- twistedTUT01

Finger演化: 创建一个简单的finger服务 -- dreamingk [2004-08-09 02:06:10]

1. 介绍

目录

(Introduction)

This is the first part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

这是《Twisted入门--Finger演化》教程的第一章。

By the end of this section of the tutorial, our finger server will answer TCP finger requests on port 1079, and will read data from the web.

在本章的结尾，我们的finger服务器会在1079端口响应TCP的finger请求，并且从web上读取数据。

2. 拒绝连接(Refuse Connections)

切换行号显示

```
1 #Source listing - listings/finger/finger01.py
2
3 from twisted.internet import reactor
4 reactor.run()
```

This example only runs the reactor. Nothing at all will happen until we interrupt the program. It will consume almost no CPU resources. Not very useful, perhaps -- but this is the skeleton inside which the Twisted program will grow.

这个例子只运行了反应器(reactor)。所以什么也不会发生。它几乎不会耗费任何处理器(cpu)资源。也许没什么用，不过这是Twisted程序成长的骨架。

2.1. 反应器(Reactor)

You don't call Twisted, Twisted calls you. The reactor is Twisted's main event loop. There

is exactly one reactor in any running Twisted application. Once started it loops over and over again, responding to network events, and making scheduled calls to code.

不是你的程序调用Twisted，而是Twisted来调用你的程序。反应器是Twisted的主事件循环。每一个运行着的Twisted程序都有一个反应器，一旦启动它，它就不停循环，响应网络事件，然后执行预定的调用。

3. 什么也不做(Do Nothing)

切换行号显示

```
1 #Source listing - listings/finger/finger02.py
2
3 from twisted.internet import protocol, reactor
4 class FingerProtocol(protocol.Protocol):
5     pass
6 class FingerFactory(protocol.ServerFactory):
7     protocol = FingerProtocol
8 reactor.listenTCP(1079, FingerFactory())
9 reactor.run()
```

Here, we start listening on port 1079. The 1079 is a reminder that eventually, we want to run on port 79, the standard port for finger servers. We define a protocol which does not respond to any events. Thus, connections to 1079 will be accepted, but the input ignored.

这里，我们开始监听1079端口。1079端口只是临时使用，最终我们会使用79端口，79端口是finger服务器的标准端口。我们定义了一个协议(protocol)，它不响应任何事件。因此可以连接到1079端口，但是任何输入都得不到响应。

4. 断开连接(Drop Connections)

切换行号显示

```
1 #Source listing - listings/finger/finger03.py
2
3 from twisted.internet import protocol, reactor
4 class FingerProtocol(protocol.Protocol):
5     def connectionMade(self):
6         self.transport.loseConnection()
7 class FingerFactory(protocol.ServerFactory):
8     protocol = FingerProtocol
9 reactor.listenTCP(1079, FingerFactory())
10 reactor.run()
```

Here we add to the protocol the ability to respond to the event of beginning a connection -- by terminating it. Perhaps not an interesting behavior, but it is already close to behaving according to the letter of the protocol. After all, there is no requirement to send any data to the remote connection in the standard. The only problem, as far as the standard is concerned, is that we terminate the connection too soon. A client which is slow enough will see his send() of the username result in an error.

现在我们改进了协议(protocol), 使它有能力对建立连接的事件作出反应——断开连接。也许不是一个有趣的做法，但是反应器已经能够根据协议(protocol)来作出反

应。毕竟，标准中也没有规定一定要在建立连接的时候向远端发数据。唯一的问题是，我们关闭连接太快。一个很慢的客户端会发现它用send()函数发送用户名时会出现错误。

5. 读用户名，断开连接(Read Username, Drop Connections)

切换行号显示

```
1 #Source listing - listings/finger/finger04.py
2
3 from twisted.internet import protocol, reactor
4 from twisted.protocols import basic
5 class FingerProtocol(basic.LineReceiver):
6     def lineReceived(self, user):
7         self.transport.loseConnection()
8 class FingerFactory(protocol.ServerFactory):
9     protocol = FingerProtocol
10 reactor.listenTCP(1079, FingerFactory())
11 reactor.run()
```

Here we make FingerProtocol inherit from LineReceiver, so that we get data-based events on a line-by-line basis. We respond to the event of receiving the line with shutting down the connection.

现在我们从LineReveiver类继承了一个新类叫FingerProtocol， 这样当数据一行一行到达时，就会产生事件，而我们响应事件，然后关闭连接。

Congratulations, this is the first standard-compliant version of the code. However, usually people actually expect some data about users to be transmitted.

恭喜你，这是第一个符合finger标准的代码版本。但是，人们通常希望能得到用户的有关信息。

6. 读用户名，输出错误信息，断开连接(Read Username, Output Error, Drop Connections)

切换行号显示

```
1 #Source listing - listings/finger/finger05.py
2
3 from twisted.internet import protocol, reactor
4 from twisted.protocols import basic
5 class FingerProtocol(basic.LineReceiver):
6     def lineReceived(self, user):
7         self.transport.write("No such user\r\n")
8         self.transport.loseConnection()
9 class FingerFactory(protocol.ServerFactory):
10     protocol = FingerProtocol
11 reactor.listenTCP(1079, FingerFactory())
12 reactor.run()
```

Finally, a useful version. Granted, the usefulness is somewhat limited by the fact that this version only prints out a No such user message. It could be used for devastating effect in honey-pots, of course.
最终我们得到了一个有用的版本。但是这个版本仅仅返回“没有这个用户”的信息，所以不是很有用。当然它也许可以用来搞恶作剧。

7. 从一个空的工厂(Factory)中输出(Output From Empty Factory)

切换行号显示

```
1 #Source listing - listings/finger/finger06.py
2
3 # Read username, output from empty factory, drop connections
4 from twisted.internet import protocol, reactor
5 from twisted.protocols import basic
6 class FingerProtocol(basic.LineReceiver):
7     def lineReceived(self, user):
8         self.transport.write(self.factory.getUser(user)+"\r\n")
9         self.transport.loseConnection()
10 class FingerFactory(protocol.ServerFactory):
11     protocol = FingerProtocol
12     def getUser(self, user): return "No such user"
13 reactor.listenTCP(1079, FingerFactory())
14 reactor.run()
```

The same behavior, but finally we see what usefulness the factory has: as something that does not get constructed for every connection, it can be in charge of the user database. In particular, we won't have to change the protocol if the user database back-end changes.
这个例子和上面的例子有同样的行为，但是最终我们会看到工厂(Factory)有什么用处：它不需要为每次连接都创建一次，它可以来管理用户数据库。尤其是当后台用户数据库发生了变化时，我们不用改变协议（protocol）。

8. 从一个非空的工厂(Factory)中输出(Output from Non-empty Factory)

切换行号显示

```
1 #Source listing - listings/finger/finger07.py
2
3 # Read username, output from non-empty factory, drop connections
4 from twisted.internet import protocol, reactor
5 from twisted.protocols import basic
6 class FingerProtocol(basic.LineReceiver):
7     def lineReceived(self, user):
8         self.transport.write(self.factory.getUser(user)+"\r\n")
9         self.transport.loseConnection()
10 class FingerFactory(protocol.ServerFactory):
11     protocol = FingerProtocol
```

```
12     def __init__(self, **kwargs): self.users = kwargs
13     def getUser(self, user):
14         return self.users.get(user, "No such user")
15 reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
16 reactor.run()
```

Finally, a really useful finger database. While it does not supply information about logged in users, it could be used to distribute things like office locations and internal office numbers. As hinted above, the factory is in charge of keeping the user database: note that the protocol instance has not changed. This is starting to look good: we really won't have to keep tweaking our protocol.

最终，一个有用的finger数据库出现了。虽然它不能提供登录用户的任何信息，但是可以发布一些类似办公地点，内部办公室编号的信息。 就像上面指出的那样，工厂(factory)负责管理用户数据库，而协议(protocol)实例却没有发生改变。这看起来不错，因为我们真的不用老是改协议(protocol)了。

9. 使用 Deferreds(Use Deferreds)

切换行号显示

```
1 #Source listing - listings/finger/finger08.py
2
3 # Read username, output from non-empty factory, drop connections
4 # Use deferreds, to minimize synchronicity assumptions
5 from twisted.internet import protocol, reactor, defer
6 from twisted.protocols import basic
7 class FingerProtocol(basic.LineReceiver):
8     def lineReceived(self, user):
9         self.factory.getUser(user
10        ).addErrback(lambda _: "Internal error in server"
11        ).addCallback(lambda m:
12                      (self.transport.write(m+"\r\n"),
13                       self.transport.loseConnection()))
14 class FingerFactory(protocol.ServerFactory):
15     protocol = FingerProtocol
16     def __init__(self, **kwargs): self.users = kwargs
17     def getUser(self, user):
18         return defer.succeed(self.users.get(user, "No such user"))
19 reactor.listenTCP(1079, FingerFactory(moshez='Happy and well'))
20 reactor.run()
```

But, here we tweak it just for the hell of it. Yes, while the previous version worked, it did assume the result of getUser is always immediately available. But what if instead of an in memory database, we would have to fetch result from a remote Oracle? Or from the web? Or, or...

但是，现在我们改了它（协议）, 仅仅是为了好玩。是的，尽管上一个版本可以工作，但那是在假设“getUser”操作总是能立即执行完的情况下。假如我们需要从另一台机器上运行的Oracle数据库取数据，或者是从web上和其它地方，我们该怎么办呢？

10. 在本地运行'finger'(Run 'finger' Locally)

切换行号显示

```
   1 #Source listing - listings/finger/finger09.py
   2
   3 # Read username, output from factory interfacing to OS, drop
connections
   4 from twisted.internet import protocol, reactor, defer, utils
   5 from twisted.protocols import basic
   6 class FingerProtocol(basic.LineReceiver):
   7     def lineReceived(self, user):
   8         self.factory.getUser(user
   9         ).addErrback(lambda _: "Internal error in server"
  10         ).addCallback(lambda m:
  11                       (self.transport.write(m+"\r\n"),
  12                        self.transport.loseConnection()))
  13 class FingerFactory(protocol.ServerFactory):
  14     protocol = FingerProtocol
  15     def getUser(self, user):
  16         return utils.getProcessOutput("finger", [user])
  17 reactor.listenTCP(1079, FingerFactory())
  18 reactor.run()
```

...from running a local command? Yes, this version runs finger locally with whatever arguments it is given, and returns the standard output. This is probably insecure, so you probably don't want a real server to do this without a lot more validation of the user input. This will do exactly what the standard version of the finger server does.

...来自于一个本地运行的命令(程序)? 是的，这个版本在本地运行系统提供的finger服务，并且传给它任何参数，然后返回它的标准输出。这也许不够安全，所以你可能不会要一个真正运行的服务器在不进行大量的用户输入检查的情况下就提供这样的服务。这个版本确实做到了标准版的finger服务器应该做的了。

11. 从web上读取信息(Read Status from the Web)

The web. That invention which has infiltrated homes around the world finally gets through to our invention. Here we use the built-in Twisted web client, which also returns a deferred. Finally, we manage to have examples of three different database back-ends, which do not change the protocol class. In fact, we will not have to change the protocol again until the end of this tutorial: we have achieved, here, one truly usable class.

web，这个渗透到全世界千千万万家庭中的发明最终结合到我们的程序中来了，这里我们使用了Twisted内置（built in）的web客户端组件，它同样也返回一个deferred对象。 最后，我们展示了如何在不修改协议(protocol)类的情况下，实现三个不同的数据库后端。实际上，直到这个教程结束，我们都不用修改协议（protocol）类，我们已经有了一个真正可用的类。

切换行号显示

```
   1 #Source listing - listings/finger/finger10.py
   2
   3 # Read username, output from factory interfacing to web, drop
connections
```

```
   4 from twisted.internet import protocol, reactor, defer, utils
   5 from twisted.protocols import basic
   6 from twisted.web import client
   7 class FingerProtocol(basic.LineReceiver):
   8     def lineReceived(self, user):
   9         self.factory.getUser(user
  10         ).addErrback(lambda _: "Internal error in server"
  11         ).addCallback(lambda m:
  12                       (self.transport.write(m+"\r\n"),
  13                        self.transport.loseConnection()))
  14 class FingerFactory(protocol.ServerFactory):
  15     protocol = FingerProtocol
  16     def __init__(self, prefix): self.prefix=prefix
  17     def getUser(self, user):
  18         return client.getPage(self.prefix+user)
  19 reactor.listenTCP(1079,
FingerFactory(prefix='http://livejournal.com/~'))
  20 reactor.run()
```

12. 使用Application对象(Use Application)

Up until now, we faked. We kept using port 1079, because really, who wants to run a finger server with root privileges? Well, the common solution is privilege shedding: after binding to the network, become a different, less privileged user. We could have done it ourselves, but Twisted has a built-in way to do it. We will create a snippet as above, but now we will define an application object. That object will have uid and gid attributes. When running it (later we will see how) it will bind to ports, shed privileges and then run.

直到现在我们一直在使用1079端口，而不是标准的79端口，是因为没有谁希望在root权限下运行一个finger服务器。通常的解决办法是：在绑定到端口以后，成为另一个权限较低的用户。我们可以自己做这件事，但是Twisted提供了一种内置（built in）的方法。我们可以使用上面那段代码，但是现在我们会定义一个Application对象，它有uid和gid的属性。当我们运行这个Appliction对象的时候，它会绑定(bind)到端口，改变用户权限然后运行。

After saving the next example (finger11.py) as finger.tac, read on to find out how to run this code using the twistd utility.

把下一个例子程序另存为finger.tac，接着往下读，你会知道如何用Twisted提供的工具来运行这个文件。

切换行号显示

```
   1 #Source listing - listings/finger/finger11.py
   2
   3 # Read username, output from non-empty factory, drop connections
   4 # Use deferreds, to minimize synchronicity assumptions
   5 # Write application. Save in 'finger.tpy'
   6 from twisted.application import internet, service
   7 from twisted.internet import protocol, reactor, defer
   8 from twisted.protocols import basic
   9 class FingerProtocol(basic.LineReceiver):
  10     def lineReceived(self, user):
```

```
11          self.factory.getUser(user
12          ).addErrback(lambda _: "Internal error in server"
13          ).addCallback(lambda m:
14                        (self.transport.write(m+"\r\n"),
15                         self.transport.loseConnection()))
16 class FingerFactory(protocol.ServerFactory):
17     protocol = FingerProtocol
18     def __init__(self, **kwargs): self.users = kwargs
19     def getUser(self, user):
20         return defer.succeed(self.users.get(user, "No such user"))
21
22 application = service.Application('finger', uid=1, gid=1)
23 factory = FingerFactory(moshez='Happy and well')
24 internet.TCPServer(79, factory).setServiceParent(
25     service.IServiceCollection(application))
```

13. twistd(twisted)

This is how to run Twisted Applications -- files which define an 'application'. twistd (TWISTed Daemonizer) does everything a daemon can be expected to -- shuts down stdin/stdout/stderr, disconnects from the terminal and can even change runtime directory, or even the root filesystems. In short, it does everything so the Twisted application developer can concentrate on writing his networking code.

这段代码展示了如何用twisted来运行定义了Twisted Application对象的.tac文件。twisted(Twisted式的daemon)实现了一个daemon应该做的全部工作——关闭stdin/stdout/stderr，断开与终端的连接，甚至可以改变运行时目录和根(root)文件系统。总之有它在，你就只管写你的网络部分代码吧。

```
root% twistd -ny finger.tac # 和以前一样
root% twistd -y finger.tac # 守护进程化，保存pid到twistd.pid文件中去
root% twistd -y finger.tac --pidfile=finger.pid
root% twistd -y finger.tac --rundir=/
root% twistd -y finger.tac --chroot=/var
root% twistd -y finger.tac -l /var/log/finger.log
root% twistd -y finger.tac --syslog # 转向log到syslog中
root% twistd -y finger.tac --syslog --prefix=twistedfinger # 使用指定的路径
```

Version: 1.3.0

return index-->TwistedTUT

twistedTUT01 (2004-08-09 02:06:10由dreamingk编辑)

- twistedTUT02

Finger 演化：给 finger 服务器添加新的特性 (The Evolution of Finger: adding features to the finger service) -- dreamingk [2004-08-09 02:08:21]

目录

1. 介绍 (Introduction)

这是《Twisted 入门 - Finger 演化》的第二部分。

This is the second part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在教程的这个部分，我们的 finger 服务器将继续进化出新的特性：用户设置 finger 公告的能力；通过 finger 服务向 web 上、向 IRC 上、或通过 XML-RPC 向外界发布这些公告的能力。

In this section of the tutorial, our finger server will continue to sprout features: the ability for users to set finger announces, and using our finger service to send those announcements on the web, on IRC and over XML-RPC.

2. 让本地用户来设置信息 (Setting Message By Local Users)

现在，1079端口空闲出来了；也许我们可以用它运行另外一个服务器，让人们可以设置他们的信息内容。初步设想，它不进行访问控制，所以任何可以登录到这台机器的人都可以设置任何信息——假设这就是我们计划中的需求

如果要测试服务，可以输入下面的命令：

Now that port 1079 is free, maybe we can run on it a different server, one which will let people set their messages. It does no access control, so anyone who can login to the machine can set any message. We assume this is the desired behavior in our case. Testing it can be done by simply:

```
% nc localhost 1079   # 如果你不知道nc是什么，哪么就 telnet localhost 1079
moshez
Giving a tutorial now, sorry!
^D
```

切换行号显示

```
1 # 让我们试试设置公告信息，怎么样？
```

```
 2 # But let's try and fix setting away messages, shall we?
 3 from twisted.application import internet, service
 4 from twisted.internet import protocol, reactor, defer
 5 from twisted.protocols import basic
 6 class FingerProtocol(basic.LineReceiver):
 7     def lineReceived(self, user):
 8         self.factory.getUser(user
 9         ).addErrback(lambda _: "Internal error in server"
10         ).addCallback(lambda m:
11                       (self.transport.write(m+"\r\n"),
12                        self.transport.loseConnection()))
13
14 class FingerFactory(protocol.ServerFactory):
15     protocol = FingerProtocol
16     def __init__(self, **kwargs): self.users = kwargs
17     def getUser(self, user):
18         return defer.succeed(self.users.get(user, "No such user"))
19
20 class FingerSetterProtocol(basic.LineReceiver):
21     def connectionMade(self): self.lines = []
22     def lineReceived(self, line): self.lines.append(line)
23     def connectionLost(self, reason):
24         self.factory.setUser(*self.lines[:2])
25         # first line: user    second line: status
26
27 class FingerSetterFactory(protocol.ServerFactory):
28     protocol = FingerSetterProtocol
29     def __init__(self, ff): self.setUser = ff.users.__setitem__
30
31 ff = FingerFactory(moshez='Happy and well')
32 fsf = FingerSetterFactory(ff)
33
34 application = service.Application('finger', uid=1, gid=1)
35 serviceCollection = service.IServiceCollection(application)
36 internet.TCPServer(79,ff).setServiceParent(serviceCollection)
37 internet.TCPServer(1079,fsf).setServiceParent(serviceCollection)
```

源程序 - listings/finger/finger12.py

3. 用服务让彼此间的依赖健壮 (Use Services to Make Dependencies Sane)

上一个版本里我们在 finger 工厂里建立了一个设置属性用的“钩子”。一般来说，这么做不太好；现在我们将通过把两个工厂视为一个对象来实现它们的均衡。当一个对象不再关联于一个特定的网络服务器时，服务（Service）将会大有用处。现在我们把两个创建工厂的所有职责都转移到服务里。注意，我们并不是使用建立子类的方法：这个服务只是简单的把原来工厂里的方法和属性照搬进来。这样的架构在协议设计方面比原来更优秀了：我们用于协议的两个类都不必做任何修改；而且直到这个教程的最后也不用再变动什么东西了。

The previous version had the setter poke at the innards of the finger factory. It's usually

not a good idea: this version makes both factories symmetric by making them both look at a single object. Services are useful for when an object is needed which is not related to a specific network server. Here, we moved all responsibility for manufacturing factories into the service. Note that we stopped subclassing: the service simply puts useful methods and attributes inside the factories. We are getting better at protocol design: none of our protocol classes had to be changed, and neither will have to change until the end of the tutorial.

切换行号显示

```
# 修正不对称的设计
# Fix asymmetry
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.factory.getUser(user
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))

class FingerSetterProtocol(basic.LineReceiver):
    def connectionMade(self): self.lines = []
    def lineReceived(self, line): self.lines.append(line)
    def connectionLost(self,reason):
self.factory.setUser(*self.lines[:2])
    # first line: user   second line: status

class FingerService(service.Service):
    def __init__(self, *args, **kwargs):
        self.parent.__init__(self, *args)
        self.users = kwargs
    def getUser(self, user):
        return defer.succeed(self.users.get(user, "No such user"))
    def getFingerFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.getUser = FingerProtocol, self.getUser
        return f
    def getFingerSetterFactory(self):
        f = protocol.ServerFactory()
        f.protocol, f.setUser = FingerSetterProtocol,
self.users.__setitem__
        return f

application = service.Application('finger', uid=1, gid=1)
f = FingerService('finger', moshez='Happy and well')
serviceCollection = service.IServiceCollection(application)
internet.TCPServer(79,f.getFingerFactory()
                   ).setServiceParent(serviceCollection)
```

```
41 internet.TCPServer(1079,f.getFingerSetterFactory()
42                     ).setServiceParent(serviceCollection)
```

源程序 - listings/finger/finger13.py

4. 读取状态文件 (Read Status File)

这一版本展示了如何从一个集中管理的文件中读取用户消息，而不仅仅是让用户设定它们。我们缓存结果，并每30秒刷新一次。服务对于这种调度性质的任务非常有用。

This version shows how, instead of just letting users set their messages, we can read those from a centrally managed file. We cache results, and every 30 seconds we refresh it. Services are useful for such scheduled tasks.

```
moshez: happy and well
shawn: alive
```

/etc/users 示例文件 - listings/finger/etc.users

切换行号显示

```
 1 # 从文件读取资料
 2 # Read from file
 3 from twisted.application import internet, service
 4 from twisted.internet import protocol, reactor, defer
 5 from twisted.protocols import basic
 6
 7 class FingerProtocol(basic.LineReceiver):
 8     def lineReceived(self, user):
 9         self.factory.getUser(user
10         ).addErrback(lambda _: "Internal error in server"
11         ).addCallback(lambda m:
12                       (self.transport.write(m+"\r\n"),
13                        self.transport.loseConnection()))
14
15 class FingerService(service.Service):
16     def __init__(self, filename):
17         self.users = {}
18         self.filename = filename
19     def _read(self):
20         for line in file(self.filename):
21             user, status = line.split(':', 1)
22             user = user.strip()
23             status = status.strip()
24             self.users[user] = status
25         self.call = reactor.callLater(30, self._read)
26     def startService(self):
27         self._read()
28         service.Service.startService(self)
29     def stopService(self):
30         service.Service.stopService(self)
```

```
31          self.call.cancel()
32      def getUser(self, user):
33          return defer.succeed(self.users.get(user, "No such user"))
34      def getFingerFactory(self):
35          f = protocol.ServerFactory()
36          f.protocol, f.getUser = FingerProtocol, self.getUser
37          return f
38
39  application = service.Application('finger', uid=1, gid=1)
40  f = FingerService('/etc/users')
41  finger = internet.TCPServer(79, f.getFingerFactory())
42
43  finger.setServiceParent(service.IServiceCollection(application))
44  f.setServiceParent(service.IServiceCollection(application))
```

源程序 - listings/finger/finger14.py

5. 同时在 web 上发布 (Announce on Web, Too)

同样类型的服务对于其他协议也可以提供有价值的东西。例如，在 twisted.web 中，工厂本身（site）几乎从来不被用来生成子类。取而代之的是，它提供了一种资源，可以通过网络地址（URL）的方式来表现可用的资源树。这种层次结构由 site 来操控，并且可以通过 getChild 来动态重载它。

The same kind of service can also produce things useful for other protocols. For example, in twisted.web, the factory itself (the site) is almost never subclassed -- instead, it is given a resource, which represents the tree of resources available via URLs. That hierarchy is navigated by site, and overriding it dynamically is possible with getChild.

切换行号显示

```
1  # 从文件读取资料，并把它们发布在 web 上!
2  # Read from file, announce on the web!
3  from twisted.application import internet, service
4  from twisted.internet import protocol, reactor, defer
5  from twisted.protocols import basic
6  from twisted.web import resource, server, static
7  import cgi
8
9  class FingerProtocol(basic.LineReceiver):
10     def lineReceived(self, user):
11         self.factory.getUser(user
12         ).addErrback(lambda _: "Internal error in server"
13         ).addCallback(lambda m:
14                       (self.transport.write(m+"\r\n"),
15                        self.transport.loseConnection()))
16
17 class MotdResource(resource.Resource):
18
19     def __init__(self, users):
20         self.users = users
21         resource.Resource.__init__(self)
```

```
22
23      # we treat the path as the username
24      def getChild(self, username, request):
25          motd = self.users.get(username)
26          username = cgi.escape(username)
27          if motd is not None:
28              motd = cgi.escape(motd)
29              text = '<h1>%s</h1><p>%s</p>' % (username,motd)
30          else:
31              text = '<h1>%s</h1><p>No such user</p>' % username
32          return static.Data(text, 'text/html')
33
34  class FingerService(service.Service):
35      def __init__(self, filename):
36          self.filename = filename
37          self._read()
38      def _read(self):
39          self.users = {}
40          for line in file(self.filename):
41              user, status = line.split(':', 1)
42              user = user.strip()
43              status = status.strip()
44              self.users[user] = status
45          self.call = reactor.callLater(30, self._read)
46      def getUser(self, user):
47          return defer.succeed(self.users.get(user, "No such user"))
48      def getFingerFactory(self):
49          f = protocol.ServerFactory()
50          f.protocol, f.getUser = FingerProtocol, self.getUser
51          f.startService = self.startService
52          return f
53
54      def getResource(self):
55          r = MotdResource(self.users)
56          return r
57
58  application = service.Application('finger', uid=1, gid=1)
59  f = FingerService('/etc/users')
60  serviceCollection = service.IServiceCollection(application)
61  internet.TCPServer(79, f.getFingerFactory()
62                      ).setServiceParent(serviceCollection)
63  internet.TCPServer(8000, server.Site(f.getResource())
64                      ).setServiceParent(serviceCollection)
```

源程序 - listings/finger/finger15.py

6. 同时在 IRC 上发布 (Announce on IRC, Too)

这是教程里头一回有客户端的代码。IRC 客户端也常常扮演服务器的角色：对网上的事件作出应答。重联客户工厂（reconnecting client factory）可以确保中断的联接

被重新建立，它使用智能加强幂指数支持算法（呵呵，这是什么算法，这么长的名字:）。IRC 客户本身很简单：它唯一真正做得事情就是在 connectionMade 函数里从工厂获得昵称。

This is the first time there is client code. IRC clients often act a lot like servers: responding to events from the network. The reconnecting client factory will make sure that severed links will get re-established, with intelligent tweaked exponential back-off algorithms. The IRC client itself is simple: the only real hack is getting the nickname from the factory in connectionMade.

切换行号显示

```
  1 # 从文件读取数据，发布到 web 和 IRC
  2 # Read from file, announce on the web, irc
  3 from twisted.application import internet, service
  4 from twisted.internet import protocol, reactor, defer
  5 from twisted.protocols import basic, irc
  6 from twisted.web import resource, server, static
  7 import cgi
  8 class FingerProtocol(basic.LineReceiver):
  9     def lineReceived(self, user):
 10         self.factory.getUser(user
 11         ).addErrback(lambda _: "Internal error in server"
 12         ).addCallback(lambda m:
 13                       (self.transport.write(m+"\r\n"),
 14                        self.transport.loseConnection()))
 15 class FingerSetterProtocol(basic.LineReceiver):
 16     def connectionMade(self): self.lines = []
 17     def lineReceived(self, line): self.lines.append(line)
 18     def connectionLost(self,reason):
self.factory.setUser(*self.lines[:2])
 19 class IRCReplyBot(irc.IRCClient):
 20     def connectionMade(self):
 21         self.nickname = self.factory.nickname
 22         irc.IRCClient.connectionMade(self)
 23     def privmsg(self, user, channel, msg):
 24         user = user.split('!')[0]
 25         if self.nickname.lower() == channel.lower():
 26             self.factory.getUser(msg
 27             ).addErrback(lambda _: "Internal error in server"
 28             ).addCallback(lambda m: irc.IRCClient.msg(self, user,
msg+': '+m))
 29
 30 class FingerService(service.Service):
 31     def __init__(self, filename):
 32         self.filename = filename
 33         self._read()
 34     def _read(self):
 35         self.users = {}
 36         for line in file(self.filename):
 37             user, status = line.split(':', 1)
 38             user = user.strip()
 39             status = status.strip()
```

```
  40             self.users[user] = status
  41         self.call = reactor.callLater(30, self._read)
  42     def getUser(self, user):
  43         return defer.succeed(self.users.get(user, "No such user"))
  44     def getFingerFactory(self):
  45         f = protocol.ServerFactory()
  46         f.protocol, f.getUser = FingerProtocol, self.getUser
  47         return f
  48     def getResource(self):
  49         r = resource.Resource()
  50         r.getChild = (lambda path, request:
  51                       static.Data('<h1>%s</h1><p>%s</p>' %
  52                       tuple(map(cgi.escape,
  53                       [path,self.users.get(path,
  54                       "No such user <p/> usage: site/user")])),
  55                       'text/html'))
  56         return r
  57
  58     def getIRCBot(self, nickname):
  59         f = protocol.ReconnectingClientFactory()
  60         f.protocol,f.nickname,f.getUser =
IRCReplyBot,nickname,self.getUser
  61         return f
  62
  63 application = service.Application('finger', uid=1, gid=1)
  64 f = FingerService('/etc/users')
  65 serviceCollection = service.IServiceCollection(application)
  66 internet.TCPServer(79, f.getFingerFactory()
  67                    ).setServiceParent(serviceCollection)
  68 internet.TCPServer(8000, server.Site(f.getResource())
  69                    ).setServiceParent(serviceCollection)
  70 internet.TCPClient('irc.freenode.org', 6667,
f.getIRCBot('fingerbot')
  71                    ).setServiceParent(serviceCollection)
```

源程序 - listings/finger/finger16.py

7. 添加 XML-RPC 支持 (Add XML-RPC Support)

在 twisted 中，只是把对 XML-RPC 支持当作另外一个资源来处理。这个资源通过 render() 方法仍然可以支持 GET 访问，但这个方法一般都预留为一个还没有实现的接口函数。注意：有可能会从 XML-RPC 方法中返回延期对象。当然，客户端是只有等到这个延期对象被触发才会获得应答的。

In Twisted, XML-RPC support is handled just as though it was another resource. That resource will still support GET calls normally through render(), but that is usually left unimplemented. Note that it is possible to return deferreds from XML-RPC methods. The client, of course, will not get the answer until the deferred is triggered.

切换行号显示

```
   1 # 从文件读取资料，发布到 web、IRC、XML-RPC
```

```
  2 # Read from file, announce on the web, irc, xml-rpc
  3 from twisted.application import internet, service
  4 from twisted.internet import protocol, reactor, defer
  5 from twisted.protocols import basic, irc
  6 from twisted.web import resource, server, static, xmlrpc
  7 import cgi
  8 class FingerProtocol(basic.LineReceiver):
  9     def lineReceived(self, user):
 10         self.factory.getUser(user
 11         ).addErrback(lambda _: "Internal error in server"
 12         ).addCallback(lambda m:
 13                       (self.transport.write(m+"\r\n"),
 14                        self.transport.loseConnection()))
 15 class FingerSetterProtocol(basic.LineReceiver):
 16     def connectionMade(self): self.lines = []
 17     def lineReceived(self, line): self.lines.append(line)
 18     def connectionLost(self,reason):
self.factory.setUser(*self.lines[:2])
 19 class IRCReplyBot(irc.IRCClient):
 20     def connectionMade(self):
 21         self.nickname = self.factory.nickname
 22         irc.IRCClient.connectionMade(self)
 23     def privmsg(self, user, channel, msg):
 24         user = user.split('!')[0]
 25         if self.nickname.lower() == channel.lower():
 26             self.factory.getUser(msg
 27             ).addErrback(lambda _: "Internal error in server"
 28             ).addCallback(lambda m: irc.IRCClient.msg(self, user,
msg+': '+m))
 29
 30 class FingerService(service.Service):
 31     def __init__(self, filename):
 32         self.filename = filename
 33         self._read()
 34     def _read(self):
 35         self.users = {}
 36         for line in file(self.filename):
 37             user, status = line.split(':', 1)
 38             user = user.strip()
 39             status = status.strip()
 40             self.users[user] = status
 41         self.call = reactor.callLater(30, self._read)
 42     def getUser(self, user):
 43         return defer.succeed(self.users.get(user, "No such user"))
 44     def getFingerFactory(self):
 45         f = protocol.ServerFactory()
 46         f.protocol, f.getUser = FingerProtocol, self.getUser
 47         return f
 48     def getResource(self):
 49         r = resource.Resource()
```

```
  50         r.getChild = (lambda path, request:
  51                       static.Data('<h1>%s</h1><p>%s</p>' %
  52                       tuple(map(cgi.escape,
  53                       [path,self.users.get(path, "No such
user")]))),
  54                       'text/html'))
  55         x = xmlrpc.XMLRPC()
  56         x.xmlrpc_getUser = self.getUser
  57         r.putChild('RPC2', x)
  58         return r
  59     def getIRCBot(self, nickname):
  60         f = protocol.ReconnectingClientFactory()
  61         f.protocol,f.nickname,f.getUser =
IRCReplyBot,nickname,self.getUser
  62         return f
  63
  64 application = service.Application('finger', uid=1, gid=1)
  65 f = FingerService('/etc/users')
  66 serviceCollection = service.IServiceCollection(application)
  67 internet.TCPServer(79, f.getFingerFactory()
  68                    ).setServiceParent(serviceCollection)
  69 internet.TCPServer(8000, server.Site(f.getResource())
  70                    ).setServiceParent(serviceCollection)
  71 internet.TCPClient('irc.freenode.org', 6667,
f.getIRCBot('fingerbot')
  72                    ).setServiceParent(serviceCollection)
```

源程序 - listings/finger/finger17.py

8. 一个用来测试 XMLRPC finger 的简易客户端 (A simple client to test the XMLRPC finger:)

切换行号显示

```
1 # 测试 xmlrpc finger
2 # testing xmlrpc finger
3
4 import xmlrpclib
5 server = xmlrpclib.Server('http://127.0.0.1:8000/RPC2')
6 print server.getUser('moshez')
```

源程序 - listings/finger/fingerXRclient.py

返回目录 --> TwistedTUT

twistedTUT02 (2004-08-09 02:08:21由dreamingk编辑)

- twistedTUT03

Finger演化：整理finger代码

The Evolution of Finger: cleaning up the finger code -- dreamingk [2004-08-09 02:09:38]

目录

1. 介绍(Introduction)

这是Twisted教程《Twisted入门 - Finger演化》的第三部分。

This is the third part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在本节教程中，我们将整理代码，以便让它更接近可读和可扩展的风格。

In this section of the tutorial, we'll clean up our code so that it is closer to a readable and extendable style.

1.1. 编写可读代码(Write Readable Code)

Application的上一个版本有许多的伎俩。我们避开了子类化，不支持在web上的用户列表之类的事情，和去掉所有的空行 -- 所有这些是为了让程序更短。这里我们将后退一步，子类化(更自然地说是一个子类)可以做一些接受多行的工作来处理它们，等等此类事情。这里显示了一种开发Twisted应用的更好的风格，尽管在前一阶段中的伎俩有时可以用在被丢弃的原型中。

The last version of the application had a lot of hacks. We avoided sub-classing, didn't support things like user listings over the web, and removed all blank lines -- all in the interest of code which is shorter. Here we take a step back, subclass what is more naturally a subclass, make things which should take multiple lines take them, etc. This shows a much better style of developing Twisted applications, though the hacks in the previous stages are sometimes used in throw-away prototypes.

Toggle line numbers

```
 1 # Do everything properly
 2 from twisted.application import internet, service
 3 from twisted.internet import protocol, reactor, defer
 4 from twisted.protocols import basic, irc
 5 from twisted.web import resource, server, static, xmlrpc
 6 import cgi
 7
 8 def catchError(err):
 9     return "Internal error in server"
10
11 class FingerProtocol(basic.LineReceiver):
12
13     def lineReceived(self, user):
14         d = self.factory.getUser(user)
15         d.addErrback(catchError)
```

```
16          def writeValue(value):
17              self.transport.write(value+'\n')
18              self.transport.loseConnection()
19          d.addCallback(writeValue)
20
21
22  class FingerSetterProtocol(basic.LineReceiver):
23
24      def connectionMade(self):
25          self.lines = []
26
27      def lineReceived(self, line):
28          self.lines.append(line)
29
30      def connectionLost(self, reason):
31          self.factory.setUser(*self.lines[:2])
32
33
34  class IRCReplyBot(irc.IRCClient):
35
36      def connectionMade(self):
37          self.nickname = self.factory.nickname
38          irc.IRCClient.connectionMade(self)
39
40      def privmsg(self, user, channel, msg):
41          user = user.split('!')[0]
42          if self.nickname.lower() == channel.lower():
43              d = self.factory.getUser(msg)
44              d.addErrback(catchError)
45              d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
46              d.addCallback(lambda m: self.msg(user, m))
47
48
49  class UserStatusTree(resource.Resource):
50      def __init__(self, service):
51          resource.Resource.__init__(self)
52          self.service = service
53
54      def render_GET(self, request):
55          d = self.service.getUsers()
56          def formatUsers(users):
57              l = ['<li><a href="%s">%s</a></li>' % (user, user)
58                   for user in users]
59              return '<ul>'+''.join(l)+'</ul>'
60          d.addCallback(formatUsers)
61          d.addCallback(request.write)
62          d.addCallback(lambda _: request.finish())
63          return server.NOT_DONE_YET
64
65      def getChild(self, path, request):
```

```
66          if path=="":
67              return UserStatusTree(self.service)
68          else:
69              return UserStatus(path, self.service)
70
71  class UserStatus(resource.Resource):
72
73      def __init__(self, user, service):
74          resource.Resource.__init__(self)
75          self.user = user
76          self.service = service
77
78      def render_GET(self, request):
79          d = self.service.getUser(self.user)
80          d.addCallback(cgi.escape)
81          d.addCallback(lambda m:
82                        '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
83          d.addCallback(request.write)
84          d.addCallback(lambda _: request.finish())
85          return server.NOT_DONE_YET
86
87
88  class UserStatusXR(xmlrpc.XMLRPC):
89
90      def __init__(self, service):
91          xmlrpc.XMLRPC.__init__(self)
92          self.service = service
93
94      def xmlrpc_getUser(self, user):
95          return self.service.getUser(user)
96
97
98  class FingerService(service.Service):
99
100     def __init__(self, filename):
101         self.filename = filename
102         self._read()
103
104     def _read(self):
105         self.users = {}
106         for line in file(self.filename):
107             user, status = line.split(':', 1)
108             user = user.strip()
109             status = status.strip()
110             self.users[user] = status
111         self.call = reactor.callLater(30, self._read)
112
113     def getUser(self, user):
114         return defer.succeed(self.users.get(user, "No such user"))
115
```

```
116     def getUsers(self):
117         return defer.succeed(self.users.keys())
118 
119     def getFingerFactory(self):
120         f = protocol.ServerFactory()
121         f.protocol = FingerProtocol
122         f.getUser = self.getUser
123         return f
124 
125     def getResource(self):
126         r = UserStatusTree(self)
127         x = UserStatusXR(self)
128         r.putChild('RPC2', x)
129         return r
130 
131     def getIRCBot(self, nickname):
132         f = protocol.ReconnectingClientFactory()
133         f.protocol = IRCReplyBot
134         f.nickname = nickname
135         f.getUser = self.getUser
136         return f
137 
138 application = service.Application('finger', uid=1, gid=1)
139 f = FingerService('/etc/users')
140 serviceCollection = service.IServiceCollection(application)
141 internet.TCPServer(79, f.getFingerFactory()
142                    ).setServiceParent(serviceCollection)
143 internet.TCPServer(8000, server.Site(f.getResource())
144                    ).setServiceParent(serviceCollection)
145 internet.TCPClient('irc.freenode.org', 6667,
f.getIRCBot('fingerbot')
146                    ).setServiceParent(serviceCollection)
```

Source listing - listings/finger/finger18.py

return index-->TwistedTUT

twistedTUT03 (2004-08-09 02:09:38由dreamingk编辑)

- twistedTUT04

The Evolution of Finger: moving to a component based architecture
Finger的演化：步入基于组件的结构

-- dreamingk [2004-08-09 02:10:32]

目录

1. 简介 Introduction

This is the fourth part of the Twisted
tutorial Twisted from Scratch, or The Evolution of Finger.

这是Twisted教程，Finger的演化的第四部分。

In this section of the tutorial, we'll move our code to a component architecture so that adding new features is trivial.

在这一部分，我们将把代码放到一个组件结构中，这样加入新的特性就很简单了。

2. 写可维护的代码 Write Maintainable Code

In the last version, the service class was three times longer than any other class, and was hard to understand. This was because it turned out to have multiple responsibilities. It had to know how to access user information, by rereading the file every half minute, but also how to display itself in a myriad of protocols. Here, we used the component-based architecture that Twisted provides to achieve a separation of concerns. All the service is responsible for, now, is supporting getUser/getUsers. It declares its support via the implements
keyword. Then, adapters are used to make this service look like an appropriate class for various things: for supplying a finger factory to TCPServer, for supplying a resource to site's constructor, and to provide an IRC client factory for TCPClient. All the adapters use are the methods in FingerService
they are declared to use: getUser/getUsers. We could, of course, skip the interfaces and let the configuration code use things like FingerFactoryFromService(f) directly. However, using interfaces provides the same flexibility inheritance gives: future subclasses can override the adapters.

