在我们这里发布 `warning`——`warn` 的模块。`stacklevel` 意味着 `warning` 报告它在模块 `warn`（最后一张多米诺骨牌）里出现。也许不同的模块调用不支持的 `oldFunction`。这个不同的模块就像倒下的从第二张到最后一张的多米诺骨牌。如果你想 `warning` 报告从第二个到最后一个的模块，使用 `stacklevel 2`。从第三个到最后一个的模块是 `stacklevel 3`。记得如果没有任何从第三个到最后一个的对象存在，使用等于 3 的 `stacklevel` 创建一个报告为 `KeyError` 的错误：`_name_`。

如果你想要或需要使 `warnings` 失效该怎么办？`warnings` 模块包含了 `warnings.filterwarnings` 函数，该函数允许你修改 `warnings` 的行为，包括使它们的显示失效。程序清单 3-14 证实了使用 `warnings.filterwarnings` 函数来使从模块 `warn` 里发布的 `warning` 消息失效。

### 程序清单 3-14 过滤 warning

```
# file: warntest.py
import warnings
```

```
warnings.filterwarnings(action = 'ignore',
                        module = 'warn')

import warn
warn.oldFunction()
```

---

运行 jython warntest.py 不产生任何输出。

在程序清单 3-14 里没有任何警告出现是因为过滤器。实际上 `warnings.filter` (`action`, `message`, `category`, `module`, `lineno` 和 `append`) 有六个参数, 但仅 `action` 参数是需要的。`action` 参数可以是 `ignore`、`error`、`always`、`default`、`module` 或 `once`。`message` 参数必须是与你正在测试过滤器的警告消息匹配的字符串。`category` 必须是 `warning` 的派生类。`module` 参数限制过滤器到指定模块的 `warnings`, 并且它必须是字符串。也可以有 `lineno` 和 `append` 参数, 如果提供, `lineno` 参数必须是正整数并限制过滤器到最合适的指定行的 `warnings`。`append` 参数指明你正在创建的过滤器是应被追加到已存在的过滤器的末尾还是插入到过滤器列表的前部。

对 `warnings` 来说, Python 语言描述包含了一个对指定的某一行为的 `-w` 命令行选项。在 Jython 里写时这不会被实现, 因此可检查 Jython 站点或与该书相关的站点来得到关于此的更多信息。

### 3.7 比较 Jython 和 Java

当你在 Jython 和 Java 两种语言之间转换时, 它们之间错误处理的细微差别可能是混淆之源。我经常发现自己混淆了 `throw` 和 `raise`、`catch` 与 `except`。除了在语句名字上的明显差别外, Java 允许一个如下的完整的 `try/catch/finally`:

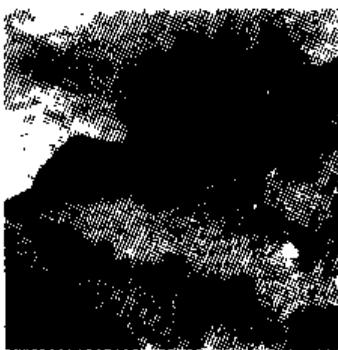
```
try {
    throw new IllegalAccessException("Illegal access message");
} catch (IllegalAccessException e) {
    e.printStackTrace();
} finally {
    //Add some cleanup code here that is always executed
}
```

Jython 仅有 `try/finally` 或 `try/except /else`, 在 Jython 里 `try/except/finally` 是一个语法错误。

用 Jython 的 `assert` 语句提供的简单的声明工具从 Java 1.4 版本后是可用的, 但在早期的 Java 版本是不可用的。

警告框架对 Jython 也是惟一的。对库开发者来说, 在 Java 里没有兼容的工具。是的, 虽然有文档和编译时的错误, 但是当代码进入时不提供任何预先警告。作为能使 Python 自省——核心语言发展的方法的警告框架也在发展, 但它对任何模块都是有用的, 可能对迅速改变的内部工具特别有用。

Java 也有编译时的检查来保证无论抛出什么都可被声明成 `thrown` 或 `caught`。如果方法 A 调用方法 B, 而 B 抛出了一个异常, 则 A 必须捕获它, 或者也把它声明成 `thrown`。Jython 是没有这种需要的。



## 第 4 章 用户定义的函数和变量的作用域

函数是那些能在一个类外面存在的可调用的对象。函数也在类定义中出现，它们在此被对象属性查找转变为方法。然而本章集中于类外面函数的定义。Java 不允许这样的无类实体；而 Jython 是允许的。在 Jython 中，函数是第一类对象，意味着此语言允许它们的动态创建和它们作为参数或返回值传递的能力。Jython 函数是 `Org.PyThon.core.PyFunction` 类的实例（`pyreflectedfunction` 是内置函数）。用内置函数 `type()` 检查 Jython 函数告诉它是 `org.python.core.PyFuncion` 类的一个实例：

```
>>> def myFunction():
...     pass
...
>>> type(myFunction)
<jclass org.python.core.PyFunction at 6879429>
```

本章描述怎样定义一个函数——包括语法、文档和参数。理解 Jython 的名空间对于书写函数是重要的，因此在这里也保证包含着名空间。本章的后面部分谈到这些工具在 Jython 中的可用性，通常它们和函数编程相关。通过包含函数编程的迂回方法来平衡具有一些关于此语言能处理的线索的艰巨的语言描述。在本章中规定，Jython 能首先处理类函数和有效的函数编程。

### 作者注释

本章在把内部类加到 Java 1.1 前将会更深奥难懂，它为第一类函数需要服务，但 Jython 中函数编程的简明和效率仍然是引人注目的。注意对于函数编程的参考并不意味着在整章的例子中展示了在严格的函数语言 Erlang、Haskell、Clean 之中出现的函数编程形式（与 Ada, C/C ++, Java，以及 Pascal 的必要形式是相对立的）。在大多数章节中例子是非常必要的，而函数编程例子出现在最后被注释的地方。

### 4.1 定义函数

Jython 函数语法如下：

```
"def" function_name([parameter_list]):"
    code block
```

函数名称，像所有的标志符一样，可以是任何没有空格的数字、字母和下划线，只要第一字符是字母字符或下划线。名称是区分大小写的，并且使用下划线有特殊意义，在第 5 章“模块和包”中将会详细解释。与函数名字相关的惟一要注意的是偶然重绑定到另外的同名函数，例如内置函数之一。程序清单 4-1 表明，如果你将定义一个名为 `type` 的函数，它将使内置的 `type` 函数无法访问。

**程序清单 4-1 定义一个函数来代替内置 type 函数()**

```
>>> S = "A test string"
>>> type(S)                  # test the built-in type()
<jclass org.python.core.PyString at 6879429>
>>> def type(arg):
...     print "Not the built-in type() function"
...     print "Local variables: ", vars() # look for 'arg'
...
>>> type(S)
Not the built-in type() function
Local variables: {'arg': 'A test string'}
>>>
>>> # to restore the builtin "type()" function, do the following:
>>> from org.python.core import __builtin__
>>> type = __builtin__.type
```

参数表是出现在跟有函数名的括号里的一个可选列表。传递给函数的变量在本地名空间里绑定到可用的参数名。在程序清单 4-1 里被定义的函数接受一个参数，它被绑定到局部名 arg。一个冒号 “:” 结束函数声明并指定相关程序代码块的开始。

**4.1.1 缩进**

缩进（Indentation）限定 Jython 中的程序代码块。一个函数的关联程序代码块必须在它的 def 语句中启动一个缩进层次。因为缩进是编码块的惟一分隔符，所以你不能有一个空程序代码块——在一个函数内必须有某种占位符。如果你需要一个占位符，就使用 pass 语句。程序清单 4-2 有示范 Jython 的相关缩进层次的函数。

程序清单 4-2 决定哪个数字是一个作为参数提供的列表之外的素数。素数测试是非常低效的，但是能很好地示范函数语法和结构。isPrime 函数是简单地返回 1 的函数，1 是真，表示那些数字是素数，0 是假，表示那些数字不是素数。因为整章讨论素数的产生，所以更有效的素数查找示例将被讨论。大于数字 2 的所有的素数是奇数。所有非素数数字能通过素数相乘被创建。知道这些，对排除不必要的候选条件以及减少不必要的重复方面更简便易行。程序清单 4-2 是低效的，因为它忽略以后的原则并仅仅把除数查找限制为奇数。在本章中，由于对 Jython 规则的简单明了的例子而非算法本身优先，其余素数生成程序将包含类似的缺点。

**程序清单 4-2 用嵌套函数查找素数**

```
#file primes.py
def isPrime(num):
    """Accepts a number and returns true if it's a prime"""
    for base in [2] + range(3, num, 2):
        if (num%base==0):
            return 0
    return 1

def primes(S1):
    """Accepts a list of numbers and returns a
       list of those numbers that are prime numbers"""
    primes = []
```

```

for i in S1:
    if isPrime(i):
        primes.append(i)
return primes

list1 = range(20000, 20100)
print 'primes are:', primes(list1)

```

运行 `python primes.py` 的结果是：

```
primes are: [20011, 20021, 20023, 20029, 20047, 20051, 20063, 20071, 20089]
```

因为在程序清单 4-2 中 `isPrime` 函数返回 Jython 所理解的真和假的值（对于数字对象，`0 = false`，`non-zero = true`），函数可以是一个 `if` 语句的条件部分。`if` 语句在这种情况下放弃 `isPrime` 函数为值因为 `isPrime` 函数返回 0。

#### 4.1.2 返回值

程序清单 4-2 使用 `return` 语句来指定函数返回值。函数返回跟有 `return` 语句表达式的结果。如果没有提供 `return` 语句，函数返回 `None`。如果 `return` 后没有跟有值，函数也返回 `None`。如果返回值多于一个，这些值就会作为一个元组返回。值得提及的是当一个 `return` 语句被调用时函数退出。这种 call-return 的编程形式假设一个 `return` 语句只是那么做——把值和程序流返回到 calling 语句中。

程序清单 4-3 定义一个函数，它决定一个点与有着确定圆心和半径的圆的关系。在这个程序清单中，根据点是在圆内、圆周上或圆外的位置不同，`bounds` 函数明确地返回 1、0 或 -1。在程序清单 4-3 中，只有当第一个 `if` 语句求值为假时，最后的 `return` 语句才能得到。因为在那个 `return` 语句后，缺省一个表达式，它就返回 `None` 值。因为没有显式返回，这最后的 `return` 语句可能已被省掉，函数不管如何，返回 `None` 值。

#### 程序清单 4-3 圆包含函数

```

# file: bounds.py
def bounds(p, c, r):
    """Determine a points 'p' relationship to a circle of
    designated center 'c' and radius 'r'.
    1 = within circle, 0 = on circle, -1 = outside circle"""
    ## Ensure args are not empty nor 0
    if p and c and r:
        ## Find distance from center "dfc"
        dfc = ((p[0]-c[0])**2 + (p[1]-c[1])**2)**.5

        return cmp(r, dfc)

print bounds((1,2), (3,1), 3)
print bounds((-3, 1), (1, 1), 4)
print bounds((4,3), (5,-2), 1)
print bounds({}, (2,2), 4)

```

从运行 `jython bounds.py` 的输出：

```
1
0
-1
None
```

### 4.1.3 文档字符串

程序清单 4-2 对两个函数中的任一个都有一个文档字符串。这个字符串从程序代码块的第一行开始，所以在 `def` 语句中必须是一个缩进层次。这字符串的内容成为 `function's __doc__ attribute` 的值（两个前端和后端下划线）。如果我们在交互的解释器中看到一个简单函数，就如同程序清单 4-4 所做的那样，当文档字符串可用时，我们看见交互地开发对象是有价值的。

#### 程序清单 4-4 函数文档字符串

```
>>> def getfile(fn):
...     """getfile(fn), Accepts a filename, fn, as a parameter and quietly
returns the resulting file object, or None if the file doesn't exist."""
...     try:
...         f = open(fn)
...         return f
...     except:
...         return None
...
>>> print getfile.__doc__
getfile(fn), Accepts a filename, fn, as a parameter and quietly returns the
resulting file object or None if the file doesn't exist.
```

许多模块在它们的 `_doc_` 字符串里被完整记录，当遇到与函数、模块或类有关的麻烦时，首先是值得查找相关的 `_doc_` 字符串的。

### 4.1.4 函数属性

正如 Jython 版本 2.1，函数也有任意属性。这些是用 `function.attribute` 语法赋值的属性。并且这种赋值可在函数程序块块内或在函数外。这是一个具有属性的函数的例子：

```
>>> def func():
...     func.a = 1
...     func.b = 10
...     print func.a, func.b, func.c
...
>>> func.c = 100
>>> func()
1 10 100
```

### 4.1.5 参数

Jython 的参数模式表是很灵活的。在 Jython 中，函数参数是一个有序列表，参数可以有缺省值，甚至你能允许有未知数量的变量存在。另外，你能使用键-值对，当调用函数时，甚至

允许有未知数量的键-值对。

### 1. 位置参数

到目前为止，函数例子使用了参数最简单形式：一个名字列表或位置参数。在这种情况下，被传递到函数的自变量按照出现的先后顺序被绑定到在参数表中指定的局部名字。在解释器中用 vars() 函数检验并阐明这一点。vars() 函数返回一个字典对象，字典对象不是保持有序的。这就意味着你看见的结果可能是无序的，但只要所有的键和值出现结果是正确的：

```
>>> def myFunction(param1, param2, param3):
...     print vars()
...
>>> myFunction("a", "b", "c")
{'param1': 'a', 'param2': 'b', 'param3': 'c'}
```

### 2. 缺省值

你可以把缺省值赋给参数。在函数定义范围内，简单地把一个值赋给参数。如果定义一个有三个参数的函数，两个参数有缺省值，当调用时只需要一个自变量，但可以用提供的一个、两个或三个变量调用。当使用缺省值时，没有缺省值的参数可以不跟有缺省值的参数。

这是一个在第三个参数之前已有两个缺省值的函数的例子。这创建了调用函数的灵活性。

```
>>> def myFunction(param1, param2="b", param3="c"):
...     print vars()
...
>>> myFunction("a")      # call with the 1, required parameter
{'param1': 'a', 'param2': 'b', 'param3': 'c'}
>>> myFunction("a", "j") # call with 2 parameters
{'param1': 'a', 'param2': 'j', 'param3': 'c'}
>>> myFunction("a", "j", "k") # call with 3 parameters
{'param1': 'a', 'param2': 'j', 'param3': 'k'}
```

下面是另一个例子，但是这个函数在一个缺省值后有一个非缺省参数。这是不允许的，通过 SyntaxError 异常证实：

```
>>> # let's test a non-default arg following a default arg
>>> def myFunction(param1, param2="b", param3):
...     print vars()
...
Traceback (innermost last):
  (no code object) at line 0
  File "<console>", line 0
SyntaxError: non-default argument follows default argument
```

### 3. \* params 和处理一未知量的位置参数

如果你需要考虑未知量的自变量，你可给一个参数名字加上星号 \* 作为前缀。星号指定一个通配符，它把所有的其他的自变量作为一个元组。这应该出现在函数定义中的其他参数后。星号是必要的语法；在它以后使用的名字是任意的：

```
>>> def myFunction(param1, param2, *params):
...     print vars()
...
>>> myFunction('a','b','c','d','e','f','g')
{'param1': 'a', 'param2': 'b', 'params': ('c', 'd', 'e', 'f', 'g')}
```

#### 4. 关键字对作参数

当调用函数时，你也可使用键-值对来指定参数，也称作关键字参数。键必须是用在函数定义中的实参数名。当使用键-值对时，顺序不再重要，但当把键-值对与明码、位置参数混合时，顺序则有关系直到最右的位置自变量：

```
>>> def myFunction(param1, param2, param3="d"):
...     print vars()
...
>>> myFunction(param2="b", param3="c", param1="a")
{'param1': 'a', 'param2': 'b', 'param3': 'c'}
>>> # next let's mix key-value pairs with positional parameters
>>> myFunction("a", param3="c", param2="b")
{'param1': 'a', 'param2': 'b', 'param3': 'c'}
```

#### 5. \*\*kw params 以及处理未知键-值对

仅当星号处理未知量的明码自变量时，双星号 \*\*，处理未知量的键 - 值对。你选择作为双星号前缀的名称成为一种包含不定键-值对的 PyDictionary 类型。

```
>>> def myFunction(param1, param2, param3, **kw):
...     print vars()
...
>>> myFunction("a", "b", "c", param4="d", param5="e")
{'param1': 'a', 'param2': 'b', 'param3': 'c', 'kw': {'param5': 'e', 'param4': 'd'}}
```

注意所有类型参数能在相同函数定义中出现：

```
>>> def myFunction(param1, param2="b", param3="c", *params, **kw):
...     print vars()
...
>>> myFunction("a", "b", "j", "k", "z", key1="t", key2="r")
{'param1': 'a', 'params': ('k', 'z'), 'param3': 'j', 'kw': {'key2': 'r',
'key1': 't'}, 'param2': 'b'}
```

## 4.2 名空间

Jython 有静态和静态嵌套的作用域。在 Jython 2.1 中出现了介于这两者之间的桥接。Jython 2.1 版本和 2.0 版本有静态的作用域；然而，你能选择性地使用静态嵌套的范围，在 2.1 版本中，也被称作词法作用域。Jython 的未来版本将只使用静态地被嵌套了的作用域（词法的作用域）。Jython 2.1 衔接两种类型作用域的原因是当提供时间反向修改受到影响的遗留代码时，Python 开发者需要一种安全方法介绍新的作用域规则。

### 4.2.1 两个静态范围

Jython 静态作用域承担两个特定的名空间：globals（全局）和 locals（局部）（如果包括内置的名空间就会有三个）。当名称绑定过程发生时，名字就出现在全局或局部名空间中。名称绑定的过程是赋值操作、一个 import 语句、函数或一个类定义，以及从语句行为中导出的赋值（例如列表中的 x）。一个名称绑定过程的定位决定哪个名空间将包含那个名称。对于函数，任

任何绑定在函数编码块中的内容，和所有的函数参数都是局部的，但有一个例外——在编码块内 global 语句的显式使用。

程序清单 4-5 包含一个能得出一些赋值的函数，并把它的局部名空间打印加以确认。程序清单 4-5 也打印全局名称以显示出全局变量是如何独一无二，并显示在 function() 中全局变量是怎样和局部变量相分离的。

#### 程序清单 4-5 全局和局部名空间

---

```
>>> a = "This is the global variable a"
>>> def function(localvar1, localvar2):
...     a = 1
...     b = 2
...     print locals()
...
>>> function(3, 4)
{'b': 2, 'localvar1': 3, 'localvar2': 4, 'a': 1}
>>>
>>> globals()
{'a': 'This is the global variable a', '__doc__': None, 'function':
<function function1 at 5135994>, '__name__': '__main__'}
```

---

因为程序清单 4-5 在 function() 中用一个局部名表示变量，而全局变量 a 未受影响。

然而一个函数能使用来自全局名空间的变量。如果名称绑定过程在这个函数以外进行，或全局语句被使用，这种情况就会发生。如果一个名称在函数内出现，但是名称绑定出现在全局名空间中，那么，名称查找过程将不考虑局部而进入全局名空间。这有个使用全局名的例子：

```
>>> var1 = [100, 200, 300]    # This is global
>>> def function():
...     print var1
...     var1[1] += 50
...
>>> function()
[100, 200, 300]
>>> function()
[100, 250, 300]
>>> print var1  #Look at the global var1
[100, 300, 300]
```

如果你试着在一个以后绑定在相同名称上的函数中使用一个全局变量，那就会产生错误。原因是不管名称绑定有序地出现在块的什么位置，名称的绑定都把一个变量指定整个程序代码块为局部符：

```
>>> var = 100    # This is global
>>> def function():
...     print var
...     var = 10  # this assignment makes ALL occurrences of var local
...
>>> function()
Traceback (innermost last):
  File "<console>", line 1, in ?
    File "<console>", line 2, in function
UnboundLocalError: local: 'var'
```

在函数内使用一个全局变量的第二种方法是用 `global` 语句声明变量全局。用 `global` 语句显式声明意向如下所示：

```
>>> var = 100 # This is global
>>> def function():
...     global var
...     print var
...     var = 10
...
>>> print var # Peak at global var
100
>>> function()
100
>>> print var # Confirm function acted on global var
10
>>>
```

#### 4.2.2 静态嵌套的范围

为什么改变到静态地被嵌套的范围？不管怎么说，Jython、Python 在使用静态作用域上并没有出错过？这个问题表明自身与嵌套函数和 `lambda` 表有关（`lambda` 表将在本章后面谈到）。这两者在双重名空间和静态作用的模式中是非直觉的。从在交互式控制台定义简单嵌套函数能清楚看到：

```
>>> a = "BacBb"
>>>
>>> def decode():
...     b = "h"
...     def inner_decode():
...         print a.replace("Bb", b)
...     inner_decode()
...
<console>:1: SyntaxWarning: local name 'b' in 'decode' shadows use as
global in nested scopes
>>> decode()Traceback (innermost last):
  File "<console>", line 1, in ?
  File "<console>", line 5, in decode
  File "<console>", line 4, in inner_decode
NameError: b
```

Jython 2.1 版本在内部函数中完全友好地提供关于变量 `b` 不可见性的控制台警告。仅仅与局部及全局的名空间，`inner-decode()` 不能在它包含的函数中看到变量。克服这个不足，通常意味着提供给内部函数一些参数。然而，在 Jython 2.1 版本的开始，你也能通过选择使用静态嵌套范围来克服这个不足。

在 Jython 2.1 中使用静态嵌套范围要求使用近来增加的 `_future_`。为了从 `_future_` 引入嵌套范围，使用下列格式：

```
>>> from __future__ import nested_scopes
```

当前，`nested_scopes` 是用户惟一能引入的未来行为，但是当完成时，解释器服从静态嵌套范围的规则。如果我们再看看这个带有引入的未用的 `nest_scopes` 的 `decode` 例子，我们得到不同结果：

```
>>> from __future__ import nested_scopes
>>>
>>> a = "BacBb"
>>>
>>> def decode():
...     b = "h"
...     def inner_decode():
...         print a.replace("Bb", b)
...     inner_decode()
...
>>> decode()
Bach
```

与词法作用域一起，内部函数能包含一些附加的、非全局的信息。假定你想在一个不同的名字下隐藏内置函数，而你使用了它的旧名字时：你能使用一个内部函数用作有如在程序清单 4-6 中的一个代理。

#### 程序清单 4-6 使用嵌套范围创建一个代理

```
# file: typeproxy.py
from __future__ import nested_scopes

def typeProxy():
    _type = type
    def oldtype(object):
        return _type(object)
    return oldtype

f = typeProxy()
type = 10 # rebind name "type"
print f(4)
```

运行 jython typeproxy.py 得到的结果：

```
org.python.core.PyInteger
```

因为词法的作用域允许嵌套的函数从它们包含的函数中使用变量，嵌套函数像运送行李一样输送非全局数据。这创建了把数据和过程联合起来的一个对象，很像一个类。这与函数语言把什么称为闭合相类似。

程序清单 4-7 使用嵌套函数和返回的内部函数闭合般的行为来产生 fibonacci 数字流。xfib 功能接受一个自变量：顺序应该以迭代开始。两个 assert 语句提供一些运行时的参数以检查并增加一些健壮性，内部函数 fibgen 被定义后，在它被返回以前，它被用来设置数据到迭代。返回这个函数的能力是第一类对象要求的一部分，并且在函数的编程中也是重要的成员。script 使用返回函数打印 fibonacci 序列的前 20 个数字。

#### 程序清单 4-7 使用词法作用域的嵌套函数

```
# file: fibonacci.py
from __future__ import nested_scopes

def xfib(n=1):
    """Returns a function that generates a stream of fibonacci numbers
```

```

Accepts 1 arg- an integer representing the iteration number of
the sequence that the stream is to start with."""
assert n > 0, "The start number must be greater than zero"
from types import IntType, LongType
assert type(n)==IntType or type(n)==LongType, \
    "Argument must be an integral number."

# make cache for penultimate, current and next fib numbers
fibs = [0L, 1L, 0L]

# Define the inner function
def fibgen():
    fibs[2] = fibs[1] + fibs[0]
    fibs[0] = fibs[1]
    fibs[1] = fibs[2]
    return fibs[1]

# Set state to the iteration designated in parameter n
for x in range(1, n):
    fibgen()

# return the primed inner function
return fibgen

fibIterator = xfib()
for x in range(20):
    print fibIterator(),
print # Print an empty newline for tidiness after execution

```

运行 `jython fibonacci.py` 得到的结果：

```
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
```

将程序清单 4-7 `fibIterator` 对象传递到另外的函数或类是一个可能的脚本。数据和过程的联合使得它在许多情况下都有作用，与类的实例对它们方法和数据的联合有很相似。

#### 4.2.3 在用户定义函数中的特殊变量

Jython 的“特殊”变量通常是那些分别有前行和拖尾下划线的变量，例如`_doc_`。然而，Jython 函数也包含有特定功能的特殊变量，并被加以前缀`func_`。一个如此特殊的函数变量功能是`func_doc`，它是`_doc_`的一个假名。当你用`dir()` 或 `vars()` 开发函数时，你将遇到这些变量，所以尽管本章的例子不直接使用它们，它们在这里也被罗列出来作为参考。

特殊函数变量如下：

- `func_doc` 与 `as_doc_` 相同。
- `func_name`（也可作`_name_`）是函数名。
- `func_defaults` 列出缺省函数值。
- `func_code` 包括函数被编辑的版本。
- `func_globals` 是函数的全局名空间。
- `func_dict`（也可作`_dict_`）是函数局部名空间。

- `func_closure` 是一个绑定在 `nested-scopes` 中使用的变量上的元组。如果不使用 `nested-scopes`, 它总为 `None` 值。

### 4.3 递归

当一个程序调用它自己时, 递归 (recursion) 就发生。递归是一种有力的控制结构, 与其他的循环结构类似, 在函数编程中也是必要成员。像在序列和图形中发现的那些一样, 递归与重复数学运算同时出现。递归另一个普通例子是求阶乘的值。阶乘的简化描述是所有整数在指定数字和 1 之间的乘积。这意味着 5 阶阶乘 (写作  $5!$ ) 是  $5 * 4 * 3 * 2 * 1$  (假设 `gamma` 函数的细节被忽略)。使递归函数实现这一计算在程序清单 4-8 被显示出来。

程序清单 4-8 实现递归函数

---

```
>>> def factorial(number):
...     number = long(number)
...     if number == 0:
...         return 1 # It just is by mathematician's decree
...     return number * factorial(long(number - 1))
...
>>> print factorial(4)
24
>>> print factorial(7)
5040
```

---

递归有它的限制。在 `StackOverflowError` 出现以前, 函数能调用它自己许多次。如果我们从上面扩展我们交互式的例子, 我们能试用越来越大的数字来判断这限制是什么。注意实际的数字将取决于你可靠的记忆。

```
>>> bignum = factorial(1000)
>>> biggernum = factorial(1500)
>>> hugenum = factorial(1452) # The breaking point on my machine
traceback (innermost last):
...
java.lang.StackOverflowError: java.lang.StackOverflowError
```

### 4.4 内置的函数编程工具

直到这个时内置函数编程工具例子在风格上是必要的。然而, 函数编程的许多重要成员存在在 Jython 中。那些已提到的是第一类函数、递归以及闭合。在函数中运行的 Jython 附加工具是 `lambda` 格式, 该格式为表处理、表生成工具 `zip()` 和表理解提供了隐函数和内置函数。本节主要讲 Jython 的函数工具, 包括 `lambda` 表、映射、过滤器、简化、压缩及表的理解。

#### 1. lambda

创建匿名函数要求使用 `lambda`。用 `lambda` 表或者 `lambda` 表达式定义函数。术语 `lambda` 表达式可能最有启发性, 就像这类函数作用时返回表达式结果。`lambda` 表达式不能包括语句。`lambda` 表达式的语法如下:

`"lambda" parameters: expression`

确定一个数字是奇数还是偶数的问题创建这个 lambda 格式：

```
>>> odd_or_even = lambda num: num%2 and "odd" or "even"
>>> odd_or_even(5)
'odd'
>>> odd_or_even(6)
'even'
```

你能看到，从功能上这与下面相同：

```
>>> def odd_or_even(num):
...     return num%2 and "odd" or "even"
```

lambda 表的参数与命名的函数参数不相同。它们可以是定位参数、有缺省值并能使用通配符参数 \* 和\*\*。

因为 lambda 表不能包含 if/else 语句，他们通常利用 and/or 运算符来提及表达式的逻辑性。在函数编程上，表达式的求值优先于语句。使用 lambda 格式中的表达式不仅是被需要的，而且在函数编程中它也是有价值的实践。

lambda 表达式的另外一个例子是重写阶乘求值程序。运用在这个例子里的控制结构是递归：lambda 格式调用本身决定每个连续值：

```
>>> factorial = lambda num: num==1 or num * factorial(num - 1)
>>> factorial(5)
120
```

lambda 表达式特别易受 python 作用域规则负作用的影响。在静态的作用域中，缺省参数值很频繁地使 lambda 编码不清楚。只有少数要求缺省值的变量变得不实用。程序清单 4-9 给出了静态高度的柱体体积及变量半径，同时，显示出 lambda 表达式中缺省自变量。

#### 程序清单 4-9 使用在 lambda 格式的缺省参数

---

```
# file: staticpi.py
# Note "nested_scopes" is not imported
from math import pi

def cylindervolume(height):
    return lambda r, pi=pi, height=height: pi * r**2 * height

vrc = cylindervolume(2) # vrc = variable radius cylinder
print vrc(5)
```

---

运行 `python staticpi.py` 得：

157.07963267948966

在词法所覆盖的 lambda 格式中，把这与丢失的缺省参数值作比较。程序清单 4-10 表明，lambda 表与词法作用域直观地表现在其中。

#### 程序清单 4-10 词法作用域的 lambda 格式

---

```
# file: lexicalpi.py
from __future__ import nested_scopes
```

```
from math import pi

def cylindervolume(height):
    return lambda r: pi * r**2 * height

vrc = cylindervolume(2) # vrc = variable radius cylinder
print vrc(5)
```

运行 jython lexicalpi.py 的结果是：

```
157.07963267948966
```

## 2. map()

映射函数是一张表处理工具。映射 (map) 语法如下：

```
map(function, sequence[, sequence, ...])
```

第一个自变量必须是函数或 None。Map 为指定序列的每个成员调用函数，并把每一个成员看成是一个自变量。返回的列表是每次调用函数的结果。创建一张整数平方的列表，你可以这样使用 map：

```
>>> def square(x):
...     return x**2
...
>>> map(square, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

lambda 格式也满足函数的要求：

```
>>> map(lambda x: x**2, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

如果一个映射函数用多重序列调用，因为有序列，函数就会得到同样多的参数。如果我们假定映射和函数（或 None 值）以及三个序列被同时调用，那么每调用一次函数就会包括三个参数：

```
>>> map(None, range(10), range(2,12), range(5,15))
[(0, 2, 5), (1, 3, 6), (2, 4, 7), (3, 5, 8), (4, 6, 9), (5, 7, 10), (6,
8, 11), (7, 9, 12), (8, 10, 13), (9, 11, 14)]
```

对于不同长度的多重序列，映射通过最长的序列以及对丢失值填入 None 来迭代。

```
>>> map(None, range(5), range(6,20), range(3))
[(0, 6, 9), (1, 7, 1), (2, 8, 2), (3, 9, None), (4, 10, None), (None, 11,
None), (None, 12, None), (None, 13, None), (None, 14, None), (None, 15,
None), (None, 16, None), (None, 17, None), (None, 18, None), (None, 19,
None)]
```

要补充数字焦点，看看用映射来处理字符串的实例。这个例子假设需要用有四个空格键的字符串来代替所有换行键。允许一个空字符串使用 join 函数，并且当一个换行键被发现时，lambda 表达式就会使用布尔 (Boolean) 求值返回四个空格。

```
>>> s = 'a\tb\tc\td\te'
>>> ''.join(map(lambda char: char=='\t' and '    ' or char, s))
'a    b    c    d    e'
```

### 3. filter()

过滤器函数与映射函数类似，它需要一个函数或 None 值作为第一参数，以及一个列表为第二参数。差别是过滤器仅仅能接受一个序列，过滤器功能是用来决定真/假值，并且过滤器的结果是包含原序列成员的，而函数就是来求值到真。函数把这些序列成员过滤出来，而对于这些成员，函数就会返回 false 值——也就是成员名。过滤器的语法如下：

```
filter(function, sequence)
```

阐明过滤器行为的例子是测试偶数。`range` 函数有可选 `step` 参数，于是这功能很小了，但它是 `filter` 行为的一个清楚例子。结果包含原序列成员，而且除了排除函数所否认的，其余不变。

```
>>> filter(lambda x: x%2==0, range(10))
[0, 2, 4, 6, 8]
```

`filter` 函数也能压缩用来集合交叉的代码。这个例子通过 `set1` 成员迭代，根据 `set2` 组成元素的内容，使用 `lambda` 格式返回 1 或 0。

```
>>> set1 = range(0, 200, 7)
>>> set2 = range(0, 200, 3)
>>> filter(lambda x: x in set2, set1)
[0, 21, 42, 63, 84, 105, 126, 147, 168, 189]
```

如果你再看看本章的程序清单 4-2，你会注意到寻找素数是很冗长且低效的。`filter` 函数能减少繁琐，并且在程序清单 4-11 讲述过。但效率没被提高。程序清单 4-11 可能效率很低，因为使用 `filter` 函数不是为了省事。`lazy` 是说一个过程仅仅求它需要什么。程序清单 4-11 `isPrime` 函数连续地测试每个潜在的除数。为了 `lazy`，`isPrime` 函数将只得测试最合适那些除数直到素数的第一永假式。当数字变得更大，有害效果是难以承担的。

#### 程序清单 4-11 素数的函数搜索

---

```
# file: functionalprimes.py
from __future__ import nested_scopes

def primes(S):
    isDiv = lambda num, x: num<3 or num%x==0
    isPrime = lambda num: filter(lambda x, num=num: isDiv(num, x),
                                  [2] + range(3,num,2))==[]
    return filter(isPrime, S)

print primes(range(4, 18))
```

---

运行 `jython functionalprimes.py` 的结果是：

```
[5, 7, 11, 13, 17]
```

### 4. reduce( )

`reduce` 函数像映射和过滤器一样需要一个函数和序列。但提供给 `reduce` 的函数必须接收两个自变量，且不能像在映射和过滤器中为 `None` 值。`reduce` 语法如下：

```
reduce(function, sequence[, initial])
```

从 reduce 返回的结果是一个单值，该值把指定函数的累积应用表示成提供的序列。列表中的前两个值，或可选的初始值和第一表值，成为函数的第一对自变量。该操作得到的结果和下一个列表项是这个函数随后的自变量，依此类推直到列表末。一个例子如下：

```
>>> reduce(lambda x, y: x+y, range(5, 0, -1))
```

这个表达式的结果是  $((((5+4)+3)+2)+1)$ ，或 15。假设可选的初始值是被提供的：

```
>>> reduce(lambda x, y: x+y, range(5, 0, -1), 10)
```

结果将是  $((((10+5)+4)+3)+2)+1$ ，或 25。

#### 5. zip ()

zip 函数把多重列表各个单元联合作为一元组，并且返回一个元组列表。返回列表的长度与所提供的最短序列一样。zip 语法如下：

```
zip(seq1 [, seq2 [...]])
```

一个例子用法如下：

```
>>> zip([1,2,3], [4,5,6])
[(1, 4), (2, 5), (3, 6)]
```

另一个例子是在构建字典时，使用 zip 提供键-值对。因为 zip 生成一个元组列表，lambda 格式对于单个元组自变量只有一个参数：

```
>>> d = {}
>>> map(lambda b: d.setdefault(b[0],b[1]), zip(range(4), range(4,8)))
[4, 5, 6, 7]
>>> d
{3: 7, 2: 6, 1: 5, 0: 4}
```

#### 6. List comprehension

List comprehension 是根据一套规则生成列表方式。List comprehension 语法确实是一条捷径。它做的可能有另外的 Jython 语法参与，但 list comprehension 显得更清楚。基本的语法如下：

```
[ expression for expr in sequence1
    for expr2 in sequence2 ...
    for exprN in sequenceN
    if condition]
```

list comprehension 的求值如下所示，等价于嵌套的 for 语句：

```
for expr1 in sequence1:
    for expr2 in sequence2:
        for exprN in sequenceN:
            if (condition):
                expression(expr1, expr2, exprN)
```

用在这里的 list comprehension 生成了求值为真的元素列表：

```
>>> L = [1, 'a', [], 0, 'b']
>>> [x for x in L if x]
[1, 'a', 'b']
```

这可以在没有 list comprehension 的情况下重写：

```
>>> L = [1, 'a', [], 0, 'b']
>>> results = []
>>> for x in L:
...     if x:
...         results.append(x)
...
>>> results
[1, 'a', 'b']
```

list comprehension 的优点是清楚明了的。随着 for 语句的增加，这个优点也增加。在程序清单 4-12 中，乍看 list comprehension 可能有点迷惑不解，那是因为满篇都是像下面嵌套结构的 lambda 表达式。在程序清单 4-12 中的实际 list comprehension 只有与以下嵌套形式相等的四个表达式：

```
for x in [num]:
    for y in ([2] + range(3, num, 2)):
        if x%y==0:
            returnlist.append(x)
```

这个 list comprehension 的结果是变量 num 的一连串除数组成的列表。lambda 格式只是常识地判断除数列表是否为空，也就是说，数字必须是一个素数。

程序清单 4-12 list comprehension 中的素数

---

```
# file: primelist.py
def primes(S):
    isPrime = lambda num: [x for x in [num]
                           for y in [2] + range(3,num,2)
                           if x%y==0]==[]
    return filter(isPrime, S)

print primes(range(2,20))
print primes(range(200, 300))
```

---

运行 `python primelist.py` 的输出：

```
[3, 5, 7, 11, 13, 17, 19]
[211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
283, 293]
```

Python 文件对象有 `readlines()` 方法，它返回一个字符串列表表示文件的每行。对于另外的 list example 例子，假设文件 `config.cfg` 是一个带有以 # 开始的注释行配置文件，且所有另外的非空白行是重要指令。在字典中，你想要这样一个文件的经解析的结果是很有可能的，但是因为我们这里集中在函数编程上，所以 `readlines()` 和 list comprehension 被用来把这个文件解析到一系列列表中。

关于 list comprehension 另外重要的一点是第一表达式调用一种方法。把第一个表达式限制到仅声明哪个值返回似乎是普通用法，但是这当然不是限制，它也能调用一个对象的方法或它是一个函数。

```

# file: configtool.py
def parseConfig(file):
    return [x.strip().replace(' ', '').split('=') for x in open(file).readlines() if not x.lstrip().startswith('#') and len(x.strip())]

print parseConfig("config.cfg")

The content of the config.cfg file used is:
    a      = b
c =      d
    # comment

dog = jack
cat = fritz

```

运行 jython configtoo.py 的结果是：

```
[['a', 'b'], ['c', 'd'], ['dog', 'jack'], ['cat', 'fritz']]
```

前面的 list comprehensions 使 for 语句清楚、紧凑，也使得这个实例更像函数形式的编程。

## 函数编程特色

定义和使用函数并不限制为函数编程。函数的编程会更多。它强调列表处理正如映射、过滤器和 reduce 所做的一样。它也有类似于 list comprehension 的要求。功能编程的另外的重要特征是在表达式上重绑定名字和强调表达式而不是典型的复合语句控制结构。大多数 Jython 编程是必要的，并且是面向对象的，但是我们能从函数的编程看到 Jython 也有函数形式。

在 Jython 中，函数编程的优点是什么？对于增加健壮性是有潜力的，可能碰到一些状况，它提高了编码的清晰性。如果每一样都被求值为表达式，对错位语句几乎没有机会。看看一些必要的代码：

```

>>> def odd_or_even(num):
...     var = 9          # potential side effect
...     if num%2:        # statement instead of expression
...         return "odd"
...     return "even"
...
>>> odd_or_even(10)
'even'
>>> odd_or_even(7)
'odd'

```

这使用了复合的 if/else 语句。函数的编程很少使用语句而多用表达式。因此，把上面的编码转变成更像函数的一些形式，使用 and 和 or 运算符：

```

>>> def odd_or_even(num):
...     return num%2 and "odd" or "even"
...
>>> odd_or_even(20)
'even'
>>> odd_or_even(33)
'odd'

```

除了避免语句，在函数编程中把名字重绑定减到最低也是很重要的。重绑定是在必要的编程中通常所涉及到的，这里有一些便捷的运算符，例如`+ =`使它更方便。函数编程把诸如重绑定看成是增加其副作用的风险。在得到`my`值之前，那些在 Perl 调试程序上碰到麻烦的人比大多数人更清楚地知道变量重绑定负作用意外和有害的方面。如果你再回看程序清单 4-12，你将看到`list comprehension`在函数中能消除许多名称绑定，所以负作用是不可能的。程序清单 4-11 一样是真；名字被绑定在`lambda`格式和参数上，但是没有其余的名字重绑定存在。

使人兴奋的是 Jython 的列表处理工具可以在任何序列上工作。不只在 Jython 序列，还有 Java 序列。通过一个`java.util.Vector`对象迭代，在映射函数中作为序列指定它：

```
>>> import java
>>> v = java.util.Vector()
>>> map(v.add, range(10)) # fill the vector with PyIntegers
[1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> filter(lambda x: x%2==0, v)
[0, 2, 4, 6, 8]
```

因为害怕影响例子的清晰性，本章没有强调 Java 对象，但是有启发性的下一步是在函数形式的脚本语言中，与合并的 Java 对象共同实验。一旦 Java 被包括进来，纯粹的函数编码便不同平常，但你能在列表处理和`lambda`格式中使用 Java 的范围是较广的。

## 4.5 同步

Java 和 Jython 允许执行多线程编程。Python 的`thread`和`threading`模块是在 Jython 和 CPython 创建多线程的程序方式。对 Jython 来说，独一无二的是同步模块，它更像 Java 的`synchronized`关键词简化同步。

因为 Jython 函数是可调用对象，所以关于函数的本章介绍了同步。Java 程序员习惯于同步的方法，但是在 Python 语言说明中，没有这样的关键词。这就提出一个你实际是怎样使 Jython 的函数或方法同步的。当进入和离开可调用的对象时，你能使用多线程模块中的工具来获得释放锁，但 Jython 使这更容易。Jython 包括有两个函数的`synchronize`模块，它容许用户同步可调用对象。这两个函数是`make_synchronized`和`apply_synchronized`。

程序清单 4-13 定义`count`方法，它以两个独立的线程开始。`count`方法仅仅打印一个序列的数字，而线程负责生成数字。这使 for 成了一个清晰的例子 因为输出没有同步被交叉，但当使其同步时它就产生有序性。

程序清单 4-13 同步一个函数

---

```
# file: sync.py
import thread
import synchronize
import sys

threadID = 0

def count(out):
    global threadID
    thisid = threadID
    threadID += 1
    threadnames = ["first thread", "second thread"]
```

```

for i in range(5):
    print threadnames[thisid], "counted to", i

count = synchronize.make_synchronized(count)

thread.start_new_thread(count, (sys.stdout,))
thread.start_new_thread(count, (sys.stdout,))

```

函数 `thread.start_new_thread()` 开始于程序清单 4-13 中的两个线程，它有下列语法：

```
thread.start_new_thread(callable_object, args[, kwargs])
```

第一参数是将运行在线程上的可调用对象的名字。第二参数是一个在第一参数中通向自变量元组的可调用对象。可选的第三个参数由关键词自变量组成，也通向可调用对象。

程序清单 4-13 与 `make_synchronized` 函数实现同步。`make_synchronized` 接受一个可调用对象并且返回一个新的可调用对象。当对象被调用时，新对象在第一自变量上同步。语法如下：

```
make_synchronized(callableObject) -> callableObject
```

对象在第一自变量上同步的事实意味着被使用的可调用的对象必须接受至少一个参数。程序清单 4-13 使用 `sys.stdout` 作为对象以保证输出是成序列的。尽管 Jython 方法还没被讨论，但注意到它们与 Jython 函数类似是有价值的；然而，他们收到一个称为 `self` 的实例引用作为它们的第一个自变量。因此，当 `make_synchronized` 按照一个方法被使用时，那个方法就与对象本身同步化了。

如果你从程序清单 4-13 执行 `sync.py` 文件，你应该看到如下的输出：

```

prompt>jython sync.py
first thread counted to 0
first thread counted to 1
first thread counted to 2
first thread counted to 3
first thread counted to 4
second thread counted to 0
second thread counted to 1
second thread counted to 2
second thread counted to 3
second thread counted to 4

```

为比较的缘故，试着这样注释同步行：

```
# count = synchronize.make_synchronized(count)
```

下一步，再运行 `sync.py` 文件，看看输出是怎样没有同步被交叉的。一个非同步的输出例子如下：

```

prompt>jython sync.py
second thread counted to 0
second thread counted to first thread 1
counted to second thread 0
counted to first thread 2

```

```
counted tosecond thread 1
counted tofirst thread 3
second thread counted to 4
counted to 2
first thread counted to 3
first thread counted to 4
```

你能交替地使用 `synchronize.apply_synchronized` 函数来同步可调用对象。`synchronize.apply_synchronized` 函数复制带有一个附加自变量的内置应用函数功能，这个自变量能使对象继续同步化。这个语句的语法如下：

```
apply_synchronized(syncObject, callableObject, args, keywords={})
```

在 `apply_synchronized` 中执行的运算在第一自变量上被同步（`syncObject`）。



## 第 5 章 模块和包

在 Jython 中，一个人必须管理 Jython 模块和包，以及 Java 的包和类。Jython 引入和使用所有这样的项，但是从哪儿引入的呢？Java 使用 classpath 定位 Java 包和类，而 Python 有 sys.path 定位并装载它的模块。Jython 则要求两者都使用。当实现两种这样的机制时，这就成为一个有趣的困境，但幕后的东西显得模模糊糊的，仅仅为 Jython 用户创建选择。

对于这些选择的学习是从 import 语句开始的，而紧跟其后的是 Java 和 Python 习惯的快速比较。本章的中间部分分成两个部分：Python 包和模块，以及 Java 包和类。交互地工作或许可以与动态地被产生的代码一起运行，就意味着模块能在被引入后变化。Jython 的“重装载”设备在这些状况下起到帮助作用，本章最后会讨论这个问题。

### 5.1 import 语句

import 语句出现在以下四行中。如果变量 X、Y 和 Z 代替模块和包的名称，那这四行看起来如下所示：

```
import X
import X as Y
from X import Y
from X import Y as Z
```

#### 1. import X

这是 import 语句的第一种基本形式。这在形式上与 Java 的 import 语句类似，但是它绑定在最左边的名字上。这与 Java 的 import 是相反的，Java 的 import 绑定在用点标注法表示的 package.class 层次的最右的名字上。一个例子 Java import 可能是：

```
import java.awt.TextComponent;
```

Java import 语句允许你直接使用名字 TextComponent，或最右名字，没有任何包命名限制它。然而，在 Jython 中，一个相似的 import 语句把命名 java 绑定在名空间中，import 语句出现在此，以致于后面所有提到 TextComponent 的地方都必须用 java.awt 限制。用 dir() 函数检查名空间名字的绑定是怎样在 Jython 中工作的：

```
>>> import java.awt.TextComponent
>>> dir()
['__doc__', '__name__', 'java']
```

如果一个名为 b 的模块存在于包 A 中，它的 import 语句如下所示：

```
import A.b
```

然而限定了名字的是名字 A。模块 b 的使用将需要包名字 A 来限制 b。这意味着在模块 b 中的函数 c 被如下引用：

```
import A.b
A.b.c() # call function c
```

sys 模块是一个 Jython 模块，而 Jython 模块不是一个包的部分，因此它在引入期间不要求任何点标注：

```
>>> import sys
```

在 sys 模块被引入后，你用点标注访问内容。这意味着访问下列变量路径要求如下：

```
>>> import sys
>>> sys.path
['', '.', '/jython/Lib']
```

sys.path 变量是 Java classpath 的 Python 版本。这个例子也作为一个检查 sys.path 变量的机会。sys.path 确实取决于你的安装目录，并很可能不同于上面的结果。

import 语句形式也接受一个要引入的模块或包的逗号分隔的列表。模块 os 和 sys 是 Jython 模块，而 java 是合适的 Java 根包。在一命令行上引入这些，如下使用一张逗号分隔的列表：

```
>>> import os, sys, java
```

## 2. import X as Y

增加单词 “as” 到简单的 import 语句中，允许你指定用来引用已引入的项的名字，并允许直接绑定在引入的最右元素。指定名字能避免命名冲突。绑定在最右元素省略了用来限制名字的点标注。

让我们再看看引入 sys 模块，只有这样绑定在名字 sys\_module 上以避免与一个前面被定义了的变量发生命名冲突：

```
>>> sys = 'A variable that creates a name conflict with module sys'
>>> import sys as sys_module
>>> dir()
['__doc__', '__name__', 'sys_module', 'sys']
>>> sys
'A variable that creates a name conflict with module sys'
>>> s
sys module
```

让我们再看看前面陈述的 A.b.c 情形，函数 c 是在模块 b 中，b 驻留在包 A 中。把 b 绑定在一个新名字上，使用 import X as Y 语法。这语法把最右元素 X 绑定在名字 Y 上，因此绑定模块 b 到一个如下所示的新名字：

```
>>> import A.b as mod_b
>>> dir(mod_b)
['__doc__', '__file__', '__name__', 'c']
>>> mod_b.c() # call function c
```

## 3. from X import Y

扩展在包和模块以外的控制，来实际控制模块内容的选择引入，你可以使用 `from X import Y` 语法。这没被限制为仅仅引入模块内容——同样它能和包、模块、类很好地工作。这语法允许引入与修饰语 `as` 共同使用的、与最右绑定相类似的子元素，但是在除了类似子包和子模块的子元素外，它也允许引入模块本身选择性内容。

再回顾脚本 `A.b.c`，你可以使用如下内容绑定模块 `b`:

```
from A import b
b.c() # call function c
```

从一个在使用 `from X import Y` 的语法模块中有选择地引入变量、函数或者类，添加模块名到 `X`，把所需要的模块内容加到 `Y`，这意味着从模块 `b` 中引入函数 `c`，`b` 驻留在包 `A` 中，使用如下：

```
from A.b import c
c() # call function c
```

就像使用简单的 `import` 语句，也容许使用逗号分隔列表。你能如下引入多重的子元素：

```
from A.b import c, d, e, f, etc
```

有如 `sys` 模块中的 `path`，引入模块内部名字如下所示：

```
>>> from sys import path
```

从 `sys` 模块中引入两个名字，使用如下：

```
>>> from sys import path, prefix
```

`from X import Y` 语法允许在 `Y` 位置有一个星号。星号指明所有的变量项目的引入（所有的定义显而易见，并会在以后描述）。在 Jython 中，引入语句使用星号遭到强烈反对，这在以前制造了众多问题（例如当和 `java` 包一起使用 `import *` 时），因此，最好假定这个选择确实不存在。

#### 4. From X import Y as Z

引入的最后形式语法如下：

```
from X import Y as Z
```

当 `X` 是一个被完全限制的包或模块时，`Y` 是一个包、模块或者模块全局名，而 `Z` 是一个任意的、用户定义的 `Y` 被绑定到的名字。就像包 `A` 模块 `b` 中的 `myFunc` 引入函数 `c`，如下所示：

```
from A.b import c as myFunc
```

## 5.2 Jython 和 Java 的比较

Java 有包，它能包含另外的包或类。Java 包与包含文件的目录相联系，而类是编辑这样文件的结果。Jython 有包，它能包含另外的包或模块，而非类。类定义仅仅出现在 Python 模块中。然而，两种语言使用同样的点标注贯穿于它们的各个层次。在 Java 中，如果你希望从包 `A` 引入类 `B`，你将使用如下的 `import` 语句：

```
import A.B;
```

这个 package.class 层次解释为 Jython 模块的 package.module 层次。在 Jython 中类似的 import 语句如下所示：

```
import A.B
```

这个 import 语句导致包 A 的引入，并随之引入模块或是包 B。这是在 Jython 和 Java 之间的重要区别。Jython 作为 import 过程一部分引入每个父包。另外，Jython 包在被引入后，功能和模块相似。

## 5.3 Python 的 package.module 层次

本节回顾了生成、定位和装载 Python 包和模块的 Python 约定。这详细描述了 sys.path 变量的研究以及模块和包的描述。Python 语言含有丰富的特殊变量，因此，这些变量也在本节被描述。

### 5.3.1 sys.path 变量

定位 Python 模块取决于 sys.path 变量。sys.path 列表内的目录告诉 Jython 在哪儿找模块。试着引入不驻留在 sys.path 任何地方的一个 \*.py 文件产生了一个 ImportError 异常。在 sys.path 列表中是 Jython 的 Lib 目录。这是 Jython 安装的模块库驻留的地方。为安装 Jython，检查 sys.path 变量，使用下面所示方法：

```
>>> import sys
>>> print sys.path
['', 'C:\\\\WINDOWS\\\\Desktop\\\\.', 'C:\\\\jython 2.1\\\\Lib', 'C:\\\\jython 2.1']
```

把一条路径加到 sys.path 列表中，引入 sys 模块并且添加所要求的路径。装载位于在目录 /usr/local/jython\_work (on \* nix) 的模块，你必须把这个目录这样加到 sys.path：

```
>>> import sys
>>> sys.path.append("/usr/local/jython_work")
```

对于 Windows 目录 c:\\ jython\_work，使用这些代码：

```
>>> import sys
>>> sys.path.append("c:\\\\jython_work") # notice the double-backslash
['', 'C:\\\\WINDOWS\\\\Desktop\\\\.', 'C:\\\\jython 2.1\\\\Lib', 'C:\\\\jython
2.1\\\\', 'c:\\\\jython 2.1\\\\Lib\\\\site-python', 'c:\\\\jython_work']
```

另外注意 sys.path 是特定的安装，因此，你的 sys.path 内容可能会与此书的内容不同。

反斜线指定像制表键 (\t) 和换行符 (\n) 的特殊字符，还有引号 (" \"") 或字母反斜线 (\\\ )。在 windows 路径中使用双反斜线保证反斜线被解释成字母反斜线而不是一个特殊字符的开始，是最安全的。然而，在大多数情况中，前斜线 /，能在 Jython 编码内的 windows 中，用作一个目录分隔符。

有另外方法添加到 sys.path 变量。Jython 有一个名为 registry，位于 Jython 的安装目录的文件。registry 文件包含很多变量，包括 python.path 和 python.prepath，两个都影响 Jython sys.path 列表的内容。添加路径 /usr/local/jython\_work 到 sys.path，不必明显地在使用它的每个模块中添

加这条路径，只把它加到 `python.path` 或 `Python.prepath` 变量中。这两个变量之间的差别是 `python.path` 在 `sys.pth` 的末尾添加路径，而 `python.prepath` 在 `sys.path` 余下项目前，附在指定路径前面。每个 \*nix 和 Windows 注册表入口如下所示：

```
# for windows, use the double-backslash, and the semi-colon
# path separator.
python.path = c:\\first_directory;d:\\another_directory

# For *nix, use the forward-slash, and the colon path separator
python.path = /first/directory/to/add:/usr/local/secondDir/
```

Jython 和 Python 也使用一个叫做 `site.py` 的文件帮助做好系统路径。`site.py` 文件是一个在 Jython Lib 目录中的模块，在 Jython 开始时，自动装载。`site.py` 模块的一个重要特征是它自动寻找并引入文件 `sitecustomize.py`。`sitecustomize.py` 模块没有作为 Jython 的部分被安装；它是一个你编写的文件，通常放置在 Jython 的 Lib 目录中。在开始的 `sitecustomize.py` 的自动执行例如额外的库路径使它对特定的信息有用。假定你把 Jython 安装在一台 Solaris 机器上的目录 `/usr/local/jython-2.1` 中，并且你希望在目录中组织用户定义的模块到 `sys.path`，你能在包括下列编码的目录 `/usr/local/jython-2.1/Lib` 中创建 `sitecustomize.py` 文件：

```
# file sitecustomize.py
import sys
sys.path.append(sys.prefix + "/Lib/site-python")
```

`sys.prefix` 变量是表示到 Jython 的安装目录路径的字符串。实际上，它是 `python.home` 变量的值，它会是任何任意的路径并在以后讨论，但现在，它都被实际想像成 Jython 安装的路径。Jython 的 Lib 确定为目录作为 `sys.prefix + Lib`。当引入问题出现时，打印 `sys.prefix` 和 `sys.path` 来证实它们的精确性，经常是最有启发性的。

### 5.3.2 什么是模块

`module` 是包含 Jython 编码的一个文件并且它有 `.py` 扩展。没有另外的特殊语法或指令来要求使它成为一个模块。在任何以前的例子中，你曾把 Jython 代码放置在一个 `*.py` 文件中，你那时实际上正在创建一个模块。模块也作为 `.class` 出现。这些是被引入和编辑了的版本模块。如果它们的 `associated.py` 没变化，随后的引入就把这些字节编辑文件作为优化。当他们被引入或使用 Jython 的编译器 `jythonc` 或运用模块 `compileall.py` 时（发现在 Jython 的 Lib 目录），模块是字节编辑的。

众多的模块与 Jython 分布在一起，并且它们定位在 Jython 的 Lib 目录中。这些模块主要是 Python 模块，原来包含在 CPython 分布和 CPython 的文档设置中。然而，并非所有的 CPython 模块都与 Jython 同时工作。在 CPython 中的一些模块依靠用 C.Jython 写的编码，现在不能利用除 Java 或 Python 以外语言编写的模块，这些依赖于 C 的模块将不能工作。大多数 Jython 模块的文档是 Python 库参考，当前定位在 <http://www.python.org/doc/current/lib/lib.html>，和在由大卫·贝兹利编写的《Python Essential Reference》中（New Riders 出版社）都可用。那些独一无二或正在丢失的模块在那本书的附录 C 有描述。

用户定义的模块是那些 \*.py 文件，该文件是由你或其他人编写的，并没包括在配置中。这些文件能按照被发现和装载的顺序放置在 Jython 的 Lib 目录；然而，那通常使升级 Jython 版本毫无益处。对于这些文件更需要一个独立的地点，并且一个独立的地点要求在 sys.path 中附加一个项。本章的剩余部分，假设新编写的模块放置在 sys.path 列表中一个位置。最容易的是当前工作目录 sys.prefix + /Lib/site.python，它将要求像更早前描述的那样，把这加到 sys.path 列表。

### 5.3.3 特殊的模块变量

模块有特殊的变量或“魔术”般的变量。魔力是一个口头术语，经常在 Jython 和 Python 中归功于变量，有前行和拖尾下划线。在一个模块中，魔力的变量是 `_doc_`、`_file_`、`_name_` 及 `_all_`。

在一个模块中 `_doc_` 变量与函数的 `_doc_` 字符串类似。如果一个模块以一个字符串的文字开始，那么那个字符串文字成为模块的 `_doc_` 属性。这是 Jython 的自我文档机制的一部分。假定你需要创建一个 directory-syncing 工具。你将很可能开始把你的意愿归档于一个模块的 `_doc_` 属性：

```
# file: dirsync.py
"""This module provides tools to sync a slave directory to a master
directory."""
```

在把 `dirsync.py` 文件放在 `sys.path` 的某个位置时，你能在交互的解释器中看到结果：

```
>>> import dirsync
>>> print dirsync.__doc__
This module provides tools to sync a slave directory to a master
directory.
```

一个模块的 `_name_` 属性是模块名字。对于这点的例外是 `_main_` 模块。当 Jython 以一个提供作为命令行自变量的一个模块名字开始时，例如 `jython test.py`。在命令行上提供的模块考虑了 `_main_` 模块。在 Jython 模块中，这介绍了一个重要且有用的常规。

程序清单 5-1 显示出 `dirsync.py` 文件的一个更完全的版本。它实际上不是 sync 文件，但是程序清单 5-1 计算要求这样做的工作。在程序清单 5-1 中，最后的 if 语句是有用的模块约定。首先，检验程序清单 5-1，从解释一些使用的函数开始：

- `os.path.join`: 添加多重的路径元素到一条合法的路径中。
- `os.path.isfile`: 如果一条路径指向一个文件，就返回值 1，否则，就为 0。
- `os.path.isdir`: 如果一条路径指向目录，就返回值 1，否则，就为 0。
- `os.listdir`: 返回一张指定程序清单内容的表。
- `os.stat`: 返回包含文件信息的一张 10 元素表。
- `stat.ST_MTIME`: 是一个指明哪个 stat 列表元素是修正时间的数字。

#### 程序清单 5-1 Syncing 目录的模块

---

```
# file: dirsync.py
"""This module provides tools to sync a slave directory to a master
directory."""
```

```

from __future__ import nested_scopes
import os
from stat import ST_MTIME

def __getRequiredUpdates(master, slave):
    """getRequiredUpdates(master, slave) -> list of files that are old or
    missing from the slave directory"""

    needUpdate = [] # holds results
    def walk(mpath): # recursive function that does the work
        if os.path.isfile(mpath):
            spath = mpath.replace(master, slave)
            if (not os.path.exists(spath) or
                (os.stat(mpath)[ST_MTIME] > os.stat(spath)[ST_MTIME])):
                needUpdate.append(mpath.replace(master, ""))
        if os.path.isdir(mpath):
            for item in os.listdir(mpath):
                walk(os.path.join(mpath, item))
    walk(master)
    return needUpdate

def __getExtraneousFiles(master, slave):
    """getExtraneousFiles(master, slave) -> list of files in the slave
    directory which do not exist in the master directory"""

    extraneous = [] # holds results
    def walk(spath): # recursive function that does the work
        mpath = spath.replace(slave, master)
        if not os.path.exists(mpath):
            extraneous.append(spath)
        if os.path.isdir(spath):
            for item in os.listdir(spath):
                walk(os.path.join(spath, item))
    walk(slave)
    return extraneous

def sync(master, slave):
    updates = __getRequiredUpdates(master, slave)
    extras = __getExtraneousFiles(master, slave)
    for item in updates:
        # copy those files needing updated
        print os.path.join(master, item), "need copied to slave dir."
    for item in extras:
        # delete those files that are extras
        print item, "should be deleted."

if __name__ == '__main__':
    from time import time
    import sys
    t1 = time()
    sync(sys.argv[1], sys.argv[2])
    print 'Directory sync completed in %f seconds.' % (time() - t1)

```

在程序清单 5-1 中，最后的 if 语句测试模块的名字是否是 `_main_`。如果这是真，它意味着模块用命令 `jython dirsSync.py` 运行，并被期望像主脚本一样运作或类似于在 java 程序中从具有

`public static void main (字符串 args)` 特征符的方法中期望得到的一样。然而，如果另外的脚本引入这个模块，`_name_` 不是 `_main_` 并且关联的编码没执行。许多 Jython 模块使用这个约定，或它们能起到像 `stand-alone` 脚本和模块的作用，或当它们像一个 `stand-alone` 脚本运行时，以至于它们能自检。在程序清单 5-1 的情况下，脚本 `dirsSync.py` 能像 `standalone` 脚本一样运行，以决定 sync 要求的工作是主从目录，在以下命令行中是这样提到的：

```
python dirsSync.py "c:\path\to\master\dir" "c:\path\to\slave\dir"
```

变量 `sys.argv` 是当启动 Jython 时使用的、包含所有命令行参数的列表。作为主要脚本运行的实际脚本总是 `sys.argv[0]`，这样在程序清单 5-1 中，`sys.argv[0]` 是 `dirsSync.py`，并且 `sys.argv[1]` 和 `sys.argv[2]` 将分别是主从目录。运行 `dirsSync.py` 的 shell 命令和各自的 `sys.argv` 赋值将如下：

```
python dirsSync.py c:\mywork c:\mybackups
    argv[0]    argv[1]    argv[2]
```

程序清单 5-1 介绍另外的特殊命名约定：以两个下划线开始的函数。当 `from x import *` 语法被使用时，而以两个下划线开始的标志符没有引入，那这个标志符是特殊的。实际上，在 Jython 2.1 中，这应该适用于以一个或两个下划线开始的那些标志符，但是那些有一个下划线的标志符在被编写的时候，并没有适当工作，因此程序清单 5-1 使用两个下划线。在程序清单 5-1 中，用 `dirsSync.py` 模块来测试这个标志符，首先保证它在 `sys.path` 中，然后键入：

```
>>> from dirsSync import *
>>> dir() # look to see which names were imported
['__TIME__', '__doc__', '__name__', 'nested_scopes', 'os', 'sync']
```

`dir()` 函数的结果证实没有以两个下划线开始的名字被引入。

神奇的变量 `_all_` 进一步控制从一个模块引入的名单。变量 `_all_` 是包含模块及其导出名字匹配的列表。如果程序清单 5-1 有如下的一行，那出现在 `from dirsSync import *` 语句后的惟一名字将是 `sync`：

```
_all_ = ['sync']
```

### 5.3.4 什么是包

一个 Python 包是包含在一个目录树中的模块层次。除了 Python 包内容是模块，而不是类，其他与 Java 包有着一样的概念。包系统的一个优秀例子是在 Jython 的 `Lib` 目录中发现的 Jython 的 `pawt` 包。

图 5-1 是在一台 Windows 机器上的 Jython 安装目录列表。注意根据你的 Jython 的版本和你是否创建了早先讨论的 `site-python` 目录，你的目录、内容可能不

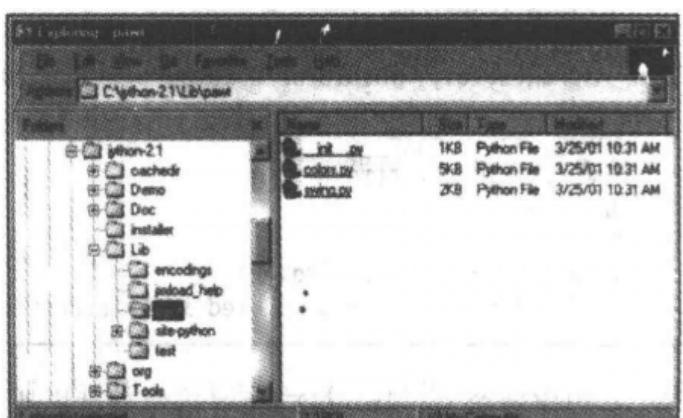


图 5-1 pawt 包

同。在 c:\python-2.1\ 的 Lib 目录就是分布在 Jython 的模块和包被存储的地方。在这个目录里，你将发现称作 pawt 的目录。然而它不只是另一个目录——它是一个 Jython 包。什么使它成为一个包？它包含一个将它初始化为文件的包。这个文件是 \_init\_.py。

当一个包被引入时，它的 \_init\_.py 文件被执行。因为 pawt 是一个包，你能用下列任何语句引入它的内容：

```
>>> import pawt.colors
>>> import pawt.swing
>>> from pawt import colors, swing
>>> from pawt import *
```

你也能用语句引入包本身：

```
>>> import pawt
```

关于 Python 包的惟一事情是一个引入语句实际上初始化所有的父包。这意味着如果你有一个层次，例如包 A 包含包 B，而包 B 含有模块 C，模块 C 的实际引入运行 A.\_init\_.py，接着运行 B.\_init\_.py，然后引入模块 C。程序清单 5-2 展示了这种情形的实现。三个文件 A.\_init\_.py、B.\_init\_.py 和 moduleC，每个仅仅打印一条消息通告它们的取值点。程序清单 5-2 在交互式解释器中引入 moduleC 来继续这个例子。显示在屏幕的消息证实父包的初始化。

#### 程序清单 5-2 包初始化

---

```
# file sys.prefix/site-python/A/__init__.py
"""Root package in a nested package demonstration"""
print "Initializing package A"

# file sys.prefix/site-python/A/B/__init__.py
print "Initializing package B"

# file sys.prefix/site-python/A/B/moduleC.py
print "Executing module 'moduleC'"
import moduleD

# file sys.prefix/site-python/A/B/moduleD.py
print "Executing module 'moduleD'

>>> # examine packages A, B, and moduleC interactively
>>> import A.B.moduleC
Initializing package A
Initializing package B
Executing module 'moduleC'
Executing 'moduleD'
```

---

程序清单 5-2 也包括一个 moduleD.py 文件，以示范一个模块在没有完全限定名字的情况下

怎么能引入同属包模块，意味着 moduleC 仅用有 moduleD 的名字能引入 moduleD。然而，在不同包中的模块必须被限定。

Jython 自我文档约定扩展到包，而包中如果一个 package's\_init\_.py 文件以字符串文字开始，那么包 \_\_doc\_\_ 属性赋给那个字符串作为值。注意到在程序清单 5-2 的文件 sys.prefix/site.python/A/\_init\_.py 里有一个字符串文字。如果我们再访问这个包，我们能看到 \_\_doc\_\_ 属性是怎么在这个包中被赋值的。你必须重启在程序清单 5-2 和其下列之间的解释器为包的初始化而如下工作：

```
>>> import A
Initializing package A
>>> A.__doc__
'Root package in a nested package demonstration'
```

一个包的 \_init\_.py 文件也能包含神奇变量 \_\_all\_\_，它指明当 from package import \* 语句使用时包所导出的名字。

一个被引入的包与一个模块类似，除开包有一个附加的特殊变量： \_\_path\_\_. \_\_path\_\_ 变量就正如你猜测的那样：到指定包的系统路径。检测 pawt 包显示出 \_\_path\_\_ 变量：

```
>>> import pawt
>>> pawt.__path__
['C:\\\\jython 2.1a1\\\\Lib\\\\pawt']
```

## 5.4 Java 的 package.class 层次

Java 包和类的装载使用像引入 Python 模块一样的语法，但是有一些额外的信息要求充分理解 Java 的装载。Java 使用 classpath 定位且装载包和类。Jython 的引入机制确实检查了 Java classpath。当运行 Jython 时，classpath 从 classpath 环境变量中取出。为了把 jar 文件 zxJDBC.jar 加到 Jython 使用的 classpath，你想用以下 shell 命令开始 Jython 前，设定 classpath 环境变量：

```
# for Windows users at a dos prompt, or within a *.bat file
>set CLASSPATH=c:\\path\\to\\zxJDBC.jar
>jython

# for *nix users at the bash prompt
>export CLASSPATH=/path/to/zxJDBC.jar
>jython
```

在任何一个平台上，保证在 classpath 中的路径遵循指定路径平台的路径准则，对于 Windows 是 “:\\” 和 “;”，而对于 \*nix 则是 “/” 和 “:”。

Jython 引入语句的所有四行命令与 Java 类和包甚至类内容一起工作。当引入 Java 项时，名称绑定规则对于引入仍保持原样。这意味着引入 java.util.Vector 仍然绑定最左名 java，而不用 java 绑定最右边名字的行为。这也意味着 from X import Y 语法能引入 X 内容，而那不是类或包。

程序清单 5-3 给出了引入 Java 类的很多引入语句，甚至包括类的内容以表明引入行为是怎么与 Java 元素工作的。注意程序清单 5-3 要求 Java 1.2（仅仅用 Sun JDK 测试）。

## 程序清单 5-3 引入和使用 Java 项

---

```
>>> import java.applet
>>> # inspect name with dir() to show leftmost binding
>>> dir()
['__doc__', '__name__', 'java']
>>>
>>> import java as j
>>> dir()
['__doc__', '__name__', 'j', 'java']
>>>
>>> # 'from X import y' even works on class contents such as...
>>> from java.security.AlgorithmParameters import getInstance
>>> from java.applet.Applet import newAudioClip as nac
>>> dir()
['__doc__', '__name__', 'getInstance', 'j', 'java', 'nac']
```

---

你不仅能看到 Jython 的引入灵活性适用于 Java 类和包，而且也能看到它允许有选择地引入类内容。

classpath 不是用来定位并装载 Java 类和包的惟一路径。他们也能从 sys.path 被定位和装载。在 Jython 中，sys.path 作为 Java's classpath 的扩展运行。这引起不能区别哪些项目是 Java 以及哪些项目是 Python 的一些问题，并且在分布于 classpath 和 sys.path 的那些包中，不把自己看作包的子集。

程序清单 5-4 显示一个最小的 Java 类。这个例子的焦点是显示出一个 Java 类怎么从 Python sys.path 中装载。程序清单 5-4 的第一半是 Java 类 mjc（为最小的 Java 类）。在程序清单 5-4 试用这个例子，保留 Java mjc 类到文件 mjc.java 并且用 javac 编辑它。然后把编辑的 mjc.class 文件放在本章一直在使用的用户定义的模块目录（sys.prefix/Lib/site-python）中。程序清单 5-4 交互式解释程序部分打印 Java 的 classpath 证实 mjc.class 不在它之中，然后打印 sys.path，再引入 mjc。在这种情况下，Java 类从 sys.path 上定位且装载。

## 程序清单 5-4 从 sys.path 装载 Java 类

---

```
// file mjc.java
public class mjc {
    String message = "This is all I do- return a string.";
    public mjc() {}

    public String doSomething() {
        return message;
    }
}

>>> from java.lang import System
>>> System.getProperty("java.class.path")
'C:\\\\jython 2.1a1\\\\jython.jar;c:\\\\windows\\\\desktop'
>>> import sys
>>> print sys.path
['', 'C:\\\\WINDOWS\\\\Desktop\\\\.', 'C:\\\\jython 2.1a1\\\\Lib', 'C:\\\\jython 2.1a1',
 'c:\\\\jython 2.1a1\\\\Lib\\\\site-python']
```

---

---

```
>>> import mjc
>>> m = mjc()
>>> m.doSomething()
'This is all I do. return a string.'
```

---

事情不总是那么简单，尽管 \*.jar 和 \*.zip 文件不在编写时从 sys.path 装载。也有 Java 包没被适当辨认出的情况。一个辨认失误的例子是当一个 jar 或 zip 文件的扩展并没有全部是小写的时候。这在一定程度上将被改正，要不是 Jython 2.1a1 版本和更早的版本，那些文件应该在缩小的扩展中重命名。从 sys.path 装载文件 \*.jar 和 \*.zip 最后也将被增加。

程序清单 5-5 使用 Java 装载设备来播放网络音频资源。注意一些防火墙安装程序为程序清单 5-4 带来了问题。

#### 程序清单 5-5 在 Jython 中使用 Java 类播放一个声音

---

```
>>> from java import net
>>> from java.applet.Applet import newAudioClip
>>>
>>> url = net.URL("http://www.digisprings.com/test.wav")
>>> audio = newAudioClip(url)
>>> audio.play()
```

---

Java 用 Class.forName 方法管理动态装载。这在 Jython 仍然是有用的，并且惟一的捕捉是大多数人们忘记 Java.lang 的内容没在 Jython 中被缺省引入。与类的显式引入或者甚至静态方法 forName 相比较，动态装载起到同样的功能。这是引入 Java.util.hashtable 类的一个例子：

```
>>> from java.lang.Class import forName
>>> forName("java.util.Hashtable")
<jclass java.util.Hashtable at 2105495>
>>>
>>> # OR
>>> from java.lang import Class
>>> Class.forName("java.util.Hashtable")
<jclass java.utilc.Hashtable at 2105495>
```

## 5.5 重载

如果你引入了一个模块，但是那个模块发生了改变，你就能用内置重新装载功能重装那个模块。当在交互式解释程序中工作且同时编辑一个模块时，这是最有用的。重新装载功能语法如下：

```
reload(modulename)
```

import 语句允许用 from X import Y 语法引入模块全局的实体，而重新装载功能不会这样。重新装载仅仅为模块和包工作。这是一些要证实的 reload 行例子：

```
>>> from pawt import colors
>>> reload(colors)
<module pawt.colors at 860800>
>>> import dumbdbm
```

```
>>> reload(dumbdbm)
<module dumbdbm at 5654181>
```

重装 Java 包和类是更困难的。Samuele Pedroni 是一个程序员和 Jython 工程成员，他为 Jython 工程做了一个意义重大允许重装 Java 实体的试验包。Samuele 的 jreload 模块是能够使 Java 重装的。引入作为 jreload load 设置部分的 Java 包和类是惟一重新装载的。因为所有包和类的访问是经由装载设置的实例，这与另外的引入大相径庭。

编写一个装载设置，你必须使用 jreload.makeLoadSet 函数。这个功能要求两个自变量：第一个是一个名字，它是你想要赋值给装载设置任意的字符串。第二个自变量必须是 PyList，它包含代表你想要在装载设置中包括的项目的 classpath 入口的一个字符串列表。假定你使对于 javax.servlet 包的 jar 文件定位在 c:\web\tomcat\lib\servlet.jar 中（这个 jar 文件不与 JDK 同时运行；它必须被单独下载——通常作为 Tomcat 网服务器的部分定位在 http://jakarta.apache.org）。为包含在文件 servlet.jar 中的包创建装载设置如下所示：

```
>>> import jreload
>>> ls = jreload.makeLoadSet('servlet classes',
                               ["c:\\web\\tomcat\\lib\\servlet.jar"])
>>> dir(ls.java)
['__name__', 'servlet']
```

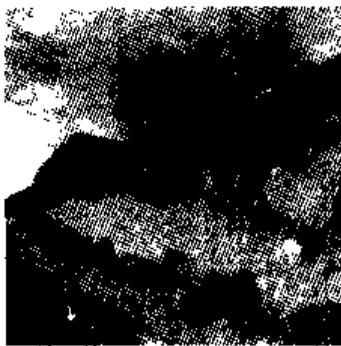
如果你想要从这个装载设置重装包和类，可能因为你已重新创建了 servlet.jar 文件，使用如下：

```
>>> ls = reload(ls)
```

当这个例子对在 servlet.jar 文件内的包是特定的时，jreload 已经不在 classpath 或 sys.path 上与任何类或包工作。用 jreload 试用另外的包或类，你必须把第二个自变量改变到 makeLoadSet 方法以包括你想引入的包的位置。把第一个自变量改变到 makeLoadSet 也是好主意，但它只是集合的一个任意名字并不影响它的功能。在 makeLoadSet 的第二个自变量中的类似 classpath 的列表是 jreload 机制的核心部分。

对于 jreload 有一些防止误解的说明。首先，类似 classpath 入口的列表提供给 makeLoadSet 函数（第二个自变量）不应该是已经在 sys.path 或 classpath 发现的入口。第二，Java 内部的类产生意外的结果。





## 第6章 类、实例和继承

这本书假定读者有一些前述的 Java 知识，或者读者从其他的资料获得了一些 Java 知识。如果你知道 Java 和对象，那么面向对象编程（OOP）是什么就不再需要再介绍了。术语类、实例、封装、多态和继承都渗入到 Java 讨论之中，因此这里只简单地提及一下。本章以封装、抽象和信息隐藏开始，接着是对 Jython 类的定义。因为有了先前的 OOP 知识，快速开始并不意味着 Jython 类和实例与 Java 类的相同。差别是肯定存在的，在 Jython 里用 Jython 类和 Java 类两者创建了当你阅读时应注意的二元性。这种二元性意味着优先选择一个类的特性可能对 Jython 和 Java 是惟一的，因此我们总要考虑类出现的上下文。幸运的是 Jython 并不改变 Java 类的特性和行为，因此关于上下文的混淆是不可能的。

在描述怎样定义 Jython 类和讨论 Jython 类属性后，再讨论关于 Jython 的继承、方法、重载、构造函数和析构函数的细节。为了完善对 Jython 类的介绍，本章用许多 Jython 类例子来总结。

### 6.1 封装、抽象和信息隐藏

本节主要是关于名字损坏和电子栅，但是让我们从更传统的对象术语开始。封装是多实体的结合或聚合来产生一种新的、常常有较高秩序的实体，例如类。类把数据和相关功能性封装到一个单对象。类用简化接口和隐藏实现的细节方式组织功能，提供抽象。继续用信息隐藏——仅把简单名字里他们需要看见的显示给用户，来简化复杂性。在其他的语言中，Java 采取另外的措施，用像 `private` 和 `protected`——电子栅方法这样的属性权限修饰语来强制抽象化。尽管当面临软件复杂性时电子栅有直接的吸引力，但其他的人主张这样的严格是不必要的，电子栅本应仅仅是对冒险者继续冒险通过的警告标记。这种类推适合把 Java 的严格方法与 Jython 的保密机制进行对照，Jython 的保密机制是非常开放的，与 Java 的严格方法相反。

程序员可以看到和修改 Jython 类里大部分的东西，但是对方法名和变量名的 Jython 约定是保密的。以下划线开始的属性是类内部的。这仅是按照约定，而不是不可更改的。你可以泰然地调用以下划线开始的属性，但是在你冒险时约定会指示你这样做。以两个下划线开始的 Jython 变量是更接近于加强保密的措施，但它仅是通过模糊来进行保密的。上面所指的模糊就是名字混淆。Jython 混淆了 Jython 类和以两个下划线开始的实例属性的名字来鼓励程序员注意它们的保密。实际的混淆就是增加一个下划线和类名以属性名开始。在类 A 里的 `变量_var` 变成 `_A_var`。混淆是保护的结束——没有更进一步的强制。通过这个被混淆的名字可以访问属性，如下例：

```
>>> class A:  
...     __classVar = 'A class variable designated as private'  
...     def __init__(self):
```

```

...     self._instVar = "An instance variable designated as private"
...     def __privateMethod(self):
...         print 'This method is designated as private.'
...
>>> inst = A()
>>> inst._A__classVar
'A class variable designated as private'
>>> inst._A__instVar
'An instance variable designated as private'
>>> inst._A__privateMethod()
This method is designated as private.

