Twisted 网络编程必备（一）

0.1 为什么使用 Twisted?

如果你并不准备使用 Twisted，你可能有很多异议。为什么使用 Twisted 而不是其他网络函数库或框架？如下是一些充分的理由：

·基于 Python

Twisted 是使用 Python 编写的，强壮的、面向对象的解释性语言。Python 使它的爱好者充满热情。使用 Python 编程是一种乐趣，易于编写、易于阅读、易于运行。因为 Python 是跨平台的，所以可以运行 Twisted 程序在 Linux、Windows、Unix 和 MAC 等等系统上。

·异步和事件驱动的

同步网络编程函数库留给开发者一个痛苦的抉择：要么允许程序在网络通信时失去响应，或者引入非常复杂的线程。Twisted 是基于事件的，异步网络通信框架允许编写的程序在处理事件时保持相应，却不需要使用线程。

·多功能

Twisted 包括大量的功能。Email、WEB、news、chat、DNS、SSH、Telnet、RPC、数据库存取或者更多。所有的都为你准备好了。

·灵活

Twisted 提供了高层类允许快速开始。而且并没有感到受限。如果需要高级功能，或者需要自定义网络协议，也是可以的。你可以实现自己的网络协议，控制每一个字节。

·开放源代码

Twisted 是免费的。它包含源代码，按照函数库协议发行。并且欢迎在你的程序中使用 Twisted，不必支付任何费用和担心法律问题。如果希望知道一个对象的运行原理也可以直接看源码。如果你开发出了自己的新功能和扩展，欢迎与我们共享。

·社区支持

Twisted 有一个活跃的社区包含开发者和用户。如果你发现了问题，也可以在邮件列表上找到很多开发者来帮助你。查看第一章的寻找 FAQ 一节。或者进入#twisted 这个 IRC 频道，来与开发者进行在线交流。

·易于整合的平台

一个 Twisted 程序可以在多种服务之间共享数据，可以整合很多工作。比如可以编写 SMTP 到 XMLRPC 的代理，一个通过 SSH 来更新网站的服务，或者一个包含 NNTP 接口的 WEB 讨论组。如果需要在平台之间交换数据，Twisted 是个很好的选择。:if expand("%") == ""|browse confirm w|else|confirm w|endif

0.2 本书包含的内容

本书并不是讲解 Twisted 框架中的每一个类，而是关注于更加真实的例子。本书也会帮助你理解 Twisted 中使用的关键技术和设计模式。如下是主要内容列表：

·安装 Twisted

第一章讲解了下载和安装还有平台相关扩展库。

·使用 TCP 连接

第二章告诉你如何使用 Twisted 建立 TCP 连接，包括服务器和客户端。

·WEB 客户端和服务器

第三、四章讲解了如何使用 Twisted 工作于 WEB 之上。第三章模拟 WEB 客户端。第四章编写了一个示例 HTTP 服务器，实现了基本的等级管理和响应功能。

·WEB 服务和 RPC

Twisted 内置支持了多种 WEB 服务和远程调用方法。第五章讲解了如何在 REST 架构下建立应用。也讲解了如何编写 XMLRPC 和 SOAP 客户端和服务器，和如何将 Python 对象转换为网络连接。

·认证

管理用户和认证权限是很多程序的核心功能之一。第六章讨论了实现方法，并提供了在多种不同应用中具有很好移植性的框架。

·电子邮件客户端和服务器

第七、八章给出了电子邮件支持。第七章的例子展示了如何编写 SMTP、POP3、IMAP 客户端来收发电子邮件。第八章展示了如何构建 SMTP、POP3、IMAP 服务器。

·Usenet 新闻组

第九章讨论了使用 NNTP 协议的 Usenet 客户端和服务器。并展示了如何阅读和发送消息，如何运行 NNTP 服务器，如何使用 NNTP 作为其他方式的接口，如 RSS 等。

·SSH 客户端和服务器

第十章讲解了 SSH 支持。例举了编写 SSH 服务器的程序，并提供了远程可用的 Python 解释器。这一章也展示了如何编写 SSH 客户端来执行远程命令。

·运行和管理 Twisted 程序

第十一章展示了如何使用 Twisted 工具运行和管理应用程序，比如作为后台精灵线程，使用 setuid 和 chroot 限制权限，和写入日志文件。

1.0 快速开始

在你使用 Twisted 进行开发之前，你应该下载并安装。这一章讲解在各种操作系统下的安装过程。也包括将 Twisted 实用工具加入到路径和熟悉文档。其他问题可以到 Twisted 社区提问。

1.1 安装 Twisted

首先应该下载 Twisted 用于安装。可以到 http://twistedmatrix.com/projects/core/下载。为了安装附加属性，还可以安装另外两个可选软件包。

Twisted 网络编程必备（二）

1.1.1 下面该如何做?

到 http://twistedmatrix.com 下载最新版本的 Twisted。然后安装 PyOpenSSL(一个 Python 开源 OpenSSL 库)，这个软件包用于给 Twisted 提供加密传输支持(SSL)。最后，安装 PyCrypto，一个包含了 Python 安全算法的包，用于提供 SSH 支持。这些软件包的下载地址在后面各个节中指定。

为了使用 Twisted 并不一定要安装 PyOpenSSL 和 PyCrypto。但是不安装这些，将无法使用 Twisted 的 SSL 和 SSH 功能，其他功能还是可用的。

1.1.1.1 Windows

到 http://twistedmatrix.com/projects/core/下载 Windows 版 Python2.4 安装包。这个二进制包包含了 Twisted 的核心功能，包括大量的扩展包和协议支持。如果需要安装更多的例子，可以到 http://twistedmatrix.com/products/download 下载"Twisted Dependencies for Windows"。这个安装包已经包含了 PyOpenSSL 和 PyCrypto。

这些页面有可能已经移动到其他页面中了，并且增加了一些新功能。如果找不到这些连接，可以尝试从 http://twistedmatrix.com 访问。

如果一次下载了这三个软件包，运行 Twisted 安装程序。这是一个小的安装包，可以自动检测 Python 安装路径，下一步运行 PyOpenSSL 和 PyCrypto 安装包。这些都是很简单的，也只需要很少的几秒钟。

当编译 Twisted 模块时，可能需要一点时间，进度条会显示当前状态。解包也会耗费几分钟。

证实是否安装成功的方法是导入 twisted 包。

```
$ python

... ...

>>> import twisted

>>>
```

如果"import twisted"执行成功而没有错误，则已经安装好了 Twisted。

1.1.1.2 MAC OS X, Linux, 与 BSD

很多 Linux 和 BSD 的发行版，包括 Debian、Ubuntu、Gentoo、Red Hat、FreeBSD 等等已经预先安装了 Twisted 了，而且 PyOpenSSL 和 PyCrypto 也是可用的。可以检查你的 OS 是否被 Twisted 下载页所提及，或者搜索已经安装包历史可以看到。如果已经安装了，最好确保版本是 Twisted 2.0 或更高，而不是 1.3 的 release 版本。

如果没有预安装 Twisted，则需要下载源码来安装。

1.2 从源码安装

如果你的操作系统上没有预安装 Twisted，则需要从源代码安装。不必担心，从源码安装对于 Python 也很简单。

1.2.1 下面如何做

首先，下载完整的 Twisted 源码包。附带的包可以让 Twisted 功能更多。运行本书的大多数例子都需要完整的安装。一旦下载了软件包，就解压到工作目录。

```
$ tar -xjvf ~/downloads/TwistedSumo-2005-03-22.tar.bz2
```

下一步进入解压后的目录。Twisted 依赖于 zope.interface 包，而这个包已经附带在了 Twisted Sumo 发行包中了。解压 ZopeInterface 压缩包：

```
$ tar -xzvf ZopeInterface-3.0.1.tgz
```

进入 ZopeInterface 目录，运行命令"python setup.py install"。这个命令将会构建和安装 zope.interface 包到 python 安装目录的 lib/site-packages/twisted 目录。这里是需要 Administrator/root 权限的。所以使用 su 或 sudo 来增加权限级别。

```
$ cd ZopeInterface-3.0.1

$ python setup.py install
```

一旦 zope.interface 安装完成，就可以安装 Twisted 了。在 Twisted 目录下，运行"python setup.py install"。这个命令将会编译 Twisted 的 C 模块，并安装 Twisted：

```
$ cd TwistedSumo-2005-03-22

$ python setup.py install
```

如果你已经安装了多个版本的 Python，确保 Twisted 安装在你正在运行的 python 命令对应的那个版本上。书的例子可以运行在 Python 2.3 或更高版本。检查你的 Python 版本通过"python -V"。

祝贺，你已经成功安装了 Twisted。现在测试一下是否安装成功，运行 Python 提示符：

```
$ python
```

```
>>> import twisted

>>>
```

如果"import twisted"语句执行成功则 Twisted 安装成功了。

下一步，下载最新版本的 PyOpenSSL(http://pyopenssl.sourceforge.net)。PyOpenSSL 运行在 OpenSSL 库之上，所以你需要确保 OpenSSL 已经安装了。Mac OS X 已经默认安装了 OpenSSL，并包含了头文件，Linux 和 BSD 的大部分发行版也默认包含了这些。

如果你正巧倒霉安装了一个默认没有 OpenSSL 库的发行版，则需要下载并安装源码包，从这里(http://www.openssl.org)。

为了安装 PyOpenSSL，需要执行如下步骤。首先解压下载文件：

```
$ tar -zxvf PyOpenSSL-0.6.tar.gz
```

然后，选择 PyOpenSSL 库并运行"python setup.py install"，使用 root 用户。

```
$ cd PyOpenSSL-0.6

$ python setup.py install
```

当安装完成了，确保 OpenSSL 包是可用的，可以用导入来测试：

```
$ python

>>> import OpenSSL

>>> import twisted.internet.ssl

>>> twisted.internet.ssl.SSL

<module 'OpenSSL.SSL' from

'....'>
```

如果没有看到任何错误，则已经为 Twisted 成功的安装了 SSL 支持。

最后一个需要安装的包是 PyCrypto。PyCrypto 是 Python 的加密工具集，是 A.M.Kuchling 开发的一个包含了众多加密算法实现的包。Twisted 使用 PyCrypto 来支持 SSH 连接。

