第一部分：Twisted 理论基础

前言：

最近有人在 Twisted 邮件列表中提出诸如”为任务紧急的人提供一份 Twisted 介绍”的的需求。值得提前透露的是，这个序列并不会如他们所愿.尤其是介绍 Twisted 框架和基于 Python 的异步编程而言，可能短时间无法讲清楚。因此,如果你时间紧急，这恐怕不是你想找的资料。

我相信如果对异步编程模型一无所知，快速的介绍同样无法让你对其有所理解，至少你得稍微懂点基础知识吧。我已经用 Twisted 框架几年了，因此思考过我当初是怎么学习它(学得很慢)并发现学习它的最大难度并不在 Twisted 本身,而在于对其模型的理解，只有理解了这个模型，你才能更好去写和理解异步程序的代码。大部分 Twisted 的代码写得很清晰，其在线文档也非常棒（至少在开源软件这个层次上可以这么说）。但如果不理解这个模型，不管是读 Twisted 源码还是使用 Twisted 的代码更或者是相关文档，你都会感到非常的伤脑筋。

因此，我会用前面几个部分来介绍这个模型以让你掌握它的机制，稍后会介绍一下 Twisted 的特点。实际上，一开始，我们并不会使用 Twisted，相反，会使用简单的 Python 来说明一个异步模型是如何工作的。我们在初次学习 Twisted 的时，会从你平常都不会直接使用的底层的实现讲起。Twisted 是一个高度抽象的体系，因此在使用它时，你会体会到其多层次性。但当你去学习尤其是尝试着理解它是如何工作时，这种为抽像而带来的多层次性会给你带来极大的理解难度。所以，我们准备来个从内到外，从低层开始学习它。

模型：

为 了更好的理解异步编程模型的特点，我们来回顾一下两个大家都熟悉的模型。在阐述过

程中，我们假设一个包含三个相互独立任务的程序。在此，除了规定这些任务 都要完成自己工作外，我们先不作具体的解释，后面我们会慢慢具体了解它们。请注意：在此我用“任务”这个词，这意味着它需要完成一些事情。

第一个模型是单线程的同步模型，如图 1 所示：

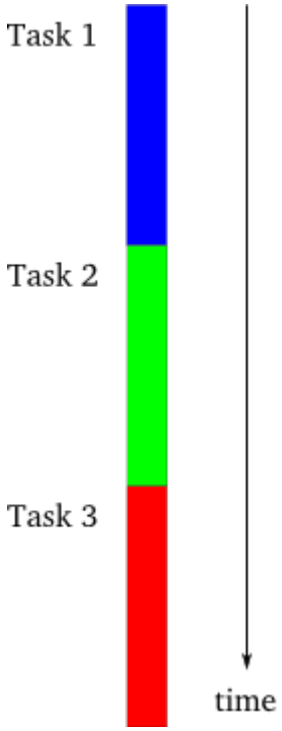


图 1 同步模型

这是最简单的编程方式。在一个时刻，只能有一个任务在执行，并且前一个任务结束后一个任务才能开始。如果任务都能按照事先规定好的顺序执行，最后一个任务的完成意味着前面所有的任务都已无任何差错地完成并输出其可用的结果—这是多么简单的逻辑。
下面我们来呈现第二个模型，如图 2 所示：

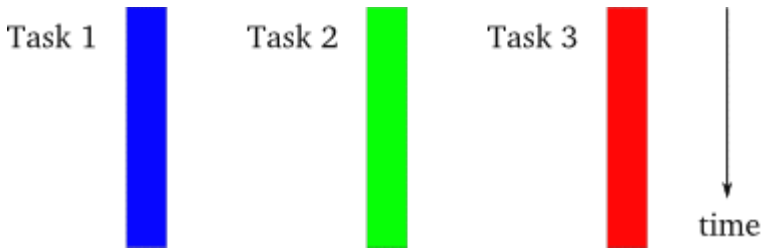


图 2 线程模型
在这个模型中，每个任务都在单独的线程中完成。这些线程都是由操作系统来管理，若在多

处理机、多核处理机的系统中可能会相互独立的运行，若在单处理机上，则会交错运行。 关键点在于，在线程模式中，具体哪个任务执行由操作系统来处理。但编程人员则只需简单地认为：它们的指令流是相互独立且可以并行执行。虽然，从图示看起来 很简单，实际上多线程编程是很麻烦的，你想啊，任务之间的要通信就要是线程之间的通信。线程间的通信那不是一般的复杂。什么邮箱、通道、共享内存、、、 唉：（
一些程序用多处理机而不是多线程来实现并行运算。虽然具体的编程细节是不同的，但对于我们要研究的模型来说是一样的。

下面我们来介绍一下异步编程模型，如图 3 所示

图 3 异步模型

在这个模型中，任务是交错完成，值得注意的是：这是在单线程的控制下。这要比多线程模型简单多了，因为编程人员总可以认为只有一个任务在执行，而其它的在停止状态。虽然在单处理机系统中，线程也是像图 3 那样交替进行。但作为程序员在使用多线程时，仍然需要使用图 2 而不是图 3 的来思考问题，以防止程序在挪到多处理机的系统上无法正常运行（考虑到兼容性）。间单线程的异步程序不管是在单处理机还是在多处理机上都 能很好的运行。

在异步编程模型与多线程模型之间还有一个不同：在多线程程序中，对于停止某个线程启动另外一个线程，其决定权并不在程序员手里而在操作系统那里，因此，程 序员在编写程序过程中必须要假设在任何时候一个线程都有可能被停止而启动另外一个线程。相反，在异步模型中，一个任务要想运行必须显式放弃当前运行的任务 的控制权。这也是相比多线程模

型来说，最简洁的地方。
值得注意的是：将异步编程模型与同步模型混合在同一个系统中是可以的。但在介绍中的绝大多数时候，我们只研究在单个线程中的异步编程模型。

动机:

我 们已经看到异步编程模型之所以比多线程模型简单在于其单令流与显式地放弃对任务的控制权而不是被操作系统随机地停止。但是异步模型要比同步模型复杂得多。 程序员必须将任务组织成序列来交替的小步完成。因此，若其中一个任务用到另外一个任务的输出，则依赖的任务（即接收输出的任务）需要被设计成为要接收系列 比特或分片而不是一下全部接收。由于没有实质上的并行，从我们的图中可以看出，一个异步程序会花费一个同步程序所需要的时间，可能会由于异步程序的性能问 题而花费更长的时间。

因此，就要问了，为什么还要使用异步模型呢？ 在这儿，我们至少有两个原因。首先，如果有一到两个任务需要完成面向人的接口，如果交替执行这些任务，系统在保持对用户响应的同时在后台执行其它的任务。因此，虽然后台的任务可能不会运行的更快，但这样的系统可能会欢迎的多。
然而，有一种情况下，异步模型的性能会高于同步模型，有时甚至会非常突出，即在比较短的时间内完成所有的任务。这种情况就是任务被强行等待或阻塞，如图 4 所示：

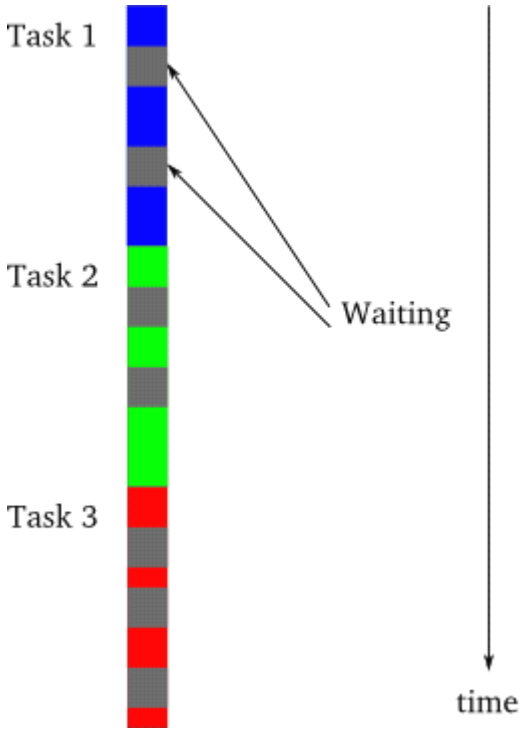


图 4 同步模型中出现阻塞
在图 4 中，灰色的部分代表这段时间某个任务被阻塞。为什么要阻塞一个任务呢？最直接的原因就是等待 I/O 的完成：传输数据或来自某个外部设备。一个典型的 CPU 处理数据的能力是硬盘或网络的几个数量级的倍数。因此，一个需要进行大 I/O 操作的同步程序需要花费大量的时间等待硬盘或网络将数据准备好。正是由于这个原因，同步程序也被称作为阻塞程

序。

从图 4 中可以看出，一个可阻塞的程序，看起来与图 3 描 述的异步程序有点像。这不是个巧合。异步程序背后的最主要的特点就在于，当出现一个任务像在同步程序一样出现阻塞时，会让其它可以执行的任务继续执行，而 不会像同步程序中那样全部阻塞掉。因此一个异步程序只有在没有任务可执行时才会出现“阻塞”，这也是为什么异步程序被称为非阻塞程序的原因。
任务之间的切换要不是此任务完成，要不就是它被阻塞。由于大量任务可能会被阻塞，异步程序等待的时间少于同步程序而将这些时间用于其它实时工作的处理（如与人打交道的接口），这样一来，前者的性能必然要高很多。

与同步模型相比，异步模型的优势在如下情况下会得到发挥：

1.有大量的任务，因此在一个时刻至少有一个任务要运行
2.任务执行大量的 I/O 操作，这样同步模型就会在因为任务阻塞而浪费大量的时间
3.任务之间相互独立，以至于任务内部的交互很少。

这些条件大多在 CS 模式中的网络比较繁忙服务器端出现（如 WEB 服务器）。每个任务代表一个客户端进行接收请求并回复的 I/O 操作。客户的请求（相当于读操作）都是相互独立的。因此一个网络服务是异步模型的典型代表，这也是为什么 twisted 是第一个也是最棒的网络库。

第二部分：异步编程初探与 **reactor** 模式

第二部分:低效的诗歌服务器来启发对 Twisted 机制的理解
这个系列是从这里开始的，欢迎你再次来到这里来。现在我们可能要写一些代码。在开始之前，我们都做出一些必要的假设。

关于对你的假设

在展开讨论前，我假设你已经有过用Python写同步程序的经历并且至少知道一点有关Python的 Sockt 编程的经验。如果你从没有写过 Socket 程序，或许你可以去看看 Socket 模块的文档，尤其是后面的示例代码。如果你没有用过 Python 的话，那后面的描述对你来说可能比看周易还痛苦。

你所使用的计算机的情况（想的真周到，：））

我一般是在 Linux 上使用 Twisted，这个系列的示例代码也是在 Linux 下完成的。首先声明的是我并没有故意让代码失去平台无关性，但我所讲述的一些内容确实可能仅仅适应于 Linux 和其它的类 Unix（比如 MAC OSX 或 FreeBSD）。WIndows 是个奇怪诡异的地方（？？为什么这么评价 Windows 呢），如果你想尝试在它上面学习这个系列，抱歉，如果出了问题，我无法提供任何帮助。

并且假设你已经安装了近期版本的 Python 和 Twisted。我所提供的示例示例代码是基于 Python2.5 和 Twisted8.2.0。

你可以在单机上运行所有的示例代码，也可以在网络系统上运行它们。但是为了学习异步编程的机制，单机上学习是比较理想的。

获取代码的方法

使用 git 工具来获取 Dave 的最新示例代码。在 shell 或其它命令行上输入以下命令（假设已经安装 git）：

git clone git://github.com/jdavisp3/twisted-intro.git

下载结束后，解压并进入第一层文件夹（你可以看到有个 README 文件）。

低效的诗歌服务器

虽然 CPU 的处理速度远远快于网络，但网络的处理速度仍然比人脑快，至少比人类的眼睛快。因此，想通过网络来获得 CPU 的视角是很困难的，尤其是在单机的回环模式中数据流全速传输时，更是困难重重。

我们所需要的是一个慢速低效诗歌服务器，其用人为的可变延时来体现影响结果。毕竟服务器要提供点东西吗，我们就提供诗歌好了。目录下面有个子目录专门存放诗歌用的。

最简单的慢速诗歌服务器在 blocking-server/slowpoetry.py 中实现。你可用下面的方式来运行它。

python blocking-server/slowpoetry.py poetry/ecstasy.txt

上面这个命令将启动一个阻塞的服务器，其提供“Ecstasy”这首诗。现在我们来看看它的源码内容，正如你所见，这里面并没有使用任何 Twisted 的内容，只是最基本的 Socket 编程操作。它每次只发送一定字节数量的内容，而每次中间延时一段时间。默认的是每隔 0.1 秒发送 10 个比特，你可以通过-delay 和
-num-bytes 参数来设置。例如每隔 5 秒发送 50 比特：

python blocking-server/slowpoetry.py --num-bytes 50 –delay 5poetry/ecstasy.txt

当服务器启动时，它会显示其所监听的端口号。默认情况下，端口号是在可用端口号池中随机选择的。你可能想使用固定的端口号，那么无需更改代码，只需要在启动命令中作下修改就 OK 了，如下所示：

python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt

如果你装有 netcat 工具，可以用如下命令来测试你的服务器（也可以用 telnet）：

netcat localhost 10000

如果你的服务器正常工作，那么你就可以看到诗歌在你的屏幕上慢慢的打印出来。对！你会注意到每次服务器都会发送过一行的内容过来。一旦诗歌传送完毕，服务器就会关闭这条连接。

默认情况下，服务器只会监听本地回环的端口。如果你想连接另外一台机子的服务器，你可以指定其 IP 地址内容，命令行参数是 -iface 选项。

不仅是服务器在发送诗歌的速度慢，而且读代码可以发现，服务器在服务一个客户端时其它连接进来的客户端只能处于等待状态而得不到服务。这的确是一个低效慢速的服务器，要不是为了学习，估计没有任何其它用处。

阻塞模式的客户端

在示例代码中有一个可以从多个服务器中顺序（一个接一个）地下载诗歌的阻塞模式的客户端。下面让这个客户端执行三个任务，正如第一个部分图 1 描述的那样。首先我们启动三个服务器，提供三首不同的诗歌。在命令行中运行下面三条命令：

python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt --num-bytes 30
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt

如果在你的系统中上面那些端口号有正在使用中，可以选择其它没有被使用的端口。注意，由于第一个服务器发送的诗歌是其它的三倍，这里我让第一个服务器使用每次发送 30 个字节而不是默认的 10 个字节，这样一来就以 3 倍于其它服务器的速度发送诗歌，因此它们会在几乎相同的时间内完成工作。

现在我们使用阻塞模式的客户端来获取诗歌，运行如下所示的命令：

python blocking-client/get-poetry.py 10000 10001 10002

如果你修改了上面服务口器的端口，你需要在这里时行相应的修改以保持一致。由于这个客户端采用的是阻塞模式，因此它会一首一首的下载，即只有在完成一首时才会开始下载另外一首。这个客户端会像下面这样打印出提示信息而不是将诗歌打印出来：

Task 1: get poetry from: 127.0.0.1:10000
Task 1: got 3003 bytes of poetry from 127.0.0.1:10000 in 0:00:10.126361
Task 2: get poetry from: 127.0.0.1:10001
Task 2: got 623 bytes of poetry from 127.0.0.1:10001 in 0:00:06.321777
Task 3: get poetry from: 127.0.0.1:10002
Task 3: got 653 bytes of poetry from 127.0.0.1:10002 in 0:00:06.617523
Got 3 poems in 0:00:23.065661

这图 1 最典型的文字版了，每个任务下载一首诗歌。你运行后可能显示的时间会与上面有所差别，并且也会随着你改变服务器的发送时间参数而改变。尝试着更改一下参数来观测一下效果。

异步模式的客户端

现在，我们来看看不用 Twisted 构建的异步模式的客户端。首先，我们先运行它试试。启动使用前面的三个端口来启动三个服务器。如果前面开启的还没有关闭，那就继续用它们好了。接下来，我们通过下面这段命令来启动我们的异步模式的客户端：
python async-client/get-poetry.py 10000 10001 10002

你或许会得到类似于下面的输出：

Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
...
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.133169
这次的输出可能会比较长，这是由于在异步模式的客户端中，每次接收到一段服务器发送来的数据都要打印一次提示信息，而服务器是将诗歌分成若干片段发送出去的。值得注意的是，这些任务相互交错执行，正如第一部分图 3 所示。
尝试着修改服务器的设置（如将一个服务器的延时设置的长一点），来观察一下异步模式的客户端是如何针对变慢的服务器自动调节自身的下载来与较快的服务器保持一致。这正是异步模式在起作用。
还需要值得注意的是，根据上面的设置，异步模式的客户端仅在 10 秒内完成工作，而同步模式的客户端却使用了 23 秒。现在回忆一下第一部分中图 3 与图 4.通过减少阻塞时间，我们的异步模式的客户端可以在更短的时间里完成下载。诚然，我们的异步客户端也有些阻塞发生，那是由于服务器太慢了。由于异步模式的客户端可以在不同的服务器来回切换，它比

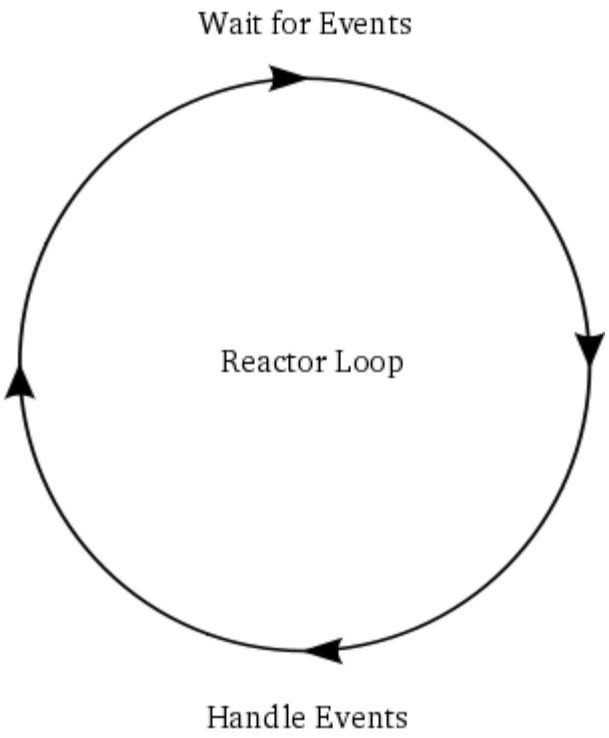
同步模式的客户产生的阻塞就少得多。

更近一步的观察

现在让我们来读一下异步模式客户端的代码。注意其与同步模式客户端的差别：

1.异步模式客户端一次性与全部服务器完成连接，而不像同步模式那样一次只连接一个。
2.用来进行通信的 Socket 方法是非阻塞模的，这是通过调用 setblocking(0 来实现的。
3.select 模块中的 select 方法是用来识别是其监视的 socket 是否有完成数据接收的，如果没有
它就处于阻塞状态。
4.当从服务器中读取数据时，会尽量多地从 Sockt 读取数据直到它阻塞为止，然后读下一个 Sockt
接收的数据（如果有数据接收的话）。这意味着我们需要跟踪记录从不同服务器传送过来诗歌的接
收情况（因为，一首诗的接收并不是连续完成，所以需要保证每个任务的可连续性，就得有冗余
的信息来完成这一工作）。
异步模式中客户端的核心就是最高层的循环体，即 get_poetry 函数。这个函数可以被拆分成两个步骤：
1.使用 select 函数等待所有 Socket，直到至少有一个 socket 有数据到来。
2.对每个有数据需要读取的 socket，从中读取数据。但仅仅只是读取有效数据，不能为了等待还没
来到的数据而发生阻塞。
3.重复前两步，直到所有的 socket 被关闭。

可以看出，同步模式客户端也有个循环体（在 main 函 数内），但是这个循环体的每个迭代都是完成一首诗的下载工作。而在异步模式客户端的每次迭代过程中，我们可以完成所有诗歌的下载或者是它们中的一些。我们 并不知道在一个迭代过程中，在下载那首诗，或者一次迭代中我们下载了多少数据。这些都依赖于服务器的发送速度与网络环境。我们只需要 select 函数告诉我们那个 socket 有数据需要接收，然后在保证不阻塞程序的前提下从其读取尽量多的数据。
如果在服务器端口固定的条件下，同步模式的客户端并不需要循环体，只需要顺序罗列三个 get_poetry
就可以了。但是我们的异步模式的客户端必须要有一个循环体来保证我们能够同时监视所有的 socket 端。这样我们就能在一次循环体中处理尽可能多的数据。
这个利用循环体来等待事件发生，然后处理发生的事件的模型非常常见，而被设计成为一个模式：reactor 模式。其图形化表示如图 5 所示：



这个循环就是个”reactor“（反应堆），因为它等待事件的发生然对其作为相应的反应。正因为如此，它也被称作事件循环。由于交互式系统都要进行 I/O 操作，因此这种循环也有时被称作 select loop,这是由于 select 调用被用来等待 I/O 操作。因此，在本程序中的 select 循环中，一个事件的发生意味着一个 socket 端处有数据来到。值得注意的是，select 并不是唯一的等待 I/O 操作的函数，它仅仅是一个比较古老的函数而已（因此才被用的如此广泛）。现在有一些新 API 可以完成 select 的工作而且性能更优，它们已经在不同的系统上实现了。不考虑性能上的因素，它们都完成同样的工作：监视一系列 sockets（文件描述符）并阻塞程序，直到至少有一个准备好时行 I/O 操作。

严格意义上来说，我们的异步模式客户端中的循环并不是 reactor 模式，因为这个循环体并没有独立于业务处理（在此是接收具体个服务器传送来的诗歌）之外。它们被混合在一起。一个真正 reactor 模式的实现是需要实现循环独立抽象出来并具有如下的功能：

1.监视一系列与你 I/O 操作相关的文件描述符（description)
2.不停地向你汇报那些准备好 I/O 操作的文件描述符

一个设计优秀的 reactor 模式实现需要做到：
1.处理所有不同系统会出现的 I/O 事件
2.提供优雅的抽象来帮助你在使用 reactor 时少花些心思去考虑它的存在
3.提供你可以在抽象层外（treactor 实现）使用的公共协议实现。

好了，我们上面所说的其实就是 Twisted—健壮、跨平台实现了 reactor 模式并含有很多附加功能。
在第三部分中，实现 Twisted 版的下载诗歌服务时，我们将开始写一些简单的 Twisted 程序。

第三部分：初步认识 Twisted

用 twisted 的方式实现前面的内容

最终我们将使用 twisted 的方式来重新实现我们前面的异步模式客户端。不过，首先我们先稍微写点简单的 twisted 程序来认识一下 twisted。
最最简单的 twisted 程序就是下面的代码，其在 twisted-intro 目录中的 basic-twisted/simple.py 中。

```
from twisted.internet import reactor
reactor.run()
```

可以用下面的命令来运行它：

```
python basic-twisted/simple.py
```

正如在第二部分所说的那样，twisted 是实现了 Reactor 模式的，因此它必然会有一个对象来代表这个 reactor 或者说是事件循环，而这正是 twisted 的核心。上面代码的第一行引入了 reactor，第二行开始启动事件循环。
这个程序什么事情也不做。除非你通过 ctrl+c 来终止它，否则它会一直运行下去。正常情况下，我们需要给出事件循环或者文件描述符来监视 I/O（连接到某个服务器上，比如说我们那个诗歌服务器）。后面我们会来介绍这部分内容，现在这里的 reactor 被卡住了。值得注意的是，这里并不是一个在不停运行的简单循环。如果你在桌面上有个 CPU 性能查看器，可以发现这个循环体不会带来任何性能损失。实际上，这个 reactor 被卡住在第二部分图 5 的最顶端，等待永远不会到来的事件发生（更具体点说是一个调用 select 函数，却没有监视任何文件描述符)。
下面我们会让这个程序丰富起来，不过事先要说几个结论：
1.Twisted 的 reactor 只有通过调用 reactor.run()来启动。
2.reactor 循环是在其开始的进程中运行，也就是运行在主进程中。
3.一旦启动，就会一直运行下去。reactor 就会在程序的控制下（或者具体在一个启动它的线程的控制下）。
4.reactor 循环并不会消耗任何 CPU 的资源。
5.并不需要显式的创建 reactor，只需要引入就 OK 了。

最后一条需要解释清楚。在 Twisted 中，reactor 是 Singleton（也是一种模式），即在一个程序中只能有一个 reactor，并且只要你引入它就相应地创建一个。上面引入的方式这是 twisted 默认使用的方法，当然了，twisted 还有其它可以引入 reactor 的方法。例如，可以使用 twisted.internet.pollreactor 中的系统调用来 poll 来代替 select 方法。
若使用其它的 reactor，需要在引入 twisted.internet.reactor 前安装它。下面是安装 pollreactor 的方法：

```
from twisted.internet import pollreactor
pollreactor.install()
```

如果你没有安装其它特殊的 reactor 而引入了 twisted.internet.reactor，那么 Twisted 会为你安装 selectreactor。正因为如此，习惯性做法不要在最顶层的模块内引入 reactor 以避免安装默认 reactor，而是在你要使用 reactor 的区域内安装。
下面是使用 pollreactor 重写上上面的程序，可以在 basic-twisted/simple-poll.py 文件中找到中找到：

```
from twited.internet import pollreactor
pollreactor.install()
from twisted.internet import reactor
reactor.run()
```

上面这段代码同样没有做任何事情。
后面我们都会只使用默认的 reactor，就单纯为了学习来说 ，所有的不同的 reactor 做的事情都一样。

你好，Twisted

我们得用 Twisted 来做什么吧。下面这段代码在 reactor 循环开始后向终端打印一条消息：

```
def hello():
    print 'Hello from the reactor loop!'
    print 'Lately I feel like I\'m stuck in a rut.'
from twisted.internet import reactor
reactor.callWhenRunning(hello)
print 'Starting the reactor.'
reactor.run()
```

这段代码可以在 basic-twisted/hello.py 中找到。运行它，会得到如下结果：

```
Starting the reactor.

Hello from the reactor loop!

Lately I feel like I'm stuck in a rut.
```

仍然需要你手动来关掉程序，因为它在打印完毕后就又卡住了。
值得注意的是，hello 函数是在 reactor 启动后被调用的。这意味是 reactor 调用的它，也就是说 Twisted 在调用我们的函数。我们通过调用 reactor 的 callWhenRunning 函数，并传给它一个我们想调用函数的引用来实现 hello 函数的调用。当然，我们必须在启动 reactor 之前完成

这些工作。
我们使用回调来描述 hello 函数的引用。回调实际上就是交给 Twisted（或者其它框架）的一个函数引用，这样 Twisted 会在合适的时间调用这个函数引用指向的函数，具体到这个程序中，是在 reactor 启动的时候调用。由于 Twisted 循环是独立于我们的代码，我们的业务代码与 reactor 核心代码的绝大多数交互都是通过使用 Twisted 的 APIs 回调我们的业务函数来实现的。
我们可以通过下面这段代码来观察 Twisted 是如何调用我们代码的：

```
import traceback
def stack():
    print 'The python stack:'
    traceback.print_stack()
from twisted.internet import reactor
reactor.callWhenRunning(stack)
reactor.run()
```

这段代码的文件是 basic-twisted/stack.py。不出意外，它的输出是：

```
The python stack:

...
  reactor.run() <-- This is where we called the reactor

...
...   <-- A bunch of Twisted function calls
...

 traceback.print_stack() <-- The second line in the stack function
```

不用考虑这其中的若干 Twisted 本身的函数。只需要关心 reactor.run()与我们自己的函数调用之间的关系即可。

有关回调的一些其它说明：

Twisted 并不是唯一使用回调的框架。许多历史悠久的框架都已在使用它。诸多 GUI 的框架也是基于回调来实现的，如 GTK 和 QT。
交互式程序的编程人员特别喜欢回调。也许喜欢到想嫁给它。也许已经这样做了。但下面这几点值得我们仔细考虑下：
1.reactor 模式是单线程的。
2.像 Twisted 这种交互式模型已经实现了 reactor 循环，意味无需我们亲自去实现它。
3.我们仍然需要框架来调用我们自己的代码来完成业务逻辑。
4.因为在单线程中运行，要想跑我们自己的代码，必须在 reactor 循环中调用它们。
5.reactor 事先并不知道调用我们代码的哪个函数
这样的话，回调并不仅仅是一个可选项，而是游戏规则的一部分。
图 6 说明了回调过程中发生的一切：

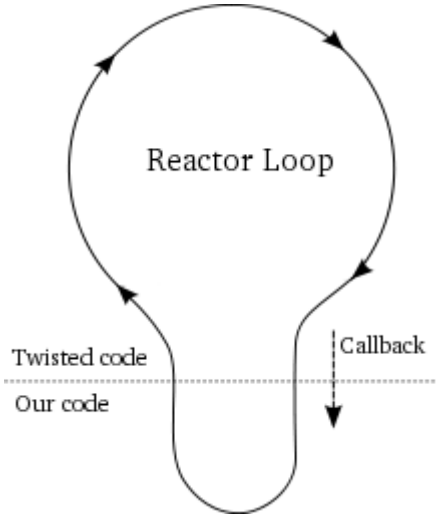


图 6 reactor 启用回调
图 6 揭示了回调中的几个重要特性：
1.我们的代码与 Twisted 代码运行在同一个进程中。
2.当我们的代码运行时，Twisted 代码是处于暂停状态的。
3.同样，当 Twisted 代码处于运行状态时，我们的代码处于暂停状态。
4.reactor 事件循环会在我们的回调函数返回后恢复运行。

在一个回调函数执行过程中，实际上 Twisted 的循环是被有效地阻塞在我们的代码上的。因此，因此我们应该确保回调函数不要浪费时间（尽快返回）。特别需要强调的是，我们应该尽量避免在回调函数中使用会阻塞 I/O 的函数。否则，我们将失去所有使用 reactor 所带来的优势。Twisted 是不会采取特殊的预防措施来防止我们使用可阻塞的代码的，这需要我们自己来确保上面的情况不会发生。正如我们实际所看到的一样，对于普通网络 I/O 的例子，由于我们让 Twisted 替我们完成了异步通信，因此我们无需担心上面的事情发生。
其它也可能会产生阻塞的操作是读或写一个非 socket 文件描述符（如管道）或者是等待一个子进程完成。
如何从阻塞转换到非阻塞操作取决你具体的操作是什么，但是也有一些 Twisted APIs 会帮助你实现转换。值得注意的是，很多标准的 Python 方法没有办法转换为非阻塞方式。例如，os.system 中的很多方法会在子进程完成前一直处于阻塞状态。这也就是它工作的方式。所以当你使用 Twisted 时，避开使用 os.system。

退出 Twisted

原来我们可以使用 reactor 的 stop 方法来停止 Twisted 的 reactor。但是一旦 reactor 停止就无法再启动了。（Dave 的意思是，停止就退出程序了），因此只有在你想退出程序时才执行这个操作。
下面是退出代码，代码文件是 basic-twisted/countdown.py：

```
class Countdown(object):
counter = 5
def count(self):
from twisted.internet import reactor
if self.counter == 0:
reactor.stop()
else:
print self.counter, '...'
self.counter -= 1
reactor.callLater(1, self.count)
from twisted.internet import reactor
reactor.callWhenRunning(Countdown().count)

print 'Start!'
reactor.run()
print 'Stop!'
```

在这个程序中使用了 callLater 函数为 Twisted 注册了一个回调函数。callLater 中的第二个参数是回调函数，第一个则是说明你希望在将来几秒钟时执行你的回调函数。那 Twisted 如何来在指定的时间执行我们安排好的的回调函数。由于程序并没有监听任何文件描述符，为什么它没有像前那些程序那样卡在 select 循环上？select 函数，或者其它类似的函数，同样会接纳一个超时参数。如果在只提供一个超时参数值并且没有可供 I/O 操作的文件描述符而超时时间到时，select 函数同样会返回。因此，如果设置一个 0 的超时参数，那么会无任何阻塞地立即检查所有的文件描述符集。

你可以将超时作为图 5 中循环等待中的一种事件来看待。并且 Twisted 使用超时事件来确保那些通过 callLater 函数注册的延时回调在指定的时间执行。或者更确切的说，在指定时间的前后会执行。如果一个回调函数执行时间过长，那么下面的延时回调函数可能会被相应的后延执行。Twisted 的 callLater 机制并不为硬实时系统提供任何时间上的保证。

下面是上面程序的输出：

```
Start!

5 ...

4 ...

3 ...

2 ...

1 ...

Stop!
```

捕获它，Twisted

由于 Twisted 经常会在回调中结束调用我们的代码，因此你可能会想，如果我们的回调函数中出现异常会发生什么状况。（Dave 的意思是说，在结束我们的回调函数后会再次回到 Twisted 代码中，若在我们的回调中发生异常，那是不是异常会跑到 Twisted 代码中，而造成不可想象的后果 ）让我们来试试，在 basic-twisted/exception.py 中的程序会在一个回调函数中引发一个异常，但是这不会影响下一个回调：

```
def falldown():
raise Exception('I fall down.')

def upagain():
print 'But I get up again.'
reactor.stop()

from twisted.internet import reactor

reactor.callWhenRunning(falldown)
reactor.callWhenRunning(upagain)

print 'Starting the reactor.'
reactor.run()
```

当你在命令行中运时，会有如下的输出：

```
Starting the reactor.
Traceback (most recent call last):

  ... # I removed most of the traceback

exceptions.Exception: I fall down.

But I get up again.
```

注意，尽管我们看到了因第一个回调函数引发异常而出现的跟踪栈，第二个回调函数依然能够执行。如果你将 reactor.stop()注释掉的话，程序会继续运行下去。所以说，reactor 并不会因为回调函数中出现失败（虽然它会报告异常）而停止运行。

网络服务器通常需要这种健壮的软件。它们通常不希望由于一个随机的 Bug 导致崩溃。也并不是说当我们发现自己的程序内部有问题时，就垂头丧气。只是想说 Twisted 能够很好的从失败的回调中返回并继续执行。

请继续讲解诗歌服务器

现在，我们已经准备好利用 Twisted 来搭建我们的诗歌服务器。在第 4 部分，我们会实现我

们的异步模式的诗歌服务器的 Twisted 版。

第四部分：由 Twisted 支持的诗歌客户端

第一个 twisted 支持的诗歌服务器

尽管 Twisted 大多数情况下用来写服务器代码，为了一开始尽量从简单处着手，我们首先从简单的客户端讲起。
让我们来试试使用 Twisted 的客户端。源码在 twisted-client-1/get-poetry.py。首先像前面一样要开启三个服务器：

```
python blocking-server/slowpoetry.py --port 10000 poetry/ecstasy.txt --num-bytes 30
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
```

并且运行客户端：

```
python twisted-client-1/get-poetry.py 10000 10001 10002
```

你会看到在客户端的命令行打印出：

```
Task 1: got 60 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 2: got 10 bytes of poetry from 127.0.0.1:10001

...

Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry

Got 3 poems in 0:00:10.134220
```

和我们的没有使用 Twisted 的非阻塞模式客户端打印的内容接近。这并不奇怪，因为它们的

工作方式是一样的。
下面，我们来仔细研究一下它的源代码。
注意：正如我在第一部分说到，我们开始学习使用 Twisted 时会使用一些低层 Twisted 的 APIs。这样做是为揭去 Twisted 的抽象层，这样我们就可以从内向外的来学习 Tiwsted。但是这就意味着，我们在学习中所使用的 APIs 在实际应用中可能都不会见到。记住这么一点就行：前面这些代码只是用作练习，而不是写真实软件的例子。
可民看到，首先创建了一组 PoetrySocket 的实例。在 PoetrySocket 初始化时，其创建了一个网络 socket 作为自己的属性字段来连接服务器，并且选择了非阻塞模式：

```
self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.sock.connect(address)
self.sock.setblocking(0)
```

最终我们虽然会提高到不使用 socket 的抽象层次上，但这里我们仍然需要使用它。在创建完 socket 后，PoetrySocket 通过方法 addReader 将自己传递给 reactor：

```
# tell the Twisted reactor to monitor this socket for reading
from twisted.internet import reactor
reactor.addReader(self）
```

这个方法给 Twisted 提供了一个文件描述符来监视要发送来的数据。为什么我们不传递给 Twisted 一个文件描述符或回调函数而是一个对象实例？并且 Twisted 内部没有任何与这个诗歌服务相关的代码，它怎么知道该如何与我们的对象实例交互？相信我，我已经查看过了，打开 twisted.internet.interfaces 模块，和我一起来搞清楚是怎么回事。

Twisted 接口

在 twisted 内部有很多被称作接口的子模块。每个都定义了一组接口类。由于在 8.0 版本中，Twisted 使用 zope.interface 作为这些类的基类。但我们这里并不来讨论它其中的细节。我们只关心其在 Twisted 的子类，就是你看到的那些。
使用接口的核心目的之一就是文档化。作为一个 python 程序员，你肯定知道 Duck Typing。（说实话我还真不懂这种编程，但通过查看资料，其实就是动态编程的思想，根据你的动作来确定你的类型）
翻阅 twisted.internet.interfaces 找到方法的 addReader 定义，它的定义在 IReactorFDSet 中可以找到：

```
def addReader(reader):
    """
    I add reader to the set of file descriptors to get read events for.
    @param reader: An L{IReadDescriptor} provider that will be checked for
                   read events until it is removed from the reactor with
                   L{removeReader}.
    @return: C{None}.
    """
```

IReactorFDSet 是一个 Twisted 的 reactor 实现的接口。因此任何一个 Twisted 的 reactor 都会一个 addReader 的方法，如同上面描述的一样工作。这个方法声明之所以没有 self 参数是因为它仅仅关心一个公共接口定义，self 参数仅仅是接口实现时的一部分（在调用它时，也没

有显式地传入一个 self 参数）。接口类永远不会被实例化或作为基类来继承实现。

注意 1： 技术上讲，IReactorFDSet 只会由 reactor 实现用来监听文件描述符。具我所知，现在所有已实现 reactor 都会实现这个接口。

注意 2： 使用接口并不仅仅是为了文档化。zope.interface 允许你显式地来声明一个类实现一个或多个接口，并提供运行时检查这些实现的机制。同样也提供代理这一机制，它可以动态地为一个没有实现某接口的类直接提供该接口。但我们这里就不做深入学习了。

注意 3： 你可能已经注意到接口与最近添加到 Python 中虚基类的相似性了。这里我们并不去分析它们之间的相似性与差异。若你有兴趣，可以读读 Ptyhon 项目的创始人 Glyph 写的一篇关于这个话题的文章。

根据文档的描述可以看出，addReader 的 reader 参数是要实现 IreadDescriptor 接口的。这也就意味我们的 PoetrySocket 也必须这样做。

阅读接口模块我们可以看到下面这段代码：

```
class IReadDescriptor(IFileDescriptor):
    def doRead():
        """
        Some data is available for reading on your descriptor.
        """
```

同时你会看到在我们的 PoetrySocket 类中有一个 doRead 方法。当其被 Twisted 的 reactor 调用时，就会采用异步的方式从 socket 中读取数据。因此，doRead 其实就是一个回调函数，只是没有直接将其传递给 reactor，而是传递一个实现此方法的对象实例。这也是 Twisted 框架中的惯例—不是直接传递实现某个接口的函数而是传递实现它的对象。这样我们通过一个参数就可以传递一组相关的回调函数。而且也可以让回调函数之间通过存储在对象中的数据进行通信。

那在 PoetrySocket 中实现其它的回调函数呢？注意到 IReadDescriptor 是 IFileDescriptor 的一个子类。这也就意味任何一个实现 IReadDescriptor 都必须实现 IFileDescriptor。若是你仔细阅读代码会看到下面的内容：

```
class IFileDescriptor(ILoggingContext):
    """
    A file descriptor.
    """
    def fileno():
        ...
    def connectionLost(reason):
        …
```

我将文档描述省略掉了，但这些函数的功能从字面上就可以理解：fileno 返回我们想监听的文件描述符，connectionLost 是当连接关闭时被调用。你也看到了，PoetrySocket 实现了这些方法。

最后，IFileDescriptor 继承了 ILooggingContext，这里我不想再展现其源码。我想说的是，这就是为什么我们要实现一个 logPrefix 回调函数。你可以在 interface 模块中找到答案。