```

换句话来说，Jython 就是没有必要这样辛苦地工作来保护程序员。Jython 保密机制的全部分类加起来就是“你不要使用，除非你确定你正在做什么”的一个下划线和“你不要使用，除非你真正确定你正在做什么”的两个下划线。

另一个命名约定是两个前部和后部的下划线。这个约定指定特殊的和不可思议的属性，该属性在 Jython 里有特殊的意义，例如 `_doc_` 属性。因此使用有两个前部和后部下划线的变量是不明智的，除非实现了预定义的特殊的属性。Jython 的特殊类属性将在下一章详述。

## 6.2 定义 Jython 类

Jython 类定义出现在模板、交互式解释程序或动态产生的代码里。类定义语法如下：

```
"class" class_name[(bases)]:
    code block
```

名字 `class` 是开始定义语句。类名是任何合法的 Jython 标志符。下面的类名是圆括号里的基类的可选列表。冒号表示类语句的结束和相关代码程序块的开始。一个几乎空的 Jython 类如下：

```

>>> class emptyClass:
...     """This is a near-empty Jython class"""
...

```

剖析这个空类强调了一个类定义的重要部分。单词 `class` 开始类定义。这个类的标志符是 `emptyClass`。这个类不是任何其他类的派生类，因为没有圆括号和它的标志符后没有列表。在 Jython 里，代码程序块遵守缩排定界，以与类语句是同一缩排级的代码程序块开始。我们也看见了类不是完全空的，因为它有文档字符串。像函数和模板一样，在类代码程序块开始的文字字符串变成了类的 `_doc_` 属性或 Python 术语里的 `docstring`。

类是可调用的对象，调用一个类返回一个实例对象。为了调用我们的例子派生类 `emptyClass`，在它的名字后增加一个空的圆括号；`emptyClass` 不需要有一定数量的自变量的构造函数，因此一个空的变量列表就足够了。为了证实类 `emptyClass` 和它返回的实例对象之间的差别，使用内置函数 `type()`，如下所示：

```

>>> c = emptyClass()
>>> type(emptyClass)
<jclass org.python.core.PyClass at 3838683>
>>> type(c)
<jclass org.python.core.PyInstance at 7833967>

```

空类有什么好处呢？在 Java 里并不很多，但 Jython 允许对 Jython 类作随机改变。除了`_doc_`外`emptyClass`例子缺少属性。然而，因为 Jython 的动态特性，动态地把属性赋给`emptyClass`或它的实例是可能的。这个例子演示如下：

```
>>> emptyClass.z = 30
>>> c = emptyClass()
>>> c.x = 10
>>> c.y = -20
>>> c.__doc__ = "This was a near-empty Jython class"
>>> c.x
10
>>> c.y
-20
>>> c.z
30
>>> c.__doc__
'This was a near-empty Jython class'
```

这样随机增加到 Java 类是不能运行的：

```
>>> from java.applet import Applet
>>> a = Applet()
>>> dir(a)
[]
>>> a.a=10
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError: can't set arbitrary attribute in java instance: a
```

### 6.3 Jython 类和实例属性

虽然调用一个类返回该类的一个实例是很清楚的，但是从那时起 Jython 的类-实例关系的清晰是欠缺的。类和实例两者都有另一个独特的属性（Python 文献使用属性就像 Java 文献使用成员一样）。实例属性是独特的，并从同名的类属性分离出来。Java 用`static`关键字来指明类成员，但 Jython 没有任何这样的关键字。因为 Jython 缺少显式的修饰语来指明属性对实例是指定的类还是惟一的类，确定它的准则需要说明。下列程序清单仅适合于 Jython 类。在 Jython 里使用 Java 类和实例不改变 Java 类和实例成员的行为。创建 Jython 类和实例属性的准则如下：

- 所有方法是实例方法。
- 在类里定义数据对象使数据对象成为类属性，而不是在方法里。

```
class A:
    x = "This is a class attribute"
```

- 赋值给一个用类名限定的或加了类名前缀的数据对象，使该数据对象成为类属性。

```
>>> class A:
...     def aMethod(self):
...         A.x = "This is a class attribute"
```

- 赋值给一个用实例引用或实例名限定的或加了实例引用或实例名前缀的标志符，使该标志符成为实例属性。

```
>>> class A:
...     def aMethod(self):
...         "Hint: 'self' is the instance ref- more on this later."
...         self.x = "This is an instance attribute"
```

访问类和实例属性的准则如下：

- 用类名限定的标志符引用类属性。

```
>>> class A:
...     x = 'Class attribute'
...     def aMethod(self):
...         self.x = 'Instance attribute'
...         print A.x
...
>>> test = A()
>>> test.aMethod()
Class attribute
```

- 用实例名限定的标志符可以引用实例属性或类属性。如果实例属性存在，则它被使用。实例属性影响任何同名的类属性或隐藏访问性。如果那个名字没有任何实例属性存在，则标志符引用类属性，但是第二个例子增加了一个同名的实例属性来显示它在查寻顺序是怎样位于类属性之前的。

```
>>> # Example 1. Only a class attribute "x" exists
>>> class A:
...     x = 'Class attribute'
...     def aMethod(self):
...         print self.x
...
>>> test = A()
>>> test.aMethod()
Class attribute
>>>
>>> # Example 2. Both a class and instance attribute "x" exist
>>> class A:
...     x = "Class attribute"
...     def aMethod(self):
...         self.x = "Instance attribute"
...         print self.x
...
>>> test = A()
>>> test.aMethod()
Instance attribute
```

惟一要记住的技巧部分是 `self.x` 引用类属性，但赋值给 `self.x` 创建影响类属性的那个名字的新实例属性。这样的一个优点是不可变类属性能作为同名实例属性的初始缺省值。

回顾列表后，似乎有什么被丢失了？Python 程序员回答“没有什么”，而 Java 程序员将不可避免地回答“类静态方法”。当许多增加这个功能的机制赢得了在 Jython 类里对类静态方法的内置支持时，却没有任何标准。尽管这种思想重复出现，也没有任何要求增加这样的支持，因为模板级函数就足够了。

类的所有实例共享类属性。另一方面，实例属性对一个类的每一个实例是惟一的。类 my-

Files 包含了一个如下所示的类属性定义：

```
>>> class myFiles:
...     path = "/home/rbill"
```

因为 path 是一个类属性，类 myFiles 的所有实例共享这个相同的 path 对象。然而，如果一个实例创建了一个称做 path 的实例属性，它不仅对同名的类属性是惟一的，而且对 myFiles 类的所有其他实例也是惟一的。为了证实 path 是静态类，需要创建两个实例，改变路径，然后检查每一个实例里 path 的值：

```
>>> c1 = myFiles()
>>> c2 = myFiles()
>>> myFiles.path = "c:\\windows\\desktop"
>>> c1.path
'c:\\windows\\desktop'
>>> c2.path
'c:\\windows\\desktop'
```

证实实例变量的惟一性需要三个实例。记住赋值给一个用实例名限定的或加了实例名前缀的标志符，创建一个实例属性。在类定义外这也同样是正确的，并在这里用做例子的实例变量：

```
>>> class myFiles:
...     path = "/home/rbill"
...
>>> c1 = myFiles()
>>> c2 = myFiles()
>>> c3 = myFiles()
>>> c2.path = "c:\\windows\\desktop" # creates a new instance attrib
>>> c3.path = "/export/home/solaris" # creates a new instance attrib
>>>
>>> c1.path # class attrib
'/home/rbill'
>>> c2.path # instance attrib unique to c2
'c:\\windows\\desktop'
>>> c3.path # instance attrib unique to c3
'/export/home/solaris'
```

在 Jython 类里定义实例方法与写一个 Jython 函数相似。Jython 方法使用非常灵活的已对 Jython 函数描述了的参数方案。Jython 可使用有序的自变量和关键字自变量，参数可以有缺省值，你可考虑到不知道参数的个数，在方法特征符里用 \* 和 \*\* 来表示。区别集中在实例引用上。在 Java 里，变量引用包含它特定的实例，这是假设或 Java 里隐式的，但是 Jython 并不为实例对象进行这样的隐式名绑定。相反地 Jython 类里每一个方法必须在参数列表第一位置为实例对象指定一个标志符。当这第一位置是一个随机标志符时，单词 self 已经成为了标准惯例。一个创建和打印实例变量的实例方法如下所示：

```
>>> class test:
...     def imethod(self):
...         self.message = "This is an instance variable"
...     print self.message
```

```

...
>>> t1 = test() # create the instance
>>> t2 = test() # create a second instance for comparison
>>> t1.imethod()
This is an instance variable
>>>
>>> # To prove the "message" data is unique to only instance "t1"...
>>> t1.message
'This is an instance variable'
>>> t2.message # imethod not called- so t2.message is not defined
Traceback (innermost last):
  File "<console>", line 1, in ?
AttributeError: instance of 'test' has no attribute 'message'

```

你将注意到 imethod 的方法定义与函数定义相同，除了需要增加的 self 参数（或其他的合适的标志符）来满足实例职责外。在类的代码程序块里，对实例属性的引用必须用 self 限定，没有任何对包含在对象里的其他属性的含蓄的理解，在 Java 也是这样。现在我们考虑带有 methodA 和 methodB 两个方法的类。因为 self 在属性查找中必须是显式的，任何在 methodB 里对 methodA 的引用必须使用 self.methodA。下面的 Calendar 类证实了在实例引用 self 前缀：

```

>>> class Calendar:
...     def setDay(self, day):
...         self.day = day
...     def setMonth(self, month):
...         self.month = month
...     def getDay(self):
...         return self.day
...     def getMonth(self):
...         return self.month
...     def getDayandMonth(self):
...         return self.getDay() + " " + self.getMonth()
...

```

设置和检索实例变量 day 和 month 都在 Calendar 类里使用 self 前缀。注意其他实例方法的使用也需要 self 前缀来限制其他方法的使用，getDay 和 getMonth 就是这样做的。

方法有它们自己的局部名空间，就像函数一样，因此仅实例里和方法的局部名空间外的变量需要 self 前缀。例子如下：

```

>>> class test:
...     def methA(self):
...         data = 10 # This is local to methA
...         self.data = 20 # this is a separate instance variable

```

类和实例属性被存储在对象的字典里或 dict\_ 变量里。这意味着在类 A 里的属性 B 通过 A.\_dict\_[B] 是可用的。内置函数 vars() 被设置来返回对象的 dict\_ 属性，在开发对象中是有用的。

## 6.4 构造函数

constructor 是一个初始化类的实例。Jython 里的构造函数是在 Jython 类里定义的称做 \_\_init\_\_ 的方法。这个 \_\_init\_\_ 方法的参数与其他方法的参数看起来相似，\_\_init\_\_ 方法参数的第一个位置属于通

常称做 `self` 的实例对象。构造函数可以不返回值，但不返回值不同于 `None`。返回语句在构造函数里是允许的，并在完成整个 `_init_` 程序块之前，万一某一条件应返回控制，返回语句是有用的。如果出现返回，它必须是简单的 `return` 或 `return None`。创建了一些实例变量的构造函数的 Jython 类如下所示：

```
>>> class contact:
...     def __init__(self):
...         self.name = ""
...         self.phone = ""
...
...
```

在构造函数中定义的参数的数量决定了当调用一个类时有多少个自变量出现。如果构造函数需要两个参数加 `self`，则当调用类时，两个自变量必须出现。如果 `contact` 类需要一个名字和电话号码作为构造函数的参数，则如下所示：

```
>>> class contact:
...     def __init__(self, name, phone):
...         self.name = name
...         self.phone = phone
...
...
```

这个 `contact` 类使用构造函数来初始化名字、该名字的电话变量和电话构造函数参数。创建这个新的 `contact` 类现在需要两个参数，如下所示：

```
>>> c = contact("Someone's name", "555-555-1234")
>>> # Or with keyword arguments
>>> c = contact(phone="555-555-1234", name="Someone's name")
```

提供缺省值使得构造函数参数可选。Java 可以有多个构造函数来实现不同的参数。Jython 不能，但要考虑参数的更大灵活性，包括在像构造函数这样的方法里的缺省值。可选的构造函数参数如下所示：

```
>>> class contact:
...     def __init__(self, name="", phone=""):
...         self.name = name
...         self.phone = phone
...
...
```

现在调用 `contact` 类可以采用以下形式的任何一种：

```
>>> c = contact()
>>> c = contact("Someone's name")
>>> c = contact("Someone's name", "555-555-1234")
>>> c = contact(name = "Someone's name")
>>> c = contact(phone = "555-555-1234")
>>> c = contact(name="Someone's name", phone="555-555-1234")
```

构造函数的通常用途是建立数据库连接。下面的例子包含了一个构造函数，该构造函数示范在对象实例化时建立一个数据库连接。注意这个例子依赖 MySQL 数据库和它的相关 Java 驱动程序，从网站 <http://www.mysql.org/> 可以得到，它们两者都是可用的。

```
>>> import java
>>> import org.gjt.mm.mysql.Driver
>>> from java.sql import DriverManager

>>> class DBAdapter:
...     def __init__(self, dbname):
...         conString = "jdbc:mysql://localhost/%s" % dbname
...         self.db = DriverManager.getConnection(conString, "", "")
...         print 'DB connection established'
```

在 DBAdapter 类被实例化后，任何实例方法可通过实例变量 self.db 访问数据库连接。因此，一旦具体到一个实例时将引起耗时的连接处理。

构造函数常常需要调用超类构造函数。为了这样，使用跟有\_init\_方法的类名，不管参数是什么，使第一个参数为 self。在 Jython 里，语法如下所示：

```
Superclass.__init__(self, otherArgs)
```

更多详细的例子是派生 java.net.Socket 类。Java 里的 Socket 类没有任何像\_init\_这样的方法，但采用同样的语法。Jython 内部处理要调用哪个构造函数。下面的例子演示了派生 Socket 类。注意这个例子假定你有活动的网络连接。

```
>>> from java.net import Socket
>>> class SocketWrapper(Socket):
...     def __init__(self, host, port):
...         Socket.__init__(self, host, port)
...
>>> s = SocketWrapper("www.digisprings.com", 80)
```

用来初始化超类的语法并不仅仅限制在 Java 超类。初始化 Jython 超类也用同样的方法，考虑到它的构造函数真正地是\_init\_，初始化 Jython 超类也更有意义。

## 6.5 终结器和析构函数

终结器和析构函数是一个当对象被垃圾收集器收集时被调用的方法。Jython 使用 Java 的垃圾收集器 (gc)，因此当 Jython 对象不可达和内存需要整理时，那么 JVM gc 线程回收对象。当在 Jython 里使用终结器时，有两件事要记住：第一，它们引起性能损耗，第二，没有任何方法来告诉你对象什么时候或是否被终结。

Jython 终结器是一个称做\_del\_的实例方法。这个方法如下所示：

```
>>> class test:
...     def __del__(self):
...         pass # close something, or clean up something here
...
```

“构造函数”一节示范了在构造函数里创建数据库连接，但为了真正地使类完成，该数据库连接应该用终结器或用一些其他显式调用的方法关闭。下面例子示范了在对象终结器里关闭数据库连接，但是这样做时会有一些警告，并描述在下面的例子里：

```
>>> import java
>>> import org.gjt.mm.mysql.Driver
>>> from java.sql import DriverManager
```

```
>>> class DBAdapter:
...     def __init__(self, dbname):
...         conString = "jdbc:mysql://192.168.1.77/%s" % dbname
...         self.db = DriverManager.getConnection(conString, "", "")
...         print "DB connection established"
...     def __del__(self):
...         if not self.db.isClosed():
...             self.db.close()
...
...
```

Java 的垃圾回收器、数据库资源的特性和数据库特许常常与上面的方法相冲突。因为 Java 虚拟机和这样的 Jython 在垃圾回收器回收对象时并没有为之提供任何保证，即使是在回收没用对象的时候也是如此，所以依赖终结器是危险的。如果一个应用创建了许多与数据库连接的对象，该连接是用终结器关闭，那么在回收这样的对象时任何的延迟都会浪费有限的数据库资源，但它总是人们关注的。前述的例子阐述了终结器，但它不是管理数据库连接的谨慎的例子。关于数据库更多的、包括更加智能的资源管理将在第 12 章“服务器端网络编程”中阐述。

## 6.6 继承

我们已经看见了在 Java 类和 Jython 类里设置随机属性的能力是不同的。在 Java 类和 Jython 类里 self 变量的显式使用又是另一个不同。两者第三个不同点是在较早出现的定义 Jython 类里。注意 Jython 类的定义允许一列基类。单词“列”的使用会使那些习惯于 Java 的单继承执行的人不满。单继承意味着 Java 类只能从单基类中继承。Java 类能够实现多接口，但是只能派生一个单个 Java 类。另一方面，Jython 允许多重继承，但还是必须遵守 Java 的约束。这又介绍了在 Jython 里发现的 Java-Jython 二元性。

Jython 的继承能力依赖于超类的类型。因为 Java 执行单继承，所以 Jython 可以仅从单个 Java 类继承；然而，因为 Python 允许多重继承，Jython 可以从多 Python (Jython) 类继承。这种二元性使得 Jython 独特。它不完全像 Python、C++ 和允许多重继承的其他语言，因为 Java 超类的数量是受限制的。另外，它也不完全像 Java，因为它可以从多 Jython 类继承。尽管 Jython 受限于单个 Java 基类，Jython 可以从连同多 Jython 类（和 Java 接口）的单个 Java 类中多重继承。接着是关于从 Jython 类、Java 接口和 Java 类继承的详述。

### 6.6.1 派生 Jython 类

Jython 类除了能从一个、两个或许多 Jython 基类中继承外不需要从任何其他的类继承。在类定义里，定义基类或超类是在类标志符之后的圆括号里。如果你希望从 threading 模板里的 Thread 类里继承和从 sched 模板里的 scheduler 类里继承，你可以用：

```
>>> from sched import scheduler
>>> from threading import Thread
>>> class threadedScheduler(scheduler, Thread):
...     pass
...
...
```

从多类继承允许你使用任一基类的方法或重定义任一基类的方法。有两种方法来使用或调

用在基类里定义的子孙类里的方法。一是使用基类名和用点标注法表示的具有 `self` 实例对象作为第一个自变量的方法名。

```
>>> class parentClass:
...     def parentMethod(self):
...         print "Parent method called"
...
>>> class child(parentClass):
...     def callSuper(self):
...         parentClass.parentMethod(self)
...
>>> c = child()
>>> c.callSuper()
Parent method called
```

上面示范用当前实例对象作为参数显式调用一个超类，有两个目的：第一，怎样初始化 Jython 里的超类；第二，即使超类的方法在当前类被重定义也允许你调用它。下面的例子用修改的派生类示范了后者：

```
>>> class parentClass:
...     def parentMethod(self):
...         print "Parent method called"
...
>>> class child(parentClass):
...     def parentMethod(self):
...         print "Overridden parent method"
...     def callSuper(self):
...         parentClass.parentMethod(self)
...
>>> c = child()
>>> c.callSuper()
Parent method called
```

调用超类方法的其他途径是使用 `self` 前缀。为了调用超类的方法 `B` 用 `self.B()`，然后通过对象属性查找机制来定位方法。假定方法 `B` 不在子实例：

```
>>> class Super:
...     def B(self):
...         print "B found in superclass"
...
>>> class A(Super):
...     def callSuper(self):
...         self.B()
...
>>> test = A()
>>> test.callSuper()
B found in superclass
```

对用 `self` 前缀限制的那些标志符的属性查寻在当前实例开始，按照超类在类定义里的次序依次进行。因此改变超类的次序就有可能找不到它的属性（假定多超类定义了同样的属性）。一个这样的例子如下：

```
>>> class SuperA:
...     def A(self):
```

```

...     print "Test method from SuperA"

...
>>> class SuperB:
...     def A(self):
...         print "Test method from SuperB"

...
>>> class C(SuperA, SuperB):
...     def callSuper(self):
...         self.A()

...
>>> test = C()
>>> test.callSuper()
Test method from SuperA
>>>
>>> class C(SuperB, SuperA):
...     def callSuper(self):
...         self.A()

...
>>> test = C()
>>> test.callSuper()
Test method from SuperB

```

### 6.6.2 派生 Java 接口

在 Java 里，你实现了接口而不是派生了它。然而在 Jython 或 Python 2.1 版本和早期版本里目前还没有相等的显式的接口。在 Jython 和 Python 里，接口或协议在目前仅仅是隐含的。这就意味着在 Jython 里实现 Java 接口，你可把接口当做基类来使用同样的语法。就像 Java 能实现多接口一样，你可以从多 Java 接口中继承。多 Jython 类和多 Java 接口的结合也是可以接受的：

```

>>> from java.text import CharacterIterator
>>> from java.io import Serializable
>>> from java.io import DataInput
>>> from threading import Thread
>>> class test(CharacterIterator, Serializable, DataInput, Thread):
...     pass
...
>>> t = test()

```

### PEP 是什么

PEP 是 Python Enhancement Proposal。PEPs 为 Python 或这样的 Jython 的设计和发展提供文件。请求新的特性发表在 PEPs 里，目前有两个有趣的关于接口的 PEPs。PEP 第 245 项为 Python 描述了一个接口语法，以便定义一个与 Java 接口相似的接口将在 Jython 本地化。关于 PEP 245 更多的可用信息在网页 <http://www.python.org/peps/pep-0245.html>。另一个 PEP 是关于“对象适应性”的提议和它假定一个对象不必要知道当写它时它支持哪个协议，但能回答请求“你支持这个协议（接口）吗？”。这将需要介绍一个新的 `adapt` 函数、`_adapt_` 方法和 `_conform_` 方法。“适应性”提议是 PEP 的 246 项，位于网页 <http://www.python.org/peps/pep-0246.html>。

这两个提议中的任何一个都影响将来 Jython 使用 Java 接口的方式；然而在这点上，

两者中的任何一个都是不可接受的。在 Jython 和 Python 2.2 版本查找关于这个的信息。关于所有 PEPs 更多的信息是在网页 <http://python.sourceforge.net/peps/> 中。

### 6.6.3 派生 Java 类

因为 Java 执行单继承，Jython 也受限于至多一个作为基类的 Java 类。企图从多余一个那样的 Java 类中继承会产生一个异常：

```
>>> from java.applet import Applet
>>> from java.util import Vector
>>> class a(Applet, Vector):
...     pass
...
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError: no multiple inheritance for Java classes: java.util.Vector and
java.applet.Applet
```

然而从 Java 类继承的 Jython 类也能从接口和 Jython 类继承。卑鄙一点的方法是通过使用隐藏在 Jython 类中的 Java 类和别的 Java 类来伪装多重继承，但这种方法并不有效。

```
>>> from java.awt import Checkbox
>>> from java.awt import Label
>>> class labelWrapper(Label):
...     pass
...
>>> class labelCheckbox(Checkbox, labelWrapper):
...     pass
...
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError: no multiple inheritance for Java classes:
org.python.proxies._main__labelWrapper$0 and java.awt.Checkbox
```

从 Java 类和 Jython 类中多重继承也能运行，但如果 Jython 类不是从 Java 继承的，就像有 Jython 的 Thread 类的情况一样应用：

```
>>> from threading import Thread
>>> from java.awt import Label
>>> class threadedLabel(Label, Thread):
...     pass
...
>>> tl = threadedLabel()
```

程序清单 6-1 是一个模块，该模块包含了从 Java 类继承的类的一个例子。对此表进行完整的剖析，首先以引入开始，然后是类定义和 FileFilter 类里的两个实例方法。这个例子是一个 Jython 模块，该模块应放在 sys.path 某个地方以便通过引入 filefilter 模块就能继续进行讨论。filefilter 模块引入下列条目：

- from java import to——Filter 类从 Java 类 java.io.FilenameFilter 继承，这个继承需要基类首先被引入。

- from os import path——这使 path.join 函数能使用。这个函数加入了 path 元素并在合适的地方增加了特定平台的分隔符。Filter 类将有序的表达式应用到整个路径，因此必须要加入 path 和 filename 元素。
- import re——这引入了 Jython 的有序表达式模板。FileFilter 类仅批准那些与特定的有序的表达式相匹配的文件。记住有序表达式是不同于通配符的。星号 (\*) 在有序表达式中是一个重复运算符，句点(.) 匹配任何字符。

程序清单 6-1 里的 FileFilter 类从 Java 类 java.io.FilenameFilter 继承，因此 java.io.File 类能用它来检索过滤目录列表。当应用到 accept 实例方法时，列表按照返回 1 的文件被过滤。

在程序清单 6-1 里两个实例方法中的 addFilter 允许把有序表达式装载到实例变量，但 accept 是一个需要满足作为 FilenameFilter 作用的方法。实例变量 self.ffilter 是一个保存有经过编译的有序表达式的 PyList。当第一个过滤器被增加时，这个实例变量被创建，那就是 if 'ffilter' not in vars(self).keys() 条件的理由。一个相似的条件作为 assert 语句的一部分出现在 accept 方法中，该 assert 语句确保在使用过滤器前至少增加一个有序表达式。

#### 程序清单 6-1 使用 Java 基类的文件过滤类

---

```
# file: filefilter.py

from java import io
from os import path
import re

class FileFilter(io.FilenameFilter):
    def addFilter(self, ffilter):
        if 'ffilter' not in vars(self).keys():
            self.ffilter = []
        self.ffilter.append(re.compile(ffilter))

    def accept(self, dir, name):
        assert 'ffilter' in vars(self).keys(), 'No filters added'
        for p in self.ffilter:
            if not p.search(path.join(str(dir), name)):
                return 0
        return 1
```

---

在交互解释程序里使用这个新的 Jython 模块确保它是在 sys.path 里，然后键入：

```
>>> import filefilter
>>> from java import io
>>> ff = filefilter.FileFilter()
>>> ff.addFilter(".*\.\tar\.\gz")
>>> ff.addFilter("eiei")
>>> # Change the following line to use a path specific to your platform
>>> L = io.File("c:\\windows\\desktop").list(ff)
>>>
>>> # print the list of files found
>>> for item in L:
...     print item
...
eiei-0_16.tar.gz
```

与 Java 类一起运行需要特别注意。首先，Java 类型需要转换成适合于方法参数的 Jython 类和返回值，反之亦然。关于 Jython 和 Java 类型是怎么转换的信息在第 2 章“运算符、类型和内置函数”是可用的。调用 Java 超类的成员需要仔细解释。第一是用点标注语法表示的类名加方法，并用提供的 self 作为第一个自变量，如下：

```
>>> from java import util
>>> class test(util.Vector):
...     def addToSuper(self, objectToAdd):
...         "Calling a java superclass looks like this:"
...         util.Vector.addElement(self, objectToAdd)
...
>>> t = test()
>>> t.addToSuper("a string")
>>> t.toString()
'[a string]'
```

第一个语法的问题是 Java 超类不理解源于派生类的调用。这个语法不对公有方法起作用，当显式初始化超类时是需要的，但它对保护的实例方法不起作用——对此使用第二个语法。

第二个语法是 self.method。加 self 前缀语法保证 Java 超类理解源于派生类的调用，当访问保护的实例方法时是需要的。程序清单 6-2 示范了用两个语法访问超类的成员。

#### 程序清单 6-2 调用 Java 超类里的方法

```
# file: javabase.py
import sys
from java import util

class cal(util.GregorianCalendar):
    def __init__(self):
        # This uses the "classname.method(self)" syntax
        # to initialize the superclass
        util.GregorianCalendar.__init__(self)

    def max(self):
        # This uses the "classname.method(self)" syntax
        return util.GregorianCalendar.getActualMaximum(self, 1)

    def min(self):
        # Another example of "classname.method(self)" syntax
        return util.GregorianCalendar.getActualMinimum(self, 1)

    def compute1(self):
        # This tries the "classname.method(self)" syntax.
        # This will not work because the method called is protected.
        try:
            util.GregorianCalendar.computeTime(self)
            return "Success"
        except AttributeError, e:
            print "compute1 failed.\n", e

    def compute2(self):
        # This uses the "self.method()" syntax so that the call appears
        # to originate from the base class.
```

```

try:
    self.computeTime()
    return "success"
except AttributeError, e:
    print "compute2 failed.", e

```

假定以上的模块是在 `sys.path` 中，你可以这样测试：

```

>>> import javabase
>>> c = javabase.cal()
>>> print "Trying max: ", c.max()
Trying max: 292278994
>>> print "Trying min: ", c.min()
Trying min: 1
>>> print "Trying compute1: ", c.compute1()
Trying compute1: compute1 failed.
class 'java.util.GregorianCalendar' has no attribute 'computeTime'
None
>>> print "Trying method compute2: ", c.compute2()
Trying method compute2: success

```

Java 的 `protected` 和 `static` 修饰语对 Jython 派生类具有特殊的关系。解释这种关系需要快速浏览 Jython。Jython 是怎么访问 Java 超类类成员的呢？一些访问是通过代理对象，该代理对象是 Java 对象的实派生类，但一些访问不是通过代理。使用 `java.util.Vector` 的派生类里的 `self.add` 方法（没有重定义 `add`）实际上要求代理对象来调用超类方法。实派生类代理能够访问保护的（`protected`）实例方法。然而目前的实现没有为类域或静态（`static`）成员创建代理入口。这意味着什么？这意味着当 Jython 派生类企图访问静态方法或类域时它不会从实 Java 派生类出现。这使得保护域（`protected field`）和保护的静态成员（`protected static members`）等同于私有的（`private`），直到 Jython 派生类被涉及到。

这是 Jython 派生类可访问的快速列表：

- 公有类（静态）方法和域。
- 公有实例方法和域。
- 保护实例方法（仅用 `self.method` 标注的）。

这里是标准 Java 派生类可访问而 Jython 派生类不能访问的列表：

- 保护类（静态）方法和域。
- 保护实例域。
- 保护包成员。

这种限制可用两种途径中的一种来正常处理。第一，你可写 Java 类作为访问需要保护的域和你需要保护的静态方法的中间件。本章末的树形部分给出了被写作不可访问成员中间件的 Java 类。第二，有用来设置 `python.security.respectJavaAccessibility` 的 Jython 注册。Jython 注册文件是一个在 Jython 的安装目录里调用 `registry` 的文件或在用户目录调用 `.jython` 的文件。在这个文件里 `respectJavaAccessibility` 设置可能是真（`true`）或假（`false`），Jython 获得了访问 Java 类（不仅是超类）的保护和私有成员。可能像你所希望的一样，当防止访问性约束这样发生时对于安全

性和稳定性有另外的含意。同样，要求你工作的并发用户也设置 `respectJavaAccessibility` 为假可能比较麻烦。当决定改变 `respectJavaAccessibility` 特性时，这个问题要仔细考虑。

使用 Java 超类最后要注意的是超类构造函数。显式调用 Java 超类构造函数使用 `classname._init_(self)` 语法。如果没有显式调用，在派生类的 `_init_` 方法完成后调用空超类构造函数。

## 6.7 方法重载

Java 广泛地使用方法重载。方法重载允许或要求若干方法对不同的参数列表使用相同的名字。参数列表不同的是参数的数量和参数的类型。例如，Jython `xrange` 函数的 Java 实现需要三个方法来实现：

```
public static PyObject xrange(int n) {
    return xrange(0,n,1);
}

public static PyObject xrange(int start, int stop) {
    return xrange(start,stop,1);
}

public static PyObject xrange(int start, int stop, int step) {
    return new PyXRange(start, stop, step);
}
```

这在 Java 是普遍深入的，而 Jython 不提供方法重载。当然缺乏方法重载并不减少 Jython 的表示，但当它与 Java 类一起运行时就成了一个问题。当 Jython 代码调用一个重载 Java 方法会发生什么呢？它通常像你所希望的那样运行。尽管 Jython 缺乏方法重载，但它对使用 Java 类里的重载方法大有帮助。这是类 `java.lang.StringBuffer` 使用的例子：

```
>>> import java
>>> sb = java.lang.StringBuffer()
>>> sb = java.lang.StringBuffer(10)
>>> sb = java.lang.StringBuffer("Jython ")
>>> sb.append(2.1)
Jython 2.1
>>> sb.append("b")
Jython 2.1b
>>> sb.append("1")
Jython 2.1b1
```

`StringBuffer` 类有三个不同的构造函数，附加方法也被重载，但是 Jython 使用所有方法都没出现问题。Jython 在内部确定要调用哪个 Java 方法首先按照参数的数量，然后按照类型。然而执行重载了的 Java 方法并不意味着在 Jython 类里重载是可能的。因为用户能从 Jython 调用重载了的 Java 方法，所以他们设想在 Jython 派生类仅重载某一 Java 方法能够运行。这不是真实的。当 Jython 派生类替换重定义的 Java 方法，它必须处理那个方法的所有重载的场合。在这种情况下 Jython 根本没有按照类型特征符使用方法重载。即使出现，参数的数量应显式地选择一个特殊方法，Jython 保持不可选择。当替换一个重载了的方法时，要么替换所有，要么什么也不做。

程序清单 6-3 检查了派生 `java.awt.Dimension` 类。程序清单 6-3 扩展 `Dimension` 类生成一个有

边界的相似的类。程序清单 6-3 里那些有趣的类是构造函数和 `setSize`。构造函数设立具有可选参数值的最大 `sizes` 或缺省值 100。构造函数必须初始化实际上有三个构造函数的超类。派生类根据构造函数获得的自变量决定每一个 `java.awt.Dimension` 构造函数。另外，`BoundaryBox` 的 `setSize` 方法替换了同名的超类的方法。超类 `Dimension` 重载有两个类型特征符的 `setSize` 方法：

```
void setSize(Dimension d)
void setSize(double width, double height)
```

记住重定义一个重载的方法影响相同名字的所有超类方法。而一个类的特定用法可能意味着这些方法中的某些对环境是无意义的，常常是派生类里的新方法必须处理多个或所有同名的超类方法特征符。既然这样，它应该考虑 `width` 和 `height` 的值或 `setSize` 方法里的 `Dimension` 实例。程序清单 6-3 要考虑这些自变量的任何一个，但由于限度测试，程序清单 6-3 仅需要使用超类的 `setSize` 方法中的一个。

### 程序清单 6-3 重定义重载的 Java 方法

```
# file: boundarybox.py
from java import awt
import types
import warnings as wrn # to shorten some lines

class BoundaryBox(awt.Dimension):
    """BoundaryBox is a java.awt.Dimension subclass with max limits.
    The constructor accepts optional width and height values
    or a java.awt.Dimension instance, which provide max values"""

    def __init__(self, *args):
        if len(args)==2: # assume this is width, height
            self.maxWidth = args[0]
            self.maxHeight = args[1]
            awt.Dimension.__init__(self, args[0], args[1])
        elif len(args)==1: # assume this is a Dimension instance
            self.maxWidth = args[0].getWidth()
            self.maxHeight = args[0].getHeight()
            awt.Dimension.__init__(self, args[0])
        elif len(args)==0:
            self.maxWidth = 100
            self.maxHeight = 100
            awt.Dimension.__init__(self)

    def setSize(self, *dim):
        if len(dim) == 2: # args must be width, height
            w = self._testWidth(dim[0])
            h = self._testHeight(dim[1])
        elif len(dim) == 1 and isinstance(dim, awt.Dimension):
            # in case arg is a Dimension instance
            w = self._testWidth(dim.getWidth())
            h = self._testHeight(dim.getHeight())
        else:
            assert 0, "'setSize accepts w, h or a Dimension inst'"
            awt.Dimension.setSize(self, w, h)

    def _testWidth(self, w):
```

```

if w > self.maxWidth:
    msg = "Width, %s, exceeds bounds. Changed to %s"
    print msg % (w, self.maxWidth)
    return self.maxWidth
return w # width within bounds

def _testHeight(self, h):
    if h > self.maxHeight:
        msg = "Height, %s, exceeds bounds. Changed to %s"
        print msg % (h, self.maxHeight)
        return self.maxHeight
    return h # height within bounds

```

为了测试 BoundaryBox 类，把 boundarybox.py 文件放在 sys.path 里，并使用以下语句：

```

>>> import boundarybox
>>> bb = boundarybox.BoundaryBox(100, 200)
>>> bb.setSize(50, 201)
Height, 201, exceeds bounds. Changed to 200
>>> bb.setSize(101, 100)
Width, 101, exceeds bounds. Changed to 100
>>> bb.width
100
>>> bb.height
100

```

## 6.8 例子类

为了更好地理解 Jython 类，最好检验示范。本节不仅提供了 Jython 的示范，而且提供了它们怎样与相似的 Java 类发生联系的示范。

### 6.8.1 单元素

这个例子对 Java 实现与 Jython 里相似的功能进行了比较。在比较中明显的区别有助于阐述 Jython 类的使用。

当一个应用请求带有特定对象的单点接口时，Java 程序员常常创建一个类，该类限制创建它自己的多个实例。当有不同的实现时，最经常是把类域和类方法结合起来返回实例。程序清单 6-4 给出了在类域里包含自己的一个实例的 Java 类，这个类也有一个类方法 getInstance，该方法决定实例在返回之前是否已存在。既然这样，如果实例已存在则该类方法返回预先存在的实例，但是它通常会产生一个异常或返回空。

#### 程序清单 6-4 Java 单元素

---

```

# file: Singleton.java

public class Singleton {
    private static Singleton single=null;

    private Singleton() {}

    public static Singleton getInstance() {

```

```

    if (single==null) single = new Singleton();
    return single;
}

//The methods which required a single point of access
//go here.
}

```

由于在 Jython 里缺乏类方法和没有能力使构造函数成为私有的，所以进行了一个有趣的比较。在 Jython 里这怎么实现？如果没有 private 修饰语来约束实构造函数，使用一个非构造函数来返回一个实例几乎没有价值。构造函数不能返回值，因此它不能选择一个预先存在的实例来返回。然而 static 类数据属性存在，这有助于为一个已存在的序列进行测试。这仅考虑测试，因此当创建第二个实例时在产生异常的 assertion 或其他表达式里，这是非常有用的。如下所示：

```

# file: SimpleSingleton.py
class Singleton:
    single = None
    def __init__(self):
        assert Singleton.single==None, "Only one instance allowed"
        Singleton.single = self

    def __del__(self):
        Singleton.single = None

```

在第二个实例创建时，这种用法将产生一个 AssertionError：

```

>>> from SimpleSingleton import Singleton
>>> s1 = Singleton()
>>> s2 = Singleton()
Traceback (innermost last):
  File "SimpleSingleton.py", line 11, in ?
    File "SimpleSingleton.py", line 4, in __init__
AssertionError: Only one instance allowed

```

虽然在许多情况下产生一个异常是足够的了，但是你可能想要预先存在的实例。为了更加接近地仿真在程序清单 6-4 里的 Java 类，你需要类方法的替代物。Java 以类为中心的编程可能劝止一些人模块级封装和函数的使用，但是你应小心不要轻视它们。把模块里的类和 factory 函数结合常常是这里的一个解决方案。程序清单 6-5 给出了这种结合。

#### 程序清单 6-5 函数 + 类 = 单元素

---

```

# file: singleton.py

class _Singleton:
    single = None
    def __init__(self):
        pass
    # put required methods here

    def __del__(self):
        _Singleton.single = None

```

```

def Singleton():
    if _Singleton.single == None:
        _Singleton.single = _Singleton()
    return _Singleton.single

#add testing code to the module
if __name__=='__main__':
    s1 = Singleton()
    s2 = Singleton()

    # set an instance data attribute in s1
    s1.data = 4
    # s2 should point to the same data value if it is really
    # the same instance
    print s2.data

```

运行 `python singleton.py` 的结果仅仅是整数 4，它证实了上述例子的 `s1` 和 `s2` 两个标志符共享相同实例。

### 6.8.2 文件 grep 效用

程序清单 6-6 是一个为特定模式搜寻行或文件的简单类。构造函数仅仅高速缓存特定目录里的文件列表。`findFiles` 方法把模式与这个高速缓存的列表进行比较，并返回一个过滤了的列表。而 `findLines` 一行一行地读每一个文件来返回包含特定模式的行的列表。这两个方法按照带有 `findFiles` 的第一个限制的文件列表很明显地用在一起，并用 `findLines` 搜寻这些文件。

#### 程序清单 6-6 用 Python 类搜寻文件

```

# file: grep.py
import os
from os.path import isfile

class directoryGrep:
    def __init__(self, directory):
        self.files = filter(isfile, [os.path.join(directory, x)
                                     for x in os.listdir(directory)])

    def findLines(self, pattern, filelist=None):
        """Accepts pattern, returns lines that contain pattern
        Optional second argument a filelist to search"""

        if not filelist:
            filelist = self.files

        results = []
        for file in filelist:
            fo = open(file)
            results += [x for x in fo.readlines()
                        if x.find(pattern) != -1]
            fo.close() # explicit close of file object
        return results

    def findFiles(self, pattern):

```

```

'Accepts pattern, returns filenames that contain pattern'
return [x for x in self.files if x.find(pattern) != -1]

# test
if __name__ == '__main__':
    g = directoryGrep("c:\\windows\\desktop")
    files = g.findFiles(".py")
    print g.findLines("java", files)

```

运行 `python grep.py` 的结果如下：

```

['from java.lang import System\n', 'from java.lang import Runtime\n', 'from
java.io import IOException\n', 'import java\n', 'from java.sql import
DriverManager\n']

```

当读文件时，程序清单 6-6 显式关闭它打开的文件对象。Python 程序员经常用到下列短句：

```
open(file).readlines()
```

这个短句从一个匿名文件对象读取并创建。它采取这样的工作方式：当垃圾收集器知道一个需要回收资源的要求时，它关闭文件和释放资源。注意，Jython 依赖 Java 垃圾回收器，因此，文件的实际关闭和资源的释放会延迟一个未知时段。在程序清单 6-6 中，对象的显式关闭避免了因潜伏时间而造成的潜在问题。

### 6.8.3 HTTP 报文头

HTTP 报文头是由一个伴随着一系列被称为首部的关键字-值对的实请求和一个用来终止报头的空白行（\r\n\r\n）组成。程序清单 6-7 通过解释字符串到数据域和再返回到字符从而简化了请求数据的操作。

#### 程序清单 6-7 解释 HTTP 请求报文头

```

# file: request.py

import string
import re

class Request:
    """Class for working with request strings:
    The constructor optionally takes an http request string.
    e.g., req = Request("GET http://localhost/index.html HTTP/1.0")"""

    def __init__(self, request=None):
        "Parses a raw request string, if supplied.
        self.headers = {}
        if not request:
            request = "GET None HTTP/1.0"
        self.method, self.url, self.version = request.split()
        self.host = self.path = self.file = ""
        match = re.match("http://(\S*)(/.*)", self.url)

```

```

if (match != None):
    self.host, self.path = match.groups()

def setHeader(self, stringHdr):
    try:
        x,y = [x.strip() for x in stringHdr.split(':')]
        self.headers[x] = y
    except ValueError:
        raise SyntaxError("A header string must be "
                           "in the format key : value")

def __repr__(self):
    d = self.__dict__ # make a local copy for convenience
    request = " ".join([d['method'], d['url'], d['version']])
    request += "\r\n"
    for key in self.headers.keys():
        request += (key + ": " + self.headers[key] + "\r\n")
    request += "\r\n"
    return request

# module testing code
if (__name__ == '__main__'):
    # Creating the instance can be without arguments...
    a = Request()
    # or with a raw request string like...
    a = Request("GET HTTP://freshmeat.net/index.html HTTP/1.0")

    # Headers are added/altered with "setHeader".
    # setHeader works with a header string as an argument...
    a.setHeader("Accept: image/jpeg, image/pjpeg, image/png, */*")
    a.setHeader("Proxy-Connection: Keep-Alive")
    a.setHeader("Accept-Encoding : gzip")

    #put it all together
    print a

```

依据起码的包含和表示法，类通常是很帮助的。这意味着类在需要时，就能通过有用的方式将相关的数据分组并提交数据。在程序清单 6-7 中的包含由两个字典组成，一个包含请求数据，另一个则包含了首部。有趣的是包含了请求数据（如 method, URL, and HTTP version）的字典就是实例字典或 `self.__dict__`。程序清单 6-7 在这个实例映射里设置入口就像它是带有 `self.__dict__.setDefault` 的标准字典，该字典使用实例属性标注 `self.attribute = x` 语法。两个方法都可更新 `self.__dict__`。

表示法是一个特殊的方法，在 Jython 类里是 `_repr_`，而在 Java 里通常是 `toString` 方法。如果这个方法存在，它必须返回对象的字符串表示法，并且当需要对象（如 `print` 语句）的字符串表示法时，这就是被调用来提供它的方法。构造函数、终止器和这种表示法方法都是到目前为止讨论的特殊类方法，但是已存在许多为定制对象行为的特殊方法。

运行 Jython `request.py` 的输出是：

```

GET HTTP://freshmeat.net/index.html HTTP/1.0
Accept: image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Proxy-Connection: Keep-Alive

```

### 6.8.4 树

树是普通的数据结构，它对演示类来说是极好的例子。树是节点（nodes）和叶子（leaves）的集合，在树里，一个节点又包含其他的节点和叶子，而叶子终止其分枝（不包含其他的节点）。因为家族树的自顶向下表示法和因为节点和叶子常常引用父——子关系，该类比更加接近相关家族树而不是树状物。程序清单 6-8 定义了两个类：leaf 类和 node 类。使用树的重要的一点是节点和叶子可能有相同的方法，因此对使用节点或叶子的接口保持相同。这与它们的继承递归结构相结合产生了清楚的、一致的和广泛适用的模式。像分类学、公司职员和文件系统这样的层次数据是一些树状数据的例子。把程序清单 6-8 里通常的 leaf 类和 node 类应用到大量相似的树状结构是可能的。对于创建适当类的树你要知道三件事：

程序清单 6-8 用 Python 类生成树

```
# file: trees.py

class Node:
    def __init__(self, name, value=None):
        self._children = []
        self.name = name
        self.value = value
        self._leaf = 0 # Means this node can have children

# These methods answer question the object must know about itself.
def isLeaf(self):
    """returns answer to, "Am I a leaf?" """
    return self._leaf

def getParent(self):
    """returns answer to, "Who is my parent?" """
    return self._parent

def getChildren(self):
    """returns answer to, "Who are my children?" """
    return tuple(self._children)

# These methods supply required functionality of the object
def addChild(self, node):
    node.setParent(self)
    self._children.append(node)

def removeChild(self, node):
    try:
        self._children.remove(node)
        return 1
    except ValueError: # in case that child doesn't exist
        return 0

# implementation detail
def setParent(self, parent):
    self._parent = parent

# A quick way to view the tree
```

```

def dump(self, level=0):
    print "%s%s %s" % (' '*level, self.name, self.value or '')
    for child in self.getChildren():
        child.dump(level+1)

def __repr__(self):
    return self.name

class leaf(node):
    def __init__(self, name, value=None):
        node.__init__(self, name, value)
        self._leaf = 1

    def addChild(self, node):
        raise ValueError, "Cannot add children to a leafNode"

    def removeChild(self, node):
        raise ValueError, "A leaf node has no children to delete."

    def getChildren(self):
        return () # returns empty tuple

# Code to test these two classes
if __name__ == '__main__':
    root = node("root", "trees")
    evergreens = node("evergreens")
    evergreens.addChild(node("Picea", "Spruce"))
    deciduous = node("deciduous")
    deciduous.addChild(node("Acer", "Maple"))
    quercus = node("Quercus", "Oak")
    quercus.addChild(leaf("Alba", "White Oak"))
    quercus.addChild(leaf("Palustris", "Pin Oak"))
    quercus.addChild(leaf("Rubra", "Red Oak"))
    deciduous.addChild(quercus)
    root.addChild(evergreens)
    root.addChild(deciduous)
    children = root.getChildren()
    print children
    print quercus.getParent()
    print # add blank line
    print "Here is a dump of the entire tree structure:"
    root.dump()

```

- 1) 树由节点组成。
- 2) 一些树可能有子节点。
- 3) 叶子也是节点，但它是特殊的，因为它们不能有子节点。

对类来说这意味着什么？它意味着对类知道它本身是什么和它能提供什么功能有特殊的期望。一个节点应该能回答关于它本身的问题是：我是一个叶子吗？谁是我的父节点？和谁是我的子节点？对非叶子节点所期望的行为是增加和删除子节点的能力。leaf 类尽管没有子节点，也包含了增加和删除子节点的方法，也回答谁是我的子节点。这保证了一致的接口。除了在

`getChildren`方法返回一个元组以便它更好地匹配节点行为的情况下，这些方法产生异常。另外一个实现细节的需要是每一个对象有一个`setParent`方法，当增加一个对象到节点时需要该方法。

因为叶子节点仅是一个减去了子节点的节点，`leaf`类派生节点并仅重定义那些与子节点有关的方法。

运行`jython trees.py`的结果应是：

```
(evergreens, deciduous)
deciduous

Here is a dump of the entire tree structure:
root trees
  evergreens
    Picea Spruce
  deciduous
    Acer Maple
    Quercus Oak
      Alba White Oak
      Palustris Pin Oak
      Rubra Red Oak
```

一个相似的树的Java实现毫无疑问是不同的，但它是怎么不同看起来是有趣的。Java树将可能有一个`abstract`类来为一个合法节点保证一致的最小。另外，一些实例变量将使用`protected`修饰符。在Java里一个可能的`abstract`类看起来如程序清单6-9。

#### 程序清单6-9 一个`abstract`Java节点类

```
// file: Node.java
import java.util.Vector;

public abstract class Node {
    protected String name;
    protected Node parent = null;
    protected boolean leaf = false;

    public abstract Vector getChildren();
    public abstract boolean addChild(Node n);
    public abstract boolean removeChild(Node n);

    public boolean isLeaf() {
        return leaf;
    }
    public String toString() {
        return name;
    }
}
```

介绍了Jython树类的一些问题，即`protected`域`leaf`、`parent`和`name`。Jython派生类不能访问`protected`Java域除非Jython的`respectJavaAccessibility`注册设置成`false`。让我们假定具有注册设置的回避访问无论什么原因都是不可接受的。可选择的办法是写一个Java类来作为Jython的

abstract Node类的中间件。程序清单 6-10 给出了这样的 Java adapter类。

#### 程序清单 6-10 Java-to-Jython adapter 类

```
// file: JyNodeAdapter.java
import Node;
import java.util.*;

public abstract class JyNodeAdapter extends Node {
    protected void setLeaf(boolean leaf) {
        this.leaf = leaf;
    }
    protected void setName(String nodeName) {
        name = nodeName;
    }
    public String getName() {
        return name;
    }
    public Node getParent() {
        return parent;
    }
    public void setParent(Node p) {
        parent = p;
    }
    public void dump() {
        this.dump(0);
    }
    public void dump(int level) {
        for (int i = 0; i < level; i++) {
            System.out.print("  ");
        }
        System.out.println(name + " " + this.value);
        Vector v = getChildren();
        for (Enumeration e = v.elements(); e.hasMoreElements();) {
            JyNodeAdapter node = (JyNodeAdapter)e.nextElement();
            node.dump(level + 1);
        }
    }
}
```

需要生成 Jython 树类的惟一修改是增加 JyNodeAdapter 作为一个超类并改变属性 fetching 和 setting 来使用在 Java 超类里指定的合适的 set 和 get 方法。另外一个变化是改变 getChildren 的返回值，getChildren 总是一个符合 abstract Node 类里的特定协议的 Vector（对 leaf 来说是一个空 Vector）。程序清单 6-11 给出了适当修改的 Jython 树实现的修改本来使用 Node 和 JyNodeAdapter 基类。

#### 程序清单 6-11 JyNodeAdapter 的 Jython 派生类

```
# file: SubclassedTree.py

import JyNodeAdapter
from java.util import Vector
```

```

class JyNode(JyNodeAdapter):
    def __init__(self, name, value=None):
        self.setName(name)
        self.value = value
        self._children = Vector()

    def getChildren(self):
        return self._children

    def addChild(self, c):
        c.setParent(self)
        self._children.add(c)
        return 1

    def removeChild(self, c):
        self._children.removeElement(c)
        return 1

    def __repr__(self):
        return self.toString()

class leaf(JyNodeAdapter):
    def __init__(self, name, value=None):
        self.setName(name)
        self.value = value
        self.setLeaf(1)

    def addChild(self, node):
        raise ValueError, "Cannot add children to a leafNode"

    def removeChild(self, node):
        raise ValueError, "A leaf node has no children to delete."

    def getChildren(self):
        return Vector() # returns empty Vector

    def __repr__(self):
        return self.toString()

if __name__=='__main__':
    root = JyNode("root", "trees")
    evergreens = JyNode("evergreens")
    evergreens.addChild(JyNode("Picea", "Spruce"))
    deciduous = JyNode("deciduous")
    deciduous.addChild(JyNode("Acer", "Maple"))
    quercus = JyNode("Quercus", "Oak")
    quercus.addChild(leaf("Alba", "White Oak"))
    quercus.addChild(leaf("Palustris", "Pin Oak"))
    quercus.addChild(leaf("Rubra", "Red Oak"))
    deciduous.addChild(quercus)
    root.addChild(evergreens)
    root.addChild(deciduous)
    children = root.getChildren()
    print children
    print quercus.getParent()
    print # add blank line
    print "Here is a dump of the entire tree structure:"
    root.dump()

```

运行 `python SubclassedTree.py` 的结果如下所示：

```
(evergreens, deciduous)
deciduous

Here is a dump of the entire tree structure:
root trees
  evergreens
    Picea Spruce
  deciduous
    Acer Maple
    Quercus Oak
      Alba White Oak
      Palustris Pin Oak
      Rubra Red Oak
```



## 第 7 章 高 级 类

用两个头部和尾部下划线标志的特殊属性在 Jython 类里是非常多的。它们是用复杂的行为创建高度定制的对象的主要途径。本章认为高级 Jython 类就是那些影响这些特殊属性的类。描述这些类是高级的可能被误解为意思是困难的或为了在 Jython 里作更多研究而保留的，但并不是这样。把这些特殊属性加到类是应付复杂性的一部分。把对象的行为调整为像列表一样作用的能力或截听属性访问的能力仅运用一些特殊方法，而在设计、重复使用能力和灵活性方面的潜在收获是巨大的。

### 7.1 预先存在的类属性

类和实例隐含有一些特殊属性——当一个类定义执行或实例被创建时自动出现的那些属性。Jython 包含 Java 类和实例以便它们也有特殊属性。Jython 类有五个特殊属性，而 Java 类有五个同样类中的三个。注意当所有这些属性是可读时，仅一些属性允许赋值给它们。

为了进一步检查这些特殊属性，让我们首先定义一个适合开发的最小 Jython 类。程序清单 7-1 是一个包含 import 语句和单类定义 LineDatum 的 Jython 模块。LineDatum 类仅仅根据提供给构造函数的 slope 和 intercept 定义一行表达式（数据）。于是实例可以用 addPoint 方法增加位于行上的 points。

程序清单 7-1 跟踪行上 Points 的 Jython 类

---

```
# file: datum.py
import java

class LineDatum(java.util.Hashtable):
    """Collects points that lie on a line.
    Instantiate with line slope and intercept: e.g. LineDatum(.5, 3)"""

    def __init__(self, slope, incpt):
        self.slope = slope
        self.incpt = incpt

    def addPoint(self, x, y):
        """addPoint(x, y) -> 1 or 0
        Accepts coordinates for a cartesian point (x,y). If point is
        on the line, it adds the point to the instance."""

        if y == self.slope * x + self.incpt:
            self.put((self.slope, self.incpt), (x, y))
            return 1
        return 0
```

---

从交互解释器里运行 LineDatum 类如下所示：

```
>>> import datum
>>> ld = datum.LineDatum(.5, 3)
>>> ld.addPoint(0, 3)
1
>>> ld.addPoint(2, 3)
0
>>> ld.addPoint(2, 4)
1
```

特殊的类变量在下一部分描述。

#### 1. `_name_`

这个只读属性包含了类名。在程序清单 7-1 里 LineDatum 类的名字是 LineDatum。即使使用 `import`, `_name_` 也是 LineDatum。一个例子如下：

```
>>> from datum import LineDatum
>>> LineDatum.__name__
'LineDatum'
>>> from datum import LineDatum as ld
>>> ld.__name__
'LineDatum'
```

用在 Jython 里的 Java 类也有`_name_` 属性，示范如下：

```
>>> import java
>>> java.util.Hashtable.__name__
'java.util.Hashtable'
```

#### 2. `_doc_`

这个属性包含类文档字符串，如果文档字符串不提供则为 `None`。你可以赋值给`_doc_`。

```
>>> from datum import LineDatum
>>> print LineDatum.__doc__
Collects points that lie on a line.
Instantiate with line slope and intercept: LineDatum(.5, 3)
```

用在 Jython 里的 Java 类没有`_doc_` 属性。

#### 3. `_module_`

这个属性包含了类被定义的模块名。你可赋值给`_module_`。程序清单 7-1 里的 LineDatum 类，它是在模块 `datum` 里定义的，并可用这个例子确认：

```
>>> from datum import LineDatum
>>> LineDatum.__module__
'datum'
```

用在 Jython 的 Java 类没有`_module_` 属性。Java 没有模块，因此这有意义。

#### 4. `_dict_`

这个属性是一个包含所有类属性的 PyStringMap 对象。我们看一看程序清单 7-1 里的 LineDatum 类定义，能够猜想到在 `LineDatum._dict_` 里能找到什么样的键：必定有`_init_` 和 `addPoint`，因为这是两个被定义的方法。也应该有以前描述的`_doc_` 和 `_module_` 键。`_name_` 键是一个好的推

测，但它实际上没有出现在 Python 里的 `class.__dict__`。程序清单 7-1 里的 `LineDatum` 类比一般的类更复杂，因为它实际上派生了一个 Java 类。需要另外属性的实现，如这个例子所见：

```
>>> from datum import LineDatum
>>> LineDatum.__dict__
{ '__doc__': 'Collects points that lie on a line.\n      Instantiate with\n      line slope and intercept: LineDatum(.5, 3)\n      ', 'rehash': <java function\n      rehash at 1298249>, '__init__': <function __init__ at 913493>, 'addPoint':\n      <function addPoint at 1939121>, 'finalize': <java function finalize at\n      1068455>, '__module__': 'datum'}
```

定义在类里的属性出现在类的 `__dict__`。在类 A 访问属性 b 的传统的表示法是 `A.b`；然而，类 `__dict__` 允许 `A.__dict__['b']` 的替换表示法。这是一个对属性访问的不同语法的例子：

```
>>> from datum import LineDatum
>>> LineDatum.__module__
'datum'
>>> LineDatum.__dict__['__module__'] # same as "LineDatum.__module__"
'datum'
```

你甚至可用替换命名来调用方法。程序清单 7-1 的 `addPoint` 方法需要一些技巧，因为这是一个实例方法。实例必须是第一个参数。幸运的是 Python 并不过分注意它是哪个实例，因此你可在测试语法时创建一个实例，并传递它作为第一个自变量：

```
>>> from datum import LineDatum
>>> inst = LineDatum(.5, 3) # get a surrogate instance
>>> LineDatum.addPoint(inst, 5, 4)
↑
>>> LineDatum.__dict__['addPoint'](inst, 5, 4) # does same as above
1
```

这种访问类的间接方式和实例属性是一些受欢迎的 Python 模式的一部分。直接使用类 `__dict__` 是许多灵活的 Python 对象设计使用的，当某些特殊方法被定义时有时候被需要，例如后面描述的 `_setattr_` 方法。甚至更有趣的是使用在 Python 里的 Java 类也有一个包含了它们的成员的 `__dict__`。看 Python 里的 `java.io.File` 类的 `__dict__` 需要下列各项：

```
>>> import java
>>> java.io.File.__dict__
{'createNewFile': <java function createNewFile at 4294600>, 'lastModified':\n<java function lastModified at 3759986>, ... }
```

由于它的长度，`java.io.File.__dict__` 的完整结果留给读者去完成得到。如果你想看成员名，使用下列各项：

```
>>> java.io.File.__dict__.keys()
['mkdirs', 'exists', ...]
```

另外，你可通过使用 `java.io.File.__dict__` 里的键来调用方法，例如 `listRoots`。用 `java.io.File.listRoots()` 传统上能调用与 `java.io.File.__dict__['listRoots']()` 一起作用，它们示范在这个例子中：

```
>>> java.io.File._dict_['listRoots']()
array([A:\, C:\, D:\, G:\], java.io.File)
```

目前，你也可赋值给 Java 类的 `dict`。虽然这对初学者来说可能是一个不好的实践，但它阐明了 `_dict_` 的特性。如果你想改变 Java 类成员的查找，你可以改变它的 `_dict_` 里的那个键值。假如你想有一个不同的为 `java.io.File.listRoots()` 调用的方法，你可用这种方式改变：

```
>>> import java
>>> def newListRoots():
...     return ['c:\\\\']
...
>>> java.io.File._dict_['listRoots'] = newListRoots
>>> java.io.File.listRoots() # try the circumvented method
['c:\\\\']
```

### 5. `_base_`

这是基类或超类元组。在 Jython 2.0 版和 Jython 2.1 的第一个初始版里指定这个变量为只读；然而，Jython 2.1a1 以后的版本允许赋值给这个变量。这个时候写的实现稍微有点不同于 Cpython，因为你可以改变 Cpython 的 `_bases_`。在程序清单 7-1 里，`LineDatum` 的超类是 `java.util.Hashtable`。特殊变量 `_bases_` 确认这点：

```
>>> from datum import LineDatum
>>> LineDatum._bases_
(<jclass java.util.Hashtable at 5012120>,)
```

使用在 Jython 里的 Java 类也有特殊的 `_bases_` 变量，它包含基类和实现的接口：

```
>>> import java
>>> java.io.File._bases_
(<jclass java.lang.Object at 7290061>, <jclass java.io.Serializable at
62789>, <jclass java.lang.Comparable at 6728374>)
```

## 7.2 预先存在的实例属性

Jython 实例有两个隐式定义的特殊变量，而在 Jython 里使用的 Java 实例有一个。这些属性是可读和可赋值的。

### 1. `_class_`

`_class_` 变量表示当前对象是它的实例的类。如果我们继续窃用程序清单 7-1，我们能示范 `LineDatum` 是怎样知道它的类：

```
>>> from datum import LineDatum
>>> ld = LineDatum(.66, -2)
>>> ld._class_
<class datum.LineDatum at 867682>
```

你可看到 `_class_` 变量不仅是一个字符串表示类，实际上也是对那个类的引用。如果你知道需要的参数，你可创建一个实例类的实例。这样来延展你的例子：

```
>>> ld2 = ld.__class__(1.2, 6)
>>> ld2.__class__
<class datum.LineDatum at 867682>
```

你可像你使用实类一样检查所有 `instance.__class__` 的类特性，当检查 Java 实例时，这是特别有利的。Java 实例的 `dir()` 并不特别提供关于它的实例成员消息因为用来访问它们的代理的特性。那就意味着能够检查 Java 实例的 `_class_` 字典有助于在交互解释器开发 Java 实例。如果你忘了 `java.io.File` 实例里的可用方法，你可为这些属性检查实例的类。

```
>>> import java
>>> f = java.io.File("c:\\\\python")
>>> dir(f)
[]
>>> dir(f.__class__)
['__init__', 'absolute', 'absoluteFile', 'absolutePath', 'canRead',
'canWrite', 'canonicalFile', 'canonicalPath', 'compareTo', 'createNewFile',
'createTempFile', 'delete', 'deleteOnExit', 'directory', 'exists', 'file',
'getAbsoluteFile', 'getAbsolutePath', 'getCanonicalFile', 'getCanonicalPath',
'getName', 'getParent', 'getParentFile', 'getPath', 'hidden', 'isAbsolute',
'isDirectory', 'isFile', 'isHidden', 'lastModified', 'length', 'list',
'listFiles', 'listRoots', 'mkdir', 'mkdirs', 'name', 'parent', 'parentFile',
'path', 'pathSeparator', 'pathSeparatorChar', 'renameTo', 'separator',
'separatorChar', 'setLastModified', 'setReadOnly', 'toURL']
```

## 2. `_dict_`

这表示实例的名字空间。除了包含实例属性外，它是像类 `_dict_` 一样的概念。你能像类 `_dict_` 一样读和改变实例的 `_dict_` 的内容。

## 7.3 一般定制的特殊方法

虽然四个对象一般定制方法的三个已在第 6 章“类、实例和继承”里被介绍了，但它们在这里重申以使本章对特殊属性有一个更加完整的介绍。

### 1. `_init_`

`_init_` 方法是 Jython 对象的构造函数；它为实例创建而调用。需要显式初始化的 Jython 超类应该用一个具有 `baseclass.__init__(self, [args...])` 的构造函数初始化。用同样的语法显式初始化 Java 超类。如果 Java 超类不是显式初始化，在 Jython 派生类的 `_init_` 方法完成时，Java 超类的空构造函数被调用。关于构造函数更多的见第 6 章。

### 2. `_del_`

`_del_` 方法是 Jython 对象的析构函数或终止器。它不接受任何自变量，因此它的参数列表仅包含 `self`。关于垃圾收集器将什么时候收集对象和调用 `_del_` 方法没有任何保证。Java 甚至根本不保证它被调用。因为这点，最好设计对象以使 `_del_` 方法的内容为最小或终止器是不必要的。也要注意定义了 `_del_` 的 Jython 类引起执行困难。Java 超类的 `finalizing` 方法与 Jython 实例的 `_del_` 方法一起被自动调用，但当需要执行 Jython 超类析构函数时，Jython 超类析构函数必须被显式调用。调用父类的析构函数的语法在这里示范：

```
>>> class superclass:
...     def __del__(self):
```

```

...
    print "superclass destroyed"
...
>>> class subclass(superclass):
...     def __del__(self):
...         superclass.__del__(self)
...         print "subclass destroyed"
...
>>> s = subclass()
>>> del s
>>>
>>> # wait for a while and hit enter a few time until GC comes around
superclass destroyed
subclass destroyed

```

Java 或 Python finalizing 方法产生的异常都被忽略。产生的异常具有惟一的影响是 finalizing 方法在异常点而不是运行到完成点返回。

### 3. \_\_repr\_\_

`__repr__`方法提供一个具有字符串转换行为的对象。反向引用或 `repr()` 内置方法的使用调用对象的`__repr__`方法。同样，如果没有任何`__str__`属性存在对象里，当它被打印时，对象的`__repr__`方法被调用。`__repr__`方法应该返回一个有效的 Python 表达式作为表示对象的形式数据结构的字符串。如果适当的表达式是不可能的，约定在角括号（`<>`）里进行技术描述。假如你有一个像列表一样作用的对象。这个对象的`__repr__`方法应该返回像列表（如`'[1,2,3,4]'`）的字符串。

### 4. \_\_str\_\_

当对象被打印或内置`str()`方法被用在对象上时，`__str__`方法提供调用对象的非形式表示。这与`__repr__`不同。`__repr__`方法返回一个对象的表达式或数据的完整表示，而`__str__`方法通常返回对象的简短描述或特性。

程序清单 7-2 示范了两个特殊方法`__str__`和`__repr__`的实现。程序清单 7-2 里的类实现了这两个方法，因此可以有规范的对象表示（`__repr__`结果）和 HTML 表示（`__str__`结果）。

#### 程序清单 7-2 实现`__str__`和`__repr__`

```

# file: html.py
class HtmlMetaTag:
    """Constructor requires "name" field of metatag.
    Use the instance's "append" method to add to the list"""
    def __init__(self, name):
        self.name = name
        self.list = []

    def append(self, item):
        self.list.append(item)

    def __repr__(self):
        return '{' + 'name':self.name, 'list':self.list}'

    def __str__(self):
        S = '<meta name="%s" content="%s">'
        return S % (self.name, ", ".join(self.list))