从(http://www.amk.ca/python/code/crypto.html)来下载，并解压：

$ tar -xzvf pycrypto-2.0.tar.gz

运行很熟悉的"python setup.py install"来安装，在 PyCrypto 的目录中。

$ cd pycrypto-2.0

$ python setup.py install

确保这个包是否安装成功可以用导入测试。你可以测试 twisted.conch.ssh.transport 模块可以使用 PyCrypto 的 RSA 实现：

$ python

>>> import Crypto

>>> import twisted.conch.ssh.transport

>>> twisted.conch.ssh.transport.RSA

<module 'Crypto.PublicKey.RSA' from

'....'>

如果是如上的显示，则你已经成功的安装了 PyCrypto。在这个是时候，你就可以利用 Twisted，并且使用 SSL 和 SSH 功能了

Twisted 网络编程必备（三）

1.3 添加 Twisted 实用工具到 PATH 路径参数

Twisted 包含了一些实用工具脚本。为了方便使用，应该将他们添加到 PATH 可以找到的地方。

1.3.1 下面如何做

一般来说，所有添加到 PATH 的路径都是可以被 OS 找到的。按照如下步骤来做。

1.3.1.1 Windows

Twisted 的使用工具将会安装到 Python 的 scripts 目录(典型为 c:\Python23\scripts)。Twisted 包含了一些菜单设置程序，可以通过 Programs->Twisted->Twisted Command Prompt。来进入。使用这个菜单入口可以使用提示符模式来运行实用工具，而且已经自动设置好了 PATH 路径了。

1.3.1.2 Linux

Twisted 的使用工具将会安装到 python 二进制目录的相同文件夹(可能是/usr/bin 或/usr/local/bin)，所以需要将这个路径添加到 PATH 路径。

1.3.1.3 Mac OS X

如果正在使用 Mac OS X 2.3 (jaguar)或更新版本，Twisted 已经默认安装在了(/System/Library/Frameworks/Python.framework/Versions/Current/bin。将这个路径添加到 PATH 变量。

```
$ set PATH=$PATH:/System/Library/Frameworks/Python.framework/Versions/Current/bin
```

1.4 使用 Twisted 的文档

Twisted 包含了一些不同类型的文档，包括扩展 API 文档、HOWTO 文档、快速入门和例子程序。十分推荐尽快熟悉这些文档，并在开发过程中提供帮助。

1.4.1 下面如何做

Twisted 的文档是在网站上在线观看的。一个完整的 API 文档在(http://twistedmatrix.com/documents/current/api)。为了编程，你需要多次使用这个文档。这份 API 文档是由代码自动生成的，所用的工具是 Twisted 的一个组件叫做 lore。

Twisted 是由一些子工程构成的，每个子工程包含自己的文档。例如，核心模块文档在(http://twistedmatrix.com/projects/core/documentation)，而关于 WEB 模块的文档在(http://twistedmatrix.com/projects/web/documentation)。在主页有进入每一个子工程的链接。

每个子工程都包含以下类型的文档：

·HOWTOs

描述特殊功能和如何使用这些功能。并不覆盖所有的只是，但可以提供对特定任务的快速入门。包含在 HOWTOs 中的快速入门(tutorial)叫做"Twisted From Scratch"，讲解如何开发一个应用程序，并扩展的讲解一些高级内容。

·Examples

简短而特别的 Twisted 代码，有点像 HOWTOs，但是可以在使用中对特定功能提供有效的帮助。

·Manual pages

这是 HTML 版本的 manpages，描述了如何使用 Twisted 实用工具。

1.4.2 关于

如果想要浏览文档而不通过浏览器，那么可以直接使用 pydoc 工具来浏览。pydoc 并不能展示 Twisted 文档中所有的标志，当仍然很有用。Figure 1-1 展示了 pydoc 展示 twisted.web.http 的信息。

当然，真的想要理解 Twisted 的工作，可以阅读 Twisted 的源代码。因为有些模块为了性能而使用了 C 语言来写，除此之外可以到 site-packages/twisted 目录中查看。或者打开 appropriate.py 文件。

1.5 为自己的问题寻找答案

大多数问题可以从文档中自己找到答案，或者可以从社区得到答案。

1.5.1 下面如何做

有一些有益的社区资源可以提供帮助。首先是邮件列表。"twisted-python"列表是提供一般讨论的。"twisted-web"列表提供了 WEB 应用程序的讨论。如果问题提错了地方，会被自动转到另外一个讨论组。可以在如下地址申请讨论组：

http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python

http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-web

第二，可以在#twisted 和#twisted.web 这两个 IRC 频道讨论问题(http://freenode.net)。这些频道是一直开放且有趣的，问题可以很及时的得到回答。如果不要求及时得到答复，最好还是到邮件列表讨论，这种方式提供更加广泛的关注，并且可以让更多的人分享。

最后的可用资源是 Planet Twisted 社区。地址在 http://planet.twistedmatrix.com，这个站点的 weblog 提供了大量的知识。这是一种很好的获取知识的方式，这也提供了 RSS 订阅方式 http://planet.twistedmatrix.com/rss.xml。

2. 建立简单的客户端和服务器

使用 Twisted 进行开发，需要学习如何使用新的类和对象。这些类是 Twisted 的核心，你将会在你的应用中使用这些类。它们提供了平滑的学习曲线，理解如何使用他们，将会使得使用 Twisted 进行开发更加简便。

这一章展示了如何编写简单的客户端和服务器。并介绍 Twisted 简单的类和工作流程，并展示如何使用它们。

Twisted 网络编程必备（四）

2.1 启动 Twisted 的事件循环

Twisted 是事件驱动的框架。这意味着不再使用特定序列来处理程序逻辑，而是通过被动调

用一些函数来实现。例如，GUI 程序里面使用的"button pressed"事件。设计者不需要事先确定事件何时发生，只需要编写事件的响应函数即可。这就是所谓的事件处理器(event handler)。

每个事件驱动的框架都包含了一个特殊的函数叫做事件循环(event loop)。每次启动，事件循环都会立即运行。这个函数运行时等待事件的发生。当事件发生时，事件循环函数会自动触发相关的事件处理函数。

使用事件循环需要改变一些顺序编程的习惯。一旦开始了事件循环，你将无法控制程序的执行顺序，只会自动响应事件。所以，你需要为程序设计事件处理器。也就是事件发生时的响应。

在 Twisted 中，有一种特殊的对象用于实现事件循环。这个对象叫做 reactor。可以把反应器(reactor)想象为 Twisted 程序的中枢神经。除了分发事件循环之外，反应器还做很多重要的工作：定时任务、线程、建立网络连接、监听连接。为了让反应器可以正常工作，需要启动事件循环。

2.1.1 下面如何做

启动反应器是很简单的，从 twisted.internet 模块导入 reactor 对象。然后调用 reactor.run()来启动反应器的事件循环。下例展示了代码：

```
from twisted.internet import reactor

print 'Running the reactor ...'

reactor.run()

print 'Reactor stopped.'
```

反应器将会一直运行，直到接到停止的通知，可以按下 Ctrl+C 来退出事件循环，并终止程序：

```
$ python runreactor.py

Running the reactor ...

<ctrl-c>

^CReactor stopped.
```

这是一个简洁的例子。尽管反应器运行着，但是没有做任何事情。下面的例子提供了更多有趣的事情，介绍了反应器的 callLater 方法。reactor.callLater 方法用于设置定时事件。这个方法在用于定时在未来执行。比如一个函数已经是一个事件处理器了，这种事件会在一定时间之后启动。下面的例子，函数调用了一定时间之后被调用：

```
from twisted.internet import reactor

import time

def printTime():

    print 'Current time is',time.strftime("%H:%M:%S")

def stopReactor():

    print "Stopping reactor"

    reactor.stop()

reactor.callLater(1,printTime)

reactor.callLater(2,printTime)

reactor.callLater(3,printTime)

reactor.callLater(4,printTime)

reactor.callLater(5,stopReactor)

print 'Running the reactor ...'

reactor.run()

print 'Reactor stopped.'
```

运行这个程序，可以看到如下结果：

```
$ python calllater.py

Running the reactor ...

Current time is 10:33:44
```

Current time is 10:33:45

Current time is 10:33:46

Current time is 10:33:47

Stopping reactor

Reactor stopped.

2.1.2 它是如何工作的

例子 2-1 简介了导入与执行 reactor.run()来启动事件循环。反应器将会一直保持执行直到按下 Ctrl-C，尽管这一段时间什么都不做。在这时，反应器停止了，程序执行到代码的最后一行，并打印出反应器已经停止的消息。

第二个例子使用 reactor.callLater 函数定时执行函数。reactor.callLater 函数包含两个必须参数，等待的秒数，和需要调用的函数。在设置了定时函数调用之后，控制就回到了反应器的事件循环 reactor.run()中了。

Tip:你可以传递附加的参数和键值对到 reactor.callLater 中，这些用于调用指定的函数。例如，reactor.callLater(1,func,True,x="Hello")，将会最终调用 func(True,x="Hello")，在一秒钟之后。

第一个定时函数是 printTime()，简单显示了当前时间。第五个定时函数是 stopReactor()，其中调用了 reactor.stop()，导致了反应器退出了事件循环。这也是为什么不需要按下 Ctrl-C 而自动退出了事件循环的原因。

Tip:在这种情况下仍然可以按下 Ctrl-C 来手动停止事件循环。

注意事件的发生顺序，反应器按照给定的时间来调用指定的函数。一旦反应器开始运行，反应器就会控制事件循环，并在指定时间调用函数。反应器在被告知停止之前会一直运行，直到 reactor.stop()调用。一旦反应器停止了，程序将继续处理最后一行，显示反应器停止的消息。

Tip:在实际应用中，reactor.callLater 是常用于超时处理和定时事件。可以设置函数按照指定的时间间隔来执行关闭非活动连接或者保存内存数据到硬盘。

2.1.3 建立(establishing)一个 TCP 连接

所有的网络应用程序都必须做一个简单的步骤，开启一个连接。接着可以发送邮件、传递文件、看电影等等，但是在所有这些之前，必须在电脑之间建立一个连接。本节讲解了如何使用 reactor 开启 TCP 连接。

2.1.4 下面如何做

调用 reactor.connectTCP()方法打开一个 TCP 连接，传递一个 ClientFactory 对象作为第三个参数。ClientFactory 对象等待连接被建立，然后创建一个 Protocol 对象来管理连接中的数据流。下面的例子 2-3 展示了如何在电脑和 Internet 之间建立一个连接(www.google.com)：

```
from twisted.internet import reactor,protocol

class QuickDisconnectedProtocol(protocol.Protocol):

    def connectionMade(self):

        print "Connected to %s."%self.transport.getPeer().host

        self.transport.loseConnection()

class BasicClientFactory(protocol.ClientFactory):

    protocol=QuickDisconnectProtocol

    def clientConnectionLost(self,connector,reason):

        print 'Lost connection: %s'%reason.getErrorMessage()

        reactor.stop()

    def clientConnectionFailed(self,connector,reason):

        print 'Connection failed: %s'%reason.getErrorMessage()

        reactor.stop()

reactor.connectTCP('www.google.com',80,BasicClientFactory())
```

reactor.run()

当运行这个例子的时候，可以看到如下输出：

$ python connection.py

Connected to www.google.com

Lost connection: Connection was closed cleanly.