在上个版本中，服务类是其它类的三倍长，而且也难懂。这是因为它具有多重职责。 必须要知道如何取得用户信息，通过每半分钟重新读取一次文件，还要知道如何通过不同的协议来显示信息。 这里，我们使用Twisted提供的基于组件的结构，由此分而治之。所有的服务都提供getUser/getUsers。 通过implement关键字。然后适配器就能让这个服务成为一个可以用于多种用途的类：向TCPServer提供finger factory， 提供资源给站点的构造器，提供IRC客户端factory给TCPClient。适配器使用的是FingerService声明可用的方法：getUser/getUsers。 我们可以，当然了，忽略接口让配置代码直接使用FingerFactoryFromService。但是使用接口则可以提供和继承一样的灵活性：将来子类可以重载适配器。

Toggle line numbers

```
1 # Do everything properly, and componentize
2 from twisted.application import internet, service
3 from twisted.internet import protocol, reactor, defer
```

```
 4 from twisted.protocols import basic, irc
 5 from twisted.python import components
 6 from twisted.web import resource, server, static, xmlrpc
 7 import cgi
 8
 9 class IFingerService(components.Interface):
10
11     def getUser(self, user):
12         """Return a deferred returning a string"""
13
14     def getUsers(self):
15         """Return a deferred returning a list of strings"""
16
17 class IFingerSetterService(components.Interface):
18
19     def setUser(self, user, status):
20         """Set the user's status to something"""
21
22 def catchError(err):
23     return "Internal error in server"
24
25 class FingerProtocol(basic.LineReceiver):
26
27     def lineReceived(self, user):
28         d = self.factory.getUser(user)
29         d.addErrback(catchError)
30         def writeValue(value):
31             self.transport.write(value+'\n')
32             self.transport.loseConnection()
33         d.addCallback(writeValue)
34
35
36 class IFingerFactory(components.Interface):
37
38     def getUser(self, user):
39         """Return a deferred returning a string"""
40
41     def buildProtocol(self, addr):
42         """Return a protocol returning a string"""
43
44
45 class FingerFactoryFromService(protocol.ServerFactory):
46
47     __implements__ = IFingerFactory,
48
49     protocol = FingerProtocol
50
51     def __init__(self, service):
52         self.service = service
53
```

```
54     def getUser(self, user):
55         return self.service.getUser(user)
56 
57 components.registerAdapter(FingerFactoryFromService,
58                            IFingerService,
59                            IFingerFactory)
60 
61 class FingerSetterProtocol(basic.LineReceiver):
62 
63     def connectionMade(self):
64         self.lines = []
65 
66     def lineReceived(self, line):
67         self.lines.append(line)
68 
69     def connectionLost(self, reason):
70         if len(self.lines) == 2:
71             self.factory.setUser(*self.lines)
72 
73 
74 class IFingerSetterFactory(components.Interface):
75 
76     def setUser(self, user, status):
77         """Return a deferred returning a string"""
78 
79     def buildProtocol(self, addr):
80         """Return a protocol returning a string"""
81 
82 
83 class FingerSetterFactoryFromService(protocol.ServerFactory):
84 
85     __implements__ = IFingerSetterFactory,
86 
87     protocol = FingerSetterProtocol
88 
89     def __init__(self, service):
90         self.service = service
91 
92     def setUser(self, user, status):
93         self.service.setUser(user, status)
94 
95 
96 components.registerAdapter(FingerSetterFactoryFromService,
97                            IFingerSetterService,
98                            IFingerSetterFactory)
99 
100 class IRCReplyBot(irc.IRCClient):
101 
102     def connectionMade(self):
103         self.nickname = self.factory.nickname
```

```
104         irc.IRCClient.connectionMade(self)
105
106     def privmsg(self, user, channel, msg):
107         user = user.split('!')[0]
108         if self.nickname.lower() == channel.lower():
109             d = self.factory.getUser(msg)
110             d.addErrback(catchError)
111             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
112             d.addCallback(lambda m: self.msg(user, m))
113
114
115 class IIRCClientFactory(components.Interface):
116
117     """
118     @ivar nickname
119     """
120
121     def getUser(self, user):
122         """Return a deferred returning a string"""
123
124     def buildProtocol(self, addr):
125         """Return a protocol"""
126
127
128 class IRCClientFactoryFromService(protocol.ClientFactory):
129
130     __implements__ = IIRCClientFactory,
131
132     protocol = IRCReplyBot
133     nickname = None
134
135     def __init__(self, service):
136         self.service = service
137
138     def getUser(self, user):
139         return self.service.getUser(user)
140
141 components.registerAdapter(IRCClientFactoryFromService,
142                            IFingerService,
143                            IIRCClientFactory)
144
145 class UserStatusTree(resource.Resource):
146
147     __implements__ = resource.IResource,
148
149     def __init__(self, service):
150         resource.Resource.__init__(self)
151         self.service = service
152         self.putChild('RPC2', UserStatusXR(self.service))
153
```

```
154     def render_GET(self, request):
155         d = self.service.getUsers()
156         def formatUsers(users):
157             l = ['<li><a href="%s">%s</a></li>' % (user, user)
158                  for user in users]
159             return '<ul>'+''.join(l)+'</ul>'
160         d.addCallback(formatUsers)
161         d.addCallback(request.write)
162         d.addCallback(lambda _: request.finish())
163         return server.NOT_DONE_YET
164 
165     def getChild(self, path, request):
166         if path=="":
167             return UserStatusTree(self.service)
168         else:
169             return UserStatus(path, self.service)
170 
171 components.registerAdapter(UserStatusTree, IFingerService,
172                            resource.IResource)
173 
174 class UserStatus(resource.Resource):
175 
176     def __init__(self, user, service):
177         resource.Resource.__init__(self)
178         self.user = user
179         self.service = service
180 
181     def render_GET(self, request):
182         d = self.service.getUser(self.user)
183         d.addCallback(cgi.escape)
184         d.addCallback(lambda m:
185                       '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
186         d.addCallback(request.write)
187         d.addCallback(lambda _: request.finish())
188         return server.NOT_DONE_YET
189 
190 
191 class UserStatusXR(xmlrpc.XMLRPC):
192 
193     def __init__(self, service):
194         xmlrpc.XMLRPC.__init__(self)
195         self.service = service
196 
197     def xmlrpc_getUser(self, user):
198         return self.service.getUser(user)
199 
200 
201 class FingerService(service.Service):
202 
203     __implements__ = IFingerService,
```

```
204
205     def __init__(self, filename):
206         self.filename = filename
207         self._read()
208
209     def _read(self):
210         self.users = {}
211         for line in file(self.filename):
212             user, status = line.split(':', 1)
213             user = user.strip()
214             status = status.strip()
215             self.users[user] = status
216         self.call = reactor.callLater(30, self._read)
217
218     def getUser(self, user):
219         return defer.succeed(self.users.get(user, "No such user"))
220
221     def getUsers(self):
222         return defer.succeed(self.users.keys())
223
224
225 application = service.Application('finger', uid=1, gid=1)
226 f = FingerService('/etc/users')
227 serviceCollection = service.IServiceCollection(application)
228 internet.TCPServer(79, IFingerFactory(f)
229                    ).setServiceParent(serviceCollection)
230 internet.TCPServer(8000, server.Site(resource.IResource(f))
231                    ).setServiceParent(serviceCollection)
232 i = IIRCClientFactory(f)
233 i.nickname = 'fingerbot'
234 internet.TCPClient('irc.freenode.org', 6667, i
235                    ).setServiceParent(serviceCollection)
236
237 Source listing - listings/finger/finger19.py
238 Advantages of Latest Version
239 Readable -- each class is short
240 Maintainable -- each class knows only about interfaces
241 Dependencies between code parts are minimized
242 Example: writing a new IFingerService is easy
243 class IFingerSetterService(components.Interface):
244
245     def setUser(self, user, status):
246         """Set the user's status to something"""
247
248 # Advantages of latest version
249
250 class MemoryFingerService(service.Service):
251
252     __implements__ = IFingerService, IFingerSetterService
253
```

```
254     def __init__(self, **kwargs):
255         self.users = kwargs
256 
257     def getUser(self, user):
258         return defer.succeed(self.users.get(user, "No such user"))
259 
260     def getUsers(self):
261         return defer.succeed(self.users.keys())
262 
263     def setUser(self, user, status):
264         self.users[user] = status
265 
266 
267 f = MemoryFingerService(moshez='Happy and well')
268 serviceCollection = service.IServiceCollection(application)
269 internet.TCPServer(1079, IFingerSetterFactory(f),
interface='127.0.0.1'
270                    ).setServiceParent(serviceCollection)
271 
272 Source listing - listings/finger/finger19a_changes.py
273 Full source code here:
274 
275 # Do everything properly, and componentize
276 from twisted.application import internet, service
277 from twisted.internet import protocol, reactor, defer
278 from twisted.protocols import basic, irc
279 from twisted.python import components
280 from twisted.web import resource, server, static, xmlrpc
281 import cgi
282 
283 class IFingerService(components.Interface):
284 
285     def getUser(self, user):
286         """Return a deferred returning a string"""
287 
288     def getUsers(self):
289         """Return a deferred returning a list of strings"""
290 
291 class IFingerSetterService(components.Interface):
292 
293     def setUser(self, user, status):
294         """Set the user's status to something"""
295 
296 def catchError(err):
297     return "Internal error in server"
298 
299 class FingerProtocol(basic.LineReceiver):
300 
301     def lineReceived(self, user):
302         d = self.factory.getUser(user)
```

```
303         d.addErrback(catchError)
304         def writeValue(value):
305             self.transport.write(value+'\n')
306             self.transport.loseConnection()
307         d.addCallback(writeValue)
308 
309 
310 class IFingerFactory(components.Interface):
311 
312     def getUser(self, user):
313         """Return a deferred returning a string"""
314 
315     def buildProtocol(self, addr):
316         """Return a protocol returning a string"""
317 
318 
319 class FingerFactoryFromService(protocol.ServerFactory):
320 
321     __implements__ = IFingerFactory,
322 
323     protocol = FingerProtocol
324 
325     def __init__(self, service):
326         self.service = service
327 
328     def getUser(self, user):
329         return self.service.getUser(user)
330 
331 components.registerAdapter(FingerFactoryFromService,
332                            IFingerService,
333                            IFingerFactory)
334 
335 class FingerSetterProtocol(basic.LineReceiver):
336 
337     def connectionMade(self):
338         self.lines = []
339 
340     def lineReceived(self, line):
341         self.lines.append(line)
342 
343     def connectionLost(self, reason):
344         if len(self.lines) == 2:
345             self.factory.setUser(*self.lines)
346 
347 
348 class IFingerSetterFactory(components.Interface):
349 
350     def setUser(self, user, status):
351         """Return a deferred returning a string"""
352 
```

```
353     def buildProtocol(self, addr):
354         """Return a protocol returning a string"""
355 
356 
357 class FingerSetterFactoryFromService(protocol.ServerFactory):
358 
359     __implements__ = IFingerSetterFactory,
360 
361     protocol = FingerSetterProtocol
362 
363     def __init__(self, service):
364         self.service = service
365 
366     def setUser(self, user, status):
367         self.service.setUser(user, status)
368 
369 
370 components.registerAdapter(FingerSetterFactoryFromService,
371                            IFingerSetterService,
372                            IFingerSetterFactory)
373 
374 class IRCReplyBot(irc.IRCClient):
375 
376     def connectionMade(self):
377         self.nickname = self.factory.nickname
378         irc.IRCClient.connectionMade(self)
379 
380     def privmsg(self, user, channel, msg):
381         user = user.split('!')[0]
382         if self.nickname.lower() == channel.lower():
383             d = self.factory.getUser(msg)
384             d.addErrback(catchError)
385             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
386             d.addCallback(lambda m: self.msg(user, m))
387 
388 
389 class IIRCClientFactory(components.Interface):
390 
391     """
392     @ivar nickname
393     """
394 
395     def getUser(self, user):
396         """Return a deferred returning a string"""
397 
398     def buildProtocol(self, addr):
399         """Return a protocol"""
400 
401 
402 class IRCClientFactoryFromService(protocol.ClientFactory):
```

```
403
404     __implements__ = IIRCClientFactory,
405
406     protocol = IRCReplyBot
407     nickname = None
408
409     def __init__(self, service):
410         self.service = service
411
412     def getUser(self, user):
413         return self.service.getUser(user)
414
415 components.registerAdapter(IRCClientFactoryFromService,
416                            IFingerService,
417                            IIRCClientFactory)
418
419 class UserStatusTree(resource.Resource):
420
421     __implements__ = resource.IResource,
422
423     def __init__(self, service):
424         resource.Resource.__init__(self)
425         self.service = service
426         self.putChild('RPC2', UserStatusXR(self.service))
427
428     def render_GET(self, request):
429         d = self.service.getUsers()
430         def formatUsers(users):
431             l = ['<li><a href="%s">%s</a></li>' % (user, user)
432                  for user in users]
433             return '<ul>'+''.join(l)+'</ul>'
434         d.addCallback(formatUsers)
435         d.addCallback(request.write)
436         d.addCallback(lambda _: request.finish())
437         return server.NOT_DONE_YET
438
439     def getChild(self, path, request):
440         if path=="":
441             return UserStatusTree(self.service)
442         else:
443             return UserStatus(path, self.service)
444
445 components.registerAdapter(UserStatusTree, IFingerService,
446                            resource.IResource)
447
448 class UserStatus(resource.Resource):
449
450     def __init__(self, user, service):
451         resource.Resource.__init__(self)
452         self.user = user
```

```
453         self.service = service
454 
455     def render_GET(self, request):
456         d = self.service.getUser(self.user)
457         d.addCallback(cgi.escape)
458         d.addCallback(lambda m:
459                       '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
460         d.addCallback(request.write)
461         d.addCallback(lambda _: request.finish())
462         return server.NOT_DONE_YET
463 
464 
465 class UserStatusXR(xmlrpc.XMLRPC):
466 
467     def __init__(self, service):
468         xmlrpc.XMLRPC.__init__(self)
469         self.service = service
470 
471     def xmlrpc_getUser(self, user):
472         return self.service.getUser(user)
473 
474 class MemoryFingerService(service.Service):
475 
476     __implements__ = IFingerService, IFingerSetterService
477 
478     def __init__(self, **kwargs):
479         self.users = kwargs
480 
481     def getUser(self, user):
482         return defer.succeed(self.users.get(user, "No such user"))
483 
484     def getUsers(self):
485         return defer.succeed(self.users.keys())
486 
487     def setUser(self, user, status):
488         self.users[user] = status
489 
490 
491 application = service.Application('finger', uid=1, gid=1)
492 f = MemoryFingerService(moshez='Happy and well')
493 serviceCollection = service.IServiceCollection(application)
494 internet.TCPServer(79, IFingerFactory(f)
495                    ).setServiceParent(serviceCollection)
496 internet.TCPServer(8000, server.Site(resource.IResource(f))
497                    ).setServiceParent(serviceCollection)
498 i = IIRCClientFactory(f)
499 i.nickname = 'fingerbot'
500 internet.TCPClient('irc.freenode.org', 6667, i
501                    ).setServiceParent(serviceCollection)
502 internet.TCPServer(1079, IFingerSetterFactory(f),
```

```
interface='127.0.0.1'
 503                    ).setServiceParent(serviceCollection)
```

Source listing - listings/finger/finger19a.py

Aspect-Oriented Programming At last, an example of aspect-oriented programming that isn't about logging or timing. This code is actually useful! Watch how aspect-oriented programming helps you write less code and have fewer dependencies!

至少可以算作是面向方面编程的一个和logging以及timing无关的例子。这段代码很有用！要知道面向方面编程让你写更少的代码，还减小依赖！

翻译：Bruce Who(whoonline@msn.com)

return index-->TwistedTUT

Version: 1.3.0

twistedTUT04 (2004-08-09 02:10:32由dreamingk编辑)

- twistedTUT05

The Evolution of Finger: pluggable backends

Finger的演化：可插入式后端

-- dreamingk [2004-08-09 02:11:23]

目录

1. 简介（Introduction）

This is the fifth part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

这是Twisted教程——“Twisted from Scratch, or The Evolution of Finger”的第五部分

In this part we will add new several new backends to our finger service using the component-based architecture developed in The Evolution of Finger: moving to a component based architecture. This will show just how convenient it is to implement new back-ends when we move to a component based architecture. Note that here we also use an interface we previously wrote, FingerSetterFactory, by supporting one single method. We manage to preserve the service's ignorance of the network.

在这个部分中，我们将为我们在前边做过的“基于组件（Component-based）”的结构构建的Finger服务上增加一些新的后端支持。 你会发现在“基于组件（Component-based）”的结构的基础上，实现新的后端支持是多么的便利。 你要注意的是，我们为了支持一个方法，会使用一个过去写过的接口——FingerSettingFactory。 我们要设法保护我们的服务对于网络环境的一知半解。

2. 另一个后端（Back-end）

Toggle line numbers

```
1 from twisted.internet import protocol, reactor, defer, utils
2 import pwd
3
4 # Another back-end
5
6 class LocalFingerService(service.Service):
7
8     __implements__ = IFingerService
9
10    def getUser(self, user):
11    # need a local finger daemon running for this to work
12        return utils.getProcessOutput("finger", [user])
13
14    def getUsers(self):
15        return defer.succeed([])
16
17
18 f = LocalFingerService()
19
```

Source listing - listings/finger/finger19b_changes.py
Full source code here:

```
# Do everything properly, and componentize
from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer, utils
from twisted.protocols import basic, irc
from twisted.python import components
from twisted.web import resource, server, static, xmlrpc
import cgi
import pwd

class IFingerService(components.Interface):

    def getUser(self, user):
        """Return a deferred returning a string"""

    def getUsers(self):
        """Return a deferred returning a list of strings"""

class IFingerSetterService(components.Interface):

    def setUser(self, user, status):
        """Set the user's status to something"""

class IFingerSetterService(components.Interface):

    def setUser(self, user, status):
        """Set the user's status to something"""

def catchError(err):
    return "Internal error in server"

class FingerProtocol(basic.LineReceiver):

    def lineReceived(self, user):
        d = self.factory.getUser(user)
        d.addErrback(catchError)
        def writeValue(value):
            self.transport.write(value+'\n')
            self.transport.loseConnection()
        d.addCallback(writeValue)


class IFingerFactory(components.Interface):

    def getUser(self, user):
        """Return a deferred returning a string"""

    def buildProtocol(self, addr):
```

```
70          """Return a protocol returning a string"""
71
72
73  class FingerFactoryFromService(protocol.ServerFactory):
74
75      __implements__ = IFingerFactory,
76
77      protocol = FingerProtocol
78
79      def __init__(self, service):
80          self.service = service
81
82      def getUser(self, user):
83          return self.service.getUser(user)
84
85  components.registerAdapter(FingerFactoryFromService,
86                             IFingerService,
87                             IFingerFactory)
88
89  class FingerSetterProtocol(basic.LineReceiver):
90
91      def connectionMade(self):
92          self.lines = []
93
94      def lineReceived(self, line):
95          self.lines.append(line)
96
97      def connectionLost(self, reason):
98          if len(self.lines) == 2:
99              self.factory.setUser(*self.lines)
100
101
102 class IFingerSetterFactory(components.Interface):
103
104     def setUser(self, user, status):
105         """Return a deferred returning a string"""
106
107     def buildProtocol(self, addr):
108         """Return a protocol returning a string"""
109
110
111 class FingerSetterFactoryFromService(protocol.ServerFactory):
112
113     __implements__ = IFingerSetterFactory,
114
115     protocol = FingerSetterProtocol
116
117     def __init__(self, service):
118         self.service = service
119
```

```
120     def setUser(self, user, status):
121         self.service.setUser(user, status)
122 
123 
124 components.registerAdapter(FingerSetterFactoryFromService,
125                            IFingerSetterService,
126                            IFingerSetterFactory)
127 
128 class IRCReplyBot(irc.IRCClient):
129 
130     def connectionMade(self):
131         self.nickname = self.factory.nickname
132         irc.IRCClient.connectionMade(self)
133 
134     def privmsg(self, user, channel, msg):
135         user = user.split('!')[0]
136         if self.nickname.lower() == channel.lower():
137             d = self.factory.getUser(msg)
138             d.addErrback(catchError)
139             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
140             d.addCallback(lambda m: self.msg(user, m))
141 
142 
143 class IIRCClientFactory(components.Interface):
144 
145     """
146     @ivar nickname
147     """
148 
149     def getUser(self, user):
150         """Return a deferred returning a string"""
151 
152     def buildProtocol(self, addr):
153         """Return a protocol"""
154 
155 
156 class IRCClientFactoryFromService(protocol.ClientFactory):
157 
158     __implements__ = IIRCClientFactory,
159 
160     protocol = IRCReplyBot
161     nickname = None
162 
163     def __init__(self, service):
164         self.service = service
165 
166     def getUser(self, user):
167         return self.service.getUser(user)
168 
169 components.registerAdapter(IRCClientFactoryFromService,
```

```
170                             IFingerService,
171                             IIRCClientFactory)
172
173 class UserStatusTree(resource.Resource):
174
175     __implements__ = resource.IResource,
176
177     def __init__(self, service):
178         resource.Resource.__init__(self)
179         self.service = service
180         self.putChild('RPC2', UserStatusXR(self.service))
181
182     def render_GET(self, request):
183         d = self.service.getUsers()
184         def formatUsers(users):
185             l = ['<li><a href="%s">%s</a></li>' % (user, user)
186                  for user in users]
187             return '<ul>'+''.join(l)+'</ul>'
188         d.addCallback(formatUsers)
189         d.addCallback(request.write)
190         d.addCallback(lambda _: request.finish())
191         return server.NOT_DONE_YET
192
193     def getChild(self, path, request):
194         if path=="":
195             return UserStatusTree(self.service)
196         else:
197             return UserStatus(path, self.service)
198
199 components.registerAdapter(UserStatusTree, IFingerService,
200                             resource.IResource)
201
202 class UserStatus(resource.Resource):
203
204     def __init__(self, user, service):
205         resource.Resource.__init__(self)
206         self.user = user
207         self.service = service
208
209     def render_GET(self, request):
210         d = self.service.getUser(self.user)
211         d.addCallback(cgi.escape)
212         d.addCallback(lambda m:
213                       '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
214         d.addCallback(request.write)
215         d.addCallback(lambda _: request.finish())
216         return server.NOT_DONE_YET
217
218
219 class UserStatusXR(xmlrpc.XMLRPC):
```

```
220
221     def __init__(self, service):
222         xmlrpc.XMLRPC.__init__(self)
223         self.service = service
224
225     def xmlrpc_getUser(self, user):
226         return self.service.getUser(user)
227
228
229 class FingerService(service.Service):
230
231     __implements__ = IFingerService,
232
233     def __init__(self, filename):
234         self.filename = filename
235         self._read()
236
237     def _read(self):
238         self.users = {}
239         for line in file(self.filename):
240             user, status = line.split(':', 1)
241             user = user.strip()
242             status = status.strip()
243             self.users[user] = status
244         self.call = reactor.callLater(30, self._read)
245
246     def getUser(self, user):
247         return defer.succeed(self.users.get(user, "No such user"))
248
249     def getUsers(self):
250         return defer.succeed(self.users.keys())
251
252 # Another back-end
253
254 class LocalFingerService(service.Service):
255
256     __implements__ = IFingerService
257
258     def getUser(self, user):
259     # need a local finger daemon running for this to work
260         return utils.getProcessOutput("finger", [user])
261
262     def getUsers(self):
263         return defer.succeed([])
264
265
266 application = service.Application('finger', uid=1, gid=1)
267 f = LocalFingerService()
268 serviceCollection = service.IServiceCollection(application)
269 internet.TCPServer(79, IFingerFactory(f)
```

```
270                     ).setServiceParent(serviceCollection)
271 internet.TCPServer(8000, server.Site(resource.IResource(f))
272                     ).setServiceParent(serviceCollection)
273 i = IIRCClientFactory(f)
274 i.nickname = 'fingerbot'
275 internet.TCPClient('irc.freenode.org', 6667, i
276                     ).setServiceParent(serviceCollection)
277
278 Source listing - listings/finger/finger19b.py
279
280 We've already written this, but now we get more for less work:
the network code is completely separate from the back-end.
281
282 Yet Another Back-end: Doing the Standard Thing
283 from twisted.internet import protocol, reactor, defer, utils
284 import pwd
285 import os
286
287
288 # Yet another back-end
289
290 class LocalFingerService(service.Service):
291
292     __implements__ = IFingerService
293
294     def getUser(self, user):
295         user = user.strip()
296         try:
297             entry = pwd.getpwnam(user)
298         except KeyError:
299             return defer.succeed("No such user")
300         try:
301             f = file(os.path.join(entry[5],'.plan'))
302         except (IOError, OSError):
303             return defer.succeed("No such user")
304         data = f.read()
305         data = data.strip()
306         f.close()
307         return defer.succeed(data)
308
309     def getUsers(self):
310         return defer.succeed([])
311
312
313
314 f = LocalFingerService()
315
316 Source listing - listings/finger/finger19c_changes.py
317 Full source code here:
318
```

```
319 # Do everything properly, and componentize
320 from twisted.application import internet, service
321 from twisted.internet import protocol, reactor, defer, utils
322 from twisted.protocols import basic, irc
323 from twisted.python import components
324 from twisted.web import resource, server, static, xmlrpc
325 import cgi
326 import pwd
327 import os
328
329 class IFingerService(components.Interface):
330
331     def getUser(self, user):
332         """Return a deferred returning a string"""
333
334     def getUsers(self):
335         """Return a deferred returning a list of strings"""
336
337 class IFingerSetterService(components.Interface):
338
339     def setUser(self, user, status):
340         """Set the user's status to something"""
341
342 class IFingerSetterService(components.Interface):
343
344     def setUser(self, user, status):
345         """Set the user's status to something"""
346
347 def catchError(err):
348     return "Internal error in server"
349
350 class FingerProtocol(basic.LineReceiver):
351
352     def lineReceived(self, user):
353         d = self.factory.getUser(user)
354         d.addErrback(catchError)
355         def writeValue(value):
356             self.transport.write(value+'\n')
357             self.transport.loseConnection()
358         d.addCallback(writeValue)
359
360
361 class IFingerFactory(components.Interface):
362
363     def getUser(self, user):
364         """Return a deferred returning a string"""
365
366     def buildProtocol(self, addr):
367         """Return a protocol returning a string"""
368
```

```
369
370 class FingerFactoryFromService(protocol.ServerFactory):
371
372     __implements__ = IFingerFactory,
373
374     protocol = FingerProtocol
375
376     def __init__(self, service):
377         self.service = service
378
379     def getUser(self, user):
380         return self.service.getUser(user)
381
382 components.registerAdapter(FingerFactoryFromService,
383                            IFingerService,
384                            IFingerFactory)
385
386 class FingerSetterProtocol(basic.LineReceiver):
387
388     def connectionMade(self):
389         self.lines = []
390
391     def lineReceived(self, line):
392         self.lines.append(line)
393
394     def connectionLost(self, reason):
395         if len(self.lines) == 2:
396             self.factory.setUser(*self.lines)
397
398
399 class IFingerSetterFactory(components.Interface):
400
401     def setUser(self, user, status):
402         """Return a deferred returning a string"""
403
404     def buildProtocol(self, addr):
405         """Return a protocol returning a string"""
406
407
408 class FingerSetterFactoryFromService(protocol.ServerFactory):
409
410     __implements__ = IFingerSetterFactory,
411
412     protocol = FingerSetterProtocol
413
414     def __init__(self, service):
415         self.service = service
416
417     def setUser(self, user, status):
418         self.service.setUser(user, status)
```

```
419
420
421 components.registerAdapter(FingerSetterFactoryFromService,
422                             IFingerSetterService,
423                             IFingerSetterFactory)
424
425 class IRCReplyBot(irc.IRCClient):
426
427     def connectionMade(self):
428         self.nickname = self.factory.nickname
429         irc.IRCClient.connectionMade(self)
430
431     def privmsg(self, user, channel, msg):
432         user = user.split('!')[0]
433         if self.nickname.lower() == channel.lower():
434             d = self.factory.getUser(msg)
435             d.addErrback(catchError)
436             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
437             d.addCallback(lambda m: self.msg(user, m))
438
439
440 class IIRCClientFactory(components.Interface):
441
442     """
443     @ivar nickname
444     """
445
446     def getUser(self, user):
447         """Return a deferred returning a string"""
448
449     def buildProtocol(self, addr):
450         """Return a protocol"""
451
452
453 class IRCClientFactoryFromService(protocol.ClientFactory):
454
455     __implements__ = IIRCClientFactory,
456
457     protocol = IRCReplyBot
458     nickname = None
459
460     def __init__(self, service):
461         self.service = service
462
463     def getUser(self, user):
464         return self.service.getUser(user)
465
466 components.registerAdapter(IRCClientFactoryFromService,
467                             IFingerService,
468                             IIRCClientFactory)
```

```
469
470 class UserStatusTree(resource.Resource):
471
472     __implements__ = resource.IResource,
473
474     def __init__(self, service):
475         resource.Resource.__init__(self)
476         self.service = service
477         self.putChild('RPC2', UserStatusXR(self.service))
478
479     def render_GET(self, request):
480         d = self.service.getUsers()
481         def formatUsers(users):
482             l = ['<li><a href="%s">%s</a></li>' % (user, user)
483                  for user in users]
484             return '<ul>'+''.join(l)+'</ul>'
485         d.addCallback(formatUsers)
486         d.addCallback(request.write)
487         d.addCallback(lambda _: request.finish())
488         return server.NOT_DONE_YET
489
490     def getChild(self, path, request):
491         if path=="":
492             return UserStatusTree(self.service)
493         else:
494             return UserStatus(path, self.service)
495
496 components.registerAdapter(UserStatusTree, IFingerService,
497                            resource.IResource)
498
499 class UserStatus(resource.Resource):
500
501     def __init__(self, user, service):
502         resource.Resource.__init__(self)
503         self.user = user
504         self.service = service
505
506     def render_GET(self, request):
507         d = self.service.getUser(self.user)
508         d.addCallback(cgi.escape)
509         d.addCallback(lambda m:
510                       '<h1>%s</h1>'%self.user+'<p>%s</p>'%m)
511         d.addCallback(request.write)
512         d.addCallback(lambda _: request.finish())
513         return server.NOT_DONE_YET
514
515
516 class UserStatusXR(xmlrpc.XMLRPC):
517
518     def __init__(self, service):
```

```
519         xmlrpc.XMLRPC.__init__(self)
520         self.service = service
521 
522     def xmlrpc_getUser(self, user):
523         return self.service.getUser(user)
524 
525 
526 class FingerService(service.Service):
527 
528     __implements__ = IFingerService,
529 
530     def __init__(self, filename):
531         self.filename = filename
532         self._read()
533 
534     def _read(self):
535         self.users = {}
536         for line in file(self.filename):
537             user, status = line.split(':', 1)
538             user = user.strip()
539             status = status.strip()
540             self.users[user] = status
541         self.call = reactor.callLater(30, self._read)
542 
543     def getUser(self, user):
544         return defer.succeed(self.users.get(user, "No such user"))
545 
546     def getUsers(self):
547         return defer.succeed(self.users.keys())
548 
549 # Yet another back-end
550 
551 class LocalFingerService(service.Service):
552 
553     __implements__ = IFingerService
554 
555     def getUser(self, user):
556         user = user.strip()
557         try:
558             entry = pwd.getpwnam(user)
559         except KeyError:
560             return defer.succeed("No such user")
561         try:
562             f = file(os.path.join(entry[5],'.plan'))
563         except (IOError, OSError):
564             return defer.succeed("No such user")
565         data = f.read()
566         data = data.strip()
567         f.close()
568         return defer.succeed(data)
```

```
569
570     def getUsers(self):
571         return defer.succeed([])
572
573
574 application = service.Application('finger', uid=1, gid=1)
575 f = LocalFingerService()
576 serviceCollection = service.IServiceCollection(application)
577 internet.TCPServer(79, IFingerFactory(f)
578                    ).setServiceParent(serviceCollection)
579 internet.TCPServer(8000, server.Site(resource.IResource(f))
580                    ).setServiceParent(serviceCollection)
581 i = IIRCClientFactory(f)
582 i.nickname = 'fingerbot'
583 internet.TCPClient('irc.freenode.org', 6667, i
584                    ).setServiceParent(serviceCollection)
```

Source listing - listings/finger/finger19c.py

Not much to say except that now we can be churn out backends like crazy. Feel like doing a back-end for Advogato, for example? Dig out the XML-RPC client support Twisted has, and get to work!

没什么多余的话好说，除非我们此时此刻我们还要竭尽全力作一个令人疯狂的backend。

感觉就像是给Advogato（译者注：一个著名的开源软件站点 www.advogato.org）作一个back-end，让我举个例子？ 发现XML-RPC客户端已经支持Twisted，并能正常工作了。

return index-->TwistedTUT

Version: 1.3.0

twistedTUT05 (2004-08-09 02:11:23由dreamingk编辑)

- twistedTUT06

The Evolution of Finger: a clean web frontend -- dreamingk [2004-08-09 02:12:35]

目录

1. 介绍 Introduction

此为 Twisted 教程的第6部分 我们示范使用 Woven 模板系统来添加一个 网络前端 web frontend

This is the sixth part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

In this part, we demonstrate adding a web frontend using the Woven templating system.

2. 使用 Woven

现在我们不再编辑HTML代码了，而是使用Woven, Woven是一个历史悠久的web模板系统。 它的主要特征是不在HTML中嵌入任何代码，而且可以和deferred对象的返回结果透明的整合在一起。

Here we convert to using Woven, instead of manually constructing HTML snippets. Woven is a sophisticated web templating system. Its main features are to disallow any code inside the HTML, and transparent integration with deferred results.

Toggle line numbers

```
 1 # Do everything properly, and componentize
 2 from twisted.application import internet, service
 3 from twisted.internet import protocol, reactor, defer
 4 from twisted.protocols import basic, irc
 5 from twisted.python import components
 6 from twisted.web import resource, server, static, xmlrpc, microdom
 7 from twisted.web.woven import page, model, interfaces
 8 import cgi
 9
10 class IFingerService(components.Interface):
11
12     def getUser(self, user):
13         """Return a deferred returning a string"""
14
15     def getUsers(self):
16         """Return a deferred returning a list of strings"""
17
18 class IFingerSetterService(components.Interface):
19
20     def setUser(self, user, status):
21         """Set the user's status to something"""
22
23 def catchError(err):
```

```
24      return "Internal error in server"
25
26  class FingerProtocol(basic.LineReceiver):
27
28      def lineReceived(self, user):
29          d = self.factory.getUser(user)
30          d.addErrback(catchError)
31          def writeValue(value):
32              self.transport.write(value+'\n')
33              self.transport.loseConnection()
34          d.addCallback(writeValue)
35
36
37  class IFingerFactory(components.Interface):
38
39      def getUser(self, user):
40          """Return a deferred returning a string"""
41
42      def buildProtocol(self, addr):
43          """Return a protocol returning a string"""
44
45
46  class FingerFactoryFromService(protocol.ServerFactory):
47
48      __implements__ = IFingerFactory,
49
50      protocol = FingerProtocol
51
52      def __init__(self, service):
53          self.service = service
54
55      def getUser(self, user):
56          return self.service.getUser(user)
57
58  components.registerAdapter(FingerFactoryFromService,
59                             IFingerService,
60                             IFingerFactory)
61
62  class FingerSetterProtocol(basic.LineReceiver):
63
64      def connectionMade(self):
65          self.lines = []
66
67      def lineReceived(self, line):
68          self.lines.append(line)
69
70      def connectionLost(self, reason):
71          if len(self.lines) == 2:
72              self.factory.setUser(*self.lines)
73
```

```
 74
 75 class IFingerSetterFactory(components.Interface):
 76
 77     def setUser(self, user, status):
 78         """Return a deferred returning a string"""
 79
 80     def buildProtocol(self, addr):
 81         """Return a protocol returning a string"""
 82
 83
 84 class FingerSetterFactoryFromService(protocol.ServerFactory):
 85
 86     __implements__ = IFingerSetterFactory,
 87
 88     protocol = FingerSetterProtocol
 89
 90     def __init__(self, service):
 91         self.service = service
 92
 93     def setUser(self, user, status):
 94         self.service.setUser(user, status)
 95
 96
 97 components.registerAdapter(FingerSetterFactoryFromService,
 98                            IFingerSetterService,
 99                            IFingerSetterFactory)
100
101 class IRCReplyBot(irc.IRCClient):
102
103     def connectionMade(self):
104         self.nickname = self.factory.nickname
105         irc.IRCClient.connectionMade(self)
106
107     def privmsg(self, user, channel, msg):
108         user = user.split('!')[0]
109         if self.nickname.lower() == channel.lower():
110             d = self.factory.getUser(msg)
111             d.addErrback(catchError)
112             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
113             d.addCallback(lambda m: self.msg(user, m))
114
115
116 class IIRCClientFactory(components.Interface):
117
118     """
119     @ivar nickname
120     """
121
122     def getUser(self, user):
123         """Return a deferred returning a string"""
```

```
124
125     def buildProtocol(self, addr):
126         """Return a protocol"""
127
128
129 class IRCClientFactoryFromService(protocol.ClientFactory):
130
131     __implements__ = IIRCClientFactory,
132
133     protocol = IRCReplyBot
134     nickname = None
135
136     def __init__(self, service):
137         self.service = service
138
139     def getUser(self, user):
140         return self.service.getUser(user)
141
142 components.registerAdapter(IRCClientFactoryFromService,
143                            IFingerService,
144                            IIRCClientFactory)
145
146 class UsersModel(model.MethodModel):
147
148     def initialize(self, *args, **kwargs):
149         self.service=args[0]
150
151     def wmfactory_users(self, request):
152         return self.service.getUsers()
153
154 components.registerAdapter(UsersModel, IFingerService,
interfaces.IModel)
155
156 class UserStatusTree(page.Page):
157
158     template = """<html><head><title>Users</title></head><body>
159     <h1>Users</h1>
160     <ul model="users" view="List">
161     <li pattern="listItem"><a view="Anchor" /></li>
162     </ul></body></html>"""
163
164     def initialize(self, *args, **kwargs):
165         self.service=args[0]
166
167     def getDynamicChild(self, path, request):
168         return UserStatus(user=path, service=self.service)
169
170     def wchild_RPC2 (self, request):
171         return UserStatusXR(self.service)
172
```

```
173 components.registerAdapter(UserStatusTree, IFingerService,
resource.IResource)
174
175
176 class UserStatus(page.Page):
177
178     template='''<html><head><title view="Text"
model="user"/></head>
179     <body><h1 view="Text" model="user"/>
180     <p model="status" view="Text" />
181     </body></html>'''
182
183     def initialize(self, **kwargs):
184         self.user = kwargs['user']
185         self.service = kwargs['service']
186
187     def wmfactory_user(self, request):
188         return self.user
189
190     def wmfactory_status(self, request):
191         return self.service.getUser(self.user)
192
193
194 class UserStatusXR(xmlrpc.XMLRPC):
195
196     def __init__(self, service):
197         xmlrpc.XMLRPC.__init__(self)
198         self.service = service
199
200     def xmlrpc_getUser(self, user):
201         return self.service.getUser(user)
202
203     def xmlrpc_getUsers(self):
204         return self.service.getUsers()
205
206
207 class FingerService(service.Service):
208
209     __implements__ = IFingerService,
210
211     def __init__(self, filename):
212         self.filename = filename
213         self._read()
214
215     def _read(self):
216         self.users = {}
217         for line in file(self.filename):
218             user, status = line.split(':', 1)
219             user = user.strip()
220             status = status.strip()
```

```
221             self.users[user] = status
222         self.call = reactor.callLater(30, self._read)
223 
224     def getUser(self, user):
225         return defer.succeed(self.users.get(user, "No such user"))
226 
227     def getUsers(self):
228         return defer.succeed(self.users.keys())
229 
230 
231 application = service.Application('finger', uid=1, gid=1)
232 f = FingerService('/etc/users')
233 serviceCollection = service.IServiceCollection(application)
234 internet.TCPServer(79, IFingerFactory(f)
235                    ).setServiceParent(serviceCollection)
236 internet.TCPServer(8000, server.Site(resource.IResource(f))
237                    ).setServiceParent(serviceCollection)
238 i = IIRCClientFactory(f)
239 i.nickname = 'fingerbot'
240 internet.TCPClient('irc.freenode.org', 6667, i
241                    ).setServiceParent(serviceCollection)
242 
243 Source listing - listings/finger/finger20.py
```

return index-->TwistedTUT

twistedTUT06 (2004-08-09 02:12:35由dreamingk编辑)

- twistedTUT07

Finger演化：使用Perspective Broker对Twisted客户端进行支持

The Evolution of Finger: Twisted client support using Perspective Broker -- dreamingk [2004-08-09 02:13:28]

目录

1. 介绍(Introduction)

这是Twisted教程(Twisted入门 - Finger演化)的第七部分。

This is the seventh part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在这一部分，我们会向finger应用增加一个Perspective Broker服务，以便Twisted客户端可以访问finger服务器。

In this part, we add a Perspective Broker service to the finger application so that Twisted clients can access the finger server.

2. 使用Perspectivev Broker(Use Perspective Broker)

我们增加对perspective broker的支持，它是Twisted的本地远程对象协议。现在，Twisted客户端不再需要通过象XML-RPC化的曲折的方法来得到用户的信息。

We add support for perspective broker, Twisted's native remote object protocol. Now, Twisted clients will not have to go through XML-RPCish contortions to get information about users.

切换行号显示

```
1 # Do everything properly, and componentize
2 from twisted.application import internet, service
3 from twisted.internet import protocol, reactor, defer
4 from twisted.protocols import basic, irc
5 from twisted.python import components
6 from twisted.web import resource, server, static, xmlrpc, microdom
7 from twisted.web.woven import page, model, interfaces
8 from twisted.spread import pb
9 import cgi
10
11 class IFingerService(components.Interface):
12
13     def getUser(self, user):
14         """Return a deferred returning a string"""
15
16     def getUsers(self):
17         """Return a deferred returning a list of strings"""
18
```

```
19 class IFingerSetterService(components.Interface):
20
21     def setUser(self, user, status):
22         """Set the user's status to something"""
23
24 def catchError(err):
25     return "Internal error in server"
26
27 class FingerProtocol(basic.LineReceiver):
28
29     def lineReceived(self, user):
30         d = self.factory.getUser(user)
31         d.addErrback(catchError)
32         def writeValue(value):
33             self.transport.write(value+'\n')
34             self.transport.loseConnection()
35         d.addCallback(writeValue)
36
37
38 class IFingerFactory(components.Interface):
39
40     def getUser(self, user):
41         """Return a deferred returning a string"""
42
43     def buildProtocol(self, addr):
44         """Return a protocol returning a string"""
45
46
47 class FingerFactoryFromService(protocol.ServerFactory):
48
49     __implements__ = IFingerFactory,
50
51     protocol = FingerProtocol
52
53     def __init__(self, service):
54         self.service = service
55
56     def getUser(self, user):
57         return self.service.getUser(user)
58
59 components.registerAdapter(FingerFactoryFromService,
60                            IFingerService,
61                            IFingerFactory)
62
63 class FingerSetterProtocol(basic.LineReceiver):
64
65     def connectionMade(self):
66         self.lines = []
67
68     def lineReceived(self, line):
```

```
 69         self.lines.append(line)
 70
 71     def connectionLost(self, reason):
 72         if len(self.lines) == 2:
 73             self.factory.setUser(*self.lines)
 74
 75
 76 class IFingerSetterFactory(components.Interface):
 77
 78     def setUser(self, user, status):
 79         """Return a deferred returning a string"""
 80
 81     def buildProtocol(self, addr):
 82         """Return a protocol returning a string"""
 83
 84
 85 class FingerSetterFactoryFromService(protocol.ServerFactory):
 86
 87     __implements__ = IFingerSetterFactory,
 88
 89     protocol = FingerSetterProtocol
 90
 91     def __init__(self, service):
 92         self.service = service
 93
 94     def setUser(self, user, status):
 95         self.service.setUser(user, status)
 96
 97
 98 components.registerAdapter(FingerSetterFactoryFromService,
 99                            IFingerSetterService,
100                            IFingerSetterFactory)
101
102 class IRCReplyBot(irc.IRCClient):
103
104     def connectionMade(self):
105         self.nickname = self.factory.nickname
106         irc.IRCClient.connectionMade(self)
107
108     def privmsg(self, user, channel, msg):
109         user = user.split('!')[0]
110         if self.nickname.lower() == channel.lower():
111             d = self.factory.getUser(msg)
112             d.addErrback(catchError)
113             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
114             d.addCallback(lambda m: self.msg(user, m))
115
116
117 class IIRCClientFactory(components.Interface):
118
```

```
119     """
120     @ivar nickname
121     """
122
123     def getUser(self, user):
124         """Return a deferred returning a string"""
125
126     def buildProtocol(self, addr):
127         """Return a protocol"""
128
129
130 class IRCClientFactoryFromService(protocol.ClientFactory):
131
132     __implements__ = IIRCClientFactory,
133
134     protocol = IRCReplyBot
135     nickname = None
136
137     def __init__(self, service):
138         self.service = service
139
140     def getUser(self, user):
141         return self.service.getUser(user)
142
143 components.registerAdapter(IRCClientFactoryFromService,
144                            IFingerService,
145                            IIRCClientFactory)
146
147 class UsersModel(model.MethodModel):
148
149     def initialize(self, *args, **kwargs):
150         self.service=args[0]
151
152     def wmfactory_users(self, request):
153         return self.service.getUsers()
154
155 components.registerAdapter(UsersModel, IFingerService,
interfaces.IModel)
156
157 class UserStatusTree(page.Page):
158
159     template = """<html><head><title>Users</title></head><body>
160     <h1>Users</h1>
161     <ul model="users" view="List">
162     <li pattern="listItem"><a view="Anchor" /></li>
163     </ul></body></html>"""
164
165     def initialize(self, *args, **kwargs):
166         self.service=args[0]
167
```

```
168     def getDynamicChild(self, path, request):
169         return UserStatus(user=path, service=self.service)
170
171     def wchild_RPC2 (self, request):
172         return UserStatusXR(self.service)
173
174 components.registerAdapter(UserStatusTree, IFingerService,
resource.IResource)
175
176
177 class UserStatus(page.Page):
178
179     template='''<html><head><title view="Text"
model="user"/></head>
180     <body><h1 view="Text" model="user"/>
181     <p model="status" view="Text" />
182     </body></html>'''
183
184     def initialize(self, **kwargs):
185         self.user = kwargs['user']
186         self.service = kwargs['service']
187
188     def wmfactory_user(self, request):
189         return self.user
190
191     def wmfactory_status(self, request):
192         return self.service.getUser(self.user)
193
194
195 class UserStatusXR(xmlrpc.XMLRPC):
196
197     def __init__(self, service):
198         xmlrpc.XMLRPC.__init__(self)
199         self.service = service
200
201     def xmlrpc_getUser(self, user):
202         return self.service.getUser(user)
203
204     def xmlrpc_getUsers(self):
205         return self.service.getUsers()
206
207
208 class IPerspectiveFinger(components.Interface):
209
210     def remote_getUser(self, username):
211         """return a user's status"""
212
213     def remote_getUsers(self):
214         """return a user's status"""
215
```

```
216 class PerspectiveFingerFromService(pb.Root):
217
218     __implements__ = pb.Root.__implements__, IPerspectiveFinger
219
220     def __init__(self, service):
221         self.service = service
222
223     def remote_getUser(self, username):
224         return self.service.getUser(username)
225
226     def remote_getUsers(self):
227         return self.service.getUsers()
228
229 components.registerAdapter(PerspectiveFingerFromService,
230                            IFingerService,
231                            IPerspectiveFinger)
232
233
234 class FingerService(service.Service):
235
236     __implements__ = IFingerService,
237
238     def __init__(self, filename):
239         self.filename = filename
240         self._read()
241
242     def _read(self):
243         self.users = {}
244         for line in file(self.filename):
245             user, status = line.split(':', 1)
246             user = user.strip()
247             status = status.strip()
248             self.users[user] = status
249         self.call = reactor.callLater(30, self._read)
250
251     def getUser(self, user):
252         return defer.succeed(self.users.get(user, "No such user"))
253
254     def getUsers(self):
255         return defer.succeed(self.users.keys())
256
257
258 application = service.Application('finger', uid=1, gid=1)
259 f = FingerService('/etc/users')
260 serviceCollection = service.IServiceCollection(application)
261 internet.TCPServer(79, IFingerFactory(f)
262                    ).setServiceParent(serviceCollection)
263 internet.TCPServer(8000, server.Site(resource.IResource(f))
264                    ).setServiceParent(serviceCollection)
265 i = IIRCClientFactory(f)
```

```
266 i.nickname = 'fingerbot'
267 internet.TCPClient('irc.freenode.org', 6667, i
268                     ).setServiceParent(serviceCollection)
269 internet.TCPServer(8889, pb.PBServerFactory(IPerspectiveFinger(f))
270                     ).setServiceParent(serviceCollection)
271
272 Source listing - listings/finger/finger21.py
273 A simple client to test the perspective broker finger:
274
275 # test the PB finger on port 8889
276 # this code is essentially the same as
277 # the first example in howto/pb-usage
278
279 from twisted.spread import pb
280 from twisted.internet import reactor
281
282 def gotObject(object):
283     print "got object:", object
284     object.callRemote("getUser","moshez").addCallback(gotData)
285 # or
286 #   object.callRemote("getUsers").addCallback(gotData)
287
288 def gotData(data):
289     print 'server sent:', data
290     reactor.stop()
291
292 def gotNoObject(reason):
293     print "no object:",reason
294     reactor.stop()
295
296 factory = pb.PBClientFactory()
297 reactor.connectTCP("127.0.0.1",8889, factory)
298 factory.getRootObject().addCallbacks(gotObject,gotNoObject)
299 reactor.run()
```

Source listing - listings/finger/fingerPBclient.py

return index-->TwistedTUT

Version: 1.3.0

twistedTUT07 (2004-08-09 02:13:28由dreamingk编辑)

- twistedTUT08

Finger演化：使用单一factory支持多种协议

The Evolution of Finger: using a single factory for multiple protocols -- dreamingk [2004-08-09 02:14:44]

目录

1. 介绍(Introduction)

这是Twisted教程《Twisted入门 - Finger演化》的第八部分。

This is the eighth part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在这部分中，我们向web前端增加 HTTPS 支持，用于展示如何让单一factory在多个端口上监听(listen)。

In this part, we add HTTPS support to our web frontend, showing how to have a single factory listen on multiple ports.