```

```

if __name__ == '__main__':
    mt = HtmlMetaTag("keywords")
    map(mt.append, ['Jython', 'Python', 'programming'])
    print "The __str__ results are:\n", mt
    print
    print "The __repr__ results are:\n", repr(mt)

```

运行 jython html.py 的结果是：

```

The __str__ results are:
<meta name="keywords" content="Jython, Python, programming">

The __repr__ results are:
{'name': 'keywords', 'list': ['Jython', 'Python', 'programming']}

```

## 7.4 动态的属性访问

Jython 允许程序员定制具有相应特殊方法 `_getattr_`、`_setattr_` 和 `_delattr_` 的实例属性的访问 (access)、设置 (setting) 和删除 (deletion)。这些方法之间有一种隐含的相互关系，但不需要定义某一种关系。如果你想动态访问，定义 `_setattr_`，如果你想动态的属性删除，定义 `_delattr_`。这些与使用一般的属性访问不同的是它们是动态的：在运行期间请求这些属性时它们需要求值。

### 1. `_getattr_`

在缺少动态的属性查找的类里，访问一个不存在的属性是 `AttributeError`：

```

>>> class test:
...     pass
...
>>> t = test()
>>> t.a
Traceback (innermost last):
  File "<console>", line 1, in ?
AttributeError: instance of 'test' has no attribute 'a'

```

一个动态属性访问的小例子是避免像 `AttributeError` 在程序清单 7-3 里被处理的能力。把动态属性访问加到实例需要定义 `_getattr_` 方法。这个方法必须有两个参数位置，第一个是为 `self`，第二个是为属性名。一旦 `_getattr_` 方法被定义，用传统方式查找失败的实例属性查找继续调用 `_getattr_` 方法来实现请求。程序清单 7-3 是一个模块，该模块包含一个通过提供一个缺省 `_getattr_` 值 `None` 来只避免 `AttributeError` 的类。

### 程序清单 7-3 增加动态属性访问类

```

# file: getattr.py
class test:
    a = 10
    def __getattr__(self, name):
        return None

if __name__ == "__main__":
    t = test()

```

---

```
print "The value of t.a is:", t.a
print "The value of t.b is:", t.b
```

---

运行 `python getattr.py` 的结果如下：

```
The value of t.a is: 10
The value of t.b is: None
```

程序清单 7-3 里的 `_getattr_` 为缺少的属性提供了一个缺省值 `None`。它是一个使用 `_getattr_` 的简洁的例子，但应该说明在设计时它有点不可信。如果 `_getattr_` 实现不能计算一个有用的值，它通常返回一个有用的值或产生一个 `AttributeError`。有时缺省值在某种程度上是适合的，但它们也能隐藏设计缺陷。程序清单 7-4 更加接近像一般的 `_getattr_` 实现。在程序清单 7-4 里使用的有价值的法则是为定位对象属性而使对象从类和实例分开。这是有价值的原因与 `_getattr_` 和 `_setattr_` 两者之间的不对称有关，这将在后面解释。在程序清单 7-4 里用来保存实例属性的数据对象是一个模块级的字典，但它容易就像是列表、另一个类的实例或网络资源一样。它都依赖于在 `_getattr_` 方法（和 `_setattr_`）里所做的。

#### 程序清单 7-4 从单独的对象提供的属性

---

```
# file extern.py
data = {"a":1, "b":2}

class test:
    def __getattr__(self, attr):
        if data.has_key(attr): # lookup attribute in module-global "data"
            return data[attr]
        else:
            raise AttributeError

if __name__=="__main__":
    t = test()
    print "attribute a =", t.a
    print "attribute b =", t.b
    print "attribute c =", t.c # doesn't exist in "data"- is error
```

---

运行 `python extern.py` 的结果如下：

```
attribute a = 1
attribute b = 2
attribute c =Traceback (innermost last):
  File "extern.py", line 15, in ?
AttributeError: instance of 'test' has no attribute 'c'
```

像早先提到的一样，仅在传统的查找失败后才调用 `_getattr_` 方法。什么是传统的查找方法？程序清单 7-3 证实它必须明显地包含类变量；否则输出将不包含 10。典型的方案是属性查找从实例字典开始，然后到初始化的基类的实例字典、类字典，最后到基类字典。仅在这些查找失败后，Python 调用 `_getattr_` 方法。在程序清单 7-3 里，查找属性 `a` 没有通过 `_getattr_`，因为 `a` 在类

`_dict_`里没有找到。

在实例初始化里的一个 `catch` 是如果你在派生类里定义一个 `_init_` 方法，你必须在 Jython 超类里显式调用 `_init_` 来确保正确的实例查找。如果你没有定义 `_init_` 方法，超类构造函数被自动调用，示范如下：

```
>>> class A:
...     def __init__(self):
...         self.val = "'val' found in instance of superclass"
...
>>> class B(A):
...     pass
...
>>> c = B()
>>> c.val
"'val' found in instance of superclass"
```

如果你在一个派生类里定义一个 `_init_`，但调用 Jython 超类的构造函数失败，属性不能在超类的实例里被解释。

```
>>> class B(A): # assume class A is the same as the previous example
...     def __init__(self):
...         pass
...
>>> c = B()
>>> c.val
Traceback (innermost last):
  File "<console>", line 1, in ?
AttributeError: instance of 'B' has no attribute 'val'
```

这个初始化 `catch` 没有应用到 Java 超类。如果一个 Java 超类不是显式地被初始化，它的空构造函数在派生类 `_init_` 方法完成时被调用，实例属性查找正常进行。

## 2. `__setattr__`

把动态属性赋值给实例需要 `__setattr__` 方法。这个方法必须有三个参数位置，第一个是为 `self`，第二个是为属性名，第三个是赋给属性的值。一旦被定义，`__setattr__` 方法截听所有除了隐式定义的 `_class_` 和 `_dict_` 以外赋给实例属性的值。因为这点，在没有创建循环查找和溢出异常的情况下，你不能在 `__setattr__` 方法里直接设置实例属性。然而你可以重新绑定没有这样错误的实例，因为它免除了 `__setattr__` 钩子，你也可以直接访问 `_dict_` 因为它免除了 `__getattr__`。

程序清单 7-5 示范了使用 `__setattr__` 方法来限制域类型在整数范围内。另外，`__setattr__` 和 `__getattr__` 方法两个都允许实例属性存储在数据对象里而不是实例 `_dict_` 里。而程序清单 7-5 使用一个调用 `_data` 的 Hashtable 来存储任何赋值的实例域。

在类构造函数里，`_data` 的赋值不使用 `self._data = ……`，而直接使用实例 `_dict_`。为什么？包括这些在构造函数里的实例赋值现在都通过 `__setattr__`；然而 `__setattr__` 希望在实例 `_dict_` 里有一个 `_data` 键。这个矛盾可通过用 `self._dict_[key] = value` 语法直接增加 `_data` 到实例字典来避免，并因此避免了 `__setattr__` 钩子。

程序清单 7-5 里另一个有价值的地方是 `__setattr__` 方法确保实例变量不是存储在实例的 `_dict_`

里。为什么这是有益的？为了理解它的作用，我们首先看一看`_setattr_`和`_getattr_`两者之间的不对称性。`_setattr_`总是截听实例属性赋值，但`_getattr_`仅当标准的属性查找失败时才调用。如果你想每次`get`和`set`执行一些对称的动作，如总是访问和存储外部数据库的值，这就坏了。使实例值在`instance_dict`外确保它们不被找到和`_getattr_`被调用。在超类里的实例变量和类变量简化了这种控制，因此在派生类里详细设计是应该的。同样，注意在调用`_getattr_`方法前考虑到标准查找失败，在一定程度上对于每个`_getattr_`方法调用会有一个与之相适应的性能。

#### 程序清单 7-5 使用`_setattr_`和`_getattr_`

```
# file: setter.py
from types import IntType, LongType
import java

class IntsOnly:
    def __init__(self):
        self.__dict__['_data'] = java.util.Hashtable()

    def __getattr__(self, name):
        if self._data.containsKey(name):
            return self._data.get(name)
        else:
            raise AttributeError, name

    def __setattr__(self, name, value):
        test = lambda x: type(x)==IntType or type(x)==LongType
        assert test(value), "All fields in this class must be integers"
        self._data.put(name, value)

if __name__ == '__main__':
    c = IntsOnly()
    c.a = 1
    print "c.a=", c.a
    c.a = 200L
    print "c.a=", c.a
    c.a = "string"
    print "c.a=", c.a # Shouldn't get here
```

运行`python setter.py`的结果如下：

```
c.a= 1
c.a= 200
Traceback (innermost last):
  File "setter.py", line 25, in ?
    File "setter.py", line 17, in __setattr__
AssertionError: All fields in this class must be integers
```

#### 3. `_delattr_`

增加动态属性删除需要定义`_delattr_`方法。这个方法必须有两个参数位置，第一个是为`self`，第二个是为属性名。当使用`del object.attribute`时，`_delattr_`方法被调用。在删除前这个属性可以是需要排空（flushing）/关闭（closing）的资源，也可以是需要从数据库或文件系统删除

的部分持久性资源，或是一个你不想你的类的用户要删除的属性。它也可以是属性被存储在数据对象里而不是实例`_dict_`里需要对删除做特殊的处理，程序清单 7-5 需要这些。`_delattr_`钩子允许程序员适当地处理这类情况。程序清单 7-6 例子使用`_delattr_`钩子来阻止属性的删除。

#### 程序清单 7-6 使用`_delattr_`来保护属性不被删除

---

```
# file: immortal.py

class A:
    def __init__(self, var):
        self.immortalVar = var
    def __delattr__(self, name):
        assert name!="immortalVar", "Cannot delete- it's immortal"

        del self.__dict__[name]

c = A("some value")print "The immortalVar=", c.immortalVar
del c.immortalVar
```

---

运行 `jython immortal.py` 的结果如下：

```
The immortalVar= some value
Traceback (innermost last):
  File "immortal.py", line 12, in ?
  File "immortal.py", line 7, in __delattr__
AssertionError: Cannot delete- it's immortal
```

## 7.5 可调用的钩子：`_call_`

特殊方法`_call_`使实例可调用。`call`方法的参数个数没有受任何限制。按照基本标准，这使实例像函数一样动作。创建一个像函数一样打印简单消息的实例，如下所示：

```
>>> class hello:
...     def __call__(self):
...         print "Hello"
...
>>> h = hello()
>>> h() # call the instance as if it were a function
Hello
```

`hello`例子有些容易被误解，因为它掩饰了`_call_`方法的真实潜力。程序清单 7-7 是一个稍微更加有趣的例子，它用一个实现`_call_`方法的内部类仿真`static`方法。程序清单 7-7 里的内部类得到一个`java.lang.Runtime`实例，并为运行系统命令和返回它的输出定义了一个`_call_`方法。程序清单 7-7 里的内部类被命名为`_static_runcommand`。用户将调用的是内部类的实例。记住，因为`_call_`实例就是什么是可调用的和什么成为`static`方法。这个类的用户将实例化外部类`commands`。让我们调用这个实例`A`，然后用`A.runcommand(command)`调用`runcommand`实例。`runcommand`实例看起来和动作都像方法。尽管外部`commands`类的许多实例，但仅需要`_static_runcommand`的一个单实例和`java.lang.Runtime`的一个单实例（这假设不同时需要）。

**程序清单 7-7 用\_call\_和内部类仿真 static 方法**

```
# file: staticmeth.py
from java import lang, io

class commands:
    class _static_runcommand:
        'inner class whose instance is used to fake a static method'
        rt = lang.Runtime.getRuntime()
        def __call__(self, cmd):
            stream = self.rt.exec(cmd).getInputStream()
            isr = io.InputStreamReader(stream)
            results = []
            ch = isr.read()
            while (ch > -1):
                results.append(chr(ch))
                ch = isr.read()
            return "".join(results)

        runcommand = _static_runcommand() # create instance in class scope

    if __name__ == '__main__':
        inst1 = commands()
        inst2 = commands()

    # now make sure runcommand is static (is shared by both instances)
    assert inst1.runcommand is inst2.runcommand, "Not class static"

    # now call the "faked" static method from either instance
    print inst1.runcommand("mem") # for windows users
    #print inst1.runcommand("cat /proc/meminfo") # for linux users
```

在 Windows 2000 上运行 jython staticmeth.py 的结果如下：

```
655360 bytes total conventional memory
655360 bytes available to MS-DOS
633024 largest executable program size

1048576 bytes total contiguous extended memory
0 bytes available contiguous extended memory
941056 bytes available XMS memory
MS-DOS resident in High Memory Area
```

## 7.6 特殊的比较方法

比较方法是运算符`==`、`!=`、`<`、`<=`、`>`和`>=`的使用。这对许多像整数（`5 > 4`）这样的对象是简单的，但对用户定义的类又怎么样呢？Jython 允许实现了比较方法的定义，但 Jython2.1 版（Python2.1 版）在实现类比较方法引进了一些改变。这个新的特征是多比较（rich comparision）。为了进行对照，旧的比较被称做少比较（poor comparisons）。

### 7.6.1 少比较方法

少比较方法需要一个称做`_cmp_`的单个特殊比较方法，它根据`self`是小于、等于还是大于另一个对象来返回-1、0还是1。`_cmp_`方法必须有两个参数位置，第一个是为`self`，第二个是为另一个对象。比较详细形式的是：

```
def __cmp__(self, other):
    if (self < other):
        return -1
    if (self == other):
        return 0
    if (self > other):
        return 1
```

程序清单7-8使用了`_cmp_`方法和一个类属性`role`来为家庭圈子的一列成员决定排列次序。内置`cmp()`函数按照被类设置的`role`决定的合适的类值被使用。如果`other`类不是`family`类的实例，它总当作小子来比较（注意“eldest”使事物颠倒，因此小实际上是多）。注意在这个例子里设置类实例`flag`、`role`不是控制更加复杂的类里的排序行为的首选方式。

#### 程序清单7-8 按照`role`进行比较

---

```
#file: poorcompare.py
class family:
    role = "familyMember" # default value

    def __init__(self, name, age, relation, communicationSkills):
        self.name = name
        self.age = age
        self.relation = relation
        self.communicationSkills = communicationSkills
        self._roles = {1:"familyMember",
                      2:"communicator",
                      3:"eldest"}

    def __cmp__(self, other):
        if other.__class__ != self.__class__:
            return -1 # non-family classes are always less
        if self.role=="familyMember":
            relations = {"mother":1, "father":1, "aunt":2, "uncle":2,
                         "cousin":3, "unrelated":4}
            return cmp(relations[self.relation],
                       relations[other.relation])
        elif self.role=="communicator":
            return cmp(self.communicationSkills,
                       other.communicationSkills)
        elif self.role=="eldest":
            return cmp(other.age, self.age) #, other.age)

    def __repr__(self):
        # This is an abuse of __repr__. 'canonical' data not returned.
        # Included only for sake of example.
        return self.name

if __name__ == '__main__':
```

```

L = []
# add ppl to list
L.append(family("Fester", 80, "uncle", 2))
L.append(family("Gomez", 50, "father", 1))
L.append(family("Lurch", 75, "unrelated", 3))
L.append(family("Cousin It", 113, "cousin", 4))
L.append("other data-type")

# print list sorted by default role
L.sort()
print "by relation:", L

# print list sorted by communication skills:
family.role = "communicator"
L.sort()
print "by communication skills:", L

# print list eldest to youngest
family.role = "eldest"
L.sort()
print "eldest to youngest:", L

```

运行 `python poorcompare.py` 的结果如下：

```

by relation: [Gomez, Fester, Cousin It, Lurch, 'other data-type']
by communication skills: [Gomez, Fester, Lurch, Cousin It, 'other data-type']
eldest to youngest: [Cousin It, Fester, Lurch, Gomez, 'other data-type']

```

### 7.6.2 多比较方法

多比较方法出现在 Jython2.1 版，并没有像 `_cmp_` 一样被限制在 -1、0、1 返回值范围内。如果比较两个列表或两个矩阵，你可以返回包含元素方向的比较、另一个对象、None、NotImplemented、一个 Boolean 或产生一个异常的列表或矩阵。特殊的 rich comparison 方法是一组表示六个 comparison 运算符的六个特殊方法。每一个方法需要两个参数：第一个是为 `self`，第二个是 `other`。表 7-1 列出了运算符和相关的多比较方法。

表 7-1 多比较方法

运算符	方 法
<	<code>_lt_(self, other)</code>
<=	<code>_le_(self, other)</code>
==	<code>_eq_(self, other)</code>
!=	<code>_ne_(self, other)</code>
>	<code>_gt_(self, other)</code>
>=	<code>_ge_(self, other)</code>

对对象 A 和 B，`A < B` 的多比较变成 `A._lt_(B)`。如果 `comparison` 是 `B > A`，求值方法 `B._gt_(A)`。每个运算符有一个自然的 compliment，但没有任何不变量的强制，例如 `A._lt_(B) == B._gt_(A)`。左边的对象首先为一个合适的多比较方法所搜寻，如果它没有被定义，则右边的对象为 compliment 方法所搜寻。如果两边定义了一个合适的方法，左边对象的方法被使用。

程序清单 7-9 定义两个类：A 和 B。类 A 定义所有六个多比较方法以便它们把 self 的类名与 other 类名进行比较。类 B 定义惟一一个 comparison 方法：greater-than (`_gt_`)。注意在类 B 的 `_gt_` 方法里的比较的手段是通过实例创建时间戳；与类 A 的可比较性定义不相宜的。这样分歧的定义是有些麻烦和在没有强烈的理由情况下会被警告。程序清单 9-7 剩下的会通过测试使用每个类的一个实例的比较结合。你可从输出看到合适的比较方法是怎么被解决的。

### 程序清单 7-9 多比较方法

```
# file: rich.py
import time

class A:
    def __init__(self):
        self.timestamp = time.time()

    def __lt__(self, other):
        print "...Using A's __lt__ method...", 
        return self.__class__.__name__ < other.__class__.__name__

    def __le__(self, other):
        print "...Using A's __le__ method...", 
        return self.__class__.__name__ <= other.__class__.__name__

    def __ne__(self, other):
        print "...using A's __ne__ method...", 
        return self.__class__.__name__ != other.__class__.__name__

    def __gt__(self, other):
        print "...Using A's __gt__ method...", 
        return self.__class__.__name__ > other.__class__.__name__

    def __ge__(self, other):
        print "...Using A's __ge__ method...", 
        return self.__class__.__name__ >= other.__class__.__name__

    def __eq__(self, other):
        print "...Using A's __eq__ method...", 
        return self.__class__.__name__ == other.__class__.__name__

class B:
    def __init__(self):
        self.timestamp = time.time()

    def __gt__(self, other):
        print "...Using B's __gt__ method...", 
        return self.timestamp > other.timestamp

if __name__ == '__main__':
    inst_b = B()

    inst_a = A()
    print 'Is a < b?', inst_a < inst_b
    print 'Is b < a?', inst_b < inst_a
    print 'Is a <= b?', inst_a <= inst_b
```

```

print "Is b <= a?", inst_b <= inst_a
print "Is a == b?", inst_a == inst_b
print "Is b == a?", inst_b == inst_a
print "Is a != b?", inst_a != inst_b
print "Is b != a?", inst_b != inst_a
print "Is a > b?", inst_a > inst_b
print "Is b > a?", inst_b > inst_a
print "Is a >= b?", inst_a >= inst_b
print "Is b >= a?", inst_b >= inst_a

```

运行 jython rich.py 的结果是：

```

Is a < b? ...Using A's __lt__ method... 1
Is b < a? ...Using A's __gt__ method... 0
Is a <= b? ...Using A's __le__ method... 1
Is b <= a? ...Using A's __ge__ method... 0
Is a == b? ...Using A's __eq__ method... 0
Is b == a? ...Using A's __eq__ method... 0
Is a != b? ...using A's __ne__ method... 1
Is b != a? ...using A's __ne__ method... 1
Is a > b? ...Using A's __gt__ method... 0
Is b > a? ...Using B's __gt__ method... 0
Is a >= b? ...Using A's __ge__ method... 0
Is b >= a? ...Using A's __le__ method... 1

```

程序清单 7-9 阐明了多比较方法，但对实际的应用是无意义的。似乎更加有理的用法是列表对象元素方向的比较。程序清单 7-10 定义了一个称做 listemulator 的类，它包含了一个 `_lt_` 方法定义。由于在列表开始引入的 `UserList` 类的帮助下，`Listemulator` 类行为像列表。仿真其他类型的细节将在本章的后面介绍，但是到现在，让我们假设 `Listemulator` 类的实例真正像 Jython 列表一样动作，除了 `_lt_comparison` 外。程序清单 7-10 里的 `_lt_` 方法做两件事。首先它比较它自己和 `other` 的长度来确保元素方向的比较是合法的（它们是相同的长度）。然后，`_lt_` 方法比较 `self` 和 `other` 的每一个元素，并返回比较结果列表。

#### 程序清单 7-10 元素方向的多比较

```

# file: richlist.py
from UserList import UserList

class listemulator(UserList):
    def __init__(self, list):
        self.data = list
        UserList.__init__(self, self.data)

    def __lt__(self, other):
        if len(self) != len(other):
            raise ValueError, ("Instance of %s differs in size from %s"
                % (self.__class__.__name__,
                   other.__class__.__name__))
        return map(lambda x, y: x < y, self, other)

L = [2,3,4,5]
LC = listemulator([2,3,3,4])
print LC < L

```

运行 `jython richlist.py` 的结果是：

```
[0, 0, 1, 1]
```

`Dictionary` 运算依赖那些用作键的对象的 `hash` 值。`Jython` 对象可以为字典键运算和定义特殊 `_hash_` 实例方法的内置 `hash` 函数确定它们自己的 `hash` 值。`_hash_` 方法仅有一个参数，就是 `self`，返回值应是整数。实现 `_hash_` 的约束是相同值的对象必须返回相同的 `hash` 值。因为字典键是不变的，定义比较方法但没有任何 `_hash_` 方法的对象不能用作字典键。

## 7.7 对象“真值”

在 `Jython` 对象里搜索真值要遵守以下规则：

- 如果一个类定义了特殊的 `_nonzero_` 方法，它的返回值（1 或 0）确定对象的“真值”。
- 如果一个对象没有定义了特殊的 `_nonzero_` 方法，则解释器调用特殊方法 `_len_`。`_len_` 方法出现在本章后面，但它的意义是非常直观的。它返回一个表示对象长度的整数。如果这个返回值是非零的，则对象是真。
- 如果类没有定义上面两个特殊方法中的任何一个，那么那个类的实例总是真。

用 `_nonzero_` 方法实现真，如下所示：

```
import random

class gamble:
    def __nonzero__(self):
        return random.choice([0,1])
```

`_nonzero_` 参数列表包含惟一的 `self`，返回值对 `true` 是 1，对 `false` 是 0。

## 7.8 仿真内置数据对象

许多场合要求创建仿真数据对象的类。或许设计需要 `Jython` 字典，仅仅需要那个字典来维持顺序，或者也许你需要具有特殊查找的列表。仿真内置对象允许你扩展它们的行为、用最小的工作增加约束和工具对象运算，但以常见的接口结束。由于对内置接口的精通，实现扩展的行为几乎不增加复杂性，这是对象的主要目的。`Jython` 的特殊方法允许用户定义的类来仿真 `Jython` 的内置数字、序列和映射对象。有相关方法的对象的仿真也像那些非特殊方法的实现，也真正仿真那些对象。

这一节里的例子常常用 `Jython` 类来举例说明这种功能，该 `Jython` 类在内部使用 `Java` 对象。这并不总是需要的。`Jython` 对于转换类型来适应环境是非常好的。`Java.util.Vector` 对象已经支持 `PyList` 索引语法 (`v[index]`)，`java.util.Hashtable` 和 `java.util.HashMap` 已经支持键赋值 (`h[key]`)，并且数字对象在需要的地方自动转换成合适的类型。由于有了这种对 `Java` 对象的实质性直观的支持，常常没有必要在特殊方法里封装它们；然而也有点像在 `Jython` 里不支持 `slice` 语法的 `java.util.Vector`。

```
>>> import java
>>> v = java.util.Vector()
>>> v.addElement(10)
```

```
>>> v.addElement(20)
>>> v[0] # this works
10
>>> v[0:2]
Traceback (innermost last):
  File "<console>", line 1, in ?
TypeError: only integer keys accepted
```

仿真内置类型允许你来指定每个行为以确保对象与非常相似于内置类型一样动作。因为 Jython 和 Java 地集成是如此好，在语言之间传递对象是普遍深入的。允许 Java 对象更好地仿真 Jython 内置是非常方便的，因此你的代码用户不必关心哪些是 Java、哪些不是 Java。Java 对象常常已经包含方法，该方法像 Jython 内置里的 comparable 方法一样做同样的事情，但命名不同。

程序清单 7-11 给出了一个方便的方式来映射这样的 Java 方法到 Jython 方法，以使仿真内置对象更加容易。在程序清单 7-11 里的 HashWrap 类是赋类标志符给 Hashtable 方法的 java.util.Hashtable 的派生类，该 Hashtable 方法已经执行了希望的行为。注意 HashWrap 类没有定义 values()、clear() 或 get()。这些名字已存在超类 java.util.Hashtable 中，并执行非常接近你所希望的。仅程序清单 7-11 的 catch 是一些 Hashtable 函数返回不希望的类型，例如从 keys() 和 items() 返回的 Enumeration。这些方法被包含在一个简单的 lambda 表达式来转换它们成一个列表。一些 Jython 的字典方法没有与 Java Hashtable 的直接并行，因此 setdefault()、popitem() 和 copy() 没有在 HashWrap 类里定义。

程序清单 7-11 也包含了特殊方法——那些以两个下划线开始和结束的方法。这些特殊方法的意义可以从它们被映射到的 Java 方法里识别，但在这点上的意义仅仅是给出了怎样映射标志符到 Java 方法。

#### 程序清单 7-11 把 Java 方法赋给 Jython 类标志符

```
# file: hashwrap.py
import java
import copy

class HashWrap(java.util.Hashtable):
    #map jython names to Hashtable names
    has_key = java.util.Hashtable.containsKey
    update = java.util.Hashtable.putAll

    # Hashtable returns an Enumeration for
    keys = lambda self: map(None, java.util.Hashtable.keys(self))
    items = lambda self: map(None, java.util.Hashtable.elements(self))

    # these don't have direct parallels in Hashtable, so define here
    def setdefault(self, key, value):
        if self.containsKey(key):
            return self.get(key)
        else:
            self.put(key, value)
            return value

    def popitem(self):
        return self.remove(self.keys()[0])
```

```

def copy(self):
    return copy.copy(self)

# These are the special methods introduced in this section.

# Read on to find out more.
__getitem__ = java.util.Hashtable.get
__setitem__ = java.util.Hashtable.put
__delitem__ = java.util.Hashtable.remove
__repr__ = java.util.Hashtable.toString
__len__ = java.util.Hashtable.size

if __name__ == '__main__':
    hw = HashWrap()
    hw["A"] = "Alpha"
    hw["B"] = "Beta"
    print hw
    print hw.setdefault("G", "Gamma")
    print hw.setdefault("D", "Delta")
    print hw['A']
    print "keys=", hw.keys()
    print "values=", hw.values()
    print "items=", hw.items()

```

运行 `python hashwrap.py` 的输出如下：

```

{A=Alpha, B=Beta}
Gamma
Delta
Alpha
keys= ['A', 'G', 'D', 'B']
values= [Alpha, Gamma, Delta, Beta]
items= ['Alpha', 'Gamma', 'Delta', 'Beta']

```

### 7.8.1 仿真序列

内置序列出现了两种风格：可变的和不可变的。Immutable sequences(PyTuples) 没有任何相关的方法，而 mutable sequences(PyLists) 有相关的方法；两种风格有相似的序列行为，例如索引(indexes) 和分片(slices)。真正地仿真一个 PyList 将包括定义它的相关方法(`append`、`count`、`extend`、`index`、`insert`、`pop`、`remove`、`reverse` 和 `sort`) 和与序列长度、索引和分片相关的特殊方法。仿真不可变序列(PyTuples) 仅需要特殊方法的一个子集作为对可变对象(它们改变对象内容)是惟一的实现操作的那些方法。用户定义的对象不必实现所有序列行为，因此你可自由地仅定义那些适合你的设计的方法，然而，定义所有序列方法允许用户不知道你的类和内置数据对象之间的不合理的区别，这对抽象、重复使用和对象梦寐以求的简化复杂性是真正有好处的。

下面小节描述序列行为和它们相关的特殊方法。这一小节的描述假设一个序列 S、一个 PyList 和一个 PyTuple，使用 `sequence` 和 S 说明通常用于序列的特殊函数，使用 PyTuple 和 T 说明不可变序列的实现，使用 PyList 和 L 说明对可变序列的注释。

## 1. \_\_len\_\_

序列应该有一个等于 len(S) 的长度。

返回一个对象长度的特殊方法是 \_\_len\_\_(self)。调用 len(S) 等同于 S.\_\_len\_\_(self)。\_\_len\_\_(self) 方法必须返回一个大于或等于零的整数。注意没有任何强制 \_\_len\_\_ 精度的方法。一个有十个元素像列表一样的对象可以定义一个返回 1 的 \_\_len\_\_ 方法。

程序清单 7-12 把序列元素保存在 java.util.Vector 实例。length 方法从 vector 的 size() 方法返回结果作为对象的长度。

程序清单 7-12 实现序列 length

```
# file: seqlen.py
import java

class usrList:
    def __init__(self):
        # use name-mangled, private identifier for vector
        self.__data = java.util.Vector()

    def append(self, o):
        self.__data.add(o)

    def __len__(self):
        return self.__data.size()

if __name__ == '__main__':
    L = usrList()
    L.append("A")
    L.append("B")
    L.append("C")
    print "The length of object L is:", len(L)
```

运行 jython seqlen.py 的输出如下：

```
The length of object L is: 3
```

## 2. \_\_getitem\_\_

S[i] 获得序列索引 i 处的值。索引 i 可以是一个正整数（从序列左边记数）、一个负整数（从右边记数）或一个 slice 对象。

用来返回被一个特定的索引或 slice 指定的对象的特殊方法是 \_\_getitem\_\_(self, index)。调用 S[i] 等同于 S.\_\_getitem\_\_(i)。当指定的索引超出范围时，\_\_getitem\_\_ 方法应产生一个 IndexError 异常，或因不支持的索引类型而产生一个 ValueError 异常。为了真正仿真内置序列，你必须考虑一个正整数、负整数或 slice 对象的 index 值。

\_\_getitem\_\_ 方法有时候与特殊类属性方法 \_\_getattribute\_\_ 相混淆，这与它们的区别没关系。\_\_getitem\_\_ 方法检索对象定义为列表或映射入口（映射实现在后面出现），而不是对象属性。另外，\_\_getitem\_\_ 为每个 item 检索所调用，不像 \_\_getattribute\_\_，是仅在标准对象属性查找失败之后才被调用。因为 \_\_getitem\_\_ 总是被调用，为实现持久和其他需要在设计和获取对象之间的对称性的特殊行为，它是

一个好的候选。

程序清单 7-13 是一个像列表一样的、包含 `java.util.Vector` 的类，在这方面它与程序清单 7-12 相似。现在我们使用这个概念来举例说明 `_getitem_` 方法。在程序清单 7-13 里的 `_getitem_` 实现按照把指定的索引值转换成一列正整数来考虑正数、负数和 slice 索引。

程序清单 7-13 用 `_getitem_` 进行序列项检索

```
# file: seqget.py
import java
import types

class usrList:
    def __init__(self, initial_values):
        data = java.util.Vector()
        map(data.add, initial_values)
        self.__data = data

    def __getitem__(self, index):
        indexes = range(self.__data.size())[index]
        try:
            if not isinstance(indexes, types.ListType):
                return self.__data.elementAt(indexes)
            else:
                return map(self.__data.elementAt, indexes)
        except java.lang.ArrayIndexOutOfBoundsException:
            raise IndexError, "index out of range: %s" % index

if __name__ == '__main__':
    S = usrList(range(1,10))
    print "S=", S[:]
    print "S[3]=", S[3]
    print "S[-2]=", S[-2]
    print "S[1:7:2]=", S[1:7:2]
    print "S[-5:8]=", S[-5:8]
    print "S[-8:]=", S[-8:]
```

实现应该考虑正数、负数和 slice 索引，应该考虑在合适的地方产生 `IndexError` 和 `ValueError` 异常。记住负数索引意味着它们是从右边开始记数，`-1` 是最后的序列项，`-2` 是倒数第二个等等。因为缺少 slice 元素 Jython 分片规则假设为缺省值，因此用户定义的像列表的对象也应该考虑这点。对 `slice[::3]`，最初两个缺少的值 Jython 假定序列的开始和序列的结束，然后使用 3 作为步长值。在用户定义的对象里实现所有这些可能听起来令人畏惧，但它不一定是困难的或复杂的。简化复杂性的重要工具影响在通常的对象里已发现的功能。所有这些的前提是 `_getitem_` 方法仿真内置 `Pylist` 对象的行为，使用 `_getitem_` 实现里的列表对象。如果内部数据是在矢量（vector）里，使一个列表为 vector 的大小，并应用索引或 slice 到列表——你现在已处理了正数、负数和 slice 索引，也处理了缺省索引和 `IndexError`、`ValueError` 异常。那是在程序清单 7-13 使用的技巧。这是使刚才的列表技巧更加清楚的一个例子：

```
>>> import java
>>> v = java.util.Vector()
>>> map(v.addElement, range(10))
```

```
[None, None, None, None, None, None, None, None, None]
>>> v # take a peek at the vector
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # create a slice object
>>> i = slice(2, -3, 4)
>>>
>>> ## Next line handles all slice logic, and appropriate exceptions
>>> indexes = range(v.size())[i]
>>> indexes
[2, 6] # remember- these are indexes of values, not values
```

而 `java.util.Vector` 对象不支持 `slice`, `PyList` 支持 `slice`。生成包含索引数目的 `vector` 范围的一个 `PyList`, 把 `index` 或 `slice` 应用到该列表。结果是 `vector` 索引的一个负的列表或一个单个 `vector` 索引数。使用 `PyList` 也确保任何 `ValueError` 或 `IndexError` 在需要时被产生。搜查方法来重新使用存在的功能, 特别是那些内置的功能, 对简化复杂性是非常重要的。程序清单 7-13 从使用列表技巧获得了单个、正索引或正索引列表。一旦合适的正索引值被确定, `_getitem_` 方法返回合适的单个值或多个值。

运行 `jython seqget.py` 的输出是:

```
S= [1, 2, 3, 4, 5, 6, 7, 8, 9]
S[3]= 4
S[-2]= 8
S[1:7:2]= [2, 4, 6]
S[-5:8]= [5, 6, 7, 8]
S[-8:]= [2, 3, 4, 5, 6, 7, 8, 9]
```

注意在程序清单 7-13 里, 使用矢量的 `elementAt()` 方法是在 `try/except` 里。如果索引超出了范围, 则更加详细的 Java 异常和 `traceback` 被捕获和用 Jython `IndexError` 异常替换。这仅是一个实现选择——没有任何义务来包含 Java 异常, 但 `IndexError` 帮助 `userList` 更加像内置一样动作。

虽然程序清单 7-13 使用 `PyList` 来做脏工作, 但是也可能有实例需要显式的类型处理。这引入了类型测试。在 Jython 里一般与标准类型进行对照测试一个变量的类型, 例如这下面三个例子中的一个:

```
>>> import types
>>> a = 1024L
>>> if type(a) == types.LongType: 'a is a LongType'
...
'a is a LongType'
>>> if type(a) == type(1L): 'a is a LongType'
...
'a is a LongType'

>>> if type(a) in [types.IntType, types.LongType]: 'a is an ok type'
...
'a is an ok type'
```

但当你引入 Java 类型时, 对适当的类型进行测试是有趣的。考虑 Java 类型的充分识别是有点奇怪, 例如, 考虑以下:

```
>>> import java
>>> v = java.util.Vector()
>>> i = java.lang.Integer(3)
>>> type(v) == type(i)
1
```

如果 `type()` 不能区别整数和矢量之间的差别，你怎么考虑有限的 Java 类型？这样做的一个途径是测试对象的类。所有 Jython 对象有一个 `_class_` 属性，同时包括 Java 对象，因此你可以这样做：

```
>>> import java
>>> i = java.lang.Integer(5)
>>> v = java.util.Vector()
>>> ok_types = [(1).__class__, (1L).__class__, java.lang.Integer,
java.lang.Long]
>>> i.__class__ in ok_types
1
>>> v.__class__ in ok_types
0
```

确定对象类型适当性的另一个方法是使用内置 `isinstance` 函数。`isinstance` 函数接受一个对象和一个类作为自变量，如果对象是指定类的实例，则返回 1，否则返回 0。当与继承层次一起作用时，因为 `isinstance` 的合适性使用它来检查类型是首选的。使用 `isinstance` 来检查类型如下所示：

```
>>> import types
>>> a = 1024L
>>> if isinstance(a, types.LongType): "a is a LongType"
...
'a is a LongType'
```

### 3. `_setitem_`

表达式 `L[i] = object` 将把 `object` 绑定到像列表的对象 `L` 里的索引 `i`。这对类是指定的，该类被设计来仿真列表而不是不可变对象（像 `PyTuple`）。只有可变对象实现这种行为。

用来绑定一个对象到指定的索引的特殊方法是 `_setitem_(self, index, value)`。当指定的索引超出范围时，这个方法将产生一个 `IndexError` 异常。这个索引值可以是正的、负的或 `slice` 对象，因为那些 `_setitem_` 实现没有考虑的索引值而产生一个 `ValueError` 异常。

赋值给 `slice` 有一些特殊的约束，至少对内置 `PyList`。在用户定义的对象里你不必考虑这种行为，但它还是值得推荐的。这种约束就是 `step` 值必须是 1，值不必像 `slice` 一样长。首先看一看给一个单索引赋值：

```
>>> S = ['a', 'b', 'c']
>>> S[1] = [1, 2, 3]
>>> S
['a', [1, 2, 3], 'c']
```

赋给索引 1 的值是一个 `list`，这作为在索引 1 里的单对象显示。赋值给 `slice` 不同：

```
>>> S = ['a', 'b', 'c']
>>> S[3:4] = [1, 2, 3]
>>> S
['a', 'b', 'c', 1, 2, 3]
```

还是仅有包含的是索引 3 的 1 索引，但赋值给 slice 意味着在右边必须是一个序列，这方面有些不同，结果列表是串联：

```
S[0:slice.start] + values + S[slice.stop:len(S)]
```

程序清单 7-14 定义一个实现 `_setitem_` 方法的类，但选择实现两个约束：所有值必须是字符串和列表必须是一个在构造函数参数指定的 static size。每个列表索引在内部都表示文件 `userList.dat` 里的一行。设置 `L[2] =some string` 改变文件的第二行到 some string。考虑到所有实例将从单文件读（而这可以用另一个构造函数自变量改变），这使内部列表相似于类 static 变量。在每个方法里的文件的打开和关闭是昂贵的，因此如果这个文件是一个共享资源这才发生。否则持久将在 `_init_` 方法和可能的 `close` 方法里被实现（注意 `_del_` 将作用，但常常被避免，因为 `_del_` 里的异常被忽略和当方法得到调用时没有任何保证）。

程序清单 7-14 用 `_setitem_` 增加持久

```
# file: seqset.py
import types
import os

class usrList:
    def __init__(self, size):
        self.__size = size
        self.__file = "usrList.dat"
        if not os.path.isfile(self.__file):
            f = open(self.__file, "w")
            print >> f, "\n" * size
            f.close()

    def __repr__(self):
        f = open(self.__file)
        L = f.readlines()[:self.__size]
        f.close()
        return str(map(lambda x: x[:-1], L))

    def __setitem__(self, index, value):
        f = open(self.__file, "r+")
        L = f.readlines()[:self.__size]
        if isinstance(index, types.SliceType):
            if len(L[index]) != len(value):
                raise ValueError, "Bad value: %s" % value
            for x in value:
                if not isinstance(x, types.StringType):
                    raise ValueError, "Only String values supported"
            L[index] = map(lambda x: x + "\n", value)

        if (isinstance(index, types.IntType) or
            isinstance(index, types.LongType)):
            if type(value) != types.StringType:
                raise ValueError, "Only String values supported"
            L[index] = value + "\n"
        f.seek(0)
        f.writelines(L)
        f.close()
```

```

if __name__ == '__main__':
    S = usrList(10)
    for x in range(10):
        S[x] = str(x)
    print "First List=", S
    S[4:-4] = "four", "five"
    print "Second List =", S
    for x in range(10, 20):
        S[x-10] = str(x)
    print "Last list = ", S

```

程序清单 7-14 支持赋值给列表 indexes 和 slice，但因为文件行被存在真正的 PyList 中作为中间件，因此这个功能是自动的。程序清单 7-14 适当地产生 ValueError 和 IndexError 异常，但注意 IndexError 将从标准列表操作而不是捕获和重新产生异常传播。

运行 `python seqset.py` 的输出如下：

```

First List= ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
Second List = ['0', '1', '2', '3', 'four', 'five', '8', '7', '8', '9']
Last list = ['10', '11', '12', '13', '14', '15', '16', '17', '18', '19']

```

`usrList` 实例里的值被存储在 `usrList.dat` 文件里，因此 `usrList` 类将来的实例化将从这些值开始。你可在交互解释器里确认这点——确定从与 `usrList.dat` 文件同一个目录里开始解释程序 (`usrList` 实例仅在当前目录查看)：

```

>>> import seqset
>>> L = seqset usrList(10)
>>> print L # <- print persistent values
['10', '11', '12', '13', '14', '15', '16', '17', '18', '19']

```

自动持久数据类型是非常方便的，但程序清单 7-14 的执行不是。使用具有快速的数据库的相同技术很有帮助。

程序清单 7-14 是指导性的，但因为内部数据是一个 PyList，几乎没有关于处理在程序清单 7-14 里赋值给 slice 的唯一约束。程序清单 7-15 使用 `java.util.Vector` 作为内部数据，因此阐述在 `_setitem_` 里支持 slice。

#### 程序清单 7-15 在 List 类里封装 Java Vector

```

#file: seqset1.py
import java
import types

class usrList:
    def __init__(self):
        self.__data = java.util.Vector()
        map(self.__data.addElement, range(10))

    def __getitem__(self, index):
        indexes = range(self.__data.size())[index]
        if isinstance(index, types.SliceType):
            return map(self.__data.elementAt, indexes)
        else:
            return self.__data.elementAt(indexes)

```

```

def __setitem__(self, index, value):
    if isinstance(index, types.SliceType):
        size = self.__data.size()
        if index.step != 1:
            raise ValueError, 'Step size must be 1 for setting list slice'
        newdata = java.util.Vector()
        map(newdata.addElement, range(0, index.start))
        map(newdata.addElement, value)
        map(newdata.addElement, range(index.stop, size))
        self.__data = newdata
    else:
        self.__data.setElementAt(value, index)

def __delitem__(self, index):
    indexes = range(self.__data.size())[index]
    indexes.reverse() # so we can delete High to Low
    for i in indexes:
        self.__data.removeElementAt(i)

def __repr__(self):
    return str(map(None, self.__data))

if __name__ == '__main__':
    L = usrList()
    print "L=", L
    print "L[1:]=", L[1:7]
    print "L[-4:9]=", L[-4:9]
    L[3:6:1] = range(100, 110)
    print L

```

运行 `python seqset1.py` 的输出如下：

```

L= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
L[1:] = [1, 2, 3, 4, 5, 6]
L[-4:9] = [6, 7, 8]
[0, 1, 2, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 6, 7, 8, 9]

```

#### 4. `_delitem_`

表达式 `del L[i]` 将从一个像列表的对象 `L` 里删除一个对象。这显然是指定到可变对象（像 `PyList` 的对象）。

用来删除指定的索引的特殊方法是 `_delitem_ (self, index, value)`。当指定的索引超出范围时，这个方法将产生 `IndexError` 异常，当索引类型不支持时，产生一个 `ValueError` 异常。

程序清单 7-16 定义了一个实现 `_delitem_` 方法的类，这个类在一个内部 `java.util.Vector` 对象里保存了它的列表内容，因此 `_delitem_` 方法必须为每个要删除的索引使用矢量的 `removeElementAt ()`（或 `remove ()`）方法。增加对 `slice` 的支持在前面的例子是常见的，但需要另一个技巧来从 `vector` 里删除项。如果索引从低到高被删除，按照每次删除一项一个一个地减少要被删除的下一个较高的索引。在程序清单 7-16 里倒转需要删除的索引列表来确保索引是按照从最高到最低的顺序。在没有减少将要删除的索引情况下允许删除。

## 程序清单 7-16 实现\_\_delitem\_\_

```
# file: seqdel.py
import java
import types

class userList:
    def __init__(self):
        self.__data = java.util.Vector()

    def append(self, object):
        self.__data.addElement(object)

    def __repr__(self):
        return str(list(self.__data))

    def __delitem__(self, index):
        if isinstance(index, types.SliceType):
            if index.step != 1:
                raise ValueError, "Step size must be 1 for setting list slice"
            delList = range(self.__data.size())[index]
        else:
            delList = [range(self.__data.size())[index]]
        delList.reverse()
        map(self.__data.removeElementAt, delList)

    if __name__ == '__main__':
        S = userList()
        map(S.append, range(5, 20, 2))
        print "Before deletes:", S
        del S[3]
        del S[4:6]
        print "After deletes:", S
```

运行 jython seqdel.py 的输出是：

```
Before deletes: [5, 7, 9, 11, 13, 15, 17, 19]
After deletes: [5, 7, 9, 13, 19]
```

## 5. 序列连接和乘法

表 7-2 包含了像序列的对象将支持的数学运算和与运算相关的特殊方法。

表 7-2 序列数学运算

运 算	描 述	特殊方法
S1 + S2	两个序列的连接	_add_ 和 _radd_
L1 += L2	用增量赋值把 L1 转换成 L1 和 L2 的连接	_iadd_
S * i	重复序列整数 i 次	_mul_ 和 _rmul_
L *= i	用增量赋值把列表 L 转换成 L 的 i 次重复	_imul_

实现连接和循环运算需要实现加和乘法的特殊运算符方法。运算符方法有三种形式：增加有 `_add_`、`_radd_` 和 `_iadd_`，而乘法有 `_mul_`、`_rmul_` 和 `_imul_`。当定义它的对象是在运算符的左边时，这些运算符方法中的第一个（`_add_` 或 `_mul_`）被调用。对表示  $S + X$  或  $S^* X$  的序列 S 正像 `S._add_(X)` 和 `S._mul_(X)` 一样被实现。这些方法的第二个（`_radd_` 和 `_rmul_`）是第一个的

修改形式——当对象在表达式的右边时。仅当左边的对象没有定义`_add_`或`_mul_`时，这个修改的方法才被调用。如果 S 定义`_radd_`和`_rmul_`，但没有 X，于是 X + S 和 X \* S 变成 S.`_radd_(X)`和 S.`_rmul_(X)`。`_radd_`和`_rmul_`方法实现增量赋值，意思是 S+=X 和 X\*=X 像 S.`_iadd_(X)`和 S.`_imul_(X)`一样被实现。

程序清单 7-17 实现了在表 7-2 列出的所有六个方法。这些方法的职责是它们因为不支持的类型而产生异常和仅增量赋值运算修改它本身。

程序清单 7-17 序列连接和循环

```
# file: seqmath.py
import types
import java

class usrList:
    def __init__(self):
        self.__data = java.util.Vector()
        map(self.__data.addElement, range(5,8)) # some default values

    def __add__(self, other):
        if isinstance(other, types.ListType):
            return map(None, self.__data) + other
        else:
            raise TypeError, "__add__ only defined for ListType"

    def __radd__(self, other):
        if isinstance(other, types.ListType):
            return other + map(None, self.__data)
        else:
            raise TypeError, "__radd__ only defined for ListType"

    def __iadd__(self, other):
        # Augmented assignments methods usually modify self, then return self
        if isinstance(other, types.ListType):
            map(self.__data.addElement, other)
            return self #map(None, self.__data) # act like a list type
        else:
            raise TypeError, "__iadd__ only defined for ListType"

    def __mul__(self, other):
        if (isinstance(other, types.IntType) or
            isinstance(other, types.LongType)):
            return map(None, self.__data) * other
        else:
            raise TypeError, "Only integers allowed for multiplier"

    def __rmul__(self, other):
        if (isinstance(other, types.IntType) or
            isinstance(other, types.LongType)):
            return map(None, self.__data) * other
        else:
            raise TypeError, "Only integers allowed for multiplier"

    def __imul__(self, other):
        if (isinstance(other, types.IntType) or
```

```

    isinstance(other, types.LongType)):
        map(self._data.addElement,
             [x for x in self._data] * (other - 1))
        return self
    else:
        raise TypeError, "Only integers allowed for multiplier"

def __repr__(self):
    return str(map(None, self._data))

if __name__ == '__main__':
    L = usrList()
    print "start      :", L
    print "__add__    :", L + [1, 0]
    print "__radd__   :", ["a", "b"] + L
    L += [1, 1]
    print "__iadd__   :", L
    print "__mul__    :", L * 2
    try:
        print "__rmul__   :", 2 * L
    except TypeError:
        print "__rmul__ raised a TypeError"

    L *= 2
    print "__imul__   :", L

```

运行 `python seqmath.py` 的输出是：

```

start      : [5, 6, 7]
__add__    : [5, 6, 7, 1, 0]
__radd__   : ['a', 'b', 5, 6, 7]
__iadd__   : [5, 6, 7, 1, 1]
__mul__    : [5, 6, 7, 1, 1, 5, 6, 7, 1, 1, 1]
__rmul__   : [5, 6, 7, 1, 1, 5, 6, 7, 1, 1, 1]
__imul__   : [5, 6, 7, 1, 1, 5, 6, 7, 1, 1, 1]

```

## 6. Slices

Python 不支持 2.0 版的下列方法。对这些方法的支持仍存在 Jython 代码基区，因此为了完整地在这里包含它们。包含它们并不是鼓励使用它们，而只是万一读者在遗留代码遇到时才被提供。对新代码，使用 `_setitem_`、`_getitem_` 和 `_delitem_`。

- `_getslice_(self, start, stop, step)` —— 返回一个序列，这个序列包含被 slice 参数（`start`、`stop` 和 `step`）指定的 `self` 的那些元素。如果 `L` 包含 `[1, 2, 3, 4, 5]`，`L[1:4:1]` 实际调用（如果定义了）`L._getslice_(1, 4, 1)`，`_getslice_(1, 4, 1)` 应该返回 `[2, 3, 4]`。
- `_setslice_(self, start, stop, step, value)` —— 赋 `self` 的索引给指定的 `value` 或 `values`。如果 `L` 包含 `[1, 2, 3, 4, 5]`，`L[2:4] = ["a", "b", "c"]` 实际调用（如果定义了）`L._setslice_(2, 4, 1, ["a", "b", "c"])`，`L._setslice_` 应该把它的内部列表设计到 `L[0:start] + value + L[stop:]`。
- `_delslice_(self, start, stop, step)` —— 删除由 slice 指定的内部序列值。如果 `L` 包含 `[1, 2, 3, 4, 5]`，`L[1:4]` 实际调用（如果定义了）`L._delslice_(1, 4, 1)`，它删除元素 1、2

和 3 指定的内部值。

### 7. `_contains_`

测试一个对象 `o` 是不是序列 `S` 的成员，通常通过 `S` 循环查找 `o`。如果你的设计需要许多像这样的成员资格 tests，你将面临严厉的、二次的执行困难。然而你可用特殊方法 `_contains_` 有选择来优化这个成员资格 test。如果 `_contains_` 方法被定义，则列表成员资格 tests 调用 `S._contains_(o)`，而不是通过 `S` 循环。`_contains_` 方法有 `self` 参数和成员资格存在问题的项的参数位置。`_contains_` 函数如果它不包含那个对象则返回 0（假），或如果包含则返回 1 或非零（真）。

程序清单 7-18 是一个仿真列表的类，但也把成员设计成内部字典里的键。字典值是对象在列表出现的次数。这允许快速的成员资格 tests 通过检查字典是否有键而不是通过序列循环。折衷办法是增加内存使用和一个较慢的项的 setting 和 deleting。程序清单 7-18 增加了 `_setitem_` 和 `_getitem_` 方法，因为这些运算必须被截听来保持内部字典和列表同步。两个帮助方法 `_incrementMember` 和 `_decrementMember` 被定义来帮助通过确定每个 key 的记数（值）和在需要时删除或创建键处理同步进程。

程序清单 7-18 用 `_contains_` 加速成员资格测试

```
# file: seqin.py
import types

class usrList:
    def __init__(self, initialValues):
        self.__data = initialValues
        self.__membership = {}
        map(self.__membership.update,
            [{key:1} for key in self.__data])

    def __contains__(self, item):
        return self.__membership.has_key(item)

    # for __contains__ to work, assignment and deletion must
    # change self.__data and self.__membership
    def __setitem__(self, index, value):
        if isinstance(index, types.SliceType):
            if index.step != 1:
                raise ValueError, "Assignment to slice requires step=1"
            indexes = self.__data[index]
        else:
            indexes = [self.__data[index]]

        # updated self.__data and self.__membership
        self.__data[index] = value
        map(self.__decrementMember, indexes)
        map(self.__incrementMember, values)

    def __delitem__(self, index):
        indexes = self.__data[index]
        del self.__data[index]
        if isinstance(indexes, types.ListType):
            map(self.__decrementMember, indexes)
        else:
```

```

    self.__decrementMember(indexes) # it's really only one index

def __incrementMember(self, member):
    if self.__membership.has_key(member):
        self.__membership[member] += 1
    else:
        self.__membership[member] = 1

def __decrementMember(self, member):
    if self.__membership.has_key(member):
        if self.__membership[member] == 1:
            del self.__membership[member]
        else:
            self.__membership[member] -= 1

def __repr__(self):
    return str(self.__data)

if __name__ == '__main__':
    from time import time

    t1 = time()
    pyList = range(0, 12000, 3)
    print "The PyList took %f seconds to fill." % (time()-t1,)

    t1 = time()
    newList = usrList(range(0, 12000, 3))
    print "The usrList took %f seconds to fill." % (time()-t1,)

    t1 = time()
    count = 0
    for x in range(10, 12000, 7):
        if x in pyList: count += 1
    print "Found %i items in pyList in %f seconds" % (count, time()-t1)

    t1 = time()
    count = 0
    for x in range(10, 12000, 7):
        if x in newList:
            count += 1
    print "Found %i items in newList in %f seconds" % (count, time()-t1)

```

运行 jython seqin.py 的输出是：

```

The PyList took 0.000000 seconds to fill.
The usrList took 0.110000 seconds to fill.
Found 571 items in pyList in 6.430000 seconds
Found 571 items in newList in 0.220000 seconds

```

程序清单 7-18 有一个详细的测试部分来举例说明项测试困难和成员资格测试有益于继承。

## 8. UserList

Jython (和 Python) 程序库包含为了使像列表的、用户定义的对象的创建容易的模块。前面例子使用了被称做 usrList 的类，该类在没有创建命名冲突的情况下（注意拼写差别）有意预先对这点进行了介绍。UserList 模块定义了一个类：UserList。这个类可任意选择使用一个 PyList

对象在内部表示列表数据，并提供与这个数据一起作用的缺省方法。如果你没有选择使用 PyList 对象作为内部数据，你需要重定义所有需要的方法。

程序清单 7-19 使用 UserList 类来对列表请求项的频率进行统计。程序清单 7-19 的重点是内部数据和定义的方法。程序清单 7-19 里的 ListStats 类选择使用 PyList 作为内部数据和传递那个对象到 UserList 构造函数。不是所有方法需要实现完全像内置列表一样动作因为 UserList 处理不是显式在程序清单 7-19 里的 ListStats 类里定义的每件事。如果它不传递 PyList 到 UserList 类，则许多工作需要完全像列表一样动作来处理。

#### 程序清单 7-19 UserList 和像列表的对象

```
# file: liststats.py
import UserList

class ListStats(UserList.UserList):
    def __init__(self, data=[]):
        self.data = data
        assert type(data)==type([]), "Constructor arg must be a list"
        UserList.UserList(data)
        self.stats = {}
        self.requestCount = 0

    def __getitem__(self, index):
        items = self.data[index]
        if type(items) != type([]):
            # make plain integers into a list for convenience
            items = [items]
        for x in items:
            self.requestCount += 1
            self.stats[x] = self.stats.setdefault(x, 0) + 1
        return items

    def printStats(self):
        for x in self.data:
            use = self.stats.setdefault(x, 0)
            if not use: continue
            print ("%0.1f, %t.3f% " %
                  (self.data[x],
                   float(use)/float(self.requestCount)*100)),
        print # to put prompt on a new line

if __name__ == '__main__':
    import random

    L = ListStats(range(10))
    for x in range(2000):
        L[random.randint(0, 9)] # access a random index
    L.printStats()
```

运行 `python liststats.py` 的输出是：

```
0, 9.100% 1, 9.600% 2, 11.000% 3, 10.900% 4, 11.350% 5, 8.750% 6,
10.850% 7, 9.500% 8, 9.550% 9, 9.400%
```

### 7.8.2 仿真映射

仿真映射类型是极其相似于仿真列表类型的，除非你用键而不用索引。内置映射对象实现方法 clear、copy、get、has\_key、items、keys、setdefault、update 和 values，因此真正地仿真映射类型需要实现这些方法。映射对象应该实现的特殊方法是\_len\_、\_getitem\_、\_setitem\_ 和 \_delitem\_。这些从列表部分看起来很常见。

- \_len\_(self) ——在映射对象里返回键：值对的数目。
- \_getitem\_(self, key) ——返回与特殊键相关的值。
- \_setitem\_(self, key, value) ——把指定的键设计到指定的在内部映射表示的值。
- \_delitem\_(self, key) ——从内部映射删除指定的键。

因为这些特殊方法的实现从仿真列表是非常常见的，下面列出的代码是足够来论证这些特殊映射方法。程序清单 7-20 从程序清单 7-14 借用概念，它也像文件一样实现数据元素，但增加了一些手法。dictionary 表示目录，keys 表示文件，values 表示文件内容。

程序清单 7-20 也考虑通过 Jython 的 pickle 模块存储数字和数据对象。Pickling 是 Jython 的序列化机制的一种。应用的两个 pickle 方法是把对象转换成字符串的 dumps() 和把字符串转换成对象的 loads()。为了序列化程序清单 7-20 里的列表，我们使用下列语句：

```
string = pickle.dumps(list)
```

为了恢复这个列表，我们使用：

```
pickle.loads(string)
```

程序清单 7-20 一个持久字典

---

```
# file: specialmap.py
import types
import os
import pickle
from stat import ST_SIZE

class mappingDirectory:
    def __init__(self, directory):
        self.__ID = None
        self.__dir = directory
        if not os.path.exists(directory):
            os.mkdir(directory)
            idfile = os.path.join(directory, "JythonIDfile")
            f = open(idfile, 'wb')
            print >> f, str(id(self))
            f.close()
        elif not os.path.isdir(directory):
            raise ValueError, 'File %s already exists.' % directory
        elif not os.path.isfile(os.path.join(directory,
                                             "JythonIDfile")):
            msg = "Directory exists, but it isn't a mapping directory."
            raise ValueError, msg

    def __repr__(self):
```

```

listing = os.listdir(self._dir)
results = {}
for x in listing:
    if x == "JythonIDfile": continue
    size = os.stat(os.path.join(self._dir, x))[ST_SIZE]
    results[x] = "<datafile: size=%i>" % size
return str(results)

def __setitem__(self, key, value):
    self._testKey(key)
    pathandname = os.path.join(self._dir, key)
    f = open(pathandname, "w+b")
    print >> f, pickle.dumps(value)
    f.close()

def __getitem__(self, key):
    self._testKey(key)
    pathandname = os.path.join(self._dir, key)
    try:
        f = open(pathandname, "rb")
    except IOError:
        raise KeyError, key
    value = f.read()
    f.close()
    return pickle.loads(value)

def __delitem__(self, key):
    self._testKey(key)
    pathandname = os.path.join(self._dir, key)
    if not os.path.isfile(pathandname):
        raise KeyError, key
    os.remove(pathandname)

def _testKey(self, key):
    if not isinstance(key, types.StringType):
        raise KeyError, "This mapping restricts keys to strings"
    if key == "JythonIDfile":
        raise KeyError, "The name JythonIDfile is reserved."
if __name__ == '__main__':
    md = mappingDirectory("c:\\windows\\desktop\\jythontestdir")
    md["odd"] = filter(lambda x: x%2, range(10000))
    md["even"] = filter(lambda x: not x%2, range(10000))
    md["prime"] = [2, 3, 5, 7, 11, 13, 17]
    print "Mapping =", md
    print "primes =", md["prime"]
    del md["prime"]
    print "primes deleted"
    print "Mapping =", md

```

在程序清单 7-20 里有另一个用作安全目的技巧。JythonIDfile 被增加到为这种映射创建的目录。没有任何检查来保证这个类创建某一个目录或在它里面的任何文件，因此它必须确定目录有一定的特殊性（以免某些人用/etc 或 c:\\windows \\ system 试这个例子）。

- `os.mkdir (directory)` —— 创建一个目录。如果它不能创建那个目录，执行文件的 OS 和目录权限可能产生一个 OSError。

- `os.path.join(path, filename)` ——连接路径和文件名，同时在适当的地方增加特定平台的路径分隔符。
- `os.path.isdir(directory)` ——如果指定的目录存在，则返回 1，否则返回 0。
- `os.path.isfile(filename)` ——如果指定的文件存在，则返回 1，否则返回 0。
- `os.listdir(directory)` ——返回在指定目录定义的所有名字的列表。
- `os.stat(file)` ——返回一个表示文件统计的九个元素元组。程序清单 7-20 使用惟一的文件 `size`，它是用 `ST_SIZE` 从 `stat` 模块指定的。
- `pickle.dumps(object)` ——返回一个字符串，它是序列化对象。
- `pickle.loads(string)` ——返回一个对象，该对象字符串是序列化的数据。

运行 `jython specialmap.py` 的输出是：

```
Mapping = {'prime': '<datafile: size=38>', 'odd': '<datafile: size=34452>',  
'even': '<datafile: size=34452>'}  
primes = [2, 3, 5, 7, 11, 13, 17]  
primes_deleted  
Mapping = {'odd': '<datafile: size=34452>', 'even': '<datafile: size=34452>'}
```

### 7.8.3 仿真数字类型

仿真数字类型需要对每个对象支持的数字运算定义特殊方法。与数字运算相关的特殊方法是那些实现一元和二元运算符、到其他类型的转换和强制的方法。特殊方法主要是为二进制运算符，这些方法也出现在三元运算符里。例如，实现 `addition` 包括当对象在 `addition` 运算符的左边时定义 `_add_` 方法和当对象在运算符的右边时定义 `_radd_` 方法，以及当使用增量赋值（`+=`）时定义 `_iadd_` 方法。这些方法有时候分别调用标准的、修改的和增量的方法。

增量赋值方法（`_i"`）是惟一的，因为它们的实现不返回值，但可修改 `self`。然而，如果一个数字对象不定义增量方法，它仍然可以在增量赋值里使用。如果对象 `N` 定义了 `_add_` 方法，但没有定义 `_iadd_`，表达式 `N+=N` 执行：

```
N = N.__add__(N)
```

表 7-3 列出了数字运算符和它们相关的方法。在表 7-3 的左边用 `N` 表示用户定义的数字对象的运算。表 7-3 的右边是相关特殊方法的方法特征符。如果一个方法应返回对象的一个指定类型，则返回类型用 `--> type` 标注。例如，运算 `N+2` 转换成 `N._add_(2)`，方法特征符是 `_add_(self, other)`。

表 7-3 数字二进制运算符和它们的特殊方法

运算符	方 法
<code>N+2</code>	<code>_add_(self, other)</code>
<code>2+N</code>	<code>_radd_(self, other)</code>
<code>N+=2</code>	<code>_iadd_(self, other) --&gt; self</code>
<code>N-2</code>	<code>_sub_(self, other)</code>
<code>2-N</code>	<code>_rsub_(self, other)</code>
<code>N-=2</code>	<code>_isub_(self, other) --&gt; self</code>

(续)

运算符	方法
N * 2	_mul_(self, other)
2 * N	_rmul_(self, other)
N *= 2	_imul_(self, other) --> self
N/2	_div_(self, other)
2/N	_rdiv_(self, other)
N/= 2	_idiv_(self, other) --> self
N%2	_mod_(self, other)
2%N	_rmod_(self, other)
N% = 2	_imod_(self, other) --> self
divmod (N, 2)	_divmod_(self, other)
divmod (2, N)	_rdivmod_(self, other)
N ** 2	_pow_(self, other)
2 ** N	_rpow_(self, other)
N ** 2 = 2	_ipow_(self, other) --> self
pow (N, 2, 2)	_pow_(self, other, mod=1)
pow (2, N, 2)	_rpow_(self, other, mod=1)
N << 2	_lshift_(self, other)
2 << N	_rlshift_(self, other)
N << = 2	_ilshift_(self, other) --> self
N&2	_and_(self, other)
2&N	_rand_(self, other)
N& = 2	_iand_(self, other) --> self
N 2	_or_(self, other)
2 N	_ror_(self, other)
N  = 2	_ior_(self, other) --> self
N^2	_xor_(self, other)
2^N	_rxor_(self, other)
N^ = 2	_ixor_(self, other) --> self
- N	_neg_(self, other)
+ N	_pos_(self, other)
- N	_invert_(self, other)
abs (N)	_abs_(self)
coerce (N, x)	_coerce_(self, other) --> some common type or None
complex (N)	_complex_(self) --> PyComplex
float (N)	_float_(self) --> PyFloat
hex (N)	_hex_(self) --> PyString
int (N)	_int_(self) --> PyInteger

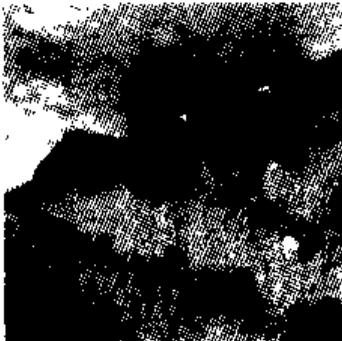
(续)

运算符	方 法
long (N)	_long_ (self) --> PyLong
oct (N)	_oct_ (self) --> PyString

在数字运算里发挥重要作用的是\_coerce\_。只要两个操作数具有不同的类型\_coerce\_方法被调用。运算 N1 + N2，而 N1 和 N2 具有不同的类型，实际调用 N1.\_coerce\_(N2)。\_coerce\_方法返回转换成通常类型的 N1 和 N2 的一个元组——让我们称它们为 T1 和 T2。于是 T1.\_add\_(T2) 被调用。如果左边的操作数没有\_coerce\_方法，则右边的操作数的\_coerce\_方法被调用。



## 第二部分 Jython 内核和用 Java 集成 Jython



### 第 8 章 用 jythonc 编译 Jython

Jython 优于其他语言是因为它广泛集成 Python 和 Java。大多数多级语言结合在语言之间有语义上的不兼容，这种不兼容阻碍了真正地广泛集成。例如，如果不采取特殊的措施和写 C 扩展名不遵循 Cpython 独特的特定的准则，CPython 就不能使用 C 库函数。另一方面，在不需要额外的措施、修改或在写 Java 类不需要特殊的处理的情况下，Jython 在使 Python 环境下任意 Java 类能无缝使用。这仅是创造的一个方面，然而，要真正广泛集成，Jython 也必须允许 Java 无缝使用 Python。由于 jythonc，即使不是非常明显 Jython 也满足了后者的需要。

jythonc 可以用来把 Jython 模块编译到 Java 的后缀名为 \*.class 的文件中去。jythonc 可以创建可运行的类。Java 能运行这些类，但要依赖 Jython 模块。jythonc 也能创建完全自包含和可运行的应用程序，这意味着它拥有所有能被定位和编译的模块，同时 Java 也能运行它。另外，jythonc 能编译 Jython 类。这些类与 Java 类几乎毫无差别，Java 代码也能输入和调用它。

为了激发你的兴趣，请多考虑 jythonc 的以下优点：能将 Jython 应用程序编译成在大多数 JVM 上可以运行的自包含的 jar 文件；Jython 里原型程序的发展利益以及在需要时能转化一部分成 Java；在 Jython 里写 servlets、applets 和 beans 这些程序非常轻松，且能让它们直接运行在 Java 的工作框中，而无须考虑它们的源语句；可替换地生成、派生、扩展 Java 和 Jython 类。Jythonc 在功能性支持上的能力是非常杰出的。

#### 8.1 jythonc 是什么

jythonc 是从 Jython 模板里生成 Java 代码的工具。jythonc 也能创建 jar 文件、跟踪从属性、冻结相关的模板和依赖所提供的选项。在同一目录里作为 Jythonc shell script（或 batch 文件）应是另一个称作 jythonc（或 jythonc.bat）的 script。该 script 真正是一个包裹文件 sys.prefix/Tools/jythonc/jythonc.py 的封装器，该文件创建一个 java 文件和调用一个 java 编译器来创建 \*.class 文件。这就意味着你必须有一个合适的 Java 编译器用来与 jythonc 连接，例如 Sun 的 javac，可在网页上 <http://www.java.com/j2se/> 或 IBM 的 jikes 得到，可从网页上得到 <http://www10.software.ibm.com/developerworksopensource/jikes/>。Java Runtime Environment (JRE) 因为不包含编译

器所以是不够的。你将需要 Java 开发工具包 (Java Development Kit, JDK) 或 Jikes 编译器。注意 Microsoft 的 jvc 编译器对编译 Jython 代码来说并不是目前一个好的选择。

编译的 Jython 对解释的 Python 并没有任何性能优势。对于那些对 Jython 不太熟悉的人来说，它应值得关注。目前的执行产生编译的 Jython 代码，该代码相似于使用 Jython 本身来执行模块一样执行。

## 8.2 用 jythonc 编译模块

如果你运行 `python A.py`，它意味着模块 A 作为顶层或 `_main_` 运行 script。如果你用 `jythonc` 来编译 `A.py`，你可改为直接使用 Java 来执行应用程序。程序清单 8-1 是一个 Jython 模块调用 `guess.py` 的例子。它将作为测试主题用来编译 Jython 模块到可运行的 Java 类文件，下面假定编译器是 `javac`，该例子是在你的路径。如果它不在你的路径下或 `jythonc` 有困难找到 Java 编译器，你可用 `jythonc` 的 `-c` 转换来指定编译器。在本章看后面的 `jythonc` 选项部分可得到更多的关于 `-c` 转换开关。另外，当 `jython.jar` 已经在 `classpath` 时，因为一些 `jythonc` 版本已失效你可能希望在运行 `jythonc` 前从你的 `classpath` 环境变量去掉 `jython.jar`

**程序清单 8-1 编译到可运行类的 Jython Guessing 模块**

```
#file: guess.py
import random

print "I'm thinking of a number between 1 and 100. Guess what it is."
num = random.randint(1, 100)

while 1:
    guess = int(raw_input("Enter guess: "))
    if guess==num:
        print "Correct, the number was %i" % num
        break
    elif abs(guess - num) < 3:
        print "Close enough. The number was %i" % num
        break
    elif guess > num:
        print "Too high."
    else: print "Too low"
```

不需要编译的程序清单 8-1 的 shell 命令是：

```
> jythonc guess.py
```

上述命令假定 `jythonc` 是在你的路径里。如果不是，你必须给出到 `jythonc` 的全路径，如下：

```
C:\jython-2.1\jythonc guess.py
```

运行 `jythonc` 命令后，你将看到这样相似的输出：

```
processing guess
Required packages:
Creating adapters:
Creating .java files:
guess module
```

```
Compiling .java to .class...
Compiling with args: ['javac', '-classpath',
'"C:\\jython\\jython.jar;..\\jpywork;C:\\jython\\Tools\\jythonc;C:\\ch8
examples\\.;C:\\jython\\Lib;C:\\jython"', '..\\jpywork\\guess.java']
0
```

依赖于你正在使用的 JDK 版本你也可能得到以下的警告：

```
Note: Some input files use or override a deprecated API.
Note: Recompile with -deprecation for details.
```

检查输出，我们看见从 guess.py 模块 jythonc 创建了一个 guess.java 文件，并调用 javac 来编译 guess.java 文件到类文件。用来编译 guess.java 文件到类文件的全部命令作为一个列表（用 join(args) 来得到实际的命令）。因为 javac 是 args 列表的第一项，所以 jythonc 调用 javac 来进行编译。注意生成的 Java 文件可能有对你正在使用的 JVM 版本特殊的代码是重要的。这就意味着当生成的代码可能含有引用不支持的类时使用-target 选项可能是不够的。对特定的 JVM 版本，为了编译 Jython 代码，当运行 jythonc 时你应使用那个 JVM 版本。

如果 javac 不是你使用的 Java 编译器，你可在注册里或命令行上指定不同的编译器。Jython 注册键 python.jythonc.compiler 和-c 命令行开关两个都可指定使用哪个 Java 编译器。用 jikes 来代替 javac 的合适的注册行是：

```
python.jythonc.compiler = /path/to/jikes # for *nix users
python.jythonc.compiler = c:\path\to\jikes # for win users
```

用-c 选项来指定 jikes 编译器是：

```
bash> jythonc -C /path/to/jikes guess.py
dos> jythonc -C c:\path\to\jikes guess.py
```

jythonc guess.py 的结果是一个称作 jpywork 的目录，该目录包含了文件 guess.java、guess.class 和 guess\$PyInner.class。guess.java 文件是 jythonc 生成的，并且两个类文件都是编译 guess.java 的结果。编译 Jython 代码至少将产生两个类文件：具有模块名的类文件和相关的\$PyInner 类。

用可执行的 JVM 运行经 jythonc 编译的类需要 jython.jar 文件和两个经过编译的类文件 (guess.class 和 guess\$PyInner.class) 都在 classpath 里。如果你与从编译程序清单 8-1 生成的两个类文件在同一个目录里，并且你从 Sun 的 JDK 使用可执行的 Java，这将是运行经过编译的 guess 文件的命令：

```
bash> java -cp /path/to/jython.jar:. guess
dos> java -cp \path\jython.jar;. guess
```