注意： 你也许注意到了，当连接关闭时，在 doRead 中返回了一个特殊的值。我是如何

知道的？说实话，没有它程序是无法正常工作的。我是在分析 Twisted 源码中发现其它相应的方法采取相同的方法。你也许想好好研究一下：但有时一些文档或书的解释是错误的或不完整的。因此可能当你搞清楚怎么回事时，我们已经完成第五部分了呵呵。

更多关于回调的知识

我们使用 Twisted 的异步客户端和前面的没有使用 Twisted 的异步客户非常的相似。两者都要连接它们自己的 socket，并以异步的方式从中读取数据。最大的区别在于：使用 Twisted 的客户端并没有使用自己的 select 循环-而使用了 Twisted 的 reactor。
doRead 回调函数是非常重要的一个回调。Twisted 调用它来告诉我们已经有数据在 socket 接收完毕。我可以通过图 7 来形象地说明这一过程：

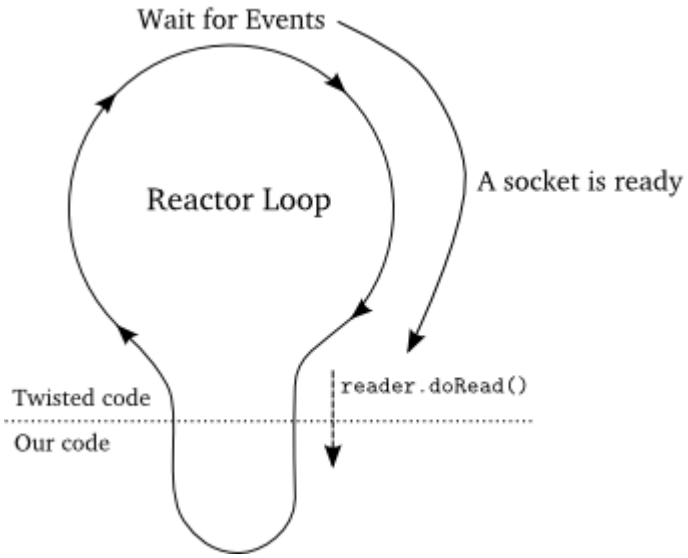


图 7 doRead 回调过程

每当回调被激活，就轮到我们的代码将所有能够读的数据读回来然后非阻塞式的停止。正如我们第三部分说的那样，Twisted 是不会因为什么异常状况（如没有必要的阻塞）而终止我们的代码。那么我们就故意写个会产生异常状况的客户端看看到底能发生什么事情。可以在 twisted-client-1/get-poetry-broken.py 中看到源代码。这个客户端与你前面看到的同样有两个异常状况出现：
1.这个客户端并不没有选择非阻塞式的 socket
2.doRead 回调方法在 socket 关闭连接前一直在不停地读 socket
现在让我们运行一下这个客户端：
python twisted-client-1/get-poetry-broken.py 10000 10001 10002

我们出得到如同下面一样的输出：

```
Task 1: got 3003 bytes of poetry from 127.0.0.1:10000
Task 3: got 653 bytes of poetry from 127.0.0.1:10002
Task 2: got 623 bytes of poetry from 127.0.0.1:10001
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry

Got 3 poems in 0:00:10.132753
```

可能除了任务的完成顺序不太一致外，和我先阻塞式客户端是一样的。这是因为这个客户端是一个阻塞式的。

由于使用了阻塞式的连接，就将我们的非阻塞式客户端变成了阻塞式的客户端。这样一来，我们尽管遭受了使用 select 的复杂但却没有享受到其带来的异步优势。

像诸如 Twisted 这样的事件循环所提供的多任务的能力是需要用户的合作来实现的。Twisted 会告诉我们什么时候读或写一个文件描述符，但我们必须要尽可能高效而没有阻塞地完成读写工作。同样我们应该禁止使用其它各类的阻塞函数，如 os.system 中的函数。除此之外，当我们遇到计算型的任务（长时间占用 CPU），最好是将任务切成若干个部分执行以让 I/O 操作尽可能地执行。

你也许已经注意到这个客户端所 花费的时间少于先前那个阻塞的客户端。这是由于这个在一开始就与所有的服务建立连接，由于服务是一旦连接建立就立即发送数据，而且我们的操作系统会缓存一 部分发送过来但尚读不到的数据到缓冲区中（缓冲区大小是有上限的）。因此就明白了为什么前面那个会慢了：它是在完成一个后再建立下一个连接并接收数据。

但这种小优势仅仅在小数据量的情况下才会得以体现。如果我们下载三首 20M 个单词的诗，那时 OS 的缓冲区会在瞬间填满，这样一来我们这个客户端与前面那个阻塞式客户端相比就没有什么优势可言了。

结束语

我没有过多地解释此部分第一个客户端的内容。你可能注意到了， connectionLost 函数会在没有 PoetrySocket 等待诗歌后关闭 reactor。由于我们的程序除了下载诗歌不提供其它服务，所以才会这样做。但它揭示了两个低层 reactor 的 APIs：removeReader 和 getReaders。

还有与我们客户端使用的 Readers 的 APIs 类同的 Writers 的 APIs，它们采用相同的方式来监视我们要发送数据的文件描述符。可以通过阅读 interfaces 文件来获取更多的细节。读和写有各自的 APIs 是因为 select 函数需要分开这两种事件（读或写可以进行的文件描述符）。当然了，可以等待即能读也能写的文件描述符。

第五部分，我们将使用 Twisted 的高层抽象方式实现另外一个客户端，并且学习更多的 Twisted 的接口与 APIs。

第五部分：由 Twisted 支持的诗歌客户端

抽象地构建客户端

在第四部分中，我们构建了第一个使用 Twisted 的客户端。它确实能很好地工作，但仍有提高的空间。

首先是，这个客户端竟然有创建网络端口并接收端口处的数据这样枯燥的代码。Twisted 理应为我们实现这些例程性功能，省得我们每次写一个新的程序时都要去自己实现。Twisted 这样做也将我们从像异步 I/O 操作中包括许多像异常处理这样的细节处理解放出来。更多的细节处理存在于多平台上运行我们的代码中。如果你那个下午有空，可以翻翻 Twisted 的 WIN32 实现源代码，看看里面有多少小针线是来处理跨平台的。

另一问题是与错误处理有关。当运行版本 1 的 Twisted 客户端来从并没有提供服务的端口上下载诗歌时，它就会崩溃。我们是可以修正这个错误，但通过下面我们要介绍 Twisted 的 APIs 来处理这些类型的错误会更简单。

最后，那个客户端也不能复用。如果有另一个模块需要通过我们的客户端下载诗歌呢？人家怎么知道你的诗歌已经下载完毕？我们不能用一个方法简单地将一首诗下 载完成后再传给人家，而在之前让人家处于等待状态。这确实是一个问题，但我们不准备在这个部分解决这个问题—在未来的部分中一定会解决这个问题。

我们将会使用一些高层次的 APIs 和接口来解决第一、二个问题。Twisted 框架是由众多抽象层松散地组合起来的。因此，学习 Twisted 也就意味着需要学习这些层都提供什么功能，例如每层都有哪些 APIs，接口和实例可供使用。接下来我们会通过剖析 Twisted 最最重要的部分来更好地感受一下 Twisted 都是怎么组织的。一旦你对 Twisted 的整个结构熟悉了，学习新的部分会简单多了。

一般来说，每个 Twisted 的抽象都只与一个特定的概念相关。例如，第四部分中的客户端使用的 IReadDescriptor，它就是“一个可以读取字节的文件描述符”的抽象。一个抽象往往会通过定义接口来指定那些想实现个抽象（也就是实现这个接口）对象的形为。在学习新的 Twisted 抽象概念时，最需要谨记的就是：

多数高层次抽象都是在低层次抽象的基础上建立的，很少有另立门户的。

因此，你在学习新的 Twisted 抽象概念时，始终要记住它做什么和不做什么。特别是，如果一个早期的抽象 A 实现了 F 特性，那么 F 特性不太可能再由其它任何抽象来实现。另外，如果另外一个抽象需要 F 特性，那么它会使用 A 而不是自己再去实现 F。（通常的做法，B 可能会通过继承 A 或获得一个指向 A 实例的引用）

网络非常的复杂，因此 Twisted 包含很多抽象的概念。通过从低层的抽象讲起，我们希望能更清楚起看到在一个 Twisted 程序中各个部分是怎么组织起来的。

核心的循环体

第一个我们要学习的抽象，也是 Twisted 中最重要的，就是 reactor。在每个通过 Twisted 搭建起来的程序中心处，不管你这个程序有多少层，总会有一个 reactor 循环在不停止地驱动程序的运行。再也没有比 reactor 提供更加基础的支持了。实际上，Twisted 的其它部分（即除了 reactor 循环体）可以这样理解：它们都是来辅助 X 来更好地使用 reactor，这里的 X 可以是提供 Web 网页、处理一个数据库查询请求或其它更加具体内容。尽管坚持像上一个客户端一样使用低层 APIs 是可能的，但如果我们执意那样做，那么我们必需自己来实现非常多的内容。而在更高的层次上，意味着我们可以少写很多代码。

但是当在外层思考与处理问题叶。很容易就忘记了 reactor 的存在了。在任何一个常见大小的 Twisted 程序中 ，确实很少会有直接与 reactor 的 APIs 交互。低层的抽象也是一样（即我们很少会直接与其交互）。我们在上一个客户端中用到的文件描述符抽象，就被更高层的抽象更好的归纳而至于我们很少会在真正的 Twisted 程序中遇到。（他们在内部依然在被使用，只是我们看不到而已）

至于文件描述符抽象的消息，这并不是一个问题。让 Twisted 掌舵异步 I/O 处理，这样我们就可以更加关注我们实际要解决的问题。但对于 reactor 不一样，它永远都不会消失。当你选择使用 Twisted，也就意味着你选择使用 Reactor 模式，并且意味着你需要使用回调与多任务合作的“交互式”编程方式。如果你想正确地使用 Twisted，你必须牢记 reactor 的存在。我们将在第六部分更加详细的讲解部分内容。但是现在要强调的是：

图 5 与图 6 是这个系列中最最重要的图

我们还将用图来描述新的概念，但这两个图是需要你牢记在脑海中的。可以这样说，我在写 Twisted 程序时一直想着这两张图。

在我们付诸于代码前，有三个新的概念需要阐述清楚：Transports,Protocols,Protocol Factoies

Transports

Transports 抽象是通过 Twisted 中 interfaces 模块中 ITransport 接口定义的。一个 Twisted 的 Transport 代表一个可以收发字节的单条连接。对于我们的诗歌下载客户端而言，就是对一条 TCP 连接的抽象。但是 Twisted 也支持诸如 Unix 中管道和 UDP。Transport 抽象可以代表任何这样的连接并为其代表的连接处理具体的异步 I/O 操作细节。

如果你浏览一下 ITransport 中的方法，可能找不到任何接收数据的方法。这是因为 Transports 总是在低层完成从连接中异步读取数据的许多细节工作，然后通过回调将数据发给我们。相似的原理，Transport 对象的写相关的方法为避免阻塞也不会选择立即写我们要发送的数据。告诉一个 Transport 要发送数据，只是意味着：尽快将这些数据发送出去，别产生阻塞就行。当然，数据会按照我们提交的顺序发送。

通常我们不会自己实现一个 Transport。我们会去实现 Twisted 提供的类，即在传递给 reactor 时会为我们创建一个对象实例。

Protocols

Twisted 的 Protocols 抽象由 interfaces 模块中的 IProtocol 定义。也许你已经想到，Protocol 对象实现协议内容。也就是说，一个具体的 Twisted 的 Protocol 的实现应该对应一个具体网络协议的实现，像 FTP、IMAP 或其它我们自己规定的协议。我们的诗歌下载协议，正如它表现的那样，就是在连接建立后将所有的诗歌内容全部发送出去并且在发送完毕后关闭连接。

严格意义上讲，每一个 Twisted 的 Protocols 类实例都为一个具体的连接提供协议解析。因此我们的程序每建立一条连接（对于服务方就是每接受一条连接），都需要一个协议实例。这就意味着，Protocol 实例是存储协议状态与间断性（由于我们是通过异步 I/O 方式以任意大小来接收数据的）接收并累积数据的地方。

因此，Protocol 实例如何得知它为哪条连接服务呢？如果你阅读 IProtocol 定义会发现一个 makeConnection 函数。这是一个回调函数，Twisted 会在调用它时传递给其一个也是仅有的一个参数，即就是 Transport 实例。这个 Transport 实例就代表 Protocol 将要使用的连接。

Twisted 包含很多内置可以实现很多通用协议的 Protocol。你可以在 twisted.protocols.basic 中找到一些稍微简单点的。在你尝试写新 Protocol 时，最好是看看 Twisted 源码是不是已经有现成的存在。如果没有，那实现一个自己的协议是非常好的，正如我们为诗歌下载客户端做的那样。

Protocol Factories

因此每个连接需要一个自己的 Portocol，而且这个 Protocol 是我们自己定义类的实例。由于我们会将创建连接的工作交给 Twisted 来完成，Twisted 需要一种方式来为一个新的连接制定一个合适的协议。制定协议就是 Protocol Factories 的 工作了。

也许你已经猜到了，Protocol Factory 的 API 由 IProtocolFactory 来定义，同样在 interfaces 模块中。Protocol Factory 就是 Factory 模式的一个具体实现。buildProtocol 方法在每次被调用时返回一个新 Protocol 实例。它就是 Twisted 用来为新连接创建新 Protocol 实例的方法。

诗歌下载客户端 2.0：第一滴心血

好吧，让我们来看看由 Twisted 支持的诗歌下载客户端 2.0。源码可以在这里 twisted-client-2/get-poetry.py。你可以像前面一样运行它，并得到相同的输出。这也是最后一个在接收到数据时打印其任务的客户端版本了。到现在为止，对于所有 Twisted 程序都是交替执行任务并处理相对较少数量数据的，应该很清晰了。我们依然通过 print 函数来展示在关键时刻在进行什么内容，但将来客户端不会在这样繁锁。

在第二个版本中，sockets 不会再出现了。我们甚至不需要引入 socket 模块也不用引用 socket 对象和文件描述符。取而代之的是，我们告诉 reactor 来创建到诗歌服务器的连接，代码如下面所示：

```
factory = PoetryClientFactory(len(addresses))

from twisted.internet import reactor

for address in addresses:
    host, port = address
    reactor.connectTCP(host, port, factory)
```

我们需要关注的是 connectTCP 这个函数。前两个参数的含义很明显，不解释了。第三个参数是我们自定义的 PoetryClientFactory 类的实例对象。这是一个专门针对诗歌下载客户端的 Protocol Factory，将它传递给 reactor 可以让 Twisted 为我们创建一个 PeotryProtocol 实例。

值得注意的是，从一开始我们既没有实现 Factory 也没有去实现 Protocol，不像在前面那个客户端中我们去实例化我们 PoetrySocket 类。我们只是继承了 Twisted 在 twisted.internet.protocol 中提供的基类。Factory 的基类是 twisted.internet.protocol.Factory，但我们使用客户端专用（即不像服务器端那样监听一个连接，而是主动创建一个连接）的 ClientFactory 子类来继承。

我们同样利用了 Twisted 的 Factory 已经实现了 buildProtocol 方法这一优势来为我们所用。我们要在子类中调用基类中的实现：

```
def buildProtocol(self, address):
    proto = ClientFactory.buildProtocol(self, address)
    proto.task_num = self.task_num
    self.task_num += 1
    return proto
```

基类怎么会知道我们要创建什么样的 Protocol 呢？注意，我们的 PoetryClientFactory 中有一个 protocol 类变量：

```
class PoetryClientFactory(ClientFactory):

    task_num = 1

    protocol = PoetryProtocol # tell base class what proto to build
```

基类 Factory 的实现 buildProtocol 过程是：安装（创建一个实例）我们设置在 protocol 变量上的 Protocol 类与在这个实例（此处即 PoetryProtocol 的实例）的 factory 属性上设置一个产生它的 Factory 的引用（此处即实例化 PoetryProtocol 的 PoetryClientFactory）。这个过程如图 8 所示：

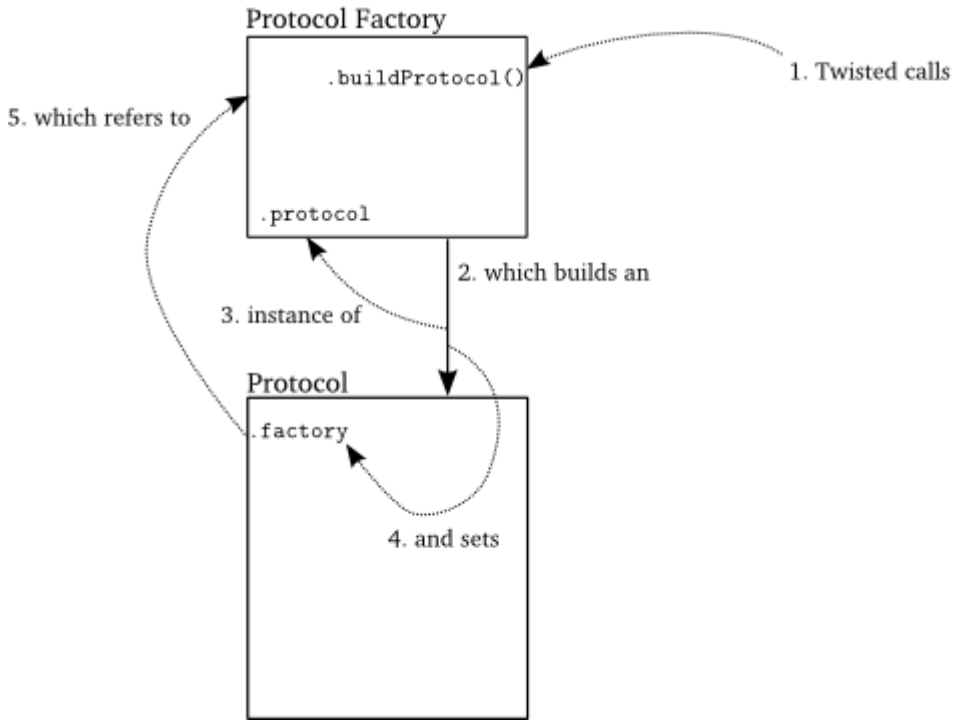


图 8：Protocol 的生成过程

正如我们提到的那样，位于 Protocol 对象内的 factory 属性字段允许在都由同一个 factory 产生的 Protocol 之间共享数据。由于 Factories 都是由用户代码来创建的（即在用户的控制中），因此这个属性也可以实现 Protocol 对象将数据传递回一开始初始化请求的代码中来，这将在第六部分看到。

值得注意的是，虽然在 Protocol 中有一个属性指向生成其的 Protocol Factory，在 Factory 中也有一个变量指向一个 Protocol 类，但通常来说，一个 Factory 可以生成多个 Protocol。

在 Protocol 创立的第二步便是通过 makeConnection 与一个 Transport 联系起来。我们无需自己来实现这个函数而使用 Twisted 提供的默认实现。默认情况是，makeConnection 将 Transport 的一个引用赋给（Protocol 的）transport 属性，同时置（同样是 Protocol 的）connected 属性为 True，正如图 9 描述的一样：

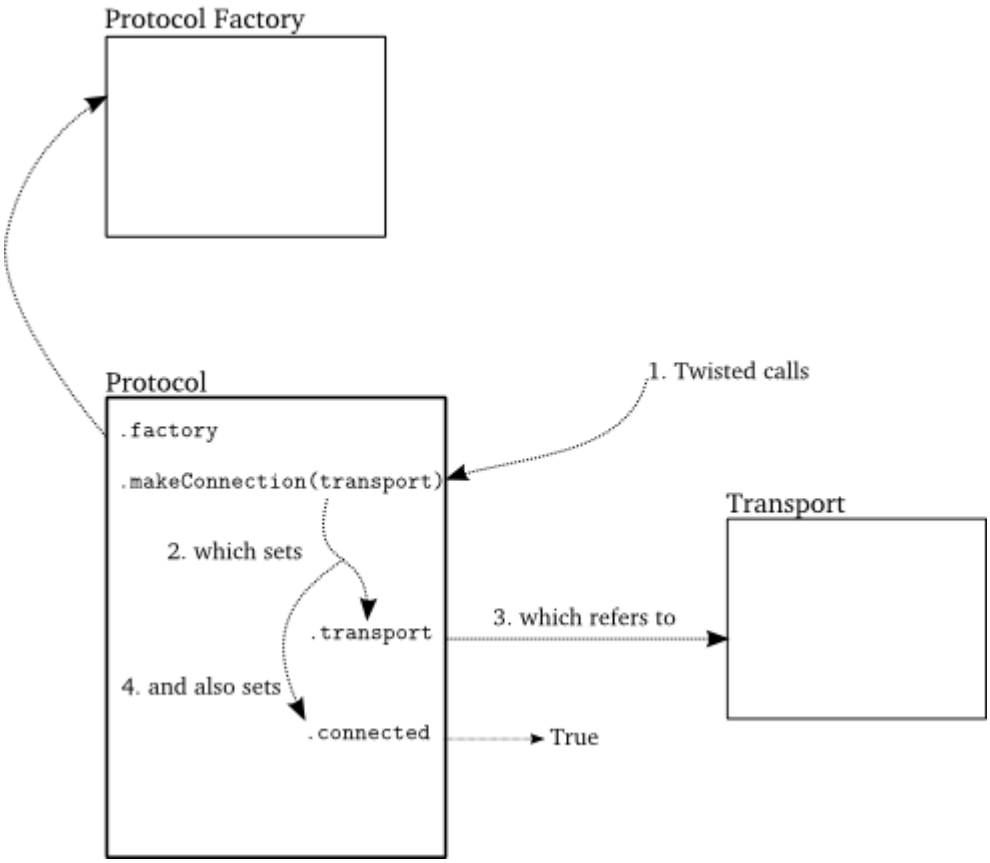


图 9：Protocol 遇到其 Transport

一旦初始化到这一步后，Protocol 开始其真正的工作—将低层的数据流翻译成高层的协议规定格式的消息。处理接收到数据的主要方法是 dataReceived，我们的客户端是这样实现的：

```
def dataReceived(self, data):
    self.poem += data
    msg = 'Task %d: got %d bytes of poetry from %s'
    print    msg % (self.task_num, len(data), self.transport.getHost())
```

每次 dateReceved 被调用就意味着我们得到一个新字符串。由于与异步 I/O 交互，我们不知道能接收到多少数据，因此将接收到的数据缓存下来直到完成一个完整的协议规定格式的消息。在我们的例子中，诗歌只有在连接关闭时才下载完毕，因此我们只是不断地将接收到的数据添加到我们的.poem 属性字段中。

注意我们使用了 Transport 的 getHost 方法来取得数据来自的服务器信息。我们这样做只是与前面的客户端保持一致。相反，我们的代码没有必要这样做，因为我们没有向服务器发送任何消息，也就没有必要知道服务器的信息了。

我们来看一下 dataReceved 运行时的快照。在 2.0 版本相同的目录下有一个 twisted-client-2/get-poetry-stack.py。它与 2.0 版本的不同之处只在于：

```
def dataReceived(self, data):
    traceback.print_stack()
    os._exit(0)
```

这样一改，我们就能打印出跟踪堆栈的信息，然后离开程序，可以用下面的命令来运行它：

```
python twisted-client-2/get-poetry-stack.py 10000
```

你会得到内容如下的跟踪堆栈：

```
File "twisted-client-2/get-poetry-stack.py", line 125, in

    poetry_main()


... # I removed a bunch of lines here


File ".../twisted/internet/tcp.py", line 463, in doRead # Note the doRead callback

    return self.protocol.dataReceived(data)

File "twisted-client-2/get-poetry-stack.py", line 58, in dataReceived
     traceback.print_stack()
```

看见没，有我们在 1.0 版本客户端的 doRead 回调函数。我们前面也提到过，Twisted 在建立新抽象层进会使用已有的实现而不是另起炉灶。因此必然会有一个 IReadDescriptor 的实例在辛苦的工作，它是由 Twisted 代码而非我们自己的代码来实现。如果你表示怀疑，那么就看看 twisted.internet.tcp 中的实现吧。如果你浏览代码会发现，由同一个类实现了 IWriteDescriptor 与 ITransport。因此 IreadDescriptor 实际上就是变相的 Transport 类。可以用图 10 来形象地说明 dateReceived 的回调过程：

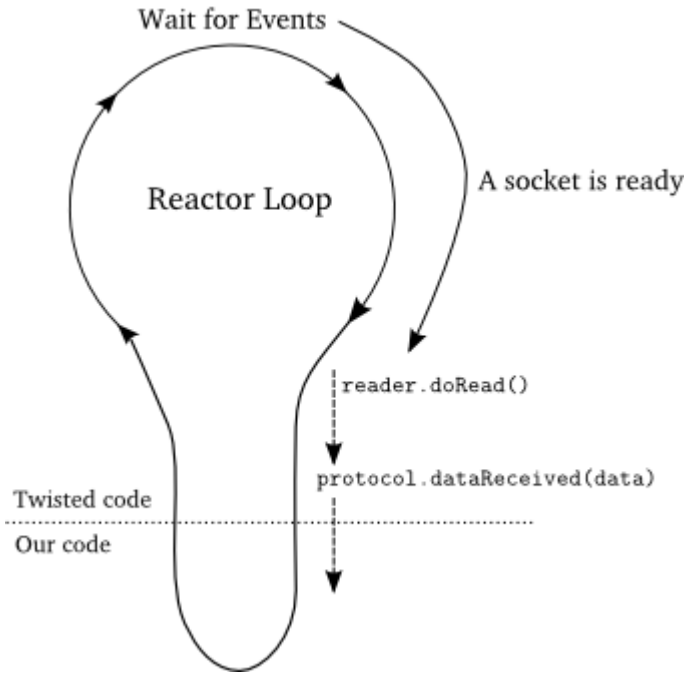


图 10：dateReceived 回调过程

一旦诗歌下载完成，PoetryProtocol 就会通知它的 PooetryClientFactory：

```
def connectionLost(self, reason):


   self.poemReceived(self.poem)

def poemReceived(self, poem):

   self.factory.poem_finished(self.task_num, poem)
```

当 transport 的连接关闭时，conncetionLost 回调会被激活。reason 参数是一个 twisted.python.failure.Failure 的实例对象，其携带的信息能够说明连接是被安全的关闭还是由于出错被关闭的。我们的客户端因认为总是能完整地下载完诗歌而忽略了这一参数。
工厂会在所有的诗歌都下载完毕后关闭 reactor。再次重申：我们代码的工作就是用来下载诗歌-这意味我们的 PoetryClientFactory 缺少复用性。我们将在下一部分修正这一缺陷。值得注意的是，poem_finish 回调函数是如何通过跟踪剩余诗歌数的：

```
...
      self.poetry_count -= 1

      if self.poetry_count == 0:
            …
```

如果我们采用多线程以让每个线程分别下载诗歌，这样我们就必须使用一把锁来管理这段代码以免多个线程在同一时间调用 poem_finish。但是在交互式体系下就不必担心了。由于 reactor 只能一次启用一个回调。
新的客户端实现在处理错误上也比先前的优雅的多，下面是 PoetryClientFactory 处理错误连接的回调实现代码：

```
def clientConnectionFailed(self, connector, reason):
      print 'Failed to connect to:', connector.getDestination()
      self.poem_finished()
```

注意，回调是在工厂内部而不是协议内部实现。由于协议是在连接建立后才创建的，而工厂能够在连接未能成功建立时捕获消息。

结束语：

版本 2 的客户端使用的抽象对于那些 Twisted 高手应该非常熟悉。如果仅仅是为在命令行上打印出下载的诗歌这个功能，那么我们已经完成了。但如果想使我们的代码能够复用，能够被内嵌在一些包含诗歌下载功能并可以做其它事情的大软件中，我们还有许多工作要做，我们将在第六部分讲解相关内容。

第五部分：由 Twisted 支持的诗歌客户端

抽象地构建客户端

在第四部分中，我们构建了第一个使用 Twisted 的客户端。它确实能很好地工作，但仍有提高的空间。

首先是，这个客户端竟然有创建网络端口并接收端口处的数据这样枯燥的代码。 Twisted 理应为我们实现这些例程性功能，省得我们每次写一个新的程序时都要去自己实现。 Twisted 这样做也将我们从像异步 I/O 操作中包括许多像异常处理这样的细节处理解放出来。更多的细节处理存在于多平台上运行我们的代码中。如果你那个下午有空，可以翻翻 Twisted 的 WIN32 实现源代码，看看里面有多少小针线是来处理跨平台的。

另一问题是与错误处理有关。当运行版本 1 的 Twisted 客户端来从并没有提供服务的端口上下载诗歌时，它就会崩溃。我们是可以修正这个错误，但通过下面我们要介绍 Twisted 的 APIs 来处理这些类型的错误会更简单。

最后，那个客户端也不能复用。如果有另一个模块需要通过我们的客户端下载诗歌呢？人家怎么知道你的诗歌已经下载完毕？我们不能用一个方法简单地将一首诗下 载完成后再传给人家，而在之前让人家处于等待状态。这确实是一个问题，但我们不准备在这个部分解决这个问题—在未来的部分中一定会解决这个问题。

我们将会使用一些高层次的 APIs 和接口来解决第一、二个问题。Twisted 框架是由众多抽象层松散地组合起来的。因此，学习 Twisted 也就意味着需要学习这些层都提供什么功能，例如每层都有哪些 APIs，接口和实例可供使用。接下来我们会通过剖析 Twisted 最最重要的部分来更好地感受一下 Twisted 都是怎么组织的。一旦你对 Twisted 的整个结构熟悉了，学习新的部分会简单多了。

一般来说，每个 Twisted 的抽象都只与一个特定的概念相关。例如，第四部分中的客户端使用的 IReadDescriptor，它就是“一个可以读取字节的文件描述符”的抽象。一个抽象往往会通过定义接口来指定那些想实现个抽象（也就是实现这个接口）对象的形为。在学习新的 Twisted 抽象概念时，最需要谨记的就是：

多数高层次抽象都是在低层次抽象的基础上建立的，很少有另立门户的。

因此，你在学习新的 Twisted 抽象概念时，始终要记住它做什么和不做什么。特别是，如果一个早期的抽象 A 实现了 F 特性，那么 F 特性不太可能再由其它任何抽象来实现。另外，如果另外一个抽象需要 F 特性，那么它会使用 A 而不是自己再去实现 F。（通常的做法，B 可能会通过继承 A 或获得一个指向 A 实例的引用）

网络非常的复杂，因此 Twisted 包含很多抽象的概念。通过从低层的抽象讲起，我们希望能更清楚起看到在一个 Twisted 程序中各个部分是怎么组织起来的。

核心的循环体

第一个我们要学习的抽象，也是 Twisted 中最重要的，就是 reactor。在每个通过 Twisted 搭建起来的程序中心处，不管你这个程序有多少层，总会有一个 reactor 循环在不停止地驱动程序的运行。再也没有比 reactor 提供更加基础的支持了。实际上，Twisted 的其它部分（即除了 reactor 循环体）可以这样理解：它们都是来辅助 X 来更好地使用 reactor，这里的 X 可以是提供 Web 网页、处理一个数据库查询请求或其它更加具体内容。尽管坚持像上一个客户端一样使用低层 APIs 是可能的，但如果我们执意那样做，那么我们必需自己来实现非常多的内容。而在更高的层次上，意味着我们可以少写很多代码。

但是当在外层思考与处理问题叶。很容易就忘记了 reactor 的存在了。在任何一个常见大小的 Twisted 程序中 ，确实很少会有直接与 reactor 的 APIs 交互。低层的抽象也是一样（即我们很少会直接与其交互）。我们在上一个客户端中用到的文件描述符抽象，就被更高层的抽象更好的归纳而至于我们很少会在真正的 Twisted 程序中遇到。（他们在内部依然在被使用，只是我们看不到而已）

至于文件描述符抽象的消息，这并不是一个问题。让 Twisted 掌舵异步 I/O 处理，这样我们就可以更加关注我们实际要解决的问题。但对于 reactor 不一样，它永远都不会消失。当你选择使用 Twisted，也就意味着你选择使用 Reactor 模式，并且意味着你需要使用回调与多任务合作的“交互式”编程方式。如果你想正确地使用 Twisted，你必须牢记 reactor 的存在。我们将在第六部分更加详细的讲解部分内容。但是现在要强调的是：

图 5 与图 6 是这个系列中最最重要的图

我们还将用图来描述新的概念，但这两个图是需要你牢记在脑海中的。可以这样说，我在写 Twisted 程序时一直想着这两张图。

在我们付诸于代码前，有三个新的概念需要阐述清楚：Transports,Protocols,Protocol Factoies

Transports

Transports 抽象是通过 Twisted 中 interfaces 模块中 ITransport 接口定义的。一个 Twisted 的 Transport 代表一个可以收发字节的单条连接。对于我们的诗歌下载客户端而言，就是对一条 TCP 连接的抽象。但是 Twisted 也支持诸如 Unix 中管道和 UDP。Transport 抽象可以代表任何这样的连接并为其代表的连接处理具体的异步 I/O 操作细节。

如果你浏览一下 ITransport 中的方法，可能找不到任何接收数据的方法。这是因为 Transports 总是在低层完成从连接中异步读取数据的许多细节工作，然后通过回调将数据发给我们。相似的原理，Transport 对象的写相关的方法为避免阻塞也不会选择立即写我们要发送的数据。告诉一个 Transport 要发送数据，只是意味着：尽快将这些数据发送出去，别产生阻塞就行。当然，数据会按照我们提交的顺序发送。

通常我们不会自己实现一个 Transport。我们会去实现 Twisted 提供的类，即在传递给 reactor 时会为我们创建一个对象实例。

Protocols

Twisted 的 Protocols 抽象由 interfaces 模块中的 IProtocol 定义。也许你已经想到，Protocol 对象实现协议内容。也就是说，一个具体的 Twisted 的 Protocol 的实现应该对应一个具体网络协议的实现，像 FTP、IMAP 或其它我们自己规定的协议。我们的诗歌下载协议，正如它表现的那样，就是在连接建立后将所有的诗歌内容全部发送出去并且在发送完毕后关闭连接。

严格意义上讲，每一个 Twisted 的 Protocols 类实例都为一个具体的连接提供协议解析。因此我们的程序每建立一条连接（对于服务方就是每接受一条连接），都需要一个协议实例。这就意味着，Protocol 实例是存储协议状态与间断性（由于我们是通过异步 I/O 方式以任意大小来接收数据的）接收并累积数据的地方。

因此，Protocol 实例如何得知它为哪条连接服务呢？如果你阅读 IProtocol 定义会发现一个 makeConnection 函数。这是一个回调函数，Twisted 会在调用它时传递给其一个也是仅有的一个参数，即就是 Transport 实例。这个 Transport 实例就代表 Protocol 将要使用的连接。

Twisted 包含很多内置可以实现很多通用协议的 Protocol。你可以在 twisted.protocols.basic 中找到一些稍微简单点的。在你尝试写新 Protocol 时，最好是看看 Twisted 源码是不是已经有现成的存在。如果没有，那实现一个自己的协议是非常好的，正如我们为诗歌下载客户端做的那样。

Protocol Factories

因此每个连接需要一个自己的 Portocol，而且这个 Protocol 是我们自己定义类的实例。由于我们会将创建连接的工作交给 Twisted 来完成，Twisted 需要一种方式来为一个新的连接制定一个合适的协议。制定协议就是 Protocol Factories 的 工作了。

也许你已经猜到了，Protocol Factory 的 API 由 IProtocolFactory 来定义，同样在 interfaces 模块中。Protocol Factory 就是 Factory 模式的一个具体实现。buildProtocol 方法在每次被调用时返回一个新 Protocol 实例。它就是 Twisted 用来为新连接创建新 Protocol 实例的方法。

诗歌下载客户端 2.0：第一滴心血

好吧，让我们来看看由 Twisted 支持的诗歌下载客户端 2.0。源码可以在这里 twisted-client-2/get-poetry.py。你可以像前面一样运行它，并得到相同的输出。这也是最后一个在接收到数据时打印其任务的客户端版本了。到现在为止，对于所有 Twisted 程序都是交替执行任务并处理相对较少数量数据的，应该很清晰了。我们依然通过 print 函数来展示在关键时刻在进行什么内容，但将来客户端不会在这样繁锁。

在第二个版本中，sockets 不会再出现了。我们甚至不需要引入 socket 模块也不用引用 socket 对象和文件描述符。取而代之的是，我们告诉 reactor 来创建到诗歌服务器的连接，代码如下面所示：

```
factory = PoetryClientFactory(len(addresses))

from twisted.internet import reactor

for address in addresses:
    host, port = address
    reactor.connectTCP(host, port, factory)
```

我们需要关注的是 connectTCP 这个函数。前两个参数的含义很明显，不解释了。第三个参数是我们自定义的 PoetryClientFactory 类的实例对象。这是一个专门针对诗歌下载客户端的 Protocol Factory，将它传递给 reactor 可以让 Twisted 为我们创建一个 PeotryProtocol 实例。

值得注意的是，从一开始我们既没有实现 Factory 也没有去实现 Protocol，不像在前面那个客户端中我们去实例化我们 PoetrySocket 类。我们只是继承了 Twisted 在 twisted.internet.protocol 中提供的基类。Factory 的基类是 twisted.internet.protocol.Factory，但我们使用客户端专用（即不像服务器端那样监听一个连接，而是主动创建一个连接）的 ClientFactory 子类来继承。

我们同样利用了 Twisted 的 Factory 已经实现了 buildProtocol 方法这一优势来为我们所用。我们要在子类中调用基类中的实现：

```
def buildProtocol(self, address):
    proto = ClientFactory.buildProtocol(self, address)
    proto.task_num = self.task_num
    self.task_num += 1
    return proto
```

基类怎么会知道我们要创建什么样的 Protocol 呢？注意，我们的 PoetryClientFactory 中有一个 protocol 类变量：

```
class PoetryClientFactory(ClientFactory):

    task_num = 1

    protocol = PoetryProtocol # tell base class what proto to build
```

基类 Factory 的实现 buildProtocol 过程是：安装（创建一个实例）我们设置在 protocol 变量上的 Protocol 类与在这个实例（此处即 PoetryProtocol 的实例）的 factory 属性上设置一个产生它的 Factory 的引用（此处即实例化 PoetryProtocol 的 PoetryClientFactory）。这个过程如图 8 所示：

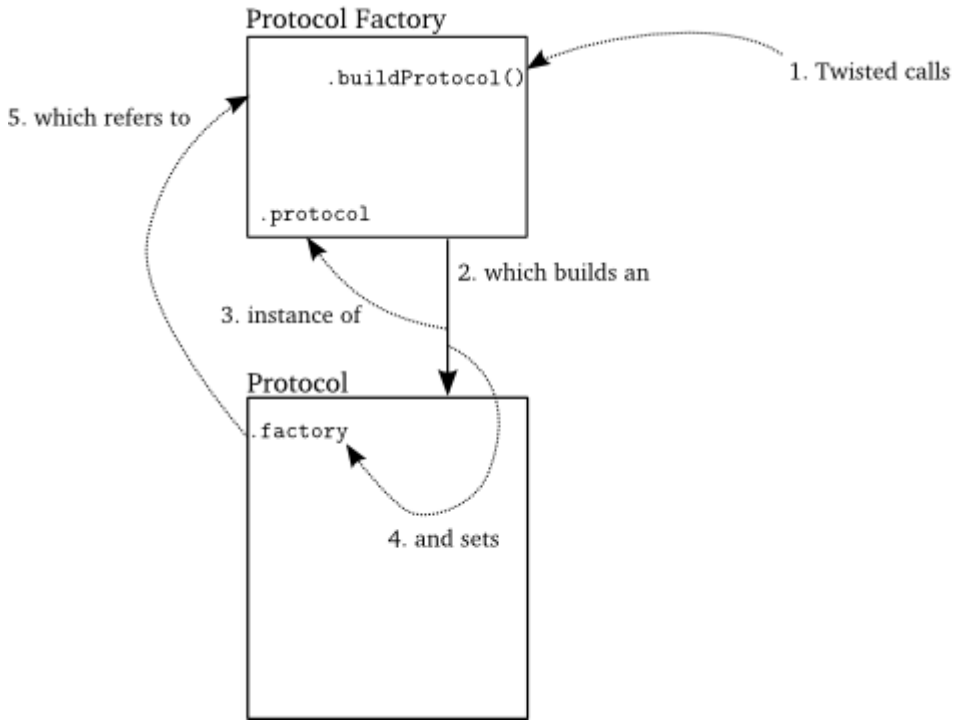


图 8：Protocol 的生成过程

正如我们提到的那样，位于 Protocol 对象内的 factory 属性字段允许在都由同一个 factory 产生的 Protocol 之间共享数据。由于 Factories 都是由用户代码来创建的（即在用户的控制中），因此这个属性也可以实现 Protocol 对象将数据传递回一开始初始化请求的代码中来，这将在第六部分看到。