2. HTTPS支持(Support HTTPS)

编写一个 HTTPS 站点我们只需要编写一个上下文(context) factory(在本例中，它会从一个特定的文件中装入证书)，接着使用twisted.application.internet.SSLServer方法。请注意一个facotry(在本例中是一个站点)可以使用多种协议在多个端口上进行监听。

All we need to do to code an HTTPS site is just write a context factory (in this case, which loads the certificate from a certain file) and then use the twisted.application.internet.SSLServer method. Note that one factory (in this case, a site) can listen on multiple ports with multiple protocols.

Toggle line numbers

```
 1 # Do everything properly, and componentize
 2 from twisted.application import internet, service
 3 from twisted.internet import protocol, reactor, defer
 4 from twisted.protocols import basic, irc
 5 from twisted.python import components
 6 from twisted.web import resource, server, static, xmlrpc, microdom
 7 from twisted.web.woven import page, model, interfaces
 8 from twisted.spread import pb
 9 from OpenSSL import SSL
10 import cgi
11
12 class IFingerService(components.Interface):
13
14     def getUser(self, user):
15         """Return a deferred returning a string"""
16
17     def getUsers(self):
```

```
18          """Return a deferred returning a list of strings"""
19
20  class IFingerSetterService(components.Interface):
21
22      def setUser(self, user, status):
23          """Set the user's status to something"""
24
25  def catchError(err):
26      return "Internal error in server"
27
28  class FingerProtocol(basic.LineReceiver):
29
30      def lineReceived(self, user):
31          d = self.factory.getUser(user)
32          d.addErrback(catchError)
33          def writeValue(value):
34              self.transport.write(value+'\n')
35              self.transport.loseConnection()
36          d.addCallback(writeValue)
37
38
39  class IFingerFactory(components.Interface):
40
41      def getUser(self, user):
42          """Return a deferred returning a string"""
43
44      def buildProtocol(self, addr):
45          """Return a protocol returning a string"""
46
47
48  class FingerFactoryFromService(protocol.ServerFactory):
49
50      __implements__ = IFingerFactory,
51
52      protocol = FingerProtocol
53
54      def __init__(self, service):
55          self.service = service
56
57      def getUser(self, user):
58          return self.service.getUser(user)
59
60  components.registerAdapter(FingerFactoryFromService,
61                             IFingerService,
62                             IFingerFactory)
63
64  class FingerSetterProtocol(basic.LineReceiver):
65
66      def connectionMade(self):
67          self.lines = []
```

```
 68
 69     def lineReceived(self, line):
 70         self.lines.append(line)
 71
 72     def connectionLost(self, reason):
 73         if len(self.lines) == 2:
 74             self.factory.setUser(*self.lines)
 75
 76
 77 class IFingerSetterFactory(components.Interface):
 78
 79     def setUser(self, user, status):
 80         """Return a deferred returning a string"""
 81
 82     def buildProtocol(self, addr):
 83         """Return a protocol returning a string"""
 84
 85
 86 class FingerSetterFactoryFromService(protocol.ServerFactory):
 87
 88     __implements__ = IFingerSetterFactory,
 89
 90     protocol = FingerSetterProtocol
 91
 92     def __init__(self, service):
 93         self.service = service
 94
 95     def setUser(self, user, status):
 96         self.service.setUser(user, status)
 97
 98
 99 components.registerAdapter(FingerSetterFactoryFromService,
100                            IFingerSetterService,
101                            IFingerSetterFactory)
102
103 class IRCReplyBot(irc.IRCClient):
104
105     def connectionMade(self):
106         self.nickname = self.factory.nickname
107         irc.IRCClient.connectionMade(self)
108
109     def privmsg(self, user, channel, msg):
110         user = user.split('!')[0]
111         if self.nickname.lower() == channel.lower():
112             d = self.factory.getUser(msg)
113             d.addErrback(catchError)
114             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
115             d.addCallback(lambda m: self.msg(user, m))
116
117
```

```
118 class IIRCClientFactory(components.Interface):
119
120     """
121     @ivar nickname
122     """
123
124     def getUser(self, user):
125         """Return a deferred returning a string"""
126
127     def buildProtocol(self, addr):
128         """Return a protocol"""
129
130
131 class IRCClientFactoryFromService(protocol.ClientFactory):
132
133     __implements__ = IIRCClientFactory,
134
135     protocol = IRCReplyBot
136     nickname = None
137
138     def __init__(self, service):
139         self.service = service
140
141     def getUser(self, user):
142         return self.service.getUser(user)
143
144 components.registerAdapter(IRCClientFactoryFromService,
145                            IFingerService,
146                            IIRCClientFactory)
147
148 class UsersModel(model.MethodModel):
149
150     def initialize(self, *args, **kwargs):
151         self.service=args[0]
152
153     def wmfactory_users(self, request):
154         return self.service.getUsers()
155
156 components.registerAdapter(UsersModel, IFingerService,
interfaces.IModel)
157
158 class UserStatusTree(page.Page):
159
160     template = """<html><head><title>Users</title></head><body>
161     <h1>Users</h1>
162     <ul model="users" view="List">
163     <li pattern="listItem"><a view="Anchor" /></li>
164     </ul></body></html>"""
165
166     def initialize(self, *args, **kwargs):
```

```
167         self.service=args[0]
168
169     def getDynamicChild(self, path, request):
170         return UserStatus(user=path, service=self.service)
171
172     def wchild_RPC2 (self, request):
173         return UserStatusXR(self.service)
174
175 components.registerAdapter(UserStatusTree, IFingerService,
resource.IResource)
176
177
178 class UserStatus(page.Page):
179
180     template='''<html><head><title view="Text"
model="user"/></head>
181     <body><h1 view="Text" model="user"/>
182     <p model="status" view="Text" />
183     </body></html>'''
184
185     def initialize(self, **kwargs):
186         self.user = kwargs['user']
187         self.service = kwargs['service']
188
189     def wmfactory_user(self, request):
190         return self.user
191
192     def wmfactory_status(self, request):
193         return self.service.getUser(self.user)
194
195
196 class UserStatusXR(xmlrpc.XMLRPC):
197
198     def __init__(self, service):
199         xmlrpc.XMLRPC.__init__(self)
200         self.service = service
201
202     def xmlrpc_getUser(self, user):
203         return self.service.getUser(user)
204
205     def xmlrpc_getUsers(self):
206         return self.service.getUsers()
207
208
209 class IPerspectiveFinger(components.Interface):
210
211     def remote_getUser(self, username):
212         """return a user's status"""
213
214     def remote_getUsers(self):
```

```
215         """return a user's status"""
216
217 class PerspectiveFingerFromService(pb.Root):
218
219     __implements__ = pb.Root.__implements__, IPerspectiveFinger
220
221     def __init__(self, service):
222         self.service = service
223
224     def remote_getUser(self, username):
225         return self.service.getUser(username)
226
227     def remote_getUsers(self):
228         return self.service.getUsers()
229
230 components.registerAdapter(PerspectiveFingerFromService,
231                            IFingerService,
232                            IPerspectiveFinger)
233
234
235 class FingerService(service.Service):
236
237     __implements__ = IFingerService,
238
239     def __init__(self, filename):
240         self.filename = filename
241         self._read()
242
243     def _read(self):
244         self.users = {}
245         for line in file(self.filename):
246             user, status = line.split(':', 1)
247             user = user.strip()
248             status = status.strip()
249             self.users[user] = status
250         self.call = reactor.callLater(30, self._read)
251
252     def getUser(self, user):
253         return defer.succeed(self.users.get(user, "No such user"))
254
255     def getUsers(self):
256         return defer.succeed(self.users.keys())
257
258
259 class ServerContextFactory:
260
261     def getContext(self):
262         """Create an SSL context.
263
264         This is a sample implementation that loads a certificate
```

```
from a file
 265         called 'server.pem'."""
 266         ctx = SSL.Context(SSL.SSLv23_METHOD)
 267         ctx.use_certificate_file('server.pem')
 268         ctx.use_privatekey_file('server.pem')
 269         return ctx
 270
 271
 272 application = service.Application('finger', uid=1, gid=1)
 273 f = FingerService('/etc/users')
 274 serviceCollection = service.IServiceCollection(application)
 275 internet.TCPServer(79, IFingerFactory(f)
 276                    ).setServiceParent(serviceCollection)
 277 site = server.Site(resource.IResource(f))
 278 internet.TCPServer(8000, site
 279                    ).setServiceParent(serviceCollection)
 280 internet.SSLServer(443, site, ServerContextFactory()
 281                    ).setServiceParent(serviceCollection)
 282 i = IIRCClientFactory(f)
 283 i.nickname = 'fingerbot'
 284 internet.TCPClient('irc.freenode.org', 6667, i
 285                    ).setServiceParent(serviceCollection)
 286 internet.TCPServer(8889, pb.PBServerFactory(IPerspectiveFinger(f))
 287                    ).setServiceParent(serviceCollection)
```

Source listing - listings/finger/finger22.py

return index-->TwistedTUT

Version: 1.3.0

twistedTUT08 (2004-08-09 02:14:44由dreamingk编辑)

- twistedTUT09

Finger演化：一个Twisted finger客户端

The Evolution of Finger: a Twisted finger client -- dreamingk [2004-08-09 02:15:36]

目录

1. 介绍(Introduction)

这是Twisted教程《Twisted入门 - Finger演化》的第九部分。

This is the ninth part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在这一部分中，我们为finger服务器开发了一个客户端：一个finger代理服务器，它可以向另一个finger服务器转发请求。

In this part, we develop a client for the finger server: a proxy finger server which forwards requests to another finger server.

2. Finger代理服务器(Finger Proxy)

用Twisted编写新的客户端很象编写新的服务器。我们实现了这样的协议，它只是收集所有的数据，并将它们发给factory。此factory维持着一个deferred，无论是连接失败或是成功，这个deferred都将被触发。当我们使用这个客户端时，首先要确保这个deferred永远不会失败，在例子中是通过产生一个消息来实现的。对只返回deferred的客户端实现一个包装器(wrapper)是通常的模式。尽管与直接使用factory相比缺少灵活性，但它要方便得多。

Writing new clients with Twisted is much like writing new servers. We implement the protocol, which just gathers up all the data, and give it to the factory. The factory keeps a deferred which is triggered if the connection either fails or succeeds. When we use the client, we first make sure the deferred will never fail, by producing a message in that case. Implementing a wrapper around client which just returns the deferred is a common pattern. While less flexible than using the factory directly, it's also more convenient.

Toggle line numbers

```
 1 # finger proxy
 2 from twisted.application import internet, service
 3 from twisted.internet import defer, protocol, reactor
 4 from twisted.protocols import basic
 5 from twisted.python import components
 6
 7
 8 def catchError(err):
 9     return "Internal error in server"
10
11 class IFingerService(components.Interface):
12
13     def getUser(self, user):
```

```
14          """Return a deferred returning a string"""
15
16      def getUsers(self):
17          """Return a deferred returning a list of strings"""
18
19
20  class IFingerFactory(components.Interface):
21
22      def getUser(self, user):
23          """Return a deferred returning a string"""
24
25      def buildProtocol(self, addr):
26          """Return a protocol returning a string"""
27
28  class FingerProtocol(basic.LineReceiver):
29
30      def lineReceived(self, user):
31          d = self.factory.getUser(user)
32          d.addErrback(catchError)
33          def writeValue(value):
34              self.transport.write(value)
35              self.transport.loseConnection()
36          d.addCallback(writeValue)
37
38
39
40  class FingerFactoryFromService(protocol.ClientFactory):
41
42      __implements__ = IFingerFactory,
43
44      protocol = FingerProtocol
45
46      def __init__(self, service):
47          self.service = service
48
49      def getUser(self, user):
50          return self.service.getUser(user)
51
52
53  components.registerAdapter(FingerFactoryFromService,
54                             IFingerService,
55                             IFingerFactory)
56
57  class FingerClient(protocol.Protocol):
58
59      def connectionMade(self):
60          self.transport.write(self.factory.user+"\r\n")
61          self.buf = []
62
63      def dataReceived(self, data):
```

```
  64         self.buf.append(data)
  65
  66     def connectionLost(self):
  67         self.factory.gotData(''.join(self.buf))
  68
  69 class FingerClientFactory(protocol.ClientFactory):
  70
  71     protocol = FingerClient
  72
  73     def __init__(self, user):
  74         self.user = user
  75         self.d = defer.Deferred()
  76
  77     def clientConnectionFailed(self, _, reason):
  78         self.d.errback(reason)
  79
  80     def gotData(self, data):
  81         self.d.callback(data)
  82
  83
  84 def finger(user, host, port=79):
  85     f = FingerClientFactory(user)
  86     reactor.connectTCP(host, port, f)
  87     return f.d
  88
  89
  90 class ProxyFingerService(service.Service):
  91     __implements__ = IFingerService
  92
  93     def getUser(self, user):
  94         try:
  95             user, host = user.split('@', 1)
  96         except:
  97             user = user.strip()
  98             host = '127.0.0.1'
  99         ret = finger(user, host)
 100         ret.addErrback(lambda _: "Could not connect to remote
host")
 101         return ret
 102
 103     def getUsers(self):
 104         return defer.succeed([])
 105
 106 application = service.Application('finger', uid=1, gid=1)
 107 f = ProxyFingerService()
 108 internet.TCPServer(7779, IFingerFactory(f)).setServiceParent(
 109     service.IServiceCollection(application))
```

Source listing - listings/finger/fingerproxy.py

return index-->TwistedTUT

Version: 1.3.0

twistedTUT09 (2004-08-09 02:15:36由dreamingk编辑)

- twistedTUT10

Finger演化：生成finger库

The Evolution of Finger: making a finger library -- dreamingk [2004-08-09 02:16:43]

目录

1. 介绍(Introduction)

这是Twisted教程《Twisted入门 - Finger演化》的第十部分。

This is the tenth part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在这个部分中，我们把装载finger服务的应用代码从定义finger服务的库代码中分离出来，将应用放在Twisted应用配置(Twisted Application Configuration)(.tac)文件中。我们同样将配置(象 HTML 模板)放入到分离的文件中。

In this part, we separate the application code that launches a finger service from the library code which defines a finger service, placing the application in a Twisted Application Configuration (.tac) file. We also move configuration (such as HTML templates) into separate files.

2. 组织(Organization)

现在这些代码，尽管设计良好并且相当模块化，但组织得不合适。在应用之上的所有东西应归于一个模块，所有的 HTML 模板应归于分离的文件。

Now this code, while quite modular and well-designed, isn't properly organized. Everything above the application belongs in a module, and the HTML templates all belong in separate files.

我们可以使用 templateFile 和 templateDirectory属性来指明，对每个页面使用什么 HTML 模板文件，并且到哪里去寻找。

We can use the templateFile and templateDirectory attributes to indicate what HTML template file to use for each Page, and where to look for it.

Toggle line numbers

```
1 # organized-finger.tac
2 # eg:  twistd -ny organized-finger.tac
3
4 import finger
5
6 from twisted.internet import protocol, reactor, defer
7 from twisted.spread import pb
8 from twisted.web import resource, server
9 from twisted.application import internet, service, strports
```

```
 10 from twisted.python import log
 11
 12 application = service.Application('finger', uid=1, gid=1)
 13 f = finger.FingerService('/etc/users')
 14 serviceCollection = service.IServiceCollection(application)
 15 internet.TCPServer(79, finger.IFingerFactory(f)
 16                     ).setServiceParent(serviceCollection)
 17
 18 site = server.Site(resource.IResource(f))
 19 internet.TCPServer(8000, site
 20                     ).setServiceParent(serviceCollection)
 21
 22 internet.SSLServer(443, site, ServerContextFactory()
 23                     ).setServiceParent(serviceCollection)
 24
 25 i = finger.IIRCClientFactory(f)
 26 i.nickname = 'fingerbot'
 27 internet.TCPClient('irc.freenode.org', 6667, i
 28                     ).setServiceParent(serviceCollection)
 29
 30 internet.TCPServer(8889,
pb.PBServerFactory(finger.IPerspectiveFinger(f))
 31                     ).setServiceParent(serviceCollection)
```

Source listing - listings/finger/organized-finger.tac Note that our program is now quite separated. We have:

Code (in the module) Configuration (file above) Presentation (templates) Content (/etc/users) Deployment (twistd) 原型并不需要这种层次的分离，因此早先的例子全都绑在了一起。然而，真正的应用需要。要感谢的是，如果我们编写的代码正确，使各部分很好的分离是容易的。

Prototypes don't need this level of separation, so our earlier examples all bunched together. However, real applications do. Thankfully, if we write our code correctly, it is easy to achieve a good separation of parts.

3. 容易的配置(Easy Configuration)

我们也可以利用makeService方法为通常的情况提供容易的配置，后面这个方法还将对建立.tap文件提供帮助：

We can also supply easy configuration for common cases with a makeService method that will also help build .tap files later:

Toggle line numbers

```
  1 # Easy configuration
  2 # makeService from finger module
  3
  4 def makeService(config):
  5     # finger on port 79
  6     s = service.MultiService()
  7     f = FingerService(config['file'])
  8     h = internet.TCPServer(79, IFingerFactory(f))
```

```
 9     h.setServiceParent(s)
10
11     # website on port 8000
12     r = resource.IResource(f)
13     r.templateDirectory = config['templates']
14     site = server.Site(r)
15     j = internet.TCPServer(8000, site)
16     j.setServiceParent(s)
17
18     # ssl on port 443
19     if config.get('ssl'):
20         k = internet.SSLServer(443, site, ServerContextFactory())
21         k.setServiceParent(s)
22
23     # irc fingerbot
24     if config.has_key('ircnick'):
25         i = IIRCClientFactory(f)
26         i.nickname = config['ircnick']
27         ircserver = config['ircserver']
28         b = internet.TCPClient(ircserver, 6667, i)
29         b.setServiceParent(s)
30
31     # Pespective Broker on port 8889
32     if config.has_key('pbport'):
33         m = internet.TCPServer(
34             int(config['pbport']),
35             pb.PBServerFactory(IPerspectiveFinger(f)))
36         m.setServiceParent(s)
37
38     return s
```

Source listing - listings/finger/finger_config.py 同时我们现在可以编写更简单的文件：

And we can write simpler files now:

Toggle line numbers

```
 1 # simple-finger.tac
 2 # eg:  twistd -ny simple-finger.tac
 3
 4 from twisted.application import service
 5
 6 import finger
 7
 8 options = { 'file': '/etc/users',
 9              'templates': '/usr/share/finger/templates',
10              'ircnick': 'fingerbot',
11              'ircserver': 'irc.freenode.net',
12              'pbport': 8889,
13              'ssl': 'ssl=0' }
14
15 ser = finger.makeService(options)
16 application = service.Application('finger', uid=1, gid=1)
```

```
17 ser.setServiceParent(service.IServiceCollection(application))
```

Source listing - listings/finger/simple-finger.tac % twisted -ny simple-finger.tac

注意：这个finger用户仍然拥有基本的能力：他可以使用makeService，或者如果有特殊需要(也许是在另一个端口上的IRC？也许我们想让non-SSL的web服务器只在本地进行监听？等等，等等)的话，他可以使用底层接口。这是一个很重要的设计原则：永远不要强加抽象层：允许抽象层的使用。

Note: the finger user still has ultimate power: he can use makeService, or he can use the lower-level interface if he has specific needs (maybe an IRC server on some other port? maybe we want the non-SSL webserver to listen only locally? etc. etc.) This is an important design principle: never force a layer of abstraction: allow usage of layers of abstractions.

3.1. 意大利面条设计理论：(The pasta theory of design:)

细面条(Spaghetti)：每一块要同其它代码块交互的代码[可以用GOTO、函数、对象来实现]
宽面条(Lasagna)：拥有仔细设计过层的代码。每一个层是指在理论上独立。然后低级的层通常不易使用，而高级的层要依赖低级的层。 面卷(Ravioli)：每一部分代码对自身都是有用的。在各部分之间有一个细的接口层[调料]。每一部分都可以用于其它地方。 ...但有时候，用户只想要订面卷，所以在它之上有一个容易定义的谷粒(coarse-grain)抽象层更加有用。

Spaghetti: each piece of code interacts with every other piece of code [can be implemented with GOTO, functions, objects] Lasagna: code has carefully designed layers. Each layer is, in theory independent. However low-level layers usually cannot be used easily, and high-level layers depend on low-level layers. Ravioli: each part of the code is useful by itself. There is a thin layer of interfaces between various parts [the sauce]. Each part can be usefully be used elsewhere.
...but sometimes, the user just wants to order Ravioli, so one coarse-grain easily definable layer of abstraction on top of it all can be useful.

return index-->TwistedTUT

Version: 1.3.0

twistedTUT10 (2004-08-09 02:16:43由dreamingk编辑)

- twistedTUT11

Finger演化：finger服务的配置和打包

The Evolution of Finger: configuration and packaging of the finger service -- dreamingk [2004-08-09 02:17:25]

目录

1. 介绍(Introduction)

这是Twisted教程(Twisted入门 - Finger演化)的第十一部分。

This is the eleventh part of the Twisted tutorial Twisted from Scratch, or The Evolution of Finger.

在这个部分中，我们将使配置一个finger服务器对非程序员来说更容易，并且展示如何以 .deb 和 RPM 包格式对其进行打包。

In this part, we make it easier for non-programmers to configure a finger server, and show how to package it in the .deb and RPM package formats.

2. 插件(Plugins)

迄今为止，用户多多少少需要是一个程序员才能够配置东西。也许我们甚至连这个都可以消除？将旧的代码移到 finger/init.py中，并且...

So far, the user had to be somewhat of a programmer to be able to configure stuff. Maybe we can eliminate even that? Move old code to finger/init.py and...

以下是finger模块的全部代码：

Full source code for finger module here:

切换行号显示

```
 1 # finger.py module
 2
 3 from twisted.application import internet, service, strports
 4 from twisted.internet import protocol, reactor, defer
 5 from twisted.protocols import basic, irc
 6 from twisted.python import components
 7 from twisted.web import resource, server, static, xmlrpc, microdom
 8 from twisted.web.woven import page, model, interfaces
 9 from twisted.spread import pb
10 from OpenSSL import SSL
11 import cgi
12
13 class IFingerService(components.Interface):
14
15     def getUser(self, user):
16         """Return a deferred returning a string"""
17
18     def getUsers(self):
19         """Return a deferred returning a list of strings"""
```

```
 20
 21 class IFingerSetterService(components.Interface):
 22
 23     def setUser(self, user, status):
 24         """Set the user's status to something"""
 25
 26 def catchError(err):
 27     return "Internal error in server"
 28
 29 class FingerProtocol(basic.LineReceiver):
 30
 31     def lineReceived(self, user):
 32         d = self.factory.getUser(user)
 33         d.addErrback(catchError)
 34         def writeValue(value):
 35             self.transport.write(value+'\n')
 36             self.transport.loseConnection()
 37         d.addCallback(writeValue)
 38
 39
 40 class IFingerFactory(components.Interface):
 41
 42     def getUser(self, user):
 43         """Return a deferred returning a string"""
 44
 45     def buildProtocol(self, addr):
 46         """Return a protocol returning a string"""
 47
 48
 49 class FingerFactoryFromService(protocol.ServerFactory):
 50     __implements__ = protocol.ServerFactory.__implements__,
IFingerFactory
 51
 52     protocol = FingerProtocol
 53
 54     def __init__(self, service):
 55         self.service = service
 56
 57     def getUser(self, user):
 58         return self.service.getUser(user)
 59
 60 components.registerAdapter(FingerFactoryFromService,
 61                            IFingerService,
 62                            IFingerFactory)
 63
 64 class FingerSetterProtocol(basic.LineReceiver):
 65
 66     def connectionMade(self):
 67         self.lines = []
 68
```

```
 69     def lineReceived(self, line):
 70         self.lines.append(line)
 71 
 72     def connectionLost(self, reason):
 73         if len(self.lines) == 2:
 74             self.factory.setUser(*self.lines)
 75 
 76 
 77 class IFingerSetterFactory(components.Interface):
 78 
 79     def setUser(self, user, status):
 80         """Return a deferred returning a string"""
 81 
 82     def buildProtocol(self, addr):
 83         """Return a protocol returning a string"""
 84 
 85 
 86 class FingerSetterFactoryFromService(protocol.ServerFactory):
 87 
 88     __implements__ =
protocol.ServerFactory.__implements__,IFingerSetterFactory
 89 
 90     protocol = FingerSetterProtocol
 91 
 92     def __init__(self, service):
 93         self.service = service
 94 
 95     def setUser(self, user, status):
 96         self.service.setUser(user, status)
 97 
 98 
 99 components.registerAdapter(FingerSetterFactoryFromService,
100                            IFingerSetterService,
101                            IFingerSetterFactory)
102 
103 class IRCReplyBot(irc.IRCClient):
104 
105     def connectionMade(self):
106         self.nickname = self.factory.nickname
107         irc.IRCClient.connectionMade(self)
108 
109     def privmsg(self, user, channel, msg):
110         user = user.split('!')[0]
111         if self.nickname.lower() == channel.lower():
112             d = self.factory.getUser(msg)
113             d.addErrback(catchError)
114             d.addCallback(lambda m: "Status of %s: %s" % (msg, m))
115             d.addCallback(lambda m: self.msg(user, m))
116 
117 
```

```
118 class IIRCClientFactory(components.Interface):
119
120     """
121     @ivar nickname
122     """
123
124     def getUser(self, user):
125         """Return a deferred returning a string"""
126
127     def buildProtocol(self, addr):
128         """Return a protocol"""
129
130
131 class IRCClientFactoryFromService(protocol.ClientFactory):
132
133     __implements__ = protocol.ClientFactory.__implements__,
IIRCClientFactory
134
135     protocol = IRCReplyBot
136     nickname = None
137
138     def __init__(self, service):
139         self.service = service
140
141     def getUser(self, user):
142         return self.service.getUser(user)
143
144 components.registerAdapter(IRCClientFactoryFromService,
145                            IFingerService,
146                            IIRCClientFactory)
147
148 class UsersModel(model.MethodModel):
149
150     def initialize(self, *args, **kwargs):
151         self.service=args[0]
152
153     def wmfactory_users(self, request):
154         return self.service.getUsers()
155
156 components.registerAdapter(UsersModel, IFingerService,
interfaces.IModel)
157
158 class UserStatusTree(page.Page):
159
160     template = """<html><head><title>Users</title></head><body>
161     <h1>Users</h1>
162     <ul model="users" view="List">
163     <li pattern="listItem"><a view="Anchor" /></li>
164     </ul></body></html>"""
165
```

```
166     def initialize(self, *args, **kwargs):
167         self.service=args[0]
168 
169     def getDynamicChild(self, path, request):
170         return UserStatus(user=path, service=self.service)
171 
172     def wchild_RPC2 (self, request):
173         return UserStatusXR(self.service)
174 
175 components.registerAdapter(UserStatusTree, IFingerService,
resource.IResource)
176 
177 
178 class UserStatus(page.Page):
179 
180     template='''<html><head><title view="Text"
model="user"/></head>
181     <body><h1 view="Text" model="user"/>
182     <p model="status" view="Text" />
183     </body></html>'''
184 
185     def initialize(self, **kwargs):
186         self.user = kwargs['user']
187         self.service = kwargs['service']
188 
189     def wmfactory_user(self, request):
190         return self.user
191 
192     def wmfactory_status(self, request):
193         return self.service.getUser(self.user)
194 
195 
196 class UserStatusXR(xmlrpc.XMLRPC):
197 
198     def __init__(self, service):
199         xmlrpc.XMLRPC.__init__(self)
200         self.service = service
201 
202     def xmlrpc_getUser(self, user):
203         return self.service.getUser(user)
204 
205     def xmlrpc_getUsers(self):
206         return self.service.getUsers()
207 
208 
209 class IPerspectiveFinger(components.Interface):
210 
211     def remote_getUser(self, username):
212         """return a user's status"""
213 
```

```
214     def remote_getUsers(self):
215         """return a user's status"""
216
217 class PerspectiveFingerFromService(pb.Root):
218
219     __implements__ = pb.Root.__implements__, IPerspectiveFinger
220
221     def __init__(self, service):
222         self.service = service
223
224     def remote_getUser(self, username):
225         return self.service.getUser(username)
226
227     def remote_getUsers(self):
228         return self.service.getUsers()
229
230 components.registerAdapter(PerspectiveFingerFromService,
231                            IFingerService,
232                            IPerspectiveFinger)
233
234
235 class FingerService(service.Service):
236
237     __implements__ = service.Service.__implements__,
IFingerService
238
239     def __init__(self, filename):
240         self.filename = filename
241         self._read()
242
243     def _read(self):
244         self.users = {}
245         for line in file(self.filename):
246             user, status = line.split(':', 1)
247             user = user.strip()
248             status = status.strip()
249             self.users[user] = status
250         self.call = reactor.callLater(30, self._read)
251
252     def getUser(self, user):
253         return defer.succeed(self.users.get(user, "No such user"))
254
255     def getUsers(self):
256         return defer.succeed(self.users.keys())
257
258
259 class ServerContextFactory:
260
261     def getContext(self):
262         """Create an SSL context.
```

```
263
264         This is a sample implementation that loads a certificate
from a file
265         called 'server.pem'."""
266         ctx = SSL.Context(SSL.SSLv23_METHOD)
267         ctx.use_certificate_file('server.pem')
268         ctx.use_privatekey_file('server.pem')
269         return ctx
270
271
272
273
274 # Easy configuration
275
276 def makeService(config):
277     # finger on port 79
278     s = service.MultiService()
279     f = FingerService(config['file'])
280     h = internet.TCPServer(79, IFingerFactory(f))
281     h.setServiceParent(s)
282
283
284     # website on port 8000
285     r = resource.IResource(f)
286     r.templateDirectory = config['templates']
287     site = server.Site(r)
288     j = internet.TCPServer(8000, site)
289     j.setServiceParent(s)
290
291     # ssl on port 443
292 #    if config.get('ssl'):
293 #        k = internet.SSLServer(443, site, ServerContextFactory())
294 #        k.setServiceParent(s)
295
296     # irc fingerbot
297     if config.has_key('ircnick'):
298         i = IIRCClientFactory(f)
299         i.nickname = config['ircnick']
300         ircserver = config['ircserver']
301         b = internet.TCPClient(ircserver, 6667, i)
302         b.setServiceParent(s)
303
304     # Pespective Broker on port 8889
305     if config.has_key('pbport'):
306         m = internet.TCPServer(
307             int(config['pbport']),
308             pb.PBServerFactory(IPerspectiveFinger(f)))
309         m.setServiceParent(s)
310
311     return s
```

finger模块 - listings/finger/finger/finger.py

finger module - listings/finger/finger/finger.py

切换行号显示

```
 1 # finger/tap.py
 2 from twisted.application import internet, service
 3 from twisted.internet import interfaces
 4 from twisted.python import usage
 5 import finger
 6
 7 class Options(usage.Options):
 8
 9     optParameters = [
10         ['file', 'f', '/etc/users'],
11         ['templates', 't', '/usr/share/finger/templates'],
12         ['ircnick', 'n', 'fingerbot'],
13         ['ircserver', None, 'irc.freenode.net'],
14         ['pbport', 'p', 8889],
15         ]
16
17     optFlags = [['ssl', 's']]
18
19 def makeService(config):
20     return finger.makeService(config)
```

finger/tap.py - listings/finger/finger/tap.py 并且完全登记：

And register it all:

切换行号显示

```
1 #finger/plugins.tml
2 register('finger', 'finger.tap',
3  description='Build a finger server tap',
4  type='tap', tapname='finger')
```

finger/plugins.tml - listings/finger/finger/plugins.tml 现在，下面可以工作了

And now, the following works

```
% mktap finger --file=/etc/users --ircnick=fingerbot
% sudo twistd -nf finger.tap
```

3. OS集成(OS Integration)

如果我们已经将finger包安装在PYTHONPATH下(例如，我们将它安装在site-packages目录下)，我们可以容易地实现集成：

If we already have the finger package installed in PYTHONPATH (e.g. we added it to site-packages), we can achieve easy integration:

4. Debian

```
% tap2deb --unsigned -m "Foo <foo@example.com>" --type=python
finger.tac
% sudo dpkg -i .build/*.deb
```

4.1. Red Hat / Mandrake

```
% tap2rpm --type=python finger.tac #[maybe other options needed]
% sudo rpm -i .build/*.rpm
```

对于给定的文件将正确地登记 tap/tac，init.d脚本，等等。

Will properly register the tap/tac, init.d scripts, etc. for the given file.

如果在你心爱的OS上不能用：我们接受补丁!

If it doesn't work on your favorite OS: patches accepted!

return index-->TwistedTUT

Version: 1.3.0

twistedTUT11 (2004-08-09 02:17:25由dreamingk编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LlNEL1

异步编程 (Asynchronous Programming) -- dreamingk [2004-08-09 01:50:15]

1. 介绍 (Introduction)

目录

要编写网络程序，有很多不同的方式。主流的有：

1. 在不同的进程中处理每个连接；
2. 在不同的线程中处理每个连接（脚注1）；
3. 在一个线程中使用非阻塞系统调用来处理所有连接。

There are many ways to write network programs. The main ones are:

1. Handle each connection in a separate process
2. Handle each connection in a separate thread (Footnode 1)
3. Use non-blocking system calls to handle all connections in one thread.

当在一个线程里处理多个连接时，调度不是由操作系统完成，它成为了应用程序的职责。当每个连接准备好读或写时，通常是通过调用一个注册函数来实现的——就是大家通常说的异步、事件驱动或基于回调的编程。

When dealing with many connections in one thread, the scheduling is the responsibility of the application, not the operating system, and is usually implemented by calling a registered function when each connection is ready to for reading or writing -- commonly known as asynchronous, event-driven or callback-based programming.

即使有很高的抽象，编写多线程的程序也需要很多技巧，Python 的“全局解释器锁”（Global Interpreter Lock）限制了潜在的性能提高。创建（fork）Python 子进程的方法也有许多缺陷，比如：Python 的引用计数在“写时拷贝”（copy-on-write）的方式下不能很好的工作，在处理共享状态方面也有问题。因此，最终事件驱动框架是最好的选择。不过，这样做的一个好处是可以让其他事件驱动型的框架（framework）来接管主循环。这样，服务器和客户端代码本质上是相同的——这在实际上形成了一个P2P（peer-to-peer）的形式。

Multi-threaded programming is tricky, even with high level abstractions, and Python's Global Interpreter Lock
limits the potential performance gain. Forking Python processes also has many disadvantages, such as Python's reference counting not playing well with copy-on-write and problems with shared state. Consequently, it was felt the best option was an event-driven framework. A benefit of such an approach is that by letting other event-driven frameworks take over the main loop, server and client code are essentially the same -- making peer-to-peer a reality.

另一方面，事件驱动型的编程方法也包含需要技巧的方面。因为每次回调都必须尽快的结束，这使得它不可能在一个函数级别（function-local）的变量中保存持久的状态。此外，象递归之类的一些编程的技术不可能使用——例如，递归消解了协议处理程序成为循环下降的语法分析程序的可能。由于经常要编写各种状态机，事件驱动编程被公认为使用起来很费劲。但 Twisted 是基于这样的想法建立的：只要使用正确的库，事件驱动方法比多进程方法更简单。

However, event-driven programming still contains some tricky aspects. As each callback must be finished as soon as possible, it is not possible to keep persistent state in function-local variables. In addition, some programming techniques, such as recursion, are

impossible to use -- for example, this rules out protocol handlers being recursive-descent parsers. Event-driven programming has a reputation of being hard to use due to the frequent need to write state machines. Twisted was built with the assumption that with the right library, event-driven programming is easier than multi-threaded programming.

注意：如果你需要，Twisted 也允许使用线程──这通常是用来与同步化的、继承来的代码进行接口的。详细信息参见“如何使用线程”。

Note that Twisted still allows the use of threads if you really need them, usually to interface with synchronous legacy code. See Using Threads for details.

2. 异步设计的观点（Async Design Issues）

在 Python 里，源代码经常被拆分到一个一般类里，它调用那些由子类实现的可重载方法。在它和与它类似的案例里，考虑合适的实现（implementation）是很重要的。如果这个实现完成一个动作（action）有可能需要很长的时间（不管是由于网络还是CPU的原因），那就应该把该方法（method）设计成异步的。通常，这意味着要把它改成基于回调的方法。在 Twisted 里，通常是让它返回一个“Deferred”。

In Python, code is often divided into a generic class calling overridable methods which subclasses implement. In that, and similar, cases, it is important to think about likely implementations. If it is conceivable that an implementation might perform an action which takes a long time (either because of network or CPU issues), then one should design that method to be asynchronous. In general, this means to transform the method to be callback based. In Twisted, it usually means returning a Deferred.

既然因为每个方法都要尽快返回，非易变（non-volatile）状态不能保存在局部变量里，所以它通常被保存到实例(instance)一级的变量中。在可能会有递归发生的案例里，这些状态通常必须保存在栈（stack）结构里──由 Python 里列表类型（list）实现，通过 append 和 pop 方法、手工控制访问。因为那些状态机频繁的转换（get non-trivial），最好是把它们分成不同层次，每个状态机只做一件事情──将事件从一个抽象层转化为下一个更高的抽象层。这样，代码更清晰，也更容易调试。

Since non-volatile state cannot be kept in local variables, because each method must return quickly, it is usually kept in instance variables. In cases where recursion would have been tempting, it is usually necessary to keep stacks manually, using Python's list and the .append and .pop method. Because those state machines frequently get non-trivial, it is better to layer them such that each one state machine does one thing -- converting events from one level of abstraction to the next higher level of abstraction. This allows the code to be clearer, as well as easier to debug.

3. 使用反映（reflection）（Using Reflection）

使用回调类型的编程方法的一个重要后果就是：需要给那些小片的代码命名。虽然看起来这是只是个小问题，只要正确的使用就可以看到它的好处。如果使用严格、一致的命名，许多分析程序中象 if/else 形式或者很长的 case 语句这样的代码就可以避免了。例如，SMTP 客户端代码有一个实例级别的变量，用来标明它正在做的动作。当从服务器接收到一个响应，它就可以直接呼叫 "do_%s_%s" % (self.state, responseCode) 方法。这样，完全避免了注册回调函数或者一个很大的 if/else 判断链。另外，子类也可以很容易的重载它或者改变接收到一些响应时的动作，却不需要新增厚重的代码。这个 SMTP 客户端实现在 twisted/protocols/smtp.py 文件中可以找到。

One consequence of using the callback style of programming is the need to name small chunks of code. While this may seem like a trivial issue, used correctly it can prove to be

an advantage. If strictly consistent naming is used, then much of the common code in parsers of the form of if/else rules or long cases can be avoided. For example, the SMTP client code has an instance variable which signifies what it is trying to do. When receiving a response from the server, it just calls the method "do_%s_%s" % (self.state, responseCode). This eliminates the requirement for registering the callback or adding to large if/else chains. In addition, subclasses can easily override or change the actions when receiving some responses, with no additional harness code. The SMTP client implementation can be found in twisted/protocols/smtp.py.

4. 脚注 (Footnotes)

1. 这种方法有一些其他的变化，比如：用一个有限尺寸的线程池来为所有连接提供服务──这其实是这种方法的一种优化。
2. There are variations on this method, such as a limited-size pool of threads servicing all connections, which are essentially just optimizations of the same idea.

<< 返回PyTwisted/LowLevelNetworkingEventLoop

PyTwisted/LowLevelNetworkingEventLoop/LlNEL1 (2004-08-09 01:50:15由dreamingk编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL2

Reactor 预览(Reactor Overview) -- dreamingk [2004-08-09 01:53:17]

1. Reacotr 基础(Reactor Basics)

目录

This HOWTO introduces the Twisted reactor, describes the basics of the reactor and links to

这份HOWTO文档介绍了Twisted中的reactor,描述了reactor的基础和相关内容

1.1. Reactor基础(Reactor Basics)

The reactor is the core of the event loop within Twisted -- the loop which drives applications using Twisted. The reactor provides basic interfaces to a number of services, including network communications, threading, and event dispatching.

Reactor是Twisted事件循环的核心-使用Twiested的应用程序就是靠这个事件循环来驱动.Reactor提供了一些基本的服务接口,包括网络通讯,线程和事件分发.

For information about using the reactor and the Twisted event loop, see:

关于如果如何使用reactor和Twisted事件循环,参见

the event dispatching howtos: Scheduling and Using Deferreds; the communication howtos: TCP servers, TCP clients, UDP networking and Using processes; and Using threads.

事件分发 howtos: "调度"和"使用"Deferreds" 通讯howtos: "TCP服务器","TCP客户端", "UDP网络"和"使用进程"; 以及 "使用线程"

There are multiple implementations of the reactor, each modified to provide better support for specialized features over the default implementation. More information about these and how to use a particular implementation is available via Choosing a Reactor.

Reactor有很多各不同的实现,每个实现都在缺省实现的基础上对某些特性提供了更好的支持.更多的信息和关于如何使用这些特性参见"选择Reactor"

Twisted applications can use the interfaces in twisted.application.service to configure and run the application instead of using boilerplate reactor code. See Using Application for an introduction to Application.

Twisted应用程序可以使用twisted.application.service中提供的接口来配置和运行应用程序,而不是使用像reactor的示例代码那样去运行.更多介绍请参考"使用应用"

1.2. 使用reactor对象(Using the reactor object)

You can get to the reactor object using the following code:

你可以通过如下代码得到reactor:

```
from twisted.internet import reactor
```

The reactor usually implements a set of interfaces, but depending on the chosen reactor and the platform, some of the interfaces may not be implemented:

Reactor通常实现了以下一组接口,但是根据选择的不同reactor和运行平台,一些接口可能没有实现:

- IReactorCore: Core (required) functionality.
- IReactorFDSet: Use FileDescriptor objects.
- IReactorProcess: Process management. Read the Using Processes document for more information.
- IReactorSSL: SSL networking support.
- IReactorTCP: TCP networking support. More information can be found in the Writing Servers and Writing Clients documents.
- IReactorThreads: Threading use and management. More information can be found within Threading In Twisted.
- IReactorTime: Scheduling interface. More information can be found within Scheduling Tasks.
- IReactorUDP: UDP networking support. More information can be found within UDP Networking.
- IReactorUNIX: UNIX socket support.
- IReactorCore: 必需的核心功能
- IReactorFDSet: 使用文件描述符对象
- IReactorProcess: 进程管理,详细信息参见"使用进程"
- IReactorSSL: SSL支持
- IReactorTCP: TCP网络支持,详细信息参见"编写服务器"和"编写客户端"
- IReactorThreads: 线程使用和管理. 详细信息参见"Twisted中的线程"
- IReactorTime: 调度接口.更多信息参见"调度任务"
- IReactorUDP: UDP支持.更多信息参见"UDP网络"
- IReactorUNIX: UNIX socket支持.