你应该给 guess 提示一个数字。运行这个程序的例子的输出如下：

```
I'm thinking of a number between 1 and 100. Guess what it is.
Enter guess: 50
```

```
Too high.
Enter guess: 25
Close enough. The number was 23
```

当首先运行经 jythonc 编译的文件时，忘记在 classpath 里包含合适的目录是一个通常的错误。因为经 jythonc 编译的文件常常被用在存在的 Java 框架里，所以编译的类文件常常被放在那个框架指定的目录中。这一步容易产生错误，例如没有复制所有需要的文件。注意 jythonc 至少创建两个文件。如果在 classpath 里没有内部文件 (guess\$PyInner.class)，你会得到以下的错误信息：

```
Error running main. Can't find: guess$PyInner
```

另外，经 jythonc 编译的类需要在 jython.jar 文件里可找到的类。这些可能包含在使用特定的 jythonc 选项的存档文件里，将在后面看到。现在我们必须保证 jython.jar 在 classpath 里。如果 jython.jar 不在 classpath 里，你将得到以下的错误信息：

```
Exception in thread "main" java.lang.NoClassDefFoundError:
org/python/core/PyObject
```

程序清单 8-1 的编译版本依赖 sys.prefix/Lib 目录里的任一模块。如果 sys.prefix 或 sys.path 变量是错误的，你将得到以下的错误：

```
Java Traceback:
...
Traceback (innermost last):
  File "C:\WINDOWS\Desktop\ch8examples\guess.py", line 0, in main
    ImportError: no module named random
```

## 8.3 路径和经过编译的 Jython

Jython 依赖称作 python.home 的 Java 特性来查找和读取 Jython 注册文件，也建立 sys.prefix 和这样某一个 sys.path 入口。Jython 也在 python.home 或 sys.prefix 目录里创建一个高速缓存目录，该目录被缺省命名为 cachedir。关于 Java 包信息被存储在这个目录里作为一个重要的优化，但是如果 python.home 或 sys.prefix 有些意外的值这可能有些麻烦。python.home 属性作为 jython 启动 script 的一部分来设置，但经过编译的 Jython 并没有这种机会。

这部分介绍了关于 Jython 的路径问题。Jython 模块编译到 Java 类需要特殊的步骤来保证它能找到 Jython 模块和它依赖的包以及使用特定的 cachedir。接下来的是一个你能从其中选择的选项列表。

### 8.3.1 在 JVM 里设置 python.home 属性

可执行的 Sun 的 Java 允许用-D 命令行选项来进行属性设置。程序清单 8-2 是一个设置用来揭示路径问题的 Jython 模块。

**程序清单 8-2 描述路径和属性的 Jython 模块**

```
# file: paths.py
import sys
import java

print "sys.path=", sys.path
print "sys.prefix=", sys.prefix
print "python.home=", java.lang.System.getProperty("python.home")

# print a random number just so this depends on a
# module in the sys.prefix/Lib directory
import random
print 'My lucky number is', random.randint(1,100)
```

用 jythonc paths.py 编译程序清单 8-2。这样编译它创建了用 Jython 行语称作可运行的类文件。单词可运行意指经过编译的 Jython 代码以便可执行的 Java 能执行它，但它仍然依赖 Jython 库和遵循标准 Jython 解释程序行为。执行一个可运行的类就像运行一个 Jython script——意味着它将缓存启动时的 Java 包信息和使用 sys.path 来查找模块。这就意味着需要设置 python.home 属性或至少指定库路径。

为了继续测试程序清单 8-2，用命令 shell 改变目录到 paths.class 和 paths\$PyInher.class 文件存在的地方。该命令看起来与下面仅使用对你的安装来说是独特的目录相似（因为命令很长，所以该命令“wraps”，但是它还是作为一个命令）。

```
bash> java -Dpython.home="/usr/local/jython-2.1" -cp /usr/local/jython.jar:. paths
```

```
dos> java -Dpython.home="c:\jython-2.1" -cp c:\jython-2.1\jython.jar;. paths
```

结果如下：

```
sys.path= ['.','.','c:\\jython 2.1\\Lib']
sys.prefix= c:\\jython 2.1
python.home= c:\\jython 2.1
My lucky number is 96
```

这意味着 Jython 的注册文件被找到，sys.prefix 值是精确的，模块能被找到和定位，例如在 sys.prefix/Lib 目录里的 random。

如果没有-D 选项会发生什么呢？没有它，该命令如下所示：

```
bash> java -cp /path/to/jython.jar:. paths
dos> java -cp c:\path\to\jython.jar;. paths
```

如果 jython.jar 文件还在 Jython 安装目录里，其结果将与下面结果相似（用你的特定平台的目录分隔符来代替/）：

```
sys.path= ['.','.','/usr/local/jython-2.1/Lib']
sys.prefix= /usr/local/jython 2.1a1/
python.home= None
My lucky number is 97
```

在没有合适的 `python.home` 值的情况下，Jython 如何知道正确的 `sys.prefix` 的值呢？这不是 Jython 的文档行为的一部分，但是，正确地说，`jython.jar` 不是正确的目录，`sys.prefix` 的值变成 `jython.jar` 文件所在的目录。如果你拷贝 `jython.jar` 文件到与 `paths.class` 文件相同的目录，将出现不同的结果：

```
dos> java -cp jython.jar;. paths
sys.path= ['.','\\Lib']
sys.prefix=.
python.home= None
Java Traceback:

    at org.python.core.Py.ImportError(Py.java:180)
...
    at paths.main(paths.java:72)
Traceback (innermost last):
  File "C:\WINDOWS\Desktop\ch8examples\paths.py", line 0, in main
ImportError: no module named random
```

最后一个例子的结果是一个等于 `None` 的 `python.home`。这意味着 Jython 注册文件没有被找到，在 Jython 的 Lib 目录库里的那个模块（random）也没找到。这可通过把 random 模块也移到同一目录来解决，因为 `sys.path` 将自动包含在当前目录，或其他方法。

### 8.3.2 显式地把目录加到模块里的 `sys.path`

有时候显式追加库路径到你打算编译的模块里的 `sys.path` 变量可能是最容易的。这确保某一个目录里的模块是可访问的。如果你想程序清单 8-2 能找到 `/usr/local/jython-2.1/Lib` 目录里的 random 模块，你可以把下列行加到 `paths.py`（当然在引入 `sys` 后）：

```
sys.path.append('/usr/local/jython-2.1/Lib') # *nix
sys.path.append('c:\\jython-2.1\\Lib') # dos
```

由于这是经过编译的类的一部分，不管 `python.home` 属性和 `sys.prefix` 变量指向哪里，在那个目录里的模块将总可以被找到。然而，这缺乏 Java 的 `classpath` 的灵活性。如果一个应用程序被移到另一个机器上，路径可能变得有些麻烦和跨平台功能受到影响。

### 8.3.3 增加 `python.path` 或 `python.prepath` 属性

正像你能用 Java 的 `-D` 开关设置 `python.home` 属性一样，你也能用 `-D` 开关设置注册的 `python.path` 或 `python.prepath` 属性。假定你已经用 `jythonc paths.py` 编译了程序清单 8-2，而且你已经把 `jython.jar` 文件移到了与 `paths.class` 和 `path$PyInner.class` 文件同--目录里，那么你能用与下面相似的命令运行应用程序：

```
dos>java -Dpython.path="c:\\jython-2.1\\Lib" -cp jython.jar;. paths
```

在这个例子里，`sys.path` 列表追加 `python.path` 值作为它的初始化的一部分，因此是在装载模块之前。我们的例子应用程序将适当地查找和装载 random 模块。在命令行设置路径属性允许批文件或 script 来动态决定这个值——一个在代码里放置静态路径的改进。

### 8.3.4 冻结应用程序

冻结一个应用程序意味着用一种方式编译 Jython 模块，该方式在编译时跟踪所有依赖性并为所有需要的模块产生类文件。一个冻结的程序是自包含的：它不依赖 Jython 的 sys.path 目录，它查找 Java 的 classpath 模块，并且它在启动时不缓冲包信息。使用以下命令来冻结程序清单 8-2：

```
dos> jythonc -d paths.py
dos> jythonc --deep paths.py
```

编译过程现在将包含 random 模块。Jythonc tracks 依赖象 random 这样的模块，并且同时编译这些模块到 Java。冻结路径应用程序（从程序清单 8-2）的结果应是四个类文件：

- paths.class
- paths\$PyInner.class
- random.class
- random\$PyInner.class

这些类文件加上在 jython.jar 里找到的类是所有需要运行路径应用程序。如果你在一个命令提示符，并且与从编译程序清单 8-2 创建的类文件在同一目录，则你能用以下命令执行冻结的路径程序：

```
bash> java -cp /usr/local/jython-2.1/jython.jar:. paths
dos> java -cp c:\jython-2.1\jython.jar;. paths
```

注意在 jython.jar 里的类仍然需要 classpath，并且必须出现在 classpath 里。

它不会分别地先编译 paths.py，然后编译 random.py。jythonc 在编译时必须知道是依赖 Java 类还是 Jython 模块。这个类/模块两分法提出了另一个重要的要点。从用 jythonc 编译一个模块创建的类文件不能用在 Jython 里，好象它们是模块似的。Jythonc 已使它们成为 Java 类，要将它们回复成模块是复杂的。

### 8.3.5 写一个定制的 import() 钩子

写自己的 import 工具允许更大的控制，但也有一些约束。有了定制的 import，你能控制模块装载的位置。这允许你使用 zip 文件、jar 文件、classpath 资源、远程 URL 资源和更多的作为适合模块的源代码。然而这种限制在处理预先编译的模块时是当前的难点。Python 模块 (\*.py 文件) 和 Py 类 (被编译成 \$py.class 文件) 的二义性使得在 Jython 里装载预先编译的模块很棘手。不冻结所有模块依赖而装载一个经过编译的 Jython 模块 (\$py.class) 的能力是人们期望的复制贫乏的人为冻结的特征，但它现在还没有实现。

程序清单 8-3 是一个定制模块从 classpath 装载那种模块 (\*.py 文件) 的简单的实现。这允许访问在 jar 文件里打包的 \*.py 文件——也常常是发送应用程序所期望的事情。在程序清单 8-3，loadmodule 是定制的 import 函数，myTest.py 模块是简单的载有 loadmodule 的伴随模块。

Loadmodule 函数的重要方面如下：

- classloader。

- 读模块。
- 用 new 模块创建一个模块实例。
- exec 函数。

当检查 sys 模块的内容时，你将看到与类装载器相关的三个函数：classLoader、getClassLoader 和 setClassLoader。当前 sys.classLoader 仅仅设置在经过 jythonc 编译的类里。程序清单 8-3 包含了适合于与没经过编译的代码（在列表里被注释出来了）运行的额外的两行。

在获得类装载器后，loadmodule 函数开始装载期望的模块。读模块的步骤如下：

- 1) 用 jarray 模块来创建一个 byte[] 缓冲器用做 stream 的 read() 方法的自变量。
- 2) 从该缓冲器读。
- 3) 追加数据到模块字符串直到 read() 方法报告文件结束 (-1)。

New 模块能够动态地创建类、函数、方法、实例和模块；loadmodule 函数利用它的动态的模块 creation 来创建一个模块对象。这个模块应该用保存在 PyStringMap sys.modules 里的所有装载了的模块的列表注册，但在当前的命名空间里需要一个引用，因此 sys.modules[name] 和 mod 两者都被分配 new 模块实例。

设置模块剩下来的工作是执行它，但它必须在被 mod 创建（否则它在顶层环境或 main\_ 里执行）的 new 模块实例的命名空间里执行

### 程序清单 8-3 定制的模块 Import

```
# file: customimp.py
import sys
import new
import jarray
import java

def loadmodule(name, bufferSize=1024):
    filename = name + ".py"
    cl = sys.getClassLoader()
    if cl == None:
        cl = java.lang.ClassLoader.getSystemClassLoader()
    r = cl.getResourceAsStream(filename)
    if not r:
        raise ImportError, "No module named %s" % name
    moduleString = ""
    buf = jarray.zeros(bufferSize, "b")
    while 1:
        readsize = r.read(buf)
        if readsize == -1: break
        moduleString += str(java.lang.String(buf[0:readsize]))
    sys.modules[name] = mod = new.module(name)
    exec(moduleString) in mod.__dict__
    return mod

myTest = loadmodule("myTest")
print myTest.test()
```

用来测试定制装载的简单 myTest 模块如下：

```
# file: myTest.py
def test():
    return "It worked"
```

程序清单 8-3 是一个很好的折衷，因为即使你冻结了应用程序，你仍然具有装载了 loadmodule 的 \*.py 文件的灵活性。没有重新编译任何东西，任何需要的 \*.py 文件能被更新、增加或从资源中删除。

为了冻结程序清单 8-3 和在一个单个的 jar 文件里包含所有需要的 Jython 类，可用以下的 jythonc 命令：

```
jythonc -all --jar myapp.jar customimp.py
```

现在你可用以下命令把 myTest.py 文件加到 myapp.jar 里：

```
jar -uf myapp.jar myTest.py
```

myapp.py 文件包含了所有你需要的来运行应用程序，但你也可在没有重新编译任何东西的情况下容易地更新 myTest.py 文件。以下命令运行 customimp 模块：

```
>java -cp myapp.jar customimp
It worked
```

## 8.4 Jythonc 选项

Jythonc 有许多选项，这些选项允许经过编译的文件、从属性、编译器选项和更多的特殊处理。Jythonc 的语法需要首先指定选项，接着模块被编译：

```
jythonc options modules
```

这些模块能够是到 python 模块的完全路径，如下所示：

```
jythonc /usr/local/jython2.1/Lib/ftplib.py
```

或者在 sys.path 里的模块仅通过模块的名字可以被列出来就像它用在 import 语句里一样：

```
jythonc ftplib
```

表 8-1 给出了每一个 jythonc 的选项和它所起的作用。注意每一个选项有一个可交换使用的长的和短的形式。一些 jythonc 选项也可用作注册键。相等的注册键在可用的地方被加了注释。

表 8-1 jythonc 选项

选 项	描 述
--deep -d	--deep 选项定位和编译所有应用程序需要的从属性。它创建的从属性称作“frozen”或自包含的应用程序。在没有访问 Jython Lib 目录下，一个 frozen 应用程序能够运行，在启动时不缓冲 Java 包信息
--jar jarfile -j jarfile	--jar 选项自动应用--deep 选项，并且它把编译的结果放到指定的 jar 文件。jar 文件名字必须用--jar 选项出现。这并不修改存在的 jar 文件，但如果一个已指定的 jar 文件已经存在，则重写

(续)

选 项	描 述
--core -c	--core 选项自动应用--deep 选项并把核心的 Java 加到用--jar 选项指定的 jar 文件。--core 选项被设置与--jar 选项联合使用。如果没有任何--jar 选项出现，则--core 选项被忽略
--all -a	--all 选项自动应用--deep 选项并把 org.python.core、org.python.parser 和 org.python.compiler 包的类加到用--jar 选项指定的 jar 文件。作为结果的 jar 文件包含所有需要的文件来运行 Jython 应用程序。--all 选项被设置与--jar 选项联合使用。如果没有任何--jar 选项出现，则--all 选项被忽略
--addpackages packages -A	--addpackages 选项把指定的 Java 包加到用--jar 选项指定的 jar 文件。当前的实现增加在包层次（目录）的类文件，但不增加 jar 文件的内容
--workingdir directory -w directory	jythonc 用称作 jpywork 的缺省目录来创建 java 文件和编译到类。--workingdir 选项允许你为这个指定另一个存在的目录
--skip modules -s modules	--skip 选项需要模块名字的逗号分隔的列表。列表里的所有模块名从编译被执行
--compiler compilerName -C compilerName	--compiler 选项允许你指定使用哪个 Java 编译器。如果希望的编译器在路径里，则它的名字就足够了；否则需要路径和名字。在创建 .java 类后你可使用 NONE 的--compiler 选项来让 jythonc 停止。Jython 的注册文件也包含了可用来设置缺省编译器的键 python.jythonc.compiler
--compileropts options -J	--compileropts 选项允许你指定被传递到使用的 Java 编译器的选项。--compileropts -O 选项将传递最佳 (-O) 选项到 javac (如果那是使用的编译器)
--package packageName -p packageName	--package 选项把经过编译的代码放到指定的 Java 包里。J 是访问经过编译的代码需要包名来限制修饰类。编译有-p test 的模块 A 意味着将来对 A 的引用需要 test.A。--package 选项在联合--deep 选项是最有用的
--bean jarfile -b jarfile	--bean 选项创建一个包含适当的 bean 声明文件的 jar 文件。如果正在被编译的 Jython 类跟有 bean 约定，而你又希望使用在 bean box 里创建的 jar 文件，这是有用的
--falsenames names -f names	--falsenames 选项设置名字的逗号分隔列表为假 (false)
--help -h	--help 选项把使用信息打印到屏幕

表 8-2 给出了 jythonc 的例子用法，并加上指定选项怎样影响编译的解释。

表 8-2 jythonc 的用法

命 令	结 果
jythonc test.py	编译 test.py 到可运行的类文件，该类文件仍将潜在地依赖 Jython 模块
jythonc -d -A utest	编译 test.py 到一个已冻结的应用程序。-d (deep) 选项跟踪所有依赖。-A utest 选项把 utest 包的 java 类加到作为结果的 jar 文件。-j A.jar 选项告诉 jythonc 把所有作为结果的文件放进文件 A.jar
-j A.jar test.py	
jythonc -p unit -j test.jar *.py	把在当前目录里的所有模块编译成 Java 类文件。因为-p 选项所有类文件将是 java 包 unit 的一部分，并因为-j 选项将被放进 test.jar 文件
jythonc -b myBean.jar namebean.py	编译 namebean.py 模块和把作为结果的类文件放进 myBean.jar 文件。这也把合适的 bean 声明加到 myBean.jar 文件
jythonc --core -j test.jar test.py	把 test.py 编译成已冻结的应用程序。因为--core 选项含有--deep 选项，所以所有依赖被跟踪和编译。另外 Jython 的核心 Java 类也被增加到 jar 文件

## 8.5 与 Java 兼容的类

如果一个经过编译的 Jython 类与 Java 是兼容的，它能像本机的 Java 类一样运行。为了使 Jython 类与 Java 兼容，你必须考虑以下三个准则：

- 这个 Jython 类必须派生一个 Java 类或接口。
- 这个 Jython 类可以包含称作 sig-strings 的 Java 方法特征提示。
- 这个 Jython 类必须是在同一名字的模块里。与 Java 兼容的类 A 应在文件 A.py 里。

第一个准则是相当简单的：派生一个 Java 类或接口。如果你不需要特定 Java 类的功能，则派生 java.lang.Object。你仍然可以从一个或多个 Jython 类继承——jythonc 并不改变 Jython 的继承规则，对那些用在 Java 里的 Jython 类来说它仅是一个额外的需要。这并不应用于可运行的或冻结的应用程序，仅当你需要一个经过编译的 Jython 类来冒充成一个 Java 类时，应用于可运行的或冻结的应用程序。

第二个准则是@sig 字符串的使用。@sig 字符串真正地是一个具有特定格式的 doc 字符串。格式是跟有有效的 Java 方法特征符的文字 @sig。假设你有以下 Jython 方法：

```
def greet(self, name):
    return "Hello " + name
```

增加了@sig 字符串的相同方法将是这样：

```
def greet(self, name):
    "@sig public String greet(String name)"
    return "Hello " + name
```

在这个例子里 @sig 指定一个具有 String 对象的 public 方法作为一个返回值和一个 String 对象作为参数 name。Java 的这个方法现在就像它是具有用 @sig 字符串指定的特征符的 Java 代码。

并不是所有的方法都需要 @sig 字符串。如果你重定义出现在 Java 超类（见准则 #1）的一个方法，你不需要指定方法特征符。Jythonc 足以巧妙地来获得 Java 超类的这个信息。另外如果你打算永不调用 Java 超类的某一方法，你不必指定一个 @sig 字符串。换句话说，如果方法 A 调用方法 B，方法 A 有一个 @sig 字符串，于是方法 A 是可从 Java 调用的，而方法 B 是不可从 Java 调用的；然而方法 B 仍然是可从方法 A 调用。不可从 Java 访问的类内部方法仍可从 Jython 访问。在某种意义上这起了加强抽象的作用。

第三个准则是类的名字和包含它的模块必须相匹配。程序清单 8-4 是一个称作 message.py 的 Jython 模块。程序清单 8-4 也包含了称做 message 的类。注意匹配类和模块名必须满足与 Java 兼容类的第三个准则。

### 8.5.1 一个与 Java 兼容的例子 Jython 类

在程序清单 8-4 里的 message 类有两个方法：\_init\_ 和 showMessage。其中的 showMessage 方法有一个 @sig 字符串。\_init\_ 方法没有 @sig 字符串，但它仍然可从 Java 访问。为什么？jythonc 能够测定这个 Java 超类的信息。Catch 是超类里的三个构造函数，现在三个构造函数都指向单个派生类里的 \_init\_ 方法，派生类仅选择实现一个特征，幸运地参数的数目使它清楚选择那一个。

注意仅没有重定义基类方法的方法需要 @sig。即使在不需要的地方你可选择提供一个 @sig——既没有害处，但也沒好处。

#### 程序清单 8-4 与 Java 兼容的 Jython 类

```
# file: message.py
import java

class message(java.awt.Frame):
    def __init__(self, name):
        self.setTitle(name)
        self.label = java.awt.Label("", java.awt.Label.CENTER)
        action = lambda x: java.lang.System.exit(0)
        button = java.awt.Button('OK', actionPerformed=action)
        p = java.awt.Panel(java.awt.GridLayout(2,1,2,2))
        p.add(self.label)
        p.add(button)
        self.add(p)

    def showMessage(self, text):
        "@sig public void showMessage(String text)"
        self.label.setText(text)
        self.pack()
        self.show()

if __name__ == '__main__':
    m = message("Test Message")
    m.showMessage("I look like Java, but I'm not")
```

为了在 Java 里使用程序清单 8-4 里的 message 类，它必须用 jythonc 编译。不需要任何特殊的选项，因此这像 jythonc message.py 一样简单（假设你与 message.py 在同一个目录下，并且 jythonc 是在你的路径）。再者注意依赖你的编译器版本，你可获取关于不支持 API 的使用的警告消息。运行 jythonc message.py 的输出得到：

```
processing message
Required packages:
java.lang
java.awt

Creating adapters:
java.awt.event.ActionListener used in message

Creating .java files:
message module
message extends java.awt.Frame

Compiling .java to .class...
Compiling with args: ['C:\\\\JDK1.3.1\\\\bin\\\\javac', '-classpath', '"C:\\\\jython2.1a1\\\\jython.jar;;.\\\\jpywork;;C:\\\\jython2.1a1\\\\Tools\\\\jythonc;C:\\\\WINDOWS\\\\Desktop\\\\ch0examples\\\\.;C:\\\\jython2.1a1\\\\Lib;C:\\\\jython2.1a1;c:\\\\jython2.1a1\\\\Lib\\\\site-python"', '.\\\\jpywork\\\\message.java']
0
```

你从jythonc获取的信息是有价值的。当编译与Java兼容类如message.py时特别重要的是当jythonc正在创建\*.java文件时打印出的信息：

```
Creating .java files:
  message module
    message extends java.awt.Frame
```

监视适当的extends消息是特别重要的，该消息保证你已满足与Java兼容要求的第一条。

既然你有message.class和message\$PyInner.class，你怎样从Java中使用它们？惟一的要求是把产生的类文件和jython.jar文件两者都放在classpath中，否则它与其他的Java类不会有任何区别。程序清单8-5是一个简单的使用message类的Java类，就好像它是本机Java。

#### 程序清单8-5 在Java里使用经过编译的Jython类

```
//file MessageTest.java
import message;

public class MessageTest {
    public static void main(String[] args) {
        message m = new message("Test");
        m.showMessageDialog("I look like Java, but I'm not");
    }
}
```

为了编译程序清单8-5，在classpath里包含jython.jar和message类。从目录里包含message.class和message\$PyInner.class及从Sun jdk使用javac，命令如下：

```
javac -classpath .;/path/to/jython.jar MessageTest.java
```

运行MessageTest类也需要jython.jar和两个message类文件都在classpath。如果message.class、message\$PyInner.class和MessageTest.class在同一个目录，用下列命令运行MessageTest：

```
java -cp .;/path/to/jython.jar MessageTest
```

相似于在图8-1那样的小的消息框(Box)将出现。

用@sig提供一个特定的方法特征符并不自动考虑到已重载的方法。是的，@sig字符串指定一个惟一的特征符，但像在第6章“类、实例和继承”中所提到的。如果Jython重定义一个Java方法，它必须处理所有同名的重载方法。你不能加一个@sig字符串以希望防止这种泛化发生。在Java超类里相似命名的方法不能自动处理不同于@sing字符串的参数类型，你也不能在Jython派生类里定义多重的、相似命名方法来仅仅基于@sig字符串处理不同的参数类型。

程序清单8-6用称作Base、sub和App的三个类来示范。

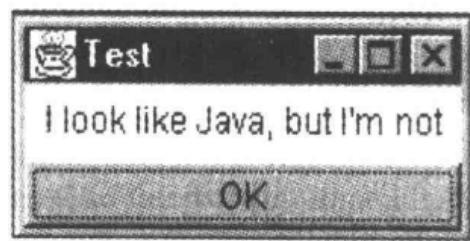


图8-1 一个Jython消息框

## 程序清单 8-6 重载的方法和 jythonc

```
// file: Base.java
public class Base {
    public String getSomething(String s) {
        return "string parameter. Base class.";
    }

    public String getSomething(int i) {
        return "int parameter. Base class.";
    }

    public String getSomething(float f) {
        return "float parameter. Base class.";
    }
}

#.....
#file: Sub.py

import Base

class Sub(Base):
    def getSomething(self, var):
        "@sig public String getSomething(String var)"
        return str(type(var)) + " Jython subclass."

//.....
//file: App.java

import Sub;

public class App {
    public static void main(String args[]) {
        Sub s = new Sub();
        System.out.println(s.getSomething("A string"));
        System.out.println(s.getSomething(1));
        System.out.println(s.getSomething((float)1.1));
    }
}
```

称作 `Base` 的基本的 Java 类有一个称做 `getSomething` 的重载方法。这个方法的三个版本共同地考虑到了 `String`、`int` 和 `float` 参数类型。称作 `sub` 的 Jython 派生类重定义 `getSomething` 方法，但有一个 `@sig` 字符串指定唯一的 `String` 参数类型。然而这不限制它重定义那个特定的方法特征符。方法特征符信息也是从超类找到的，因此超类里具有相同名字而有不同的特征符的所有方法自动地指向 Jython 派生类里那个名字的单一的方法。

为了编译程序清单 8-6 里的文件，把它们所有的都放在同一个目录中，改变控制台的工作目录到它们存储的地方，然后用下列命令编译（按指定的顺序）：

```
javac Base.java
jythonc -w . Sub.py
javac -classpath . App.java
```

用下列命令从同一目录里执行 App.java:

```
dos>java -cp .;\path\to\jython.jar App
*nix>java -c; ./path/to/jython.jar App
```

结果将与下面相似:

```
org.python.core.PyString Jython subclass.
org.python.core.PyInteger Jython subclass.
org.python.core.PyFloat Jython subclass.
```

你可看见 jythonc 已经用名字 getSomething 自动地载听每一个 Base 类的重载方法。依据这个名字的单个 Jython 方法是希望来处理这些条件的每一个和设法用没有任何修饰的 @sig 字符串防止这种泛化发生。

### 8.5.2 模块全局对象和与 Java 兼容类

增加 Java 方法特征符的严格性可能觉得与 Jython 的动态特性相矛盾。出现在 Jython 里的许多通用模式是依赖动态的特殊的方法如 `_getattr_` 和 `_setattr_` 或其他不容易转化成编译时的检查机制。另外，当用 jythonc 来创建与 Java 兼容类时模块全局对象可能是不清楚的。聚集类影响模块全局标志符 (module-global identifiers) 的值：这些对象在模块里定义，而不是在与 Java 兼容类里定义。虽然 Jython 类和那些有 @sig 字符串的方法是能从 Java 调用，但模块全局对像是不能的。然而，Jython 类可以使用模块级的代码，模块级代码甚至可以改变被 Java 使用的类的细节 (像在程序清单 8-7)。程序清单 8-7 在编译的类里使用了一点 Jython 的动态机制，也提供透明给模块全局对象。

为了给程序清单 8-7 设置阶段，设想用编译的 Jython 类原型化一个 Java 应用程序。你对静态方法做些什么呢？Jythonc 不允许你用 @sig 字符串指定 static，因此程序清单 8-7 被创造来在一个编译的 Jython 类里创建 static 方法的假象。它并不创建一个合法的静态方法——单词 “static” 不出现在它的特征符里，但它可效仿它。它可用其他的不能从 Java 调用的模块全局代码做到这样。铭记 Java 可以调用编译的 Jython 类里的最合适的方法是有用的，但 Jython 代码没有任何限制。产生的方法特征符是依附于 Jython 而不必改变它的灵活性。

程序清单 8-7 定义了一个称做 JavaPrototype 的类，该类包含了一个函数定义：test。test 函数除了 @sig 字符串外是空的。这个 test 函数所做的所有就是 jythonc 产生一个希望的 Java 方法特征符；它与运行功能没有关系。为什么？因为模块的最后一行赋了另一个类的一个可调用实例给 test。在程序清单 8-7 里的 staticfunctor 类定义了 `_call_`，这个类的实例是这个类提供希望的方法功能。如果你不知道为什么 arg 标志符出现在 test 方法的 @sig 字符串里，它是因为我们实际上是在为第二个类里的 `_call_` 方法设法产生一个特征符。下面是发生在程序清单 8-7 里的一个概括：

- 1) 产生一个 phony 方法和 @sig 字符串来巧妙地把 jythonc 引入增加希望的 Java 方法特征标记。
- 2) 定义一个实现 `_call_` 的类。`_call_` 定义实际上是希望的功能，因此方法特征符（第一步）应该符合它的需要。

3) 从 phony 方法赋标志符值给包含希望的\_call\_方法类的一个实例。

#### 程序清单 8-7 在编译的 Jython 里创建一个与静态相似的方法

```
# file: JavaPrototype.py
import java.lang.Object

class JavaPrototype(java.lang.Object):
    def test(self): # This will become static-like
        "@sig public void test(String arg)"

class staticfunctor(java.lang.Object):
    def __call__(self, arg):
        print "printing %s from static method" % arg

JavaPrototype.__dict__['test'] = staticfunctor()
```

如果 @sig 字符串不指定 static，它怎么能是 static？这是一个扩展，但在 Jython 意识里它起 static 的作用。如果你使用 Jython 里的 JavaPrototype 模块，你可以测试来看看类的多实例共享 test 函数的一个单个的通用实例：

```
>>> import JavaPrototype
>>> p1 = JavaPrototype.JavaPrototype()
>>> p2 = JavaPrototype.JavaPrototype()
>>> p1.test == p2.test
1
>>> p1.test is p2.test
1
>>> print id(p1.test), id(p2.test)
5482965 5482965
```

上面交互的例子使用了 JavaPrototype 模块，而不是编译的类。这引起了一个关于编译的 Jython 类的警告。你应该在 Jython 里使用模块和保存 jythonc 编译的类为在 Java 里使用。

程序清单 8-8 是一个小 Java test 类，该类引入和使用编译的 JavaPrototype 类来确认它也像 Java 类一样正当地运转。这假定你已使用 jythonc 来编译 JavaPrototype 模块到具有 jythonc JavaPrototype 的类文件。

#### 程序清单 8-8 从 Java 里使用 JavaPrototype 类

```
// file: Test.java
import JavaPrototype;

public class Test {
    public static void main(String[] args) {
        JavaPrototype i1 = new JavaPrototype();
        JavaPrototype i2 = new JavaPrototype();
        i1.test("THIS");
        i2.test("THAT");
    }
}
```

---

Place in the same directory as JavaPrototype.class and  
JavaPrototype\$PyInner.class, then compile with:  
javac -classpath . test.java

---

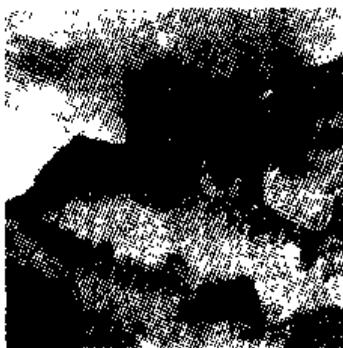
从运行

```
"java -cp .;\\path\\to\\jython.jar test" # dos  
"java -cp ./path/to/jython.jar test" # *nix:
```

输出：

```
printing THIS from static method  
printing THAT from static method
```





## 第9章 在 Java 里嵌入和扩展 Jython

将受益于 Jython 的高级、动态特性的 Java 应用程序通过嵌入解释器容易地使用 Jython。另外，需要额外速度的 Jython 应用程序可以包含用 Java 写的模块，同时允许它们在不影响像模块一样的功能性的情况下利用 Java 的有效率的字节码。这些实现被称做嵌入和扩展。两者都是在单应用程序里聚合多语言的方法。

尽管嵌入和扩展在许多语言里长期地是平凡的，但是 Jython-Java 结合在这点上非常好地超越了所有其他语言。Python 的 C 实现因为在聚合语言的效用或 Python 程序员称为“gluing（粘合）”的而被高度称赞。Jython 继承了这个成功之处，但也因为缩小了语言之间的接缝和间隙而超越了它。在 Java 和 Jython 之间传递对象是均衡地无缝——Java 对象能被传递进解释器，而 Jython 对象可以被传出到 Java。没有专用化的对象——任何对象，所有这些发生在没有引起了其他语言定制这些对象的消耗。另外，嵌入和初始化解释器、设定和获取对象和其他的相关任务用 Jython 是非常直观的和容易的，Jython 这种容易的优点加上 Java 和 Jyhton 的无缝集成使 Java 和 Jyhton 成为多语言应用程序的最佳结合。

本章的明显的组织结构分为两部分，一部分是嵌入，另一部分是扩展 Jython。

### 9.1 嵌入 Jython

在 Java 里嵌入 Jython 有许多用途，但当需要一个交互的命令解释器、创建不同的输出形式、应用动态配置和应用程序的特定元素需要频繁改变时，它是特别有用的。因为有许多其他方面为嵌入 Jython 开发出另外的用途，因此这个列表将毫无疑问地要增长。即使忽略设计动机，也有理由为嵌入 Jython 来影响它的优点。你可仅用增加嵌入的解释器得到增强的可读性、快速开发、短时间的学习曲线和更多其他的。仅有的缺点就是你引起嵌入解释器的额外内存和在应用程序里需要另外的类。

为了在 Java 里嵌入 Jython，org.python.util 包的三个类提供了必须的方法。这三个类是 PythonInterpreter、InteractiveInterpreter 和 InteractiveConsole。它们出现的顺序也反映了它们的层次。

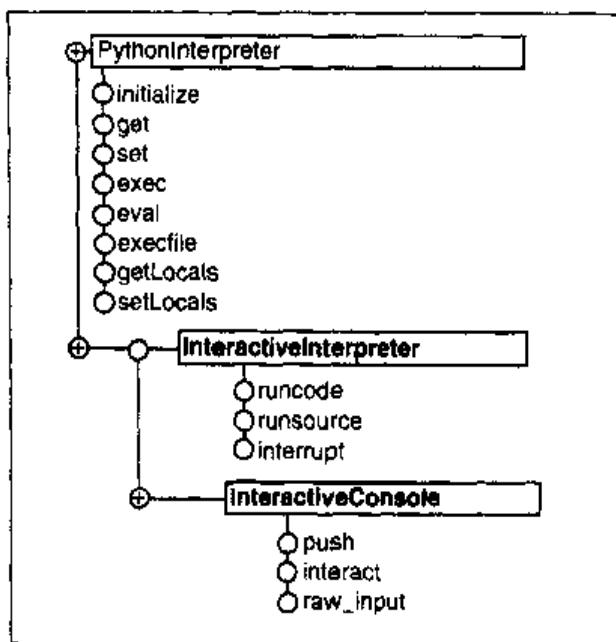


图 9-1 解释器层次和相关方法

每一个类都是列表里前面的派生类。图 9-1 给出了这些类和这些类相关的方法。

### PythonInterpreter

在 Java 应用程序里嵌入 Jython 经常等同于使用 Java 类 org.python.util.PythonInterpreter 的一个实例。用 Jython 安装的 jython.jar 文件包含了这个类。在 Java 里嵌入这个类只需要增加两个语句到 Java 代码：

```
import org.python.util.PythonInterpreter;
PythonInterpreter interp = new PythonInterpreter();
```

在上述例子里的 interp 标志符称为解释器实例或仅称为解释器，最好看作包容器或在 Java 里看作 bubble。这个类推应用于大多数对象，但在这里是特别有价值的因为它在使用 Jython 环境和使用 Jython 类之间强制区别。Jython 解释器运行在这个 bubble 里。包围解释器的 Java 代码不知道 bubble 里是什么，而 bubble 里的对象不知道外面的世界。创建 bubble 是极其简单的，就像在上面的两个例子行提到的，因此关心的是在解释器的实例化之前的步骤和在实例化它之后怎样使用它。下面部分详细说明了怎样初始化 Jython 环境、怎样实例化解释器、怎样使用解释器实例和怎样传递 Jython 和 Java 对象进和出嵌入的解释器。

#### 1. 初始化

Jython 环境依赖许多特性。像 python.home、python.path 和 python.security.respectJavaAccessibility 这样的特性通常是在启动时从注册文件装载或用命令行选项设定。这些特性和其他的特性保证正确的模块装载和确定其他方面的 Jython 行为。在 Java 里嵌入 Jython 在创建解释器对象前要考虑显式地设定这些特性的时机。当你在 PythonInterpreter 类里使用 static initialize 方法时，这些特性被设定。这种对特性、路径和设定的显式控制优于使用 jythonc 编译的类因为当嵌入时你完全有更多的控制。

#### 2. 特性和初始化

你应该在创建解释器实例之前初始化 PythonInterpreter。初始化是在 Jython 特性被设定、注册文件被处理和 Jython 包管理器搜寻并缓冲 Java 包信息时进行的。Initialize 步骤采用字面上的意义是因为它在 PythonInterpreter 使用了 static initialize 方法。下面是 initialize 方法的特征符：

```
public static void initialize(Properties preProperties,
                           Properties postProperties,
                           String[] argv)
```

参数也可读作“old properties”、“new properties”和“command-line arguments”。old 或 preProperties 常常被设定给 System.getProperties()。当 preProperties 被设定给 System.getProperties() 时，它们包含了用 Java 的-D 命令行转换开关设定的这些特性。new 或 postProperties 常常被设定给一个新的已增加了入口的 java.util.Properties 实例。Initialize 方法的第三个和最后一个自变量是一个 String 数组，这个字符串数组成为 Jython 的 sys.argv。

如果 Jython 不能找到它的 home 目录，则它可能不能读注册文件或查到模块的位置。你能

通过在 initialize 方法里设定 python.home 特性来保证嵌入的解释器能找到它的 home 目录。为了做到这样，你应使用下列语句：

```
Properties props = new Properties();
props.put("python.home", "/usr/local/jython-2.1"); // *nix
//props.put("python.home", "c:\\jython2.1"); // windows
PythonInterpreter.initialize(System.getProperties(),
    props, new String[0]);
```

初始化 PythonInterpreter 也处理 Jython 的注册文件（如果找到了）。在注册文件里设定创建了第三组特性。三组特性是 old 特性（上面例子的 System.getProperties()）、registry 特性（在注册文件里定义的特性）和作为 initialize 方法的第二个自变量出现的 new 特性（在前述例子里的“props”）。这些特性出现在前面句子的顺序是有意义的。当一个给定的特性在多个地方被定义时，这个顺序决定哪个特性优先。Old 特性首先被装载，接着是 registry 特性最后是 new 特性。这种优先意味着 registry 特性重定义（override）old 特性，而 new 特性重定义 registry 特性和 system 特性。

程序清单 9-1 是一个嵌入 PythonInterpreter 的 Java 类。程序清单 9-1 不仅示范了特性的设定和初始化解释器，而且它也是本章里许多例子的方便的基类。一个称为 org.python.demo 的包与程序清单 9-1 和有助于组织例子的将来的 Java 列表一起使用。在列表里的 Embedding 类假定一个用-D 选项在命令行显式地声明的特性有最高的优先级，因此它明确地设定在 System.getProperties() 里找到的命令行特性，System.getProperties() 在新的特性 postProps 里以字符串 python 开始。Embedding 类也在 sysProps 对象里设定 python.home 特性，但只要它不是在命令行设定。这有助于确保解释器找到和处理注册文件。在这个脚本里，确保 python.home 特性有最低的优先级，但没关系，因为它通常不是在 registry 里设定的。registry 特性装载下一个，于是 new 特性被设定和重定义特性具有的任何以前的值的副本，再改变到包括那些具有-D 转换开关的特性组。

#### 程序清单 9-1 一个简单嵌入的 PythonInterpreter

---

```
// file: Embedding.java
package org.python.demo;

import java.util.Properties;
import org.python.util.PythonInterpreter;
import java.util.Enumeration;

public class Embedding {

    protected PythonInterpreter interp;

    public static void main(String[] args) {
        Embedding embed = new Embedding();
        embed.initInterpreter(args);
        embed.test();
    }

    protected void initInterpreter(String[] argv) {
        // Get preProperties postProperties, and System properties
        Properties postProps = new Properties();
```

```

Properties sysProps = System.getProperties();

// set default python.home property
if (sysProps.getProperty("python.home")==null)
    sysProps.put("python.home", "c:\\\\python 2.1a1");

// put System properties (those set with -D) in postProps
Enumeration e = sysProps.propertyNames();
while (e.hasMoreElements()) {
    String name = (String)e.nextElement();
    if (name.startsWith("python."))
        postProps.put(name, System.getProperty(name));
}

// Here's the initialization step
PythonInterpreter.initialize(sysProps, postProps, argv);
//instantiate- note that it is AFTER initialize
interp = new PythonInterpreter();
}

public void test() {
    // Print system state values to confirm proper initialization
    interp.exec("import sys");
    interp.exec("print"); // Add empty line for clarity
    interp.exec("print 'sys.prefix=' , sys.prefix");
    interp.exec("print 'sys.argv=' , sys.argv");
    interp.exec("print 'sys.path=' , sys.path");
    interp.exec("print 'sys.cachedir=' , sys.cachedir");
    interp.exec("print"); // Another blank for clarity
}
}

```

在程序清单 9-1 里的 test() 方法是下面许多例子将重定义的方法。你将可在这个方法里测试在相关文本里讨论的解释器命令。在程序清单之前的 test 方法用解释器的 exec 方法执行一些 Jython 语句。Jython 语句引入 Jython 的 sys 模块，然后使用 sys 来打印确认特性被正确地设定的信息。使用 exec 方法的详细说明是在本节的后部，但是现在在每一个 exec 方法里注意仔细使用单个和双重引用是足够了。

程序清单 9-1 和通常使用 system 路径的特性需要特别的关注来确信它们对路径和目录分隔符遵守特定平台规则（例如，对 windows 是 ; 和 \\，而对 \*nix 是 : 和 /）。也要注意程序清单 9-1 是假定你的 python.home 目录是 c:\\python2.1 和你完全许可使用那个目录。如果那不是你的 python.home 目录，改变它到正确的值。

为了在程序清单 9-1 编译 Embedding 类，选择一个工作目录，并在它里面创建 org/python/demo 目录，然后把 Embedding.java 文件放在 {working directories}\org/python/demo 目录，因此这个目录与指定的包相匹配，再从你的工作目录里运行下列命令（对 DOS 用 \ 代替 /）：

```
javac -classpath /path/to/jython.jar org/python/demo/Embedding.java
```

执行 Embedding 类的命令应该与下面类似，注意使用指定的路径到你的平台：

```
java -classpath /path/to/jython.jar:. org.python.demo.Embedding
```

下面是运行 Embedding 类的例子输出：

```
sys.prefix= /usr/local/jython-2.1
sys.argv= []
sys.path= ['.', '/usr/local/jython-2.1/Lib']
sys.cachedir= /usr/local/jython-2.1/cachedir
```

按照推测 Embedding 类使用在命令行上设定的特性。使用一个具有增加的-D 转换开关的命令行确定正确的特性设置。下面在 Windows 上示范了如何做，注意命令是一行，但它重叠占有下面开始的两行，也要注意你必须设置 cachedir 到一个你有足够的特权的目录。

```
java -Dpython.cachedir="newcache" -Dpython.path="d:\\devel\\jython" -classpath
"c:\\jython-2.1\\jython.jar"; org.python.demo.Embedding
*sys-package-mgr*: processing new jar, 'C:\\jython-2.1\\jython.jar'
*sys-package-mgr*: processing new jar, 'C:\\jdk1.3.1\\jre\\lib\\rt.jar'
*sys-package-mgr*: processing new jar, 'C:\\jdk1.3.1\\jre\\lib\\i18n.jar'
*sys-package-mgr*: processing new jar, 'C:\\jdk1.3.1\\jre\\lib\\sunrsasign.jar'

sys.prefix= c:\\jython-2.1
sys.argv= []
sys.path= ['.', 'c:\\\\jython-2.1\\\\Lib', 'd:\\\\devel\\\\jython']
sys.cachedir= c:\\jython-2.1\\newcache
```

如果你在指定的位置还没有一个 cache 目录，你应该看到缓冲消息。当嵌入 Jython 时明显地出现搜索 Java 包和缓冲结果，就像当运行 Jython 程序本身时出现的一样。万一在你的应用程 序里时间选择有关系，这个缓冲发生在初始化而不是当 PythonInterpreter 被实例化时。注意如果不提供 system 特性给 initialize 方法（在程序清单 9-1 里的第一个参数），上面例子里提到的缓冲就不会出现，因为解释器没有找到 jars 所需要的信息。

### 3. 实例化解释器

程序清单 9-1 已经使用了 PythonInterpreter 的空构造函数，但还有另外的允许设置解释器的名空间和 system 状态的构造函数。PythonInterpreter 类的三个构造函数特征符如下：

```
public PythonInterpreter()
public PythonInterpreter(PyObject dict)
public PythonInterpreter(PyObject dict, PySystemState systemState)
```

第一个构造函数是直观的、无参数的版本，第二个构造函数接受了一个 PyObject 作为自变量，这个 PyObject 成了解释器的名空间；因此，它经常是一个 org.python.core.PyStringMap 对象。PyDictionary 或另一个实现了一些定制功能的像字典的 PyObject 也将起作用，但除非有理由这样做，否则使用 PyStringMap。除了有定制名空间对象的行为外，提供解释器的名空间允许你在解释器的名空间里在实例化对象之前设置它，或在脚本调用之间存储名空间。在 PyStringMap 里设置对象，并把它传递到 PythonInterpreter 的构造函数如下所示：

```
// the required imports
import org.python.core.*;
import org.python.util.PythonInterpreter;
```

```
// Assume the interpreter is already initialized
PyStringMap dict = new PyStringMap();
dict.__setitem__("Name", new PyString("Strategy test"));
dict.__setitem__("Context", Py.java2py(new SomeContextClassOrBean()));
PythonInterpreter interp = new PythonInterpreter(dict);
```

注意在 Java 里使用 Jython 对象利用在第 7 章“高级类”中描述的 Jython 的特殊方法。在前述的例子中，在名空间对象里 PyStringMap 的 \_\_setitem\_\_ 被用来设置 items。Java 明显不能自动应用 Jython 的动态的特殊方法（有两条头和尾下划线的方法），因此在 Java 里使用 Jython 对象时你必须在合适的地方显式应用这些特殊方法。

当你实例化解释器时，PythonInterpreter 类的第三个构造函数允许你也像 PySystemState 对象一样设置名空间。后面部分解释了很多关于 PySystemState 对象，但是到现在知道它是在它的 Java 角色里的 Jython 的 sys 模块就足够了。使用 PySystemState 对象来设置被解释器使用的 sys.path 将需要下列行：

```
PyStringMap dict = new PyStringMap();
dict.__setitem__("A", new PyInteger(1));
PySystemState sys = new PySystemState();
sys.path.append(new PyString("c:\\\\python-2.1\\\\Lib"));
sys.path.append(new PyString("c:\\\\windows\\\\desktop"));
sys.path.append(new PyString("d:\\\\cvs"));

// instantiate with namespace and system state objects
interp = new PythonInterpreter(dict, sys);
```

如果你嵌入多个解释器，它们将共享 system state。这有用构造函数使用 PySystemState 的含义，因为即使调用看起来对正在被创建的解释器实例是孤立的，当 PySystemState 对象被改变时所有嵌入的解释器的 system state 被改变，然而局部名空间对每一个解释器实例来说是惟一的。

#### 4. 设置输出流和错误流

PythonInterpreter 实例有为设置输出流和错误流的模块，这些方法是 setOut 和 setErr，每一个这些方法都接受单个自变量，但自变量可以是 java.io.OutputStream、java.io.Writer 或任何像文件对象的 org.python.core PyObject。程序清单 9-2 利用 setErr 方法来重定向错误消息到一个文件。

#### 程序清单 9-2 设置 Error 流到 FileWrite

```
// file: ErrorRedir.java
package org.python.demo;

import org.python.demo.Embedding;
import java.io.*;

public class ErrorRedir extends Embedding {

    public static void main(String[] args) {
        ErrorRedir erd = new ErrorRedir();
        erd.initInterpreter(args);
        erd.test();
    }
}
```

```

public void test() {
    // Redirect errors to a file
    try {
        interp.setErr(new FileWriter(new File("errors")));
    } catch (IOException e) {
        e.printStackTrace();
    }

    interp.exec("assert 0, 'This should end up in a file.'");
}
}

```

用 `javac -classpath c:\path\to\jython.jar;.org\python\demo\ErrorRedir.java` 编译程序清单 9-2 后，并用 `java -cp c:\path\to\jython.jar;.org.python.demo.ErrorRedir` 执行它，你在控制台里仅看到下列消息：

`Exception in thread "main"`

剩下的 traceback 信息出现在你的当前目录里的文件 `errors` 中，改变 error 流和 output 流到 network 流或其他的资源也是可能的事情。

### 5. PySystemState

在 Jython 里，`sys` 模块包含关于 `system` 状态的信息。在 Jython 里，你可通过引入 `sys` 模块和打印它里面的变量来观看 `system` 状态信息。你也可改变对象如追加到 `sys.path` 或赋新的对象到标准 `input`、`output` 和 `error` 对象。当你嵌入 Jython 时，你显然可能使用解释器实例里的 `sys` 对象，但你也可以使用解释器外面的 `sys` 模块。为了从 Java 使用 `sys` 模块目录，你可以使用它的 Java 解释器：`PySystemState`。注意使用 `PythonInterpreter` 类的方法是适合嵌入解释器的被推荐的 API。如果一个操作用 `PythonInterpreter` 方法完成，那就是它应该怎样被执行，然而 `PySystemState` 对象需要注意因为它经常出现在 Jython 的代码基区，它包含了对 Jython 的包管理器（特别是当嵌入时）是非常重要的类，由于使用 Jython 里 `sys` 模块，对用户来说已是非常熟悉的，同时也把它变成在 Java 里容易使用的工具。

`import sys` 在 Jython 的表示变成了下列在 Java 的表示

```

import org.python.core.*;
PySystemState sys = Py.getSystemState();

```

因为 `PySystemState` 类和 `Py` 类两者都出现在 `org.python.core` 包，这个包是首先被引入的，在用 `Py.getSystemState()` 查询 `PySystemState` 对象后，你可从解释器外面的 Java 代码改变解释器里面的状态。程序清单 9-3 示范了怎么通过使用 `PySystemState` 对象来追加值到 `sys.path` 变量。在初始化时或像下面一样在 Jython 代码里通过设置 `python.path` 特性正常转变到 `sys.path`。

```

import sys
sys.path.append("c:\\windows\\desktop")

```

这个相同的操作能用 `PySystemState` 对象完成就像程序清单 9-3 所示范的一样。

## 程序清单 9-3 使用 Java 里的 sys 模块

```
// file: SysState.java
package org.python.demo;

import org.python.demo.Embedding;
import org.python.core.*;

public class SysState extends Embedding {

    public static void main(String[] args) {
        SysState s = new SysState();
        s.initInterpreter(args);
        s.test();
    }

    public void test() {
        System.out.println("sys.path before changes to sys:");
        interp.exec("import sys\n" +
                   "print sys.path\n" +
                   "print");

        // Get the system state and append to its path
        PySystemState sys = Py.getSystemState();
        sys.path.append(new PyString("c:\\windows\\desktop"));

        System.out.println("sys.path after changes to sys:");
        interp.exec("print sys.path");
    }
}
```

用下列命令编译 SysState 类：

```
javac -classpath c:\path\to\jython.jar;. org.python.demo.SysState.java
```

运行 SysState 类的输出如下。注意执行类的命令尽管在例子里包含了两行但仅是一个命令（没有任何返回）：

```
dos>\jdk1.3.1\bin>java -classpath "c:\jython-2.1\jython.jar";. org.python.demo.SysState
The sys.path before changes to PySystemState:
['.', 'c:\\jython-2.1\\Lib', 'd:\\python20\\lib']

The sys.path after changes to PySystemState:
['.', 'c:\\jython-2.1\\Lib', 'd:\\python20\\lib', 'c:\\windows\\desktop']
```

这个例子第二次打印 sys.path，另外，通过 PySystemState 对象增加的 sys.path 人口出现了。如果你已嵌入了多个解释器，system 状态的变化已经影响了每一个解释器。

Jython 的 PySystemState 类 (sys 模块) 也包含了与类装载有关的三个方法：add\_package、add\_classdir 和 add\_extdir。当嵌入 Jython 时这三个方法变得重要，因为 Java 应用程序将有它们自己的类装载器。当一个应用程序的类装载器不同于 Jython 的时，Jython 并不总是正确地识别或找到某一个 Java 包。这创建了环境，在该环境里 Jython 有助于识别它可以引入的 Java 类，这也是

这三个方法被提到的理由。

`add_package`方法把指定的Java包放在Jython的包管理器里，如果company A有一个它具有自己的类装载器的应用程序，并在它里面嵌入一个Jython解释器，则嵌入的解释器可能不能正确地识别Java包，如`com.A.python`（假定它存在）。解释器不能在这个包里装载类，除非company A使用`add_package`方法来确保正确的包识别并因此装载。下面的snippet示范：

```
import org.python.core.*;
PySystemState sys = Py.getSystemState();
sys.add_package("com.A.python");
```

`add_package`方法不引入或转载包，应用程序的类装载器引入或转载包。`add_package`方法仅仅把包加到Java包的列表，Jython可以从该Java包引人类。

`add_classdir`和`add_extdir`方法在两者都把定位加到列表方面是相似的，Jython在该列表里搜索Java包。`add_classdir`方法使在指定目录里的包可用。如果你有在\*nix目录`/usr/java/devel`里开始的包和类的层次图，则如下把这个树形图加到Jython的包管理器：

```
import org.python.core.*;
PySystemState sys = Py.getSystemState();
sys.add_classdir("/usr/java/devel/");
```

`add_extdir`方法把一个目录里的存档文件的内容加到Jython搜索Java包的位置列表。如果你把应用程序需要的jar和zip文件放在\*nix目录`/usr/java/lib`里，同时把所有这些存档文件的内容放到Jython的包管理器如下：

```
import org.python.core.*;
PySystemState sys = Py.getSystemState();
sys.add_extdir("/usr/java/lib/");
```

初始化解释器和使用三个`add_*`方法的一个好的例子是伴随Jython产生的`org.python.util.PyServlet`类。程序清单9-4是一个在`PyServlet`里的`init()`方法的修订版本，在这个方法里解释器被初始化和所有需要的Java包被加到包管理器。

#### 程序清单9-4 Jython PyServlet类

---

```
public void init() {
    Properties props = new Properties();
    if (props.getProperty("python.home") == null &&
        System.getProperty("python.home") == null)
    {
        props.put("python.home", rootPath + "WEB-INF" +
                  File.separator + "lib");
    }

    PythonInterpreter.initialize(System.getProperties(),
                               props, new String[0]);

    PySystemState sys = Py.getSystemState();
    sys.add_package("javax.servlet");
    sys.add_package("javax.servlet.http");
    sys.add_package("javax.servlet.jsp");
```

```

    sys.add_package("javax.servlet.jsp.tagext");
    sys.add_classdir(rootPath + "WEB-INF" +
                     File.separator + "classes");
    sys.add_extdir(rootPath + "WEB-INF" +
                     File.separator + "lib");
}

```

记住 add\_package、add\_classdir 和 add\_extdir 不装载任何东西。应用程序的类装载器处理实际工作，而 add\_\* 方法仅仅是使另外的 Java 包可用来引入。这些方法在 Jython 的 sys 模块里是可用的。Add\_\* 方法的描述在这里出现是因为当嵌入时它们经常被使用，但是没有任何意图指它们仅被限制于嵌入 Jython。你可调用这些具有 sys.add\_\* 的嵌入的 Jython 解释器里的方法，甚至当没有嵌入时你也可能需要使用 Jython 里这样的类。下面是一个小的交互解释程序 session，在该程序 session 里用 sys.add\_extdir 方法来使所有 Tomcat web 服务器的 lib 目录里的 jar 文件里的包可以从 Jython 引入（Tomcat 是 Apache 的 Jakarta project 的可用的 Web 服务器，地址是 <http://jakarta.apache.org>。关于 Tomcat 更多的内容在第 12 章“服务器端 Web 编程”中阐述）。

```

>>> import sys
>>> sys.add_extdir("c:\\web\\tomcat\\lib")
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\ant.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\jaxp.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\servlet.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\parser.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\webserver.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\jasper.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\zxJDBC.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\tools.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\ecs.jar'
*sys-package-mgr*: processing new jar, 'C:\\web\\tomcat\\lib\\jython.jar'
>>> from javax.servlet import http
>>> dir()
['__doc__', '__name__', 'http', 'sys']

```

我们看到 http 包变得可用，但 add\_extdir 仅使包可用，不包括类。如果交互的 session 继续企图从 ext\_dirs 引入一个类，我们看到下列：

```

>>> from javax.servlet.http import HttpServlet
Traceback (innermost last):
  File "<console>", line 1, in ?
ImportError: cannot import name HttpServlet

```

## 6. 使用解释器

运行解释器里的代码需要使用解释器的 exec、execfile 的一个或 eval 方法。

### 7. exec

解释器的 exec 方法允许你执行一串 Python 代码或一个预先编译的代码对象。当执行一串 Python 代码时，你在每一次调用 exec 必须使用完整的、语法上完全正确的语句，换句话说，你不能 exec 函数的第一行，用后来的调用 exec 来完成函数定义。对执行的字符串长度没有任何

static 限制。exec 字符串能包含语句、赋值、函数和在 Jython script 文件里能正常找到的任何东西。exec 方法不返回任何对象：执行 Jython 代码字符串的全部结果保存在解释器 bubble 里。

在本章前面的程序清单示范了 exec 方法，但它们仅使用简单的语句。复合语句更加困难因为它们需要特殊的格式。用在 exec 方法里的格式字符串是嵌入手段的一部分，但格式的扩展避开双引号或用单引号代替双引号、在需要的地方增加换行和使用适当的缩排，换句话说，如果你打印你打算送到 exec 方法的字符串，它应该看起来像语法上正确的 Jython script。程序清单 9-5 给出了一个简单的 Jython 函数定义，然后给出了同一个重写的函数定义来用嵌入的解释器运行。

程序清单 9-5 为了适合解释器的 exec 方法格式化 Jython

```
# in Jython
def movingAverage(datalist, sampleLen):
    """movingAverage(list, sampleLen) -> PyList"""
    add = lambda x, y: x + y
    return [reduce(add, datalist[x:x+10])/sampleLen
            for x in range(len(datalist) - sampleLen)]

import random
L = [random.randint(1,100) for x in range(100)]
print movingAverage(L, 10)

// in Java
interp.exec("def movingAverage(datalist, sampleLen):\n" +
    "    """movingAverage(list, sampleLength) -> PyList"""\n" +
    "    add = lambda x, y: x + y\n" +
    "    return [reduce(add, datalist[x:x+10])/sampleLen" +
    "            for x in range(len(datalist)-sampleLen)]\n" +
    "import random\n" +
    "L = [random.randint(1,100) for x in range(100)]\n" +
    "print movingAverage(L, 10);
```

在前述例子里的 Java 版本连接字符串来创建等同于 Jython 版本的字符串。大量的引用，或直接在前一行引用下，并在每一行的开始放置引用使适当的 Jython 缩排的复制变得容易。Jython 代码需要的引用使用单引号来避免与 Java 引号相冲突。如果在字符串里需要双引号，确保避开它们。换行符 (\n) 也必须按照 Jython 语法加在 exec 字符串需要的地方。

嵌入的解释器常常从外部资源如 network 流、文件或存档文件资源的一部分获得代码。一些应用程序可以动态生成被解释器使用的代码，然而，如果代码是一个文件的整个内容，解释器的 execfile 方法比 exec 方法更适合这种情况。

## 8. execfile

PythonInterpreter 的 execfile 方法允许你执行一个文件或 InputStream。Execfile 方法有三种形式。第一种形式接受一个要执行的文件名的单字符串自变量，第二种形式接受一个单 InputStream 对象。因为没有与 InputStream 相关的名字，错误消息将在错误消息的文件名位置显示 < iostream >。execfile 方法的第三种形式接受一个 InputStream 对象和一个在错误消息的文件名域里使用的 String 对象。这三个方法的 Java 方法特征符如下：

```
public void execfile(String s)
public void execfile(java.io.InputStream s)
public void execfile(java.io.InputStream s, String name)
```

程序清单 9-6 包含了一个实现所有三个 execfile 方法的类。程序清单 9-6 需要一个命令行自变量来指定要执行的文件。这个自变量对那个文件不是绝对路径，而仅是实际上存在用户的 home 目录里的文件名。用户的 home 目录对创建完全限定的路径的文件名不是预先决定的。如果我的 home 目录 /home/rbill 包含文件 exectest.py，命令行自变量应该仅仅是 exectest.py。如果你不能确定你的 home 目录，用 Jython 运行下列语句：

```
>>> import java
>>> print java.lang.System.getProperty("user.home")
```

### 程序清单 9-6 execfile 方法的演示

```
// file: ExecFileTest.java
package org.python.demo;

import org.python.demo.Embedding;
import java.io.*;
import org.python.core.*;

public class ExecFileTest extends Embedding {

    public static void main(String[] args) {
        ExecFileTest eft = new ExecFileTest();
        eft.initInterpreter(args);
        eft.test();
    }

    public void test() {
        PySystemState sys = Py.getSystemState();
        if (sys.argv.__len__() == 0) {
            System.out.println("Missing filename.\n" +
                               "Usage: ExecFileTest filename");
            return;
        }

        String home = System.getProperty("user.home");
        String filename = home + File.separator +
                          sys.argv.__getitem__(0);

        // Using a file name with execfile
        interp.execfile(filename);

        // Using an InputStream with execfile
        try {
            FileInputStream s = new FileInputStream(filename);
            interp.execfile(s);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        // Using an InputStream and a name with execfile
    }
}
```

```

try {
    FileInputStream s = new FileInputStream(filename);
    interp.execfile(s, sys.argv._getitem_(0).toString());
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
}
}
}

```

你可用下列命令编译 ExecfileTest.java 文件：

```

javac -classpath c:\path\to\jython.jar;.
org\python\demo\ExecFileTest.java

```

在你的包含行 print ‘It worked’ 的 home 目录里创建一个称为 test.py 的 Python 文件。你可用下列命令在一行里执行这个文件三次——execfile 方法每个一次：

```

java -cp c:\path\to\jython.jar;. org.python.demo.ExecFileTest test.py
It worked.
It worked.
It worked.

```

### 9. eval

eval 方法在三个方面不同于 exec 方法。eval 方法总是返回表达式的结果，该表达式的值为 org.python.core PyObject——Jython 等同于 Java 的 java.lang.Object 类。作为对照，exec 和 execfile 方法的返回类型总是 void，exec 方法接受 String 或经过编译的代码对象。最后，解释器的 eval 方法仅求表达式的值，而 exec 方法执行任意的 Jython 代码。

介绍解释器的 eval 方法也是一个来介绍 Java 里的 Jython 内置函数使用的好机会。解释器的 eval 方法是一个 Jython 的内置 eval 方法的封装器（wrapper）。解释器的 eval 接受字符串代码，并依次调用具有两个小修改的内置 eval 方法。第一个修改是它增加解释器的局部名空间作为内置 eval 方法的第二个自变量，第二个修改是它把代码字符串从 java.lang.String 转变到 org.python.core.PyString 对象。下面代码使用解释器的 eval 方法和内置 eval 方法两者给出了 eval 操作：

```

// The interpreter's eval shortcut
interp.eval("1 or 0") // The interpreter's eval()
// The built-in version
__builtin__.eval(new PyString("1 or 0"), interp.getLocal())

```

你可看到 Jython 的内置函数通过类 org.python.core.\_builtin\_ 是可用的。不仅 eval 方法这样可用，大多数 Jython 的内置函数通过 \_\_builtin\_\_ 都是可用的。这些内置函数可直接从 Java 使用，当与 PyObject 一起作用时在 Java 里使用 Jython 的内置函数变得方便，因为 eval 方法总是返回一个 PyObject，你可以使用内置方法对 eval 返回的对象进行实验。

内置 eval 函数和这样的解释器的 eval 方法求取一个表达式的值。Eval 表达式有与 lambda 表达式一样的约束：它不能包含赋值或语句。相反，代码字符串必须仅使用函数、方法、数据对

象、文字和那些不执行名字绑定操作的操作。

程序清单 9-7 为用嵌入的解释器求表达式的值创建了一个交互的循环。程序清单派生本章早先讨论的 Embedding，因此它从 Embedding 类继承了解释器初始化和实例化。程序清单 9-7 最重要部分就是它确定了 eval 方法怎样返回 PyObject，并给出了怎样在那个对象上使用内置函数 type()。在 EvalTest 类里的命令循环也提供了机会来检查表达式。

#### 程序清单 9-7 一个 eval 循环

```
// file: EvalTest.java
package org.python.demo;

import org.python.demo.Embedding;
import java.io.*;
import org.python.core.*;

public class EvalTest extends Embedding {
    private static BufferedReader terminal;

    public static void main(String[] args) {
        EvalTest et = new EvalTest();
        et.initInterpreter(args);
        et.test();
    }

    public void test() {
        System.out.println("Enter strings to evaluate at the prompt");
        interact("eval> ");
    }

    public void interact(String prompt) {
        terminal = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter 'exit' to quit.");

        String codeString = "";
        while (true) {
            System.out.print(prompt);
            try {
                codeString = terminal.readLine();
                if (codeString.compareTo("exit") == 0) System.exit(0);
                processInput(codeString);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public void processInput(String input) {
        PyObject o = interp.eval(input);
        System.out.println("Results is of type " +
                           __builtin__.type(o));
        System.out.println(o);
    }
}
```

用下列命令编译 EvalTest 类：

```
javac -classpath c:\path\to\jython.jar;. org.python.demo.EvalTest.java
```

用下列命令执行 EvalTest 类：

```
java -cp c:\path\to\jython.jar;. org.python.demo.EvalTest
```

下面是运行 EvalTest 类的例子输出：

```
Enter strings to evaluate at the prompt
Enter "exit" to quit.
eval> 0 or 10
Results is of type <jclass org.python.core.PyInteger at 6851381>
10
eval> 1 and "A string"
Results is of type <jclass org.python.core.PyString at 7739053>
A string
eval> [x for x in range(2,37) if 37%x == 0] == []
Results is of type <jclass org.python.core.PyInteger at 6851381>
1
eval> a = 10 # Assignments are not allowed
Exception in thread "main" Traceback (innermost last):
  (no code object) at line 0
  File "<string>", line 1
    a = 10 # Assignments are not allowed

SyntaxError: invalid syntax
```

## 10. 为随后的使用编译代码

在有代码字符串的 Jython 解释器里，调用 exec 和 eval 方法使解释器首先编译那些代码。然后执行那些代码。编译步骤可从容进行；因此，如果一个应用程序有大量的 Jython 代码字符串或多次使用的代码，它可能受益于编译在应用程序的关键部分之外的代码。也许你有一个 statistics 应用程序，该程序循环通过许多不同的数据，但对每一个数据都使用相同的 Jython 代码。宁愿每次反复调用具有一串 Jython 代码的 exec 或 eval，最好在循环之前编译那些代码，甚至在启动时编译它，并为随后的使用保存经过编译的代码对象。处理这些需要 Jython 的内置 compile 函数。

从 Java 里使用 Jython 的内置 compile 函数来创建一个经过编译的代码对象包括三步：引入需要的类、创建一个 java.lang.String 和编译 String 到一个代码（PyCode）对象。引入必须包括在 org.python.core 包里找到的 \_builtin\_ 和 PyCode 类，通过 Java 类 org.python.core.\_builtin\_ 访问 Jython 的内置函数。Compile 函数需要一个 java.lang.String 对象作为自变量，compile 函数返回一个 org.python.core.PyCode 对象。你经常看到用来包含引入需要的 import org.python.core.\*。compile 函数返回 PyCode 对象，这也是所希望的编译的对象。在没有引起编译消耗下，你可在将来执行这个对象或求取这个对象的值（依赖 compile 函数的第三个自变量）。这些行是用 exec 方法来创建要使用的 PyCode 对象：

```
// Import required classes (PyCode and __builtin__)
import org.python.core.*;

// Create a java.lang.String
String s = "print 'Hello World'";
```

```
// compile with "<>" for the file-name, and "exec" for type of object
PyCode code = __builtin__.compile(s, "<>", "exec");
// exec with interpreter. Note: no return object
interp.exec(code);
```

内置 compile 函数需要三个自变量：代码的 PyString、文件名（用在错误消息）和模式。如果模式是一个单语句，则它可以是 exec、eval 或 single。因为在上面例子的 compile 语句用 exec 模式创建了一个代码对象，你可以把对象传递到嵌入的解释器的 exec 方法。嵌入的解释器的 exec 方法与代码对象一起作用，也与字符串一起作用。然而解释器的 eval 方法只与字符串一起作用，这是因为使用内置 eval 来执行经过编译的代码对象是足够容易的，为了使用具有经过编译的代码对象的内置 eval，调用 \_\_builtin\_\_.eval 方法，并包括解释器的作为第二个自变量的名空间，下面示范：

```
import org.python.core.*;
String s = "1 and 0 or 10";
PyCode code = __builtin__.compile(s, "<>", "eval");

// Fetch the interpreter's local namespace
PyCode locals = interp.getLocals();

// use __builtin__.eval with interp's locals
PyObject o = __builtin__.eval(code, locals);
```

前述的行常常被缩短为：

```
import org.python.core.*;
PyCode code = __builtin__.compile("1 and 0 or 10", "<>", "eval");
PyObject o = __builtin__.eval(code, interp.getLocals());
```

## 11. 处理解释器里的异常

如果在解释器里出现了错误将会发生什么呢？答案就是解释器产生异常 org.python.core.PyException。如果 Jython 的 open 函数打开文件失败，Java 代码像 Jython 一样不能得到一个 IOError 异常，但相反会得到 PyException。下面给出了 Jython 里的 try/except 语句，如果你设法把异常处理移到嵌入的解释器周围的 Java 代码，那么它看起来像什么？

```
# in Jython
try:
    f = open("SomeNonExistingFile")
except IOError:
    e, i, tb = sys.exc_info()
    print e, "\n", tb.dumpStack()

// in Java
try {
    interp.exec("f = open('SomeNonExistingFile')");
} catch (PyException e) {
    e.printStackTrace();
}
```

PyException stack trace 将包括关于实 Jython 异常的信息，但它不帮助区分 catch 子句里的 Jython 异常的类型。即使你在产生 Java 异常的解释器里使用 Java 对象，封装它的 Jython 也会把这个异常转变成 PyException。在一个 Java catch 子句里捕获一个特定的 Jython 异常是不可能的；因

此，你必须在 Jython 解释器对象里使用一个 try/excep 语句，通过解决 Java 代码周围的错误来避免 Jython 异常，否则你必须通过使用 Py.matchException 来识别 Java 的 catch 子程序块里的实异常类型。

假设你有一个命令循环和仅需要捕获它里面的 IOException。仅仅捕获 Java 里的 PyException 不允许那种特性，因此你反而能在解释器里这样使用 try/excep：

```
interp.exec("try:\n" +
    "    _file = open(" + filename + ")\n" +
    "except:\n" +
    "    print 'File not found. Try again.'");
```

你能交替地，解决 Java 代码里的错误捕获和后来的使用对象，或者你把对象传送到解释器，该对象应该经过了读写检验。下面再假设有一个已经加入了文件名的命令循环，另外，需要避免 FileNotFoundException 或 Jython 的 IOError 以便命令循环能继续。这个例子是在解释器里打开这个文件之前首先测试它。并假设可读文件从解释器里打开是安全的：

```
File file = new File(filename);
if (file.canRead()!=true) {
    System.out.println("File not found. Try again.");
    break;
}
// File was found and is readable. safe to open.
interp.exec("file = open(" + filename + ")");
```

Org.python.core.Py 类包含了函数 matchException 和 Jython 的异常对象；使用 Py.matchException 函数来比较 PyException 与特定的 Jython 异常，允许你识别 catch 子句的相关程序代码块里的特定的 Jython 异常。Py.matchException 函数把 PyException 作为第一个自变量，Jython 异常（如 Py.IOError）的 PyObject 作为第二个自变量，并返回一个 Boolean 值来显示 PyException 是不是具有特定的 Jython 异常类型。假设你在如下一个 try/catch 语句里包含一个解释器的 exec 语句：

```
try {
    interp.exec("f = open('SomeNonExistingFile')");
} catch (PyException e) {
    //fill in later
}
```

然后，你可以像这样增加对特定 Jython 异常的测试：

```
try {
    interp.exec("f = open('SomeNonExistingFile')");
} catch (PyException e) {
    if (Py.matchException(e, Py.AttributeError)) {
        //handle Jython's AttributeError here
    } else if (Py.matchException(e, Py.IOError)) {
        //handle Jython's IOError here
    }
}
```

## 12. set 和 get 方法

PythonInterpreter 的 get 和 set 方法分别返回或设置解释器的局部名空间里的对象。为了预示

`getLocals` 和 `setLocals` 方法，这个名空间是一个在解释器里标志为 `locals` 的 `PyStringMap` 对象，这是值得一提的。有 `set` 和 `get` 方法的对象在 Jython 和 Java 之间具有的双向流动性再次引入数据对象的类型转换。如果你在解释器里设置 `java.lang.Integer` 对象，它保持 `java.lang.Integer` 吗？简单的回答是：不，回答的理由在下面部分。下面部分的目的是给出了怎样在解释器里设置对象、怎样从解释器里获得对象和当遍历解释器的边界时阐述这样的对象的类型发生了什么。

### 13. set

`PythonInterpreter` 类有两个 `set` 方法，这两个 `set` 方法允许解释器实例把解释器里的标志符 (`name`) 绑定到解释器外面的对象源。这两个方法的特征符如下：

```
public void set(String name, Object value)
public void set(String name, PyObject value)
```

两个方法都为 `name` 接受一个 `java.lang.String` 对象（第一个参数），但它们的第二个参数不同。第二个参数是限定在特定 `name` 的实对象，如果这个对象是一个 Java Object（前面列出的方法的第一个），则解释器把这个对象转换成合适的 Jython 类型。如果这个对象是 `PyObject`，解释器则不做任何修改地把它直接放在名空间字典里。下面给出了怎样用 `set` 方法在嵌入的解释器里设置一些简单的对象：

```
interp.set("S", "String literals becomes a PyStrings in the interp");
interp.set("V", java.util.Vector()); // V becomes a PyInstance type
```

`String` 和 `Java` 实例是容易的，但基本的类型是什么呢？`Java int` 不是 `java.lang.Object`。下面两行都是错误：

```
interp.set("I", 1);
interp.set("C", 'c');
```

这些类型需要使用它们的 `java.lang` 对应物，使用 `java.lang` 类修改前面两行如下所示：

```
interp.set("I", new Integer(1));
interp.set("C", new Character('c'));
```

`set` 方法的逻辑部分就是它把 Java 对象转换成适当的 `PyObjects`。在第 2 章“运算符、类型和内置函数”的“Java 类型”部分描述了 Jython 怎样转换某一种类型的对象。在这部分的前面对字符串设置一个变量的例子注意了这种类型转换。当在解释器里设置时字符串对象转变为 `PyString`。按照 Jython 的类型转换规则，`java.lang.Integer` 转换成 `PyInteger`，`java.lang.Long` 转换成 `PyLong` 等等。

### 14. get

`PythonInterpreter` 的 `get` 方法从解释器里检索对象。`get` 方法有两个方法特征符，它们如下：

```
public PyObject get(String name)
public Object get(String name, Class javaclass)
```

`name`参数在解释器的名空间映射对象（`locals`）里指定一个键，并且`get`方法的返回值是与指定的键相关的值。第一个`get`方法仅需要`name`参数，返回一个`PyObject`。然而，第二个`get`方法还需要另外一个参数来指定返回对象的类。

在解释器的`set`方法里的Java对象的自动类型转换与在`get`方法里的不是互逆的。为了用`get`方法检索Java对象而不是`PyObject`，你必须指定你希望返回的类，并把它转换成所希望的对象。为了用`get`方法检索`java.lang.String`对象，你将使用这些：

```
interp.set("myStringObject", "A string");

String s = (String)interp.get("myStringObject", String.class);
```

返回值是一个对象，如果你正在把结果赋给一个不通用的类型，则把这个结果类型转换。也要注意第二个参数必须是类；描述类的字符串是不够的。你可使用以前给出的`Classname.class`或`Objectname.getClass()`：

```
interp.set("myStringObject", "A string");
String s = (String)interp.get("myStringObject", String.class);
```

程序清单9-8示范了`set`和`get`两个方法。列表所做的事是创建一些Java对象，使用`set`来把它们放到解释器，然后用`get`在Java里检索出对象。每一步打印出对象的类——在Java里、在Jython里、再回到Java里，以便产生一个详细说明解释器使用什么样的自动类型转换表。没有任何命令行自变量，`TypeTest`类使用通常的、单自变量`get`方法。当用户指定命令行选项`symmetrical`时，`TypeTest`类使用两个参数`get`方法来转换每一个对象成它的初始类。

#### 程序清单9-8 类型转换器

---

```
// file: TypeTest.java
package org.python.demo;

import org.python.demo.Embedding;
import org.python.core.*;
import java.util.Vector;

public class TypeTest extends Embedding {
    private boolean simple = true;

    public TypeTest() { ; }

    public static void main(String[] args) {
        TypeTest tt = new TypeTest();
        tt.initInterpreter(args);
        tt.test();
    }

    public void test() {
        String codeString;
        Object[] jTypes = {"A string", new Short("1"), new Integer(3),
            new Long(10), new Float(3.14), new Double(299792.458),
            new Boolean(true), new int[] {1,2,3,4,5}, new Vector(),
            new PyInteger(1), new Character('c')};
    }
}
```

```

interp.exec("print 'In Java'.ljust(20), " +
            "'In Jython'.ljust(20), " +
            "'Back in Java'");
interp.exec("print '-----'.ljust(20), " +
            "-----".ljust(20), " +
            "-----");

// get first command-line argument: argv[0]
PyObject argv = Py.getSystemState().argv;
if (argv.__len__() > 0) {
    String option = argv.__getitem__(0).toString();
    if (option.compareTo("symmetrical") == 0) simple = false;
}

for (int i=0; i < jTypes.length; i++) {
    showConversion(jTypes[i]);
}
}

public void showConversion(Object o) {
    interp.set("testObject", o);
    interp.set("o", o.getClass().toString());
    String newClass = null;

    if (simple) {
        newClass = interp.get("testObject").getClass().toString();
    } else {

        newClass = interp.get("testObject",
                              o.getClass().getClass().toString());
    }
    interp.set("n", newClass); // n for newJavaClass
    interp.exec("pyClass = str(testObject.__class__) \n" +
               "print o[o.rfind('.') + 1: ].ljust(20), " +
               "pyClass[pyClass.rfind('.') + 1: ].ljust(20), " +
               "n[n.rfind('.') + 1:]");
}
}

```

用下列命令编译程序清单 9-8 里的 TypeTest 类：

```
javac -classpath c:\path\to\jython.jar;. org\python\demo\TypeTest.java
```

在没有命令行自变量的情况下运行 TypeTest 类产生下列各项：

```

dos>java -classpath "c:\jython-2.1\jython.jar";. org.python.demo.TypeTest
In Java          In Jython          Back in Java
-----
String           PyString          PyString
Short            PyInteger         PyInteger
Integer          PyInteger         PyInteger
Long             PyLong            PyLong
Float            PyFloat           PyFloat

```

Double	PyFloat	PyFloat
Boolean	PyInteger	PyInteger
class [I	PyArray	PyArray
Vector	Vector	PyJavaInstance
PyInteger	PyInteger	PyInteger
Character	PyString	PyString

你可从结果看到解释器的自动转换是一个方式。当你设法把每一个类型转换回它的初始类时，增加 symmetrical 选项到命令将给出所发生的：

```
C:\WINDOWS\Desktop\ch9examples>java -classpath "c:\jython-2.1\jython.jar";.
org.python.demo.TypeTest symmetrical
In Java           In Jython          Back in Java
-----
String            PyString           String
Short             PyInteger          Short
Integer           PyInteger          Integer
Long              PyLong             Long
Float             PyFloat            Float
Double            PyFloat            Double
Boolean           PyInteger          Boolean
class [I          PyArray            class [I
Vector            Vector             Vector
PyInteger         PyInteger          PyInteger
Character         PyString           Character
```

也有另一个转换 Py \* 对象到 Java 对象的方法，该方法经常被使用，这就是 `_toJava_` 方法，它将出现在下一节。

### 15. `_toJava_`

Jython 类 (PyObject) 的实例有一个称为 `_toJava_` 的特殊方法。`_toJava_` 方法需要一个 Java 类作为参数，并返回强制的实例到请求的 Java 类。如果强制是不可能，则方法返回 Py.NoConversion 对象。下面行给出了把 PyString 转换到 java.lang.String 对象：

```
interp.set("A = 'A test string'");
PyObject po = interp.get("A");
String MyString = (String)po._toJava_(String.class);
```

上述的例子假设 `_toJava_` 是成功的。如果 `_toJava_` 方法把对象转换成 String 类的实例失败，则它将返回 Py.NoConversion 对象。因为 (String) 转换 Py.NoConversion 值将产生一个 ClassCastException。处理异常意思是或者在 try/excep 里包含转换或者在转换前对 Py.NoConversion 对象进行测试。程序清单 9-9 选择了对 Py.NoConversion 对象进行测试。这个程序清单里的 Convert 类是 Embedding 类的另一个派生类。这个例子里的 `test()` 方法遍查 PyInteger 对象的 getting、converting 和 casting，再进入 java.lang.Character。这个例子里的转换失败了；在转换成 Character 前对 Py.NoConversion 进行测试阻止了 ClassCastException。

### 程序清单 9-9 用 `_toJava_` 对 Py.NoConversion 进行测试

```
// file: Convert.java
package org.python.demo;
```

```

import org.python.demo.Embedding;
import org.python.core.*;

public class Convert extends Embedding {
    public Convert() { ; }

    public static void main(String[] args) {
        Convert c = new Convert();
        c.initInterpreter(args);
        c.test();
    }

    public void test() {
        // Set an identifier in the interpreter
        interp.exec("test = 'A'");

        // Get object out of interpreter as PyObject
        PyObject retrievedObject = interp.get("test");

        // Convert PyObject to instance of desired Class
        Object myObject = retrievedObject.__tojava__(Character.class);

        // See if conversion failed- meaning Py.NoConversion returned
        if (myObject == Py.NoConversion) {
            System.out.println("Unable to convert.");
        } else {
            Character myChar = (Character)myObject;
            System.out.println("The Character is: " + myChar);
        }
    }
}

```

用`__tojava__`方法转换用户定义的 Jython 类成 Java 对象允许用 Jython 写的类在某一个 Java 类需要的地方被使用。考虑一个使用 Java 类 A 的应用程序和你需要许多这个类的惟一形式，你可以用 Jython 写这些类作为 A 的派生类，当被某一个方法特征符需要时则把它们转换成类 A 的对象。程序清单 9-10、9-11 和 9-12 形成应用这样一个过程的三元文件。程序清单 9-10 里的 Report 类有意模拟报表生成工具。无可否认避免分散`__tojava__`聚焦的报表生成部分的配合留下了许多想象，但是这个模式对 Jython scripts 是一个有效的集合点。Report 类在命令行装载一个特定的 Jython script。程序清单 9-12 包含了一个处理特定报表细节的 Jython script。在没有重新编译 Report 类的情况下这允许多种报表格式存在。这样使用 scripts 的技巧让它们从 Java 类继承。程序清单 9-11 是一个简单的抽象 ReportSpec 类，该类在程序清单 9-12 里作为 Jython script 的一个合适的超类。

#### 程序清单 9-10 使用`__tojava__`作为报表生成器

```

// file: Report.java
package org.python.demo.reports;

import org.python.demo.Embedding;
import org.python.demo.reports.ReportSpec;
import java.io.*;
import org.python.core.*;

```

```

public class Report extends Embedding {

    public static void main(String[] args) {
        Report rpt = new Report();
        rpt.initInterpreter(args);
        try {
            rpt.generateReport();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void generateReport() throws FileNotFoundException, PyException {
        String fileName;
        ReportSpec rs = null;

        // Check #1- user supplied command-line arg
        PySystemState sys = Py.getSystemState();
        if (sys.argv._len_() == 0) {
            System.out.println("Missing filename.\n" +
                               "Usage: 'Report' filename");
            return;
        } else {
            fileName = sys.argv._getitem_(0).toString();
        }

        // Check #2- Command-line arg is in fact a *.py file
        if (new File(fileName).isFile() != true)
            throw new FileNotFoundException(fileName);
        if (fileName.endsWith(".py") != true)
            throw new PyException(
                new PyString(fileName + " is not a *.py file"));
        try {
            rs = getReportSpecInstance(fileName);
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
        rs.fillTitle();
        rs.fillHeadings();
        while (rs.fillRow()) {}
    }

    protected ReportSpec getReportSpecInstance(String fileName)
    throws InstantiationException
    {
        String className;

        // Exec the file
        interp.execfile(fileName);
        // Get the name of the file without path and extension.
        // This should be the name of the class within the file
        int start = fileName.lastIndexOf(File.separator);
        if (start < 0)
            start = 0;
        else
            start++;
    }
}

```

```

        className = fileName.substring(start, fileName.length() - 3);
        PyObject reportSpecClass = interp.get(className);
        if (reportSpecClass == null)
            throw new InstantiationException(
                "No ReportSpec Class named " + className +
                " exists in " + fileName);
        PyObject m_args = (PyObject) new PyInteger(70);
        PyObject reportSpecInstance = reportSpecClass.__call__(m_args);
        ReportSpec rs =
            (ReportSpec)reportSpecInstance.__tojava__(ReportSpec.class);
        if (rs == Py.NoConversion)
            throw new InstantiationException(
                "Unable to create a ReportSpec instance from " +
                className);
        return rs;
    }
}

```

程序清单 9-11 Jython Report Scripts 派生的 Java 类

```

// file: ReportSpec.java
package org.python.demo.reports;

public abstract class ReportSpec extends Object {
    public abstract String fillTitle();
    public abstract void fillHeadings();
    public abstract boolean fillRow();
}

```

程序清单 9-12 Jython Report 生成脚本

```

# file: ReportTest.py
from org.python.demo.reports import ReportSpec

class ReportTest(ReportSpec):
    def __init__(self, reportWidth):
        # @sig public ReportTest(int reportWidth)
        # plug in data (a database in a real implementation)
        self.width = reportWidth
    self.data = [ [1,2,3],
                 [4,5,6],
                 [7,8,9] ]
    self.pad = reportWidth/(len(self.data[0]) - 1)

def fillTitle(self):
    print "Test of Report Generator".center(self.width)

def fillHeadings(self):
    """Prints column headings."""
    # This would be database metadata in a real implementation
    for x in ["A", "B", "C"]:
        print x.ljust(self.pad - 1),
    print

def fillRow(self):
    if not self.data:
        return 0

```

---

```

row = self.data.pop()
for x in row:
    print str(x).ljust(self.pad + 1),
print
return 1

```

---

report script 必须继承的抽象 Java 类是程序清单 9-11 里的抽象 ReportSpec 类。Jython script 像程序清单 9-12 里的 ReportTest.py 类一样仅仅需要满足需要的超类的协议规范。注意是从解释器检索类，而不是从实例。在解释器里创建实例和获取它似乎是可能的；然而，在一个生存期长的进程里，减少解释器的名空间污染可能更容易些。注意实例的创建使用类对象的\_call\_() 方法。因为\_call\_方法对可调用的 Jython 对象的重要性，重新浏览第 7 章“高级类”。

用下列命令编译程序清单 9-10 里的 Report 类和程序清单 9-11 的 ReportSpec 类：

```
javac -classpath \path\to\jython.jar;.
org\python\demo\reports\Report.java
```

```
javac -classpath \path\to\jython.jar;.
org\python\demo\reports\ReportSpec.java
```

用程序清单 9-12 作为命令行自变量运行 Report 类产生以下各项：

```
>java -classpath "c:\jython-2.1\jython.jar";.
org.python.demo.reports.Report ReportTest.py
                                Test of Report Generator
      A           B           C
      7           8           9
      4           5           6
      1           2           3
```

#### 16. getLocals 和 setLocals 方法

Locals 意思是指解释器的局部名空间。在上面描述的 set 和 get 方法实际上是在局部映射对象里绑定了一个键或检索一个键的相关对象。setLocals 和 getLocals 方法反而允许你设置或检索整个名空间映射对象。有许多理由这样做，但显然的一个是对多 scripts 保持分隔的名空间。如果模块 A 和模块 B 在解释器里顺序执行，两个都定义了相同的模块全局变量，然后模块 B 重新定义那个变量。如果模块 A 又执行，则将可能得到一个不正确的结果。取出和恢复 locals 对象消除这种偶然的副作用的危险。

getLocals 和 setLocals 方法分别返回或需要一个像字典的 PyObject，这通常是一个 org.python.core.PyStringMap 对象。getLocals 和 setLocals 的方法特征符如下：

```

public PyObject getLocals()

public void setLocals(PyObject d)
```

程序清单 9-13 也是早先的 Embedding 例子的另一个派生类。在这个程序清单里的类设置了一个 test 值到在解释器里设置的 PyStringMap 对象。在解释器里，test 值被打印出来确认它是在那里，然后在解释器里设置另一个 test 值。getLocals 方法检索 locals 和打印它来确认两个标

志符都在它里面。

#### 程序清单 9-13 使用 setLocals 和 getLocals

```
// file: LocalsTest.java
package org.python.demo;
import org.python.demo.Embedding;
import org.python.core.*;
public class LocalsTest extends Embedding {
    public LocalsTest() { ; }
    public static void main(String[] args) {
        LocalsTest L = new LocalsTest();
        L.initInterpreter(args);
        L.test();
    }
    public void test() {
        PyStringMap locals = new PyStringMap();
        locals.__setitem__("Test1", new PyString("A test string"));
        interp.setLocals(locals);
        interp.exec("print Test1");
        interp.exec("Test2 = 'Another teststring'");
        PyObject dict = interp.getLocals();
        System.out.println(dict);
    }
}
Compile with javac -classpath c:\path\to\jython.jar;. org\python\demo\LocalsTest.java
```

运行程序清单 9-13 的输出如下：

```
dos>java -cp "c:\jython\jython.jar";. org.python.demo.LocalsTest
A test string
{'Test2': 'Another teststring', 'Test1': 'A test string'}
```

#### 17. imp 和顶层 Script

许多 Jython 模块利用顶层 script 环境来有条件地运行代码的 main 部分。如果这个描述没有想起来，相关的语句应该是非常熟悉的：

```
if __name__ == '__main__':
```

当 script 是 main script 时，script 应用这个来运行特定的代码。在 Jython 里 name\_ 标志符是自动的，但当嵌入 Jython 时，如果你需要 script 来执行与 `name_ == '__main__'` 条件相关的代码你必须显式地增加这个到解释器。程序清单 9-14 是 Embedding 类的另一个派生类，它执行在命令行指定的文件。然而这个例子增加了 `_main_` 以便 `name_ == '__main__'` 是真 (true)。

#### 程序清单 9-14 设置 `name_ == '__main__'`

```
// file: topLevelScript.java
package org.python.demo;
import org.python.demo.Embedding;
```

```

import java.io.*;
import org.python.core.*;

public class TopLevelScript extends Embedding {

    public static void main(String[] args) {
        TopLevelScript tls = new TopLevelScript();
        tls.initInterpreter(args);
        tls.test();
    }

    public void test() {
        PySystemState sys = Py.getSystemState();
        if (sys.argv._len_() == 0) {
            System.out.println("Missing filename.\n" +
                               "Usage: TopLevelScript filename");
            return;
        }
        String filename = sys.argv._getitem_(0).toString();

        // Set __name__
        PyModule mod = imp.addModule("__main__");
        interp.setLocals(mod.__dict__);

        // Using a file name with execfile
        interp.execfile(filename);
    }
}

Compile with javac -classpath \path\to\jython.jar
org\python\demo\TopLevelScript.java

```

你可用一个小的 Jython 模块像下面一样来测试\_\_name\_设置：

```

# file: mainname.py
if __name__ == '__main__':
    print 'It worked.'

```

运行这个的命令和输出如下：

```

dos> java -classpath \path\to\jython.jar;. org.python.demo.TopLevelScript
mainname.py
It worked.

```

通过用\_\_name\_键设置到\_\_main\_创建一个 PyStringMap 将获得同样的效果：

```

// Set __name__
PyStringMap dict = new PyStringMap();
dict.__setitem__("__name__", new PyString("__main__"));
interp.setLocals(dict);

```

因为 Jython 这样做和它引进 imp 类，所以程序清单 9-14 反而使用 imp.addModule 方法。用在程序清单 9-14 的 addModule 方法装载了一个模块，然后返回那个 PyModule 对象。它也把模块

名放在引入的模块的列表：`sys.modules`。关于 `imp` 类更多的细节将在后面的“扩展 Jython”一节讲述。

## 9.2 嵌入 InteractiveInterpreter

`InteractiveInterpreter` 也是 `PythonInterpreter` 的派生类。`InteractiveInterpreter` 为更高级的交互提供了 `runcode`、`runsource` 和 `interrupt` 方法。这个类的两个重要行为是异常捕获和语句完整性的管理。如果异常发生在 `runcode` 或 `runsource` 方法里，它在方法里被捕获以便它不是致命的，同时允许交互继续。解释器打印这样的异常以便用户知道发生了什么。另外，`runsource` 方法返回一个指明源字符串完整性的 Boolean 值。这有助于决定要打印哪个提示符和什么时候重置语句缓冲器。

表 9-1 给出了方法特征符和每一个 `InteractiveInterpreter` 方法的使用。

表 9-1 `InteractiveInterpreter` 方法

方法特征符	概要
<code>public InteractiveInterpreter ()</code>	这些是 <code>InteractiveInterpreter</code> 类的构造函数。第二个构造函数就像 <code>PythonInterpreter</code> 的一个自变量构造函数一样接受一个 <code>PyStringMap</code> 对象
<code>public InteractiveInterpreter (PyObject locals)</code>	
<code>public Boolean runsource (String source)</code>	<code>runsource</code> 方法设法编译和执行显示异常信息的代码，但不传播异常。返回值如下：
<code>public Boolean runsource (String source, String filename)</code>	<code>Success -&gt; false</code> <code>Exception occurred -&gt; false</code> <code>Incomplete statement -&gt; true</code>
<code>public Boolean runsource (String source, String filename, String symbol)</code>	当不完整时，真就是允许容易的搜集用户输入的循环控制。第二个和第三个参数是与内置 <code>compile</code> 方法需要的一样
<code>public void runcode (PyObject code)</code>	<code>runcode</code> 方法执行代码对象并显示任何出现在执行时发生的异常。注意异常仅被显示；它不传播
<code>public void showexception (PyException exc)</code>	写异常信息到 <code>sys.stderr</code> 。这主要为内部使用
<code>public void write (String data)</code>	写字符串到 <code>sys.stderr</code> 。这主要为内部使用
<code>public void interrupt (ThreadState ts)</code>	<code>interrupt</code> 方法暂停代码来用友元线程方式插入一个异常
<code>throws InterruptedException</code>	
<code>interrupt</code>	

下列循环影响 `runsource` 返回循环的值，直到用户输入一个语法上完整的语句：

```
while (interp.runsource(codeString)) {
    System.out.print(ps2);
    codeString += "\n" + terminal.readLine();
}
```

程序清单 9-15 示范了异常的特殊处理和 `runsource` 方法的返回值。注意 `initialize` 步骤仍然是需要的。当嵌入 `PythonInterpreter` 时，`initialize` 方法是被使用的同一个方法，因为它是 `Interac-`

tiveInterpreter 的超类。

#### 程序清单 9-15 嵌入 InteractiveInterpreter

```
// file: InteractiveEmbedding.java
package org.python.demo;

import org.python.demo.Embedding;
import org.python.util.InteractiveInterpreter;
import java.util.Properties;
import java.io.*;

public class InteractiveEmbedding extends Embedding {

    protected InteractiveInterpreter interp;

    public static void main(String[] args) {
        InteractiveEmbedding ie = new InteractiveEmbedding();
        ie.initInterpreter(args);
        ie.test();
        ie.interact();
    }

    public void initInterpreter(String[] argv) {
        // set Properties
        if (System.getProperty("python.home") == null)
            System.setProperty("python.home", "c:\\jython-2.1");

        // no postProps, all properties but python.home put in registry file
        InteractiveInterpreter.initialize(System.getProperties(), null, argv);

        //instantiate- note that it is AFTER initialize
        interp = new InteractiveInterpreter();
    }

    public void test() {
        interp.runsource("print 'this is a syntax error'");
        interp.runsource('print 'This is not''');
    }

    public void interact() {
        String ps1 = ">>>";
        String ps2 = "...";
        BufferedReader terminal = new BufferedReader(
            new InputStreamReader(System.in));
        interp.write("Enter \"exit\" to quit.");

        String codeString = "";
        interp.write('\n');
        while (true) {
            interp.write(ps1);
            try {
                codeString = terminal.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        }
        if (codeString.compareTo("exit")==0) System.exit(0);

        while (interp.runsource(codeString)) {
            interp.write(ps2);
            try {
                codeString += "\n" + terminal.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

程序清单 9-15 初始化解释器、运行快速的 test 方法，然后开始一个交互循环。test 方法有意地包含一个错误引用的字符串来产生一个 SyntaxError。这示范了 runsource 方法在没有终止解释器的情况下怎样报告错误。在 test 方法之后这个例子继续循环命令。这个循环命令打印提示符、读输入和反馈输入到解释器的 runsource 方法。如果语句是不完整的，runsource 方法返回真 (true)，这个返回值就是允许简洁的内部循环，该循环为复合语句收集输入直到语句是完整的。这个复合语句的内部循环也使用第二个提示符 (...) 来提示语句的继续。

用与下列相似的命令来编译程序清单 9-15：

```
javac -classpath \path\to\jython.jar;
org\python\demo\InteractiveEmbedding.java
```

例子使用 InteractiveEmbedding 类的输出如下。

记住在执行开始时的 SyntaxError 和 This is not 语句是 test 方法的一部分：

```

dos>java -cp \path\to\jython;. org.python.demo.InteractiveEmbedding
Traceback (innermost last):
  (no code object) at line 0
  File "<input>", line 2
SyntaxError: Lexical error at line 2, column 0.  Encountered: <EOF> after :
""

This is not
Enter "exit" to quit.
>>>print 'Hello World!'
Hello World!
>>>try:
...     assert 0, 'Assertion error for testing compound statements'
...except AssertionError:
...     import sys
...     e = sys.exc_info()
...     print "%s\n%s\n%s" % e
...
exceptions.AssertionError
Assertion error for testing compound statements
<traceback object at 2147462>
>>>exit

```

### 9.3 嵌入 InteractiveConsole

InteractiveConsole 增加了对嵌入多于一层的抽象，但是，这个 InteractiveConsole 对在 Jython 是普通的控制台交互是特定的，如在程序清单 9-13 使用。使用 InteractiveConsole 的三个方法创建控制台交互：interact、raw\_input 和 push。表 9-2 总结了这些方法。

表 9-2 InteractiveConsole 方法

方 法	概 要
public InteractiveConsole()	
public InteractiveConsole(PyObject locals)	InteractiveConsole 的三个构造函数考虑有选择地设置解释器的局部名空间和设置在错误消息里使用的文件名
public InteractiveConsole(PyObject locals, String filename)	
public void interact()	interact 方法仿真 Jython 解释器。可选的 banner 自变量是仅在第一个交互前打印的消息
public void interact(String banner)	
public boolean push(String line)	push 方法把一个没有用 \n 结束的单行推入解释器。这个方法就像 InteractiveInterpreter 的 runsource 方法一样返回真 (true) 或假 (false) —— 真意味着需要另外的输入
public String raw_input(PyObject prompt)	InteractiveConsole 的 raw_input 方法与内置的 raw_input 方法相同

程序清单 9-16 给出了创建一个嵌入的控制台是多么容易。Initialize 步骤像 interact 方法处理所有工作一样总是并更需要。

程序清单 9-16 嵌入 InteractiveConsole

```
// file: Console.java
package org.python.demo;

import org.python.util.InteractiveConsole;
import java.util.Properties;
import java.io.*;

public class Console {
    protected InteractiveConsole interp;

    public Console() {
        // set Properties
        if (System.getProperty("python.home") == null)
            System.setProperty("python.home", "c:\\\\jython-2.1");

        // no postProps, registry values used
        InteractiveConsole.initialize(System.getProperties(),
                                      null, new String[0] );
        interp = new InteractiveConsole();
    }

    public static void main(String[] args) {
        Console con = new Console();
```

```

        con.startConsole();
    }

    public void startConsole() {
        interp.interact("Welcome to your first embedded console");
    }
}

```

程序清单 9-16 初始化 InteractiveConsole，并调用它的 interact 方法。初始化是与以前一样的过程，但交互循环用 interact 方法被大大简化了。执行程序清单 9-16 产生了 Jython 的交互控制台，因此例子是不必要的。注意 InteractiveConsole 的 push 和 raw\_input 方法是可用的，但对这个列表是没有必要的。这些方法为交互而设置，就像 InteractiveInterpreter 的 runsource 方法，因此对执行临界情形是不理想的。如果你的应用程序需要与解释器进行交互，使用 PythonInterpreter 对象和它的 exec 和 eval 方法。

## 9.4 扩展 Jython

扩展 Jython 意思是用 Java 写 Jython 模块。Jython 模块在这里用作按特性像 Jython 模块一样运转的普通的 Java 类。这和仅写 Java 类有区别。Jython 能使用任何大多数的 Java 类，因此没有必要采取其他步骤来使 Java 类看起来和行动像 Jython 模块，然而有些情况下设计需要真正的 Jython 模块。当一个类允许像有序的和关键字参数这样的 Jython 机制时，就会出现差别。

程序清单 9-17 给出了一个简单的用 Java 写的 Jython 模块。你可看到这仅仅是一个 Java 类。然而，有趣的部分是 classDictInit 接口、相关的 classDictInit 方法、每一个方法上的 static 修饰符、\_\_doc\_\_ 字符串和 PyException。虽然 mymod 类非常简单，但它说明了用 Java 创建 Jython 模块的许多方面。

程序清单 9-17 用 Java 写的 Jython 模块

```

// file mymod.java
package org.python.demo.modules;
import org.python.core.*;

public class mymod implements ClassDictInit {
    public static void classDictInit(PyObject dict) {
        dict.__setitem__("__doc__",
                        new PyString("Test class to confirm " +
                                    "builtin module"));
        dict.__delitem__("classDictInit");
    }

    public static PyString __doc_fibTest =
        new PyString("fibTest(iteration) " +
                    "-> integer");

    public static int fibTest(PyObject[] args, String[] kw) {
        ArgParser ap = new ArgParser("fibTest", args, kw, "iteration");
        int iteration = ap.getInt(0);
        if (iteration < 1)
            throw new PyException(Py.ValueError,

```

```

        new PyString("Only integers >=1 allowed");
if (iteration == 1 || iteration == 2)
    return iteration;

return fibTest(new PyObject[] { new PyInteger(iteration-1) },
    new String[0]) +
    fibTest(new PyObject[] { new PyInteger(iteration-2) },
    new String[0]);
}
}

```

下面是在 Jython 的交互 shell 里运行 mymod 类例子的输出：

```

>>> from org.python.demo.modules import mymod
>>> import time
>>>
>>> def test(method, iteration):
...     t1 = time.time()
...     results = apply(method, (iteration,))
...     print "The 25th fibonacci iteration is: ", results
...     print "Time elapsed: ", time.time() - t1
...
>>> test(mymod.fibTest, 25)
The 25th fibonacci iteration is: 121393
Time elapsed: 0.8799999952316284

```

如果我们用 Jython 里的可比较的 fibonacci 函数来延伸这个例子，它阐明写 Java 模块的优点：性能。如果你传递实现 Jython 的灵活参数和仅使用 public static int fibTest (int iteration)，你可以在 fibTest 方法里将一些实例化减到最小。

```

>>> def fib(iteration):
...     if iteration < 1: raise ValueError, "Iteration must be >=1"
...     if iteration < 3: return iteration
...     return fib(iteration - 1) + fib(iteration - 2)
...
>>> test(fib, 25)
The 25th fibonacci iteration is: 121393
Time elapsed: 1.590000033378601

```

#### 9.4.1 ClassDictInit

如果被写作 Java 类的 Jython 模块实现了 ClassDictInit 接口，它可以控制模块的 `_dict_` 属性。实现了 ClassDictInit 接口的类必须有一个如下所示的方法：

```
public static void classDictInit(PyObject dict)
```

当类被初始化时 Jython 调用 classDictInit 方法，同时允许对 Jython 里属性名 `visible` 和它们的实现进行控制。程序清单 9-17 里的 mymod 类使用 classDictInit 来设置 `_doc_` 字符串并从 Jython 的名字 `visible` 删除它自己 classDictInit。实际上这是不必要的，因为 `_doc_` 字符串不能像下面一样定义：

```
public static String __doc__="Test class to confirm builtin module";
```

如果在 classDictInit 里没有 `_doc_` 字符串赋值，`classDictInit` 不必包含在类里，并且无论如何不出现在 Jython 里。当你有一个复杂的在多个类里实现的模块或类包含多个你需要从 Jython 隐含的属性时，真正的作用才开始。

因为 Jython 的简单性，在有希望的 Jython 里有另外一个途径控制 Java 方法的 visibility，但在写时是稍微有点试验性的。这包括异常 `org.python.core.PyIgnoreMethodTag`。异常永远不会真正被抛出，但声明它抛出这个异常的 Java 方法自动地从 Jython 视图移出。

#### 9.4.2 `_doc_` 字符串

通过包含命名为 `_doc_` 的 static `PyString` 成员，你能定义一个模块的 `doc` 字符串：

```
public static __doc__ = new PyString("Some documentation");
```

类里的 static 方法成为 Jython 里的模块函数。在程序清单 9-17，static `fibTest` 方法就像 Jython 模块里的 `fibTest` 函数一样运转。增加一个 `doc` 字符串到这个函数就意味着增加一个称作 `_doc_fibTest` 的 `PyString`。程序清单 9-15 使用它把文档分派给 `fibTest` 方法，而不能像 `mymod.fibTest._doc_` 一样从 `python` 检索到。

#### 9.4.3 异常

产生一个 Jython 异常需要用两个自变量抛出 `PyException` 类：实际 Jython 异常和异常消息（作为在 Jython 里的异常自变量被提到的）。Jython 的内置 exceptions 被定位在 `org.python.core.Py` 类。因此，`ValueError` 真正地是 `Py.valueError`。

```
raise ValueError, "Invalid Value"
```

上面的 Jython 语句用 Java 形式显示如下：

```
throw PyException(Py.ValueError, "Invalid Value")
```

程序清单 9-17 在解释器里使用 `PyException` 来产生 Jython `ValueError` 异常。

#### 9.4.4 参数

Jython 函数受益于多参数模式，该参数模式包含有序参数、关键字参数、缺省值和通配符。在 Java 里仿效这种行为是可能的。下列方法允许 Java 方法来处理位置和关键字自变量：

```
public PyObject MyFunction(PyObject[] args, String[] kw);
```

`args` 数组保存了所有自变量值，而 `kw` 数组仅保存特定的键值。下面是前面提到的 `MyFunction` 和例子调用：

```
MyFunction(1, 2, D=4, C=9)
```

`args` 和 `kw` 数组如下所示：

```
args = [1, 2, 4, 9]
kw = ["D", "C"]
```

类 org.python.core.ArgParser 使解释这些参数成更加有用的形式变得容易。ArgParser 有四个构造函数：

```
public ArgParser(String funcname, PyObject[] args,
                 String[] kws, String p0)

public ArgParser(String funcname, PyObject[] args,
                 String[] kws, String p0, String p1)

public ArgParser(String funcname, PyObject[] args,
                 String[] kws, String p0, String p1, String p2)

public ArgParser(String funcname, PyObject[] args,
                 String[] kws, String[] paramnames)
```

每个构造函数需要函数名作为第一个参数。使用 ArgsParser 的函数应该使用参数 PyObject [] args 和 String [] kw。这两个对象成为 ArgParser 构造函数的第二个和第三个自变量。剩余的参数是方法希望的自变量列表。如果有三个或更少些，这些可以是单独的 args，否则它们是一个String[]。程序清单 9-17 在 ArgParser 的帮助下给出了实现 Jython 自变量格式的 Java 方法。一些 Jython 方法和它们的 Java 方法加上 ArgParser 副本的例子依次来阐述：

```
Jython Method:
def test(A, B, C=2)

Java implementation:
public static PyObject test(PyObject[] args, String[] kws) {
    ArgParser ap = new ArgParser("test", args, kws, "A", "B", "C");
}

Jython Method:
def addContact(name, addr, ph=None)

Java implementation:
public static PyObject addContact(PyObject[] args, String[] kws) {
    ArgParser ap = new ArgParser("addContact", args, kws,
                                 new String[] {"name", "addr", "ph"});
}
```

参数值从具有一个它的 get \* 方法的 ArgParser 实例检索。这里列出的 get \* 方法有一个或两个参数。具有两个参数的这些 get \* 方法用缺省值检索参数。注意自变量的位置 (pos) 是从位置 0 开始的而不是位置 1。

```
public String getString(int pos)

public String getString(int pos, String def)

public int getInt(int pos)

public int getInt(int pos, int def)

public PyObject getPyObject(int pos)
```

```
public PyObject getPyObject(int pos, PyObject def)
public PyObject getList(int pos)
```

对 Jython 方法：

```
def test(A, B, C=2)
```

考虑缺省值的 Java 实现是这样：

```
public static PyObject test(PyObject[] args, String[] kws) {
    ArgParser ap = new ArgParser("test", args, kws, "A", "B", "C");
    int A = ap.getInt(0);
    // or...
    // String A = ap.getString(0);
    // PyObject A = ap.getPyObject(0);
    // PyTuple A = (PyTuple)ap.getList(0);

    String B = ap.getString(1);
    // or...
    // int B = ap.getInt(1);
    // PyObject B = ap.getPyObject(1);
    // PyObject B = ap.getList(1);

    // here's the two argument version to allow for defaults
    int C = ap.getInt(2, 2); // first 2=position, second 2=default value
}
```

#### 9.4.5 在 Java 里引入 Jython 模块

在 Java 里引入 Jython 模块通常使用 `_builtin_.__import__` 函数。`__import__` 函数有四个特征符：

```
public static PyObject __import__(String name)
public static PyObject __import__(String name, PyObject globals)
public static PyObject __import__(String name, PyObject globals,
                                PyObject locals)
public static PyObject __import__(String name, PyObject globals,
                                PyObject locals, PyObject fromlist)
```

从 Java 里引入 `random` 模块如下所示：

```
PyObject module = __builtin__.__import__(random);
```

#### 9.4.6 使用 PyObject

在 Java 里调用 Jython 类和用 Java 写 Jython 模块需要 Jython 特殊方法的扩展使用。特殊方法是那些以两个下划线开始和结束并在第 7 章描述的类。Jython 的灵活操作创建了从 Java 里调用 PyObjects 的特殊方法。

Jython的动态操作意思是属性的查找(finding)、获取(getting)、设置(setting)和调用(calling)通过提供了定制或扩展每一步机会的方法发生。对于getting、setting和calling第7章包含了特殊方法，但对于finding有另外的方法。这些方法是\_findattr\_和\_finditem\_。\_findattr\_方法返回对象属性，而\_finditem\_方法返回被键指定的序列或字典值。使用这些方法创建以下特征符：

```
public PyObject __finditem__(PyObject key)
public PyObject __finditem__(int index)
public PyObject __finditem__(String key)

public PyObject __findattr__(String name)
public PyObject __findattr__(PyObject name)
```

接受String对象作为参数的\_finditem\_和\_findattr\_方法都需要字符串对象被拘留。字符串文字自动被拘留，否则这个字符串必须被显式拘留。

如果你引入random模块并打算使用randint方法，你不能这样调用它的方法：

```
// This doesn't work
PyObject random = __builtin__.__import__("random");
random.randint(new PyInteger(10), new PyInteger(100));
```

相反，你应该结合call方法使用\_findattr\_方法

```
PyObject random = __builtin__.__import__("random");
random.__findattr__("randint").__call__(new PyInteger(10),
                                         new PyInteger(20));
```

简捷方法invoke有助于从Java调用关于PyObject的方法。调用关于各种Jython映射对象(如PyDictionary和PyStringMap)方法的正确的或通常的途径是用invoke方法。调用关于PyObject的方法的invoke方法等同于调用下列各项：

```
zyPyObject.__getattr__(name).__call__(args, keywords)
```

下面是invoke方法的不同方法特征符：

```
public PyObject invoke(String name)
public PyObject invoke(String name, PyObject arg1)
public PyObject invoke(String name, PyObject arg1, PyObject arg2)
public PyObject invoke(String name, PyObject[] args)
public PyObject invoke(String name, PyObject[] args, String[] keywords)
```

不同的特征符分别考虑不同的传递给调用方法的自变量集，相同的是第一个自变量是一个表示要调用的PyObject方法的名字。注意字符串文字是自动被拘留的。

下面例子给出了PyDictionary的创建和它使用invoke方法从Java里检索的键值。这个例子使用了PyDictionary的空构造函数，并接着用\_setitem\_特殊方法增加值：

```
PyDictionary dct = new PyDictionary();
dct.__setitem__(new PyString("G"), new PyInteger(1));
dct.__setitem__(new PyString("D"), new PyInteger(2));
dct.__setitem__(new PyString("A"), new PyInteger(3));
dct.__setitem__(new PyString("E"), new PyInteger(4));
```

```

dct.__setitem__(new PyString("B"), new PyInteger(5));
PyObject keys = dct.invoke("keys");
PyObject keys = dict.invoke("keys");

```

上面使用 PyDictionary 对象的 keys 方法的 invoke 方法来检索它的 keys。因为方法名 (keys) 是前面例子的字符串文字，它自动被拘留。

通过上述例子返回的 keys 进行循环也需要关注。因为 PyObject 的 \_len\_ 方法可能返回错误值，通过 Java 里 Jython 序列安全循环的唯一方法是对每一个索引值进行测试直到返回一个不存在的索引。如果我们通过这些 keys 循环延展上面的 dictionary keys 例子，通过它们循环的正确和通常的途径是如下被实现：

```

PyObject key;
for (int i = 0; (key = keys.__finditem__(i)) != null; i++) {
    System.out.println("K: " + key + ", V: " + dict.__getitem__(key));
}

```

#### 9.4.7 用 Java 写 Jython 类

当 Java 类仿效 Jython 模块时，那个模块里的函数常常被作为 static 成员被实现。在 Java 里仿效 Jython 类从而可能被作为 static 内部类实现；然而，在这点上 ClassDictInit 接口允许更多的灵活性。用 Java 仿效 Jython 类，最好派生在 org.python.core 包得到的最合适的 Py\* 类，并为所希望类型的类实现所有需要的特殊方法。用 Java 写的所有类可以派生 PyObject 和实现 \_findattr\_、\_setattr\_ 和 \_delattr\_ 方法。Mapping 对象将派生 PyDictionary 或 PyStringMap，并实现 \_finditem\_、\_setitem\_、\_delitem\_ 和相关的 mapping 方法。对 Jython 其他的数据类型同样是正确的。

#### 9.4.8 增加 Java 类作为内置 Jython 模块

当你用 Java 写 Jython 模块时，你也有选择来指定这个模块作为内置模块。注册键 python.modules.builtin 允许你把模块增加到用 Java 写的内置模块列表。python.modules.builtin 特性是一个用逗号分隔的人口列表。这个人口有三种形式。表 9-3 给出了模块的每种形式和它的使用说明。

表 9-3 内置模块入口语法和描述

入口语法	描述
name	Java 类的名字。假设类在 org.python.modules 包里。把这个入口加到 python.modules.builtin 列表允许你在 Jython 里使用 “import name”。如果你希望增加类 org.python.modules.jnios，这个入口将是这样： Python.modules.builtin = jnios
name: class	这种形式需要名字和完全限定的类，并用冒号分开。这个名称不必是类名；它仅仅是 Jython 用来引用这个类的名字。如果这个 name 复制预先存在的名字，则预先存在的模块被重定义。增加类 com.mycompany.python.agent 作为名字 mycompanyagent 使用下列： python.modules.builtin = mycompanyagent:org.mycompany.python.agent
name: null	从内置模块列表删除模块名。为了删除 os 模块，使用下列语句： python.modules.builtin = os: null

Python.modules.builtin 特性怎样被设置是独特的。目前用-D 命令行开关设置这个特性不增加模块到这个内置列表。它必须在注册文件里或在 initialize 方法 post-properties (第二个 arg) 里设置。

增加 mymod 作为模块，编译它，并确保它对它的包是在正确的目录树和 classpath 里。然后编辑注册文件，同时加入下列语句：

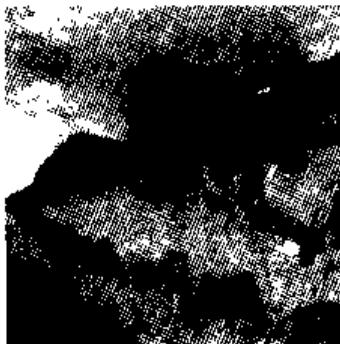
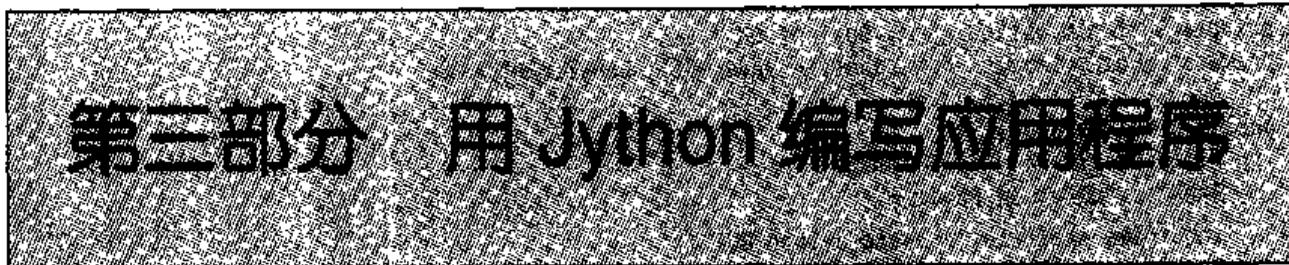
```
python.modules.builtin = "mymod:org.python.demo.modules.mymod"
```

name 或冒号左边部分可以是任何合法的标志符，但 class 或右边部分必须是一个完整的惟一标志合适的类的 package.class 字符串。一旦 Java 类作为一个内置被增加，仅用它简单的名字的引入就可在 Jython 里使用。前面例子的 mymod 可用下列语句使用：

```
>>> import mymod
```

对定义一个内置的相同机制允许你为一个存在的内置替换模块或删除内置。所有这些对容易用 Java 扩展和定制 Jython 作出了贡献。





## 第 10 章 GUI 开发

在 Jython 中开发图形用户接口 (GUI) 的主要工具是 Java 的 AWT (Abstract Windowing Toolkit) 和 Swing 类。考虑到 Jython 是用 Java 写的，这些工具在逻辑上将是 Jython 的首选，它们胜过在 Python 的语法中使用 Java 类。当然这并不意味着 AWT 和 Swing 是实现 GUIs 的惟一选择。现在有很多其他的工具可供 Java 当然也包括 Jython 使用。本章仅限于用 Jython 写的 AWT 和 Swing GUI。

本章并不包含 AWT 和 Swing 的基本知识。本书假设读者已具有 Java 的基础知识，故略去了基础知识部分。另外这些相关的知识也超过了本书的范围。幸运的是与 AWT 和 Swing 相关的书籍在各地书店的书架上都很容易找到。

你在本书中会看到利用 AWT 和 Swing 工具的 Jython 如何开发 GUI 应用的细节。Jython 有一些自动的 bean 属性、事件特性、常规的关键字参数和一些在 Jython pawt 包中的常用工具。

将 Java 的 GUI 例子直接翻译成 Jython 代码是理解 Jython 开发 GUI 的最快方法，所以本章从 Java 到 Jython 的翻译开始。Jython 对使用 AWT 和 Swing 开发 GUI 有两个很重要的改进使其开发更加简便快速：自动 bean 属性和 pawt 包。本章详细介绍了它们，并在本章余下的部分将详细介绍一些更大的例子来加深读者对用 Jython 开发 GUI 的理解。

### 10.1 比较 Java 和 Jython 的 GUI

在 Jython 中编写一个 GUI 与在 Java 中很类似。大部分 Java GUI 都能很容易翻译成为 Jython 的语言，并能运行正常。一旦熟悉了在 Jython 中使用 AWT 和 Swing 以及在 Jython 中 GUI 的原型，再将它翻译成为 Java（需要的话）就变得很有意义。程序清单 10-1 列出了一个用 Java 编写的简单的 GUI，程序清单 10-2 是用 Jython 编写的相同的 GUI。

程序清单 10-1 一个简单的 Java GUI

```
// file SimpleJavaGUI.java
import java.awt.*;
import java.awt.event.*;
```

```

class SimpleJavaGUI implements ActionListener {
    private Button mybutton = new Button("OK");
    private Label mylabel = new Label("A Java GUI", Label.CENTER);

    public SimpleJavaGUI() {
        Frame top_frame = new Frame();
        Panel panel = new Panel();
        top_frame.setTitle("A Basic Jython GUI");
        top_frame.setBackground(Color.yellow);

        //WindowListener needed for window close event
        top_frame.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
    }

    mybutton.addActionListener(this);
    panel.add(mylabel);
    panel.add(mybutton);
    top_frame.add(panel);

    // pack and show
    top_frame.pack();
    Dimension winSz = Toolkit.getDefaultToolkit().getScreenSize();
    top_frame.setLocation(winSz.width/2 - top_frame.getWidth()/2,
                          winSz.height/2 - top_frame.getHeight()/2);
    top_frame.setVisible(true);
}

public static void main(String[] args) {
    SimpleJavaGUI s = new SimpleJavaGUI();
}

public void actionPerformed(ActionEvent event) {
    System.exit(0);
}
}

```

运行程序清单 10-1 的类所产生的 GUI 如图 10-1 所示。

注意在程序清单 10-1 中的 AWT 类、bean 属性、bean 事件的使用。本例将使用 AWT 的 Frame、Panel、Button 和 Label 类。这些类使用类似 .setVisible () 方法来设置 bean 属性。另外这些类使用类似 .addWindowListener 方法来设置事件监听者。这些特性在与 Jython 比较时都是很重要的。

在 Jython 中实现同样的 GUI 需要在语法上做明显的改变。Java 的类型声明、访问修饰语、分号和括号在 Jython 中都不再存在。下面看一个在 Java 中 Button 组件的实例：

```

import java.awt.*;
private Button mybutton = new Button("OK");

```

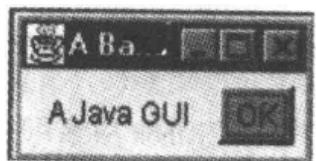


图 10-1 一个简单的 Java GUI

它在 Jython 中将变为：

```
from java import awt
mybutton = awt.Button("OK")
```

在 Jython 中实例化 button 使用了 awt 包的前缀。原因是在 Jython 中不主张使用 from java.awt import \* (我们从第 5 章“模块和包”中已知)，因此你最好使用 import 包或类的名字。使用 from java import awt 更为常见，但这需要在其后的 AWT 类的引用中使用包的名字，如 awt.Button。

Jython 与 Java 之间会做自动的类型转换，翻译程序清单 10-1 需要特别注意在 Jython 中所使用的对象类型。例如 setVisible 方法需要一个 Java 的布尔参数。Jython 将 PyInteger 的 1 转换成为布尔 true，而将 0 转换成为布尔 false。这意味着在 Jython 中 visible 属性必须设置为 1。能够显示 Jython 自动 bean 属性的有趣的一点是 Jython 不需要使用 setVisible 方法，它可直接设置 visible 的属性值为 1。

在 Jython 中不需要类。一个 Jython 类当然可以运行，而且可能是程序清单 10-1 的更精确的翻译，但在 Jython 中类是可选的。程序清单 10-2 并未用类，但实现了与程序清单 10-1 几乎一样的 GUI。如果已使用一个类，有一点需要特别指出，Jython 的方法需要在第一个参数位置有一个明显的 self 参数。前面几章已阐明此问题，但在将 Java 代码转换成为 Jython 代码时依然要注意几个其余的问题。程序清单 10-2 没有用一个类仅仅是为了表明如何在没有类的情况下实现一个 GUI；当然 AWT 和 Swing 类的特性使不使用类的复杂的 GUI 难以实现。

在程序清单 10-1 中的 window 事件和 button 事件都是基本 AWT 应用的简单实现；然而这些任务已开始形成著名的 Gordian 结。一个匿名的内部类实现了 WindowAdapter，故它能够处理 windowClosing 事件。SimpleJavaGUI 类自身也实现了 ActionListener 接口，故它能够在 actionPerformed

#### 程序清单 10-2 一个简单的 Jython GUI

---

```
# File: simpleJythonGUI.py
import java
from java import awt

# Make an exit function
def exit(e):
    java.lang.System.exit(0)

# Make root frame and the panel it will contain
top_frame = awt.Frame(title="A Basic Jython GUI",
                      background = awt.Color.yellow,
                      windowClosing=exit)

panel = awt.Panel()

# Add stuff to look at inside frame
mylabel = awt.Label("A Jython GUI", awt.Label.CENTER)
mybutton = awt.Button("OK", actionPerformed=exit)

panel.add(mylabel)
panel.add(mybutton)
top_frame.add(panel)
```

```
# pack and show
top_frame.pack()
# set location
toolkit = awt.Toolkit.getDefaultToolkit()
winSz = toolkit.screenSize
top_frame.setLocation(winSz.width/2 - top_frame.size.width/2,
                      winSz.height/2 - top_frame.height/2)
top_frame.visible = 1
```

方法中处理 button 事件。从程序清单 10-2 中我们注意到 WindowAdapter、addWindowListener 及 addActionListener 在 SimpleJythonGUI 中并不存在！相反，Jython 版本使用设置语句来处理这些特性和事件。这些设置语句是另一个 Jython 自动作为类属性增加 bean 属性和事件的简化工具。

图 10-2 为上述简单 GUI 的 Jython 实现。我们可看出由 Jython 实现的 GUI 除名字改变之外，其他的与图 10-1 看起来一样。

## 10.2 Bean 属性和事件

在描述 Jython 的自动 bean 属性之前先简要回顾一下 beans。bean 是指在一定的规范下有方法、特性以及事件的类。Bean 的方法是指所有的在 bean 中指定为公有的方法。bean 的特性是指通过 get 和 set 方法操作的数据对象。Bean 通过指定带有 add 和 remove 方法的监听者对象来定义事件。下列的类可读为“拥有只读特性名的 bean A”。

```
public class A {
    private String name;
    public String getName() {
        return name;
    }
}
```

虽然在上述 bean 中 Java 可直接使用 getName() 方法，但 Jython 增加了另外一个极有用快捷方式——不仅包括与上述的 name 特性一样的特性，还包括其他特性和方法。Jython 利用监听来知道 bean 的特性和事件，然后你就可以将这些 bean 的属性和事件作为实例属性来使用。Java 将用下列方法来检索上述 bean 的 name 属性：

```
import A;
bean = new A();
String name = bean.getName();
```

在 Jython 中，你可以将 name 特性简单地处理为一般的类属性，这样你就可以用下列方法来操作 name 属性：

```
>>> import A
>>> b = A()
>>> name = b.name
```

你不需要调用任何其他方法来检索 name 特性，因为 Jython 自动地将 bean 属性当作类属性来对待。在 Jython 中，你依然可以使用 getName() 的方法，但你不必一定如此，因为它的功能

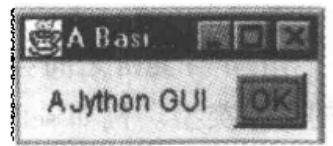


图 10-2 一个简单的 Jython GUI

已由 Jython 的自动 bean 属性所完成。setter 方法是同样的道理。Bean A 的一个 setName 方法能够翻译成下列的 Java 语句：

```
import A;
A bean = new A();
bean.setName("new name");
```

在 Jython 中，仅需要属性设置：

```
import A
bean = A()
bean.name = "new name"
```

虽然这些例子仅针对简单的特性，但事件的原理是相同的。该原理与 Jython 的关键字参数在很多方面简化了 GUI。下面程序清单 10-2 中的行在构造函数中使用 Jython 的关键字参数在组件中设置一对特性：

```
top_frame = awt.Frame(title="A Basic Jython GUI",
                      background = awt.Color.yellow,
                      windowClosing=exit)
mybutton = awt.Button("OK", actionPerformed=exit)
```

Jython 的自动 bean 属性和 bean 的事件属性如前所述非常快捷方便，但这并不意味着比在 Java 中使用这些方法低效。自动 bean 属性的使用在 Jython 语法中是十分合适的，尤其在与 GUI 同时运行时由于 beans 充满了 AWT 和 Swing 类而显得尤其有效。在 Jython 中使它们显得十分适当的是关键字参数。虽然仅在 Java 构造函数中运行，但使用关键字参数来设置 bean 属性依然十分好。下面是一段设置一个框架的 Java 伪码和 Jython 代码。注意关键字参数导致的区别。

```
// In Java (pseudo-code)
import java.awt.*;
Frame myframe = new Frame("A test Frame");
myframe.setBackground(Color.gray);
myframe.addWindowListener(new SomeWindowListenerClass());
myframe.setBounds(20, 20, 200, 300);

# In Jython
from java import awt
myframe = awt.Frame("A test Frame", background=awt.Color.gray,
                     bounds=(20, 20, 200, 300),
                     actionPerformed=someCallableObject)
```

虽然这并不总是节约空间、行数或打字，但由于通过关键字参数定义后其组件所有的良好的视觉相关特性和事件使其具有良好的可读性。

回到程序清单 10-1，我们可看到 Java 类如何通过明显地使用 get 和 set 方法来与 bean 属性作用。程序清单 10-1 通过下列方法使用：

```
Dimension winSz = Toolkit.getDefaultToolkit().getScreenSize();
top_frame.setLocation(winSz.width ... winSz.height);
top_frame.setTitle("A Basic Jython GUI");
top_frame.setBackground(Color.yellow);
top_frame.setVisible(true);
```

另外，Java 通过 add 和 remove 方法为 bean 指定事件。其中在程序清单 10-1 中，add 方法出现了两次：

```
top_frame.addWindowListener(new exitClass());
mybutton.addActionListener(this);
```

Jython 能使用与 Java 应用同样的方法。top\_frame.setBounds (...) 方法与上面所提的别的方法一样能在 Jython 中使用。然而 Jython 像常规 Jython 类属性一样增加了 bean 属性和事件。回到程序清单 10-2，我们可看到已经取代 bean 属性方法的属性赋值方法：

```
winSz = toolkit.screenSize
title="A Basic Jython GUI"
background = awt.Color.yellow
actionPerformed=exit
```

另外，程序清单 10-2 用指定一个 Jython 函数的简单属性赋值方法来注册事件处理器：

```
windowClosing=exit
actionPerformed=exit
```

转向 bean 属性的理由在于 AWT 和 Swing 类中突出的 bean 属性。只要看以下 `java.awt.Frame` 类就可看出 AWT 中的基本 bean 属性。Frame 类自身就有 13 个 bean 属性列于 jdk1.3 API 文档之中：

```
getAccessibleContext, getCursorType, getFrames, getIconImage,
getMenuBar, getState, getTitle, setCursor, setIconImage,
setMenuBar, setResizable, setState, setTitle.
```

Frame 类也从 `java.awt.Window` 类中继承了 10 个附加的访问器方法：

```
getFocusOwner, getGraphicsConfiguration, getInputContext, getListeners,
getLocale, getOwnedWindows, getOwner, getToolkit, getWarningString, setCursor
```

Frame 类还从 `java.awt.Container` 类中继承了 14 个访问器方法：

```
getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt,
getComponentCount, getComponents, getInsets, getLayout, getMaximumSize,
getMinimumSize, getPreferredSize, setFont, setLayout
```

该类还从 `java.awt.Component` 类中继承了 39 个访问器方法：

```
getBackground, getBounds, getBounds, getColorModel, getComponentOrientation,
getCursor, getDropTarget, getFont, getFontMetrics, getForeground,
getGraphics, getHeight, getInputMethodRequests, getLocation, getLocation,
getLocationOnScreen, getName, getParent, getPeer, getSize, getSize,
getTreeLock, getWidth, getX, getY, setBackground, setBounds, setBounds,
setComponentOrientation, setDropTarget, setEnabled, setForeground, setLocale,
 setLocation, setLocation, setName, setSize, setSize, setVisible
```

另外不要忘记在 `java.lang.Object` 中定义的最基本的 bean 属性：

```
getClass
```

一个简单的 Frame 容器就包括像 Jython 的类属性一样的很多 bean 属性。下面的交互式例子

显示了通过 Frame 对象的 bean 属性 Jython 能做什么。最令人感兴趣的是在实例化 frame 类时使用的关键字参数：

```
>>> import java, sys
>>> myframe = java.awt.Frame(title="Test", background=(220,220,4))
>>> myframe.bounds=(40, 30, 200, 200)
>>> myframe.visible=1
>>> myframe.background = 150, 240, 25 # Change color interactively
>>> # Try clicking on the window close icon here. it does nothing.
>>> myframe.windowClosing = lambda x: sys.exit(0)
>>> # now try closing the frame. it works.
```

Jython 的关键字参数使在组件构造函数中建立很多的 bean 属性和事件成为可能。即使忽略关键字参数，Frame 容器现在也不包含一个允许这样的参数的构造函数。相反，Jython 的很好的 bean 属性使在构造函数中建立如此多的 bean 状态成为可能。但更好的是 Jython 在正如前面交互例子所示的设置给 bounds 和 background 的元组参数下的行为。

### 10.2.1 Bean 属性

Bean 主要是一个习惯。一个叫 extent 的 bean 属性有相关的访问器和变异器方法 getExtent 和 setExtent。注意大小写的习惯。属性及其一对相关的方法都是第一个词小写而每一个其后的词的首字母大写。该习惯决定了在 Jython 中定义的 bean 属性的名字。如果一个 Java 类有一个方法叫 setAMultiWordName，然后 Jython 会增加一个自动 bean 属性名 aMultiWordName。

如果一个 bean 属性只定义了 get 方法，则该 bean 属性是只读的，而一个只有相关的 set 方法的属性是只写的。如果二者都定义了则产生了读-写属性。get 和 set 方法必须互相补充以实现完全的读写访问。这意味着如果一个 set 方法有一个字符串参数，则其相对应的 get 方法必须返回一个字符串对象。下面的 Java bean 错误地给普通的对象设置 setName 参数：

```
public class A {
    private String name = "";

    public String getName() {
        System.out.println("Getting name");
        return name;
    }

    public void setName(Object s) {
        name = (String)s;
        System.out.println("Setting name");
    }
}
```

使用前面的有 Jython 的自动 bean 属性的 bean 将会产生下面的输出：

```
>>> import A
>>> b = A()
>>> b.name
Getting name
 ''
>>> b.name = "this"
Traceback (innermost last):
  File "<console>", line 1, in ?
AttributeError: read-only attr: name
```

如果重写该类使其具有合适的互补参数，它将能与 Jython 的 bean 属性设置相容：

```
public class A {
    private String name = "";

    public String getName() {
        System.out.println("Getting name");
        return name;
    }

    public void setName(String s) {
        name = s;
        System.out.println("Setting name");
    }
}
```

对新的 bean 做一个快速的交互式的测试就可确认对 name 属性的合适的读写访问：

```
>>> import A
>>> b = A()
>>> b.name = "New name"
Setting name
>>> b.name
Getting name
'New name'
```

Bean 的属性也可是一个序列。如果你想拥有对一个序列或索引 bean 有完全的读写访问权限，它将会产生四个访问器。四个方法如下：

```
setProperty([])           // Sets the entire sequence
setProperty(int, Object) // Sets a specific index
getProperty()            // Gets the entire sequence
getProperty(int)         // Gets a specific index
```

然而 Jython 的自动 bean 属性仅使用这四个方法中的 getProperty() 和 setProperty() 方法。程序清单 10-3 是一个有一个叫 items 的索引属性的 Java bean。

### 程序清单 10-3 一个有索引属性的 Bean

---

```
// file: ItemBean.java
public class ItemBean {
    private String[] data = {"A", "B", "C"};

    public ItemBean() { ; }

    public String[] getItems() {
        return data;
    }

    public String getItems(int i) {
        return data[i];
    }

    public void setItems(String[] items) {
        data = items;
```

```

    }

    public void setItems(int index, String item) {
        data[index] = item;
    }
}

```

下面的屏幕交互会话显示了程序清单 10-3 中的 items 属性的作用：

```

>>> import ItemBean
>>> ib = ItemBean()
>>>
>>> # This calls ib.getItems()
>>> ib.items
array(['A', 'B', 'C'], java.lang.String)
>>>
>>> # The following does not call ib.getItems(int i), but instead
>>> # calls ib.getItems() and applies the index to the returned array
>>>
>>> # The following calls ib.setItems([])
>>> ib.items = ["1", "2", "3"]

```

### 10.2.2 Bean 属性和元组

Jython 自动地解释设置给 bean 属性的元组作为要设置的属性类的构造函数。设置给 background 属性的三个元素的元组自动地变成 Java.awt.Color 类的构造函数的值是因为 bean 属性类型为 java.awt.Color。设置给边界属性的元组自动地变成 Rectangle 类型。

这种自动的属性类型构建进一步简化了 Jython 的 GUI 代码。同时要注意到 Jython 的自动 bean 属性在与 Jython 的关键字参数结合后是最有效的。f.setVisible(1) 和 f.visible = 1 的区别可以忽略，但关键字参数通常会产生一个有价值的区别。

```

// In Java (pseudo-code)
import java.awt.*;
Frame f = new Frame();
Label L = new Label("This is a Label");
L.setBackground( new Color(50, 160, 40));
L.setForeground( new Color(50, 255, 50));
L.setFont(new Font("Helvetica Bold", 18, 24));
f.add(L);
f.setVisible(true);

# In Jython
from java import awt
f = awt.Frame()
L = awt.Label("This is a Label", background=(50, 160, 40),
               foreground=(50, 255, 50), font=("Helvetica Bold", 18, 24))
f.add(L)
f.visible = 1

```

### 10.2.3 Bean 事件

Bean 事件是那些由 addEventListener 和 removeEventListener 注册的事件。(Event 是事件实际类

型的占位符) 例如, `java.awt.Window` 类或 `Window bean`, 用 `addWindowListener` 方法注册 `Window` 事件。`addWindowListener` 方法需要一个将 `WindowListener` 接口作为一个参数实现的类。

如果你想在 Java GUI 中处理事件, 你必须增加一个实现那个事件类型的事件监听器的类。如 `windowClosing` 事件是在接口 `WindowListener` 中定义的。所以 Java 的应用首先必须使用 bean 事件 `addWindowListener` 来增加实现 `WindowListener` 接口的类或扩展一个有上述功能的类。然后该类将被要求实现 `windowClosing` 方法。Java 也允许一个无名的内部类来满足该接口, 然后处理事件。

Jython 当然也使用自动事件属性作为属性。这使得 Jython 能改变它的第一类函数。虽然使用第一类函数是它的通常实现方法, 但实际上它能与任何可调用的 Jython 对象一起工作。Jython 的自动 bean 事件属性可用来设置一个可调用的对象给一个事件的 bean 名。要实现 `windowClosing` 事件, 只要设置一个可调用的对象给容器的 `windowClosing` 属性。可调用的对象必须接受一个参数——事件。事件的作用不必是一个类, 也不需有一个特定的名字。换句话说, 事件名字的作用像 Jython 的类属性。正如下面交互式的解释器中的例子所示, 像 `windowClosing`、`windowOpened`、`windowActivated` 这样的事件是 Jython 中简单的可设置的属性。

```
>>> def wclose(e):
...     print "Processing the windowClosing Event"
...     e.getSource().dispose()
...
>>> def wopen(e):
...     print 'Processing the windowOpened Event'
...
>>> def wact(e):
...     print "Processing the windowActivated event"
...
>>> import java
>>> f = java.awt.Frame("Jython AWT Example")
>>> f.windowClosing = wclose
>>> f.windowOpened = wopen
>>> f.windowActivated = wact
>>> f.visible = 1
>>> Processing the windowActivated event
Processing the windowOpened Event      # click out of window here
Processing the windowActivated event    # click on window here
Processing the windowClosing Event      # click on window close box
```

所有的事件属性设置与关键字参数一样可包含在构造函数中。

```
f = java.awt.Frame("Jython AWT Example", windowOpened=wopen,
                    windowClosing=wclose, windowActivated=wact)
```

#### 10.2.4 名字优先权

Jython 支持实例方法、类的静态属性、bean 属性和 bean 事件属性。这在一个方法的 bean 属性或事件属性是同样的名字时会产生潜在的命名冲突。那么, 在多个属性有同样的名字时会发生什么? Jython 通过设置优先权解决了这个问题。最先给的是实例方法。一个方法的名字总是优先于其他与之冲突的属性名。在这个层次序列中, 类的静态属性是下一个。下面的 Jython 类定义了一个叫作 `windowClosing` 类的静态属性。这是一个类属性, 不是一个事件, 由于优先权

的原因，它将优先同样名字的事件：

```
>>> class f(java.awt.Frame):
...     windowClosing="This"
...
>>> myf = f()
>>> myf.windowClosing = lambda x: java.lang.System.exit(0)
>>> myf.setVisible(1) # try to close the window now
```

事件属性和 bean 属性在这个层次序列中按这个次序紧随其后。该次序如下：

- 1) Methods
- 2) Class-static attributes
- 3) Event Properties
- 4) Bean properties

### 10.3 pawt 包

Jython 的 pawt 包为在 Jython 中使用 AWT 和 Swing 增加了几个便利的模块。GridBag、colors、test 和 swing 都是 pawt 包所引入的功能。

#### 1. GridBag

pawt.GridBag 类是一个设计用来辅助 `java.awt.GridBagLayout` 和 `java.awt.GridBagConstraints` 的封装器。在 GridBag 类中有两个方法：add 和 addRow。Jython 支持大部分布局管理器如 BorderLayout、CardLayout、FlowLayout 和 GridLayout，其用法与 Java 中的用法相差无几。Jython 支持在容器的构造函数中布局管理器的快捷设置。下面的一部分程序显示了如何通过构造函数中的关键字来建立一个容器的布局管理器：

```
from java import awt
from java.lang import System

f = awt.Frame("Test", bounds=(100,100,200,200),
               layout=awtFlowLayout(),
               windowClosing=lambda e: System.exit(0)
)
```

`GridBagLayout` 与 Jython 中其他布局有所不同是因为它通过一个类在复杂的布局管理器中增加了便利。组件根据与它们相联的布局 `GridBagLayout` 的约束而显示，但同时与 `GridBagLayout` 和 `GridBagConstraints` 一起工作是棘手的。Bean 属性这时在解决这个棘手的问题时又发挥了作用，而 Jython 的 `pawt.GridBag` 类也能有所帮助。

要在 Java 容器中使用 `pawt.GridBag` 类就要将该容器的实例提交给 `pawt.GridBag` 的构造函数。下而一行创建了一个 `java.awt.Frame` 实例并在 `pawt.GridBag` 类中增加那个框架。一旦加上后，在框架就使用 `GridBagLayout`：

```
from java import awt
import pawt
top_frame = awt.Frame("GridBag frame")
bag = pawt.GridBag(top_frame)
```

pawt.GridBag 类的实例化可以有选择地为缺省约束接受关键字参数。下面的 pawt.GridBag 类的实例化将 fill 作为缺省值设置给 GridBagConstraints.VERTICAL。注意在引号中“VERTICAL”的使用与更冗长的 GridBagConstraints.VERTICAL 的区别：

```
bag = pawt.GridBag(top_frame, fill="VERTICAL")
```

pawt.GridBag 类用两个方法在使用 GridBagLayout 的容器中加入组件：add 和 addRow。用在 Java 中 GridBagConstraints 加入一个标签和一个 TextField 可参照下面的代码（伪码）：

```
import java.awt.*;

Panel pane = new Panel();
GridBagLayout bag = new GridBagLayout();
GridBagConstraints bagconstraints = new GridBagConstraints();

pane.setLayout(bag);
Label nameLabel = new Label("name: ");
bagconstraints.anchor = GridBagConstraints.WEST;
bagconstraints.fill = GridBagConstraints.NONE;
bagconstraints.gridwidth = 1;
pane.add(nameLabel, bagconstraints);

TextField nameField = new textField(25);
bagconstraints.anchor = GridBagConstraints.WEST;
bagconstraints.fill = GridBagConstraints.HORIZONTAL;
bagconstraints.gridwidth = 3;
pane.add(nameLabel, bagconstraints);
```

通过 Jython 的 pawt.GridBag 类，你可将其缩短为：（伪码）

```
from java import awt
from pawt import GridBag

pane = awt.Panel()
bag = GridBag(pane, fill='HORIZONTAL')
nameLabel = awt.Label("Name: ")
nameField = awt.TextField(25)
bag.add(nameLabel, anchor="WEST", fill="NONE", gridwidth=1)
bag.addRow(nameField, anchor="WEST", fill="HORIZONTAL", gridwidth=3)
```

add 方法增加一个有约束的特定组件，其参数由关键字参数指定。AddRow 方法同样，只是它需要完成一行，而其后的组件在下一行开始。

程序清单 10-4 使用 pawt.GridBag 模块来简化地址簿条目的设置。该程序清单只需要最小限度地使用关键字参数。使用 GridBagLayout 的容器是 pawt.GridBag 类的第一个参数。其后的是缺省的约束，在本例中 fill 约束缺省设置为“HORIZONTAL”。增加的类别选择组件是惟一使用关键字约束的组件。要设置 fill 约束为“NONE”，关键字参数 fill 将会设置为字符串“NONE”而不是 GridBagConstraints.NONE。该简写对所有的 GridBagConstraints 域都有效。这使得我们可用简单的关键字参数 anchor = “WEST” 来设置一个组件的 anchor 约束：

#### 程序清单 10-4 GridBagConstraints 的方便方法：pawt.GridBag

---