除非你的电脑没有在线。在这种情况下，会看到如下错误信息：

$ python connection.py

Connection failed: DNS lookup failed: address 'www.google.com' not found.

2.1.5 它是如何工作的

这里有两个主要的类用于作为客户端工作，ClientFactory 和 Protocol。这些类被设计成处理连接中所有可能运到的事件：成功建立连接、连接失败、连接断开、数据传送等等。

ClientFactory 和 Protocol 有严格的不同。ClientFactory 的工作是管理连接事件，并且创建 Protocol 对象处理每一个成功的连接。一旦连接建立，Protocol 对象就接管下面的工作了，包括收发数据和决定是否关闭连接。

Tip:名字"Factory"在 ClientFactory 是来自于 Protocol 的请求，响应每一个成功的连接。

例子 2-3 定义了自定义的 Protocol 叫做 QuickDisconnectProtocol，继承自 protocol.Protocol。它重载了一个方法 connectMade。这个方法连接成功时运行，在 reactor 刚刚成功建立了连接，然后 ClientFactory 创建了 QuickDisconnectProtocol 的实例时。有如他的名字，QuickDisconnectProtocol 对象在打印信息之后就马上关闭了。

Tip:Protocol 对象有一个属性叫做 transport，包含了当前活动连接对象。

BasicClientFactory 是继承自 protocol.ClientFactory 的类。它首先设置了类变量 protocol 为 QuickDisconnectProtocol。这个类的实例被创建用于管理成功的连接。

BasicClientFactory 重载了 ClientFactory 的两个方法，clientConnectionLost 和 clientConnectionFailed。这两个方法是事件处理器。clientConnectionFailed 在反应器无法建立

连接时被调用。clientConnectionLost 在建立的连接被关闭或断开时调用。

告知反应器建立 TCP 连接，按照例子 2-3 的方法是调用 reactor.connectTCP：

reactor.connectTCP('www.google.com',80,BasicClientFactory())

这一行告知反应器建立一个 TCP 连接到服务器 www.google.com 的 80 端口，通过 BasicClientFactory 来管理连接。

Twisted 网络编程必备（五）

2.2 使用非同步的方式工作的结果

除了反应器 reactor 之外，Deferred 可能是最有用的 Twisted 对象。你可能在 Twisted 程序中多次用到 Deferred，所有有必要了解它是如何工作的。Deferred 可能在开始的时候引起困惑，但是它的目的是简单的：保持对非同步活动的跟踪，并且获得活动结束时的结果。

Deferred 可以按照这种方式说明：可能你在饭店中遇到过这个问题，如果你在等待自己喜欢的桌子时，在一旁哼哼小曲。带个寻呼机是个好主意，它可以让你在等待的时候不至于孤零零的站在那里而感到无聊。你可以在这段时间出去走走，到隔壁买点东西。当桌子可用时，寻呼机响了，这时你就可以回到饭店去你的位置了。

一个 Deferred 类似于这个寻呼机。它提供了让程序查找非同步任务完成的一种方式，而在这时还可以做其他事情。当函数返回一个 Deferred 对象时，说明获得结果之前还需要一定时间。为了在任务完成时获得结果，可以为 Deferred 指定一个事件处理器。

2.2.1 下面如何做?

当编写一个启动非同步操作的函数时，返回一个 Deferred 对象。当操作完成时，调用 Deferred 的 callback 方法来返回值。如果操作失败，调用 Deferred.errback 函数来跑出异常。例子 2-4 展示了程序使用 Deferred 使用非同步操作的连接例子，连接一个服务器的端口。

当调用一个可以返回 Deferred 的函数时，使用 Deferred.addCallback 方法指定返回结果时调用的函数。使用 Deferred.addErrback 方法指定执行发生异常时调用的函数。

```
from twisted.internet import reactor,defer,protocol
class CallbackAndDisconnectProtocol(protocol.Protocol):
    def connectionMade(self):
        self.factory.deferred.callback("Connected!")
        self.transport.loasConnection()
class ConnectionTestFactory(protocol.ClientFactory):
    protocol=CallbackAndDisconnectProtocol
    def __init__(self):
        self.deferred=defer.Deferred()
    def clientConnectionFailed(self,connector,reason):
        self.deferred.errback(reason)
def testConnect(host,port):
    testFactory=ConnectionTestFactory()
    reactor.connectTCP(host,port,testFactory)
    return testFactory.deferred
def handleSuccess(result,port):
    print "Connect to port %i"%port
    reactor.stop()
def handleFailure(failure,port):
    print "Error connecting to port %i: %s"%(
        port,failure.getErrorMessage())
    reactor.stop()
```

```
if __name__=="__main__":
    import sys
    if not len(sys.argv)==3:
        print "Usage: connectiontest.py host port"
        sys.exit(1)
    host=sys.argv[1]
    port=int(sys.argv[2])
    connecting=testConnect(host,port)
    connecting.addCallback(handleSuccess,port)
    connecting.addErrback(handleFailure,port)
    reactor.run()
```

运行这个脚本并加上服务器名和端口这两个参数会得到如下输出：

```
$ python connectingtest.py oreilly.com 80
Connecting to port 80.
```

或者，如果连接的端口是关闭的：

```
$ python connectiontest.py oreilly.com 81
Error connecting to port 81: Connection was refused by other side: 22: Invalid argument.
```

或者连接的服务器并不存在：

```
$ python connectiongtest.py fakesite 80
Error connecting to port 80: DNS lookup failed: address 'fakesite' not found.
```

2.2.2 它是如何工作的?

类 ConnectionTestFactory 是 ClientFactory 的子类，并且含有 Deferred 对象作为属性。当连接完成时，CallbackAndDisconnectProtocol 的 connectionMade 方法会被调用。connectionMade 会调用 self.factory.deferred.callbakc 加上任意值标识调用成功。如果连接失败，ConnectionTestFactory 的 clientConnectionFailed 方法会被调用。第二个参数 clientConnectionFailed 的 reason 是 twisted.python.failure.Failure 对象，封装了异常，并描述了异常发生的原因。clientConnectionFailed 传递了 Failure 对象到 self.deferred.errback 来标识操作失败。

testConnecn 函数需要两个参数 host 和 port。用于创建叫做 testFactory 的 ConnectionTestFactory 的类实例，并且传递给 reactor.connectTCP 参数 host 和 port。当连接成功时返回 testFactory.deferred 属性，就是用于跟踪的 Deferred 对象。

例子 2-4 夜展示了两个事件处理器函数：handleSuccess 和 handleFailure。当从命令行运行时，它们需要 host 和 port 两个参数来调用 testConnect，指定了结果 Deferred 到变量 connecting。此时可以使用 connecting.addCallback 和 connecting.addErrback 来设置事件处理器函数。在每一种情况下，传递 port 作为附加参数。因为扩展的参数或者关键字参数将会由 addCallback 或 addErrback 传递给事件处理器，这将会在调用 handleSuccess 和 handleFailure 时得到端口参数作为第二个参数。

Tip:当函数返回 Deferred 时，可以确定事件调用了 Deferred 的 callback 或者 errback 方法。否则代码将永远等待结果。

在调用了 testConnect 和设置事件处理器之后，控制劝将会交给 reactor.run()。依赖于 testConnect 的执行成功与否，将会调用 handleSuccess 或者 handleFailure，打印西那个关的信息并停止反应器。

2.2.3 关于

是否需要保持一串的 Deferred 呢？有时确实需要同时保持多个非同步任务，且并非同时完成。例如，可能需要建立一个端口扫描器运行 testConnect 函数来对应一个范围的端口。为了实现这个，使用 DeferredList 对象，作为例子 2-5，来挂历众多的 Deferred。

```
from twisted.internet import reactor,defer

from connectiontester import testConnect

def handleAllResules(results,ports):
```

```
    for port,resultInfo in zip(ports,results):
        success,result=resultInfo
        if success:
            print 'Connected to port %i' % port
    reactor.stop()
import sys
host=sys.argv[1]
ports=range(1,201)
testers=[testConnect(host,port) for port in ports]
defer.DeferredList(testers,consumeErrors=True).addCallback(handleAllResults,ports)
reactor.run()
```

运行 portscan.py 脚本并传递 host 参数。将会自动扫描 1-200 端口并报告结果。

```
$ python portscan.py localhost
Connected to port 22
Connected to port 23
Connected to port 25
... ...
```

例子 2-5 使用了 Python 的列表(list)的形式来创建一个列表存储由 testConnect()返回的 Deferred 对象的列表。每一个 testConnect()使用 host 参数，并测试 1-200 端口。

```
testers=[testConnect(host,port) for port in ports]
```

例子 2-5 使用列表方式包装了 Deferred 对象成为 DeferredList 对象。DeferredList 将会跟踪所有的 Deferred 对象的结果并传递作为首参数。当它们都完成了的时候，将会回调按照(success,result)的格式。在 Deferred 完成时，第一个回传值是 TRue 和第二个参数是 Deferred 传回结果。如果执行失败，则第一个参数是 False 且第二个参数是 Failure 对象包装的异常。consumeErrors 键设置为 True 时告知 DeferredList 完成了吸收 Deferred 的所有错误。否则将

会看到弹出的错误信息。

当 DeferredList 完成时，results 将会被传给 handleAllResults 函数，列表中的每个元素都会被扫描。handleAllResults 使用了 zip 函数来匹配每个结果的端口。对每一个端口如果有成功的连接则打印信息。最终反应器会结束并结束程序。

Twisted 网络编程必备（六）

2.3 发送和接收数据

一旦 TCP 连接被建立，就可以用于通讯。程序可以发送数据到另一台计算机，或者接收数据。

2.3.1 下面如何做?

使用 Protocol 的子类可以发送和接收数据。重载 dataReceived 方法用于控制数据的接收，仅在接收到数据时才被调用。使用 self.transport.write 来发送数据。

例子 2-6 包括了 DataForwardingProtocol 的类，可以将接收到的数据写入 self.output。这个实现创建了简单的应用程序，类似于 netcat，将收到的数据传递到标准输出，用于打印接收的数据到标准输出。

```
from twisted.internet import stdio,reactor,protocol

from twisted.protocols import basic

import re

class DataForwardingProtocol(protocol.Protocol):

    def __init__(self):

        self.output=None

        self.normalizeNewLines=False

    def dataReceived(self,data):
```

```
            if self.normalizeNewLines:
                data=re.sub(r"(\r\n|\n)","\r\n",data)
            if self.output:
                self.output.write(data)
class StdioProxyProtocol(DataForwardingProtocol):
    def connectionMade(self):
        inputForwarder=DataForwardingProtocol()
        inputForwarder.output=self.transport
        inputForwarer.normalizeNewLines=True
        stdioWarpper=stdio.StandardIO(inputForwarder)
        self.output=stdioWarpper
        print "Connected to server. Press Ctrl-C to close connection."
class StdioProxyFactory(protocol.ClientFactory):
    protocol=StdioProxyProtocol
    def clientConnectionLost(self,transport,reason):
        reactor.stop()
    def clientConnectionFailed(self,transport,reason):
        print reason.getErrorMessage()
        reactor.stop()
if __name__=='__main__':
    import sys
    if not len(sys.argv)==3:
```

```
        print "Usage: %s host port "__file__

        sys.exit(1)

reactor.connectTCP(sys.argv[1],int(sys.argv[2]),StdioProxyFactory())

reactor.run()
```

运行 dataforward.py 并传入 host 和 port 两个参数。一旦连接就可以将所有来自服务器的消息送回服务器，所有接收的数据也同时显示在屏幕上。例如可以手动连接一个 HTTP 服务器，并发送 HTTP 请求到 oreilly.com：

```
$ python dataforward.py oreilly.com 80

Connected to server.    Press Ctrl-C to close connection.

HEAD / HTTP/1.0

Host: oreilly.com

                                    <--空行

HTTP/1.1 200 OK

……
```

2.3.2 它们是如何工作的?

例子 2-6 开始于定义类 DataForwardingProtocol。这个协议用于接受数据并存入 self.output，这个属性可以用 write 方法访问，如同 self.output.write。DataForwardingOutputProtocol 包含一个叫做 normalizeNewLines 的属性。如果这个属性设置为 True，将会由 Unix 风格的\n 换行改变为\r\n 这种常见的网络换行方式。

作为 DataForwardingProtocol 的子类 StdioProxyProtocol 类具体负责工作。一旦连接被建立，将会创建一个叫做 inputForwarder 的 DataForwardingProtocol 实例，并设置输出为 self.transport。然后包装 twisted.internet.stdio.StandardIO 的实例 inputForwarder，以标准 IO 的方式代替网络连接。这一步对所有的通过 StdioProxyProtocol 方式的网络连接都有效。最终设置 StdioProxyProtocol 的 output 属性到值 stdioWarpper，所以数据的接收工作定义到了标准输出。

协议定义之后，很容易定义 StdioProxyFactory，将其 protocol 属性设置为 StdioProxyProtocol，并且处理反应器的停止和连接、失败工作。调用 reactor.connectTCP 连接，然后依靠 reactor.run() 来控制事件循环。

2.4 接受客户端的连接

前面的实验都是讲解如何作为客户端进行连接。Twisted 也同样适合于编写网络服务器，用于等待客户端连接。下面的实验将会展示如何编写 Twisted 服务器来接受客户端连接。

2.4.1 下面如何做?

创建一个 Protocol 对象来定义服务器行为。创建一个 ServerFactory 对象并引用 Protocol，并传递给 reactor.listenTCP。例子 2-7 展示了简单的 echo 服务器，简单的返回客户端发来的信息。

```
from twisted.internet import reactor,protocol
from twisted.protocols import baisc
class EchoProtocol(basic.LineReceiver):
    def lineReceived(self,line):
        if line=='quit':
            self.sendLine("Goodbye.")
            self.transport.loseConnection()
        else:
            self.sendLine("You said: "+line)
class EchoServerFactory(protocol.ServerFactory):
    protocol=EchoProtocol
```

```
if __name__=="__main__":
    port=5001
    reactor.listenTCP(port,EchoServerFactory())
    reactor.run()
```

当这个例子运行时，将会在 5001 端口监听，并报告已经建立的连接。

```
$python echoserver.py
Server running, Press Ctrl-C to stop.
Connection from  127.0.0.1
Connection from  127.0.0.1
```

在另一个终端，使用 netcat、telnet 或者 dataforward.py(例子 2-6)来连接服务器。将会返回键入的例子。输入 quit 来关闭连接。

```
$ python dataforward.py localhost 5001
Connected to server.  Press Ctrl-C to close connection.
hello
You said: hello
twisted is fun
You said: twisted is fun
quit
Goodbye.
```

2.4.2 它们是如何工作的?

Twisted 服务器使用相同的 Protocol 类作为客户端。为了复用，EchoProtocol 继承了 twisted.protocols.basic.LineReceiver，作为 Protocol.LineReceiver 的轻量级实现，可以自动按照获得的行来产生处理。当 EchoProtocol 接收到一个行时，就会返回收到的行，除非遇到了

'quit'将退出。

下一步，定义了 EchoServerFactory 类。EchoServerFactory 继承自 ServerFactory，作为 ClientFactory 的服务器端，并设置了 EchoProtocol 为 protocol 属性。EchoServerFactory 的实例作为第二个参数传递给 reactor.listenTCP，第一个参数是端口号。

Twisted 网络编程必备（七）

3.0 WEB 客户端

大部分上网活动都是通过 WEB 浏览器来访问 WEB 的。所以通过 HTTP 协议制作客户端来访问 WEB 是很有意义的。这一章讲解如何使用 twisted.web.client 模块来操作互联网资源，包括下载页面，使用 HTTP 认证，上传文件，使用 HTTP 字段等。

3.1 下载网页

最简单和常用的任务莫过于通过 WEB 客户端来下载网页了。客户端连接服务器，发送 HTTP 的 GET 请求，接收包含网页的 HTTP 响应。

3.1.1 下面如何做?

可以依靠 Twisted 内置的协议来快速进入工作。twisted.web 包包含了完整的 HTTP 实现，免去自行开发 Protocol 和 ClientFactory 的工作。在未来，它还将包括建立 HTTP 请求的工具函数。获取一个网页，使用 twisted.web.client.getPage。例子 3-1 是 webcat.py 脚本，用于通过 URL 获得网页。

```
from twisted.web import client

from twisted.internet import reactor

import sys

def printPage(data):

    print data
```

```
    reactor.stop()

def printError(failure):
    print >> sys.stderr, "Error: ", failure.getErrorMessage()
    reactor.stop()

if len(sys.argv)==2:
    url=sys.argv[1]
    client.getPage(url).addCallback(printPage).addErrback(printError)
    reactor.run()
else:
    print "Usage: webcat.py <URL>"
```

给 webcat.py 脚本一个 URL，将会显示网页代码：

```
$ python webcat.py http://www.oreilly.com/
<!DOCTYPE HTML PUBLIC "-//....
```

3.1.2 它们如何工作?

函数 printPage 和 printError 是简单的事件处理器，用于打印下载的网页和错误信息。最重要的一行是 client.getPage(url)。这个函数返回了 Deferred 对象，用于非同步状态下通知下载完成事件。

注意如何在一行中添加 Deferred 的回调函数。这是可能的，因为 addCallback 和 addErrback 都返回 Deferred 对象的引用。因此语句：

```
d=deferredFunction()
d.addCallback(resultHandler)
```

d.addCallback(errorHandler)

等同于：

deferredFunction().addCallback(resultHandler).addErrback(errorHandler)

这两种方式都可以用，下面那种方式在 Twisted 代码中更加常见。

3.1.3 关于

是否有需要将网页写入磁盘上呢？这个高级功能在使用 3-1 例子脚本下载巨大文件时可能会出问题。一个更好的解决方法是在下载时将数据写入临时文件，而在下载完成时在从临时文件中全部读出。

twisted.web.client 包含了 downloadPage 函数，类似于 getPage，但是将文件写入文件。调用 downloadPage 并传入 URL 做首参数，文件名或文件对象做第二个参数。如下例 3-2：

```
from twisted.web import client
import tempfile
def downloadToTempFile(url):
    """
    传递一个 URL，并返回一个 Deferred 对象用于下载完成时的回调
    """
    tmpfd,tempfilename=tempfile.mkstemp()
    os.close(tmpfd)
    return client.downloadPage(url,tempfilename).addCallback(returnFilename,tempfilename)
def returnFilename(result,filename):
    return filename
if __name__=='__main__':
```

```
import sys,os
from twisted.internet import reactor
def printFile(filename):
    for line in  file(filename,'r+b'):
        sys.stdout.write(line)
    os.unlink(filename) #删除文件
    reactor.stop()
def printError(failure):
    print >> sys.stderr, "Error: ",failure.getErrorMessage()
    reactor.stop()
if len(sys.argv)==2:
    url=sys.argv[1]
    downloadToTempFile(url).addCallback(printFile).addErrback(printError)
    reactor.run()
else:
    print "Usage: %s <URL>"%sys.argv[0]
```

函数 downloadToTempFile 在调用 twisted.web.client.downloadPage 时返回 Deferred 对象。downloadToTempFile 还添加了 returnFilename 作为 Deferred 的回调，将临时文件名作为附加参数。这意味着当 downloadToTempFile 返回时， reactor 将会调用 returnFileName 作为 downloadToTempFile 的首个参数，而文件名作为第二个参数。

例子 3-2 注册了另外两个回调函数到 downloadToTempFile。记住 downloadToTempFile 返回的 Deferred 已经包含了 returnFilename 作为回调处理器。因此，在结果返回时，returnFilename 将会首先被调用。这个函数的结果被用于调用 printFile。

3.2 存取受到密码保护的页面

有些网页需要认证。如果你正在开发 HTTP 客户端应用，那么最好准备好处理这些，并在需要时给出用户名和密码。

3.2.1 下面如何做?

如果一个 HTTP 请求以 401 的状态码执行失败，则是需要认证的。这时传递用户提供的登录用户名和密码到 Authorization 字段，有如例子 3-3 中那样：

```
from twisted.web import client,error as weberror
from twisted.internet import reactor
import sys,getpass,base64
def printPage(data):
    print data
    reactor.stop()
def checkHTTPError(failure,url):
    failure.trap(weberror.Error)
    if failure.value.status=='401':
        print >> sys.stderr,failure.getErrorMessage()
        #要求用户名和密码
        username=raw_input("User name: ")
        password=getpass.getpass("Password: ")
        basicAuth=base64.encodestring("%s:%s"%(username,password))
        authHeader="Basic "+basicAuth.strip()
```

```
            #尝试再次获取页面，已经加入验证信息
            return client.getPage(url,headers={"Authorization":authHeader})
        else:
            return failure
def printError(failure):
        print >> sys.stderr, 'Error: ',failure.getErrorMessage()
        reactor.stop()
if len(sys.argv)==2:
        url=sys.argv[1]
        client.getPage(url).addErrback(
            checkHTTPError,url).addCallback(
            printPage).addErrback(
            printError)
        reactor.run()
else:
        print "Usage: %s <URL>"%sys.argv[0]
```

运行 webcat3.py 并传递 URL 作为附加参数，将会尝试下载页面。如果收到了 401 错误，则会询问用户名和密码之后自动重试：

```
$ python webcat3.py http://example.com/protected/page
401 Authorization Required
User name: User
Password: <输入密码>
<html>
```

……

3.2.2 它们如何工作?

这个例子使用了扩展的错误处理器。它首先给client.getPage添加了Deferred，在添加printPage和printError之前。这样做给了checkHTTPError以机会在其他处理器之前来处理client.getPage的错误。

作为一个errback的错误处理器，checkHTTPError将被twisted.python.failure.Failure对象调用。Failure对象封装了跑出的异常，记录了异常时的traceback，并添加了几个有用的方法。checkHTTPError开始使用了Failure.trap方法来证实异常是twisted.web.error.Error类型的。如果不是，trap将会重新抛出异常，退出当前函数并允许错误传递到下一个errback处理器，printError。

然后，checkHTTPError检查HTTP响应状态码.failure.value是一个Exception对象，并且是twisted.web.error.Error的对象，已知的status属性包含了HTTP响应状态码。如果状态码不是401，则返回原始错误，通过把错误传递给printError。

如果状态码是401，checkHTTPError会执行。它会提示输入用户名和密码，并编码成HTTP的Authorization头字段。然后调用client.getPage，返回Deferred对象。这将导致一些很cool的事情发生，反应器等待第二次调用结果，然后调用printPage或者printError来处理结果。实际上，checkHTTPError将会提示"通过另外一个Deferred来处理错误，等待Deferred的结果，并且在另外一行中进行事件处理"。这个技术是很强大的，并且可以在Twisted程序中多次使用。

最终的结果有如前面的例子：使用printPage或者printError来输出结果。当然，如果初始化请求成功了，则不需要认证，checkHTTPError将不会被调用，返回的结果也将直接调用printPage。

Twisted网络编程必备（八）

3.3 上传文件

以用户的观点来说，没有什么简单的方法使得上传文件到网页。使用HTML表单选择文件并按下提交按钮才可以上传。正因为如此很多站点提供了难用的上传方式。有时当你需要上传文件而又不能使用浏览器时。可能需要自己开发程序来上传照片、基于WEB的文件管理

系统等等。下面的例子展示了使用 Twisted 开发 HTTP 客户端来上传文件的方法。

3.3.1 下面如何做?

首先，对键值对编码，并将上传文件编入 multipart/form-data 的 MIME 文件。Python 和 Twisted 都无法提供简便的方法来实现，但你可以也可以不需要太大努力的自己实现一个。然后传递已编码的表单数据到 formdata 键，并传递给 client.getPage 或 client.downloadPage，并使用 HTTP 的 POST 方法。然后就可以执行 getPage 或 downloadPage 来获得 HTTP 响应了。例子 3-4(validate.py)展示了如何按照 W3C 标准上传文件，并保存响应到本地文件，然后显示在用户浏览器中。

```
from twisted.web import client
import os,tempfile,webbrowser,random
def encodeForm(inputs):
    """
    传递一个字典参数 inputs 并返回 multipart/form-data 字符串，包含了 utf-8 编码的
    数据。键名必须为字符串，值可以是字符串或文件对象。
    """
    getRandomChar=lambda:chr(random.choice(range(97,123)))
    randomChars=[getRandomChar() for x in range(20)]
    boundary="---%s---"%''.join(randomChars)
    lines=[boundary]
    for key,val in  inputs.items():
        header='Content-Disposition: form-data; name="%s"'%key
        if hasattr(val,'name'):
            header+=';filename="%s"'%os.path.split(val.name)[1]
```

```
            lines.append(header)
            if hasattr(val,'read'):
                lines.append(val.read())
            else:
                lines.append(val.encode('utf-8'))
            lines.append('')
            lines.append(boundary)
        return "\r\n".join(lines)
def showPage(pageData):
        #将数据写入临时文件并在浏览器中显示
        tmpfd,tmp=tempfile.mkstemp('.html')
        os.close(tmpfd)
        file(tmp,'w+b').write(pageData)
        webbrowser.open('file://'+tmp)
        reactor.stop()
def handleError(failure):
        print "Error: ",failure.getErrorMessage()
        reactor.stop()
if __name__=='__main__':
        import sys
        from twisted.internet import reactor
        filename=sys.argv[1]
```

```
    fileToCheck=file(filename)
    form=encodeForm({'uploaded_file':fileToCheck})
    postRequest=client.getPage(
        'http://validator.w3.org/check',
        method="POST",
        headers={'Content-Type':'multipart/form-data; charset=utf-8',
                 'Content-Length':str(len(form))},
        postdata=form)
    postRequest.addCallback(showPage).addErrback(handleError)
    reactor.run()
```

运行 validate.py 脚本并在第一个参数给出 HTML 文件名：

```
$ python validate.py test.html
```

一旦发生了错误，必须获得 validation 的在浏览器中的错误报告。

3.3.2 它们是如何做的?

按照 W3C 标准页面，例如 http://validator.w3.org 包含表单如下：

```
<form method="POST" enctype="multipart/form-data" action="check">
    <label title="Choose a Local file to Upload and Validate"
        for="uploaded_file">Local File:
    <input type="file" name="uploaded_file" size="30" /></label>
    <label title="Submit file for validation">
```

<input type="submit" value="Check" /></label>

</form>

这样可以建立一个 HTTP 客户端发送与浏览器相同的数据。函数 encodeForm 输入一个字典包含了以键值对方式引入的上传文件对象，并返回已经按照 multipart/form-data 的 MIME 编码文档的字符串。当 validate.py 运行时，打开特定文件作为第一个参数并传递 encodeForm 作为'uploaded_file'的值。这将会返回待提交的有效数据。

validate.py 然后使用 client.getPage 来提交表单数据，传递头部字段包括 Content-Length 和 Content-Type。showPage 回调处理器得到返回的数据并写入临时文件。然后使用 Python 的 webbrowser 模块调用默认浏览器来打开这个文件。

3.4 检测网页更新

当使用流行的 HTTP 应用打开一个 RSS 收集时，将会自动下载最新的 RSS(或者是 Atom)的 blog 更新。RSS 收集定期下载最新的更新，典型值是一个小时。这个机制可以减少浪费大量的带宽。因为内容很少更新，所以客户端常常重复下载相同数据。

为了减少浪费网络资源，RSS 收集(或者其他请求页面的方式)推荐使用条件 HTTP GET 请求。通过在 HTTP 请求的头部字段添加条件，客户端可以告知服务器仅仅返回已经看过的时间之后的更新。当然，一种条件是在上次下载后是否修改过。

3.4.1 下面如何做?

首次下载网页时保持跟踪头部字段。查找 ETag 头部字段，这定义了网页的修正版本，或者是 Last-Modified 字段，给出了网页修改时间。下次请求网页时，发送头部 If-None-Match 加上 ETag 值，或者 If-Modified-Since 加上 Last-Modified 值。如果服务器支持条件 GET 请求，如果网页没有修改过则返回 304 Unchanged 响应。

getPage 和 downloadPage 函数是很方便的，不过对于控制条件请求是不可用的。可以使用稍微低层次的 HTTPClientFactory 接口来实现。例子 3-5 演示了 HTTPClientFactory 测试网页是否更新。

```
from twisted.web import client

class HTTPStatusChecker(client.HTTPClientFactory):
```

```
    def __init__(self,url,headers=None):
        client.HTTPClientFactory.__init__(self,url,headers=headers)
        self.status=None
        self.deferred.addCallback(
            lambda data: (data,self.status,self.response_headers))
    def noPage(self,reason): #当返回非 200 响应时
        if self.status=='304' #页面未曾改变
            client.HTTPClientFactory.page(self,'')
        else:
            client.HTTPClientFactory.noPage(self,reason)
def checkStatus(url,contextFactory=None,*args,**kwargs):
    scheme,host,port,path=client._parse(url)
    factory=HTTPStatusChecker(url,*args,**kwargs)
    if scheme=='https':
        from twisted.internet import ssl
        if contextFactory is None:
            contextFactory=ssl.ClientContextFactory()
        reactor.connectSSL(host,port,factory,contextFactory)
    else:
        reactor.connectTCP(host,port,factory)
    return factory.deferred
def handleFirstResult(result,url):
```

```
    data,status,headers=result
    nextRequestHeaders={}
    eTag=headers.get('etag')
    if eTag:
        nextRequestHeaders['If-None-Match']=eTag[0]
    modified=headers.get('last-modified')
    if modified:
        nextRequestHeaders['If-Modified-Since']=modified[0]
    return checkStatus(url,headers=nextRequestHeaders).addCallback(
        handleSecondResult)
def handleSecondResult(result):
    data,status,headers=result
    print 'Second request returned status %s:'%status,
    if status=='200':
        print 'Page changed (or server does not support conditional request).'
    elif status=='304':
        print 'Page is unchanged.'
    else:
        print 'Unexcepted Response.'
    reactor.stop()
def handleError(failure):
    print 'Error',failure.getErrorMessage()
```

```
    reactor.stop()

if __name__=='__main__':
    import sys
    from twisted.internet import reactor
    url=sys.argv[1]
    checkStatus(url).addCallback(
        handleFirstResult,url).addErrback(
        handleError)
    reactor.run()
```

运行 updatecheck.py 脚本并传入 URL 作为首参数。它首先尝试下载 WEB 页面，使用条件 GET。如果返回了 304 错误，则表示服务器未更新网页。常见的条件 GET 适用于静态文件，如 RSS 种子，但不可以是动态生成页面，如主页。

```
$ python updatecheck.py http://slashdot.org/slashdot.rss
Second request returned status 304: Page is unchanged
$ python updatecheck.py http://slashdot.org/
Second request returned status 200: Page changed
(or server does not support conditional requests).
```

3.4.2 它们如何做?