翻译 -- Jerry Marx 7/30/2004

返回 PyTwisted/LowLevelNetworkingEventLoop

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL2 (2004-08-09 01:53:17由dreamingk编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL3

编写TCP服务器(Writing a TCP Server) -- dreamingk [2004-08-09 01:56:45]

1. 概要(Overview)

目录

Twisted is a framework designed to be very flexible and let you write powerful servers. The cost of this flexibility is a few layers in the way to writing your server.

Twisted被设计为一个非常灵活的框架,可以使用它编写强大的服务器.这种灵活性的代价是当你编写服务器的时候需要分好几层.

This document describes the Protocol layer, where you implement protocol parsing and handling. If you are implementing an application then you should read this document second, after first reading the top level overview of how to begin writing your Twisted application, in Writing Plug-Ins for Twisted. This document is only relevant to TCP, SSL and Unix socket servers, there is a separate document for UDP.

本文档描述协议层,在这里定义协议的解析和处理.如果你正在编写一个应用的话这应该是你读的第二份文档,第一份是关于如何开始编写你的Twsited服务器,参见"编写Twised插件".本文档关注TCP,SSL和Unix socket服务器,关于UDP有另一份文档.

Your protocol handling class will usually subclass twisted.internet.protocol.Protocol. Most protocol handlers inherit either from this class or from one of its convenience children. An instance of the protocol class might be instantiated per-connection, on demand, and might go away when the connection is finished. This means that persistent configuration is not saved in the Protocol.

通常你的协议处理类应该是twisted.internet.protocol.Protocol的子类.大部分协议处理函数或者继承自这个类,或者继承自它的一个更适合的子类.依据需求,协议类的实例可能为每个连接实例化,可能在连接结束的时候被析构.这就意味着不要在协议类中包含需要持久配置.

The persistent configuration is kept in a Factory class, which usually inherits from twisted.internet.protocol.Factory. The default factory class just instantiates each Protocol, and then sets on it an attribute called factory which points to itself. This lets every Protocol access, and possibly modify, the persistent configuration.

持久配置应该包含在工厂类中,工厂类通常继承自twisted.internet.protocol.Factory.缺省的工厂类只是实例化协议类,并且设置协议类的属性factory指向自己.这样就可以让每个协议类可以访问甚至修改持久配置.

It is usually useful to be able to offer the same service on multiple ports or network addresses. This is why the Factory does not listen to connections, and in fact does not know anything about the network. See twisted.internet.interfaces.IReactorTCP.listenTCP, and the other IReactor*.listen* APIs for more information.

在不同的网络地址和端口提供同样的服务通常是很有用的.所以工厂并不去监听连接,实事上工厂对于网络一无所知.请阅读twisted.internet.interfaces.IReactorTCP.listenTCP和IReactor*.listen*得到更多的信息.

This document will explain each step of the way.

本文档将解释上述的每个步骤.

2. 协议(Protocols)

As mentioned above, this, along with auxiliary classes and functions, is where most of the code is. A Twisted protocol handles data in an asynchronous manner. What this means is that the protocol never waits for an event, but rather responds to events as they arrive from the network.

就像上面提到的,在大多数代码中,协议和其它辅助类和函数协同工作.Twisted协议以异步方式处理数据.这意味着协议从来不会去等待一个事件,而是当事件从网络到来的时候响应它.

Here is a simple example:

这里有个简单的例子

切换行号显示

```
1     from twisted.internet.protocol import Protocol
2
3     class Echo(Protocol):
4
5         def dataReceived(self,data):
6             self.transport.write(data)
```

This is one of the simplest protocols. It simply writes back whatever is written to it, and does not respond to all events. Here is an example of a Protocol responding to another event:

这是一个最简单的协议,它只是把收到的内容写回去,并没有响应所有的事件.下一个例子演示响应其它事件

切换行号显示

```
1     from twisted.internet.protocol import Protocol
2
3     class QOTD(Protocol):
4
5         def connectionMade(self):
6             self.transport.write("An apple a day keeps the doctor
away\r\n")
7             self.transport.loseConnection()
```

This protocol responds to the initial connection with a well known quote, and then terminates the connection.

这个协议在连接建立后发送一条格言,然后断开连接.

The connectionMade event is usually where set up of the connection object happens, as well as any initial greetings (as in the QOTD protocol above, which is actually based on RFC 865). The connectionLost event is where tearing down of any connection-specific objects is done. Here is an example:

connectionMake事件通常在建立连接的时候触发,发送一条欢迎信息(就像上面的QOTD协议,该协议基于RFC865).conectionLost事件在连接断开后触发.看这个例子

切换行号显示

```
1     from twisted.internet.protocol import Protocol
2
3     class Echo(Protocol):
4
```

```
    5          def connectionMake(self):
    6              self.factory.numProtocols = self.factory.numProtocols +
1
    7              self.factory.numProtocols > 100:
    8                  self.transport.write("Too many connections, try
later")
    9                  self.transport.loseConnection()
   10
   11          def connectionLost(self,reason):
   12              self.factory.numProtocols = self.factory.numProtocols
- 1
   13
   14          def dataReceived(self,data):
   15              self.transport.write(data)
```

Here connectionMade and connectionLost cooperate to keep a count of the active protocols in the factory. connectionMade immediately closes the connection if there are too many active protocols.

这里connnectionMade事件和connectionLost事件协同维护工厂中的活动计数.如果活动计数太高,connectionMake就立即断开连接.

2.1. 使用协议(Using the Protocol)

In this section, I will explain how to test your protocol easily. (In order to see how you should write a production-grade Twisted server, though, you should read the Writing Plug-Ins for Twisted HOWTO as well).

在这一节中,我将解释如何使你的协议便于测试.(尽管如果你想写出产品级的Twisted服务器,"编写Twisted插件"才是你最应该读的).

Here is code that will run the QOTD server discussed earlier

下面的代码运行前面讨论过的QOTD服务器

切换行号显示

```
    1      from twisted.internet.protocol import Protocol, Factory
    2      from twisted.internet import reactor
    3
    4      class QOTD(Protocol):
    5
    6          def connectionMake(self):
    7              self.transport.write("An apple a day keeps the doctor
away\r\n")
    8              self.transport.loseConnection()
    9
   10      # 魔术发生在下面一行:
   11      factory = Factory()
   12      factory.protocol = QOTD
   13
   14      # 8007是你想服务运行的端口,其实大于1024都可以
   15      reactor.listenTCP(8007,factory)
   16      reactor.run()
```

Don't worry about the last 6 magic lines -- you will understand what they do later in the document.
别为不明白具有魔力的最后六行担心--在文档的后面部分你会明白他们在干什么的.

2.2. 助手协议(Helper Protocols)

Many protocols build upon similar lower-level abstraction. The most popular in internet protocols is being line-based. Lines are usually terminated with a CR-LF combinations.
很多协议的底层数据提取都是一样的.大部分流行的因特网协议都是基于行的.行通常由"回车换行"这个组合来结束

However, quite a few protocols are mixed - they have line-based sections and then raw data sections. Examples include HTTP/1.1 and the Freenet protocol.
也有非常少的协议是混合的--他们有基于行的部分也有原始数据部分.这样的例子包括HTTP/1.1和Freenet协议

For those cases, there is the LineReceiver protocol. This protocol dispatches to two different event handlers - lineReceived and rawDataReceived. By default, only lineReceived will be called, once for each line. However, if setRawMode is called, the protocol will call rawDataReceived until setLineMode is called again.

LineReceiver
协议适用于这些情况.该协议分发两种不同的事件处理器 - lineReceived和rawDataReceived.缺省情况下每行只有lineReceived会被调用.然而,如果setRawMode被调用后,协议就会触发rawDataReceived,直到setLineMode被调用.

Here is an example for a simple use of the line receiver:
这里有个行接受的简单示例:

切换行号显示

```
1     from twisted.protocols.basic import LineReceiver
2
3     class Answer(LineReceiver):
4         answers = {'How are you?': 'Fine', None: "I don't know
what you mean"}
5
6         def lineReceived(self, line):
7             if self.answers.has_key(line):
8                 self.sendLine(self.answers[line])
9             else:
10                self.sendLine(self.answers[None])
```

Note that the delimiter is not part of the line.
注意分隔符('\r\n')不是行的一部分

Several other, less popular, helpers exist, such as a netstring based protocol and a prefixed-message-length protocol.
还有一些不是特别流行的助手协议,比方说netstring based协议和prefixed-message-length协议

2.3. 状态机(State Machines)

Many Twisted protocol handlers need to write a state machine to record the state they are

at. Here are some pieces of advice which help to write state machines:

很多Twisted协议处理器需要实现状态机来纪录它目前所处的状态.这里有一些关于状态机的建议:

- Don't write big state machines. Prefer to write a state machine which deals with one level of abstraction at a time.
- Use Python's dynamicity to create open ended state machines. See, for example, the code for the SMTP client.
- Don't mix application-specific code with Protocol handling code. When the protocol handler has to make an application-specific call, keep it as a method call.
- 不要写大的状态机.只要处理一层抽象更好
- 使用Pyhton的动态特征创建最终的状态机.SMTP客户端是个很好的例子
- 不要把应用特性代码和协议处理代码混起来.当协议必须要调用应用特性的时候,调用另一个模块.

3. 工厂(Factories)

As mentioned before, usually the class twisted.internet.protocol.Factory works, and there is no need to subclass it. However, sometimes there can be factory-specific configuration of the protocols, or other considerations. In those cases, there is a need to subclass Factory.

如同前面提到的,通常twisted.internet.protocol.Factory就可以工作了,而不需要派生一个子类.然而,有时协议需要一些工厂维护的持续信息或者其它情况,我们就需要一个派生自Factory的子类

For a factory which simply instantiates instances of a specific protocol class, simply instantiate Factory, and sets its protocol attribute:

对于一个工厂来说,就是实例化协议类,实例化工厂,设置工厂的protocol属性

切换行号显示

```
1     from twisted.internet.protocol import Factory
2     from twisted.protocols.wire import Echo
3     myFactory = Factory()
4     myFactory.protocol = Echo
```

If there is a need to easily construct factories for a specific configuration, a factory function is often useful:

如果需要方便的为特殊配置构造工厂,一个工厂函数很有用:

切换行号显示

```
1     from twisted.internet.protocol import Factory, Protocl
2
3     class QOTD(Protocol):
4
5         def connectionMake(self):
6             self.transport.write(elf.factory.quote+'\r\n')
7             self.transport.loseConnection()
8
9     def makeQOTDFactory(quote=None):
10        factory = Factory()
11        factory.protocol = QOTD
12        factory.quote = quote or 'An apple a day keeps the doctor
away'
13        return factory
```

A Factory has two methods to perform application-specific building up and tearing down (since a Factory is frequently persisted, it is often not appropriate to do them in init or del, and would frequently be too early or too late).

工厂有两个执行特殊应用的方法building up和tearing down(由于工程是持续的,调用init或者del都是不合适的)<<<这两句没太看明白>>>

Here is an example of a factory which allows its Protocols to write to a special log-file:

下面这个例子演示了允许协议写一个Log文件.

切换行号显示

```
1     from twisted.internet.protocol import Factory
2     from twisted.protocols.basic import LineReceiver
3
4     class LoggingProtocol(LineReceiver):
5
6         def lineReceived(self, line):
7             self.factory.fp.write(line + '\n')
8
9     class LogfileFactory(Factory):
10        protocol = LoggingProtocol
11
12        def __init__(self, fileName):
13             self.file = fileName
14
15        def startFactory(self):
16             self.fp = open(self.file, 'a')
17
18        def stopFactory(self):
19             self.fp.close()
```

3.1. 组织在一起(Putting it All Together)

So, you know what factories are, and want to run the QOTD with configurable quote server, do you? No problems, here is an example.

现在,你知道了什么是工厂,下来想要运行QOTD为一个可配置的引用服务器,是吧?没问题,示例如下:

切换行号显示

```
1     from twisted.internet.protocol import Factory, Protocl
2     from twisted.internet import reactor
3
4     class QOTD(Protocol):
5         def connectionMade(self):
6             self.transport.write(self.factory.quote + '\r\n')
7             self.transport.loseConnection()
8
9     class QOTDFactory(Factory):
10        protocol = QOTD
11
12        def __init__(self, quote=None):
```

```
  13             self.quote = quote or 'An apple a day keeps the doctor
away'
  14
  15     reactor.listenTCP(8007, QOTDFactory("configurable quote"))
  16     reactor.run()
```

The only lines you might not understand are the last two.

只有最后两行你还可能不理解

listenTCP is the method which connects a Factory to the network. It uses the reactor interface, which lets many different loops handle the networking code, without modifying end-user code, like this. As mentioned above, if you want to write your code to be a production-grade Twisted server, and not a mere 20-line hack, you will want to use the Application object.

listenTCP是一个连接到网络的方法.它使用reactor接口,reactor接口使我们不用修改代码就可以完成不同的处理循环.就像前面提及的,如果你像写一个产品级的Twisted服务器而不是只有20行代码,你可能想使用"应用程序对象"

翻译 -- Jerry Marx. 7/30/2004

version 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL3 (2004-08-09 01:56:45由dreamingk编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL4

编写客户端(Writing Clients) -- JerryMarx [2004-08-03 03:33:44]

1. 概要

目录

(Overview)

Twisted is a framework designed to be very flexible, and let you write powerful clients. The cost of this flexibility is a few layers in the way to writing your client. This document covers creating clients that can be used for TCP, SSL and Unix sockets, UDP is covered in a different document.

Twisted被设计为一个非常灵活的框架,你可以用它写强大的客户端.这种灵活性的代价是编写客户端的时候需要分好几层来实现.本文档叙述如何写一个可用于TCP,SSL和Unix socket的客户端,关于UDP另有文档叙述.

At the base, the place where you actually implement the protocol parsing and handling, is the Protocol class. This class will usually be decended from twisted.internet.protocol.Protocol. Most protocol handlers inherit either from this class or from one of its convenience children. An instance of the protocol class will be instantiated when you connect to the server, and will go away when the connection is finished. This means that persistent configuration is not saved in the Protocol.

作为基础,协议类实现了协议解析和处理,通常继承自twisted.internet.protocol.Protocol.大部分协议处理类或者直接继承自这个类或者继承自这个类的某个合适的子类.协议类的的实例在连接到服务器的时候被实例化,在断开连接的时候被销毁.这就意味着不能使用协议类来做持续持有配置.

The persistent configuration is kept in a Factory class, which usually inherits from twisted.internet.protocol.ClientFactory. The default factory class just instantiate the Protocol, and then sets on it an attribute called factory which points to itself. This let the Protocol access, and possibly modify, the persistent configuration.

持续持有的地方应该在工厂类,它通常继承自twisted.internet.protocol.ClientFactory.缺省的工厂类的行为只是实例化Protocol类,并把Protocol类的factory属性设置为指向自己,这样协议类就可以访问甚至修改持续持有的配置.

2. 协议(Protocol)

As mentioned above, this, and auxiliary classes and functions, is where most of the code is. A Twisted protocol handles data in an asynchronous manner. What this means is that the protocol never waits for an event, but rather responds to events as they arrive from the network.

就像上面所说的,这就是基本框架.Twisted协议以异步方式处理数据.就是说它从来不等待一个事件,而是在事件从网络到来的时候响应事件.

Here is a simple example:

这里有一个例子:

切换行号显示

```
1 from twisted.internet.protocol import Protocol
2 from sys import stdout
3 class Echo(Protocol):
4
5     def dataReceived(self, data):
6         stdout.write(data)
```

This is one of the simplest protocols. It simply writes to standard output whatever it reads from the connection. There are many events it does not respond to. Here is an example of a Protocol responding to another event.

这是最简单的协议了,它只是把任何从网络收到的数据写到标准输出.有很多事件它并没有处理,下一个例子演示如何响应其它事件:

切换行号显示

```
1 from twisted.internet.protocol import Protocol
2 class WelcomeMessage(Protocol):
3
4     def connectionMade(self):
5         self.transport.write("Hello server, I am the client!\r\n")
6         self.transport.loseConnection()
```

This protocol connects to the server, sends it a welcome message, and then terminates the connection.

这个协议连接到服务器,发送一条欢迎信息,然后断开.

The connectionMade event is usually where set up of the Protocol object happens, as well as any initial greetings (as in the WelcomeMessage protocol above). Any tearing down of Protocol-specific objects is done in connectionLost.

事件connectionMade通常在协议对象被创建的时候触发,伴随着初始化问候(就像上面的WelcomMessage协议).在connectionLost事件中做销毁特定协议对象的处理.

3. 客户工厂(ClientFactory)

We use reactor.connect* and a ClientFactory. The ClientFactory is in charge of creating the Protocol, and also receives events relating to the connection state. This allows it to do things like reconnect on the event of a connection error. Here is an example of a simple ClientFactory
that uses the Echo protocol (above) and also prints what state the connection is in.

我们使用reactor.connect*和一个ClientFactory.ClientFactory的职责是创建协议,它也会收到和连接状态相关的事件,这样就可以在连接出错的时候重新建立连接.这里有一个例子,它使用了上面的Echo协议,它也可以打印连接的状态.

切换行号显示

```
1 from twisted.internet.protocol import Protocol, ClientFactory
2 from sys import stdout
3
4 class Echo(Protocol):
5
6     def dataReceived(self, data):
```

```
 7         stdout.write(data)
 8
 9 class EchoClientFactory(ClientFactory):
10
11     def startedConnecting(self, connector):
12         print 'Started to connect.'
13
14     def buildProtocol(self, addr):
15         print 'Connected.'
16         return Echo()
17
18     def clientConnectionLost(self, connector, reason):
19         print 'Lost connection.  Reason:', reason
20
21     def clientConnectionFailed(self, connector, reason):
22         print 'Connection failed. Reason:', reason
```

To connect this EchoClientFactory to a server, you could use this code:

可以使用一下代码连接到服务器

切换行号显示

```
1 from twisted.internet import reactor
2 reactor.connectTCP(host, port, EchoClientFactory())
3 reactor.run()
```

Note that clientConnectionFailed is called when a connection could not be established, and that clientConnectionLost is called when a connection was made and then disconnected.

注意当连接不能建立的时候clientConnectionFailed会被调用,而当连接建立之后断开则clientConnectionLost会被调用.

3.1. 再次连接(Reconnection)

Many times, the connection of a client will be lost unintentionally due to network errors. One way to reconnect after a disconnection would be to call connector.connect() when the connection is lost:

很多时候,客户端会因为网络错误而意外失去连接.再连接的一种方式是在失去连接后调用connector.connect().

切换行号显示

```
1 from twisted.internet.protocol import ClientFactory
2 class EchoClientFactory(ClientFactory):
3
4     def clientConnectionLost(self, connector, reason):
5         connector.connect()
```

The connector passed as the first argument is the interface between a connection and a protocol. When the connection fails and the factory receives the clientConnectionLost event, the factory can call connector.connect() to start the connection over again from scratch.

connector是connection和protocol之间接口的第一个参数.当连接失败,工厂收到clientConnectionLost事件,就调用connector.connect()重新开始连接.

However, most programs that want this functionality should implement ReconnectingClientFactory instead, which tries to reconnect if a connection is lost or fails, and which exponentially delays repeated reconnect attempts.

而然,大部分的程序员向实现一种ReconnectingClientFactory机制,当连接断开或者失败的时候,它以指数函数关系延迟重复的连接意图.

Here is the Echo protocol implemented with a ReconnectingClientFactory:

下面这个例子使用ReconnectingClientFactory实现Echo协议:

切换行号显示

```
   1 from twisted.internet.protocol import Protocol,
ReconnectingClientFactory
   2 from sys import stdout
   3
   4 class Echo(Protocol):
   5
   6     def dataReceived(self, data):
   7         stdout.write(data)
   8
   9 class EchoClientFactory(ReconnectingClientFactory):
  10
  11     def startedConnecting(self, connector):
  12         print 'Started to connect.'
  13
  14     def buildProtocol(self, addr):
  15         print 'Connected.'
  16         print 'Resetting reconnection delay'
  17         self.resetDelay()
  18         return Echo()
  19
  20     def clientConnectionLost(self, connector, reason):
  21         print 'Lost connection.  Reason:', reason
  22         ReconnectingClientFactory.clientConnectionLost(self,
connector, reason)
  23
  24     def clientConnectionFailed(self, connector, reason):
  25         print 'Connection failed. Reason:', reason
  26         ReconnectingClientFactory.clientConnectionFailed(self,
  27                 connector, reason)
```

4. 一个稍微复杂的例子:irc日志机器人(A Higher-Level Example: ircLogBot)

4.1. irc日志机器人概述(Overview of ircLogBot)

The clients so far have been fairly simple. A more complicated example comes with Twisted in the doc/examples directory.

客户端一直以来都使用简单清楚的例子,这次我们来看Twisted中doc/examples附带的一个稍微复杂的例子:

切换行号显示

```
"""An example IRC log bot - logs a channel's events to a file.

If someone says the bot's name in the channel followed by a ':',
e.g.

  <foo> logbot: hello!

the bot will reply:

  <logbot> foo: I am a log bot

Run this script with two arguments, the channel name the bot
should
connect to, and file to log to, e.g.:

  $ python ircLogBot.py test test.log

will log channel #test to the file 'test.log'.
"""


# twisted imports
from twisted.protocols import irc
from twisted.internet import reactor, protocol
from twisted.python import log

# system imports
import time, sys


class MessageLogger:
    """
    An independant logger class (because separation of application
    and protocol logic is a good thing).
    """
    def __init__(self, file):
        self.file = file

    def log(self, message):
        """Write a message to the file."""
        timestamp = time.strftime("[%H:%M:%S]",
time.localtime(time.time()))
        self.file.write('%s %s\n' % (timestamp, message))
        self.file.flush()

    def close(self):
        self.file.close()


class LogBot(irc.IRCClient):
```

```
 49      """A logging IRC bot."""
 50
 51      def __init__(self):
 52          self.nickname = "twistedbot"
 53
 54      def connectionMade(self):
 55          irc.IRCClient.connectionMade(self)
 56          self.logger = MessageLogger(open(self.factory.filename,
"a"))
 57          self.logger.log("[connected at %s]" %
 58                          time.asctime(time.localtime(time.time())))
 59
 60      def connectionLost(self, reason):
 61          irc.IRCClient.connectionLost(self, reason)
 62          self.logger.log("[disconnected at %s]" %
 63                          time.asctime(time.localtime(time.time())))
 64          self.logger.close()
 65
 66
 67      # callbacks for events
 68
 69      def signedOn(self):
 70          """Called when bot has succesfully signed on to server."""
 71          self.join(self.factory.channel)
 72
 73      def joined(self, channel):
 74          """This will get called when the bot joins the channel."""
 75          self.logger.log("[I have joined %s]" % channel)
 76
 77      def privmsg(self, user, channel, msg):
 78          """This will get called when the bot receives a
message."""
 79          user = user.split('!', 1)[0]
 80          self.logger.log("<%s> %s" % (user, msg))
 81          if msg.startswith("%s:" % self.nickname):
 82              # someone is talking to me, lets respond:
 83              msg = "%s: I am a log bot" % user
 84              self.say(channel, msg)
 85              self.logger.log("<%s> %s" % (self.nickname, msg))
 86
 87      def action(self, user, channel, msg):
 88          """This will get called when the bot sees someone do an
action."""
 89          user = user.split('!', 1)[0]
 90          self.logger.log("* %s %s" % (user, msg))
 91
 92      # irc callbacks
 93
 94      def irc_NICK(self, prefix, params):
 95          """Called when an IRC user changes their nickname."""
```

```
  96         old_nick = prefix.split('!')[0]
  97         new_nick = params[0]
  98         self.logger.log("%s is now known as %s" % (old_nick,
new_nick))
  99
 100
 101 class LogBotFactory(protocol.ClientFactory):
 102     """A factory for LogBots.
 103
 104     A new protocol instance will be created each time we connect
to the server.
 105     """
 106
 107     # the class of the protocol to build when new connection is
made
 108     protocol = LogBot
 109
 110     def __init__(self, channel, filename):
 111         self.channel = channel
 112         self.filename = filename
 113
 114     def clientConnectionLost(self, connector, reason):
 115         """If we get disconnected, reconnect to server."""
 116         connector.connect()
 117
 118     def clientConnectionFailed(self, connector, reason):
 119         print "connection failed:", reason
 120         reactor.stop()
 121
 122
 123 if __name__ == '__main__':
 124     # initialize logging
 125     log.startLogging(sys.stdout)
 126
 127     # create factory protocol and application
 128     f = LogBotFactory(sys.argv[1], sys.argv[2])
 129
 130     # connect factory to this host and port
 131     reactor.connectTCP("irc.freenode.net", 6667, f)
 132
 133     # run bot
 134     reactor.run()
```

Source listing - ../examples/ircLogBot.py ircLogBot.py connects to an IRC server, joins a channel, and logs all traffic on it to a file. It demonstrates some of the connection-level logic of reconnecting on a lost connection, as well as storing persistent data in the Factory.

ircLogBot.py 连接到IRC服务器,加入一个频道,将各种信息纪录到文件.它演示了连接-再次连接的逻辑,演示了工厂中的持续数据.

4.2. 在工厂中保存持续数据(Persistent Data in the Factory)

Since the Protocol instance is recreated each time the connection is made, the client needs some way to keep track of data that should be persisted. In the case of the logging bot, it needs to know which channel it is logging, and where to log it to.

由于Protocol实例会随着每个连接的建立而实例化,而客户端又需要保存一些持续数据.在上面这个例子里面,需要保存的数据就是机器人要加入哪个频道和要把日子写在什么地方.

切换行号显示

```
  1 from twisted.internet import protocol
  2 from twisted.protocols import irc
  3
  4 class LogBot(irc.IRCClient):
  5
  6     def connectionMade(self):
  7         irc.IRCClient.connectionMade(self)
  8         self.logger = MessageLogger(open(self.factory.filename,
"a"))
  9         self.logger.log("[connected at %s]" %
 10                         time.asctime(time.localtime(time.time())))
 11
 12     def signedOn(self):
 13         self.join(self.factory.channel)
 14
 15
 16 class LogBotFactory(protocol.ClientFactory):
 17
 18     protocol = LogBot
 19
 20     def __init__(self, channel, filename):
 21         self.channel = channel
 22         self.filename = filename
```

When the protocol is created, it gets a reference to the factory as self.factory. It can then access attributes of the factory in its logic. In the case of LogBot, it opens the file and connects to the channel stored in the factory.

在protocol被创建的时候,它得到了对工厂的引用.这样它就可以访问工厂的属性.在上面这个例子里面,它打开的文件和要加入的频道就存储在工厂.

翻译 -- Jerry Marx.

(目录)Index

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL4 (2004-08-18 17:51:17由218编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL5

UDP网络(UDP Networking) -- JerryMarx [2004-08-02 09:44:02]

目录

1. 概要(Overview)

Unlike TCP, UDP has no notion of connections. A UDP socket can receive datagrams from any server on the network, and send datagrams to any host on the network. In addition, datagrams may arrive in any order, never arrive at all, or be duplicated in transit.

和TCP不同,UDP没有连接的概念.UDP socket可以接受任何网络服务器发来的数据报,也可以向任何网络地址发送数据报.而且,数据报可能以任何顺序到达,可能永远也不会到达,也可能重复到达.

Since there are no multiple connections, we only use a single object, a protocol, for each UDP socket. We then use the reactor to connect this protocol to a UDP transport, using the twisted.internet.interfaces.IReactorUDP reactor API.

由于不存在多重连接,我们所有的UDP socket使用一个单独的对象,一个protocol.我们使用reactor把这个协议连接到UDP传输,使用twisted.internet.interface.IReactorUDP中提供的reactor API.

2. 数据报协议(DatagramProtocol)

At the base, the place where you actually implement the protocol parsing and handling, is the DatagramProtocol
class. This class will usually be decended from twisted.internet.protocol.DatagramProtocol. Most protocol handlers inherit either from this class or from one of its convenience children. The DatagramProtocol class receives datagrams, and can send them out over the network. Received datagrams include the address they were sent from, and when sending datagrams the address to send to must be specified.

数据报协议(DatagramProtocol)类是基础中的基础,这个类实现了协议的解析和处理.这个类通常位于twisted.internet.protocol.DatagramProtocol.大部分协议或者直接继承自这个类或者继承自这个类的某个合适的子类.数据报协议类接收数据报,或者发送数据报.接收到的数据报包含数据报的发送地址,发送的数据报必须指定特定的目标地址.

Here is a simple example:

这里有个简单的例子:

切换行号显示

```
1 from twisted.internet.protocol import DatagramProtocol
2 from twisted.internet import reactor
3
4 class Echo(DatagramProtocol):
5
6     def datagramReceived(self, data, (host, port)):
7         print "received %r from %s:%d" % (data, host, port)
8         self.transport.write(data, (host, port))
9
10 reactor.listenUDP(9999, Echo())
```

```
11 reactor.run()
```

As you can see, the protocol is registed with the reactor. This means it may be persisted if it's added to an application, and thus it has twisted.internet.protocol.DatagramProtocol.startProtocol and twisted.internet.protocol.DatagramProtocol.stopProtocol methods that will get called when the protocol is connected and disconnected from a UDP socket.

如你所见,protocol和reactor一起注册.这意味着如果它被添加到一个应用程序,它将持续存在.当协议被连接或者断开的时候它的方法 twisted.internet.protocol.DatagramProtocol.startProtocol和 twisted.internet.protocol.DatagramProtocol.stopProtocol会被调用.

The protocol's transport attribute will implement the twisted.internet.interfaces.IUDPTransport interface. Notice that the host argument should be an IP, not a hostname. If you only have the hostname use reactor.resolve() to resolve the address (see twisted.internet.interfaces.IReactorCore.resolve).

Protocol的transport属性实现了twisted.internet.interfaces.IUDPTransport接口.注意参数host应该是一个IP地址,而不是域名(机器名).如果你只有域名(机器名)可用,可以调用reactor.resolve()得到IP地址(参见twisted.internet.interfaces.IReactorCore.resolve).

3. 连接UDP(Connected UDP)

A connected UDP socket is slighly different from a standard one - it can only send and receive datagrams to/from a single address, but this does not in any way imply a connection. Datagrams may still arrive in any order, and the port on the other side may have no one listening. The benefit of the connected UDP socket is that it is faster.

连接的UDP同标准UDP只有一些细微差别 -- 它只能和一个确定的地址通信(发送/接收数据报),但是这并不意味着我们有一个连接.数据报仍然可能以任何顺序到达,目标地址的端口甚至可能都没有打开.连接UDP的好处只是在于它更快一些.

Unlike a regular UDP protocol, we do not need to specify where to send datagrams to, and are not told where they came from since they can only come from address the socket is 'connected' to.

和标准UDP协议不同的是,我们不需要在发送数据报的时候指定地址,也不需要从数据报中得到来源地址的信息,因为我们只能和我们"连接"到的地址通信.

切换行号显示

```
1 from twisted.internet.protocol import DatagramProtocol
2 from twisted.internet import reactor
3
4 class Helloer(DatagramProtocol):
5
6     def startProtocol(self):
7         d = self.transport.connect("192.168.1.1", 1234)
8         d.addCallback(self._cbConnected)
9
10    def _cbConnected(self, (host, port)):
11        print "we can only send to %s now" % str((host, port))
12        self.transport.write("hello") # no need for address
13
14    def datagramReceived(self, data, (host, port)):
15        print "received %r from %s:%d" % (data, host, port)
16
```

```
17 # 0 means any port, we don't care in this case
18 reactor.listenUDP(0, Helloer())
19 reactor.run()
```

Note that connect(), like write() will only accept IP addresses, not unresolved domain names. To obtain the IP of a domain name use reactor.resolve(), e.g.:

注意connect()和write()一样只接受IP地址作为参数,不能使用域名.调用reactor.resolve()可以从域名得到IP地址.例如:

切换行号显示

```
1 from twisted.internet import reactor
2
3 def gotIP(ip):
4         print "IP of 'example.com' is", ip
5
6 reactor.resolve('example.com').addCallback(gotIP)
```

Connecting to a new address after a previous connection, or making a connected port unconnected are not currently supported, but will likely be supported in the future.

在连接前一地址后连接到一个新地址,或者断开已经连接的端口目前还不支持,但是会在将来支持.

(目录)Index

翻译 -- Jerry Marx 8/2/2004

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL5 (2005-01-11 01:03:33由limodou编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL6

使用进程(Using Processes)

-- Jerry Marx 于 [2004-08-18 18:28:47] 最后编辑

1. 概述

目录

(Overview)

Along with connection to servers across the internet, Twisted also connects to local processes with much the same API. the API is described in more detail in the documentation of:

就像通过网络连接到服务器上一样,Twisted也可以使用相同的API连接到本地进程.这些API的详细文档如下:

- "twisted.internet.interfaces.IReactorProcess"
- "twisted.internet.interfaces.IProcessTransport"
- "twisted.internet.protocol.ProcessProtocol"

2. 运行另一个进程(Running Another Processes)

Processes are run through the reactor, using reactor.spawnProcess(). Pipes are created to the child process, and added to the reactor core so that the application will not block while sending data into or pulling data out of the new process. reactor.spawnProcess() requires two arguments, processProtocol and executable, and optionally takes six more: arguments, environment, path, userID, groupID, and usePTY.

进程通过反应器(reactor)的成员函数reactor.spawnProcess()运行. 同时创建一个通向子进程的管道(pipe),由于管道是被加到反应器核心的,所以应用程序不会被阻塞. 在向新进程发送数据或者从新进程接收数据的时候, reactor.spawnProcess()需要两个参数, processProtocol和excutable,还有其它留个可选参数: arguments,environment,path, userID,groupID和usePTY.

切换行号显示

```
   1 from twisted.internet import reactor
   2
   3 mypp = MyProcessProtocol()
   4 reactor.spawnProcess(processProtocol, executable, args=[program,
arg1, arg2],
   5                       env={'HOME': os.environ['HOME']}, path,
```

```
6                       uid, gid, usePTY, childFDs)
```

3. 写一个进程协议(Writing a ProcessProtocol)

The ProcessProtocol
you pass to spawnProcess is your interaction with the process. It has a very similar signature to a regular Protocol, but it has several extra methods to deal with events specific to a process. In our example, we will interface with 'wc' to create a word count of user-given text. First, we'll start by importing the required modules, and writing the initialization for our ProcessProtocol.

传给spawnProcess()的ProcessProtocol是和进程的交互方法.它的签名式和通常的Protocol很相似,只是多了一些专为处理进程事件的方法.在我们的例子中,我们使用"wc"接口来创建一个可以计算用户输入文本的单词数量的应用.首先,我们从包含模块和初始化我们的ProcessProtocol开始.

切换行号显示

```
1 from twisted.internet import protocol
2 class WCProcessProtocol(protocol.ProcessProtocol):
3
4     def __init__(self, text):
5         self.text = text
```

When the ProcessProtocol
is connected to the protocol, it has the connectionMade method called. In our protocol, we will write our text to the standard input of our process and then close standard input, to the let the process know we are done writing to it.

当ProcessProtocol
连接到protocol的时候,connectionMade()方法会被调用.在我们的协议中,我们把文字写到进程的标准输入然后关闭标准输入通知进程我们已经写完了.

切换行号显示

```
1 def connectionMade(self):
2         self.transport.write(self.text)
3         self.transport.closeStdin()
```

在这里进程收到了数据,该是我们读结果的时候了.这里没有使用dataReceived()来接收数据,而是使用了从标注输出接收数据的outReceived().这样就可以和标准错误输出的数据区别开来.

切换行号显示

```
1 def outReceived(self, data):
2         fieldLength = len(data) / 3
3         lines = int(data[:fieldLength])
4         words = int(data[fieldLength:fieldLength*2])
5         chars = int(data[fieldLength*2:])
6         self.transport.loseConnection()
7         self.receiveCounts(lines, words, chars)
```

Now, the process has parsed the output, and ended the connection to the process. Then it sends the results on to the final method, receiveCounts. This is for users of the class to override, so as to do other things with the data. For our demonstration, we will just print

the results.

现在进程已经解析了标准输出的数据,然后断开了连接,然后把数据发送给最后一个函数:receiveCounts().这个函数应该被用户类重写来实现他们自己对数据的处理.在我们的例子里面,只是把它们打印出来

切换行号显示

```
1 def receiveCounts(self, lines, words, chars):
2         print 'Received counts from wc.'
3         print 'Lines:', lines
4         print 'Words:', words
5         print 'Characters:', chars
```

We're done! To use our WCProcessProtocol, we create an instance, and pass it to spawnProcess.

完成了!创建一个WCProcessProtocol实例,传给spawnProcess()给可以用了.

切换行号显示

```
1 from twisted.internet import reactor
2 wcProcess = WCProcessProtocol("accessing protocols through Twisted
is fun!\n")
3 reactor.spawnProcess(wcProcess, 'wc', ['wc'])
4 reactor.run()
```

4. 进程协议可以做什么(Things that can happen to your ProcessProtocol)

These are the methods that you can usefully override in your subclass of ProcessProtocol:

通常,派生自ProcessProtocol类的子类应该改写以下这些函数

- .connectionMade: This is called when the program is started, and makes a good place to write data into the stdin pipe (using self.transport.write()).
- .outReceived(data): This is called with data that was received from the process' stdout pipe. Pipes tend to provide data in larger chunks than sockets (one kilobyte is a common buffer size), so you may not experience the random dribs and drabs behavior typical of network sockets, but regardless you should be prepared to deal if you don't get all your data in a single call. To do it properly, outReceived ought to simply accumulate the data and put off doing anything with it until the process has finished.
- .errReceived(data): This is called with data from the process' stderr pipe. It behaves just like outReceived.
- .inConnectionLost: This is called when the reactor notices that the process' stdin pipe has closed. Programs don't typically close their own stdin, so this will probably get called when your ProcessProtocol
  has shut down the write side with self.transport.loseConnection().
- .outConnectionLost: This is called when the program closes its stdout pipe. This usually happens when the program terminates.
- .errConnectionLost: Same as outConnectionLost, but for stderr instead of stdout.
- .processEnded(status): This is called when the child process has been reaped, and receives information about the process' exit status. The status is passed in the form of a Failure instance, created with a .value that either holds a Failure object if the process terminated normally (it died of natural causes instead of receiving a signal, and if the exit code was 0), or a ProcessTerminatedobject (with an .exitCode

attribute) if something went wrong. This scheme may seem a bit weird, but I trust that it proves useful when dealing with exceptions that occur in asynchronous code.

This will always be called afterinConnectionLost, outConnectionLost, and errConnectionLost are called.

- .connectionMade: 程序开始的时候会调用这这个函数,这里是写数据到标准输入管道的合适时机(使用self.transport.write()).
- .outReceived(data): 进程在标准输出管道收到数据的时候会调用这个函数.管道趋向于处理比套接字数据量大很多的数据(千字节的buffer是很普通的).也许你没有从套接字获取零星数据的经验,但是如果不知道自己在干什么就不要一次就从管道中取出所有数据.合适的方式是,ouReceived只是简单的收集数据,在程序结束之前处理它们.
- .errReceived(data): 进程在标准错误管道收到数据的时候会调用这个函数.它的行为和.outReceived(data)类似
- .inConnnectionLost: 当reactor发现进程的标准输入管道被关闭的时候这个函数会被调用,通常一个程序不会关闭它自己的标准输入,因此这个函数一般会在你的ProcessProtocol调用self.transport.loseConnection()关闭写入端的时候被调用.
- .outConnectionLost: 这个函数通常在程序关闭他自己的标准输出管道的时候被调用.这通常发生在程序结束的时候.
- .errConnectinoLost: 标准错误输出管道,其它同.outConnectionLost.
- .processEnded(status): 当子进程完程的时候会被调用,收到关于进程退出的信息.状态以Failure
  实例的方式传回.它的成员.value有两种情况,如果进程是正常结束(自然结束而不是因为收到一个信号,并且推出码是0), .value就是个ProcessDone对象,如果有什么地方出错了, .value就是个ProcessTerminated对象(有一个.exitCode属性).这样的安排也许看起来有些怪异,不过我相信对于处理异步代码的异常是非常有用的.

  > 这个函数总会在 inConnectionLost, outConnectionLost 和 errConnectionLost 之后被调用.

The base-class definitions of these functions are all no-ops. This will result in all stdout and stderr being thrown away. Note that it is important for data you don't care about to be thrown away: if the pipe were not read, the child process would eventually block as it tried to write to a full pipe.

基类中这些函数的定义都是空操作(no-ops).这样会导致所有的标准输出和标准错误都被丢弃.注意丢弃你不关心的数据是很重要的: 如果关掉不能读,子进程最终会被阻塞的,因为它可能会试图写一个已经满的管道.

5. 可以对进程做什么(Things you can do from your ProcessProtocol)

The following are the basic ways to control the child process:

下面是控制子进程的基本方法:

- self.transport.write(data): Stuff some data in the stdin pipe. Note that this write method will queue any data that can't be written immediately. Writing will resume in the future when the pipe becomes writable again.
- self.transport.closeStdin: Close the stdin pipe. Programs which act as filters (reading from stdin, modifying the data, writing to stdout) usually take this as a sign that they should finish their job and terminate. For these programs, it is important to close stdin when you're done with it, otherwise the child process will never quit.
- self.transport.closeStdout: Not usually called, since you're putting the process into a

state where any attempt to write to stdout will cause a SIGPIPE error. This isn't a nice thing to do to the poor process.
- self.transport.closeStderr: Not usually called, same reason as closeStdout.
- self.transport.loseConnection: Close all three pipes.
- os.kill(self.transport.pid, signal.SIGKILL): Kill the child process. This will eventually result in processEnded being called.
- self.transport.write(data): 往标准输入管道里面塞数据. 这个write操作如果发现不能立即写入的话就会把数据放入队列,等待管道再次可用的时候再写进去
- self.transport.closeStdin: 关闭标准输入管道. 扮演过滤器(从标准输入读取数据,修改数据,写到标准输出)角色的程序通常使用这种方式来表示它完成了它的所有工作,要结束了.对于这些程序来说这样做很重要,不然子进程永远不会推出.
- self.transport.closeStdout: 通常不会被调用,因为这样做会导致进程进入一个如果尝试向标准输出写数据就会引发SIGPIPE错误的状态.这对于可怜的进程来说可不是什么好事.
- self.transport.closeStderr: 通常不会被调用,原因如上.
- self.transport.loseConnection: 关闭所有的三种管道
- os.kill(self.transport.pid, signal.SIGKILL): 杀掉子进程,这么做会导致processEnded()会被调用.

6. Verbose例子(Verbose Example)

Here is an example that is rather verbose about exactly when all the methods are called. It writes a number of lines into the wc program and then parses the output.

下面的例子详细的演示了这些函数是如何被调用的.它写了多行数据到wc程序然后分析它的输出.