```
from java import awt
```

```
from java.awt.event import ActionListener
from pawt import GridBag
from java.lang import System
frame = awt.Frame("GridBag Test", visible=1, size=(200,200),
                  windowClosing=lambda e: System.exit(0))

pane=awt.Panel(background=(200,200,255))
frame.add(pane)

bag = GridBag(pane, fill='HORIZONTAL')

labelFactory = lambda x: awt.Label(x, background=awt.Color.yellow)

title = awt.Label("Jython GridBag Address Book", awt.Label.CENTER)
bag.addRow(title)

category = awt.Choice()
map(category.add, ["Family", "Friends", "Business"])
bag.add(labelFactory("Category: "))
bag.addRow(category, anchor="WEST", fill='NONE')

name = awt.TextField(25)
bag.add(labelFactory("Name: "))
bag.addRow(name)

address = awt.TextField(35)
bag.add(labelFactory("Address: "))
bag.addRow(address)

city = awt.TextField(25)
bag.add(labelFactory("City: "))
bag.addRow(city)

state = awt.TextField(2)
bag.add(labelFactory("State: "))
bag.addRow(state)

zip = awt.TextField(5)
bag.add(labelFactory("zip: "))
bag.addRow(zip)

homephone_areacode = awt.TextField(3)
homephone_prefix = awt.TextField(3)
homephone_suffix = awt.TextField(4)

cellphone_areacode = awt.TextField(3)
cellphone_prefix = awt.TextField(3)
cellphone_suffix = awt.TextField(4)

frame.pack()
```

执行程序清单 10-4 中的模块将会产生如图 10-3 的输出。

## 2. 颜色

在 `pawt` 包中的 `colors` 模块是一个很多有名字的颜色的集合。由于与数字相比，人们通常更容易记住名字，这些预定义的颜色加上名字仅仅是为了好记。例如，颜色 `papayawhip` 很容易记住，但 RGB 数 (255, 239, 213) 却难于记住。

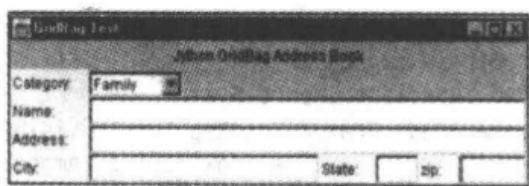


图 10-3 用 `pawtGridBag` 的地址簿条目

要获得一个在 `pawt.colors` 中定义的颜色的全部列表，检查文件 `sys.prefix/Lib/pawt/colors.py` 的内容或在 Jython 的交互式解释器中执行下列命令即可：

```
>>> import pawt
>>> dir(pawt.colors)
['__doc__', '__file__', '__name__', 'aliceblue', 'antiquewhite', 'aqua',
'aquamarine', 'azure', 'beige', 'bisque', 'black', 'blanchedalmond', 'blue',
'blueviolet', 'brown', 'burlywood',
...
'snow', 'springgreen', 'steelblue', 'tan', 'teal', 'thistle', 'tomato',
'turquoise', 'violet', 'wheat', 'white', 'whitesmoke', 'yellow',
'yellowgreen']
```

### 3. test

在 `pawt` 包中的 `test` 函数能对图形组件做下列简单的测试。如要测试一个按钮的颜色而不用 `Frame` 或 `Panel` 来查看，可使用 `pawt.test`：

```
>>> import java
>>> import pawt
>>> b = java.awt.Button("help", background=(212,144,100))
>>> pawt.test(b)
```

`test` 函数可选地接受一个范围参数。范围参数必须为一个长、宽为整数的元组或列表：

```
>>> from java.awt import Label
>>> import pawt
>>> L = Label("Test", background=(255,10,10), foreground=(10, 10, 100))
>>> pawt.test(L, (200,140))
```

`test` 函数返回使用的根框架使你可继续与 `test` 交互：

```
>>> from java.awt import Label
>>> import pawt
>>> L = Label("Test", background=(110,10,10), foreground=(10,110,100))
>>> f = pawt.test(L, (200,250))
>>> f.resize(100,140)
>>> L.font="Helvetica Bold", 20,24
>>> f.dispose()
>>> # note the prompt is still available for further tests this way
```

### 4. pawt.swing

`pawt.swing` 模块完成两件事：一是为你的 JDK 选择合适的 Swing 库；二是它有一个专用于 Swing 组件的 `test` 函数。

对于那些同时使用 JDK1.1 和 JDK1.2 的人，检索正确的 Swing 库是非常有用的。它需要减少时间，但它允许开发人员用 `java1.1` 和 `java 1.2` 来写 Jython Swing 代码而无须任何改变。

这个模块也加入了等价于 `pawt.test` 的测试函数，但使用了 Swing 部件：

```
>>> import pawt
>>> cb = pawt.swing.JCheckBox("Test Check Box")
>>> myframe = pawt.swing.test(cb)
>>> myframe.dispose()
>>> myframe = pawt.swing.test(cb, (100,200))
```

然而使用 `pawt.swing` 模块也有缺点。它是一个非常动态的模块，在编译 Jython 时产生问题。因为这一点，当你创建一个 swing 组件的派生类或将 Jython 应用编译成类文件时最好不要用 `pawt.swing` 模块，相反你应直接使用 `javax.swing`。

## 10.4 例子

本节包含很多说明在 Jython 中使用 AWT 和 Swing 的例子。

### 10.4.1 简单的 AWT 图形

程序清单 10-5 是一个简单的用 `java.awt.Canvas` 的 `paint` 方法绘制的标志图。`paint` 方法接

**程序清单 10-5 一个 Jython 标志图**

```
# file: jythonBanner.py
from java import awt
import java

class Banner(awt.Canvas):
    def paint(self, g):
        g.color = 204, 204, 204
        g.fillRect(15, 15, self.width-30, self.height-30)
        g.color = 102, 102, 153
        g.font = 'Helvetica Bold', awt.Font.BOLD, 28
        message = ">>> Jython"
        g.drawString(message,
                     self.width/2 - len(message)*10, #approx. center
                     self.height/2 + 14)           #same here

top_frame = awt.Frame('Jython Banner', size=(350, 150),
                      windowClosing=lambda e: java.lang.System.exit(0))
top_frame.add(Banner())
top_frame.visible = 1
```

收一个 `Graphics2D` 对象，程序清单用该对象打印 Jython 旗帜图。程序清单 10-5 对于画布的高度和宽度属性和图形的字体、颜色属性使用 Jython 的自动 bean 属性。它也使用 Jython 的自动 bean 属性来在 `Frame` 的构造函数中创建 `windowClosing` 事件。

运行 `jythonBanner.py` 脚本就可产生如图 10-4 的标志图。`paint` 方法在每次 `resize` 事件中调用。你可重设置标志图的大小来了解它是如何调节的。

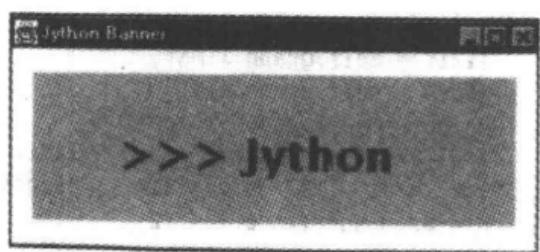


图 10-4 一个 Jython 标志图

### 10.4.2 增加事件

Jython 的事件，正如前面所提到的，是一个简单的属性设置。前面的例子通过 windowClosing 事件和一个按钮的 actionPerformed 事件展示了它。然而程序清单 10-6 展示了在 java.awt.event.MouseListener 和 java.awt.event.MouseMotionListener 接口中鼠标事件的使用。由于 Jython 的自动事件属性，对这些事件实现事件处理器正如一个属性赋值一样简单。

程序清单 10-6 是一个极坐标图，它显示了在鼠标遍历画布时鼠标位置的半径 (r) 和角度 (theta)。PolarCanvas 类是一个实现该图形的组件，它是 java.awt.Canvas 的派生类。创建 java.awt.Canvas 的派生类可使 PolarCanvas 实现画图的 paint 方法。该应用的目的是当鼠标在 PolarCanvas 组件上时以标签的方式显示坐标。这意味着 PolarCanvas 类是一个必须注册鼠标动作监听器的类。下面一行定义了一个 mouseMoved 事件处理器：

```
self.graph.mouseMoved = self.updateCoords
```

#### 程序清单 10-6 图形化极坐标

```
# file: PolarGraph.py
import java
from java.lang import System
from java import awt
from java.lang import Math

class Main(awt.Frame):
    def __init__(self):
        self.background=awt.Color.gray
        self.bounds=(50,50,400,400)
        self.windowClosing=lambda e: System.exit(0)
        self.r = awt.Label("r:      ")
        self.theta = awt.Label("theta:      ")
        infoPanel = awt.Panel()
        infoPanel.add(self.r)
        infoPanel.add(self.theta)
        self.graph = PolarCanvas()
        self.add("North", infoPanel)
        self.add("Center", self.graph)
        self.visible = 1
        self.graph.mouseMoved = self.updateCoords

    def updateCoords(self, e):
        limit = self.graph.limit
        width = self.graph.width
        height = self.graph.height
        x = (2 * e.x * limit)/width - limit
        y = limit - (2 * e.y * limit)/height
        r = Math.sqrt(x**2 + y**2)
        if x == 0:
            theta = 0
        else:
            theta = Math.atan(Math.abs(y)/Math.abs(x))
        if x < 0 and y > 0:
            theta = Math.PI - theta
```

```
elif x < 0 and y <= 0:
    theta = theta + Math.PI
elif x > 0 and y < 0:
    theta = 2 * Math.PI - theta
self.r.text = "r: %0.2f" % r
self.theta.text = "theta: %0.2f" % theta

class PolarCanvas.awt.Canvas):
    def __init__(self, interval=100, limit=400):
        self.background=awt.Color.white
        self.mouseReleased=self.onClick
        self.interval = interval
        self.limit = limit
        self.points = []

    def onClick(self, e):
        x = (2 * e.x * self.limit)/self.width - self.limit
        y = self.limit - (2 * e.y * self.limit)/self.height
        self.points.append(awt.Point(x, y))
        self.repaint()

    def paint(self, g):
        rings = self.limit/self.interval
        step = (self.width/(rings*2), self.height/(rings*2))
        # Draw rings

        for x in range(1, rings + 1):
            r = awt.Rectangle(x*step[0], x*step[1],
                               step[0] *(rings-x)*2,
                               step[1]* (rings-x)*2)
            lambda x, y: max(y - x * 20, 10)
            g.color = (max(140-x*20,10), max(200-x*20,10),
                       max(240-x*20,10))
            g.fillOval(r.x, r.y, r.width, r.height)
            g.color = awt.Color.black
            g.drawOval(r.x, r.y, r.width, r.height)
            g.drawString(str((rings*self.interval)-(x*self.interval)),
                         r.x - 8, self.height/2 + 12)

        # draw center dot
        g.fillOval(self.width/2-2, self.height/2-2, 4, 4)

        # draw points
        g.color = awt.Color.red
        for p in self.points:
            x = (p.x * self.width)/(2 * self.limit) + self.width/2
            y = self.height/2 + (p.y * self.height)/(2 * self.limit)
            g.fillOval(x, y, 4, 4)

if __name__ == '__main__':
    app = Main()
```

前面一行的设置意味着只要在鼠标移动到 PolarCanvas 组件上时，Main 类的 updateCoords 方法就会被调用。updateCoords 方法是在窗口顶部显示所期望坐标的方法。

除显示鼠标坐标以外，程序清单 10-6 还显示了通过释放鼠标按钮注册的点。注册一个显示的点的 mouseReleased 事件在 PolarCanvas 的构造函数中设置。mouseReleased 事件由 onClick 方法处理，它增加了一个显示的点并重绘该画布。

图 10-5 为运行 PolarGraph.py 模块的结果。

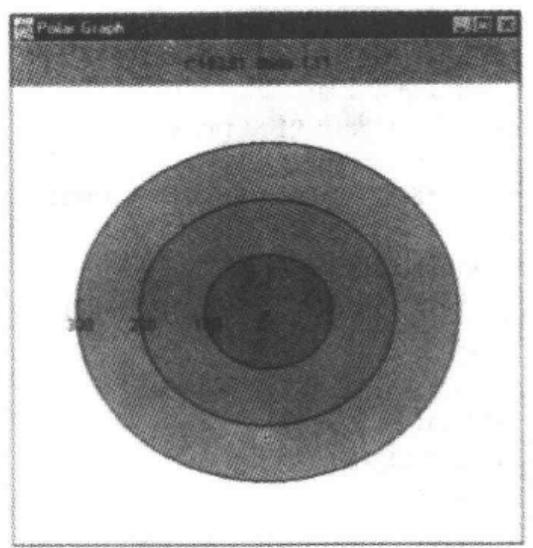


图 10-5 鼠标事件与极坐标图

#### 10.4.3 图像

虽然在 Jython 中显示图像只有一点革新，但与 Java 比较还是很有意义的。程序清单 10-7 显示了在 AWT 框架中的一个简单图像。程序清单 10-7 的实现与 Java 中的实现类似，尤其考虑到应用的顶框架竟然是一个 `java.awt.Frame` 类的派生类。这可通过使用组件的 `paint` 方法来实现，但通常由于更新和重画的方式导致不创建派生类就难以使用该方法。与 Java 的区别还在于在 `jyimage` 中出现的 `bean` 属性、事件属性和关键字设置。

##### 程序清单 10-7 显示图像

```
# file jyimage.py
from java.lang import System
from java import awt

class jyimage(awt.Frame):
    def __init__(self, im):
        self.im = awt.Toolkit.getDefaultToolkit().getImage(im)
        self.bounds=100,100,200,200
        self.title="Jython Image Display"
        self.windowClosing=lambda e: System.exit(0)

        mt = awt.MediaTracker(self)
        mt.addImage(self.im, 0)
        mt.waitForID(0)
        self.addNotify() # add peer to get insets
        i = self.insets
        self.resize(self.im.width + i.left + i.right,
                   self.im.height + i.top + i.bottom)

    def paint(self, g):
        g.drawImage(self.im, self.insets.left, self.insets.top, self)
```

```
if __name__ == '__main__':
    f = jyimage("jython-new-small.gif")
    f.visible = 1
```

程序清单 10-7 在 `jyimage` 的构造函数中显示了被一个名字指定的图像。在这个例子中被显示的图像是 Jython 的网站 (<http://www.jython.org>) 中的标志图。你能下载这个图像并重新生成与程序清单 10-7 相同的结果。程序清单 10-7 假设该图像在当前的工作目录中，因此你能毫无限制的使用这个文件名。

在程序清单 10-7 中有趣的是 `getDefauktToolkit()` 方法和 `MediaTracker` 对象。通常假设 `getDefauktToolkit()` 被翻译成可由 `awt.Toolkit.defaultToolkit` 访问的 `bean` 属性，这还不能保证其作为一个静态的类方法，故你必须使用完全的 `awt.Toolkit.getDefaultToolkit()` 调用。程序清单 10-7 使用缺省的 `toolkit` 来装载显示的图像。为保证整个图像在绘制前已装载，程序清单 10-7 使用了方法 `awt.MediaTracker`。这使定义 `imageUpdate` 和其他与图像增长式装载相关的问题变得不必要。Java 的应用需要在 `mt.waitForID()` 方法外有一个 `try/catch` 语句。在 Jython 中虽然这也是很好的习惯，但并非必要。

`AddNotify` 和插边在 Java 中是很一般的常识，但这儿有必要再重复一下。框架的 (0, 0) 坐标在标题条下面。这使得使用框架插边信息来计算整个框架的大小很有必要，只有在用 `addNotify()` 方法创建了一个等同物之后 `getInsets` 才是可用的。

图 10-6 显示了执行 `imagedisplay.py` 后的结果。



图 10-6 用 Jython 显示图像

#### 10.4.4 菜单和菜单事件

框架有一个菜单条属性，它指定了 `java.awt.MenuBar` 类的实例所对应的菜单条。`java.awt.MenuBar` 的实例包括了对应你想实现的每个菜单项的 `java.awt.MenuItem` 类的实例。在 Jython 中创建一个菜单条属性的步骤包括用 `MenuItem` 实例填充一个 `MenuBar` 实例，然后设置框架的 `menuBar` 的属性为 `MenuBar`。

实现菜单项的动作只需要给每个 `MenuItem` 的 `actionPerformed` 属性设置一个值。程序清单 10-8 显示了在 Jython 中加入菜单和与其相关的动作的容易性。在该程序清单中 Jython 的 `getattr` 函数的使用使菜单动作快速对应其方法。

#### 程序清单 10-8 AWT 菜单

```
# file: jythonmenus.py
from java import awt
from java.awt import event
from java.lang import System

menus = [
    ('File', ['New', 'Open', 'Save', 'Saveas', 'Close']),
    ('Edit', ['Copy', 'Cut', 'Paste']),
]
class MenuTest(awt.Frame):
```

```

def __init__(self):
    bar = awt.MenuBar()
    for menu, menuitems in menus:
        menu = awt.Menu(menu)
        for menuitem in menuitems:
            method = getattr(self, 'on%s' % menuitem)
            item = awt.MenuItem(menuitem, actionPerformed=method)
            menu.add(item)
        bar.add(menu)
    self.menuBar = bar
    self.windowClosing = lambda e: System.exit(0)
    self.eventLabel = awt.Label("Event: ")
    self.bounds = 100, 100, 200, 100
    self.add(self.eventLabel)

def onNew(self, e):
    self.eventLabel.text = "Event: onNew"

def onOpen(self, e):
    self.eventLabel.text = "Event: onOpen"

def onSave(self, e):
    self.eventLabel.text = "Event: onSave"

def onSaveas(self, e):
    self.eventLabel.text = "Event: onSaveas"

def onClose(self, e):
    self.eventLabel.text = "Event: onClose"
    System.exit(0)

def onCopy(self, e):
    self.eventLabel.text = "Event: onCopy"

def onCut(self, e):
    self.eventLabel.text = "Event: onCut"

def onPaste(self, e):
    self.eventLabel.text = "Event: onPaste"

f = MenuTest()
f.visible = 1

```

图 10-7 和图 10-8 显示了程序清单 10-8 的 jythonmenus 模块。图 10-7 显示了下拉菜单，图 10-8 显示了作为菜单选择结果的更新后的标签。

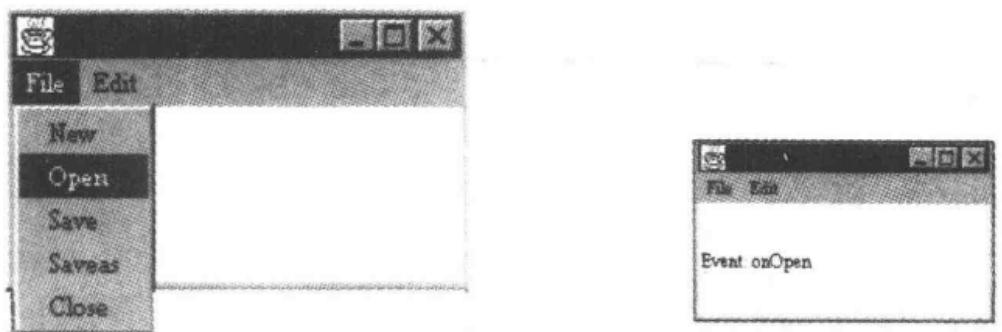


图 10-7 AWT 菜单和事件文件菜单

图 10-8 标签更新后的 AWT 菜单和事件

### 10.4.5 拖放

Jython 没有实现拖放的简便机制。正如在 Java 中一样，其实现包括实现所需要的 DnD 接口：DragGestureListener, DragSourceListener 和 DropTargetListener。然后实现这些接口的类建立了一个拖动源和放置目标。

建立一个拖放源需要两步：实例化 java.awt.dnd.DragSource 类，然后使用该类来创建拖动源识别器。上述两步的 Jython 代码如下：

```
from java.awt import dnd
myDragSource = dnd.DragSource()
myDragRecognizer = myDragSource.createDefaultDragGestureRecognizer(
    some_component,
    dnd.DnDConstants.ACTION_COPY_OR_MOVE,
    implementation_of_dnd.DragGestureListener)
```

创建识别器的三个参数分别为拖动源组件、动作 (action) 和拖动监听器。拖动源是任何你要拖动东西的组件。动作是一个指定对该拖动方式合适动作的值。动作的值在 java.awt.dnd.DnDConstants 类中定义。第三个参数为实现 java.awt.dnd.DragGestureListener 接口的类的实例。在程序清单 10-9 中的 JythonDnD 类为拖动源用一个 java.awt.List 实例化了方式识别器，为动作及其自身 (self) 作为 DragGestureListener 而拷贝或移动。

一旦建立了拖动源和拖动方式识别器，应用还需要一个放置拖动项目的地方。程序清单 10-9 使用了一个适当地扩展了 java.awt.List 的类叫 DropDownList。DnD 放下的部分要求一个实现 java.awt.dnd.DropTargetListener 的类，正如程序清单 10-9 中的 DropDownList 类。

程序清单 10-9 拖放程序清单

```
# file: ListDnD.py
from java import awt
from java.awt import dnd
from java.lang import System
from java.awt import datatransfer

class JythonDnD(awt.Frame, dnd.DragSourceListener, dnd.DragGestureListener):
    def __init__(self):
        self.title='Jython Drag -n- Drop Implementation'
        self.bounds = 100, 100, 300, 200
        self.windowClosing = lambda e: System.exit(0)

        self.draglist = awt.List()
        map(self.draglist.add, ["1","2","3","4","5"])
        self.dropList = DropDownList()
        self.dropTarget = dnd.DropTarget(self,
```

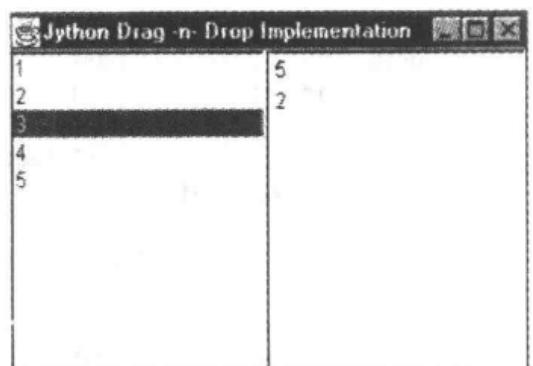


图 10-9 拖放例子的显示

```

dnd.DnDConstants.ACTION_COPY_OR_MOVE, self.droplist)

self.layout = awt.GridLayout(1, 2, 2, 2)
self.add(self.draglist)
self.add(self.droplist)
self.dragSource = dnd.DragSource()
self.recognize = self.dragSource.createDefaultDragGestureRecognizer(
    self.draglist,
    dnd.DnDConstants.ACTION_COPY_OR_MOVE,
    self)

def dragGestureRecognized(self, e):
    item = self.draglist.getSelectedItem()
    e.startDrag(self.dragSource.DefaultCopyDrop,
                datatransfer.StringSelection(item), self)

def dragEnter(self, e): pass
def dragOver(self, e): pass
def dragExit(self, e): pass
def dragDropEnd(self, e): pass
def dropActionChanged(self, e): pass

class droplist(awt.List, dnd.DropTargetListener):
    def __init__(self, datalist):
        map(self.add, datalist)
        self.dropTarget = dnd.DropTarget(self, 3, self)

    def drop(self, e):
        transfer = e.getTransferable()
        data = transfer.getTransferData(datatransfer.DataFlavor.stringFlavor)
        self.add(data)
        e.dropComplete(1)

    def dragEnter(self, e):
        e.acceptDrag(dnd.DnDConstants.ACTION_COPY_OR_MOVE)

    def dragExit(self, e): pass
    def dragOver(self, e): pass
    def dropActionChanged(self, e): pass

win = JythonOnD()
win.visible=1

```

程序清单 10-9 的结果为两列。左边的一列包括你能拖放到空的右边的一列的项目。图 10-9 显示了在几个项目已被拖放到右边后的 DnD 的例子所产生的 GUI。

#### 10.4.6 Swing

使用 Swing 类与使用 AWT 类没有什么太大的区别。Swing 是对 GUI 内部实现的重大改进。它提供了很丰富的组件来让开发者选择，但它并未改变图形应用构建的根本。通过与持续流行的 beans 结合，Jython 也能够对开发 Swing 应用同样有效。

在 Jython 中写一个 Swing 应用依然从其自动 bean 属性和事件属性中获益。通过与关键字参

## 程序清单 10-10 一个 Swing 树

```
# file: SwingTree.py
from javax import swing
from javax.swing import tree
import sys

top_frame = swing.JFrame("A Simple Tree in Jython and Swing",
                         windowClosing=lambda e: sys.exit(0),
                         background=(180,180,200),
                         )
data = tree.DefaultMutableTreeNode("Root")
data.add(tree.DefaultMutableTreeNode("a leaf"))
childNode = tree.DefaultMutableTreeNode("a node")
childNode.add(tree.DefaultMutableTreeNode("another leaf"))
data.add(childNode)
t = swing.JTree(data)

t.putClientProperty('JTree.lineStyle', 'Angled')
top_frame.contentPane.add(t)
top_frame.bounds = 100, 100, 200, 200
top_frame.visible = 1
```

数的组合，它们能很大地改进通过 Swing 组件的运用，正如其对 AWT 组件一样。

程序清单 10-10 为一个在 Swing 应用中简单的树的显示。程序清单 10-10 的 JFrame 和前面的 AWT Frame 的惟一的细微区别在于 JFrame 的 contentPane 的使用。除了这些细微的区别外，研究这个例子很大程度上加深对 AWT 和 Swing 工作一致性的理解。Jython 的作用是改进组件属性和事件，使组件具有自动 bean 属性和事件属性。

程序清单 10-10 产生的图形如图 10-10 所示。

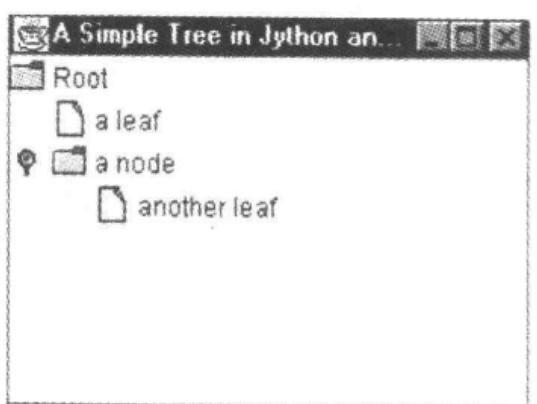


图 10-10 Jython 树的显示





## 第 11 章 数据库编程

本章详细介绍 Jython 的数据库使用。Jython 能够使用 Java 所使用的所有数据库工具，但本章重点讲述 DBM 数据库文件，MySQL 数据库服务器和 PostgreSQL 面向对象的数据库管理系统（ORDBMS）。本章也包括 Jython 对象的序列化与持久。本章的主题为：DBM、序列化、持久、MySQL、PostgreSQL、JDBC API、zxJDBC API 与事物处理。

### 11.1 DBM 文件

DBM 文件为杂凑型的与字典化对象相似的数据库文件。但这种相似性由于目前的实现并未用到所有的字典化方法以及所有的 DBM 关键字和值必须为字符串等原因而受到限制。各种 DBM 工具都能在 Jython 中运行，但目前只有一个与 Jython 绑定：dumbdbm。这个名字不是很吸引人并且其实现简单、缓慢，但它却是一个合适的有效的 DBM 的克隆。在将来，其他的 DBM 模块也将能在 Jython 中使用。

下面一个简单的使用 dumbdbm 的交互式的例子表明了使用 dumbdbm 是如何需要调用模块的 open() 方法的，然后你可以像一个字典那样有限制地使用打开的对象。

```
>>> import dumbdbm
>>> dbm = dumbdbm.open("dbmtest")
>>> dbm['Value 1'] = 'A'
>>> dbm['Value 2'] = 'B'
>>> dbm['Value 3'] = 'B'
>>> for key in dbm.keys():
...     print key, ":", dbm[key]
...
Value 3 : B
Value 2 : B
Value 1 : A
>>>
>>>dir(dbm.__class__)
['__delitem__', '__doc__', '__getitem__', '__init__', '__len__',
 '__module__', '__setitem__', '__addkey__', '__addval__', '__commit__', '__setval__',
 '__update__', 'close', 'has_key', 'keys']
>>>
>>> dbm.close()
```

dir (dbm.\_\_class\_\_) 方法显示了 dbm 对象与字典的不同。在字典对象中，只有 keys 和 has\_key 实现了。字典对象所需的最小特殊方法（\_\_setitem\_\_，\_\_delitem\_\_，\_\_getitem\_\_，\_\_and\_len\_\_）出现了。由于 DBM 对象的行为像字典，这儿不需要解释其使用。虽然它缺少一些字典方法，但它有一个明显的优点：它是持久的。重打开 dbmtest 编目可看到前一次输入的值。

```
>>> import dumbdbm
>>> dbm = dumbdbm.open("dbmtest")
>>> for key in dbm.keys():
...     print key, ":", dbm[key]
...
Value 3 : B
Value 2 : B
Value 1 : A
>>> dbm.close()
```

打开一个新的 `dumbdbm` 编目实际上创建了两个文件：一个以 `.dir` 作后缀，另一个以 `.dat` 作后缀。打开函数时不需要扩展名，只需要目录名。其他 DBM 实现允许在打开函数中有标志和模式参数，但 `dumbdbm` 忽略了这些参数。

## 11.2 序列化

序列化使一个对象变成一个便于传输与存储的流。存储部分保证包含数据库的内容，因为那是序列化对象常驻之地。要序列化一个 Jython 对象，可采用 `marshal`、`pickle`、`cPickle` 模块。`marshal` 模块只针对内部对象，而 `pickle` 模块可针对内部对象、模块全局类和函数以及大部分的实例对象。`cPickle` 模块是一个高性能的 `pickle` 模块的 Java 版本。它的名字是从 CPython 的 `cPickle` 模块中来，但这并不意味着它是用 C 所写。`shelve` 模块将 `pickle` 模块和数据库组合起来以创建持久对象的存储。Jython 的 `PyObjectInputStream` 是一个在反序列化从 Java 类继承的 Jython 对象时用来分解类的 `ObjectInputStream`。

### 11.2.1 marshal 模块

`marshal` 模块序列化代码对象和内部数据对象。它总是出现在下一节所讲的 `cPickle` 模块之前，但它仍然出现在很多项目之中。注意人们为 CPython 创造 `marshal` 模块是由于其编译代码对象的能力。然而目前这在 Jython 中不再有效，因而 `marshal` 模块很少有人用。

`marshal` 模块有四个方法：`dump`、`dumps`、`load` 和 `loads`。方法 `dump` 将对象序列化到文件，而方法 `load` 从文件中恢复对象。所以 `dump` 的参数是要序列化的对象和数据对象，而 `load` 的参数仅为文件对象。函数 `dump` 和 `load` 所用的文件对象必须是已经以二进制方式打开的。方法 `dumps` 将对象序列化到字符串，而方法 `loads` 从字符串中恢复对象。

下面交互式的例子说明了用列表对象使用 `marshal`。记住 `marshal` 模块并不序列化任何对象，它只序列化普通的内部对象。在目前的实现中，如果 `marshal` 不能序列化一个对象，就会产生一个 `KeyError` 错误。

```
>>> import marshal
>>> file = open("myMarshallledData.dat", "w+b") # Must be Binary mode
>>> L = range(30, 50, 7)
>>> marshal.dump(L, file) # serialize object L to 'file'
>>>
>>> file.flush() # flush or close after the dump to ensure integrity
>>> file.seek(0) # put file back at beginning for load()
>>>
>>> restoredL = marshal.load(file)
>>> print L
[30, 37, 44]
```

```
>>> print restoredL
[30, 37, 44]
```

### 11.2.2 pickle 和 cPickle 模块

pickle 和 cPickle 模块是序列化 Python 对象的更好的方法。pickle 和 cPickle 的区别仅在于其实现和性能上，二者在使用上并未有何不同。大多数对象，除了 Java 对象、代码对象和有复杂的\_getattr\_ 和\_setattr\_ 方法的 Python 对象之外，都能通过 pickle 和 cPickle 模块序列化。注意本文下面对 pickle 模块的引用既指 pickle 模块，也指 cPickle 模块，但所有的例子均指 cPickle 模块是由于其性能的优势。

pickle 模块定义了表 11-1 所示的 4 个函数。dump 和 dumps 函数序列化对象，而 load 和 loads 函数反序列化对象。

表 11-1 Pickle 方法

方 法	描 述
Dump (object, file [, bin])	序列化一个内部对象到前面打开的文件对象中。如果该对象不能被序列化，则产生一个 PicklingError 异常。Pickle 能使用文本或二进制的格式。第三个参数为 0 或没有第三个参数意味着文本格式，非 0 的第三个参数表示二进制方式
Load (file)	从前面打开的文件对象中读取或反序列化数据。如果 pickled 的数据是二进制模式的，提供给 load 的文件对象也必须是二进制模式
dumps (object [, bin])	序列化一个对象到字符串对象而不是文件中。如果该对象不能被序列化，产生一个 PicklingError 异常。第一个参数为 0 或没有第二个参数意味着文本格式，非 0 的第二个参数表示二进制方式
loads (string)	反序列化一个字符串到数据对象中

pickle 模块在遇到不能被 pickled 的对象时也定义一个 PicklingError 异常。

下面是一个简单的使用 pickle 来存储一个叫 product 类的实例的交互式的例子。该例子指定文件对象为二进制模式，并通过给 cPickle.dump 增加一个非零的第三个参数来指定二进制 pickle 模式：

```
>>> import cPickle
>>> class product:
...     def __init__(self, productCode, data):
...         self.__dict__.update(data)
...         self.productCode = productCode
...
>>> data = {'name': 'widget', 'price': 112.95, 'inventory': 1000}
>>> widget = product('1123', data)
>>> f = open(widget.productCode, "wb")
>>> cPickle.dump(widget, f, 1)
>>> f.close()
```

现在 widget product 存储在文件名为 1123 (product 的 productCode) 的文件之中。恢复实例使用 cPickle.load 函数，但为了正确地重新创建实例，其类必须存在于 unpickled 的名字空间之中。假设目前的例子是一个重新创建 widget product 的新的会话。Product 类必须重定义来保证其对实

例重新创建的有效性。由于原始的文件是以二进制方式打开的，所以其后的对文件的访问必须指定二进制模式。

```
>>> import cPickle
>>> class product:
...     def __init__(self, productCode, data):
...         self.__dict__.update(data)
...         self.productCode = productCode
...
>>> f = open("1123", "rb")
>>> widget = cPickle.load(f)
>>> f.close()
>>> vars(widget)
{'name': 'widget', 'productCode': '1123', 'price': '112.95', 'inventory':
1000}
```

一个对象能通过定义特殊的方法`_getstate_`和`_setstate_`控制其 pickling 和 unpickling。如果定义了`_getstate_`方法，则其返回值为在 pickled 对象时其序列化的对象。如果定义了`_setstate_`方法，在重新创建对象时通过反序列化的数据（`_getstate_`方法返回的）调用它。方法`_getstate_`和`_setstate_`是互补的，但不需要二者同时存在。如果未定义`_setstate_`方法，那方法`_getstate_`将返回一个字典。如果未定义方法`_getstate_`，则`_setstate_`在其装载时将接收到一个字典（实例`_dict_`）。

你也可以交替的使用`pickler`和`unpickler`对象。要创建这些对象，只要给类`pickle.Pickler`或`pickle.Unpickler`提供合适的文件对象即可。

```
>>> import cPickle
>>> f = open("picklertest", "w+b")
>>> p = cPickle.Pickler(f)
>>> u = cPickle.Unpickler(f)
>>>
>>> L = range(10)
>>> p.dump(L) # use pickler object to serialize
>>> f.seek(0)
>>> print u.load()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`pickle`能处理递归的数据结构、引用自身的对象和嵌套的数据结构，例如包含一个含有元组等的字典。`Pickler`对象存储了前面所有的 pickled 对象的引用，保证对同一对象的再引用时无需序列化，不管另外的引用是来自递归还是嵌套。重用一个`Pickler`对象允许共享的对象只需一次 pickled，`Pickler.dump()`只碰到那些前面 pickled 了的对象时要编写一个短的引用。

### 11.2.3 Shelves

Jython 的`shelve`模块综合了 DBM 编目的便利性和`pickle`方法的序列化来创建一个持久的对象存储。与`dumbdbm`模块类似，`shelve`的作用像字典，区别在于`shelve`允许任何`pickle`的对象作为值，即使其关键字依然为字符串。下面是一个使用`shelve`模块的交互式的例子。

```
>>> import shelve, dumbdbm
>>> dbm = dumbdbm.open('dbmdata')
>>> shelf = shelve.Shelf(dbm)
>>> shelf['a list'] = range(10)
>>> shelf['a dictionary'] = {1:1, 2:2, 3:3}
```

```
>>> print shelf.keys()
['a list', 'a dictionary']
>>> shelf.close()
```

你也可以使用 shelve 自身的 open 方法，正如下面所示，跳过 dumbdbm.open (...) 步骤：

```
>>> import shelve
>>> shelf = shelve.open("dbmdata")
```

#### 11.2.4 PythonObjectInputStream

在 Java 中序列化 Jython 对象与序列化 Java 对象一样，惟一的例外是：org.python.util.PythonObjectInputStream。一般情况下，Jython 对象的序列化与反序列化与 Java 对象一样，但在反序列化那些从 Java 类继承而来的 Jython 对象时，PythonObjectInputStream 能帮助分解类。这并不意味它只作用于那些从 Java 类继承而来的派生类。它能作用于任何 Jython 对象，但对那些从 Java 超类继承而来的 Jython 对象尤其有价值。

从 Java 中序列化 Jython 对象需要按下面伪码所写的步骤来实现：

```
// Import required classes
import java.io.*;
import org.python.core.*;

// Identify the resource
String file = "someFileName";

// To serialize, make an OutputStream (FileOutputStream in this case)
// and then an ObjectOutputStream.
FileOutputStream oStream = new FileOutputStream(file);
ObjectOutputStream objWriter = new ObjectOutputStream(oStream);

// Write a simple Jython object
objWriter.writeObject(new PyString("some string"));

// clean up
objWriter.flush();
oStream.close();
```

反序列化上述例子可以用 ObjectInputStream 或 PythonObjectInputStream。它需要按下面伪码所写的步骤来实现：

```
// Import required classes
import java.io.*;
import org.python.core.*;
import org.python.util.PythonObjectInputStream;

// Identify the resource
String file = "someFileName";

// Open an InputStream (FileInputStream in this case)
FileInputStream iStream = new FileInputStream(file);

// Use the input stream to open an ObjectInputStream or a
```

```

// PythonObjectInputStream.
PythonObjectInputStream objReader =
    new PythonObjectInputStream(iStream);

// It could be this for objects without java superclasses
// ObjectInputStream objReader = new ObjectInputStream(iStream);

// Read the object
PyObject po = (PyObject)objReader.readObject();

// clean up
iStream.close();

```

## 11.3 数据库管理系统

本节详细介绍如何使用 Jython 与 MySQL 和 PostgreSQL 数据库管理系统（DBMS）进行交互。Jython 支持任何有 Java 驱动器的数据库，然而选择这两个数据库的原因是它们非常流行、免费、非常稳定、已被普遍地配置以及有很多的文档。另外对新手，Paul Dubois 写了一本《MySQL》，Barry Stinson 写了一本《PostgreSQL Essential Reference》。这些书对研究这两种数据库系统都是很有用的资源，你能从 <http://www.newriders.com> 上获取更多的关于这些书的信息。本章所有的例子均是用其中某种数据库编写的，但大部分也适合于另外一种数据库，甚至只要微小的修改就可以运用到任何其他本书所未提到的数据库。唯一的例外是那些使用了 PostgreSQL 数据库系统有的而 MySQL 没有的特征的例子，显然那些例子将不能在没有这些基本特征（如事务）的数据库中运行。

MySQL 和 PostgreSQL 数据库管理系统的差别在于速度和特性。这对于比较复杂的数据系统而言可能过于简单化，但通常 MySQL 快-很快，而 PostgreSQL 数据库管理系统则有很高级的特性。MySQL 中数据的更新和选择操作更快，而 PostgreSQL 提供事务完整性、约束、规则、触发器和继承。MySQL 是一个基于关系型数据库的 SQL 数据库服务器（它以单个表的形式存储数据），而 PostgreSQL 是面向对象的数据库管理系统（ORDBMS）。MySQL 只使用列表数据，但在使用面向对象的语言时，有时会发现使对象适合于列表数据很困难。这是使用 ORDBMS 的原因，它在关系模型的基础上允许一些非常规数据的表示。

使用 PostgreSQL 和 MySQL 都有安全问题，因为它们都能支持网络连接。这些数据库系统的管理已超出本书的范围，但我们还是鼓励那些使用这两种数据库的新手去研究一下其安全性和管理文档。

### 11.3.1 MySQL

MySQL 是一个在大部分 UNIX 型平台和 Windows 平台上都能运行的 SQL 数据库服务器。为运行本书中针对 MySQL 的例子，你必须下载并安装 MySQL 和与其相关的 JDBC 驱动器，否则你必须改动例子使其适合于你想要用的数据库服务器。MySQL 可从 <http://www.mysql.com/> 的“downloads”区下载。MySQL JDBC 驱动器也在同一站点的同一下载区，但一些最新的驱动器和版本信息在站点 <http://maven.mysql.sourceforge.net/> 上。本书所用的 JDBC 驱动器的版本为 mm.mysql-

2.0.4-bin.jar。这实际上只是 MySQL 适用的驱动器中的一个，但它是目前 MySQL 适用的最常用的最好的 JDBC 驱动器。

安装指导显得有些多余，因为 MySQL 以 Windows 的 installer 或 \* nix 包的形式出现，只需要运行 installer 或包管理器即可。MySQL JDBC 不需要安装，只须将其加入 classpath 中即可。在安装完 MySQL 后，你必须增加一个用户和数据库。除非另外指出，本章的例子将使用的用户名为 jyuser，密码为 beans，数据库为 test。要启动服务器，可根据你的平台选择下面一种合适的命令行即可：

```
# *nix. "user" designates the name of the user to own the process
/pathToMysql/bin/safe_mysqld &

# windows
\pathToMysql\bin\winmysqladmin.exe
```

本章中所用的 test 数据库在 MySQL 安装时系统会自动创建，但如果由于某些原因，test 数据库不存在，你要创建一个。要创建一个数据库 test，先通过 MySQL 的客户端程序 mysql 的命令语言连接到服务器，然后输入 CREATE DATABASE 命令。客户端程序 mysql 位于 MySQL 安装的 bin 目录中。你必须以某个有适当特权的用户名来运行这个程序。注意 SQL 语句是以 ;， \g 或 \G 结束的。

```
C:\mysql\bin>mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.23.39

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.33 sec)
```

你可通过 SHOW DATABASES 来确认你所创建的数据库。

```
mysql> SHOW DATABASES\g
+-----+
| Database |
+-----+
| mysql    |
| test     |
+-----+
2 rows in set (0.00 sec)mysql>
```

下一步增加一个用户 jyuser。这个用户拥有所有对 test 数据库的权限。这可通过 GRANT 语句来实现。GRANT 语句有四个部分，分别为 what、where、who 和 how。下面是一个 GRANT 语句的语法：

```
GRANT what ON where TO who IDENTIFIED BY how
```

What 指明要准予的权限。我们必须对用户 jyuser 准予其 ALL 的权限。Where 的意思是权限设置在哪个数据库和表上——我们这儿是 test 数据库。Who 不仅是指用户名，还包括请求来源的源机器名。本例中的用户为 jyuser，而源位置在你用来连接数据库的机器中。下面的 GRANT

语句假设所有的 jyuser 连接都来源于 localhost。要加入新的源地址，只要使用百分号指明即可，如 192.168.1.%。甚至于只要用%就可表示任意地方。注意允许用户从任意主机上连接是麻烦的。How 是认证的方法，换句话说就是口令。本章使用口令 beans。一个完全满足本章需要的 GRANT 语句的例子为：

```
mysql> USE test;
mysql> GRANT ALL ON test TO jyuser@localhost IDENTIFIED BY "beans";
Query OK, 0 rows affected (0.05 sec)
```

本章中例子的设置是完整的。下一节讲述对 PostgreSQL 数据库同样的设置步骤，如果你只想用 MySQL 数据库，你可跳过这一节。

### 11.3.2 PostgreSQL

PostgreSQL 是一个先进的完全面向对象的 ORDBMS。PostgreSQL 在大部分 UNIX 型平台和 Windows2000/NT 平台下都能运行。本章中的例子必须使用 PostgreSQL 服务器和 PostgreSQL JDBC 驱动器，这可在 <http://www.postgresql.org/> 上下载。本章假定安装在 \*nix 系统上。由于 Windows 缺少很多 UNIX 型系统的功能，所以在 Windows 系统上安装 PostgreSQL 需要额外的包。这通常更为麻烦。要在 UNIX 型的系统上安装，先下载适合于你所使用的系统的包，然后运行包管理器。要在 Windows2000/NT 平台下安装 PostgreSQL，首先必须下载 Cygwin 和 cygipc 包来在 Windows 操作系统上模拟 UNIX 环境。Cygwin 包可在 <http://sources.redhat.com/cygwin/> 上下载，cygipc 包可在 <http://www.neuro.gatech.edu/users/cwilsou/cygutils/V1.1/cygipc/> 上下载。然后按在 PostgreSQL 发布的文件 doc/FAQ\_MSWIN 中的说明或可在 <http://www.ca.postgresql.org/users-lounge/docs/7.1/admin/install-win32.html> 上免费下载的 PostgreSQL 管理员向导中的第 2 章中的说明操作。

在服务器启动后，你必须启动数据库簇集。这必须通过管理 PostgreSQL 数据库的用户（通常是 postgres）才能做。你不能以 root 或 administrator 的名义来执行管理操作。启动数据库簇集的命令行如下：

```
initdb -D /path/to/databdirectory
```

PostgreSQL 自动带有 initdb 文件。数据目录通常是在 /usr/local/pgsql/data。按下面的命令：

```
initdb -D /usr/local/pgsql/data
```

在前面命令所创建的数据目录中的文件是 pg\_hba.conf。该文件控制哪一个主机连到数据库。在你能用 JDBC 驱动器连接数据库前，你要在连接的机器的 pg\_hba.conf 文件中必须有一个主机入口。主机入口需要字“host”、主机要连接的数据库名、所连接机器的 IP 地址、位掩码来表明 IP 地址中重要的位及认证类型。如果你想从本地机上连接，主机条目如下所示：

```
host test 127.0.0.1 255.255.255.255 crypt
```

认证类型可是下面任何类型中的一种：

- Trust——无认证需要。

- Password——匹配一个带密码的用户名。
- Crypt——与“密码”一样，但在网络中传输时是以加密的方式传送的。
- Ident——通过 ident 服务器认证。
- Krb4——使用 Kerberos V4 认证。
- Krb5——使用 Kerberos V5 认证。
- Reject——从指定的 IP 连接被拒。

下一步创建一个用户 jyuser。PostgreSQL 使用一个叫作 `createuser` 的工具程序来创建新的用户。其命令如下：

```
[shell prompt]$ createuser -U postgres -d -P jyuser
Enter password for user "jyuser":
Enter it again:
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

密码 beans 在密码提示下输入，但它不会返回终端。输出 CREATE USER 证明创建成功。在创建了新的用户之后，你可以减少以 `postgres` 用户名义做的工作。以 `jyuser` 的名义来执行下面的操作是安全的，如创建 `test` 数据库。

创建 `test` 数据库要使用 PostgreSQL 的工具程序 `createdb`。下面是用 `createdb` 来创建 `test` 数据库的命令：

```
[shell prompt]$ createdb -U jyuser test
CREATE DATABASE
```

输出 CREATE DATABASE 保证创建是成功的。

现在你可启动服务器。你可以以管理服务器的用户名（通常是 `postgres`）来启动服务器。为启动服务器，可使用下面的命令（用你安装时所选择的数据目录的路径来代替其中的数据目录的路径）：

```
postmaster -i -D /usr/local/pgsql/data &
```

你必须用 `-i` 选项是由于它是 PostgreSQL 能接收网络套接字连接的唯一办法（需要用来与 JDBC 连接）。

现在 PostgreSQL 可用来运行本书的例子了。

## 11.4 JDBC

Java 使用 JDBC 和 `java.sql` 包来与 SQL 数据库交互。这意味着 Jython 也能使用 JDBC 和 `java.sql` 包。使用 JDBC API 和 `java.sql` 包来与 SQL 数据库交互仅需要对所使用的数据库合适的 JDBC 驱动器（当然也包括数据库本身）。本章的例子依赖于前面一节“数据库管理系统”中所讲的 MySQL 和 PostgreSQL 驱动器。实际的步骤与 Java 中一样，只是采用了 Jython 的语法。交互需要一个连接、一个语句、一个结果集以及不断需要处理的错误与警告。这些都是基本的并使用了本节中 Jython 的一些基础知识。另外像事务、存储过程等高级特性也将出现。

## 连接到数据库

数据库的连接需要一个 JDBC URL，一般也需要一个用户名和密码。在连接之前，你必须理解 JDBC URL。

### 1. JDBC URI

URL 是一个惟一标志数据库连接的字符串。语法为：

```
jdbc:<subprotocol>:<subname>
```

URL 以 jdbc: 开头，后面跟着子协议，通常跟数据库厂商的产品名（如 Oracle、MySQL、PostgreSQL）一致。另外 URL 与驱动器相关，这意味着 subname 依赖于 subprotocol。下面分别是 MySQL 数据库的 JDBC URL 语法和 PostgreSQL 的 JDBC URL 语法：

```
jdbc:mysql://[hostname][:port]/databaseName[parameter=value]
jdbc:postgresql://[hostname][:port]/databaseName[parameter=value]
```

上述二个 URL 只有一个区别，那就是子协议（产品）名。为连接到本地机的 test 数据库，使用下面的与你想要连接的数据库系统匹配的协议相对应的命令：

```
jdbc:mysql://localhost:3306/test
jdbc:postgresql://localhost:5432/test
```

缺省的主机为 localhost，故实际上 URL 中不需要它也可以。缺省的端口号对 MySQL 与 PostgreSQL 分别是 3306 和 5432，故在缺省值为正确的情况下这些值也不需要。对 test 数据库最短的 localhost 连接为：

```
jdbc:mysql:///test
jdbc:postgresql:///test
```

注意斜线的数目仍然不变，但在没指定端口时区分主机和端口号的分号省略了。

下面是一些对其他数据库驱动器的 JDBC URL 的例子。在这些 URL 中均假设数据库名为 test：

```
Driver: COM.ibm.db2.jdbc.net.DB2Driver
URL: jdbc:db2://localhost/test
```

```
Driver: oracle.jdbc.driver.OracleDriver
URL: jdbc:oracle:thin:@localhost:ORCL
```

```
Driver: sun.jdbc.odbc.JdbcOdbcDriver
URL: JDBC:ODBC:test
```

```
Driver: informix.api.jdbc.JDBCDriver
URL: jdbc:informix-sqli://localhost/test
```

在 JDBC URL 的最后，你可以指定连接参数。Parameters 可以是数据库驱动器所允许的任何以 & 分隔的参数。表 11-2 列出了每一个参数，其意义、缺省值、以及是 MySQL 与 PostgreSQL 两个数据库系统中哪一个允许该参数。java.sql 包也包含类 DriverPropertyInfo，它是探究和设置

连接性质的另一办法。

表 11-2 驱动器参数

参 数	含 义	缺 省 值	数 据 库
User	你的数据库用户名	none	Both
password	你的密码	none	Both
autoReconnect	如果连接失效后尝试重新连接? (true/false)	false	MySQL
maxReconnects	驱动器要尝试多少次重新连接? (假设 autoReconnect 为 true。)	3	MySQL
initialTimeout	在重新连接前要等多少秒? (假设 autoReconnect 为 true。)	2	MySQL
maxRows	返回的行的最大数目 (0 = 所有的行)	0	MySQL
useUnicode	使用 Unicode-true 或 false?	false	MySQL
characterEncoding	使用哪个 Unicode 字符编码 (如果 useUnicode 为 true)	none	MySQL

下面是一个在本地机上指定 test 数据库的 URL 的例子，它指定用 Unicode 对字符串编码。除非要用缺省值以外的情况，端口号一般省略：

```
jdbc:mysql://localhost/test?useUnicode=true
jdbc:postgresql://localhost/test?useUnicode=true
```

如果由于 localhost 已经是缺省值，主机名又被包括，那么这些 URL 变成：

```
jdbc:mysql:///test?useUnicode=true
jdbc:postgresql:///test?useUnicode=true
```

指定在 IP 地址为 192.168.1.10 的机器上，端口号为 8060，用户名为 bob，密码为 letmein 的数据库 products 的 URL 为：

```
jdbc:mysql://192.168.1.10:8060/products?user=bob&password=letmein
jdbc:postgresql://192.168.1.10:8060/products?user=bob&password=letmein
```

实际上，用户名和密码很少作为参数指定，因为实际连接的方法已经接收了这样的参数。URL 字符串的参数可能会很繁杂。

## 2. JDBC 连接

使用 Java 的数据库连接建立数据库连接的步骤如下：

- 1) 在 classpath 中包含合适的驱动器。
- 2) 用 JDBC DriverManager 来注册驱动器。
- 3) 给 java.sql.DriverManager.getConnection 方法提供一个 JDBC URL，用户名和密码。

你必须保证在 classpath 中存在合适的数据库驱动器，或者作为环境变量或者作为在 Jython 启动脚本中 Java 的 -classpath 选项。本章中 MySQL 和 PostgreSQL 的 JDBC 驱动器所使用的 jar 文件的文件名分别为 mm.mysql-2\_0\_4-bin.jar 和 jdbc7\_1-1\_2.jar。将这些 jar 文件加入 classpath 需要下列的 shell 命令：

```
# For MySQL
set CLASSPATH=%path%\to\mm_mysql-2_0_4-bin.jar;%CLASSPATH%
# For PostgreSQL
set CLASSPATH=%path%\to\jdbc7.1-1.2.jar;%CLASSPATH%
```

一旦驱动器在 classpath 中存在，则 JDBC 需要驱动器已装载，或用 DriverManager 注册。有两个基本的方法来装载驱动器。你可以利用命令行的 -D 选项来给合适的驱动器设置 jdbc.drivers 特性，或你可使用 java.lang.Class.forName (classname) 方法。

利用 -D 选项来注册一个驱动器需要创建新的批处理或脚本来启动 Jython。下面为两个命令行，一个为注册 MySQL 驱动器，一个为注册 PostgreSQL 驱动器。

```
dos>java -Djdbc.drivers=org.gjt.mm.mysql.Driver \
-Dpython.home=c:\jython-2.1 \
-classpath c:\jython-2.1\jython.jar;%path%\to\mm.mysql-2_0_4-bin.jar \
org.python.util.jython

dos>java -Djdbc.drivers=org.postgresql.Driver \
-Dpython.home=c:\jython-2.1 \
-classpath c:\jython-2.1\jython.jar;%path%\to\jdbc7.1-1.2.jar \
org.python.util.jython
```

你也可用 Java 的动态 Class.forName (classname) 语法来装载一个驱动器。记住 java.lang 包中的类在 Jython 中不再自动有效，故在调用这个方法之前你必须引入 java.lang.Class 或它的父包中的一个。下面的例子用类 Class.forName 装载 MySQL 和 PostgreSQL 驱动器：

```
import java
try:
    java.lang.Class.forName("org.gjt.mm.mysql.Driver")
    java.lang.Class.forName("org.postgresql.Driver")
except java.lang.ClassNotFoundException:
    print "No appropriate database driver found"
    raise SystemExit
```

正如 Java 中一样，前面的例子会捕捉一个 Java 的异常 ClassNotFoundException。

在 classpath 中有了驱动器并用 DriverManager 注册后，我们可连接数据库了。这时，我们使用 java.sql.DriverManager 的 getConnection 方法。getConnection 方法有三种形式：

```
public static Connection getConnection(String url) throws SQLException
public static Connection getConnection(String url, Properties info)
    throws SQLException
public static Connection getConnection(
    String url, String user, String password) throws SQLException
```

假设装载的本地数据库 test 存在且你准备用用户名为 jyuser 和密码 beans 来连接数据库。连接的可能语法为：

```
from java.sql import DriverManager
from java.util import Properties
# Using the getConnection(URL) method
```

```

mysqlConn = DriverManager.getConnection(
    "jdbc:mysql://localhost/test?user=jyuser&password=beans")

postgresqlConn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/test?user=jyuser&password=beans")

# Using the getConnection(URL, Properties) method
params = Properties()
params.setProperty("user", "jyuser")
params.setProperty("password", "beans")

mysqlConn = DriverManager.getConnection("jdbc:mysql://localhost/test",
    params)

postgresqlConn =
    DriverManager.getConnection("jdbc:postgresql://localhost/test",
        params)

# Using the getConnection(URL, String, String) method
mysqlConn =
    DriverManager.getConnection("jdbc:mysql://localhost/test",
        "jyuser", "beans")

postgresqlConn =
    DriverManager.getConnection("jdbc:postgresql://localhost/test",
        "jyuser", "beans")

```

每一个方法返回一个数据库连接使你可与数据库系统交互。

### 连接脚本

Jython 的交互解释使其可作为一个理想的数据库客户端工具。MySQL 和 PostgreSQL 都有交互式的、控制台的客户端应用，所以使用 Jython 时可与这些工具同时使用。然而你也要继承 Jython 的函数、类等等并对任何有 JDBC 驱动器的数据库都有一致的接口。

Jython 要成为交互式用户惟一要做的事情就是连接。每次在你要用数据库时在 Jython 中交互式地输入数据库连接语句不是一个好的方法。相反，你可使用 Jython 的 -i 命令行。当以 -i 选项运行时，Jython 以交互式的方式继续，即使在脚本执行以后。这使得你可以在脚本中设置数据库连接，且在脚本完成后继续与由脚本创建的对象交互。用 -i 选项的问题是没有机制来明显地自动关闭数据库资源。在数据库上工作时，最好总是明显地关闭任何数据库资源。这意味着在每次完成数据库的工作退出解释器之前都要输入 close 方法，故这始终不是一个最好的方法。

将交互的控制打包到所需要的连接和关闭语句中或用 Java 的 Runtime.addShutdownHook 可保证每次连接都是关闭的。因为 ShutdownHook 不是在每个 Java 的版本中都有，最好的方法似乎是将 InteractiveConsole 实例打包到所需要的连接和关闭语句中。程序清单 11-1 做了这个工作。数据库连接和语句对象创建后，内部的 InteractiveConsole 就在这些对象的名字空间中启动了。当这个内部的 InteractiveConsole 存在时，连接和语句对象就关闭了。

## 程序清单 11-1 数据库客户端启动脚本

```
# File: mystart.py

from java.lang import Class
from java.sql import DriverManager
from org.python.util import InteractiveConsole
import sys

url = "jdbc:mysql://localhost/%s"
user, password = "jyuser", "beans"
# Register driver
Class.forName("org.gjt.mm.mysql.Driver")

# Allow database to be optionally set as a command-line parameter
if len(sys.argv) == 2:
    url = url % sys.argv[1]
else:
    url = url % "test"

# Get connection
dbconn = DriverManager.getConnection(url, user, password)

# Create statement
try:
    stmt = dbconn.createStatement()
except:
    dbconn.close()
    raise SystemExit

# Create inner console
interp = InteractiveConsole({"dbconn":dbconn, "stmt":stmt}, "DB Shell")

try:
    interp.interact("Jython DB client")
finally:
    # Close resources
    stmt.close()
    dbconn.close()
    print
```

要连接 test 数据库，你所需要做的是要保证 classpath 中有合适的驱动器，然后运行 jython mystart.py。下面是一个使用 mystart.py 的简单交互式会话的例子：

```
shell-prompt> jython mystart.py
Jython DB Client
>>> dbconn.catalog
'test'
>>> # Exit the interpreter on the next line
>>>
```

PostgreSQL 的启动脚本除在 JDBC URL 中 postgresql 的名字外将与表 11-1 所列出的一样。注意 PostgreSQL 的 dbconn.catalog 是空的。

**连接会话**

另一个常见的连接问题是用对话框获取连接信息。需要数据库登录的客户端应用通常需要用对话窗口来获取连接信息。该问题的常见性需要用一个例子来解释，程序清单 11-2 正是这样一个例子。程序清单 11-2 是一个建立数据库连接并返回该连接给注册的客户端对象的对话框。客户端的对象是任何的可调用的对象，该对象只接受一个参数——连接。当检索到连接成功以后，对话框将该连接导向注册的客户端对象。然后关闭连接就是客户端的任务。

程序清单 11-2 连接的获得发生在 DBLoginDialog 类的 login 方法之中。在 login 方法之中有两个重要的 try/except 子句。第一个 try/except 子句在 classpath 中没有合适的驱动器时捕获 ClassNotFoundException，第二个 try/except 子句在想获得一个数据库的连接时捕获 SQLException。

当模块作为主脚本运行时，在程序清单 11-2 中定义的 dummyClient 函数作为对话框的客户端。成功的连接发送到 dummyClient，然后由 dummyClient 来关闭对象。

#### 程序清单 11-2 数据库连接对话框

```
# File: DBLoginDialog.py

import sys
import java
from java import awt
from java import sql
import pawt

class DBLoginDialog(awt.Dialog):
    '''DBLoginDialog prompts a user of database information and
    establishes a database login. A connection receiver is registered
    as client. for example:
        def connectionClient(dbconnection):
            # do something with database connection
        dbl = DBLoginDialog(parentframe, message)
        dbl.client = connectionClient'''

    def __init__(self, parentFrame, message):
        awt.Dialog.__init__(self, parentFrame)
        self.client = None
        self.background = pawt.colors.lightgrey
        bag = pawt.GridBag(self, anchor='WEST', fill='NONE')

        # Make required components
        self.Hostname = HighlightTextField("localhost", 24)
        self.DBMS = awt.Choice(itemStateChanged=self.updatePort)
        self.Port = HighlightTextField("3306", 5)
        self.Database = HighlightTextField("test", 12)
        self.User = HighlightTextField("jyuser", 10)
        self.Password = HighlightTextField("", 10, echoChar='*')

        # Fill the choice component with options
        self.DBMS.add("MySQL")
        self.DBMS.add("PostgreSQL")

        # add message
        bag.addRow(awt.Label(message), anchor='CENTER')

        # Put components in the bag
        for x in ["Hostname", "DBMS", "Port", "Database",
                  "User", "Password"]:
```

```

bag.add.awt.Label(x + ':'))
bag.addRow(self.__dict__[x])

# Add action buttons
bag.add.awt.Button("Login", actionPerformed=self.login),
    fill='HORIZONTAL')
bag.addRow.awt.Button("Cancel", actionPerformed= self.close),
    anchor='CENTER')

self.pack()
bounds = parentFrame.bounds
self.bounds = (bounds.x + bounds.width/2 - self.width/2,
               bounds.y + bounds.height/2 - self.height/2,
               self.width, self.height)

def login(self, e):
    db = self.DBMS.selectedItem
    if db == "MySQL":
        driver = "org.gjt.mm.mysql.Driver"
    else:
        driver = "org.postgresql.Driver"
    try:
        java.lang.Class.forName(driver)
    except java.lang.ClassNotFoundException:
        self.showError("Unable to load driver %s" % driver)
        return
    url = "jdbc:%s://%s:%s/%s" % (db.lower(), self.Hostname.text,
                                    self.Port.text,
                                    self.Database.text)
    try:
        dbcon = sql.DriverManager.getConnection(url,
                                                self.User.text,
                                                self.Password.text)
        self.dispose()
        self.client(dbcon)
    except sql.SQLException:
        self.showError("Unable to connect to database")

def updatePort(self, e):
    if self.DBMS.selectedItem == 'MySQL':
        port = '3306'
    elif self.DBMS.selectedItem == 'PostgreSQL':
        port = '5432'
    self.Port.text= port

def setClient(client):
    self.client = client

def showError(self, message):
    d = awt.Dialog(self.parent, "Error")
    panel = awt.Panel()
    panel.add.awt.Label(message))
    button = awt.Button("OK")
    panel.add(button)
    d.add(panel)
    d.pack()
    bounds = self.parent.bounds
    d.bounds = (bounds.x + bounds.width/2 - d.width/2,
                bounds.y + bounds.height/2 - d.height/2,
                d.width, d.height)
    d.windowClosing = lambda e, d=d: d.dispose()
    button.actionPerformed = d.windowClosing

```

```

d.visible = 1

def close(self, e):
    self.dispose()

class HighlightTextField.awt.TextField, awt.event.FocusListener):
    def __init__(self, text, chars, **kw):
        awt.TextField.__init__(self, text, chars)
        self.addFocusListener(self)
        for k in kw.keys():
            exec("self." + k + "=kw[k]")

    def focusGained(self, e):
        e.source.selectAll()

    def focusLost(self, e):
        e.source.select(0, 0)

def dummyClient(connection):

    if connection != None:
        print "\nDatabase connection successfully received by client."
        print "Connection=", connection
        connection.close()
        print "Database connection properly closed by client."

if __name__ == '__main__':
    # make a dummy frame to parent the dialog window
    f = awt.Frame("DB Login", windowClosing=lambda e: sys.exit())
    screensize = f.toolkit.screenSize
    f.bounds = (screensize.width/2 - f.width/2,
                screensize.height/2 - f.height/2,
                f.width, f.height)

    # create and show the dialog window
    dbi = DBLoginDialog(f, "Connection Information")
    dbi.client = dummyClient
    dbi.windowClosing = dbi.windowClosed = lambda e: sys.exit()
    dbi.visible = 1

```

要执行 DBLoginDialog，你所需要做的是保证 classpath 中有合适的数据库驱动器，然后运行 jython DBLoginDialog.py。你将会看到如图 11-1 所示的登录接口。

### 3. 数据库元数据

一旦连接上以后，你就可以探究数据库和连接，或元数据的信息。Java 的连接对象 (java.sql.Connection)

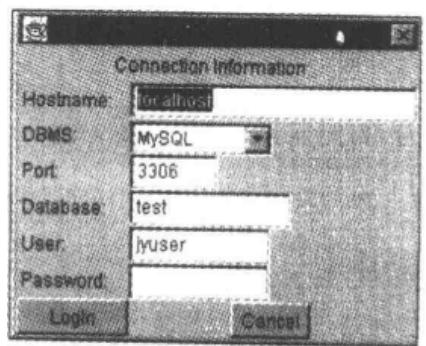


图 11-1 DB 登录对话框

有一个方法 `getMetaData`，它返回一个 `java.sql.DatabaseMetaData` 的对象。`DatabaseMetaData` 对象提供了其关系数据库连接的完全信息。`DatabaseMetaData` 对象有很多方法，要获得详细的细节你必须查看 javadoc 文档。

当学习一个新的数据库工具和试图支持多个数据库时，元数据变得非常重要。你可以用 Jython 交互式地发现数据库的特征和特性。假设你要知道 MySQL 的系统函数、数值函数和对事务的支持。交互式的发现可按下列方法进行：

```
>>> import java
>>> from java.sql import DriverManager
>>>
>>> # Register driver
>>> java.lang.Class.forName("org.gjt.mm.mysql.Driver")
<jclass org.gjt.mm.mysql.Driver at 650329>
>>>
>>> # Get Connection
>>> url, user, password = "jdbc:mysql:///test", "jyuser", "beans"
>>> dbconn = DriverManager.getConnection(url, user, password)
>>>
>>> # Get the DatabaseMetaData object
>>> md = dbconn.getMetaData()
>>>
>>> # Use the metadata object to find info about MySQL
>>> print md.systemFunctions
DATABASE,USER,SYSTEM_USER,SESSION_USER,PASSWORD,ENCRYPT,LAST_INSERT_ID,
VERSION
>>> md.numericFunctions
'ABS,ACOS,ASIN,ATAN,ATAN2,BIT_COUNT,CEILING,COS,COT,DEGREES,EXP,FLOOR,LOG,
LOG10,MAX,MIN,MOD,PI,POW,POWER,RADIANS,RAND,ROUND,SIN,SQRT,TAN,TRUNCATE'
>>> md.supportsTransactions()
0
>>>
>>> # Don't forget to close connections
>>> dbconn.close()
```

在 `DatabaseMetaData` 对象大量的方法中，只有一些能说明 MySQL 和 PostgreSQL 数据库的区别。程序清单 11-3 使用了那些方法来创建一个可视化的报表来比较 MySQL 和 PostgreSQL 的区别。很多区别都是很琐碎的，但其他一些如事务、模式和联合却不是。程序清单 11-3 使用了在程序清单 11-1 中定义的 `DBLoginDialog` 来创建一个连接。故 `DBLoginDialog.py` 必须在 `sys.path` 之中，最好在目前的工作目录。当运行程序清单 11-3 时，对数据库连接有两个对话框。第一个必须选择 MySQL 数据库，而第二个选择 PostgreSQL 数据库。脚本通过调用一系列所期望的方法的循环来产生一个报表。在这个循环中，Jython 的 `exec` 语句执行一个构建了的语句字符串，通常是如下的语句：

```
value = metaData.supportsTransactions()
```

将变成类似于：

```
cmd = "supportsTransactions"
exec("value = metaData.%s()" % cmd)
```

在所有的信息都收集后，每个数据库的结果都将在 TextArea 中出现。

### 程序清单 11-3 用 JDBC 检索元数据

```
# File: jdbcMetaData.py
from DBLoginDialog import DBLoginDialog
from java import awt
import sys

class MetaData(awt.Frame):
    def __init__(self):
        self.windowClosing=lambda e: sys.exit()
        self.databases = ["MySQL", "PostgreSQL"]
        self.panel = awt.Panel()
        self.infoArea = awt.TextArea("", 40, 60,
                                      awt.TextArea.SCROLLBARS_VERTICAL_ONLY,
                                      font=("Monospaced", awt.Font.PLAIN, 12))
        self.panel.add(self.infoArea)
        self.add(self.panel)
        self.pack()
        screensize = self.toolkit.screenSize
        self.bounds = (screensize.width/2 - self.width/2,
                       screensize.height/2 - self.height/2,
                       self.width, self.height)
        self.data = {}
        self.visible = 1
        self.dialog = DBLoginDialog(self, "Select a Connection")
        self.dialog.client = self.gatherMetaInfo
        self.dialog.visible = 1

    def showResults(self):
        infoWidth = self.infoArea.columns
        info = 'Method' + 'MySQL  PostgreSQL\n'.rjust(infoWidth - 7)
        info += '-' * (infoWidth - 1) + '\n'
        keys = self.data.keys()
        keys.sort()
        for x in keys:
            info += x
            mysql = str(self.data[x]['MySQL'])
            postgresql = str(self.data[x]['PostgreSQL'])
            results = mysql.ljust(18 + len(postgresql)) + postgresql
            info += results.rjust(self.infoArea.columns + len(x) - 2)
            info += "\n"
        self.infoArea.text = info

    def nextDatabase(self):
        if len(self.databases):
            self.dialog.visible = 1
        else:
            self.showResults()

    def gatherMetaInfo(self, dbconn):
        if dbconn==None:
            return
        metaData = dbconn.getMetaData()
        dbname = metaData.databaseProductName
        for cmd in self.getCommands():
```

```

value = ""
try:
    exec("value = metaData.%s()" % cmd)
except:
    value = "Test failed"
if self.data.has_key(cmd):
    self.data[cmd][dbname] = value
else:
    self.data.update({cmd:{dbname:value}})
dbconn.close() # close the database!
self.databases.remove(dbname)
self.nextDatabase()

def getCommands(self):
    return ['getCatalogTerm', 'getMaxBinaryLiteralLength',
        'getMaxCatalogNameLength', 'getMaxCharLiteralLength',
        'getMaxColumnsInGroupBy', 'getMaxColumnsInIndex',
        'getMaxColumnsInOrderBy', 'getMaxColumnsInSelect',
        'getMaxColumnsInTable', 'getMaxConnections',
        'getMaxCursorNameLength', 'getMaxIndexLength',
        'getMaxProcedureNameLength', 'getMaxRowSize',
        'getMaxStatementLength', 'getMaxStatements',
        'getMaxTablesInSelect', 'getMaxUserNameLength',
        'supportsANSI92EntryLevelSQL', 'supportsBatchUpdates',
        'supportsCatalogsInDataManipulation',
        'supportsCoreSQLGrammar',
        'supportsDataManipulationTransactionsOnly',
        'supportsDifferentTableCorrelationNames',
        'supportsFullOuterJoins', 'supportsGroupByUnrelated',
        'supportsMixedCaseQuotedIdentifiers',
        'supportsOpenStatementsAcrossCommit',
        'supportsOpenStatementsAcrossRollback',
        'supportsPositionedDelete', 'supportsUnion',
        'supportsSubqueriesInComparisons',
        'supportsTableCorrelationNames', 'supportsTransactions']

if __name__ == '__main__':
    md = MetaData()

```

要执行程序清单 11-3，必须保证两个数据库驱动器都在 classpath 之中。然后运行下面的命令：

```
python jdbcMetaData.py
```

你首先会看到与程序清单 11-2 同样的对话框。在那儿输入一个 MySQL 连接。对话框将返回；然后再输入一个 PostgreSQL 连接。这样，你就可看到如图 11-2 所示的在两个数据库之间的元数据比较。

#### 4. 语句

大部分与 MySQL 和 PostgreSQL 的交互都包含 SQL 语句。产生 SQL 语句需要一个 statement 对象。要创建一个 statement 对象，使用 connection 的 createStatement() 方法。

```
>>> import java
>>> java.lang.Class.forName("org.gjt.mm.mysql.Driver")
```

```
<jclass org.gjt.mm.mysql.Driver at 1876475>
>>> db, user, password = "test", "jyuser", "beans"
>>> url = "jdbc:mysql://localhost/%s" % db
>>> dbconn = java.sql.DriverManager.getConnection(url, user, password)
>>> stmt = dbconn.createStatement()
```

The screenshot shows a Jupyter Notebook cell with a table comparing MySQL and PostgreSQL methods. The table has two columns: 'Method' and 'MySQL' (with sub-columns 'database', 'Catalog', and 'Procedure'). The 'PostgreSQL' column is also present. The data is as follows:

Method	MySQL	PostgreSQL
getCatalogTerm	database	Catalog
getMaxBinaryLiteralLength	16777208	0
getMaxCatalogNameLength	32	0
getMaxCharLiteralLength	16777208	65535
getMaxColumnsInGroupBy	16	1600
getMaxColumnsInIndex	16	1600
getMaxColumnsInOrderBy	16	1600
getMaxColumnsInSelect	256	1600
getMaxColumnsInTable	512	1600
getMaxConnections	0	8192
getMaxCursorNameLength	64	32
getMaxIndexLength	128	65535
getMaxProcedureNameLength	0	32
getMaxRowSize	2147483639	65535
getMaxStatementLength	65531	65535
getMaxStatements	0	1
getMaxTablesInSelect	256	1024
getMaxUserNameLength	16	32
supportsANSI92EntryLevelSQL	1	0
supportsBatchUpdates	0	1
supportsCatalogsInDataManipulation	1	0
supportsCoreSQLGrammar	1	0
supportsDataManipulationTransactionsOnly	0	1

图 11-2 MySQL - PostgreSQL 比较结果

`createStatement` 方法也可为结果集设置类型和 `concurrency` 参数。在一个结果集能用它的 `update*` 方法之前，语句必须被创建并且 `concurrency` 设置为可更新的。下面例子通过设置类型和 `concurrency` 参数来创建一个可更新的结果集：

```
>>> import java
>>> from java import sql
>>> java.lang.Class.forName("org.gjt.mm.mysql.Driver")
<jclass org.gjt.mm.mysql.Driver at 1876475>
>>> db, user, password = "test", "jyuser", "beans"
>>> url = "jdbc:mysql://localhost/%s" % db
>>> dbconn = sql.DriverManager.getConnection(url, user, password)
>>>
>>> type = sql.ResultSet.TYPE_SCROLL_SENSITIVE
>>> concurrency = sql.ResultSet.CONCUR_UPDATABLE
>>> stmt = dbconn.createStatement(type, concurrency)
```

`statement` 对象允许你用三个方法 `execute()`、`executeQuery()`、`executeUpdate()` 来执行 SQL 语句。这三个方法的返回值有所不同。`execute()` 方法返回 1 或 0，它们表示有无 `ResultSet`。实际上，它返回一个 Python 作为 1 或 0 解释的 Java 布尔值。`executeQuery()` 方法总返回一个 `ResultSet`。`executeUpdate()` 方法返回一个表示受影响的行数。

```
>>> query = "CREATE TABLE random ( number tinyint, letter char(1) )"
>>> stmt.execute(query)
0
```

在前面的例子中，查询更新了数据库，这意味着没有 ResultSet。你可通过 statement 对象的 getResultSet() 方法或 resultSet 的自动 bean 特性来确认它。

```
>>> print stmt.resultSet
None
```

现在表已存在，你可在表中填充数据。这可以使用 executeUpdate() 方法。下面的交互例子通过将一些数据加入 random 表中继续了前面的例子（statement 对象已存在）：

```
>>> import string, random
>>> for i in range(20):
...     number = random.randint(0, 51)
...     query = "INSERT INTO random (number, letter) VALUES (%i, '%s')"
...     query = query % (number, string.letters[number - 1])
...     stmt.executeUpdate(query)
1
...
1
```

你可看出许多 1s 的输出，这表明在每次更新后行改变了。注意在查询字符串时 ‘%s’ 中的单引号。字母与所有的字符串都必须在引号内。使用内部单引号很方便，但如果字符串是一个单引号呢？该字母的单引号将导致一个 SQLException 的异常。单引号在查询字符串中必须重复，这意味着在查询串中字符串 “It's a bird” 必须变为 “Its a bird”。即使是这样，你还必须适应 Jython 的语法，在 “之前加入 \ 号以保证适当的 SQL 语句。对单引号使用 \” 号的例子如下：

```
>>> query = "INSERT INTO random (number, letter) VALUES (%i, \"%s\")"
>>> query = query % (4, "")
```

现在要注意 random 表中有数据，你可使用 executeQuery() 方法来选择数据。然后继续上的交互会话，下面的例子从 random 表中选择所有的数据：

```
>>> rs = stmt.executeQuery("SELECT * FROM random")
```

现在我们必须使用 ResultSet 对象 (rs) 来检查 SELECT 语句的结果。

## 5. ResultSet

当你使用 executeQuery 方法时，返回的对象是 ResultSet。java.sql.ResultSet 对象包含很多在结果中导航并以 Java 固有类型的方式检索的方法。记住 Jython 会自动将 Java 的固有类型转变为合适的 Jython 类型。使用 ResultSet 的 getByte() 方法将返回一个变为 PyInteger 的值，getDouble() 方法将返回一个 PyFloat 等等。这些都是按照第 2 章 “运算符、类型和内部函数” 中的 Java 类型转换规则进行。

要查询前面创建的 random 表且在它所有的条目中循环可使用 ResultSet 的 next() 和 get\* 方法。在下面的例子中，数字字段通过 getInt() 检索，而字母字段通过 getString() 来检索。这产生了 Jython 的类型 PyInteger 和 PyString：

```
>>> import java
>>> java.lang.Class.forName("org.gjt.mm.mysql.Driver")
<jclass org.gjt.mm.mysql.Driver at 1876475>
```

```

>>> db, user, password = "test", "jyuser", "beans"
>>> url = "jdbc:mysql://localhost/%s" % db
>>> dbconn = java.sql.DriverManager.getConnection(url, user, password)
>>> stmt = dbconn.createStatement()
>>>
>>> rs = stmt.executeQuery("SELECT * FROM random")
>>> while rs.next():
...     print rs.getString('letter'), ': ', rs.getInt('number')
...
...
U : 46
K : 36
h : 7
Q : 42
l : 11
u : 20
n : 13
z : 25
U : 46
Y : 50
j : 9
o : 14
i : 8
s : 18
d : 3
A : 26
K : 36
j : 9
n : 13
g : 6
>>> stmt.close()
>>> dbconn.close()

```

注意到由于 random 模块的使用导致输出有所不同。

在结果集中的导航包括使用 ResultSet 的导航方法在记录中移动。这主要依你所使用的 JDBC 不同和你所创建的语句的类型不同而有所区别。在 JDBC 2.0 以前，导航仅限制于 next() 方法。目前的 JDBC 版本包括了 next()、first()、last()、previous()、absolute()、relative()、afterLast()、beforeFirst()、moveToCurrentRow()、moveToInsertRow() 等等。其中 moveToInsertRow() 和 moveToCurrentRow() 在 PostgreSQL 驱动器中未实现，在 MySQL 中则需要特定的条件。

这些导航方法的大部分从名字上就可看出其意义，但其中有四个却不那么明显。导航方法 relative (int) 使光标从目前的位置移动 int 数目。导航方法 absolute (int) 不管光标目前的位置而将光标移到结果集的 int 行。导航方法 moveToInsertRow() 目前在 MySQL 中有效。它将光标移到一个特定的行，该行表示与数据库检索结果无关的构造区域。在这个区域中，你可构造你想要插入的行。在你调用了 moveToInsertRow() 并想返回你跳到构造区域前你所在的那一行，你采用 moveToCurrentRow() 方法。注意 moveToInsertRow() 和 moveToCurrentRow() 方法需要一个可更新的 ResultSet 对象。

除了可在行之间导航，结果集还能在驱动器支持的情况下更换行。目前 PostgreSQL 不支持，但 MySQL 支持。当然也有条件限制哪个结果集为可更新的。在 MySQL 中，只有在查询只

包含一个表、查询时未采用联接且查询时选择表的主键等条件下结果集才是可更新的。一个对所有的 JDBC 驱动器的附加要求是要有语句设置 ResultSet.CONCUR\_UPDATABLE 为 concurrency 类型。要执行插入，你必须保证所有产生可更新结果集的查询必须包含所有无缺省值的行和所有非 NULL 行。

由于前面创建的 random 表缺少主键，在它的结果集能被更新之前必须创建一个主键。下面的交互例子说明了通过滚动和更新演示如何来更新 random 表：

```
>>> import java
>>> from java import sql
>>> java.lang.Class.forName("org.gjt.mm.mysql.Driver")
<jclass org.gjt.mm.mysql.Driver at 1876475>
>>> user, password = "jyuser", "beans"
>>> url = "jdbc:mysql://localhost/test"
>>> dbconn = sql.DriverManager.getConnection(url, user, password)
>>>
>>> type = sql.ResultSet.TYPE_SCROLL_SENSITIVE
>>> concurrency = sql.ResultSet.CONCUR_UPDATABLE
>>> stmt = dbconn.createStatement(type, concurrency)
>>>
>>> # Update table to include primary key (excuse the wrap)
>>> query = "ALTER TABLE random ADD pkey INT UNSIGNED NOT NULL AUTO_INCREMENT
PRIMARY KEY"
>>> stmt.execute(query)
0
>>>
>>> # Now get an updatable result set.
>>> rs = stmt.executeQuery("select * from random")
>>> rs.concurrency # 1008 is "updatable"
1008
>>>
>>> # Insert a row
>>> rs.moveToInsertRow()
>>> rs.updateInt('number', 7)
>>> rs.updateString('letter', 'g')
>>> rs.updateInt('pkey', 22)
>>> rs.insertRow() # This puts it in the database
>>> rs.moveToCurrentRow()
>>>
>>> rs.relative(5) # scroll 5 rows
1
>>> # Print current row data
>>> # Remember, data is random- odds are yours will differ
>>> print rs.getString('letter'), rs.getInt('number')
h 8
>>> rs.updateInt('number', 3)
>>> rs.updateString('letter', 'c')
>>> rs.updateRow() # this puts it in the database
>>> stmt.close()
>>> dbconn.close()
```

## 6. 预编译语句

预编译 SQL 语句常常可用来提高效率。java.sql.PreparedStatement 可用来创建一个预编译语句。要创建一个预编译语句，可使用 connection 对象的 prepareStatement 方法而不是 createState-

ment。一个在 random 数据库中更新一行的简单的预编译语句如下所示：

```
>>> import java
>>> from java import sql
>>> java.lang.Class.forName("org.postgresql.Driver")
<jclass org.gjt.mm.mysql.Driver at 1876475>
>>> dbconn = sql.DriverManager.getConnection(url, user, password)
>>> query = "INSERT INTO random (letter, number) VALUES (?, ?)"
>>> preppedStmt = dbconn.prepareStatement(query)
```

注意在查询中的问号 (?)。一个预编译语句可保留一些值的占位符在运行时填充。要填充这些占位符需要你根据这些占位符的位置标志来用 set<sup>\*</sup> 方法给每一个占位符填充值。set 方法支持所有的 Java 类型。另外在设置新参数和执行更新时，你必须清空所有的参数：

```
>>> # continued from previous interactive example
>>> preppedStmt.clearParameters()
>>> preppedStmt.setString(1, "f")
>>> preppedStmt.setInt(2, 6)
>>> preppedStmt.executeUpdate()
1
>>> preppedStmt.close()
>>> dbconn.close()
```

## 7. 事务

事务就是一个语句的集合，该事务的执行只有在这些语句全部执行完成后才能成功，否则全部语句将不会执行（回滚）。PostgreSQL 是一个我们使用事务的数据库例子，这意味着 random 表现在必须在 PostgreSQL test 数据库中创建。下面是一个创建 random 表的交互式例子。

```
>>> import java
>>> from java import sql
>>> java.lang.Class.forName("org.postgresql.Driver")
<jclass org.postgresql.Driver at 5303656>
>>> db, user, password = "test", "jyuser", "beans"
>>> url = "jdbc:postgresql://localhost/%s" % db
>>> dbconn = sql.DriverManager.getConnection(url, user, password)
>>> stmt = dbconn.createStatement()
>>> query = "CREATE TABLE random ( number int, letter char(1),
    pkey INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY)"
>>> stmt.execute(query)
0
>>> import string, random
>>> for i in range(20):
...     number = random.randint(0, 51)
...     query = "INSERT INTO random (number, letter) VALUES (%i, '%$')"
...     query = query % (number, string.letters[number - 1])
...     stmt.executeUpdate(query)
1
1
...
>>> stmt.close()
>>> dbconn.close()
```

要在 Jython 中使用 JDBC 事务处理，你首先必须设置连接的 autoCommit bean 属性为 0，然后开始语句集合，在任何一个语句失败时调用 connection 的 rollback() 方法。下面是一个用 Post-

greSQL 和 random 表的事务的例子：

```
>>> import java
>>> from java import sql
>>> java.lang.Class.forName("org.postgresql.Driver")
<jclass org.postgresql.Driver at 5711596>
>>> url = "jdbc:postgresql://localhost/test"
>>> con = sql.DriverManager.getConnection(url, "jyuser", "beans")
>>> stmt = con.createStatement()
>>> con.setAutoCommit = 0
>>> try:
...     # Insert an easy-to-find character for letter
...     query = "INSERT INTO random (letter, number) VALUES ('.', 0)"
...     stmt.executeUpdate(query)
...     print "First update successful."
...     stmt.execute("Create an exception here")
...     con.commit()
... except:
...     print "Error encountered, rolling back."
...     con.rollback()
...
1
First update successful.
Error encountered, rolling back.
>>>
>>> # now confirm that the first update statement was in fact undone
>>> rs = stmt.executeQuery("SELECT * FROM random WHERE letter='.'")
>>> rs.next()
0
```

在前面的例子中结尾处最后的 0 表示查询没有结果集。这表明第一个 SQL 插入语句被回滚了。

## 11.5 zxJDBC

从 Jython 中使用 JDBC 无疑是很有价值的。它需要 Jython 中的原型系统并考虑 JDBC 的技巧。然而从很多 Java 类型专用的方法可看出它是一个 Java API。Java、数据库和 JDBC 都是类型丰富的。不足之处是 Java 固有类型专用的方法看来与 Jython 的高级的多态的动态类型有矛盾。

相反，Python 有一个指 Python DB API 的数据库 API，目前的版本为 2.0。Python 的 DB API 2.0 有一个与 CPython 中的数据库交互的标准 API，然而 CPython 使用的数据库驱动器通常由于其是用 C 语言实现的而并不适合于 Jython。虽然 Jython 能方便使用 Java 的数据库连接，但 Python 的 DB API 依然需要 Java 实现。Brian Zimmer，一个热情的 Jython、Java 和 Python 开发者，为填补这一空白开发了 zxJDBC。实际上，zxJDBC 不仅仅实现了 DB API，它还扩展了这个 API。Brian 的 zxJDBC 工具包括其源代码都是免费的，并包含有详尽的文档，读者可从 <http://sourceforge.net/projects/zxjdbce/> 或 <http://www.zielix.com/zxjdbce/> 上下载。ZxJDBC 工具在你读到本书时可能已加入到 Jython 之中。如果需要了解更多的信息，可查看 <http://www.jython.org> 或 <http://www.newrider.com/> 中的 Jython 信息。如你的 Jython 版本中未包含 zxJDBC，你必须下载它，并在你的 classpath 中包含文件 zxJDBC.jar。

zxJDBC 包所包含的工具不只是这儿所说明的，它还包含实现管道模式的一个包和数据句柄的简便创建以及 DataHandlerFilters。

### 11.5.1 连接到数据库

当你使用 zxJDBC 包时，在调用连接函数之前要准备的是 zxJDBC.jar 和所需要的 JDBC 驱动器在 classpath 中。驱动器的实际装载是在创建一个数据库连接时就已经建立了。用 zxJDBC 建立数据库连接分为下面两步：

- 1) zxJDBC.jar 文件和所需要的 JDBC 驱动器包含在 classpath 中。
  - 2) 提供一个 JDBC URL、用户名、密码和数据库 Driver 类的名字给 zxJDBC.connect() 方法。
- 使用 zxJDBC 的合适的 classpath 的设置为：

```
# For MySQL
set CLASSPATH=mm_mysql-2_0_4-bin.jar;\path\to\zxJDBC.jar;%CLASSPATH%
# For PostgreSQL
set CLASSPATH=\path\to\jdbc7.1-1.2.jar;\path\to\zxJDBC.jar;%CLASSPATH%
```

zxJDBC.connect 方法返回一个数据库连接并有下列语法：

```
zxJDBC.connect(URL, user, password, driver) -> connection
```

使用 zxJDBC.connect 方法来检索连接的方法如下：

```
from com.ziclix.python.sql import zxJDBC
mysqlConn = zxJDBC.connect("jdbc:mysql://localhost/test",
                            "jyuser", "beans",
                            "org.gjt.mm.mysql.Driver")

postgresqlConn = zxJDBC.connect("jdbc:postgresql://localhost/test",
                                 "jyuser", "beans",
                                 "org.postgresql.Driver")
```

驱动器所需要的特殊参数可作为连接函数的关键字参数。在连接 MySQL 数据库时要设置 autoReconnect 为真可按下列方式作为关键字参数包含上述参数：

```
url = "jdbc:mysql://localhost/test"
user = "jyuser"
password = "beans"
driver = "org.gjt.mm.mysql.Driver"
mysqlConn = zxJDBC.connect(url, user, password, driver,
                           autoReconnect="true")
```

连接的错误导致异常 DatabaseError，故在尝试连接时处理错误需要下列的 except 语句：

```
url = "jdbc:mysql://localhost/test"
user = "jyuser"
password = "beans"
driver = "org.gjt.mm.mysql.Driver"
try:
    mysqlConn = zxJDBC.connect(url, user, password, driver,
                               autoReconnect="true")
except zxJDBC.DatabaseError:
    pass
    #handle error here
```

如果你从 javax.sql 包或是实现 javax.sql.DataSource 或 javax.sql.ConnectionPoolDataSource 的类中使用连接工厂，你就可以使用 zxJDBC.connectx 方法来连接。注意 javax.sql 包在常规的 JDK 安装中并未包含，除非为企业版。然而 MySQL JDBC 驱动器包含下面例子中使用的 MysqlDataSource 类。zxJDBC.connectx 方法需要 DataSource 类和所有的数据库连接参数作为关键字参数或字典对象：

```
from com.ziclix.python.sql import zxJDBC
userInfo = {'user':'jyuser', 'password':'beans'}
con = zxJDBC.connectx("org.gjt.mm.mysql.MysqlDataSource",
                      serverName="localhost", databaseName='test',
                      port=3306, **userInfo)
```

bean 属性的名字在前面例子中是用关键字参数来设置的，但也可包含在包含用户名和密码信息的字典之中：

```
from com.ziclix.python.sql import zxJDBC
userInfo = {'user':'jyuser', 'password':'beans',
            'databaseName':'test', 'serverName':'localhost',
            'port':3306}
con = zxJDBC.connectx("org.gjt.mm.mysql.MysqlDataSource", **userInfo)
```

你也可通过 zxJDBC.lookup 方法的 jndi lookup 来获得一个连接。lookup 方法仅需要在特定连接或你所期望的 DataSource 中表示 JNDI 名字的一个字符串。其中可以包含关键字参数，当关键字匹配场景的静态字段名时则将被转换为 javax.jndi.Context 的静态字段值。

### 11.5.2 游标

zxJDBC 游标实际是用来与数据库中的数据进行交互的对象。zxJDBC 游标实际上是一个包含 JDBC 语句和 ResultSet 对象的封装器。静态和动态游标类型的区别在于结果集的处理。动态游标是惰性的，它只在需要时在结果集中移动。这节省了存储空间并最终节省了分配处理时间。静态游标是非惰性的。它立即在整个结果集中移动，这样会导致内存开销。静态游标的好处是在执行一个语句之后你能迅速知道行数，这是你使用动态游标时所无法得到的。

要获得一个游标，你可以调用 zxJDBC 连接对象的 cursor() 方法。下面为一连接数据库并检索游标对象的例子：

```
from com.ziclix.python.sql import zxJDBC
url = "jdbc:mysql://localhost/test"
user = "jyuser"
password = "beans"
driver = "org.gjt.mm.mysql.Driver"
con = zxJDBC.connect(url, user, password, driver,
                      autoReconnect="true")
cursor = con.cursor() # Static cursor

# Alternatively, you can create a dynamic cursor
cursor = con.cursor(1) # Optional boolean arg for dynamic
```

游标对象的 execute 方法执行 SQL 语句。下面例子表明如何执行一个从 random 表中选择所

有数据的 SQL 语句：

```
>>> from com.ziclix.python.sql import zxJDBC
>>> url = "jdbc:mysql://localhost/test"
>>> user, password, driver = "jyuser", "beans", "org.gjt.mm.mysql.Driver"
>>> con = zxJDBC.connection(url, user, password, driver)
>>> cursor = con.cursor()
>>> cursor.execute("SELECT * FROM random")
```

要在一个语句的结果集中移动，你必须使用游标的 `fetchone`、`fetchmany` 和 `fetchall` 方法。`fetchone` 和 `fetchall` 方法的作用顾名思义，是获取一行结果或所有的行。而 `fetchmany` 方法接收一个可选的参数来指定返回的行数。每一次都返回多行，它们作为序列（元组列表）的一个序列而返回。你可查看这三个方法的使用来作为前面例子的继续：

```
>>> cursor.fetchone()
(41, 'O', 1)
>>> cursor.fetchmany()
[(6, 'f', 2)]
>>> cursor.fetchmany(4)
[(49, 'W', 4), (35, 'I', 5), (43, 'Q', 6), (37, 'K', 3)]
>>> cursor.fetchall() # All remaining in this case
[(3, 'c', 7), (17, 'q', 8), (29, 'C', 9), (36, 'J', 10), (43, 'Q', 11), (23,
    'w', 12), (49, 'W', 13), (25, 'y', 14), (40, 'N', 15), (50, 'X', 16), (46,
    'T', 17), (51, 'Y', 18), (8, 'h', 19), (25, 'y', 20), (7, 'g', 21), (11, 'k',
    22), (1, 'a', 23)]
```

在一个查询执行完后，你可在结果集中查看在结果集中有游标 `description` 属性的那些行的行信息。`description` 属性是只读的并对结果集的每一行包含一个序列。每一个序列包含一列行集的名字、类型、显示大小、内部大小、精度、标度和可空的信息。前面查询的 `description` 如下所示：

```
>>> cursor.description
[('number', -6, 4, None, None, None, 1), ('letter', 1, 1, None, None, None,
1), ('pkey', 4, 10, None, 10, 0, 0)]
```

表 11-3 列出了全部游标对象的方法和属性。

表 11-3 cursor 对象的方法和属性

方法/属性	描述
<code>description</code>	信息描述了在查询结果中出现的每一列。该信息是一个 7 元素的元组，包括名字、类型码、显示尺寸、内部尺寸、精度、标度和可空否
<code>rowcount</code>	结果中的行数。这只在游标为一静态游标或用动态游标完全遍历结果集后有效
<code>callproc (procedureName, [parameters])</code>	调用存储过程，只应用于那些实现了存储过程的数据库
<code>close ()</code>	关闭游标
<code>execute (statement)</code>	执行一个语句
<code>executemany (statement, parameterList)</code>	执行一个有参数列表的语句。其中，你可在语句中为值使用问号。parameterList 为一个值的元组，它将在语句中被替代
<code>fetchone ()</code>	检索查询的一行

(续)

方法/属性	描述
fetchmany([size])	如无参数，则检索行的 arraysize 数。如果提供了 arg 参数，则它返回了结果行的 arg 数
fetchall()	检索所有剩余的结果行
nextset()	继续下一个结果集。这应用于那些支持多结果集的数据库
arraysize	fetchmany() 返回的行数，不带参数

### 11.5.3 zxJDBC 和元数据

Python DB API 不包含元数据规范，但 zxJDBC 包含有很多带有连接和游标属性的连接元数据。这些属性匹配了本章前面所讨论的 JDBC java.sql.DatabaseMetaData 对象中的 bean 属性。表 11-4 列出了游标字段及其相关的 DatabaseMetaData bean 方法。

表 11-4 zxJDBC 元数据

zxJDBC 属性	DatabaseMetaData 访问符
connection dbname	getDatabaseProductName
connection dbversion	getDatabaseProductVersion
cursor tables (catalog, schemapattern, tablepattern, types)	getTables
cursor columns (catalog, schemapattern, tablenamepattern, columnnamepattern)	getColumns
cursor foreignkeys (primarycatalog, primaryschema, primarytable, foreigncatalog, foreignschema, foreigntable)	getCrossReference
cursor primarykeys (catalog, schema, table)	getPrimaryKeys
cursor procedures (catalog, schemapattern, procedurepattern)	getProcedures
cursor procedurecolumns (catalog, schemapattern, procedurepattern, columnpattern)	getProcedureColumns
cursor statistics (catalog, schema, table, unique, approximation)	getIndexInfo

下面是一个从前面的 MySQL random 数据库中析取元数据的例子：

```
>>> from com.ziclix.python.sql import zxJDBC
>>> url = "jdbc:mysql://localhost/test"
>>> driver = "org.gjt.mm.mysql.Driver"
>>> dbconn = zxJDBC.connect(url, "jyuser", "beans", driver)
>>> dbconn dbname
'MySQL'
>>> dbconn dbversion
'3.23.32'
```

剩下的元数据可通过游标对象来访问。当游标检索信息时，它以内部的方式存储它并等待用户来获取。要查看游标所提供的元数据，必须调用所有的元数据方法，然后可用游标来检索数据：

```
>>> cursor.primarykeys(None, "%", "random")
>>> cursor.fetchall()
[(' ', ' ', 'random', 'pkey', 1, 'pkey')]
>>> cursor.tables(None, None, "%", None)
```

```
>>> cursor.fetchall()
[('', '', 'random', 'TABLE', '')]
>>> cursor.primarykeys('test', '%', 'random')
>>> cursor.fetchall()
[('test', '', 'random', 'pkey', 1, 'pkey')]
>>> cursor.statistics('test', '', 'random', 0, 1)
>>> cursor.fetchall()
[('test', '', 'random', 'false', '', 'PRIMARY', '3', 1, 'pkey', 'A', '23',
None, '')]
```

#### 11.5.4 预编译语句

`executemany()` 游标方法是 Python DB API Java 的预编译语句。实际上，别的被执行的语句也是预编译的，但 `executemany()` 方法使你能对那些在 SQL 语句中的值使用问号。`executemany()` 的第二个参数是在 SQL 语句中代替问号的值的元组：

```
>>> sql = "INSERT INTO random (letter, number) VALUES (?, ?)"
>>> cur.executemany(sql, ('Z', 51))
>>>
>>> # view the row
>>> cur.execute("SELECT * from random where letter='Z'")
>>> cur.fetchall()
[('Z', 51, 24)]
```

#### 11.5.5 错误和警告

`zxJDBC` 中可能导致的异常包括：

- `Error`——这是一个一般的异常。
- `DatabaseError`——这是一个数据库特有的异常。连接对象和所有的连接方法（`connect`, `lookup`, `connectx`）都将导致这种异常。
- `ProgrammingError`——在发生程序错误时，如遗漏参数，错误的 SQL 语句等，所产生的异常。游标对象和查找连接可能会导致这样的异常。
- `NotSupportedError`——当一个方法不能实现时产生的异常。

所有的这些异常都在 `zxJDBC` 包中，所以 `except` 子句应为：

```
>>> try:
...     pass # Assume a method that raises and error is here
... except zxJDBC.DatabaseError:
...     pass # Handle DatabaseError
... except zxJDBC.ProgrammingError:
...     pass # handle ProgrammingError
... except notSupportedError:
...     pass # handle not supported error
... except zxJDBC.Error:
...     pass # Handle the generic Error exception
```

你可通过 `cursor.warnings` 属性来获取警告信息。如果没有警告，所使用的 `cursor.warnings` 属性为 `None`。

`dbexts`

zxJDBC 包的另一个有用的扩展是 dbexts。dbexts 是一个在 Python DB API 2.0 基础上增加了一个抽象层的 Python 模块。使用 BDexts，你可以在配置文件中指定连接信息，然后在定义的连接中使用高级的 dbexts 方法。为了使用 dbexts，与 zxJDBC 一起的模块 dbexts.py 必须加入 sys.path 之中。

配置文件可以是任何文件，但缺省的文件为一个名为 dbexts.ini 的文件，它与文件 dbexts.py 在同一目录之中。要使用另一文件，将该文件的文件名包含到 dbexts 的构造函数中。程序清单 11-4 是一个使用本章的方法定义两个数据库连接的配置文件。

#### 程序清单 11-4 dbexts 配置的例子

```
[default]
name=mysqltest

[jdbc]
name=mysqltest
url=jdbc:mysql://localhost/test
user=jyuser
pwd=beans
driver=org.gjt.mm.mysql.Driver

[jdbc]
name=pstest
url=jdbc:postgresql://localhost/test
user=jyuser
pwd=beans
driver=org.postgresql.Driver
```

一旦配置文件定义以后，你就可以通过实例化 dbexts 类来连接数据库。在下面的例子中，文件 dbexts.ini 放在目前的工作目录之中。你也可以将其放在你放 dbexts.py 模块的 sys.path 的目录之中。

```
>>> from dbexts import dbexts
>>> mysqlcon = dbexts("mysqltest", "dbexts.ini")
>>> psgrscon = dbexts("pstest", "dbexts.ini")
>>>
>>> # execute raw sql and get a list of headers and results
>>> psgrscon.raw("SELECT * from random")
[(['letter', 1, 1, None, None, None, 1], ('number', 4, 11, None, 10, 0, 1)],
 [('A', 4), ('f', 6), ('a', 1)])
>>>
>>> # execute interactive sql
>>> psgrscon.isql("select * from random")

LETTER | NUMBER
-----
A      | 4
f      | 6
a      | 1

3 rows affected
>>>
>>> # Display schema. this works with postgresql, not MySQL
```

```

>>> psgscon.schema("random")
Table
random

Primary Keys

Imported (Foreign) Keys

Columns
letter          bpchar(1), nullable
number          int4(4), nullable

Indices
>>>
>>> # show tables in MySQL 'test' database
>>> mysqlcon.table()

TABLE_CAT | TABLE_SCHEMA | TABLE_NAME | TABLE_TYPE | REMARKS
-----+-----+-----+-----+-----+
| random | TABLE | random | TABLE | random |
1 row affected

```

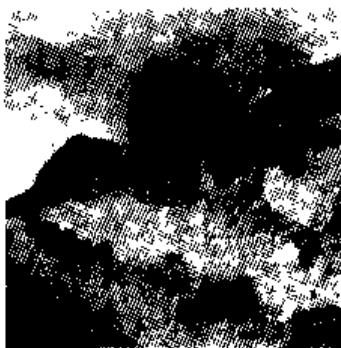
表11-5列出了主要的dbexts方法。一些方法有附加的可选参数，也有一些附加的方法，但已略为超出本书的范围。如要更详细的资料，可查看zxJDBC所带的详尽的文档。

表11-5 dbexts方法

方 法	描 述
_init_ (dbname, cfg, resultformatter, autocommit)	dbexts构造函数。所有的参数都有缺省值，故它们都是可选的。dbname是为在dbexts.ini文件中的连接而指定的名字。如果它不与文件dbexts.py在同一目录则cfg为dbexts.ini文件的位置。resultformatter是一个可调用的参数，它接受一个行的列表作为第一个参数，可选地接受一个头的列表。resultformatter对象是用来显示数据。autocommit参数缺省设置为1或true，如果包含在对构造函数的调用之中也可设置为0
isql (sql, params, bindings, maxrows)	isql方法交互式地执行SQL语句。交互式意味着结果在执行完语句后立即以resultformatter的方式显示，这很像MySQL和pSQL的客户端。除sql语句外所有的参数都有缺省值。sql参数就是SQL语句自身。params参数是一个用来在SQL语句中代替?的值参数的元组。bindings参数允许你绑定一个datahandler。查看关于zxJDBC的文档以获取更多的关于datahandlers的知识。maxrows参数指定返回行的最大数目，其中0或None意味着没有限制
raw (sql, params, bindings)	执行sql语句并返回一个包含头和结果的元组。params和bindings参数与isql方法一样
schema (table, full, sort)	显示表的索引、外键、主键和列。如果full参数非0，则结果包括表的参考键。非0的sort值意味着表的列名必须排序
table (table)	table参数是可选的。在没有table参数时，该方法将显示一个所有表的列表。否则将显示指定表的列

(续)

方 法	描 述
proc (proc)	proc 参数是可选的。在没有 proc 参数时，该方法将显示一个所有程序的列表。否则将显示指定程序
bcp (src, table, where = '(1 = 1)')	该方法将数据从指定的数据库和表 (src 和 table) 复制到目前实例的数据库中
begin (self)	创建一个新的游标
rollback (self)	回滚在一个新的游标创建后执行的语句
commit (self, cursor = None, maxrows = None)	提交在一个新的游标创建后执行的语句
display (self)	使用目前的格式显示结果



## 第 12 章 服务器端 Web 编程

Jython 是怎样适合于 Web 开发的？答案与 Java 一样。Jython 对于那些需要更快速的开发与更大的灵活性的情况尤其合适。服务器端 Java 的 Web 编程主要是以 Servlets 和 Java Server Pages (JSP) 的方式实现，故 Jython 的主要实现方式是 Servlets 和 JSP。当然这并不是全部的方式。Java 的企业包 (J2EE) 在 Web 应用中有极为重要的作用。EJB、JNDI、JDBC 等都是 Java Web 应用的组成部分，这些技术对 Jython 同样可用。然而本章主要介绍的是 Servlets 和 JSP。

当然也有一些 Jython 不适用的技术。CPython 是实现 CGI 脚本的常用语言，而 Jython 不是。在 CGI 中，Web 服务器通过接收请求、启动一个进程来响应。当响应结束后关闭该子进程。可以以这种方式使用 Jython，但这并不是一个好方法。JVM 的启动时间限制了它。Servlet 和 JSP 是持久的，在 Web 请求之间它们依然驻留在内存。

使用 Jython 进行 Web 编程有许多优势。高质量的 Servlet container 很容易使用与配置。Java Servlet 应用能从 Jython 的灵活性和高级语言特性中受益，并影响所提供的 Java 和 Jython 的库函数。高性能的 Servlet container 包括 WebLogic、WebSphere、Tomcat、Jigsaw、Resin 以及 Jetty。已有很多公司安装了上述的 Servlet container，这使 Jython 在很多情况下都能很快使用上。

本章将介绍使用 Jython 的 Servlet 和 JSP。基本内容包括设置一个 Servlet container，基本 Servlet 类，实现 cookie、session 和数据库连接，在 JSP 页面中使用 Jython。在此基础上再建立你自己的 Jython Servlet 映射。本章最后将讨论一些面向实现的内容，包括模板、XML、Cocoon、IBM 的 Bean Scripting Framework 以及 EJB (Enterprise JavaBeans)。

### 12.1 Jython Servlet Container

Jython 能与各种兼容的 Java Servlet container 同时运行，Java Servlet container 有很多产品可供选择。本章使用 Tomcat，它是一个 Servlet 和 JSP 规范所推荐使用的。一些常用的免费的 Servlet container 包括 Apache 的 Jakarta 项目的 Tomcat、Apache 的 JServ、W3C 的 Jigsaw Web 服务器、Mort Bay Consulting 的 Jetty 等。下面对这些工具做一下简述。

Jakarta 的 Tomcat 是 Servlet 和 JSP 规范所推荐使用的标准 Containers，本章使用该服务器。Tomcat 在 <http://jakarta.apache.org/> 上可免费下载，本章写作时使用的最新版本为 3.2.3。该版本的 Tomcat 支持 2.2 Servlet 和 1.1 JSP 规范。在本书出版时读者将可以使用到 Tomcat 4.0，它将支持 2.3 Servlet 和 1.2 JSP 规范。对本章中的 Jython Servlet 我们均使用 Tomcat 3.2.3 来测试。根据 Sun 公司“编写一次，处处可用”的格言，本书所用的例子均可在 2.2 Servlet / 1.1 JSP 规范兼容的 container 上运行。Tomcat 已包括本章所需要的 Servlet 和 JSP 类，故不需要其他的下载。

Apache 的 JServ 是一个为 Apache 创建的 Servlet (2.0 版本)，它已被广泛使用。这是在 Jython 中使用 Servlets 的容易的方法，并且如果像其他的一样你目前的开发已应用了 JServ 这也是一个好的选择。关于 Apache 和 JServ 更多的信息可在 <http://java.apache.org/> 中得到。JServ 需要在 <http://java.sun.com/products/servlet/index.html> 中的独立存在的伴随 Java Servlet Development kit 2.0. Java Server Pages 则需要 <http://java.apache.org/jserv/> 中的内部模型。

Jigsaw 是 W3C 的一个实验性的 Web 服务器。“实验性”可能会导致误解，其实它是相当成熟的。Jigsaw 是一个完全兼容 HTTP/1.1 的 Web 服务器而且它是一个完全用 Java 编写的高速缓冲代理服务器。Jigsaw 也支持 2.2 Servlet 和 1.1 JSP 规范。Jigsaw 及所需要的 Servlet 和 JSP 文件能从 <http://www.w3.org/Jigsaw/> 下载。

Jetty 是一个支持 2.2 Servlet 和 1.1 JSP 规范的紧凑的高效的 Java Web 服务器，它支持 HTTP 1.1，包括 SSL 支持，它能方便地与 EJB 服务器如 JBoss 集成。Jetty 能从 <http://jetty.mortbay.com/> 下载。

这些工具的文档都相当多，它们的安装文档都能从各自的网站上下载。

## 12.2 定义简单的 Servlet 类

本节比较了一个简单的 Java Servlet 和一个简单的 Jython Servlet。通过测试这些 Servlet 描述了如何在 Tomcat Web 应用中安装 `python.jar` 文件。

### 12.2.1 一个简单的 Java Servlet

程序清单 12-1 为最基本的 Java Servlet。

**程序清单 12-1 基本的 Java Servlet**

```
'''// Filename: JavaServlet.java
import javax.servlet.*;
import java.io.*;

public class JavaServlet extends GenericServlet {
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter toClient = res.getWriter();
        toClient.println("<html><body>" +
            "This is a Java Servlet." +
            "</body></html>");
    }
}
```

一个 Servlet API 中基类是必要的，程序清单 12-1 在第 1 行中使用了从 `javax.servlet` 包引入的 `GenericServlet`。在 `GenericServlet` 中，`service` 方法是抽象的，故它必须在任意派生类中实现。这是为响应 `JavaServlet` (程序清单 12-1 中的文件名) 的 Web 请求而调用的方法。客户端的输出是通过从 `ServletResponse` 对象得到的 `PrintWriter` 对象而发送。对二进制数据而言，这可以是 `OutputStream`。

### 12.2.2 一个简单的 Jython Servlet

与程序清单 12-1 相比，实现一个 Jython Servlet 在引入（import）、继承与实现方面应是同样的，只是用 Jython 语法实现而已。程序清单 12-2 为一个例子。

#### 程序清单 12-2 一个简单的 Jython Servlet

---

```
# Filename: JythonServlet.py
from javax import servlet
import random # Not used, just here to test module imports

class JythonServlet(servlet.GenericServlet):
    def service(self, req, res):
        res.setContentType("text/html")
        toClient = res.getWriter()
        toClient.println("""<html><body>
            This is a Servlet of the Jython variety.
        </body></html>""")
```

---

程序清单 12-2 是 GenericServlet 的一个派生类，正如程序清单 12-1 中的 Java 所对应的。它也在需要时实现 GenericServlet 的 service() 方法。程序清单 12-2 的语法差别包括类定义后面用来指定其超类的圆括号，throws 语句的省略，在 service 方法中直接的 self 参数，当然也包括分号的省略和直接的类型声明。另外，Jython 的引入语句不同。正如前面提到的，我们建议不要使用 from module import \* 的语法。相反，程序清单 12-2 使用了 parent 包。在程序清单 12-2 中额外的引入包括 random 模块。该模块并未实际使用，仅用来测试模块的引入。

### 12.2.3 测试 Java 和 Jython Servlet

测试程序清单 12-1 和 12-2 的 Servlets 需要安装 Tomcat。程序清单 12-2 的 Jython Servlet 需要在 Tomcat 中另外包含 jython.jar 文件。本节首先描述了安装 Tomcat 的步骤，然后谈了上面讨论的两个 Servlet 的安装和测试。

#### 1. 安装 Tomcat

第一步是从 <http://jakarta.apache.org> 下载 Tomcat。本节主要讨论在独立模式下安装 Tomcat 的二进制版本。推荐的版本为 jakarta-tomcat-3.2.3。根据你的平台下载 zip 或 tar.gz 文件。

下一步，将下载的文档 unzip 或 untar 到你有足够使用权限的目录之中。如果你将文档解压到 c:\jakarta-tomcat-3.2.3 目录中，它将变成 Tomcat 的主目录。如果你使用 /usr/local/jakarta-tomcat-3.2.3，它将变成 Tomcat 的主目录。你必须设置一个 Tomcat 主目录的环境变量。对 Windows 系统中的 c:\jakarta-tomcat-3.2.3，在你的环境设置中加入：

```
set TOMCAT_HOME=c:\jakarta-tomcat-3.2.3
```

对 \* nix 系统中的 /usr/local/jakarta-tomcat-3.2.3，在你的环境设置中加入：

```
export TOMCAT_HOME=/usr/local/jakarta-tomcat-3.2.3
```

如果你不设置 TOMCAT\_HOME 环境变量，你必须从 TOMCAT 的主目录或 bin 目录中启动

TOMCAT。

下一步，将环境变量 JAVA\_HOME 设置为你的 JDK 安装的根目录。下面是一个使用 JDK 1.3.1 的例子：

```
# on Windows
set JAVA_HOME=c:\jdk1.3.1
# bash (*nix) setup
export JAVA_HOME=/usr/java/jdk1.3.1
```

安装结束了。你现在能使用针对你所使用的平台的启动脚本来启动 Tomcat：

```
# Windows
%TOMCAT_HOME%\bin\startup.bat
# bash (*unix)
$TOMCAT_HOME/bin/startup.sh
```

你必须在 Tomcat 服务器启动时查看屏幕上的启动信息。下面一行是需要特别注意的：

```
date time - PoolTcpConnector: Starting HttpConnectionHandler on 8080
```

这指明了你用来连接到 Servlet container 的端口号：8080 是缺省的。当你看到该信息时，Tomcat 正在运行并接受端口 8080 的连接。

当你想要中止 Tomcat 时，使用下面脚本：

```
# Windows
%TOMCAT_HOME%\bin\shutdown.bat
# bash (*nix)
$TOMCAT_HOME/bin/shutdown.sh
```

Servlet 2.2 规范为 Web 应用指定了一个目录层次，该层次以 %TOMCAT\_HOME%\webapps 为开始。在该目录中的文件夹为 Web 应用或场景，每一个都对应一个特定的目录层次。本章使用名叫 jython 的场景。jython 场景所需要的目录结构如下：

%TOMCAT_HOME%\webapps\	
%TOMCAT_HOME%\webapps\jython	The context's root
%TOMCAT_HOME%\webapps\jython\WEB-INF	
%TOMCAT_HOME%\webapps\jython\WEB-INF\classes	Servlet classes
%TOMCAT_HOME%\webapps\jython\WEB-INF\lib	Library archives

你必须在运行这些例子之前创建这些目录。如果你重启 Tomcat，你将在其重启时看到一些额外的行的信息。下面一行确认 Tomcat 已装入新的场景：

```
date time - ContextManager: Adding context Ctx( /jython )
```

## 2. 安装 Java Servlet

为安装程序清单 12-1 中的 Java Servlet，首先 将文件 JavaServlet.java 放入目录 %TOMCAT\_HOME%\webapps\jython\WEB-INF\classes 中。该目录为类文件的根目录。由于 JavaServlet 不在某个包中，所以它属于 classes 目录的根目录。注意到程序清单 12-1 并不在某个包之中；如果你选择指定某个包，如例子中所示的包 demo，你就必须将编译好的类文件放入 classes\demo 目

录中以便保持与 Java 类目录结构的一致性。这是那些在包中的 Servlet 应注意的，与我们这儿所讨论的例子无关。

从上述目录，利用下列命令编译文件 JavaServlet.java：

```
javac -classpath %TOMCAT_HOME%\lib\servlet.jar JavaServlet.java
```

在编译了 JavaServlet 后，你可以浏览它。首先启动 Tomcat 服务器，然后将你的浏览器定位到 <http://localhost:8080/jython/servlet/JavaServlet>。你就可以看到最简单的字符串信息 “This is a Java Servlet”。

### 3. 安装 Jython Servlet

有两种方法使用 Jython Servlet。其一是用 jythonc 编译 Servlet 并将所得到的类文件放入 \$TOMCAT\_HOME%\jython\WEB-INF\classes 目录中。另一办法是用 Jython 的 PyServlet 映射类。本节使用 jythonc。PyServlet 映射类通常是一个更好的办法，但使用 jythonc 编译的 Servlets 现在也同样可行。

在 Tomcat 中安装用 Jythonc 编译的 Tomcat 需要三步：

- 1) 利用 jythonc 编译 Jython Servlet 模块。
- 2) 将 jython.jar 文件加入 Web 应用中。
- 3) 使 Jython 的 lib 目录的各模块能为 Jython Servlets 所使用。

### 4. 利用 jythonc 编译一个 Jython Servlet

使用 jythonc 编译需要 servlet.jar 文件在 classpath 之中。servlet.jar 文件包含 javax.Servlet.\* 类和包，它可在 Tomcat 的 lib 目录中 (%TOMCAT\_HOME%\lib) 找到。如果你在没有 servlet.jar 文件的 classpath 之中用 jythonc 编译一个 Servlet 时，编译将没有错误或警告；但当你运行上述编译的 Servlet 时，你将会得到一个 java.lang.ClassCastException 异常（至少在本书写作时的 jythonc 编译器是如此）。

将程序清单 12-2 中的 JythonServlet.py 文件放至目录 %TOMCAT\_HOME%\jython\WEB-INF\classes 中。保证你的环境变量 CLASSPATH 实际上包含 Servlet.jar，然后通过在目录 %TOMCAT\_HOME%\jython\WEB-INF\classes 中使用如下命令用 jythonc 将 Jython 代码转换成 Java 类：

```
jythonc -w . JythonServlet.py
```

通过选项-w 指定目前的工作目录消除了从 jpywork 目录中拷贝产生类文件的必要性。在 classes 目录中必须有两个类文件。注意编译好的 Jython 文件至少有两个相关的类文件。用 jythonc 来编译 JythonServlet.py 的文件为 JythonServlet.java、JythonServlet.class 和 JythonServlet\$PyInner.class。这些类文件均必须使用 Servlet，故它们都必须在 WEB-INF\classes 目录中。

在用 jythonc 编译的过程中，必须仔细查看如下行的内容：

```
Creating .java files:  
  JythonServlet module  
    JythonServlet extends javax.servlet.GenericServlet
```

如果你没注意上面的内容，而使某些东西错了，Servlet 将不会运行。再检查一次 CLASSPATH 和 setup 的选项并编译文件，直到你能看到上述内容为止。

### 5. 将 jython.jar 加入 classpath 之中

所有的 Jython Servlet 必须能访问 jython.jar 文件中的类。有下列三种方法将 jython.jar 文件加入 Tomcat 的 classpath 之中：

- 将 jython.jar 加入场景的 lib 目录。这是最好的方法。
- 将 jython.jar 加入 Tomcat 的 lib 目录。这是次好的方法。
- 将 jython.jar 留在 Jython 的安装目录之中，并在运行 Tomcat 时将其加入 classpath 之中。这是可行的，但不如将其加入场景的 lib 目录那样好。

本书推荐的办法是将 jython.jar 文件加入场景的 lib 目录。场景必须包括目录 context\WEB-INF\lib。对本章中所使用 jython 场景，目录为 %TOMCAT\_HOME%\webapps\jython\WEB-INF\lib。类存档文件如 jython.jar 属于这个目录。我们推荐这样使用是由于它使 Web 应用能自恰。只要 Web 应用需要访问它自己场景以外的其他文件，在别的服务器上的存档、打包和安装都变得很麻烦。我们强烈推荐使所有的 Web 应用都自恰，除非你认为它没有必要。

你也可将文件 jython.jar 放入 Tomcat 的 lib 目录 (%TOMCAT\_HOME%\lib) 中。该目录中的 Jar 文件会自动加入 classpath 之中。然而，这是这三个方法中最差的方法。你可以减少复制的 jython.jar 文件，但你的 Web 应用不再自恰。另外，像第三种方法一样你不能自动访问 Jython 的 lib 目录。

第三个方法是将 jython.jar 文件留在 Jython 的安装目录中，并在启动 Tomcat 之前将它加入 classpath 之中。这也消除复制的 jython.jar 文件。另外它还有访问注册文件和 Jython 的 lib 目录的优点。注意注册文件在 python.home 目录中，如果没有 python.home 特性就在 jython.jar 文件的目录中。这样将 jython.jar 文件留在 Jython 的安装目录中优于将其放在 Tomcat 的 lib 目录中。这儿要再次强调，建议保持自恰的场景。

### 6. 使 Jython 的 lib 目录对 Servlet 可用

有三种方法来使 Jython 的 lib 目录对 Servlet 可用：

- 设置 python.home 特性。
- 冻结所需要的模块。
- 每一个 Servlet 将模块位置明显地加到 sys.path 中。

如果你已选择将 jython.jar 文件留在 Jython 的安装目录中，则不需要额外的步骤来获得对 Jython 的 lib 目录的访问。如果你选择将 jython.jar 文件加入场景的 lib 目录，你就必须设置 python.home 特性，将目录明显地加到 sys.path 中或在你的 Jython Servlet 能使用 Jython 的模块库前冻结所需要的模块。

要设置 python.home 特性，你可设置 TOMCAT\_OPTS 环境变量。在做之前，你必须决定模块放在哪个位置。另外，最好的方法是创建一个自恰的 Web 应用。一个好的建议是在场景的 WEB-INF 目录之中创建一个额外的目录。由于本节的缘故，这个目录名叫 jylib。创建目录 %TOMCAT\_HOME%\webapps\jython\WEB-INF\jylib。由于 Jython 在 python.home 中寻找 lib 目录，继续创建目录 %TOMCAT\_HOME%\webapps\jython\WEB-INF\jylib\Lib。将任何所需要的模块放入该目录中，并指定 jylib 目录作为 python.home。这儿是一些设置 TOMCAT\_OPTS 给合适的 python.home 属性的例子：

```
# Windows
set TOMCAT_OPTS=-Dpython.home=%TOMCAT_HOME%\webapps\jython\WEB-INF\jylib

# bash (*nix)
export TOMCAT_OPTS=-Dpython.home=$TOMCAT_HOME/webapps/jython/WEB-INF/jylib
```

注意有些 Windows 的版本在环境字符串中不准出现等号。在这种情况下你必须编辑 tomcat.bat 文件来包含 python.home 的设置。

也可以通过直接将模块目录加入 sys.path 来启动 Servlets。问题在于这经常需要依赖于机器的直接的路径，这就限制了跨平台的移植性。下面是当你想直接加入 sys.path 时在 Servlet 的头部将会出现的：

```
import sys
libpath = "c:/jakarta-tomcat_3.2.3/webapps/jython/WEB-INF/jylibs/Lib"
sys.path.append(libpath)
```

另一个使 Jython 模块可用的方法是冻结它们。jythonc -deep 选项编译所有需要的模块，从而使 python.home 和 Jython 的 lib 目录变得不重要。要冻结程序清单 12-2 的 JythonServlet.py 文件，可从 Jython 场景的 classes 目录中使用下列命令行语句即可：

```
jythonc -w . --deep JythonServlet.py
```

JythonServlet 及其所需要的所有模块的类文件现在都在 Jython 场景的 classes 目录中。换句话说，Servlet 及其模块都安装在一个自恰的 Web 应用之中。注意按该方法编译其他 Jython Servlet 将会覆盖前面编译的所有模块。这意味着在更新模块时要非常小心，一个新的版本会对老的 Servlet 有副作用。使用--deep 选项编译会产生很多文件，但所产生的 \*.java 文件可在编译完成后删除。

由于模块的变化很频繁，冻结对于保证一个完全自恰的 Web 应用是很有用的。你不用设置 python.home 属性。一个这样设置的 Web 应用能够简单地作为 \*.war 文件归档，在不需要额外的安装步骤后就能插入其他相容的服务器之中。

## 7. 测试 Jython Servlet

在用 jythonc 在 classpath 中的 jython.jar Jython 的模块编译程序清单 12-2 中的 Servlet，你就能直接查看它。将你的浏览器定位在 <http://localhost:8080/jython/servlet/jythonServlet>。你将会看到简单的信息 “This is a Servlet of the Jython variety”。如果你看到的信息不是这样的，那你将看到如下三种错误信息中的一种。如果在编译 JythonServlet 时 Servlet.jar 文件不在 classpath 中，你将会看到 ClassCastException。如果文件名和类名不同，即使是大小写不同，你也将看到 ClassCastException。如果用 jythonc 编译 Servlet 时所产生的某一个类文件在场景的 classes 目录中，你将会看到 AttributeError。如果 Jython 的模块是无效的，你将会看到 ImportError。

## 12.3 GenericServlet 的更多内容

程序清单 12-2 从 javax.Servlet.GenericServlet 中继承而来。正如其名字一样，该类意味着一个不针对特定协议的 Servlet 类。HttpServlet 类由于其针对 HTTP 协议而使其在 web 开发上更加常

见。然而 GenericServlet 是 HttpServlet 的超类，知道它的方法是很重要的。表 12-1 为对 GenericServlet 的 Java 方法特性的总结，并举例说明了 GenericServlet 的 Jython 派生类如何使用这些方法。表中并未包含所有的方法-而仅包含那些 Jython 中易于使用的方法。表 12-1 将方法按两类列出：

- 在 Jython 派生类中覆盖的方法，以 def 语句标出。
- 从 Jython 派生类中调用超类的方法，以 self 语句标出。

表 12-1 GenericServlet 方法

Java 特性	Jython 派生类中的用法
派生类的方法	
public void init(ServletConfig config) throws ServletException;	def init(self, config):
public void destroy();	def destroy(self):
public abstract void service( ServletRequest request, ServletResponse response) throws ServletException, IOException;	def service(self, request, response):
public String getServletInfo();	def getServletInfo(self): return "Info string"
使用 SELFMETHOD 语法的方法：	
public void log(String message);	self.log("message")
public ServletConfig getServletConfig();	config = self.getServletConfig()
public java.util.enumeration getInitParameterNames();	nameList = self.getInitParameterNames()
public String getInitParameter(String name)	param = self.getInitParameter( 'paramName')
public ServletContext getServletContext()	context = self.getServletContext

你已在程序清单 12-2 中看到重载 public abstract void service () 方法的例子。其他对重载有用的方法将在下面点击记数 (HitCounter) 的 Servlet 中使用。

点击记数 (HitCounter) 的 Servlet 像在编程教学中的例子 Hello World 一样是必须的。这儿用的名字是一个惯例。程序清单 12-3 的点击记数的 Servlet 说明了 init、service 和 destroy 方法的使用。这些方法的使用将在下面的程序清单中讲述。

### 程序清单 12-3 HitCounter Servlet

```
# filename: HitCounter.py
from javax import servlet
from time import time, ctime
```

```

import os

class HitCounter(servlet.GenericServlet):
    def init(self, cfg=None):
        if cfg:
            servlet.GenericServlet.init(self, cfg)
        else:
            servlet.GenericServlet.init(self)

    # Construct a path + filename to file storing hit data
    contextRoot = self.servletContext.getRealPath(".")

    self.file = os.path.join(contextRoot, "counterdata.txt")

    if os.path.exists(self.file):
        lastCount = open(self.file, "r").read()
        try:
            # within 'try' just in case the file is empty
            self.totalHits = int(lastCount)
        except:
            self.totalHits = 0
    else:
        self.totalHits = 0

    def service(self, req, res):
        res.setContentType("text/html")
        toClient = res.getWriter()
        toClient.println("<html><body>")
        toClient.println("Total Hits: %i<br>" % (self.totalHits,))
        self.totalHits += 1
        toClient.println("</body></html>")

    def destroy(self):
        f = open(self.file, "w")
        f.write(str(self.totalHits))
        f.close()

```

将 HitCounter.py 放到场景的 classes 目录中（%TOMCAT\_HOME%\webapps\WEB-INF\classes），并在同一目录中用 jythonc 编译：

```
jythonc -w . --deep HitCounter.py
```

将你的浏览器定位在 <http://localhost:8080/jython/servlet/HitCounter>，你就可以测试 HitCounter Servlet 了。在你第一次访问该 URL 时，你应看到消息“Total Hits: 0”。每一次点击都会相应地增加计数。如果你用%TOMCAT\_HOME%\bin\shutdown.bat（对 bash 为\$TOMCAT\_HOME/bin/shutdown.sh）关闭 Tomcat 并重新启动 Tomcat，计数器将从关闭前的最后一次计数开始。

三个方法 init、service 和 destroy 对 Servlet 很重要，每一个都与 Servlet 的生命周期中的一个阶段对应。Servlet 第一次装载并初试化：init() 方法。Servlet 然后处理客户端的请求：service() 方法。Servlet 处理请求直到被卸载或从 Servlet container() 中消除时：destroy 方法。

### 12.3.1 init (ServletConfig) 方法

init() 方法在 Servlet 启动时装载，并只在那时调用一次。这使得它对一些时间不敏感的任务如设置数据库连接、编译常规表达式或装载一些辅助文件有益。在程序清单 12-3 中包含 init() 方法的目的在于建立存储点击信息的文件，并将该文件中的最后一个存储的整数设置为计数器的值。这种实现方法保证除非 Servlet 重新启动且 counter.txt 文件丢失，否则点击计数器的值不会被复位到 0。

Servlet API 的 2.2 版本有两种 init() 方法：无参数版本和能接收 ServletConfig 类的版本。程序清单 12-3 使用了这个对象的 cfg 参数更重要的是记住当 Jython 覆盖一个 Java 方法时，它用那个名字覆盖了所有的方法。这意味着在 Jython Servlet 中定义一个 init() 方法覆盖了无参数 init() 和 init (ServletConfig cfg) 方法。程序清单 12-3 通过指定一个对 cfg 变量的 None 的缺省参数来处理两种方法的功能，然后当调用超类的 init() 方法时如果 cfg 变量被用到则测试的 None 被决定了。

### 12.3.2 service (ServletRequest, ServletResponse) 方法

该方法是为响应客户端的请求而调用的方法。由于在 GenericServlet 中该方法为抽象，故 Servlet 首先必须定义此方法。该方法的参数为 ServletRequest 和 ServletResponse 对象。ServletRequest 包括客户端通过请求送到服务器端的信息，包括请求头、请求方法和请求参数。ServletResponse 包括按合适的 mime 编码方式传递给客户端的响应流。

### 12.3.3 destroy() 方法

该方法在 Servlet 被关闭或卸载时调用。该方法包括一些清除代码，也包括关闭数据库连接、删除和关闭文件等。

程序清单 12-3 中的 destroy 方法将计数器的值写到文件中去使得当 Servlet 卸载时信息不会丢失。这不是保持持久的最好方法。对这种方法而言，只有在服务器正常关闭时计数器的值才会被保存。如果 Servlet 容器被非正常地关闭，那 destroy 方法将不会被调用。当你使用 Tomcat 时，你通常使用启动和关闭的 shell 脚本来启动和关闭服务器。这些都是启动和关闭 Tomcat 的正常方法。然而如果你采用“Ctrl + C”的非正常方法来关闭 Tomcat，正常的关闭次序被跳过。另外一个意外终止 Tomcat 的错误也会导致同样的错误，计数器的值将不再保存。高质量的持久要求在数据存储中的赋值和提交都在事务处理的框架内进行。

## 12.4 HttpServlet

GenericServlet 因为对所有会话型的协议都有用而显得十分通用，然而 Web 开发大都是 HTTP 协议的。在 HTTP 协议中会话，javax.Servlet.http.HttpServlet 是最好扩展的。HttpServlet 是 GenericServlet 的派生类。所以在扩展 HttpServlet 时，从 GenericServlet 继承的三个方法 init、service、destroy 都有效。

HttpServlet 对每个 HTTP 方法定义了一个方法。这些方法是：doGet、doPost、doPut、doOp-

tions、doDelete 和 doTrace。当 Tomcat 接受一个 GET 型的客户端请求时，请求的 Servlet 的 doGet() 方法被调用来响应该客户。另外，HttpServlet 有一个 HTTP 特有的 service 方法的版本。HttpServlet service 方法与 GenericServlet 版本相比仅有的改变是 HTTP 特有的请求和响应对象都为参数 (javax.servlet.http.HttpServletRequest 和 javax.servlet.http.HttpServletResponse)。

为通过 HttpServlet 派生类来编写 Servlet，需要实现每个 HTTP 方法（如 doGet 或 doPost），或定义 service 方法。HttpServlet 类也有一个 getLastModified 方法可覆盖，你可用来返回一个表示 Servlet 或相关数据更新的最后时间（从 1970 纪元开始计算的毫秒数）。该信息通过高速缓存而使用。

#### 12.4.1 HttpServlet 方法

表 12-2 包括所有 javax.servlet.http.HttpServlet 类的方法及其在 Jython 中使用的例子。所有覆盖的方法均在 Jython 中有 def 语句，所有在超类中调用的方法均以 self 开头。表 12-2 的 Java 特性不包含返回值或允许的修饰符。对访问权限而言，所有的方法除 service (ServletRequest req, ServletResponse res) 外都保护起来且没有权限修饰符 (package private)。除 getLastModified 方法外所有的返回值都为 void。getLastModified 方法返回一个表示从纪元开始计算的毫秒数的长整型。

表 12-2 HttpServlet 方法

Java 函数	Jython 派生类中的用法
doDelete (HttpServletRequest req, HttpServletResponse resp)	def doDelete (self, req, res):
doGet (HttpServletRequest req, HttpServletResponse resp)	def doGet (self, req, res):
doHead (HttpServletRequest req, HttpServletResponse resp)	def doHead (self, req, res): 在 J2EE1.3 版本中
doOptions (HttpServletRequest req, HttpServletResponse resp)	def doOptions (self, req, res):
doPost (HttpServletRequest req, HttpServletResponse resp)	def doPost (self, req, res):
doPut (HttpServletRequest req, HttpServletResponse resp)	def doPut (self, req, res):
doTrace (HttpServletRequest req, HttpServletResponse resp)	def doTrace (self, req, res):
getLastModified (	
HttpServletRequest req)	def getLastModified (self, req):
service (HttpServletRequest req, HttpServletResponse resp)	def service (self, req, res):
service (ServletRequest req, ServletResponse res)	def service (self, req, res):

接受 HTTP 特有的请求和响应的 service 方法将这些请求重新分配到相应的 do \* 函数上（如果它未被覆盖）。接受通用 Servlet 请求和响应的 service 方法将这些请求重新分配 HTTP 特有的 service 方法上。重新分配使得在实现 Servlet 映射时 service 方法有效。

#### 12.4.2 HttpServlet 例子

程序清单 12-4 展示了一个为 javax.servlet.http.HttpServlet 派生类的 Servlet。该例子通过实现 doGet 和 doPost 方法来说明这些方法如何通过客户端接受的请求的类型来调用这些方法。Web 客户端在没有确认的情况下不允许重复 POST 操作。由于这个原因，数据库更新、定单提交等等必须用 doPost 方法，而表格可安全地存在 doGet 之中。

程序清单 12-4 中的 get\_post.py 文件说明了如何通过 HttpServletRequest (req) 的 getParameterNames 和 getParameterValues 来获取参数的名字和值。从 getParameterNames 返回的列表为 java.util.Enumeration，而 doPost 方法用来显示请求的参数。在 Jython 中可以使用 for x in list; 从 getParameterNames 返回的列举。注意 getParameterValues() 方法是多值的。正如在隐藏表单域中所列出的一样，参数可有多个值。为了防止 Servlet 中 doGet 和 doPost 方法的冗余，程序清单 12-4 增加了一个 \_params 方法。该方法在 HttpServlet 及其基类中都未定义，且由于没有 Java 类调用它，不需要 @sig 字符串。该方法的目的仅仅是为了只在一个地方保持相同的操作。

程序清单 12-4 用 HttpServlet 实现 Servlet

```
#file get_post.py
from time import time, ctime
from javax import servlet
from javax.servlet import http

class get_post(http.HttpServlet):
    head = "<head><title>Jython Servlets</title></head>"
    title = "<center><H2>%s</H2></center>"

    def doGet(self, req, res):
        res.setContentType("text/html")
        out = res.getWriter()

        out.println('<html>')
        out.println(self.head)
        out.println('<body>')
        out.println(self.title % req.method)

        out.println("This is a response to a %s request" %
                   (req.getMethod(),))
        out.println("<P>In this GET request, we see the following " +
                   "header variables.</P>")
        out.println("<UL>")
        for name in req.headerNames:
            out.println(name + " : " + req.getHeader(name) + "<br>")
        out.println("</UL>")

        out.println(self._params(req))
        out.println("")
```

```

<P>The submit button below is part of a form that uses the
    "POST" method. Click on this button to do a POST request.
</P>"")}

out.println('<br><form action="get_post" method="POST">' +
    '<INPUT type="hidden" name="variable1" value="one">' +
    '<INPUT type="hidden" name="variable1" value="two">' +
    '<INPUT type="hidden" name="variable2" value="three">' +
    '<INPUT type="submit" name="button" value="submit">')

out.println('<br><font size="-2">time accessed: %s</font>' %
    % ctime(time()))
out.println('</body></html>')

def doPost(self, req, res):
    res.setContentType("text/html");
    out = res.getWriter()

    out.println('<html>')
    out.println(self.head)
    out.println('<body>')
    out.println(self.title % req.method)

    out.println("This was a %s<br><br>" % (req.getMethod(),))
    out.println(self._params(req))
    out.println('<br> back to <a href="get_post">GET</a>')
    out.println('<br><font size="-2">time accessed: %s</font>' %
        % ctime(time()))
    out.println('</body></html>')

def _params(self, req):
    params = "Here are the parameters sent with this request:<UL>"
    names = req.getParameterNames()

    if not names.hasMoreElements():
        params += "None<br>"
    for name in names:
        value = req.getParameterValues(name)
        params += "%s : %r<br>" % (name, tuple(value))
    params += "</UL>"
    return params

```

在将文件 get\_post.py 放到 %TOMCAT\_HOME/webapps/jython/WEB-INF/classes 目录中之后，使用如下命令编译它：

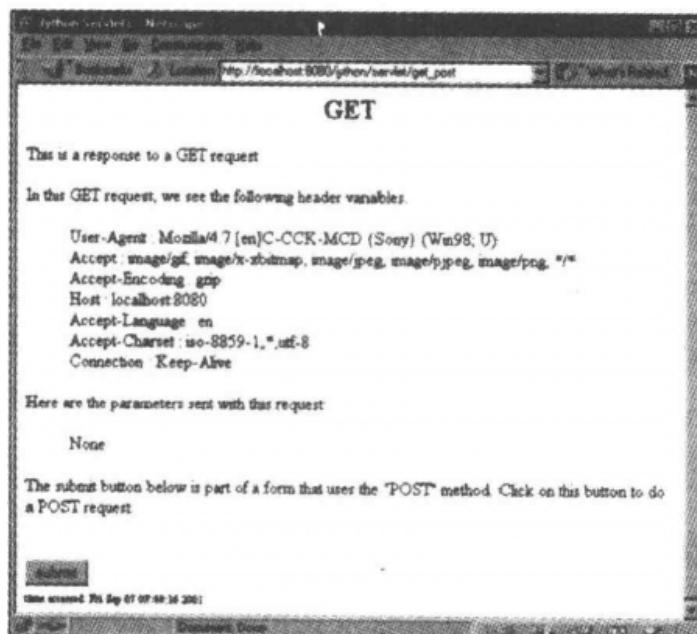
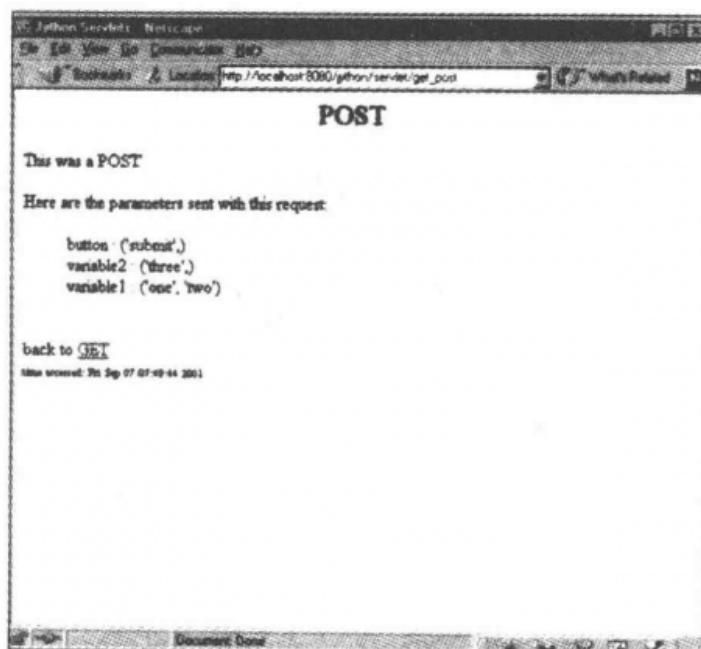
```
javac -w . --deep get_post.py
```

如要测试，只要将你的浏览器定位在 http://localhost: 8080/jython/servlet/get-post，你就可以看到图 12-1 所示的浏览器窗口。

在该例子中 GET 操作的参数为 None，但测试其他的 doGet 方法中的参数时必须在 URL 后加入一些参数（如 http://localhost: 8080/servlet/get-post? variable1 = 1&variable2 = 2）。

由于在第一个视图底部的 Submit 按钮是 POST 实现的表格的一部分，点击 Submit 按钮将执

行同一个 Servlet 的 `doPost` 方法。`doPost` 方法的结果将如图 12-2 所示。

图 12-1 从 `get_post.py` 来的 GET 视图图 12-2 从 `get_post.py` 来的 POST 视图

#### 12.4.3 HttpServletRequest 和 HttpServletResponse

与客户端的通信可通过 `HttpServletRequest` 和 `HttpServletResponse` 对象。这些都是从客户端和到客户端的文字上的请求流的抽象。这些对象中的每一个将增加高级的、HTTP 特有的方法来简化请求和响应。

表 12-3 是 `HttpServletRequest` 对象内这些方法的列表。大部分方法都是组件特性的访问器，这意味着你能通过 Jython 的自动组件特性来调用它们。这可能不像在 GUI 编程中有那样大的好处，因为这儿没有机会在关键字参数的方法中来利用这种简便方法。表 12-3 列出了 `HttpServletRequest` 对象的方法和组件特性。

表 12-3 `HttpServletRequest` 方法和属性

方法和属性	描述
方法： <code>getAuthType()</code>	返回一个描述认证类型的字符串 (PyString)
属性名： <code>AuthType</code>	如果用户未认证，值为 None
方法： <code>getContextPath()</code>	返回一个描述确认场景请求的路径信息的字符串 (PyString)
属性名： <code>contextPath</code>	
方法： <code>getCookies()</code>	返回所有的作为一列 <code>javax.servlet.http.Cookie</code> 对象的客户端请求发送的 cookie
属性名： <code>cookies()</code>	

(续)

方法和属性	描述
方法： getDateHeader (name)	检索作为长类型的特定头的值
方法： getHeader (name)	返回作为字符串 (PyString) 的特定头的值
方法： getHeaderNames ( )	返回包含在请求之中作为列举的所有头名
属性名： headerNames	
方法： getHeaders (name)	返回包含作为列举的特定的头名
方法： getIntHeader (name)	检索一个作为 Java int 的特定头的值，在 Jython 中 转变为 PyInteger
方法： getMethod ( )	返回请求类型，产生一个字符串
属性名： method	
方法： getPathInfo ( )	客户端发送的外部路径信息
属性名： pathInfo	
方法： getPathTranslated	返回客户端请求的外部路径信息中得到的实际路 径
属性名： pathTranslated	
方法： getQueryString ( )	返回从客户端请求的查询字符串（路径后的字符 串）
属性名： queryString	
方法： getRemoteUser ( )	返回客户端的登录名。如果客户端未认证，返回 空值
属性名： remoteUser	
方法： getRequestedSessionId ( )	返回客户端 session ID
属性名： requestedSessionId	
方法： getRequestURI ( )	返回协议名和查询字符串之间的那部分
属性名： requestURI	
方法： getServletPath ( )	返回指定目前 Servlet 的 URL 部分
属性名： servletPath	
方法： getSession ( )	返回目前的 session，如果必要则创建一个。Session 是 javax.servlet.http.HttpSession 的实例
属性名： session	

(续)

方法和属性	描述
方法： getSession (create)	如果存在，返回目前的 session。如果不存在且创建值为真，则创建一个新的 session
方法： getUserPrincipal ( )	返回一个带有目前认证信息 userPrincipal 的 java.security.Principal 对象
属性名： userPrincipal	
方法： isRequestedSessionIdFromCookie( )	根据目前的 session ID 是否从一个 cookie 中来而返回 1 或 0
方法： isRequestedSessionIdFromURL ( )	根据目前的 session ID 是否从请求 URL 字符串中来而返回 1 或 0
方法： isRequestedSessionIdValid ( )	根据请求的 session ID 是否还有效而返回 1 或 0
方法： isUserInRole (role)	根据用户是否在角色表中列出而返回 1 或 0

HttpServletResponse 对象用来将 mime 编码的信息流送回客户端。HttpServletResponse 定义了额外的在通用的 ServletResponse 中不存在的 HTTP 特有的方法。HttpServletResponse 对象的方法在表 12-4 中列出。尽管 Jython 在 HttpServletRequest 对象中增加了很多自动组件特性，但在 HttpServletResponse 对象中只有一个：status。

表 12-4 HttpServletResponse 方法和特性

方法和特性	描述
addCookie (cookie)	增加一个 cookie 到响应中
addDateHeader (headerName, date)	增加一个有日期（长整型）值的头名字
addHeader (headerName, value)	增加一个头名字和值
addIntHeader (headerName, value)	增加一个有整数值的头名字
containsHeader (headerName)	根据是否为指定的头文件返回 1 或 0
encodeRedirectUrl (url)	为 sendRedirect 方法编一个 URL。对 2.1 版本及更高的版本，则使用 encodeRedirectURL
encodeRedirectURL (url)	对 sendRedirect 方法编一个 URL
encodeURL (url)	编一个 URL，其中加入 session ID
sendError (sc)	利用状态码发送一个错误
sendError (sc, msg)	利用指定的状态码和信息发送一个错误
sendRedirect (location)	对一个指定的位置发送一临时的 redirect
setDateHeader (headerName, date)	设置头名字为一个指定的日期（长整型）值
setHeader (headerName, value)	设置头名字为一个指定的值
setIntHeader (headerName, value)	设置头名字为一个指定的整数值
setStatus (statusCode)	设置响应状态码
status	

HttpServletResponse 类也包含对应的标准的 HTTP 响应码的域。你可通过 sendError (int) 和 setStatus (int) 来使用它们。表 12-5 列出了这些状态码的数字和错误代码。

表 12-5 HttpServletResponse 状态错误码

码	状态
100	SC_CONTINUE
101	SC_SWITCHING_PROTOCOLS
200	SC_OK
201	SC_CONTINUE
202	SC_ACCEPTED
203	SC_NON_AUTHORITATIVE_INFORMATION
204	SC_NO_CONTENT
205	SC_RESET_CONTENT
206	SC_PARTIAL_CONTENT
300	SC_MULTIPLE_CHOICES
301	SC_MOVED_PERMANENTLY
302	SC_MOVED_TEMPORARILY
303	SC_SEE_OTHER
304	SC_NOT_MODIFIED
305	SC_USE_PROXY
400	SC_BAD_REQUEST
401	SC_UNAUTHORIZED
402	SC_PAYMENT_REQUIRED
403	SC_FORBIDDEN
404	SC_NOT_FOUND
405	SC_METHOD_NOT_ALLOWED
406	SC_NOT_ACCEPTABLE
407	SC_PROXY_AUTHENTICATION_REQUIRED
408	SC_REQUEST_TIMEOUT
409	SC_CONFLICT
410	SC_GONE
411	SC_LENGTH_REQUIRED
412	SC_PRECONDITION_FAILED
413	SC_REQUEST_ENTITY_TOO_LARGE
414	SC_REQUEST_URI_TOO_LONG
415	SC_UNSUPPORTED_MEDIA_TYPE
500	SC_INTERNAL_SERVER_ERROR
501	SC_NOT_IMPLEMENTED
502	SC_BAD_GATEWAY
503	SC_SERVICE_UNAVAILABLE
504	SC_GATEWAY_TIMEOUT
505	SC_HTTP_VERSION_NOT_SUPPORTED

## 12.5 PyServlet

对 Jython 程序员而言，HttpServlet 编程的最大优点在于 Jython 的发布是与 Servlet

org.python.util.PyServlet 一起进行的。该 Servlet 安装、执行和缓冲 Jython 文件使你能无须经过中间编译步骤就可编写和查看 Jython Servlet。这是通过 Servlet 映像而工作的。Servlet 映像是一个特定 Servlet（本例中为 PyServlet）和某一个 URL 模式（如 \*.py）之间的关联，例如 \*.py。通过合适的 web.xml 文件，Jython 的 PyServlet 类映像到所有的 \*.py 文件，所以一旦请求文件 \*.py 的信息到达后，PyServlet 类就装载、缓冲并调用响应所需要的文件 \*.py 中的方法。

PyServlet 的设计对其服务的 Jython 文件产生了一些限制。一个 Jython 文件首先要包含一个类，该类是 javax.servlet.http.HttpServlet 的派生类。该类的名字必须与不带 .py 扩展名的文件名匹配。换句话说，文件 Test.py 必须包含类 Test，且该类为类 javax.servlet.http.HttpServlet 的派生类。另外，使用除类 HttpServlet 的派生类之外的其他类的模块全局标志符是不安全的。

命名上的一点提示：用 Servlets、PyServlet 和实现 servlets 的 Jython 模块，命名有一点混淆。为使其清楚化，servlet 是通用意义上的术语，而不管其实现的语言。PyServlet 指特定的 Servlet org.python.util.PyServlet。在这些 PyServlet 实际服务的 Jython 模块中，混淆出现了。在场景中匹配 \*.py 模式的通过 PyServlet 装载、执行并缓冲的文件需要一个它们自己的名字来区别它们。以前都没有这样的术语，但为了本章故采用 jylets 的称号。

### 12.5.1 安装 PyServlet

安装 PyServlet 所需要的是定义一个 Servlet 映射并保证 python.jar 文件在场景的 lib 目录中。Servlet 映射在场景的配置描述文件 web.xml 中定义。Jython 场景的 web.xml 文件在 \$TOMCAT\_HOME/webapps/jython/WEB-INF/web.xml 中。在 Tomcat 中，\$TOMCAT\_HOME/conf/web.xml 中缺省的 web.xml 文件用来设置所有在场景的 web.xml 中所未明显包含的设置。所以你的场景中可能还未有一个 web.xml 的文件，而且你不必定义所有该场景中的属性，因为已经存在缺省值。

程序清单 12-5 是 Jython 场景的一个示例的配置描述器，它建立了作为匹配 \*.py 模式的场景响应处理器的 PyServlet。加入文件 web.xml 的最基本之处是一个 PyServlet 的 Servlet 定义和一个将 URLs 与 PyServlet 相关联的 Servlet 映射。由于 Tomcat 有一些未定义项目的缺省值，程序清单 12-5 是一个为 Tomcat 服务器的非常完全的 web.xml 文件。

**程序清单 12-5 为 PyServlet 的示例配置描述器**

```
<web-app>
  <servlet>
    <servlet-name>PyServlet</servlet-name>
    <servlet-class>
      org.python.util.PyServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>PyServlet</servlet-name>
    <url-pattern>*.py</url-pattern>
  </servlet-mapping>
</web-app>
```

Servlet 的定义定义了 Servlet 的名字和 Servlet 的类。程序清单 12-5 定义了名字 PyServlet 和类

org.python.util.PyServlet。该类是一个完全的包-惟一标志 Servlet 类：PyServlet 类的层次。实际类驻留在 jython.jar 文件中，该文件必须在场景的 lib 目录中。

PyServlet 的 Servlet 定义可选地包含初始化参数 (init-params)。PyServlet 使用这样一些参数来初始化 Jython，所以你必须把各种 Jython 属性设置如 python.home、python.path、python.respectJavaAccessibility 或任何其他的通常在 Jython 注册文件中的其他属性放置在这儿。程序清单 12-6 是一个提供 python.home 和 python.path 值作为 init-params 的 web.xml 文件。

程序清单 12-6 在 web.xml 文件中的 PyServlet init-params

---

```
<web-app>
  <servlet>
    <servlet-name>PyServlet</servlet-name>
    <servlet-class>
      org.python.util.PyServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
      <param-name>python.home</param-name>
      <param-value>c:\jython-2.1</param-value>
    </init-param>
    <init-param>
      <param-name>python.path</param-name>
      <param-value>
        c:\jython-2.1\lib\site-packages
      </param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>PyServlet</servlet-name>
    <url-pattern>*.py</url-pattern>
  </servlet-mapping>
</web-app>
```

---

定义 Jython 资源位置的属性，如 python.home 和 python.path，需要引起特别注意。这些将影响到场景是否自恰以及其跨平台的可移植性。如果 python.home 属性指向了场景以外，或 python.path 属性包含一个场景以外的目录，则场景将不再自恰。另外，属性值必须是与平台相关的路径。注意在程序清单 12-6 中的 python.home 和 python.path 属性。它们在别的平台下将不能工作，并且没有机制来为这些属性值创建与平台无关的路径。幸运的是 PyServlet 有一个 python.home 的缺省值，它产生了一个自恰的与平台无关的场景。

PyServlet 的缺省 python.home 值是场景的 lib 目录。这使 Jython 场景缺省的 python.home 值为下面的一种（依赖于平台）：

```
%TOMCAT_HOME%\webapps\jython\WEB-INF\lib
$TOMCAT_HOME/webapps/jython/WEB-INF/lib
```

另外，Jython 的 lib 目录将自动变为下列二者之一：

```
%TOMCAT_HOME%\webapps\jython\WEB-INF\lib\Lib
$TOMCAT_HOME/webapps/jython/WEB-INF/lib/Lib
```

这保证了所有的 Jython 资源在场景之内，使其能自恰。另外的特点是 PyServlet 以平台无关的方式增加了缺省的 python.home 路径，使得只用这些缺省值的场景与平台无关。

### 12.5.2 测试 PyServlet

你可通过启动 Tomcat 和在浏览器中查看一个简单的 jylet (Jython-Servlet) 来确认 Servlet 映射正在工作。下面是一个简单的测试 jylet：

```
# File ServletMappingTest.py
from javax.servlet.http import HttpServlet

class ServletMappingTest(HttpServletRequest):
    def doGet(self, req, res):
        out = res.getWriter()
        res.setContentType("text/html")
        print >> out, "Greetings from a jylet."
```

保存这个测试文件为 %TOMCAT\_HOME%\webapps\jython\ServletMappingTest.py，然后将你的浏览器定位在 http://localhost:8080/jython/ServletMappingTest.py。如果你看到问候信息，则 PyServlet 映射是正确的。要完成这个安装，创建 Jython 的 lib 目录 %TOMCAT\_HOME%\webapps\jython\WEB-INF\lib\Lib。这是一个你要把在你的 jylets 中所用的任何 Jython 模块都放入其中的目录。到此为止，安装完成了。

## 12.6 cookie

cookie 是存储在客户端并能为其后的请求所用的信息。为从 Jython 中创建和操作 cookie，使用 javax.Servlet.http.cookie 类。创建一个新的 cookie 需要用名字和值作为参数来实例化 javax.servlet.http.cookie。如你想用 cookie 来跟踪书本的销售，你可用 cookie 来按下列的方式设置作者和书名：

```
from javax.servlet import http
name = "book1"
value = "Lewis Carroll, Alice In Wonderland"
MyNewCookie = http.cookie(name, value)
```

现在你有一个名为 book1 的新的 cookie，其值为 Rev.Dodgson 的笔名，书名《Alice in Wonderland》。将该 cookie 发送到客户端，你可使用 HttpServletResponse 的 addcookie (cookie) 方法：

```
res.addCookie(MyNewCookie)
```

增加 cookie 必须在通过响应流发送其他内容之前发生。

cookie 必须引起更多的注意，这样 cookie 的实例有方法来设置 cookie 的特性。每个 cookie 实例必须使用 get 和 set 方法或 Jython 的自动 bean 特性来操作下列的属性：

- comment
- domain
- maxAge

- name
- path
- secure
- value
- version

程序清单 12-7 使用了 cookie 对象的自动 bean 特性来创建和读取从 Web 表单中定义的 cookie。

#### 程序清单 12-7 Jython 的 cookie

```
# File: cookies.py
from javax import servlet
from javax.servlet import http

class cookies(http.HttpServlet):
    def doGet(self, req, res):
        res.setContentType("text/html")
        out = res.getOutputStream()

        print >>out, "<html><head><title>cookies with Jython</title></head>"
        print >>out, """
            <body>\n<H2>cookie list:</H2>
            Remember, cookies must be enabled in your browser.<br><br>"""

        # Here's the list of cookies
        for c in req.cookies:
            print >>out, """
                <b>Cookie Name</b>= %s &nbsp;
                <b>Value</b>= %s<br><br>"" % (c.name,c.value)

        print >>out, """<br><br><br>
        <HR><P>Use this form to add a new cookie</P>
        <form action="cookies.py" method="POST">
        <P>Name:<br><INPUT type="text" name="name" size="30"></P>
        <P>Value:<br><INPUT type="text" name="value" size="30"></P>
        <P>Use the MaxAge field to set the cookie's time-to-expire.
            A value of "0" deletes the cookie immediately, a value of
            "-1" saves the cookie until the browser exits, and
            any other integer represents seconds until it expires
            (i.e.- using "10" would expire 10 seconds after being set).</P>
        <P>MaxAge:<br><INPUT type="text" name="maxAge" size="30"></P>
        <INPUT type="submit" name="button" value="submit">
        \n</body>
        \n</html>
        """

    def doPost(self, req, res):
        res.setContentType('text/html');
        out = res.getWriter()
        name = req.getParameterValues("name")[0]
        value = req.getParameterValues("value")[0]
        maxAge = req.getParameterValues("maxAge")[0]

        if name:
            newcookie = http.cookie(name, value)
```

```

newcookie.setMaxAge( int(maxAge or -1) )
newcookie.setComment( "Jython test cookie" )
res.addCookie(newCookie)

print >>out, """
<html><body>Cookie set successfully\n\n
<P>click <a href="cookies.py">here</a>
to view the new cookie.</P>
<P>If cookies are enabled in your
browser that is.</P>
</body>
</html>"""

else:
    print >>out, """
<html>\n<body>
Cookie not set
<P>No cookie "Name" provided</P>
<P>click <a href="cookies">here</a>
to try again</P>
</body>
</html>"""

```

为测试程序清单 12-7 中的 jylet，首先要确认你的浏览器设置为允许 cookies。然后将 cookies.py 文件放入目录 %TOMCAT\_HOME%\webapps\jython，将你的浏览器定位在 <http://localhost:8080/jython/cookies.py>。在你第一次访问 servlet 时，你只能看到一个头和表单。继续加入表单条目——名字可能是“Jython Cookie”，值可能是“My first Jython cookie”，然后点击提交按钮（maxAge 可选）。你将确认增加 cookie 是否是成功的。要确认 cookie 是真正增加了，可返回 doGet 方法看它在 cookie 列表中是否存在。图 12-3 显示了在返回 doGet 方法后浏览器可能显示的内容。注意图 12-3 假定已输入某一个 cookie 的名字和值，你所看到的结果将决定于你所使用的名字和值。

## 12.7 Session

Cookie 是创建与客户端会话的最常用的方法。Sessions 是一种通过一系列的请求来跟踪客户信息的方法。cookie 例子（如程序清单 12-7 所示）可用来存储一个会话的 ID，但 Java 的 HttpSession 类使会话跟踪更加容易。

要创建一个会话，使用 HttpRequest 的 getSession() 方法。它返回一个 HttpSession 实例。Ht-

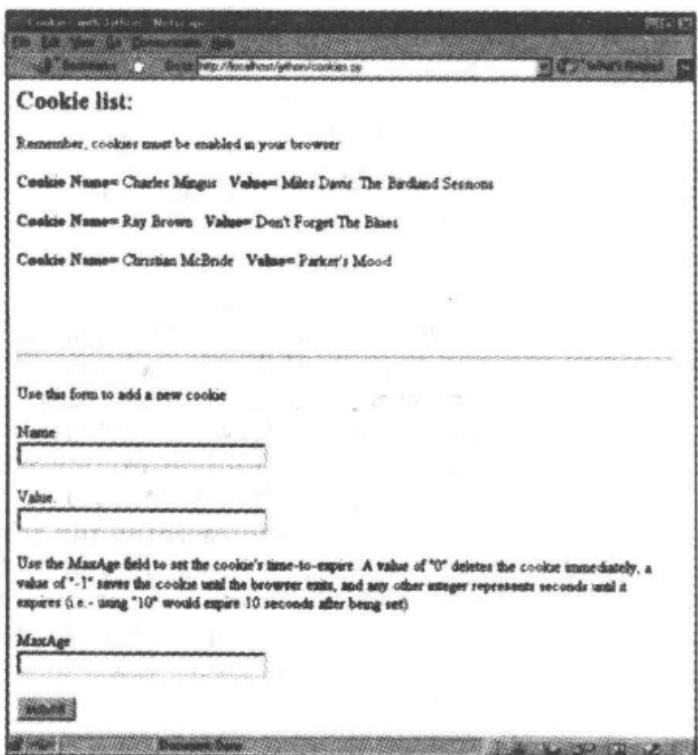


图 12-3 用 cookies.py 定义 Cookie

`tpSession` 是一个为会话管理子系统的更复杂行为而设计的简单接口。在 Python 中使用 `HttpSession` 与在 Java 中使用它仅在语法以及对象的 `get*` 方法的自动 bean 属性上有所不同。

程序清单 12-8 通过简单的 `req.session` bean 属性创建了一个 `session` 对象。程序清单 12-8 允许 cookie 或 URL 包含 session 值。因为 cookie 可存储 session 值，你可在发送其余数据到输出流前使用 `req.session` 属性或 `req.getSession()` 方法。即使客户端的 cookie 为非有效，然而 session 对象将继续工作，这是因为所有的 URL 都以 `encodeUrl` 方法重写。实际上，仅有一个 URL 需要重写，那就是表单动作。如果没有别的 URL，它们也将通过 `res.encodeUrl()` 方法来支持无 cookie 的会话。

一旦你有了 `HttpSession` 实例，将数据输入到 `session` 中仅仅是一个使用 `key`、`value` 以及 `putValue(key, value)` 和 `value = getValue(key)` 的事情。

#### 程序清单 12-8 会话管理

```
# File: session.py
from javax import servlet
from javax.servlet import http

class session(http.HttpServlet, servlet.RequestDispatcher):
    def doGet(self, req, res):
        sess = req.session
        res.contentType = "text/html"
        out = res.getWriter()

        name = req.getParameterValues("name")
        value = req.getParameterValues("value")
        if name and value:
            sess.putValue(name[0], value[0])

        print >>out, """
            <html>
                <body>
                    <H3>Session Test</H3>
                    Created at %s<br>
                    Last Accessed = %s<br>
                    <u>Values:</u>"""
            % (sess.creationTime, sess.maxInactiveInterval)
    }

    print >>out, "<ul>"
    for key in sess.getValueNames():
        print >>out, "<li>%s: %s</li>" % (key, sess.getValue(key))
    print >>out, "</ul>

    print >>out, """
        <HR><P>Use this form to add a new values to the session</P>
        <form action="session.py" method="GET">
        <P>Name:<br><INPUT type="text" name="name" size="30"></P>
        <P>Value:<br><INPUT type="text" name="value" size="30"></P>
        <INPUT type="submit" name="button" value="submit">
        </body>
        </html>
        """
```

在保存程序清单 12-8 中的 session.py 文件到 %TOMCAT\_HOME%\webapps\jython 目录后，将你的浏览器定位在 `http://localhost:8080/jython/Session.py?name=key+2&value=valu`。使用 Web 表单来增加一些变量到 session 中以保证它在工作，甚至可尝试使 cookie 无效来看 URL 是怎样重写的。图 12-4 是在增加一系列任意值（结果依赖于其所加的值）后 session.py 的结果。

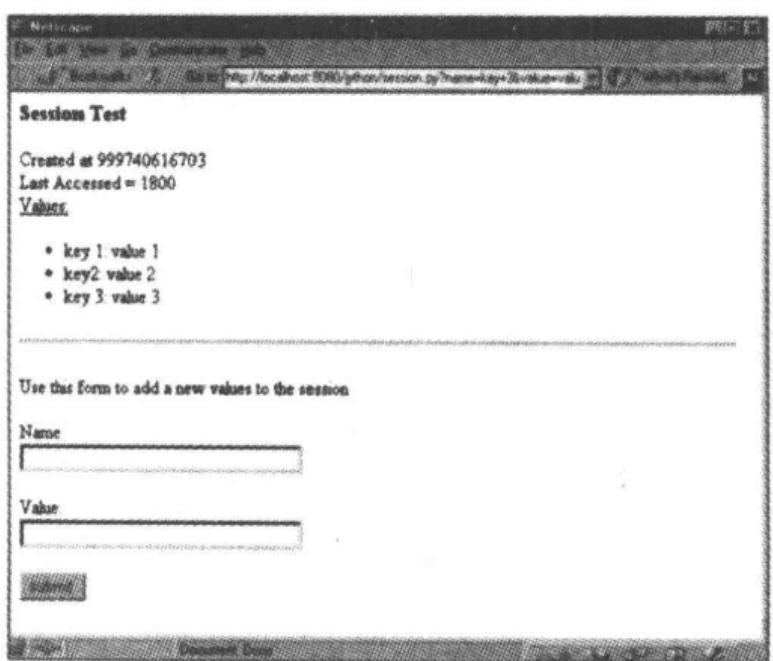


图 12-4 session.py 的 session 变量

## 12.8 数据库和 Servlet

在 jylet 中连接数据库、执行语句和遍历结果集与 Jython 的其他数据库应用没有什么区别。然而由于许多 Web 应用包含使用数据库连接的 jylet，所以管理连接和其他数据库资源尤其需要引起注意。在每个 jylet 中创建一个数据库连接会很快消耗数据库资源，这会导致一些不良后果。同样，对每个请求创建和关闭数据库连接也是一个不可接受的开销。

有很多方法可以用来管理数据库资源，但两个最基本的方法是每个 jylet 一个连接的方法或连接池的方法。每个 jylet 一个连接是一种很普遍且耗费资源的方法。该方法包括用 jylet 的 init 方法建立一个数据库连接，在 jylet 卸载时关闭该数据库连接。这消除了在响应客户端请求时的任何连接开销。然而这仅在你所需要的数据库连接在资源管理的极限以内才有意义。在很多情况下需要更谨慎的资源管理。

程序清单 12-9 实现了一个 jylet，在其 init 方法中包含且在 destroy 方法中关闭数据库连接和游标对象。所以 jylet 仅在初始化时需要连接开销而不是在每次客户请求时都如此。程序清单 12-9 的数据库实现使用了第 11 章“数据库编程”中所讲的 zxJDBC 和 MySQL 数据库。所以你必须在场景的 lib 目录中加入 MySQL 和 zxJDBC 所必须的类。对程序清单 12-9 而言，文件 mm\_mysql-2\_0\_4-bin.jar 和 zxJDBC.jar 必须放入目录 %TOMCAT\_HOME%\webapps\jython\WEB-INF\lib 之中。你必须在将 jar 文件加入 lib 目录后重启 Tomcat 来保证它能检测到新的 jar 文件。

## 程序清单 12-9 带有数据库连接的 Jython Servlet

```

# file: DBDisplay.py
from javax.servlet import http
from com.ziclix.python.sql import zxJDBC

class DBDisplay(http.HttpServlet):
    def init(self, cnfg):
        #define the JDBC url
        url = "jdbc:mysql://192.168.1.77/products"
        usr = "productsUser"      # replace with real user name
        passwd = "secret"         # replace with real password
        driver = "org.gjt.mm.mysql.Driver"

        #connect to the database and get cursor object
        self.db = zxJDBC.connect(url, usr, passwd, driver)
        self.c = self.db.cursor()

    def doGet(self, req, res):
        res.setContentType("text/html")
        out = res.getWriter()

        print >>out, """
            <html>
            <head>
                <title>Jylet Database Connection</title>
            </head>
            <body>
                <table align="center">
                    <tr>
                        <td><b>ID</b></td>
                        <td><b>Title</b></td>
                        <td><b>Description</b></td>
                        <td><b>Price</b></td>
                    </tr>"""
            self.c.execute("select code, name, description, price from products")
            for row in self.c.fetchall():
                print >>out, """
                    <tr>
                        <td>%s</td>
                        <td>%s</td>
                        <td>%s</td>
                        <td>%s</td>"" % row

            print >>out, """
                </table>
            </body>
            </html>"""

    def destroy(self):
        self.c.close()
        self.db.close()

```

程序清单 12-9 假设存在一个名为 products 的数据库，该数据库包含 products 表，该表至少

有字段码、名字、描述和价格。要创建这样一个数据库，使用下面的 SQL 语句：

```
create database products
```

要创建 products 表，使用下面的 SQL 语句：

```
CREATE TABLE products (
    primarykey int(11) NOT NULL auto_increment,
    code varchar(55) default NULL,
    name varchar(255) default NULL,
    description text,
    price float(5,2) default NULL,
    PRIMARY KEY (primarykey)
) TYPE=MyISAM;
```

在创建完数据库和表后，给每个字段创建一些任意的值并将 DBDisplay.py 文件放至场景的根目录中，将你的浏览器定位在 <http://localhost:8080/jython/DBDisplay.py> 就可查看数据库的数据。

如果你的 Web 应用开始需要很多的连接，可以考虑使用连接池。连接池具有谨慎的资源管理和消除了连接开销的特点。一个连接池保留一定数量的 jylets 在响应客户端请求时借用且在完成后返回给连接池的活动数据库连接。这就创建了一个可预测的静态数目的连接。也可以使用一个语句池，赋予语句同样的优点。一个常用的免费的测试过的且有完善文档的连接池工具是 PoolMan，可从 <http://www.codestudio.com/> 上下载。当然也有其他的 Java 连接池包，它们都需要与 Jython 无缝集成。

## 12.9 JSP

JSP（Java Server Pages）是由 Sun 公司所支持的一个为 Servlets 而设计的模板系统。一个 JSP 文件包含一个不变的 Web 页的标记代码和文本，但也包含一些指定动态内容的特殊标记。目前，Tomcat 仅在 JSP 中实现 Java 语言，所以最大的问题是如何在 JSP 中使用 Jython。目前的答案是你不能直接使用 Jython。你不能使用其中的代码为 Jython 语言的代码标记（`<% code %>`）。你所能做的是使用 jythonc 编译好的类，使用一个嵌入的 PythonInterpreter 或创建一个面向 Jython 的用户化的标记库。

### 12.9.1 jythonc 编译类和 JSP

在 JSP 中使用 jythonc 编译类需要创建一个 Java 兼容的 Jython 类。该类的名字与其所包含的模块一样，从 Java 类继承而来且每一个不是从超类继承的方法中都包含字符串 `@sig`。将由 jythonc 所产生的类文件放入场景的类目录之中可使 JSP 页面像 Java 固有类一样使用这些类。当然，这也需要 jythonc.jar 文件放在场景的 lib 目录之中。

一个 JSP 文件能够以两种方法使用 jythonc 编译的类：以 scriptlets 或作为 bean。如果其以 bean 的方式使用，则 Jython 类必须符合 bean 的习惯且必须为每个读写属性包含一个 `getProperty` 和 `setProperty` 的方法。而一个 scriptlet 能使用任何类。不管是以 scriptlet 或作为 bean 的方式，你首先必须将 jythonc 编译好的类装载到合适的位置。要引入一个非 bean 的类，可使用下面的页

面引入指示:

```
<%@ page import="fully.qualified.path.to.class" %>
```

对一个 bean, 使用 jsp:useBean 标记来装载 bean:

```
<jsp:useBean name="beanName"
    class="fully.qualified path.to.class"
    scope="scope(page or session)">
```

要使用一个非 bean 的类, 在 scriptlet 标记 (`<% %>`) 或表达式标记 (`<% = %>`) 中使用类来包含 Java 代码。如果你想引入假想的类 ProductListing, 你可以用与下面设计的 JSP 页面类似的方法来使用该类:

```
<%@ page import="ProductListing" %>

<html>
<body>

<!--The next line begins the scriptlet -->
<% ProductListing pl = new ProductListing(); %>

<table>
  <tr>
    <td><%= pl.productOne %></td>
    <td><%= pl.productTwo %></td>
  </tr>
</table>
```

通常不鼓励 scriptlet, 因为它会使 JSP 页面变得很复杂。较好的方法是通过 JSP 的 useBean、setProperty 和 getProperty 标记来使用 jythonc 编译的 beans。程序清单 12-10 是一个用 Jython 编写的存储一个用户名的非常简单的 bean。它将作为一个例子显示如何在 Jython 中使用 bean。

程序清单 12-10 简单的用 Jython 编写 bean

---

```
# file: NameHandler.py
import java

class NameHandler(java.lang.Object):
    def __init__(self):
        self.name = "Fred"
    def getUsername(self):
        "@sig public String getname()"
        return self.name
    def setUsername(self, name):
        "@sig public void setname(java.lang.String name)"
        self.name = name
```

---

将程序清单 12-10 中的文件 nameHandler.py 放到目录 `%TOMCAT_HOME%\webapps\jython\WEB-INF\classes` 中, 可用下列命令在该目录中编译上述文件:

```
jythonc -w . Namehandler.py
```

程序清单 12-11 是一个使用 NameHandler bean 的简单 JSP 页面。

#### 程序清单 12-11 在一个 JSP 页面中使用 Jython bean

```
<!--file: name.jsp -->
<%@ page contentType="text/html" %>

<jsp:useBean id="name" class="NameHandler" scope="session"/>

<html>
<head>
    <title>hello</title>
</head>
<body bgcolor="white">
Hello, my name is

<jsp:getProperty name='name' property="username"/>
<br>
No, wait...

<jsp:setProperty name="name" property="username" value="Robert"/>
, It's really <%= name.getUsername() %>.

</body>
</html>
```

注意 `jsp: setProperty` 与表达式 `<% = beanName.property = value%>` 具有相同的功能，`jsp: getProperty` 与表达式 `<% = beanName.property %>` 具有相同的功能。

要在程序清单 12-11 中使用 JSP 页面，只要将文件 `name.jsp` 放到场景的根目录中（`%TOMCAT_HOME%\webapps\jython`）即可，然后将你的浏览器定位在 `http://localhost: 8080/jython/name.jsp`，你就可以看到缺省的名字 Fred 和修订者名字 Robert。

#### 12.9.2 在 JSP 中嵌入 PythonInterpreter

如果你想在 JSP scriptlet 中使用 Jython 代码，你可以间接地用 PythonInterpreter 实现。这需要你用一个引入指示来引入 `org.python.util.PythonInterpreter`。程序清单 12-12 显示了一个简单的使用 `PythonInterpreter` 对象来在 JSP 页面中包含 Jython 代码的简单 JSP 页面。

#### 程序清单 12-12 在一个 JSP 页面中嵌入 PythonInterpreter

```
<!--name: interp.jsp-->
<%@ page contentType="text/html" %>
<%@ page import="org.python.util.PythonInterpreter" %>

<% PythonInterpreter interp = new PythonInterpreter();
   interp.set("out, out"); %>

<html>
```

---

```
<body bgcolor="white">
<% interp.exec("out.printIn('Hello from JSP and the Jython interpreter.')"); %>
</body>
</html>
```

---

要使用 `interp.jsp` 文件，必须保证 `jython.jar` 文件在场景的 `lib` 目录中，再将 `interp.jsp` 文件放到场景的根目录中 (`%TOMCAT_HOME%\webapps\jython`)。如果将你的浏览器定位在 `http://localhost:8080/jython/interp.jsp`，你就可以看到从 Jython 解释器来的简单消息。

注意考虑到 `scriptlets` 的实际运用尚不多，所以在 `scriptlets` 中从 Java 使用 Jython 而增加复杂性的方法显然是令人怀疑的。有很多更好的办法来创建动态内容，如 `bean` 类和 `taglibs`。

### 12.9.3 一个 Jython Taglib

`Taglibs` 是一些你能在 JSP 页面中使用的用户化的标记库。你能通过 `jythonc` 将 Jython `taglib` 模块编译成 Java 类来创建 Jython 中的 `taglibs`。Jython `taglib` 模块在它能透明地作为 Java 类使用前必须满足一定的限制条件。模块必须包括与模块名一样的类（无 `.py` 扩展名）。类必须有 `javax.servlet.jsp.tagext.Tag` 接口或为实现该接口的派生类。`org.python.*` 包和类以及 Jython 库模块（如果 `taglib` 引入）必须为编译的类文件所能访问。

要使所有所需求的类和库都能为 `taglib` 所用，你必须用 `jythonc` 的 `-all` 模式来编译，这很像本章前面所提到的用 `jythonc` 来编译 Servlets。这样做以后会产生一个 `JAR` 文件，然后将该文件放入场景的 `lib` 目录中。问题在于很可能很多资源将会使用 Jython 的核心文件，故重复地用 `core`、`--deep` 或 `-all` 编译会产生很多不必要的冗余。更好的方法是将 `jython.jar` 文件包含在场景的 `lib` 目录之中，推荐为 Jython 模块在场景 (`{context}\WEB-INF\lib\Lib`) 的某个位置创建一个 Jython 的 `lib` 目录（可以看以前的“PyServlet”来理解），然后可以独立编译 Jython `taglib` 模块。

程序清单 12-13 是一个简单的作为样本来深入研究 `taglibs` 的 Jython `taglib` 模块。程序清单 12-13 中所做的只是增加一个消息，但它实现了 `taglib` 所有重要的部分。实现程序清单 12-13 的基本要求如下：

- 在程序清单 12-13 中的文件名与它所包含的类名一样。
- 类必须实现 `javax.servlet.jsp.tagext.Tag` 接口。超类（接口）的其他选择包括 `BodyTag` 和 `IterationTag` 接口，`javax.servlet.jsp.tagext` 包中的 `TagSupport` 或 `BodyTagSupport` 类。
- `JythonTag` 类实现所有完成 `Tag` 接口所需要的方法。

#### 程序清单 12-13 一个 Jython Taglib

---

```
# file: JythonTag.py
from javax.servlet.jsp import tagext

class JythonTag(tagext.Tag):
    def __init__(self):
        self.context = None
        self.parent = None

    def doStartTag(self):
```

```

    return tagext.Tag.SKIP_BODY

def doEndTag(self):
    out = self.context.out
    print >>out, "Message from a taglib"
    return tagext.Tag.EVAL_PAGE

def release(self):
    pass

def setPageContext(self, context):
    self.context = context

def setParent(self, parent):
    self.parent = parent

def getParent(self):
    return self.parent

```

保存 pageContext 和父 tag 信息 (self.context 和 self.parent) 的实例变量需要特别提醒。这些变量要么不能作为 self.pageContext 和 self.parent 来标志，要么所有对这些变量的访问都必须通过实例的\_dict\_。Tag 接口需要 setPageContext()、setParent() 和 getParent() 方法的实现，但由于这些方法存在，Jython 为这些相关的属性名创建了一个自动 bean 属性。这使得循环的引用和 StackOverflowException 容易实现。设想如下代码：

```

def setPageContext(self, context):
    self.context = context

```

setPageContext 方法赋予实例属性 self.context 的场景。然而，self.context 是自动 bean 属性，这意味着 self.context 实际已调用了 self.setPageContext (context)。在 Java 框架中使用 jythonc 编译的类时要常常特别小心这种类型循环的引用，但这儿由于接口导致的对 get \* 和 set \* 方法的明显需要而增加了这种可能性。

要安装程序清单 12-13 中的 taglib 所需要的类，首先必须保证 jython.jar 文件包含在场景的 lib 目录之中，然后用下面的命令来编译 JythonTag.py 文件：

```
jythonc -w %TOMCAT_HOME%\webapps\jython\WEB-INF\classes JythonTag.py
```

这将在场景的 classes 目录中产生文件 JythonTag.class 和 JythonTag\$PyInner.class。这些类本身不足以使用 taglib。你必须在 JSP 文件中使用 tag 之前创建一个 taglib 库描述文件。程序清单 12-14 是适合于描述程序清单 12-13 中定义的标记的 taglib 库的描述文件。

#### 程序清单 12-14 标记库描述器

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>

```

```

<jspversion>1.1</jspversion>
<shortname>JythonTag</shortname>
<info>A simple Jython tag library</info>
<tag>
  <name>message</name>
  <tagclass>JythonTag</tagclass>
</tag>
</taglib>

```

将程序清单 12-14 中的标记库描述器信息放到文件 %TOMCAT\_HOME%\webapps\jython\WEB-XML\Jython-taglibs.tld。JSP 页面将使用该文件来标识在该页中使用的标记。

标记库描述器标志标记库的特性如它的版本号、用它的 JSP 的版本号。它也指定了一个来引用该标记库的名字 <shortname>。余下的元素定义一个程序清单 12-13 中创建的特定的标记库。该标记元素标志完全认证的类名并给该类赋予一个名字。JythonTag 不在一个包中，所以它自己是一个完全认证的名字且该类与名字 taglet 关联。

在 JythonTag 编译好且标记库描述器保存后，剩下的步骤是从 JSP 页面中使用标记库。一个 JSP 页面必须包括一个指定到标记库路径的提示并给该库一个简单的名字。所有其后的在该库中对标记的引用将以在这个提示中赋予的名字开头。在程序清单 12-13 和程序清单 12-14 中创建的标记库的 JSP 提示将会变成：

```
<%@ taglib uri="/WEB-INF/jython-taglibs.tld" prefix="jython" %>
```

记住 .tld 文件给我们的例子 tag 指定了名字 message，故在声明这个标记库后，你可接着使用下面的消息标记：

```
<jython:message/>
```

该标记的 jython 部分来自在 JSP 声明中赋予的名字，而 message 部分来自在标记库描述器中指定的标记类的名字。程序清单 12-15 是一个使用程序清单 12-13 所定义的标记和程序清单 12-14 中的标记库描述器的 JSP 文件。保存程序清单 12-14 中的 test.jsp 为 %TOMCAT\_HOME%\webapps\jython\test.jsp，然后将你的浏览器定位在 http://localhost:8080/jython/test.jsp，你就可以看到用户化的标记信息。

#### 程序清单 12-15 实现 Jython Tag 的 JSP 文件

```

<%@ taglib uri="/WEB-INF/jython-taglibs.tld" prefix="jython" %>
<html>
<body>

<jython:message />

</body>
</html>

```

使用编译 jython 模块来实现 taglibs 将再次产生 Jython 的主目录和 Jython 的 lib 目录的问题。

除非信息在 PySystemState 中确定，一个 jythonc 编译的 taglib 将不知道 python.home 在哪或在哪儿寻找库模块。你可在 TOMCAT\_OPTS 环境变量中设置 python.home 属性，但你也必须通过 PyServlet 来建立你的系统状态信息。如果你所在的场景已装载 PyServlet 类，那 python.home 和 sys.path 信息对需要它的 taglibs 而言已经是正确的了。在缺省 PyServlet 设置的情况下，python.home 是 %TOMCAT\_HOME%\webapps\jython\WEB-INF\lib，故 Jython 的 lib 目录为 %TOMCAT\_HOME%\webapps\jython\WEB-INF\lib\Lib。在 PyServlet 装载以后，taglibs 可从同样的 lib 目录中装载 Jython 模块。

#### 12.9.4 BSF

IBM 的 BSF (Bean Scripting Framework) 是一个实现各种脚本语言的 Java 工具。Jython 是 BSF 目前支持的一种语言。Apache Jakarta 项目包括一个叫做 taglibs 的子项目，它是目前使用的 Java 标记库的一个扩充；然而真正有趣的 taglib 是 BSF 标记库。BSF 标记库与 BSF jar 文件自身能使你快速地将 Jython 的 scriptlets 和表达式直接插入 JSP 页面中。

BSF 目前的版本还需要一点改进才能与 Jython 一起工作。因为 BSF 最终会包含这些小的改变，所以在这儿详细地介绍 BSF 没有必要；当然这意味着你要确认你的 BSF 版本包括了这些变化。要弄清楚这些东西，本书的网站 (<http://www.newriders.com/>) 包含一个你能下载正确的 bsf.jar 文件的链接。在下载 bsf.jar 文件后，将其放入场景的 lib 目录中 (%TOMCAT\_HOME%\webapps\jython\WEB-INF\lib)。本书的网站也包含一个你能下载 BSF taglibs 和 bsf.tld 文件的链接。将包含 BSF taglibs 的 jar 文件放入场景的 lib 目录中并将 bsf.tld 文件放入场景的 WEB-INF 目录 (%TOMCAT\_HOME%\webapps\jython\WEB-INF\bsf.tld) 中即可。

在这些文件安装完以后，你可在 JSP 文件中使用 Jython 的 scriptlets。你首先必须包含一个提示来标志 BSF taglib：

```
<%@ taglib uri="/WEB-INF/bsf.tld" prefix="bsf" %>
```

然后你可以使用 bsf.scriptlet 标记，在标记体中按下列方法加入 Jython 代码：

```
<bsf.scriptlet language="jython">
import random
print >>, out random.randint(1, 100)
</bsf.scriptlet>
```

scriptlet 有很多在解释器中自动设置的 JSP 对象，这些对象及其 Java 类名均在下面列出：

- request javax.servlet.http.HttpServletRequest
- response javax.servlet.http.HttpServletResponse
- pageContext javax.servlet.jsp.PageContext
- application javax.servlet.ServletContext
- out javax.servlet.jsp.JspWriter
- config javax.servlet.ServletConfig
- page java.lang.Object

- exception java.lang.Exception
- session javax.servlet.http.HttpSession

大部分的对象从 Servlets 中来，但 BSF scriptlet 标记使你能在 JSP 页面中使用它们。程序清单 12-16 显示了一个使用 bsf: scriptlet 标记来在 JSP 文件中创建一个时间标志的 JSP 页面。

程序清单 12-16 BSF Scriptlet

```
<%@ taglib uri="/WEB-INF/bsf.tld" prefix="bsf" %>
<html>
<body>

<center><H2>BSF scriptlets</H2></center>
<b>client info:</b><br>

<bsf:scriptlet language="jython">
for x in request.headerNames:
    print >>out, "%s: %s<br>\n" % (x, request.getHeader(x))
</bsf:scriptlet>

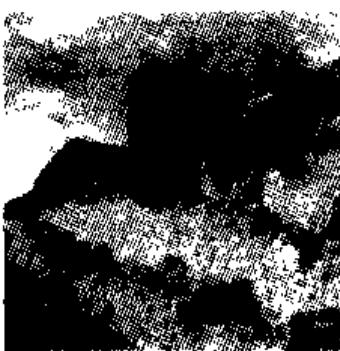
<br><br>
<b>Time of request:</b><br>

<bsf:scriptlet language="jython">
import time
print >>out, time.ctime(time.time())
</bsf:scriptlet>

</body>
</html>
```

在将程序清单 12-16 保存为文件%TOMCAT\_HOME%\webapps\python\scriptlets.jsp 后，将你的浏览器定位 http://localhost: 8080/python/scriptlets.jsp。你就可以看到所有的客户端信息与访问时间。





## 附录 A Jython 语句和内置函数快速参考

本附录包括两个按字母顺序的快速参考表。表 A-1 为内部函数，表 A-2 为 Jython 语句。

表 A-1 内部函数快速参考

函 数	描 述
abs	返回一个数 x 的绝对值 语法： <code>abs (number)</code> 例子： <code>&gt;&gt;&gt; abs (-5.43)</code> 5.43
apply	调用一个对象，其作为第一个参数提供给应用函数。它可以是一个函数、类或任何其他可调用的对象。如果可选的第二个参数提供，它必须为一个元组，当对象被调用时作为序参数。可选的第三个参数可以是 PyDictionary 或 PyStringMap，当对象被调用时作为关键字参数 语法： <code>apply (object [, args [, kwargs]])</code> 例子： <code>&gt;&gt;&gt; def printKwargs (** kw):</code> ... <code>print kw</code> ... <code>&gt;&gt;&gt; apply (printKwargs, (), globals ())</code>   '_name_': '__main__', 'printKwargs': <function printKwargs at 2744202>, '__doc__': None >>> <code>&gt;&gt;&gt; def product (x, y):</code> ... <code>return x * y</code> ... <code>&gt;&gt;&gt; print apply (product, (3, 4))</code> 12
callable	根据对象是否为可调用的类型返回 1 或 0 (true 或 false) 语法： <code>callable (object)</code> 例子： <code>&gt;&gt;&gt; def myFunction ():</code> ... <code>return</code> ... <code>&gt;&gt;&gt; callable (myFunction)</code> 1 <code>&gt;&gt;&gt; callable ("a string")</code> 0
chr	对 $\leq 65535$ 的整数，chr 返回指定整数值的字符（长度为 1 的 PyString）

(续)

函 数	描 述
	<p>语法:</p> <pre>chr (integer)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; chr (119) 'w' &gt;&gt;&gt; chr (88) 'X' &gt;&gt;&gt; chr (50000) u'\uC350'</pre>
cmp	<p>需要两个参数。在 <math>x &lt; y</math>、<math>x = y</math> 和 <math>x &gt; y</math> 时分别返回值 -1、0、1</p> <p>语法:</p> <pre>cmp (x, y)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; cmp (1, 3) -1 &gt;&gt;&gt; cmp (3, 1) 1 &gt;&gt;&gt; cmp (3, 3) 0</pre>
coerce	<p>接受两个对象作为参数，测试有无一个公用类型能表示每个对象的值。如果存在，返回两个在相同类型的元组中的值。如果不存在，coerce 产生一个 TypeError</p> <p>语法:</p> <pre>coerce (x, y)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; coerce (3.1415, 6) # float and int (3.1415, 6.0) &gt;&gt;&gt; results = coerce ("a", 2.3) Traceback (most recent call last):   File "&lt;stdin&gt;", line 1, in? TypeError: number coercion failed</pre>
compile	<p>编译一个字符串到代码对象。第一个参数为源参数。第二个参数为表示文件名的参数。第二个参数仅用来使错误信息更加可描述，故它可为任何字符串而不影响函数的功能。第三个参数为编译的模式，可以为下列三个模式中的一种：</p> <p>exec: 对模块或大的代码字符串      single: 对单个语句      eval: 对表达式</p> <p>返回的代码对象能够在 exec 或 eval 语句下执行</p> <p>语法:</p> <pre>compile (source, filename, mode)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; co = compile ("0 and 'This' or 1 and 'that'", "-", "eval") &gt;&gt;&gt; exec (co) &gt;&gt;&gt; eval (co) 'that' &gt;&gt;&gt; co = compile ("for x in range (10): \n\tprint x", "&lt;console&gt;", "single") &gt;&gt;&gt; exec (co) 0 1 2 3 4 5 6 7 8 9</pre>

(续)

函 数	描 述
complex	返回一个复数，其实部和虚部分别来自函数的两个参数。虚数部分（第二个参数）可选，当未提供第二个参数时，则为 0 语法： <code>complex (real [, imag])</code> 例子： <code>&gt;&gt;&gt; complex (2) # int to complex (2+0j)</code> <code>&gt;&gt;&gt; complex (3.1, 0.123) # floats to complex (3.1+0.123j)</code> <code>&gt;&gt;&gt; complex ("2.1","0.1234") (2.1000000000000001+0j)</code>
delattr	从一个对象中删除一个指定的属性 语法： <code>delattr (object, name)</code> 例子： <code>&gt;&gt;&gt; class cache: ...     pass # empty class to store arbitrary objects ... &gt;&gt;&gt; c = cache () &gt;&gt;&gt; c.a = 1 &gt;&gt;&gt; c.b = 2 &gt;&gt;&gt; vars (c) {'a': 1, 'b': 2} &gt;&gt;&gt; delattr (c, "a") &gt;&gt;&gt; vars (c) {'b': 2}</code>
dir	如果指定了对象，dir 返回那个对象中定义的名字列表。如果对象没指定，则返回当前范围内定义的名字 语法： <code>dir ([object])</code> 例子： <code>&gt;&gt;&gt; a = 1 &gt;&gt;&gt; b = "a string" &gt;&gt;&gt; def aFunction (): ...     return ... &gt;&gt;&gt; dir () ['__doc__', '__name__', 'a', 'aFunction', 'b', 'file'] &gt;&gt;&gt; dir (aFunction) ['__dict__', '__doc__', '__name__', 'func_closure', 'func_code', 'func_defaults', 'func_doc', 'func_globals', 'func_name']</code>
divmod	以元组的形式返回 y/x 的整数部分和模数 语法： <code>divmod (x, y)</code> 例子： <code>&gt;&gt;&gt; divmod (9, 4) (2, 1)</code> <code>&gt;&gt;&gt; divmod (7, 2) (3, 1)</code>

(续)

函数	描述
eval	<p>计算 compile 创建的代码串或代码对象。如果名字空间未指定，使用目前的名字空间。如果只提供 globals 名字空间，则也适合于 locals。名字空间可为 PyDictionary 或 PyStringMap</p> <p>语法：</p> <pre>eval (source [, globals [, locals]])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; eval ("x&gt;3 and x or 0", {'x': 4}) 4 &gt;&gt;&gt; eval ("x&gt;3 and x or 0", {'x': 2}) 0 &gt;&gt;&gt; co = compile ("map (lambda x: divmod (x, 3), L)", "console","eval") &gt;&gt;&gt; eval (co, {'L': range (15)}) [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), (4, 2)]</pre>
execfile	<p>执行一个在指定名字空间中包含 Python 代码的文件。如果名字空间未指定，使用目前的名字空间。如果只提供 globals 名字空间，则也适合于 locals。名字空间可为 PyDictionary 或 PyStringMap</p> <p>语法：</p> <pre>execfile (filename [, globals [, locals]])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; # first use an anonymous dictionary for globals &gt;&gt;&gt; execfile ("c:\windows\desktop\testfile.py", {}) a b 7 &gt;&gt;&gt; globals () {'__name__': '__main__', '__doc__': None} &gt;&gt;&gt; &gt;&gt;&gt; # Now use current namespaces &gt;&gt;&gt; execfile ("c:\windows\desktop\testfile.py",) a b 7 &gt;&gt;&gt; globals () {'var2': 'b', 'var1': 'a', '__doc__': None, 'var3': 7, '__name__': '__main__'}</pre>
filter	<p>filter 函数需要一个函数或序列作为其参数，返回那些 sequence 中为“true”的那些元素列表。函数能为 None。在那种情况下，filter 返回计算为“true”的参数序列的那些元素列表</p> <p>语法：</p> <pre>filter (function, sequence)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; filter (lambda x: x%2, [1, 2, 3, 4, 5, 6, 7, 8, 9]) [1, 3, 5, 7, 9] &gt;&gt;&gt; filter (None, [1, 0, [], 3-3, {}, 7]) [1, 7]</pre>
float	<p>返回第一个参数，在可能时将其转换为 float (PyFloat 型)</p> <p>语法：</p> <pre>float (x)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; float (2) # int to float</pre>

(续)

函 数	描 述
	2.0 <pre>&gt;&gt;&gt; float(7L) # long to float 7.0 &gt;&gt;&gt; float(abs(1.1+2.1j)) 2.3706539182259396 &gt;&gt;&gt; float("2.1") # string to float 2.1</pre>
getattr	返回指定对象属性的引用。所需要的两个参数为对象和作为字符串的属性名。第三个参数可选，如果该对象不包含指定的属性，则第三个参数为返回的缺省值。在没有所需要的属性而又没有第三个属性时，产生一个 AttributeError。 语法： <code>getattr(object, name [, default])</code> 例子： <pre>&gt;&gt;&gt; from time import time &gt;&gt;&gt; class util: ...     def __init__(self): ...         self.inittime = time() ... &gt;&gt;&gt; u = util() &gt;&gt;&gt; getattr(u, "inittime") 9.8829678293E8 &gt;&gt;&gt; getattr(u, "currenttime", time()) 9.8829681275E8</pre>
globals	返回一个表示在 globals 名字空间中定义的变量的字典型对象。 语法： <code>globals()</code> 例子： <pre>&gt;&gt;&gt; num = 1 &gt;&gt;&gt; string = "a String" &gt;&gt;&gt; globals() {'num': 1, '__name__': '__main__', '__doc__': None, 'string': 'a String'}</pre>
hasattr	测试给定的对象有无指定的属性，根据结果返回 1 或 0。 语法： <code>hasattr(object, name)</code> 例子： <pre>&gt;&gt;&gt; class add: ...     def __init__(self): ...         pass ...     def methodA(self): ...         pass ... &gt;&gt;&gt; hasattr(add, "methodA") 1 &gt;&gt;&gt; hasattr(add, "methodB") 0</pre>
hash	返回一个表示指定对象的 hash 值的整数。 语法： <code>hash(object)</code> 例子：

(续)

函 数	描 述
	<pre>&gt;&gt;&gt; a = 1 &gt;&gt;&gt; hash (1) 1 &gt;&gt;&gt; hash (a) 1 &gt;&gt;&gt; c = "dog" &gt;&gt;&gt; hash (c) 1528775661</pre>
hex	<p>返回一个作为字符串的整数的十六进制</p> <p>语法:</p> <pre>hex (number)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; hex (16) '0x10' &gt;&gt;&gt; hex (15) '0xf'</pre>
id	<p>返回一个代表指定对象的惟一标志符的整数</p> <p>语法:</p> <pre>id (object)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; string = "A" &gt;&gt;&gt; id ("A") 6262933</pre>
input	<p>函数 input 与 raw_input 除在 prompt 输入的值的计算不同外, 其余都一样</p> <p>语法:</p> <pre>input ([prompt])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; input ("Enter a Jython expression:") Enter a Jython expression: 2 + 3 5 &lt;- the evaluated result of 2 + 3</pre>
int	<p>返回第一个参数, 在可能时将其转换为 integer (PyInteger 型)。可选的第二个参数为转换时的基 (hex 为 16 进制, octal 为 8 进制)。将复数转换为整数时只有在用复数的绝对值 abs () 才可能。浮点数为截取尾数而非四舍五入</p> <p>语法:</p> <pre>int (x [, base])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; int (3.5) # float to int 3 &gt;&gt;&gt; int (5L) # long to int 5 &gt;&gt;&gt; int (abs (2.2 + 1.72j)) # complex to int 2 &gt;&gt;&gt; int ("012", 8) # octal string to int 10 &gt;&gt;&gt; int ("0xA", 16) # hex string to int 26</pre>
intern	<p>该字符串将给定的字符串放入持久字符串的 PyStringMap 中, 并返回内部字符串对象本身</p> <p>语法:</p>

(续)

函 数	描 述
	<pre>intern (string) 例子: &gt;&gt;&gt; string1 = "a" &gt;&gt;&gt; Istring1 = intern (string1) &gt;&gt;&gt; id (string1), id (Istring1) (7347538, 7347538)</pre>
isinstance	<p>根据实例是否为所提供的类的实例而返回 1 或 0 (true 或 false)</p> <p>语法:</p> <pre>isinstance (instance, class).</pre> <p>例子:</p> <pre>&gt;&gt;&gt; import java &gt;&gt;&gt; hm = java. util. HashMap () &gt;&gt;&gt; isinstance (hm, java. util. HashMap) 1 &gt;&gt;&gt; isinstance (hm, java. util. AbstractMap) 1 &gt;&gt;&gt; isinstance (hm, java. util. Vector) 0</pre>
len	<p>返回序列或映像类型的长度</p> <p>语法:</p> <pre>len (object)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; len ("a string") 8 &gt;&gt;&gt; len ([1, 2, 3, 4, 5]) 5 &gt;&gt;&gt; len (range (0, 100, 7)) 15</pre>
list	<p>接受一个为序列的参数，并返回一个元素列表，其元素与参数中的一样</p> <p>语法:</p> <pre>list (sequence)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; list ("abcdefg") ['a', 'b', 'c', 'd', 'e', 'f', 'g'] &gt;&gt;&gt; list (( "a", "b", "c", "d", "e", "f", "g")) ['a', 'b', 'c', 'd', 'e', 'f', 'g']</pre>
locals	<p>返回一个 PyStringMap，它包含了在 locals 名字空间中定义的变量</p> <p>语法:</p> <pre>locals ()</pre> <p>例子:</p> <pre>&gt;&gt;&gt; def aFunction (): ...     aVariable = "String var" ...     print locals () ... &gt;&gt;&gt; aFunction () {'aVariable': 'String var'}</pre>
long	<p>返回第一个参数，在可能时将其转换为 long (PyLong 型)。可选的第二个参数为转换时的基 (hex 为 16 进制，octal 为 8 进制)。将复数转换为 long 时只有在用复数的绝对值 abs () 才可能。浮点数为截取尾数而非四舍五入</p>

(续)

函 数	描 述
	<p>语法:</p> <pre>long (x, base)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; long (2.7) # float to long 2L &gt;&gt;&gt; long (abs (1.1+2.3j)) # complex to long 2L &gt;&gt;&gt; long ("021", 8) # octal string to long 17L &gt;&gt;&gt; long ("0xff", 16) # hex string to long 255L</pre>
map	<p>map 函数需要一个函数，一个或多个序列作参数。函数对每一个序列的元素进行调用，其参数等同于所提供的序列数。如果存在不同长度的序列，迭代一直持续到最长的序列用尽，此时较短的序列用 None 代替。返回的是一个对函数的每次调用的返回值的列表。函数可为 None，此时返回的列表为通过序列迭代产生的参数集</p> <p>语法:</p> <pre>map (function, sequence [, sequence, ...])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; map (None, range (6), ["a","b","c","d"]) [(0,'a'), (1,'b'), (2,'c'), (3, None), (4, None), (5, None)] &gt;&gt;&gt; def pad (string) ...     return string.rjust (10) ... &gt;&gt;&gt; map (pad, ["a","b","c"]) ['         a', '         b', '         c', 'd']</pre>
max	<p>返回指定序列的最大元素。参数必为一个序列</p> <p>语法:</p> <pre>max (sequence)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; max (1, 2, 3, 4, 5, 6, 7, 8, 9) 9 &gt;&gt;&gt; T = ("strings", are,"compared", "lexigraphically") &gt;&gt;&gt; max (T) 'strings'</pre>
min	<p>返回指定序列的最小元素。参数必为一个序列</p> <p>语法:</p> <pre>min (sequence)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; min (1, 2, 3, 4, 5) 1 &gt;&gt;&gt; T = "min returns the smallest char of a string" &gt;&gt;&gt; min (T) ' '</pre>
oct	<p>返回一个作为字符串的整数的十六进制表示</p> <p>语法:</p> <pre>oct (number)</pre> <p>例子:</p>

(续)

函 数	描 述
	<pre>&gt;&gt;&gt; oct(8) '010' &gt;&gt;&gt; oct(7) '07'</pre>
open	<p>需要一个与平台有关的文件路径和名字，然后打开指定的文件并返回相关的文件对象。可选的模式参数指明文件打开是否是用来读、写、追加或连接。模式也要指明文件是否为二进制。Jython 需要二进制模式来读写二进制数，这与 CPython 不同。如果没有模式参数，则为非二进制数、只读。可能的模式为：</p> <pre>r = reading w = writing a = appending + = 追加到 r, w, a 后来指定 reading 加 writing 或 reading 加 appending b = binary mode (二进制模式)  open 函数可选的第三个参数指定缓冲，目前 Jython 忽略第三个参数语法： open(filename [, mode [, buffering]])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; # open file in current directory for reading &gt;&gt;&gt; # plus writing. &gt;&gt;&gt; fo = open("config.cfg", "r+") &gt;&gt;&gt; &gt;&gt;&gt; # Open file for binary reading &gt;&gt;&gt; fo = open("c:\soot\ba\Baf.class", "rb")</pre>
ord	<p>返回字符的整数值。该函数与 chr(integer) 相反。对字符 c, chr(ord(c)) == c</p> <p>语法：</p> <pre>ord(character)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; ord('w') 119 &gt;&gt;&gt; ord('x') 88 &gt;&gt;&gt; ord(u'\u0350') 50000</pre>
pow	<p>返回 <math>x^{**}y</math>。如提供 z, 返回 <math>x^{**}y \% z</math></p> <p>语法：</p> <pre>pow(x, y [, z])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; pow(3, 3) 27 &gt;&gt;&gt; pow(3, 3, 2) 1</pre>
range	<p>返回基于 start、stop 和 step 的整数列表。所有的参数必须为整数 (PyInteger 或 PyLong)。计数从 start 或 0 开始。计数一直到最后的 step 整数 (但不包括它)。如果可选的 step 参数提供，则它作为增量</p> <p>语法：</p> <pre>range([start,] stop [, step])</pre> <p>例子：</p>

(续)

函数	描述
	<pre>&gt;&gt;&gt; range(4) [0, 1, 2, 3] &gt;&gt;&gt; range(3, 6) [3, 4, 5] &gt;&gt;&gt; range(2, 10, 2) [2, 4, 6, 8]</pre>
raw_input	<p>从标准输入中读入字符串，并将尾部的换行符删除</p> <p>语法：</p> <pre>raw_input([prompt])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; name = raw_input("Enter your name:") Enter your name: Bilbo Baggins &gt;&gt;&gt; print name Bilbo Baggins</pre>
reduce	<p>将一列值缩减为一个。两个需要的参数为函数和序列。第三个参数可选，为初始值。提供的函数必须有两个参数。缩减操作是将函数作用在前两个序列参数之上或在提供初值的情况下，作用在初值和第一个序列参数之上。函数下一步再作用在第一次作用的结果和下一个序列参数之上直到序列用尽，返回一个值</p> <p>语法：</p> <pre>reduce(function, sequence [, initial])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; def add(x, y): ...     return x + y ... &gt;&gt;&gt; reduce(add, [5, 4, 3, 2, 1]) 15 &gt;&gt;&gt; def stripspaces(x, y): ...     return x.strip() + y.strip() ... &gt;&gt;&gt; reduce(stripspaces, "A string with spaces") 'Astringwithspaces'</pre>
reload	<p>重载入一个指定的模块</p> <p>语法：</p> <pre>reload(module)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; import math &gt;&gt;&gt; # If the math module changed-reload it. &gt;&gt;&gt; reload(math) &lt;jclass org.python.modules.math at 1401745&gt;</pre>
repr	<p>返回一个表示指定对象的字符串</p> <p>语法：</p> <pre>repr(object)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; repr(123) '123' &gt;&gt;&gt; repr(type) '&lt;java function type at 4737674&gt;' &gt;&gt;&gt; repr("A string") '"A string"'</pre>

(续)

函数	描述
round	<p>返回一个表示数值参数的浮点数。如果可选的要四舍五入的参数省略，则四舍五入到小数点位置。否则，四舍五入到小数点位前后给定数值的位置。负的 <code>ndigits</code> 数表示小数点的左边，正的 <code>ndigits</code> 数表示小数点的右边。在 Jython 中，<code>ndigits</code> 参数必须为整数。而在 CPython 中，情况有一点点不同，CPython 可接受可转换为整数的任何内容</p> <p>语法：</p> <pre>round (number [, ndigits])</pre> <p>例子：</p> <pre>&gt;&gt;&gt; round (1.23434, 3L) 1.234 &gt;&gt;&gt; round (23.2124) 23.0 &gt;&gt;&gt; round (2) 2.0</pre>
setattr	<p>设置对象的属性值。三个所需要的参数分别为对象、作为字符串的属性名字和值。在 Jython 中，你可对 Jython 对象设置任何属性，但不能对 Java 对象设置。这意味着 setattr 设置一个 Jython 对象先前没有的属性。但对 Java 对象，则返回一个 <code>TypeError</code></p> <p>语法：</p> <pre>setattr (object, name, value)</pre> <p>例子：</p> <pre>&gt;&gt;&gt; class split: ...     def __init__ (self, token): ...         self. token = token ... ... &gt;&gt;&gt; s = split (";") &gt;&gt;&gt; vars (s) {'token': ';' } &gt;&gt;&gt; setattr (s, 'count', 0) &gt;&gt;&gt; setattr (s, 'token', ",") &gt;&gt;&gt; vars (s) {'token': ',', 'count': 0} &gt;&gt;&gt; &gt;&gt;&gt; # Java objects are different for arbitraryattributes &gt;&gt;&gt; # Note that javax. midi is an optional package-you may need to download it &gt;&gt;&gt; # for this example to work. &gt;&gt;&gt; from javax. sound. midi import VoiceStatus &gt;&gt;&gt; v = VoiceStatus () &gt;&gt;&gt; v. channel 0 &gt;&gt;&gt; setattr (v, "channel", 3) &gt;&gt;&gt; v. channel # "channel" already exists-this works 3 &gt;&gt;&gt; v. channel 3 &gt;&gt;&gt; v. arbitraryVar = 6 # "arbitraryVar doesn't exist" Traceback (innermost last):   File "&lt;console&gt;", line 1, in ? TypeError: can't set arbitrary attribute in java instance: arbitraryVar</pre>

(续)

函 数	描 述
slice	<p>slice 函数与序列截取语法有同样的功能。从该函数返回的 slice 对象能代替传统的 slice 符号</p> <p>语法:</p> <pre>slice ( [ start, ] stop [ , step])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; L = ["s", "o", "r", "c", "q", "a", "p", "j"] &gt;&gt;&gt; sliceobj = slice(7, 0, -2) &gt;&gt;&gt; L[sliceobj] ['j', 'a', 'c', 'o']</pre>
str	<p>返回一个对象的字符串表示</p> <p>语法:</p> <pre>str (object)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; str(1) # int to string '1' &gt;&gt;&gt; str(4L) # long to string '4' &gt;&gt;&gt; str(2.2 + 1.3j) '(2.2 + 1.3j)' &gt;&gt;&gt; from java.util import Vector &gt;&gt;&gt; str(Vector) 'java.util.Vector' &gt;&gt;&gt; v = Vector() &gt;&gt;&gt; str(v) '[]' &gt;&gt;&gt; v.add("typy") # put a function in the vector 1 &gt;&gt;&gt; str(v) ' [&lt;java function type at 5229978&gt; ]'</pre>
tuple	<p>接受一个序列作参数，返回一个与参数元素数目相同的元组</p> <p>语法:</p> <pre>tuple (sequence)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; tuple([1, 2, 3, 4, 5]) # list to tuple (1, 2, 3, 4, 5) &gt;&gt;&gt; tuple("This is a test") # string to tuple ('T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 't', 'e', 's', 't')</pre>
type	<p>返回一个对象的类型，通常为 org.python.core 包中的类名，该类表示该类型的对象</p> <p>语法:</p> <pre>type (object)</pre> <p>例子:</p> <pre>&gt;&gt;&gt; type("a string") &lt;jclass org.python.core.PyString at 1923370&gt;</pre>
unichr	<p>与 chr 功能一样。这两个方法，chr 和 unichr 为与 CPython 兼容而存在</p> <p>语法:</p> <pre>unichr (i)</pre> <p>例子:</p> <pre>See chr</pre>
unicode	<p>用指定的编码方式产生一个新的 PyString 对象。有效的编码存在于编码文件夹的 Python Lib 目录之中。可选的 errors 参数为 strict、ignore 或 replace</p>

(续)

函 数	描 述
vars	<p>语法:</p> <pre>unicode (string [, encoding [, errors]])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; unicode ("Abé","utf_8","replace") u'Ab \ uFFFD'</pre>
xrange	<p>如果指定了对象, vars 返回在那个对象内的名字字典。对象必须有内部 dict_ 属性。如果没指定对象, vars 与 locals () 一样</p> <p>语法:</p> <pre>vars ([object])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; class aClass: ...     def __init__(self): ...         attrib1 = "An instance variable" ...         attrib2 = "Another instance variable" ... &gt;&gt;&gt; vars (aClass) {'__init__': &lt;function __init__ at 634037&gt;,  '__module__': '__main__', '__doc__': None} &gt;&gt;&gt; vars () {'aClass': &lt;class __main__. aClass at 7033304&gt;,  '__name__': '__main__', '__doc__': None}</pre>
zip	<p>xrange 函数与 range 函数功能一样。差别在于 xrange 函数返回一个 xrange 对象, 它能在需要时产生数, 而不是一次性地产生所有的数。这对于范围很大时有很大的帮助</p> <p>语法:</p> <pre>xrange ([start,] stop [, step])</pre> <p>例子:</p> <p>See range</p> <p>返回一个长度等于参数中最短序列的元组列表。元组包含有同样索引的序列项, 每个元组的长度等于所提供的序列数</p> <p>语法:</p> <pre>zip (seq1 [, seq2 [...]])</pre> <p>例子:</p> <pre>&gt;&gt;&gt; zip ([1, 2, 3, 4, 5, 6, 7],"abcdefg") [(1,'a'), (2,'b'), (3,'c'), (4,'d'), (5,'e'), (6,'f'), (7,'g')] &gt;&gt;&gt; zip (range (5), range (5, -1, -1)) [(0, 5), (1, 4), (2, 3), (3, 2), (4, 1)]</pre>

表 A-2 Jython 语句快速参考

语 句	描 述
assert	<p>assert 语句测试表达式是否为 true, 如果非 true 则产生一个异常。如果提供两个表达式, 它们将作为 OR 对待, 故只要有一个为 true, 则不会产生异常。设置 debug_=0, 可能在将来的版本中用 -O 命令行选择就可使 asserts 无效</p> <p>语法:</p> <pre>"assert" expression [, expression]</pre> <p>例子:</p>

(续)

语句	描述
	>>> a = 21 >>> assert a < 10 Traceback (innermost last): File "< console >", line 1, in ? AssertionError: break 语句终止一个循环的执行，并从循环块后开始执行。这意味着如果存在 else 块，它将被跳过 语法： "break" 例子： >>> for x in (1, 2, 3, 4): ...     if x == 3: ...         break ...     print x ... 1 2 >>>
class	指定类定义的开始 语法： "class" name [ (base-class-name (s))] 例子： >>> class test:      # no base class ...     pass      # place holder ... >>> t = test () # Calls class to create an instance
continue	continue 语句终止当前循环块的执行，并从下一步迭代开始运行 语法： "continue" 例子： >>> for x in (1, 2, 3, 4): ...     if x == 3: ...         continue ...     print x ... 1 2 4 >>>
def	指定函数或方法定义的开始 语法： "def" name ( [parameters]) code-block 例子： >>> def caseless_srch_n_replace (string, srch, rplc): ...     return string.lower ().replace (srch, rplc) ... >>> S = "Some UseLess text" >>> caseless_srch_n_replace (S, "useless", "useful") 'some useful text'
del	从 del 调用的名字空间中消去一个变量

(续)

语句	描述
	<p>语法： "del" identifier</p> <p>例子： &gt;&gt;&gt; a = "foo" &gt;&gt;&gt; print a foo &gt;&gt;&gt; del a &gt;&gt;&gt; print a Traceback (innermost last): File "&lt; console &gt;", line 1, in ? NameError: a</p>
exec	<p>执行一个字符串，打开一个文件对象或作为 exec 语句后第一个表达式的代码对象。第二个和第三个表达式为可选，分别表示用来作全局和局部名字空间的 PyDictionary 和 PyStringMapping。如果只提供全局变量，则 locals 为缺省。如果未提供任何名字空间，则使用目前的名字空间</p> <p>语法： "exec" expression ["in" expression ["," expression]]</p> <p>例子： &gt;&gt;&gt; exec ("print 'The exec method is used to print this' ") The exec method is used to print this</p>
for	<p>for 语句是一个循环结构，对每个序列元素循环执行相关的代码块直至遇到 break 语句为止。所以所提供的表达式的计算结果必须为序列。在将每个序列值绑定到 for 语句后面指定的变量时有一个隐含的名字。可选的 else 语句执行是在所有的序列都已结束时（意味着如果 break 语句在循环块中，它将不会被执行）</p> <p>语法： "for" variable "in" expression ":"     code-block     ["else:" ]         code-block</p> <p>例子： &gt;&gt;&gt; for x in (1, 2, 3): ...     print x, "in first code-block" ... else: ...     print "In second code-block" ... 1 in first code-block 2 in first code-block 3 in first code-block In second code-block</p>
global	<p>显式地指明指定的名字从 globals 而非 locals 名字空间引用</p> <p>语法： "global" identifier ["," identifier] *</p> <p>例子： &gt;&gt;&gt; var = 10 &gt;&gt;&gt; def test (): ...     global var</p>

(续)

语句	描述
	<pre> ...     print var # try and print the global identifier 'var' ...     var = 20 # assign to 'var' in local namespace ... &gt;&gt;&gt; test () 10 </pre>
if	<p>if语句指定一个有条件执行的代码块。代码块可用 if、elif 和 else 语句定义。只有在它所对应的表达式为真时，该代码块才被执行。执行的代码块是第一个表达式计算为真的代码块。如果所有的表达式都为假则执行 else 代码块</p> <p>语法：</p> <pre> "if" expression:     code-block "elif" expression:     code-block "else":     code-block </pre> <p>例子：</p> <pre> &gt;&gt;&gt; a, b = 0, 1 &gt;&gt;&gt; if a == b: ...     print "variable a equals variable b" ... elif a &gt; b: ...     print "variable a is greater than b" ... else: ...     print "variable a is less than b" ... variable a is less than b </pre>
import	<p>引入指定的 Python 包、Python 模块、Java 包或 Java 类。它创建一个与调用的名字空间引入绑定的名字。你用 as 修饰符则可选择性地改变其名字</p> <p>语法：</p> <pre> import module-name OR from module-name import names OR import module-name as new-name </pre> <p>例子：</p> <pre> &gt;&gt;&gt; import sys &gt;&gt;&gt; from java import util &gt;&gt;&gt; import os as myOS &gt;&gt;&gt; from sys import packageManager as pm </pre>
pass	<p>该语句没有什么功能，只为一个占位符</p> <p>语法：</p> <pre> "pass" </pre> <p>例子：</p> <pre> &gt;&gt;&gt; for x in (1, 2, 3, 4) ...     pass ... &gt;&gt;&gt; def doNothing(): ...     pass </pre>
print	<p>计算一个表达式，并将结果转换成所需的字符串，将字符串写至 sys.stdout 或任何 &gt; 所指向的文件型对象。文件型的对象是定义了 write 方法的对象</p>

(续)

语句	描述
print	<p>语法： "print" [expression] OR "print &gt;&gt;" fileLikeObject, [expression]</p> <p>例子： &gt;&gt;&gt; print "Hello world" Hello world &gt;&gt;&gt; print &gt;&gt; myFile, "Hello world" &gt;&gt;&gt;</p>
raise	<p>raise 语句产生一个异常。它之后有三个可选的表达式分别表示异常的类型、值与跟踪 (traceback)。类型可以是异常类，一个表示异常类的字符串或异常类的实例。值或第二个表达式为异常类的构建参数。异常的构建函数只有在第二个参数为元组时才接受一个参数，那样元组的索引为单独的参数。如果 raise 语句的第一个表达式为实例，第二个表达式或值表达式必须为 None。可选的第三个表达式必须为跟踪 (traceback) 对象</p> <p>语法： "raise" [expression [, expression [, traceback]]] 例子： &gt;&gt;&gt; raise ValueError,"No value provided" Traceback (innermost last):   File "&lt; console &gt;", line 1, in ? ValueError: no value provided</p>
return	<p>终止它所在的方法或函数的执行，返回括号内表达式的值。如果没有表达式，返回 None</p> <p>语法： "return" [expression] 例子： &gt;&gt;&gt; def someFunction (): ...     return "This string is the return value" ... &gt;&gt;&gt; print someFunction () This string is the return value</p>
try/except	<p>try/except 是 Python 的异常处理机制，正如 Java 的 try/catch 语句一样。try 代码块一直执行到它完成或遇到一个错误为止。如果遇到错误（产生异常），系统会立即转向合适的 except 语句去处理异常。如果使用 try/finally 语句，则不管 try 代码块有何异常，finally 块都会执行</p> <p>语法： "try:" code-block "except" [expression [, "target"]]: "code-block"   ["else:" code-block] OR "try:" code-block "finally:" code-block 例子： &gt;&gt;&gt; try: ...     1/0 ... except ZeroDivisionError, e: ...     print "You cannot divide by zero: ", e ...</p>

(续)

语句	描述
	<pre>You cannot divide by zero: integer division or modulo &gt;&gt;&gt; &gt;&gt;&gt; try: ...     pass ... finally: ...     print "This block always executes" ... This block always executes</pre>
while	<p>while 语句为一循环结构，只要表达式计算的结果为真就一直执行循环体内的语句，直到遇到 break 语句为止</p> <p>语法： "while" expression ":" 例子：</p> <pre>&gt;&gt;&gt; x = 10 &gt;&gt;&gt; while x &gt; 0: ...     print x, ...     x -= 1 ... 10 9 8 7 6 5 4 3 2 1 &gt;&gt;&gt;</pre>

(续)

语句	描述
	<pre>You cannot divide by zero: integer division or modulo &gt;&gt;&gt; &gt;&gt;&gt; try: ...     pass ... finally: ...     print "This block always executes" ... This block always executes</pre>
while	<p>while 语句为一循环结构，只要表达式计算的结果为真就一直执行循环体内的语句，直到遇到 break 语句为止</p> <p>语法： "while" expression ":" 例子：</p> <pre>&gt;&gt;&gt; x = 10 &gt;&gt;&gt; while x &gt; 0: ...     print x, ...     x -= 1 ... 10 9 8 7 6 5 4 3 2 1 &gt;&gt;&gt;</pre>