HTTPStatusChecker 类是 client.HTTPClientFactory 的子类。它负责两件重要的事情。在初始化时使用 lambda 添加回调函数到 self.deferred。这个匿名函数将会在结果传递到其他处理器之前截获 self.deferred。这将会替换结果(已下载数据)为包含更多信息的元组：数据、HTTP 状态码、self.response_headers，这是一个响应字段的字典。

HTTPStatusChecker 也重载了 noPage 方法，就是由 HTTPClientFactory 调用的成功的响应码。如果响应码是 304(未改变状态码)，则 noPage 方法调用 HTTPClientFactory.page 替换原来的 noPage 方法，表示成功的响应。如果执行成功了，HTTPStatusChecker 的 noPage 同样返回重载的 HTTPClientFactory 中的 noPage 方法。在这个时候，它提供了 304 响应来替换错误。

checkStatus 函数需要一个被 twisted.web.client._parse 转换过的 URL 参数。它看起来像是 URL 的一部分，可以从中获取主机地址和使用 HTTP/HTTPS 协议(TCP/SSL)。下一步，checkStatus 创建一个 HTTPStatusChecker 工厂对象，并打开连接。所有这些代码对 twisted.web.client.getPage 都是很简单的，并推荐使用修改过的 HTTPStatusChecker 工厂来代替 HTTPClientFactory。

当 updatecheck.py 运行时，会调用 checkStatus，设置 handleFirstResult 作为回调事件处理器，按照循序，第二次请求会使用 If-None-Match 或 If-Modified-Since 条件头字段，设置 handleSecondResult 作为回调函数处理器。handleSecondResult 函数报告服务器是否返回 304 错误，然后停止反应器。

handleFirstResult 实际上返回的是 handleSecondResult 的 deferred 结果。这允许 printError，就是被指定给 checkStatus 的错误事件处理器，用于所有在第二次请求中调用 checkStatus 的其他错误。

3.5 监视下载进度

以上的例子暂时还没有办法监视下载进度。当然，Deferred 可以在下载完成时返回结果，但有时你需要监视下载进度。

Twisted 网络编程必备（九）

3.5.1 下面如何做?

再次，twisted.web.client 没能提供足够的实用函数来控制进度。所以定义一个 client.HTTPDownloader 的子类，这个工厂类常用于下载网页到文件。通过重载一对方法，你可以保持对下载进度的跟踪。webdownload.py 脚本是 3-6 例子，展示了如何使用：

```
from twisted.web import client

class HTTPProgressDownloader(client.HTTPDownloader):

    def gotHeaders(self,headers):
```

```
            if self.status=='200':
                if headers.has_key('content-length'):
                    self.totalLength=int(headers['content-length'][0])
                else:
                    self.totalLength=0
                self.currentLength=0.0
                print ''
            return client.HTTPDownloader.gotHeaders(self,headers)
        def pagePart(self,data):
            if self.status=='200':
                self.currentLength+=len(data)
                if self.totalLength:
                    percent="%i%%"%(
                        (self.currentLength/self.totalLength)*100)
                else:
                    percent="%dK"%(self.currentLength/1000)
                    print "\033[1FProgress: "+percent
            return client.HTTPDownloader.pagePart(self,data)
    def downloadWithProgress(url,file,contextFactory=None,*args,**kwargs):
        scheme,host,port,path=client._parse(url)
        factory=HTTPProgressDownloader(url,file,*args,**kwargs)
        if scheme=='https':
```

```
            from twisted.internet import ssl
            if contextFactory is None:
                contextFactory=ssl.ClientContextFactory()
            reactor.connectSSL(host,port,factory,contextFactory)
        else:
            reactor.connectTCP(host,port,factory)
        return factory.deferred
if __name__=='__main__':
        import sys
        from twisted.internet import reactor
        def downloadComplete(result):
            print "Download Complete."
            reactor.stop()
        def downloadError(failure):
            print "Error: ",failure.getErrorMessage()
            reactor.stop()
        url,outputFile=sys.argv[1:]
        downloadWithProgress(url,outputFile).addCallback(
            downloadComplete).addErrback(
            downloadError)
        reactor.run()
```

运行 webdownload.py 脚本并加上 URL 和存储文件名这两个参数。作为命令工作会打印出下载进度的更新。

$ python webdownload.py http://www.oreilly.com/ oreilly.html

Progress: 100% <- 下载过程中的更新

Download Complete.

如果 WEB 服务器并不返回 Content-Length 头字段，则无法计算下载进度。在这种情况下，webdownload.py 打印已经下载的 KB 数量。

$ python webdownload.py http://www.slashdot.org/ slashdot.html

Progress: 60K <- 下载过程中更新

Download Complete.

3.5.2 它们如何工作?

HTTPProgressDownloader 是 client.HTTPDownloader 的子类。重载了 gotHeaders 方法并检查 Content-Length 头字段用于计算下载总量。它还同时提供重载了 pagePart 方法，在每次接收到一块数据时发生，用来保持对下载量的跟踪。

每次有数据到达时，HTTPProgressDownloader 打印进度报告。字符串\033[1F 是一个终端控制序列，可以确保每次替换掉当前行。这个效果看起来可以在原位置更新。

downloadWithProgress 函数包含了例子 3-5 中相同的代码用来转换 URL，创建 HTTPProgressDownloader 工厂对象，并初始化连接。downloadComplete 和 downloadError 是用于打印消息和停止反应器的简单回调函数。

4.0 WEB 服务器

即使是很保守的说，现在的很多软件是基于 WEB 开发的。人们将大量时间花费在 WEB 浏览器上面，包括阅读 HTML 页面、电子邮件、管理日志、进入数据库的记录、更新 Wiki 页面和写 weblog。

即使你不打算写严格的 WEB 应用，WEB 界面也更加容易提供适合于跨平台的 UI。在你的应用中包含轻量级的 WEB 服务器将会提供更多的附属功能。这一章将会展示如何使用 Twisted 开发一个 WEB 服务器，并介绍你一些构建 WEB 应用的方法。当然还提供了 HTTP

代理服务器。

这一章提供了一些 HTTP 协议的常识。这些一些构建 WEB 服务器所必须的知识。实际上，本书所涉及的 HTTP 知识还远远不够，还需要如《HTTP: The Definitive Guide》等等的书，还有不可替代的 HTTP 的 RFC 文档 RFC2616(http://www.faqs.org/rfcs/rfc2616.html)。

Twisted 网络编程必备（十）

4.1 响应 HTTP 请求

HTTP 是一个简单的协议接口。客户端发送请求，服务器端发送响应，然后关闭连接。你可以自己实现一个 HTTP 的 Protocol 来练习接收连接，读取请求，并发送 HTTP 格式的响应。

4.1.1 下面如何做?

每个 HTTP 请求开始于单行的 HTTP 方法，紧接着是 URL，然后是 HTTP 版本。随后是一些行的头字段。一个空行标志着头字段的结束。头字段后面就是请求主体，例如提交的 HTML 表单数据。

如下是一个 HTTP 请求的例子。这个请求询问服务器通过 GET 方法获取 www.example.com/index.html 资源，并使用 HTTP 版本 1.1。

```
GET /index.html HTTP/1.1

Host: www.example.com
```

服务器响应的第一行告知客户端响应的 HTTP 版本和状态码。有如请求一样，响应包含了头字段，并用空行隔开消息主体。如下是 HTTP 响应的例子：

```
HTTP/1.1 200 OK

Content-Type: text/plain

Content-Length: 17

Connection: Close
```

Hello HTTP world!

设置简单的 HTTP 服务器，需要写一个 Protocol 允许客户端连接。查找空行标志着头字段的结束。然后发送 HTTP 响应，例子 4-1 展示了简单的 HTTP 实现。

```
from twisted.protocols import basic

from twisted.internet import protocol,reactor

class HttpEchoProtocol(basic,LineReceiver):

    def __init__(self):

        self.lines=[]

        self.gotRequest=False

    def lineReceived(self,line):

        self.lines.append(line)

        if not line and not self.gotRequest:

            self.sendResponse()

            self.gotRequest=True

    def sendResponse(self):

        responseBody="You said: \r\n\r\n"+"\r\n".join(self.lines)

        self.sendLine("HTTP/1.0 200 OK")

        self.sendLine("Content-Type: text/plain")

        self.sendLine("Content-Length: %i"%len(responseBody))

        self.sendLine("")

        self.transport.write(responseBody)

        self.transport.loseConnection()
```

```
f=protocol.ServerFactory()

f.protocol=HttpEchoProtocol

reactor.listenTCP(8000,f)

reactor.run()
```

运行 webecho.py 脚本启动服务器。你可以看到服务器运行在 http://localhost:8000。你可以获得请求的回显，即原请求的报文。

4.1.2 它们如何工作?

HTTPEchoProtocol 懂得如何响应每一个请求。从客户端收到的数据将会存储在 self.lines 中。当看到一个空行时，可以知道头字段结束了。它发送回一个 HTTP 响应。第一行包含了 HTTP 版本和状态码，在这种情况下，200 是成功("OK"是附加的便于阅读的状态码描述)。下一对行是 Content-Type 和 Content-Length 头字段，用于告知客户端内容格式和长度。HTTPEchoProtocol 发送一个空行来结束头字段，然后发送响应主体，回显客户端请求报文。

4.2 解析 HTTP 请求

HTTPEchoProtocol 类提供了有趣的 HTTP 入门，但是距离使用功能还很遥远。它并不去分析请求头和资源位置，HTTP 方法也只支持一种。如果想要建立一个真正的 WEB 服务器，你可以用一种更好的方式来解析和响应请求。下面的实现展示这些。

4.2.1 下面如何做?

写 twisted.web.http.Request 的子类并重载 process 方法来处理当前请求。Request 对象已经包含了 process 需要调用的 HTTP 请求所有信息，所以只需要决定如何响应。例子 4-2 展示了如何运行基于 http.Request 的 HTTP 服务器。

```
from twisted.web import http
```

```
class MyRequestHandler(http.Request):
    pages={
        '/':'<h1>Home</h1>Home Page',
        '/test':'<h1>Test</h1>Test Page',
        }
    def process(self):
        if self.pages.has_key(self.path):
            self.write(self.pages[self.path])
        else:
            self.setResponseCode(http.NOT_FOUND)
            self.write("<h1>Not Found</h1>Sorry, no such page.")
        self.finish()
class MyHttp(http.HTTPChannel):
    requestFactory=MyRequestHandler
class MyHttpFactory(http.HTTPFactory):
    protocol=MyHttp
if __name__=='__main__':
    from twisted.internet import reactor
    reactor.listenTCP(8000,MyHttpFactory())
    reactor.run()
```

运行 requesthandler.py 将会在 8000 端口启动一个 WEB 服务器。可以同时阅览主页(http://localhost:8000/)和测试页/test(http://localhost:8000/test)。如果你指定了想要访问其他页面，将会得到错误信息。

4.2.2 它们如何工作?

http.Request 类解析了进入的 HTTP 请求并提供了制造响应的接口。在例子 4-2 中，MyRequestHandler 是 http.Request 的一个子类提供了 process 方法。process 方法将会被请求调用并接收。有责任在产生一个响应之后调用 self.finish() 来指出响应完成了。MyRequestHandler 使用 path 属性来寻找请求路径。并在 pages 字典中指定匹配路径。如果匹配成功，MyRequestHandler 使用 write 方法发送响应文本到响应。

注意 write 仅在写入响应的部分实体主体时才使用，而不是在生成原始 HTTP 响应时。setResponseCode 方法可以用于改变 HTTP 状态码。twisted.web.http 模块提供了所有 HTTP 状态码的定义，所以可用 http.NOT_FOUND 来代替 404 错误。

Tip:Request.setResponseCode 带有一个可选的第二个参数，一个易读的状态消息。你可以感觉到很方便于 twisted.web.http 模块包含了内置列表描述普通的状态码，也就是缺省使用的。

Request 类也提供 setHeader 方法来添加响应头字段。MyRequestHandler 使用 setHeader 来设置 Content-Type 头为 text/html，这个设置告诉浏览器对响应主体使用 HTML 格式。

twisted.web.http 模块提供了两种附加类允许将 Request 的子类转换成功能性的 WEB 服务器。HTTPChannel 类是一个 Protocol，可以创建 Request 对象来应付每个连接。创建 HTTPChannel 时使用你的 Request 子类，重载 requestFactory 类属性。HTTPFactory 是一个 ServerFactory，添加附加特性，包含日志方法加上 Request 对象，这种日志格式包括 Apache 和其他多种日志格式。

4.3 处理 POST 数据和 HTML 表单

前面的实验展示了通过客户端请求生成静态 HTML。这个实验展示了允许书写代码控制响应的生成，并且处理 HTML 表单提交的数据。

4.3.1 下面如何做?

写一个函数来处理 Request 对象并产生响应。设置字典来映射每一个可用路径到 WEB 站点来让函数出路路径请求。使用 Request.args 字典存取提交的 HTML 表单数据。例子 4-3 展示

了生成单页 HTML 表单的 WEB 服务器，另一个页面是通过表单数据显示的。

```
from twisted.web import http

def renderHomePage(request):

    colors='red','blue','green'

    flavors='vanilla','chocolate','strawberry','coffee'

    request.write("""

<html>

<head>

    <title>Form Test</title>

</head>

<body>

    <form action="posthandler" method="POST">

        Your Name:

        <p>

            <input type="text" name="name">

        </p>

        What's your favorite color?

        <p>

""")

    for color in colors:

        request.write(

            "<input type='radio' name='color' value='%s'>%s<br/>"%(
```

```
                color,color.capitalize()))
        request.write("""
            </p>
            What kinds of ice cream do you like?
            <p>
            """)
        for flavor in flavors:
            request.write(
                "<input type='checkbox' name='flavor' value='%s'>%s<br/>"%(
                flavor,flavor.capitalize()))
        request.write("""
            </p>
            <input type='submit'/>
        </form>
</body>
</html>
""")
        request.finish()
def handlePost(request):
        request.write("""
        <html><head><title>Posted Form Datagg</title>
            </head>
```

```
            <body>
            <h1>Form Data</h1>
        """)
        for key,values in request.args.items():
            request.write("<h2>%s</h2>"%key)
            request.write("<ul>")
            for value in values:
                request.write("<li>%s</li>"%value)
            request.write("</ul>")
        request.write("""
            </body></html>
        """)
        request.finish()
class FunctionHandleRequest(http.Request):
    pageHandlers={
        '/':renderHomePage,
        '/posthandler':handlePost,
    }
    def process(self):
        self.setHeader("Content-Type","text/html")
        if self.pageHandlers.has_key(self.path):
            handler=self.pageHandlers[self.path]
```

```
            handler(self)
        else:
            self.setResponseCode(http.NOT_FOUND)
            self.write("<h1>Not Found</h1>Sorry, no such page.")
            self.finish()
class MyHttp(http.HTTPChannel):
    requestFactory=FunctionHandledRequest
class MyHttpFactory(http.HTTPFactory):
    protocol=MyHttp
if __name__=='__main__':
    from twisted.internet import reactor
    reactor.listenTCP(8000,MyHttpFactory())
    reactor.run()
```

运行 formhandler.py 脚本。将会在 8000 端口运行 WEB 服务器。进入 http://localhost:8000 可以找到表单主页。按照如下填写一些字段信息。

然后点击提交按钮，你的浏览器将会发送表单数据到页面 formhandler 使用 HTTP 的 POST 请求。当它接受到表单数据时，formhandler 回展示提交过的字段和值。

4.3.2 它们是如何工作的?

例子 4-3 定义了两个函数来处理请求，renderHomePage 和 handlePost。FunctionHandleRequest 是 Request 的子类，其属性 pageHandler 定义了路径映射功能。process 方法查找路径，并尝试在 pageHandlers 中匹配路径。如果匹配成功，则 FunctionHandleRequest 传递自身到匹配函数，并且由对方负责处理；如果匹配失败，则返回 404 Not Found 响应。

renderHomePage 函数设置处理器到/，站点的根路径。它生成的 HTML 表单将会提交数据到页面/formhandler。这个处理器函数/formhandler 是 handlePost，将会响应页面列表并提交数据。handlePost 遍历值 Request.args，这个字典属性包含了请求提交的所有数据。

Tip:在这种情况下，发送的表单数据在 HTTP POST 请求的主体中。当请求发送 HTTP GET 时，Request.args 将会包含所有提交的 URI 查询字段值。你可以修改这个行为，通过改变表单生成器 renderHomePage 的 method 属性，从 POST 到 GET，重启服务器，就可以重新提交表单。

一个 HTML 表单可以有多个字段具有相同的名字。例如，表单 4-3 中允许选中多个复选框，所有的名字都是 flavor。不像很多其他的框架，http.Request 并不对你隐藏什么：代替了映射字段名到字符串，Request.args 映射每个字段值到列表。如果你知道要取哪一个值，那么可以只获取列表的第一个值。

4.4 管理资源等级

WEB 应用中的路径通常使用分级目录管理。例如如下 URL：

http://example.com/people

http://example.com/people/charles

http://example.com/people/charles/contact

这里可以很清楚的看出等级划分。页面/people/charles 是/people 的子页面，而页面/people/charles/contact 是/people/charles 的子页面。等级中的每个页面都有特定的意义：/people/charles 是一个人，而/people/charles/contact 是一项数据，charles 的数据。

WEB 服务器的缺省行为是将 PATH 等级映射到磁盘上的文件。客户端的每次请求都是对应特定的资源，服务器查找文件并定位磁盘路径，然后将文件内容或者可执行文件的执行结果作为响应。在 WEB 应用中可以人为的生成一个对应路径文件的内容。例如，你的数据并没有存储在磁盘上，而是在一个关系数据库或者另一个服务器上。或者你想要在请求时自动创建资源。类似这种情况，最好的办法是为等级浏览创建自己的逻辑。

编写自己的资源管理逻辑可以帮助你管理安全。而不是打开 WEB 服务器的整个目录，你可以有选择的控制哪些文件是可以访问的。

4.4.1 下面如何做?

twisted.web.resource 和 twisted.web.static 还有 twisted.web.server 模块提供了比 twisted.web.http.Resource 更高层次的请求管理类，你可以使用这些来设置一个 WEB 服务器来处理多种逻辑等级的资源。例子 4-4 使用了这些类来建立十六进制颜色代码。请求资源/color/hex，hex 是十六进制的颜色代码，你可以得到一个背景为#hex 的页面。对应每一种可能出现的颜色可能，服务器动态创建资源。

```
from twisted.web import resource,static,server

class ColorPage(resource.Resource):

    def __init__(self,color):

        self.color=color

    def render(self,request):

        return """

        <html>

        <head>

            <title>Color: %s</title>

            <link type='text/css' href='/styles.css' rel='Stylesheet' />

        </head>

        <body style='background-color: #%s'>

            <h1>This is #%s.</h1>

            <p style='background-color: white'>

            <a href='/color/'>Back</a>

            </p>

        </body>

        </html>

        """ % (self.color, self.color, self.color)
```

```
class ColorRoot(resource.Resource):
    def __init__(self):
        resource.Resource.__init__(self)
        self.requestedColors=[]
        self.putChild('',ColorIndexPage(self.requestColors))
    def render(self,request):
        # redirect /color -> /color/
        request.redirect(request.path+'/')
        return 'please use /colors/ instead.'
    def getChild(self,path,request):
        if path not in self.requestedColors:
            self.requestedColors.append(path)
        return ColorPage(path)
class ColorIndexPage(resource.Resource):
    def __init__(self,requestColorsList):
        resource.Resource.__init__(self)
        self.requestedColors=requestedColorsList
    def render(self,request):
        request.write("""
        <html>
        <head>
            <title>Colors</title>
```

```
            <link type='text/css' href='/styles.css' rel='Stylesheet' />
        </head>
        <body>
        <h1>Colors</h1>
        To see a color, enter a url like
        <a href='/color/ff0000'>/color/ff0000</a>. <br />
        Colors viewed so far:
        <ul>""")
        for color in self.requestedColors:
            request.write(
                "<li><a href="/blog/%s" style='color: #%s'>%s</a></li>" % (
                color, color, color))
        request.write("""
        </ul>
        </body>
        </html>
        """)
        return ""
class HomePage(resource.Resource):
    def render(self,request):
        return """
        <html>
```

```
            <head>
                <title>Colors</title>
                <link type='text/css' href='/styles.css' rel='Stylesheet' />
            </head>
            <body>
            <h1>Colors Demo</h1>
            What's here:
            <ul>
                <li><a href='/color'>Color viewer</a></li>
            </ul>
            </body>
            </html>
            """
if __name__=='__main__':
    from twisted.internet import reactor
    root=resource.Resource()
    root.putChild('',HomePage())
    root.putChild('color',ColorRoot())
    root.putChild('styles.css',static.File('styles.css'))
    site=server.Site(root)
    reactor.listenTCP(8000,site)
    reactor.run()
```

例子 4-4 引用了静态文件。所以需要在 resourcetree.py 脚本目录下创建一个 styles.css 文件。内容如下：

```
body {
    font-family: Georgia, Times, serif;
    font-size: 11pt;
}
h1 {
    margin: 10px 0;
    padding: 5px;
    background-color: black;
    color: white;
}
a {
    font-family: monospace;
}
p {
    padding: 10px;
}
```

运行 resourcetree.py 脚本，将会在 8000 端口启动一个 WEB 服务器。下面是服务器全部可用路径：

```
/              主页
/css           虚拟的 CSS 资源
/css/styles.css      静态文件 styles.css
```

/colors/ 颜色查看页面

/colors/hexcolor 按照背景色为#hexcolor 的页面

尝试通过 http://localhost:8000/colors/00abef 来访问，将会得到背景色为#00abef 的页面，大约是亮蓝色。

可以随便试试其他颜色。同样可以进入 http://localhost:8000/，选择可选答案。

Twisted 网络编程必备（十三）

4.4.2 它们如何工作?

例子 4.4 从 twisted.web 包中引入了几个类：resource.Resource、static.File、server.Site。每个 resource.Resource 对象做两件事。首先，定义请求的资源如何处理。第二，定义请求子资源的 Resource 对象。

例如查看类 ColorRoot。在这个类的实例稍后被加入了/colors 这个等级资源。初始化时，ColorRoot 使用 putChild 方法插入 ColorIndexPage 这个资源座位"资源。这意味着所有对/colors/的请求都由 ColorIndexPage 对象来处理。

你可以把他们想象为等价的，但是/stuff 和/stuff/是不同的。浏览器在解释相对路径时，对是否加上斜线的处理方法是不同的。在第一个例子中，对 "otherpage" 的请求会解释为 "http://example.com/otherpage"，在第二个例子中解释为"http://example.com/stuff/otherpage"。

如果你不清楚(explicit)服务器代码，这个问题可能会再次郁闷你。最好是预先设计好是否需要在 URI 末尾加上斜线，并重定向请求。Resource 类将会简化这些操作。如果设置了 addSlash 属性为 True，一个 Resource 会自动在找不到对应资源时自动在 URL 末尾添加斜线来再次查找。

4.4 管理资源等级

4.6 运行 HTTP 代理服务器

除了 HTTP 服务器和客户端以外，twisted.web 还包含了 HTTP 代理服务器的支持。一个代理服务器是一个服务器和一个客户端。他接受来自客户端的请求(作为服务器)并将他们转发到

服务器(作为客户端)。然后将响应发送回客户端。HTTP 代理服务器可以提供很多有用的服务：缓存、过滤和使用情况报告。下面的例子展示了如何使用 Twisted 构建一个 HTTP 代理服务器。

4.6.1 下面如何做?

twisted.web 包包含了 twisted.web.proxy，这个模块包含了 HTTP 代理服务器。例子 4-7 构建了一个简单的代理服务器。

```
from twisted.web import proxy,http

from twisted.internet import reactor

from twisted.python import log

import sys

log.startLogging(sys.stdout)

class ProxyFactory(http.HTTPFactory):

    protocol=proxy.Proxy

reactor.listenTCP(8001,ProxyFactory())

reactor.run()
```

运行 simpleproxy.py 脚本将会在 8001 端口启动代理服务器。在浏览器中设置这个代理服务器可以作为代理进行测试。对 log.startLogging 的调用将会把 HTTP 日志信息记录在 stdout 中，并可以直接查看。

```
$ python simpleproxy.py

2005/06/13 00:22 EDT [-] Log opened.
```

2005/06/13 00:22 EDT [-] __main__.ProxyFactory starting on 8001

... ...

这虽然给出了一个代理服务器，但是实际上没什么用处。例子 4-8 提供了更多的功能，可以跟踪最常使用的网页。

```
import sgmllib.re
from twisted.web import proxy,http
import sys
from twisted.python import log
log.startLogging(sys.stdout)
WEB_PORT=8000
PROXY_PORT=8001
class WordParser(sgmllib.SGMLParser):
    def __init__(self):
        sgmllib.SGMLParser.__init__(self)
        self.chardata=[]
        self.inBody=False
    def start_body(self,attrs):
        self.inBody=True
    def end_body(self):
        self.inBody=False
```

```
    def handle_data(self,data):
        if self.inBody:
            self.chardata.append(data)
    def getWords(self):
        #解出单词
        wordFinder=re.compile(r'\w*')
        words=wordFinder.findall("".join(self.chardata))
        words=filter(lambda word: word.strip(), words)
        print "WORDS ARE", words
        return words
class WordCounter(object):
    ignoredWords="the a of in from to this that and or but is was be can could i you they we at".split()
    def __init__(self):
        self.words=()
    def addWords(self,words):
        for word in words:
            word=word.lower()
            if not word in self.ignoredWords:
                currentCount=self.words.get(word,0)
                self.words[word]=currentCount+1
class WordCountProxyClient(proxy.ProxyClient):
    def handleHeader(self,key,value):
```

```
            proxy.ProxyClient.handleHeader(self,key,value)
            if key.lower()=="content-type":
                if value.split(';')[0]=='text/html':
                    self.parser=WordParser()
        def handleResponsePart(self,data):
            proxy.ProxyClient.handleResponsePart(self,data)
            if hasattr(self,'parser'):
                self.parser.feed(data)
        def handleResponseEnd(self):
            proxy.ProxyClient.handleResponseEnd(self)
            if hasattr(self,'parser'):
                self.parser.close()
                self.father.wordCounter.addWords(self.parser.getWords())
                del(self.parser)
    class WordCountProxyClientFactory(proxy.ProxyClientFactory):
        def buildProtocol(self,addr):
            client=proxy.ProxyClientFactory.buildProtocol(self,addr)
            #升级 proxy.proxyClient 对象到 WordCountProxyClient
            client.__class__=WordCountProxyClient
            return client
    class WordCountProxyRequest(proxy.ProxyRequest):
        protocols={'http':WordCountProxyClientFactory)
```

```
        def __init__(self,wordCounter,*args):
            self.wordCounter=wordCounter
            proxy.ProxyRequest.__init__(self,*args)
    class WordCountProxy(proxy.Proxy):
        def __init__(self,wordCounter):
            self.wordCounter=wordCounter
            proxy.Proxy.__init__(self)
        def requestFactory(self,*args)
            return WordCountProxyRequest(self.wordCounter,*args)
    class WordCountProxyFactory(http.HTTPFactory):
        def __init__(self,wordCount):
            self.wordCounter=wordCounter
            http.HTTPFactory.__init__(self)
        def buildProtocol(self,addr):
            protocol=WordCountProxy(self.wordCounter)
            return protocol
    #使用 WEB 接口展示记录的接口
    class WebReportRequest(http.Request):
        def __init__(self,wordCounter,*args):
            self.wordCounter=wordCounter
            http.Request.__init__(self,*args)
        def process(self):
```

```
            self.setHeader("Content-Type",'text/html')
            words=self.wordCounter.words.items()
            words.sort(lambda(w1,c1),(w2,c2): cmp(c2,c1))
            for word,count in words:
                self.write("<li>%s %s</li>"%(word,count))
            self.finish()
class WebReportChannel(http.HTTPChannel):
    def __init__(self,wordCounter):
        self.wordCounter=wordCounter
        http.HTTPChannel.__init__(self)
    def requestFactory(self,*args):
        return WebReportRequest(self.wordCounter,*args)
class WebReportFactory(http.HTTPFactory):
    def __init__(self,wordCounter):
        self.wordCounter=wordCounter
        http.HTTPFactory.__init__(self)
    def buildProtocol(self,addr):
        return WebReportChannel(self.wordCounter)
if __name__=='__main__':
    from twisted.internet import reactor
    counter=WordCounter()
    prox=WordCountProxyFactory(counter)
```

```
reactor.listenTCP(PROXY_PORT,prox)

reactor.listenTCP(WEB_PORT,WebReportFactory(counter))

reactor.run()
```

运行 wordcountproxy.py 将浏览器的代理服务器设置到 8001 端口。浏览其他站点，或者访问 http://localhost:8000/，将可以看到访问过的站点的单词频率。

4.6.2 它是如何工作的?

在例子 4-8 中有很多个类，但大多数是连接用的。只有很少的几个做实际工作。最开始的两个类 WordParser 和 WordCounter，用于从 HTML 文档中解出单词符号并计算频率。第三个类 WordCountProxy 客户端包含了查找 HTML 文档并调用 WordParser 的任务。

因为代理服务器同时作为客户端和服务器，所以需要使用大量的类。有一个 ProxyClientFactory 和 ProxyClient，提供了 Factory/Protocol 对来支持客户端连接。为了响应客户端的连接，需要使用 ProxyRequest，它是 HTTPFactory 的子类，还有 Proxy，是 http.HTTPChannel 的子类。它们对建立一个普通的 HTTP 服务器是有必要的：HTTPFactory 使用 Proxy 作它的协议(Protocol)，而代理 Proxy HTTPChannel 使用 ProxyRequest 作为它的 RequestFactory。如下是客户端发送请求的事件处理流程：

·客户端建立到代理服务器的连接。这个连接被 HTTPFactory 所处理。

·HTTPFactory.buildProtocol 会创建一个 Proxy 对象用来对客户端发送和接收数据。

·当客户端发送请求时，Proxy 创建 ProxyRequest 来处理。

·ProxyRequest 查找客户端请求的服务器。并创建 ProxyClientFactory 并调用 reactor.connectTCP 来通过 factory 连接服务器。

·一旦 ProxyClientFactory 连接到服务器，就会创建 ProxyClient 这个 Protocol 对象来发送和接收数据。

·ProxyClient 发送原始请求。作为响应，它将客户端发来的请求发送给服务器。这是通过调用 self.father.transport.write 实现的。self.father 是一个 Proxy 对象，正在管理着客户端连接对象。

这是一大串类，但实际上是分工明确的将工作由一个类传递到另一个类。但这是很重要的。

为代理模块的每一个类提供一个子类，你可以完成每一步的控制。

整个例子 4-8 中最重要的一个技巧。ProxyClientFactory 类有一个 buildProtocol 方法，可以包装好，供 ProxyClient 作为 Protocol。它并不提供任何简单的方法来提供你自己的 ProxyClient 子类。解决办法是使用 Python 的__class__属性来替换升级 ProxyClient 对象来放回 ProxyClientFactory.buildProtocol，这些将 ProxyClient 改变为 WordCountProxyClient。

作为代理服务器的附加功能。例子 4-8 提供了标准的 WEB 服务器，在 8000 端口，可以显示从代理服务器来的当前单词统计数量。由此可见，在你的应用中包含一个内嵌的 HTTP 服务器是多么的方便，这种方式可以很好的提供给远程来显示相关状态信息。