切换行号显示

```
#! /usr/bin/python

from twisted.internet import protocol
from twisted.internet import reactor
import re

class MyPP(protocol.ProcessProtocol):
    def __init__(self, verses):
        self.verses = verses
        self.data = ""
    def connectionMade(self):
        print "connectionMade!"
        for i in range(self.verses):
            self.transport.write("Aleph-null bottles of beer on the wall,\n" +
                                 "Aleph-null bottles of beer,\n" +
                                 "Take one down and pass it around,\n" +
                                 "Aleph-null bottles of beer on the wall.\n")
        self.transport.closeStdin() # tell them we're done
    def outReceived(self, data):
        print "outReceived! with %d bytes!" % len(data)
        self.data = self.data + data
```

```
  22     def errReceived(self, data):
  23         print "errReceived! with %d bytes!" % len(data)
  24     def inConnectionLost(self):
  25         print "inConnectionLost! stdin is closed! (we probably did
it)"
  26     def outConnectionLost(self):
  27         print "outConnectionLost! The child closed their stdout!"
  28         # now is the time to examine what they wrote
  29         #print "I saw them write:", self.data
  30         (dummy, lines, words, chars, file) = re.split(r'\s+',
self.data)
  31         print "I saw %s lines" % lines
  32     def errConnectionLost(self):
  33         print "errConnectionLost! The child closed their stderr."
  34     def processEnded(self, status_object):
  35         print "processEnded, status %d" %
status_object.value.exitCode
  36         print "quitting"
  37         reactor.stop()
  38
  39 pp = MyPP(10)
  40 reactor.spawnProcess(pp, "wc", ["wc"], {})
  41 reactor.run()
```

The exact output of this program depends upon the relative timing of some un-synchronized events. In particular, the program may observe the child process close its stderr pipe before or after it reads data from the stdout pipe. One possible transcript would look like this:

这个程序的准确输出依赖于一些异步事件的发生时机.比方说,子进程可能在它从标准输出读数据之前或之后关闭标准错误管道.一个可能的输出如下:

```
% ./process.py
connectionMade!
inConnectionLost! stdin is closed! (we probably did it)
errConnectionLost! The child closed their stderr.
outReceived! with 24 bytes!
outConnectionLost! The child closed their stdout!
I saw 40 lines
processEnded, status 0
quitting
Main loop terminated.
%
```

7. 容易一些(Doing ti the Easy Way)

Frequently, one just needs a simple way to get all the output from a program. In the blocking world, you might use commands.getoutput from the standard library, but using that in an event-driven program will cause everything else to stall until the command finishes. (in addition, the SIGCHLD handler used by that function does not play well with Twisted's own signal handling). For these cases, the twisted.internet.utils.getProcessOutput function can be used. Here is a simple example:

经常需要通过简单的方法来得到一个程序的所有输出(指标准输出).在阻塞的世界里,你可以使用标准库中的commands.getoutput,但是如果在事件驱动的程序里面这么做的话就会导致所有的事情都停下来等待命令结束(此外,这个函数使用的SIGCHLD信号处理也不能和Twisted的信号处理方法很好的共处).这时应该使用"twisted.internet.utils.getProcessOutput",下面是个简单的例子:

切换行号显示

```
 1 from twisted.internet import protocol, utils, reactor
 2 from twisted.python import failure
 3 from cStringIO import StringIO
 4
 5 class FortuneQuoter(protocol.Protocol):
 6
 7     fortune = '/usr/games/fortune'
 8
 9     def connectionMade(self):
10         output = utils.getProcessOutput(self.fortune)
11         output.addCallbacks(self.writeResponse, self.noResponse)
12
13     def writeResponse(self, resp):
14         self.transport.write(resp)
15         self.transport.loseConnection()
16
17     def noResponse(self, err):
18         self.transport.loseConnection()
19
20
21 if __name__ == '__main__':
22     f = protocol.Factory()
23     f.protocol = FortuneQuoter
24     reactor.listenTCP(10999, f)
25     reactor.run()
```

If you only need the final exit code (like commands.getstatusoutput(cmd)[0]), the twisted.internet.utils.getProcessValue function is useful. Here is an example:

如果你只是想得到最后的退出码(就像commands.getstatusoutput(cmd)[0])."twisted.internet.utils.getProcessValue"对你来说非常有用,下面是个例子:

切换行号显示

```
 1 from twisted.internet import utils, reactor
 2
 3 def printTrueValue(val):
 4     print "/bin/true exits with rc=%d" % val
 5     output = utils.getProcessValue('/bin/false')
 6     output.addCallback(printFalseValue)
 7
 8 def printFalseValue(val):
 9     print "/bin/false exits with rc=%d" % val
10     reactor.stop()
11
```

```
12 output = utils.getProcessValue('/bin/true')
13 output.addCallback(printTrueValue)
14 reactor.run()
```

7.1. 加入文件描述符的(ProcessProtocols with extra file descriptors)

When you provide a childFDs dictionary with more than the normal three fds, you need addtional methods to access those pipes. These methods are more generalized than the .outReceived ones described above. In fact, those methods (outReceived and errReceived) are actually just wrappers left in for compatibility with older code, written before this generalized fd mapping was implemented. The new list of things that can happen to your ProcessProtocol is as follows:

如果你提供了通常使用的三种fd以外的fd,你就需要编写额外的函数来访问这些管道.这些函数比上面描述的.outReceived()更通用.事实上,那些函数(.outReceived()和.errReceived())只是为了兼容性而对通用函数做了包装,主要是为了兼容在通用fd之前就已经实现的标准输入输出和标准错误输出.在你的ProcessProtocol中的新的事件列表如下:

- .connectionMade: This is called when the program is started.
- .childDataReceived(childFD, data): This is called with data that was received from one of the process' output pipes (i.e. where the childFDs value was r. The actual file number (from the point of view of the child process) is in childFD. For compatibility, the default implementation of .dataReceived dispatches to .outReceived or .errReceived when childFD is 1 or 2.
- .childConnectionLost(childFD): This is called when the reactor notices that one of the process' pipes has been closed. This either means you have just closed down the parent's end of the pipe (with .transport.closeChildFD), the child closed the pipe explicitly (sometimes to indicate EOF), or the child process has terminated and the kernel has closed all of its pipes. The childFD argument tells you which pipe was closed. Note that you can only find out about file descriptors which were mapped to pipes: when they are mapped to existing fds the parent has no way to notice when they've been closed. For compatibility, the default implementation dispatches to .inConnectionLost, .outConnectionLost, or .errConnectionLost.
- .processEnded(status): This is called when the child process has been reaped, and all pipes have been closed. This insures that all data written by the child prior to its death will be received before .processEnded is invoked.
- .connectionMade: 程序开始的时候被调用
- .childDataReceived(childFD, data): 数据从进程的一个输出管道(就是childFDs的值为"r")上到达的时候被调用.真正的文件描述符(从子进程的观点来看)在childFD中.考虑到兼容性, .dataReceived()的缺省实现是当childFD为1或者2的时候就分发数据到.outReceived()或者.errReceived()
- .childConnectionLost(childFD): 当reactor发现进程的一个管道被关闭的时候被调用.这意味着或者是你在父进程端关闭了管道(用.transport.closeChildFD()),或者是子进程显式的关闭了管道(用EOF标志),或者是子进程已经结束核心关闭了所有它的管道.参数childFD告诉你哪一个管道被关闭了.要注意的是你只能发现已经映射到管道上的文件安描述符的关闭消息,当它们映射到一句能够从在的fd的时候父进程是没有办法知道它们被关闭的.同样为了兼容性,缺省实现会分发事件到.inConnectinoLost() .outConnectLost() 或者是 .errConnectionLost()
- .processEnded(status): 当子进程完成任务,所有管道都被关闭的时候被调用.这样

可以确保所有在之前写入的数据都可以在.processEnded()调用之前收到.

In addition to those methods, there are other methods available to influence the child process:

除此之外,还有几个对子进程有影响的方法:

- self.transport.writeToChild(childFD, data): Stuff some data into an input pipe. .write simply writes to childFD=0.
- self.transport.closeChildFD(childFD): Close one of the child's pipes. Closing an input pipe is a common way to indicate EOF to the child process. Closing an output pipe is neither very friendly nor very useful.
- os.kill(self.transport.pid, signal.SIGKILL): Kill the child process. This will eventually result in processEnded being called.
- self.tranport.writeToChild(childFD, data): 往输入管道里面写数据. .write()只是写到childFD=0的管道
- self.transport.closeChildFD(childFD): 关闭子进程的一个管道.关闭输入管道是发送EOF给子进程的一种常用方法.而关闭输出管道则不怎么有用也不太友好.
- os.kill(self.transport.pid, signal.SIGKILL): 杀掉子进程.会导致processEnded()被调用.

7.2. 例子

GnuPG, the encryption program, can use additional file descriptors to accept a passphrase and emit status output. These are distinct from stdin (used to accept the crypttext), stdout (used to emit the plaintext), and stderr (used to emit human-readable status/warning messages). The passphrase FD reads until the pipe is closed and uses the resulting string to unlock the secret key that performs the actual decryption. The status FD emits machine-parseable status messages to indicate the validity of the signature, which key the message was encrypted to, etc.

GnuPG,一个加密程序,使用额外的文件描述符来得到passphrase和输出状态.和标准输入(用来接收crypttext)输出(用来打印plaintext)和标准错误输出(用来输出友好的状态/警告消息)截然不同.passphrase文件描述从管道中读入数据直到管道被关闭,然后用得到的字串解锁真正加密用的秘钥.向状态文件描述符输出易于机器解析的状态消息来指出签名的有效性.

gpg accepts command-line arguments to specify what these fds are, and then assumes that they have been opened by the parent before the gpg process is started. It simply performs reads and writes to these fd numbers.

gpg接受命令行参数来指定这些文件描述符,假定他们在gpg进程开始前已经被父进程打开.它只是通过这些文件描述符来读写.

To invoke gpg in decryption/verification mode, you would do something like the following:

为了以解密/验证模式调用gpg,可以这么做:

切换行号显示

```
1 class GPGProtocol(ProcessProtocol):
2     def __init__(self, crypttext):
3         self.crypttext = crypttext
4         self.plaintext = ""
5         self.status = ""
6     def connectionMade(self):
```

```
   7         self.transport.writeToChild(3, self.passphrase)
   8         self.transport.closeChildFD(3)
   9         self.transport.writeToChild(0, self.crypttext)
  10         self.transport.closeChildFD(0)
  11     def childDataReceived(self, childFD, data):
  12         if childFD == 1: self.plaintext += data
  13         if childFD == 4: self.status += data
  14     def processEnded(self, status):
  15         rc = status.value.exitCode
  16         if rc == 0:
  17             self.deferred.callback(self)
  18         else:
  19             self.deferred.errback(rc)
  20
  21 def decrypt(crypttext):
  22     gp = GPGProtocol(crypttext)
  23     gp.deferred = Deferred()
  24     cmd = ["gpg", "--decrypt", "--passphrase-fd", "3",
"--status-fd", "4",
  25            "--batch"]
  26     p = reactor.spawnProcess(gp, cmd[0], cmd, env=None,
  27                              childFDs={0:"w", 1:"r", 2:2, 3:"w",
4:"r"})
  28     return gp.deferred
```

In this example, the status output could be parsed after the fact. It could, of course, be parsed on the fly, as it is a simple line-oriented protocol. Methods from LineReceiver could be mixed in to make this parsing more convenient.

在这个例子中,状态输出可以被解析,它当然可以在任何时候解析(这个不会译原文on the fly),它只是个面向行的协议.LineReciver()方法调用的其它方法使的解析过程非常方便.

The stderr mapping (2:2) used will cause any GPG errors to be emitted by the parent program, just as if those errors had caused in the parent itself. This is sometimes desireable (it roughly corresponds to letting exceptions propagate upwards), especially if you do not expect to encounter errors in the child process and want them to be more visible to the end user. The alternative is to map stderr to a read-pipe and handle any such output from within the ProcessProtocol (roughly corresponding to catching the exception locally).

标准错误输出映射(2:2)使任何GPG错误都能被它的父程序输出,就像这些错误是父程序自己发生的一样.有时这就是我们想要的(就像异常的向上抛出一样),尤其是你向不想在子进程中处理错误而是使这些错误对于最终用户更易读的时候.另一个方法是映射标准错误输出管道到一个由类似ProcessProtocol处理的读管道(就像直接在本地处理异常.)

翻译 -- Jerry Marx.

(目录)Index

Version: 1.3.0

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL7

使用延迟机制(Using Deferreds)

-- -- Jerry Marx 于 [2004-08-23 04:35:55] 最后编辑

目录

1. 介绍(Introduction)

Twisted is a framework that allows programmers to develop asynchronous networked programs. twisted.internet.defer.Deferred objects are one of the key concepts that you should understand in order to develop asynchronous code that uses the Twisted framework: they are a signal to the calling function that a result is pending.

Twisted是一个运行程序员以异步方式进行网络编程的框架.如果要理解Twisted框架并用它来开发异步程序"twisted.internet.defer.Deferred"是一个非常关键的概念:它给了调用函数一个操作未决的信号

This HOWTO first describes the problem that Deferreds solve: that of managing tasks that are waiting for data without blocking. It then illustrates the difference between blocking Python code, and non-blocking code which returns a Deferred; describes Deferreds in more details; describes the details of the class interfaces; and finally describes DeferredList.

这份HOWTO首先解释Deferreds要解决什么问题:它以不阻塞的方式管理等待数据的任务.然后举例说明返回Defrred的非阻塞代码和阻塞Python代码的区别;描述了Deferreds的更多细节;详细讨论类接口;最后描述DeferredList

2. Defferreds解决什么问题(The Problem

that Deferreds Solve)

Deferreds are designed to enable Twisted programs to wait for data without hanging until that data arrives.

Deferreds被设计为一种可以让程序等待数据而不用在数据到来之前一直挂起的方式.

Many computing tasks take some time to complete, and there are two reasons why a task might take some time:

很多计算任务需要很长时间才能完成,下面列出两个可能的原因

1. it is computationally intensive (for example factorising large numbers) and requires a certain amount of CPU time to calculate the answer; or
2. it is not computationally intensive but has to wait for data to be available to produce a result.
3. 可能是高强度计算(比方说找很大数字的因子)需要大量CPU时间来计算结果;或者
4. 虽然不是高强度计算但是需要等待数据变得有效以产生结果

It is the second class of problem — non-computationally intensive tasks that involve an appreciable delay — that Deferreds are designed to help solve. Functions that wait on hard drive access, database access, and network access all fall into this class, although the time delay varies.

第二种问题--不是高强度计算,引起一个可感知的延迟--就是Deferreds要解决的问题. 函数等待硬盘访问,等待数据库访问,等待网络访问都是这类问题,尽管等待的时间各不相同.

The basic idea behind Deferreds, and other solutions to this problem, is to keep the CPU as active as possible. If one task is waiting on data, rather than have the CPU (and the program!) idle waiting for that data (a process normally called "blocking"), the program performs other operations in the meantime, and waits for some signal that data is ready to be processed before returning to that process.

Deferreds和其他解决这个问题的方法的基本思想就是尽可能的让CPU可用.如果一个任务正在等待数据,就应该在等待数据到达的信号的同时进行另一个任务,而不是让CPU(和程序!)空闲等待数据(对于进程来说一般叫做"阻塞").

In Twisted, a function signals to the calling function that it is waiting by returning a Deferred. When the data is available, the program activates the callbacks on that Deferred to process the data.

在Twisted中,函数发信号给调用函数等待返回的Deferred(这句怎么都译不通).当数据可用的时候,程序调用注册到Deferred的回调函数来处理数据.

3. 今生前世(The Context)

3.1. 用阻塞来解决(Dealing with Blocking Code)

When coding I/O based programs - networking code, databases, file access - there are many APIs that are blocking, and many methods where the common idiom is to block until a result is gotten.

在编写基于I/O的程序--网络,数据库,文件访问--的时候,会用到很多会阻塞的API函数,很多方法的惯用法就是阻塞等待.

Toggle line numbers

```
1 class Getter:
2     def getData(self, x):
```

```
3         # imagine I/O blocking code here
4         print "blocking"
5         import time
6         time.sleep(4)
7         return x * 3
8
9 g = Getter()
10 print g.getData(3)
```

3.2. 不要打电话给我,我会打给你的(Don't Call Us, We'll Call You)

Twisted cannot support blocking calls in most of its code, since it is single threaded, and event based. The solution for this issue is to refactor the code, so that instead of blocking until data is available, we return immediately, and use a callback to notify the requester once the data eventually arrives.

由于Twisted以事件驱动的单线程方式工作,所以其大部分代码都不支持阻塞调用.解决这个问题的方法就是重构,我们直接返回而不是阻塞等待数据到达,然后在数据到达候使用回调函数机制通知请求者.

Toggle line numbers

```
1 from twisted.internet import reactor
2
3 class Getter:
4     def getData(self, x, callback):
5         # this won't block
6         reactor.callLater(2, callback, x * 3)
7
8 def printData(d):
9     print d
10
11 g = Getter()
12 g.getData(3, printData)
13
14 # startup the event loop, exiting after 4 seconds
15 reactor.callLater(4, reactor.stop);
16 reactor.run()
```

There are several things missing in this simple example. There is no way to know if the data never comes back; no mechanism for handling errors. The example does not handle multiple callback functions, nor does it give a method to merge arguments before and after execution. Further, there is no way to distinguish between different calls to gotData from different producer objects. Deferred solves these problems, by creating a single, unified way to handle callbacks and errors from deferred execution.

这个例子缺少了一些东西.没有办法知道数据是否永远不会到达,没有错误处理机制.这个例子也不能处理多个回调函数,也不能提供一个函数在执行前或执行后合并参数.更进一步,它不能分辨从不同数据源获取的不同的数据.Deferred提供了一种简单,统一的方法来处理回调和错误从而可以解决上面的所有问题.

4. 延迟机制(Deferreds)

A "twisted.internet.defer.Deferred" is a promise that a function will at some point have a result. We can attach callback functions to a Deferred, and once it gets a result these callbacks will be called. In addition Deferreds allow the developer to register a callback for an error, with the default behavior of logging the error. The deferred mechanism standardizes the application programmer's interface with all sorts of blocking or delayed operations.

一个"twisted.internet.defer.Deferred"许诺一个函数在某一点会返回结果.我们可以把一些回调函数注册到Deferred,一旦有结果这些回调函数就会被调用.此外Deferred也允许开发者注册对于错误处理的回调函数取代缺省的写入日值的错误处理行为.Deferred机制提供了处理各种阻塞和延迟的一种标准化的接口.

Toggle line numbers

```
 1 from twisted.internet import reactor, defer
 2
 3 class Getter:
 4     def getData(self, x):
 5         # this won't block
 6         d = defer.Deferred()
 7         reactor.callLater(2, d.callback, x * 3)
 8         return d
 9
10 def printData(d):
11     print d
12
13 g = Getter()
14 d = g.getData(3)
15 d.addCallback(printData)
16
17 reactor.callLater(4, reactor.stop); reactor.run()
```

Deferreds do not make the code magically not block. Once you have rewritten your code to not block, Deferreds give you a nice way to build an interface to that code.

Defrreds并不是对代码施加了什么魔力使得它们不会阻塞.一旦你以非阻塞的方式重写你的代码,Deferreds可以给你创建这样接口的一种非常好的方法.

As we said, multiple callbacks can be added to a Deferred. The first callback in the Deferred's callback chain will be called with the result, the second with the result of the first callback, and so on. Why do we need this? Well, consider a Deferred returned by twisted.enterprise.adbapi - the result of a SQL query. A web widget might add a callback that converts this result into HTML, and pass the Deferred onwards, where the callback will be used by twisted to return the result to the HTTP client. The callback chain will be bypassed in case of errors or exceptions.

我们前面提到,可以注册多个回调到一个Deferred对象.Deferred的回调函数链中的第一个函数调用时会收到Deferred返回的结果,第二个回调函数被调用的时候收到第一个回调函数返回的结果,依此类推.为什么我们需要这样作?嗯,假设一个Deferred对象返回了twisted.enterprise.adbapi - SQL查询的结果.一个Web部件可以加入一个回调把这个结果转换为HTML,然后继续传给Deferred,下一个函数会把这个结果返回给HTTP客户.回调函数链在错误或者异常发生的时候会通过旁路返回.

Toggle line numbers

```
 1 from twisted.internet import reactor, defer
 2
```

```
  3 class Getter:
  4     def gotResults(self, x):
  5         """The Deferred mechanism provides a mechanism to signal
error
  6            conditions.  In this case, odd numbers are bad.
  7         """
  8         if x % 2 == 0:
  9             self.d.callback(x*3)
 10         else:
 11             self.d.errback(ValueError("You used an odd number!"))
 12
 13     def _toHTML(self, r):
 14         return "Result: %s" % r
 15
 16     def getData(self, x):
 17         """The Deferred mechanism allows for chained callbacks.
 18            In this example, the output of gotResults is first
 19            passed through _toHTML on its way to printData.
 20         """
 21         self.d = defer.Deferred()
 22         reactor.callLater(2, self.gotResults, x)
 23         self.d.addCallback(self._toHTML)
 24         return self.d
 25
 26 def printData(d):
 27     print d
 28
 29 def printError(failure):
 30     import sys
 31     sys.stderr.write(str(failure))
 32
 33 # this will print an error message
 34 g = Getter()
 35 d = g.getData(3)
 36 d.addCallback(printData)
 37 d.addErrback(printError)
 38
 39 # this will print "Result: 12"
 40 g = Getter()
 41 d = g.getData(4)
 42 d.addCallback(printData)
 43 d.addErrback(printError)
 44
 45 reactor.callLater(4, reactor.stop); reactor.run()
```

4.1. 形象解释(Visual Explanation)

<<<图1>>>

1. Requesting method (data sink) requests data, gets Deferred object.

2. Requesting method attaches callbacks to Deferred object.
3. 请求数据(从数据接收器),得到Deferred对象
4. 注册回调到Deferred对象.

<<<图2>>>

When the result is ready, give it to the Deferred object. .callback(result) if the operation succeeded, .errback(failure) if it failed. Note that failure is typically an instance of a twisted.python.failure.Failure instance. Deferred object triggers previously-added (call/err)back with the result or failure. Execution then follows the following rules, going down the chain of callbacks to be processed. Result of the callback is always passed as the first argument to the next callback, creating a chain of processors. If a callback raises an exception, switch to errback.
An unhandled failure gets passed down the line of errbacks, this creating an asynchronous analog to a series to a series of except: statements. If an errback doesn't raise an exception or return a twisted.python.failure.Failure instance, switch to callback.

当数据抵达就返回给Deferred对象. 操作成功就调用.callback(result),失败就调用.errback(failure).failure是一个"twisted.python.failure.Failure"实例 Deffered对象触发先前加入的回调/错误回调函数,传给result或failure.回调函数链中的函数会依次被执行. 如果回调过程中发生异常,就切换到错误回调中去执行. 一个未处理错误会依次调用错误回调函数链中的函数,创建一系列的和异常相似的异步相似物:状态. 如果异常回调函数不继续抛出异常或者返回"twisted.python.failure.Failure"实例,就切换到回调函数链执行.

4.2. 关于回调(More about callbacks)

注册多个回调函数到Deferred:

Toggle line numbers

```
1 g = Getter()
2 d = g.getResult(3)
3 d.addCallback(processResult)
4 d.addCallback(printResult)
```

Each callback feeds its return value into the next callback (callbacks will be called in the order you add them). Thus in the previous example, processResult's return value will be passed to printResult, instead of the value initially passed into the callback. This gives you a flexible way to chain results together, possibly modifying values along the way (for example, you may wish to pre-process database query results).

每个回调函数都把自己的返回值传给下一个回调函数(回调函数会按照注册的顺序被调用).在上门的例子中,processResult()的返回值会传给printResult(),而不是最初传给processResult()那个值.这样就可以灵活的组织回调函数链,可以在这个链的传递过程中修改数据(例如可以对数据库查询结果预处理).

4.3. 关于出错回调(More about errbacks)

Deferred's error handling is modeled after Python's exception handling. In the case that no errors occur, all the callbacks run, one after the other, as described above.

Deferred的错误处理是对Python异常处理的模拟.没有错误发生的情况下所有的回调函数(callback)会一个接一个的被调用,如上所述.

If the errback is called instead of the callback (e.g. because a DB query raised an error), then a twisted.python.failure.Failure is passed into the first errback (you can add multiple errbacks, just like with callbacks). You can think of your errbacks as being like except blocks of ordinary Python code.

如果调用了错误回调函数(errback)而不是回调函数(callback)(例如因为一个数据库查询异常),一个twisted.python.failure.Failure对象就会传给第一个错误回调函数(errback)(你可以定义多个错误回调函数,就像回调函数一样).你可以认为你的错误回调就像是原生Python代码的except块.

Unless you explicitly raise an error in except block, the Exception is caught and stops propagating, and normal execution continues. The same thing happens with errbacks: unless you explicitly return a Failure or (re-)raise an exception, the error stops propagating, and normal callbacks continue executing from that point (using the value returned from the errback). If the errback does returns a Failure or raise an exception, then that is passed to the next errback, and so on.

除非你显式的在except块抛出(raise)一个错误,错误会被捕捉而不是继续传播,正常的执行会继续.错误回调(errback)也是这样:除非你显式的返回(return)一个Failure或者(重新)抛出一个异常,错误不会继续传播,正常的回调函数会在这点开始继续执行(使用错误回调的返回值).如果错误回调确实返回一个Failure或者抛出一个异常,下一个错误回调就会被调用,依此类推.

Note: If an errback doesn't return anything, then it effectively returns None, meaning that callbacks will continue to be executed after this errback. This may not be what you expect to happen, so be careful. Make sure your errbacks return a Failure (probably the one that was passed to it), or a meaningful return value for the next callback.

注意:如果错误回调没有返回任何值,它就返回None,意味着在这个错误回调之后正常回调序列会继续执行.这也许不是你想要的,所以要小心.确定你在错误回调中返回了一个Failure(很可能就是别的函数传给它的那个),或者返回有意义的值给下一个正常回调函数(callback).

Also, twisted.python.failure.Failure instances have a useful method called trap, allowing you to effectively do the equivalent of:

此外.twisted.python.failure.Failure有一个非常有用的方法叫做陷阱(trap),你可以用它有效的做相同的事情:

Toggle line numbers

```
1 try:
2     # code that may throw an exception
3     cookSpamAndEggs()
4 except (SpamException, EggException):
5     # Handle SpamExceptions and EggExceptions
6     ...
```

You do this by:

你可以这样做:

Toggle line numbers

```
1 def errorHandler(failure):
2     failure.trap(SpamException, EggException)
3     # Handle SpamExceptions and EggExceptions
4
5 d.addCallback(cookSpamAndEggs)
6 d.addErrback(errorHandler)
```

If none of arguments passed to failure.trap match the error encapsulated in that Failure, then it re-raises the error.

如果传给failure.trap()的任何一个参数都不能匹配Failure包装的错误,这个错误就会被重新抛出.

There's another potential gotcha here. There's a method twisted.internet.defer.Deferred.addCallbacks which is similar to, but not exactly the same as, addCallback followed by addErrback. In particular, consider these two cases:

这是另一个潜在的陷阱.这里有个类似的但又不完全相同的方法twisted.internet.defer.Deferred.addCallbacks就是,在addErrBack()之后调用addCallBack().特别的,考虑下面两个例子:

Toggle line numbers

```
 1 # Case 1
 2 d = getDeferredFromSomewhere()
 3 d.addCallback(callback1)        # A
 4 d.addErrback(errback1)          # B
 5 d.addCallback(callback2)
 6 d.addErrback(errback2)
 7
 8 # Case 2
 9 d = getDeferredFromSomewhere()
10 d.addCallbacks(callback1, errback1)  # C
11 d.addCallbacks(callback2, errback2)
```

If an error occurs in callback1, then for Case 1 errback1 will be called with the failure. For Case 2, errback2 will be called. Be careful with your callbacks and errbacks.

如果在callback1()中发生了错误,Case 1中的errback1()会被传入failure来调用.在Case 2中,errback2()会被调用.运用callback和errback的时候千万要小心.

What this means in a practical sense is in Case 1, "A" will handle a success condition from getDeferredFromSomewhere, and "B" will handle any errors that occur from either the upstream source, or that occur in 'A'. In Case 2, "C"'s errback1 will only handle an error condition raised by getDeferredFromSomewhere, it will not do any handling of errors raised in callback1.

在Case 1中代码到底意思是什么呢?"A"会处理getDeferredFromSomeWhere()的成功情况,"B"会处理任何之前发生的错误,或者在"A"中发生的错误.在Case 2中,"C"的errback1只会处理getDeferredFromSomeWhere()中发生的错误,它不会处理任何callback1中抛出的错误.

4.4. 未处理的错误(Unhandled Errors)

If a Deferred is garbage-collected with an unhandled error (i.e. it would call the next errback if there was one), then Twisted will write the error's traceback to the log file. This means that you can typically get away with not adding errbacks and still get errors logged. Be careful though; if you keep a reference to the Deferred around, preventing it from being garbage-collected, then you may never see the error (and your callbacks will mysteriously seem to have never been called). If unsure, you should explicitly add an errback after your callbacks, even if all you do is:

如果Deferred在垃圾回收的时候遇到了一个未处理的错误(也就是说如果有下一个errback的话就会被调用),Twisted会把错误跟踪信息写到日志文件.这意味着你可以不

用添加errback就拥有错误日志.要小心的是,如果你保留了Deferred的一个引用而阻止它被回收的话,你可能永远都看不到错误(并且你的callback也看似神秘的永远不会被调用).如果你不是很确定的话,就应该显式的在callback后面添加一个errback,如下所示:

```
# Make sure errors get logged
from twisted.python import log
d.addErrback(log.err)
```

5. 类概述(Class Overview)

This is the overview API reference for Deferred. It is not meant to be a substitute for the docstrings in the Deferred class, but can provide guidelines for its use.

这里是浏览一下Deferred的API参考.我们的目的并不是要取代Deferred类的docstring,只是提供一个使用的指导.

5.1. 基本回调函数(Basic Callback Functions)

- addCallbacks(self, callback[, errback, callbackArgs, errbackArgs, errbackKeywords, asDefaults])

  This is the method you will use to interact with Deferred. It adds a pair of callbacks parallel to each other (see diagram above) in the list of callbacks made when the Deferred is called back to. The signature of a method added using addCallbacks should be myMethod(result, *methodArgs, **methodKeywords). If your method is passed in the callback slot, for example, all arguments in the tuple callbackArgs will be passed as *methodArgs to your method. 这是你用来和Deferred交互的方法.它添加了一对callback到平行的两条(参见上面的示意图)回调处理链中.使用addCallbacks添加的函数的签名式应该是myMethod(result, *methodArgs, **methodKeywords).举个例子:如果你的方法传给了callback槽,在元组callbackArgs中的所有参数都会作为*meghodArgs传给你的方法. There are various convenience methods that are derivative of addCallbacks. I will not cover them in detail here, but it is important to know about them in order to create concise code. 还有一些派生自addCallbacks的方便的方法.在这里我不会列出所有细节,但是如果想要写出简练的代码的话知道下面这些是很重要的.

  - addCallback(callback, *callbackArgs, **callbackKeywords)

    Adds your callback at the next point in the processing chain, while adding an errback that will re-raise its first argument, not affecting further processing in the error case. 将你的callback天加到处理链的下一点,在添加errback的时候将会重新抛出它的第一个参数,但在错误情况下不会影响未来的处理. Note that, while addCallbacks (plural) requires the arguments to be passed in a tuple, addCallback (singular) takes all its remaining arguments as things to be passed to the callback function. The reason is obvious: addCallbacks (plural) cannot tell whether the arguments are meant for the callback or the errback, so they must be specifically marked by putting them into a tuple. addCallback (singular) knows that everything is destined to go to the callback, so it can use Python's * and ** syntax to collect the remaining arguments. 注意:

addCallbacks(复数)要求传入参数以元组(tuple)的方式, addCallback(单数)要求所有要传给callback函数的参数.原因是显而易见的:addCallbacks(复数)不能分辨参数是传给callback的还是errback的,所以它们必须以元组的方式来组织,而addCallback(单数)知道要传给callback的梭鱼东西,所以它使用了Python的*和**的语法形式来获得参数.

- addErrback(errback, *errbackArgs, **errbackKeywords)
  Adds your errback at the next point in the processing chain, while adding a callback that will return its first argument, not affecting further processing in the success case. 将你的errback添加到处理链的下一点,在添加callback的时候将会返回它的第一个参数,对于在成功的情况下不影响未来的处理
- addBoth(callbackOrErrback, *callbackOrErrbackArgs, **callbackOrErrbackKeywords)
  This method adds the same callback into both sides of the processing chain at both points. Keep in mind that the type of the first argument is indeterminate if you use this method! Use it for finally: style blocks. 这个方法添加同一个callback到两个处理链.在你使用这个方法的时候记住第一个参数是不确定的.把它看作是finally块来看待吧.

- callback(result)
  Run success callbacks with the given result. This can only be run once. Later calls to this or errback will raise twisted.internet.defer.AlreadyCalledError. If further callbacks or errbacks are added after this point, addCallbacks will run the callbacks immediately.
  使用给定的result调用成功执行时的callback.这个函数只能被调用一次.再次调用这个函数或者errback会导致一个twisted.internet.defer.AlreadyCalledError错误.如果更多的callback或者errback在这一点后被添加.addCallbacks会立即调用callback.
- errback(failure)
  Run error callbacks with the given failure. This can only be run once. Later calls to this or callback will raise twisted.internet.defer.AlreadyCalledError. If further callbacks or errbacks are added after this point, addCallbacks will run the callbacks immediately.
  使用给定的failure调用错误回调函数.这个函数只能被调用一次.再次调用这个函数或者callback会导致一个twisted.internet.defer.AlreadyCalledError错误.如果更多的callback或者errback在这一点后被添加,addCallbacks会立即调用callbacks.

5.2. 延迟处理链(Chaining Deferreds)

If you need one Deferred to wait on another, all you need to do is return a Deferred from a method added to addCallbacks. Specifically, if you return Deferred B from a method added to Deferred A using A.addCallbacks, Deferred A's processing chain will stop until Deferred B's .callback() method is called; at that point, the next callback in A will be passed the result of the last callback in Deferred B's processing chain at the time.

如果需要一个Deferred等待另一个,只需要从添加到addCallbacks的方法中返回一个Deferred.特别的,如果从一个添加到Deferred A(使用A.addCallbacks)的方法返回Deferred B, Deferred A的处理链会等待Deferred B的.callback()方法被调用;然后,A的下

一个callback调用时就会传入B的处理链的最后一个callback返回的结果.

If this seems confusing, don't worry about it right now -- when you run into a situation where you need this behavior, you will probably recognize it immediately and realize why this happens. If you want to chain deferreds manually, there is also a convenience method to help you.

可能你现在会觉得有些乱,不过先不要着急 -- 当你遇到需要这种行为的情况的时候,你会马上想起这些并且明白到底发生了什么.如果你想手工来使处理链延迟,也有一个方便的方法可以调用

- chainDeferred(otherDeferred)

  Add otherDeferred to the end of this Deferred's processing chain. When self.callback is called, the result of my processing chain up to this point will be passed to otherDeferred.callback. Further additions to my callback chain do not affect otherDeferred 添加otherDeferred到这个Deferred的处理链的末尾.当self.callback被调用的时候,我的处理链在该点的最后结果会传递给otherDeferred.callback. 后面添加到我的回调链的方法不会对otherDeferred产生影响.

  This is the same as self.addCallbacks(otherDeferred.callback, otherDeferred.errback)

  这和self.addCallbacks(otherDeferred.callback, otherDeferred.errback)是一样的.

5.3. 自动化错误情况(Automatic Error Conditions)

- setTimeout(seconds[, timoutFunc])

  Set a timeout function to be triggered if this Deferred is not called within that time period. By default, this will raise a TimeoutError after seconds.

  设置一个超时回调函数,如果在指定的一段时间内Deferred没有被调用,这个函数就会被触发.缺省情况会抛出一个TimeoutError.

5.4. 打断一下,马上回来:技术细节(A Brief Interlude: Technical Details)

Deferreds greatly simplify the process of writing asynchronous code by providing a standard for registering callbacks, but there are some subtle and sometimes confusing rules that you need to follow if you are going to use them. This mostly applies to people who are writing new systems that use Deferreds internally, and not writers of applications that just add callbacks to Deferreds produced and processed by other systems. Nevertheless, it is good to know.

Deferred通过提供标准的注册回调的方法极大的简化了编写同步代码的过程,但是有一些容易混淆的地方和细节一定要搞清楚.这个主要适用于那些编制信的系统而在内部使用Deferred的人们,而不是那些编写需要处理和其他系统交叉使用callback和Deferred应用程序的人.当然,他们也最好知道这些.

Deferreds are one-shot. A generalization of the Deferred API to generic event-sources is in progress -- watch this space for updates! -- but Deferred itself is only for events that occur once. You can only call Deferred.callback or Deferred.errback once. The processing chain continues each time you add new callbacks to an already-called-back-to Deferred.

Deferred只有一次.Deferred API概括来讲就是在执行过程中的一般事件源 -- 这里会在将来被更新! -- 但是,Deferred自己只用于只能被触发一次的事件.Deferred.callback或者Deferred.errback都只能被调用一次.在每次添加新的callback到已经调用过callback的Deferred的调用链的时候调用链都会继续执行.

The important consequence of this is that sometimes, addCallbacks will call its argument synchronously, and sometimes it will not. In situations where callbacks modify state, it is highly desirable for the chain of processing to halt until all callbacks are added. For this, it is possible to pause and unpause a Deferred's processing chain while you are adding lots of callbacks.

这里有一个重要的推论就是,addCallbacks会同步的调用它的参数,但并不总是这样.在callbacks修改状态的情况下,希望处理能够停止直到所有的callbacks都被添加.这就希望在添加大量callbacks的时候可以暂停,恢复处理链的执行.(Jerry:就是说添加回调会改变回调链本身,所以这个时候希望回调链的执行停下来).

Be careful when you use these methods! If you pause a Deferred, it is your responsibility to make sure that you unpause it; code that calls callback or errback should never call unpause, as this would negate its usefulness!

使用这些方法的时候一定要小心!如果你暂停了Deferred,你就有责任去恢复它;千万不要callback和errback中调用unpause,这样作没有用.

5.5. 高级用法之处理链控制(Advanced Processing Chain Control)

- pause()

  Cease calling any methods as they are added, and do not respond to callback, until self.unpause() is called. 暂停调用添加的任何方法,也不再响应callback,直到self.unpause()被调用.

- unpause()

  If callback has been called on this Deferred already, call all the callbacks that have been added to this Deferred since pause was called. 如果Deferred的callback已经被调用,就调用所有在pause()调用以后被添加到处理链的callbacks.

  Whether it was called or not, this will put this Deferred in a state where further calls to addCallbacks or callback will work as normal. 无论它是否已经被调用,这样做都会使Deferred回到普通的状态,在这个状态下addCallbacks或者callback都会正常工作.

6. 处理同步或异步结果(Handling either synchronous or asynchronous results)

In some applications, there are functions that might be either asynchronous or synchronous. For example, a user authentication function might be able to check in memory whether a user is authenticated, allowing the authentication function to return an immediate result, or it may need to wait on network data, in which case it should return a Deferred to be fired when that data arrives. However, a function that wants to check if a user is authenticated will then need to accept both immediate results and Deferreds.

在一些应用程序中,函数可能是同步执行的也可能是异步执行的.例如,一个用户验证函数可能只是在内存中检查用户是否是认证的然后立即返回结果,也可能需要等待网络数据,这时就需要返回Deferred,它可以在数据到达的时候被触发.而作为一个需要检查用户是否被认证的函数来说,它就应该可以接受直接结果也可以接受Deferred.

In this example, the library function authenticateUser uses the application function isValidUser to authenticate a user:

在这个例子中,库函数authenticateUser()调用了应用函数isValidUser来验证用户:

Toggle line numbers

```
1 def authenticateUser(isValidUser, user):
2     if isValidUser(user):
3         print "User is authenticated"
4     else:
5         print "User is not authenticated"
```

However, it assumes that isValidUser returns immediately, whereas isValidUser may actually authenticate the user asynchronously and return a Deferred. It is possible to adapt this trivial user authentication code to accept either a synchronous isValidUser or an asynchronous isValidUser, allowing the library to handle either type of function. It is, however, also possible to adapt synchronous functions to return Deferreds. This section describes both alternatives: handling functions that might be synchronous or asynchronous in the library function (authenticateUser) or in the application code.

不管怎样,它假设isValidUser()会立即返回结果.但是isValidUser()却有可能实际上是用异步的方式验证用户而返回Deferred.可以改写这个函数让它可以接受同步的IsValidUser()也可以接受异步的IsValidUser(),这样库函数就可以接受两种类型的验证调用.其实,还可以改写同步函数让它也返回Deferred.这节描述了这两种方法:改写库函数(authenticateUser)使得可以适应同步和异步两种方式,或者改写应用程序代码.

6.1. 在库代码中处理可能的延迟(Handling possible Deferreds in the library code)

Here is an example of a synchronous user authentication function that might be passed to authenticateUser:

下面是可以传给authenticateUser()的同步用户认证函数的例子:

Toggle line numbers

```
1 def synchronousIsValidUser(d, user):
2     return user in ["Alice", "Angus", "Agnes"]
```

However, here's an asynchronousIsValidUser function that returns a Deferred:

下面是返回Deferred的asynchronousIsValidUser()函数:

Toggle line numbers

```
1 from twisted.internet import reactor
2
3 def asynchronousIsValidUser(d, user):
4     d = Deferred()
5     reactor.callLater(2, d.callback, user in ["Alice", "Angus",
"Agnes"])
6     return d
```

Our original implementation of authenticateUser expected isValidUser to be synchronous, but now we need to change it to handle both synchronous and asynchronous implementations of isValidUser. For this, we use maybeDeferred to call isValidUser, ensuring that the result of isValidUser is a Deferred, even if isValidUser is a synchronous

function:

原来authenticateUser()的实现期望isValidUser()以同步方式工作,但现在我们需要改变它以处理同步和异步两种方式的isValideUser()的实现.我们使用maybeDeferred来调用isValidUser(),保证了isValidUser()的结果一定是Deferred,就算它是同步方式工作也一样:

Toggle line numbers

```
 1 from twisted.internet import defer
 2
 3 def printResult(result):
 4     if result:
 5         print "User is authenticated"
 6     else:
 7         print "User is not authenticated"
 8
 9 def authenticateUser(isValidUser, user):
10     d = defer.maybeDeferred(isValidUser, user)
11     d.addCallback(printResult)
```

Now isValidUser could be either synchronousIsValidUser or asynchronousIsValidUser.

现在isValidUser()可以是synchronousIsValidUser()也可以是asynchronousIsValidUser().

6.2. 在异步函数中返回Deferred(Returning a Deferred from synchronous functions)

An alternative is for authenticateUser to require that the implementation of isValidUser return a Deferred:

authenticateUser()的另一个选择是把isValidUser()实现为返回Deferred:

Toggle line numbers

```
1 def printResult(result):
2     if result:
3         print "User is authenticated"
4     else:
5         print "User is not authenticated"
6
7 def authenticateUser(isValidUser, user):
8     d = isValidUser(user)
9     d.addCallback(printResult)
```

In this case, the author of synchronousIsValidUser would use defer.succeed to return a fired Deferred which will call the first callback with result, rather than returning the result itself:

在这个例子中,synchronousIsValidUser()的作者使用defer.succeed返回一个已经触发的Deferred,它会以result为参数调用第一个callback,而不是返回result自己.

Toggle line numbers

```
1 from twisted.internet import defer
2
3 def immediateIsValidUser(d, user):
```

```
4     result = user in ["Alice", "Angus", "Agnes"]
5     return defer.succeed(result)
```

7. 延迟链表(DeferredList)

Sometimes you want to be notified after several different events have all happened, rather than waiting for each one individually. For example, you may want to wait for all the connections in a list to close. twisted.internet.defer.DeferredList is the way to do this.

有时你想要在几个不同的事件都发生后才收到通知,而不是每个事件都触发一次.例如:你可能想等待一个列表中所有的连接都关闭.twisted.internet.defer.DeferredList可以做这件事.

To create a DeferredList
from multiple Deferreds, you simply pass a list of the Deferreds you want it to wait for:

从多个Deferred创建DeferredList,只需要简单的传递列表给想要等待的Deferred.

Toggle line numbers

```
1 # Creates a DeferredList
2 dl = defer.DeferredList([deferred1, deferred2, deferred3])
```

You can now treat the DeferredList like an ordinary Deferred; you can call addCallbacks and so on. The DeferredList
will call its callback when all the deferreds have completed. The callback will be called with a list of the results of the Deferreds it contains, like so:

你可以像对待普通的Deferred一样对待DeferredList;可以调用addCallbacks等等.DeferredList
会在所有的deferred都完成的时候调用他的callback.callback会被传入一列包含Deferred的结果,就像下面这样:

Toggle line numbers

```
1 def printResult(result):
2     print result
3 deferred1 = defer.Deferred()
4 deferred2 = defer.Deferred()
5 deferred3 = defer.Deferred()
6 dl = defer.DeferredList([deferred1, deferred2, deferred3])
7 dl.addCallback(printResult)
8 deferred1.callback('one')
9 deferred2.errback('bang!')
10 deferred3.callback('three')
11 # At this point, dl will fire its callback, printing:
12 #     [(1, 'one'), (0, 'bang!'), (1, 'three')]
13 # (note that defer.SUCCESS == 1, and defer.FAILURE == 0)
```

A standard DeferredList will never call errback.

标准的DeferredList永远不会调用errback.

```
Note:
If you want to apply callbacks to the individual Deferreds that go into
the DeferredList, you should be careful about when those
callbacks are added. The act of adding a Deferred to a DeferredList
inserts a callback into that Deferred (when that callback is
```

```
run, it checks to see if the DeferredList has been completed yet). The
important thing to remember is that it is this callback
which records the value that goes into the result list handed to the
DeferredList's callback.

注意:
如果你想为DeferredList中的每个Deferred单独的应用callback,就应该在添加那些
callback的时候十分小心.添加一个Deferred到DeferredList会
插入一个callback到那个Deferred(当callback运行的时候,它会检查DeferredList是否已
经都完成了).要记住的是:就是这个记录值的callback,会
被加入到处理DeferredList的callback的结果列表
```

Therefore, if you add a callback to the Deferred after adding the Deferred to the DeferredList, the value returned by that callback will not be given to the DeferredList's callback. To avoid confusion, we recommend not adding callbacks to a Deferred once it has been used in a DeferredList.