值得注意的是，虽然在 Protocol 中有一个属性指向生成其的 Protocol Factory，在 Factory 中也有一个变量指向一个 Protocol 类，但通常来说，一个 Factory 可以生成多个 Protocol。

在 Protocol 创立的第二步便是通过 makeConnection 与一个 Transport 联系起来。我们无需自己来实现这个函数而使用 Twisted 提供的默认实现。默认情况是，makeConnection 将 Transport 的一个引用赋给（Protocol 的）transport 属性，同时置（同样是 Protocol 的）connected 属性为 True，正如图 9 描述的一样：

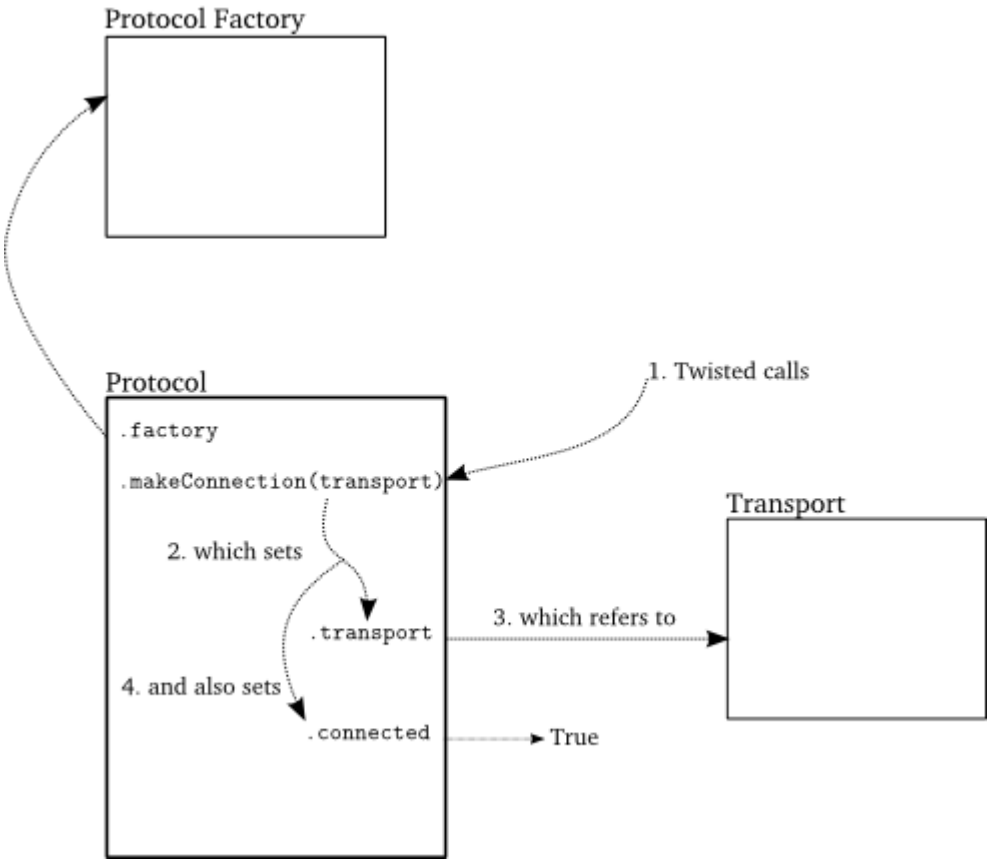


图 9：Protocol 遇到其 Transport

一旦初始化到这一步后，Protocol 开始其真正的工作—将低层的数据流翻译成高层的协议规定格式的消息。处理接收到数据的主要方法是 dataReceived，我们的客户端是这样实现的：

```
def dataReceived(self, data):
    self.poem += data
    msg = 'Task %d: got %d bytes of poetry from %s'
    print    msg % (self.task_num, len(data), self.transport.getHost())
```

每次 dateReceved 被调用就意味着我们得到一个新字符串。由于与异步 I/O 交互，我们不知道能接收到多少数据，因此将接收到的数据缓存下来直到完成一个完整的协议规定格式的消息。在我们的例子中，诗歌只有在连接关闭时才下载完毕，因此我们只是不断地将接收到的数据添加到我们的.poem 属性字段中。

注意我们使用了 Transport 的 getHost 方法来取得数据来自的服务器信息。我们这样做只是与前面的客户端保持一致。相反，我们的代码没有必要这样做，因为我们没有向服务器发送任何消息，也就没有必要知道服务器的信息了。

我们来看一下 dataReceved 运行时的快照。在 2.0 版本相同的目录下有一个 twisted-client-2/get-poetry-stack.py。它与 2.0 版本的不同之处只在于：

```
def dataReceived(self, data):
    traceback.print_stack()
    os._exit(0)
```

这样一改，我们就能打印出跟踪堆栈的信息，然后离开程序，可以用下面的命令来运行它：

```
python twisted-client-2/get-poetry-stack.py 10000
```

你会得到内容如下的跟踪堆栈：

```
File "twisted-client-2/get-poetry-stack.py", line 125, in

    poetry_main()


... # I removed a bunch of lines here


File ".../twisted/internet/tcp.py", line 463, in doRead # Note the doRead callback

    return self.protocol.dataReceived(data)

File "twisted-client-2/get-poetry-stack.py", line 58, in dataReceived
     traceback.print_stack()
```

看见没，有我们在 1.0 版本客户端的 doRead 回调函数。我们前面也提到过，Twisted 在建立新抽象层进会使用已有的实现而不是另起炉灶。因此必然会有一个 IReadDescriptor 的实例在辛苦的工作，它是由 Twisted 代码而非我们自己的代码来实现。如果你表示怀疑，那么就看看 twisted.internet.tcp 中的实现吧。如果你浏览代码会发现，由同一个类实现了 IWriteDescriptor 与 ITransport。因此 IreadDescriptor 实际上就是变相的 Transport 类。可以用图 10 来形象地说明 dateReceived 的回调过程：

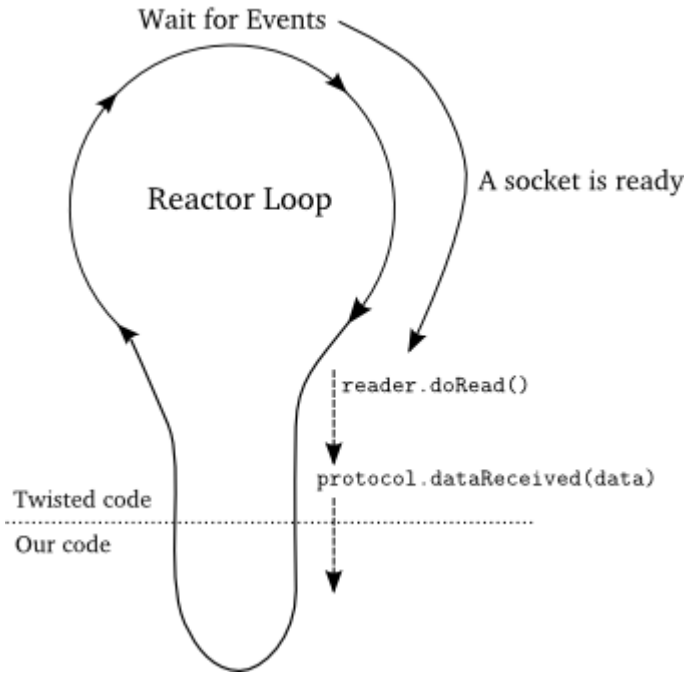


图 10：dateReceived 回调过程

一旦诗歌下载完成，PoetryProtocol 就会通知它的 PooetryClientFactory：

```
def connectionLost(self, reason):


   self.poemReceived(self.poem)

def poemReceived(self, poem):

   self.factory.poem_finished(self.task_num, poem)
```

当 transport 的连接关闭时，conncetionLost 回调会被激活。reason 参数是一个 twisted.python.failure.Failure 的实例对象，其携带的信息能够说明连接是被安全的关闭还是由于出错被关闭的。我们的客户端因认为总是能完整地下载完诗歌而忽略了这一参数。
工厂会在所有的诗歌都下载完毕后关闭 reactor。再次重申：我们代码的工作就是用来下载诗歌-这意味我们的 PoetryClientFactory 缺少复用性。我们将在下一部分修正这一缺陷。值得注意的是，poem_finish 回调函数是如何通过跟踪剩余诗歌数的：

```
...
      self.poetry_count -= 1

      if self.poetry_count == 0:
            …
```

如果我们采用多线程以让每个线程分别下载诗歌，这样我们就必须使用一把锁来管理这段代码以免多个线程在同一时间调用 poem_finish。但是在交互式体系下就不必担心了。由于 reactor 只能一次启用一个回调。
新的客户端实现在处理错误上也比先前的优雅的多，下面是 PoetryClientFactory 处理错误连接的回调实现代码：

```
def clientConnectionFailed(self, connector, reason):
      print 'Failed to connect to:', connector.getDestination()
      self.poem_finished()
```

注意，回调是在工厂内部而不是协议内部实现。由于协议是在连接建立后才创建的，而工厂能够在连接未能成功建立时捕获消息。

结束语：

版本 2 的客户端使用的抽象对于那些 Twisted 高手应该非常熟悉。如果仅仅是为在命令行上打印出下载的诗歌这个功能，那么我们已经完成了。但如果想使我们的代码能够复用，能够被内嵌在一些包含诗歌下载功能并可以做其它事情的大软件中，我们还有许多工作要做，我们将在第六部分讲解相关内容。

第六部分：抽象地利用 Twisted

打造可以复用的诗歌下载客户端

我们在实现客户端上已经花了大量的工作。最新版本的（2.0）客户端使用了 Transports，Protocols 和 Protocol Factories，即整个 Twisted 的网络框架。但仍有大的改进空间。2.0 版本的客户端只能在命令行里下载诗歌。这是因为 PoetryClientFactory 不仅要下载诗歌还要负责在下载完毕后关闭程序。但这对于”PeotryClientFactory“的确是一项分外的工作，因为它除了做好生成一个 PoetryProtocol 的实例和收集下载完毕的诗歌的工作外最好什么也别做。
我需要一种方式来将诗歌传给开始时请求它的函数。在同步程序中我们会声明这样的 API:

```
def get_poetry(host, post):
    """Return a poem from the poetry server at the given host and port."""
```

当然了，我们不能这样做。诗歌在没有全部下载完前上面的程序是需要被阻塞的，否则的话，就无法按照上面的描述那样去工作。但是这是一个交互式的程序，因此对于阻塞在 socket 是不会允许的。我们需要一种方式来告诉调用者何时诗歌下载完毕，无需在诗歌传输过程中将其阻塞。这恰好又是 Twisted 要解决的问题。Twisted 需要告诉我们的代码何时 socket 上可以读写、何时超时等等。我们前面已经看到 Twisted 使用回调机制来解决问题。因此，我们也可以使用回调：

```
def get_poetry(host, port, callback):
    """
    Download a poem from the given host and port and invoke

      callback(poem)

    when the poem is complete.
    """
```

现在我们有一个可以与 Twisted 一起使用的异步 API，剩下的工作就是来实现它了。
前面说过，我们有时会采用非 Twisted 的方式来写我们的程序。这是一次。你会在第七和八部分看到真正的 Twisted 方式（当然，它使用了抽象）。先简单点讲更晚让大家明白其机制。

客户端 3.0

可以在 twisted-client-3/get-poetry.py 看到 3.0 版本。这个版本实现了 get_poetry 方法：

```
def get_poetry(host, port, callback):
    from twisted.internet import reactor
    factory = PoetryClientFactory(callback)
    reactor.connectTCP(host, port, factory)
```

这个版本新的变动就是将一个回调函数传递给了 PoetryClientFactory。这个 Factory 用这个回调来将下载完毕的诗歌传回去。

```
class PoetryClientFactory(ClientFactory):
    protocol = PoetryProtocol

    def __init__(self, callback):
        self.callback = callback

    def poem_finished(self, poem):
        self.callback(poem)
```

值得注意的是，这个版本中的工厂因其不用负责关闭 reactor 而比 2.0 版本的简单多了。它也将处理连接失败的工作除去了，后面我们会改正这一点。PoetryProtocol 无需进行任何变动，我们就直接复用 2.1 版本的：

```
class PoetryProtocol(Protocol):
    poem = ''
    def dataReceived(self, data):
        self.poem += data
    def connectionLost(self, reason):
        self.poemReceived(self.poem)
    def poemReceived(self, poem):
        self.factory.poem_finished(poem)
```

通过这一变动，get_poetry,PoetryClientFactory 与 PoetryProtocol 类都完全可以复用了。它们都仅仅与诗歌下载有关。所有启动与关闭 reactor 的逻辑都在 main 中实现：

```
def poetry_main():
    addresses = parse_args()
    from twisted.internet import reactor
    poems = []
    def got_poem(poem):
        poems.append(poem)
        if len(poems) == len(addresses):
            reactor.stop()
    for address in addresses:
        host, port = address
        get_poetry(host, port, got_poem)
    reactor.run()
    for poem in poems:
        print poem
```

因此，只要我们需要，就可以将这些可复用部分放在任何其它想实现下载诗歌功能的模块中。顺便说一句，当你测试 3.0 版本客户端时，可以重配置诗歌下载服务器来使用诗歌下载的快点。现在客户端下载的速度就不会像前面那样让人”应接不暇“了。

讨论

我们可以用图 11 来形象地展示回调的整个过程：

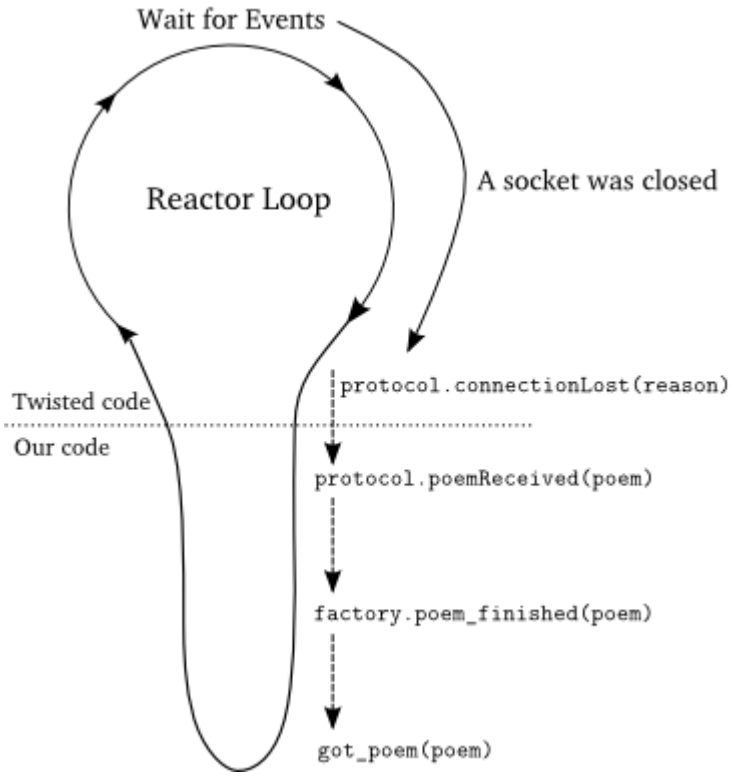


图 10 ：回调过程

图 11 是值得好好思考一下的。到现在为止，我们已经完整描绘了一个一直到向我们的代码发出信号的整个回调链条。但当你用 Twisted 写程序时，或其它交互式的系统时，这些回调中会包含一些我们的代码来回调其它的代码。换句话说，交互式的编程方式不会在我们的代码处止步（Dave 的意思是说，我们的回调函数中可能还会回调其它别人实现的代码，即交互方式不会止步于我们的代码，这个方式会继续深入到框架的代码或其它第三方的代码）。

当你在选择 Twisted 实现你的工程时，务必记住下面这几条。当你作出决定：

I'm going to use Twisted!

即代表你已经作出这样的决定：

我将要构造我的程序如由 reactorz 牵引的一系列的异步回调链

现在也许你还不会像我一样大声地喊出，但它确实是这样的。那就是 Twisted 的工作方式。貌似大部分 Python 程序与 Python 模块都是同步的。如果我们正在写一个同样需要下载诗歌的同步方式的程序，我可能会通过在我们的代码中添加下面几句来实现我们的同步方式的下载诗歌客户端版本：

```
...
import poetrylib # I just made this module name up
poem = poetrylib.get_poetry(host, port)
...
```

然后我们继续。如果我们决定不需要这个这业务那我们可以将这几行代码去掉就 OK 了。如果我们真的要用 Twisted 版本的 get_poetry 来实现同步程序，那么我们需要对异步方式中的回调进行大的改写。这里，我并不想说改写程序不好。而是想说，简单地将同步与异步的程序混合在一直是不行的。

如果你是一个 Twisted 新手或初次接触异步编程，建议你在试图复用其它异步代码时先写点异步 Twisted 的程序。这样你不用去处理因需要考虑各个模块交互关系而带来的复杂情况下，感受一下 Twisted 的运行机制。

如果你的程序原来就是异步方式，那么使用 Twisted 就再好不过了。Twisted 与 pyGTK 和 pyQT 这两个基于 reactor 的 GUI 工具包实现了很好的可交互性。

异常问题的处理

在版本 3.0 中，我们没有去检测与服务器的连接失败的情况，这比在 1.0 版本中出现时带来的麻多得多。如果我们让 3.0 版本的客户端到一个不存在的服务器上下载诗歌，那么不是像 1.0 版本那样立刻程序崩溃掉而是永远处于等待状态中。clientConncetionFailed 回调仍然会被调用，但是因为其在 ClientFactory 基类中什么也没有实现（若子类没有重写基类函数则使用基类的函数）。因此，got_poem 回调将永远不会被激活，这样一来，reactor 也不会停止了。我们已经在第 2 部分也遇到过这样一个不做任何事情的函数了。

因此，我们需要解决这一问题，在哪儿解决呢？连接失败的信息会通过 clientConnectionFailed 函数传递给工厂对象，因此我们就从这个函数入手。但这个工厂是需要设计成可复用的，因此如何合理处理这个错误是依赖于工厂所使用的场景的。在一些应用中，丢失诗歌是很糟糕的;但另外一些应用场景下，我们只是尽量尝试，不行就从其它地方下载 。换句话说，使用 get_poetry 的人需要知道会在何时出现这种问题，而不仅仅是什么情况下会正常运行。在一个同步程序中，get_poetry 可能会抛出一个异常并调用含有 try/excep 表达式的代码来处理异常。但在一个异步交互的程序中，错误信息也必须异步的传递出去。总之，在取得 get_poetry 之前，我们是不会发现连接失败这种错误的。下面是一种可能：

```
def get_poetry(host, port, callback):
    """
    Download a poem from the given host and port and invoke

      callback(poem)

    when the poem is complete. If there is a failure, invoke:
```

```
        callback(None)

    instead.
    """
```

通过检查回调函数的参数来判断我们是否已经完成诗歌下载。这样可能会避免客户端无休止运行下去的情况发生，但这样做仍会带来一些问题。首先，使用 None 来表示失败好像有点牵强。一些异步的 API 可能会将 None 而不是错误状态字作为默认返回值。其次，None 值所携带的信息量太少。它不能告诉我们出的什么错，更不说可以在调试中为我呈现出一个跟踪对象了。好的，也可以尝试这样：

```
def get_poetry(host, port, callback):
    """
    Download a poem from the given host and port and invoke

        callback(poem)

    when the poem is complete. If there is a failure, invoke:

        callback(err)

    instead, where err is an Exception instance.
    """
```

使用 Exception 已经比较接近于我们的异步程序了。现在我们可以通过得到 Exception 来获得相比得到一个 None 多的多的出错信息了。正常情况下，在 Python 中遇到一个异常会得到一个跟踪异常栈以让我们来分析，或是为了日后的调试而打印异常信息日志。跟踪栈相当重要的，因此我们不能因为使用异步编程就将其丢弃。

记住，我们并不想在回调激活时打印跟踪栈，那并不是出问题的地方。我们想得到是 Exception 实例用其被抛出的位置。

Twisted 含有一个抽象类称作 Failure，如果有异常出现的话，其能捕获 Exception 与跟踪栈。Failure 的描述文档说明了如何创建它。将一个 Failure 对象付给回调函数，我们就可以为以后的调试保存跟踪栈的信息了。

在 twisted-failure/failure-examples.py 中有一些使用 Failure 对象的示例代码。它演示了 Failure 是如何从一个抛出的异常中保存跟踪栈信息的，即使在 except 块外部。我不用在创建一个 Failure 上花太多功夫。在第七部分中，我们将看到 Twisted 如何为我们完成这些工作。好了，看看下面这个尝试：

```
def get_poetry(host, port, callback):
    """
    Download a poem from the given host and port and invoke
        callback(poem)
    when the poem is complete. If there is a failure, invoke:
        callback(err)
      instead, where err is a twisted.python.failure.Failure instance.
    """
```

在这个版本中，我们得到了 Exception 和出现问题时的跟踪栈。这已经很不错了！

大多数情况下，到这个就 OK 了，但我们曾经遇到过另外一个问题。使用相同的回调来处理

正常的与不正常的结果是一件莫名奇妙的事。通常情况下，我们在处理失败信息进，相比成功信息要进行不同的操作。在同步 Python 编程中，我们经常在处理失败与成功两种信息上采用不同的处理路径，即 try/except 处理方式：

```
try:
    attempt_to_do_something_with_poetry()
except RhymeSchemeViolation:
    # the code path when things go wrong
else:
    # the code path when things go so, so right baby
```

如果我们想保留这种错误处理方式，那么我们需要独立的代码来处理错误信息。那么在异步方式中，这就意味着一个独立的回调：

```
def get_poetry(host, port, callback, errback):
    """
    Download a poem from the given host and port and invoke
      callback(poem)
    when the poem is complete. If there is a failure, invoke:
      errback(err)
    instead, where err is a twisted.python.failure.Failure instance.
    """
```

版本 3.1

版本 3.1 实现位于 twisted-client-3/get-poetry-1.py。改变是很直观的。PoetryClientFactory，获得了 callback 和 errback 两个回调，并且其中我们实现了 clientConnectFailed：

```
class PoetryClientFactory(ClientFactory):
    protocol = PoetryProtocol
    def __init__(self, callback, errback):
        self.callback = callback
        self.errback = errback
    def poem_finished(self, poem):
        self.callback(poem)
    def clientConnectionFailed(self, connector, reason):
        self.errback(reason)
```

由于 clientConncetFailed 已经收到一个 Failure 对象（其作为 reason 参数）来解释为什么会发生连接失败，我们直接将其交给了 errback 回调函数。直接运行 3.1 版本（无需开启诗歌下载服务）的代码你会得到如下输出：

```
Poem failed: [Failure instance: Traceback (failure with no frames): : Connection was refused by other side: 111: Connection refused.
]
```

这是由 poem_failed 回调中的 print 函数打印出来的。在这个例子中，Twisted 只是简单将一个 Exception 传递给了我们而没有抛出它，因此这里我们并没有看到跟踪栈。因为这并不一

个 Bug，所以跟踪栈也不需要，Twisted 只是想通知我们连接出错。

总结：

我们在第六部分学到：

一、我们为 Twisted 程序写的 API 必须是异步的
二、不能将同步与异步代码混合起来使用
三、我们可以在自己的代码中写回调函数，正如 Twisted 做的那样
四、并且，我们需要写处理错误信息的回调函数

使用 Twisted 时，难道在写我们自己的 API 时都要额外的加上两个参数：正常的回调与出现错误时的回调。幸运的是，Twisted 使用了一种机制来解决了这一问题，我们将在第七部分学习这部分内容。

第七部分：小插曲，Deferred

回调函数的后序发展

在第六部分我们认识这样一个情况：回调是 Twisted 异步编程中的基础。除了与 reactor 交互外，回调可以安插在任何我们写的 Twisted 结构内。因此在使用 Twisted 或其它基于 reactor 的异步编程体系时，都意味需要将我们的代码组织成一系列由 reactor 循环可以激活的回调函数链。
即使一个简单的 get_poetry 函数都需要回调，两个回调函数中一个用于处理正常结果而另一个用于处理错误。作为一个 Twisted 程序员，我们必须充分利用这一点。应该花点时间思考一下如何更好地使用回调及使用过程中会遇到什么困难。
分析下 3.1 版本中的 get_poetry 函数：

```
...
def got_poem(poem):
    print poem
    reactor.stop()
def poem_failed(err):
```

```
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
    reactor.stop()


get_poetry(host, port, got_poem, poem_failed)

reactor.run()
```

我们想法很简单：

1.如果完成诗歌下载，那么就打印它

2.如果没有下载到诗歌，那就打印出错误信息

3.上面任何一种情况出现，都要停止程序继续运行

同步程序中处理上面的情况会采用如下方式：

```
...
try:
    poem = get_poetry(host, port) # the synchronous version of get_poetry
except Exception, err:
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
    sys.exit()
else:
    print poem
    sys.exit()
```

即 callback 类似 else 处理路径，而 errback 类似 except 处理路径。这意味着激活 errback 回调函数类似于同步程序中抛出一个异常，而激活一个 callback 意味着同步程序中的正常执行路径。

两个版本有什么不同之外吗？可以明确的是，在同步版本中， Python 解释器可以确保只要 get_poetry 抛出何种类型的异步都会执行 except 块。即只要我们相信 Python 解释器能够正确的解释执行 Python 程序，那么就可以相信异常处理块会在恰当的时间点被执行。

不异步版本相反的是：poem_failed 错误回调是由我们自己的代码激活并调用的，即 PeotryClientFactory 的 clientConnectFailed 函数。是我们自己而不是 Python 来确保当出错时错误处理代码能够执行。因此我们必须保证通过调用携带 Failure 对象的 errback 来处理任何可能的错误。

否则，我们的程序就会因为等待一个永远不会出现的回调而止步不前。

这里显示出了同步与异步版本的又一个不同之处。如果我们在同步版本中没有使用 try/except 捕获异步，那么 Python 解释器会为我们捕获然后关掉我们的程序并打印出错误信息。但是如果我们忘记抛出我们的异步异常（在本程序中是在 PoetryClientFactory 调用 errback），我们的程序会一直运行下去，还开心地以为什么事都没有呢。

显而易见，在异步程序中处理错误是相当重要的，甚至有些严峻。也可以说在异步程序中处理错误信息比处理正常的信息要重要的多，这是因为错误会以多种方式出现，而正确的结果出现的方式是唯一的。当使用 Twisted 编程时忘记处理异常是一个常犯的错误。

关于上面同步程序代码的另一个默认实事是：else 与 except 块两者只能是运行其中一个（假

设我们的 get_poetry 没有在一个无限循环中运行）。Python 解释器不会突然决定两者都运行或突发奇想来运行 else 块 27 次。对于通过 Python 来实现那样的动作是不可能的。
但在异步程序中，我们要负责 callback 和 errback 的运行。因此，我们可能就会犯这样的错误：同时调用了 callback 与 errback 或激活 callback27 次。这对于使用 get_poetry 的用户来说是不幸的。虽然在描述文档中没有明确地说明，像 try/except 块中的 else 与 except 一样，对于每次调用 get_poetry 时 callback 与 errback 只能运行其中一个，不管是我们是否成功得下载完诗歌。
设想一下，我们在调试某个程序时，我们提出了三次诗歌下载请求，但是得到有 7 次 callback 被激活和 2 次 errback 被激活。可能这时，你会下来检查一下，什么时候 get_poetry 激活了两次 callback 并且还抛出一个错误出来。
从另一个视角来看，两个版本都有代码重复。异步的版本中含有两次 reactor.stop，同步版本中含有两次 sys.exit 调用。我们可以重构同步版本如下：

```
...
try:
    poem = get_poetry(host, port) # the synchronous version of get_poetry
except Exception, err:
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
else:
    print poem

sys.exit()
```

我们可以以同样的方式来重构异步版本吗？说实话，确实不太可能，因为 callback 与 errback 是两个不同的函数。难道要我们回到使用单一回调来实现重构吗？
好下面是我们在讨论使用回调编程时的一些观点：
1.激活 errback 是非常重要的。由于 errback 的功能与 except 块相同，因此用户需要确保它们的存在。他们并不可选项，而是必选项。
2.不在错误的时间点激活回调与在正确的时间点激活回调同等重要。典型的用法是，callback 与 errback 是互斥的即只能运行其中一个。
3.使用回调函数的代码重构起来有些困难。
来下面的部分，我们还会讨论回调，但是已经可以明白为什么 Twisted 引入了 deferred 抽象机制来管理回调了。

Deferred

由于架设在异步程序中大量被使用，并且我们也看到了，正确的使用这一机制需要一些技巧。因此，Twisted 开发者设计了一种抽象机制-Deferred-以让程序员在使用回调时更简便。
一个 Deferred 有一对回调链，一个是为针对正确结果，另一个针对错误结果。新创建的 Deferred 的这两条链是空的。我们可以向两条链里分别添加 callback 与 errback。其后，就可以用正确的结果或异常来激活 Deferred。激活 Deferred 意味着以我们添加的顺序激活 callback 或 errback。图 12 展示了一个拥有 callback/errback 链的 Deferred 对象：

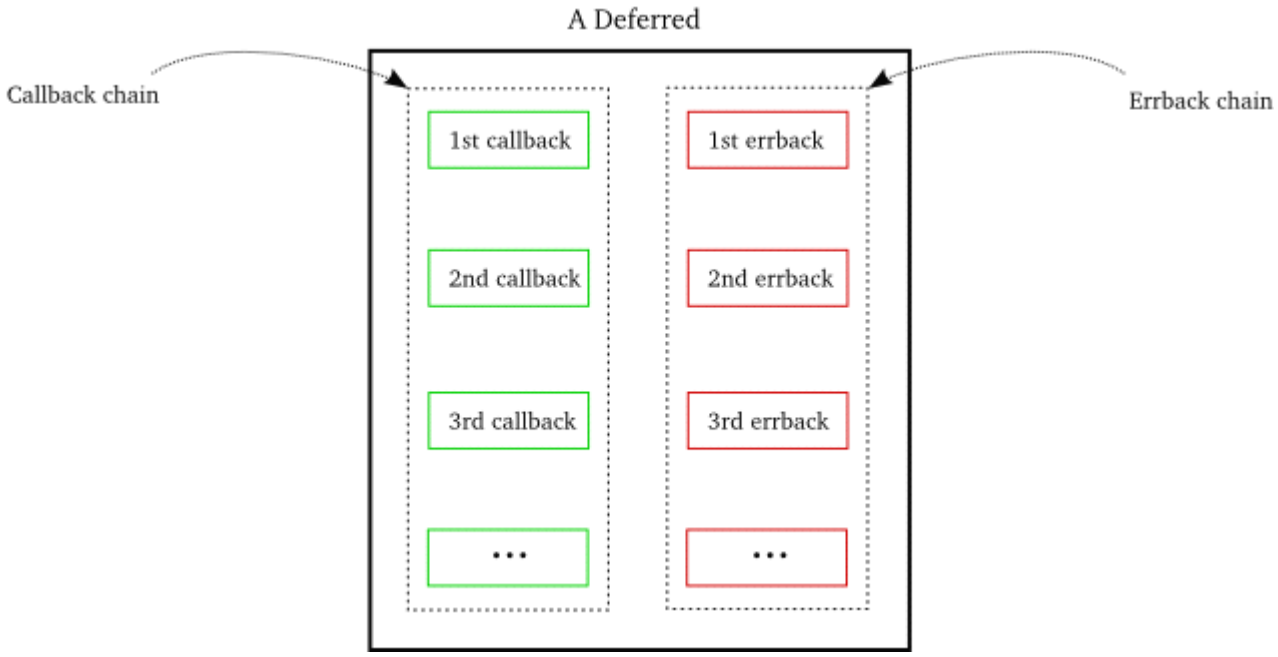


图 12： Deferred

由于 defered 中不使用 reactor，所以我们可以不用在事件循环中使用它。也许你在 Deferred 中发现一个 seTimeout 的函数中使用了 reactor。放心，它将来将来的版本中删掉。
下面是我们第一人使用 deferred 的例子 twisted-deferred/defer-1.py:

```
from twisted.internet.defer import Deferred

def got_poem(res):
    print 'Your poem is served:'
    print res

def poem_failed(err):
    print 'No poetry for you.'

d = Deferred()

# add a callback/errback pair to the chain
d.addCallbacks(got_poem, poem_failed)

# fire the chain with a normal result
d.callback('This poem is short.')

print "Finished"
```

代码开始创建了一个新 deferred，然后使用 addCallbacks 添加了 callback/errback 对，然后使

用 callback 函数激活了其正常结果处理回调链。当然了，由于只含有一个回调函数还算不上链，但不要紧，运行它：
Your poem is served:

This poem is short.

Finished

有几个问题需要注意：
1.正如 3.1 版本中我们使用的 callback/errback 对，添加到 deferred 中的回调函数只携带一个参数，正确的结果或出错信息。其实，deferred 支持回调函数可以有多个参数，但至少得有一个参数并且第一个只能是正确的结果或错误信息。
2.我们向 deferred 添加的是回调函数对
3.callbac 函数携带仅有的一个参数即正确的结果来激活 deferred
4.从打印结果顺序可以看出，激活的 deferred 立即调用了回调。没有任何异步的痕迹。这是因为没有 reactor 参与导致的。
好了，让我们来试试另外一种情况，twisted-deferred/defer-2.py 激活了错误处理回调：

```
from twisted.internet.defer import Deferred
from twisted.python.failure import Failure

def got_poem(res):
    print 'Your poem is served:'
    print res

def poem_failed(err):
    print 'No poetry for you.'

d = Deferred()

# add a callback/errback pair to the chain
d.addCallbacks(got_poem, poem_failed)

# fire the chain with an error result
d.errback(Failure(Exception('I have failed.')))

print "Finished"
```

运行它打印出的结果为：

```
No poetry for you.

Finished
```

激活 errback 链就调用 errback 函数而不是 callback，并且传进的参数也是错误信息。正如上面那样，errback 在 deferred 激活就被调用。
在前面的例子中，我们将一个 Failure 对象传给了 errback。deferred 会将一个 Exception 对象

转换成 Failure，因此我们可以这样写：

```
from twisted.internet.defer import Deferred

def got_poem(res):
    print 'Your poem is served:'
    print res

def poem_failed(err):
    print err.__class__
    print err
    print 'No poetry for you.'

d = Deferred()

# add a callback/errback pair to the chain
d.addCallbacks(got_poem, poem_failed)

# fire the chain with an error result
d.errback(Exception('I have failed.'))
```

运行结果如下：

```
twisted.python.failure.Failure
[Failure instance: Traceback (failure with no frames): : I have failed.
]

No poetry for you.
```

这意味着在使用 deferred 时，我们可以正常地使用 Exception。其中 deferred 会为我们完成向 Failure 的转换。

下面我们来运行下面的代码看看会出现什么结果：

```
from twisted.internet.defer import Deferred
def out(s): print s
d = Deferred()
d.addCallbacks(out, out)
d.callback('First result')
d.callback('Second result')
print 'Finished'
```

输出结果：

```
First result
Traceback (most recent call last):
  ...
twisted.internet.defer.AlreadyCalledError
```

很意外吧，也就是说 deferred 不允许别人激活它两次。这也就解决了上面出现的那个问题：一个激活会导致多个回调同时出现。而 deferred 设计机制控制住了这种可能，如果你非要在

一个 deferred 上要激活多个回调，那么正如上面那样，会报异常错。
那 deferred 能帮助我们重构异步代码吗？考虑下面这个例子：

```
import sys

from twisted.internet.defer import Deferred

def got_poem(poem):
    print poem
    from twisted.internet import reactor
    reactor.stop()

def poem_failed(err):
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'
    from twisted.internet import reactor
    reactor.stop()

d = Deferred()

d.addCallbacks(got_poem, poem_failed)

from twisted.internet import reactor

reactor.callWhenRunning(d.callback, 'Another short poem.')

reactor.run()
```

这基本上与我们上面的代码相同，唯一不同的是加进了 reactor。我们在启动 reactor 后调用了 callWhenRunning 函数来激活 deferred。我们利用了 callWhenRunning 函数可以接收一个额外的参数给回调函数。多数 Twisted 的 API 都以这样的方式注册回调函数，包括向 deferred 添加 callback 的 API。下面我们给 deferred 回调链添加第二个回调：

```
import sys

from twisted.internet.defer import Deferred

def got_poem(poem):
    print poem

def poem_failed(err):
    print >>sys.stderr, 'poem download failed'
    print >>sys.stderr, 'I am terribly sorry'
    print >>sys.stderr, 'try again later?'

def poem_done(_):
```

```
    from twisted.internet import reactor
    reactor.stop()

d = Deferred()

d.addCallbacks(got_poem, poem_failed)
d.addBoth(poem_done)

from twisted.internet import reactor

reactor.callWhenRunning(d.callback, 'Another short poem.')

reactor.run()
```

addBoth 函数向 callback 与 errback 链中添加了相同的回调函数。在这种方式下，deferred 有可能也会执行 errback 链中的回调。这将在下面的部分讨论，只要记住后面我们还会深入讨论 deferred。

总结：

在这部分我们分析了回调编程与其中潜藏的问题。我们也认识到了 deferred 是如何帮我们解决这些问题的：

1.我们不能忽视 errback，在任何异步编程的 API 中都需要它。Deferred 支持 errbacks。

2.激活回调多次可能会导致很严重的问题。Deferred 只能被激活一次，这就类似于同步编程中的 try/except 的处理方法。

3.含有回调的程序在重构时相当困难。有了 deferred，我们就通过修改回调链来重构程序。

关于 deferred 的故事还没有结束，后面还有大量的细节来讲。但对于使用它来重构我们的客户端已经够用的了，在第八部分将讲述这部分内容。

第八部分：使用 **Deferred** 的诗歌下载客户端

客户端 4.0

我们已经对 deferreds 有些理解了，现在我们可以使用它重写我们的客户端。你可以在 twisted-client-4/get-poetry.py 中看到它的实现。
这里的 get_poetry 已经再也不需要 callback 与 errback 参数了。相反，返回了一个用户可能根据需要添加 callbacks 和 errbacks 的新 deferred。

```
def get_poetry(host, port):
    """
    Download a poem from the given host and port. This function
    returns a Deferred which will be fired with the complete text of
    the poem or a Failure if the poem could not be downloaded.
    """
    d = defer.Deferred()
    from twisted.internet import reactor
    factory = PoetryClientFactory(d)
    reactor.connectTCP(host, port, factory)
    return d
```

这里的工厂使用一个 deferred 而不 callback/errback 对来初始化。一旦我们获取到 poem 后或者没有连接到服务器上，deferred 就会以返回一首诗歌或一个 failure 的被激活。

```
class PoetryClientFactory(ClientFactory):

protocol = PoetryProtocol

def __init__(self, deferred):

self.deferred = deferred

def poem_finished(self, poem):
if self.deferred is not None：

d, self.deferred = self.deferred, None

d.callback(poem)

def clientConnectionFailed(self, connector, reason)：

  if self.deferred is not None：
d, self.deferred = self.deferred, None

d.errback(reason)
```

注意我们在 deferred 被激活后是如何销毁其引用的。这种方式普便存在于 Twisted 的源代码中，这样做可以保证我们不会激活一个 deferred 两次。这也为 Python 的垃圾回收带来的方便。
这里仍然不用去改变 poetryProtocol。我们只需要更新 poetry_main 函数即可：

```
def poetry_main():

addresses = parse_args()

from twisted.internet import reactor

poems = []

errors = []

  def got_poem(poem):

poems.append(poem)

     def poem_failed(err):

print >>sys.stderr, 'Poem failed:', err

errors.append(err)

     def poem_done(_):

if len(poems) + len(errors) == len(addresses):

reactor.stop()

     for address in addresses:

host, port = address

d = get_poetry(host, port)

d.addCallbacks(got_poem, poem_failed)

d.addBoth(poem_done)

   reactor.run()

     for poem in poems:
```

print poem

注意我们是如何利用 deferred 的回调链在不考虑两个主要的 callback 与 errback 回调外，重构 poem_done 调用的。
由于 deferred 在 Twisted 大量被使用，使用小写字母 d 来表示当前正在工作中的 deferred 已经成为惯例。

讨论

新版本的客户端与我们前面的同步版本的客户端一样，get_poetry 得到的参数都是诗歌下载服务器的地址。同步版本返回的是诗歌内容，而异步版本返回的却是一个 deferred。返回一个 deferred 是 Twisted 的 APIs 或用 Twisted 写的程序常见的，这样一来我们可以这样来理解 deferred：
一个 Deferred 代表了一个“异步的结果”或者“结果还没有到来”
在图 13 中可以更加清晰地表达出两者之间的不同：

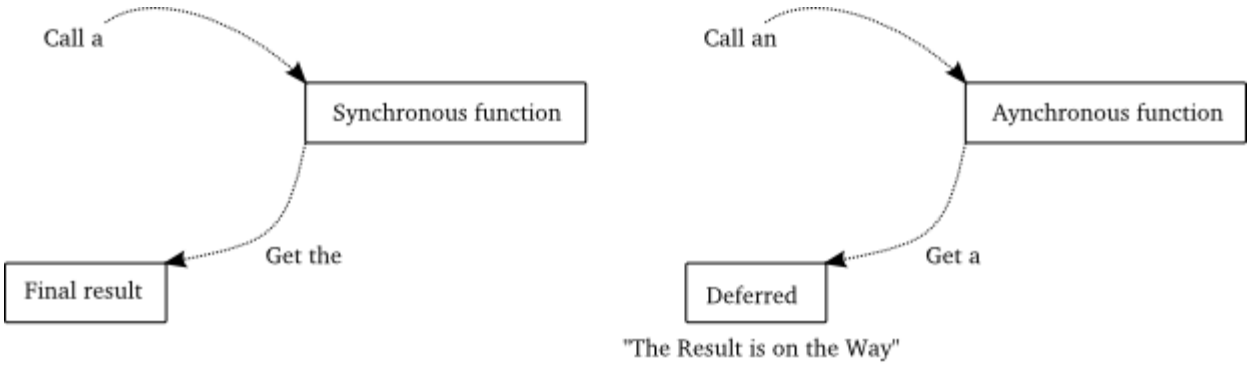


图 13：同步 VS 异步
异步函数返回一个 deferred，对用户意味着：
我是一个异步函数。不管你想要什么，可能现在马上都得不到。但当结果来到时，我会激活这个 deferred 的 callback 链并返回结果。或者当出错时，相应地激活 errback 链并返回出错信息。
当然，这个函数是不能随意激活这个 deferred 的，因为它已经返回了。但这个函数已经启动了一系列事件，这些事件最终将会激活这个 deferred。
因此，deferred 是为适应异步模式的一种延迟函数返回的方式。函数返回一个 deferred 意味着其是异步的，代表着将来的结果，也是对将来能够返回结果的一种承诺。
同步函数也能返回一个 deferred，因此严格点说，返回 deferred 只说可能是异步的。我们会在将来的例子中看到同步函数返回 deferred。
由于 deferred 的行为已经很好的定义与理解，因此在实现自己的 API 时返回一个 deferred 更容易让其它的 Twisted 程序理解你的代码。如果没有 deferred，可能每个人写的模块都使用不同的方式来处理回调。这要一来就增加了相互理解的工作量。

当你使用 Deferred 时，你仍然在使用回调，它们仍然由 reactor 来调用。
当首次学习 Twisted 时，经常犯的一个错误就是：会给 deferred 增加一些它本身不能实现的功能。尤其是：经常假设在 deferred 上添加一个函数就可以使其变成异步函数。这可能会让你产生这样的想法：在 Twisted 中可以通过将 os.system 的函数添加到 deferred 的回调链中。我认为，这可能是没有弄清楚异步编程的原因才产生这样的想法。由于 Twisted 代码使用了大量的 deferred 但却很少会涉及到 reactor，可能会认为 deferred 做了大部分工作。如果你是从开始阅读这个系列的，你就会知道事情远不是这样。虽然 Twisted 是由众多部分组合在一起来工作的，但实现异步的主要工作都是由 reactor 来完成的。Deferred 是一个很好的抽象概念，但前面几个例子中的客户端我们却没有使用它，而 reactor 却都用到了。
来看看我们第一个回调激活时的跟踪栈信息。运行 twisted-client-4/get-poetry-stack.py 让其连接你打开的服务器：

```
File "twisted-client-4/get-poetry-stack.py", line 129, in

   poetry_main()

File "twisted-client-4/get-poetry-stack.py", line 122, in poetry_main

   reactor.run()
  ... # some more Twisted function calls

   protocol.connectionLost(reason)

File "twisted-client-4/get-poetry-stack.py", line 59, in connectionLost

   self.poemReceived(self.poem)

File "twisted-client-4/get-poetry-stack.py", line 62, in poemReceived

   self.factory.poem_finished(poem)

File "twisted-client-4/get-poetry-stack.py", line 75, in poem_finished

   d.callback(poem) # here's where we fire the deferred
```

... # some more methods on Deferreds

File "twisted-client-4/get-poetry-stack.py", line 105, in got_poem

traceback.print_stack()

这很像版本 2.0 的跟踪栈，图 14 可以很好地说明具体的调用关系：

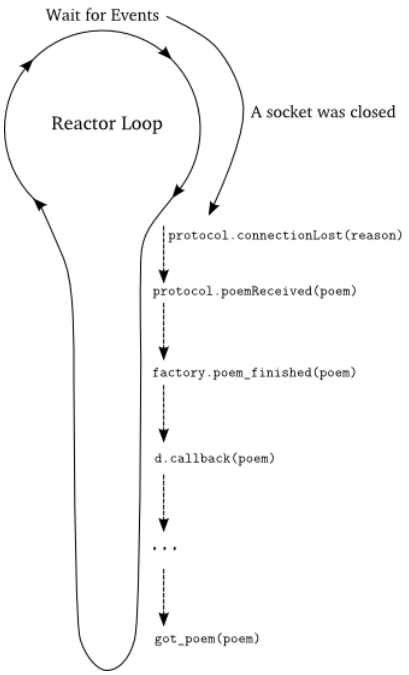


图 14 deferred 的回调

这很类似于我们前面的 Twisted 客户端，虽然这张图的调用关系并不清晰而会你摸不着头脑。但我们先不深入分析这张图。有一个细节并没有在这张图上反映出来：callback 链直到第二个回调 poem_done 激活前才将控制权还给 reactor。

通过使用 deferred，我们在由 Twisted 中的 reactor 启动的回调中加入了一些自己的东西，但我们并没有改变异步程序的基础架构。回忆下回调编程的特点：

1.在一个时刻，只会有一个回调在运行

2.当 reactor 运行时，那我们自己的代码则得不到运行

3,反之则反之

4.如果我们的回调函数发生阻塞，那么整个程序就跟着阻塞掉了

在一个 deferred 上追加一个回调并不会改变上面这些实事。尤其是，第 4 条。因此当一个 deferred 激活时被阻塞，那么整个 Twisted 就会陷入阻塞中。因此我们会得到如下结论：Deferred 只是解决回调函数管理问题的一种解决方案。它并不一种替代回调方式也不能将阻塞式的回调变成非阻塞式回调的。

我通过构建一个添加阻塞式回调的 deferred 来验证最后一点。验证代码文件为 twisted-deferred/defer-block.py。第二个 callback 通过使用 time.sleep 来达到阻塞的效果。如果你运行该代码来观察打印信息顺序时，你会发现 deferred 中阻塞回调仍然会阻塞掉。

aaa 总结
函数通过返回一个 Deferred，向使用者暗示“我是采用异步方式的”并且当结果到来时会使用一种特殊的机制（在此处添加你的 callback 与 errback）来获得返回结果。Defered 被广泛地运用在 Twisted 的每个角落，当你浏览 Twisted 源码时你就会不停地遇到它。
4.0 版本客户端是第一个使用 Deferred 的 Twisted 版的客户端，其使用方法为在其异步函数中返回一个 deferred 来。可以使用一些 Twisted 的 APIs 来使客户端的实现更加清晰些，但我觉得它能够很好地体现出一个简单的 Twisted 程序是怎么写的了，至少对于客户端可以如此肯定。事实上，后面我们会重构我们的服务器端。
但我们对 Deferred 的讲解还没有结束。使用如此少量的代码，Deferred 就能提供如此之多的功能。我们将在第 9 部分探讨其更多的功能和功能背后的动机。

第九部分：第二个小插曲，Deferred

更多关于回调的知识

稍微停下来再思考一下回调的机制。尽管对于以 Twisted 方式使用 Deferred 写一个简单的异步程序已经非常了解了，但 Deferred 提供更多的是只有在比较复杂环境下才会用到的功能。因此，下面我们自己想出一些复杂的环境，以此来观察当使用回调编程时会遇到哪些问题。然后，再来看看 deferred 是如何解决这些问题的。
因此，我们为诗歌下载客户端添加了一个假想的功能。设想一些计算机科学家发明了一种新诗歌关联算法，
Byronification 引擎。这个漂亮的算法根据一首诗歌生成一首使用 Lord Byron 式的同样的诗歌。另外，专家们提供了其 Python 的接口，即：

```
class IByronificationEngine(Interface):
    def byronificate(poem):
        """
        Return a new poem like the original, but in the style of Lord Byron.
        Raises GibberishError if the input is not a genuine poem.
        """
```

像大多数高尖端的软件一样，其实现都存在着许多 bugs。这意外着除了已知的异常外，这个 byronificate 方法可能会抛出一些专家当时没有预料到的异常出来。
我们还可以假设这个引擎能够非常快的动作以至于我们可以在主线程中调用到而无需考虑使用 reactor。下面是我们想让程序实现的效果：

1.尝试下载诗歌
2.如果下载失败，告诉用户没有得到诗歌
3.如果下载到诗歌，则转交给 Byronificate 处理引擎一份
4.如果引擎抛出 GibberishError，告诉用户没有得到诗歌
5.如果引擎抛出其它异常，则将原始式样的诗歌立给用户
6.如果我们得到这首诗歌，则打印它
7.结束程序
这里设计是当遇到 GibberishError 异常则表示没有得到诗歌，因此我们直接告诉用户下载失败即可。这也许对调试没什么用处，但我们的用户关心的只是我们下载到诗歌没有。另一方面，如果引擎因为一些其它的原因而出现处理失败，那么我们将原始诗歌交给用户。毕竟，有诗歌呈现总比没有好，虽然不是用户想要的 Byron 样式。

下面是同步模式的代码：

```
try:
    poem = get_poetry(host, port) # synchronous get_poetry
except:
    print >>sys.stderr, 'The poem download failed.'
else:
    try:
        poem = engine.byronificate(poem)
    except GibberishError:
        print >>sys.stderr, 'The poem download failed.'
    except:
        print poem # handle other exceptions by using the original poem
    else:
        print poem
sys.exit()
```

这段代码可能经过一些重构会更加简单，但已经足以说明上面的逻辑流程。我们想升级那些最近使用 deferred 的客户端来使用这个功能。但这部分内容我准备把它放在第十部分。现在，我们来考虑一下，用版本 3.1 来实现这个功能，最后一个没有使用 deferred 的客户端。假设我们无需考虑处理异常，那么只是改变一下 got_poem 回调即可：

```
def got_poem(poem):
    poems.append(byron_engine.byronificate(poem))
    poem_done()
```

那么如果 byronificate 抛出 GibberishError 异常或其它异常会发生什么呢？看看第六部分的图 11,我们可以得到：
1.这个异常会传播到工厂中的 poem_finished 回调，即激活 got_poem 的方法
2.由于 poem_finished 并没有捕获这个异常，因此其会传递到 protocol 中的 poemReceive 函数
3.然后来到 connectionLost 函数，仍然在 protocol 中
4.然后就来到 Twisted 的核心区，最后止步于 reactor。
前面已经了解到，reactor 会捕获异常并记录它而不是“崩溃”掉。但它却不会告诉用户我们的诗歌下载失败的消息。reactor 并不知道任何诗歌或 GibberishErrors 的信息，它只是一段被设计成适应所有网络类型的通用代码，即便与诗歌无关的网络服务 。（Dave 这里想强调的是

reactor 只是做一些具有普遍意义的事情，不会单独去处理特定的问题，例如这里原 GibberishErrors 异常）
注意异常是如何顺着调用链传递到具有通用性代码区域。并且看到，在 got_poem 后面任何一步都没有可望以我们客户端的具体要求来处理异常的。这与同步代码中的方式恰恰相反。图 15 揭示了一个同步客户端的调用栈：

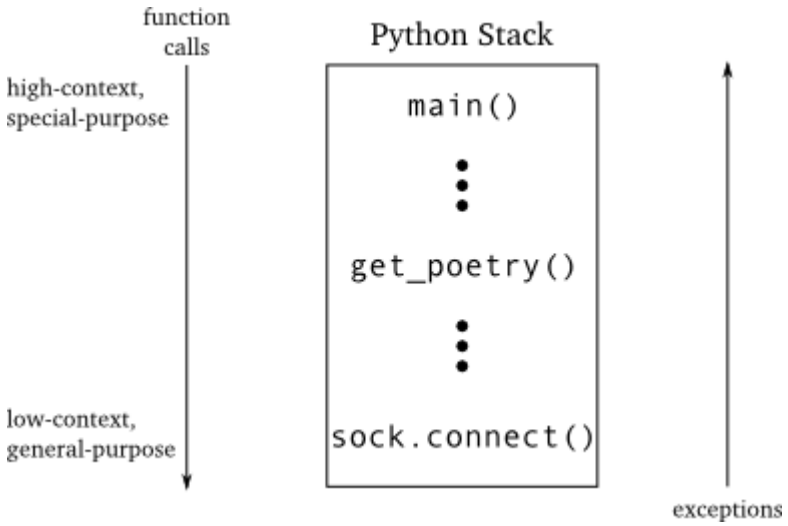


图 15：同步调用栈
main 函数是最高层，意味着它可以触及整个程序，它为什么要存在，并且它是如何在整体上表现的。典型的，main 函数可以触及到用户在命令行输入想让程序做什么的参数。并且它还有一个特殊的目的：为一个命令行式的客户端打印结果。
socket 的 connet 函数，恰恰相反，其为最低层。它所知道的就是提供到指定地址的连接。它并不知道另一端是什么及我们为什么要进行连接。但 connect 有通用性，不管你因为何种服务要进行网络连接都可以使用它。
get_poetry 在中间，它知道要取一些诗歌，但并不知道如果得不到诗歌会发生什么。因此，从 connect 抛出的异常会向上传递，从低层的具有通用性的代码区到高层的具有针对性的代码区，直到其传递到知道如何处理这个异常的代码区。
现在，我们再回来看看对 3.1 版的假想功能的实现。我们在图 16 里对调用栈进行了分析，当然只是说明了其中关键的函数：

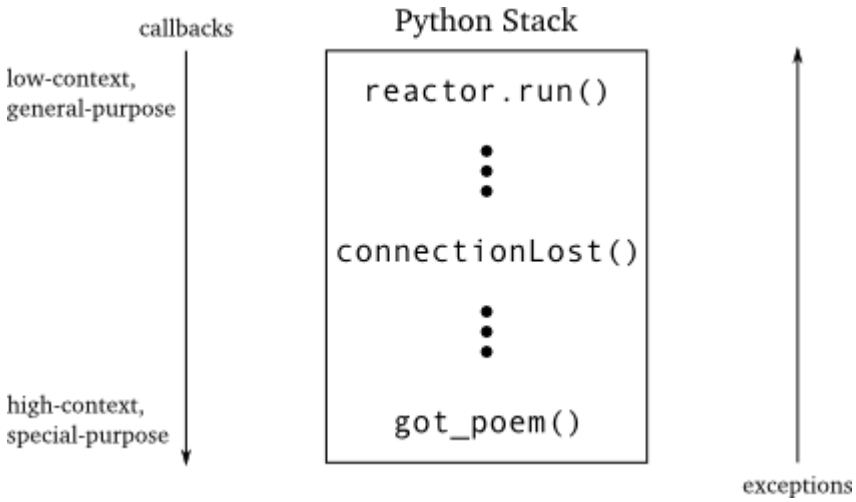


图 16 异步调用栈

现在问题非常清晰了：在回调中，低层的代码（reactor）调用高层的代码，其甚至还会调用更高层的代码。因此一旦出现了异常，它并不会立即被其附件（在调用栈中可触及）的代码捕获，当然附近的代码也不可能处理它。由于异常每向上传递一次，就越靠近低层那些更加不知如何处理该异常的代码。

一旦异常来到 Twisted 的核心代码区，游戏也就结束了。异常并不会被处理，只是被记录下来。因此我们在以最原始的回调方式使用回调时（不使用 deferred），必须在其进入 Twisted 之间很好地处理各种异常，至少是我们知道的那些在我们自己设定的规则下会产生的异常。当然其也应该包括那些由我们自己的 BUG 产生的异常。

由于 bug 可能存在于我们代码中的每个角落，因此我们必须将每个回调都放入 try/except 中，这样一来所有的异常都才有可能被捕获。这对于我们的 errback 同样适用，因为 errback 中也可能含有 bugs。

Deferred 的优秀架构

最终还得由 Deferred 来帮我们解决这类问题。当一个 deferred 激活了一个 callback 或 errback 时，它就会捕获各种由回调抛出的异常。换句话说，deferred 扮演了 try/except 模块，这样一来，只要我们使用 deferred 就无需自己来实现这一层了。那 deferred 是如何解决这个问题的？很简单，它传递异常给在其链上的下一个 errback。

我们添加到 deferred 中的第一个 errback 回调来处理任何出错信息，信息是在 deferred 的 errback 函数调用时发出的。但第二个 errback 会处理任何由第一个 errback 或第一个 callback 抛出的异常，并一直按这种规则传递下去。

回忆下图 12.我们假设第一对 callback/errback 是 stage0,下面则是 stage1，stage2。。。依次类推。对于 stage N 来说，如果其 callback 或 errback 出错，那么 stage N+1 的 errback 就会被调用并收到一个 Failure 对象作为参数，同时 stage N+1 的 callback 就不会被调用了。

通过将回调函数产生的异常向在链中传递，deferred 将异常抛向了高层代码。这也意味着调用 deferred 的 callback 与 errback 永远不会在调用都本身处引发异常（只要你仅激活 deferred 一次），因此，底层的代码可以放心的激活 deferred 而无需担心会引发异常。相反，高层代码通过向 deferred 中添加 errback（使用 addErrback）来捕获异常。

在同步代码中，异常会在其被捕获而停止传递，那么一个 errback 如何发出其捕获了异常这一信号呢？同样很简单：不再引发异常。这样一来，执行权就转移到了 callback 中来。因此对于 stage N 来说，不管是 callback 还是 errback 成功执行而没有抛出异常，那么 stage N+1 的 callback 就会被调用，同样，stage N+1 的 errback 就不会被调用了。

我们来总结一下吧：

1.一个 deferred 有一个 callback/errback 对链，它们以添加到 deferred 中的顺序依次排列

2.stage 0，即第一对 errback/callbac，会在 deferred 激活时调用，具体调用那个看激活 deferred 的方式，若是通过.errback 激活，则调用 errback；同样若是通过.callback 激活则调用 callback。（这里的 errback/callback 实际是指通过 addBoth 添加的函数）

3.如果 stage N 执行出现异常，则 stage N+1 的 errback 被调用，并且其参数即为 stage N 出现的异常

4.同样，如果 stage N 成功，即没有抛出异常，则 N+1 的 callback 被调用，其第一个参数为 stage N 的返回值。

图 17 更加直观的描述上述操作：

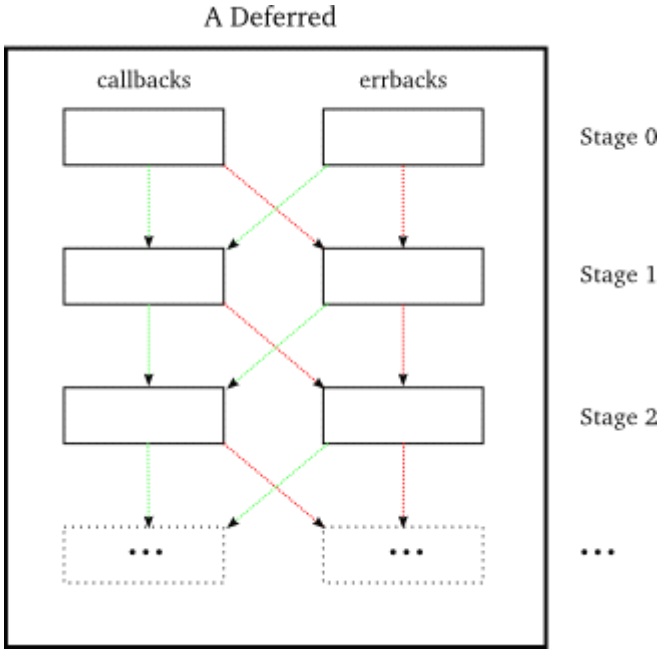


图 17：deferred 中的控制流程

绿色的线表示 callback 和 errback 成功执行没抛出异常，而红线表示出现了异常。这些线不仅说明了控制流程还说明了异常与返回值在链中流动的情况。图 17 显示了所有 deferred 能出现的可能路径，但实际只有一条路径会存在。图 18 显示了一条可能的路径：

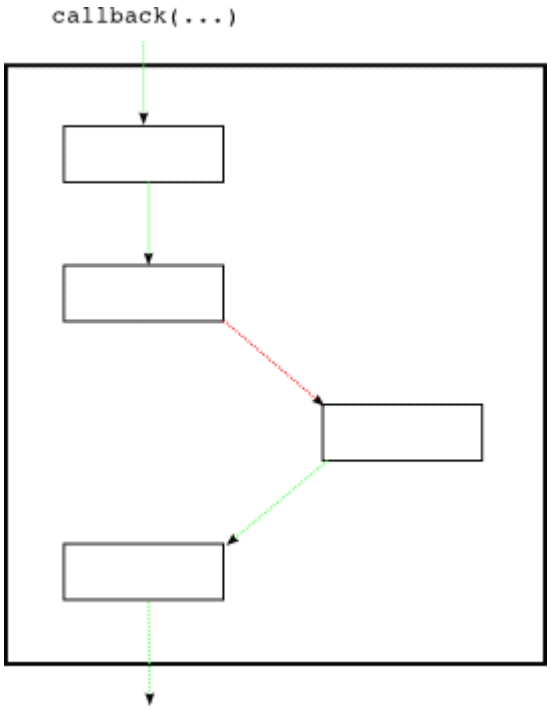


图 18：可能的 deferred 激活路线

图 18 中，deferred 的.callback 函数被调用了，因此激活了 stage 0 的 callback。这个 callback 成功的执行而没有抛出异常，因此控制权传给了 stage 1 的 callback。但这个 callbac 执行失败而抛出异常，因此控制权传给了 stage 2 的 errback。errback 成功的处理了异常，而没有再抛出异常，因此控制权传给了 stage 3 的 callback，并且将 errback 的返回值作为第一个参数传了进来（即 stage 3 的 callback 中）。

图 18 中，可以看出，最后一个 stage 上的所有的回调出现异常时，都由下一层的 errback 来捕获并处理，但如果最后一个 stage 的 callback 或 errback 执行失败而抛出异常，怎么办呢？那么这个异常就会成为 unhandled（未处理）。

在同步代码中，未处理的异常会导致解释器崩溃，在原始方式使用回调的代码中未处理异常会由 reactor 捕获并记录下来。那么未处理异常出现在 deferred 中会怎样呢？让我们来做个试验。运行 twisted-deferred/defer-unhandled.py 试试。下面是输出：

Finished

Unhandled error in Deferred:
Traceback (most recent call last):

...

--- <exception caught here> ---

...

exceptions.Exception: oops

如下几点需要引起我们的注意：
1.最后一个 print 函数成功执行，意味着程序并没有因为出现未处理异常而崩溃。
2.其只是将跟踪栈打印出来，而没有宕掉解释器
3.跟踪栈的内容告诉我们 deferred 在何处捕获了异常
4.“’Unhandle”的字符在“Finished”之后出现。
之所以出现第 4 条是因为，这个消息只有在 deferred 被垃圾回收时才会打印出来。我们将在下面的部分看到其中的原因。
在同步代码中，我们可以使用 raise 来重新抛出一个异常而无需其它参数。同样，我们也可以在 errback 中这样做。deferred 通过以下两点来判断 callback/errback 是否执行成功：
1.callback/errback “raise”一个异常，或
2.callbakc/errback 返回一个 Failure 对象
因为 errback 的第一个参数就是一个 Failure，因此一个 errback 可以在进行完其处理后可以再次抛出这个 Failure。

Callbacks 与 Errbacks，成对出现

上面讨论内容中的一个问题必须要清楚：你添加 callback 与 errback 到一个 defered 的顺序会决定这个 deferred 的的整体运行情况。另一个必须搞清楚的是：在一个 deferred 中 callback 与 errback 往往是成对出现。有四个方法可以向一个 deferred 的回调链中添加 callback/errback 对：
addCallbacks
addCallback
addErrback
addBoth
很明显的是，第一个与第四个是向链中添加函数对。当然中间两个也向链中添加函数对。AddCallback 向链中添加一个显式的 callback 函数与一个隐式的”pass-through“函数（实在想不出一个对应的词）。一个 pass-through 函数只是虚设的函数，只将其第一个参数返回。由于 errback 回调函数的第一个参数是 Failure，因此一个“path-through”的 errback 总是执行“失败”，即将异常传给下个 errback 回调。

deferred 模拟器

这部分内容，没有译。其主要是帮助理解 deferred，但你会发现，读其中的代码，根本更好的理解 deferred。主要是我还没有理解，嘿嘿。所以就不知为不知吧。

aaa 总结
经过这些对回调的考虑，发现由于回调式编程改变了低层代码与高层代码的关系，因此让回调产生的异常直接抛到栈中并不件好事。Deferred 通过将异常捕获然后将其顺着回调链传递

来解决了这个问题。
我们同样意识到，原始数据（返回值）在链中被传递。结合这个两事实也就带来了这样一种场景：根据每个 stage 收到的结果的不同，deferred 在 callback 与 errback 链中来回交错传递数据并执行。
我们将在第十部分使用些学到的知识来更新我们的客户端。

第十部分：增强 defer 功能的客户端

版本 5.0

现在我们将要向诗歌下载客户端添加一些新的处理逻辑，包括在第九部分提到要添加的功能。不过，首先我要说明一点：我并不知道如何实现 Byronification 引擎。那超出了我的编程能力范围。取而代之的，我想实现一个简单的功能，即 Cummingsifier。其只是将诗歌内容转换成小写字母：

```
def cummingsify(poem)
    return poem.lower()
```

这个方法如此之简单以至于它永远不会出错。版本 5.0 的实现代码在 twisted-client-5/get-poetry.py 文件中。我们使用了修改后的 cummingsify，其会随机地选择以下行为：
1.返回诗歌的小写版本
2.抛出一个 GibberishError 异常
3.抛出一个 ValueError
这样，我们便模拟出来一个会因为各种意料不到的问题而执行失败的复杂算法。其它部分的仅有的改变在方法 poetry_main 中：

```
def poetry_main():
    addresses = parse_args()
    from twisted.internet import reactor
    poems = []
    errors = []
    def try_to_cummingsify(poem):
        try:
            return cummingsify(poem)
        except GibberishError:
            raise
        except:
            print 'Cummingsify failed!'
            return poem
    def got_poem(poem):
```

```
        print poem
        poems.append(poem)
    def poem_failed(err):
        print >>sys.stderr, 'The poem download failed.'
        errors.append(err)
    def poem_done(_):
        if len(poems) + len(errors) == len(addresses):
            reactor.stop()
    for address in addresses:
        host, port = address
        d = get_poetry(host, port)
        d.addCallback(try_to_cummingsify)
        d.addCallbacks(got_poem, poem_failed)
        d.addBoth(poem_done)
    reactor.run()
```

因此，当从服务器上下载一首诗歌时，可能会出现如下情况：

1.打印诗歌的小写版本

2.打印 ”Cummingsify failed“并附上原始形式的诗歌

3.打印”The peom download failed”。

为了实现下面内容的效果，你可以打开多个服务器或开一个服务器而打开此程序次，直到你观察到所有不同的结果，当然也尝试一下去连接一个没有服务器值守的端口。

图 19 是我们给 deferred 添加回调后形成的 callback/errback 链：

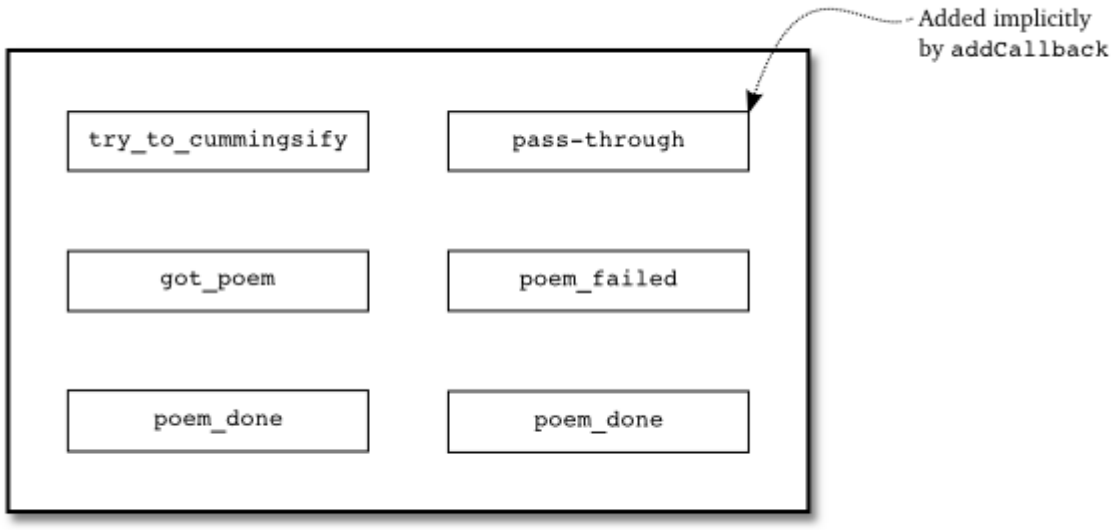


图 19 deferred 中的回调链

注意到，"pass-throug”errback 通过 addCallback 添加到链中。它会将任何其接收到的 Failure 传递给下一个 errback（即 poem_failed 函数）。因此 poem_failed 函数可以处理来自 get_poetry

与 try_to_commingsify 两者的 failure。下面让我们来分析下 deferred 可能会出现的激活情况，图 20 说明了我们能够下载到诗歌并且 try_to_commingsify 成功执行的路线图：

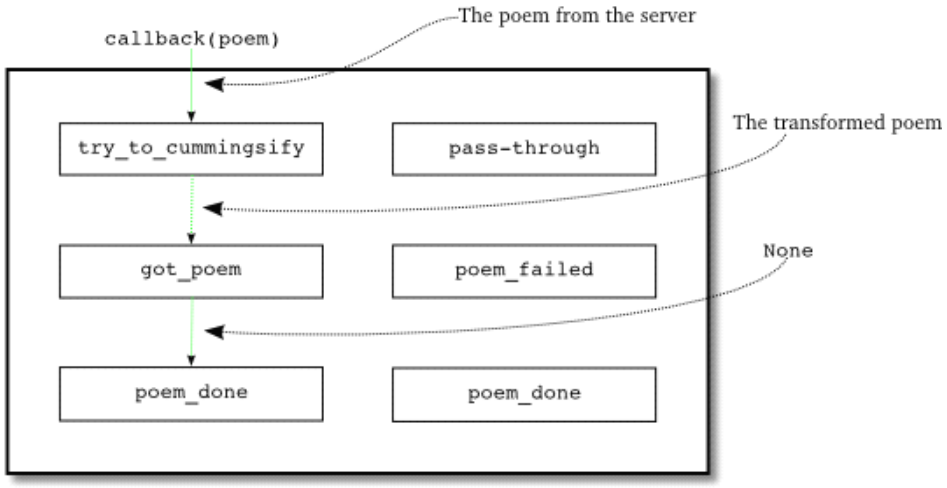


图 20 成功下载到诗歌并且成功变换其格式

在这种情况中，没有回调执行失败，因此控制权一直在 callback 中流动。注意到 poem_done 收到的结果是 None，这是因为它并没有返回任何值。如果我们想让后续的回调都能触及到诗歌内容，只要显式地让 got_poem 返回诗歌即可。

图 21 说明了我们在成功下载到诗歌后，但在 try_to_cummingsify 中抛出了 GibberishError：

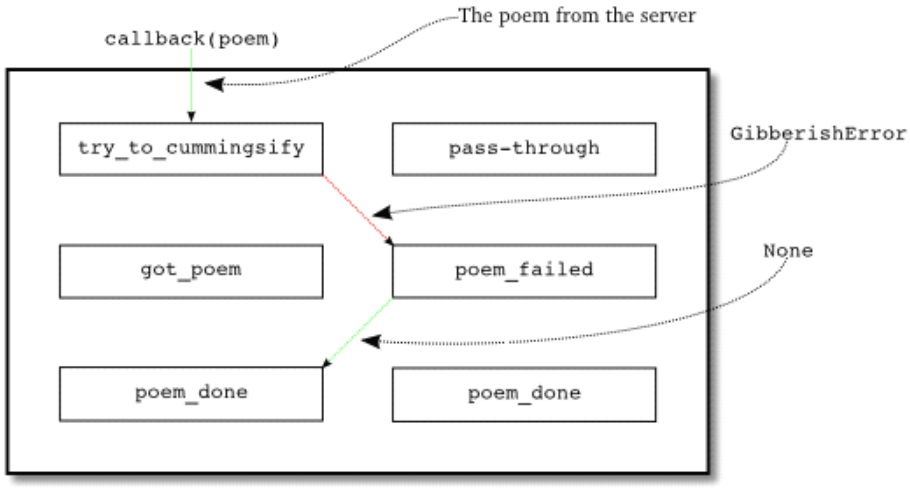


图 21 成功下载到诗歌但出现了 GibberishError

由于 try_to_cummingsify 回调抛出了 GibberishError，所以控制权转移到了 errback 链，即 poem_fail 回调被调用并传入的捕获的异常作为其参数。

由于 poem_failed 并没有抛出获异常或返回一个 Failure，因此在它执行完后，控制权又回到了 callback 链中。如果我们想让 poem_fail 完全处理好传进来的错误，那么返回一个 None

是再好不过的做法了。相反，如果我们只想让 poem_failed 采取一部分行动，但继续传递这个错误，那么我们需要改写 poem_failed，即将参数 err 作为返回值返回。如此一来，控制权交给了下一个 errback 回调。

注意到，迄今为止，got_poem 与 poem_failed 都不可能出现执行失败的情况，因此 errback 链上的 poem_done 是不可能被激活的。但在任何情况下这样做都是安全的，这体现了“防御式”编程的思想。比如在 got_poem 或 poem_failed 出现了 bugs，那么这样做就不会让这个 bugs 的影响进入 Twisted 的核心代码区。鉴于上面的描述，可以看出 addBoth 类型于 try/except 中的 finally 语句。

下面我们再来看看第三种可能情况，即成功下载到诗歌但 try_to_cummingsify 抛出了 VauleError，如图 22：

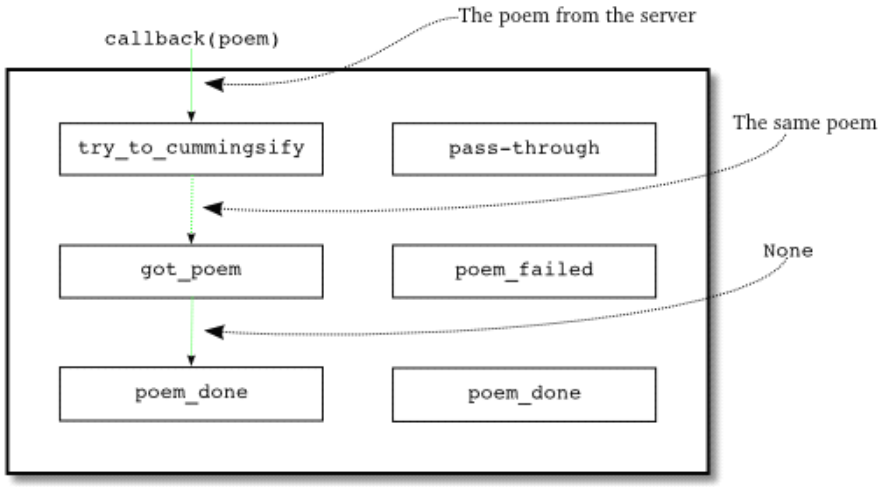


图 22：成功下载到诗歌当 cummingsify 执行失败

除了 got_poem 得到是原始式样的诗歌而不是小写版的外，与图 20 描述的情况完全相同。当然，控制权还是在 try_to_cummingsif 中进行了转移，即使用了 try/except 捕获了 ValueError 并返回了原始式样的诗歌。而这一切 deferred 并不知晓。

最后，我们来看看当试图连接一个无服务器值守的端口会出现什么情况，如图 23 所示：

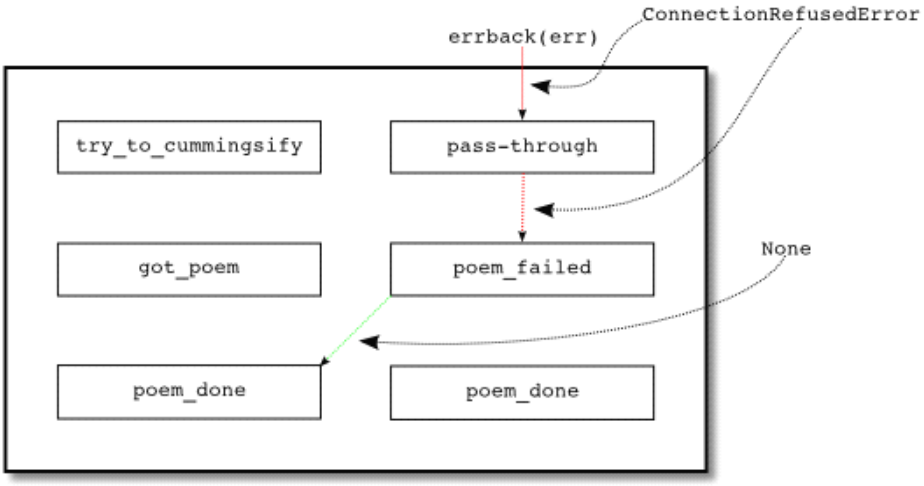


图 23 连接服务器失败
由于 poem_failed 返回了一个 None，因此控权又回到了 callback 链中。

版本 5.1

在版本 5.0 中我们使用普通的 try/except 来捕获 try_to_cummingsify 中的异常，而没有让 deferred 来捕获这个异常。这其实并没有什么错误，但下面我们将采取一种新的方式来处理异常。
设想一下，我们让 deferred 来捕获 GibberishError 与 ValueError 异常，并将其传递到 errback 链中进行处理。如果要保留原有的行为，那么需要下面的 errback 来判断错误类型是否为 Valuerror，如果是，那么返回原始式样的诗歌，这样一来，控制权再次回到 callback 链中并将原始式样的诗歌打印出来。
但有一个问题：errback 并不会得到原始诗歌内容 。它只会得到由 cummingsify 抛出的 vauleError 异常。为了让 errback 处理这个错误，我们需要重新设计它来接收到原始式样的诗歌。
一种方法是改变 cummingsify 以让异常信息中包含原始式样的诗歌。这也正是我们在 5.1 版本中做的，其代码实现在 twisted-client-5/get-poetry-1.py 中。我们改写 ValueError 异常为 CannotCummingsify 异常，其能将诗歌作为其第一个参数来传递。
如果 cummingsify 中外部模块中一个真实存在的函数，那么其最好是通过另一个函数来捕获非 GibberishError 并抛出一个 CannotCummingsify 异常。这样，我们的 poetry_main 就成为：

```
def poetry_main():
    addresses = parse_args()
    from twisted.internet import reactor
    poems = []
    errors = []
    def cummingsify_failed(err):
```

```
            if err.check(CannotCummingsify):
                print 'Cummingsify failed!'
                return err.value.args[0]
            return err
        def got_poem(poem):
            print poem
            poems.append(poem)
        def poem_failed(err):
            print >>sys.stderr, 'The poem download failed.'
            errors.append(err)
        def poem_done(_):
            if len(poems) + len(errors) == len(addresses):
                reactor.stop()
        for address in addresses:
            host, port = address
            d = get_poetry(host, port)
            d.addCallback(cummingsify)
            d.addErrback(cummingsify_failed)
            d.addCallbacks(got_poem, poem_failed)
            d.addBoth(poem_done)
```

而新的 deferred 结构如图 24 所示：

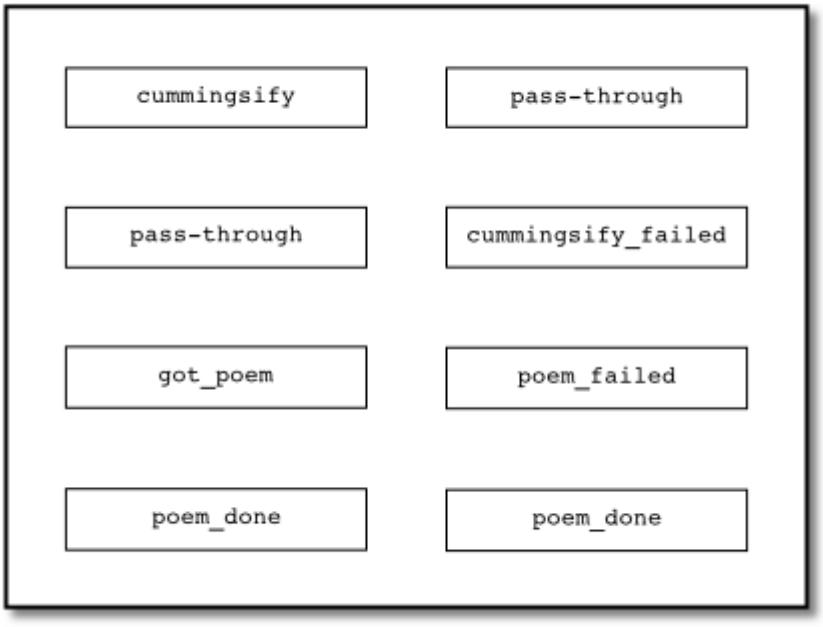


图 24：版本 5.1 的 deferrd 调用链结构

来看看 cummingsify_failed 的 errback 回调：

```
def cummingsify_failed(err):
    if err.check(CannotCummingsify):
        print 'Cummingsify failed!'
```

```
            return err.value.args[0]
        return err
```

我们使用了 Failure 中的 check 方法来确认嵌入在 Failure 中的异常是否是 CannotCummingsify 的实例。如果是，我们返回异常的第一个参数（即原始式样诗歌）。因此，这样一来返回值就不是一个 Failure 了，控制权也就又回到 callback 链中了。否则（即异常不是 CannotCummingsify 的实例），我们返回一个 Failure，即将错误传递到下一个 errback 中。
图 25 说明了当我们捕获一个 CannotCummingsify 时的调用过程：

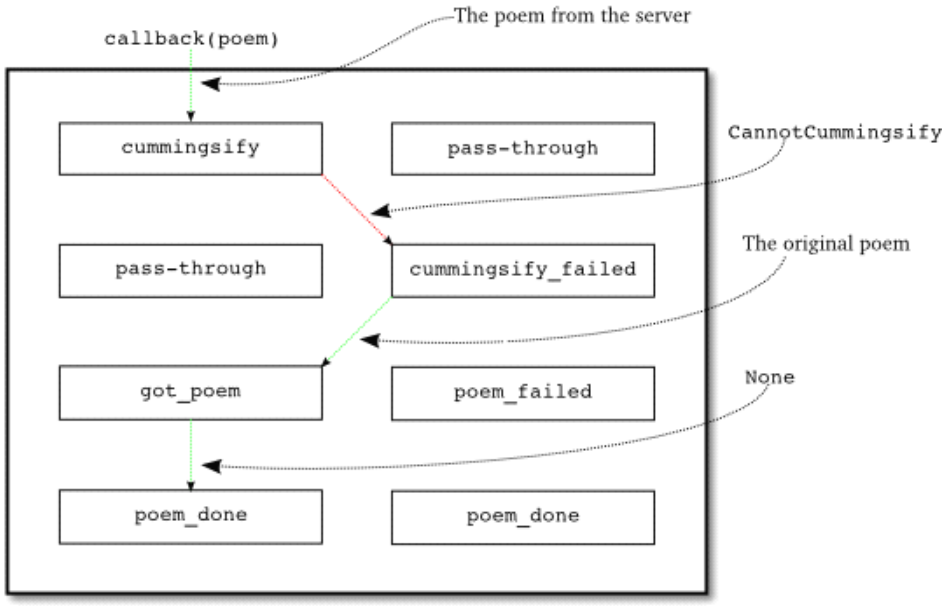


图 25：捕获一个 CannotCummingsify 异常
因此，当我们使用 deferrd 时，可以选择使用 try/except 来捕获异常，也可以让 deferred 来将异常传递到 errback 回调链中进行处理。

总结：

在这个部分，我们增强了客户端的 Deferred 的功能，实现了异常与结果在 callback/errback 链中“路由”。（你可以将各个回调看作成路由器，然后根据传入参数的情况来决定其返回值进入下一个 stage 的哪条链，或者说控制权进入下一个 stage 的哪个类型的回调）。虽然示例程序是虚构出来的，但它揭示了控制权在 deferred 的回调链中交错传递具体方向依赖于返回值的类型。
那我们是不是已经对 deferred 无所不知了？不，我们还会在下面的部分继续讲解 deferred 的更多的功能。但在第十一部分，我们先不讲这部分内容，而是实现我们的 Twisted 版本的诗歌下载服务器。
可以从这里从头开始阅读这个系列。

第十一部分：改进诗歌下载服务器

诗歌下载服务器

到目前为止，我们已经学习了大量关于诗歌下载客户端的 Twisted 的知识，接下来，我们使用 Twisted 重新实现我们的服务器端。利益于 Twisted 的抽象机制，接下来你会发现我们前面已经几乎全部学习到这部分知识了。其实现源码在 twisted-server-1/fastpoetry.py 中。之所以称其为 fastpoetry 是因为其并没有任何延迟的传输诗歌。注意到，其代码量比客户端少多了。

让我们一部分一部分地来看服务端的实现，首先是 poetryProtocol：

```
class PoetryProtocol(Protocol):
    def connectionMade(self):
        self.transport.write(self.factory.poem)
        self.transport.loseConnection()
```

如同客户端的实现，服务器端使用 Protocol 来管理连接（在这里，连接是由客户端发起的）。这里的 Protocol 实现了我们的诗歌下载逻辑的服务器端。由于我们协议逻辑处理的是单向的，服务器端的 Protocol 只负责发送数据。如果你访问服务器端，协议请求服务器在连接建立后立即发送诗歌，因此我实现了 connectionMade 方法，其会在 Protocol 中创建一个连接时被激活执行。

这个方法告诉 Transport 做两件事：将整首诗歌发送出去然后关闭连接。当然，这两个动作都是同步操作。因此调用 write 函数也可以说成“一定要将整首诗歌发送到客户端”，调用 loseConnection 意味着“一旦将要求下载的诗歌发送完毕就关掉这个连接”。

也许你看到了，Protocol 是从 Factory 中获得诗歌内容的：

```
class PoetryFactory(ServerFactory):
    protocol = PoetryProtocol
    def __init__(self, poem):
        self.poem = poem
```

这么简单！除了创建 PoetryProtocol 工厂的仅有的工作是存储要发送的诗歌。

注意到我们继承了 ServerFactory 而不是 ClientFactory。这是因为服务器是要被动地监听连接状态而不是像客户端一样去主动的创建。我们何以如此肯定呢？因为我们使用了 listenTCP 方法，其描述文档声明 factory 参数必须是 ServerFactory 类型的。

我们在 main 函数中调用了 listenTCP 函数：

```
def main():
    options, poetry_file = parse_args()
    poem = open(poetry_file).read()
    factory = PoetryFactory(poem)
```

```
    from twisted.internet import reactor
    port = reactor.listenTCP(options.port or 0, factory,nterface=options.iface)
    print 'Serving %s on %s.' % (poetry_file, port.getHost())
    reactor.run()
```

其做了三件事：

1.读取我们要发呈现的诗歌

2.创建 PoetryFactory 并传入这首诗歌

3.使用 listenTCP 来让 Twisted 监听指定的端口，并使用我们提供的 factory 来为每个连接创建一个 protocol

剩下的工作就是 reactor 来运转事件循环了。你可以使用前面任何一个客户端来测试这个服务器。

讨论

回忆下第五部分中的图 8 与图 9.这两张图说明了一个协议在 Twisted 创建一个连接后如何创建一个协议并初始化它的。其实对于 Twisted 在其监听的端口处接听到一个连接之后的整个处理机制也是如此。这也是为什么 connectTCP 与 listenTCP 都需要一个 factory 参数的原因。我们在图 9 中没有展示是的，connectionMake 其实也是 Protocol 初始化的一部分。无论在哪儿都一样（Dave 是想说，connectionMade 都会在 Protocol 初始化时执行），但我们在客户端处没有用到这个方法。并且我们在客户端的协议实现中的方法并没有在服务器处用到。因此，如果我们有这个需要，可以创建一个共享式的单一 PoetryProtocol 供客户端与服务器端使用。这各方式在 Twisted 经常见到。例如，NetstringReceiver protocol 即能读从一个连接中读也能向一个连接中写 netstrings。

我们略去了写从低层来实现服务器端的内容，但我们仍要来思考一下下面发生的事情。首先，调用 listenTCP 来告诉 Twisted 创建一个 listening socket 并将其添加到事件循环中。在 listening socket 有事件发生并不意味有数据要读，而是说明有客户端在等待连接自己。

Twisted 会自动接受连接请求，并创建一个新客户端式连接来连接客户端与服务器（中间桥梁）。这个新的连接也要加入事件循环中，并且 Twisted 为其创建了一个 Transport 与一个专门为这个连接服务的 PoetryProtocol。因此，Protocol 实例总是连接到客户端式的 socket，而不是监听式 socket。

我们可以在图 26 中形象地看到这一结果：

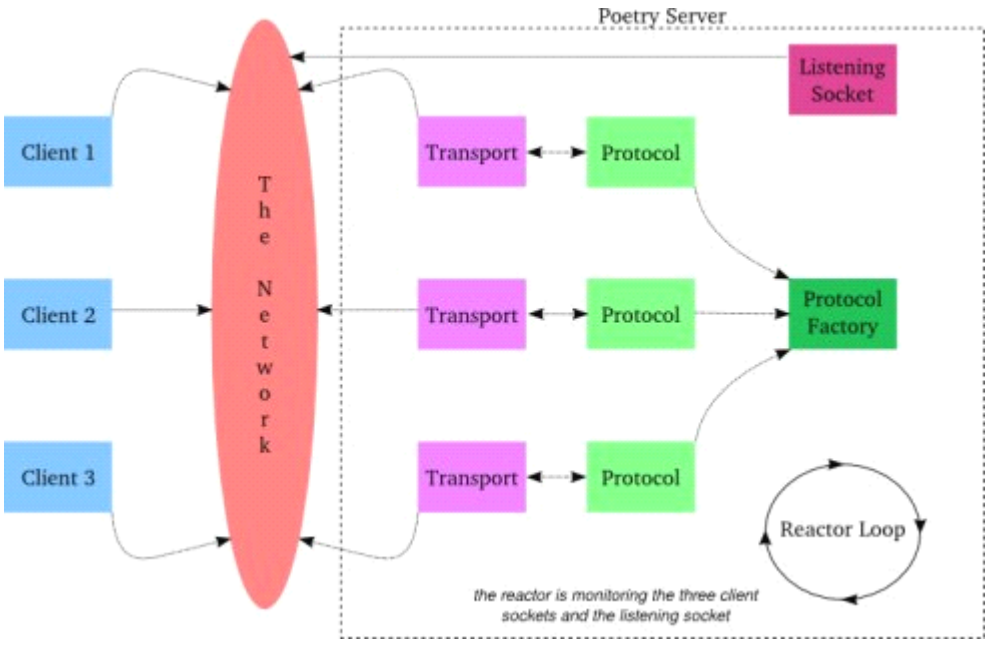


图 26：服务器端的网络连接

在图中，有三个客户端连接到服务器。每个 Transport 代表一个 client socket，加上 listening socket 总共是四个被 select 循环监听的文件描述符（file descriptor).当一个客户端断开与其相关的 transport 的连接时，对应的 PoetryProtocol 也会被解引用并当作垃圾被回收。而 PoetryFactory 只要我们还在监听新的连接就会一直不停地工作（即 PoetryFactory 不会将 PoetryProtocol 会随着一个连接的断开而被销毁）。

如果我们提供的诗歌很短的话，那么这些 client socket 与其相关的各种对象的生命期也就很短。但也有可能会是一台相当繁忙的服务器以至于同时有千百个客户端同时请求较长的诗歌。那没关系，因为 Twisted 并没有连接建立的限制。当然，当下载量持续的增加，在某个结点处，你会发现已经到达了 OS 的上限。对于那些高下载量的服务器，仔细的检测与测试是每天都必须的工作。

并且 Twisted 在监听端口的数量上亦无限制。实际上，一个单一的 Twisted 线程可以监听数个端口并为其提供不同的服务（通过使用不同的 factory 作为 listenTCP 的参数即可）。并且经过精心的设计，使用一个 Twisted 线程来提供多个服务还是使用多个 Twisted 线程来实现可以推迟到部署阶段来做决定。

我们这个版本的服务器有些功能是没有的。首先，它无法产生任何日志来帮助我们调试和分析网络出现的问题。另外，服务器也不是做为一个守护进程来运行，很容易通过 ctrl+c 来中止其执行。我们将会在第十二部分来分析这部分内容。

第十二部分：改进诗歌下载服务器

新的服务器实现

这里我们要新写一个 Twisted 版的服务器。然后，再来讨论一些 Deferred 的新功能。

在第九、十部分，我们提出了诗歌转换引擎这个概念。由于其实现太过简单，因此我们用随机选择来模拟了可能会出现转换失败的情景。但如果转换引擎位于服务器端，那么当服务器宕机就会出现真实的转换失败的情景了。
因此，在这部分我们要实现一个诗歌样式转换服务器，然后在一个部分，我们会重写我们的诗歌下载客户端来使用这一服务并且学习 Deferred 的新功能。

设计协议

到目前为止，服务器端与客户端之间的交互都是单向的。但样式转换服务需要两者进行双向交互-客户端将原始式样的诗歌发送给乳服务器，然后服务器将转换格式并将其发送给对应的客户端。因此，我们需要使用、或自己实现一个协议来实现这种交互。
我们设计服务器端可以提供若干种转换服务，而让客户端来进行选择。因此客户端需要向服务器端发送两部分信息：转换方式名与诗歌原始内容。服务器只是将转换格式之后的诗歌发送给客户端。这里使用到了简单的运程调用。

Twisted 支持需要种协议来解决这个问题：XML-RPC,Perspective Broker,AMP。
但介绍使用其中任何一种都需要大量的时间，因此我们使用自己实现的协议。我们约定客户端发送内容格式如下：转换名称.诗歌内容
我们将其以 netstring 格式编码，当然服务器回发的信息也是以 netstring 格式编码。由于 netstring 使用了 length-encoding，因此客户端能够识别出服务器没有将完整诗歌回发的情况。如果你尝试一下前面的协议，其无法检测到中途中断传输的情况。

代码

新的服务器实现代码在 twisted-server-1/transformedpoetry.py 中。首先，我们定义了一个 TransformService 类：
class TransformService(object):

```
    def cummingsify(self, poem):
        return poem.lower()
```

这里我们仅仅实现了一种转换方法，我们可以不回新格式转换方法。有一个重要的地方需要注意：格式转换服务与具体协议的实现是完全分分离的。将协议逻辑与服务逻辑分开是 Twisted 编程中常见的模式。这样做可以通过多种协议实现同一种服务，以增加了代码的重用性。

下面看看 factory 的实现代码：

```
class TransformFactory(ServerFactory):
    protocol = TransformProtocol
    def __init__(self, service):
        self.service = service
    def transform(self, xform_name, poem):
        thunk = getattr(self, 'xform_%s' % (xform_name,), None)
        if thunk is None: # no such transform
            return None
        try:
            return thunk(poem)
        except:
            return None # transform failed
    def xform_cummingsify(self, poem):
        return self.service.cummingsify(poem)
```

factory 提供了一个 transform 的函数，protocol 就是用它来代表客户端连接请求进行诗歌格式转换。

如果发现没有客户端请求的转换方法或转换失败，那么返回 None。和 TransformService 一样，factory 与具体的协议逻辑实现也是相互独立的。

有一个地方需要引起注意：我们通过 xfomr_前缀式方法来获取服务方法。这种方法在 Twisted 中很常见，尽管前缀经常发生变化，并且他们经常是依赖于独立于 factory 的一个对象（如此处的 TransformService）这是一种防止客户端使用蓄意恶性代码来让服务器端执行的方法。这种方法也提供了实现由服务提供具体协议代理的机制。

下面是协议实现代码：

```
class TransformProtocol(NetstringReceiver):
    def stringReceived(self, request):
        if '.' not in request: # bad request
            self.transport.loseConnection()
            return
        xform_name, poem = request.split('.', 1)
        self.xformRequestReceived(xform_name, poem)
    def xformRequestReceived(self, xform_name, poem):
        new_poem = self.factory.transform(xform_name, poem)
        if new_poem is not None:
            self.sendString(new_poem)
        self.transport.loseConnection()
```

在这个协议的实现中，我们通过继承 NetstringReceiver 来利用了 Twisted 对 netstrings 的实现。基类很好的处理了编码与解码功能，我们需要做的就是实现 stringReceived 方法。换句话说，

stringReceived 接收的参数是客户端编码之后的诗歌，而无需我们再去添加额外的编码信息。而且基类同样管理着缓冲区，即当一首诗歌完整接收完再进行解码。
如果一切进展正常的话，我们会使用 NetstringReceiver 的 sendString 方法来将格式转换成功后的诗歌发送给客户端。
注意我们是如何通过定义 xformRequestReceived 方法将收到的信息一步步推向更高的抽象层而实现了 Twisted 的模式。

一个简单的客户端

我们会在下一个部分来实现相应的客户端，这里使用一个简单的脚本来实现客户端，代码位于 twisted-server-1/transform-test 中。如果你运行服务器端于 11000 端口：
python twisted-server-1/transformedpoetry.py --port 11000

相应的运行脚本为：
./twisted-server-1/transform-test 11000

那么你会看到如下输出（经过 netstring 编码）：
15:here is my poem,

讨论

在这个部分介绍了如下几个方面内容：
1.双向通信
2.基于 Twisted 已有的协议实现新协议
3.将协议实现与服务功能实现独立分开
双向通信的基本机制是很简单的。我们使用前面服务器端与客户端使用的相同的技术来写与读数据，唯一不同的是我们这次两者都使用了（读与写）。当然，一个复杂的协议需要复杂的代码来处理接收到的数据流与格式化输出的信息。这也是为什么使用已经存在的协议的原因。
如果你开始觉得写简单的协议已经很上手了，那么最好就开始看看 Twisted 对不同协议的实现。尽管写一些简单的协议有助理解 Twisted 的编程风格，但在一个真实的程序中，最好是复用那些已经实现并证明性能良好的协议。
最后一点是将协议解析逻辑与服务实现逻辑分开，这是 Twisted 编程中非常重要的一个模式。我们这个服务器程序只是一个演示，你可以想象一下真实的网络服务是相当复杂的。通过将服务与协议逻辑分开，你可以通过复用已有的服务代码来运行于其它的协议实现上。
图 27 展示了一个格式转换服务器通过两种协议提供格式转换服务（当然，我们的服务器只提供了一种协议）：

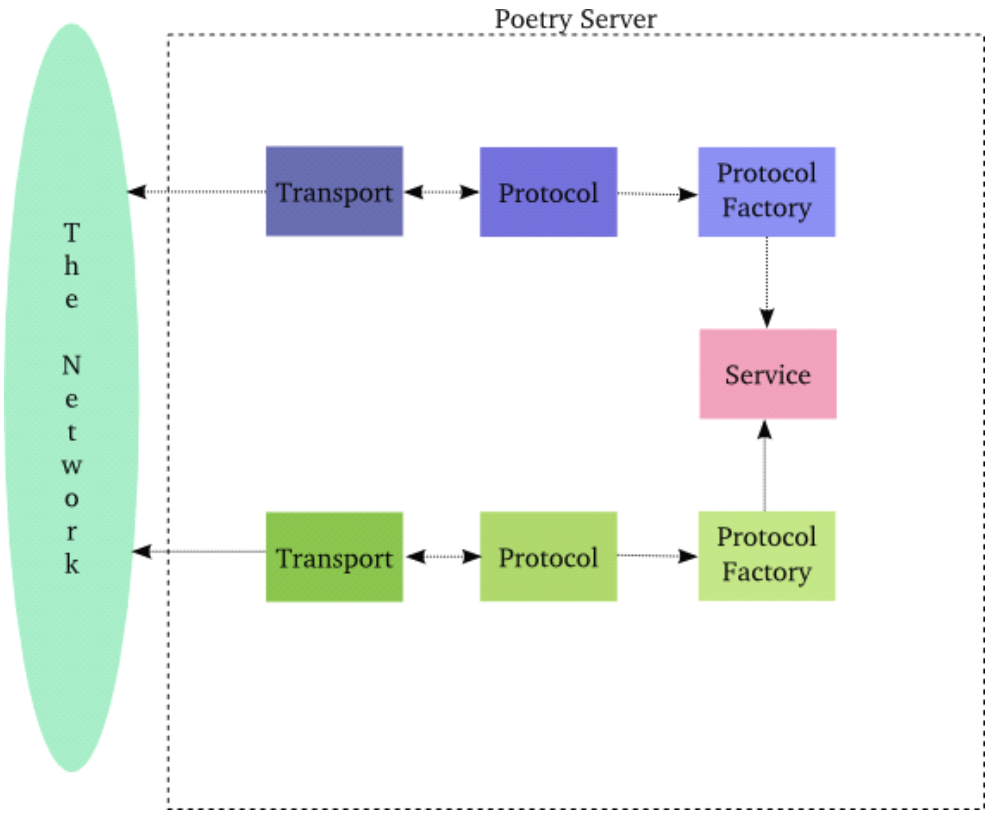


图 27 提供两种协议支持的格式转换服务器
虽然在图 27 中使用了两种协议，但他们也许是只有几个协议属性不同。factory 共享相同的服务。这样实现了代码的复用。

第十三部分：使用 Deferred 新功能实现新客户端

介绍

回忆下第 10 部分中的客户端 5.1 版。客户端使用一个 Deferred 来管理所有的回调链，其中包括一个格式转换引擎的调用。在那个版本中，这个引擎的实现是同步的。（即等待其执行再切到其它函数或任务中）
现在我们想实现一个新的客户端，其使用我们在第十二部分实现的格式服务器提供的格式转换服务。但这里有一个问题需要说清楚：由于格式转换服务是通过网络获取的，因此我们需要使用异步 I/O。这也就意味着我们获取格式转换服务的 API 必须是异步实现的。换句话说，try_to_cummingsify 回调将会在新客户端中返回一个 deferred。
如果在一个 deferred 的回调链中的一个回函数又返回了一个 deferred 会发生什么现象呢？我

们规定前一个 deferred 为外层 deferred，而后者则为内层 deferred。假设回调 N 在外层 deferred 中返回一个内层的 deferred。意味着这个回调宣称“我是一个异步函数，结果不会立即出现！”。由于外层的 deferred 需要调用回调链中下一个 callback 或 errback 并将回调 N 的结果传下去，因此，其必须等待直到内层 deferred 被激活。当然了，外层的 deferred 不可能处于阻塞状态，因为控制权此时已经转交给了 reactor 并且阻塞了。

那么外层的 deferred 如何知晓何时恢复执行呢？很简单，在内层 deferred 上添加 callback 或 errback 即可（即激活内层的 deferred）。因此，当内层 deferrd 被激活时，外层的 deferred 恢复其回调链的执行。当内层 deferred 回调执行成功，那么外层 deferred 会调用第 N+1 个 callback 回调。相反，如果内层 deferred 执行失败，那么外层 deferred 会调用第 N+1 个 errback 回调。

图 28 形象地解释说明了这一过程：

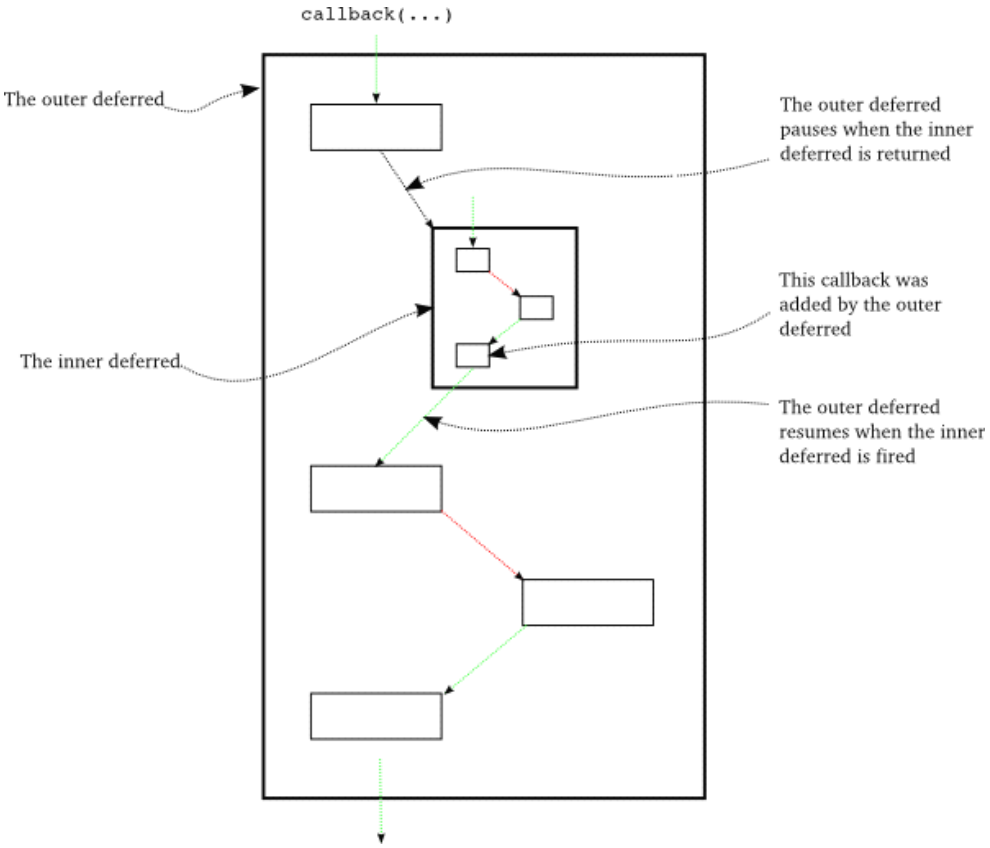


图 28 内层与外层 deferred 的交互

在这个图示中，外层的 deferred 有四个 callback/errback 对。当外围的 deferred 被激活后，其第一个 callback 回调返回了一个 deferred（即内层 deferred）。从这里开始，外层的 deferred 停止激活其回调链并且将控制权交还给了 reactor（当然是在给内层 deferred 添加 callback/errback 之后）。过了一段时间之后，内层 deferred 被激活，然后执行它的回调链并执行完毕后恢复外层 deferred 的回调执行过程。注意到，外层 deferred 是无法激活内层 deferred 的。这是不可能的，因为外层的 deferred 根本就无法获知内层的 deferred 何时能把结果准备好及结果内容是什么。相反，外层的 deferred 只可能等待（当然是异步方式）内部 deferred 的激活。

注意到外层 deferred 的产生内层 deferred 的回调的连线是黑色的而不是红色或蓝色，这是因为我们在内层 deferred 激活之前是无法获知此回调返回的结果是执行成功还执行失败。只有

在内层 deferred 激活时，我们才能决定下一个回调是 callback 还是 errback。
图 29 从 reactor 的角度来说明了外层与内层 deferred 的执行序列：

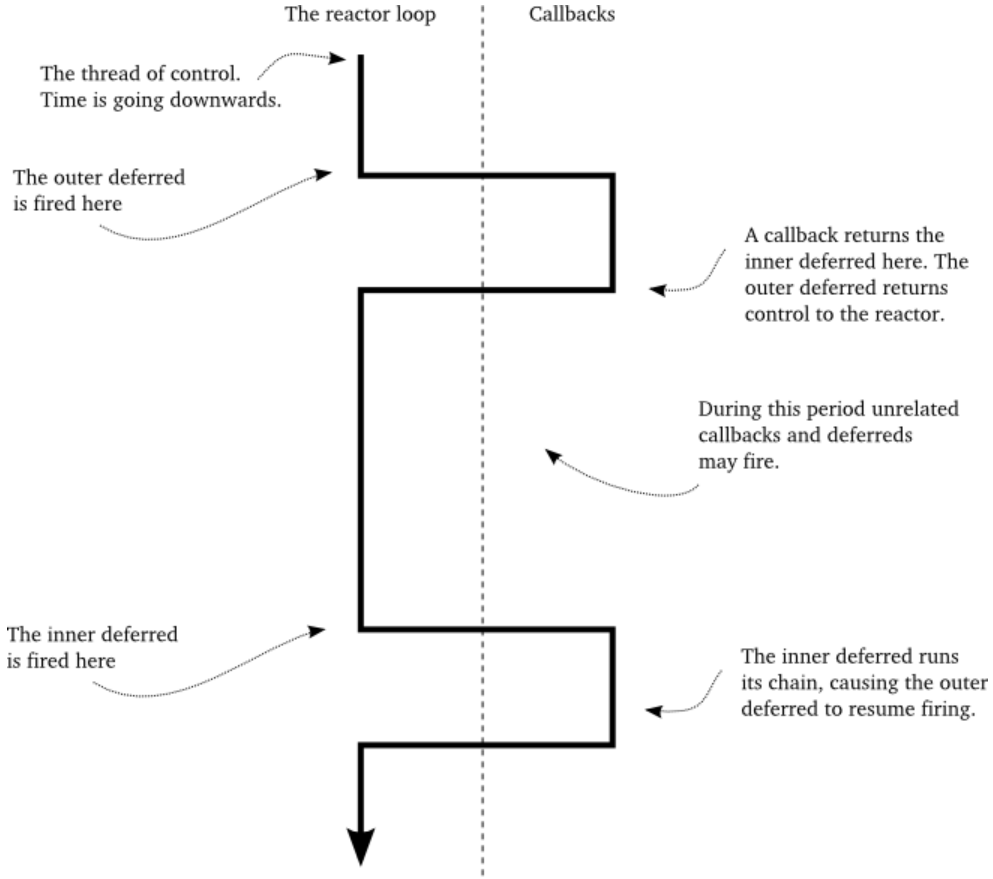


图 29 控制权的转换

这也许是 Deferred 类最为复杂的功能，但无需担心你可能会花费大量时间来理解它。我们将在示例 twisted-deferred/defer-10.py 中说明如何使用它。这个例子中，我们创建了两个外层 deferred，一个使用了简单的回调，另一个其中的一个回调返回了一个内部 deferred。通过阅读这段代码，我们可以发现外层 deferred 是在内层 deferred 激活后才开始继续执行回调链的。

客户端版本 6.0

我们将使用新学的 deferred 嵌套来重写我们的客户端来使用由服务器提供的样式转换服务。其实现代码在
twisted-client-6/get-poetry.py 中。与前几个版本一样，协议与工厂都没有改变。但我们添加了进行格式转换服务请求的协议与工厂实现。下面是协议实现代码：

```
class TransformClientProtocol(NetstringReceiver):
    def connectionMade(self):
        self.sendRequest(self.factory.xform_name, self.factory.poem)
    def sendRequest(self, xform_name, poem):
        self.sendString(xform_name + '.' + poem)
```

```
    def stringReceived(self, s):
        self.transport.loseConnection()
        self.poemReceived(s)
    def poemReceived(self, poem):
        self.factory.handlePoem(poem)
```

使用 NetstringReceiver 作为基类可以很简单地实现我们的协议。只要连接一旦建立我们就发出格式转换服务的请求。当我们得到格式转换之后的诗歌后交给工厂进行处理，下面是工厂代码：

```
class TransformClientFactory(ClientFactory):
    protocol = TransformClientProtocol
    def __init__(self, xform_name, poem):
        self.xform_name = xform_name
        self.poem = poem
        self.deferred = defer.Deferred()
    def handlePoem(self, poem):
        d, self.deferred = self.deferred, None
        d.callback(poem)
    def clientConnectionLost(self, _, reason):
        if self.deferred is not None:
            d, self.deferred = self.deferred, None
            d.errback(reason)
    clientConnectionFailed = clientConnectionLost
```

值得注意的是，工厂是如何处理两种类型错误：连接失败与在诗歌未全部接收完就中断连接。并且 clientConncetionLost 可能会在我们已经接收完诗歌后激活执行（即连接断开了），但在这种情况下，self.deferred 已经是个 None 值，这利益于 handePoem 中对 deferredr 处理。

这个工厂创建了一个 deferred 并且最后激活了它，这在 Twisted 编程中是一个好的习惯，即通常情况下，一个对象创建了一个 deferred，那么它应当负责激活它。

除了格式转换工厂外，还有一个 Proxy 类开包装了具体创建一个 TCP 连接到格式转换服务器：

```
class TransformProxy(object):
    """
    I proxy requests to a transformation service.
    """
    def __init__(self, host, port):
        self.host = host
        self.port = port
    def xform(self, xform_name, poem):
        factory = TransformClientFactory(xform_name, poem)
        from twisted.internet import reactor
        reactor.connectTCP(self.host, self.port, factory)
        return factory.deferred
```

这个类提供了一个 xform 接口，以让其它程序请求格式转换服务。这样一来其它代码只需要提出请求并得到一个 deferred，而无需考虑什么端口与 IP 地址之类的问题。

剩下的代码除了 try_to_cummingsify 外都没有改变：

```
def try_to_cummingsify(poem):
    d = proxy.xform('cummingsify', poem)
    def fail(err):
        print >>sys.stderr, 'Cummingsify failed!'
        return poem
    return d.addErrback(fail)
```

这个作为外层 deferred 的回调返回了一个内层的 deferred，但我们仍然需要更改 main 方法，除了创建了一个 Proxy 对象。由于 try_to_cummingsify 已经是 deferred 回调链中的一部分，因此其早已使用了异步方式。因此我们说 main 函数无需更改。
你可能注意到 return d.addErrback(fail)这句，其它它等于

```
d.addErrback(fail)
return d
```

结束语

这一部分我们学习了关于 deferred 如何透明地完成了回调链内部再次处理 deferred。并由此，我们可以无需考虑内部实现细节并放心地在外部 deferred 上添加回调。
在第十四部分，我们将讲解 deferred 的另外一个特性。

第十四部分：Deferred 用于同步环境

介绍

这部分我们要介绍 Deferred 的 另外一个功能。便于讨论，我们设定如下情景：假设由于众多的内部网请求一个外部诗歌下载服务器，但由于这个外部下载服务器性能太差或请求负荷太重。因此， 我们不想将所有的内部请求全部发送到外部服务器。我们的处理办法是，在中间添加一个缓存代理。当一个请求来到后，我们可以从中间缓存代理中找到缓存的备份（如果有缓存）或者直接从外部服务器获得。部署图如图 30 所示：

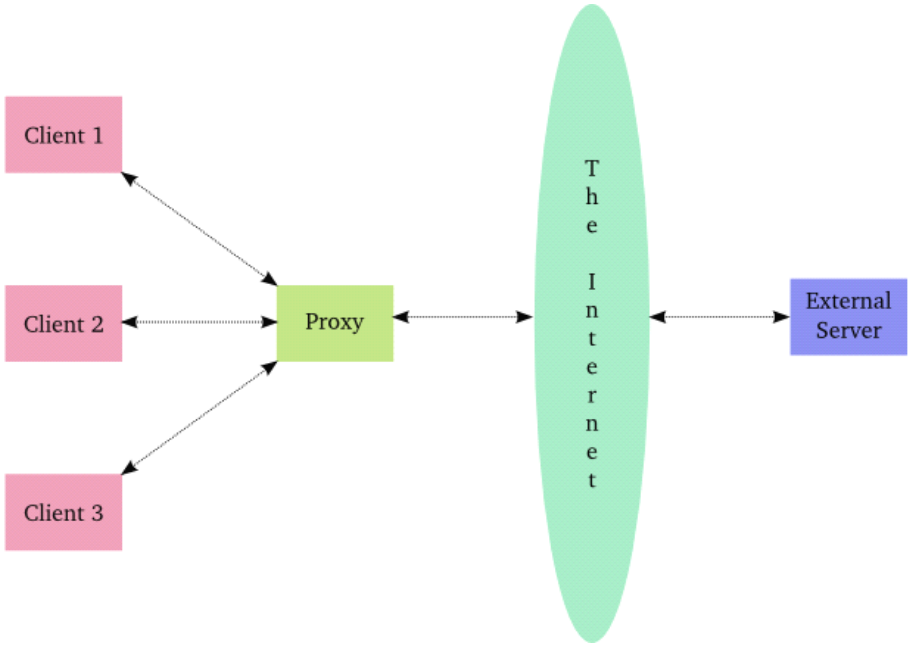


图 30 缓存代理服务器

考虑到，客户端端发送请求来时，此缓存代理可能会将本地的缓冲的诗歌取出并回复，也有可能需要异步等待外部诗歌下载服务器的诗歌回复。如此一来，就会出现这样的情景：客户端发送来的请求，缓存代理处理请求可能是同步也可能是异步。

要解决这个需要，就用到了 Deferred 的另一个特性，即可以在将 Deferred 返回前就激活这个 Deferred。之所以可以这样做，是因为你可以在一个已经激活的 deferred 上添加回调处理函数。一个非常值得注意的是：已经被激活的 deferred 可以立即激活新添加的回调处理函数。

图 31 表示一个已经激活的 deferred：

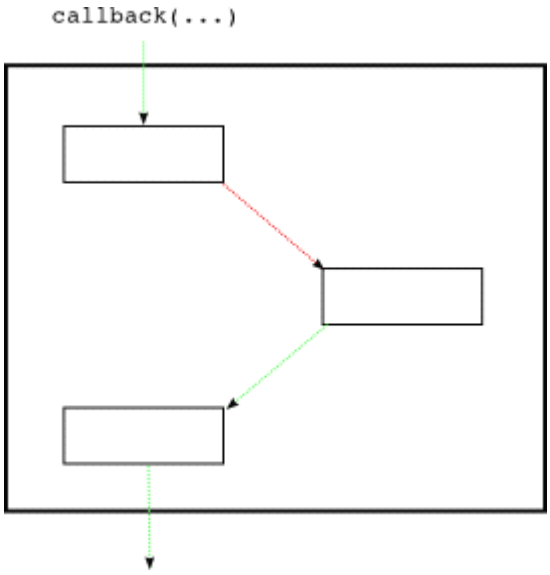


图 31 已经激活的 deferred

如果在此时，我们再为其另一对 callback/errback，那么会立即激活新的回调并执行。如图 32

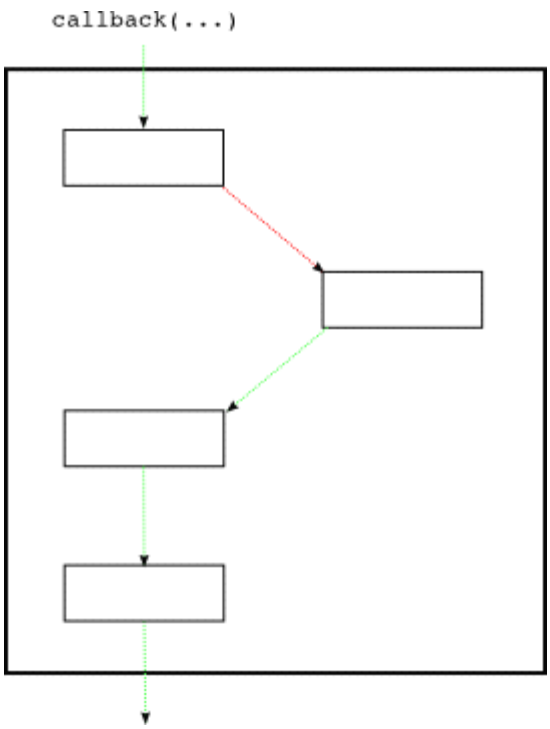


图 32 同一个 deferred 在添加新的回调之后

后面的 callback 回调被执行，是因为前面的 callback 执行成功。如果其执行失败，那么接下来执行的将是新添加的 errback 回调。

我们可以通过 twisted-deferred/defer-11.py 示例来检测我们这里说到的特性。其中第二组例子，演示了 deferred 中的 pause 与 unpause 函数的功能，即可以暂停一个已经激活的 deferred 对其回调链上回调的激活。并可以用 unpause 来解除暂停设置。这两个函数同样完成了在回调中继续产生 deferred 期间的控制。

代理 1.0 版本

让我们来看看第一个版本的缓存代理的实现 twisted-server-1/poetry-proxy.py。由于该服务器既作为服务器向客户端请求提供本地缓存的诗歌，同时也要作为向外部诗歌下载服务器提出下载请求的客户端。因此，其有两套协议/工厂，一套实现服务器角色，另一套实现客户端角色。

首先我们先来看看 ProxyService 的实现部分：

```
class ProxyService(object):
    poem = None # the cached poem
    def __init__(self, host, port):
        self.host = host
        self.port = port
    def get_poem(self):
        if self.poem is not None:
            print 'Using cached poem.'
```

```
            return self.poem
        print 'Fetching poem from server.'
        factory = PoetryClientFactory()
        factory.deferred.addCallback(self.set_poem)
        from twisted.internet import reactor
        reactor.connectTCP(self.host, self.port, factory)
        return factory.deferred
    def set_poem(self, poem):
        self.poem = poem
        return poem
```

主要的函数是 get_poem。如果缓存中没有请求的诗歌，那么就会建立连接从外部服务器中异步取得而返回一个 deferred，并将取得的诗歌放到缓冲区中。相反，若缓冲区中存在请求的诗歌，则直接返回诗歌。

我们如何来处理这样一个返回值不确定的函数呢，让我们来看看实现服务器角色的协议/工厂：

```
class PoetryProxyProtocol(Protocol):
    def connectionMade(self):
        d = maybeDeferred(self.factory.service.get_poem)
        d.addCallback(self.transport.write)
        d.addBoth(lambda r: self.transport.loseConnection())
class PoetryProxyFactory(ServerFactory):
    protocol = PoetryProxyProtocol
    def __init__(self, service):
        self.service = service
```

这里使用了 maybeDeferred 函数解决了这个问题。此函数的功能就是如果作为其参数返回值为 defer，那么其不作任何处理，原样将 defer 返回。但如何返回值不是 defer 而是一个值（正如我们的缓存代理将本地缓冲的诗歌返回一样），那么这个 maybeDeferred 会将该值重新打包成一个已经激活的 deferred 返回，注意是已经激活的 deferred。当然，如果返回的是一个异常，其也会将其打包成一个已经激活的 deferred，只不过就不是通过 callback 而是 errback 激活的。

代理 2.0 版本

前面我们已经提到，有另一种替代方法来实现这一机制。这在 twisted-server-2/poetry-proxy.py 中很好的说明了。即我们可以返回一个已经激活的 defer，放在这儿就是如果缓存代理中有请求的诗歌，那么就通过返回一个激活的 deferred：

```
def get_poem(self):
    if self.poem is not None:
        print 'Using cached poem.'
        # return an already-fired deferred
        return succeed(self.poem)
    print 'Fetching poem from server.'
```

```
        factory = PoetryClientFactory()
        factory.deferred.addCallback(self.set_poem)
        from twisted.internet import reactor
        reactor.connectTCP(self.host, self.port, factory)
        return factory.deferred
```

如果我们去看 succeed 的源码会发现，其只是在返回一个 deferred 之前，将其激活。同样，如果想要返回一个以失败的方式激活的 deferred，可以调用函数 defer.fail

在这个版本中，由于 get_poem 返回的是 deferred 而不像前一个版本存在不确定性因素。因此协议实现就无需使用 maybeDeferred：

```
class PoetryProxyProtocol(Protocol):
    def connectionMade(self):
        d = self.factory.service.get_poem()
        d.addCallback(self.transport.write)
        d.addBoth(lambda r: self.transport.loseConnection())
```

总结

这个部分我们学习到了 deferred 可以在返回之前被激活，这样我们就可以将其用于同步环境中。并且我们已经知道了有两种方法来实现。其一是使用 maybeDeferred 函数，其二是使用 succeed/fail。两者返回的都是 deferred，不同的是前者返回的可能是异步的也可能是同步的，而后者返回的肯定是同步的，即已经激活。

Deferred 可以在激活后添加新的回调也间接说明了我们在第九部分提到的，deferred 中会在最后一个回调中遇到未处理异常，并在此 deferred 被垃圾回收（即其已经没有任何外界引用）时才将该异常的情况打印出来。即 deferred 回在其销毁前一直持有异常，等待可能还会添加进来的回调来处理。

我们已经将 deferred 中的大部分功能都介绍完了，当然 Twisted 开发人员可能不会增强 deferred 的功能。我们下一部分将讲讲 Twisted 的其它内容。

第十五部分：测试诗歌

你可以从":doc:`p01`"开始阅读；也可以从":doc:`index`"浏览索引.

简介

在我们探索 Twisted 的过程中写了很多代码,但目前我们却忽略了一些重要的东西 —— 测

试.你也是会怀疑怎样用像 unittest 这样 Python 自带的同步框架测试异步代码.答案是你不能.正如我们已经发现的,同步代码和异步代码是不能混合的,至少不容易.
幸运地是,Twisted 包含自己的测试框架,叫 trial,它支持测试异步代码(当然你也可以用它测试同步代码).
我们假设你已经熟悉了 unittest 的机理和相似的测试框架,它允许你通过定义类创建测试.这个类通常是一个父类(通常叫"TestCase")的子类,类中的方法以"test"开头并被视作一个测试.框架负责发现所有的测试,一个接一个地运行它们,并伴有可选项 setUp 和 tearDown 步骤,之后报告结果.

例子

你可以在 tests/test_poetry.py 中找到一些关于测试的例子.为了确保我们所有的例子是自包含的(以便你不用担心 PYTHONPAYH 设置),我们将所有需要的代码拷贝到测试模块中.当然正常情况,你只需导入需要测试的模块.
这个例子既测试诗歌客户端又测试服务器,通过使用客户端从测试服务器抓取一首诗. 为了提供一个可供测试的诗歌服务器, 我们在测试案例中实现 setUp 方法:

```
class PoetryTestCase(TestCase):

    def setUp(self):
        factory = PoetryServerFactory(TEST_POEM)
        from twisted.internet import reactor
        self.port = reactor.listenTCP(0, factory, interface="127.0.0.1")
        self.portnum = self.port.getHost().port
```

这个 setUp 方法用一首测试诗建立诗歌服务器,然后监听一个随机开放端口.我们保存了端口号,以便实际测试需要时可以利用.当然测试结束时我们会用 tearDown 清除测试服务器:

```
def tearDown(self):
    port, self.port = self.port, None
    return port.stopListening()
```

这把我们带到了第一个测试, test_client, 用 get_poetry 从测试服务器获取诗歌并且验证这就是我们所期望的诗歌:

```
def test_client(self):
    """The correct poem is returned by get_poetry."""
    d = get_poetry('127.0.0.1', self.portnum)

    def got_poem(poem):
        self.assertEquals(poem, TEST_POEM)

    d.addCallback(got_poem)

    return d
```

注意我们的测试函数返回一个 deferred.在 trial 中,每个测试方法都以回调的方式运行.这意

味着 reactor 正在运行并且我们可以以测试的一部分执行异步操作.我们仅仅需要让框架知道测试是异步的,这可以通过采用常规的 Twisted 方式 —— 返回 deferred 来实现.
trial 框架在调用 tearDown 方法之前将等待直到 deferred 激发,并且当 deferred 失败时将使测试失败(如,最后一个回调/错误回调对失败).如果我们的 deferred 反应太慢(默认2分钟)它同样会使测试失败.这意味着如果测试完成,我们知道 deferred 激发了,因此我们的回调激发了并且运行了 assertEquals 测试方法.
我们的第二个测试, test_failure, 证实 get_poetry 以适当的方式失败了,如果不能连接到服务器:

```
def test_failure(self):
    """The correct failure is returned by get_poetry when
    connecting to a port with no server."""
    d = get_poetry('127.0.0.1', -1)
    return self.assertFailure(d, ConnectionRefusedError)
```

这里我们打算连接到一个无效端口,之后使用 trial 提供的 assertFailure 方法.这个方法类似于熟悉的 assertRaises 方法但是是针对异步代码的.它返回一个 deferred,如果给定的 deferred 失败则返回成功,否则返回失败.
你可以用 trial 脚本自己运行这些测试,如下:

```
trial tests/test_poetry.py
```

你将看到显示每个测试案例的输出,OK 表示测试通过了.

讨论

由于当谈到基本 API 时,trial 与 unittest 十分相似,所以开始写测试十分容易.如果你的测试使用异步代码,仅仅返回 deferred 就可以了,trial 将负责其余的事情.你也可以从 setUp 或 tearDown 方法返回一个 deferred,如果它们也需要异步.
任何来自测试的日志消息将被收集到当前文件夹下的一个文件中,即"_trial_temp", trial 会自动创建它. 除了打印到屏幕的错误,日志是调试失败测试的实用入口.
图33显示了一个正在进行中的假想测试：

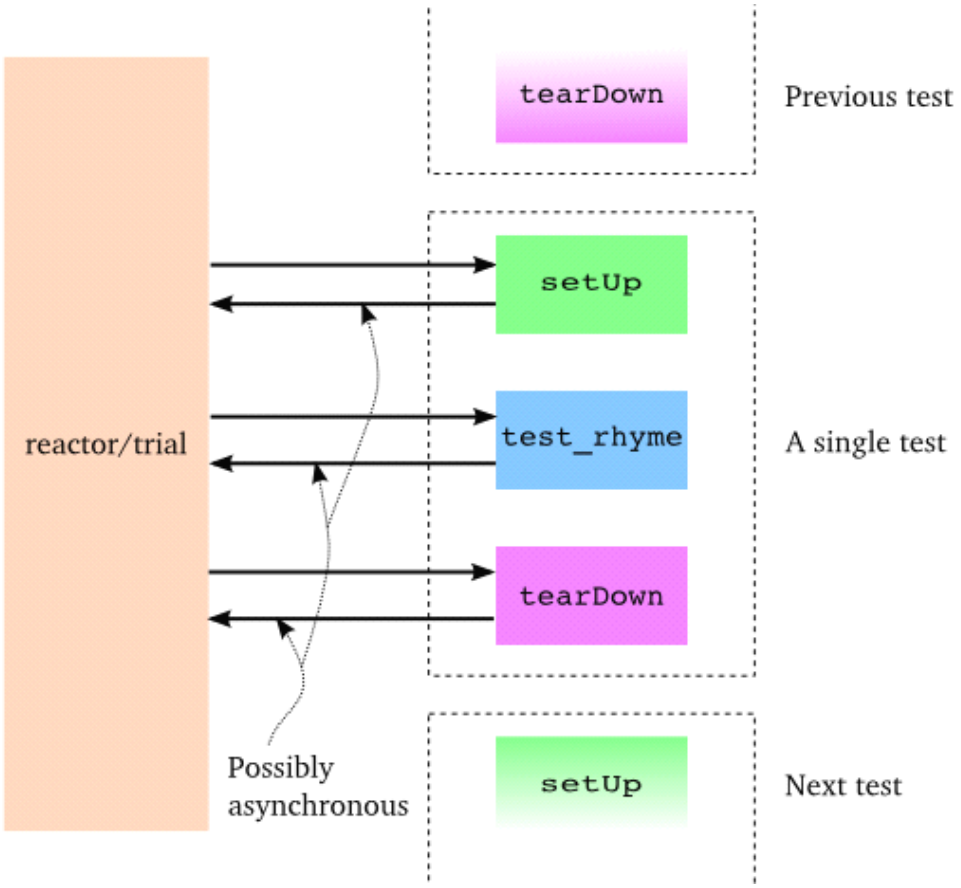


图33: 进行中的 trial 测试

如果你之前使用过类似的框架,这是一个熟悉的模型,除了所有测试相关的方法可能返回 deferreds.

trial 框架是一个关于如何"异步运作"的很好例子,包括级联在整个程序中的变化.为了使一个测试(或任何函数,方法)是异步的,它必须:

1 非阻塞并且,通常
2 返回一个 deferred.

但这意味着无论什么调用,那个函数必须愿意接收一个 deferred,并且非阻塞(如此又好像返回了一个 deferred).如此这般一层又一层.这样就呼唤出现 trial 一样的框架,可以处理返回 deferreds 的异步测试.

总结

这就是关于单元测试的内容.如果你想了解更多关于如何为 Twisted 代码写单元测试的例子,你只需要看看 Twisted 代码本身.Twisted 框架自带了一套非常庞大的单元测试,而且每个新的发布又会加入很多.由于这些测试在被接受入代码库之前,经过严格的代码评论以及 Twisted 专家们的仔细审查,故而它们是告诉你如何以正确方式测试 Twisted 代码的极好例子.

在 :doc:`p16` 中,我们将使用 Twisted 工具将诗歌服务器转化为一个真正的守护进程.

参考练习

3 改变测试之一使其失败,然后再次运行 trial 看看输出结果.
4 阅读 trial 在线文档.
5 为我们这个系列中所创建的其他诗歌服务写测试.
6 探索 Twisted 中的 一些测试.

第十六部分：Twisted 进程守护

你可以从":doc:`p01`"开始阅读；也可以浏览":doc:`index`"的索引

简介

目前我们所写的服务器仅仅运行在终端窗口，结果通过 print 语句输出到屏幕.这对于开发来说已经足够，但对于产品级的部署还远远不够. 健壮的产品级服务器应该：

1 运行一个 daemon 进程,这个进程不与任何终端或用户会话相关.因为没有人愿意当某用户登出时服务自动关闭.
2 将调试和错误信息发送到一系列滚转日志文件, 或者 syslog 服务.
3 放弃过高的权限,比如,在运行前切换到较低权限.
4 保存它的 pid 文件以便管理员方便地向 daemon 发送信号.

我们可以利用 Twisted 提供的 twistd 脚本获得所有以上功能. 但是首先需要稍稍修改我们的代码.

IService

IService 接口定义了一个可以启动或停止的服务. 这个服务究竟做了些什么? 答案是任何你喜欢的事情——这个接口只需要自提供的一些通用属性和方法,无须用户定义特定的函数.
这边有两个需要的属性: name 和 running.其中 name 属性是一个字符串,如 "fastpoetry",或者 None 如果你不想给这个服务起名字. running 属性是 **Boolean** 变量,如果服务成功启动,值为 True.
下面我们只涉及 IService 的某些方法, 跳过那些很显然的或者在更简单的 Twisted 程序中用不到的高级方法. startService 和 stopService 是 IService 的两个关键方法：

```
def startService():
    """
    Start the service.
    """

def stopService():
```

```
        """
        Stop the service.

        @rtype: L{Deferred}
        @return: a L{Deferred} which is triggered when the service has
                 finished shutting down. If shutting down is immediate, a
                 value can be returned (usually, C{None}).
        """
```

同样,这些方法做什么取决于服务的需求,比如 startService 可能会:

- 加载配置数据,或
- 初始化数据库,或
- 开始监听某端口,或
- 什么也不做.

stopService 可能会:

- 储存状态,或
- 关闭打开的数据库连接,或
- 停止监听某端口,或
- 什么也不做.

当我们写自定义服务时，要恰当地实现这些方法.对于一些通用的行为,比如监听某端口,Twisted 提供了现成的服务可以使用.

注意 stopService 可以选择地返回 deferred,要求当服务完全关闭时被激发.这允许我们的服务在结束之后与整个程序终止之前完成清理工作.如果你需要服务立即关闭,可以仅仅返回 None 而不是 deferred.

服务可以被组织成集合以便一起启动和停止.下面来看看这里最后一个 IService 方法: setServiceParent,它添加一个服务到集合:

```
def setServiceParent(parent):
        """
        Set the parent of the service.

        @type parent: L{IServiceCollection}
        @raise RuntimeError: Raised if the service already has a parent
                             or if the service has a name and the parent already has a child
                             by that name.
        """
```

任何服务都可以有双亲,这意味着服务可以被组织为层级结构.这把我们引向了今天讨论的另一个接口.

IServiceCollection

IServiceCollection 接口定义了一个对象，它可包含若干个 IService 对象.一个服务集合仅仅是一个普通的类容器,具有以下方法:

- 通过名字查找某服务(getServiceNamed)

- 遍历集合中的服务(__iter__)
- 添加一个服务到集合(addService)
- 从集合中移除一个服务(removeService)

Application

一个 Twisted Application 不是通过一个单独的接口定义的.相反, Application 对象需要实现 IService 和 IServiceCollection 接口以及一些我们未曾涉及的接口.
Application 是一个代表你整个 Twisted 应用的最顶层的服务. 在你 daemon 中的所有其他服务将是这个 Application 对象的儿子(甚至孙子,等等.).
其实需要你自己实现 Application 的机会很小,Twisted 已经提供了一个当下常用的实现.

Twisted Logging

Twisted 在其模块 twistd.python.log 中包含了其自身的日志架构.由于写日志的基本 **API** 非常简单, 我们仅仅介绍一个小例子： basic-twisted/log.py,如果你感兴趣更多细节可以浏览 Twisted 模块.
我们也不详细介绍安装日志处理程序的 **API**,因为 twistd 脚本会帮我们做.

FastPoetry 2.0

好吧,让我们看看代码.我们已经将快诗服务器升级为使用 twistd. 源码在 twisted-server-3/fastpoetry.py. 首先我们有了 诗歌协议:

```
class PoetryProtocol(Protocol):

    def connectionMade(self):
        poem = self.factory.service.poem
        log.msg('sending %d bytes of poetry to %s'
                % (len(poem), self.transport.getPeer()))
        self.transport.write(poem)
        self.transport.loseConnection()
```

注意没有使用 print 语句,而是使用 twisted.python.log.msg 函数去记录每个新连接.
这里是 工厂类:

```
class PoetryFactory(ServerFactory):

    protocol = PoetryProtocol

    def __init__(self, service):
        self.service = service
```

正如你看到的，诗不再储存在工厂中，而是储存在一个被工厂引用的服务对象上。注意这边

协议是如何通过工厂从服务获得诗歌.最后,看一下 服务类:

```
class PoetryService(service.Service):

    def __init__(self, poetry_file):
        self.poetry_file = poetry_file

    def startService(self):
        service.Service.startService(self)
        self.poem = open(self.poetry_file).read()
        log.msg('loaded a poem from: %s' % (self.poetry_file,))
```

就像许多其他接口类一样,Twisted 提供了一个基类供自定义实现,同时具有方便的默认行为.我们使用 twisted.application.service.Service 类实现 PoetryService.
这个基类提供了所有必要方法的默认实现,所以我们只需要实现个性化的行为.在上面的例子中,我们只重载了 startService 方法来加载诗歌文件.注:我们仍然调用了相应的基类方法(它为我们设置 running 属性).
另外值得一提的是： PoetryService 对象不知道关于 PoetryProtocol 的任何细节.这里服务的任务仅仅是加载诗歌以及为其他需要诗歌的对象提供接口.也就是说, PoetryService 只关心提供诗歌的更高层的细节,而不是关心诸如通过 **TCP** 连接发送诗歌这样的更底层的细节.所以同样的服务可以被另外的协议使用,如 **UDP** 或 **XML-RPC**.虽然对于简单的服务好处不大,但你可以想象其在更实际服务实现中的优势.
如果这是一个典型的 Twisted 程序,到目前我们看到的代码都不该出现在这个文件里.它们应该在一些模块当中(也许是 fastpoetry 和 fastpoetry.service).但是,遵循我们的惯例会使这些例子自包含,也就是在一个脚本中包含了所有东西.

Twisted tac files

这个脚本的其余部分包含通常作为完整内容的 Twisted tac 文件. tac 文件是一个 Twisted Application Configuration 文件,它告诉 twistd 怎样去构建一个应用.作为一个配置文件,它负责选择设置(如端口,诗歌文件位置,等)来以一种特定的方式运行这个应用.换句话说, tac 代表我们服务的一个特定部署(在这个端口服务这首诗),而不是启动任何诗歌服务的一般脚本.如果我们在同一个域运行多个诗歌服务,我们将为每一个服务准备一个 tac 文件(因此你可以明白为什么 tac 文件通常不包含任何一般目的的代码).在我们的例子中, tac 文件被配置为使 poetry/ecstasy.txt 运行在回环接口的10000号端口:

```
# configuration parameters
port = 10000
iface = 'localhost'
poetry_file = 'poetry/ecstasy.txt'
```

注意 twistd 并不知道这些特定变量,我们仅仅将这些配置值统一的放在这里.事实上, twistd 只关心整个文件中的一个变量,我们即将看到.下面我们开始建立我们的应用:

```
# this will hold the services that combine to form the poetry server
top_service = service.MultiService()
```

我们的诗歌服务器将包含两个服务，上文定义的 PoetryService,和一个 Twisted 的内置服务,它将建立服务我们诗歌的监听套接字.由于这两个服务明显的相关,我们用 MultiService 将它们组织在一起,一个实现 IServiceCollection 和 IService 的类.
作为一个服务集合, MultiService 把我们的诗歌服务组织在一起.同时作为一个服务,MultiService 启动时将启动它的子服务,关闭时将关闭它的子服务.让我们向服务集合 添加第一个诗歌服务:

```
# the poetry service holds the poem. it will load the poem when it is
# started
poetry_service = PoetryService(poetry_file)
poetry_service.setServiceParent(top_service)
```

这是非常简单的内容.我们仅创建了 PoetryService,然后用 setServiceParent 方法将其添加到服务集合.下面我们添加 **TCP** 监听器:

```
# the tcp service connects the factory to a listening socket. it will
# create the listening socket when it is started
factory = PoetryFactory(poetry_service)
tcp_service = internet.TCPServer(port, factory, interface=iface)
tcp_service.setServiceParent(top_service)
```

Twisted 为创建连接到任意工厂的 **TCP** 监听套接字提供了 TCPServer 服务(这里是PoetryFactory),我们没有直接调用 reactor.listenTCP 因为 tac 文件的工作是使我们的应用准备好开始,而不是实际启动它. 这里 TCPServer 将在被 twistd 启动后创建套接字.
你可能注意到我们没有为任何服务起名字.为服务起名不是必需的,而仅是一个可选项,如果你希望在运行时查找服务.因为我们不需要这个功能,所以这里没有为服务命名.
既然我们已经将两个服务绑定到服务集合.现只需创建我们的应用,并且将它添加到集合:

```
# this variable has to be named 'application'
application = service.Application("fastpoetry")

# this hooks the collection we made to the application
top_service.setServiceParent(application)
```

在这个脚本中 twistd 所关心的唯一变量就是 application. twistd 正是通过它找到那个需要启动的应用(所以这个变量必须被命名为 applicaton).当应用被启动时,我们添加到它的所有服务都会被启动.
图34显示了我们刚刚建立的应用的结构:

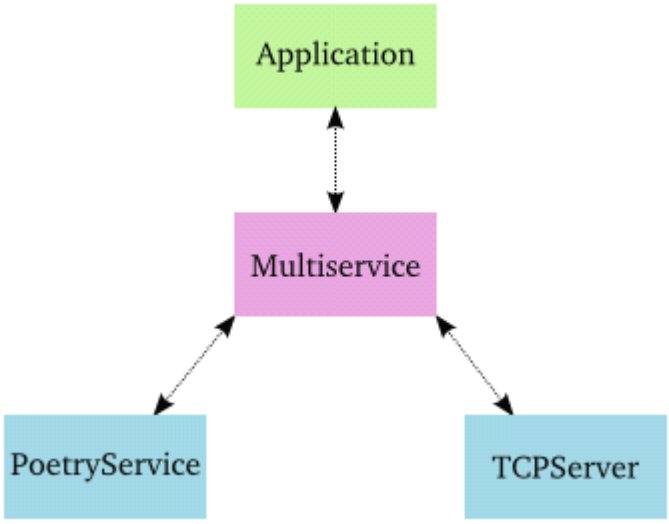


图34: fastpoetry 应用的结构图

Running the Server

让我们的新服务器运转起来.作为 tac 文件,我们需要用 twistd 启动它.当然,它仅仅是一个普通的 Python 文件.所以我们首先用 python 命令启动,再看看会发生什么:
python twisted-server-3/fastpoetry.py

如果你这样做,会发现什么也没有发生！正如前文所述, tac 文件的工作是使我们的应用准备好运行,而不是实际运行它.作为 tac 文件这个特殊目的的提醒,人们将它的扩展名规定为 .tac 而不是 .py.但是 twistd 脚本实际并不区分扩展名.
让我们用 twistd 脚本来实际运行这个服务器:
twistd --nodaemon --python twisted-server-3/fastpoetry.py

运行以上命令后会看到如下输出:
2010-06-23 20:57:14-0700 [-] Log opened.
2010-06-23 20:57:14-0700 [-] twistd 10.0.0 (/usr/bin/python 2.6.5) starting up.
2010-06-23 20:57:14-0700 [-] reactor class: twisted.internet.selectreactor.SelectReactor.
2010-06-23 20:57:14-0700 [-] __builtin__.PoetryFactory starting on 10000
2010-06-23 20:57:14-0700 [-] Starting factory <__builtin__.PoetryFactory instance at 0x14ae8c0>
2010-06-23 20:57:14-0700 [-] loaded a poem from: poetry/ecstasy.txt

需要注意的几点:

5 你可以看到 Twisted 日志系统的输出, 包括 PoetryFactory 调用 log.msg.但是我们在 tac 文件中没有安装 logger, 所以 twistd 会帮我们安装.

6 你可以看到我们的两个主要服务 PoetryService 和 TCPServer 启动了.

7 shell 提示符不会返回. 这表明我们的服务器没有以守护进程方式运行. 默认地, twistd 会以守护进程方式运行服务器(这正是 twistd 存在的原因), 但是如果你包含 "--nodaemon" 选项,那么 twistd 将以一个常规 shell 进程的方式运行你的服务器,同时会将日志输出导向到标准输出. 这对于调试 tac 文件非常有用.

下面测试取诗服务器, 通过我们的诗歌代理或者 netcat 命令:
netcat localhost 10000

这将从服务器抓取诗歌,并且你可以看到一行如下的日志:
2010-06-27 22:17:39-0700 [__builtin__.PoetryFactory] sending 3003 bytes of poetry to IPv4Address(TCP, '127.0.0.1', 58208)

这个日志来自 PoetryProtocol.connectionMade 方法调用 log.msg.当你向服务器发送更多请求时, 你将看到更多的日志条目.
现在可以用 Ctrl-C 来终止这个服务器. 你可以看到如下输出:
^C2010-06-29 21:32:59-0700 [-] Received SIGINT, shutting down.
2010-06-29 21:32:59-0700 [-] (Port 10000 Closed)
2010-06-29 21:32:59-0700 [-] Stopping factory <__builtin__.PoetryFactory instance at 0x28d38c0>
2010-06-29 21:32:59-0700 [-] Main loop terminated.
2010-06-29 21:32:59-0700 [-] Server Shut Down.

正如你看到的, Twisted 并没有简单地崩溃, 而是优雅地关闭并将日志信息告诉你.
好啦, 现在再次启动服务器:
twistd --nodaemon --python twisted-server-3/fastpoetry.py

现在打开另一个 shell 并切换到 twisted-intro 目录. 其中有一个叫 twistd.pid 的文件. 它是被 twistd 创建的, 包含我们这个运行服务器进程号. 试一下下面的方法来关闭服务器:
kill `cat twistd.pid`

注意当服务器关闭后, twistd.pid 文件消失了, 它被 twistd 清理了.

A Real Daemon

现在让我们以守护进程的方式启动服务器, 这是 twistd 的默认方式:
twistd --python twisted-server-3/fastpoetry.py

这次我们立即看到 shell 提示符返回. 当你列出目录中的文件时,会发现除了 twistd.pid 文件, 又出现了 twistd.log 文件,它记录了之前显示在 shell 窗口的日志信息.
当启动一个守护进程时, twistd 安装一个日志管理器将条目写入一个文件而不是标准输出. 默认的日志文件是 twistd.log, 它出现在你运行 twistd 的目录中,但是你可以通过"--logfile" 来改变它的位置. twistd 安装的的日志管理器将滚动输出日志信息, 确保其不超过 1M.
你可以通过列出操作系统上的所有进程来查看正在运行的服务器. 你不妨通过取另一首诗来测试这个服务器. 你可以看到记录每个诗歌请求的新条目出现在日志文件中.
由于这个服务器不再与 shell 相连(或者除了 init 的任何其他进程), 你不能通过 Ctrl-C 关闭它. 作为一个真的守护进程, 即使你登出它也继续运行.但是你可以通过 twistd.pid 文件终止这个进程:
kill `cat twistd.pid`

随后, 关闭消息出现在日志文件中, twistd.pid 文件被移除, 服务器停止.
检查一下其他的 twistd 启动选项是个不错的主意. 例如,你可以告诉 twistd 在启动进程守护前切换到另一个用户或组账户(是一种当你的服务器不需要安全防范措施取消权限的典型方法). 我们就不进一步探讨那些额外的选项了,你可以通过 twistd 的 --help 自己研究它们.

Twisted 插件系统

现在我们已经通过 twistd 启动真正的守护进程服务器. 这非常完美,而且事实上我们的配置文件是纯 Python 源码文件,这一点为我们设置带来巨大便利. 但是我们有时用不到这样的便利性.对于诗歌服务器,我们通常只关心一小部分选项:

8 需要服务的诗歌
9 服务端口
10 监听接口

为了几个简单的变量建立一个 tac 文件显得有点小题大做. 如果我们能够通过 twistd 选项指定这些值将非常方便. Twisted 的插件系统允许我们可以这样做.
Twisted 插件通过定义 Application 提供了一种方法, 可以实现个性化的命令行选项, 进而 twistd 动态的发现和运行. Twisted 本身具有一套插件,你可以通过运行不带参数的 twistd 命令来查看它们. 现在就试一试, 在 twisted-intro 目录外. 在帮助部分后面,你可以看到如下输出:

```
...
ftp                 An FTP server.
telnet              A simple, telnet-based remote debugging service.
socks               A SOCKSv4 proxy service.
...
```

每一行显示了一个 Twisted 内置的插件, 你可以用 twistd 运行它们.
每个插件同样有它们自己的选项,你可以通过 --help 来发现它们. 让我们看看 ftp 插件有什么选项:

```
twistd ftp --help
```

注意我们需要将 --help 放在 ftp 后面而不是 twistd 后面, 因为我们想得到 ftp 的可选项. 我们可以像运行诗歌服务器一样运行 ftp 服务器. 但由于它是一个插件,我们可以仅仅通过它的名字运行:

```
twistd --nodaemon ftp --port 10001
```

以上命令以非守护进程的方式在端口 10001 上运行 ftp 插件. 注意 twistd 的 nodaemon 选项出现在插件名字的前面,插件特定选项 port 出现在插件名字的后面. 正如我们的诗歌服务器一样,你可以用 Ctrl-C 停止它.
OK, 让我们把诗歌服务器转化为 Twisted 的插件. 首先我们需要介绍一些新概念.

IPlugin

任何 Twisted 插件都需要实现 twisted.plugin.IPlugin 接口. 如果你浏览这个接口的声明, 你

会发现它没有指定任何方法. 实现 IPlugin 接口仅仅相当于一个插件在说："你好,我是插件!"以便 twistd 找到它. 当然,出于实用考虑,它需要实现一些其他接口,我们很快会介绍.
但是你怎样知道一个对象实现了一个空接口? zope.interface 包含了一个叫做 implements 的函数,它可以用来声明一个特定类实现了一个特定的接口. 我们将在插件版的诗歌服务器中看到这种使用.

IServiceMaker

除了 IPlugin,我们的插件还实现 IServiceMaker 接口. 一个实现了 IServiceMaker 接口的对象知道如何创建 IService,它将成为运行程序的核心. IServiceMaker 指定了三个属性和一个方法:

11 tapname: 代表插件名字的字符串. "tap"代表"Twisted Application Plugin". 注:老版本的 Twisted 还使用"tapfiles"文件,不过这个功能现在已经取消了.
12 description: 插件的描述, twistd 将以它作为帮助信息输出.
13 options: 一个代表这个插件接受的命令行选项的对象.
14 makeService: 一个创建 IService 对象的方法,需提供一些特定的命令行选项.

我们将在下一个版本的诗歌服务器中看到怎样将上述内容组织在一起.

Fast Poetry 3.0

现在我们已经为插件版本的"Fast Poetry"做好准备, 它位于 twisted/plugins/fastpoetry_plugin.py.
你可能注意到与其他例子不同, 我们命名了一个不同的目录. 这是因为 twistd 需要插件文件位于 twisted/plugins 目录中, 同时在你的 Python 搜索路径上. 这个目录不必是一个包(也就是, 不必包含任何 __init__.py 文件), 而且在路径上可以有多个 twisted/plugins 目录, twistd 都会找到它们. 这个插件的实际文件名是什么也没有关系, 但是一个好的方案是根据应用所代表的含义来命名, 就像我们在这里做的.
我们的插件开头部分同样包括诗歌协议,工厂,以及像 tac 文件中所实现的服务.如前所述,这些代码通常应该单独的存在于一个模块中,但出于我们例子自包含的目的,还是将它们放在插件文件中.
下面将 声明 这个插件的命令行选项:

```
class Options(usage.Options):

    optParameters = [
        ['port', 'p', 10000, 'The port number to listen on.'],
        ['poem', None, None, 'The file containing the poem.'],
        ['iface', None, 'localhost', 'The interface to listen on.'],
    ]
```

以上代码指定可以放在 twistd 命令后面使用的插件特定选项的名字.
这里就不必进一步解释上述选项的含义了,其含义很显然. 下面我们来看一下插件的主要部分 服务制造类:

```
class PoetryServiceMaker(object):
```

```
    implements(service.IServiceMaker, IPlugin)

    tapname = "fastpoetry"
    description = "A fast poetry service."
    options = Options

    def makeService(self, options):
        top_service = service.MultiService()

        poetry_service = PoetryService(options['poem'])
        poetry_service.setServiceParent(top_service)

        factory = PoetryFactory(poetry_service)
        tcp_service = internet.TCPServer(int(options['port']), factory,
                                         interface=options['iface'])

        tcp_service.setServiceParent(top_service)

        return top_service
```

这里你可以看到如何使用 zope.interface.implements 函数来声明我们的类同时实现 IServiceMaker 和 IPlugin 接口.
你应该从之前的 tac 文件辨认出 makeService 中的代码, 但是这次我们不需要自己建立一个 Application 对象, 我们仅仅创建并返回最顶层服务,这样我们的程序就可以运行, twistd 来处理其余的事情. 注意我们是如何使用 options 参数来提取插件传递给 twistd 的特定命令行选项.
定义了上述类, 还有 一步 :

```
service_maker = PoetryServiceMaker()
```

twistd 脚本会发现我们插件的实例并使用它构建最顶层服务. 与 tac 文件不同的是, 选择什么变量名没有关系, 关键是我们的对象实现了 IPlugin 和 IServiceMaker 接口.
既然已经创建了插件, 让我们运行它. 确保你位于 twisted-intro 目录中, 或者 twisted-intro 位于 Python 的搜索目录中. 下面单独运行 twistd,你会看到"fastpoetry"是列出的插件之一,后面显示插件文件中定义的描述文字.
你同样会注意到 twisted/plugins 目录中出现了一个 dropin.cache 的新文件. 这个文件由 twistd 创建, 用来加速后续扫描插件的.
现在让我们获取一些关于插件的帮助信息:

```
twistd fastpoetry --help
```

你可以看到关于 fastpoetry 插件选项的帮助性文字. 最后,运行这个插件:

```
twistd fastpoetry --port 10000 --poem poetry/ecstasy.txt
```

这将以守护进程方式启动 fastpoetry 服务器. 与前面例子一样, 你会在当期文件夹看到

twistd.pid 和 twistd.log 文件. 测试完我们的服务器, 用一下命令关闭:
kill `cat twistd.pid`

这就是如何制作 Twisted 插件的方法.

总 结

在这个部分, 我们学习了将 Twisted 服务器转换到支持长时间运行的守护进程模式. 我们还涉及了 Twisted 日志系统以及如何使用 twistd 以守护进程模式启动一个 Twisted 应用程序, 即或者通过 tac 配置文件或者 Twisted 插件. 在 第十七部分 <`p17>`_ 部分, 我们将转向异步编程的更基本的主题和另外一种结构化 Twisted 回调函数的方法.

参 考 练 习

15 修正 tac 文件以在另外一个端口服务另外一首诗. 使用另外一个 MultiService 对象以保持每首诗的服务是分离的.
16 创建一个新的 tac 文件来启动一个诗歌代理服务器.
17 修正插件文件使其可接受第二个可选诗歌文件和服务端口.
18 为诗歌代理服务器创建一个新的插件.

第十七部分：构造"回调"的另一种方法

你可以从":doc:`p01`"开始阅读；也可以浏览":doc:`index`"的索引

简介

这部分我们将回到"回调"这个主题.我们将介绍另外一种写回调函数的方法,即在 Twisted 中使用 generators. 我们将演示如何使用这种方法并且与使用"纯" Deferreds 进行对比. 最后, 我们将使用这种技术重写诗歌客户端. 但首先我们来回顾一下 generators 的工作原理,以便弄清楚它为何是创建回调的候选方法.

简要回顾生成器

你可能知道, 一个 Python 生成器是一个"可重启的函数",它是在函数体中用 yield 语句创建的. 这样做可以使这个函数变成一个"生成器函数",它返回一个"iterator"可以用来以一系列步骤运行这个函数. 每个迭代循环都会重启这个函数,继续执行到下一个 yield 语句.
生成器(和迭代器)通常被用来代表以惰性方式创建的值序列. 看一下以下文件中的代码 inline-callbacks/gen-1.py:

```
def my_generator():
    print 'starting up'
    yield 1
    print "workin'"
```

```
    yield 2
    print "still workin'"
    yield 3
    print 'done'

for n in my_generator():
    print n
```

这里我们用生成器创建了1,2,3序列. 如果你运行这些代码,会看到在生成器上做迭代时,生成器中的 print 与循环语句中的 print 语句交错出现.
以下自定义迭代器代码使上面的说法更加明显(inline-callbacks/gen-2.py):

```
def my_generator():
    print 'starting up'
    yield 1
    print "workin'"
    yield 2
    print "still workin'"
    yield 3
    print 'done'

gen = my_generator()

while True:
    try:
        n = gen.next()
    except StopIteration:
        break
    else:
        print n
```

视作一个序列,生成器仅仅是获取连续值的一个对象.但我们也可以以生成器本身的角度看问题:

1 生成器函数在被循环调用之前并没有执行(使用 next 方法).
2 一旦生成器开始运行,它将一直执行直到返回"循环"(使用 yield)
3 当循环中运行其他代码时(如 print 语句),生成器则没有运行.
4 当生成器运行时, 则循环没有运行(等待生成器返回前它被"阻滞"了).
5 一旦生成器将控制交还到循环,再启动需要等待任意可能时间(其间任意量的代码可能被执行).

这与异步系统中的回调工作方式非常类似. 我们可以把 while 循环视作 reactor, 把生成器视作一系列由 yield 语句分隔的回调函数. 有趣的是, 所有的回调分享相同的局部变量名空间, 而且名空间在不同回调中保持一致.
进一步,你可以一次激活多个生成器(参考例子 inline-callbacks/gen-3.py),使得它们的"回调"互相交错,就像在 Twisted 系统中独立运行的异步程序.

然而,这种方法还是有一些欠缺.回调不仅仅被 reactor 调用, 它还能接受信息.作为 deferred 链的一部分,回调要么接收 Python 值形式的一个结果,要么接收 Failure 形式的一个错误.
从 Python2.5开始,生成器功能被扩展了.当你再次启动生成器时,可以给它发送信息,如 inline-callbacks/gen-4.py 所示:

```
class Malfunction(Exception):
    pass

def my_generator():
    print 'starting up'

    val = yield 1
    print 'got:', val

    val = yield 2
    print 'got:', val

    try:
        yield 3
    except Malfunction:
        print 'malfunction!'

    yield 4

    print 'done'

gen = my_generator()

print gen.next() # start the generator
print gen.send(10) # send the value 10
print gen.send(20) # send the value 20
print gen.throw(Malfunction()) # raise an exception inside the generator

try:
    gen.next()
except StopIteration:
    pass
```

在 Python2.5以后的版本中, yield 语句是一个计算值的表达式.重新启动生成器的代码可以使用 send 方法代替 next 决定它的值(如果使用 next 则值为 None), 而且你还可以在迭代器内部使用 throw 方法抛出任何异常. 是不是很酷?

内联回调

根据我们刚刚回顾的可以向生成器发送值或抛出异常的特性,可以设想它是像 deferred 中的一系列回调,即可以接收结果或错误. 每个回调被 yield 分隔,每一个 yield 表达式的值是

```
def my_generator(arg1, arg2):
  blah = blah * arg1                    | First Callback
  blah2 = blahblah(blah)                |
  result = yield blah * 3               |

  foo = result + 7                      | Second Callback
  result = yield something()            |

  try:                                  |
    something_else(result)              | Third Callback
  except BadThings:                     |
    handle_bad_things(arg2)             |
```

下一个回调的结果(或者 yield 抛出异常表示错误).图35显示相应概念:

图35:作为回调序列的生成器

现在一系列回调以 deferred 方式被链接在一起,每个回调从它前面的回调接收结果.生成器很容易做到这一点——当再次启动生成器时,仅仅使用 send 发送上一次调用生成器的结果(yield 产生的值).但这看起来有点笨,既然生成器从开始就计算这个值,为什么还需要把它发送回来? 生成器可以将这个值储存在一个变量中供下一次使用. 因此这到底是为什么呢?

回忆一下我们在 :doc:`p13` 中所学, deferred 中的回调还可以返回 deferred 本身. 在这种情况下, 外层的 deferred 先暂停等待内层的 deferred 激发,接下来外层 deferred 链使用内层 deferred 的返回结果(或错误)激发后续的回调(或错误回调).

所以设想我们的生成器生成一个 deferred 对象而不是一个普通的 Python 值. 这时生成器会自动"暂停";生成器总是在每个 yield 语句后暂停直到被显示的重启.因而我们可以延迟它的重启直到 deferred 被激发, 届时我们会使用 send 方法发送值(如果 deferred 成功)或者抛出异常(如果 deferred 失败).这就使我们的生成器成为一个真正的异步回调序列,这正是 twisted.internet.defer 中 inlineCallbacks 函数背后的概念.

进一步讨论内联回调

考虑以下例程, 位于 inline-callbacks/inline-callbacks-1.py:

```
from twisted.internet.defer import inlineCallbacks, Deferred

@inlineCallbacks
def my_callbacks():
    from twisted.internet import reactor

    print 'first callback'
    result = yield 1 # yielded values that aren't deferred come right back

    print 'second callback got', result
    d = Deferred()
    reactor.callLater(5, d.callback, 2)
    result = yield d # yielded deferreds will pause the generator
```

```
    print 'third callback got', result # the result of the deferred

    d = Deferred()
    reactor.callLater(5, d.errback, Exception(3))

    try:
        yield d
    except Exception, e:
        result = e

    print 'fourth callback got', repr(result) # the exception from the deferred

    reactor.stop()

from twisted.internet import reactor
reactor.callWhenRunning(my_callbacks)
reactor.run()
```

运行这个例子可以看到生成器运行到最后并终止了 reactor, 这个例子展示了 inlineCallbacks 函数的很多方面.首先, inlineCallbacks 是一个修饰符,它总是修饰生成器函数, 如那些使用 yield 语句的函数. inlineCallbacks 的全部目的是将一个生成器按照上述策略转化为一系列异步回调.

第二,当我们调用一个用 inlineCallbacks 修饰的函数时,不需要自己调用 send 或 throw 方法.修饰符会帮助我们处理细节,并确保生成器运行到结束(假设它不抛出异常).

第三,如果我们从生成器生成一个非延迟值,它将以 yield 生成的值立即重启.

最后,如果我们从生成器生成一个 deferred,它不会重启除非此 deferred 被激发.如果 deferred 成功返回,则 yield 的结果就是 deferred 的结果.如果 deferred 失败了,则 yield 会抛出异常. 注这个异常仅仅是一个普通的 Exception 对象,而不是 Failure,我们可以在 yield 外面用 try/except 块捕获它们.

在上面的例子中,我们仅用 callLater 在一小段时间之后去激发 deferred.虽然这是一种将非阻塞延迟放入回调链的实用方法,但通常我们会生成一个 deferred,它是被生成器中其他的异步操作(如 get_poetry)返回的.

OK,现在我们知道了 inlineCallbacks 修饰的函数是如何运行的,但当你实际调用时会返回什么值呢?正如你认为的,将得到 deferred.由于不能确切地知道生成器何时停止(它可能生成一个或多个 deferred),装饰函数本身是异步的,所以 deferred 是一个合适的返回值.注:这个返回的 deferred 不是生成器中 yield 生成的 deferred.相反,它在生成器完全结束(或抛出异常)后才被激发.

如果生成器抛出一个异常,那么返回的 deferred 将激发它的错误回调链,把异常包含在一个 Failure 中. 但是如果我们希望生成器返回一个正常值,必须使用 defer.returnValue 函数. 就像普通 return 语句一样,它也会终止生成器(实际会抛出一个特殊异常).例子 inline-callbacks/inline-callbacks-2.py 说明了这两种可能.

客户端 7.0

让我们在新版本的诗歌客户端中加入 inlineCallbacks,你可以在 twisted-client-7/get-poetry.py 中查看源代码.也许你需要与客户端6.0—— twisted-client-6/get-poetry.py 进行对比,它们的相对变化位于 poetry_main:

```
def poetry_main():
    addresses = parse_args()

    xform_addr = addresses.pop(0)

    proxy = TransformProxy(*xform_addr)

    from twisted.internet import reactor

    results = []

    @defer.inlineCallbacks
    def get_transformed_poem(host, port):
        try:
            poem = yield get_poetry(host, port)
        except Exception, e:
            print >>sys.stderr, 'The poem download failed:', e
            raise

        try:
            poem = yield proxy.xform('cummingsify', poem)
        except Exception:
            print >>sys.stderr, 'Cummingsify failed!'

        defer.returnValue(poem)

    def got_poem(poem):
        print poem

    def poem_done(_):
        results.append(_)
        if len(results) == len(addresses):
            reactor.stop()

    for address in addresses:
        host, port = address
        d = get_transformed_poem(host, port)
        d.addCallbacks(got_poem)
        d.addBoth(poem_done)
```

```
    reactor.run()
```

在这个新版本里, inlineCallbacks 生成函数 get_transformed_poem 负责取回诗歌并且应用变换(通过变换服务).由于这两个操作都是异步的,我们每次生成一个 deferred 并且隐式地等待结果.与客户端 6.0 一样,如果变换失败则返回原始诗歌.我们可以使用 try/except 语句捕获生成器中的异步错误.
我们以先前的方式测试新版客户端. 首先启动一个变换服务:
python twisted-server-1/tranformedpoetry.py --port 10001

然后启动两个诗歌服务器:
python twisted-server-1/fastpoetry.py --port 10002 poetry/fascination.txt
python twisted-server-1/fastpoetry.py --port 10003 poetry/science.txt

现在可以运行新的客户端:
python twisted-client-7/get-poetry.py 10001 10002 10003

试试关闭一个或多个服务器,看一看客户端如何捕获错误.

讨论

就像 Deferred 对象, inlineCallbacks 函数给我们一种组织异步回调的新方式.同时,如同使用 deferred, inllineCallbacks 没有改变游戏规则.特别地,我们的回调仍然一次调用一个回调,它们仍然被 reactor 调用.我们可以通过打印内联回调的回溯跟踪信息来证实这一点,参见脚本 inline-callbacks/inline-callbacks-tb.py.运行此代码你将首先获得一个关于 reactor.run() 的回溯,然后是许多帮助函数信息,最后是我们的回调.
图29解释了当 deferred 中一个回调返回另一个 deferred 时会发生什么,我们调整它来展示当一个 inlineCallbacks 生成器生成一个 deferred 时会发生什么,参考图36:

图36: inlineCallbacks 函数中的控制流
同样的图对两种情况都适用,因为它们表示的想法都是一样的 —— 一个异步操作正在等待另一个.
由于 inlineCallbacks 和 deferred 解决许多相同的问题,在它们之间如何选择呢?下面列出一些 inlineCallbacks 的潜在优势.

- 由于回调分享相同的名空间,因此没有必要传递额外状态.
- 回调的顺序很容易看到,因为它总是从上到下执行.
- 节省了每个回调函数的声明和隐式控制流,通常减少输入.
- 可以使用熟悉的 try/except 语句处理错误.

当然也存在一些缺陷:

- 生成器中的回调不能被单独调用,这使代码重用比较困难.而构造 deferred 的代码则能够以任意顺序自由地添加任何回调.
- 生成器的紧致性可能混淆一个事实,其实异步回调非常晦涩.尽管生成器看起来像一个普通的函数序列,但是它的行为却非常不一样. inlineCallbacks 函数不是一种避免学习异步编程模型的方式.

就像任何技术,实践将积累出必要的经验,帮你做出明智选择.

总结

在这个部分,我们学习了 inlineCallbacks 装饰符以及它怎样使我们能够以 Python 生成器的形式表达一系列异步回调.
在 :doc:`p18` 中,我们将学习一种管理 **一组** "并行"异步操作的技术.

参考练习

6 为什么 inlineCallbacks 函数是复数(形式)?
7 研究 inlineCallbacks 的实现以及它们帮助函数 _inlineCallbacks. 并思考短语"魔鬼在细节处".
8 有 N 个 yield 语句的生成器中包含多少个回调,假设其中没有循环或者 if 语句?
9 诗歌客户端7.0可能同时运行三个生成器.概念上,它们之间有多少种不同的交错方式?考虑诗歌客户端和 inlineCallbacks 的实现,你认为实际有多少种可能?
10 把客户端7.0中的 got_poem 放入到生成器中.
11 把 poem_done 回调放入生成器.小心!确保处理所有失败情况以便无论怎样 reactor 都会关闭.与使用 deferred 关闭 reactor 对比代码有何不同?
12 一个在 while 循环中使用 yield 语句的生成器代表一个概念上的无限序列.那么同样的装饰有 inlineCallbacks 的生成器又代表什么呢?

第十八部分：**Deferreds** 全貌

你可以从 ":doc:`p01`" 开始阅读；也可以浏览 ":doc:`index`" 的索引

简介

在上一个部分,我们学习了使用生成器构造顺序异步回调的新方法.这样,包括 deferreds,我们现在有两种将异步操作链接在一起的方法.
有时,然而,我们需要"并行"的运行一组异步操作.由于 Twisted 是单线程的,它实际并不会并发运行,但我们希望使用异步 I/O 在一组任务上尽 可能快的工作.以我们的诗歌客户端为例,它从多个服务器同时下载诗歌,而不是一个接一个的方式.这就是使用 Twisted 下载诗歌的全部特点.
作为一个结果,所有诗歌客户端需要解决一个问题:你怎样得知你启动的所有异步操作已经完成?目前我们通过将结果集总到一个列表(如客户端 7.0中的 结果 列表)并检查这个列表的长度来解决这个问题.除了收集成功的结果,我们还必须小心地对待失败,否则一个失败将使程序进入死循环,以为还有工作需要做.
正如你所料,Twisted 包含一个抽象层可以用来解决这个问题,我们来看一看.

DeferredList

DeferredList 类使我们可以将一个 defered 对象列表视为一个 defered 对象.通过这种方法我们启动一族异步操作并且在它们全部完成后获得通知(无论它们成功或者失败).让我们看一些例子.

在 deferred-list/deferred-list-1.py 中,可以找到如下代码:

```
from twisted.internet import defer

def got_results(res):
    print 'We got:', res

print 'Empty List.'
d = defer.DeferredList([])
print 'Adding Callback.'
d.addCallback(got_results)
```

如果运行它,将得到如下输出:

```
Empty List.
Adding Callback.
We got: []
```

注意以下几点:

- DeferredList 由 Python 列表创建.在这种情况下,列表是空的,但我们很快将看到列表元素必须是 Deferred 对象.
- DeferredList 本身是一个 deferred (它继承 Deferred).这意味着你可以像对待普通 deferred 一样向其添加回调和错误回调.
- 在以上例子中,回调被添加时立即激发,所以 DeferredList 也必须立即激发.我们一会儿将讨论.
- deferred 列表的结果本身也是一个列表(空).

下面看一下 deferred-list/deferred-list-2.py:

```
from twisted.internet import defer

def got_results(res):
    print 'We got:', res

print 'One Deferred.'
d1 = defer.Deferred()
d = defer.DeferredList([d1])
print 'Adding Callback.'
d.addCallback(got_results)
print 'Firing d1.'
d1.callback('d1 result')
```

现在我们创建了包含一个 deferred 元素的 DeferredList 列表,得到如下输出:

```
One Deferred.
Adding Callback.
Firing d1.
We got: [(True, 'd1 result')]
```

注意以下几点:

- 这次 DeferredList 没有激发它的回调,直到我们激发列表中的 deferred.
- 结果同样是一个列表,但这次包含一个元素.
- 这个元素是一个元组,它的第二个值是列表中 deferred 的结果.

让我们向列表添加两个 deferreds (deferred-list/deferred-list-3.py):

```
from twisted.internet import defer

def got_results(res):
    print 'We got:', res

print 'Two Deferreds.'
d1 = defer.Deferred()
d2 = defer.Deferred()
d = defer.DeferredList([d1, d2])
print 'Adding Callback.'
d.addCallback(got_results)
print 'Firing d1.'
d1.callback('d1 result')
print 'Firing d2.'
d2.callback('d2 result')
```

得到如下输出:

```
Two Deferreds.
Adding Callback.
Firing d1.
Firing d2.
We got: [(True, 'd1 result'), (True, 'd2 result')]
```

现在 DeferredList 的结果非常清晰,至少以我们的使用方式,它是一个列表,元素个数与传入构造器的 deferred 列表元素个数相同. 而且结果列表的元素包含原始的 deferreds 结果信息,至少当这些 deferred 成功返回.这意味着 DeferredList 本身并不激发直到所有的原始列表中的 deferreds 都被激发. 而且以一个空列表创建的 DeferredList 会立即激发,因为它不需要等待任何 deferreds.

那么最终结果列表中的元素顺序如何? 考虑以下代码(deferred-list/deferred-list-4.py):

```
from twisted.internet import defer

def got_results(res):
    print 'We got:', res
```

```
print 'Two Deferreds.'
d1 = defer.Deferred()
d2 = defer.Deferred()
d = defer.DeferredList([d1, d2])
print 'Adding Callback.'
d.addCallback(got_results)
print 'Firing d2.'
d2.callback('d2 result')
print 'Firing d1.'
d1.callback('d1 result')
```

这里我们先激发 d2 然后再激发 d1,注意构造参数中的 deferred 列表里 d1, d2 仍是原先的顺序.输出结果如下:

```
Two Deferreds.
Adding Callback.
Firing d2.
Firing d1.
We got: [(True, 'd1 result'), (True, 'd2 result')]
```

输出列表中结果的顺序与原始 deferred 列表顺序相对应,而不是 deferred 碰巧被激发的顺序.这一点非常好,因为我们可以很容易地将每个结果与生成它的相应的操作联系在一起(如哪首诗来自哪个服务器).

好了,那如果列表中一个或多个 deferreds 失败了怎么办呢? 上面结果中的 True 有什么用? 再看一个例子(deferred-list/deferred-list-5.py):

```
from twisted.internet import defer

def got_results(res):
    print 'We got:', res

d1 = defer.Deferred()
d2 = defer.Deferred()
d = defer.DeferredList([d1, d2], consumeErrors=True)
d.addCallback(got_results)
print 'Firing d1.'
d1.callback('d1 result')
print 'Firing d2 with errback.'
d2.errback(Exception('d2 failure'))
```

现在我们以正常结果激发 d1,以错误激发 d2.先暂时忽略 consumerErrors 选项,稍候介绍.这里是输出结果:

```
Firing d1.
Firing d2 with errback.
We got: [(True, 'd1 result'), (False, <twisted.python.failure.Failure <type 'exceptions.Exception'>>)]
```

这次对应 d2 的元组在第二个位置出现了一个 Failure,并且第一个位置是 False.至此 DeferredList 的工作原理非常清晰(但继续浏览以下讨论):

- DeferredList 是以一个 deferred 对象列表创建的.
- DeferredList 本身是一个 deferred,它返回的结果是一个列表,长度与 deferred 列表相同.
- 当原始列表中所有 deferred 被激发后, DeferredList 将会被激发.
- 结果列表中的每个元素以相同顺序对应原始列表中相应的 deferred.如果那个 deferred 成功返回,相应元素是(True,result),如果失败则为(False,failure).
- DeferredList 不会失败,因为无论每个 deferred 的返回结果是什么都会被集总到结果列表中(同样,请看下面讨论).

现在让我们讨论一下被传入 DeferredList 的 consumeErrors 选项,如果我们运行以上相同代码而不传入此选项(deferred-list/deferred-list-6.py),则得到以下输出:

```
Firing d1.
Firing d2 with errback.
We got: [(True, 'd1 result'), (False, >twisted.python.failure.Failure >type 'exceptions.Exception'<<)]
Unhandled error in Deferred:
Traceback (most recent call last):
Failure: exceptions.Exception: d2 failure
```

如果你还记得,"Unhandled error in Deferred"消息是在 deferred 垃圾回收时被生成的,而且它表示最后一个回调失败了.这个消息告诉我们并没有完全捕获潜在的异步错误.在我们例子中,它是从哪里来的呢? 很明显不是来自 DeferredList,因为它已经成功返回了.所以它一定是来自 d2.

DeferredList 需要知道它所监视的 deferred 何时激发. DeferredList 以通常的方式向每个 deferred 添加一个回调和错误回调. 默认地,这个回调(或错误)返回原始结果(或错误)在将它们放入最终结果列表之后.由于错误回调返回原始 failure 后将触发下一个错误回调, d2 在它被激发后仍然保持失败状态.

但是如果我们将 consumeErrors=True 传递给 DeferredList, 它将向每个 deferred 添加返回 None 的错误回调, 即"消耗"掉这个错误并且取消警告信息. 我们同样可以向 d2 添加自己的错误回调来处理错误,如 deferred-list/deferred-list-7.py.

客户端 8.0

获取诗歌客户端 8.0 发布啦! 客户端使用 DeferredList 去发现所有诗歌何时完成(或失败). 新版客户端位于 twisted-client-8/get-poetry.py. 同样,唯一的变化在于 poetry_main, 我们来看一下重要的变化:

```
...
ds = []

for (host, port) in addresses:
    d = get_transformed_poem(host, port)
    d.addCallbacks(got_poem)
    ds.append(d)
```

```
dlist = defer.DeferredList(ds, consumeErrors=True)
dlist.addCallback(lambda res : reactor.stop())
```

你可以与 客户端 7.0 中的相应部分比较.
在客户端 8.0中,我们不需要 poem_done 回调和 results 列表.相反,我们把每个从 get_transformed_poem 返回的 deferred 放入 ds 列表,之后创建一个 DeferredList.由于 DeferredList 不会在所有诗歌完成或失败之前激发,我们仅仅向 DeferredList 添加一个回调以便关闭 reactor. 在我们这个情况中,没有使用 DeferredList 返回的结果,我们仅仅需要知道所有事情何时结束.仅此而已!

讨论

可视化 DeferredList 的工作方式:

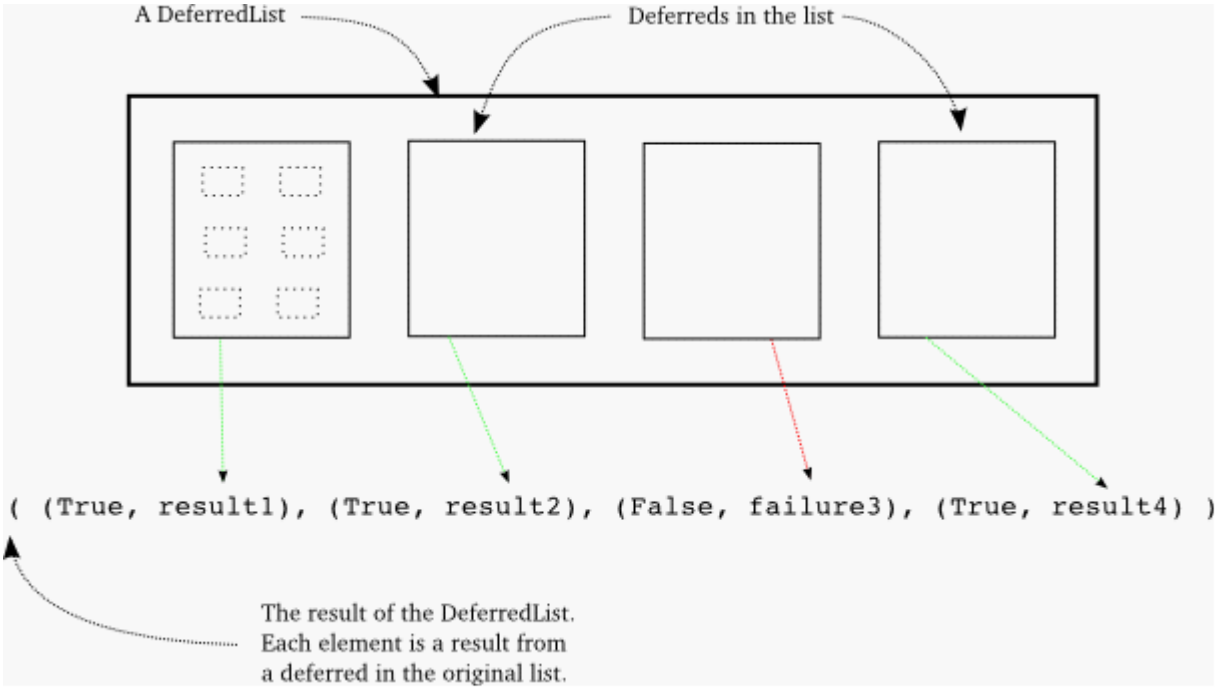


图37: DeferredList 的结果
非常简单,真的. 还有一些关于 DeferredList 的选项我们没有涉及,以及那些改变我们以上所描述行为的选项.我们在参考练习中把这些留给读者自己探索.
在 :doc:`p19` 中我们将进一步介绍 Deferred 类, 包括 Twisted 10.1.0 提出的最新特性.

参考练习

1 阅读 DeferredList 的源代码.
2 修改 deferred-list 中的例子去实现可选的构造器参数 fireOnOneCallback 和 fireOnOneErrback. 实现你将用其中一个(或两个都使用)的情景.
3 你可以使用 DeferredLists 列表创建一个 DeferredList 吗? 如果是这样,结果将是什么?
4 修改客户端8.0在所有诗歌完成下载前不打印任意信息. 这次你将使用 DeferredList 的结果.

5 定义 DeferredDict 的句法并且实现它.

第十九部分：改变之前的想法

你可以从 ":doc:`p01`" 开始阅读；也可以从 ":doc:`index`" 浏览索引.

简介

Twisted 是一个正在进展的项目,它的开发者会定期添加新的特性并且扩展旧的特性.
随着 Twisted 10.1.0发布,开发者向 Deferred 类添加了一个新的特性—— cancellation ——这正是我们今天要研究的.
异步编程将请求和响应解耦了,如此又带来一个新的可能性:在请求结果和返回结果之间,你可能决定不再需要这个结果了.考虑一下 :doc:`p14` 中的诗歌代理服务器.下面是这个如何工作的,至少对于诗歌的第一次请求:

1 一个对诗歌的请求来了.
2 这个代理联系实际服务器以得到这首诗
3 一旦这首诗完成,将其发送给原发出请求的代理

看起来非常完美,但是如果客户端在获得诗歌之前挂了怎么办?也许它们先前请求 Paradise Lost 的全部内容,随后它们决定实际想要的是 Kojo 的俳句.我们的代理将陷入下载前者,并且那个慢服务器会等好一会.最好的策略便是关闭连接,让慢服务器回去顺觉.
回忆一下 :ref:`figure15`,展示了同步程序控制流的概念.在那张图中我们可以看到函数调用自上而下,异常是自下而上.如果我们希望取消一个同步调用(这仅是假设),控制流的传递方向与函数调用的方向一致,都是从高层传向底层,如图38所示:

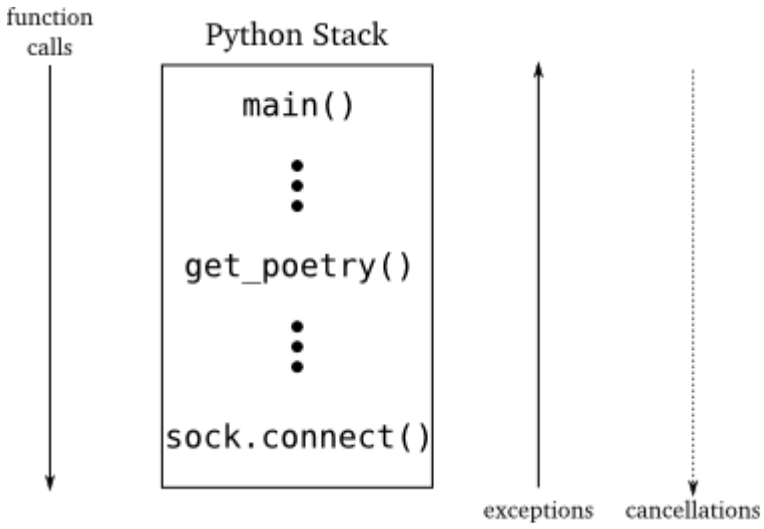


图38:同步程序流,含假想取消操作
当然,在同步程序中这是不可能的,因为高层的代码在底层操作结束前没有恢复运行,自然也就没有什么可取消的.但是在异步程序中,高层代码在底层代码完成前具有控制权,至少具有在底层代码完成之前取消它的请求的可能性.

在Twisted程序中,底层请求被包含在一个 Deferred 对象中,你可以将其想象为一个外部异步操作的"句柄". deferred 中正常的信息流是向下的,从底层代码到高层代码,与同步程序中返回的信息流方向一致.从 Twisted 10.1.0开始,高层代码可以反向发送信息 —— 它可以告诉底层代码它不再需要其结果了.如图39:

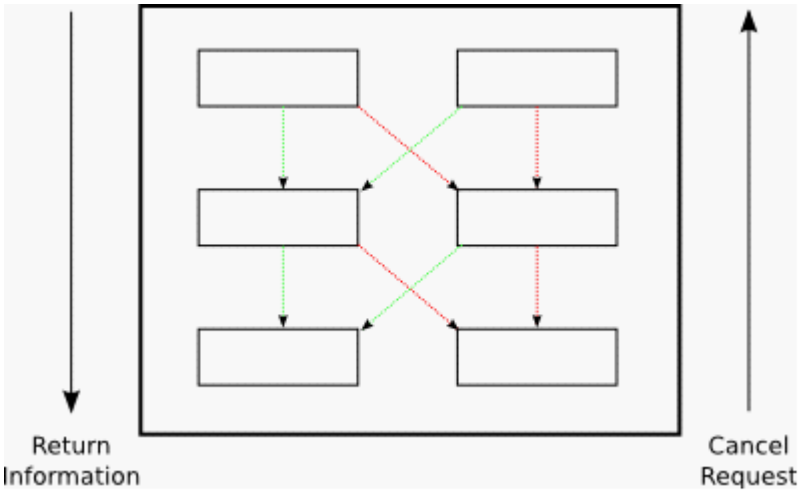


图39： deferred 中的信息流,包含取消

取消 Deferreds

让我们看一些例程,来了解下取消 deferreds 的实际工作原理.注:为了运行这些列子以及本部分中的其他代码,你需要安装 Twisted 10.1.0 或更高 版本. 考虑 deferred-cancel/defer-cancel-1.py:

```
from twisted.internet import defer

def callback(res):
    print 'callback got:', res

d = defer.Deferred()
d.addCallback(callback)
d.cancel()
print 'done'
```

伴随着新的取消特性, Deferred 类获得一个名为 cancel 的新方法.上面代码创建了一个新的deferred,添加了一个回调,这后取消了这个 deferred 而没有激发它.输出如下:

```
done
Unhandled error in Deferred:
Traceback (most recent call last):
Failure: twisted.internet.defer.CancelledError:
```

OK,取消一个 deferred 看起来像使错误回调链运行,常规的回调根本没有被调用.同样注意到这个错误是: twisted.internet.defer.CancelledError,一个意味着 deferred 被取消的个性化异常(但请继续阅读).让我们添加一个错误回调,如 deferred-cancel/defer-cancel-2.py

```
from twisted.internet import defer
```

```
def callback(res):
    print 'callback got:', res

def errback(err):
    print 'errback got:', err

d = defer.Deferred()
d.addCallbacks(callback, errback)
d.cancel()
print 'done'
```

得到以下输出:

```
errback got: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
done
```

所以我们可以'捕获'从 cancel 产生的错误回调,就像其他 deferred 错误一样.
OK,让我们试试激发 deferred 然后取消它,如 deferred-cancel/defer-cancel-3.py:

```
from twisted.internet import defer

def callback(res):
    print 'callback got:', res

def errback(err):
    print 'errback got:', err

d = defer.Deferred()
d.addCallbacks(callback, errback)
d.callback('result')
d.cancel()
print 'done'
```

这里我们用常规 callback 方法激发 deferred,之后取消它.输出结果如下:

```
callback got: result
done
```

我们的回调被调用(正如我们所预期的)之后程序正常结束,就像 cancel 根本没有被调用.所以取消一个 deferred 好像根本没有效果如果它已经被激发(但请继续阅读!).
如果我们在取消 deferred 之后激发它会怎样?参看 deferred-cancel/defer-cancel-4.py:

```
from twisted.internet import defer

def callback(res):
    print 'callback got:', res
```

```
def errback(err):
    print 'errback got:', err

d = defer.Deferred()
d.addCallbacks(callback, errback)
d.cancel()
d.callback('result')
print 'done'
```

这种情况的输出如下:

```
errback got: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
done
```

有意思!与第二个例子的输出一样,当时没有激发 deferred.所以如果 deferred 被取消了,再激发它没有效果.但是为什么 d.callback('result') 没有产生错误,考虑到不能激发 deferred 大于一次,错误回调链为何没有运行?

再次考虑 figure39.用结果或失败激发一个 deferred 是底层代码的工作,然而取消 deferred 是高层代码的行为.激发 deferred 意味着"这是你的结果",然而取消 deferred 意味着"我不再想要这个结果了".同时记住 cancel 是一个新特性,所以大部分现有的 Twisted 代码并没有处理取消的操作.但是 Twisted 的开发者使我们取消 deferred 的想法变得有可能,甚至包括那些在 Twisted 10.1.0之前写的代码.

为了实现以上想法, cancel 方法实际上做两件事:

4 告诉 Deferred 对象本身你不想要那个结果,如果它还没有返回(如, deferred 没有被激发),这样忽略任何回调或错误回调的后续调用.

5 同时,可选地,告诉正在产生结果的底层代码需要采取何种步骤来取消操作.

由于旧版本的 Twisted 代码会上前去激发任何已经被取消的 deferred, step#1确保我们的程序不会垮掉如果我们取消一个旧有库中的 deferred.

这意味着我们可以随心所欲地取消一个 deferred,同时可以确定不会得到结果如果它还没有到来(甚至那些 **将要** 到来的).但是取消 deferred 可能并没有取消异步操作.终止一个异步操作需要一个上下文的具体行动.你可能需要关闭网络连接,回滚数据库事务,结束子进程,等等.由于 deferred 仅仅是一般目的的回调组织者,它怎么知道具体要做什么当你取消它时?或者,换种说法,它怎样将 cancel 请求传递给首先已经创建和返回了 deferred 的底层代码? 和我一起说:

I know, with a callback!

本质上取消 Deferreds

好吧,首先看一下 deferred-cancel/defer-cancel-5.py:

```
from twisted.internet import defer

def canceller(d):
```

```
    print "I need to cancel this deferred:", d

def callback(res):
    print 'callback got:', res

def errback(err):
    print 'errback got:', err

d = defer.Deferred(canceller) # created by lower-level code
d.addCallbacks(callback, errback) # added by higher-level code
d.cancel()
print 'done'
```

这个例子基本上跟第二个例子相同,除了有第三个回调(canceller).这个回调是我们在创建 Deferred 的时候传递给它的,不是之后添加的.这个回调负责执行终止异步操作时所需的上下文相关的具体操作(当然,仅当 deferred 被实际取消). canceller 回调是返回 deferred 的底层代码的必要部分,不是接收 deferred 的高层代码为其自己添加的回调和错误回调.
运行这个例子将产生如下输出:

```
I need to cancel this deferred: <Deferred at 0xb7669d2cL>
errback got: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
done
```

正如你所看到, 不需要返回结果的 deferred 被传递给 canceller 回调.在这里我们可以做任何需要做的事情以便彻底终止异步操作.注意 canceller 在错误回调链激发前被调用.其实我们可以在取消回调中选择使用任何结果或错误自己激发 deferred (这样就会优先于 CancelledError 失败). 这两种情况在 deferred-cancel/defer-cancel-6.py 和 deferred-cancel/defer-cancel-7.py 中进行了说明.
在激发 reactor 之前先做一个简单的测试.我们将使用 canceller 回调创建一个 deferred,正常的激发它,之后取消它.你可以在 deferred-cancel/defer-cancel-8.py 中看到代码.通过检查那个脚本的输出,你将看到取消一个被激发的 deferred 不会调用 canceller 回调.这正是我们所要的,因为没什么可取消的.
我们目前看到的例子都没有实际的异步操作. 让我们构造一个调用异步操作的简单程序,之后我们将指出如何使那个操作可取消.
参见代码 deferred-cancel/defer-cancel-9.py:

```
from twisted.internet.defer import Deferred

def send_poem(d):
    print 'Sending poem'
    d.callback('Once upon a midnight dreary')

def get_poem():
    """Return a poem 5 seconds later."""
```

```
    from twisted.internet import reactor
    d = Deferred()
    reactor.callLater(5, send_poem, d)
    return d

def got_poem(poem):
    print 'I got a poem:', poem

def poem_error(err):
    print 'get_poem failed:', err

def main():
    from twisted.internet import reactor
    reactor.callLater(10, reactor.stop) # stop the reactor in 10 seconds

    d = get_poem()
    d.addCallbacks(got_poem, poem_error)

    reactor.run()

main()
```

这个例子中包含了一个 get_poem 函数,它使用 reactor 的 callLater 方法在被调用 5 秒钟后异步地返回一首诗.主函数调用 get_poem,添加一个回调/错误回调对,之后启动 reactor.我们(同样使用 callLater)安排 reactor 在 10 秒钟之后停止.通常我们向 deferred 添加一个回调来实现,但你很快就会知道我们为何这样做.
运行程序(适当延迟后)产生如下输出:

```
Sending poem
I got a poem: Once upon a midnight dreary
```

10 秒钟后程序终止.现在来试试在诗歌被发送前取消 deferred.只需加入以下代码在 2 秒钟后取消(在 5 秒钟延迟发送诗歌之前):

```
reactor.callLater(2, d.cancel) # cancel after 2 seconds
```

完整的例子参见 deferred-cancel/defer-cancel-10.py,这将产生如下输出:

```
get_poem failed: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
Sending poem
```

这个例子清晰地展示了取消一个 deferred 并没有取消它背后的异步请求.2秒钟后我们看到了错误回调输出,打印出如我们所料的 CancelledError 错误.但是5秒钟后我们看到了 send_poem 的输出(但是这个 deferred 上的回调并没有激发).

这时我们与 deferred-cancel/defer-cancel-4.py 的情况一样."取消" deferred 仅仅是使最终结果被忽略,但实际上并没有终止这个操作.正如我们上面所学,为了得到一个真正可取消的deferred,必须在它被创建时添加一个 cancel 回调.
那么这个新的回调需要做什么呢? 参考一下关于 callLater 方法的 文档. 它的返回值是另一个实现了 IDelayedCall 的对象,用 cancel 方法我们可以阻止延迟的调用被执行.
这非常简单,更新后的代码参见 deferred-cancel/defer-cancel-11.py.所有相关变化都在 get_poem 函数中:

```
def get_poem():
    """Return a poem 5 seconds later."""

def canceler(d):
    # They don't want the poem anymore, so cancel the delayed call
    delayed_call.cancel()

    # At this point we have three choices:
    #   1. Do nothing, and the deferred will fire the errback
    #        chain with CancelledError.
    #   2. Fire the errback chain with a different error.
    #   3. Fire the callback chain with an alternative result.

d = Deferred(canceler)

from twisted.internet import reactor
delayed_call = reactor.callLater(5, send_poem, d)

return d
```

在这个新版本中,我们保存 callLater 的返回值以便能够在 cancel 回调中使用. cancel 回调的唯一工作是调用 delayed_call.cancel(). 但是正如之前讨论的,我们可以选择激发自定义的deferred. 最新版本的程序产生如下输出:

```
get_poem failed: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
```

正如你看到的, deferred 被取消了并且异步操作被真正地终止了(我们看不到 send_poem 的输出了).

诗歌代理 3.0

正如在简介中所讨论,诗歌代理服务器是实现取消的很好的候选者,因为这可以让我们取消诗歌下载如果事实证明没有人想要它(如客户端已经在我们发送诗歌前关闭了连接).版本 3.0 的代理位于 twisted-server-4/poetry-proxy.py, 实现了 deferred 取消. 变化首先位于 PoetryProxyProtocol:

```
class PoetryProxyProtocol(Protocol):
```

```
    def connectionMade(self):
        self.deferred = self.factory.service.get_poem()
        self.deferred.addCallback(self.transport.write)
        self.deferred.addBoth(lambda r: self.transport.loseConnection())

    def connectionLost(self, reason):
        if self.deferred is not None:
            deferred, self.deferred = self.deferred, None
            deferred.cancel() # cancel the deferred if it hasn't fired
```

你可以与 旧版本 对比一下.两个主要的变化是:

6 保存我们从 get_poem 得到的 deferred,以便之后在需要时取消它.

7 当连接关闭时取消 deferred.注这个操作同样会取消 deferred 当我们实际得到诗歌之后,但正如前例所发现的,取消一个被激发的 deferred 不会有任何效果.

现在我们需要确保取消 deferred 将实际终止诗歌的下载. 所以我们需要改变 ProxyService:

```
class ProxyService(object):

    poem = None # the cached poem

    def __init__(self, host, port):
        self.host = host
        self.port = port

    def get_poem(self):
        if self.poem is not None:
            print 'Using cached poem.'
            # return an already-fired deferred
            return succeed(self.poem)

        def canceler(d):
            print 'Canceling poem download.'
            factory.deferred = None
            connector.disconnect()

        print 'Fetching poem from server.'
        deferred = Deferred(canceler)
        deferred.addCallback(self.set_poem)
        factory = PoetryClientFactory(deferred)
        from twisted.internet import reactor
        connector = reactor.connectTCP(self.host, self.port, factory)
        return factory.deferred

    def set_poem(self, poem):
        self.poem = poem
```

return poem

同样,可以与 旧版本 对比一下. 这个类具有一些新的变化:

8 我们保存 reactor.connetTCP 的返回值,一个 IConnector 对象.我们可以使用这个对象上的 disconnect 方法关闭连接.

9 我们创建带 canceler 回调的 deferred.那个回调是一个闭包,它使用 connector 关闭连接. 但首先须设置 factory.deferred 属性为 None. 否则,工厂会以 "连接关闭"错误回调激发 deferred 而不是以 CancelledError 激发. 由于 deferred 已经被取消了, 以 CancelledError 激发更加合适.

你同样会注意到我们是在 ProxyService 中创建 deferred 而不是 PoetryClientFactory. 由于 canceler 回调需要获取 IConnector 对象, ProxyService 成为最方便创建 deferred 的地方.
同时,就像我们之前的例子, canceler 回调作为一个闭包实现.闭包看起来在取消回调的实现上非常有用.
让我们试试新的代理.首先启动一个慢服务器.它需要很慢以便我们有时间取消:
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt

现在可以启动代理(记住你需要 Twisted 10.1.0):
python twisted-server-4/poetry-proxy.py --port 10000 10001

现在我们可以用任何客户端从代理下载一首诗,或者仅使用 curl:
curl localhost:10000

几秒钟后,按 Ctrl-C 停止客户端或者 curl 进程. 在终端运行代理你将看到如下输出:
Fetching poem from server.
Canceling poem download.

你应该看到慢服务器已经停止了向输出打印它所发送诗歌的片段,因为我们的代理挂了.
你可以多次启动和停止客户端来证实每个下载每次都被取消了.但是如果你让整首诗运行完,那么代理将缓存它并且在此之后立即发送它.

另一个难点

以上我们曾不止一次说取消一个已经激发的 deferred 是没有效果的.然而,这不是十分正确. 在 :doc:`p13` 中,我们学习了附加给一个 deferred 的回调和错误回调也可能返回另一个 deferred.在那种情况下,原始的(外层) deferred 暂停执行它的回调链并且等待内层 deferred 激发(参见 `figure28`_).
如此, 即使一个 deferred 激发了发出异步请求的高层代码,它也不能接收到结果,因为在等待内层 deferred 完成之前回调链暂停了. 所以当高层代码取消这个外部 deferred 时会发生什么情况呢? 在这种情况下,外部 deferred 不仅仅是取消它自己(它已经激发了);相反地,这个 deferred 取消内部的 deferred.
所以当你取消一个 deferred 时,你可能不是在取消主异步操作,而是一些其他的作为前者结果所触发的异步操作.呼!
我们可以用一个例子来说明.考虑代码 deferred-cancel/defer-cancel-12.py:
from twisted.internet import defer

```
def cancel_outer(d):
    print "outer cancel callback."

def cancel_inner(d):
    print "inner cancel callback."

def first_outer_callback(res):
    print 'first outer callback, returning inner deferred'
    return inner_d

def second_outer_callback(res):
    print 'second outer callback got:', res

def outer_errback(err):
    print 'outer errback got:', err

outer_d = defer.Deferred(cancel_outer)
inner_d = defer.Deferred(cancel_inner)

outer_d.addCallback(first_outer_callback)
outer_d.addCallbacks(second_outer_callback, outer_errback)

outer_d.callback('result')

# at this point the outer deferred has fired, but is paused
# on the inner deferred.

print 'canceling outer deferred.'
outer_d.cancel()

print 'done'
```

在这个例子中,我们创建了两个 deferred, outer 和 inner,并且有一个外部回调返回内部 deferred. 首先,我们激发外部 deferred,然后取消它. 输出结果如下:

```
first outer callback, returning inner deferred
canceling outer deferred.
inner cancel callback.
outer errback got: [Failure instance: Traceback (failure with no frames): <class
'twisted.internet.defer.CancelledError'>:
]
done
```

正如你看到的,取消外部 deferred 并没有使外部 cancel 回调被激发. 相反,它取消了内部 deferred,所以内部 cancel 回调被激发了,之后外部错误回调收到 CancelledError (来自内部 deferred).
你可能需要仔细看一看那些代码,并且做些变化看看如何影响结果.

讨论

取消 deferred 是非常有用的操作,使我们的程序避免去做不需要的工作. 然而正如我们看到的,它可能有一点点棘手.
需要明白的一个重要事实是取消一个 deferred 并不意味着取消了它后面的异步操作.事实上,当写这篇文章时,很多 deferreds 并不会被真的"取消",因为大部分 Twisted 代码写于 Twisted 10.1.0之前并且还没有被升级.这包括很多 Twisted 本身的 APIs！检查文档或源代码去发现"取消 deferred"是否真的取消了背后的请求,还是仅仅忽略它.
第二个重要事实是从你的异步 APIs 返回的 deferred 并不一定在完整意义上可取消. 如果你希望在自己的程序中实现取消,你应该先研究一下 Twisted 源代码中的许多例子. Cancellation 是一个暂新的特性,所以它的模式和最好实践还在制定当中.

展望未来

现在我们已经学习了关于 Deferreds 的方方面面以及 Twisted 背后的核心概念. 这意味着我们没什么需要介绍的了,因为 Twisted 的其余部分主要包括一些特定的应用,如网络编程或异步数据库处理.故而,在 接下来 的部分中,我们想走点弯路,看看其他两个使用异步 I/O 的系统跟 Twisted 有何理念相似之处.之后,在尾声中,我们会打个包并且建议一些帮助你继续学习 Twisted 的方法.

参考练习

10 你知道你可以用多种方式拼写"cancelled"吗? 真的. 这取决于你的心情.
11 细读 Deferred 类的源代码,关注 cancellation 的实现.
12 在 Twisted 10.1.0的 源码 中找具有取消回调的 deferred 的例子.研究它们的实现.
13 修改我们诗歌客户端中 get_poetry 方法返回的 deferred, 使其可取消.
14 做一个基于 reactor 的例子展示取消外部 deferred,它被内层 deferred 暂停了.如果使用 callLater 你需要小心选择延迟时间,以确保外层 deferred 在正确的时刻被取消.
15 找一个 Twisted 中还不支持"本质上取消操作"的异步 API,为它实现本质取消. 并向 Twisted 项目 提交一个 补丁.不要忘记单元测试!

第二十部分：轮子中的轮子：Twisted 和 Erlang

你可以从 ":doc:`p01`" 开始阅读；也可以从 ":doc:`index`" 浏览索引.

简介

在这个系列中,有一个事实我们还没有介绍,即混合同步的"普通 Python"代码与异步 Twisted 代码不是一个简单的任务,因为在 Twisted 程序中阻滞不定时间将使异步模型的优势丧失殆尽.

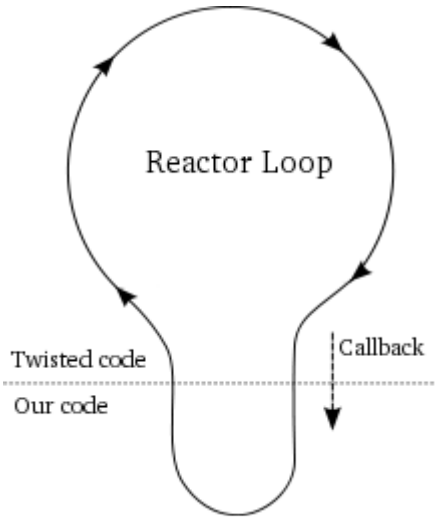
如果你是初次接触异步编程,那么你得到的知识看起来有一些局限.你可以在 Twisted 框架内使用这些新技术,而不是在更广阔的一般 Python 代 码世界中.同时,当用 Twisted 工作时,你仅仅局限于那些专门为作为Twisted程序一部分所写的库,至少如果你想直接从 reactor 线程调用它们.

但是异步编程技术已经存在了很多年并且几乎不局限于 Twisted.其实仅在 Python 中就有令人吃惊数目的异步编程模型. 搜索 一下就会看到很多. 它们在细节方面不同于 Twisted,但是基本的思想(如异步 I/O,将大规模数据流分割为小块处理)是一样的.所以如果你需要,或者选择,使用一个不同的框架,你将由于学习了 Twisted 而具备一个很好的开端.

当我们移步 Python 之外,同样会发现很多语言和系统要么基于要么使用了异步编程模型.你在 Twisted 学习到的知识将继续为你在异步编程方面开拓更广阔的领域而服务.

在这个部分,我们将简单地看一看 Erlang,一种编程语言和运行 时系统,它广泛使用异步编程概念,但是以一种独特的方式.请注意我们不是要开始写 Erlang 入门.而是稍稍探索一下 Erlang 中包含的一些思想,看看这些与 Twisted 思想的联系.基本主题就是你通过学习 Twisted 得到的知 识可以应用到学习其他技术.

回顾回调



考虑 图6 ,回调的图形表示. 是 :doc:`p06` 中介绍的 诗歌代理3.0 的回调和 dataReceived 方法中的顺序诗歌客户端的原理. 每次从一个相连的诗歌服务器下载一小部分诗歌时将激发回调.

假设我们的客户端从3个不同的服务器下载3首诗.以 reactor 的角度看问题(这是在这个系列中一直主张的),我们得到一个单一的大循环,当每次轮到时激发一个或多个回调,如图40:

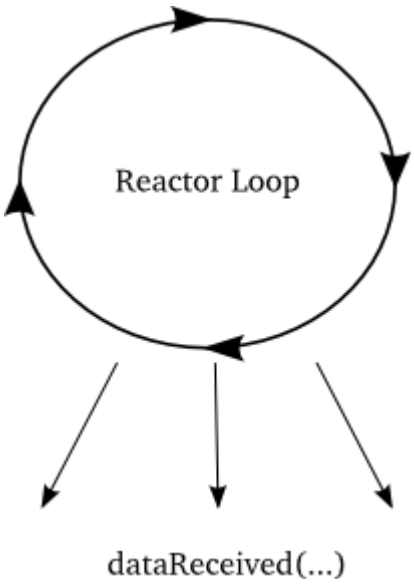


图40: 以 reactor 角度的回调

此图显示了 reactor 欢快地运转,每次诗歌到来时它调用 dataReceived. 每次 dataReceived 调用应用于一个特定的 PoetryProtocal 类实例. 我们知道一共有3个实例因为我们正在下载3首诗(所以必须有3个连接).

以一个 Protocol 实例的角度考虑这张图.记住每个 Protocol 只有一个连接(一首诗). 那个实例可“看到”一个方法调用流,每个方法承载着诗歌的下一部分,如下:

```
dataReceived(self, "When I have fears")
dataReceived(self, " that I may cease to be")
dataReceived(self, "Before my pen has glea")
dataReceived(self, "n'd my teeming brain")
...
```

然而这不是严格意义上的 Python 循环,我们可以将其概念化为一个循环:

```
for data in poetry_stream(): # pseudo-code
    dataReceived(data)
```

我们可以设想"回调循环",如图 41:

图41:一个虚拟回调循环

同样,这不是一个 for 循环或 while 循环. 在我们诗歌客户端中唯一重要的 Python 循环是 reactor. 但是我们可以把每个 Protocol 视作一个虚拟循环,当有诗歌到来时它会启动循环. 根据这种想法, 我们可以用图42重构整个客户端:

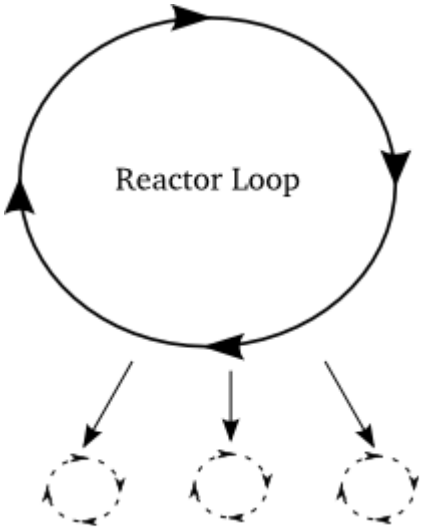


图42: reactor 转动虚拟循环
在这张图中,有一个大循环 —— reactor 和三个虚拟循环 —— 诗歌协议实例个体.大循环转起来,如此,使得虚拟循环也转起来了,就像一组环环相扣的齿轮.

进入 **Erlang**

Erlang,与 Python 一样,源自一种八十年代创建的一般目的动态类型的编程语言.不像 Python 的是,Erlang 是功能型的而不是面向对象的,并且在句法上类似怀旧的 Prolog, Erlang 最初就是由其实现的. Erlang 被设计为建立高度可靠的分布式电话系统,这样 Erlang 包含广泛的网络支持.
Erlang 的一个最独特的特性是一个涉及轻量级进程的并发模型. 一个 Erlang 进程既不是一个操作系统进程也不是线程.而它是在 Erlang 运行环境中一个独立运行的函数,它有自己的堆栈.Erlang 进程不是轻量 级的线程,因为 Erlang 进程不能共享状态(许多数据类型也是不可变的,Erlang 是一种功能性编程语言).一个 Erlang 进程可以与其他 Erlang 进程交互,但仅仅是通过发送消息,消息总是,至少概念上,被复制的而不是共享.
所以一个 Erlang 程序看起来如图43:

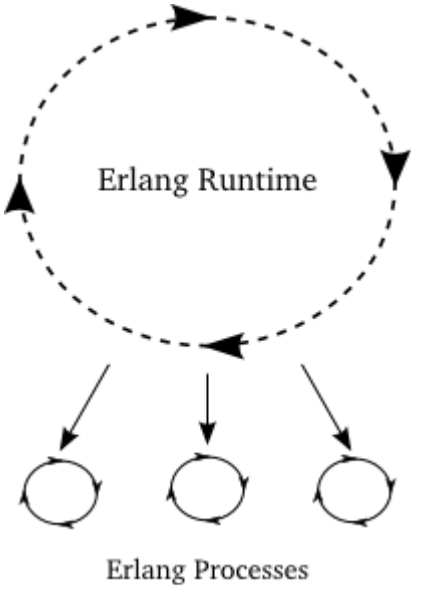


图43:有3个进程的 Erlang 程序

在此图中,个体进程变成了"真实的".因为进程在 Erlang 中是第一构造,就像 Python 中的对象.但运行时变成了"虚拟的",不是由于它不存 在,而是由于它不是一个简单的循环.Erlang 运行时可能是多线程的,因为它必须去实现一个全面的编程语言,还要负责很多除异步 I/O 之外的东西.进一 步,一个语言运行时也就是允许 Erlang 进程和代码执行的媒介,而不是像 Twisted 中的 reactor 那样的额外构造.

所以一个 Erlang 程序的更好表示如下图44:

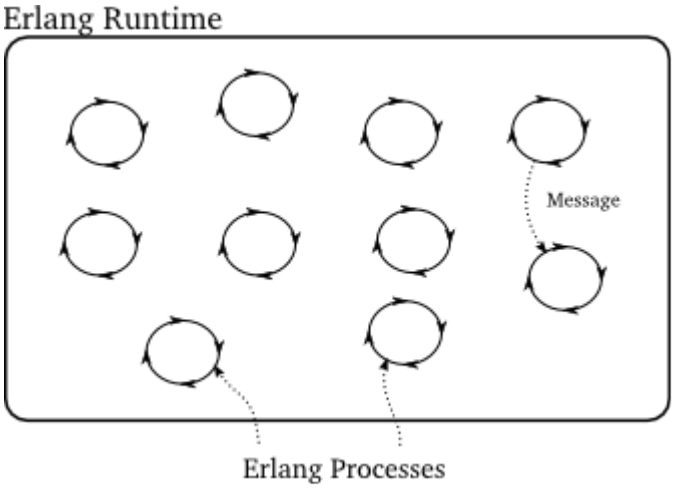


图44: 有若干进程的 Erlang 程序

当然, Erlang 运行时确实需要使用异步 I/O 以及一个或多个选择循环,因为 Erlang 允许你创建**大量** 进程. 大规模 Erlang 程序可以启动成千上万的 Erlang 进程,所以为每个进程分配一个实际地 OS 线程是问题所在.如果 Erlang 允许多进程执行 I/O,同时允许其他进程运行即便那个 I/O 阻塞了,那么异步 I/O 就必须被包含进来了.

注我们关于 Erlang 程序的图说明了每个进程是"靠它自己的力量"运行,而不是被回调旋转着.随着 reactor 的工作被归纳成 Erlang 运行时的结构,回调不再扮演中心角色. 原来在 Twisted 中需要通过回调解决的问题,在 Erlang 中将通过从一个进程向另一个进程发送异步消息来解决.

一个 Erlang 诗歌代理

让我们看一下 Erlang 诗歌客户端. 这次我们直接跳入工作版本而不是像在 Twisted 中慢慢地搭建它.同样,这不是意味着完整版本的 Erlang 介绍. 但如果这激起了你的兴趣,我们在本部分最后建议了一些深度阅读资料.

Erlang 客户端位于 erlang-client-1/get-poetry. 为了运行它,你当然需要安装 Erlang.

下面代码是 main 函数代码,与 Python 客户端中的 main 函数具有相同的目的:

```
main([]) ->
    usage();

main(Args) ->
    Addresses = parse_args(Args),
    Main = self(),
    [erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
|| {TaskNum, Addr} <- enumerate(Addresses)],
        collect_poems(length(Addresses), []).
```

如果你从来没有见过 Prolog 或者相似的语言,那么 Erlang 的句法将显得有一点奇怪.但是有一些人也这样认为 Python.
main 函数被两个分离的句群定义,被分号分割. Erlang 根据参数选择运行哪一个句群,所以第一个句群只在我们执行客户端时不提供任何命令行参数的情况下运行,并且它只打印出帮助信息.第二个句群是所有实际的行动.
Erlang 函数中的每条语句被逗号分隔,所以函数以句号结尾.让我们看一看第二个句群,第一行仅仅分析命令行参数并且将它们绑定到一个变量(Erlang 中所有变量必须大写).第二行使用 self 函数来获取当下正在运行的 Erlang 进程(而非 OS 进程)的 ID.由于这是主函数,你可以认为它等价于 Python 中的 __main__ 模块. 第三行是最有趣的:

```
[erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
    || {TaskNum, Addr} <- enumerate(Addresses)],
```

这个语句是对 Erlang 列表的理解,与 Python 有相似的句法.它产生新的 Erlang 进程,对应每个需要连接的服务器. 同时每个进程将运行相同的 get_poetry 函数, 但是根据特定的服务器用不同的参数.我们同时传递主进程的 PID 以便新的进程可以把诗歌发送回来(你通常需要一个进程的 PID 来向它发送消息)
main 函数中的最后一条语句调用 collect_poems 函数,它等待诗歌传回来和 get_poetry 进程结束.我们可以看一下其他函数,但首先你可能会对比一下 Erlang 的 main 函数与等价地 Twisted 客户端中的 main 函数.
现在让我们看一下 Erlang 中的 get_poetry 函数.事实上在我们的脚本中有两个函数叫 get_poetry.在 Erlang 中,一个函数被名字和元数同时确定,所以我们的脚本包含两个不同的函数, get_poetry/3 和 get_poetry/4,它们分别接收3个或4个参数.这里是 get_poetry/3,它是被 main 生成的:

```
get_poetry(Tasknum, Addr, Main) ->
    {Host, Port} = Addr,
    {ok, Socket} = gen_tcp:connect(Host, Port,
                                   [binary, {active, false}, {packet, 0}]),
    get_poetry(Tasknum, Socket, Main, []).
```

这个函数首先做一个 TCP 连接,就像 Twisted 客户端中的 get_poetry.但之后,不是返回,而是继续使用那个 TCP 连接,通过调用 get_poetry/4,如下:

```
get_poetry(Tasknum, Socket, Main, Packets) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Packet} ->
            io:format("Task ~w: got ~w bytes of poetry from ~s\n",
                      [Tasknum, size(Packet), peername(Socket)]),
            get_poetry(Tasknum, Socket, Main, [Packet|Packets]);
        {error, _} ->
            Main ! {poem, list_to_binary(lists:reverse(Packets))}
    end.
```

这个 Erlang 函数正在做 Twisted 客户端中 PoetryProtocol 的工作,不同的是它使用阻塞函数调用. gen_tcp:recv 函数等待在套接字上一些数据的到来(或者套接字关闭),无论要等多长时间.但 Erlang 中的"阻塞"函数仅阻塞正在运行函数的进程,而不是整个 Erlang 运行时.那个

TCP 套接字并不是一个真正的阻塞套接字(你不能在纯 Erlang 代码中创建一个真正的阻塞套接字).对于 Erlang 中的每个 套接字,在运行时的某处,一个"真正的"TCP 套接字被设置为非阻塞模型并且用作选择循环的一部分.
但是 Erlang 进程并不知道这些.它仅仅等待一些数据的到来,如果阻塞了,其他 Erlang 进程会代替运行.甚至一个进程从不阻塞,Erlang 运行时可以在任何时刻自由地在进程间切换.换句话说,Erlang 具有一个非协同并发机制.
注意 get_poetry/4,在收到一小部分诗歌后,继续递归地调用它自己.对于一个急迫的语言程序员这看起来像耗尽内存的良方,但 Erlang 编译器却可 以优化"尾"调用(函数调用一个函数中的最后一条语句)为循环.这照亮了又一个有趣的 Erlang 客户端和 Twisted 客户端之间的平行对比.在 Twisted 客户端中,"虚拟"循环是被 reaactor 创建的,它一次又一次地调用相同的函数(dataReceived).同时在 Erlang 客户端中,"真正"的运行进程(get_poetry/4)形成通过"尾调优化"一次又一次调用它们自己的循环.感觉怎么样.
如果连接关闭了, get_poetry 做的最后一件事情是把诗歌发送到主进程.同时结束 get_poetry 正在运行的进程,因为剩下没什么可做的了.
我们 Erlang 客户端中剩下的关键函数是 collect_poems:

```
collect_poems(0, Poems) ->
    [io:format("~s\n", [P]) || P <- Poems];
collect_poems(N, Poems) ->
    receive
        {'DOWN', _, _, _, _} ->
            collect_poems(N-1, Poems);
        {poem, Poem} ->
            collect_poems(N, [Poem|Poems])
    end.
```

这个函数被主进程运行,就像 get_poetry,它对自身递归循环.它同样阻塞. receive 告诉进程等待符合给定模式的消息到来,并且从"信箱"中提取消息.
collect_poems 函数等待两种消息: 诗歌和"DOWN"通知.后者是发送给主进程的, 当 get_poetry 进程之一由于某种原因死了的情况发送(这是 spawn_monitor 的监控部分).通过数 DOWN 消息,我们知道何时所有的诗歌都结束了. 前者是来自 get_poetry 进程的包含完整诗歌的消息.
OK,让我们运行一下 Erlang 客户端.首先启动 3 个慢服务器:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30
```

现在我们可以运行 Erlang 客户端了,与 Python 客户端有相似的命令行句法.如果你在 Linux 或其他 UNIX-样的系统,你应该可以直接运 行客户端(假设你安装了 Erlang 并使得它在你的 PATH 上).在 Windows 中,你可能需要运行 escript 程序,将指向 Erlang 客户端的路径作为第一个参数(其他参数留给 Erlang 客户端自身的参数).

```
./erlang-client-1/get-poetry 10001 10002 10003
```

之后,你可以看到如下输出:

```
Task 3: got 30 bytes of poetry from 127:0:0:1:10003
```

Task 2: got 10 bytes of poetry from 127:0:0:1:10002
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...

这就像之前的 Python 客户端之一,打印我们得到的每一小部分诗歌的信息.当所有诗歌都结束后,客户端应该打印每首诗的完整内容.注意客户端在所有服务器之间切换,这取决于哪个服务器可以发送诗歌.
图45展示了 Erlang 客户端的进程结构:

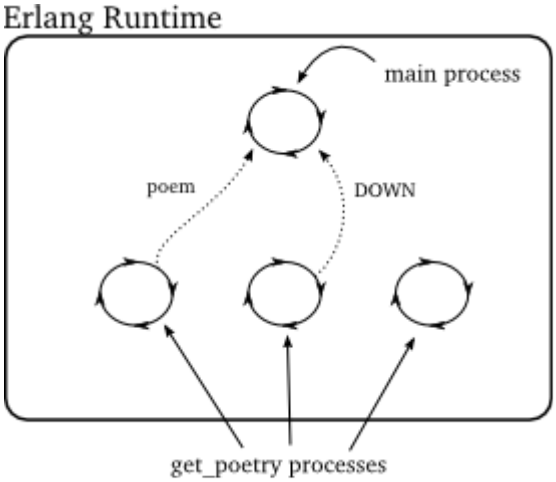


图45: Erlang 诗歌客户端
这张图显示了 3 个 get_poetry 进程(每个服务器一个)和一个主进程.你可以看到消息从诗歌进程流向主进程.
那么当一个服务器失败了会发生什么呢? 让我们试试:
./erlang-client-1/get-poetry 10001 10005

上面命令包含一个活动的端口(假设你没有终止之前的诗歌服务器)和一个未激活的端口(假设你没有在 10005 端口运行任一服务器). 我们得到如下输出:
Task 1: got 10 bytes of poetry from 127:0:0:1:10001

=ERROR REPORT==== 25-Sep-2010::21:02:10 ===
Error in process <0.33.0> with exit value: {{badmatch,{error,econnrefused}},[{erl_eval,expr,3}]}

Task 1: got 10 bytes of poetry from 127:0:0:1:10001
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...

最终客户端从活动的服务器完成诗歌下载,打印出诗歌并退出.那么 main 函数是怎样得知那两个进程完成工作了? 那个错误消息就是线索. 这个错误源自当 get_poetry 尝试连接到服务器时没有得到期望的值({ok, Socket}),而是得到一个连接被拒绝的错误.
Erlang 进程中一个未处理的异常将使其"崩溃",这意味着进程停止运行并且它们所有资源被回收了.但主进程,它监视所有 get_poetry 进程,当任何进程无论因为何种原因停止运行时将收到一个 DOWN 消息.这样,我们的客户端就退出了而不是一直运行下去.

讨论

让我们总结一下 Twisted 和 Erlang 客户端关于并行化的特点:

1 它们都是同时连接到所有诗歌服务器(或尝试连接).

2 它们都是从服务器即刻接收诗歌,无论是哪个服务器发送的.

3 它们都是以小段方式处理诗歌,因此必须保存迄今为止收到的诗歌的一部分.

4 它们都创建一个"对象"(或者 Python 对象或者 Erlang 进程)来为一个特定服务器处理所有工作.

5 它们都需要小心地确定诗歌何时结束,无论一个特定的下载成功与否.

在最后, 两个客户端中的 main 函数异步地接收诗歌和"任务完成"通知.在 Twisted 客户端中这个信息是通过 Deferred 发送的,而在 Erlang 中客户端接收来自内部进程消息.

注意到两个客户端非常像,无论它们的整体策略还是代码架构.但机理有一点点不同,一个是使用对象, deferreds 和回调,另一个是使用进程和消息.然而在高层的思想模型方面,两个客户端是十分相似的,如果你熟悉两种语言可以很方便地把一种转化为另一种.

甚至 reactor 模式在 Erlang 客户端中以小型化形式重现.我们诗歌客户端中的每个 Erlang 进程终究转变为一个递归循环:

6 等待一些事情发生(一小部分诗歌到来,一首诗传递完毕,另一个进程结束),以及

7 采取相应的行动.

你可以把 Erlang 程序视作一系列小 reactor 的大集合,每个都自己旋转着并且偶尔向另一个小 reactor 发送一个信息(它将以另一个事件来处理这个信息).

另外如果你更加深入 Erlang,你将发现回调露面了. Erlang 的 gen_server 进程是一个通用的 reactor 循环,你可以用一系列回调函数来"实例化"它,这是一种在 Erlang 系统中重复出现的模式.

进一步阅读

在这个部分我们关注 Twisted 与 Erlang 的相似性,但它们毕竟有很多不同.Erlang 的一个独特特性之一是它处理错误的方式.一个大的 Erlang 程序被结构化为一个树形结构的进程组,在高一层有"监管者",在叶子上有"工作者".如果一个工作进程崩溃了,监管进程会注意到并采取相应行 动(通常重启失败的进程).

如果你有兴趣学习 Erlang,那么很幸运.许多关于 Erlang 的书已经出版或将要出版:

- Programming Erlang—— 作者是 Erlang 的发明者之一.这个语言的精彩入门.
- Erlang Programming—— 此书补充了 Armstrong 的书,并且在许多关键部分深入更多细节.
- Erlang and OTP in Action—— 此书尚未出版,但我正在等待.前两本书都没有介绍 OTP, 构造大型应用的 Erlang 框架.完全披露:这本书的两个作者是我的朋友.

关于 Erlang 先就这么多.在 下一部分 我们会看一看 Haskell,另一种功能性语言,但与 Python 和 Erlang 的感觉都不同.然而,我们将努力去发现一些共同点.

建议练习(为高度热情的人)

8 浏览 Erlang 和 Python 客户端,并且确定它们在哪里相同哪里不同.它们是怎样处理错误的(比如连接到诗歌服务器的一个错误)?

9 简化 Erlang 客户端以便它不再打印到来的诗歌片段(故而你也不需要跟踪任务号).

10 修改 Erlang 客户端来计量下载每个诗歌所用的时间.

11 修改 Erlang 客户端打印诗歌,并使诗歌的顺序与它们在命令行给定的相同.

12 修改 Erlang 客户端来打印一个更加可读的错误信息当我们不能连接到诗歌服务器时.
13 写一个 Erlang 版本的诗歌服务器正如我们在 Twisted 中写的.

第二十一部分：惰性不是迟缓：Twisted 和 Haskell

你可以从 ":doc:`p01`" 开始阅读；也可以从 ":doc:`index`" 浏览索引.

简介

在上一个部分我们对比了 Twisted 与 Erlang,并将注意力集中在它们共有的一些思想上.结果表明使用 Erlang 也是非常简便的,因为异步 I/O 和反应式编程是 Erlang 运行时和进程模型的关键元素.
今天我们想走得更远一点,去看一看 Haskell —— 另一种功能性语言,然而与 Erlang 有很大不同(当然与 Python 也不同).这里面没有太多的平行概念,但我们仍然会发现藏在下面的异步 I/O 概念.

F —— 功能性

虽然 Erlang 是功能性语言,它主要关注可靠的并发模型.Haskell,另一方面,是彻头彻尾功能性的,它无耻地利用了范畴论的概念,如 函子 和 单子.
不要慌.我们这里不会涉及那些复杂的东西(虽然我们可以).相反,我们将关注一个 Haskell 的更加传统的功能性特性:惰性. 像许多功能性语言一样(除了 Erlang), Haskell 支持 惰性计算. 在懒惰计算语言中,程序的文字并不过多的描述怎样计算需要计算的东西.具体实施计算的细节一般留给了编译器和运行时系统.
同时,需要进一步指出,作为惰性计算推进的运行时可能一次只计算表达式的一部分(惰性的)而不是全部.一般地,运行时只提供维持当前计算继续所需的最小计算量.
这里有一个使用 Haskell head 语句的简单例子,这是一个提取列表第一个元素的函数,对于列表[1,2,3](Haskell 与 Python 分享一些列表句法):

```
head [1,2,3]
```

如果你安装了 GHC Haskell 运行时,你可以自己试一试:

```
[~] ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  : ? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> head [1,2,3]
1
Prelude>
```

结果是 1, 正如所料.
Haskell 列表的句法包含从前几个元素定义列表的使用功能.例如,列表[2,4,..]是从 2 开始的偶数序列.到哪结束呢?实际并不结 束.Haskell 列表[2,4,..]和其他如此表述的都是(概念上)无限列表.你可以在交互式 Haskell 提示符下计算它,这将试图打印这个表达式 的结果如下:

```
Prelude> [2,4 ..]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66,
68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100,102,104,106,108,110,112,114,116,118,120,
122,124,126,128,130,132,134,136,138,140,142,144,146,
...
```

你不得不按 Ctrl-C 终止计算因为它自己不会停下来.但由于是惰性计算,在 Haskell 中应用无限列表是没有问题的:

```
Prelude> head [2,4 ..]
2
Prelude> head (tail [2,4 ..])
4
Prelude> head (tail (tail [2,4 ..]))
6
```

这里我们分别获取无限列表的第一、二、三个元素,没看到任何无限循环.这就是惰性计算的本质.Haskell 运行时只构造完成 head 函数所需的列表,而不是先构造整个列表(这将导致无限循环),再将整个列表传递给 head.这个列表的其余部分跟本没有被构造,因为它们对继续推进计算毫无意义.
当我们引入 tail 函数时,Haskell 被迫进一步构造列表,但是又一次仅仅构造了满足下一次计算所需的列表.同时,一旦计算结束,列表(未完成的)被丢弃了.
这里是一些部分计算无限列表的 Haskell 代码：

```
Prelude> let x = [1..]
Prelude> let y = [2,4 ..]
Prelude> let z = [3,6 ..]
Prelude> head (tail (tail (zip3 x y z)))
(3,6,9)
```

zip 函数将所有列表压缩在一起,之后抓取尾部的尾部的头部.又一次,Haskell 没有发生任何问题,仅仅构造了计算所需的列表.我们可以将 Haskell 运行时"消耗"这些无限列表的过程可视化：

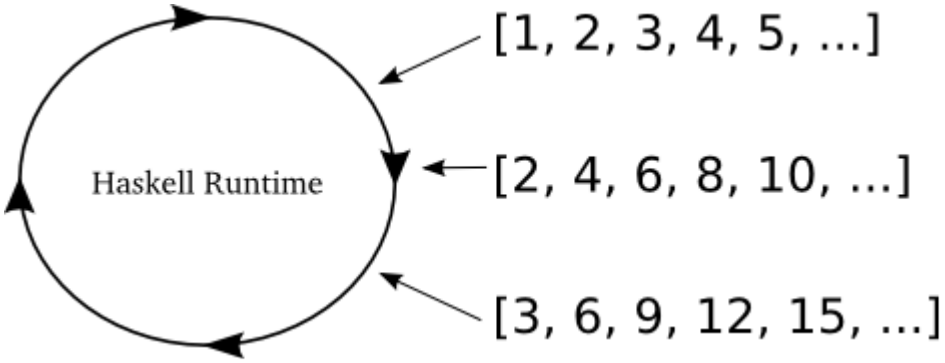


图46: Haskell 消耗一些无限列表

虽然我们将 Haskell 运行时画为一个简单的循环,它可能被多线程实现(并且很可能如果你使用 GHC 版本的 Haskell).但这幅图的关键点在于它十分像一个 reactor 循环,消耗从网络套接字传来的数据片段.

你可以把异步 I/O 和 reactor 模式视为一种有限形式的惰性计算.异步 I/O 的格言是:"仅仅推进你所拥有的数据".同时惰性计算的格言是:"仅仅推进你所需的数据".进一步,一个惰性计算语言在任何地方都使用这个格言,并不仅仅是有限范围的 I/O.

但关键点在于,对于惰性计算语言,做异步 I/O 没什么大不了的. 编译器和运行时已经被设计为一点一点地处理数据结构,因而惰性地处理到来的 I/O 数据流是标准问题. 如此 Haskell 运行时,就像 Erlang 运行时,简单地集成异步 I/O 为套接字抽象的一部分. 我们以实现一个 Haskell 诗歌客户端来展示这个概念.

Haskell 诗歌

我们第一个 Haskell 诗歌客户端位于 haskell-client-1/get-poetry.hs. 同 Erlang 一样,我们直接给出了完成版的客户端,如果你希望学习更多,我们指出进一步阅读的参考.

Haskell 同样支持轻量级线程或进程,尽管它们不是 Haskell 的核心,我们的 Haskell 客户端为每首需要下载的诗歌创建一个进程.关键函数是 runTask,它连接到一个套接字并且以轻量级线程启动 getPoetry 函数.

在这个代码中,你将看到许多类型定义. Haskell,不像 Python 和 Erlang,是静态类型的.我们没有为每个变量定义类型因为 Haskell 可以自动地推断没有显示定义的变量(或者 报告错误如果不能推断).许多函数包含 IO 类型(技术上叫单子)因为 Haskell 要求我们将有副作用的代码从纯函数中干净地分离(如,执行 I/O 的代 码).

getPoetry 函数包含如下行:

```
poem <- hGetContents h
```

看起来像从句柄一次读入整首诗(如 TCP 套接字).但是 Haskell,像往常一样,是惰性的.Haskell 运行时包含一个或更多实际线程,它们在一个选择循环中执行异步 I/O,如此便保存了惰性处理 I/O 流的可能性.

仅仅为说明异步 I/O 正在进行,我们引入一个"回调"函数, gotLine,它为诗歌的每一行打印一些任务信息.但这不是一个真正的回调函数,无论我们用不用它程序都会使用异步 I/O.甚至叫它"gotLine"反映了一个必要的语言思维,它是 Haskell 程序外的一部分.无论怎样,我们将一点点清扫它,但先使 Haskell 客户端运转起来.

启动一些慢诗歌服务器:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
```

python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30

现在编译 Haskell 客户端:
cd haskell-client-1/
ghc --make get-poetry.hs

这将创建一个二进制 get-poetry.最后,针对我们的服务器运行客户端:
/get-poetry 10001 10002 1000

你将看到如下输出:
Task 3: got 12 bytes of poetry from localhost:10003
Task 3: got 1 bytes of poetry from localhost:10003
Task 3: got 30 bytes of poetry from localhost:10003
Task 2: got 20 bytes of poetry from localhost:10002
Task 3: got 44 bytes of poetry from localhost:10003
Task 2: got 1 bytes of poetry from localhost:10002
Task 3: got 29 bytes of poetry from localhost:10003
Task 1: got 36 bytes of poetry from localhost:10001
Task 1: got 1 bytes of poetry from localhost:10001
...

输出与前一个异步客户端有点不同,因为我们只打印一行而不是任意块的数据.但,你可以清楚地看到,客户端是从所有服务器一起处理数据,而不是一个接一个.你同样可以注意到客户端立即打印第一首完成的诗,不等其他还在继续处理的诗.
好了,让我们清除还剩下的一点讨厌东西并且发布一个仅仅抓取诗歌而不介意任务序号的新版本.它位于 haskell-client-2/get-poetry.hs. 注意它短多了,对于每个服务器,仅仅连接到套接字,抓取所有数据,之后将其发送回去.
OK,让我们编译新的客户端：
cd haskell-client-2/
ghc --make get-poetry.hs

针对相同的诗歌服务器组运行它:
./get-poetry 10001 10002 10003

最终,你将看到屏幕上出现每首诗的文字.
你将注意到每个服务器同时向客户端发送数据.更重要的,客户端以最快速度打印出第一首诗的每一行,而不去等待其余的诗,甚至当它正在处理其它两首诗.之后它快速地打印出之前积累的第二首诗.
同时这所有发生的一切都不需要我们做什么.这里没有回调,没有传来传去的消息,仅仅是一个关于我们希望程序做什么的简洁地描述,而且很少需要告诉它应该怎样做.其余的事情都是由 Haskell 编译器和运行时处理的.漂亮!

讨论与进一步阅读

从 Twisted 到 Erlang 之后到 Haskell,我们可以看到一个平行的移动,从前景到背景逐步深入异步编程背后的思想.在 Twisted 中,异步编程是其存在的核心激励理念. Twisted 实现作为一个与 Python 分离的框架(Python 缺乏核心的异步抽象如轻量级线程),将异步模型置于首位与核心,当你用 Twisted 写程序时.

在 Erlang 中,异步对于程序员仍然是可见的,但细节成为语言材料的一部分和运行时系统,形成一个抽象使得异步消息在同步进程之间交换.

最后,在 Haskell 中,异步 I/O 仅仅是运行时中的另一个技术,大部分对于程序员是不可见的,因为提供惰性计算是 Haskell 的中心理念.

对于以上情况,我们还没有介绍任何深邃的思想.我们仅仅指出许多并且有趣的异步模型出现的地方,这种模型可以被多种方式表达.

如果任何这些激起你对 Haskell 的兴趣,那么我们建议"Real World Haskell"继续你的学习.这本书是介绍语言学习的典范.

同时虽然我没有读过它,我却听说"Learn You a Haskell"的饱受赞誉.

现在到了结束探索 Twisted 之外异步系统的时刻,并且完成了本系列的倒数第二部分. 在":doc:`p22`"中,我们将做一个总结,以及建议一些学习 Twisted 的方法.

建议练习(献给令人吃惊的狂热者)

1 互相对比 Twisted,Erlang 和 Haskell 客户端.

2 修改 Haskell 客户端来处理连接诗歌服务器的失败,以便它们能够下载所有的能够下载的诗歌并为那些不能下载的诗歌输出合理的错误消息.

3 写 Haskell 版本的对应 Twisted 中的诗歌服务器.