Python Twisted介绍

原文链接：http://www.aosabook.org/en/twisted.html

作者：Jessica McKellar

Twisted是用Python实现的基于事件驱动的网络引擎框架。Twisted诞生于2000年初，在当时的网络游戏开发者看来，无论他们使用哪种语言，手中都鲜有可兼顾扩展性及跨平台的网络库。Twisted的作者试图在当时现有的环境下开发游戏，这一步走的非常艰难，他们迫切地需要一个可扩展性高、基于事件驱动、跨平台的网络开发框架，为此他们决定自己实现一个，并从那些之前的游戏和网络应用程序的开发者中学习，汲取他们的经验教训。

Twisted支持许多常见的传输及应用层协议，包括TCP、UDP、SSL/TLS、HTTP、IMAP、SSH、IRC以及FTP。就像Python一样，Twisted也具有“内置电池”（batteries-included）的特点。Twisted对于其支持的所有协议都带有客户端和服务器实现，同时附带有基于命令行的工具，使得配置和部署产品级的Twisted应用变得非常方便。

21.1 为什么需要Twisted

2000年时，Twisted的作者Glyph正在开发一个名为Twisted Reality的基于文本方式的多人在线游戏。这个游戏采用Java开发，里面尽是一堆线程——每个连接就有3个线程处理。处理输入的线程会在读操作上阻塞，处理输出的线程将在一些写操作上阻塞，还有一个“逻辑”线程将在等待定时器超时或者事件入队列时休眠。随着玩家们在虚拟世界中移动并交互时，线程出现死锁，缓存被污染，程序中的加锁逻辑几乎从来就没对过——采用多线程使得整个软件变得复杂、漏洞百出而且极难扩展。

为了寻求其他的解决方案，作者发现了Python，特别是Python中用于对流式对象比如socket和pipe进行多路I/O复用的select模块（UNIX规范第3版（SUSv3）描述了select）。那时，Java并没有提供操作系统的select接口或者任何其他的异步I/O API（针对非阻塞式I/O的包java.nio已经在J2SE 1.4中加入了，2002年发布）。通过用Python中的select模块快速搭建起游戏的原型，这迅速降低了程序的复杂度，并且比多线程版本要更加可靠。

Glyph迅速转向了Python、select以及基于事件驱动的编程。他使用Python的select模块为游戏编写了客户端和服务器。但他想要的还不止于此。从根本上说，他希望能将网络行为转变为对游戏中的对象的方法调用。如果你能在游戏中收取邮件会怎样，就像Nethack mailer这种守护进程一样？如果游戏中的每位玩家都拥有一个主页呢？Glyph发现他需要优秀的IMAP以及HTTP客户端和服务器的Python实现，而这些都要采用select。

他首先转向了Medusa，这是一个在90年代中期开发的平台，在这里可以采用Python中的asyncore模块来编写网络服务。asyncore是一个异步化处理socket的模块，在操作系统的select API之上构建了一个调度器和回调接口。

这对于Glyph来说是个激动人心的发现，但Medusa有两个缺点：

1. 这个项目到2001年就不再维护了，那正是glyph开发Twisted Reality的时候。
2. asyncore只是对socket的一个薄封装层，应用程序的编写者仍然需要直接操作socket。这意味着程序可移植性的担子仍然落在程序员自己身上。此外，那时asyncore对Windows的支持还有问题，Glyph希望能在Windows上运行一个带有图形用户界面的客户端。

Glyph需要自己实现一个网络引擎平台，而且他意识到Twisted Reality已经打开了问题的大门，这和他的游戏一样有趣。

随着时间的推移，Twisted Reality这个游戏就演化成了Twisted网络引擎平台。它可以做到当时Python中已有的网络平台所无法做到的事情：

- 使用基于事件驱动的编程模型，而不是多线程模型。

- 跨平台：为主流操作系统平台暴露出的事件通知系统提供统一的接口。
- “内置电池”的能力：提供流行的应用层协议实现，因此Twisted马上就可为开发人员所用。
- 符合RFC规范，已经通过健壮的测试套件证明了其一致性。
- 能很容易的配合多个网络协议一起使用。
- 可扩展。

21.2 Twisted架构概览

Twisted是一个事件驱动型的网络引擎。由于事件驱动编程模型在Twisted的设计哲学中占有重要的地位，因此这里有必要花点时间来回顾一下究竟事件驱动意味着什么。

事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

让我们用例子来比较和对比一下单线程、多线程以及事件驱动编程模型。图21.1展示了随着时间的推移，这三种模式下程序所做的工作。这个程序有3个任务需要完成，每个任务都在等待I/O操作时阻塞自身。阻塞在I/O操作上所花费的时间已经用灰色框标示出来了。

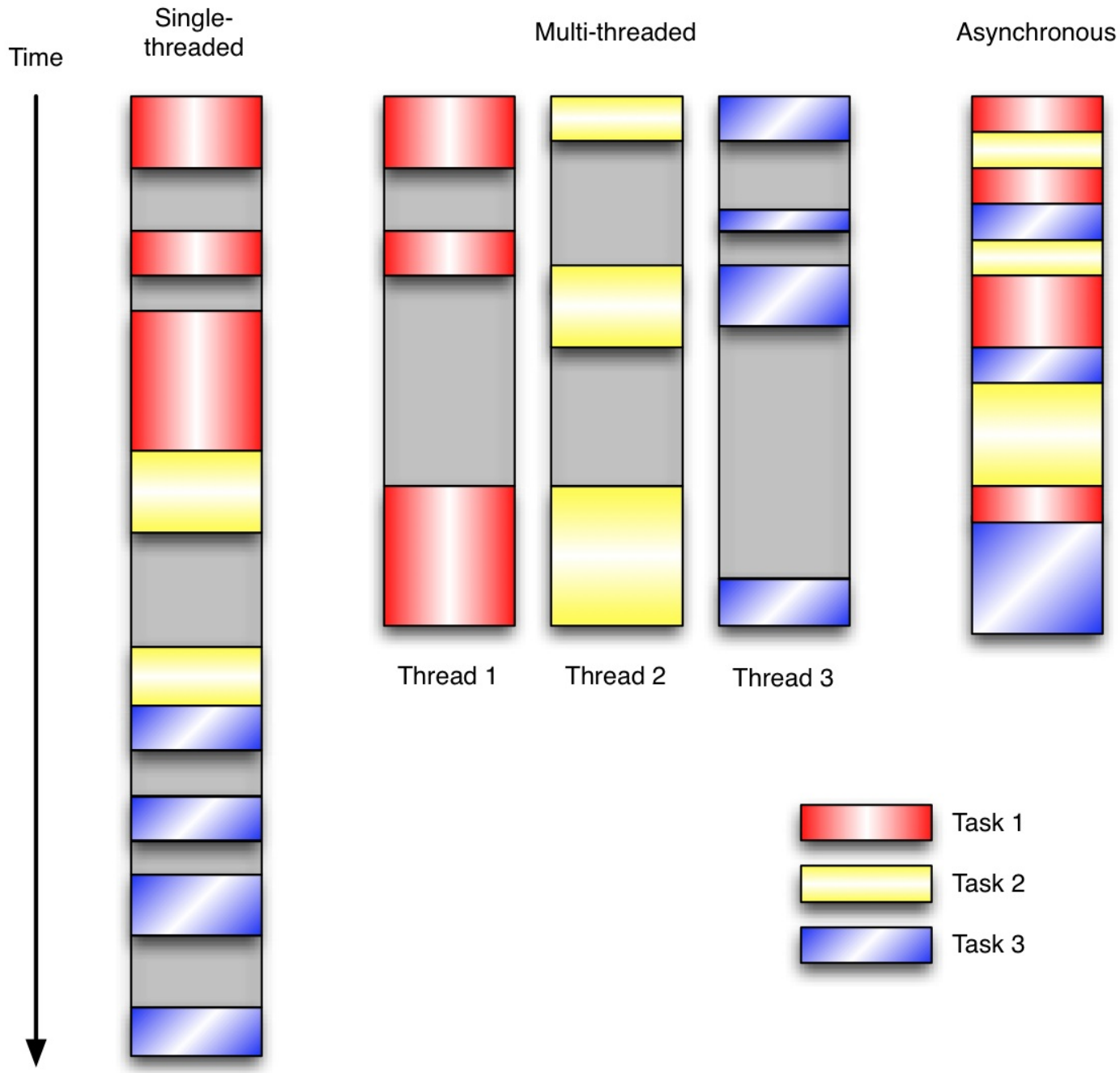

图21.1 线程模型

在单线程同步模型中，任务按照顺序执行。如果某个任务因为I/O而阻塞，其他所有的任务都必须等待，直到它完成之后它们才能依次执行。这种明确的执行顺序和串行化处理的行为是很容易推断得出的。如果任务之间并没有互相依赖的关系，但仍然需要互相等待的话这就使得程序不必要的降低了运行速度。

在多线程版本中，这3个任务分别在独立的线程中执行。这些线程由操作系统来管理，在多处理器系统上可以并行处理，或者在单处理器系统上交错执行。这使得当某个线程阻塞在某个资源的同时其他线程得以继续执行。与完成类似功能的同步程序相比，这种方式更有效率，但程序员必须写代码来保护共享资源，防止其被多个线程同时访问。多线程程序更加难以推断，因为这类程序不得不通过线程同步机制如锁、可重入函数、线程局部存储或者其他机制来处理线程安全问题，如果实现不当就会导致出现微妙且令人痛不欲生的bug。

在事件驱动版本的程序中，3个任务交错执行，但仍然在一个单独的线程控制中。当处理I/O或者其他昂贵的操作时，注册一个回调到事件循环中，然后当I/O操作完成时继续执行。回调描述了该如何处理某个事件。事件循环轮询所有

的事件，当事件到来时将它们分配给等待处理事件的回调函数。这种方式让程序尽可能的得以执行而不需要用到额外的线程。事件驱动型程序比多线程程序更容易推断出行为，因为程序员不需要关心线程安全问题。

当我们面对如下的环境时，事件驱动模型通常是一个好的选择：

1. 程序中有许多任务，而且…
2. 任务之间高度独立（因此它们不需要互相通信，或者等待彼此）而且…
3. 在等待事件到来时，某些任务会阻塞。

当应用程序需要在任务间共享可变的数据时，这也是一个不错的选择，因为这里不需要采用同步处理。

网络应用程序通常都有上述这些特点，这使得它们能够很好的契合事件驱动编程模型。

重用已有的应用

在Twisted创建之前就已经有了许多针对多种流行的网络协议的客户端和服务器实现了。为什么Glyph不直接用Apache、IRCd、BIND、OpenSSH或者任何其他已有的应用，而要为Twisted从头开始重新实现各个协议的客户端和服务器呢？

问题在于所有这些已有的实现都存在有从头写起的网络层代码，通常都是C代码。而应用层代码直接同网络层耦合在一起，这使得它们非常难以以库的形式来复用。当要一起使用这些组件时，如果希望在多个协议中暴露相同的数据，则它们必须以黑盒的形式来看待，这使得开发者根本没机会重用代码。此外，服务器和客户端的实现通常是分离的，彼此之间不共享代码。要扩展这些应用，维护跨平台的客户端-服务器兼容性的难度本不至于这么大。

Twisted中的客户端和服务器是用Python开发的，采用了一致性的接口。这使得开发新的客户端和服务器变得很容易实现，可以在客户端和服务器之间共享代码，在协议之间共享应用逻辑，以及对某个实现的代码做测试。

Reactor模式

Twisted实现了设计模式中的反应堆（reactor）模式，这种模式在单线程环境中调度多个事件源产生的事件到它们各自的事件处理例程中去。

Twisted的核心就是reactor事件循环。Reactor可以感知网络、文件系统以及定时器事件。它等待然后处理这些事件，从特定于平台的行为中抽象出来，并提供统一的接口，使得在网络协议栈的任何位置对事件做出响应都变得简单。

基本上reactor完成的任务就是：

```
while True:
    timeout =
time_until_next_timed_event()
    events = wait_for_events(timeout)
    events += timed_events_until(now())
    for event in events:
        event.process()
```

Twisted目前在所有平台上的默认reactor都是基于poll API的（UNIX规范第3版（SUSv3）中描述）。此外，Twisted还支持一些特定于平台的高容量多路复用API。这些reactor包括基于FreeBSD中kqueue机制的KQueue reactor，支持epoll接口的系统（目前是Linux 2.6）中的epoll reactor，以及基于Windows下的输入输出完成端口的IOCP reactor。

在实现轮询的相关细节中，Twisted需要考虑的包括：

- 网络和文件系统的限制

- 缓冲行为
- 如何检测连接丢失
- 出现错误时的返回值

Twisted的reactor实现同时也考虑了正确使用底层的非阻塞式API，并正确处理各种边界情况。由于Python中没有暴露出IOCP API，因此Twisted需要维护自己的实现。

管理回调链

回调是事件驱动编程模型中的基础，也是reactor通知应用程序事件已经处理完成的方式。随着程序规模不断扩大，基于事件驱动的程序需要同时处理事件处理成功和出错的情况，这使得程序变得越来越复杂。若没有注册一个合适的回调，程序就会阻塞，因为这个事件处理的过程绝不会发生。出现错误时需要通过应用程序的不同层次从网络栈向上传递回调链。

下面是两段Python伪码，分别是同步和异步模式下获取URL的玩具代码。让我们相互比较一下这两个版本，看看基于事件驱动的程序有什么缺陷：

以同步的方式获取URL：

```
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Shutting
down..."
    exit(0)

url = "http://google.com"
try:
    page = getPage(url)
    processPage(page)
except Error, e:
    logError(error)
finally:
    finishProcessing()
```

以异步的方式获取URL：

```
from twisted.internet import
reactor
import getPage

def processPage(page):
    print page
    finishProcessing()

def logError(error):
    print error
    finishProcessing()

def finishProcessing(value):
    print "Shutting down..."
    reactor.stop()

url = "http://google.com"
# getPage takes: url,
# success callback, error callback
getPage(url, processPage, logError)

reactor.run()
```

在异步版的URL获取器中，reactor.run()启动reactor事件循环。在同步和异步版程序中，我们假定getPage函数处理获取页面的工作。如果获取成功就调用processPage，如果尝试获取页面时出现了Exception（异常），logError就得到调用。无论哪种情况，最后都要调用finishProcessing。

异步版中的logError回调正对应于同步版中的try/except块。对processPage的回调对应于else块，无条件回调的finishProcessing就对应于finally块。

在同步版中，代码结构直接显示出有一个try/except块，logError和processPage这两者间只会取其一调用一次，而finishProcessing总是会被调用一次。在异步版中需要由程序员自己负责正确调用成功和失败情况下的回调链。如果由于编程错误，在processPage或者logError的回调链之后没有调用finishProcessing，reactor事件循环将永远不会停止，程序就会卡住。

这个玩具式的例子告诉我们在开发Twisted的头几年里这种复杂性令程序员感到非常沮丧。而Twisted应对这种复杂性的方式是新增一个称为Deferred（延迟）的对象。

Deferreds

Deferred对象以抽象化的方式表达了一种思想，即结果还尚不存在。它同样能够帮助管理产生这个结果所需要的回调链。当从函数中返回时，Deferred对象承诺在某个时刻函数将产生一个结果。返回的Deferred对象中包含所有注册到事件上的回调引用，因此在函数间只需要传递这一个对象即可，跟踪这个对象比单独管理所有的回调要简单的多。

Deferred对象包含一对回调链，一个是针对操作成功的回调，一个是针对操作失败的回调。初始状态下Deferred对象的两条链都为空。在事件处理的过程中，每个阶段都为其添加处理成功的回调和处理失败的回调。当一个异步结果到来时，Deferred对象就被“激活”，那么处理成功的回调和处理失败的回调就可以以合适的方式按照它们添加进来的顺序依次得到调用。

异步版URL获取器采用Deferred对象后的代码如下：

```
from twisted.internet import reactor
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Shutting down..."
    reactor.stop()

url = "http://google.com"
deferred = getPage(url) # getPage returns a
Deferred
deferred.addCallbacks(success, failure)
deferred.addBoth(stop)

reactor.run()
```

在这个版本中调用的事件处理函数与之前相同，但它们都注册到了一个单独的Deferred对象上，而不是分散在代码各处再以参数形式传递给getPage。

Deferred对象创建时包含两个添加回调的阶段。第一阶段，addCallbacks将 processPage和logError添加到它们各自归属的回调链中。然后addBoth再将finishProcessing同时添加到这两个回调链上。用图解的方式来看，回调链应该如图21.2所示：

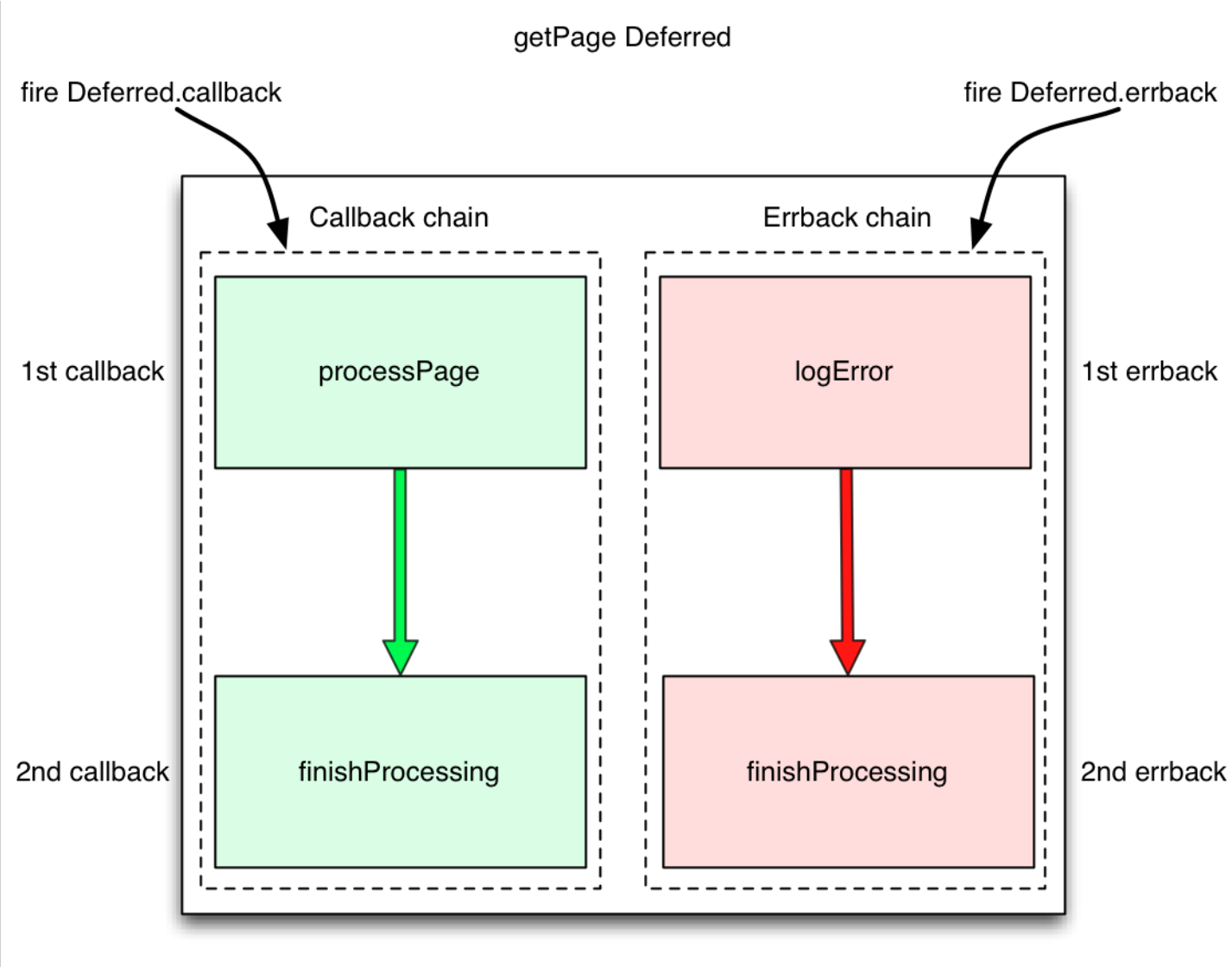

图21.2 回调链

Deferred对象只能被激活一次，如果试图重复激活将引发一个异常。这使得Deferred对象的语义相当接近于同步版中的try/except块。从而让异步事件的处理能更容易推断，避免由于针对单个事件的回调调用多了一个或少了一个而产生微妙的bug。

理解Deferred对象对于理解Twisted程序的执行流是非常重要的。然而当使用Twisted为我们提供的针对网络协议的高层抽象时，通常情况下我们完全不需要直接使用Deferred对象。

Deferred对象所包含的抽象概念是非常强大的，这种思想已经被许多其他的事件驱动平台所借用，包括jQuery、Dojo和Mochikit。

Transports

Transports代表网络中两个通信结点之间的连接。Transports负责描述连接的细节，比如连接是面向流式的还是面向数据报的，流控以及可靠性。TCP、UDP和Unix套接字可作为transports的例子。它们被设计为“满足最小功能单元，同时具有最大程度的可复用性”，而且从协议实现中分离出来，这让许多协议可以采用相同类型的传输。Transports实现了ITransports接口，它包含如下的方法：

write	以非阻塞的方式按顺序依次将数据写到物理连接上
writeSequence	将一个字符串列表写到物理连接上
loseConnection	将所有挂起的数据写入，然后关闭连接
getPeer	取得连接中对端的地址信息
getHost	取得连接中本端的地址信息

将transports从协议中分离出来也使得对这两个层次的测试变得更加简单。可以通过简单地写入一个字符串来模拟传输，用这种方式来检查。

Protocols

Protocols描述了如何以异步的方式处理网络中的事件。HTTP、DNS以及IMAP是应用层协议中的例子。Protocols实现了IProtocol接口，它包含如下的方法：

makeConnection	在transport对象和服务器之间建立一条连接
connectionMade	连接建立起来后调用
dataReceived	接收数据时调用
connectionLost	关闭连接时调用

我们最好以一个例子来说明reactor、protocols以及transports这三者之间的关系。以下是完整的echo服务器和客户端的实现，首先来看看服务器部分：

```
from twisted.internet import protocol, reactor

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        # As soon as any data is received, write it
back
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
```

接着是客户端部分：

```
from twisted.internet import reactor, protocol

class EchoClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("hello, world!")

def dataReceived(self, data):
    print "Server said:", data
        self.transport.loseConnection()

def connectionLost(self, reason):
    print "connection lost"

class EchoFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return EchoClient()

def clientConnectionFailed(self, connector,
reason):
    print "Connection failed - goodbye!"
        reactor.stop()

def clientConnectionLost(self, connector, reason):
    print "Connection lost - goodbye!"
        reactor.stop()

reactor.connectTCP("localhost", 8000, EchoFactory())
reactor.run()
```

运行服务器端脚本将启动一个TCP服务器，监听端口8000上的连接。服务器采用的是Echo协议，数据经TCP transport对象写出。运行客户端脚本将对服务器发起一个TCP连接，回显服务器端的回应然后终止连接并停止reactor事件循环。这里的Factory用来对连接的双方生成protocol对象实例。两端的通信是异步的，connectTCP负责注册回调函数到reactor事件循环中，当socket上有数据可读时通知回调处理。

Applications

Twisted是用来创建具有可扩展性、跨平台的网络服务器和客户端的引擎。在生产环境中，以标准化的方式简化部署这些应用的过程对于Twisted这种被广泛采用的平台来说是非常重要的一环。为此，Twisted开发了一套应用程序基础组件，采用可重用、可配置的方式来部署Twisted应用。这种方式使程序员避免堆砌千篇一律的代码来将应用程序同已有的工具整合在一起，这包括精灵化进程（daemonization）、日志处理、使用自定义的reactor循环、对代码做性能剖析等。

应用程序基础组件包含4个主要部分：服务（Service）、应用（Application）、配置管理（通过TAC文件和插件）以及twistd命令行程序。为了说明这个基础组件，我们将上一节的Echo服务器转变成一个应用。

Service

Service就是IService接口下实现的可以启动和停止的组件。Twisted自带有TCP、FTP、HTTP、SSH、DNS等服务以及其他协议的实现。其中许多Service都可以注册到单独的应用中。IService接口的核心是：

`startService` 启动服务。可能包含加载配置数据，设定数据库连接或者监听某个端口
`stopService` 关闭服务。可能包含将状态保存到磁盘，关闭数据库连接或者停止监听端口

我们的Echo服务使用TCP协议，因此我们可以使用Twisted中IService接口下默认的TCPServer实现。

Application

Application是处于最顶层的Service，代表了整个Twisted应用程序。Service需要将其自身同Application注册，然后就可以用下面我们将介绍的部署工具twistd搜索并运行应用程序。我们将创建一个可以同Echo Service注册的Echo应用。

TAC文件

当在一个普通的Python文件中管理Twisted应用程序时，需要由开发者负责编写启动和停止reactor事件循环以及配置应用程序的代码。在Twisted的基础组件中，协议的实现都是在一个模块中完成的，需要使用到这些协议的Service可以注册到一个Twisted应用程序配置文件中（TAC文件）去，这样reactor事件循环和程序配置就可以由外部组件来进行管理。

要将我们的Echo服务器转变成一个Echo应用，我们可以按照以下几个简单的步骤来完成：

1. 将Echo服务器的Protocol部分移到它们自己所归属的模块中去。
2. 在TAC文件中：
    1. 创建一个Echo应用。
    2. 创建一个TCPServer的Service实例，它将使用我们的EchoFactory，然后同前面创建的应用完成注册。

管理reactor事件循环的代码将由twistd来负责，我们下面会对此进行讨论。这样，应用程序的代码就变成这样了：

echo.py文件：

```
from twisted.internet import protocol,
reactor

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()
```

twistd

twistd（读作“twist-dee”）是一个跨平台的用来部署Twisted应用程序的工具。它执行TAC文件并负责处理启动和停止应用程序。作为Twisted在网络编程中具有“内置电池”能力的一部分，twistd自带有一些非常有用的配置标志，包括将应用程序转变为守护进程、定义日志文件的路径、设定特权级别、在chroot下运行、使用非默认的reactor，甚至是在profiler下运行应用程序。

我们可以像这样运行这个Echo服务应用：

```
$ twistd -y
echo_server.tac
```

在这个简单的例子里，twistd将这个应用程序作为守护进程来启动，日志记录在twistd.log文件中。启动和停止应用后，日志文件内容如下：

```
2011-11-19 22:23:07-0500 [-] Log opened.
2011-11-19 22:23:07-0500 [-] twistd 11.0.0 (/usr/bin/python 2.7.1) starting up.
2011-11-19 22:23:07-0500 [-] reactor class:
twisted.internet.selectreactor.SelectReactor.
2011-11-19 22:23:07-0500 [-] echo.EchoFactory starting on 8000
2011-11-19 22:23:07-0500 [-] Starting factory <echo.EchoFactory instance at
0x12d8670>
2011-11-19 22:23:20-0500 [-] Received SIGTERM, shutting down.
2011-11-19 22:23:20-0500 [-] (TCP Port 8000 Closed)
2011-11-19 22:23:20-0500 [-] Stopping factory <echo.EchoFactory instance at
0x12d8670>
2011-11-19 22:23:20-0500 [-] Main loop terminated.
2011-11-19 22:23:20-0500 [-] Server Shut Down.
```

通过使用Twisted框架中的基础组件来运行服务，这么做使得开发人员能够不用再编写类似守护进程和记录日志这样的冗余代码了。这同样也为部署应用程序建立了一个标准的命令行接口。

Plugins

对于运行Twisted应用程序的方法，除了基于TAC文件外还有一种可选的方法，这就是插件系统。TAC系统可以很方便的将Twisted预定义的服务同应用程序配置文件注册，而插件系统能够方便的将用户自定义的服务注册为twistd工具的子命令，然后扩展应用程序的命令行接口。

在使用插件系统时：

1. 由于只有plugin API需要保持稳定，这使得第三方开发者能很容易地扩展软件。
2. 插件发现能力已经集成到系统中了。插件可以在程序首次运行时加载并保存，每次程序启动时会重新触发插件发现过程，或者也可以在程序运行期间反复轮询新插件，这使得在程序已经启动后我们还可以判断是否有新的插件安装上了。

当使用Twisted插件系统来扩展软件时，我们要做的就是创建IPlugin接口下实现的对象并将它们放到一个特定的位置中，这里插件系统知道该如何去找到它们。

我们已经将Echo服务转换为一个Twisted应用程序了，而将其转换为一个Twisted插件也是非常简单直接的。在我们之前的Echo模块中，除了包含有Echo协议和EchoFactory的定义之外，现在我们还要添加一个名为twistd的目录，其中还包含着一个名为plugins的子目录，这里正是我们需要定义echo插件的地方。通过这个插件，我们可以启动一个echo服务，并将需要使用的端口号作为参数指定给twistd工具。

```
from zope.interface import implements

from twisted.python import usage
from twisted.plugin import IPlugin
from twisted.application.service import IServiceMaker
from twisted.application import internet

from echo import EchoFactory

class Options(usage.Options):
    optParameters = [["port", "p", 8000, "The port number to listen
on."]]

class EchoServiceMaker(object):
    implements(IServiceMaker, IPlugin)
    tapname = "echo"
    description = "A TCP-based echo server."
    options = Options

def makeService(self, options):
    """
    Construct a TCPServer from a factory defined in myproject.
    """
    return internet.TCPServer(int(options["port"]), EchoFactory())

serviceMaker = EchoServiceMaker()
```

现在，我们的Echo服务器将作为一个服务选项出现在twistd –help的输出中。运行twistd echo –port=1235将在端口1235上启动一个Echo服务器。

Twisted还带有一个可拔插的针对服务器端认证的模块twisted.cred，插件系统常见的用途就是为应用程序添加一个认证模式。我们可以使用twisted.cred中现成的AuthOptionMixin类来添加针对各种认证的命令行支持，或者是添加新的认证类型。比如，我们可以使用插件系统来添加基于本地Unix密码数据库或者是基于LDAP服务器的认证方式。

twistd工具中附带有许多Twisted所支持的协议插件，只用一条单独的命令就可以完成启动服务器的工作了。这里有一些通过twistd启动服务器的例子：

```
twistd web -port 8080 -path
.
```

这条命令将在8080端口启动一个HTTP服务器，在当前目录中负责处理静态和动态页面请求。

```
twistd dns -p 5553
-hosts-file=hosts
```

这条命令在端口5553上启动一个DNS服务器，解析指定的文件hosts中的域名，这个文件的内容格式同/etc/hosts一样。

```
sudo twistd conch -p
tcp:2222
```

这条命令在端口2222上启动一个SSH服务器。ssh的密钥必须独立设定。

```
twistd mail -E -H localhost -d
localhost=emails
```

这条命令启动一个ESMTP POP3服务器，为本地主机接收邮件并保存到指定的emails目录下。

我们可以方便的通过twistd来搭建一个用于测试客户端功能的服务器，但它同样是可装载的、产品级的服务器实现。

在部署应用程序的方式上，Twisted通过TAC文件、插件以及命令行工具twistd的部署方式已经获得了成功。但是有趣的是，对于大多数大型Twisted应用程序来说，部署它们仍然需要重写一些这类管理和监控组件；Twisted的架构并没有对系统管理员的需求呈现出太多的友好性。这也反映了一个事实，那就是对于系统管理员来说Twisted历来就没有太多架构可言，而这些系统管理员才是部署和维护应用程序的专家。在这方面，Twisted在未来架构设计的决策上需要更积极的征求这类专家级用户的反馈意见。

21.3 反思与教训

Twisted最近刚刚渡过了其10周年的诞辰。自项目成立以来，由于受2000年早期的网络游戏启发，目前的Twisted已经在很大程度上实现了作为一个可扩展、跨平台、事件驱动的网络引擎的目标。Twisted广泛使用于生产环境中，从Google、卢卡斯电影到Justin.TV以及Launchpad软件协作平台都有在使用。Twisted中的服务器端实现是多个开源软件的核心，包括BuildBot、BitTorrent以及TahoeLAFS。

Twisted从最初开发到现在，其架构已经经历了几次大的变动。Deferred对象作为一个关键部分被增加了进来。如前文所述，这是用来管理延后的结果以及相应的回调链。

还有一个重要的部分被移除掉了，在目前的实现中已经几乎看不到任何影子了，这就是Twisted应用持久化（Twisted Application Persistence）。

Twisted应用持久化

Twisted应用持久化（TAP）是指将应用程序的配置和状态保存在一个pickle中。要运行采用了这种方案的应用需要两个步骤：

1. 使用mktap工具创建一个代表该应用的pickle（该工具现已废弃不用）。
2. 使用twistd命令行工具进行unpickle操作，然后运行该应用。

这个过程是受Smalltalk images的启发，因为我们讨厌那种临时性的且难以使用的专用配置语言，不希望它们在项目中不断扩散。我们更希望在Python中表示配置的细节。

很快，TAP文件就引入了不必要的复杂性。修改Twisted中的类并不会使pickle中这些类的实例得到改变。在pickle对象上使用新版本的类方法或属性时可能会使整个应用崩溃。因此“升级版”的概念得以引入，即将pickle对象升级到新的API版本。但这就会出现升级版本的矩阵化现象，出现各种不同版本的pickle对象，因此单元测试时需要维护涵盖所有可能的升级路径。想全面地跟踪所有的接口变化依然很难，而且容易出错。

TAP以及相关的组件全部被废除了，最终从Twisted中完全剔除掉。取而代之的是TAC文件和插件系统。TAP这个缩写被重新定义为Twisted Application Plugin（Twisted应用插件），如今已经很难在Twisted中找到pickle系统的踪迹了。

我们从TAP的惨败中得到的教训是：如果可维护性要达到合理化的程度，则持久性数据就需要有一个明确的模式。更一般的是，我们学到了如何为项目增加复杂度：为了解决某个问题而需要引入一个新系统时，我们要正确理解这个方案的复杂性，并经过测试。新系统所带来的价值应该明显大于其复杂性。确保了这一点之后我们才能将方案付诸于项

目中。

web2：重构的教训

虽然这基本上不属于架构设计上的决策，但从项目管理的角度来看，重写Twisted的Web实现对于Twisted的外在形象以及维护者对代码库中其他部分做架构改善的能力却有着长远的影响，因此这里值得我们简单讨论一下。

在2000年中期，Twisted的开发者决定完全重写twisted.web API，在Twisted代码库中将其作为一个单独的项目实现，这就是web2。web2将包含许多针对原有twisted.web的改善和提升，包括完全支持HTTP1.1，以及对流式数据的API支持。

web2最初只是试验性的项目，但最终被大型项目所采用，甚至意外的得以在Debian系统上打包发布。twisted.web和web2的开发一直并行持续了多年，新用户常常被这两个并行的项目搞混，关于究竟应该使用哪种实现缺乏明确的提示，这使得新用户很沮丧。转换到web2的情况从未出现，终于在2011年开发者将其从代码库中移除，官方主页上再也看不到它了。web2中做出的一些改进也被慢慢地移植回twisted.web中。

Twisted获得了难以导航且结构混乱，容易使新开发者感到困惑的“恶名”，这个印象部分归功于web2。以至于数年之后，Twisted社区仍然在同这种不和谐的名声做斗争。

我们从web2中汲取的教训是：从头开始重构一个项目通常都是糟糕的主意。但如果必须这么做，请确保开发者社区能够懂得这么做的长远意义，而且在用户社群中要有明确的选择该使用哪种实现。

如果Twisted能够倒退回web2的时代，开发者们应该会对twisted.web做一系列向后兼容型的修改而不是去重构。

紧跟互联网的浪潮

我们使用互联网的方式还在持续演进中。把多种协议的实现作为软件核心的一部分，这个技术决策使得Twisted背负了维护这些协议的沉重负担。随着标准的改变以及对新协议的采纳，原有的实现必须跟着演进，同时需要严格的保证向后兼容性。

Twisted基本上是一个志愿者驱动型的项目，项目发展的限制因素不是技术社区的热情，而在于志愿者的时间。比如说，1999年的RFC 2616中定义了HTTP 1.1规范，而在Twisted的HTTP协议实现中增加对HTTP 1.1的支持却在2005年才开始，等到完成时已经是2009年了。1998年RFC 2460中定义了对IPv6的支持，而Twisted对其的支持还在进行中，但是直到2011年都未能合并进去。

随着所支持的操作系统的接口改变，实现也要跟着演进。比如，epoll事件通知机制是在2002年加入到Linux 2.5.44版中的，Twisted随之也发展出基于epoll的reactor事件循环来利用这个新的系统接口。2007年时，苹果公司发布的OS 10.5 Leopard系统中，系统调用poll的实现居然不支持外设，对于苹果公司来说这个问题足以让他们在系统自带的Python中屏蔽掉select.poll接口。Twisted不得不自行解决这个问题，并从那时起就对用户提供文档说明。

有时候，Twisted的开发并没有紧跟网络世界的变化，有一些改进被移到核心层之外的程序库中去了。比如Wokkel project，这是对Twisted的Jabber/XMPP支持的改进合集，已经作为“待合入”的独立项目有几年之久了，但还没有看到合入的希望。在2009年也曾经尝试过增加WebSocket到Twisted中，因为浏览器已经开始采纳对新协议的支持了。但开发计划最终却转到其他外部项目中去了，因为开发者们决定暂不包含新的协议，直到IETF把它从草案转变成标准以后再说。

所有这一切都在说明，库和附加组件的扩散有力的证明了Twisted的灵活性和可扩展性。通过采用严格的测试驱动开发策略以及文档化和编码规范标准，这样做能够帮助项目避免出现需要“回炉”的情况。在维护大量所支持的协议和平台的同时保持向后兼容性。Twisted是一个成熟、稳定的项目，并继续保持有非常活跃的开发状态。

Twisted期待着在下一个十年里成为你遨游互联网的引擎。