然而,如果添加一个callback给一个已经添加到DeferredList的Deferred,callback的返回值不会传递给DeferredList的callback.为了避免这种混乱,建议一旦把Deferred添加到DeferredList后就不要再给这个Deferred添加任何callback. }}}

Toggle line numbers

```
 1 def printResult(result):
 2     print result
 3 def addTen(result):
 4     return result + " ten"
 5
 6 # Deferred gets callback before DeferredList is created
 7 deferred1 = defer.Deferred()
 8 deferred2 = defer.Deferred()
 9 deferred1.addCallback(addTen)
10 dl = defer.DeferredList([deferred1, deferred2])
11 dl.addCallback(printResult)
12 deferred1.callback("one") # fires addTen, checks DeferredList,
stores "one ten"
13 deferred2.callback("two")
14 # At this point, dl will fire its callback, printing:
15 #     [(1, 'one ten'), (1, 'two')]
16
17 # Deferred gets callback after DeferredList is created
18 deferred1 = defer.Deferred()
19 deferred2 = defer.Deferred()
20 dl = defer.DeferredList([deferred1, deferred2])
21 deferred1.addCallback(addTen) # will run *after* DeferredList gets
its value
22 dl.addCallback(printResult)
23 deferred1.callback("one") # checks DeferredList, stores "one",
fires addTen
24 deferred2.callback("two")
25 # At this point, dl will fire its callback, printing:
26 #     [(1, 'one'), (1, 'two')]
```

7.1. 其它的行为(Other behaviours)

DeferredList
accepts two keywords arguments that modify its behaviour: fireOnOneCallback, fireOnOneErrback and consumeErrors. If fireOnOneCallback is set, the DeferredList will immediately call its callback as soon as any of its Deferreds call their callback. Similarly, fireOnOneErrback will call errback as soon as any of the Deferreds call their errback. Note that DeferredList
is still one-shot, like ordinary Deferreds, so after a callback or errback has been called the DeferredList
will do nothing further (it will just silently ignore any other results from its Deferreds).

DeferredList
接受两个关键字参数来更改它的行为: fireOnOneCallback, fireOnOneErrback和consumeErrors.如果fireOnOneCallback被设置,只要任何DeferredList中的一个Deferred调用callback,DeferredList
就会立即调用它的callback.类似的,fireOnOneErrback会在任何一个Deferred调用errback的时候调用它的errback.注意DeferredList也是只会有一次,就像普通的Deferred,在一个callback或者errback被调用后DeferredList以后就不会再被调用(只会悄无声息的忽略任何Deferred的结果).

The fireOnOneErrback option is particularly useful when you want to wait for all the results if everything succeeds, but also want to know immediately if something fails.
当你想等待所有的结果都成功的可以使用选项fireOnOneErrback,这样也可以在任何一个结果失败的时候立即收到通知.

The consumeErrors argument will stop the DeferredList from propagating any errors along the callback chains of any Deferreds it contains (usually creating a DeferredList has no effect on the results passed along the callbacks and errbacks of their Deferreds). Stopping errors at the DeferredList
with this option will prevent Unhandled error in Deferred warnings from the Deferreds it contains without needing to add extra errbacks[1].

参数consumeErrors会使DeferredList从它包含的任何一个Deferred的处理链中的错误传播中停止(通常情况下创建一个DeferredList不会对它包含的Deferred的callback和errback产生任何影响).使用这个选项停止DeferredList的错误处理会阻止在Deferred(该Deferred指已经加入到DeferredList中的Deferred)的未处理错误处理而不需要添加额外的errback[脚注1].

8. 脚注(Footnotes)

Unless of course a later callback starts a fresh error — but as we've already noted, adding callbacks to a Deferred after its used in a DeferredList is confusing and usually avoided.
当然,除非callback又产生新的错误 - 但是就像我们前面提到的,在将Deferred添加到DeferredList之后再给Deferred添加callback是应该被避免的.

翻译 -- Jerry Marx.

(目录)Index

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL7 (2004-09-11 02:23:27由Jerry Marx编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL8

为未来安排任务(Scheduling tasks for the future)

-- Jerry Marx 于 [2004-09-11 02:42:25] 最后编辑该页

目录

Let's say we want to run a task X seconds in the future. The way to do that is defined in the reactor interface twisted.internet.interfaces.IReactorTime:

我们想再x秒后执行一个任务,可以使用twisted.internet.interfaces.IReactorTime:

切换行号显示

```
1 from twisted.internet import reactor
2
3 def f(s):
4     print "this will run 3.5 seconds after it was scheduled: %s" %
s
5
6 reactor.callLater(3.5, f, "hello, world")
```

If we want a task to run every X seconds repeatedly, we can use twisted.internet.task.LoopingCall:

如果想每x秒就重复执行一个任务,可以使用twisted.internet.task.LoopingCall:

切换行号显示

```
1 from twisted.internet import task
2
3 def runEverySecond():
4     print "a second has passed"
5
6 l = task.LoopingCall(runEverySecond)
7 l.start(1.0) # call every second
8
9 # l.stop() will stop the looping calls
```

If we want to cancel a task that we've scheduled:

如果想要取消一个已经安排的任务:

切换行号显示

```
1 from twisted.internet import reactor
2
3 def f():
4     print "I'll never run."
5
6 callID = reactor.callLater(5, f)
7 callID.cancel()
```

翻译 -- Jerry Marx.

(目录)Index

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL8 (2004-09-11 02:42:25由Jerry Marx编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL9

在Twisted中使用线程(Using Threads in Twisted) -- JerryMarx [2004-08-03 02:38:44]

1. 介绍(Introduction)

Before you start using threads, make sure you do at the start of your program:

使用线程之前,请在程序开始包含如下代码

切换行号显示

```
1 from twisted.python import threadable
2 threadable.init()
```

This will make certain parts of Twisted thread-safe so you can use them safely. However, note that most parts of Twisted are not thread-safe.

这样可以使Twisted中的某些部分变成线程安全的,你可以安全的使用它们.然而要注意的是:Twisted中的大部分代码并不是线程安全的.

2. 以线程安全的方式运行(Running code in a thread-safe manner)

Most code in Twisted is not thread-safe. For example, writing data to a transport from a protocol is not thread-safe. Therefore, we want a way to schedule methods to be run in the main event loop. This can be done using the function twisted.internet.interfaces.IReactorThreads.callFromThread:

Twisted中的大部分代码并不是线程安全的.例如:在protocol中写数据到transport就不是线程安全的.因此,我们想要一种可以在主事件循环中调度方法的途径.可以使用这个函数:twisted.internet.interfaces.IReactorThreads.callFromThread:

切换行号显示

```
1 from twisted.internet import reactor
2 from twisted.python import threadable
3 threadable.init(1)
4
5 def notThreadSafe(x):
6       """do something that isn't thread-safe"""
7       # ...
8
9 def threadSafeScheduler():
10      """Run in thread-safe manner."""
11      reactor.callFromThread(notThreadSafe, 3) # will run
'notThreadSafe(3)'
12                                               # in the event loop
```

3. 在线程中运行(Running code in threads)

Sometimes we may want to run methods in threads - for example, in order to access blocking APIs. Twisted provides methods for doing so using the IReactorThreads API (twisted.internet.interfaces.IReactorThreads). Additional utility functions are provided in twisted.internet.threads. Basically, these methods allow us to queue methods to be run by a

thread pool.

有时我们想要在一个线程中运行方法--例如:访问会阻塞的API函数.Twisted在IReactorThreads(twisted.internet.interfaces.IReactorThreads)提供了这样做的方法.还有一些工具函数在twisted.internet.threads中提供.基本上,这些方法允许我们对方法进行排队安排它们运行在线程池.

For example, to run a method in a thread we can do:

例如:我们可以这样使一个方法运行在一个线程中.

切换行号显示

```
1 from twisted.internet import reactor
2
3 def aSillyBlockingMethod(x):
4     import time
5     time.sleep(2)
6     print x
7
8 # run method in thread
9 reactor.callInThread(aSillyBlockingMethod, "2 seconds have
passed")
```

4. 工具方法(Utility Methods)

The utility methods are not part of the twisted.internet.reactor APIs, but are implemented in twisted.internet.threads.

工具方法并不是twisted.internet.reactor的一部分,而是在twisted.internet.threads.

If we have multiple methods to run sequentially within a thread, we can do:

如果我们想要多个方法在一个线程中顺序执行,我们可以这样做:

切换行号显示

```
1 from twisted.internet import threads
2
3 def aSillyBlockingMethodOne(x):
4     import time
5     time.sleep(2)
6     print x
7
8 def aSillyBlockingMethodTwo(x):
9     print x
10
11 # run both methods sequentially in a thread
12 commands = [(aSillyBlockingMethodOne, ["Calling First"], {})]
13 commands.append((aSillyBlockingMethodTwo, ["And the second"], {}))
14 threads.callMultipleInThread(commands)
```

For functions whose results we wish to get, we can have the result returned as a Deferred:

如果我们想得到函数的返回值,我们可以让它返回Deferred:

切换行号显示

```
1 from twisted.internet import threads
2
3 def doLongCalculation():
```

```
    # .... do long calculation here ...
    return 3

def printResult(x):
    print x

# run method in thread and get result as defer.Deferred
d = threads.deferToThread(doLongCalculation)
d.addCallback(printResult)
```

5. 管理线程池(Managing the Thread Pool)

The thread pool is implemented by twisted.python.threadpool.ThreadPool.

线程池在twisted.python.threadpool.ThreadPool中实现

We may want to modify the size of the threadpool, increasing or decreasing the number of threads in use. We can do this do this quite easily:

我们也许想要改变线程池的大小,增加或者减少可用线程的数目,很容易做到:

切换行号显示

```
from twisted.internet import reactor

reactor.suggestThreadPoolSize(30)
```

The default size of the thread pool depends on the reactor being used; the default reactor uses a minimum size of 5 and a maximum size of 10. Be careful that you understand threads and their resource usage before drastically altering the thread pool sizes.

线程池的缺省大小取决于选择的是那种reactor;缺省reactor使用的最小值是5最大值是10.当你确实要改变线程数之前先确认你知道自己在做什么.

翻译 -- Jerry Marx

(目录)Index

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL9 (2004-08-09 02:03:16由dreamingk编辑)

- PyTwisted/LowLevelNetworkingEventLoop/LLNEL10

选择Reactor和GUI综合工具包(Choosing a Reactor and GUI Toolkit Integration)

-- Jerry Marx 于 [2004-09-11 05:29:50] 最后修改本页

1. 概述(Overview)

Twisted provides a variety of implementations of the twisted.internet.reactor. The specialized implementations are suited for different purposes and are designed to integrate better with particular platforms.

Twisted提供了["http://twistedmatrix.com/documents/TwistedDocs/TwistedDocs-1.3.0/api/twisted.internet.reactor.l "twisted.internet.reactor]的很多种实现.为不同的目的和设计整合到不同平台提供各种特定的实现.

The general purpose reactor implementations are:

通常目的实现的reactor实现有:

- The select()-based reactor
- The poll()-based reactor

Platform-specific reactor implementations exist for:

特定平台的实现有:

- cReactor for Unix
- KQueue for FreeBSD
- Win32
- Mac OS X

The remaining custom reactor implementations provide support for integrating with the native event loops of various graphical toolkits. This lets your Twisted application use all of the usual Twisted APIs while still being a graphical application.

其它的reactor的定制实现支持和各种图形工具包的事件循环的整合.这样Twisted应用程序就可以在图形界面程序中使用通常的Twisted API了.

Twisted currently integrates with the following graphical toolkits:

Twisted现在已经整合到下面的图形工具包中:

- GTK+ 1.2 and 2.0
- Qt
- Tkinter
- WxPython
- Win32
- Cocoa
- PyUI

When using applications that runnable using twistd, e.g. TAPs or plugins, there is no need to choose a reactor explicitly, since this can be chosen using twistd's -r option.

当使用可以用twisted(比方说TAPs或者plugins)应用程序,不需要显示的选择reactor,可以使用twisted的 -r 选项

In all cases, the event loop is started by calling reactor.run(). In all cases, the event loop should be stopped with reactor.stop().

所有的情况下,实现循环都是通过调用reactor.run()开始的,通过调用reactor.stop()结束.

IMPORTANT: installing a reactor should be the first thing done in the app, since any code that does from twisted.internet import reactor will automatically install the default reactor if the code hasen't already installed one.

重要提示:应用程序做的第一件事应该是安装一个reactor,如果还没有安装reactor的话执行from twisted.internet import reactor会自动的安装一个缺省的reactor.

2. Reactor功能(Reactor Functionality)

	TCP	SSL	UDP	Threading	Processes	Scheduling	Platforms
select()	Y	Y	Y	Y	Y(Unix only)	Y	Unix,Win32
poll()	Y	Y	Y	Y	Y	Y	Unix
Win32	Y	Y	Y	Y	Y	Y	Win32
cfreactor	Y	Y	Y	Y	Y	Y	OS X
GTK+	Y	Y	Y	Y	Y(Unix only)	Y	Unix,Win32
Qt	Y	Y	Y	Y	Y(Unix only)	Y	Unix,Win32
kqueue	Y	Y	Y	Y	Y	Y	FreeBSD
C	Y	N	N	Y	Y	Y	Unix

3. Reactor的通用意图(General Purpose Reactors)

3.1. Select()-based Reactor

The SelectReactor is the default reactor.

SelectReactor是缺省的reactor.

切换行号显示

```
1 from twisted.internet import reactor
```

The SelectReactor may be explicitly installed by:

SelectReactor可以通过如下方式显示安装:

切换行号显示

```
1 from twisted.internet import default
2 default.install()
```

3.2. Poll()-based Reactor

The PollReactor
will work on any platform that provides poll(). With larger numbers of connected sockets, it may provide for better performance.

PollReactor可以在任何提供poll()的平台上.它可以通过大量的连接套接字而提供更好的性能.

切换行号显示

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

4. Reactor平台细节(Platform-Specific Reactors)

4.1. cReactor for Unix

The cReactor is a high-performance C implementation of the Reactor interfaces. It is currently experimental and under active development.

cReactor是一个高性能的C语言实现的Ractor接口.当前它还是试验性的,还在开发中.

切换行号显示

```
1 from twisted.internet import cReactor
2 cReactor.install()
```

4.2. KQueue

The KQueue Reactor allows Twisted to use FreeBSD's kqueue mechanism for event scheduling. See instructions in the twisted.internet.kqreactor's docstring for installation notes.

KQueue Reactor允许Twisted使用FreeBSD的kqueue机制来为事件调度服务.参考twisted.internet.kqreactor的docstring以获得安装信息.

切换行号显示

```
1 from twisted.internet import kqreactor
2 kqreactor.install()
```

4.3. Win32

The Win32 reactor is not yet complete and has various limitations and issues that need to be addressed. The reactor supports GUI integration with the win32gui module, so it can be used for native Win32 GUI applications.

Win32 reactor还没有完成,目前有一些限制和问题.这个reactor支持和win32gui模块的GUI整合,它可以被用于原生的win32 GUI应用程序.

切换行号显示

```
1 from twisted.internet import win32eventreactor
2 win32eventreactor.install()
```

5. GUI整合Reactor(GUI Integration Reactors)

5.1. GTK+

Twisted integrates with PyGTK, versions 1.2 and 2.0. Sample applications using GTK+ and Twisted are available in the Twisted CVS.

Twisted和pyGTK的整合,版本号1.2和2.0. 使用GTK+和Twisted的样例程序可以在Twisted CVS中找到.

切换行号显示

```
1 from twisted.internet import gtkreactor
2 gtkreactor.install()
```

6. GUI整合Reactor(GUI Integration Reactors

6.1. Cocoa

Twisted integrates with PyObjC, version 1.0. Sample applications using Cocoa and Twisted are available in the examples directory under Cocoa.

Twisted和PyObjC的整合,版本号1.0. 使用Cocoa和Twisted的样例程序可以在例子目录中的Cocoa目录中找到

切换行号显示

```
1 from twisted.internet import cfreactor
2 cfreactor.install()
```

6.2. Qt

An example Twisted application that uses Qt can be found in doc/examples/qtdemo.py.

使用Qt的Twisted应用程序可以在doc/examples/qtdemo.py找到.

When installing the reactor, pass a QApplication instance, and if you don't a new one will be created for you.

在安装reactor的时候,传递一个QApplication实例,如果你没有这么做的话系统会自动为你创建一个.

切换行号显示

```
1 from qt import QApplication
2 app = QApplication([])
```

```
3
4 from twisted.internet import qtreactor
5 qtreactor.install(app)
```

7. 没有Reactor整合的GUI(Non-Reactor GUI Integration)

7.1. Tkinter

The support for Tkinter doesn't use a specialized reactor. Instead, there is some specialized support code:

对于TKinter的支持没有使用特定的reactor,而是一些特别的支持代码

切换行号显示

```
1 from Tkinter import *
2 from twisted.internet import tksupport
3
4 root = Tk()
5 root.withdraw()
6
7 # Install the Reactor support
8 tksupport.install(root)
9
10 # at this point build Tk app as usual using the root object,
11 # and start the program with "reactor.run()", and stop it
12 # with "reactor.stop()".
```

7.2. wxPython

As with Tkinter, the support for integrating Twisted with a wxPython application uses specialized support code rather than a simple reactor.

如同Tkinter,整合Twisted和wxPython也是使用特定得支持代码而不是实现一个简单得reactor.

切换行号显示

```
1 from wxPython.wx import *
2 from twisted.internet import wxsupport, reactor
3
4 myWxAppInstance = wxApp(0)
5 wxsupport.install(myWxAppInstance)
```

However, this has issues when runnin on Windows, so Twisted now comes with alternative wxPython support using a reactor. Using this method is probably better. Initialization is done in two stages. In the first, the reactor is installed:

在Windows上运行的时候目前还有一些问题,因此现在也可以使用reactor来支持wxPython.或许使用这种方法更好.初始化会分成两个阶段执行.首先,reactor被安装:

切换行号显示

```
1 from twisted.internet import wxreactor
```

```
2 wxreactor.install()
```

Later, once a wxApp instance has been created, but before reactor.run() is called:
然后,wxApp进程被创建,而在reactor.run()被调用前:

切换行号显示

```
1 myWxAppInstance = wxApp(0)
2 reactor.registerWxApp(myWxAppInstance)
```

An example Twisted application that uses WxWindows can be found in doc/examples/wxdemo.py
使用WxWindows的Twisted应用可以在doc/examples/wxdemo.py找到.

7.3. PyUI

As with Tkinter, the support for integrating Twisted with a PyUI application uses specialized support code rather than a simple reactor.
如同Tkinter,Twisted对于PyUI的支持也是使用特定的代码而不是一个reactor.

切换行号显示

```
1 from twisted.internet import pyuisupport, reactor
2
3 pyuisupport.install(args=(640, 480), kw={'renderer': 'gl'})
```

An example Twisted application that uses PyUI can bve found in doc/examples/pyuidemo.py 使用PyUI的Twisted样例程序可以在 doc/examples/pyuidemo.py找到.

翻译 -- Jerry Marx

(目录)Index

Version: 1.3.0

PyTwisted/LowLevelNetworkingEventLoop/LLNEL10 (2004-10-22 05:09:00由Jerry Marx编辑)

- PyTwisted/HighLevelTwisted

高阶Twisted (High-Level Twisted) -- 令狐虫

1. 工具的使用(Using the utilities)

目录

(Using the utilities)

1.1. Application

Twisted程序通常和twisted.application.service.Application协作。这个类通常管理着一个运行中的服务器的所有持久化配置信息 —— 需要绑定的端口、必须被保持或者被尝试的连接位置、需要周期性完成的动作，以及几乎所有的一切。它是服务树中的根对象，实现了IService接口。

Twisted programs usually work with twisted.application.service.Application. This class usually holds all persistent configuration of a running server -- ports to bind to, places where connections to must be kept or attempted, periodic actions to do and almost everything else. It is the root object in a tree of services implementing IService.

其他的HOWTO们描述了如何为一个应用编写客户代码，但是本节描述的是如何使用已经写好的代码（这些代码可能是Twisted的一部分，也可能来自一个第三方的Twisted插件开发者）。Twisted的发布中包含了用户建立和维护*Application*所需的各种各样的工具。

Other HOWTOs describe how to write custom code for Applications, but this one describes how to use already written code (which can be part of Twisted or from a third-party Twisted plugin developer). The Twisted distribution comes with an assortment of tools to create and manipulate *Application*S.

Application

是一种Python对象，可以像其他对象一样被建立和维护。特别的一点是，它们可以被序列化到文件中。Twisted支持几种序列化格式。

*Application*S are just Python objects, which can be created and manipulated in the same ways as any other object. In particular, they can be serialized to files. Twisted supports several serialization formats.

1.2. 序列化(Serialization)

- **TAP**
  Twisted Application Pickle。这种格式由本地的Python pickle包提供支持。这种格式虽然人类看不懂，但是存取是最快的。 A Twisted Application Pickle. This format is supported by the native Python pickle support. While not being human readable, this format is the fastest to load and save.
- **TAX**
  Twisted 包含 twisted.persisted.marmalade, 一个支持对遵循XML标准的一种格式进行序列化和解序列化的模块。这种格式是人类可读可编辑的。 Twisted contains twisted.persisted.marmalade, a module that supports serializing and deserializing from a format which follows the XML standard. This format is human readable and editable.
- **TAS**
  Twisted 包含 twisted.persisted.aot，一个支持序列化为Python代码的模块。它的好处在于使用了Python自身的语法分析器，并且以后可以手工将这段代码加入到文件中去。
  Twisted contains twisted.persisted.aot, a module that supports serializing into Python source. This has the advantage of using Python's own parser and being able to later manually add Python code to the file.

1.3. mktap和tapconvert(mktap and tapconvert)

mktap(1)工具是建立TAP(或者TAX或者TAS)文件的主要途径。它可以用来建立所有主流Twisted服务器类型——比如web、ftp或IRC——的应用。它同时也支持插件，因此当你安装一个Twisted插件（其实就是将它解压到你PYTHONPATH所指定的目录中）它会自动检测并在所有支持该插件的Twisted应用中使用它。它可以生成上述的任何一种应用格式。

The mktap(1) utility is the main way to create a TAP (or TAX or TAS) file. It can be used to create an Application for all of the major Twisted server types like web, ftp or IRC. It also supports plugins, so when you install a Twisted plugin (that is, unpack it into a directory on your PYTHONPATH) it will automatically detect it and use any Twisted Application support in it. It can create any of the above Application formats.

想了解哪些服务器类型是可用的，使用 mktap --help。mktap --help <名称> 显示一个给定服务器的可能的配置选项。mktap支持一些配置应用所需的通用选项——要了解完整的细节，可以查看man page。

In order to see which server types are available, use mktap --help. For a given server, mktap --help <name> shows the possible configuration options. mktap supports a number of generic options to configure the application -- for full details, read the man page.

有一个重要的选项是 --append <文件名>。它用于向一个已经被序列化的Twisted应用中增加一个服务器。举例来说，它可以用来增加一个telnet服务器，使你可以通过telnet来检测和重新配置应用。

One important option is --append <filename>. This is used when there is already a Twisted application serialized to which a server should be added. For example, it can be used to add a telnet server, which would let you probe and reconfigure the application by telnetting into it.

另一个有用的工具是tapconvert(1)，它用于在三种应用格式之间进行转换。

Another useful utility is tapconvert(1), which converts between all three Application

formats.

1.4. twistd

拥有一个表现为各种格式的*Application*，也许在美学角度上是令人愉悦的，但是却不能实际的导致任何事情的发生。因此，我们需要一个程序为死的应用带来生机。在UNIX系统中（并且，在有其他的选择之前，对于别的操作系统也一样），这个程序就是twistd(1)。严格的说，twistd并不是必须的——解序列化应用，得到*IService*组件，调用*startService*，当反映器(reactor)关闭时调用*stopService*，然后调用*reactor.run()*都能手工完成。然而twistd(1)提供了许多选项可以非常有效的对程序进行设置。

Having an *Application*
in a variety of formats, aesthetically pleasing as it may be, does not actually cause anything to happen. For that, we need a program which takes a dead Application and brings life to it. For UNIX systems (and, until there are are alternatives, for other operating systems too), this program is twistd(1). Strictly speaking, twistd is not necessary -- unserializing the application, getting the *IService* component, calling *startService*, scheduling *stopService* when the reactor shuts down, and then calling *reactor.run()* could be done manually. twistd(1), however, supplies many options which are highly useful for program set up.

twistd支持选择反映器(reactor)（更对关于reactors的信息，参见“选择反映器（Choosing a Reactor）”），向日志文件记录日志，开启守护进程以及其他功能。twistd支持上面提到的所有应用——和一个额外的。有时，直接用Python写代码来构建一个类很方便。在*doc/examples*目录里有一个这样的大型例子。当使用在Python文件里直接定义一个叫做*application*的*Application*对象的方法的时候，使用 -y 选项。

twistd supports choosing a reactor (for more on reactors, see Choosing a Reactor), logging to a logfile, daemonizing and more. twistd supports all Applications mentioned above -- and an additional one. Sometimes is is convenient to write the code for building a class in straight Python. One big source of such Python files is the *doc/examples* directory. When a straight Python file which defines an *Application* object called *application* is used, use the -y option.

当twistd运行的时候，它把它的进程id记录在twistd.pid文件中（这可以通过命令行开关来进行配置）。为了关闭twistd进程，杀掉那个进程的pid（你通常需要 `kill `cat twistd.pid`` ）。当这个进程被以正常形式杀掉之后，它会留下一个shutdown Application，并在这个Application的原名后面加上-shutdown。它包含一个新的，被应用更改了的配置信息。例如，关闭web.tap会产生一个web-shutdown.tap的新文件。

When twistd runs, it records its process id in a twistd.pid file (this can be configured via a command line switch). In order to shutdown the twistd process, kill that pid (usually you would do kill `cat twisted.pid`). When the process is killed in an orderly fashion it will leave behind the shutdown Application which is named the same as the original file with a -shutdown added to its base name. This contains the new configuration information, as changed in the application. For example, web.tap when shutdown will have an additional file, web-shutdown.tap.

同样的，可怕的细节都在手册页中。

As always, the gory details are in the manual page.

1.5. tap2deb

Twisted 为那些想在Debian上部署基于Twisted服务器的应用的开发者们提供了tap2deb 程序。这个程序将Twisted应用文件（的任意一种格式——Python、源代码、xml或pickle）封装成一个Debian包，包括正确的安装和删除脚本，以及init.d脚本。这将安装者从人工停止和启动服务的工作中解放出来，并且保证了它在启动和关闭时正确的遵循init级别。

For Twisted-based server application developers who want to deploy on Debian, Twisted supplies the tap2deb program. This program wraps a Twisted Application file (of any of the supported formats -- Python, source, xml or pickle) in a Debian package, including correct installation and removal scripts and init.d scripts. This frees the installer from manually stopping or starting the service, and will make sure it goes properly up on startup and down on shutdown and that it obeys the init levels.

tap2deb同样也可以为那些Debian的高级用户产生源码包，允许她修改和改进那些自动软件不能检测的东西（比如和虚拟包的依赖或关联）。另外，Twisted小组自己也打算出品一些常见服务的Debian包，比如Web服务器和inetd的替代品。那些包将会使全世界的精英们感到高兴——无论是依赖tap2deb产生的，还是Debian维护者的精美手工维护工具，都可以保证和Debian的完美集成。

For the more savvy Debian users, the tap2deb also generates the source package, allowing her to modify and polish things which automated software cannot detect (such as dependencies or relationships to virtual packages). In addition, the Twisted team itself intends to produce Debian packages for some common services, such as web servers and an inetd replacement. Those packages will enjoy the best of all worlds -- both the consistency which comes from being based on the tap2deb and the delicate manual tweaking of a Debian maintainer, insuring perfect integration with Debian.

目前，在Moshe的文档里有一个Web服务器的beta版Debian档案。

Right now, there is a beta Debian archive of a web server available at Moshe's archive.

1.6. tap2rpm

tap2rpm类似于tap2deb，只是它产生的是用于Redhat以及其他相关平台的RPM包。

tap2rpm is similar to tap2deb, except that it generates RPMs for Redhat and other related platforms.

2. Twisted组件：接口和适配器 (Twisted Components: Interfaces and Adapters)

本节翻译由Jerry Marx完成，令狐虫排版

2.1. twisted.python.components: Twisted的接口和组件的实现 (twisted.python.components: Twisted's implementation of Interfaces and Components)

2.1.1. 组件和继承Components and Inheritance

面向对象的编程语言允许程序员以创建已有类子类对象的方式来重用已有的代码。一个类子类化其他类，我们也称之为继承她的所有行为。子类可以重写(override)或者扩展超类的行为。继承在很多情况下是非常有用的，因为继承使用起来太方便了，以至于在一些大的软件系统中被滥用，涉及到多继承的时候尤其如此。对于这样的问题有一个解决方法就是在合适的地方用委托(delegation)代替继承。委托只是简单的要求另一个对象执行委托对象的任务。因为委托涉及一些互动的小组件，这种模式通常被称为组件模式 (components)。为了支持这种设计模式。Zope 3小组创造了接口(interfaces)和适配器(adapters)。

Object oriented programming languages allow programmers to reuse portions of existing code by creating new classes of objects which subclass another class. When a class subclasses another, it is said to inherit all of its behaviour. The subclass can then override and extend the behavior provided to it by the superclass. Inheritance is very useful in many situations, but because it is so convenient to use, often becomes abused in large software systems, especially when multiple inheritance is involved. One solution is to use delegation instead of inheritance where appropriate. Delegation is simply the act of asking another object to perform a task for an object. To support this design pattern, which is often referred to as the components pattern because it involves many small interacting components, interfaces and adapters were created by the Zope 3 team.

接口只是一个简单的可以被一个对象用来说“我实现了这个接口”的标志。其它的对象就可以发出类似“请给我一个实现了为Y对象使用的X接口”的请求。为另一种对象实现一个接口的对象就称之为适配器。

Interfaces are simply markers which objects can use to say I implement this interface. Other objects may then make requests like Please give me an object which implements interface X for object type Y. Objects which implement an interface for another object type are called adapters.

超类-子类的关系是一种is-a关系。在设计对象继承层次的时候，对象模块使用子类化技术，这样就可以说子类是父类同样的类。例如：

The superclass-subclass relationship is said to be an is-a relationship. When designing object hierarchies, object modellers use subclassing when they can say that the subclass is the same class as the superclass. For example:

Toggle line numbers

```
 1 class Shape:
 2     sideLength = 0
 3     def getSideLength(self):
 4         return self.sideLength
 5
 6     def setSideLength(self, sideLength):
 7         self.sideLength = sideLength
 8
 9     def area(self):
10         raise NotImplementedError, "Subclasses must implement
area"
11
12 class Triangle(Shape):
13     def area(self):
14         return (self.sideLength * self.sideLength) / 2
15
16 class Square(Shape):
17     def area(self):
```

```
  18         return self.sideLength * self.sideLength
```

在上面的例子中，三角形(Triangle)是一种(is-a)形状(Shape)，因此Triangle类子类化了Shape类，正方形(Square)也是一种(is-a)形状(Shape)，因此Square类也子类化了Shape类。

In the above example, a Triangle is-a Shape, so it subclasses Shape, and a Square is-a Shape, so it also subclasses Shape.

然而，子类化可以变的非常复杂，尤其在多继承的情况下。多继承允许一个类继承一个以上基类。过分依赖于继承的软件通常形成扩展很广层次很深的继承树，一个类继承自多个超类可能会贯穿到整个系统中。由于多继承意味着实现继承(implementation inheritance)，定位一个方法的实现和确认调用正确的方法变成了一种挑战。例如：

However, subclassing can get complicated, especially when Multiple Inheritance enters the picture. Multiple Inheritance allows a class to inherit from more than one base class. Software which relies heavily on inheritance often ends up having both very wide and very deep inheritance trees, meaning that one class inherits from many superclasses spread throughout the system. Since subclassing with Multiple Inheritance means implementation inheritance, locating a method's actual implementation and ensuring the correct method is actually being invoked becomes a challenge. For example:

Toggle line numbers

```
   1 class Area:
   2     sideLength = 0
   3     def getSideLength(self):
   4         return self.sideLength
   5 
   6     def setSideLength(self, sideLength):
   7         self.sideLength = sideLength
   8 
   9     def area(self):
  10         raise NotImplementedError, "Subclasses must implement
area"
  11 
  12 class Color:
  13     color = None
  14     def setColor(self, color):
  15       self.color = color
  16 
  17     def getColor(self):
  18       return self.color
  19 
  20 class Square(Area, Color):
  21     def area(self):
  22         return self.sideLength * self.sideLength
```

程序员们喜欢使用实现继承因为这样可以使代码可读性更好，因为计算面积的代码和颜色的代码在不同的地方。这很好，有一些对象可以有颜色但是没有面积，另一些对象有面积，但是没有颜色。问题在于，正方形(Square)并不是一个面积(Area)，也不是一个颜色(Color)。因此我们应该使用另外一种叫做组件的面向对象技术，它使用委托而不是继承来实现代码的重用。我们将继续使用这个经常在各种练习中用到的多继承的例子。

The reason programmers like using implementation inheritance is because it makes code easier to read since the implementation details of Area are in a separate place than the implementation details of Color. This is nice, because conceivably an object could have a color but not an area, or an area but not a color. The problem, though, is that Square is not really an Area or a Color, but has an area and color. Thus, we should really be using another object oriented technique called composition, which relies on delegation rather than inheritance to break code into small reusable chunks. Let us continue with the Multiple Inheritance example, though, because it is often used in practice.

如果Color和Area这两个基类定义了同名的方法-比方说叫做calculate-会怎么样呢？在哪里实现它呢？Square().calculate()函数实现的地方依赖于方法推演顺序(MRO: method resolution order)，并且也许会被程序员重构其它看似不相关的系统中的其他代码而影响，产生晦涩的bug。我们首先想到的也许是将calculate这个函数名改为calculateArea和calculateColor来避免命名冲突。显然这样的改变会影响到整个系统的很大部分，特别是当你在整合两个不是自己写的不同的系统的时候很容易出错。

What if both the Color and the Area base class defined the same method, perhaps calculate? Where would the implementation come from? The implementation that is located for Square().calculate() depends on the method resolution order, or MRO, and can change when programmers change seemingly unrelated things by refactoring classes in other parts of the system, causing obscure bugs. Our first thought might be to change the calculate method name to avoid name clashes, to perhaps calculateArea and calculateColor. While explicit, this change could potentially require a large number of changes throughout a system, and is error-prone, especially when attempting to integrate two systems which you didn't write.

我们来看另外的例子。我们有一个电吹风(hair dryer)，它使用美国的电压标准，而我们有两个插座，一个是110伏另一个是220伏(非美国标准)。如果将电吹风查到220伏的插座上就会出现一个错误，因为它期望的是110伏的电压。让电吹风去支持plug110Volt和 plug220Volt这样两个方法是很无聊的事情，那么如果我们需要把电吹风插到另一个插座上去该怎么办呢？例如：

Let's imagine another example. We have an electric appliance, say a hair dryer. The hair dryer is american voltage. We have two electric sockets, one of them an american 110 Volt socket, and one of them a foreign 220 Volt socket. If we plug the hair dryer into the 220 Volt socket, it is going to expect 110 Volt current and errors will result. Going back and changing the hair dryer to support both plug110Volt and plug220Volt methods would be tedious, and what if we decided we needed to plug the hair dryer into yet another type of socket? For example:

Toggle line numbers

```
 1 class HairDryer:
 2     def plug(self, socket):
 3         if socket.voltage() == 110:
 4             print "I was plugged in properly and am operating."
 5         else:
 6             print "I was plugged in improperly and "
 7             print "now you have no hair dryer any more."
 8
 9 class AmericanSocket:
10     def voltage(self):
11         return 110
12
13 class ForeignSocket:
```

```
14      def voltage(self):
15          return 220
```

有了这些类，就可以做以下操作了：

Given these classes, the following operations can be performed:

```
>>> hd = HairDryer()
>>> am = AmericanSocket()
>>> hd.plug(am)
I was plugged in properly and am operating.
>>> fs = ForeignSocket()
>>> hd.plug(fs)
I was plugged in improperly and
now you have no hair dryer any more.
```

我们准备为ForeignSocket
开发一个适配器来解决这个问题，这个适配器可以将电压转换为适合电吹风使用的电压。转换器是一个有一个单参数构造函数的类，这个参数是adaptee或者叫做original object在twisted.python.components.Adapter中有一个简单的实现，它定义了如下所示的`__init__`，你可以在希望的时候继承它。在这个例子中，我们清楚的展示了全部代码：

We are going to attempt to solve this problem by writing an Adapter for the ForeignSocket which converts the voltage for use with an American hair dryer. An Adapter is a class which is constructed with one and only one argument, the adaptee or original object. There is a simple implementation in twisted.python.components.Adapter which defines the `__init__`
shown below, so you can subclass it if you desire. In this example, we will show all code involved for clarity:

Toggle line numbers

```
1 class AdaptToAmericanSocket:
2     def __init__(self, original):
3         self.original = original
4
5     def voltage(self):
6         return self.original.voltage() / 2
```

现在，我们可以这样来使用它:

Now, we can use it as so:

```
>>> hd = HairDryer()
>>> fs = ForeignSocket()
>>> adapted = AdaptToAmericanSocket(fs)
>>> hd.plug(adapted)
I was plugged in properly and am operating.
```

如你所见，适配器可以改写(override)原有的实现，它也可以通过提供原有对象没有的方法扩展(extend)原有的接口。注意适配器必须显式的(explicitly)委托任何不想改变的原有方法。否则就不能在使用原有对象的地方使用该适配器。通常这不会是问题，因为适配器通常为了是一个对象适合用于特定接口，然后就被丢弃了。

So, as you can see, an adapter can 'override' the original implementation. It can also

'extend' the interface of the original object by providing methods the original object did not have. Note that an Adapter must explicitly delegate any method calls it does not wish to modify to the original, otherwise the Adapter cannot be used in places where the original is expected. Usually this is not a problem, as an Adapter is created to conform an object to a particular interface and then discarded.

== twisted.python.components: Twisted的接口和组件的实现 (twisted.python.components: Twisted's implementation of Interfaces and Components) ==

适配器对于使用多个类来组织一个离散模块(discrete chunks)的时候是一个很有用的方法。可是在没有基础组件的时候也不是件有趣的事情。如果每段希望使用的代码都必须显示的构造一个适配器的话，组件间的耦合就太紧了。而我们想要的是一种松耦合，于是就有了twisted.python.components。

Adapters are a useful way of using multiple classes to factor code into discrete chunks. However, they are not very interesting without some more infrastructure. If each piece of code which wished to use an adapted object had to explicitly construct the adapter itself, the coupling between components would be too tight. We would like to achieve loose coupling, and this is where twisted.python.components comes in.

首先，我们需要讨论接口的更多细节。前面提到过，接口就是一个被用作标识(used as a marker)的类。接口应该是twisted.python.components.Interface的子类，Python程序员不这样用的话实在很奇怪

First, we need to discuss Interfaces in more detail. As we mentioned earlier, an Interface is nothing more than a class which is used as a marker. Interfaces should be subclasses of twisted.python.components.Interface, and have a very odd look to python programmers not used to them:

Toggle line numbers

```
1 from twisted.python import components
2
3 class IAmericanSocket(components.Interface):
4     def voltage(self):
5       """Return the voltage produced by this socket object, as an
integer.
6       """
```

注意，除了继承自components.Interface之外这只是通常的类定义。可是在类定义中的只有空方法！由于Python不具有像Java那样对于接口的语言级的内建支持，这就是区别类定义和接口定义的方法。

Notice how it looks just like a regular class definition, other than inheriting from components.Interface. However, the method definitions inside the class block do not have any method body! Since Python does not have any native language-level support for Interfaces like Java does, this is what distinguishes an Interface definition from a Class.

现在我们有了一个接口的定义，我们可以使用如下的术语来讨论：类 AmericanSocket 实现了接口IAmericanSocket ， 请可给我一个可用于接口 IAmericanSocket 的ForeignSocket 对象的适配器。我们可以声明(declarations) 实现类实现了哪个接口，然后请求为特定对象适合某个接口的适配器。

Now that we have a defined Interface, we can talk about objects using terms like this: The AmericanSocket
class implements the IAmericanSocket interface and Please give me an object which adapts ForeignSocket
to the IAmericanSocket interface. We can make declarations about what interfaces a certain class implements, and we can request adapters which implement a certain interface

for a specific class.

我们来看我们是如何声明一个类实现一个接口的：

Let's look at how we declare that a class implements an interface:

Toggle line numbers

```
1 class AmericanSocket:
2     __implements__ = (IAmericanSocket, )
3     def voltage(self):
4         return 110
```

声明一个类实现一个接口，只是简单的设置接口元组(tuple of interfaces)给类成员implements。在Python中圆括号括起来的后面有个逗号的变量表示一个只有一项的元组。

So, to declare that a class implements an interface, we simply set the implements class variable to a tuple of interfaces. A single item tuple in Python is created by enclosing an item in parentheses and placing a single trailing comma after it.

现在我们将AdaptToAmericanSocket改写为一个真正的适配器。我们只是简单的从components.Adapter继承然后实现接口IamericanSocket的方法。

Now, let's say we want to rewrite the AdaptToAmericanSocket class as a real adapter. We simply subclass components.Adapter and provide implementations of the methods in the IAmericanSocket interface:

Toggle line numbers

```
1 class AdaptToAmericanSocket(components.Adapter):
2     __implements__ = (IAmericanSocket, )
3     def voltage(self):
4         return self.original.voltage() / 2
```

注意在adapter中的实现声明。迄今为止，我们还没有为达成目标做任何事情。为了让这些组件变得有用，必须使用组件注册(component registry)。AdaptToAmericanSocket实现了IamericanSocket并且把电压调整为可用于对象ForeignSocket，我们可以注册AdaptToAmericanSocket为IamericanSocket 可用于ForeignSocket的适配器。直接阅读这些代码比描述它们更容易。

Notice how we placed the implements declaration on this adapter class. So far, we have not achieved anything by using components other than requiring us to type more. In order for components to be useful, we must use the component registry. Since AdaptToAmericanSocket implements IAmericanSocket and regulates the voltage of a ForeignSocket object, we can register AdaptToAmericanSocket as an IAmericanSocket adapter for the ForeignSocket
class. It is easier to see how this is done in code than to describe it:

Toggle line numbers

```
1 from twisted.python import components
2
3 class IAmericanSocket(components.Interface):
4     def voltage(self):
5       """Return the voltage produced by this socket object, as an
integer.
6       """
7
8 class AmericanSocket:
```

```
 9      __implements__ = (IAmericanSocket, )
10      def voltage(self):
11          return 110
12
13  class ForeignSocket:
14      def voltage(self):
15          return 220
16
17  class AdaptToAmericanSocket(components.Adapter):
18      __implements__ = (IAmericanSocket, )
19      def voltage(self):
20          return self.original.voltage() / 2
21
22  components.registerAdapter(
23      AdaptToAmericanSocket,
24      ForeignSocket,
25      IAmericanSocket)
```

现在，如果在解释器中运行这段脚本，我们就可以更了解于组件。第一件要做的事情就是去检查一个对象是否实现了一个接口： Now, if we run this script in the interactive interpreter, we can discover a little more about how to use components. The first thing we can do is discover whether an object implements an interface or not:

```
>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> components.implements(as, IAmericanSocket)
1
>>> components.implements(fs, IAmericanSocket)
0
```

如你所见，AmericanSocket 声明它实现接口IAmericanSocket, 但是 ForeignSocket 没有。如果想一起使用HairDryer和AmericanSocket，我们可以通过检查它是否实现了接口IamericanSocket来得知是否可以安全的使用。无论如何，当我们想一起使用HairDryer和ForeignSocket的时候，我们就必须首先将它适配到IAmericanSocket。我们使用接口对象来做:

As you can see, the AmericanSocket instance claims to implement IAmericanSocket, but the ForeignSocket does not. If we wanted to use the HairDryer with the AmericanSocket, we could know that it would be safe to do so by checking whether it implements IAmericanSocket. However, if we decide we want to use HairDryer with a ForeignSocket instance, we must adapt it to IAmericanSocket before doing so. We use the interface object to do this:

```
>>> IAmericanSocket(fs)
<__main__.AdaptToAmericanSocket instance at 0x1a5120>
```

以一个对象作为参数调用一个接口(的构造函数)的时候，接口就会去在适配器注册表中查找可用的适配器(为传入对象转换到该接口的适配器)。如果找到这样的适配器，就以原始对象作为参数构造一个适配器的一个实例并返回这个实例。现在HairDryer就可以和经过转换的ForeignSocket一起安全的工作了。但是，当我们试图转换一个已经实现了IamericanSocket的对象会发生什么呢？我们简单的得回原始的那个实例：

When calling an interface with an object as an argument, the interface looks in the adapter registry for an adapter which implements the interface for the given instance's class. If it finds one, it constructs an instance of the Adapter class, passing the constructor the original instance, and returns it. Now the HairDryer can safely be used with the adapted ForeignSocket. But what happens if we attempt to adapt an object which already implements IAmericanSocket? We simply get back the original instance:

```
>>> IAmericanSocket(as)
<__main__.AmericanSocket instance at 0x36bff0>
```

好了，我们可以写一个具有智能的电吹风(smartHairDryer)了，它可以自动的寻找可用的适配器：

So, we could write a new smartHairDryer which automatically looked up an adapter for the socket you tried to plug it into:

Toggle line numbers

```
1 class HairDryer:
2     def plug(self, socket):
3         adapted = IAmericanSocket(socket)
4         assert socket.voltage() == 110, "BOOM"
5         print "I was plugged in properly and am operating"
```

现在我们创建smartHairDryer的一个实例并试图把它接到各种插座上去，HairDryer会自动根据插座的类型去使用合适的适配器：

Now, if we create an instance of our new smartHairDryer and attempt to plug it in to various sockets, the HairDryer will adapt itself automatically depending on the type of socket it is plugged in to:

```
>>> as = AmericanSocket()
>>> fs = ForeignSocket()
>>> hd = HairDryer()
>>> hd.plug(as)
I was plugged in properly and am operating
>>> hd.plug(fs)
I was plugged in properly and am operating
```

瞧瞧！我们有了一个具有魔力的组件。

Voila; the magic of components.

2.2. 组件和继承 (Components and Inheritance)

如果继承了一个已经声明实现(implements)了某个接口的类，而子类又需要定义了自己的`__implements__`
元组(因为它又实现了另外的接口)，自列必须显式的包含基类的实现接口表(base clase interface list)。实现接口声明不会从继承树中得到(因为会严重影响性能)，没有采用递归查找类属性`__implements__`的方式。

If you inherit from a class which `__implements__` some interface, and your new subclass defines its own `__implements__` tuple (because it implements an additional interface), you must explicitly include the base clase interface list. The Interface code does not walk the complete inheritance tree (because that would be a significant performance

hit), but instead will recursively expand any tuples it finds in each class's `__implements__` attribute.

举个例子，pb.Root (定义在flavors.Root) 是一个实现了接口IPBRoot的类。 这个接口表明它是一个具有远程调用方法的对象，这个远程调用方法可以使用一个新的Broker实例提供初始化对象的服务。它又一个`__implements__`属性：

For example, pb.Root (actually defined in flavors.Root) is a class which implements IPBRoot. This interface indicates that an object has remotely-invokable methods and can be used as the initial object served by a new Broker instance. It has an `__implements__` attribute like:

Toggle line numbers

```
1 class Root(Referenceable):
2     __implements__ = IPBRoot,
```

假设你有一个实现了你自己接口IMyInterface 的类

Suppose you have your own class which implements your IMyInterface interface:

Toggle line numbers

```
1 class MyThing:
2     __implements__ = IMyInterface,
```

如果你想让这个类继承自pb.Root，你必须自己包括pb.Root接口列表。 Now if you want to make this class inherit from pb.Root, you must manually include pb.Root's interface list:

Toggle line numbers

```
1 class MyThing(pb.Root):
2     __implements__ = (IMyInterface, pb.Root.__implements__)
```

PyTwisted/HighLevelTwisted (2004-09-10 05:21:39由61编辑)

- PyTwisted/TwistedUtilities/ParsingCommandLines

Using usage.Options

-- Jerry Marx [2004-09-15 21:41:07]

Using

目录

usage.Options

1. 介绍

(Introduction)

There is frequently a need for programs to parse a UNIX-like command line program: options preceded by - or --, sometimes followed by a parameter, followed by a list of arguments. The twisted.python.usage provides a class, Options, to facilitate such parsing.

程序需要频繁的分析类UNIX的命令行:由-或者--引导的选项,有时后面跟着一个选项,有时是一系列."twisted.python.usage"提供了一个类Options来做这样的命令行解析.

While Python has the getopt module for doing this, it provides a very low level of abstraction for options. Twisted has a higher level of abstraction, in the class twisted.python.usage.Options. It uses Python's reflection facilities to provide an easy to use yet flexible interface to the command line. While most command line processors either force the application writer to write her own loops, or have arbitrary limitations on the command line (the most common one being not being able to have more then one instance of a specific option, thus rendering the idiom program -v -v -v impossible), Twisted allows the programmer to decide how much control she wants.

Python本身的模块getopt也可以做这件事,它提供了一个非常低层次的选项提取.Twisted有一个高层次的选项提取,就是类twisted.python.usage.Options.他使用了Python的反射机制,提供更容易使用的非常灵活的命令行接口.大多数命令行处理或者要求应用程序有自己的处理循环,或者对于命令行格式有一些武断的限制(最常见的就是不能处理一个选项多次出现,解析类似于 program -v -v -v 是不可能的),而Twisted则允许程序员自己决定他(她)控制到什么程度.

The Options class is used by subclassing. Since a lot of time it will be used in the twisted.tap package, where the local conventions require the specific options parsing class to also be called Options, it is usually imported with

类Options一般通过子类来使用.由于在twisted.tag包种也使用了它,而且因为一些习惯和其它原因它自己的选项分析类也叫做Options,因此通常用如下的方式来引入这个类

Toggle line numbers

```
1 from twisted.python import usage
```

2. Boolean Options

For simple boolean options, define the attribute optFlags like this:

对于简单的布尔选项,可以定义属性optFlags:

Toggle line numbers

```
1 class Options(usage.Options):
2
3     optFlags = [["fast", "f", "Act quickly"], ["safe", "s", "Act
safely"]]
```

optFlags should be a list of 3-lists. The first element is the long name, and will be used on the command line as --fast. The second one is the short name, and will be used on the command line as -f. The last element is a description of the flag and will be used to generate the usage information text. The long name also determines the name of the key that will be set on the Options instance. Its value will be 1 if the option was seen, 0 otherwise. Here is an example for usage:

optFlags应该是一个每个表项都是一个三项列表的列表,第一个项是长名称,可以这样使用: --fast. 第二个项是短名称,可以这样使用: -f. 最后一个项是用于使用说明文字的关于这个选项的描述文字.长名字也决定了在Options实例这个选项对应的变量的名称.如果有这个选项,变量就设置为1,如果没有就设置为0.下面是个例子:

Toggle line numbers

```
1 class Options(usage.Options):
2
3     optFlags = [
4         ["fast", "f", "Act quickly"],
5         ["good", "g", "Act well"],
6         ["cheap", "c", "Act cheaply"]
7     ]
8
9 command_line = ["-g", "--fast"]
10
11 options = Options()
12 try:
13     options.parseOptions(command_line)
14 except usage.UsageError, errortext:
15     print '%s: %s' % (sys.argv[0], errortext)
16     print '%s: Try --help for usage details.' % (sys.argv[0])
17     sys.exit(1)
18 if options['fast']:
19     print "fast",
20 if options['good']:
21     print "good",
22 if options['cheap']:
23     print "cheap",
24 print
```

The above will print fast good.

上面的程序会打印: fast good.

Note here that Options fully supports the mapping interface. You can access it mostly just like you can access any other dict. Options are stored as mapping items in the Options instance: parameters as 'paramname': 'value' and flags as 'flagname': 1 or 0.

注意这里Options完全支持mapping接口.可以像访问其它的字典一样访问它.在Options实例中各个选项作为字典的项被存入,parameters作为'paramname': 'value'和flags作为'flagname': 1或者0.

2.1. 继承，或者说：我如何才能爱上Superclass，不再因它而烦心？

(Inheritance, Or: How I Learned to Stop Worrying and Love the Superclass)

Sometimes there is a need for several option processors with a unifying core. Perhaps you want all your commands to understand -q/--quiet means to be quiet, or something similar. On the face of it, this looks impossible: in Python, the subclass's optFlags would shadow the superclass's. However, usage.Options uses special reflection code to get all of the optFlags defined in the hierarchy. So the following:

有时需要用同样的方式来处理几个选项,也许你想让你所有的命令都理解 -q/--quiet 的意思就是安静,或者类似的其它东西.从表面上看起来这是不可能的,子类的optFlags会覆盖父类中的optFlags.然而,usage.Options使用了特殊的反射代码获得整个继承体系的定义,所以下面这样:

Toggle line numbers

```
1 class BaseOptions(usage.Options):
2
3     optFlags = [["quiet", "q", None]]
4
5 class SpecificOptions(BaseOptions):
6
7     optFlags = [
8         ["fast", "f", None], ["good", "g", None], ["cheap", "c",
None]
9     ]
```

Is the same as:

和下面这样是相同的

Toggle line numbers

```
1 class SpecificOptions(BaseOptions):
2
3     optFlags = [
4         ["quiet", "q", "Silence output"],
5         ["fast", "f", "Run quickly"],
6         ["good", "g", "Don't validate input"],
7         ["cheap", "c", "Use cheap resources"]
8     ]
```

3. Parameters

Parameters are specified using the attribute optParameters. They must be given a default. If you want to make sure you got the parameter from the command line, give a non-string default. Since the command line only has strings, this is completely reliable.

Parameters被指定使用了属性optParameters.他们必需给定一个缺省值.如果你想确认你从命令行获得了一个parameter,给他一个非字符串作为缺省值.因命令行只会产生字

符串,这样做是可靠的.

Here is an example:

下面是个例子:

Toggle line numbers

```
 1 from twisted.python import usage
 2
 3 class Options(usage.Options):
 4
 5     optFlags = [
 6         ["fast", "f", "Run quickly"],
 7         ["good", "g", "Don't validate input"],
 8         ["cheap", "c", "Use cheap resources"]
 9     ]
10     optParameters = [["user", "u", None, "The user name"]]
11
12 config = Options()
13 try:
14     config.parseOptions() # When given no argument, parses
sys.argv[1:]
15 except usage.UsageError, errortext:
16     print '%s: %s' % (sys.argv[0], errortext)
17     print '%s: Try --help for usage details.' % (sys.argv[0])
18     sys.exit(1)
19
20 if config['user'] is not None:
21     print "Hello", config['user']
22 print "So, you want it:"
23
24 if config['fast']:
25     print "fast",
26 if config['good']:
27     print "good",
28 if config['cheap']:
29     print "cheap",
30 print
```

Like optFlags, optParameters works smoothly with inheritance.

就像optFlags一样,optPrameters也可以在继承关系下很好的工作.

4. 可选择的子命令(Option Subcommands)

It is useful, on occassion, to group a set of options together based on the logical action to which they belong. For this, the usage.Options class allows you to define a set of subcommands, each of which can provide its own usage.Options instance to handle its particular options.

在某些场合下,把一组逻辑相关的选项组织在一起很有用.使用usage.Options类可以定义一组子命令,每个子命令都有一个它自己的usage.Options实例来处理他自己的选项.

Here is an example for an Options class that might parse options like those the cvs program takes

下面是一个可以处理类似cvs命令的例子:

Toggle line numbers

```
  1 from twisted.python import usage
  2
  3 class ImportOptions(usage.Options):
  4     optParameters = [
  5         ['module', 'm', None, None], ['vendor', 'v', None, None],
  6         ['release', 'r', None]
  7     ]
  8
  9 class CheckoutOptions(usage.Options):
 10     optParameters = [['module', 'm', None, None], ['tag', 'r',
None, None]]
 11
 12 class Options(usage.Options):
 13     subCommands = [['import', None, ImportOptions, "Do an
Import"],
 14                    ['checkout', None, CheckoutOptions, "Do a
Checkout"]]
 15
 16     optParameters = [
 17         ['compression', 'z', 0, 'Use compression'],
 18         ['repository', 'r', None, 'Specify an alternate
repository']
 19     ]
 20
 21 config = Options(); config.parseOptions()
 22 if config.subCommand == 'import':
 23     doImport(config.subOptions)
 24 elif config.subCommand == 'checkout':
 25     doCheckout(config.subOptions)
```

The subCommands attribute of Options directs the parser to the two other Options subclasses when the strings "import" or "checkout" are present on the command line. All options after the given command string are passed to the specified Options subclass for further parsing. Only one subcommand may be specified at a time. After parsing has completed, the Options instance has two new attributes - subCommand and subOptions - which hold the command string and the Options instance used to parse the remaining options.

当命令行宗出现字符串"import"或者"checkout"地时候,选项的子命令属性就会指引分析器去分析另外两个选项子类.在给定命令串后所有的选项都会传递给特定的Options子类去进一步解析.每次只会指定一个子命令.解析完成后,Options实例有两个新属性 - subCommand 和 subOptions - 他们持有用来解析命令行剩余选项所需要的命令字符串和Options实例.

5. 实现选项的通用代码(Generic Code For Options)

Sometimes, just setting an attribute on the basis of the options is not flexible enough. In those cases, Twisted does not even attempt to provide abstractions such as counts or lists, but rathers lets you call your own method, which will be called whenever the option is encountered.

有时,只是把属性设置给options并不够灵活.对应这种情况,Twisted并没有试图提供像counts或者lists这样的属性,而是允许你调用你自己的方法,这些方法可以在选项被计数的任何时候调用.

Here is an example of counting verbosity

下面是一个对冗长参数的计数的例子:

Toggle line numbers

```
 1 from twisted.python import usage
 2
 3 class Options(usage.Options):
 4
 5     def __init__(self):
 6         usage.Options.__init__(self)
 7         self['verbosity'] = 0 # default
 8
 9     def opt_verbose(self):
10         self['verbosity'] = self['verbosity']+1
11
12     def opt_quiet(self):
13         self['verbosity'] = self['verbosity']-1
14
15     opt_v = opt_verbose
16     opt_q = opt_quiet
```

Command lines that look like command -v -v -v will increase verbosity to 4, while command -q -q will decrease verbosity to -3.

如 command -v -v -v 的命令行将会使verbosity增加到4, command -q -q -q 会使verbosity增加到-3.

The usage.Options class knows that these are parameter-less options, since the methods do not receive an argument. Here is an example for a method with a parameter:

usage.Options类知道哪些无参数的选项,因此那些方法不接受参数.下面是个例子:

Toggle line numbers

```
 1 from twisted.python import usage
 2
 3 class Options(usage.Options):
 4
 5     def __init__(self):
 6         usage.Options.__init__(self)
 7         self['symbols'] = []
 8
 9     def opt_define(self, symbol):
10         self['symbols'].append(symbol)
11
12     opt_D = opt_define
```

This example is useful for the common idiom of having command -DFOO -DBAR to define symbols

这个例子对于定义符号的习惯用法 -DFOO -DBAR 很有用.

6. 解析参数(Parsing Arguments)

usage.Options does not stop helping when the last parameter is gone. All the other arguments are sent into a function which should deal with them. Here is an example for a cmp like command.

usage.Options并没有在分析完最后一个参数就停止.所有的其它的参数都被送到相应的处理函数中去处理,下面是一个类似cmp的例子:

Toggle line numbers

```
1 from twisted.python import usage
2
3 class Options(usage.Options):
4
5     optParameters = [["max_differences", "d", 1, None]]
6
7     def parseArgs(self, origin, changed):
8         self['origin'] = origin
9         self['changed'] = changed
```

The command should look like command origin changed.

这个命令看起来应该是 command origin changed.

If you want to have a variable number of left-over arguments, just use def parseArgs(self, *args):. This is useful for commands like the UNIX cat(1).

如果你想有不定参数,只需要使用定义 parseArgs(self, *args):.这对于像UNIX cat(1)之类的命令很有用.

7. 后处理(Post Processing)

Sometimes, you want to perform post processing of options to patch up inconsistencies, and the like. Here is an example:

有时,你想在处理完选项后处理冲突,或者其它事情.下面是个例子:

Toggle line numbers

```
1 from twisted.python import usage
2
3 class Options(usage.Options):
4
5     optFlags = [
6         ["fast", "f", "Run quickly"],
7         ["good", "g", "Don't validate input"],
8         ["cheap", "c", "Use cheap resources"]
9     ]
10
11     def postOptions(self):
12         if self['fast'] and self['good'] and self['cheap']:
13             raise usage.UsageError, "can't have it all, brother"
```

翻译 -- Jerry Marx

"目录(Index)"

Version 1.3.0

- PyTwisted/TwistedUtilities/UsingDirdbm

1. DirDBM: 基于目录的存储系统(DirDBM: Directory-based Storage)

1.1. dirdbm.DirDBM

twisted.persisted.dirdbm.DirDBM是一种类似于DBM的存储系统。它可以像Python里的字典(Dictionary)一样保存键-值映射。与字典不同的是，DirDBM的每一条映射都保存为一个单独的文件，而且键与值都要求是字符串。除此之外，DirDBM的行为与Python里的字典是基本一致的。

twisted.persisted.dirdbm.DirDBM is a DBM-like storage system. That is, it stores mappings between keys and values, like a Python dictionary, except that it stores the values in files in a directory - each entry is a different file. The keys must always be strings, as are the values. Other than that, DirDBM objects act just like Python dictionaries.

如果你想保存少量的数据，而且又不想用复杂的关系数据库系统或其它高深的数据库，那么，DirDBM非常的适合您。与Python自带的DBM模块不同的是，它小巧，易用，跨平台而且还不用任何外部的C库。

DirDBM is useful for cases when you want to store small amounts of data in an organized fashion, without having to deal with the complexity of a RDBMS or other sophisticated database. It is simple, easy to use, cross-platform, and doesn't require any external C libraries, unlike Python's built-in DBM modules.

Toggle line numbers

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.DirDBM("/tmp/dir")
>>> d["librarian"] = "ook"
>>> d["librarian"]
'ook'
>>> d.keys()
['librarian']
>>> del d["librarian"]
>>> d.items()
[]
```

1.2. dirdbm.Shelf

我们有时会想要保存一些比字符串更复杂的对象。稍微小心一点的话，dirdbm.Shelf可以保存这些更复杂的对象。Shelf和DirDBM的功能非常相似，而且Shelf的值(不是键)可以为任意对象，只要这个对象可以被保存。不过，一个对象在保存后被改变过的话，后来的改变不会被保存。如果改变了对象，你需要在改变后显式的再保存一次：

Sometimes it is neccessary to persist more complicated objects than strings. With some care, dirdbm.Shelf can transparently persist them. Shelf works exactly like DirDBM,

except that the values (but not the keys) can be arbitrary picklable objects. However, notice that mutating an object after it has been stored in the Shelf has no effect on the Shelf. When mutating objects, it is neccessary to explictly store them back in the Shelf afterwards:

Toggle line numbers

```
>>> from twisted.persisted import dirdbm
>>> d = dirdbm.Shelf("/tmp/dir2")
>>> d["key"] = [1, 2]
>>> d["key"]
[1, 2]
>>> l = d["key"]
>>> l.append(3)
>>> d["key"]
[1, 2]
>>> d["key"] = l
>>> d["key"]
[1, 2, 3]
```

PyTwisted/TwistedUtilities/UsingDirdbm (2005-07-22 16:27:24由LeoJay编辑)

- PyTwisted/TwistedUtilities/UsingTelnetToManipulateaTwistedServer

1. 用telnet操作一个twisted服务器(Using telnet to manipulate a twisted server)

在开始之前，我们要建立一个只允许远程访问Python解释器的服务器。我们将用telnet来访问这个服务器

To start things off, we're going to create a simple server that just gives you remote access to a Python interpreter. We will use a telnet client to access this server.

在命令行运行mktap telnet -p 4040 -u admin -w admin。如果你现在查看你的当前目录，你会发现多了一个新文件：telnet.tap。然后你再运行twistd -f telnet.tap。这个telnet服务器将会侦听所有连接到你所指定的4040端口的连接。试着用你喜欢的telnet工具连接到127.0.0.1的404端口。

Run mktap telnet -p 4040 -u admin -w admin at your shell prompt. If you list the contents of your current directory, you'll notice a new file -- telnet.tap. After you do this, run twistd -f telnet.tap. Since the Application has a telnet server that you specified to be on port 4040, it will start listening for connections on this port. Try connecting with your favorite telnet utility to 127.0.0.1 port 4040.

```
$ telnet localhost 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

twisted.manhole.telnet.ShellFactory
Twisted 1.1.0
username: admin
password: admin
>>>
```

现在，你应该可以看到Python的提示符：>>>。你可以在这里输入任何合法的Python代码，我们来试试吧：

Now, you should see a Python prompt -- >>>. You can type any valid Python code here. Let's try looking around.

Toggle line numbers

```
1 >>> dir()
2 ['__builtins__']
```

很好，再来一些复杂点的：

Ok, not much. let's play a little more:

Toggle line numbers

```
1 >>> import __main__
2 >>> dir(__main__)
3 ['__builtins__', '__doc__', '__name__', 'os', 'run', 'string',
'sys']
```

```
4
5 >>> service
6 <twisted.application.internet.TCPServer instance at 0x10270f48>
7 >>> service._port
8 <twisted.manhole.telnet.ShellFactory on 4040>
9 >>> service.parent
10 <twisted.application.service.MultiService instance at 0x1024d7a8>
```

service对象是用来响应telnet连接的服务器，它正在用一个叫ShellFactory的东西来侦听4040端口。它的父类是一个服务的集合，叫做twisted.application.service.MultiService。我们可以一直得到服务父类的属性，直到我们到达这个tap中所有服务的根。

The service object is the service used to serve the telnet shell, and that it is listening on port 4040 with something called a ShellFactory. Its parent is a twisted.application.service.MultiService, a collection of services. We can keep getting the parent attribute of services until we hit the root of all services in this tap.

正如你所看到的，这个真的很有用，我们可以观察一个正在运行的进程，可以看它内部的对象，甚至可以改变这些对象的属性。我们可以往已有的tap添加telnet支持，如：mktap --append=foo.tap telnet -p 4040 -u user -w pass。这个telnet服务器当然也可以由python代码直接使用。你可以通过看twisted.tap.telnet的代码来了解具体怎么做。

As you can see, this is quite useful - we can introspect a running process, see the internal objects, and even change their attributes. We can add telnet support to existing tap like so: mktap --append=foo.tap telnet -p 4040 -u user -w pass. The telnet server can of coursed be used from straight Python code as well. You can see how to do this by reading the code for twisted.tap.telnet.

最后，如果你想要更安全的访问，你可以让telnet服务器使用SSL。如果你有了适当的证书和私钥文件，你可以运行mktap telnet -p ssl:443:privateKey=mykey.pem:certKey=cert.pem -u admin -w admin。想到了解更多有关侦听端口的参数请参考twisted.application.strports。

A final note - if you want access to be more secure, you can even have the telnet server use SSL. Assuming you have the appropriate certificate and private key files, you can mktap telnet -p ssl:443:privateKey=mykey.pem:certKey=cert.pem -u admin -w admin. See twisted.application.strports for more examples of options for listening on a port.

PyTwisted/TwistedUtilities/UsingTelnetToManipulateaTwistedServer (2005-07-22 17:08:00由LeoJay编辑)

- PyTwisted/TwistedUtilities/TwistedRdbmsSupport

Twisted的关系数据库支持(Twisted RDBMS support)

-- Jerry Marx [2004-09-15 04:30:49]

目录

1. 概要(Abstract)

Twisted is an asynchronous networking framework, but most database API implementations unfortunately have blocking interfaces -- for this reason, twisted.enterprise.adbapi was created. It is a non-blocking interface to the standardized DB-API 2.0 API, which allows you to access a number of different RDBMSes.

Twisted是异步的网络编程框架,不幸的是大多数数据库API的实现都是阻塞的接口,因为这个原因创造了twisted.enterprise.adbapi.它是一个符合DB-API 2.0 标准的非阻塞接口,使用它可以访问很多关系数据库

2. 你应该已经知道的(What you should already know)

- Python
- How to write a simple Twisted Server (see this tutorial to learn how) 如何用Twisted写一个简单的Server(参见教程学习如何做)
- Familiarity with using database interfaces (see the documentation for DBAPI 2.0 or this article by Andrew Kuchling) 对使用数据库接口很熟悉(参见DBAPI 2.0 的文档或者由 Andrew Kuchling 写的这份文档)

3. 快速回顾(Quick Overview)

Twisted is an asynchronous framework. This means standard database modules cannot be used directly, as they typically work something like:

Twisted是一个异步编程框架,这就意味着标准的数据库模块不能直接使用,他们的典型用法如下:

Toggle line numbers

```
# Create connection...
db = dbmodule.connect('mydb', 'andrew', 'password')
# ...which blocks for an unknown amount of time

# Create a cursor
cursor = db.cursor()

# Do a query...
resultset = cursor.query('SELECT * FROM table WHERE ...')
# ...which could take a long time, perhaps even minutes.
```

Those delays are unacceptable when using an asynchronous framework such as Twisted. For this reason, twisted provides twisted.enterprise.adbapi, an asynchronous wrapper for

any DB-API 2.0-compliant module.

对于像Twisted这样的异步编程框架来说延迟是不能接受的.因为这个原因,twisted提供了twisted.enterprise.adbapi,它是一个对于任何兼容DB-API 2.0 的异步包装模块.

enterprise.adbapi will do blocking database operations in seperate threads, which trigger callbacks in the originating thread when they complete. In the meantime, the original thread can continue doing normal work, like servicing other requests.

enterprise.adbapi在独立的线程里面执行阻塞的数据库操作,而是在操作完成后触发原线程的回调函数.同时,原始线程可以继续执行,比方说处理其他的请求.

4. 怎么使用adbapi(How do I use adbapi?)

Rather than creating a database connection directly, use the adbapi.ConnectionPool class to manage a connections for you. This allows enterprise.adbapi to use multiple connections, one per thread. This is easy:

使用adbapi.ConnectionPool类来管理你的数据库连接,而不要直接连接.这个类可以使用多个连接,比方说每个线程一个连接,非常容易:

Toggle line numbers

```
1 # Using the "dbmodule" from the previous example, create a ConnectionPool
2 from twisted.enterprise import adbapi
3 dbpool = adbapi.ConnectionPool("dbmodule", 'mydb', 'andrew', 'password')
```

Things to note about doing this:

这么做的时候要注意以下这些事情:

- There is no need to import dbmodule directly. You just pass the name to adbapi.ConnectionPool's constructor.
- The parameters you would pass to dbmodule.connect are passed as extra arguments to adbapi.ConnectionPool's constructor. Keyword parameters work as well.
- 不需要直接import dbmodule.只需要把名称传递给adbapi.ConnectionPool的构造函数.
- 要传递给dbmodule.connect的参数就是传递给adbapi.ConnectionPool的构造函数的参数,关键字参数可以很好的工作.

Now we can do a database query:

现在我们可以做数据库查询了:

Toggle line numbers

```
1 # equivalent of cursor.execute(statement), return cursor.fetchall():
2 def getAge(user):
3     return dbpool.runQuery("SELECT age FROM users WHERE name = ?", user)
4
5 def printResult(l):
6     if l:
7         print l[0][0], "years old"
8     else:
9         print "No such user"
10
```

```
11 getAge("joe").addCallback(printResult)
```

This is straightforward, except perhaps for the return value of getAge. It returns a twisted.internet.defer.Deferred, which allows arbitrary callbacks to be called upon completion (or upon failure). More documentation on Deferred is available here.

除了getAge的返回值可能有些奇怪以外,上面的代码一目了然.getAge返回了一个twisted.internet.defer.Deferred,可以给它添加任何的成功回调(或者错误回调).关于Deferred的更多信息可以查阅文档.

In addition to runQuery, there is also runOperation, and runInteraction that gets called with a callable (e.g. a function). The function will be called in the thread with a twisted.enterprise.adbapi.Transaction, which basically mimics a DB-API cursor. In all cases a database transaction will be commited after your database usage is finished, unless an exception is raised in which case it will be rolled back.

除了runQuery外,还有runOperation和runInteraction可以杯一个可调用实体(比方说一个函数)来调用.这个函数会在一个拥有twisted.enterprise.adbapi.Transaction的线程里调用,它简单模拟了一个DB-API游标.

Toggle line numbers

```
 1 def _getAge(txn, user):
 2     # this will run in a thread, we can use blocking calls
 3     txn.execute("SELECT * FROM foo")
 4     # ... other cursor commands called on txn ...
 5     txn.execute("SELECT age FROM users WHERE name = ?", user)
 6     result = txn.fetchall()
 7     if result:
 8         return result[0][0]
 9     else:
10         return None
11
12 def getAge(user):
13     return dbpool.runInteraction(_getAge, user)
14
15 def printResult(age):
16     if age != None:
17         print age, "years old"
18     else:
19         print "No such user"
20
21 getAge("joe").addCallback(printResult)
```

Also worth noting is that these examples assumes that dbmodule uses the qmarks paramstyle (see the DB-API specification). If your dbmodule uses a different paramstyle (e.g. pyformat) then use that. Twisted doesn't attempt to offer any sort of magic paramater munging -- runQuery(query, params, ...) maps directly onto cursor.execute(query, params, ...).

另外还要注意这个例子假设dbmodule使用qmarks paramstyle(参见DB-API规范).如果你的dbmodule使用了不同的paramstyle(比方说pyformat)你就要改相应的地方.Twisted没有提供任何参数排序 -- runQuery(query, params, ...)直接映射到cursor.execute(query, params, ...).

5. 这样!(And that's it!)

That's all you need to know to use a database from within Twisted. You probably should read the adbapi module's documentation to get an idea of the other functions it has, but hopefully this document presents the core ideas.

以上就是在Twisted种使用数据库需要知道的全部知识.你或许应该阅读adbapi模块的问道那个来获得另一些函数的信息,希望这个文档给了你基本的概念.

翻译 -- Jerry Marx

"目录(Index)"

version 1.3.0

PyTwisted/TwistedUtilities/TwistedRdbmsSupport (2004-09-15 19:50:00由Jerry Marx编辑)

- PyTwisted/TwistedUtilities/TheRowDatabaseAbstraction

系统说明书 文章模板

-- Jerry Marx [2004-09-17 01:52:14]

目录

The twisted.enterprise.row module is a method of interfacing simple python objects with rows in relational database tables. It has two components: the RowObject class which developers sub-class for each relational table that their code interacts with, and the Reflector which is responsible for updates, inserts, queries and deletes against the database.

twisted.enterprise.row模块是简单Python对象和关系数据库数据表中的行之间交互的一种方法.它有两个组件:RowObject类,开发人员通过可以子类化这个类来和关系数据库的表交互;Reflector类,它的职责是更新,插入,查询和删除数据库中的数据.

The row module is intended for applications such as on-line games, and websites that require a back-end database interface. It is not a full functioned object-relational mapper for python - it deals best with simple data types structured in ways that can be easily represented in a relational database. It is well suited to building a python interface to an existing relational database, and slightly less suited to added database persistance to an existing python application.

行模式是为例如在线游戏和网站而设计的,这些应用需要一个后台的数据库接口.它并不是一个关系对象函数在Python的完整映射 - 它和关系数据库表现的简单数据类型结构协作的非常好.很适合用来构建一个python接口来和已有的关系数据库交互,而对于为已有的python应用添加数据库实现持久数据稍微有些不适合.

If row does not fit your model, you will be best off using the low-level database API directly, or writing your own object/relational layer on top of it.

如果row不适合你的模型,最好直接使用底层的数据库API,或者在API的基础上包装自己的对象/相关层.

1. 类定义(Class Definitions)

To interface to relational database tables, the developer must create a class derived from the twisted.enterprise.row.RowObject class for each table. These derived classes must define a number of class attributes which contains information about the database table that class corresponds to. The required class attributes are:

为了访问关系数据库表,开发人员必须为每个表创建一个继承自twisted.enterprise.row.RowObject类的子类.这些子类需要定义一些属性来描述对应的表的信息.必须的类属性如下所示:

- rowColumns - list of the column names and types in the table with the correct case
  表中列名和类型的列表(并且有正确的大小写).
- rowKeyColumns - list of key columns in form: [(columnName, typeName)]
  以[(columnName, typeName)]形式纪录的列关键字列表.
- rowTableName - the name of the database table
  数据表名称.

There are also two optional class attributes that can be specified:

还有两个可以指定的可选属性.

- rowForeignKeys - list of foreign keys to other database tables in the form: [(tableName, [(childColumnName, childColumnType), ...], [(parentColumnName, parentColumnType), ...], containerMethodName, autoLoad]
  外部关键字列表,以形式[(tableName, [(childColumnName, childColumnType), ...], [(parentColumnName, parentColumnType), ...], containerMethodName, autoLoad]列出.
- rowFactoryMethod - a method that creates instances of this class
  创建这个类的实例的方法.

For example:

例如:

行番号表示/非表示切替

```
 1 class RoomRow(row.RowObject):
 2     rowColumns        = [("roomId",   "int"),
 3                          ("town_id",  "int"),
 4                          ("name",     "varchar"),
 5                          ("owner",    "varchar"),
 6                          ("posx",     "int"),
 7                          ("posy",     "int"),
 8                          ("width",    "int"),
 9                          ("height",   "int")]
10     rowKeyColumns     = [("roomId",   "int4")]
11     rowTableName      = "testrooms"
12     rowFactoryMethod = [testRoomFactory]
```

The items in the rowColumns list will become data members of classes of this type when they are created by the Reflector.

rowcolumns的各项会在他们被Reflector创建的时候成为类的数据成员.

2. 初始化(Initialization)

The initialization phase builds the SQL for the database interactions. It uses the system catalogs of the database to do this, but requires some basic information to get started. The class attributes of the classes derived from RowClass are used for this. Those classes are passed to a Reflector when it is created.

初始化阶段构在为和数据库交互的SQL语句.它使用数据库的系统目录,但还是要求一些基本的信息才能开始.继承自RowClass的类的属性就是为了提供这些信息.当Reflector创建的时候这些类被传递给它.

There are currently two available reflectors in Twisted Enterprise, the SQL Reflector for relational databases which uses the python DB API, and the XML Reflector which uses a file system containing XML files. The XML reflector is currently extremely slow.

现在Twisted Enterprise里有两个可用的reflector,为关系数据库服务的SQL Reflector使用了python的DB API,还有一个XML Reflector使用了包含XML文件的文件系统.XML reflector现在非常慢.

An example class list for the RoomRow class we specified above using the SQLReflector:

下面这个例子使用了SQLRelfector:

行番号表示/非表示切替

```
1 from twisted.enterprise.sqlreflector import SQLReflector
2
3 dbpool = adbapi.ConnectionPool("pyPgSQL.PgSQL")
4 reflector = SQLReflector( dbpool, [RoomRow] )
```

3. 创建一个行对象(Creating Row Objects)

There are two methods of creating RowObjects - loading from the database, and creating a new instance ready to be inserted.

创建RowObject有两个方法 - 从数据库加载,创建一个将要被插入的新的实例.

To load rows from the database and create RowObject instances for each of the rows, use the loadObjectsFrom method of the Reflector. This takes a tableName, an optional user data parameter, and an optional where clause. The where clause may be omitted which will retrieve all the rows from the table. For example:

从数据库加载行并且为每行都创建一个RowObject实例,可以使用Reflector的方法loadObjectFrom.它需要一个tableName,一个可选的用户数据参数,一个可选的where子句.当需要找到表的所有行的时候where子句可以被忽略.例如:

行番号表示/非表示切替

```
1 def gotRooms(rooms):
2     for room in rooms:
3         print "Got room:", room.id
4
5 d = reflector.loadObjectsFrom("testrooms",
6                                 whereClause=[("id", reflector.EQUAL,
5)])
7 d.addCallback(gotRooms)
```

For more advanced RowObject construction, loadObjectsFrom may use a factoryMethod that was specified as a class attribute for the RowClass derived class. This method will be called for each of the rows with the class object, the userData parameter, and a dictionary of data from the database keyed by column name. This factory method should return a fully populated RowObject
instance and may be used to do pre-processing, lookups, and data transformations before exposing the data to user code. An example factory method:

行番号表示/非表示切替

```
1 def testRoomFactory(roomClass, userData, kw):
2     newRoom = roomClass(userData)
3     newRoom.__dict__.update(kw)
4     return newRoom
```

The last method of creating a row object is for new instances that do not already exist in the database table. In this case, create a new instance and assign its primary key attributes and all of its member data attributes, then pass it to the insertRow method of the Reflector. For example:

newRoom = RoomRow()

newRoom.assignKeyAttr("roomI", 11) newRoom.town_id = 20 newRoom.name = 'newRoom1' newRoom.owner = 'fred' newRoom.posx = 100 newRoom.posy = 100 newRoom.width = 15 newRoom.height = 20
reflector.insertRow(newRoom).addCallback(onInsert)

This will insert a new row into the database table for this new RowObject instance. Note that the assignKeyAttr method must be used to set primary key attributes - regular attribute assignment of a primary key attribute of a rowObject will raise an exception. This prevents the database identity of RowObject from being changed by mistake.

4. 表之前的关系(Relationships Between Tables)

5. 复制行对象(Duplicate Row Objects)

6. 更新行对象(Updating Row Objects)

7. 删除行对象(Deleting Row Objects)

PyTwisted/TwistedUtilities/TheRowDatabaseAbstraction (2004-09-17 01:52:14由Jerry Marx编辑)

- PyTwisted/tasteTwisted

040721 开始定期的进行UC组中的,会课,进行Twisted 的讲解,大家的练习成果在此展示 -- dreamingk [2004-08-09 02:20:07]

1. TwistedTUT 01

目录

- Twisted TUT 练习1 要求
    - 完成一个服务器侦听2020端口
    - 用户telnet上去后输入一个url，服务器将这个url的信息抓回返回给用户
    - 完成一个客户端，传入一个参数，将得到的信息显示出来

1.1. *inux版本

利用 unix 平台的优点!

1.2. M$ 版本

Python!哪里也可以!

1.2.1. limodou版本

切换行号显示

```
 1 #reactor version
 2
 3 from twisted.internet import protocol, reactor, defer, utils
 4 from twisted.protocols import basic
 5 from twisted.web import client
 6 class FingerProtocol(basic.LineReceiver):
 7     def lineReceived(self, url):
 8         self.factory.getUrl(url
 9         ).addErrback(lambda _: "Internal error in server"
10         ).addCallback(lambda m:
11                       (self.transport.write(m+"\r\n"),
12                        self.transport.loseConnection()))
13 class FingerFactory(protocol.ServerFactory):
14     protocol = FingerProtocol
15     def getUrl(self, url):
16         return client.getPage(url)
17 reactor.listenTCP(2020, FingerFactory())
18 reactor.run()
```

去掉了不必要的**init**函数，改了函数名及参数

切换行号显示

```
#application version

from twisted.application import internet, service
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
from twisted.web import client

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, url):
        self.factory.getUrl(url
        ).addErrback(lambda _: "Internal error in server"
        ).addCallback(lambda m:
                      (self.transport.write(m+"\r\n"),
                       self.transport.loseConnection()))
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def getUrl(self, url):
        return client.getPage(url)

application = service.Application('finger', uid=1, gid=1)
factory = FingerFactory()
internet.TCPServer(2020, factory).setServiceParent(
service.IServiceCollection(application))
```

切换行号显示

```
#client.py

from socket import *

ADDR = '127.0.0.1'
PORT = 2020

def sendraw(data):
    sendSock = socket(AF_INET, SOCK_STREAM)
    sendSock.connect((ADDR, PORT))
    sendSock.send(data+'\r\n')
    fp = file('data.txt', 'wb')
    while 1:
        text = sendSock.recv(1024)
        if text:
            fp.write(text)
        else:
            break
    sendSock.close()
    fp.close()

if __name__ == '__main__':
    url = raw_input('enter url:')
    sendraw(url)
```

存在问题，如果使用telnet好象数据接收不全，但使用客户端程序还可以。

1.2.2. 0.706版本

切换行号显示

```
#server.py
from twisted.internet import protocol, reactor, defer
from twisted.protocols import basic
from twisted.web import client
class MyProtocol(basic.LineReceiver):
    def connectionMade(self):
        self.transport.write("Welcome,Input A URL:\r\n")

    def lineReceived(self, url):
        self.factory.getUser(url).addErrback(lambda _:
        "Get %s error in server \r\n"%url).addCallback(lambda m:
                       (self.transport.write(m+"\r\n"),
                        self.transport.loseConnection()))
class MyFactory(protocol.ServerFactory):
    protocol = MyProtocol
    def __init__(self):
        pass
    def getUser(self, url):
        return client.getPage(url)
reactor.listenTCP(2020, MyFactory())
print "Server Started!"
reactor.run()
```

我可不喜欢对着一个毫无提示的屏幕.

切换行号显示

```
#client.py
import sys
from socket import *

def main():
    Sock = socket(AF_INET, SOCK_STREAM)
    err=Sock.connect_ex(("127.0.0.1", 2020))
    if(err):
        print "Connect Error"
    else:
        page = Sock.recv(8192)
        sys.stdout.write(page)

        url = raw_input('')

        Sock.send(url+'\r\n')
        page = Sock.recv(8192)
        sys.stdout.write(page)

    Sock.close()


```

```
23 if __name__ == '__main__':
24     main()
```

通过测试，发现在输入URL时得输全才能得到结果,如输入 "http://www.163.com" 会出错,而输入 "http://www.163.com/" 就可以

1.2.3. Zoom.Quiet版本

1.2.3.1. for reactor

切换行号显示

```
1 # -*- coding: gbk -*-
2 # file zwget.py
3 """ reactor version
4 """
5 from twisted.internet import protocol, reactor, defer, utils
6 from twisted.protocols import basic
7 from twisted.web import client
8 import sys, os, string, time, re
9 class FingerProtocol(basic.LineReceiver):
10     def connectionMade(self):
11         self.transport.write("Hollo! 输入URL吧::\r\n")
12     def lineReceived(self, url):
13         self.factory.getUrl(url
14         ).addErrback(lambda _: "网络连接错误，或是URL错误!"
15         ).addCallback(lambda m:
16                       (self.echo(url,m))
17                       )
18     def echo(self,url,messg):
19         label =
time.strftime("%m%d%H%M%S",time.localtime(time.time()))
20         if "bye"==url:
21             self.transport.write("再见!\r\n")
22             self.transport.loseConnection()
23         else:
24             if "URL错误" in messg:
25                 self.transport.write(messg+"\r\n")
26             else:
27                 open("zgweb-%s.html"%label,"w").write(messg)
28                 self.transport.write("抓取内容%s字节\r\n写入
zgweb-%s.html\r\n 嗯嗯! 再来! \r\n"%(
29                                                 len(messg),label)
30                                     )
31 class FingerFactory(protocol.ServerFactory):
32     protocol = FingerProtocol
33     def getUrl(self, url):
34         return client.getPage(url)
35 reactor.listenTCP(2020, FingerFactory())
36 reactor.run()
```

- 不断根据URL输入,将结果输出为.html文件,直到输入 bye ;
- 感谢 0.706 的提示,知道了:

```
BaseProtocol --+
               |
        Protocol --+
                   |
                 LineReceiver
中,有从协议继承的  connectionFailed(self)
 用以响应协议创建成功!
```

connectionFailedin Twisted API

1.2.3.2. for application

切换行号显示

```
  1 # -*- coding: gbk -*-
  2 # file zawget.py
  3 """ application version
  4 """
  5 from twisted.internet import protocol, reactor, defer
  6 from twisted.protocols import basic
  7 from twisted.web import client
  8 from twisted.application import internet, service
  9 import sys, os, string, time, re
 10 class FingerProtocol(basic.LineReceiver):
 11     def connectionMade(self):
 12         self.transport.write("Hollo! 输入URL吧::\r\n")
 13     def lineReceived(self, url):
 14         self.factory.getUrl(url
 15         ).addErrback(lambda _: "网络连接错误，或是URL错误!"
 16         ).addCallback(lambda m:
 17                       (self.echo(url,m))
 18                       )
 19     def echo(self,url,messg):
 20         label =
time.strftime("%m%d%H%M%S",time.localtime(time.time()))
 21         if "bye"==url:
 22             self.transport.write("再见!\r\n")
 23             self.transport.loseConnection()
 24         else:
 25             if "URL错误" in messg:
 26                 self.transport.write(messg+"\r\n")
 27             else:
 28                 open("zgweb-%s.html"%label,"w").write(messg)
 29                 self.transport.write("抓取内容%s字节\r\n写入
zgweb-%s.html\r\n 嗯嗯! 再来! \r\n"%(
 30                                                 len(messg),label)
 31                                         )
 32 class FingerFactory(protocol.ServerFactory):
```

```
33      protocol = FingerProtocol
34      def getUrl(self, url):
35          return client.getPage(url)
36  application = service.Application('wget', uid=1, gid=1)
37  factory = FingerFactory()
38  internet.TCPServer(2020, factory).setServiceParent(
39      service.IServiceCollection(application))
```

果然方便!其实就改最后的服务创建部分,三行而已!

1.2.3.3. for client

切换行号显示

```
1  # -*- coding: gbk -*-
2  # file zcwget.py
3  """ socket client version
4  """
5  from socket import *
6  ADDR = '127.0.0.1'
7  PORT = 2020
8  def client():
9      Sock = socket(AF_INET, SOCK_STREAM)
10     err=Sock.connect_ex((ADDR, PORT))
11     if(err): print "连接失败!"
12     else:
13         page = Sock.recv(8192)
14         print page
15         while 1:
16             url = raw_input('')
17             if "bye" in url: break
18             else:
19                 Sock.send(url+'\r\n')
20                 page = Sock.recv(8192)
21                 print page
22     Sock.close()
23 if __name__ == '__main__':
24     client()
```

- 不明白这客户端是什么意思呢? telnet 也算客户端吧? 嗯嗯,就是可以访问我们创建的服务的脚本是也乎!?
- 不熟!参考前人的代码,唯一奇怪的是:
  - Sock.send(url+'\r\n') 为什么不能少 '\r\n' ?? 否则死也!

> 我最早是只使用\n就出现你所说的问题。
> 后来看到transform返回时使用的是\r\n，
> 我想起来许多协议都要求行结束为\r\n，改过来就行了。
> 我想finger协议也是如此。使用\r\n而不是\\n应该是标准了。
> ======= 2004-07-24 22:57:48 您在来信中写道: =======

---Limodou

PyTwisted/tasteTwisted (2004-08-09 16:07:07由219编辑)

- LawMe/2005-12-01

开始TwistedStudyRecord写作 ::-- LawMe [2005-12-01 01:17:25]

开始Twisted学习写作

目录

1. 啃嚼Twisted的初感

啃嚼快一星期了，不再痛苦难受，逐渐尝出twisted的香甜美味、柔顺可口，开始适应twisted的套路。

twisted的套路，有哪些显著特点呢？接下去说说我品尝出的滋味。

前面把twisted的套路概括成一句话，“一个中心，两个基本点”，现在就从这个“中心”聊起。

Twisted 官方说，“ Twisted is an event-driven networking framework ”。事实的确如此。从其运行机制上看，event 是 Twisted 运转的引擎，是发生各种动作的启动器，是牵一发而动全身的核心部件。从其架构组成上看，它是紧密围绕event设计的；它的具体应用application，主要是定义、实现各式各样的event，由此完成不同网络协议的连接和输入输出任务，满足用户的实际需求；从其application的文本形式上，可以直接看到，它的应用程序，基本上由一系列event构成。

由此可见，说它以event为中心，符合实际情况。

Twisted 对event 的管理机制，可划分为后台和前台两种形式。

后台的管理，是Twisted 框架的内在机制，自动运行，对程序员透明无须干预，在程序文本中不见其踪迹。

前台的管理，是Twisted 授权程序员，在程序文本中显式写码来实现。程序员的工作，主要是按照既定的方式，实现 event。我们所关心、所用到的，是这部分东西（API）。

Twisted 众多的 event，分门别类、层次有序。前台管理中，有两个特别的 object，一个叫 reactor ，另一个叫defered。特别之处，在于它俩起着“事件管理器”的作用。下面，说说它俩。

1.1. 一、统领全局的 reactor

在 Twisted 应用中，reactor 的任务是为程序运行建立必须的全局循环（event loop），所起的作用，相当于 Python 应用中的 MainLoop()。

reactor 的用法很简单，一般只用两个：reactor.run() 启动全局循环，reactor.stop() 停止全局循环（程序终止）。

如果程序中没有调用reactor.stop() 的语句，程序将处于死循环，可以按键 Ctrl-C 强制退出。

下面是一个例子：

Toggle line numbers

```
1 from twisted.internet import reactor
2 import time
3 def printTime( ):
4     print "Current time is", time.strftime("%H:%M:%S")
5
```

```
def stopReactor( ):
    print "Stopping reactor"
    reactor.stop( )

reactor.callLater(1, printTime)
#定时器，1秒钟后调用printTime()

reactor.callLater(2, printTime)

reactor.callLater(3, printTime)

reactor.callLater(5, stopReactor)
#定时器，5秒钟后调用stopReactor()

print "Running the reactor..."

reactor.run( )

print "Reactor stopped."
```

1.2. 二、提升效率的 deferred

Twisted 官方称，“Twisted is event-based, asynchronous framework ”。这个“异步”功能的代表就是 deferred。

deferred 的作用类似于“多线程”，负责保障多头连接、多项任务的异步执行。

当然，deferred “异步”功能的实现，与多线程完全不同，具有以下特点：

1 . deferred 产生的 event，是函数调用返回的对象；

2 . deferred 代表一个连接任务，负责报告任务执行的延迟情况和最终结果；

3 . 对deferred 的操作，通过预定的“事件响应器”（event handler）进行。

有了deferred，即可对任务的执行进行管理控制。防止程序的运行，由于等待某项任务的完成而陷入阻塞停滞，提高整体运行的效率。

请看下面的例子：

建议只关注**有特殊注释**的语句，它们反映了deferred的用法。涉及的两个class，是Twisted建立网络连接的固定套路，后面会专门说它。

Toggle line numbers

```
# connectiontest.py
from twisted.internet import reactor, defer, protocol

class CallbackAndDisconnectProtocol(protocol.Protocol):
# Twisted建立网络连接的固定套路

    def connectionMade(self):

        self.factory.deferred.callback("Connected!")
        # “事件响应器”handleSuccess对此事件作出处理

        self.transport.loseConnection( )
```

```
13
14
15 class ConnectionTestFactory(protocol.ClientFactory):
16 # Twisted建立网络连接的固定套路
17
18     protocol = CallbackAndDisconnectProtocol
19
20     def __init__(self):
21
22         self.deferred = defer.Deferred( )
23         # 报告发生了延迟事件，防止程序阻塞在这个任务上
24
25     def clientConnectionFailed(self, connector, reason):
26
27         self.deferred.errback(reason)
28         # "事件响应器"handleFailure对此事件作出处理
29
30 def testConnect(host, port):
31
32     testFactory = ConnectionTestFactory( )
33
34     reactor.connectTCP(host, port, testFactory)
35
36     return testFactory.deferred
37     # 返回连接任务的deferred
38
39
40 def handleSuccess(result, port):
41 # deferred"事件响应器": 连接任务完成的处理
42
43     print "Connected to port %i" % port
44
45     reactor.stop( )
46
47
48 def handleFailure(failure, port):
49 # deferred"事件响应器": 连接任务失败的处理
50
51     print "Error connecting to port %i: %s" % (
52
53     port, failure.getErrorMessage( ))
54
55     reactor.stop( )
56
57
58 if __name__ == "__main__":
59
60     import sys
61
62     if not len(sys.argv) == 3:
```

```
63
64      print "Usage: connectiontest.py host port"
65
66      sys.exit(1)
67
68
69      host = sys.argv[1]
70
71      port = int(sys.argv[2])
72
73      connecting = testConnect(host, port)
74      # 调用函数，返回deferred
75
76      connecting.addCallback(handleSuccess, port)
77      # 建立deferred"事件响应器"
78
79      connecting.addErrback(handleFailure, port)
80      # 建立deferred"事件响应器"
81
82      reactor.run( )
```

LawMe/2005-12-01 (2005-12-16 00:59:26由riverfor编辑)

- PyTwisted/UnderstandReactor

关于reactor的内部机制(浅析) -- dreamingk [2004-08-09 02:21:09]

刚才向hd讨教了一些关于reactor的问题，受益非浅，这里把它写下来。

目录

1. 精髓描述

reactor内部有一个event loop, 根据事件的类型，
调用不同的事件响应函数，这些函数是事先注册的callback object.

1.1. 详细

::

切换行号显示

```
   1 register_event(event1, fun1)   #这里简化了，twisted是用继承类，重载类函数来完成的
   2 register_event(event2, fun2)
   3
   4 #event_loop:
   5 while(True):
   6     event = get_event(); #从事件队列里面取事件
   7     switch(event):
   8         case event1:
   9             fun1()
  10         case event2:
  11             fun2()
  12     ....
```

- 这里调用fun1,fun2是阻塞调用的，就是说只有fun1执行完才能接着下一次loop，这样如果fun1执行的时间太长，

事件就会阻塞在事件队列里面。

而实际项目中，fun1,fun2很难保证能马上返回，可能要查询数据库什么的，因此这里必须想办法，比如多线程什么的，而twisted提供了deffered对象来提供帮助。

在Cpython中,reactor对网络数据的读写是用select实现的,每个需要回调的对象都是abstract.FileDescriptor派生类的实例。该对象
实现了fileno方法,以返回fd供select用,还要实现了doWrite和doRead方法供回调。该对象的startReading,startWriting方法会将回调的
对象加入到reactor的读写队列中(实际在reactor中是dict)。当reactor的主循环工作时,它首先去检查那些用Calllator登记过的函数是否到
了时间,如到了就运行它们,然后调用select,并将读写队列中的对象传递给它,此外还需根据CallLater函数队列中的时间设置一个合适的超时。
当select返回时,再根据返回的结果调用相应对象的doRead,doWrite方法。
此外，如上面提到的，CallLater函数可以将函数登记到回调队列中。

---0.706

1.2. 后记体会

这里其实和UI框架中的主事件循环很象，只是各种框架的响应事件的机制不同，MFC是用映射表，Java的swing使用interface,跟twisted有些象，qt使用的是信号/信号槽机制，各有利弊吧。

这些道理其实很简单，俺只是以为twisted为每个连接开一线程呢，所以不知道为什么要用deffered。

PyTwisted/UnderstandReactor (2004-09-16 20:45:16由218编辑)

Deferreds are beautiful! (A Tutorial)

Introduction

Deferreds are quite possibly the single most confusing topic that a newcomer to Twisted has to deal with. I am going to forgo the normal talk about what deferreds are, what they aren't, and why they're used in Twisted. Instead, I'm going show you the logic behind what they **do**.

对Twisted的新手来说，deferred或许是最令他们感到困惑的东西。我准备先把“它是什么”， “不是什么”以及“为什么Twisted必须要用它”之类的普通问题先放一放。这里我先给你介绍一下它的工作原理。

A deferred allows you to encapsulate the logic that you'd normally use to make a series of function calls after receiving a result into a single object. In the examples that follow, I'll first show you what's going to go on behind the scenes in the deferred chain, then show you the deferred API calls that set up that chain. All of these examples are runnable code, so feel free to play around with them.

deferred的作用是，能让你把通常情况下，当一个对象收到一个结果之后会产生一系列方法调用的处理逻辑封装起来。下面我们会举一些例子，我会告诉你在deferred链背后究竟发生了些什么。此外我还会演示怎样用deferred API来构建这个链。这些代码都能运行，所以尽管试吧。

A simple example

First, a simple example so that we have something to talk about:

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
here we have the simplest case, a single callback and a single errback
"""

num = 0
def handleFailure(f):
    print "errback"
```

```
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    global num; num += 1
    print "callback %s" % (num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"


def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    global num; num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex.py

And the output: (since both methods in the example produce the same output, it will only be shown once.)

输出如下：(鉴于两次调用的输出是相同的，这里就只打印一次了。)

```
callback 1
        got result: success
```

Here we have the simplest case. A deferred with a single callback and a single errback. Normally, a function would create a deferred and hand it back to you when you request an operation that needs to wait for an event for completion. The object you called then does `d.callback(result)` when the results are in.

这是最简单的例子。只带一个callback和一个errback的defrred。通常你会建一个会返回deferred的函数，然后当你发出一个需要等一段时间才能结束的请求时，你就可以调用这个方法会获取一个deferred。等到结果出来了，你再用d.callback(result)。

The thing to notice is that there is only one result that is passed from method to method, and that the result returned from a method is the argument to the next method in the chain. In case of an exception, result is set to an instance of `Failure` that describes the exception.

值得注意的是，所有方法都只能传一个参数，在这个链里前一个方法的返回值就是后一个方法的参数。万一碰到异常，twisted就会把返回值设成相应的Failure实例。

Errbacks

Failure in requested operation

操作请求失败了

Things don't always go as planned, and sometimes the function that returned the deferred needs to alert the callback chain that an error has occurred.

事情不会总是按着计划走，有时你得在函数里设计好，万一发生了错误，callback链该做什么调整。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
this example is analagous to a function calling .errback(failure)
"""


class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"


def behindTheScenes(result):
    if not isinstance(result, failure.Failure): # ---- callback
        try:
```

```
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


def deferredExample(result):
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)

    d.errback(result)


if __name__ == '__main__':
    result = None
    try:
        raise RuntimeError, "*doh*! failure!"
    except:
        result = failure.Failure()
    behindTheScenes(result)
    print "\n---------------------------------------------------\n"
    Counter.num = 0
    deferredExample(result)
```

Source listing - ../listings/deferred/deferred_ex1a.py

```
errback
we got an exception: Traceback (most recent call last):
--- exception caught here ---
  File "deferred_ex1a.py", line 73, in ?
    raise RuntimeError, "*doh*! failure!"
exceptions.RuntimeError: *doh*! failure!
```

The important thing to note (as it will come up again in later examples) is that the callback isn't touched, the failure goes right to the errback. Also note that the errback trap()s the expected exception type. If you don't trap the exception, an error will be logged when the deferred is garbage-collected.

值得注意的是(因为在后面的例子中还会碰到)，callback根本没机会发言，failure直接走到errback那里去了。同样errback trap()(捕捉到了)它所希望得到的异常。如果你没有捕获异常，那么当垃圾回收器回收deferred的时候，它就会往日志里写一条error记录。

Exceptions raised in callbacks

callback抛出了异常

Now let's see what happens when *our callback* raises an exception

现在我们来看看当*callback*抛出异常的时候会发生些什么。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
here we have a slightly more involved case. The deferred is called back w
result. the first callback returns a value, the second callback, however
raises an exception, which is handled by the errback.
"""


class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"


def behindTheScenes(result):
    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
        try:
            result = handleFailure(result)
```

```
        except:
            result = failure.Failure()


def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex1b.py

And the output: (note, tracebacks will be edited slightly to conserve space)

```
callback 1
        got result: success
callback 2
        got result: yay! handleResult was successful!
        about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py",
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex1.py", line 32, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error
```

If your callback raises an exception, the next method to be called will be the next errback in your chain.

如果callback抛出了异常，那么下一个调用的将是跟在这个callback之后的第一个errback。

Exceptions will only be handled by errbacks

只有errback才会去管Exception

If a callback raises an exception the next method to be called will be next errback in the chain. If the chain is started off with a failure, the first method to be called will be the first errback.

如果callback抛出了异常，那么下一个调用的将是跟在这个callabck后面的第一个errback。如果callback链一开始就出了错，那么它就会调用第一个errback。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
this example shows an important concept that many deferred newbies
```

```
(myself included) have trouble understanding.

when an error occurs in a callback, the first errback after the error
occurs will be the next method called. (in the next example we'll
see what happens in the 'chain' after an errback)

"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"



def behindTheScenes(result):
    # equivalent to d.callback(result)

    # now, let's make the error happen in the first callback

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    # note: this callback will be skipped because
    # result is a failure

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
```

```
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()



def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex2.py

```
callback 1
        got result: success
        about to raise exception
errback
we got an exception: Traceback (most recent call last):
  File "./deferred_ex2.py", line 85, in ?
    nonDeferredExample("success")
--- <exception caught here> ---
  File "./deferred_ex2.py", line 46, in nonDeferredExample
    result = failAtHandlingResult(result)
  File "./deferred_ex2.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error
```

You can see that our second callback, handleResult was not called because failAtHandlingResult raised an exception

Handling an exception and continuing on

处理异常，然后继续

In this example, we see an errback handle an exception raised in the preceeding callback. Take note that it could just as easily been an exception from **any other** preceeding method. You'll see that after the exception is handled in the errback (i.e. the errback does not return a failure or raise an exception) the chain continues on with the next callback.

在这个例子里，errback将会处理由上一个callback所抛出的一样。其实这个异常可以是**任意**一个callback所产生的。你会发现当errback处理完异常之后(比方说errback不再返回failure或者抛出异常)，callback链会继续下一个callback。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util
```

```
"""
now we see how an errback can handle errors. if an errback
does not raise an exception, the next callback in the chain
will be called

"""

class Counter(object):
    num = 0


def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def callbackAfterErrback(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)



def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
```

```
    else:                                           # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = callbackAfterErrback(result)
        except:
            result = failure.Failure()
    else:                                           # ---- errback
        pass



def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addErrback(handleFailure)
    d.addCallback(callbackAfterErrback)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex3.py

```
callback 1
        got result: success
        about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py",
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex2.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error
```

addBoth: the deferred version of *finally*

addBoth：deferred版的*finally*

Now we see how deferreds do **finally**, with .addBoth. The callback that gets added as addBoth will be called if the result is a failure or non-failure. We'll also see in this example, that our doThisNoMatterWhat() method follows a common idiom in deferred callbacks by acting as a passthru, returning the value that it received to allow processing the chain to continue, but appearing transparent in terms of the result.

现在我们来看看deferred是怎样实现**finally**的，答案就是.addBoth。不管参数是不是failure，addBoth所加载的callback都会被调用。此外我们还会看的，doThisNoMatterWhat()方法以充当passthru的方式遵守了deferred的callback约定。它将收到的数据原封不动地发出去，这样处理链就能继续下去了。对数据来说，它是完全透明的。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
now we'll see what happens when you use 'addBoth'

"""

class Counter(object):
    num = 0


def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def doThisNoMatterWhat(arg):
    Counter.num += 1
    print "both %s" % (Counter.num,)
    print "\tgot argument %r" % (arg,)
    print "\tdoing something very important"
    # we pass the argument we received to the next phase here
    return arg



def behindTheScenes(result):
    # equivalent to d.callback(result)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass
```

```
    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    # ---- this is equivalent to addBoth(doThisNoMatterWhat)

    if not isinstance(result, failure.Failure):
        try:
            result = doThisNoMatterWhat(result)
        except:
            result = failure.Failure()
    else:
        try:
            result = doThisNoMatterWhat(result)
        except:
            result = failure.Failure()


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


def deferredExample():
    d = defer.Deferred()
    d.addCallback(handleResult)
    d.addCallback(failAtHandlingResult)
    d.addBoth(doThisNoMatterWhat)
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex4.py

```
callback 1
        got result: success
callback 2
        got result: yay! handleResult was successful!
        about to raise exception
both 3
        got argument <twisted.python.failure.Failure exceptions.RuntimeEr
        doing something very important
errback
we got an exception: Traceback (most recent call last):
```

```
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py",
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex4.py", line 32, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error
```

You can see that the errback is called, (and consequently, the failure is trapped). This is because doThisNoMatterWhat method returned the value it received, a failure.

addCallbacks: decision making based on previous success or failure

addCallbacks: 根据前面的运行结果加载callback

As we've been seeing in the examples, the callback is a pair of callback/errback. Using addCallback or addErrback is actually a special case where one of the pair is a pass statement. If you want to make a decision based on whether or not the previous result in the chain was a failure or not (which is very rare, but included here for completeness), you use addCallbacks. Note that this is **not** the same thing as an addCallback followed by an addErrback.

正如我们前面所讲的，回调函数是一组callback/errback。无论是addCallback还是addErrback其实都是特例，因为在加载一个函数的同时，我们也pass了另一个另一个函数。如果你想根据前一次调用是否成功来判断该加载哪个函数(这种情况十分罕见，不过出于完整性的考虑，twisted也提供了)，那么可以使用addCallbacks。注意，这和addCallback之后再addErrback是两码事

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
now comes the more nuanced addCallbacks, which allows us to make a
yes/no (branching) decision based on whether the result at a given point
a failure or not.

"""

class Counter(object):
    num = 0


def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
```

```
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def yesDecision(result):
    Counter.num += 1
    print "yes decision %s" % (Counter.num,)
    print "\twasn't a failure, so we can plow ahead"
    return "go ahead!"

def noDecision(result):
    Counter.num += 1
    result.trap(RuntimeError)
    print "no decision %s" % (Counter.num,)
    print "\t*doh*! a failure! quick! damage control!"
    return "damage control successful!"



def behindTheScenes(result):

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = failAtHandlingResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    # this is equivalent to addCallbacks(yesDecision, noDecision)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = yesDecision(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        try:
            result = noDecision(result)
        except:
            result = failure.Failure()


    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                       # ---- errback
        pass


    # this is equivalent to addCallbacks(yesDecision, noDecision)

    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = yesDecision(result)
        except:
            result = failure.Failure()
```

```
    else:                                           # ---- errback
        try:
            result = noDecision(result)
        except:
            result = failure.Failure()


    if not isinstance(result, failure.Failure): # ---- callback
        try:
            result = handleResult(result)
        except:
            result = failure.Failure()
    else:                                           # ---- errback
        pass


    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                           # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)
    d.addCallbacks(yesDecision, noDecision) # noDecision will be called
    d.addCallback(handleResult) # - A -
    d.addCallbacks(yesDecision, noDecision) # yesDecision will be called
    d.addCallback(handleResult)
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex5.py

```
callback 1
        got result: success
        about to raise exception
no decision 2
        *doh*! a failure! quick! damage control!
callback 3
        got result: damage control successful!
yes decision 4
        wasn't a failure, so we can plow ahead
callback 5
        got result: go ahead!
```

Notice that our errback is never called. The noDecision method returns a non-failure so processing continues with the next callback. If we wanted to skip the callback at "- A -" because of the error, but do some kind of processing in response to the error, we would have used a

passthru, and returned the failure we received, as we see in this next example:

注意，我们的errback一直没被调用。noDecision方法返回了一个非failure的值，所以处理链接下来调用的是callback。如果你想跳过位于"- A -"的callback，但又不想放过错误，那么你就得用passthru把收到的failure再发出去。这就是下一段例程所演示的：

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
now comes the more nuanced addCallbacks, which allows us to make a
yes/no (branching) decision based on whether the result at a given point
a failure or not.

here, we return the failure from noDecisionPassthru, the errback argument
the first addCallbacks method invocation, and see what happens

"""

class Counter(object):
    num = 0


def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)
    return "okay, continue on"

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def yesDecision(result):
    Counter.num += 1
    print "yes decision %s" % (Counter.num,)
    print "\twasn't a failure, so we can plow ahead"
    return "go ahead!"

def noDecision(result):
    Counter.num += 1
    result.trap(RuntimeError)
    print "no decision %s" % (Counter.num,)
    print "\t*doh*! a failure! quick! damage control!"
    return "damage control successful!"

def noDecisionPassthru(result):
    Counter.num += 1
    print "no decision %s" % (Counter.num,)
```

```
        print "\t*doh*! a failure! don't know what to do, returning failure!"
        return result


    def behindTheScenes(result):

        if not isinstance(result, failure.Failure): # ---- callback
            try:
                result = failAtHandlingResult(result)
            except:
                result = failure.Failure()
        else:                                       # ---- errback
            pass


        # this is equivalent to addCallbacks(yesDecision, noDecision)

        if not isinstance(result, failure.Failure): # ---- callback
            try:
                result = yesDecision(result)
            except:
                result = failure.Failure()
        else:                                       # ---- errback
            try:
                result = noDecisionPassthru(result)
            except:
                result = failure.Failure()


        if not isinstance(result, failure.Failure): # ---- callback
            try:
                result = handleResult(result)
            except:
                result = failure.Failure()
        else:                                       # ---- errback
            pass


        # this is equivalent to addCallbacks(yesDecision, noDecision)

        if not isinstance(result, failure.Failure): # ---- callback
            try:
                result = yesDecision(result)
            except:
                result = failure.Failure()
        else:                                       # ---- errback
            try:
                result = noDecision(result)
            except:
                result = failure.Failure()


        if not isinstance(result, failure.Failure): # ---- callback
            try:
                result = handleResult(result)
            except:
                result = failure.Failure()
        else:                                       # ---- errback
            pass
```

```
    if not isinstance(result, failure.Failure): # ---- callback
        pass
    else:                                       # ---- errback
        try:
            result = handleFailure(result)
        except:
            result = failure.Failure()


def deferredExample():
    d = defer.Deferred()
    d.addCallback(failAtHandlingResult)

    # noDecisionPassthru will be called
    d.addCallbacks(yesDecision, noDecisionPassthru)
    d.addCallback(handleResult) # - A -

    # noDecision will be called
    d.addCallbacks(yesDecision, noDecision)
    d.addCallback(handleResult) # - B -
    d.addErrback(handleFailure)

    d.callback("success")


if __name__ == '__main__':
    behindTheScenes("success")
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex6.py

```
callback 1
        got result: success
        about to raise exception
no decision 2
        *doh*! a failure! don't know what to do, returning failure!
no decision 3
        *doh*! a failure! quick! damage control!
callback 4
        got result: damage control successful!
```

Two things to note here. First, "- A -" was skipped, like we wanted it to, and the second thing is that after "- A -", noDecision is called, because **it is the next errback that exists in the chain**. It returns a non-failure, so processing continues with the next callback at "- B -", and the errback at the end of the chain is never called

有两点值得注意。第一，正如我们所希望的，它跳过了“- A -”，第二，跳过“— A —”之后它调用了noDecision，因为**这是链中的下一个errback**。这个errback返回的是一个非failure的值，所以接下来执行的是“— B —”的callback，而最有一个errback是不可能被调到的。

Hints, tips, common mistakes, and miscellaney

提示，技巧，常见错误及其它

The deferred callback chain is stateful

deffered的callback链是带状态的

A deferred that has been called back will call it's addCallback and addErrback methods as appropriate in the order they are added, when they are added. So we see in the following example, deferredExample1 and deferredExample2 are equivalent. The first sets up the processing chain beforehand and then executes it, the other executes the chain as it is being constructed. This is because deferreds are *stateful*.

deferred的回调顺序就是addCallback和addErrback的顺序。所以我们看下面这个例子，deferredExample1和deferredExample2是一样的。第一个先建处理链再调用，第二个则一边建一边调用。这是因为deferred是*带状态的*。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
The deferred callback chain is stateful, and can be executed before
or after all callbacks have been added to the chain
"""

class Counter(object):
    num = 0

def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    f.trap(RuntimeError)

def handleResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    return "yay! handleResult was successful!"

def failAtHandlingResult(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    print "\tabout to raise exception"
    raise RuntimeError, "whoops! we encountered an error"

def deferredExample1():
    # this is another common idiom, since all add* methods
    # return the deferred instance, you can just chain your
    # calls to addCallback and addErrback

    d = defer.Deferred().addCallback(failAtHandlingResult
                        ).addCallback(handleResult
                        ).addErrback(handleFailure)

    d.callback("success")

def deferredExample2():
    d = defer.Deferred()

    d.callback("success")

    d.addCallback(failAtHandlingResult)
```

```
    d.addCallback(handleResult)
    d.addErrback(handleFailure)


if __name__ == '__main__':
    deferredExample1()
    print "\n--------------------------------------------------\n"
    Counter.num = 0
    deferredExample2()
```

Source listing - ../listings/deferred/deferred_ex7.py

```
callback 1
        got result: success
        about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py",
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex7.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error


--------------------------------------------------

callback 1
        got result: success
        about to raise exception
errback
we got an exception: Traceback (most recent call last):
--- <exception caught here> ---
  File "/home/slyphon/Projects/Twisted/trunk/twisted/internet/defer.py",
326, in _runCallbacks
    self.result = callback(self.result, *args, **kw)
  File "./deferred_ex7.py", line 35, in failAtHandlingResult
    raise RuntimeError, "whoops! we encountered an error"
exceptions.RuntimeError: whoops! we encountered an error
```

This example also shows you the common idiom of chaining calls to addCallback and addErrback.

Don't call .callback() on deferreds you didn't create!

别去调用还没建好的deferred的.callback()方法

It is an error to reinvoke deferreds callback or errback method, therefore if you didn't create a deferred, **do not under any circumstances** call its callback or errback. doing so will raise an exception

重复调用deferred的callback或errback办法是一种错误，所以如果你还没建好deferred，那就**别去调用**它的callback或errback，这么做只会导致错误。

Callbacks can return deferreds

callback能返回deferred

If you need to call a method that returns a deferred within your callback chain, just return that deferred, and the result of the secondary deferred's processing chain will become the result that gets passed to the next callback of the primary deferreds processing chain

如果callback链里的方法会返回deferred，那就让它返回deferred。这个deferred的处理结果会被当作参数传给当前这个deferred的下一个callback。

```
#!/usr/bin/python2.3

from twisted.internet import defer
from twisted.python import failure, util

"""
"""

class Counter(object):
    num = 0
    let = 'a'

    def incrLet(cls):
        cls.let = chr(ord(cls.let) + 1)
    incrLet = classmethod(incrLet)


def handleFailure(f):
    print "errback"
    print "we got an exception: %s" % (f.getTraceback(),)
    return f

def subCb_B(result):
    print "sub-callback %s" % (Counter.let,)
    Counter.incrLet()
    s = " beautiful!"
    print "\tadding %r to result" % (s,)
    result += s
    return result

def subCb_A(result):
    print "sub-callback %s" % (Counter.let,)
    Counter.incrLet()
    s = " are "
    print "\tadding %r to result" % (s,)
    result += s
    return result

def mainCb_1(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
    result += " Deferreds "

    d = defer.Deferred().addCallback(subCb_A
                        ).addCallback(subCb_B)
    d.callback(result)
    return d

def mainCb_2(result):
    Counter.num += 1
    print "callback %s" % (Counter.num,)
    print "\tgot result: %s" % (result,)
```

```
def deferredExample():
    d = defer.Deferred().addCallback(mainCb_1
                        ).addCallback(mainCb_2)

    d.callback("I hope you'll agree: ")


if __name__ == '__main__':
    deferredExample()
```

Source listing - ../listings/deferred/deferred_ex8.py

```
callback 1
        got result: I hope you'll agree:
sub-callback a
        adding ' are ' to result
sub-callback b
        adding ' beautiful!' to result
callback 2
        got result: I hope you'll agree:  Deferreds  are  beautiful!
```

Conclusion

结论

Deferreds can be confusing, but only because they're so elegant and simple. There is a lot of logical power that can expressed with a deferred's processing chain, and once you see what's going on behind the curtain, it's a lot easier to understand how to make use of what deferreds have to offer.

Index

Version: 2.0.0

Twisted 入门教程精解（一）

HD （ hdcola@gmail.com）

http://220.248.2.35:7080/moin/twistedTUT01

备注:

拒绝连接

- reactor （反应器）
  - twisted 使用 reactor 来进行事件调度
  - 事件调度的基础是使用 select 或是 poll 来进行 socket 操作
  - reactor 会有多种实现方式，不同的实现方式适应于不同的操作系统和不同的处理机制（线程 / 进程）

备注:

reactor 精解

- 一般可见的 socket 服务方法
  - 阻塞等待请求
  - 每个请求对应一个新的线 / 进程
- twisted 使用的 select/poll
  - 所有的连接放在一个调度器中，使用事件的方式通知应用程序
  - 使用一个线 / 进程进行数据发送 / 接收处理
- 参考 http://www.twistedmatrix.com/documents/current/

备注:

reactor 在哪？

- twisted.internet.interfaces.IReactor*
  - 怎么发现的
  - http://blog.huangdong.com/comments.php?id=150_0_1_10_C
  - 怎么使用 select 和 poll
  - http://blog.huangdong.com/comments.php?id=
  - 有多少 reactor 可以用
  - http://blog.huangdong.com/comments.php?id=153_0_1_0_C

备注：

什么也不做

- 一个完整的 server 需要了解的基础类
  - reactor：进行事件调度
  - ServerFactory 的子类：进行协议的初始化
  - Protocol 的子类：进行 socket 事件的响应
- 服务器初始化流程

reactor->ServerFactory->Protocol

备注:

断开连接

- Protocol 抽像类提供了所有的事件接口
  - 继承 Protocol 的事件方法就等同于响应事件
  - 缺省是不做任何工作
- 事件的响应流程

reactor->Protocol

- Protocol 中的 transport 是用来进行 socket 流操作的主体对象

备注:

读用户名，断开连接

- twisted 提供了一些常用的 Protocol 实现
  - NetstringReceiver/LineOnlyReceiver/LineReceiver....
  - 如果可行， Protocol 是最应该在设计时进行分层以减少代码量和出错可能的层次
- 可继承的方法的意义
  - 可以进行响应的事件

备注:

transport 在哪里？

- twisted.internet.interfaces.I*Transport
  - 依据你建立的协议来寻找
  - reactor.listenTCP 的提示

备注：

从一个空的工厂 (Factory) 中输出

- ServerFactory 的作用二
  - 为各个连接、各个事件的处理共享数据
- Protocol 中使用共享数据的方式
  - Protocol.factory 就是初始化 Protocol 的 Factory

备注:

从一个非空的工厂 (Factory) 中输出

- Factory 的方法都可以重载
- __init__ 的重载可以加入系统初始化的参数

备注:

使用 Deferreds

- 如果每个处理都会使用尽量久的时间，就会发生所有用户的响应缓慢
- Deferreds 将事物的处理放入 reactor 的事件调度器中统一调度
- 使用方法的 addErrback 和 addCallback 加入回调时的事务处理

备注:

为什么使用 Deferreds

- 使用线 / 进程
  - 启动需要更久的时间，而操作可能会比启动的时间短
  - 使用操作系统的调度
  - 需要更多的代码
- 使用 Deferreds
  - 不用启动，直接运行。只是在调度器中注册回调就可以
  - 使用 reactor 的调度
  - 加入两个事件的响应

备注:

在本地运行 'finger'

- utils.getProcessOutput 是一个进程调用的包装
- 包装的重要特点：
  - Deferred 处理
  - 进程的输出处理

备注：

从 web 上读取信息

- 程序不可运行
  - http://livejournal.com/~不存在了
- 使用 twisted.web.client 进行 HTTP 协议的客户机操作

备注:

使用应用程序 (Application) 对象

- reactor 由 internet 来取代
- application 形成了 reactor 的容器
- application 中设置运行时的所属用户和组

- 注意：
  - 不再由 python 运行应用，而是使用 twisted 来启动应用
  - 用户和组 ID 只在 Unix 下有效，用户对应的 ID 见 /etc/passwd，组 ID 见 /etc/group
  - 文件后嫒名： tac 和 py 都可以
  - 在 Unix 下非 root 用户不可启动 1024 端口以内的侦听

备注：

twistd

- twisted 命令行工具实现了守护进程的包装
- 包装的内容
  - stdin/stdout/stderr 的定向管理
  - 守护进程的后台运行机制
  - 守护进程的用户和组权限
  - 运行时的目录
  - 启动时刻确定所使用的 reactor
  - chroot 方式运行
- 使用说明　twisted --help

备注:

提示

- 教程中会有许多没见过的类和名词怎么办？
  - 按照教程中的说明先进行，适当的时候来回味，会明白很多。
- 为什么有 Factory 这样的命名？
  - 设计模式中工厂是一种对象生成机制，有空了看看这部分内容，很不错的
- 为什么有 lambda 这样难以使用的语言呀
  - Python 的 FP （函数编程）是一个非常需要理解的语法模式，但是能灵活使用却实也是不错的手法。

备注:

提问时间

- 你可以：
  - 喝茶
  - 起立跳跳
- 当想起问题时发问
  - 注意需要一个个的来
- 你也可以回答的哟

备注：

资源提示 - 开发工具

- Eclipse （ http://www.eclipse.org）是一个很好的开发工具（下载 3.0 版本）
- PyDev （ http://pydev.sourceforge.net）提供了在 Eclipse 下开发 Python 的功能
- 用 Eclipse 和 Ant 进行 Python 开发（ http://www-900.ibm.com/developerWorks/c ）

备注:

练习实例

- 完成一个服务器侦听 2020 端口
- 用户 telnet 上去后输入一个 url，服务器将这个 url 的信息抓回返回给用户
- 完成一个客户端，传入一个参数，将得到的信息显示出来

备注:

Twisted 入门教程精解
（二）

HD （ hdcola@gmail.com）

http://220.248.2.35:7080/moin/twistedTUT02

巧夺天工的设计，就在自然之间

备注：

让本地用户来设置信息

- serviceCollection 支持将多个服务进行统一的管理
- 数据的共享通过 Factory 之前的实例化顺序和互相引用实现

巧夺天工的设计，就在自然之间

备注:

用服务让彼此间的依赖健壮

- 使用 service.Service 再构造 Factory 的容器
- 使用 Service 形成 Factory 的持续数据共享

巧夺天工的设计，就在自然之间

备注:

读取状态文件

- reactor.callLater 的作用
  - 在指定的时间运行指定的函数
  - 放弃 cpu 的控制，让出时间片
  - 非阻赛式等待

巧夺天工的设计，就在自然之间

备注:

统一服务的结构

- Service
  - 存储 Factory，使 factory 间共享数据
- Factory
  - 存储 Protocol，使 protocol 处理时共享数据
- Protocol
  - 由 ractor 调度，进行网络事件的响应

巧夺天工的设计，就在自然之间

备注:

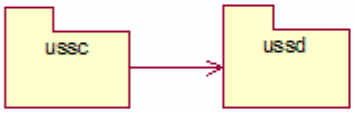
benchmarks0731a 代码精解

HD （ hdcola@gmail.com）

巧夺天工的设计，就在自然之间

备注:

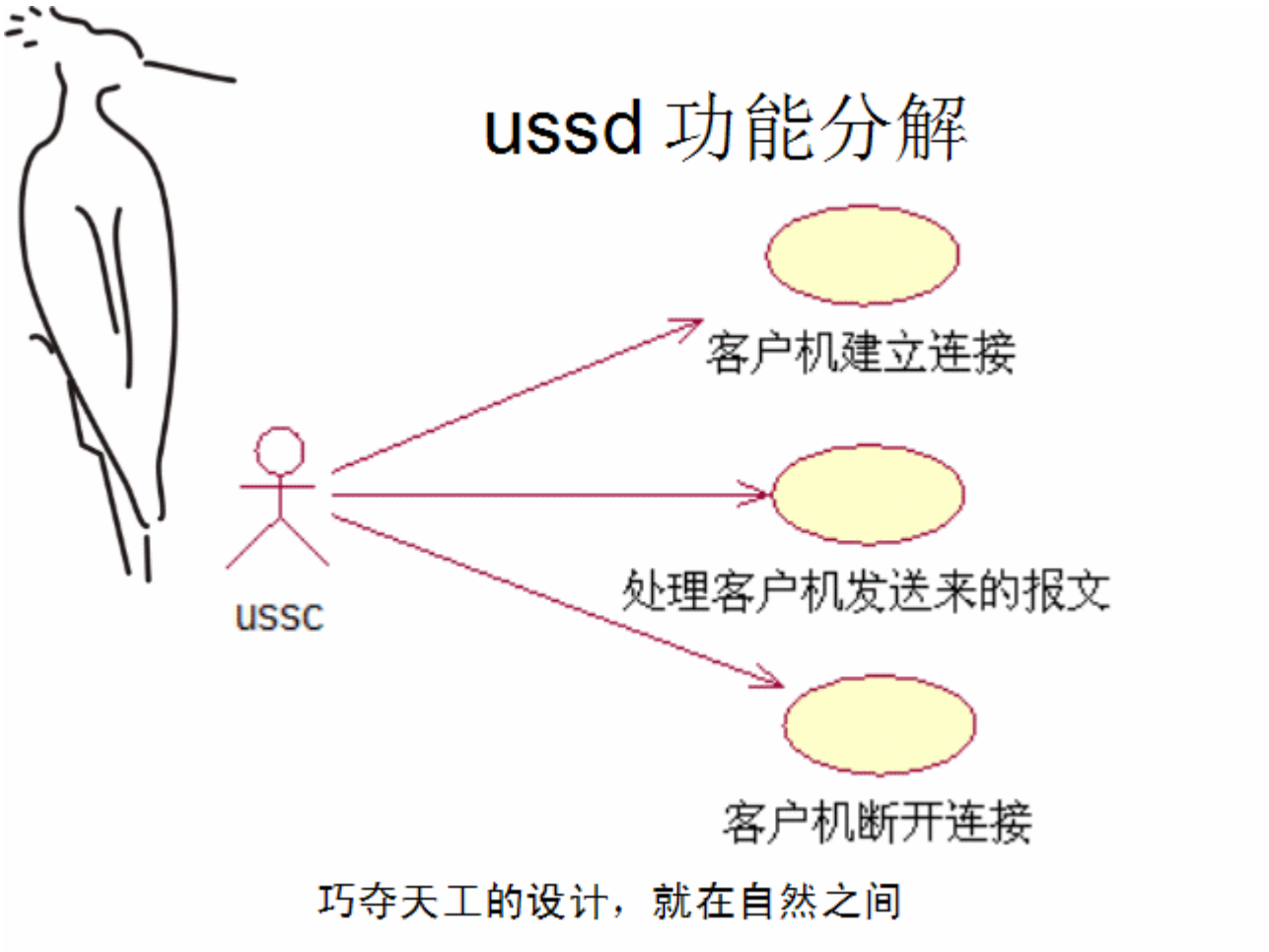
benchmarks 中的子系统



- ussc
  - 统一存储协议的模拟客户端
- ussd
  - 统一存储协议的模拟服务器端
- ussp
  - 二进制报文
  - 可并行处理的报文

巧夺天工的设计，就在自然之间

备注:


ussd 功能分解
客户机建立连接
处理客户机发送来的报文
客户机断开连接
USSC
巧夺天工的设计，就在自然之间

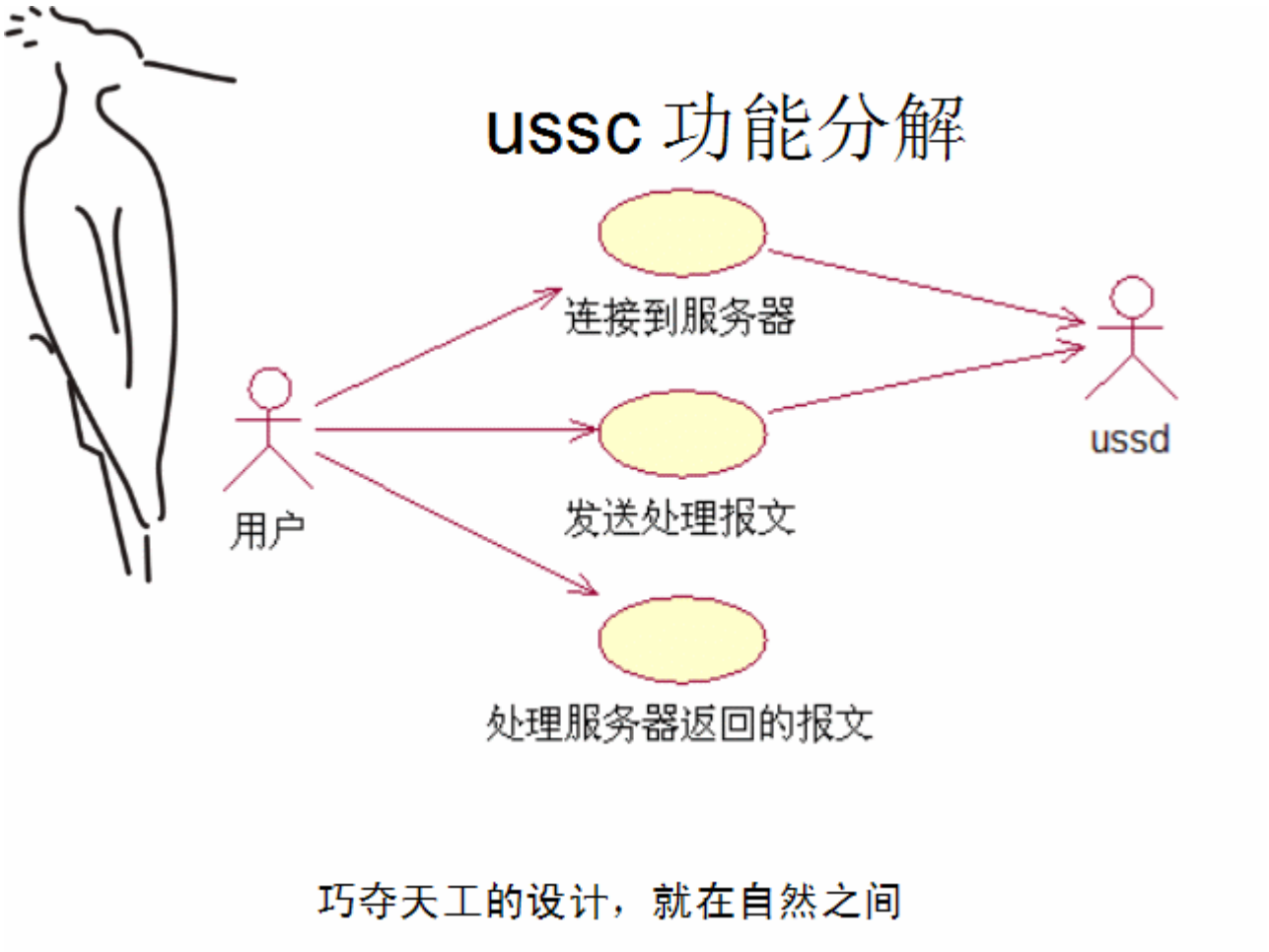
备注:

第一期要做到的目标

- 客户机连接
  - 不做处理
- 处理客户机发送来的报文
  - 解晰报文，形成待处理对象
  - 构造事件处理框架，形成可扩展、便于开发的结构
  - 将处理后的结果返回给客户机
- 客户机断开连接
  - 不做处理

巧夺天工的设计，就在自然之间

备注:



备注:

第一期要做到的目标

- 连接到服务器
  - 发送 connect 报文
- 发送处理报文
  - 在 connect_resp 处理中发送指定个数的报文
  - 统计发送的报文数
- 处理服务器返回的报文
  - 对返回的报文正确定判断
  - 统计返回的报文数

巧夺天工的设计，就在自然之间

备注: