Введение в асинхронное программирование и Twisted

Dave Peticolas
перевод Nina Evseenko
верстка Андрей Березовский

Содержание

1. С чего мы начнем вначале

Данная статья является переводом. Оригинал, который можно найти на странице Twisted Introduction, был написан Dave Peticolas.

1.1. Предисловие

В рассылке Twisted не так давно появился вопрос: можно ли где-нибудь найти описание, которое позволит быстро овладеть техникой использования Twisted? На данный момент такого описания не существует, и данное руководство им не является.

Если вы новичок в асинхронном программировании, то лучше изучать основные принципы постепенно. Исходя из продолжительного опыта использования Twisted и познания всех тонкостей и сложностей, данное пособие начинается с общего понятия модели асинхронного программирования. Большая часть кода Twisted понятна и хорошо написана, документация на сайте хорошая, по меньшей мере с точки зрения стандартов свободного ПО. Но без понятия модели, чтение кода Twisted или кода, использующего Twisted, или даже чтение документации, превратится в головную боль.

Таким образом, первые главы помогают осознать модель асинхронного программирования, а последующие - постичь особенности программирования с использованием Twisted. В самом начале мы совсем не будем использовать Twisted. Вместо этого, для иллюстрации функционирования асинхронной системы мы будем использовать простые программы на Python'е. И как только мы начнем использовать Twisted, мы начнем с очень низкого уровня, который вы не использовали бы в повседневном программировании. Twisted - высоко абстрактная система, и это дает вам огромное преимущество при решении проблем. Но, когда вы изучаете Twisted, в особенности, когда вы пытаетесь понять как Twisted реально работает, многоуровневые абстракции могут быть препятствием. Поэтому мы начнем с самых основ.

После того, как вы осознаете модель, вам будет проще читать документацию и исходный код Twisted. Так что - начнем.

1.2. Модели

Мы начнем с рассмотрения двух похожих моделей, для того чтобы сравнить с ними асинхронную модель. Для иллюстрации мы представим программу, которая состоит из трех концептуально отличных задач, которые должны быть выполнены, для того чтобы программа завершилась. Мы конкретизируем эти задачи позже, сейчас мы не будем говорить ничего о них, кроме того, что программа должна выполнить эти задачи. Заметьте, что термин “задача” используется не в техническом смысле: “нечто, что нужно выполнить”.

Первая модель, которую мы рассмотрим, - однопоточная синхронная модель, изображенная на рисунке 1 ниже:

Это самый простой стиль программирования. Одновременно выполняется только одна задача, следующая задача выполняется только после того, как предыдущая завершилась. И, если задачи всегда выполняются в определенном порядке, то запуск следующей задачи может осуществиться, если все предыдущие задачи завершились без ошибок.

Мы можем сравнить синхронную модель с многопоточной моделью, изображенной на рисунке 2:

В этой модели каждая задача выполняется в отдельном потоке. Потоки управляются операционной системой, и в случае наличия нескольких процессоров и/или ядер, могут выполняться параллельно, или их выполнение может чередоваться в случае одного

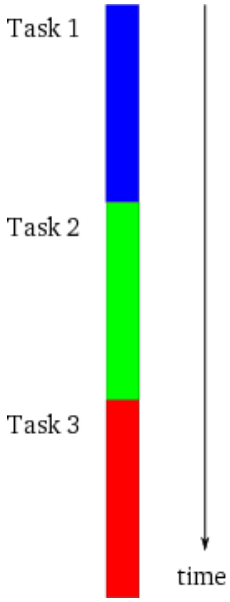

Рис. 1: Синхронная модель

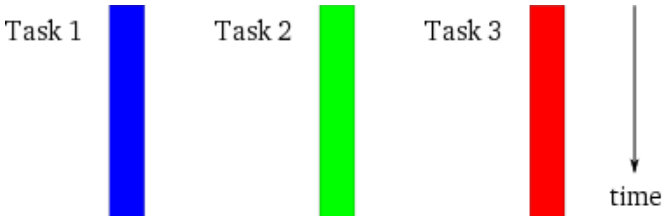

Рис. 2: Модель на основе потоков

процессора. Смысл в том, что в тредовой модели деталями управляет операционная система, и программист просто думает в терминах независимых потоков инструкций, которые могут выполняться одновременно. Хотя диаграмма простая, на практике потоковые программы могут быть достаточно сложными, поскольку нужно координировать потоки между собой. Потоковая координация и взаимодействие являются отдельной, достаточно сложной темой.

Некоторые программы реализуют параллельность, используя несколько процессов вместо нескольких потоков. Хотя, с точки зрения реализации есть отличия, для наших целей это одна и та же модель, как на рисунке 2.

Теперь мы можем перейти к асинхронной модели на рисунке 3:

В этой модели задачи выполняются поочередно в одном потоке. Это проще, чем в случае потоков, поскольку программист всегда знает, что когда одна задача выполняется, другая - нет. Хотя в однопроцессорной системе в потоковой программе задачи также будут выполняться поочередно, программист, использующий потоки, все равно должен думать в терминах рисунка 2, а не 3, чтобы программа работала корректно при переходе на многопроцессорную систему. Но однопоточная асинхронная система всегда будет чередовать выполнение задач даже на многопроцессорной системе.

Существует еще одно отличие между асинхронной и потоковой моделями. В потоковой модели решение приостановить один поток и выполнить другой в большей степени не регулируется программистом. Скорее, это контролируется операционной системой, и программист должен предполагать, что поток может приостановиться и смениться другим практически в любой момент времени. Что отличается от задач в асинхронной мо-

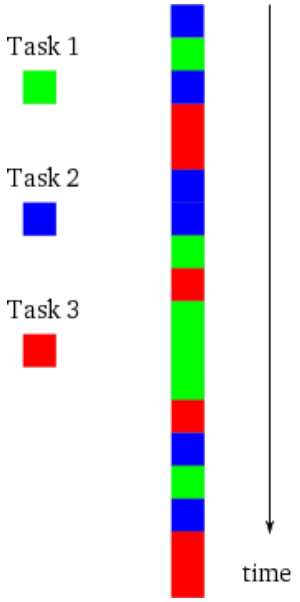

Рис. 3: Асинхронная модель

дели, которые выполняются до того момента, когда явно не передают контроль другим задачам. Это значительно проще, чем в случае потоков.

Заметим, что возможно смешивать асинхронные и потоковые модели, и использовать оба варианта в одной системе. Но, для большей части этого введения, мы будем придерживаться “чистых“ асинхронных систем с управлением в один поток.

1.3. Мотивация

Мы увидели, что асинхронная модель проще, чем потоковая, поскольку есть только единственный поток инструкций и задач, который явно передает управление, вместо приостановки в произвольный момент времени. Но асинхронная модель является явно более сложной, чем синхронная.

Программист должен организовать задачу как последовательность маленьких шагов, которые периодически запускаются.

И, если одна задача использует вывод другой, то зависящая задача должна быть написана так, чтобы уметь принимать на вход результат по порциям, вместо одного большого куска.

Поскольку здесь нет параллельности, что видно из наших диаграмм, то асинхронная программа будет выполняться также долго как и синхронная, возможно дольше, так как асинхронная программа может быть хуже с точки зрения локальности ссылок.

Так почему же мы бы выбрали асинхронную модель? Существует по меньшей мере две причины. Первая - если одна или более задач отвечают за реализацию интерфейса, то путем чередования задач, система сможет отвечать на пользовательский ввод даже тогда, когда в “фоне” выполняется другая задача. Так как фоновые задачи не могут выполниться быстрее, система будет более вежливой для того, кто ее использует. Однако, существуют условия, когда асинхронная система будет лучше по сравнению с синхронной, иногда намного, в смысле выполнения всех задач за более короткий промежуток времени. Эти условия связаны с задачами, которые вынуждены ожидать событий или блокироваться, как это изображено на рисунке 4:

На рисунке серые секции представляют собой периоды времени, когда определенная задача ожидает (блокируется) и не выполняется. Почему же задача может быть заблоки-

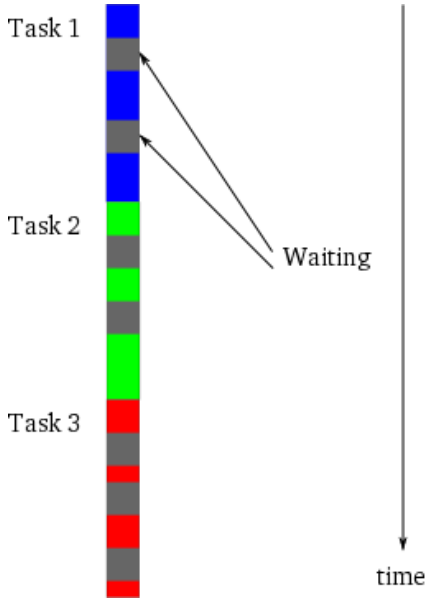

Рис. 4: Блокирование в синхронной программе

рована? Частая причина - ожидание выполнения ввода-вывода, связанное с перенесением данных из или во внешнее устройство. Обычный процессор может управлять переносом данных на порядок быстрее, по сравнению с максимальной скоростью переноса данных для диска или сети. Таким образом, синхронная программа, которая выполняет много операций ввода-вывода, будет часто блокироваться в момент обращения к диску или сети. Такая синхронная программа часто называется блокирующаяся программа (blocking program).

Заметьте, что на рисунке 4 изображена блокирующаяся программа, которая немного похожа на асинхронную программу на рисунке 3. Это не совпадение. Основная идея в асинхронной модели - это то, что, когда асинхронная программа сталкивается с тем, что блокирует синхронную программу, асинхронная будет выполняться. Асинхронная программа будет “блокироваться” только тогда, когда нет задач на выполнение, поэтому такая программа называется неблокирующейся (non-blocking program). И каждое переключение с одной задачи на другую соответствует тому, что первая задача или выполнилась, или находится в месте, где она заблокирована. С большим количеством потенциально блокирующихся задач, асинхронная программа может превосходить синхронную тем, что проводит в целом меньше времени ожидая.

По сравнению с синхронной моделью, асинхронная модель лучше когда:

1. Когда есть много задач таких, что почти всегда есть по меньшей мере одна задача, которая может выполняться.

2. Задачи выполняют много операций ввода-вывода, приводя к тому, что синхронная задача проводит много времени впустую, блокируясь, в то время когда другие задачи могли бы выполняться.

3. Задачи независимы друг от друга и не нуждаются во взаимодействии.

Эти условия почти полностью характеризуют типичный сетевой сервер (например, web сервер) в среде клиент-сервер. Каждая задача представляет собой один клиентский запрос с операциями ввода-вывода в форме получения запроса и отправления ответа. И клиентские запросы по большому счету независимы. Так что реализация сетевого сервера является подходящим кандидатом для асинхронной модели, поэтому Twisted - это одна из первых и передовых сетевых библиотек.

1.4. Дальше и больше

Это окончание первой части. Во второй части мы будем писать некоторые сетевые программы, обе - блокирующиеся и неблокирующиеся, настолько простые, насколько это возможно (без использования Twisted), для того, чтобы осознать как действительно работает асинхронная программа, написанная на Python’е.

2. Медленная поэзия и апокалипсис

На этот раз мы запачкаем наши руки и напишем некоторый код. Но сначала - некоторые предположения.

2.1. Предположения о навыках

Предполагается, что вы имеете некоторые навыки написания синхронных программ на Python'е и немного знаете о сокетном программировании на Python'е. Если вы никогда раньше не использовали сокеты, вы можете почитать документацию о Python-модуле socket, особенно примеры ближе к концу. Если вы никогда раньше не использовали Python, то все остальное вам, вероятно, покажется странным.

2.2. Предположения о компьютере

Примеры разрабатывались и отлаживались на Linux'е. Возможно, что код работает под другие Unix подобные системы (Mac OSX или FreeBSD).

Далее предполагается, что вы установили относительно свежые версии Python и Twisted. Примеры разрабатывались с использованием Python 2.5 и Twisted 8.2.0.

Вы можете запускать все примеры на одном компьютере или сконфигурировать их для запуска по сети на нескольких машинах. Но для изучения основных механизмов асинхронного программирования использование одной машины предпотчтительнее.

Код с примерами доступен ввиде zip или tar файла, или как git репозиторий. Если вы можете использовать git или другую систему контроля версий, которая может читать git репозиторий, то рекомендуется использовать этот метод, так как примеры время от времени обновляются, и в этом случае обновление будет осуществляться гораздо проще. Репозиторий помимо всего прочего включает исходники рисунков в формате SVG. Клонировать репозиторий можно следующим образом:

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

И последнее предположение: наличие нескольких консолей, открытых в директории с примерами (в одной из них предполагается открытым файл README).

2.3. Медленная поэзия

Хотя процессоры намного быстрее, чем сети, а большинство сетей все же намного быстрее, чем мозг, или по меньшей мере быстрее, чем скорость движения глаз. Сложно получить заметную задержку в сети, особенно, в случае одной машины и байтов, проносящихся со свистом на loopback интерфейсе. То что нам нужно - медленный сервер, с искуственными задержками, которые мы можем менять для изучения их воздействия. Поскольку серверы должны что-то обслуживать, то наш сервер будет обслуживать поэзию. В исходниках примеров находится директория poetry с поэзией авторов John Donne, W.B. Yeats, Edgar Allen Poe. Конечно же, вы можете использовать свои собственные поэмы.

Медленный поэтический сервер реализован в blocking-server/slowpoetry.py. Вы можете запустить пример сервера следующим образом:

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

Эта команда запустит блокирующий сервер с поэмой “Ecstasy” John Donne. Пойдем дальше и посмотрим на исходный код блокирующего сервера. Как можно заметить, в коде не используется Twisted, только основные сокетные операции. Сервер отправляет ограниченное количество байт за раз, с фиксированной временной задержкой между ними. По умолчанию, сервер отправляет 10 байт каждые 0.1 секунды, но вы можете поменять эти параметры с помощью опций командной строки *--num-bytes* и *--delay*. Например, чтобы отправлять 50 байт каждые 5 секунд, нужно запустить:

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

Когда сервер запускается, он печатает номер порта, который он слушает. По умолчанию, это первый доступный случайный порт на вашей машине. Запуская сервер с различными настройками, может понадобится использовать тот же порт, что и до этого, для того, чтобы не нужно было перенастраивать клиент. Вы можете задать определенный порт следующим образом:

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

Если у вас есть программа netcat (или nc), можно протестировать запущенный сервер следующим образом:

```
netcat localhost 10000
```

При запущенном сервере вы увидите медленно ползущую вниз по экрану поэму. Экстаз! Вы также заметите, что сервер печатает строки с количеством отосланных байт. Сразу после того, как поэма была отослана, сервер закрывает соединение.

По умолчанию, сервер слушает только локальный ”loopback” интерфейс. Если вы хотите получить доступ к серверу с другой машины, вы можете задать сетевой интерфейс через опцию –iface.

Сервер не только медленно отправляет поэму, по коду видно, что во время отправления сервером поэмы одному клиенту, другие клиенты должны ожидать окончания скачивания поэмы первым клиентом до того, как другие смогут получить хотя бы первую строку. Это реально медленный сервер и не особо полезен, кроме как в обучающих целях.

С другой стороны, если большинство пемиссемистичных людей из Peak Oil правы, и нашему миру грозит глобальный энергетический и социальный кризис, то, однажды, низкопропускной и маломощный сервер может стать именно тем, что нам нужно. Представьте, что после длительного дня забот о ваших садах, производстве собственной одежды, обслуживания Центрального Организационного Комитета Вашей комунны, борьбы с радиокативными зомби, бродящими по пост-апокалиптическим пустошам, вы сможете покрутить ваш генератор и скачать несколько строк высокой культуры из исчезнувшей цивилизации. Вот тогда наш сервер реально вступит в свои права.

2.4. Блокирующий клиент

Также в исходных кодах можно найти пример блокирующего клиента, который скачивает поэмы из серверов одну за другой. Давайте дадим нашему клиенту три задачи для выполнения так, как это изображено на рисунке 1. Сначала мы запустим три сервера, обслуживающих три различных поэмы:

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

Вы можете выбрать другие номера портов, если те, что выбраны выше, уже используются в вашей системе. Заметьте, что первый сервер использует chunk'и по 30 байт, вместо 10, что установлены по умолчанию, поскольку поэма, которую он обслуживает примерно в три раза длиньше остальных. При таких настройках скачивание каждой из них должно завершаться приблизительно в одно и тоже время.

Теперь мы можем запустить блокирующий клиент из blocking-client/get-poetry.py, для того, чтобы скачать немного поэзии. Запустите клиент следующим образом:

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

Вы также можете поменять номера портов, если сервера слушают на других портах. Поскольку мы имеем дело с блокирующим клиентом, то он будет скачивать одну поэму из каждого порта по очереди, ожидая пока поэма будет полностью получена до того, как начать скачивать следующую поэму. Вместо печати поэм, блокирующий клиент выводит нечто вроде:

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

Это очень похоже на текстовую версию рисунка 1, где каждая задача качает одну поэму. В вашем выводе времена скачиваний могут немного отличаться. Ими можно варьировать через параметр сервера *--delay*. Попробуйте поменять его и увидеть как меняется время скачивания.

Теперь можно посмотреть на исходный код блокирующего сервера и клиента и отметить места в коде, где они блокируются при отправлении или получении сетевых данных.

2.5. Асинхронный клиент

Давайте теперь посмотрим на реализацию простого асинхронного клиента, написанного без ипользования Twisted. Сначала давайте его запустим. Запустите три сервера, как мы это делали выше. Если они все еще запущены, то мы можем снова их использовать. Теперь запустим асинхронный клиент, расположенный в async-client/get-poetry.py, как написано ниже:

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

В результате получится примерно такой вывод:

```
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
...
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.133169
```

На этот раз вывод намного длиньше, поскольку клиент выводит строку после каждого скачивания порции данных с любого сервера. Сервера, как и раньше, понемногу отдают поэзию. Заметьте, что выполнение отдельных задач чередуется, как это изображено на рисунке 3.

Попробуйте поменять опцию *--delay* для серверов (например, сделать один сервер медленнее, чем остальные) для того, чтобы увидеть как асинхронный клиент автоматически ”подстроится” под скорость медленных серверов, в тоже время успевая за быстрыми серверами. Это асинхронность в действии.

Также заметьте, что при найстройках сервера, которые были установлены выше, асинхронный клиент завершается примерно за 10 секунд, в то время как синхронному клиенту требуется примерно 23 секунды для получения всех поэм. Теперь вернемся к отличиям между рисунками 3 и 4. Тратя меньше времени на блокирование, наш асинхронный клиент может скачать все поэмы за более короткий промежуток времени, поскольку он переключается между всеми медленными серверами.

Технически, наш асинхронный клиент выполняет блокирующую операцию: он пишет в стандартный вывод, используя оператор print! Это не является проблемой в нашем случае. На локальной машине с терминальным shell’ом, готовым принять вывод, оператор print не будет реально блокироваться и будет выволняться досточно быстро по сравнению с нашими медленными серверами. Но, если бы мы захотели, чтобы наша программа была частью pipeline и все еще выполнялась асинхронно, нам нужно было бы использовать асинхронный ввод-вывод для стандартного ввода и вывода. Twisted имеет модули, реализующие это, но для простоты, мы будем использовать оператор print, даже в программах, написанных с использованием Twisted.

2.6. Пристальный взгляд

Теперь давайте посмотрим на исходный код асинхронного клиента. Отметим основные отличия между ним и синхронным клиентом:

1. Вместо соединения только с одним сервером в один момент времени асинхронный клиент соединяется со всеми серверами одновременно.

2. Сокетные объекты, использованные для соединения, устанавливаются в неблокирующее состояние с использованием вызова setblocking(0).

3. Используется метод select из модуля select для ожидания (блокирования) до того момента, когда хотя бы один из сокетов готов отдать некоторую порцию данных.

4. При чтении данных из серверов мы читаем по максимуму до момента блокирования сокета, затем переходим к чтению данных из следующего имеющегося готового к чтению сокета.

Ядром асинхронного клиента является внешний цикл в функции get_poetry. Этот цикл можно разбить на несколько шагов:

1. Ожидание (блокирование) всех открытых сокетов с использованием select’а, до тех пор пока один или более сокетов будут иметь данные для считывания.

2. Для каждого сокета с готовыми для считывания данными, прочитать их, но только в количестве, доступном на данный момент, чтобы избежать блокирования.

3. Повторять шаги выше до момента закрытия сокетов.

Синхронный клиент также имеет цикл (в функции main), но при каждой итерации цикла в синхронном клиенте скачивается полностью одна поэма. В то время как при каждой итерации асинхронного клиента скачиваются куски поэм. И мы не знаем, над какими из них мы будем работать в данной итерации, или сколько данных мы получим от каждой из

них. Все это зависит от относительных скоростей серверов и состояния сети. Мы только позволяем select'у сообщать нам о том, какие сокеты готовы на чтение, и читаем столько данных, сколько можем из каждого сокета, не блокируясь.

Если бы синхронный клиент соединялся бы с фиксированным количеством серверов (скажем тремя), нам бы совершенно не понадобился внешний цикл, поскольку можно было бы последовательно вызвать три раза функцию get_poetry. Но асинхронный клиент не может обходиться без внешнего цикла, поскольку нужно одновременно опрашивать все сокеты и обрабатывать столько данных, сколько было доставлено за заданную итерацию.

Использование подобного цикла, ожидающего событий и управляющего ими, является общепринятым, и известно как шаблон проектирования reactor. Изложенное выше проиллюстрировано на рисунке 5:

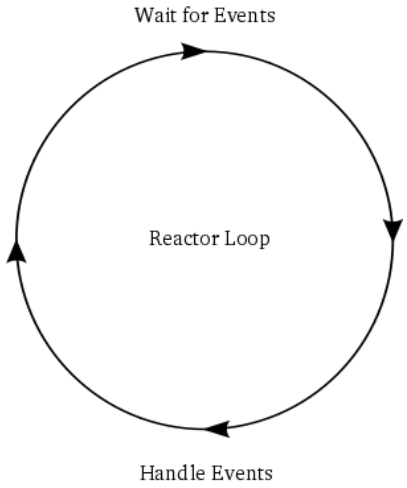

Рис. 5: Цикл reactor'а

Цикл является “reactor’ным”, поскольку он ожидает событий и затем реагирует на них. По этой причине, такой цикл известен также как ”event loop”. Поскольку реактивные системы зачастую ожидают ввода-вывода, такие циклы зачастую называют select циклы, поскольку вызов select используется для ожидания ввода-вывода. Таким образом в select циклах, ”событием” является момент, когда сокет становится доступным на чтение или запись. Надо отметить, что select - не единственный способ ожидания ввода-вывода, это один из самых старых (поэтому широко доступных) методов. Существует несколько более новых API, доступных на различных операционных системах, которые делают те же вещи, что и select, но имеют лучшую производительность. Но, невзирая на производительность, они все делают одно и тоже: берут множество сокетов (реально файловых дескрипторов) и блокируются до тех пор, пока один или более из них не станут готовы для операций ввода-вывода.

Нужно отметить, что можно использовать select и его товарищей для простой проверки на готовность множества файловых дескрипторов для операций ввода-вывода без блокирования. Это свойство позволяет реагирующей системе (reactive system) выполнять что-либо, не являющееся вводом-выводом, внутри цикла. Но в реагирующих системах в основном вся работа связана с вводом-выводом, таким образом блокирование на всех файловых дескрипторах сохранит ресурсы процессора.

Говоря точно, цикл в нашем асинхронном клиенте не соответсвует шаблону проектирования reactor, поскольку логика в цикле не реализована отдельно от основной логики, которая является специфичной для поэтических серверов. Они просто перемешаны друг с другом. Реальная реализация шаблона проектирования reactor реализовывала бы цикл как отдельную абстракцию, способную:

1. Принимать множество файловых дескрипторов для выполнения операций ввода-вывода.

2. Циклически сообщать о том, когда любой из файловых дескриторов готов на чтение или запись.

А реально хорошая реализация шаблона проектирования reactor делала бы:

1. Управление всеми проблемными случаями, которые происходят на различных системах.

2. Обеспечение множеством абстракций, помогающих использовать reactor с минимальными усилиями.

3. Предоставление реализаций основных протоколов, которые можно тут же использовать.

В целом, Twisted - это устойчивая, кросс-платформенная реализация шаблона проектирования reactor со множественными дополнениями. И в следующей главе мы начнем писать простые программы с использованием Twisted, так что сейчас мы движемся напрямую к получению поэзии со вкусом Twisted!

2.7. Упражнения

1. Проделайте несколько экспериментов с блокирующей и асинхронной версиями клиента, меняя количество и настройки поэтических серверов.

2. Может ли асинхронный клиент предоставить функцию get_poetry, которая возвращала бы текст поэмы? Почему нет?

3. Если бы вы хотели реализовать функцию get_poetry в асинхронном клиенте аналогичную синхронной версии, как бы это могло работать? Какие аргументы и возвращаемые значения были бы в этом случае?

3. Наш первый взгляд на Twisted

3.1. Ничего не делать - путь Twisted

На данный момент мы нацелены на улучшение кода асинхронного поэтического клиента, используя Twisted. Сначала давайте напишем несколько простых Twisted программ, чтобы осознать суть вещей. Как было упомянуто в главе 2, примеры были отлажены с использованием Twisted 8.2.0. API в Twisted меняется, но основные API, которые мы собираемся использовать меняются медленно, поэтому примеры будут актуальны и для следующих релизов. Если у вас не установлен Twisted, вы можете получить его здесь.

Самая простая Twisted программа представлена ниже. Код можно найти в директории с примерами в файле basic-twisted/simple.py.

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

Пример запускается следующим образом:

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

В предыдущей главе мы выяснили, что Twisted - реализация шаблона проектирования Reactor, таким образом Twisted содержит объект, который представляет reactor или event loop, который является ядром любой Twisted программы. Первая строка нашей программы импортирует объект reactor, чтобы его можно было использовать, вторая строка запускает цикл реактора.

Программа выше ничего не делает. Остановить ее можно, нажав Ctrl-C, иначе она будет вечно ничего не делать. Обычно в цикле мониторятся на ввод-вывод какие-нибудь файловые дескрипторы (например, соединенные с поэтическим сервером). Позже, мы сделаем это, но сейчас цикл реактора простаивает. Заметьте, что это не busy loop, который реально выполняет цикл. Если посмотреть на загруженность процесса, то не будет видно никаких всплесков. Фактически, наша программа совсем не использует процессор. Вместо этого reactor простаивает наверху цикла рисунка 5, ожидая события, которое никогда не произойдет (в реальности, ожидая возврата вызова select без файловых дескрипторов).

Сделаем несколько выводов:

1. Цикл реактора в Twisted запускается явно вызовом reactor.run().
2. Цикл реактора запускается в том же треде, где он был запущен. В примере выше, он был запущен в единственном main треде.
3. После запуска цикл бесконечно продолжается.
4. Если нет задач, цикл реактора не потребляет процессор.
5. reactor явно не создается, он просто импортируется.

Последний пункт стоит обсудить. Twisted reactor - Singleton. Существует только один объект reactor, и он создается неявно при импорте. Если открыть модуль reactor в пакете twisted.internet, то кода там будет очень мало. Реальная реализация находится в другом файле:
(twisted.internet.selectreactor.py).

В Twisted есть много различных реализаций реактора. Как было упомянуто в главе 2, вызов select'a - это один из способов ожидания событий на файловых дескрипторах.

По умолчанию, Twisted использует select, но Twisted также включает и другие типы реакторов. Например, twisted.internet.pollreactor, который использует системный вызов poll вместо select.

Для использования другого реактора нужно установить его до импортирования twisted.intern Вот так можно установить pollreactor:

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

Если проимпортировать twisted.internet.reactor без установки определенной реализации реактора, то Twisted установит reactor на основе select. По этой причине не импортируйте reactor в начале модулей, чтобы избежать случайной установки реактора по умолчанию, вместо этого импортируйте в том же месте, где собираетесь использовать.

Стоит заметить, что на момент написания статьи, Twisted двигается в сторону архитектуры, которая позволила бы использовать несколько реакторов одновременно. В такой схеме объект reactor мог бы подставляться как ссылка, а не как объект, импортированный из модуля.

Сейчас можно переписать нашу первую Twisted программу, используя pollreactor, которую можно найти в basic-twisted/simple-poll.py:

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

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

Таким образом, мы имеем poll цикл, который ничего не делает, вместо ничего не делающего select цикла.

Далее мы везде будем придерживаться реактора по умолчанию. Все реакторы Twisted делают одно и тоже.

3.2. Привет, Twisted

Давайте сделаем программу Twisted, которая что-нибудь делает. Например, программу, печатающую сообщение в окне терминала, после запуска цикла реактора:

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

from twisted.internet import reactor

reactor.callWhenRunning(hello)

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

Код программы находится в basic-twisted/hello.py. После запуска получится следующий вывод:

```
Starting the reactor.
Hello from the reactor loop!
Lately I feel like I'm stuck in a rut.
```

Программу выше все еще надо завершать самостоятельно, поскольку она бесконечно выполняется после того, как напечатает строки выше.

Заметьте, что функция hello вызывается после того, как reactor начал выполняться. Это означает, что функция вызывается реактором, то есть код Twisted должен вызывать эту функцию. Это происходит благодаря вызову метода реактора callWhenRunning с ссылкой на функцию, которую должен вызвать Twisted. И, конечно же, мы должны вызвать callWhenRunning до того, как мы запустим reactor.

Термин callback означает ссылку на функцию (например, на функцию hello), которую мы передаем Twisted (или какой-то другой системе), которую Twisted будет использовать для того, чтобы ”вызывать” в определенное время. В примере выше такой вызов происходит сразу же после того, как reactor запустится. Поскольку цикл в Twisted отделен от нашего кода, большинство взаимодействий между ядром реактора и логикой программы будет осуществляться через callback’и, которые мы передали Twisted, используя различные API.

Мы можем увидеть то, как Twisted вызывает наш код, используя следующую программу:

```
import traceback

def stack():
    print 'The python stack:'
    traceback.print_stack()

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

Код находится в basic-twisted/stack.py, и он напечатает что-то вроде этого:

```
The python stack:
...
  reactor.run() <-- This is where we called the reactor
...
...  <-- A bunch of Twisted function calls
...
  traceback.print_stack() <-- The second line in the stack function
```

Не обращайте внимание на все строки, просто осознайте взаимосвязь между вызовом reactor.run() и нашим callback’ом.

3.3. Кто использует callback’и?

Twisted не единственная система, которая использует callback’и. Более старые асинхронные Python системы Medusa и asyncore также их используют. Помимо этого, многие GUI библиотеки, в том числе GTK и QT, основаны на цикле реактора.

Отметим некоторые вещи:

1. Шаблон проектирования reactor однопоточный.
2. Реактивная система, подобная Twisted, реализует цикл реактора так, что наш код не должен его реализовывать.
3. Наш код должен реализовывать основную логику программы.

4. Поскольку программа выполняется в одном потоке, цикл реактора будет вызывать наш код.

5. reactor не может знать наперед какую часть нашего кода нужно вызвать.

Использование callback’ов, является неотъемлемой частью асинхронного программирования.

Рисунок 6 показывает, что случается во время обратного вызова:

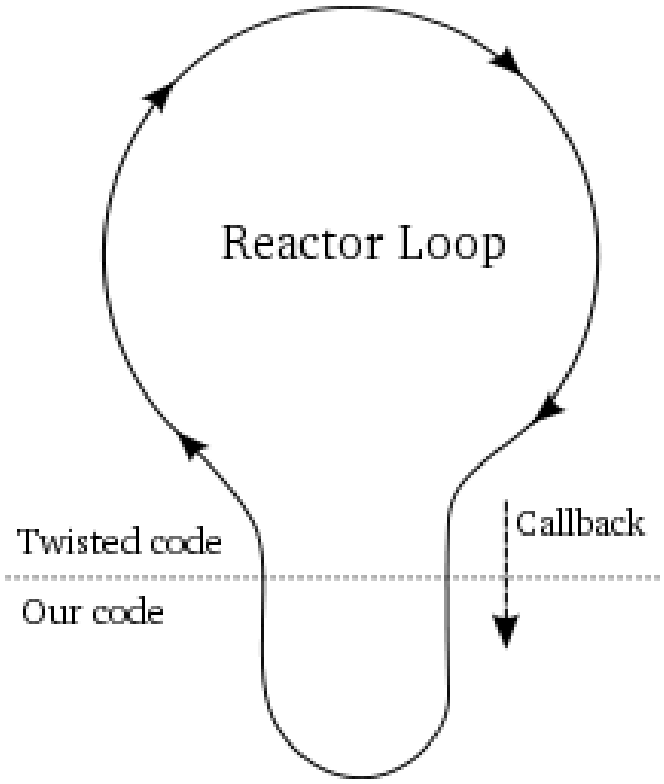

Рис. 6: reactor, делающий обратный вызов

Рисунок 6 иллюстрирует некоторые важные особенности callback'ов:

1. Код нашего callback'а запускается в том же потоке, что и Twisted цикл.
2. Когда наши callback'и выполняются, цикл Twisted не выполняется.
3. И наоборот.
4. Цикл реактора начинает выполняться после возврата из callback'ов.

Во время вызова callback-функции, цикл Twisted блокируетяся на нашем коде. Поэтому нам следует убедиться, что код нашего callback'а не тратит время зря. То есть нам нужно избегать реализации блокирующего ввода-вывода в коде callback'ов. Иначе, использование реактора окажется бесмысленным. Twisted не имеет никаких специальных мер для предотвращения кода от блокирования, мы сами должны заботиться об этом. Как мы увидим позже, в большинстве случаев сетевого ввода-вывода нам не нужно беспокоится о блокировании, позволяя Twisted делать асинхронные взаимодействия за нас.

Другие примеры потенциально блокирующих операций включают чтение и запись из несокетных файловых дескрипторов (побобных pipe) или ожидание завершения подпроцесса. Как вы будете переходить из блокирующихся к неблокирующимся операциям, зависит от того, что вы делаете, но зачастую есть уже реализованные Twisted API, которые могут помочь. Заметьте, что многие стандартные Python функции не имеют способа стать неблокирующимися. Например, вызов функции os.system всегда блокируется до тех пор, пока подпроцесс не остановится. При использовании Twisted, вам следует избегать os.system и отдать предпотчтение Twisted API для запуска подпроцессов.

3.4. До свидания, Twisted

В свою очередь, вы можете сказать реактору Twisted остановить выполнение, используя метод stop. Но однажды остановленный reactor не может быть перезапущен, поэтому в основном это делается, когда программа завершается.

В рассылке Twisted обсуждалась тема о том, чтобы сделать reactor ”перезапускаемым”. В версии 8.2.0, можно запустить и остановить reactor один раз.

Далее программа из примера basic-twisted/countdown.py, которая останавливает reactor примерно через 5 секунд:

```
class Countdown(object):

    counter = 5

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

from twisted.internet import reactor

reactor.callWhenRunning(Countdown().count)

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

Программа использует API callLater для регистрации callback’а с помощью Twisted. В вызове callLater callback - второй аргумент, а первый аргумент - количество секунд, через которое надо запустить передаваемый callback. Вы также можете использовать число с плавающей точкой для задания дробного числа секунд.

Так как же Twisted выполняет callback в нужное время? Эта программа не слушает никакие файловые дескрипторы, так почему же она не зависает на select’е подобно другим? Вызов select и подобные ему имеют опцию timeout. Если задать опцию timeout и не передать файловые дескрипторы, то вызов select’а возвратится через время timeout. Если подставить нулевой timeout, то можно быстро опросить множество файловых дескрипторов без блокирования.

timeout можно представить как разновидность события, который ожидает event loop из рисунка 5. Twisted использует timeout’ы для того, чтобы убедиться, что callback’и будут вызваны в нужное время или в приблизительно нужное время. Если какой-то другой callback займет действительно много времени на выполнение, запланированный callback может быть отложен. Использование метода callLater не является своего рода гарантией, которая требуется в системах hard realtime.

Далее вывод программы выше:

```
Start!
5 ...
4 ...
```

```
3 ...
2 ...
1 ...
Stop!
```

Заметьте, что строка ”Stop!” в конце показывает нам, что, когда reactor выходит, то происходит возврат функции reactor.run. И мы получаем программу, которая останавливает сама себя.

3.5. Возьми это на себя, Twisted

Поскольку Twisted зачастую завершается вызывая наш код ввиде callback’ов, вы можете удивиться тому, что происходит, когда callback генерирует исключение. Давайте попробуем это сделать. В программе basic-twisted/exception.py в одном callback’е генерируется исключение, в другом код выполняется нормально:

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

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

from twisted.internet import reactor

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

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

Если вы запустите это, то увидите следующее:

```
Starting the reactor.
Traceback (most recent call last):
  ... # I removed most of the traceback
exceptions.Exception: I fall down.
But I get up again.
```

Заметьте, что второй callback запустился после первого, хотя мы видим traceback из исключения от первого. И если мы закоментируем вызов reactor.stop(), то программа будет выполняться вечно. Так что reactor продолжает работать, даже когда callback завершается с ошибкой (хотя при этом он будет сообщать об ошибке).

Сетевые сервера в основном нуждаются в достаточно устойчивом ПО. Они не предполагают краха системы в случае каких-то ошибок, но сообщают о них. Приятно осознавать, что Twisted помогает обрабатывать случайные ошибки.

3.6. Поэзию, пожалуйста

Теперь мы готовы скачать немного поэзии с помощью Twisted. В следующей главе мы реализуем Twisted версию нашего асинхронного поэтического клиента.

3.7. Упражнения

1. Обновите программу countdown.py так, чтобы она имела три независимых счетчика, убывающих с разной скоростью. Остановите reactor, когда все счетчики завершатся.

2. Посмотрите на класс LoopingCall из twisted.internet.task. Перепишите программу countdow с использованием LoopingCall. Вам нужны только методы start и stop, и вам не нужно использовать возврат deferred'а. Про deferred'ы мы изучим позже.

4. Twisted поэзия

4.1. Наш первый Twisted клиент

Twisted используется в основном для написания серверов. Конечно, можно его использовать для написать клиентов, что существенно проще, чем написание серверов. Мы начнем с простого: с написания клиента. Давайте попробуем наш первый поэтический клиент, написанный с использованием Twisted. Исходный код можно найти в twisted-client-1/get-poetry.py. Сначала нужно запустить поэтические сервера:

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

Затем запустите клиент следующим образом:

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

И получится следующий вывод:

```
Task 1: got 60 bytes of poetry from 127.0.0.1:10000
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 1: got 30 bytes of poetry from 127.0.0.1:10000
Task 3: got 10 bytes of poetry from 127.0.0.1:10002
Task 2: got 10 bytes of poetry from 127.0.0.1:10001
...
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.134220
```

Также как мы это делали для нашего асинхронного клиента, не использующего Twisted. И это неудивительно, что клиент выполняет тоже, что и раньше. Давайте посмотрим на исходный код для того, чтобы понять как он работает. Откройте их в редакторе, чтобы можно было смотреть на то, что мы будем обсуждать.

Заметьте, что как было упомянуто, мы начнем изучать Twisted с очень низкоуровневых API. Проходя по абстракциям Twisted снизу вверх, мы сможем изучить Twisted изнутри и сверху. Это означает, что мы изучим многие API, которые обычно не используются при написании реального кода. Запомните, что первоначальные примеры - это не то, как надо писать боевой код, а лишь способ изучения Twisted.

Twisted клиент начинается с создания множества объектов типа PoetrySocket. PoetrySocket инициализирует себя сам созданием сетевого сокета, соединенного с сервером и выставленного в неблокирующий режим:

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

Конечно, в дальнейшем мы будет иметь дело с абстрациями, при использовании которых не надо работать с сокетами, но сейчас нам нужно работать с ними. После создания сетевого соединения, PoetrySocket подставляет себя в reactor через метод addReader:

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

Этот метод дает Twisted файловый дескриптор, который он должен проверить на предмет приходящих данных. Почему мы передаем Twisted объект вместо файлового дескриптора и callback'а? И как Twisted узнает, что делать с этим объектом, ведь Twisted не содержит кода, относящегося к поэзии? Откройте twisted.internet.interfaces.py, и мы посмотрим.

4.2. Twisted интерфейсы

В Twisted много подмодулей, называемых интерфейсами. Каждый из них определяет множество классов Interface. Начиная с версии 8.0, Twisted использует zope.interface в качестве основы для этих классов, но реально детали этого пакета нам неинтересны. Мы будем рассматривать только подклассы класса Interface в самом Twisted, подобно тому, на который мы смотрим сейчас.

Одна из основных целей интерфейсов - документация. Будучи Python программистом, вы несомненно знакомы с Duck typing нотацией, при которой тип объекта не определяется его позицией в классовой иерархии, а определяется public интерфейсом, который он предоставляет. Таким образом два объекта, представляющие один и тот же public интерфейс (например, ходит как утка, крякает как ...) являются, согласно duck typing, одним и тем же видом объекта (уткой!). Таким образом, Interface - некий формализованный способ определения того, что означает "ходит как утка".

Найдите в twisted.internet.interfaces определение метода addReader. Он определен в IReactorFDSet интерфейсе и должен быть таким:

```
def addReader(reader):
    """
    I add reader to the set of file descriptors to get read events for.

    @param reader: An L{IReadDescriptor} provider that will be checked for
                   read events until it is removed from the reactor with
                   L{removeReader}.

    @return: C{None}.
    """
```

IReactorFDSet - это один из интерфейсов, которые реализуют Twisted реакторы. Таким образом, любой Twisted reactor имеет метод с названием addReader, который работает как написано выше в строке с документацией. Объявление метода не имеет аргумента self, поскольку он имеет отношение только к определению public интерфейса, в то время как аргумент self - часть реализации (например, при вызове не надо явно передавать self). Из интерфейса никогда не создают объекты и они никогда не используются как базовые классы при реализации.

Некоторые замечания:

1. Исходя из названия, IReactorFDSet был бы интерфесом к реакторам, которые ожидают на файловых дескрипторах, но на данный момент IReactorFDSet - интерфейс ко всем имеющимся реализациям реакторов.

2. Интерфейсы можно использовать не только как документацию. Модуль zope.interface позволяет явно объявить, что класс реализует один или более интерфейсов, обеспечивая проверку во время запуска. Также поддерживается концепция адаптации, способности динамически предоставлять заданный интерфейс для объекта, который может не поддерживать этот интерфейс напрямую. Мы не будем углубляться в эти более сложные виды использования.

3. Можно заметить сходство между Interface'ми и абстрактыми базовыми классами, недавно добавленными в Python. Мы не будем изучать их сходство и отличия, но, возможно, будет интересно почитать эссэ [1], написанное Glyph'ом, основателем проекта Twisted, который затрагивает эту тему.

Согласно строке с документацией выше, аргумент reader метода addReader должен реализовывать интерфейс IReadDescriptor. И это означает, что наши PoetrySocket объекты тоже должны это делать.

Давайте просмотрим модуль и найдем в нем этот интерфейс:

```
class IReadDescriptor(IFileDescriptor):

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

Реализацию doRead можно найти в классе PoetrySocket. Внутри метода происходит асинхронное чтение из сокета при каждом вызове Twisted реактора. Так что doRead - это callback, но мы его не подставляем напрямую в Twisted, а передаем объект, который имеет метод doRead. Это общепринятая идиома в системе Twisted - вместо того, чтобы передать функцию, вы передаете объект, который реализует заданный Interface. Это позволяет передавать множество callback'ов (методов, определенных в Interface'е) одним аргументом. Это также позволяет callback'ам взаимодейстсвовать друг с другом через общее состояние, хранимое в объекте.

Так что по поводу других callback'ов, реализованных в PoetrySocket? Заметьте, что IReadDescriptor - подкласс IFileDescriptor. Это означает, что любой объект, который реализует IReadDescriptor должен также реализовывать IFileDescriptor. И, если вы посмотрите в модуль, то найдете:

```
class IFileDescriptor(ILoggingContext):
    """
    A file descriptor.
    """

    def fileno():
        ...

    def connectionLost(reason):
        ...
```

Комментарии выше немного урезаны, но из названия понятно, что метод fileno возвращает файловый дескриптор, который нам нужно мониторить, и метод connectionLost

[1] http://glyph.twistedmatrix.com/2009/02/explaining-why-interfaces-are-great.html

вызывается при закрытии соединения. Наши объекты типа PoetrySocket также реализуют эти методы.

IFileDescriptor выведен из ILoggingContext. Далее мы не будем на него смотреть, но это это является причиной тому, почему мы реализовали callback logPrefix. Детали можно изучить в модуле interfaces.

Заметьте, что doRead возвращает специальные значения, указывающие на то, что сокет закрыт. Как узнать об этих значениях? В основном, код не будет работать без них и возникла необходимость просмотреть реализацию этого же интерфейса в Twisted, чтобы понять, что делать. Вам нужно осознать, что порой, документация неточная и неполная.

4.3. Еще про callback'и

Наш новый Twisted клиент очень похож на наш первоначальный асинхронный клиент. Оба клиента соединяются с сокетами и асинхронно читают данные из этих сокетов. Основным отличием является то, что в клиенте, использующем Twisted, не нужно своего собственного select цикла, вместо этого он использует Twisted reactor.

doRead callback - самый важный callback. Twisted вызывает его, сообщая нам о том, есть ли какие-нибудь данные, готовые на чтение из нашего сокета. Иллюстрация этого на рисунке 7:

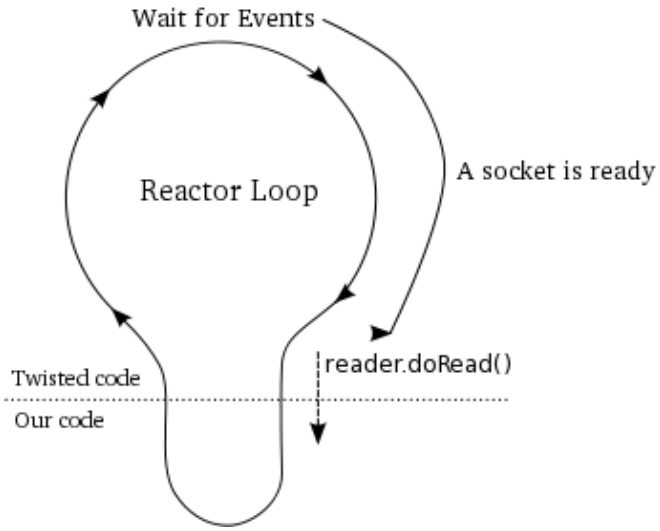

Рис. 7: doRead callback

Каждый раз, когда вызывается callback, из сокета считывается максимальное количество данных, после чего чтение прерывается без блокирования. Как было упомянуто в главе 3, Twisted не может предотвратить какие-то ни было проблемы в нашем коде, в том числе и блокирование. Чтобы это понять это, подправим наш код и посмотрим что получится. В той же директории, где находится Twisted клиент, расположен сломанный клиент в twisted-client-1/get-poetry-broken.py. Этот клиент отличается следующим:

1. Сломанный клиент не заботится о том, чтобы установить socket в неблокирующий режим.

2. callback doRead считываниет данные до закрытия сокета (то есть в этом случае происходит блокирование в момент, когда на сокете нет данных).

Теперь попробуем запустить сломанный клиент:

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

Получится примерно такой вывод:

```
Task 1: got 3003 bytes of poetry from 127.0.0.1:10000
Task 3: got 653 bytes of poetry from 127.0.0.1:10002
Task 2: got 623 bytes of poetry from 127.0.0.1:10001
Task 1: 3003 bytes of poetry
Task 2: 623 bytes of poetry
Task 3: 653 bytes of poetry
Got 3 poems in 0:00:10.132753
```

Несмотря на порядок задач, вывод похож на тот, что был у блокирующегося клиента. Это потому что наш сломанный клиент и является блокирующимся клиентом. Используя блокирующийся вызов recv в нашем callback'е, мы превратили нашу асинхронную Twisted программу в синхронную.

Своего рода многозадачность, которую предоставляет event loop в Twisted, кооперативная. Twisted скажет нам, когда можно читать или писать в файловый дескриптор, но мы должны передавать данные без блокирования. И мы должны избегать блокирующих вызов, подобных os.system. Более того, если мы имеем длительную вычислительную задачу (потребляющую процессор), за нами разделить ее на несколько меньших кусков так, чтобы другие задачи могли выполнять ввод-вывод.

Заметьте, что есть смысл в том, что наш сломанный клиент все еще работает: он скачивает запрашиваемую поэзию. Единственное, что он не использует преимущества асинхронного ввода-вывода. Можно заметить, что сломанный клиент работает намного быстрее, чем первоначальный блокирующийся клиент. Это потому что сломанный клиент соединяется со всеми серверами при старте программы. Поскольку сервера начинают отправлять данные немедленно, и поскольку операционная система буферизирует некоторые приходящие данные для нас (до какого-то фиксированного предела), даже если мы их не читали, наш блокирующий клиент эффективно получает данные из других серверов, даже если происходит поочереденое считывание.

Но этот "трюк" работает только для небольшого объема данных, подобных нашим поэмам. Если бы мы скачивали, скажем, эпические саги с 20 милионнами слов, являющиеся записями одного хакера в попытке написать великолепный интерпретатор Lisp, буферы операционной системы быстро бы заполнились, и наш сломанный клиент был бы не эффективнее нашего первоначального блокирующего клиента.

4.4. Резюме

Больше нечего сказать о нашей первой Twisted версии клиента. Заметьте, что connectionLost callback завершает reactor после того, как закончились все ожидающие поэм PoetrySocket'ы. Это не особо хороший способ, поскольку он предполагает, что в программе мы ничего кроме скачивания поэзии не делаем, но позволяет проиллюстрировать несколько низкоуровневых реакторных API: removeReader и getReaders.

Существуют Writer API, эквивалентные Reader API, которые мы использовали, и они работают также для файловых дескрипторов, которые мы хотим мониторить, чтобы отправить данные. Просмотрите interface'ы для ознакомления с деталями. Причина, по которой чтение и запись имеют отдельные API, связана с тем, что вызов select отличает виды событий (файловый дескриптор доступен на чтение или запись). Конечно же, можно ожидать обоих событий на одном и том же файловом дескрипторе.

В следующей главе, мы напишем вторую версию нашего Twisted поэтического клиента, используя некоторые высокоуровневые абстракции, и по пути изучим несколько Twisted интерфейсов и API.

4.5. Упражнения

1. Исправьте клиент так, чтобы сбой при соединении с сервером не рушил программу.
2. Воспользуйтесь callLater, для того, чтобы сделать клиентский timeout, в случае, если поэма не скачалась за заданный временной интервал. Прочитайте о возвращаемом значении callLater, для того, чтобы вы могли отменить таймер, в случае, если поэма скачалась во время.

5. Улучшенная Twisted поэзия

5.1. Абстрактный экспрессионизм

В предыдущей главе мы сделали наш первый поэтический клиент, использующий Twisted. Он работает достаточно хорошо, но в нем определенно есть что улучшить.

В первую очередь, клиент влючает код с такими деталями, как создание сетевых сокетов, получение данных из сокетов. Twisted поддерживает эти вещи, так что нам не нужно реализовывать их самим каждый раз, когда мы пишем новую программу. Это особенно удобно, поскольку асинхронный ввод-вывод требует нескольких хитростей, включающих обработку исключений, так как это делается в коде клиента. И есть много всяких хитростей, для того, чтобы сделать код работающим на нескольких платформах. Если у вас есть свободное время - поищите в исходниках Twisted слово ”win32”, для того, чтобы увидеть как много всяких проблем исходит от этой платформы.

Другая проблема в текущем клиенте - обработка ошибок. Попробуйте запустить версию 1.0 Twisted клиента и укажите порт, на котором не запущен сервер. Программа просто разрушится. Мы могли бы исправить текущий клиент, но управление ошибками проще делать с помощью Twisted API, которые мы сегодня будем использовать.

И наконец, клиент не является переиспользуемым. Как другой модуль мог бы получить поэму с помощью нашего клиента? Как ”вызывающий” модуль узнал бы, когда поэма скачалась? Мы не можем написать функцию, которая просто возвращает текст поэмы, так как это потребовало бы блокирования до тех пор, пока вся поэма не прочитана. Это реальная проблема, но мы не собираемся исправлять ее сегодня - мы это сделаем в следующих главах.

Мы собираемся исправить первую и вторую проблемы, используя высокоуровневое множество API и Interface’ов.

В целом, каждая Twisted абстрация касается только одной определенной концепции. Например, клиент 1.0 из главы 4 использует IReadDescriptor: абстракция ”файлового дескритора, из которого можно прочитать данные”. Twisted абстракция обычно определяется Interface’ом, задающим то, как объекту, воплощающему абстракцию, следует себя вести. Наиболее важная деталь, про которую надо помнить, следующая:

Большинство высокоуровневых абстракций в Twisted построены с использованием низкоуровневых, не заменяя их.

Поэтому, когда вы изучаете новую Twisted абстракцию, поймите, что она делает и что не делает. Особенно, если некоторая более ранняя абстракция A реализует свойство F, то F вероятно не реализована другой абстракцией. Пожалуй, если другой абстракции B нужно свойство F, она будет переиспользовать A нежели, чем реализовывать F сама (реализация B будет либо наследовать реализацию A, либо ссылаться на другой объект, который реализует A).

Сети - сложный предмет, так что Twisted содержит множество абстракций. Начиная с низкого уровня, мы получим более ясную картину того, как все соединяется вместе в работающей Twisted программе.

5.2. Без цикла в мозге

Наиболее важная абстракция, которую мы узучили до сего момента, являющаяся самой важной абстракцией в Twisted, - reactor. В центре каждой программы, построенной с помощью Twisted, невзирая на количество уровней, которые программа может иметь, существует реакторный цикл, в котором осуществляются все запуски. В Twisted функциональность реактора реализована только в реакторе. Об остальном в Twisted можно думать как о ”вещах, упрощающих реализацию X, использующего reactor”, где X - может

быть ”обслужить web страницу” или ”сделать запрос к базе”, или что-то подобное. Хотя можно придерживаться низкоуровневых API, подобных тому, что делает клиент 1.0, но в этом случае мы должны реализовать больше всяких мелочей. Перемещение на более высокий уровень абстракций в основном означает, что кода будет меньше (и позволяет Twisted управлять зависимыми от платформы ньюансами).

Но, когда мы работаем на внешних уровнях Twisted, можно просто забыть про reactor. В Twisted программе разумного размера, относительно мало частей кода будут действительно использовать API реактора напрямую. Тоже самое верно для других низкоуровневых абстракций. Абстракции файлового дескриптора, которые мы использовали в клиенте 1.0, являются также тщательно включенными высокоуровневыми концепциями, они в основном не видны в реальных Twisted программах (они все еще используются внутри, но мы их просто не видим как таковых).

Поскольку абстракции файловых дескрипторов работают, то можно позволить Twisted управлять механизмами асинхронного ввода-вывода и освободить нас для концентрации над тем, что мы действительно хотим решить. Но reactor отличается. Он в действительности никогда не исчезнет. Когда вы выбираете использование Twisted, вы также выбираете использование шаблона проектирования Reactor, и это означает программированко-оперативнойном стиле”, с использованием callback’ов и кооперативной многозадачности (cooperative multi-tasking). Если вы хотите использовать Twisted корректно, вы должны помнить про существование реактора и про то, как он работает. В главе 6 будет больше сказано про это, резюмируя можно сказать:

Рисунки 5 и 6 - самые важные в данном введении.

Мы будет иллюстрировать новые концепции, но эти два рисунка обязательно нужно запомнить и держать в голове при написании программ с использованием Twisted.

До того, как мы погрузимся в код, нужно ознакомиться с тремя новыми абстракциями: Transport, Protocol и Protocol Factory.

5.3. Транспорты

Абстракция Transport определяется в ITransport в модуле Twisted interfaces. Twisted Transport представляет одно соединение, которое может отправлять и/или получать байты. Для наших поэтических клиентов, Transport’ы - абстрактные TCP соединения подобно тем, которые мы делали в ранних версиях. Twisted также поддерживает ввод-вывод через Unix pipe’s и UDP сокеты. Абстракция Transport предствляет любое такое соединение и управляет деталями асинхронного ввода-вывода для любого типа соединения, которое оно представляет.

Если посмотреть на методы, определенные для ITransport, то там ничего не будет про получение данных. Поскольку Transport’ы всегда управляют низкоуровневыми деталями асинхронного чтения данных из их соединений, и дают нам данные через callback’и. Среди подобных строк, методы объектов типа Transport, имеющие отношение к записи, могут не записывать данные тут же, избегая блокирования. Говоря Transport’у записать некоторые данные означает ”отправьте эти данные так скоро, как вы сможете это сделать, избегая блокирования”. Данные будут записаны, конечно же, в том же порядке, в котором мы их предоставили.

В общем мы не реализовываем свой свои собственные Transport объекты или не создаем их в нашем коде. Скорее, мы используем реализации, которые Twisted уже предоставил и которые создаются для нас, когда мы говорим реактору сделать соединение.

5.4. Протоколы

Twisted Protocol'ы определены в IProtocol в том же модуле interfaces. Как Вы могли бы ожидать, объекты типа Protocol реализуют протоколы. Нужно сказать, что определенный Twisted Protocol реализует один определенный сетевой протокол, такой как FTP или IMAP, или какой-нибудь безымяный протокол, который мы изобрели для наших целей. Наш поэтический протокол, так как он есть, просто отправляет все байты поэмы в момент установления соединения до момента соединения, означающего конец поэмы.

Строго говоря, каждый экземпляр объекта типа Twisted Protocol реализует протокол для одного определенного соединения. Поэтому каждое соединение, которое делает наша программа (или, в случае серверов, принимает), будет требовать один экземпляр Protocol. Это делает экземпляры типа Protocol естественным местом для хранения состояния протоколов и собранных данных из частично полученных сообщений (поскольку мы получаем куски произвольного размера, используя асинхронный ввод-вывод).

Так как же экземпляры Protocol узнают за какое соединение они ответственны? Если вы посмотрите на определение IProtocol, вы обнаружите метод с названием makeConnection. Этот метод - callback, и Twisted код вызывает его с экземпляром типа Transport в качестве единственного аргумента. Transport - соединение, которое Protocol будет использовать.

Twisted включает большое количество реализаций для различных общеиспользуемых протоколов. Вы можете найти несколько простых в twisted.protocols.basic. Хорошая идея проверить исходники Twisted до того, как вы напишите новый Protocol, возможно, что уже есть готовая реализация. Но, если ваш протокол еще не реализован, то нет проблем реализовать его самим, так как мы сделаем это для наших поэтических клиентов.

5.5. Протокольные фабрики

Так как каждому соединению нужен свой собственный Protocol, и этот Protocol может быть экземпляром класса, который мы сами реализовали. Поскольку мы позволим Twisted управлять созданием соединений, Twisted нужнен способ, которым он будет делать соответсвующие Protocol "по запросу" в момент создания нового соединения. Создавать экземпляры Protocol - задача протокольных фабрик (Protocol Factory).

Как вы уже вероятно догадались, Protocol Factory API определены IProtocolFactory, также в модуле interfaces. Protocol Factory - пример шаблона проектирования Factory, и они работают просто. Метод buildProtocol возвращает новый экземпляр Protocol каждый раз при вызове. Это метод, который использует Twisted для того, чтобы сделать новый Protocol для каждого нового соединения.

5.6. Получение поэзии 2.0: первая кровь.0

Давайте посмотрим на версию 2.0 Twisted поэтического клиента. Код находится в twisted-client-2/get-poetry.py. Вы можете запустить его, подобно другим и получить похожий вывод, так что не будем надоедать здесь выводом. Это также последняя версия клиента, которая печатает номера задач и сколько для каждой задачи было получено байт. Сейчас должно быть ясно, что все Twisted программы работают, чередуя задачи и обрабатывая относительно маленький кусок данных за раз. Мы все еще будем использовать оператор print для того, чтобы показать, что происходит в ключевые моменты, но в будущем клиенты не будут такими многословными.

В клиенте 2.0 сокеты исчезли. Мы даже не импорируем модуль socket, и мы никогда не ссылаемся на объект типа socket или файловый дескриптор. Вместо этого мы говорим реактору делать соединения к поэтическим серверам на нашей стороне следующим образом:

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

from twisted.internet import reactor

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

Сфокусируемся на методе connectTCP. Первые два аргумента понятны из названия. Третий аргумент - экземпляр нашего класса PoetryClientFactory.

Этот класс - Protocol Factory для наших поэтических клиентов, и передача его реактору позволяет Twisted создавать экземпляры нашего PoetryProtocol на лету.

Заметьте, что мы не реализуем ни Factory, ни Protocol с нуля, подобно объектам типа PoetrySocket в нашем предыдущем клиенте. Вместо этого, мы наследуем базовые реализации, которые предоставляет Twisted в twisted.internet.protocol. Первоначальный базовый класс Factory находится в twisted.internet.protocol.Factory, но мы наследуем класс ClientFactory, который специализирован для клиентов (процессы создают соединения вместо того, чтобы их слушать, как в случае сервера).

Мы также воспользовались тем фактом, что класс Factory в Twisted реализует для нас метод buildProtocol. Мы вызываем реализацию этого метода в базовом классе из нашего подкласса:

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

Как базовый класс узнает какой Protocol создать? Заметьте, что мы также устанавливаем атрибут класса protocol в PoetryClientFactory:

```
class PoetryClientFactory(ClientFactory):

    task_num = 1

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

Базовый класс Factory реализует buildProtocol созданием экземпляра класса, который мы установили в атрибут protocol (например, PoetryProtocol), и устанавливает атрибут factory новому объект, чтобы у него была ссылка на ”родительский” Factory. Это проиллюстрировано на рисунке 8:

Как мы упомянули выше, атрибут factory в объектах типа Protocol позволяют протоколам, созданным одним и тем же объектом типа Factory, разделять состояние. И поскольку фабрики создаются пользовательским кодом, одинаковый атрибут позволяет объектам типа Protocol передавать результаты обратно в код, который задал запрос, как мы увидим в главе 6.

Заметьте, что в то время как атрибут factory в Protocol’ах ссылается на экземпляр Protocol Factory, атрибут protocol в Factory ссылается на класс Protocol. В общем случае, один объект Factory может создать много экземпляров Protocol.

Вторая стадия создания Protocol соединяет Protocol с Transport, используя метод makeConne
Мы не должны реализовывать этот метод сами, поскольку базовый класс в Twisted предоставляет реализацию по умолчанию. По умолчанию, makeConnection сохраняет ссылку

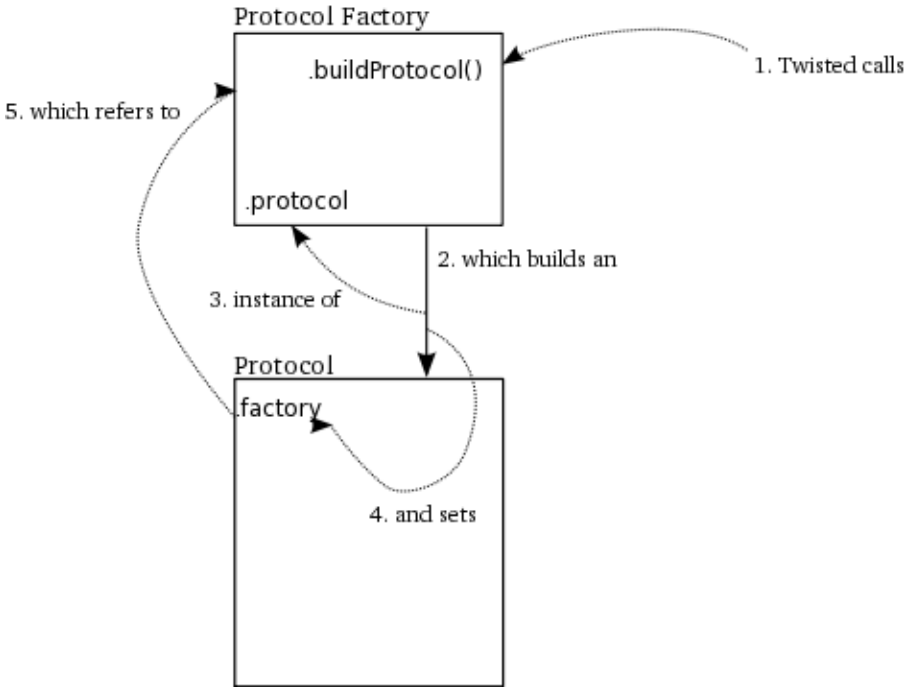

Рис. 8: Рождение Protocol'а

на Transport в атрибуте transport и устанавливает атрибут connected в значение True, как это изображено на рисунке 9:

Однажды проинициализировав таким образом, Protocol может начать выполнение своей реальной работы - передачу низкоуровневого потока данных в высокоуровневый поток протокольных сообщений (и, наоборот, для двунаправленных соединений). Ключевой метод для обработки приходящих данных - dataReceived, который наш клиент реализовывает следующим образом:

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

Каждый раз при вызове dataReceived мы получаем новую последовательность байт (данных) в форме строки. Как всегда с асинхронным вводом-выводом, мы не знаем как много данных мы получим, так что мы должны буферизовать их, до тех пор пока мы не получим завершенное протокольное сообщение. В нашем случае, поэма не завершается до тех пор, пока не закроется соединение, поэтому мы добавляем байты в наш атрибут poem.

Заметьте, что мы используем метод getHost нашего Transport'а для идентификации того, какой с какого сервера пришли данные. Мы делаем это только для согласованности с более ранними клиентами. Иначе, нашему коду не нужно было бы использовать Transport явно, поскольку мы никогда не отправляем данные серверу.

Давайте посмотрим на то, что происходит, когда вызвается метод dataReceived. В той же директории, где находится наш клиент 2.0, есть еще один клиент в файле twisted-client-2/get-poetry-stack.py. Это тоже самое, что и клиент 2.0, за исключением того, что метод dataReceived изменен следующим образом:

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

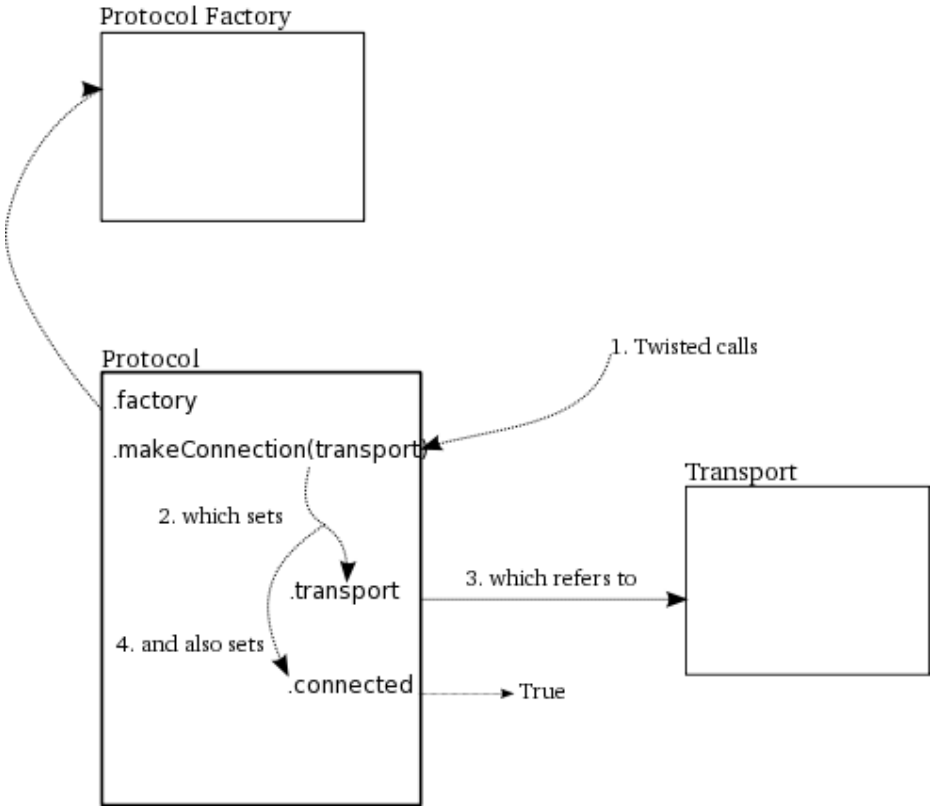

Рис. 9: Ссылка из Protocol в Transport

С этим изменением программа будет печатать stack trace и затем завершится сразу после получения некоторых данных. Вы можете запустить эту версию следующим образом:

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

И вы получите следующий stack trace:

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

... # I removed a bunch of lines here

File ".../twisted/internet/tcp.py", line 463, in doRead  # Note the doRead callback
    return self.protocol.dataReceived(data)
File "twisted-client-2/get-poetry-stack.py", line 58, in dataReceived
    traceback.print_stack()
```

В выводе мы видим doRead callback, который мы использовали в клиенте 1.0! Как было замечено ранее, Twisted создает новые абстракции, используя старые, но не заменяя их. То есть здесь в работе все еще реализация IReadDescriptor, но она реализована внутри Twisted, а не в нашем коде. Если вы любопытны, то реализация находится в twisted.internet.tcp. Если вы посмотрите в код, то найдете похожий объект, реализующий IWriteDescriptor и ITransport. Так что IReadDescriptor - это в действительности замаскированный объект типа Transport. Мы можем визуализировать dataReceived callback как на рисунке 10:

Сразу же, когда поэма скачалась, объект PoetryProtocol сообщает об этом PoetryClientFactor

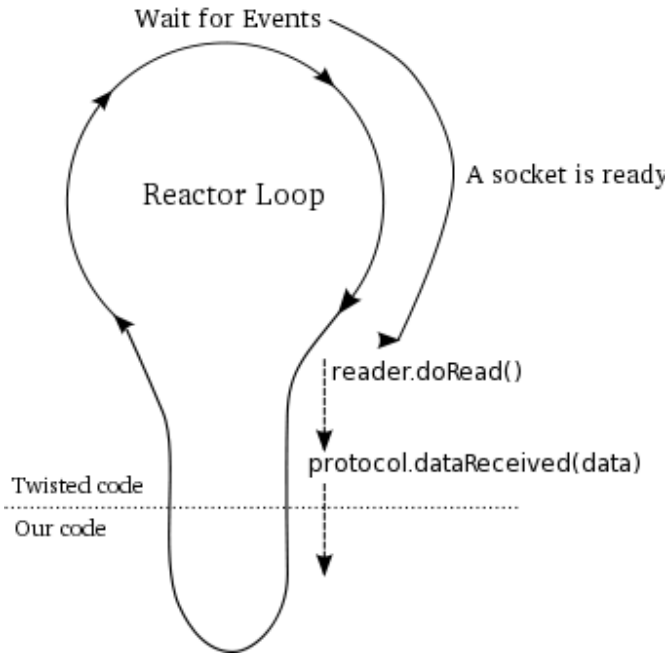

Рис. 10: dataReceived callback

```
def connectionLost(self, reason):
    self.poemReceived(self.poem)

def poemReceived(self, poem):
    self.factory.poem_finished(self.task_num, poem)
```

callback connectionLost вызывается при закрытии транспортного соединения. Аргумент reason - это объект типа twisted.python.failure.Failure с дополнительной информацией о том закрылось ли соединение без ошибок или из-за ошибок. Наш клиент просто игнорирует это значение и предполагает, что мы получили поэму полностью.

factory останавливает reactor после того, как все поэмы скачались. И снова мы предполагаем, что единственное, что наша программа делает - скачивает поэмы, что делает объекты типа PoetryClientFactory менее переиспользуемыми. Мы исправим это в следующей главе, но заметьте то, как poem_finished callback сохраняет количество оставшихся поэм:

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

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

Если бы мы писали многотредовую программу, где каждая поэма скачивается в отдельном потоке, нам бы понадобилось защищать участок программы выше с помощью lock'а в случае двух или более потоков, вызывающих poem_finished в одно и то же время. Иначе, мы могли бы завершить reactor дважды (и получить исключение). Но с реактивной системой нам не нужно об этом заботиться: reactor может вызывать только один callback за раз, так что такая проблема не может произойти.

Наш новый клиент управляет ошибкой соединения изящнее, чем в клиент 1.0. Следующий callback в классе PoetryClientFactory делает это:

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

Заметьте, что callback находится в Factory, а не в Protocol. Поскольку Protocol создается только после создания соединения, а Factory получает новости о том, что соединение не установилось.

5.7. Упрощение клиента

Хотя наш клиент уже достоточно прост, мы можем еще больше его упростить, если обойдемся без номеров задач. Клиент ведь реально о поэзии в конце концов. Упрощенная версия клиента 2.1 находится в twisted-client-2/get-poetry-simple.py.

5.8. Резюме

Клиент 2.0 использует Twisted абстрации, которые знакомы любому Twisted хакеру. И если все, что мы хотели - это клиент, запускающийся из командной строки, печатающий некоторую поэзию и после - завершающийся, то мы могли бы здесь остановится и сказать, что наша программа сделана. Но если вы хотите некоторый переиспользуемый код, который вы могли бы встроить в большую программу, которой надо было скачать немного поэзии и сделать также другие вещи, то у нас все еще есть над чем работать. В главе 6 мы сделаем первый удар в эту сторону.

5.9. Упражнения

1. Используйте callLater для того, чтобы сделать timeout для клиента, если поэма не скачалась после заданного интервала времени. Используйте метод loseConnection класса Transport для того, чтобы закрыть соединение при возникновлении timeout’а, и не забудьте сбросить timeout, если поэма скачалась во время.

2. Используйте метод stacktrace для анализа последовательности callback’ов, которые вызываются в момент вызова connectionLost.

6. Дальнейшие улучшения

6.1. Поэзия для всех

Мы сделали большой прогресс с нашим поэтическим клиентом. Наша последняя версия (2.0) использует Transport, Protocol и Protocol Factory. Но есть что улучшить. Клиент 2.0 (а также 2.1) может быть использован только для скачивания поэзии из командной строки. Это потому что PoetryClientFactory не только отвечает за получение поэзии и создание PoetryProtocols, но также отвечает за завершение программы по окончанию скачивания, и это случайная работа для PoetryClientFactory.

Нам нужен способ отправить поэму коду, который запрашивает поэму. В синхронной программе, мы могли бы сделать такое API:

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

Конечно, мы не можем сделать тоже самое. Функция выше блокируется до того, как поэма не будет получена полностью, иначе - она бы не работала согласно документации. Но у нас реактивная программа, поэтому блокирование на сетевом сокете вне вопроса. Нам нужен способ сказать вызывающему коду, когда поэма готова, без блокирования на тот момент, когда поэма доставляется. Но в этом то и проблема. Twisted должен сказать нашему коду, когда сокет будет готов для ввода-вывода, или когда некоторые данные получены, или когда произошел timeout и т.д. Мы видели, что Twisted решает эти проблемы, используя callback'и, поэтому мы также можем их использовать:

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

      callback(poem)

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

Теперь у нас есть асинхронное API, которое мы можем использовать в Twisted, поэтому давайте двигаться вперед и его реализуем.

Как мы сказали до этого, временами мы будем писать код так, как бы типичный Twisted программист не писал бы. Это один из этих моментов и один из этих способов. Мы увидим в главе 7 и 8 как это делается "Twisted способом" (к удивлению, при этом используются абстракции!), но начиная с простого, мы сможем более глубого понять окончательную версию.

6.2. Клиент 3.0

Вы можете найти версию 3.0 нашего поэтического клиента в twisted-client-3/get-poetry.py. Эта версия имеет реализацию функции get_poetry:

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

Здесь единственная новая загогулина - передача callback'а функции PoetryClientFactory. factory использует callback для доставки поэмы:

```
class PoetryClientFactory(ClientFactory):

    protocol = PoetryProtocol

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

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

Заметьте, что factory намного проще, чем в версии 2.1, поскольку теперь не отвечает за завершение реактора. Также отсутсвует код для обнаружения ошибок при соединении, но мы исправим это немного позже. Сам PoetryProtocol не нуждается ни в каких изменениях, так что мы просто скопируем его из клиента 2.1:

```
class PoetryProtocol(Protocol):

    poem = ''

    def dataReceived(self, data):
        self.poem += data

    def connectionLost(self, reason):
        self.poemReceived(self.poem)

    def poemReceived(self, poem):
        self.factory.poem_finished(poem)
```

С этим изменением, функция get_poetry и классы PoetryClientFactory и PoetryProtocol теперь полностью могут быть переиспользованы. Они делают только скачивание поэзии и ничего больше. Вся логика для запуска и завершения реактора находится в функции main нашего скрипта:

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

    from twisted.internet import reactor

    poems = []

    def got_poem(poem):
        poems.append(poem)
        if len(poems) == len(addresses):
            reactor.stop()

    for address in addresses:
        host, port = address
        get_poetry(host, port, got_poem)
```

```
reactor.run()

for poem in poems:
    print poem
```

Так что, если мы бы захотели, мы могли бы взять переиспользуемые части и положить их в общий модуль, где каждый мог бы использовать их для получения поэзии (конечно же, при условии использования Twisted).

Кстати, при тестировании клиента 3.0, можно было бы переконфигурировать поэтические сервера для отправки поэзии быстрее или большими кусками. И теперь, поскольку клиент менее разговорчив в смысле вывода, он не столь интересен в наблюдениях во время скачивания поэм.

6.3. Обсуждение

Мы можем визуализировать цепочки callback'ов в месте, когда поэма доставлена на рисунке 11:

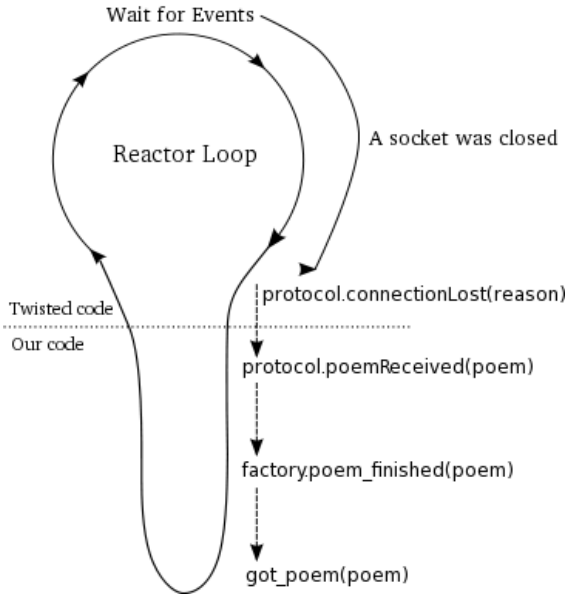

Рис. 11: поэтические callback'и

Рисунок 11 стоит созерцания. До сих пор мы изображали цепочки callback'ов, которые завершались одним вызовом "нашего кода". Но когда вы программируете с Twisted, или любой другой однопоточной реактивной системой, эти цепочки callback'ов могут включать небольшие куски вашего кода, которые делают вызовы других небольших кусков кода. Другими словами, реактивный стиль программирования не прекращается, когда он дошел до кода, который мы сами написали. В системе, основанной на реакторе, все callback'и последовательно вызываются.

Запомните этот факт при выборе Twisted для вашего проекта. Когда вы сделаете решение:

> Я собираюсь использовать Twisted!

Вы также делаете решение:

> Я собираюсь структурировать мою программу как серию асинхронной callback цепочки, вызовы элементов которой осуществляются из цикла реактора!

Теперь может быть вы не будете восклицать это так громко как это написано, но это тем не менее это повод. Это то, как работает Twisted.

Вероятно, что большинство Python программ - синхронные, и большинство Python модулей - тоже синхронные. Если бы мы писали синхронную программу и внезапно поняли, что нам нужно немного поэзии, мы могли бы использовать синхронную версию нашей функции get_poetry, добавляя несколько строк кода к нашему скрипту пободно следующим:

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

Давайте продолжим наш путь. Если позже мы решили бы, что, в конце концов, мы не хотим поэму, мы бы просто убрали бы эти строки. Но, если бы мы писали синхронную программу и затем решили бы использовать Twisted версию get_poetry, нам нужно было бы перестроить нашу программу в асинхронном стиле, используя callback'и. Нам, вероятно,нужно было бы сделать значительные изменения в коде. При этом не утверждается, что переписать программу - это ошибка. Может быть очень хорошо так и сделать. Но это не будет так просто, как добавить строку с import и дополнительным вызовом функции. Или проще: синхронный и асинхронный код не смешиваются.

Если вы новичок в Twisted и в асинхронном программирования, то рекомендуется написать несколько Twisted программ с нуля до того, как вы попробуете портировать существующий код. Этим способом вы получите ощущение от использования Twisted без дополнительной сложности, чем пытаясь подумать в обоих режимах, когда вы портируете с одного на другой.

Если, однако, ваша программа уже асинхронная, то использование Twisted может быть намного проще. Twisted интегрирован относительно хорошо с pyGTK и pyQT, python API для двух основанных на реакторе GUI библиотек.

6.4. Когда дела плохи

В клиенте 3.0 мы больше не будем определять обработчик ошибки соединения с поэтическим сервером, упущение которой вызывает гораздо больше проблем, чем в клиенте 1.0. Если мы скажем клиенту 3.0 скачать поэму из несуществуюшего сервера, то вместо того, чтобы завершиться с ошибкой, он будет крутиться в вечном цикле. Метод clientConnectionFailed все еще вызовется, но реализация по умолчанию базового класса ClientFactory ничего не делает. Поэтому мы получим никогда не вызывающийся callback got_poem, reactor никогда не остановится и мы получим еще одну ничего не делающую программу, подобно той, что мы делали в главе 2.

Ясно, что нам нужно управлять этой ошибкой, но где? Информация об ошибке соединения доставляется объекту Factory через clientConnectionFailed, поэтому мы должны начать здесь. Но предполагается, что Factory переиспользуема, и правильный путь - управлять ошибкой взависимости от контекста, в котором Factory используется. В некоторых приложениях, отсутствие поэзии может быть бедствием (Нет поэзии? Может быть равносильно критической ситуации). В других - можно продолжать работать и попробовать скачать другую поэму из другого места.

Другими словами, пользователям get_poetry нужно знать, когда дела плохи, и не только, когда все хорошо. В синхронной программе, get_poetry могла бы генерировать исключение и вызывающий код мог бы его обрабатывать с помощью конструкции try/except. Но в реактивной программе, условие ошибки должно доставляться асинхронно. После этого, мы даже не обнаружим ошибку в соединении до тех пор, пока не возвратится get_poetry. Далее одна из возможностей:

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

      callback(poem)

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

      callback(None)

    instead.
    """
```

Проверяя аргумент callback'а (например, если poem is None), клиент может определить действительно ли мы имеем поэму или нет. Это могло бы быть достаточным для нашего клиента для того, чтобы избежать вечного запуска, но это метод все еще имеет некоторые проблемы. Прежде всего, использовать None для индикации ошибки - это нечно специальное. Некоторые асинхронные API могут захотеть использовать None в качестве возвращаемого значения по умолчанию вместо условия ошибки. Во-вторых, значение None приносит очень мало информации. Оно не может сказать нам, что пошло не так, или включить объект traceback, который мы могли бы использовать для отладки. Хорошо, вторая попытка:

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

      callback(poem)

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

      callback(err)

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

Использование Exception ближе к тому, что мы используем в синхронном программировании. Теперь мы посмотрим на исключение для получения большей информации о том, что плохого произошло и освободим None для использования в качестве регулярного значения. Хотя это нормально, что когда мы сталкиваемся с исключением в Python'е, мы также получаем traceback, мы можем анализировать или печатать лог для отладки немногим позже. Traceback'и особенно полезны, поэтому мы должны их передавать наверх при использовании асинхронного программирования.

Запомните, что мы не хотим получить объект traceback в месте вызова нашего callback'а, так как это не то место, где произошла проблема. Что мы действительно хотим так это экземпляр Exception и traceback в месте, где произошло исключение (предполагая, что оно произошло, а не просто было создано).

Twisted включает абстракцию, называемую Failure, которая обертывает Exception и traceback. Строка с документацией Failure объясняет как создать этот объект. Передавая объекты Failure в callback'и, мы сохраняем информацию о traceback'е, что очень удобно при отладке.

В twisted-failure/failure-examples.py находится пример кода, который использует объекты типа Failure. В этом примере показано как Failures может сохранять информацию о traceback'е из возникшего исключения, даже вне контекста блока except. Мы не будем подробно останавливаться на создании экземпляров типа Failure. В главе 7 мы посмотрим на то, как Twisted в целом создает их для нас.

Хорошо, третья попытка:

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

      callback(poem)

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

      callback(err)

    instead, where err is a twisted.python.failure.Failure instance.
    """
```

В этой версии мы получили оба - Exception и traceback на момент проблемы. Прекрасно.

Мы практически все сделали, но у нас есть еще одна проблема. Использовать один и тот же callback для нормального и проблемного результатов - это странно. Обычно, нам нужно сделать нечто отличное в случае проблемы, чем в успешном исходе. В синхронной Python программе мы обычно управляем успешными и проблемными исходами в двух различных ветках оператора try/exception примерно так:

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

Если мы хотим сохранить такой стиль управления ошибками, то нам нужно использовать отдельную ветку в случае возникновления проблем. В асинхронном программировании отдельная ветка означает отдельный callback:

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

      callback(poem)

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

      errback(err)

    instead, where err is a twisted.python.failure.Failure instance.
    """
```

6.5. Клиент 3.1

Теперь у нас есть API с разумной семантикой управления ошибками, которую мы можем реализовать. Клиент 3.1 расположен в twisted-client-3/get-poetry-1.py. Изменения достаточно простые. PoetryClientFactory получает оба: callback и errback, и теперь в классе реализован метод clientConnectionFailed:

```
class PoetryClientFactory(ClientFactory):

    protocol = PoetryProtocol

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

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

    def clientConnectionFailed(self, connector, reason):
        self.errback(reason)
```

Поскольку clientConnectionFailed получает объект типа Failure (в аргументе reason), который объясняет почему не произошло соединение, нам нужно только передать этот аргумент в errback.

Другие изменения мы не будем обсуждать, их можно посмотреть в коде. Вы можете запустить клиент 3.1, используя порт без сервера следующим образом:

```
python twisted-client-3/get-poetry-1.py 10004
```

И вы получите следующий вывод:

```
Poem failed: [Failure instance: Traceback (failure with no frames): : Connection was refused
]
```

Это напечатано оператором print в нашем poem_failed errback'е. В этом случае Twisted просто передает исключение вместо его порождения, поэтому в данном случае мы не получаем traceback. Но traceback реально не нужен, поскольку это не ошибка, и Twisted корректно информирует нас, что мы не можем установить соединение по адресу.

6.6. Резюме

Вот что мы изучили в этой главе:

- API, которые мы пишем для Twisted программ, должны быть асинхронными.
- Мы не можем смешивать синхронный и асинхронный код.
- Таким образом, мы должны использовать callback'и в нашем коде, подобно тому как это делает Twisted.
- Мы должны управлять ошибками в callback'ах.

Значит ли, что каждый API, который мы написали с помощью Twisted должен включать два дополнительных аргумента, callback и errback? Это звучит не так приятно. К счастью, Twisted имеет абстракцию, которую мы можем использовать для того, чтобы исключить оба этих аргумента и преобрести несколько дополнительных свойств. Мы узучим это в следующей главе.

6.7. Упражнения

1. Обновите клиент 3.1 и установите timeout, который активизируется, если поэма не получена после заданного интервала времени. В этом случае вызывайте errback с пользовательским исключением. Не забудьте закрыть соединение.

2. Изучите метод trap объектов Failure. Сравните его с условием except в операторе try/except.

3. Используйте оператор print для того, чтобы убедиться, что clientConnectionFailed вызывается после выхода из get_poetry.

7. Отложенные вызовы

7.1. Обратные вызовы и их последователи

В главе 6 мы столкнулись лицом к лицу с тем фактом, что callback'и это фундамент асинхронного программирования с Twisted. Обратные вызовы вплетаются в структуру каждой нашей Twisted программы, а не являются только способом взаимодействия с реактором. Таким образом, использование Twisted или любой другой асинхронной системы, основанной на реакторе, означает организацию нашего кода определенным образом: как серию цепочек обратных вызовов, вызываемых реактором.

Даже, когда API настолько простое как наша фукнция get_poetry, требуются callback'и (фактически два callback'а): один - для нормальных результатов, другой - для ошибочных. Как Twisted программисты, мы должны будем сделать много callback'ов, и нам нужно потратить немного времени, подумав о том, как лучше использовать callback'и, и с какими подводными камнями мы можем столкнуться.

Рассмотрим следующий кусок кода, который использует Twisted версию get_poetry из клиента 3.1:

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

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

get_poetry(host, port, got_poem, poem_failed)

reactor.run()
```

Основной план здесь понятен:

1. Если мы получили поэму - напечатаем ее.
2. Если мы не получили поэму - напечатает ошибку.
3. В любом случае завершаем программу.

Синхронный аналог кода выше выглядит примерно так:

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

Таким образом callback подобен блоку else, и errback подобен except. Это означает, что вызов errback - асинхронный аналог генерации исключения, и вызов callback'а соответсвует нормальному потоку программы.

Какие различия между двумя этими версиями? Первое: в синхронной версии интерпретатор Python'а будет гарантировать, что как только get_poetry сгенерирует любой exception, в любом случае будет выполняться блок except. Если мы верим интерпретатору, который запускает код Python'а, то мы можем довериться тому, что блок except будет выполнен вовремя.

Сравните с асинхронной версией: errback poem_failed вызывается нашим кодом методом clientConnectionFailed из PoetryClientFactory. Мы, а не Python, ответственны за то, чтобы код управления ошибками запустился в случае, если что-то пошло не так. Поэтому мы должны убедиться, что управляем каждой возможной ошибкой, вызывая errback с объектом типа Failure. Иначе наша программа застынет, ожидая callback, который никогда не придет.

Это показывает различие между синхронной и асинхронной версиями. Если мы не будем заботиться об отлавливании исключений в синхронной версии (не будем использовать try/except), интерпретатор Python'а поймает их за нас и прервет выполнение программы для того, чтобы показать ошибку. Но если мы не будем заботиться о вызове функции errback в PoetryClientFactory, наша программа будет вечно работать, не зная, что что-то не так.

Ясно, что управление ошибками в асинхронной программе - важное и хитрое дело. Вы могли бы сказать, что управление ошибками в асинхронном коде действительно важнее, чем управление нормальными случаями, так как ошибки могут случаться гораздо большими способами, чем не случаться. Забывать управлять ошибочными случами - общая ошибка при программировании с помощью Twisted.

Еще один факт о синхронном коде выше: либо блок else один раз запустится, либо блок except запустится только один раз (в предположении, что синхронная версия get_poetry не входит в тело бесконечного цикла). Интерпретатор Python'а не решит внезапно запустить их оба или запустить блок else 27 раз. Тогда было бы совершенно невозможно программировать на Python't, если бы это было так!

Снова, в асинхронном случае мы ответсвенны за запущенные callback или errback. Зная себя, мы можем сделать какие-нибудь ошибки. Мы могли бы вызвать callback или errback или вызвать callback 27 раз. Это было бы нежелательно для пользователей get_poetry. Подобно блокам else и except в операторе try/except, предполагается, что либо callback, либо errback запустится один раз для каждого определенного вызова get_poetry. Либо мы получим поэму, либо - нет.

Представьте попытку отладить программу, которая делает три поэтических запроса и получает семь вызовов callback'ов и два вызова errback'а. Где вы бы начали отлаживать? Вы, вероятно, закончили бы писать свои callback'и и errback'и для обнаружения, когда они вызываются во второй раз для одного и того же вызова get_poetry, и сгенерировали исключение в таком случае.

Еще одно наблюдение: обе версии имеют дублированный код. Асинхронная версия имеет два вызова reactor.stop и синхронная версия имеет два вызова sys.exit. Мы могли улучшить синхронную версию следующим образом:

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

```
    print >>sys.stderr, 'try again later?'
else:
    print poem

sys.exit()
```

Можем ли мы улучшить асинхронную версию подобным образом? Не очень ясно, что мы можем, так как callback и errback - две разные функции. Должны ли мы вернуться обратно к одному callback'у, чтобы сделать это возможным?

Хорошо, давайте вспомним то, что мы открыли о программировании с callback'ми:

1. Вызов errback'ов очень важен. Поскольку errback'и являются заменой блокам except, пользователям нужно их учитывать. Это не опциональное свойство нашего API.

2. Не вызывать callback'и в неправильный момент также важно как и вызывать их в правильный. Обычно, callback и errback взаимозаменяемы и вызываются только один раз.

3. Улучшать код может быть сложнее в случае использования callback'ов.

Мы скажем больше о callback'ах в следующих главах, но сейчас этого достаточно для того, чтобы увидеть почему Twisted может иметь управляющую абстракцию.

7.2. Deferred

Поскольку callback'и много используются в асинхронном программировании, и поскольку их корректное использование, как мы увидели, может быть непростым, разработчики Twisted создали абстракцию, называемую Deffered, для того, чтобы упросить программирование с использованием callback'ов. Класс Deferred определен в twisted.internet.defer.

Слово "deferred" (отложенный) - это либо глагол, либо прилагательное в повседневном английском, поэтому это может звучать немного странно при использовании его как существительного. Зная это, с этого момента, когда используется фраза "deferred", то это означает экземпляр класса Deferred. Мы обсудим то, почему класс называется Deferred в следующей главе. Нам может помочь добавление слова "результат" к каждой фразе, то есть "отложенный результат". Как мы увидим, это действительно так.

deferred содержит пару callback цепочек: одну для нормальных результатов, другую - для ошибочных. Вновь созданный deferred имеет пустые цепочки. Мы можем заселить цепочки, добавляя callback'и и errback'и, и затем активизировать deferred с нормальным результатом (здесь ваша поэма!) или исключением (я не смог получить поэму, и вот почему). Активизированный deferred будет вызывать либо соответсвующие callback'и, либо errback'и в порядке, в котором они были добавлены. Рисунок 12 иллюстрирует экземпляр Deffered с его callback и errback цепочками:

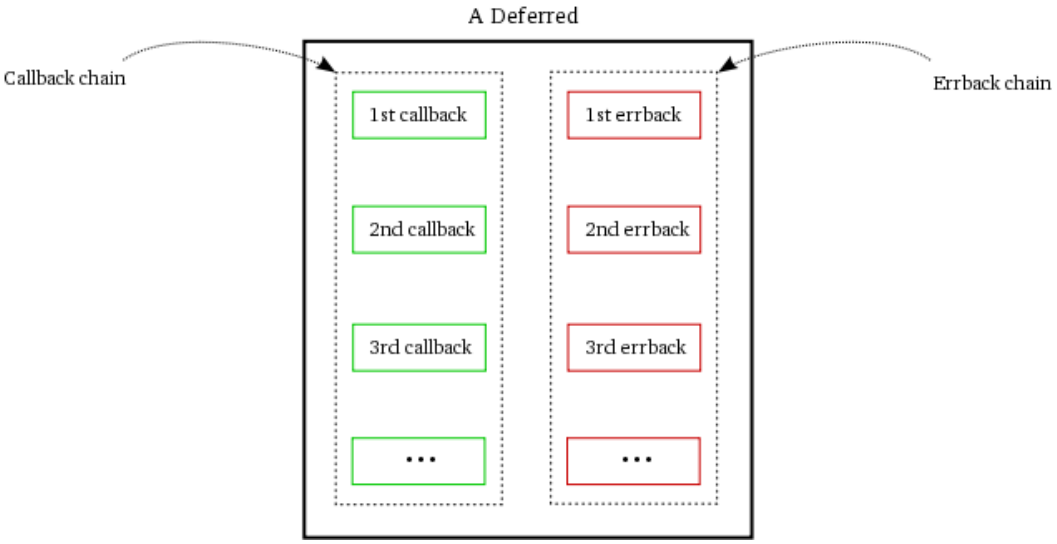

Рис. 12: Deferred

Давайте опробуем это. Поскольку deferred'ы не используют reactor, мы можем их попробовать, не запуская цикла.

Вы могли бы заметить, что метод setTimeout класса Deferred использует reactor. Эта часть устарела, и, возможно, прекратит существовать в следующем релизе. Сделайте вид, что этого здесь нет и не используйте.

Наш первый пример находится в twisted-deferred/defer-1.py:

```
from twisted.internet.defer import Deferred

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

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

d = Deferred()

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

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

print "Finished"
```

Этот код создает новый deferred, добавляет пару callback/errback с методом addCallbacks, и затем активизирует цепочку с «нормальным результатом» с помощью метода callback. Запустите код, и он напечатает следующее:

```
Your poem is served:
This poem is short.
Finished
```

Это достаточно просто. Вот, что стоит отметить:

1. Подобно парам callback/errback, которые мы использовали в клиенте 3.1, callback'и, которые мы добавили к этому deferred'у, берут в качестве аргумента либо нормальный результат, либо ошибку. Оказывается, что deferred'ы поддерживают callback'и и errback'и с несколькими аргументами, но они всегда имеют по меньшей мере один аргумент: нормальный результат или результат с ошибкой.
2. Мы добавляем callback'и и errback'и в deferred парами.
3. Метод callback активизирует deferred с нормальным результатом, единственным аргументом метода.
4. Смотря на порядок вывода print'а, можно заметить, что активизация deferred'а вызывает callback'и немедленно. Здесь нет ничего асинхронного и не могло бы быть, поскольку reactor не запущен. Это реально сводится к обычному вызову функции в Python'е.

Хорошо, давайте нажмем на другую кнопку. Пример в twisted-deferred/defer-2.py активизирует errback цепочку deferred'а:

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

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

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

d = Deferred()

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

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

print "Finished"
```

После запуска этой программы, мы получим вывод:

```
No poetry for you.
Finished
```

Таким образом, активизация цепочки errback состоит только в вызове метода errback вместо callback, где в качестве аргумента использовали результат с ошибкой. Также как с callback'ми, errback'и вызываются сразу же после активизации.

В предыдущем примере мы передаем объект Failure к методу errback подобно тому, как мы делали это в клиенте 3.1. Это отлично, но deferred обертывает для нас обычные Exception в Failure. Мы можем увидеть это в twisted-deferred/defer-3.py:

```
from twisted.internet.defer import Deferred

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

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

d = Deferred()

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

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

Здесь мы подставляем обычный Exception в метод errback. В errback'е, мы печатаем класс и сам результат. Мы получаем такой вывод:

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

Это означает, что когда мы используем deferred'ы, мы можем вернуться обратно к работе с обычными исключениями, и объекты типа Failure создадутся для нас автоматически. deferred будет гарантировать, каждый errback вызывается с действительным экземпляром Failure.

Мы попробовали нажать на кнопку callback, и мы попробовали нажать на кнопку errback. Подобно любым хорошим инженерам, вы вероятно хотите нажать еще раз. Чтобы сделать код короче, мы будем использовать ту же функцию для обоих callback и errback. Только запомните, что они получают различные возвращаемые значения: один - результат, другой - ошибка. Посмотрите twisted-deferred/defer-4.py:

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

Теперь мы получаем этот вывод:

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

Это интересно! deferred не позволяет нам активизировать нормальный результат во второй раз. Фактически, deferred не может быть активизирован во второй раз, что демонстрируется в следующих примерах:

- twisted-deferred/defer-4.py
- twisted-deferred/defer-5.py
- twisted-deferred/defer-6.py
- twisted-deferred/defer-7.py

Заметьте, что последние операторы print никода не вызовутся. Методы calback и errback вызывают исключения, чтобы дать нам понять, что мы уже активизировали этот deferred. Deferred'ы помогают нам избежать ловушек, которые мы идентифицировали как повторный вызов. Когда мы используем deferred для управления нашими callback'ми, мы просто не можем сделать ошибку вызвав одновременно callback и errback, или вызывая callback 27 раз. Мы можем попробовать, но deferred сгенерирует исключение, вместо того, чтобы подставить нашу ошибку в сами callback'и.

Могут ли deferred'ы помочь нам улучшить асинхронный код? Рассмотрим пример в twisted-deferred/defer-8.py:

```
import sys

from twisted.internet.defer import Deferred

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

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

d = Deferred()

d.addCallbacks(got_poem, poem_failed)

from twisted.internet import reactor

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

reactor.run()
```

Код выше - это наш первоначальный пример, с дополнительным запуском реактора. Заметьте, что мы используем callWhenRunning для активизации deferred'a после запуска реактора. Мы пользуемся тем фактом, что callWhenRunning принимает дополнительные позиционные и ключевые (keyword) аргументы для того, чтобы подставить их в callback

во время его запуска. Многие Twisted API, регистрирующие callback'и, следуют этому соглашению, включая API для добавления callback'ов в deferred'ы.

И callback, и errback останавливают reactor. Поскольку deferred'ы поддерживают цепочки callback'ов и errback'ов, мы можем улучшить общий код, создав вторую ссылку в цепочках, что проиллюстрировано в twisted-deferred/defer-9.py:

```
import sys

from twisted.internet.defer import Deferred

def got_poem(poem):
    print poem

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

def poem_done(_):
    from twisted.internet import reactor
    reactor.stop()

d = Deferred()

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

from twisted.internet import reactor

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

reactor.run()
```

Метод addBoth добавляет одну и ту же функцию в обе цепочки callback и errback. И, в конце концов, мы можем улучшить асинхронный код.

Замечание: существует тонкость в способе, когда deferred действительно выполнял бы свою errback цепочку. Мы обсудим это в следующей главе, но запомните, что еще есть что изучить о deferred'ах.

7.3. Резюме

В этой главе мы анализировали программирование с использованием callback'ов и идентифицировали некоторые потенциальные проблемы. Мы также увидели, как класс Deferred может выручить нас в следующих ситуациях:

1. Мы не можем игнорировать errback'и, так как они требуются для любого асинхронного API. Deferred'ы имеют встроенную поддержку errback'ов.

2. Вызов callback'ов несколько раз может вызвать зависание программы, и это трудно отлаживать. Deferred'ы могут активизироваться только один раз, что делает их похожими на семантику оператора try/except.

3. Программирование с помощью обычных callback'ов делает рефакторинг сложным. С помощью deferred'ов, мы можем рефакторить, добавляя ссылки в цепочку и перемещая код из одной ссылки в другую.

Мы еще не завершили рассмотрение deferred'ов, так как существует большое количество деталей их обоснования и поведения. Но теперь мы имеем достаточно для того, чтобы начать использовать их в нашем поэтическом клиенте, и мы сделаем это в следующей главе.

7.4. Упражнения

1. Последний пример игнорирует аргумент poem_done. Распечатайте его. Сделайте так, чтобы got_poem возвращал значение и посмотрите на изменения в аргументе poem_done.

2. Поменяйте два последних примера с deferred'ми и активизируйте errback цепочки. Убедитесь, что активизировали errback с Exception.

3. Прочитайте строки с документацией для методов addCallback и addErrback класса Deferred.

8. Отложенная поэзия

8.1. Клиент 4.0

Теперь, когда мы знаем что-то о deferred'ах, мы можем переписать наш Twisted поэтический клиент с их использованием. Вы можете найти клиент 4.0 в twisted-client-4/get-poetry.py.

Нашей функции get_poetry не нужны больше аргументы callback или errback. Вместо этого она возвращает новый deferred, к которому пользователь может прикрепить необходимые callback'и и errback'и.

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

Наш объект factory инициализируется с deferred'ом вместо пары callback/errback. Как только мы получили поэму или обнаружили, что не можем присоединиться к серверу, deferred активизируется либо поэмой, либо ошибкой:

```
class PoetryClientFactory(ClientFactory):

    protocol = PoetryProtocol

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

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

    def clientConnectionFailed(self, connector, reason):
        if self.deferred is not None:
            d, self.deferred = self.deferred, None
            d.errback(reason)
```

Заметьте способ, которым особобождается ссылка на deferred после активизации. Этот подход можно найти в нескольких местах исходных кодов Twisted, и этот способ помогает убедиться в том, что мы не активизируем один и тотже deferred дважды. Это также упрощает работу сборщика мусора в Python'е.

И снова, нет необходимости менять PoetryProtocol, он хорош как есть. Все, что осталось - обновить функцию poetry_main:

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

    from twisted.internet import reactor

    poems = []
    errors = []

    def got_poem(poem):
        poems.append(poem)

    def poem_failed(err):
        print >>sys.stderr, 'Poem failed:', err
        errors.append(err)

    def poem_done(_):
        if len(poems) + len(errors) == len(addresses):
            reactor.stop()

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

    reactor.run()

    for poem in poems:
        print poem
```

Заметьте, что мы пользуемся способностью создавать цепочки вызовов в deferred'е для рефакторинга вызова poem_done из нашего первоначального callback'а и errback'а.

Поскольку deferred'ы используются повсеместно в Twisted, то распостраненной практикой является использование однобуквенной локальной переменной d для хранения deferred, над которым вы сейчас работаете. Для длительного хранения, подобно атрибутам объекта, зачастую используется название "deferred".

8.2. Обсуждение

С нашим новым клиентном асинхронная версия get_poetry принимает ту же информация, что и асинхронная версия: только адрес поэтического сервера. Синхронная версия возвращает поэму, в то время как асинхронная версия возвращает deferred. Возвращать deferred - это типично для асинхронных API в Twisted и программах, написанных с помощью Twited, и это указывает на другой способ концептуализации deferred'ов:

Объект типа Deferred представляет "асинхронный результат" или "результат в пути".

Мы можем сравнить эти два стиля программирования на рисунке 13:

Возвращая deferred, асинхронное API сообщает следующее пользователю:

Я асинхронная функция. Требуемый результат еще не получен. Но, когда он будет получен, я активизирую цепочку callback'ов этого deferred'а с результатом. С другой стороны, если что-то пошло не так, то вместо этого я активизирую цепочку errback этого deferred'а.

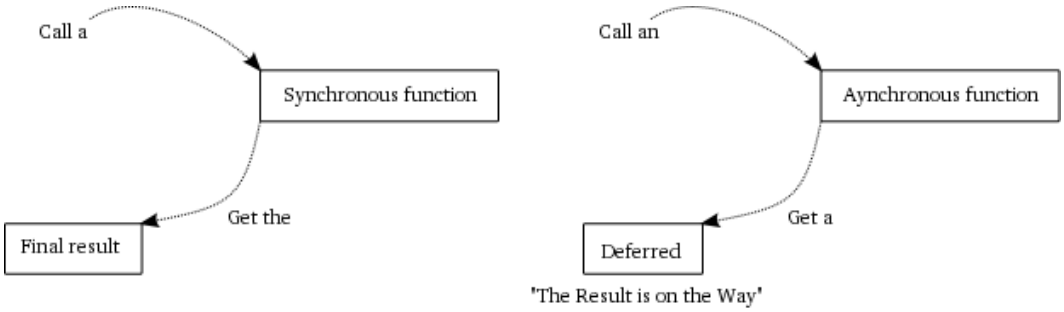

Рис. 13: sync против async

Конечно же, эта функция буквально сама не будет активизировать deferred, ее выполнение уже завершилось. Скорее, функция установила движение в цепочке событий, которые в конечном итоге приведут к активизации deferred'а.

Таким образом deferred'ы - способ "сдвига по времени" результатов функций для приспосабливания к нуждам асинхронной модели. И deferred, возвращенный функцией, - это знак того, что функция асинхронная, воплощение будущего результата и обещание того, что результат будет доставлен.

В случае синхронной функции возвратить deferred возможно, так что технически возврат deferred'а означает, что функция потенциально асинхронная. В следующих главах мы увидим примеры синхронных функций, возвращающих deferred'ы.

Поскольку поведение deferred'ов хорошо определено и хорошо изучено (для людей, имеющих некоторый опыт программирования с Twisted), возвращая deferred'ы из наших собственных API, вы упрощаете другим Twisted программистам понимание и использование вашего кода. Без deferred'ов, каждая Twisted программа или даже каждый Twisted компонент, может иметь свой собственный уникальный метод для управления callback'ми, которые вам надо изучить, для того, чтобы их использовать.

8.3. Связь между deferred'ами, callback'ми, реактором

При первом изучении Twisted, общераспостраненной ошибкой присваивать большую функциональность deferred'ам, нежели они в действительности имеют. Особенно, зачастую предполагается, что добавление функции в цепочку deferred'а, автоматически делает эту функцию асинхронной. Это могло бы навести на мысль, что вы могли бы использовать, скажем, os.system с Twisted добавляя его в deferred с помощью addCallback.

Думается, что эта ошибка вызвана попыткой изучить Twisted без первоначального изучения асинхронной модели. Поскольку типичный Twsisted код использует много deferred'ов, и только иногда ссылается на reactor, может показаться, что deferred'ы делают всю работу. Если вы читали введение с самого начала, то должно быть ясно, что это далеко от реальности. Хотя Twisted составлен из многих частей, которые работают вместе, первоначальная ответственность за реализацию асинхронной модели ложится на reactor. Deferred'ы - полезная абстракция, но мы написали несколько версий нашего поэтического Twisted клинта без их использования.

Давайте посмотрим на stack trace в месте, когда вызывается наш первый callback. Запустите пример программы из twisted-client-4/get-poetry-stack.py с адресом запущенного поэтического сервера. Вы получите следующий вывод:

```
File "twisted-client-4/get-poetry-stack.py", line 129, in
  poetry_main()
File "twisted-client-4/get-poetry-stack.py", line 122, in poetry_main
```

```
    reactor.run()

  ... # some more Twisted function calls

    protocol.connectionLost(reason)
  File "twisted-client-4/get-poetry-stack.py", line 59, in connectionLost
    self.poemReceived(self.poem)
  File "twisted-client-4/get-poetry-stack.py", line 62, in poemReceived
    self.factory.poem_finished(poem)
  File "twisted-client-4/get-poetry-stack.py", line 75, in poem_finished
    d.callback(poem) # here's where we fire the deferred

  ... # some more methods on Deferreds

  File "twisted-client-4/get-poetry-stack.py", line 105, in got_poem
    traceback.print_stack()
```

Это очень похоже на stack trace, который мы создали для клиента 2.0. Визуализируем последний trace на рисунке 14:

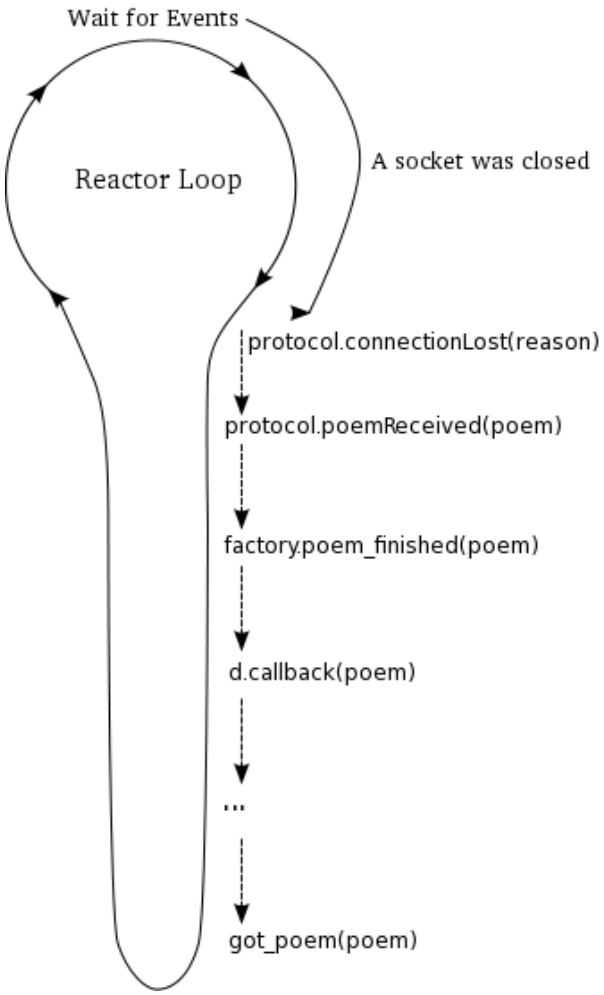

Рис. 14: callback с использованием deferred'a

И снова, это подобно нашим предыдущим Twisted клиентам, хотя визуальное представление становится неясным. Одно замечание, не касающееся рисунка: цепочка callback'ов

выше не возвратит управление реактору до того, как не вызовется второй callback в deferred'е (poem_done), что произойдет сразу после возврата из первого callback'а (got_poem).

Есть еще одно отличие в нашем новом stack trace'е. Граница, разделяющая "код Twisted" от "нашего кода" немного размыта, поскольку deferred'ы - это в реальности Twisted код. Это чередование Twisted и пользовательского кода в callback-цепочке является общепринятым в большинстве программ, написанных с использованием Twisted, активно использующих различные Twisted абстракции.

Используя deferred, мы добавили несколько шагов в callback-цепочку, которая начинается в реакторе Twisted, но мы не поменяли фундаментальных механизмов асинхронной модели. Вспомним факты о программировании с использованием callback'ов:

1. Только один callback выполняется в один момент.
2. Когда выполняется reactor, наши callback'и - нет.
3. И наоборот.
4. Если наш callback блокируется, то вся программа блокируется.

Присоединение callback'а к deferred'у никак не меняет эти факты. В особенности, callback, который блокируется, все еще будет блокироваться, даже после присоединения к deferred'у. Поэтому deferred заблокируется при активизации (d.callback), поэтому все остальное тоже заблокируется. Поэтому мы заключаем, что:

Deferred'ы - решение (разработанное разработчиками Twisted) проблемы управления callback'ми. Они не являются ни способом мзбежать callback'ов, ни способом превратить блокирующиеся callback'и в неблокирующиеся.

Мы можем подтвердить наш последний вывод созданием deferred'а с блокирующм callback'ом. Рассмотрим пример кода в twisted-deferred/defer-block.py. Второй callback блокируется, используя функцию time.sleep. Если вы запустите этот скрипт и проверите порядок операторов print, то будет ясно, что блокирующийся callback также блокируется, находясь внутри deferred'а.

8.4. Резюме

Возвращая Deferred, функция говорит пользователю "Я асинхронная" и обеспечивает механизм (добавь здесь callback'и и errback'и!) получения асинхронного результата в момент, когда он прибудет. Deferred'ы повсеместно используются в Twisted, поэтому с ними надо ознакомиться и знать, как применять.

Клиент 4.0 - первая версия нашего Twisted поэтического клиента, которая действительно написана в стиле Twisted, используя deferred'ы в качестве возвращаемых значений асинхронного вызова функций. Есть еще другие Twisted API, которые мы могли бы использовать для того, чтобы сделать код понятнее, но я думаю, что он представляет достаточно хороший пример того, как написать простые Twisted программы, по меньшей мере на стороне клиента. В конечном итоге, мы также перепишем наш поэтический сервер с использованием Twisted.

Но мы еще не окончили с deferred'ми. Для относительно небольшого куска кода, класс Deferred предоставляется удивительный ряд возможностей. Мы остановимся подоробнее на некоторых из этих возможностях и их мотифированности в следующей главе.

8.5. Упражнения

1. Обновите клиент 4.0 для установки timeout'а, в случае, если поэмы не была получена за заданный период времени. Активизируйте в этом случае errback deferred'а, используя пользовательское исключение. Не забудьте закрыть соединение.

2. Обновите клиент 4.1 для того, чтобы распечатать соответствующий адрес сервера при ошибки скачивания поэмы так, чтобы пользователь смог сказать какой сервер виновник. Не забудьте, что вы можете добавить дополнительные позиционные и keyword-аргументы, при присоединении callback'ов и errback'ов.

9. Deferred'ы, часть вторая

9.1. Дальнейшие выводы про callback'и

Мы ненадолго приостановимся, чтобы снова подумать о callback'ах. Хотя сейчас мы достаточно знаем о deferred'ах, для того, чтобы писать асинхронные программы в стиле Twisted, класс Deferred предоставляет больше свойств, которые имеют значение в более сложных настройках. Поэтому мы придумаем более сложные настройки и посмотрим какие преимущества мы получаем при программировании с использованием callback'ов. Затем мы изучим то, как deferred'ы используют эти преимущества.

Для мотивации нашей дискуссии, добавим гипотетическое возможность к нашему поэтическому клиенту. Допустим, что некий профессор изобрел новый алгоритм, имеющий отношение к поэзии: Byronification Engine. Ловкий алгоритм берет на вход одну поэму и производит новую поэму, подобную первоначальной, но написанной в стиле Лорда Байрона. Помимо этого, наш профессор любезно предоставляет ссылку на реализацию на Python'е со следующим интерфейсом:

```
class IByronificationEngine(Interface):

    def byronificate(poem):
        """
        Return a new poem like the original, but in the style of Lord Byron.

        Raises GibberishError if the input is not a genuine poem.
        """
```

Подобно большиству передовых программ, реализация имеет какие-то баги. Это означает, что в дополнение к документированным исключениям, метод byronificate иногда выкидывает произвольные исключения, когда натыкается на случаи, которые профессор забыл отловить.

Мы будем также предполагать, что движок выполняется достаточно быстро, поэтому мы можем просто его вызывать из основного треда, не беспокоясь о связывании с реактором. Вот как хочется, чтобы работала программа:

1. Попробовать скачать поэму.
2. Если скачивание не произошло из-за ошибки, сказать пользователю, что мы не можем получить поэму.
3. Если мы получили поэму, преобразовать ее с помощью движка Byronification.
4. Если движок выкинул исключение GibberishError, сказать пользователю, что мы не можем получить поэму.
5. Если движок выкинул какое-то другое исключение, сохранить первоначальную поэму.
6. Если мы имеем поэму - распечать ее.
7. Завершить программу.

Здесь идея в том, что GibberishError означает, что в конце концов мы не получили действительную поэму, поэтому мы просто скажем пользователю, что скачивание завершилось с ошибкой. Это не особо полезно при отладке, но наши пользователи просто

хотят знать: мы получили поэму или нет. С другой стороны, если движок завершается с ошибкой по какой-то причине, то мы будем использовать поэму, которую мы получили из сервера. В конце концов, какая-нибудь поэзия лучше, чем никакой, даже если она не под торговой маркой ”стиль Байрона”.

Далее синхронная версия нашего кода:

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

sys.exit()
```

Это набросок программы мог бы быть проще с некоторыми улучшениями, но он достаточно ясно иллюстрирует логический поток. Мы хотим обновить наш последний поэтический клиент (который использует deferred’ы), чтобы реализовать эту же схему. Но мы не будем делать до следующей главы. Сейчас, вместо этого, давайте представим то, как мы могли бы сделать это с клиентом 3.1, нашим последний клиентом, который не использует deferred’ы. Предположим. что мы не заботимся управлением исключениями, но вместо этого мы меняем got_poem callback следующим образом:

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

Что происходит, когда метод byronificate генерирует GibberishError или какое-то другое исключение? Смотря на рисунок 11, мы можем увидеть что:

1. Исключение будет передаваться к callback’у poem_finished в factory, методу, который в действительности вызывает callback.
2. Поскольку poem_finished не ловит исключения, далее продолжится выполнение в методе poemReceived протокола.
3. Затем в connectionLost также протокола.
4. Затем управление перейдет самому Twised, окончательно завершись в реакторе.

Как мы изучили, reactor ловит и логирует исключения, а не завершается с ошибкой. Но то, что он определенно не сделает - не скажет пользователю, что мы не смогли скачать поэму. reactor ничего не знает о поэмах или об исключениях типа GibberishError, это код общего назначения для всех видов сетевых взаимодейтсвий, даже не относящихся к поэзии.

Заметьте, как на каждом шаге в списке выше, исключение перемещается к коду, имеющему более и более общее назначение. В got_poem нет кода специфичного для нашего клиента управления исключениями. Эта ситуация противоположна тому, как исключения распостраняются в синхронном коде.

Давайте посмотрим на рисунок 15, иллюстрацию стека вызова, которую мы можем увидеть в синхронном поэтическом клиенте:

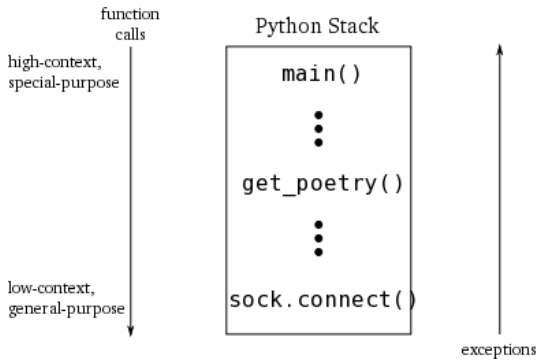

Рис. 15: Синхронный код и исключения

Функция main - это ”высокоуровневая”, что означает, что она много знает о всей программе, почему она выходит, предполагаемое поведение в целом. Типично, main имела бы доступ к опциям командной строки, которая означает то, как пользователь хочет, чтобы программа работала (и, возможно, что надо делать, если что-то пошло не так).

С другой стороны, метод connect модуля socket, ”низкоуровневый”. Все что он знает - это то, что предполагается присоединиться по некоторому сетевому адресу. Он даже не знает, что на другом конце, или почему нам нужно соединиться прямо сейчас. Но connect - метод общего назначения, вы можете его использовать, невзирая на тип сервиса, к которому вы присоединяетесь.

А метод get_poetry находится по середине. Он знает, что он получает некоторую поэзию (и это единственное, чем он занимается), но не знает, что должно произойти, если он не сможет получить поэзию.

Таким образом, исключение, произошеднее в connect’е, будет перемещаться вверх по стеку, из низкоуровнего контекста и кода общего назначения в высокоуровневый контекст и кода специального назначения до тех пор, пока он не достигнет некоторого кода с контекстом, который будет знать, что делать, когда что-то пошло не так (или оно поймается интерпретатором Python’а и программа развалится).

Конечно же, исключение в действительности просто перемещается вверх по стеку, не разыскивая, что является более ”высококонтекстным кодом”. Это потому что в типичной синхронной программе ”ввер по стеку” и ”в направлении высокоуровневого контекста” - это одно и то же.

Теперь вспомним гипотетическую модификацию клиента 3.1 выше. Стек вызова, который мы анализировали, изображен на рисунке 16, сокращенный только до нескольких функций:

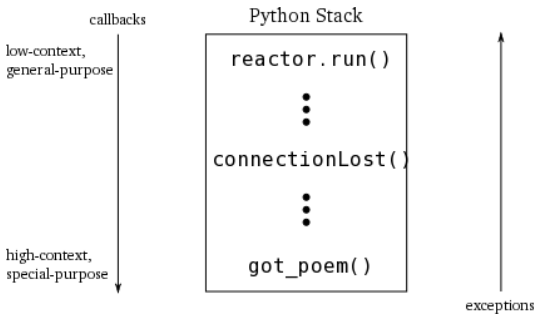

Рис. 16: Асинхронные callback'и и исключения

Проблема теперь ясна: во время callback'а, низкоуровневый код (reactor) вызывает высокоуровневый код, который в свою очередь может вызвать более высокоуровневый код и так далее. Таким образом, если происходит исключение, и оно немедленно не обрабатывается примерно в том же stack frame'е, где оно произошло, то оно, вероятно, не будет обработано вовсе. Поскольку каждый раз, когда исключение перемещается вверх по стеку, оно перемещается к низкоуровневому коду, который еще менее вероятно знает, что делать.

Как только исключение перейдет в Twisted код, то все пропало. Исключение не будет обработано, оно будет только замечено (когда reactor окончательно его поймает). Поэтому, когда мы программируем с обычными callback'ми (без использования deferred'ов), мы должны тщательно отлавливать каждое исключение до того, как оно вовзвратится в Twisted, по меньшей мере, если мы хотим иметь какой-то шанс управления ошибками согласно нашим правилам. Это также касается ошибок, вызванных нашими багами!

Поскольку баг может присутствовать в любом месте нашего кода, нам нужно было бы оборачивать каждый callback, который мы пишем, в дополнительный уровень из операторов try/except так, чтобы исключения из наших нелепостей, могли быть обработаны. Тоже самое касается наших errback'ов, поскольку код для управления ошибками, может также иметь баги.

И это не особо хорошо.

9.2. Прекрасная структура Deferred'ов

Оказывается, класс Deferred помогает решить нам эту проблему. В момент, когда deferred вызывает callback и errback, он ловит любое исключение, которое могло бы произойти. Другими словами, deferred действует как "внешний уровень" из операторов try/except, так что, в конце концов, нам не нужно писать этот уровень, в случае использования deferred'ов. Но что делает deferred с исключением, которые он ловит? Все просто: он подставляет исключение (в виде Failure) следующему errback'у в цепочке.

Так как первый errback, который мы добавляем в deferred, является тем местом, где обрабатывается какое-нибудь условие ошибки, которое сигнализируется вызовом метода deferred'а errback. Второй errback будет обрабатывать исключение, возникшее или в первом callback'е, или в первом errback'е, и так далее вниз по цепочке.

Вспомним 12, визуальное представление deferred'а с некоторыми callback'ми и errback'ми в цепочке. Давайте назовем первую пару callback/errback этапом 0, следующую - 1 и т.д.

В заданной стадии N, если callback или errback (какой бы ни выполнялся) завершаются с ошибкой, то будет вызван errback из этапа N+1 с соответсвующим объектом типа Failure, и callback из этапа N+1 не будет вызван.

Подставляя исключения, сгерерированных callback'ми, "вниз по цепочке", deferred перемещает исключения в направлеии "высокоуровневого контекста". Это также означает, что вызывающиеся методы deferred'а callback и errback никогда не сгенерируют исключение тому, кто их вызвал (конечно, если вы активизируете deferred только один раз!), поэтому низкоуровневый код может безопасно активизировать deferred, не заботясь об отлавливании исключений. В то время как, высокоуровневый код поймает исключения, добавляя errback'и в deferred (с помощью, например, addErrback).

В синхронном коде, исключения прекращают распостраняться после их первого отлавливания. А как же errback сигнализирует о том, что он "поймал" ошибку? Также просто: не генерируя исключение. В этом случае, выполнение переключится на callback-цепочку. Так что если на заданном этапе N, либо callback, либо errback завершаются успешно (например, не генерируют исключение), то вызывается callback из этапа N+1 со значением, возвращенным из этапа N, и errback из этапа N+1 не вызывается.

Давайте подведем итоги того, что мы знаем об активизации deferred'а:

1. deferred содержит цепочку упорядоченных пар callback/errback (этапов). Пары находятся в порядке, в котором они добавлялись в deferred.

2. Этап 0, первая пара callback/errback, вызывается, когда активизируется deferred. Если deferred активизируется с методом callback, то вызывается callback из этапа 0. Если deferred активизируется методом errback, то вызывается errback из этапа 0.

3. Если на этапе N происходит ошибка, то вызывается errback из этапа N+1 с исключением (обернутым в Failure) в качестве первого аргумента.

4. Если этап N успешно завершается, то вызывается callback из этапа N+1 со значением, возвращенным этапом N+1, в качестве первого аргумента.

Этот шаблон проиллюстрирован на рисунке 17:

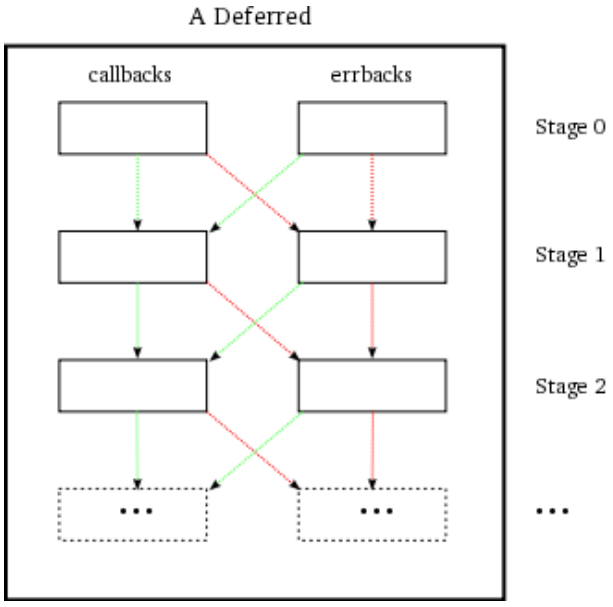

Рис. 17: Поток управления в deferred'е

Зеленые линии показывают, что происходит, когда callback или errback успешно завершаются, и красные линии, если завершаются с ошибками. Линии показывают оба поток управления и исключений и возвращаемые значения вниз по цепочке. Рисунок 17 показывает все возможные пути, которыми deferred может быть пройти, но только один путь возьмется из определенного случая. Рисунок 18 показывает один возможный путь для ”активизации”:

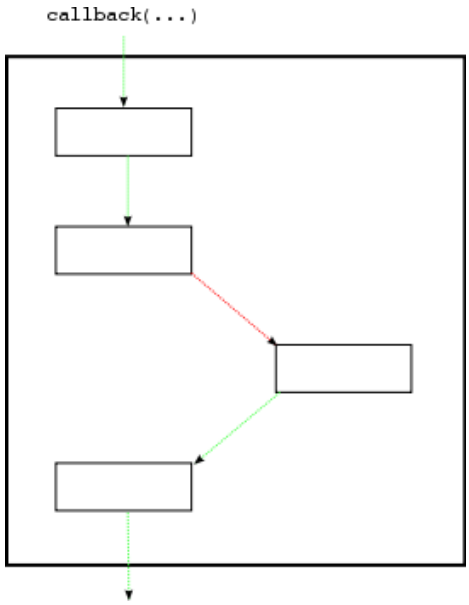

Рис. 18: Один из возможных способов активизации deferred’a

На рисунке 18 вызывается метод callback deferred’a, который вызывает callback из этапа 0. Этот callback успешно завершается, поэтому управление (и возвращаемое значение из этапа 0) передается callback’у из этапа 1. Но callback из этапа 1 завершается с ошибкой (генерирует исключение), поэтому управление переключается на errback из этапа 2. errback ”обрабатывает” ошибку (не генерирует исключение), поэтому контроль переходит обратно к callback’у из этапа 3.

Заметьте, что любой путь из рисунка 17 будет проходить через каждый этап цепочки, но только один из пары callback/errback для каждой стадии будет вызван.

На рисунке 18, мы показали, что на этапе 3 callback успешно завершается, нарисовав выходящую зеленую стрелку, но поскольку нет больше этапов в deferred’e, то результат этапа 3 никуда в действительности не отправится. Если callback завершается успешно, то это в действительности не проблема, но что, если callback завершился с ошибкой? Если последний этап в deferred’e завершился с ошибкой, то мы говорим, что ошибка не обработывается, поскольку нет errback’a, который ее ”поймает”.

В синхронном коде необработанные исключения рушат интерпретатор, и необработанные исключение в обычных callback’ах асинхронного кода ловятся реактором и логируются. Что происходит с необработанными исключениями в deferred’ах? Давайте попробуем так сделать и посмотрим. Посмотрите на пример в twisted-deferred/defer-unhandled.py. Этот код активизирует deferred с одним callback’ом, который всегда генерирует исключение. Далее вывод программы:

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

```
--- <exception caught here> ---
  ...
exceptions.Exception: oops
```

Надо отметить следующее:

1. Последний оператор print выполняется, так что программа не рушится исключением.
2. Это означает, что traceback просто печатается, и это не разрушенный интерпретатор.
3. Текст traceback'а говорит нам, где deferred сам поймал исключение.
4. Сообщение "Unhandled" печатается после "Finished".

Поэтому, когда мы используем deferred'ы, необработанные исключения в callback'ах все еще будут замечены для целей отладки, но они не будут рушить программу (фактически, они не будут передаваться реактору, deferred'ы первыми их поймают). Кстати, причина почему "Finish" печатается первым связана с тем, что сообщение "Unhandled" в действительности не печаетается до тех пор, пока deferred не утилизируется сборщиком мусора. Мы увидим причину в следующей главе.

В синхронном коде мы можем "перевызвать" исключение, используя ключевое слово raise без аргументов. Делая так, мы вызываем первоначальное исключение, которое мы обрабатывали, что позволяет нам произвести какие дополнительные действия без полной его обработки. В свою очередь, мы может сделать тоже самое с errback'ом. deferred будет считать, что callback/errback завершился с ошибкой, если:

- callback/errback вызывает любой тип исключения, или
- callback/errback возвращает объект типа Failure.

Поскольку первый аргумент errback'а всегда типа Failure, errback может "перевызвать" исключение, возвратив свой первый аргумент, после выполнения какого-либо действия.

9.3. Callback'и и Errback'и, по двое

Одно должно быть ясно из обсуждения выше - это то, что порядок, в котором вы добавляете callback'и и errback'и в deferred, определяет порядок, в котором они будут активизированы. Что еще должно быть ясно - это то, что в deferred'е callback'и и errback'и всегда появляются парами. Есть 4 метода в классе Deffered, которые вы можете использовать для добавления пар в цепочку:

1. addCallbacks
2. addCallback
3. addErrback
4. addBoth

Очевидно, что первый и последний методы добавляют пару в цепочку. Но два других метода также добавляют пару callback/errback. Метод addCallback добавляет явный

callback (тот, который вы передали в метод) и неявный ”сквозной” errback. Сквозная функция - это функция-заглушка, которая просто возвращает свой первый аргумент. Поскольку первый аргумент в errback’е всегда типа Failure, сквозной errback всгда завершается с ошибкой и отправляет ошибку следующему errback’у в цепочке.

Как вы несомненно догадались, функция addErrback добавляет явный errback и неявный сквозной callback. И поскольку первый аргумент в callback’е никогда не является Failure, сквозной callback отправляет свой результат следующему callback’у в цепочке.

9.4. Deferred симулятор

Хорошая идея познакомиться о том, как deferred’ы активизируют свои callback’и и errback’и. В twisted-deferred/deferred-simulator.py находится python скрипт, являющийся ”deferred симулятором”, небольшая программа, которая позволяет вам раскрыть то, как deferred’ы активизируют свои цепочки. Когда вы запустите скрипт, он попросит вам ввести список пар callback/errback в одну строку. Для каждого callback’а или errback’а вы задаете одно из:

- Он возвращает заданное значение (выполнение без ошибок)
- Он вызывает данное исключение (выполнение с ошибкой)
- Он возвращает свой аргумент (сквозное выполнение)

После того, как вы ввели все пары, и вы хотите симулировать, скрипт напечатает диаграмму, показывающую содержимое цепочки и шаблоны активизации для методов callback и errback. Вы можете использовать терминальное окно, которое настолько широко, насколько это возможно, для того, чтобы увидеть все корректно. Вы также можете использовать опцию –narrow для распечатки диаграмма одну за другой, но проще увидеть их взаимосвязь при их распечатке бок о бок.

Конечно же, в реальном коде callback не будет возвращать одно и тоже значение каждый раз, и данная функция может иногда завершиться успешно, иногда - с ошибкой. Но симулятор может дать представление того, что произойдет для заданной комбинации результатов и ошибок, в заданной композиции callback’ов и errback’ов.

9.5. Резюме

Немного больше подумав о callback’ах, мы осознали, что позволяя callback исключениям всплывать вверх по стеку, не приведет ни к чему хорошему, поскольку программирование с использованием callback’ов инвертирует обычную взаимосвязь между низкоуровневым и высокоуровневым кодом. Класс Deferred решает эту проблему, отлавливая исключения и отправляя из вниз по цепочке, вместо вверх, в reactor.

Мы также изучили, что обычные результаты (возвращаемые значение) также перемещаются вниз по цепочке. Комбинирую оба факта вместе, дают в результате шаблон активизации крест-накрест, так как deferred переключается между callback и errback цепочками, взависимости от результата на каждом этапе.

Вооружившись эти знанием, в следующей главе мы обновим наш поэтический клиент с некоторой поэтической трансформирующей логикой.

9.6. Упражнения

1. Изучите реализацию каждого из четырех методов в классе Deferred,

   которые добавляют callback'и и errback'и. Проверьте, что все методы добавляют пару callback/errback.

2. Используйте deferred симулятор для того, чтобы изучить отличие между этим кодом:

   ```
   deferred.addCallbacks(my_callback, my_errback)
   ```

   и этим кодом:

   ```
   deferred.addCallback(my_callback)
   deferred.addErrback(my_errback)
   ```

   Вспомните, что два последних метода добавляют неявные сквозные функции в качестве одного из членов пары.

10. Преобразованная поэзия

10.1. Клиент 5.0

Теперь мы будем добавлять некоторую трансформирующую логику в наш поэтический клиент к строкам, предложенным в части 9, с использованием упрощенного преобразования - Cummingsifier. Cummingsifier - алгоритм, который на вход берет поэму и возвращает поэму в стиле e.e.cummings. Далее алгоритм, который выполняет преобразование:

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

К сожалению, этот алгоритм очень простой и, реально нем никогда не произойдет ошибки, поэтому в клиенте 5.0, расположенном в twisted-client-5/get-poetry.py, мы используем модифицированную версию cummingsify, которая произвольно выполняет следующее:

1. Возвращает трансформированную версию
2. Генерирует GibberishError
3. Генерирует ValueError

Таким образом мы эмулируем более сложный алгоритм, который иногда может глючить.

Изменения в клиенте 5.0 также сделаны в функции poetry_main:

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

    from twisted.internet import reactor

    poems = []
    errors = []

    def try_to_cummingsify(poem):
        try:
            return cummingsify(poem)
        except GibberishError:
            raise
        except:
            print 'Cummingsify failed!'
            return poem

    def got_poem(poem):
        print poem
        poems.append(poem)

    def poem_failed(err):
        print >>sys.stderr, 'The poem download failed.'
        errors.append(err)
```

```
def poem_done(_):
    if len(poems) + len(errors) == len(addresses):
        reactor.stop()

for address in addresses:
    host, port = address
    d = get_poetry(host, port)
    d.addCallback(try_to_cummingsify)
    d.addCallbacks(got_poem, poem_failed)
    d.addBoth(poem_done)

reactor.run()
```

Таким образом, когда программа скачивает поэму с сервера, она будет выполнять одно из:

1. Печатать трансформированную версию поэмы
2. Печатать “Cummingsify failed!” и оригинальную версию поэмы
3. Печатать “The poem download failed.”

Хотя мы сохранили возможность скачивать с нескольких серверов, когда вы будете проверять клиент 5.0, проще использовать один сервер и запускать программу несколько раз, до тех пор пока вы не увидите различные выводы. Также попробуйте запустить клиент с портом, на котором не запущен сервер.

Давайте нарисуем цепочку callback/errback, которые создаются для каждого Deferred’a, создаваемого в функции get_poetry:

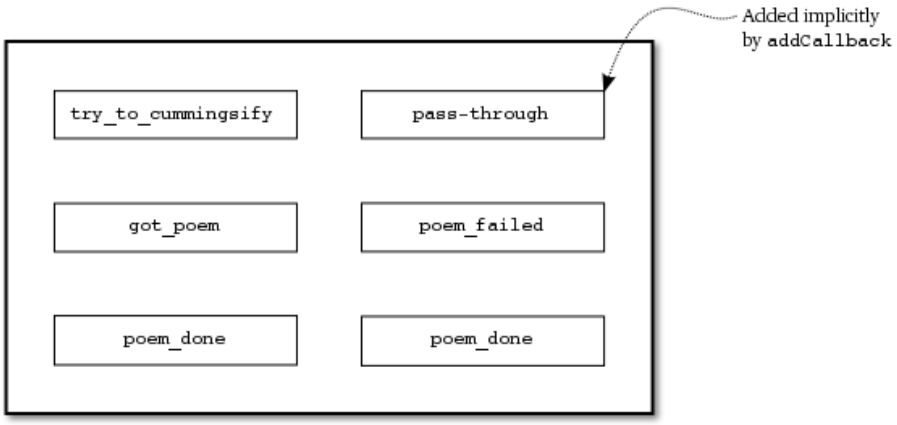

Рис. 19: Цепочка deferred’a в клиенте 5.0

Обратите внимание, что pass-through добавляется неявно в errback функцией addCallback. Функция pass-through ничего не делает и передает свой аргумент типа Failure следующему errback’у (poem_failed). Таким образом, poem_failed может управлять ошибками из двух функций: get_poetry (в этом случае deferred poem_failed активизируется предыдущим deferred’ом из цепочки errback) и функцией cummingsify.

Нужно заметить, что великолепная тень на рисунке 19 была сделана в Inkscape[1].

Давайте проанализируем различные способы, которые могут активизировать наши deferred'ы. Случай, когда мы получаем поэму и функция cummingsify работает корректно, изображен на рисунке 20:

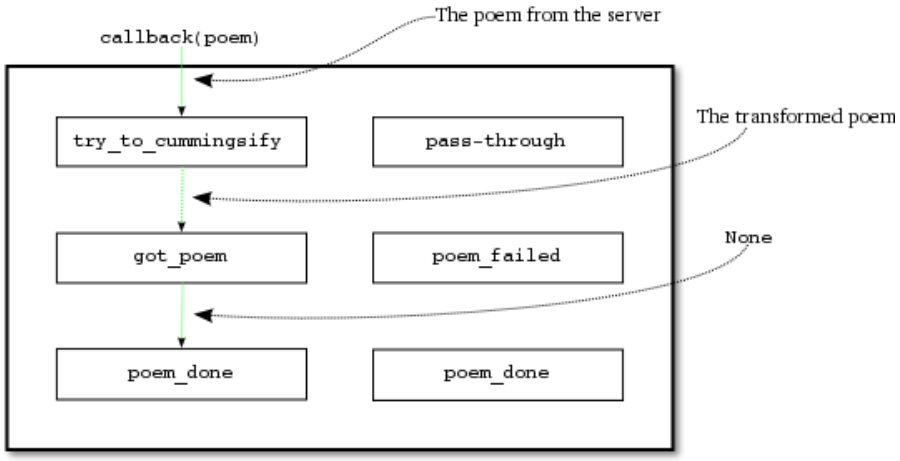

Рис. 20: Цепочка deferred'а в клиенте 5.0

В этом случае, ни в одном callback'е не произошло ошибок, так что управление полностью проходит вниз по линии callback. Заметим, что функция poem_done получает None в качестве аргумета, поскольку got_poem в действительности не возвращает значение. Если мы хотим, чтобы каждая функция из линии callback имела доступ к поэме, то нам нужно поменять функцию got_poem так, чтобы она возвращала поэму.

Рисунок 21 иллюстрирует случай, когда мы получием поэму, но в функции cummingsify генерирует исключение GibberishError:

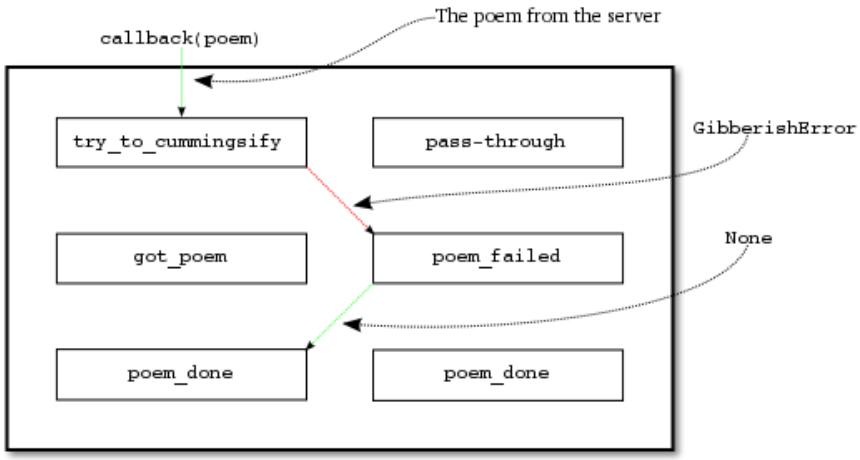

Рис. 21: Случай, когда мы скачали поэму и получили GibberishError

[1]http://inkscape.org/

После того как try_to_cummingsify callback генерирует во второй раз ислючение GibberishErro управление переключается на линию errback и вызывается poem_failed с исключением в качестве своего параметра (естественно, обернутого в Failure).

И, так как poem_failed не вызывает исключение, или не возвращает Failure, то после того, как она завершит работу, контроль переключается обратно на линии callback. Если мы предполагаем, что poem_failed полностью управляет ошибками, то возвратить None - это разумное поведение. С другой стороны, если мы хотим, чтобы poem_failed производила некоторое дейтсвие, но все еще передавала ошибку, мы могли бы поменять poem_failed так, чтобы она возвращала свой аргумент err и, обработка продолжилась бы ниже по линии errback.

Заметим, что ни в got_poem, ни в poem_failed никогда не происходит ошибок, так что функция poem_done для errback никогда не будет вызвана. Но, безопасней добавить добавить такую функцию в errback, чтобы явится примером “оборонительного” программирования, так как или got_poem, или poem_failed могут иметь ошибки, о которых мы не знаем. Так как метод addBoth гарантирует, что определенная функция запустится независимо от того как deferred был активизирован (ошибкой или результатом), использование addBoth является аналогичным добавлению finally в оператор try/except.

Теперь исследуем случай, который изображен на рисунке 22, когда мы скачали поэму, и функция cummingsify сгнерировала исключение ValueError.

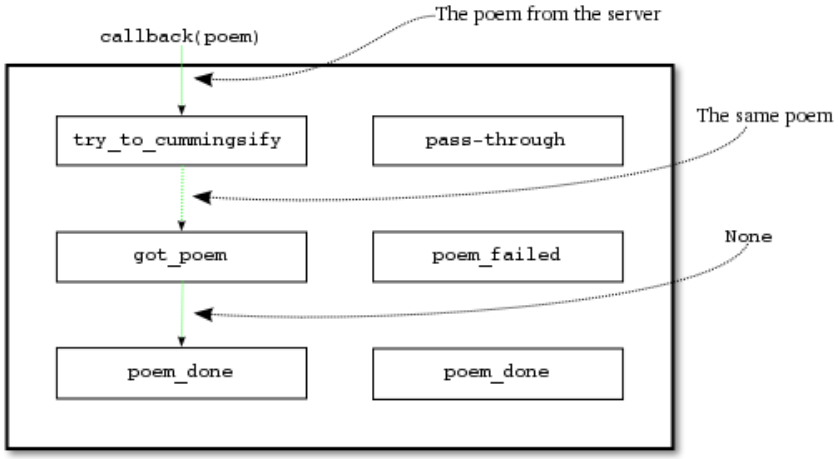

Рис. 22: Случай, когда мы скачали поэму и в cummingsify произошла ошибка

Эта ситуация аналогична ситуации на рисунку 20, за исключением, что got_poem получает оригинальную версию поэмы вместо трансформированной версии. Переключение происходит полностью внутри callback’a try_to_cummingsify, который отлавливает исключение ValueError обычным оператором try/except и возвращает оригинальную поэму. Объект deferred не обнаруживает ошибки.

И наконец, рисунок 23 - мы рассматриваем случай, когда мы пытались скачать поэму из несуществующего сервера:

Как и прежде, poem_failed возвращает None, поэтому далее контроль переключается на линию callback.

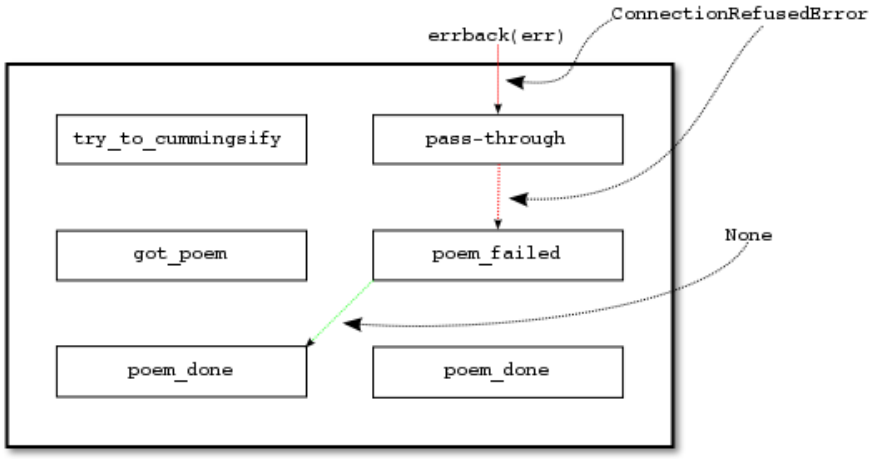

Рис. 23: Случай, когда мы не можем соединиться с сервером

10.2. Клиент 5.1

В клиенте 5.0 мы отлавливаем исключение из функции cummingsify в нашем callback'е try_to_cummingsify, используя обычный оператор try/except, не позволяя deferred'у ловить его первым. Такой подход не является обязательно ошибычным, но является поучительным рассмотреть как мы можем сделать это по-другому.

Давайте предположим, что мы хотим позволить deferred'у ловить оба исключения GibberishError и ValueError и отправлять их по цепи errback. Для того, чтобы сохранить поведение нашей последовательности errback, нужно проверить: если мы видим, что ошибка ValueError, то нужно возвратить оригинальную поэму, так чтобы управление вернулось обратно на цепь callback и, чтобы оригинальная поэма напечаталась.

Но здесь есть проблема: errback не получил бы оригинальную поэму, вместо этого получил бы завернутое в Failure исключение ValueError, сгенерированное функцией cummingsify. Для того, чтобы позволить errback управлять ошибкой, нам нужно сделать так, чтобы errback получал оригинальную поэму.

Один из способов сделать такое - это поменять функцию cummingsify так, чтобы оригинальная поэма была добавлена в исключение. Это то, что мы сделали в клиенте 5.1, расположенном в twisted-client-5/get-poetry-1.py. Мы поменяли исключение ValueError на специальное исключение CannotCummingsify, которое принимает оригинальную поэму в качестве своего первого аргумента.

Если бы cummingsify была бы реальной функцией во внешнем модуле, то, вероятно, лучшим решением было бы обернуть ее другой функцией, которая отлавливала бы любое исключение кроме GibberishError и генерировала бы исключение CannotCummingsify. С нашей новой структурой, функция poetry_main выглядит так:

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

    from twisted.internet import reactor

    poems = []
    errors = []
```

```
def cummingsify_failed(err):
    if err.check(CannotCummingsify):
        print 'Cummingsify failed!'
        return err.value.args[0]
    return err

def got_poem(poem):
    print poem
    poems.append(poem)

def poem_failed(err):
    print >>sys.stderr, 'The poem download failed.'
    errors.append(err)

def poem_done(_):
    if len(poems) + len(errors) == len(addresses):
        reactor.stop()

for address in addresses:
    host, port = address
    d = get_poetry(host, port)
    d.addCallback(cummingsify)
    d.addErrback(cummingsify_failed)
    d.addCallbacks(got_poem, poem_failed)
    d.addBoth(poem_done)
```

И каждый deferred имеет структуру как на рисунке 24:

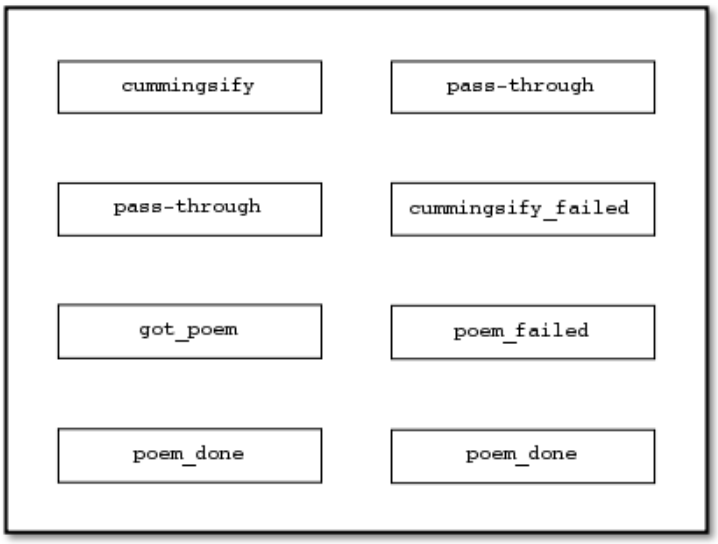

Рис. 24: Цепочка deferred’а в клиенте 5.1

Исследуем cummingsify_failed errback:

```
def cummingsify_failed(err):
    if err.check(CannotCummingsify):
        print 'Cummingsify failed!'
        return err.value.args[0]
    return err
```

Мы используем метод check объектов типа Failure для проверки является ли исключение, встроенное в Failure экземпляром типа CannotCummingsify. Если это так, то мы возвращаем первый аргумент исключения (изначальную поэму) и, таким образом управляем ошибкой. Поскольку возвращаемое значение не является Failure, то контроль возвращается цепи callback. Иначе, мы возвращаем Failure и отправляем ошибку далее по цепи errback. Как вы можете это видеть, исключение доступно как значение атрибута объекта Failure.

На рисунке 25 показано что случится, если мы получим исключение CannotCummingsify:

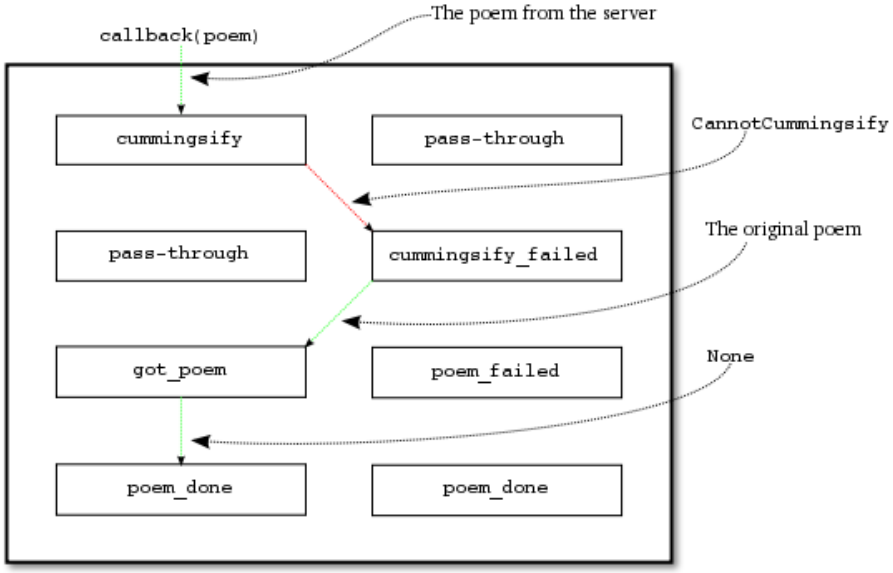

Рис. 25: Когда мы получаем ошибку CannotCummingsify

Таким образом, когда мы используем deferred'ы, иногда мы можем выбрать хотим ли мы использовать операторы try/except для управления исключениями или позволить deferred'м отправлять ошибки в errback.

10.3. Резюме

В главе 10 мы обновили наш поэтический клиент для того, чтобы использовать возможность Deferred маршрутизовать ошибки и результаты по цепи callback'в и errback'в. Хотя пример был достаточно искусственным, он проиллюстрировал как контролирующий поток в deferred'х переключается между callback и errback цепями взависимости от результата на каждой стадии.

Так что теперь мы знаем все, что можно знать про deferred'ы, верно? Нет еще! Мы будем исследовать остальные особенности deferred'в в следующей части. Но сначала, мы поедем немного в объезд и, в главе 11, реализуем Twisted версию нашего поэтического сервера.

10.4. Упражнения

1. На рисунке 25 показан один из четырех возможных способах активизации deferred'а в клиенте 5.1. Нарисуйте три других.

2. Используйте deferred simulator для эмулирования всех возможных активизаций для клиетов 5.0 и 5.1. Для начала, эта программа может представлять случай, где функция try_to_cummingsify успешно завершается в клиенте 5.0:

```
r poem p
r None r None
r None r None
```

11. Ваша поэзия обслуживается

11.1. Twisted поэтический сервер

Теперь мы узучили достаточно много о написании клиентов с использованием Twisted, давайте развернемся и реализуем наш поэтический сервер тоже с использованием Twisted. И благодаря общности абстракций Twisted, оказывается, что мы уже изучили практически все, что нам надо знать. Давайте посмотрим на поэтический сервер, написанный с использованием Twisted, расположенный в twisted-server-1/fastpoetry.py. Он называется fastpoetry, поскольку этот сервер отправляет поэзию так быстро, насколько это возможно, совсем без задержек. Заметьте, что кода меньше, чем для клиента!

Давайте рассмотрим куски кода сервера один за другим. Сначала, PoetryProtocol:

```
class PoetryProtocol(Protocol):

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

Подобно клиенту, сервер использует экземпляр класса Protocol для управления соединениями (в этом случае, это соединения, которые клиенты делают к серверу). Поскольку наш протокол однонаправленный, то экземпляру серверного Protocol нужно только заботиться о посылке данных. Если вы помните, нашему протоколу требуется сервер, чтобы начать отправление поэму сразу же после соединения, поэтому мы реализовали метод connectionMade, который является callback'м и который вызывается после того как экземпляр Protocol соединяется с Transport.

Наш метод говорит Transport сделать две вещи: отправить полный текст поэмы (self.transport.write) и закрыть соединение
(self.transport.loseConnection). Конечно, обе эти операции асинхронные. Таким образом, вызов write() реально означает “со временем отправить все эти данные клиенту” и, вызов loseConnection() означает “закрыть это соединение сразу же, после того как все данные, которые я запрашивал для записи, были записаны”.

Как вы видите, Protocol получает текст поэмы из Factory, поэтому давайте это будет следующее, на что мы посмотрим:

```
class PoetryFactory(ServerFactory):

    protocol = PoetryProtocol

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

Теперь это ужасно просто. Реальной работой Factory, помимо создания экземпляра PoetryProtocol по требованию, является хранение поэмы, которую каждый PoetryProtocol отправляет клиенту.

Заметим, что мы создали подкласс ServerFactory вместо ClientFactory. Поскольку наш сервер только пассивно слушает соединения вместо активного их создания, нам не нужно создавать дополнительных методов, которые предоставляет ClientFactory. Как мы можем в этом убедиться? Мы используем метод реактора listenTCP, а документация по этому методу сообщает нам, что аргумент factory должен быть экземпляром ServerFactory.

Далее функция main, в которой мы вызваем listenTCP:

```
def main():
    options, poetry_file = parse_args()

    poem = open(poetry_file).read()

    factory = PoetryFactory(poem)

    from twisted.internet import reactor

    port = reactor.listenTCP(options.port or 0, factory,
                             interface=options.iface)

    print 'Serving %s on %s.' % (poetry_file, port.getHost())

    reactor.run()
```

Тут происходит в основном три вещи:

1. Чтение текста поэмы, которую мы собираемся обслуживать
2. Создание PoetryFactory с этой поэмой
3. Использование listenTCP для того, чтобы сообщить Twisted, что надо слушать соединения по указанному порту, и использовать нашу Factory для создания экземпляров Protocol для каждого нового соединения.

После этого, единственное, что осталось сделать - это сказать реактору запустить цикл. вы можете использовать любой из наших поэтических клиентов (или просто netcat), для того, чтобы проверить сервер.

11.2. Обсуждение

Вспомним рисунок 8 и рисунок 9 из главы 5. Они иллюстрируют как создается новый экземпляр Protocol и инициализируется после того как Twisted создает новое соединение на нашей стороне. Оказывается, используется тот же самый механизм, когда Twisted принимает новое соединение по порту, на котором мы слушаем. Поэтому оба, connectTCP и listenTCP, требуют аргумент factory.

Одна вещь, которую мы не показали на рисунке 9, - это то, что callback connectionMade также вызывается как часть инициализации
Protocol. Это происходит не смотря ни на что, но нам не нужно использовать это в клиентском коде. И методы Protocol, которые мы использовали в клиенте, не используются в реализации сервера. Поэтому, если бы мы хотели, то мы могли бы сделать shared библиотеку с один единственным PoetryProtocol, который работает для обоих клиента и сервера. Таким способом реально обычно используется в самом Twisted. Например, NetstringReceiver Protocol может и читать, и писать netstring из и в Transport.

Мы пропустили написание низкоуровневой версии нашего сервера, но давайте думать о том, что происходит под капотом. Сначала, вызов listenTCP говорит Twisted'у создать слушающий сокет и добавляет его в event loop. “Событие” (event) на слушающем сокете не означает, что есть данные для чтения; вместо этого это означает, что есть клиент, который ожидает соединения к нам.

Twisted будет автоматически принимать входящие запросы на соединение, таким образом создавая новый клиентский сокет, который связывает сервер напрямую с определенным клиентом. Такой клиентский сокет также добавляется в цикл обработки событий

(event loop), и Twisted создает новый Transport и (через PoetryFactory) новый экземпляр PoetryProtocol для обработки определенного клиента. Таким образом, экземпляр Protocol всегда присоединен к клиентским сокетам, и никогда к слушающему сокету.

Мы можем визуализировать все это на рисунке 26.

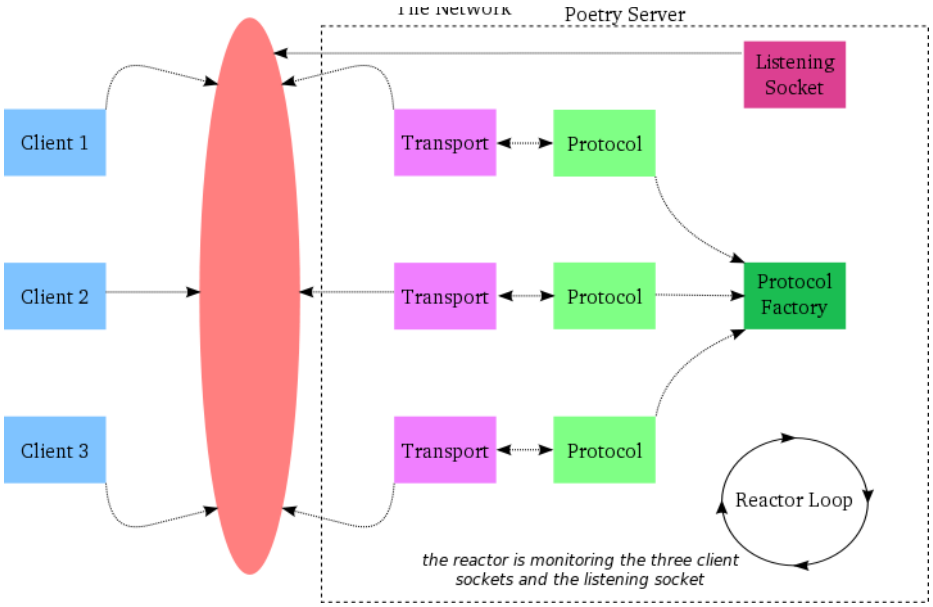

Рис. 26: Поэтический сервер в действии

На рисунке три клиента присоеденены к серверу. Каждый Transport представляет единственный клиентский сокет, и слушающий сокет делает в общей сложности 4 файловых дескриптора для мониторящего цикла операций ввода-вывода на основе select. Когда клиент отсоединяется, связанный с ним Transport и PoetryProtocol будет разыменован и утилизирован сборщиком мусора (предполагая, что мы не запрятали ссылку где-нибудь на один из них, нам не нужно предупреждать утечки памяти). PoetryFactory, между тем, не будет завершаться до тех пор пока мы будем слушать новые соединения, то есть, для нашего поэтического сервера, он будет висеть всегда.

Клиентские сокеты и связанные с ними Python объекты не будут жить долго, если поэма, которую мы обслуживаем, относительно короткая. Но, с огромной поэмой и с относительно занятым сервером, мы могли бы оборваться из-за сотни или тысячи одновременных клиентов. И тут нет проблем, поскольку Twisted не имеет встроенных ограничений на количество соединений, которыми он может управлять. Конечно же, если вы увеличите нагрузку на любой сервер, то с какого-то момента, вы обнаружите, что сервер не может поддерживать такую нагрузку или, будет достигнуто внутреннее ограничение операционной системы. Для высоко-нагруженных серверов нужно тщательно измерять и тестировать пределльную нагрузку.

Twisted также не налагает ограничений на количество портов, которые вы можете слушать. Фактически, единственный Twisted процесс может слушать десятки портов и предоставлять различные сервисы на каждый (используя различные Factory-классы для каждого listenTCP вызова). И, при тщательном дизайне, решение, о том предоставлять ли несколько сервисов одним Twisted процессом или несколькими, потенциально можно отложить до этапа выкладки.

Есть несколько вещей, которые пропущены в нашем сервере. Прежде всего, он не создает никаких логов, которые могли бы помочь отладить проблемы или проанализировать наш сетевой трафик. Более того, сервер не запущен как демон, что делает его уязвимом в смысле завершения, при случайном нажатии Ctrl-C (или просто выхода из

системы). Мы исправим эти две проблемы в следующей главе, но сначала, в главе 12, мы напишем еще один сервер, выполняющий поэтическое преобразование.

11.3. Упражнения

1. Напишите асинхронный поэтический сервер без использования Twisted, подобно тому, который мы сделали для клиента в главе 2. Заметьте, что слушающие сокеты нужно мониторить для чтения и, “читабельность” слушающего сокета означает, что мы можем принимать новый клиентский сокет.

2. Напишите высокоуровневую версию поэтического сервера “медленный сервер” с использованием Twisted, используя callLater или
   LoopingCall, для создания нескольких вызовов к transport.write(). Добавьте опции *--num-bytes* и *--delay* в строку запуска, поддерживаемые блокирующим сервером. Не забудьте про случай, когда сервер отсоединяется до того, как получена вся поэма.

3. Расширьте высокоуровневый Twisted сервер так, чтобы он мог обслуживать несколько поэм (на различных портах).

4. Какие есть причины делать обслуживание нескольких сервисов в одном и том же Twisted процессе? Какие есть причины так не делать?

12. Сервер, преобразующий поэзию

12.1. Еще один сервер

Не смотря на то, что мы уже написали один Twisted сервер, давайте теперь напишем другой, и затем мы опять вернемся к изучению Deferred'в.

В главе 9 и 10 мы ознакомились с идеей движка, трансформирующего поэзию. Один из такик движков (cummingsifier) мы, в конечном итоге, реализовали, это было настолько просто, что нам пришлось добавить случайные исключения для эмуляции ошибки. Но, если бы трансформирующий движок был размещен на другом сервере, обеспечивая сетевой "сервис, по трансформации поэзии", то ошибка "трансформирующий сервер не доступен" была бы более реальной.

Таким образом, в главе 12 мы собираемся реализовать поэтический трансформирующий сервер и, затем, в следующей главе, мы обновим наш поэтический клиент, для того, чтобы он мог использовать внешний трансформирующий сервис, и в процессе изучим несколько новых вещей по поводу Deferred'в.

12.2. Проектирование протокола

До сих пор взаимодействия между клиентом и сервером были строго одностороннее. Сервер отправляет поэму клиенту, в то время как клиент никогда вовсе ничего не отправляет серверу. Но, трансформирующий сервис является двунаправленным - клиент отправляет поэму серверу, затем сервер отправляет трансформированную поэму обратно.

Позволим серверу поддерживать несколько видов трансформаций и разрешим клиенту выбирать какой вид использовать. Клиент будет отправлять два куска информации: название трансформации и полный текст поэмы. Сервер будет возвращать один кусок информации, а именно: текст трансформированной поэмы. В итоге, мы получаем очень простую разновидность удаленного вызова процедур (Remote Procedure Call).

Twisted включает поддержку нескольких протокролов, которые мы могли бы использовать для решения этой проблемы, включая XML-RPC, Perspective Broker и AMP.

Но, включение любой из этих полнофункциональных протоколов увлекло бы нас вдаль, поэтому мы вместо этого будем использовать наш собственный скромный протокол. Давайте, клиент будет отправлять строку в форме (без угловых скобок):

```
<                              >.<              >
```

Это только название преобразования с последующей точкой и полным текстом поэмы. Все это мы будем кодировать в формате netstring. Сервер будет отправлять текст преобразованной поэмы, также в формате netstring. Поскольку netstring'и хранят длину строки, клиент сможет обнаружить случай, когда на стороне сервера возникает ошибка, в момент отправки результата (например, в середине операции сервер мог сломаться). Если вы вспомните, то наш изначальный поэтический протокол имел проблемы по обнаружению прерывания доставки поэзии.

Возможно, что наш протокол не самый лучший, но его достаточно для цели изучения.

12.3. Код

Давайте посмотрим на код нашего трансформирующего сервера, расположенного в twisted-server-1/tranformedpoetry.py. Сначала мы определяем класс TransformService:

```
class TransformService(object):

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

Трансформирующий сервис на данный момент поддерживает только одну трансформацию - cummingsify, через метод с тем же названием. Мы могли бы добавить дополнительные алгоритмы используя дополнительные методы. Тут важно отметить следующее: трансформирующий сервис полностью независим от определенных деталей протокола, про который мы договорились ранее. Отделение логики протокола от логики сервиса - это общий шаблон программирования в Twisted. Использование этого шаблона позволяет без дублирования кода предоставлять один и тот же сервис через различные протоколы.

Теперь давайте посмотрим на protocol factory (мы будем смотреть на протокол после):

```
class TransformFactory(ServerFactory):

    protocol = TransformProtocol

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

    def transform(self, xform_name, poem):
        thunk = getattr(self, 'xform_%s' % (xform_name,), None)

        if thunk is None: # no such transform
            return None

        try:
            return thunk(poem)
        except:
            return None # transform failed

    def xform_cummingsify(self, poem):
        return self.service.cummingsify(poem)
```

Данная factory предоставляет метод transform, который может использоваться экземпляром protocol для запроса трансформации поэзии на стороне присоединенного клиента. Метод возвращает None, если не существует запрашиваемой трансформации или, если преобразование не выполнилось. И, подобно TransformService, protocol factory независит от типа сетевого протокола, детали которого реализуются в самом классе protocol.

Одно замечание: мы защищаем доступ до сервиса через методы, имеющие префикс xform_. Это шаблон, который вы найдете в исходниках Twisted, хотя префиксы варьируются. Это один из способов предотвратить клиентский код от выполнения произвольного метода над сервисным объектом, поскольку клиент может отправить любое название трансформации. Также это является местом для выполнения протокольно-специфичной адаптации к API, предоставленную сервисным объектом.

Теперь мы посмотрим на реализацию протокола:

```
class TransformProtocol(NetstringReceiver):

    def stringReceived(self, request):
```

```
        if '.' not in request: # bad request
            self.transport.loseConnection()
            return

        xform_name, poem = request.split('.', 1)

        self.xformRequestReceived(xform_name, poem)

    def xformRequestReceived(self, xform_name, poem):
        new_poem = self.factory.transform(xform_name, poem)

        if new_poem is not None:
            self.sendString(new_poem)

        self.transport.loseConnection()
```

В реализации протокола мы воспользуемся тем фактом, что Twisted поддерживает netstring'и в реализации протокола NetstringReceiver. Этот базовый класс кодирует и декодирует netstring'и, и все, что мы должны реализовать, - метод stringReceived. Метод stringReceived вызывается с содержимым netstring'а, отправленым клиентом, без дополнительных байт, добавляемых при кодировании строки в netstring (без заголовка). Базовый класс NetstringReceiver также заботится о буферизации входящих байт до тех пор пока у нас не будет достаточно для декодирования.

Если все прошло хорошо (и если мы не закрыли соединение), мы отправляем трансформированную поэму обратно клиенту, используя метод sendString, предоставляемый NetstringReceiver (и который в итоге вызывает transport.write()). Мы не будем больше надоедать функцией main, поскольку она подобна рассмотренным ранее.

Заметьте, что мы продолжаем Twisted шаблон, переводящий входный поток байт, к более и более высоким уровням абстракции путем определения метода xformRequestReceived, который вызывается с двумя независимыми аргументами: название трансформации и сама поэма.

12.4. Простой клиент

Мы реализуем Twisted клиент для преобразующего сервиса в следующей части, а на данный момент мы просто обойдемся простым скриптом, расположенным в twisted-server-1/transform-test. В скрипте используется программа netcat, которая отправляет поэму серверу и печатает его ответ (который будет кодирован как netstring). Допустим, что вы запустили трансформирующий сервер на порту 11000:

```
python twisted-server-1/tranformedpoetry.py --port 11000
```

Тестирующий скрипт можно запустить так:

```
./twisted-server-1/transform-test 11000
```

И вы увидите примерно следующее:

```
15:here is my poem,
```

Это трансформированная поэма в виде netstring (оригинальная в верхнем регистре).

12.5. Обсуждение

В этого главе мы ознакомились с несколькими новыми идеями:

1. Двунаправленной коммуникацией
2. Встраивание протокола, реализованного в Twisted
3. Использование сервисного объекта для разделения функциональной и протокольной логики

Основные механизмы двунаправленной коммуникации простые. Мы использовали те же самые приемы для чтения и записи данных в предыдущих клиентах и серверах; единственное отличие в том, что мы их использовали вместе. Конечно же, более сложный протокол потребует более сложного кода для обработки байтового потока и форматирования выходных сообщений. И есть огромная причина использовать существующую реализацию протокола подобно тому, как мы это сделали сегодня.

Как только вам становится комфортно писать базовые протоколы, хорошая идея - посмотреть на различные реализации протоколов в Twisted. Вы можете начать с просмотра twisted.protocols.basic модуля и продолжить оттуда. Написание простых протоколов - это прекрасный способ познакомить себя со стилем программирования в Twisted, но в «реальной» программе, вероятнее, что более общим является использование готовой реализации, предполагая, что она есть.

Последняя идея, с которой мы познакомились, - использование объекта Service для разделения функциональной и протокольной логики, является действительно выжным шаблоном проектирования с использованием Twisted. Хотя, сервисный объект, который мы сегодня сделали является тривиальным, вы можете представить более реалистичный сетевой сервис, который мог бы быть достаточно сложным. Создавая Service независимым от деталей протокола, мы можем быстро предоставить тот же самый сервис по новому протоколу без дублирования кода.

Рисунок 27 показывает трансформирующий сервер, предоставляющий трансформацию поэзии через два различных протокола (версия сервера, который мы представили выше предоставляет только один протокол):

Хотя нам нужно две различных protocol factory, как это изображено на рисунке 27, они могли бы отличаться только их классовым атрибутом protocol и были в остальном идентичными. Фабрики (factory) могли бы разделять один и тот же объект Service, и только сами Protocol требовали бы разной релизации.

12.6. Взгляд в будущее

В главе 13, мы обновим наш поэтический клиент так, чтобы он мог использовать трансформирующий сервер вместо реализации трансформаций в самом клиенте.

12.7. Упражнения

1. Прочитайте исходный код для класса NetstringReceiver. Что произойдет, есди клиент отправит неправильный netstring? Что случится, если клиент попытается отправить огромный netstring?
2. Изобретите другой трансформирующий алгоритм и добавьте его в трансформирующий сервис и protocol factory. Протестируйте его, поменяв netcat клиент

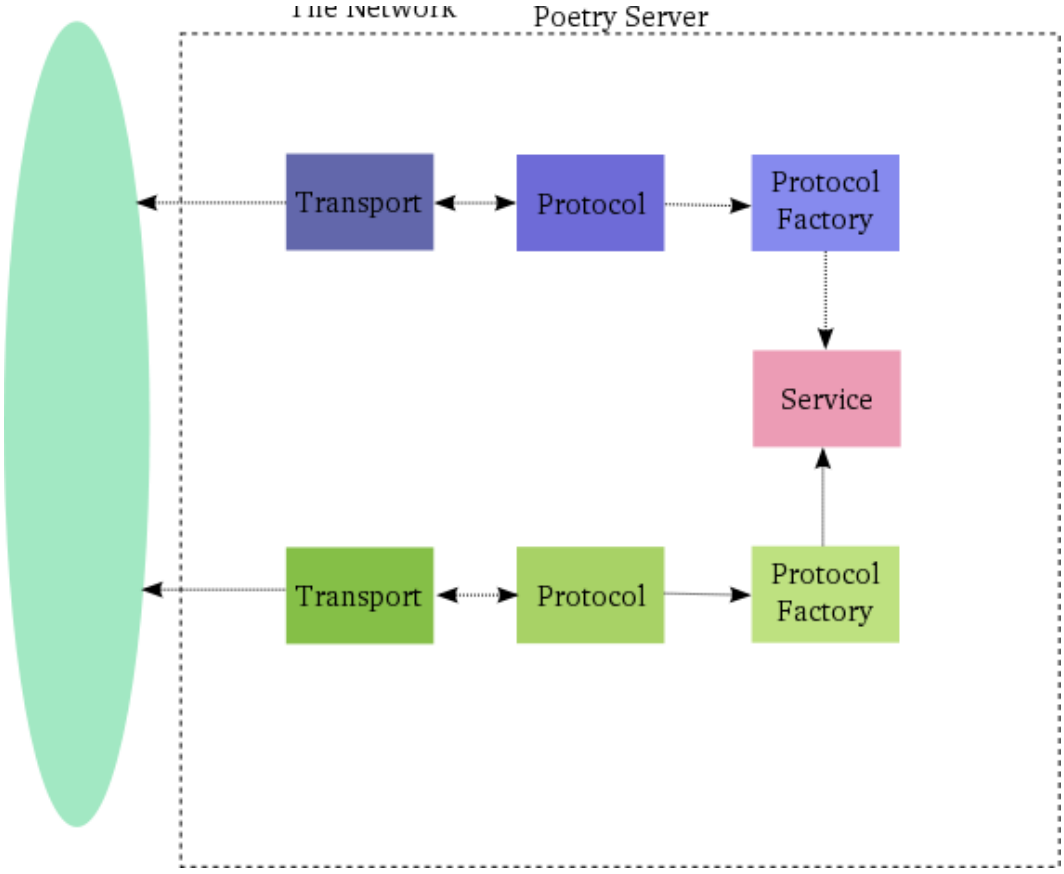

Рис. 27: Трансформирующий сервер с двумя протоколами

3. Изобретите другой протокол для запрашивания поэтических трансформаций и поменяйте сервер так, чтобы он мог управлять обоими версиями протоколов (на различных портах). Используйте тот же экземпляр TransformService для обоих.

4. Как нужно было бы поменять код, если методы для TransformService были бы асинхронные (например, они возращали бы Deferred'ы)?

5. Напишите синхронный клиент для трансформирующего сервера.

6. Обновите изначальный клиент и сервер так, чтобы они использовали netstring'и при посылке поэзии.

13. Deferred'ы из Deferred'ов

13.1. Введение

Вспомним поэтический клиент 5.1 из главы 10. Он использовал Deferred для управления цепочкой callback'ов, в которую было включено обращение к поэтическому трансформирующему движку. В клиенте 5.1, движок релизован как синхронный вызов фукнкции, реализованный в самом клиенте.

Теперь мы хотим сделать новый клиент, который использует сетевой сервис преобразования поэзии из главы 12. Здесь есть один ньюанс: поскольку трансформирующий сервер доступен по сети, то нам нужно будет использовать асинхронный ввод-вывод. И это означает, что наш API, запрашивающий трансформацию, должен быть асинхронным. Другими словами, callback try_to_cummingsify в нашем клиенте будет возвращать Deferred.

Что происходит, когда callback в цепочке deferred'а возвращает другой deferred? Давайте назовем первый deferred "внешним" deferred'м, а второй - "внутренним". Предположим, что N-ый callback во внешнем deferred'е возвратил внутренний deferred. Этот callback говорит: "Я асинхронный, моего результата еще здесь нет". Поскольку внешнему deferred'у нужно вызывать следующий callback или errback в цепочке с результатом, то внешнему deferred'у придется ждать активизации внутреннего deferred'а. Конечно же, внешний deferred не может блокировать, так что вместо этого он приостанавливает выполнение цепочки callback и возвращает управление реактору (или тому, кто его активизировал).

И каким образом внешний deferred узнает, когда возобновиться? Просто - путем добавления пары callback/errback во внутренний deferred. Таким образом, когда внутренний deferred активизируется, для внешнего deferred'а выполнение его цепочки будет возобновлено. Если внутренний deferred успешно завершается (например, он вызвал callback, добавленный внешним deferred'ом), то внешний deferred вызывает свой N+1'ый callback с результатом. И если внутренний deferred завершился с ошибкой (вызвал errback, добавленный внешний deferred'ом), то внешний deferred вызовет N+1'ый errback с failure.

Это нужно осознать, поэтому давайте проиллюстрируем эту идею на рисунке 28.

На этом рисунке внешний deferred имеет 4 слоя пар callback/errback. Когда внешний deferred активизируется, первый callback в цепочке возвращает deferred (внутренний). С этого места, внешний deferred остановит активизацию своей цепочки и возвратит управление реактору (после добавления пары callback/errback во внутренний deferred). Затем, через некоторое время, внутренний deferred активизирует внешний deferred, и он возобновит обработку своей callback цепочки. Заметим, что внешний deferred сам не активизирует внутренний deferred. Это было бы невозможно, поскольку внешний deferred не знает, когда доступен результат для внутреннего deferred'а, или какой результат для него мог бы быть. Вместо этого, внешний deferred просто ожидает (асинхронно) активизации внутреннего deferred'а.

Заметим, что линия, соединяющая callback и внутренний deferred на рисунке 28 черная, а не зеленая или красная. Это потому, что мы не знаем до того момента как внутренний deferred не активизируется был ли этот callback успешно выполнен, или он был выполнен с ошибками.

На рисунке 29 показана та же последовательность внутренних и внешних deferred'в, что и на рисунке 28, с точки зрения реактора.

Это вероятно самое сложное свойство класса Deffered, поэтому не беспокойтесь, если вам нужно некоторое время, что бы понять. Мы проиллюстрируем это еще одним способом, используя пример кода из twisted-deferred/defer-10.py. Этот пример создает два внешних deferred'а: один - с обычными callback'ми, другой - с callback'ми, один из кото-

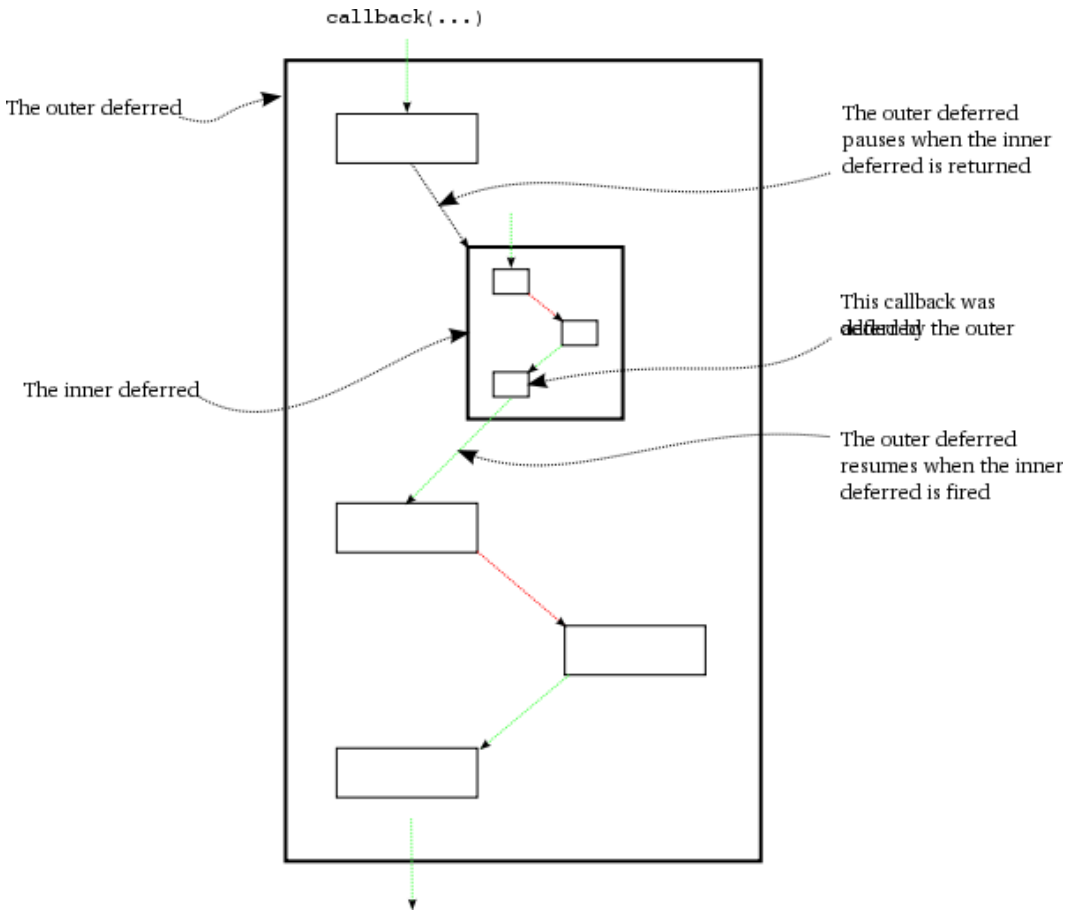

Рис. 28: Обработка внутреннего и внешнего deferred'ов

рых возвращает внутренний deferred. Изучая код, и то, что он выводит, вы можете увидеть, как второй, внешний, deferred останавливает выполнение своей цепочки в момент, когда внутренний deferred возвращается, и после этого возобновляет выполнение, когда внутренний deferred активизирован.

13.2. Клиент 6.0

Давайте использовать наше новое знание о вложенных deferred'х и реализуем заново наш поэтический клиент для использования сетевого трансформирующего сервиса из главы 12. Вы можете найти его код в twisted-client-6/get-poetry.py. Поэтические Protocol и Factory не изменялись и остались такими же как и в предыдущей версии. Но теперь появились новые Protocol и Factory для создания запросов на преобразование. Далее Protocol трансформирующего клиента:

```
class TransformClientProtocol(NetstringReceiver):

    def connectionMade(self):
        self.sendRequest(self.factory.xform_name, self.factory.poem)

    def sendRequest(self, xform_name, poem):
        self.sendString(xform_name + '.' + poem)

    def stringReceived(self, s):
        self.transport.loseConnection()
```

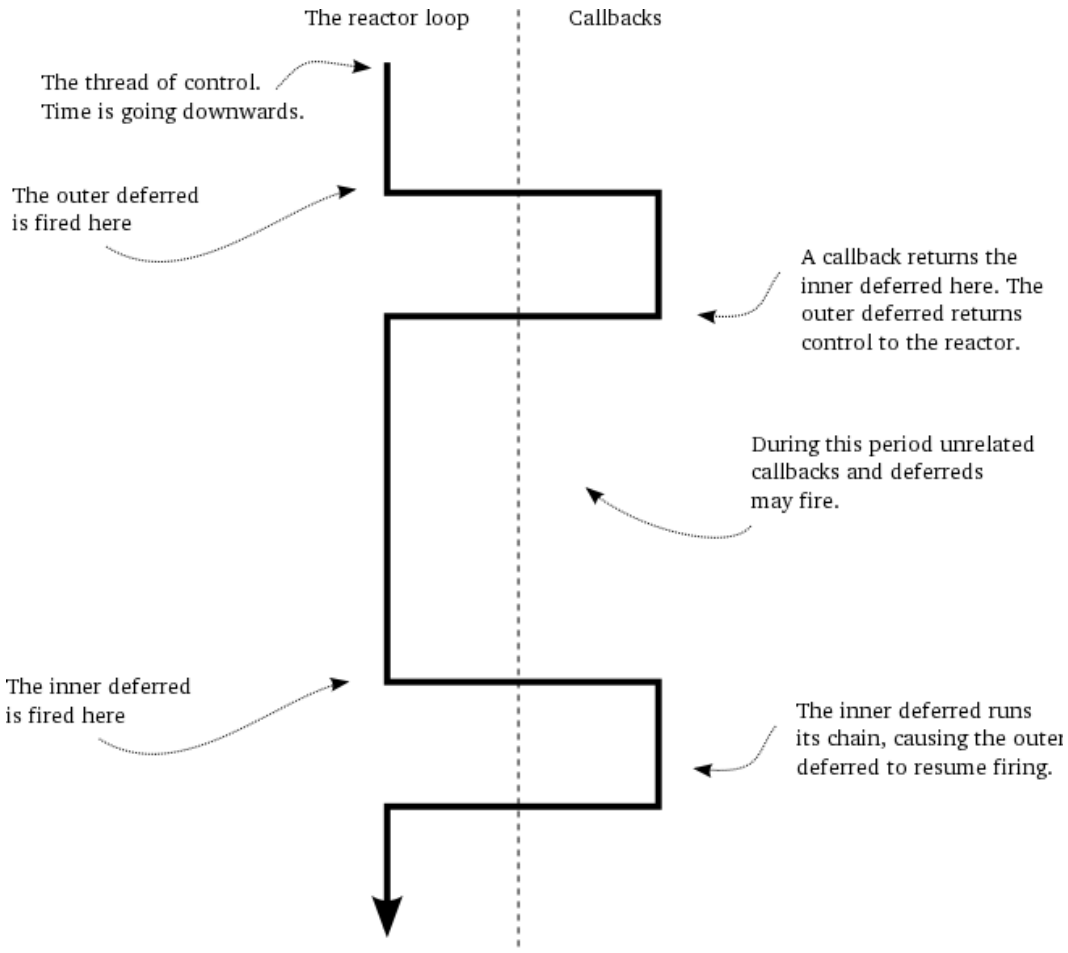

Рис. 29: Поток выполнения при обработке внутренних и внешних deferred'в

```
        self.poemReceived(s)

    def poemReceived(self, poem):
        self.factory.handlePoem(poem)
```

Используя NetstringReceiver как базовый класс делает реализацию достаточно простой. Как только соединение установлено, мы отправляем запрос на трансформацию серверу, получая название трансформации и поэмы из TransformClientFactory. И, когда мы получаем поэму обратно, мы подставляем его в factory для обработки. Далее следует код для TransformClientFactory:

```
class TransformClientFactory(ClientFactory):

    protocol = TransformClientProtocol

    def __init__(self, xform_name, poem):
        self.xform_name = xform_name
        self.poem = poem
        self.deferred = defer.Deferred()

    def handlePoem(self, poem):
        d, self.deferred = self.deferred, None
        d.callback(poem)
```

```
    def clientConnectionLost(self, _, reason):
        if self.deferred is not None:
            d, self.deferred = self.deferred, None
            d.errback(reason)

    clientConnectionFailed = clientConnectionLost
```

Эта factory спроектирована для клиентов и управляет одним запросом на трансформацию, сохраняя название трансформации и поэму для использования Protocol'ом. Factory создает единственный Deferred, который представляет результат трансформирующего заспроса. Заметим, что TransformClientFactory управляет двумя случаями ошибок: ошибка соединения и соединение, разорванное в процессе получения поэмы. Также нужно отметить, что метод clientConnectionLost вызывается даже, если мы получили поэму, и не возникли ошибки, но в этом случае self.deferred будет None, благодаря методу handlePoem.

Класс TransformClientFactory создает Deferred, который он также активизирует. Хорошее правило следовать такой схеме в Twisted программировании, поэтому подчеркнем это:

__ Объекту, который создает Deferred, следует нести ответственность за активизацию этого Deferred'a.

Правила "ты делаешь - ты и активизируешь" помогает гарантировать, что данный deferred будет активизирован только один раз, и улучшает читаемость Twisted программы.

В дополнение к TransformClientFactory, есть также класс Proxy, который прячет детали создания TCP соединения к определенному трансформирующему сервису:

```
class TransformProxy(object):
    """
    I proxy requests to a transformation service.
    """


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

    def xform(self, xform_name, poem):
        factory = TransformClientFactory(xform_name, poem)
        from twisted.internet import reactor
        reactor.connectTCP(self.host, self.port, factory)
        return factory.deferred
```

Этот класс представляет единственный интерфейс xform(), который может использоваться для запросов на преобразование. Так что в остальном коде можно просто запрашивать трансформацию и получать обратно deferred, не заботясь о названиях хостов и номерах портов.

Оставшаяся часть программы не изменилась, за исключением try_to_cummingsify callback'a:

```
    def try_to_cummingsify(poem):
        d = proxy.xform('cummingsify', poem)

        def fail(err):
            print >>sys.stderr, 'Cummingsify failed!'
```

```
            return poem

        return d.addErrback(fail)
```

Этот callback теперь возвращает deferred, но мы не должны менять ничего кроме создания экземпляра Proxy в оставшейся части функции. Поскольку try_to_cummingsify был частью цепочки deferred'а (возвращаемого get_poetry), то он уже используется асинхронно, и ничего не надо менять.

Вы заметите, что мы возвращаем результат вызова d.addErrback(fail). Это только немного синтаксического сахара. Методы addCallback и addErrback возвращают оригинальный deferred. Мы могли бы написать двустрочный эквивалент:

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

13.3. Тестирование клиента

Новый клиент имеет немного отличающийся от ранее написанных клиентов синтаксис. Если бы вы имели трансформирующий сервис, запущенный на порту 10001, и два поэтических сервера, запущенных на портах 10002 и 10003, вы бы запустили:

```
python twisted-client-6/get-poetry.py 10001 10002 10003
```

Для того, чтобы скачать две поэмы и обе их трансформировать, вы можете запустить трансформирующий сервер следующим образом:

```
python twisted-server-1/tranformedpoetry.py --port 10001
```

И поэтические сервера так:

```
python twisted-server-1/fastpoetry.py --port 10002 poetry/fascination.txt
python twisted-server-1/fastpoetry.py --port 10003 poetry/science.txt
```

Затем вы можете запустить поэтический клиент как это было написано выше. После этого, попробуйте сломать трасформирующий сервер и перезапустить клиент той же командой.

13.4. Резюме

В этой главе мы изучили то, как deferred'ы могут прозрачно управлять другими deferred'ми в callback-цепочке, таким образом, мы можем безопасно добавлять асинхронные callback'ки во "внешние" deferred'ы не заботясь о деталях. Это очень удобно, поскольку большинство наших функций, в конечном счете, собираются стать асинхронными.

Знаем ли мы все о deferred'ах? Еще нет! Существует еще одна важная особенность, про которую мы поговорим в главе 14.

13.5. Упражнения

1. Модифицируйте клиент так, чтобы мы могли запрашивать определенный вид трансформации по названию.

2. Модифицируйте клиент так, чтобы адрес преобразующего сервера являлся необязательным аргументом. Если он не задан, то пропускать стадию преобразования.

3. Сейчас PoetryClientFactory нарушает правило для deferred'ов "ты создавал - ты и активизируй". Улучшите get_poetry и PoetryClientFactory для того, чтобы это вылечить.

4. Хотя, мы это не демонстрировали, но случай, когда errback возвращает deferred является симметричным. Поменяйте пример twisted-deferred/defer-10.py, чтобы это проверить.

5. Найдите место в реализации Deferred, которое управляет случаем, когда callback/errback возвращает еще один Deferred.

14. Случай, когда Deferred не является Deferred'ом

14.1. Введение

В этой части мы изучим другую сторону класса Deffered. Для мотивации нашего обсуждения, давайте добавим еще один сервер в наши стабильные поэтические сервисы. Предположим, что есть большое количество внутренних клиентов, которые хотят получить поэзию из одного и того же внешнего сервера. Но этот внешний сервер медленный и уже перегружен ненасытными сетевыми поэтическими запросами. Мы не хотим перегружать этот несчастный сервер, отправляя туда всех наших клиентов.

Поэтому вместо этого мы сделаем кеширующий прокси сервер. Когда клиент подсоединяется к прокси, прокси либо вытащит поэму из внешнего сервера, либо вернет закешированную копию поэмы, полученную ранее. Затем мы можем указать клиентам на прокси, после чего нагрузка на сервер спадет. Эта система показана на рисунке 30:

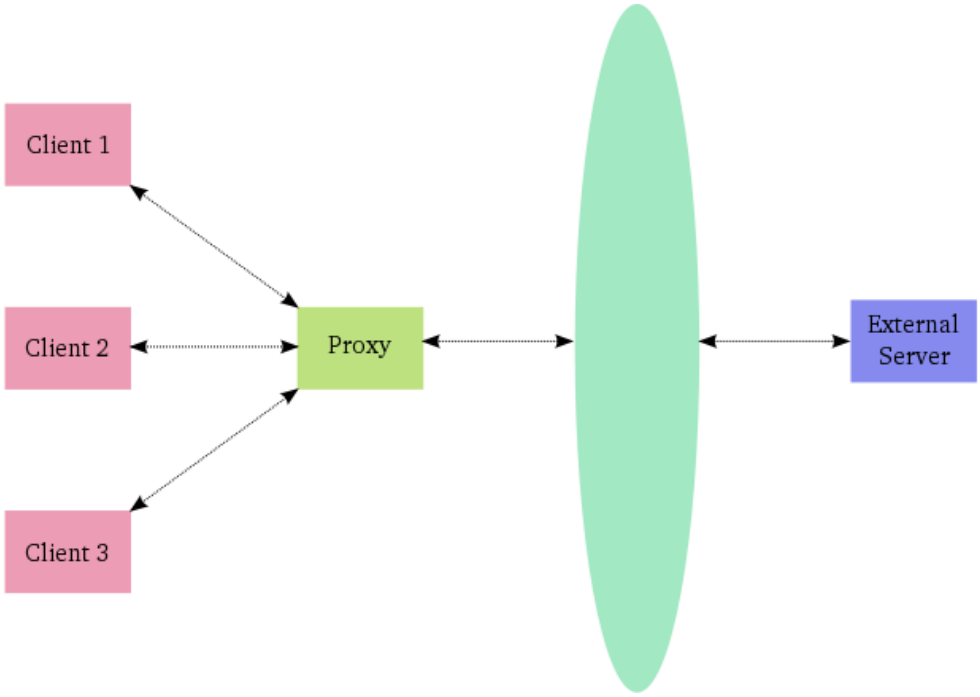

Рис. 30: Кеширующий прокси сервер

Рассмотрим что случится, когда клиент подсоединится к прокси для получения поэмы. Если кеш пустой, то прокси должен ждать (асинхронно) пока внешний сервер ответит до того, как отправить поэму. Пока все хорошо, и мы уже знаем как управлять такой ситуацией используя асинхронную функцию, которая возвращает deferred. С другой стороны, если в кеше уже есть поэма, то прокси может отправить ее обратно немедленно без ожидания. Поэтому внутренний механизм прокси для получения поэмы будет иногда асинхронным, а иногда - синхронным.

Что нам делать, если мы хотим иметь функцию, которая асинхронная только иногда? Twisted предоставляет ряд опций, зависящих от свойств класса Deferred, которые мы еще не использовали: вы можете активизировать deferred до того, как вы возвратите его тому, кто его вызывал.

Это работает, поскольку вы не можете активизировать deferred дважды, вы можете добавить callback'и и errback'и в deferred после того, как он уже был активизирован. И когда вы так делаете, deferred просто продолжает активизировать цепочку с того места, на

котором он остановился. Одна важная вещь, про которую надо упомянуть: один раз активизированный deferred может активизировать новый callback (или errback, взависимости от состояния deferred'а) сразу же, например, после его добавления.

Рассмотрим рисунок 31, показывающий deferred, который был активизирован:

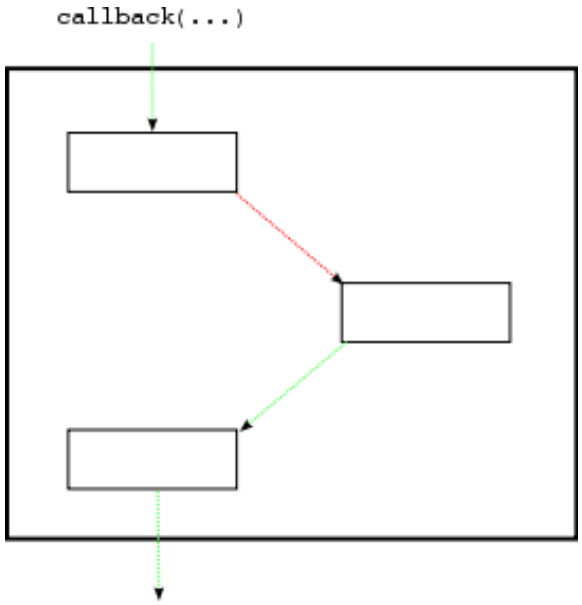

Рис. 31: Активизированный deferred

Если бы мы добавили другую пару callback/errback в этом месте, то deferred сразу же активизировал новый callback, как это показано на рисунке 32.

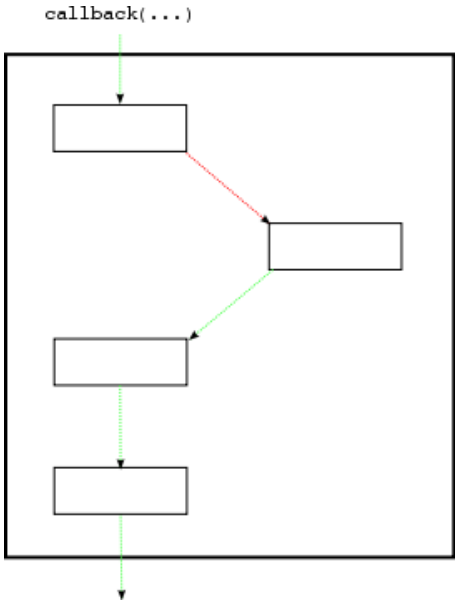

Рис. 32: Тот же deferred с новым callback'м

Активизация callback (не errback) происходит, потому что предыдущий callback завершился без ошибок. Если бы он завершился с ошибкой (вызвал Exception или возвратил Failure), то был бы вызван новый errback.

Мы можем проверить описанные новые свойства, используя пример кода из twisted-deferred/defer-11.py. Прочитайте и запустите этот скрипт, для того, чтобы посмотреть как

deferred ведут себя, когда вы их сначала активизируете и затем добавляете callback'и. Заметьте, что в первом примере каждый новый callback вызывается сразу же (вы можете сказать, что со скоростью порядка вывода с использованием print).

Второй пример в этом скрипте показывает, как вы можете приостановить, используя pause(), deferred так, чтобы он не активизировал callback'и прямо сейчас. Когда мы готовы для активизации callback'в, мы вызовем unpause(). Это в точности тот же механизм, который используют deferred'ы для приостановки самих себя, когда один из их callback'в возвращает другой deferred.

14.2. Прокси 1.0

Теперь давайте посмотрим на первую версию поэтического прокси в twisted-server-1/poetry-proxy.py. С того момента как прокси работает и как клиент, и как сервер, он имеет две пары Protocol/Factory классов, одна из них обслуживает поэзию, другая пара - получает поэзию из внешнего сервера. Мы не будем рассматривать код клиентской пары, она аналогична рассмотренной ранее.

Но до того как мы посмотрим на серверную пару, мы посмотрим на ProxyService, который использует протокол на стороне сервера для получения поэмы:

```
class ProxyService(object):

    poem = None # the cached poem

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

    def get_poem(self):
        if self.poem is not None:
            print 'Using cached poem.'
            return self.poem

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

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

Здесь ключевой метод - get_poem. Если в кеше уже есть поэма, то метод просто возвращает саму поэму. С другой стороны, если мы еще не получили поэму, мы инициируем соединение к внешнему серверу и возвращаем deferred, который будет активизирован, когда прийдет поэма. Поэтому get_poem - функция, являющаяся иногда асинхронной.

Как управлять функцией, подобной этой? Давайте посмотрим на пару protocol/factory на стороне сервера:

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

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

class PoetryProxyFactory(ServerFactory):

    protocol = PoetryProxyProtocol

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

Класс PoetryProxyFactory простой: он просто сохраняет ссылку на прокси сервис, так что экземпляры PoetryProxyProtocol могут вызывать метод get_poem. PoetryProxyFactory - место, где происходят все действия. Вместо вызова get_poem напрямую, PoetryProxyFactory использует обертку из модуля twisted.internet.defer maybeDeferred.

Функция maybeDeferred берет ссылку на другую функцию, плюс некоторые опциональные аргументы для этой функции (мы их не используем). После этого maybeDeferred действительно вызовет эту функцию и:

- Если функция возвращает deferred, maybeDeferred возвращает тот же deferred
- Если функция возвращает Failure, maybeDeferred возвращает новый deferred, который активизировался (через .errback()) с этим Failure
- Если функция возвращает обычное значение, maybeDeferred возвращает deferred, который активизировался с этим значением в качестве аргумента result
- Если функция сгенерировала исключение, maybeDeferred возвращает deferred, который активизировался (через .errback()) с этим исключением, обернутым в Failure.

Другими словами, возвращаемое значение функции maybeDeferred является гарантированно deferred'ом, даже если функция, которую вы подставляете, никогда не возвращает deferred. Это позволяет нам безопасно вызывать синхронные функции (даже те, выполнение которых завершается исключением) и обрабатывать их как асинхронные функции, возвращающие deffered.

Замечание 1: Хотя, тонкое отличие все же будет. deferred, возвращенный синронной функцией, уже активизирован, поэтому любые callback'и или errback'и, которые вы добавили, запустятся немедленно, а не в некотором будущем взаимодействии цикла реактора.

Замечание 2: По недосмотрительности, пожалуй, называть функцию, которая всегда возвращает deferred, “maybeDeferred” - было не лучшим выбором.

Поскольку PoetryProxyProtocol имеет реальный deferred для управления, он может добавить некоторые callback'и, которые отправляют поэму клиенту и затем закрывают соединение.

14.3. Запуск прокси

Для того, чтобы попробовать прокси, запустите поэтический сервер:

```
python twisted-server-1/fastpoetry.py --port 10001 poetry/fascination.txt
```

И теперь запустите прокси сервер:
то

```
python twisted-server-1/poetry-proxy.py --port 10000 10001
\e   nd{verbatim}



                    ,
      10000                    10001.
                   :

 \begin{verbatim}
python twisted-client-4/get-poetry.py 10000
```

Мы будем использовать предыдущую версию клиента, которая не связана с поэтическими преобразованиями. вы увидите поэму в клиентском окне и текст в окне с запущенным прокси, в котором будет запись о том, что прокси скачивает поэму с сервера. Теперь снова запустите клиент, и прокси должен подтвердить, что он использует закешированную версию поэмы, в то время как клиент показывает ту же поэму, что и раньше.

14.4. Прокси 2.0

Как мы упомянули ранее, существует альтернативный способ реализовать такую схему. Это показано в поэтическом прокси версии 2.0, расположенном в twisted-server-2/poetry-proxy.py. Поскольку мы можем активизировать deferred'ы до того, как мы их возвратим, мы можем создать прокси сервис, возвращающий уже активизированный deferred, в случае присутсвия поэмы в кеше. Далее новая версия метода get_poem в прокси сервисе:

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

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

Функция defer.succeed - это просто удобный способ создать уже активизированный deferred передавая ему результат. Просмотрите реализацию этой функции, и вы увидите, что это просто, по сути, создание нового deferred'а и его активизация с использованием .callback(). Если бы вы хотели возвратить активизированный deferred с ошибкой, вы могли бы использовать defer.fail.

В этой версии, поскольку get_poem всегда возвращает deferred, класс PoetryProxyProtocol не нуждается больше в использование метода maybeDeferred (хотя, из изученного ранее, ясно, что он бы работал, если бы мы его оставили):

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

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

Кроме этих двух изменений вторая версия прокси аналогична первой, и вы можете запустить ее тем же образом, что и первоначальную.

14.5. Резюме

В этой главе мы изучили как deferred'ы могут быть активизированы до того, как они были возвращены, и таким образом мы можем использовать их в синхронном (или частично синхронном) коде. И мы имеет два способа сделать это:

- Мы можем использовать maybeDeferred для управления функцией, которая иногда возвращает deferred, а иногда - обычное значение (или бросает исключение);
- Мы можем предактивизировать свои собственные deferred'ы, используя defer.succeed или defer.fail, поэтому наша "полусинхронная" функция всегда возвращает deferred.

Какую технику вы выбирете - дело ваше. Первый вариант подчеркивает, что наша функция не всегда асинхронная, в то время как второй вариант упрощает клиентский код. Пожалуй, не существует однозначного аргумента при выборе одного вместо другого.

Оба приема нужны для того, чтобы мы могли добавлять callback'и и errback'и к deferred'м, после того как они были активизированы. И это объясняет любопытный факт, который мы обнаружили в главе 9, и примере twisted-deferred/defer-unhandled.py. Мы изучили, что "неуправляемая ошибка" в deferred'е возникает в случае, когда последний callback или последний errback завершился с ошибкой, и это не сообщается до тех пор, пока deferred не будет утилизирован сборщиком мусора (например, когда в коде нет на него ссылок). Теперь мы знаем почему: поскольку мы могли всегда добавить еще одну callback/errback пару в deferred, которая управляет той ошибкой, и не активизируется, чтобы Twisted мог сказать, что ошибка не обработана, до тех пор, пока есть ссылки.

Теперь, когда вы потратили так много времени, исследуя класс Deferred, который расположен в пакете twisted.internet, вы могли бы заметить, что он ничего не должен делать с Internet. Это просто абстракция для управления callback'ми. Так что же он там делает? Это артифакт истории Twisted. В лучшем из всех возможных миров, модуль для deferred'в вероятнее был бы в twisted.python. Полагаю, что такова жизнь.

Так что это за deferred'ы? Знаем ли мы все о их свойствах? По большей части - да. Но Twisted позволяет использовать альтернативные способы использования deferred'в, которые мы еще не исследовали (мы доберемся!). Тем временем, разработчики Twisted упорно работают над добавлением новой вещи. В новом релизе, класс Deferred преобретен новую возможность. Мы ознакомимся с ней в следующей главе, но сначала, мы прервем изучение deferred'в и посмотрим на некоторые стороны Twisted, включая тестирование.

14.6. Упражнения

1. Модифицируйте пример twisted-deferred/defer-11.py для иллюстрации предактивизированный ошибкой deferred'в, используя .errback(). Прочтите докуменатцию и реализацию функции defer.fail.

2. Модифицируйте прокси так, чтобы закешированная более 2 часов поэма отбрасывалась, вызывая перезапрос из сервера следующего запроса.

3. Предполагается, что прокси избегает присоединения к серверу более одного раза, но при нескольких клиентских запросах, приходящих одновременно во время отсутвия поэмы в кеше, прокси будет делать несколько поэтических запросов. Это проще увидеть, если вы используете медленный сервер для проверки.

   Поменяйте прокси-сервис так, чтобы генерировался только один запрос. Сейчас сервис имеет только два состояния: либо поэма к кеше, либо - не в кеше. вам нужно добавить третье состояние, показывающее, что запрос был сделан, но еще не завершен. Когда метод get_poem вызывается в третьем состоянии, добавьте новый deferred в список “ожидающих”. Этот новый deferred будет результатом от метода get_poem. Когда поэма окончательно вернется, активизируйте все ожидающие deferred’ы с поэмой и перейдите в закешированной состояние. Если в процессе ожидания поэмы возникла ошибка, то активизируйте метод .errback() для всех ожидающих deferred’ов и перейдите в незакешированное состояние.

4. Добавьте преобразующий прокси в прокси-сервис. Этот сервис должен работать так же, как изначальный преоразующий сервис, но использовать внешний сервер для того, чтобы осуществить трансформации.

5. Предположим следующий гипотетический кусок кода:

   ```
   d = some_async_function() # d is a Deferred
   d.addCallback(my_callback)
   d.addCallback(my_other_callback)
   d.addErrback(my_errback)
   ```

   Предположим, что когда deferred d возвращен в строке 1, он не активизирован. Возможно ли для этого deferred’a, что он будет активизирован в момент, когда мы добавляем наши callback’и и errback’и в строках 2-4? Почему да или почему нет?

15. Протестированная поэзия

15.1. Введение

Мы написали много кода для того, чтобы изучить Twisted, но до сих пор мы пренебрегали написанием тестов. Вы можете удивиться: “Как можно тестировать асинхронный код используя синхронную систему, подобную пакету unittest из Python’a?” Короткий ответ: вы не можете. Как мы обнаружили, синхронный и асинхронный код не смешиваем, по меньшей мере не без труда.

К счастью, Twisted включает свою собственную тестирующую систему, названную trial, которая поддерживает тестирование асинхронного кода (также ее можно использовать для тестирования сихронного кода).

Мы будем предполагать, что вы уже знакомы с основными механизмами unittest и подобными тестирущими системами, в которых вы создаете тесты определением класса с определенным родительским классом (обычно с названием TestCase), и каждый метод того класса, начинающийся со слова “test”, считается одним тестом. Система заботится об обнаружении всех тестов, запускаемых один за другим с опциональными шагами setUp и tearDown, а также сообщает о результатах.

15.2. Пример

Вы найдете некоторые примеры тестов в tests/test_poetry.py . Для гарантии автономности (то есть не надо беспокоиться о настройках PYTHONPATH) мы скопировали весь необходимый код в модуль test. Обычно, конечно же, вы могли бы просто импортировать модули, которые вы бы хотели проверить.

Слудующий тест проверяет поэтический клиент и сервер, используя клиент для поэмы из тестового сервера. Для того, чтобы запустить сервер для тестирования, мы реализовали метод setUp в качестве нашего testcase’a:

```
class PoetryTestCase(TestCase):

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

Метод setUp создает сервер с тестовой поэмой и слушает случайный, открытый порт. Мы сохраняем номер порта, поэтому тесты могут его использовать. И, конечно же, мы завершаем тестовый сервер в функции tearDown, когда тесты завершены:

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

В нашем первом тесте test_client мы используем get_poetry для того, чтобы получить поэму из тестового сервера, и проверяем, что это та самая поэма, которую мы ожидаем:

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

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

        d.addCallback(got_poem)

        return d
```

Заметим, что наша тестовая функция возвращает deferred. При использовании trial, каждый метод выполняется как callback. Это означает, что reactor выполняется и мы можем производить асинхронные операции как часть теста. Нам только нужно дать системе знать, что наш тест асинхронный, и мы делаем это обычным способом - возвращаем deferred.

Система trial будет ждать до тех пор пока deferred не активизируется вызовом метода tearDown, и завершит тест с ошибкой, если deferred не выполнится (например, если последяя пара callback/errback завершилась с ошибкой). Наш тест также провалится, если наш deferred слишком долго активизируется (по умолчанию - 2 минуты). Это означает, что если тест завершился, мы знаем, что наш deferred активизировался, и поэтому наш callback активизировался и запускал метод assertEquals.

Наш второй тест, test_failure, проверяет, что get_poetry завершается с ошибкой определенным образом, если мы не можем соединиться с сервером:

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

Здесь мы пытаемся подсоединиться к невалидному порту и затем используем метод assertFailure из trial'a. Этот метод подобен иметоду assertRaises из модуля unittest, но используется для тестирования асинхронного кода. Он возвращает deferred, который успешно завершается, если заданный deferred генерирует заданное исключение, иначе - завершается с ошибкой.

вы можете самостоятельно запускать тесты, используя скрипт trial следующим образом:

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

И вы увидете некоторый вывод, показывающий каждый testcase и OK, сообщающий, что каждый тест пройден успешно.

15.3. Обсуждение

Поскольку trial похож на unittest в отношении к основному API, то достаточно просто начать писать тесты. Нужно только возвратить deferred, если Ваш тест использует асинхронный код, и trial позаботится обо всем. вы также можете возвращать deferred из методов setUp и tearDown, если они должны быть также асинхронными.

Любые логи из Ваших тестов будут собираться в файле внутри директории с названием _trial_temp, которую trial создаст автоматически, если ее нет. В дополнение к ошибкам, напечатанным на экране, лог - полезная начальная точка при отладке тестов с ошибками.

Рисунок 33 показывает гипотетический тест в процессе запуска:

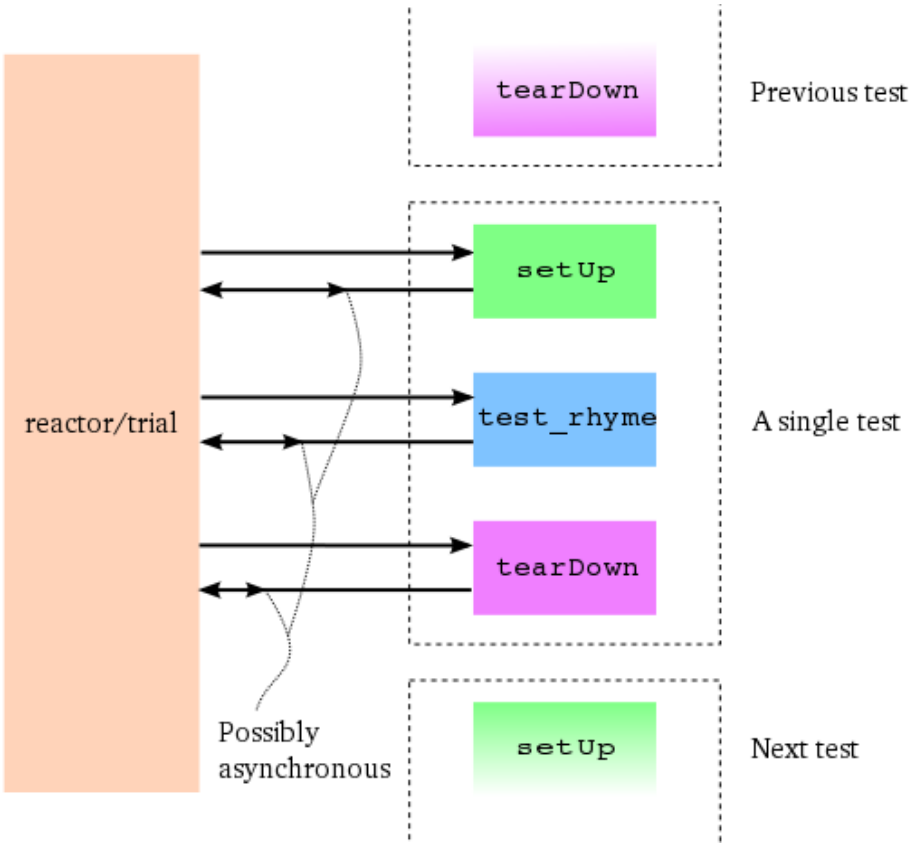

Рис. 33: Запущенный trial тест

Если вы когда-нибудь использовали подобные системы ранее, то вы должны быть знакомы с моделью, за исключением того, что все относящиеся к тестам методы могут возвращать deferred'ы.

Система trial - хорошая иллюстрация того как "работающая асинхронность" вызывает изменения во всей программе. Для того, чтобы тест (или любая функция) был асинхронным, он должен:

1. Не блокироваться
2. Зачастую, возвращать deferred

Но это означает, что любые вызовы функции должны готовы принять deferred и не блокироваться (возможно, также возвращать deferred). И так это условие распространяется далее. Таким образом, возникает необходимость в системе, подобной trial, которая может управлять асинхронными тестами, которые возвращают deferred'ы.

15.4. Резюме

Если вы хотите посмотреть больше примеров о том как писать unit тесты для кода Twisted, вам нужно смотреть не дальше, чем сам Twisted. Twisted имеет очень много unit тестов, с новыми добавленными при выходе каждого релиза. Поскольку эти тесты изучены Twisted экспертами, они являются прекрасным примером того, как правильно тестировать Twisted код.

В главе 16 мы будем использовать Twisted для того, чтобы сделать наш поэтический сервер подлинным демоном.

15.5. Упражнения

1. Поменяйте один из тестов так, чтобы вызвать ошибку, и запустите trial снова, чтобы посмотреть вывод программы.
2. Почитайте документацию по тестирования в Twisted[1].
3. Напишите тесты для других поэтических сервисов.
4. Исследуйте некоторые тесты в Twisted[2].

[1] http://twistedmatrix.com/documents/current/core/howto/testing.html
[2] http://twistedmatrix.com/trac/browser/trunk/twisted/test

16. Демонизация Twisted

16.1. Введение

Серверы, которые мы написали до сих пор, запускались только в терминальном окне, с выводом на экран через оператор print. Это подходит для программы на стадии разработки, но это едва ли подходит для релизов. Хорошо написанная программа, готовая для выкладке, должа:

1. Запускаться как демон, отсоединяясь от любой терминальной или пользовательской сессии. Сервис не должен завершаться после того, как администратор отлогился.
2. Отправлять отладочный и ошибочный вывод во множество ротируемых лог файлов или в сервис syslog.
3. Убирать лишние привилегии, например, переключаться к к низкопривелигированному пользователю до запуска.
4. Записывать свой pid в файл так, чтобы администратор мог без проблем отправить сигналы в демон.

Мы можем добраться до всех этих вещей, используя скрипт twistd, предоставляемый Twisted. Но сначала, мы должны быть немного поменять наш код.

16.2. Концепции

Для того, чтобы понять twisted, необходимо изучить несколько новых понятий в Twisted, наиболее важное из них - Service. Как всегда, несколько новых концепций сопровождаются новыми Interface'ми.

16.2.1. IService

Интерфейс IService опредлеяет именнованный сервис, который может быть запущен и остановлен. Интерфейс требует наличия небольшого множества основных атрибутов и методов.

Существует два обязательных атрибута: name и running. Атрибут name - это просто строка, например 'fastpoetry', или None, если вы не хотите давать Вашему сервису название. Атрибут running - булево значение, которое устанавливается в true, если сервис успешно запущен.

Мы собираемся коснуться лишь некоторых методов IService. Мы пропустим некоторые, являющиеся очевидными, и другие, которые наиболее продвинуты и в основном не используются в простых Twisted программах. Два основных метода IService - startService и stopService:

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


def stopService():
    """
```

```
        Stop the service.

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

Что эти методы делают будет зависеть от сервиса. Например, метод startService может любое из:

- Загружать некоторые конфигурационные данные
- Инициализировать базу данных
- Начать слушать порт
- Ничего не делать

Метод stopService может:

- Сохранять некоторое состояние
- Закрывать открытие соединения к базе данных
- Завершать слушать порт
- Ничего не делать

Когда мы пишем наши пользовательские сервисы, мы должны соответсвующе реализовать эти методы. Для некоторых общих случаев, подобно прослушиванию порта, Twisted предоставляет готовые сервисы, которые мы можем использовать.

Заметим, что stopService может опционально возвратить deferred, который нужно активизировать, когда сервис окончательно прекратил работать. Это позволяет нашим сервисам убрать за собой до того как приложение завершится. Если Ваш сервис сразу же завершается, вы можете возвратить None вместо deferred'а.

Сервисы могут быть организованы в коллекции, которые вместе стартуют и завершаются. Последний метод IService, на который мы будем смотреть - setServiceParent, добавляет Service в коллекцию:

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

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

Любой сервис может иметь родительский, что означает, что сервисы могут быть организованы в иерархию. И это подводит нас к следующему Interface'у, на который мы собираемся сегодня посмотреть.

16.2.2. IServiceCollection

The IServiceCollection interface defines an object which can contain IService objects. A service collection is a just plain container class with methods to:

Интерфейс IServiceCollection определяет объект, который может содержать объекты IService. Коллекция сервисов - просто обычный контейнерный класс с методами:

- Искать сервис по названию (getServiceNamed)
- Пройтись по всем сервисам в коллекции (__iter__)
- Добавить сервис в коллекцию (addService)
- Удалить сервис из коллекции (removeService)

Заметим, что реализация IServiceCollection не является автоматически реализацией IService, но нет препятсвий одному классу реализовать оба интерфейса (и мы увидим пример).

16.2.3. Application

Twisted Application не определяется отдельным интерфейсом. Скорее, объект Application требуется для реализации обоих IService и IServiceCollection, также как и для нескольких других, которые мы собираемся рассмотреть.

Приложение - это сервис верхнего уровня, который представляет наше полное Twisted приложение. Все другие сервисы в вашем демоне будут детьми (или внуками, и т.д.) объекта Application.

Редко, когда надо реализовывать свое собственное Application. Twisted предоставляет реализацию, которую мы будем сегодня использовать.

16.2.4. Twisted логирование

Twisted включает свое собственную инфраструктуру логирования в модуле twisted.python.log Основные API для записи в лог простые, поэтому мы просто включим небольшой пример, расположенный в basic-twisted/log.py, и вы можете просмотреть интересующие Вас детали в соответсвующем модуле Twisted.

Мы не будем вам докучать, показывая API для установки логирующих обработчиков, поскольку twistd делает это за нас.

16.3. FastPoetry 2.0

Хорошо, давайте взглянем на некоторый код. Мы обновили наш поэтический сервер для того, чтобы запускать его с помощью twistd. Его исходный код расположен в twisted-server-3/fastpoetry.py. Сначала мы посмотрим на PoetryProtocol:

```
class PoetryProtocol(Protocol):

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

Заметьте, что вместо использования оператора print, мы используем функцию twisted.python для записи каждого соединения. Далее класс PoetryFactory:

```
class PoetryFactory(ServerFactory):

    protocol = PoetryProtocol

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

Как вы видите, поэма больше не хранится в PoetryFactory, вместо этого она хранится в сервисном объекте, на которую есть ссылка из PoetryFactory. Заметьте, как PoetryProtocol получает поэму из service через factory. Наконец, далее сам сервисный класс:

```
class PoetryService(service.Service):

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

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

Так же как и с другими многочисленными классами Interface, Twisted предоставляет базовый класс, который мы можем использовать для того, чтобы сделать нашу собственную реализацию используя поведение по умолчанию. Здесь мы используем класс twisted.application.service.Service для реализации нашего PoetryService.

Базовый класс предоставляет реализацию по умолчанию всех необходимых методов, поэтому нам нужно реализовать только пользовательское поведение. В нашем случае, мы просто перегружаем startService для загрузки файла с поэзией. Заметим, что мы все равно вызываем метод базового класса (который устанавливает для нас атрибут running).

Другая сторона здесь плохо упомянута. Объект PoetryService не знает ничего о деталях PoetryProtocol. Сервис только должен загрузить поэму и обеспечить доступ до него любого объекта, который нуждается в поэме. Другими словами, PoetryService обеспокоен только высокоуровневыми деталями в преоставлении поэзии, а не низкоуровневыми деталями отправки по TCP соединению. Таким образом, этот сервисный класс мог бы быть использован другим протоколом, скажем UDP или XML-RPC. В случае нашего простого сервиса пользы от этого мало, но можно представить преимущества в более реалистичной сервисной реализации.

Если бы это была обычная Twisted программа, весь код, на который мы посмотрели до сих пор не был бы в одном файле, он был бы в нескольких разных модулях (возможно, fastpoetry.protocol и fastpoetry.service). Но, следуя нашей обычной практике создания примеров самодостаточными, мы включаем все что нам надо в один скрипт.

16.3.1. Twisted tac файлы

Оставшийся код в скрипте содержит то, что обычно бывает содержимым Twisted tac файла. Файл tac - Twisted Application Configuration файл, который говорит twistd как создавать приложение. Поскольку это конфигурационный файл, то он ответственнен за выбор настроек (подобно номерам портов, расположениями файлов с поэзией и т.д.) для запуска приложения определенным образом. Другими словами, tac файл представляет

определенный вид запуска для нашего сервиса (обслужить эту поэму на этом порту), а не общий скрипт для запуска любого поэтического сервера.

Если бы мы запускали несколько поэтических сервером на одной машине, мы бы имели несколько tac файлов для каждого (вот почему tac файлы обычно не содержат кода общего назначения). В нашем примере, tac файл сконфигурирован для обслуживания poetry/ecstasy.txt на порту 10000 loopback интерфейса:

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

Заметим, что twistd ничего не знает об этих переменных, мы их определяем здесь для того, чтобы сохранить все наши конфигурационные переменные в одном месте. Фактически, twistd заботится только об одной переменной во всем файле, как мы это вскоре поймем. Далее мы начинаем построение нашего приложения:

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

Наш поэтический сервер состоит из двух сервисов: PoetryService, который мы определили выше, и встроенный Twisted сервис, который создает слушающий сокет нашей поэмы, из которого она будет предоставляться. Поскольку эти два сервисы очевидным образом связаны друг с другом, мы сгруппируем их вмести, используя MultiService - класс Twisted, который реализует IService вместе с IServiceCollection.

Будучи сервисной коллекцией, MultiService сгруппирует наши два поэтических сервиса вместе. И будучи сервисом, MultiService запустит оба дочерних сервиса, когда MultiService сам запустится, и остановит их, когда сам остановится. Давайте добавим наш первый поэтический сервис в коллекцию:

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

Эта достаточно удобная вещь. Мы только создаем PoetryService и затем добавляем его в коллекцию, используя setServiceParent, который унаследован из базового класса Twisted. Затем мы слушаем TCP сокет:

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

Twisted предоставляет сервис TCPServer для создания слушающего TCP сокета, присоединенного к произвольной factory (в нашем случае - PoetryFactory). Мы не вызываем reactor.listenTCP напрямую, поскольку работа tac файла - сделать наше приложение готовым к старту не запуская его. TCPServer создаст сокет после того, как он запустится скриптом twistd.

Вы могли заметить, что мы не беспокоились заданием названия сервиса. Именование сервисов не является обязательным, а наоборот - опциональным свойством, которое вы

можете использовать, если хотите «просматривать» сервисы во время запуска. Поскольку нам не нужно этого делать в нашем маленьком приложении, то мы не беспокоимся об этом.

Хорошо, мы получили оба наших сервиса, объединенных в коллекцию. Теперь мы только сделаем Application и добавим нашу коллекцию в него:

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

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

Единственная переменная в этом скрипте, которая интересует twistd - переменная application. Это то, как twistd найдет приложение, которое предполагается запустить (поэтому переменная должна иметь название "application"). И когда приложение запустится, все сервисы, которые мы туда добавили, также будут запущены.

На рисунке 34 показана структура приложения, которое мы только что построили:

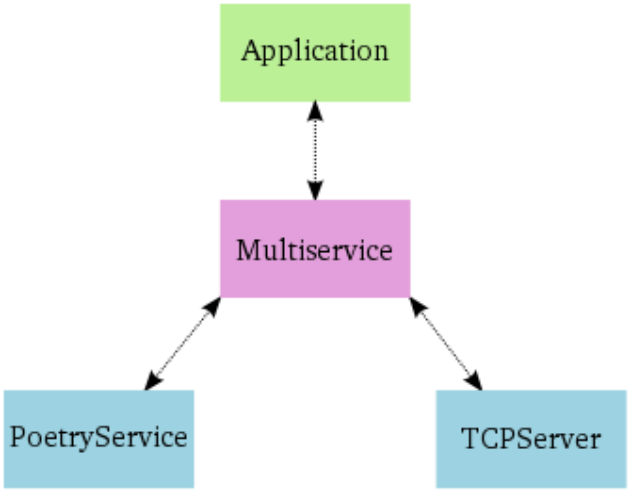

Рис. 34: Структура fastpoetry приложения

16.3.2. Запуск сервера

Давайте запустим наш сервер. Поскольку наш скрипт - tac файл, то нам нужно запустить его с помощью twistd. Конечно же, tac файлы - обычные Python скрипты. Давайте сначала запустим tac файл без twistd, просто используя интерпретатор Python и посмотрим, что получается:

```
python twisted-server-3/fastpoetry.py
```

Если вы сделаете это, вы обнаружите, что ничего не произошло! Как мы говорили до этого, работа tac файла - сделать приложение готовым для запуска, без его запуска. В качестве напоминание об этом специальном назначении tac файлов, некоторые люди называют их с расширением .tac вместо .py. Скрипт twistd не заботится о расширении файла.

Давайте запустим наш сервер по-настоящему, используя twistd:

```
twistd --nodaemon --python twisted-server-3/fastpoetry.py
```

После запуска такой команды вы увидите похожий вывод:

```
2010-06-23 20:57:14-0700 [-] Log opened.
2010-06-23 20:57:14-0700 [-] twistd 10.0.0 (/usr/bin/python 2.6.5) starting up.
2010-06-23 20:57:14-0700 [-] reactor class: twisted.internet.selectreactor.SelectReactor.
2010-06-23 20:57:14-0700 [-] __builtin__.PoetryFactory starting on 10000
2010-06-23 20:57:14-0700 [-] Starting factory <__builtin__.PoetryFactory instance at 0x14a
2010-06-23 20:57:14-0700 [-] loaded a poem from: poetry/ecstasy.txt
```

Здесь нужно отметить несколько вещей:

1. вы видите вывод системы логирования Twisted, в том числе вызов log.msg в PoetryFactory. Но мы не устанавливали logger в нашем tac файле, так как twistd должен это сделать для нас.

2. вы также можете увидеть, что наши два основных сервиса, PoetryService и TCPServer, запустились.

3. Командная строка не возвращается. Это означает, что наш сервер не запущен как демон. По умолчанию, twistd запускает сервер как демон (это основная причина существования twistd), но если вы используете опцию –nodaemon, то twistd запустит ваш сервер как обычный консольный процесс, и лог будет выводится на стандартный вывод. Это полезно при отладки ваших tac файлов.

Теперь протестируем сервер путем скачивания поэмы одним из наших поэтических клиентов или даже netcat'м:

```
netcat localhost 10000
```

Такая команда должна скачать поэму из сервера, и вы увидите новую запись в логе подобную такой:

```
2010-06-27 22:17:39-0700 [__builtin__.PoetryFactory] sending 3003 bytes of poetry to IPv4Ad
```

Эта запись появляется в результате вызова log.msg в PoetryProtocol.connectionMade. Если вы сделаете еще запросы к серверу, то на каждый запрос вы увидите новую запись в логе.

Теперь остановим сервер, нажав Ctrl-C. вы должны увидеть строки подобные:

```
^C2010-06-29 21:32:59-0700 [-] Received SIGINT, shutting down.
2010-06-29 21:32:59-0700 [-] (Port 10000 Closed)
2010-06-29 21:32:59-0700 [-] Stopping factory <__builtin__.PoetryFactory instance at 0x28d
2010-06-29 21:32:59-0700 [-] Main loop terminated.
2010-06-29 21:32:59-0700 [-] Server Shut Down.
```

Как вы видите, Twisted не просто разрушается, а завершает сам себя чисто и говорит вам о завершении в логе. Заметим, что два наших основных сервис также сами себя завершают.

Хорошо, теперь запустим сервер еще раз:

```
twistd --nodaemon --python twisted-server-3/fastpoetry.py
```

Откройте другую консоль и зайдите в директорию twisted-intro. В директории должен появится файл twistd.pid. Этот файл создан twistd и содержит идентификатор процесса нашего запущенного сервера. Попробуйте запустить эту команду для завершения сервера:

```
kill `cat twistd.pid`
```

Заметьте, что twistd стирает файл с pid'м, когда наш сервис завершается.

16.3.3. Реальный демон

Теперь давайте запустим наш сервер как демон, который даже проще сделать, поскольку по умолчанию twistd демонизирует процесс:

```
twistd --python twisted-server-3/fastpoetry.py
```

На этот раз мы сразу же получаем командную строку обратно. Если вы посмотрите на содержимое вашей директории, то в дополнение к файлу twistd.pid, созданное для сервера, который мы только что запустили, вы увидите файл с логами — twistd.log, который отображает тоже самое, что и ранее отображалось в консоли.

При запуске демона, twistd устанавливает обработчик логов, который пишет в файл вместо стандартного вывода. Лог-файл по умолчанию — это twistd.log, расположенный в той же директории, где вы запустили twistd, но, если вы хотите, то можете менять файл использованием опции *--logfile*. Обработчик, который устанавливает twistd также ротирует лог, когда размер превышает один мегабайт.

Вы можете увидеть запущенный сервер, просмотрев список всех процессов в вашей системе. Продолжим и проверим сервер, скачав еще одну поэму. Вы увидите новые записи в лог файле для каждого запроса на скачивание поэмы.

Поскольку сервер больше не присоединен к консоли (и другим процессам, кроме init), вы не можете завершить его нажав Ctrl-C. Как реальный демон, процесс продолжит работу, даже если вы отлогинетесь. Но мы все еще можем использовать файл twistd.pid, чтобы остановить процесс:

```
kill `cat twistd.pid`
```

И когда это происходит, то в логе появляются сообщения об остановке сервера, файл twistd.pid удаляется, и наш сервер завершается.

Это хорошая идея проверить другие опции twistd. Например, вы можете сказать twistd переключиться на другого пользователя или группу до запуска демона (это типичный случай понижения привелегий Вашего сервера, которые ему не нужны, в качестве меры предосторожности). Мы не будем беспокоиться тщательным изучением этих дополнительных опций, вы можете найти их, используя опцию –help.

16.4. Системы плагинов в Twisted

Хорошо, теперь мы можем использовать twistd для запуска наших серверов как подлинных демонов. Это все очень здорово, и тот факт, что наш “конфигурационные” файлы являются реально файлами с исходниками на Python’е, дает нам огромную гибкость во многих вещах. Но, нам не всегда нужно так много гибкости. Для наших поэтических серверов, обычно мы заботимся только о нескольких опциях:

1. Поэма для обслуживания
2. Порт для обслуживания поэмы
3. Интерфейс для прослушивания

Создавать новые tac файлы с простыми изменениями переменных кажется чрезмерным. Система плагинов в Twisted позволяет нам сделать тоже самое.

Twisted плагины предоставляют способ определения именованных приложений с пользовательскими опциями, которые twistd может автоматически распознать и запустить. Twisted сам построен из множества встроенных плагинов. вы можете их увидеть, запуская twisted без аргументов. Попробуйте запустить сейчас вне директории twisted-intro. После раздела с помощью, вы увидите вывод подобный:

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

Каждая строка показывает один из встроенных плагинов из Twisted. И вы можете любой из них, используя twistd.

Каждый плагин также имеет свое множество опций, которые вы можете изучить, используя опицию –help. Давайте посмотрим, какие опции имеет плагин ftp:

```
twistd ftp --help
```

Заметим, что нужно поместить опцию –help после команды ftp, поскольку вам нужны опции ftp, а не опции twistd. вы можете запустить ftp сервер с помощью twistd также, как мы запускали наш поэтический сервер. Но, поскольку это плагин, то мы запускаем его по его названию:

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

Эта команда запускает ftp плагин в недемонизированном виде на порту 10001. Заметим, что опция twistd *--nodaemon* помещается до названия плагины, в то время как опции, специфичные для плагина, помещаются после его названия. Так же как и наш поэтический сервер, вы можете остановить плагин с помощью Ctrl-C.

Ok, давайте превратим наш поэтический сервер в Twisted плагин. Но сначала, нам нужно осознать несколько новый концепций.

16.4.1. IPlugin

Любой Twisted плагин должен реализовывать интерфейс twisted.plugin.IPlugin. Если вы посмотрите на его объявление, то вы обнаружите, что он в действительности не определяет никакие методы. Реализация IPlugin - просто способ для плагина, сказать “Привет, я - плагин!” так, чтобы twistd мог найти его. Конечно же, чтобы его можно было использовать, он должен реализовать некоторый другой интерфейс, и мы вскоре до этого доберемся.

Но, как узнать, действительно ли объект реализует пустой интерфейс? Пакет zope.interface имеет функцию, которая позволяет указать то, что определенный класс реализует определенный интерфейс. Мы рассмотрим пример использования этой функции в plugin-версии нашего поэтического сервера.

16.4.2. IServiceMaker

В дополнение к IPlugin, наш плагин будет реализовывать интерфейс IServiceMaker. Объект, который реализует IServiceMaker, знает как создавать IService, который будет формировать основной элемент запуска application. IServiceMaker определяет три атрибута и метод:

1. tapname: строка с названием нашего плагина. Аббревиатура “tap” означает Twisted Application Plugin. Заметим, что более старые версии Twisted также использовали pickled application files, называемые “tapfiles”, но это функциональность устарела.

2. description: описание плагина, который twistd будет отображать как часть своего help текста.

3. options: объект, который описывает опции командной строки, которые принимает плагин.

4. makeService: метод, который создает новый объект IService по заданному множеству опций командной строки.

Мы увидим то, как все это соединяется вместе в следующей версии нашего поэтического сервера.

16.5. Fast Poetry 3.0

Теперь мы готовы посмотреть на плагин версию Fast Poetry, расположенную в twisted/plugins

Как вы могли заметить, название директорий отличается от других примеров. Это потому что twistd требует, чтобы плагин файлы были расположены в директории twisted/plugins (вам не нужно создавать никаких __init__.py файлов); вы можете иметь несколько директорий twisted/plugins в PYTHONPATH, и twisted все их найдет. Название файла, которое вы используете для плагина, не имеет значения, но все таки хорошая идея именовать его согласно тому приложению, которое он представляет, подобно тому как мы это сделали.

Первая часть нашего плагина состоит из тех же protocol, factory и service реализаций, как и наш tac файл. И так же как и раньше, этот код обычно находится в разных модулях, но мы поместим его в плагин, для того, чтобы сделать пример самодостаточным.

Далее идет объявление комнадных опций плагина:

```
class Options(usage.Options):

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

Этот код определяет опции, специфичные для плагина, которые пользователь может поместить после названия плагина при использовании twistd. Мы не будем вдаваться в подробности, так как тут все достаточно понятно. Теперь, мы перейдем к основной части нашего плагина - классу PoetryServiceMaker:

```
class PoetryServiceMaker(object):

    implements(service.IServiceMaker, IPlugin)

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

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

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

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

```
                                        interface=options['iface'])
        tcp_service.setServiceParent(top_service)

        return top_service
```

Здесь вы видите как функция zope.interface.implements используется для объявления того, что наш класс реализует оба IServiceMaker и IPlugin. На этот раз, нам не нужно создавать объект Application, мы только создаем и возвращаем сервис более высокого уровня, который запустит наш application, и twistd позаботится обо всем остальном. Заметим то, как мы используем аргумент options для того, чтобы получить опции командной строки, специфичные для плагина, переданные twistd.

После объявления класса, осталась только одна вещь, которую надо сделать:

```
service_maker = PoetryServiceMaker()
```

Скрипт twistd обнаружит экземпляр нашего плагина и использует его для создания сервиса высокого уровня. В отличие от tac файла, переменная name, которую мы выбираем, неважна. Важно, что наш объект реализует оба IPlugin и IServiceMaker.

Посколе того, как мы создали наш плагин, давайте запустим его. Убедитесь, что вы находитесь в директории twisted-intro или, что эта директория находится в PYTHONPATH. Затем, попробуйте запустить twistd. вы должны увидеть, что “fastpoetry” в списке плагинов вместе с описанием из нашего файла с плагином.

Вы также заметите, что создался новый файл dropin.cache в директории twisted/plugins. Этот файл создается twistd для ускорения последующих поисков плагинов.

Теперь давайте получим некоторую помощь по использованию нашего плагина:

```
twistd fastpoetry --help
```

Вы увидите опции, специфичные для плагина fastpoetry в тексте помощи. И наконец, давайте запустим наш плагин:

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

Эта команда запустит сервер fastpoetry как демон. Как и раньше, вы увидите оба файла twistd.pid и twistd.log в текущей директории. После тестирования нашего сервера, вы можете остановить его:

```
kill `cat twistd.pid`
```

Вот как создавать Twisted плагины.

16.6. Резюме

В этой главе мы изучили то, как превращать Twisted сервера в долгоживущие демоны. Мы затронули систему логирования Twisted и как использовать twistd для запуска Twisted приложений ввиде демонизированных процессов, как создавать tac конфигурационные файлы, как создавать и запускать Twisted плагины. В главе 17 мы возвратимся к более фундаментальной теме асинхронного программирования и посмотрим на другой способ структурирования наших callback’в в Twisted.

16.7. Упражнения

1. Поменяйте tac файл для обслуживания второй поэмы на другом порту. Сохраните сервисы для каждой поэмы отдельно, используя другой объект MultiService.
2. Создайте новый tac файл, который запустит поэтический прокси сервер.
3. Модифицируйте файл plugin для того, чтобы он принимал необязательный второй поэтический файл и второй порт, для обслуживания.
4. Создайте новый плагин для поэтического прокси сервера.

17. Еще один способ написания callback'в

17.1. Введение

В этой главе мы возвратимся к предмету callback'в. Мы изучим еще одну технику написания callback'в в Twisted, которая использует генераторы. Мы покажем как эта техника работает и сравним с использованием «чистых» Deferred'ов. Наконец, мы перепишем один из наших поэтических клиентов, используя эту технику. Но сначала, давайте посмотрим на то, как работают генераторы, чтобы мы могли увидеть, почему они являются кандидатами для создания callback'в.

17.1.1. Краткий обзор генераторов

Как вы вероятно знаете, генератор в Python'е - это "перезапускаемая функция", которую вы создаете используя выражение yield в ее теле. Делая так, функция становится "генераторной функцией", которая возвращает итератор, который вы можете использовать для запуска функции как серии шагов. Каждый цикл итератора перезапускает функцию, которая продолжает выполняться до тех пор пока не достигнет следующего вызова yield.

Генераторы (и итераторы) зачастую используются для представления лениво созданных последовательностей значений. Давайте посмотрим на пример кода в inline-callbacks/gen-1.py:

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

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

Здесь мы имеем генератор, который создает последовательность 1, 2, 3. Если вы запустите этот код, то вы увидите, что операторы print в генераторе, чередуются с оператором print в цикле, проходящему по генератору.

Вы можете сделать этот код более явным путем создания самого генератора (inline-callbacks/gen-2.py):

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

gen = my_generator()
```

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

Генератор — это просто объект для получения последовательных значений. Мы также можем рассмотреть вещи с точки зрения самого генератора:

1. Функция-генератор не начинает запуск до тех пор, пока она не будет вызвана циклом (используя метод next).

2. Запущенный генератор продолжает выполняться до тех пор, пока он не возвращается циклу (используя yield).

3. Когда цикл запускает другой код (подобный оператору print в примере), генератор не выполняется.

4. Когда генератор выполняется, цикл не выполняется (блокируется, ожидая возврата генератора).

5. Когда генератор уступает контроль циклу, может пройти произвольное количество времени (может быть выполнено произвольное количество кода) до того, как генератор запустится снова.

Это очень похоже на то, как работают callback'и в асинхронной системе. Мы можем представить, что цикл while - reactor, а генератор - серия callback'в, разделенных операторами yield, с интересным фактом, что все callback'и разделяют одно и то же локальное пространство переменных, и пространство имен сохраняется от между всеми вызовами callback'в.

Кроме того, вы можете иметь несколько одновременно активных генераторов (посмотрите пример в inline-callbacks/gen-3.py), с их чередующимися "callback'ми", также как вы можете иметь независимые асинхронные задачи, запущенные в системе, подобной Twisted.

Хотя, что-то еще пропущено. Callback'и не только вызываются реактором, они также получают информацию. Как часть цепочки deferred'а, callback либо получает результат, в форме одного Python значения, либо ошибку в форме объекта типа Failure.

Начиная с Python 2.5, функциональность генераторов была расширена, что позволило отправлять информацию генератору, когда он перезапускается; пример inline-callbacks/gen-4.py иллюстрирует это свойство:

```
class Malfunction(Exception):
    pass

def my_generator():
    print 'starting up'

    val = yield 1
    print 'got:', val

    val = yield 2
```

```
    print 'got:', val

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

    yield 4

    print 'done'

gen = my_generator()

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

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

В Python 2.5 и выше, оператор yield - выражение, которое возвращает значение. А код, который перезапускает генератор, может установить это значение, используя метод send вместо next (если вы используете next, то значение - None). Более того, вы можете генерировать произвольное исключение внутри генератора, используя метод throw. И это круто.

17.2. Встроенные callback'и

Взяв за основу то, что мы только что просмотрели об отправке значений и генерировании исключений в генераторе, мы можем представить генератор как серию callback'в, подобных callback'ам в deferred'е, которые получают либо результаты, либо ошибки. callback'и разделены yield'ми и значения каждого yield выражения - результат для следующего callback'а (или yield генерирует исключение в случае ошибки). На рисунке 35 показано соответствие:

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

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

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

Рис. 35: Генератор как последовательность callback'в

Когда серия callback'в соединена вместе в deferred, каждый callback получает результат из предыдущего. Это достаточно просто сделать и с генератором - просто отправить значение, которое имеется от предыдущего запуска генератора, в следующий раз, когда вы перезапускаете его. Это кажется немного глупо. Поскольку генератор вычислил значение, почему нужно заботится об отправке его назад? Генератор может просто сохранить значение в переменной до следующего раза, когда она понадобится. Так какой в этом смысл?

Вспомним факт, который мы изучили в главе 13, когда callback'и в deferred'е могли возвращать deferred'ы. И когда такое происходило, внешний deferred останавливался до тех пор, пока внутренный deferred активизировался, и затем следующий callback (или errback) в цепочке внешнего deferred'а вызывался с результатом (или ошибкой) из внутреннего deferred'а.

Представим, что наш генератор производит объект deferred вместо обычного Python значения. Генератор приостановится и это происходит автоматически; генераторы всегда приостанавливаются после каждого оператора yield до тех пор, пока они явно не перезапустятся. Таким образом мы можем приостановить перезапуск генератора до тех пор, пока deferred активизируется, в этот момент мы либо отправляем значение (если deferred завершился успешно), либо генерируем исключение (если deferred завершился с ошибкой). Это могло бы сделать наш генератор подлинной последовательностью асинхронных callback'в, что является основной идеей в реализации функции inlineCallbacks в twisted.internet.defer.

17.2.1. inlineCallbacks

Рассмотрим теперь пример inline-callbacks/inline-callbacks-1.py:

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

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

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

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

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

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

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

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

```
    reactor.stop()

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

Запустите пример, и вы увидите выполнение генератора, который по завершению останавливает reactor. Пример иллюстрирует несколько свойств функции inlineCallbacks.

Во-первых, inlineCallbacks - декоратор, и он всегда декорирует генераторные функции, то есть функции, которые используют yield. Основная цель inlineCallbacks - превратить генератор в серию асинхронных callback'в, согласно тому, как мы обсуждали это ранее.

Во-вторых, когда мы вызываем декоратор inlineCallbacks, то нам не нужно самим вызывать методы next, send или throw. Декоратор сам вызывает эти методы и гарантирует, что генератор выполнит весь код (предполагая, что в нем не произошло исключения).

В-третьих, если в генераторе порождается значение, не являющееся deferred'ом, то генератор сразу же возвратит себе управление, и yield вернет это значение.

И наконец, если мы порождаем deferred из генератора, то генератор не получит управления обратно до тех пор, пока deferred не активизируется. После успешного завершения deferred'a, управление отдается обратно к генератору, и yield возвращает то, что возвратил завершившийся deferred. Если deferred завершается с ошибкой, то оператор yield генерирует исключение. Заметим, что исключение - это просто обычный Exception объект, а не объект типа Failure, и мы можем поймать его с помощью оператора try/except, обернутого вокруг yield.

В примере мы использовали callLater для активизации deferred'в после короткого периода времени. Это удобный способ установить неблокирующую задержку в нашу callback цепочку, в реальных более сложных програмах, мы бы порождали deffered, возвращенный некоторой другой асинхронной операцией (например, get_poetry), вызванной из нашего генератора.

Хорошо, теперь мы знаем как запускать функции с декоратором inlineCallbacks, но какое значение возвратит 'yield 1'? Ответ - deferred. Поскольку мы не можем точно знать, когда генератор остановится (он может породить один или несколько deferred'ов), декорированная функция сама асинхронная и deferred является подходящим возвращяемым значением.

Поскольку мы заранее не можем сказать, когда завершится выполнение генератора, так как он может породить несколько deferred'ов, то декоратор inlineCallbacks - асинхронная функция и deferred - самое подходящее значение в качестве возвращаемого значения yield. Заметим, что для возвращаемого deferred'a, генератор не может вызвать yield. Заметим, что возвращаемый генератором deferred не порождается оператором yield. Напротив, этот deferred может активизироваться только после того как генератор полностью завершился (или выкинул исключенение).

Если генератор выкидывает исключение, то возвращаемый deferred активизирует свою цепочку errback с исключением, обернутым в Failure. Но, если вы хотите, чтобы генератор возвращал обычное значение, вы должны использовать функцию defer.returnValue. Подобно обычному оператору return, он также остановит генератор (в действительности, он генерирует специальное исключение). Пример inline-callbacks/inline-callbacks-2.py иллюстрирует обе возможности.

17.3. Клиент 7.0

Давайте включим inlineCallbacks в новую версию нашего поэтического клиента. Код находится в twisted-client-7/get-poetry.py. Можно сравнить с кодом клиента 6.0 в twisted-client-6/get-poetry.py. Изменения в основном коснулись poetry_main:

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

    xform_addr = addresses.pop(0)

    proxy = TransformProxy(*xform_addr)

    from twisted.internet import reactor

    results = []

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

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

        defer.returnValue(poem)

    def got_poem(poem):
        print poem

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

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

    reactor.run()
```

В новой версии генератор get_transformed_poem с декоратором inlineCallbacks отвечают за скачивание поэмы и ее преобразование (через трасформирующий сервер). Поскольку обе операции асинхронные, мы порождаем (оператором yield) deferred и затем

явно ждем результат. Так же как и клиент 6.0, если преобразование завершилось с ошибкой, то поэма не меняется. Заметим, что мы можем использовать операторы try/except для управления асинхронными ошибками внутри генератора.

Мы можем проверить новый клиент так же как и раньше. Сначала запустим трансформирующий сервер:

```
python twisted-server-1/tranformedpoetry.py --port 10001
```

Затем запустим несколько поэтических серверов:

```
python twisted-server-1/fastpoetry.py --port 10002 poetry/fascination.txt
python twisted-server-1/fastpoetry.py --port 10003 poetry/science.txt
```

Теперь можно запустить новый клиент:

```
python twisted-client-7/get-poetry.py 10001 10002 10003
```

Попробуйте отключить один или несколько серверов, для того, чтобы посмотреть как клиент управляет ошибками.

17.4. Обсуждение

Подобно объекту Deferred, функция inlineCallbacks дает новый способ организации наших асинхронных callback'в. Так же как и с deferred'ми, inlineCallbacks не меняет правила игры. Наши callback'и запускаются один за раз и вызываются реактором. Мы можем убедиться в этом, распечатав traceback из внутреннего callback'а, так, как это сделано в примере inline-callbacks/inline-callbacks-tb.py. Запустите пример, и вы увидите traceback, начинающийся с reactor.run(), завершающийся нашим callback'м и несколько вспомогательных функций между ними.

Мы можем адаптировать рисунок 29, который объясняет как один callback в deferred'е возвращает другой deferred, чтобы показать, что случается, когда генератор inlineCallbacks порождает deferred (при помощи yield). Посмотрите на рисунок 36.

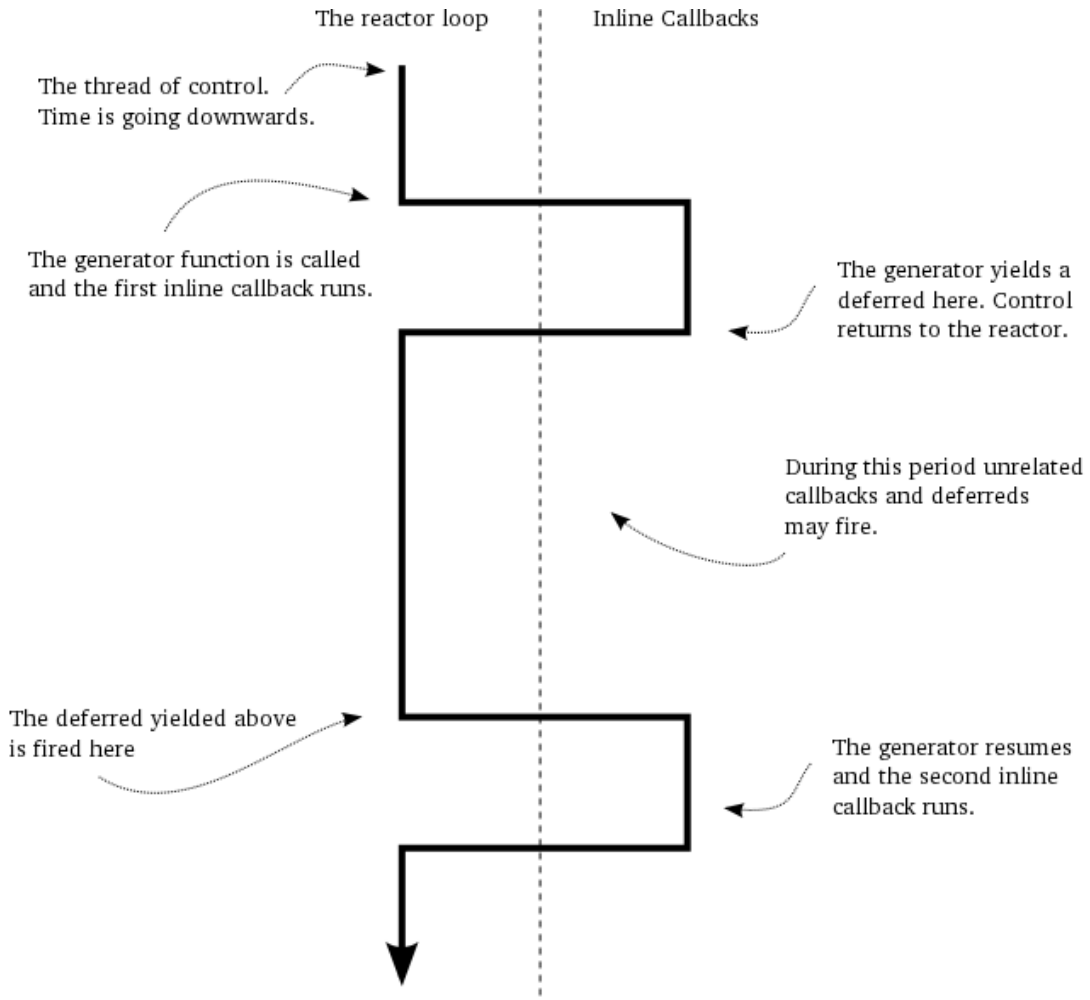

Рис. 36: Поток выполнения в функции inlineCallbacks

Оба рисунка иллюстрируют одну и ту же идею: одна асинхронная операция ожидает другую.

Поскольку inlineCallbacks и deferred'ы решают много идентичных проблем, что лучше выбрать? Существует несколько преимуществ inlineCallbacks:

- Посколько callback'и разделяют пространство имен, не нужно использовать дополнительное состояние.
- Порядок callback'в легче просматривается, так как они выполняются сверху вниз.
- Без объявлений функций для индивидуальных callback'ов и явным потоком управления в общем случае получается меньше текста.
- Ошибки обрабатываются с помощью оператора try/except.

Также возможны следующие проблемы:

- Callback'и внутри генератора не могут быть вызваны индивидуально, что делает сложным переиспользование кода. С deferred'ми, при конструировании кода, можно просто добавить произвольные callback'и в произвольном порядке.
- Компактная форма генератора делает неявным факт использования асинхронных callback'ов. Несмотря на визуальное сходство с обычными последовательными функциями, генератор выполняется отличным образом. Функция inlineCallbacks - не способ избежать изучения асинхронной модели программирования.

Также как и с любой другой технологией, практика обеспечит необходимым опытом для правильного выбора между callback'ми и генератором, обернутым в декоратор inlineCallbac

17.5. Резюме

В этой главе мы изучили декоратор inlineCallbacks и о том, как он позволяет нам выражать последовательность асинхронных callback'ов в форме Python генератора.

В следующей главе мы изучим технику управления множеством "параллельных" асинхронных операций.

17.6. Упражнения

1. Почему название функции inlineCallbacks во множественном числе?
2. Изучите реализацию функции inlineCallbacks и ее управляющую функцию _inlineCallbacks. Подумайте над фразой "дьявол в деталях".
3. Сколько callback'в в генераторе с N операторами yield, предполагая, что нет циклов и условных операторов (if)?
4. Поэтический клиент 7.0 может иметь три генератора, запущенных одновременно. Концептуально, сколько различных способов чередования их выполнения? Учитывая способ, которым они вызываются в поэтическом клиенте и реализацию inlineCallbacks как вы думаете, сколько способов действительно возможно?
5. Переместите callback got_poem в клиент 7.0 внутрь генератора.
6. Переместите callback poem_done внутрь генератора. Будьте внимательны! Убедитесь, что управляете всеми случаями ошибок, чтобы reactor завершился не смотря ни на что.
7. Генератор с оператором yield внутри цикла while представляет собой бесконечный цикл. Что представляет собой генератор с декоратором inlineCallbacks?

18. Deferred'ы в целом

18.1. Введение

В предыдущей главе мы изучили новый способ структурирования последовтельных асинхронных callback'ов с использованием генератора. Таким образом, включая deferred'ы, у нас теперь две техники для связывания асинхронных операций вместе.

Иногда нужно запустить группу асинхронных операций "параллельно". Поскольку Twisted однотредовый, то асинхронные операции реально не запускаются параллельно, мы просто указываем место, где хотим использовать асинхронный ввод-вывод. Например, наши поэтические клиенты скачивают поэмы из разных серверов одновременно, а не с одного сервера, а потом с другого.

Как узнать, что все асинхронные операции, которые были начаты, завершены? До сих пор мы решали эту задачу, собирая наши результаты в список (подобно списку результатов в клиенте 7.0) и проверяли длину списка. Мы должны были тщательно собирать как успешные запуски, так и неуспешные, иначе один неуспешный запуск мог вызвать то, что программа запускалась всегда, предполагая, что еще остались незавершенные задачи.

Как вы догадываетесь, Twisted включает абстракцию, которую вы можете использовать для решения этой проблемы, и мы посмотрим на это в этой главе.

18.2. DeferredList

Класс DeferredList позволяет нам обрабатывать список объектов типа Deferred как один deferred. Таким способом мы можем запустить пачку асинхронных операций и получить уведомление, только тогда, когда все они завершены (несмотря на успешный или ошибочный результат). Давайте рассмотрим несколько примеров.

В примере deferred-list/deferred-list-1.py вы найдете следующий код:

```
from twisted.internet import defer

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

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

Если вы запустите, то получите следующий вывод:

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

Нужно отметить следующее:

- DefferedList создается из Python списка. В первом примере список пустой, но мы увидим, что элементы списка должны быть объектами типа Deffered.
- DeferredList - сам является deferred'ом, так как унаследован от Deferred. Это означает, что вы можете добавлять callback'и и errback'и в него, так, как будто это обычный deferred.

- В примере выше, наш callback был активизирован сразу же после добавления, так что DeferredList должен активизироваться сразу же. Мы обсудим это позже.
- Результат deferred списка - список (пустой).

Давайте теперь помотрим на deferred-list/deferred-list-2.py:

```
from twisted.internet import defer

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

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

Теперь мы создаем DeferredList с одним элементом в списке. Получаем следующий вывод:

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

Нужно отметить следующее:

- На этот раз DeferredList не активизировал свой callback, до тех пор пока мы не активизировали deferred в списке.
- Результат - это все еще список, но теперь с одним элеметом.
- Элемент в возвращенном списке - кортеж, у которого второе значение - результат deferred'а в списке.

Давайте попробуем добавить два deferred'а в список(deferred-list/deferred-list-3.py):

```
from twisted.internet import defer

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

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

Вывод следующий:

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

С этого момента достаточно ясен результат DeferredList, по меньшей мере способ, которым мы его использовали, - список элементов по количеству deferred'ов в списке, который мы передали в конструктор. И элементы списка результатов содержат результаты первоначальных deferred'ов, по меньшей мере deferred'ов, которые успешно завершились. Это означает, что сам по себе DeferredList не активизируется до тех пор пока все deferred'ы в списке не активизированы. И DeferredList, созданный с пустым списком, активизируется сразу же, поскольку нет deferred'в, завершения которых надо ожидать.

Что по поводу порядка резульатов в списке с результатами? Рассмотрим deferred-list/deferred-list-4.py:

```
from twisted.internet import defer

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

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

На этот раз мы активизировали сначала d2, а потом d1. Заметим, что список deferred'ов все еще создается с d1 и d2 в их первоначальном порядке. Далее вывод:

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

Результирующий список имеет результаты в то же самом порядке, что и первоначальный список deferred'ов, порядок не соответвует очередности запуска deferred'ов. Что очень хорошо, поскольку мы можем связать каждый индивидуальный результат с действием, которое их порождает (например, какая поэма с какого сервера пришла).

Что происходит, если один или более deferred'ов в списке завершились с ошибкой? И что здесь делают значения True? Давайте попробуем пример deferred-list/deferred-list-5.py:

```
from twisted.internet import defer

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

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

Теперь мы активизировали d1 с положительным результатом, а d2 - отрицательным. Игнорируя опцию consumerErrors, мы вернемся к этому, вывод будет следующий:

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

Теперь кортеж соответсвующий d2 имеет Failure в качестве второго элемента и False - в качестве первого. С этого момента должно быть достаточно понятно, как работает DeferredList:

- DeferredList конструируется со списком объектов типа Deferred.
- DeferredList сам deferred, с результатом ввиде списка той же длины, что и список deferred'ов.
- DeferredList активизируется после того как все deferred'ы в первоначальном списке активизированы.
- Каждый элемент резуьтирующего списка соответсвует deferred'у в той же позиции, что и в первоначальном списке. Если deferred завершился успешно, то элемент - (True, result), иначе - (False, failure).
- DeferredList никогда не завершается с ошибкой, поскольку результат каждого отдельного deferred'а собирается в список в любом случае.

Давайте теперь поговорим об опции consumeErrors в конструкторе DeferredList. Если мы запускаем код deferred-list/deferred-list-6.py, в котором не используется эта опция, вывод получается такой:

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

Если вы вспомните, то сообщение “Unhandled error in Deferred” генерируется, когда deferred собирается сборщиком мусора, и последний callback в deferred’е завершился с ошибкой. Это сообщение говорит нам, что мы не поймали все потенциальные асинхронные ошибки в нашей программе. И откуда это приходит в нашем примере? Ясно, что это не приходит от DeferredList, посколько DeferredList всегда запускается успешно. Поэтому, это приходит от d2.

DeferredList нужно знать, когда каждый из deferred’ов, которые он мониторит, активизировались. И DeferredList делает это обычным образом - путем добавления callback и errback к каждому deferred’у. По умолчанию, callback (и errback) возвращают первоначальный результат (или ошибку) после их помещения в окончательный список. Возвращая первоначальный Failure из errback тригеров следующему errback, d2 остается в состоянии ошибки после активизации.

Но если мы подставляем consumeErrors=True в DeferredList, errback, добавленный в DeferredList для каждого deferred’а, возвратит None, таким образом потребляя ошибку и выдавая предуреждение. Мы также можем управлять ошибкой, добавляя свой собственный errback в d2, так как это делается в deferred-list/deferred-list-7.py.

18.3. Клиент 8.0

В версии 8.0 поэтического клиента используется DeferredList для поиска, когда вся поэзия успешно завершилась (или завершилась с ошибкой). Новый поэтический клиент можно найти в twisted-client-8/get-poetry.py. Изменения коснулись только poetry_main. Давайте посмотрим на важные изменения:

```
...
ds = []

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

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

В клиенте 8.0 вам не нужен callback poem_done или список результатов. Вместо этого, мы помещаем каждый deferred, который мы получаем от get_transformed_poem в список (ds) и затем создаем DeferredList. Посколько DeferredList не будет активизироваться до тех пор пока все поэмы не завершаться успешно или с ошибкой, мы добавляем в DeferredList callback, останавливающий reactor. В этом случае, мы не используем результат из DeferredList, мы только хотим знать, когда все завершилось.

18.4. Обсуждение

На рисунке 37 визуализация того, как работает DeferredList:

Все достаточно просто. Существует несколько опций в DeferredList, которые мы не обсудили, и которые меняют поведение того, что мы обсудили выше. Мы вернемся к ним позже.

В следующей части мы узучим еще одну особенность класса Deferred, которая была недавно добавлена в Twisted 10.1.0.

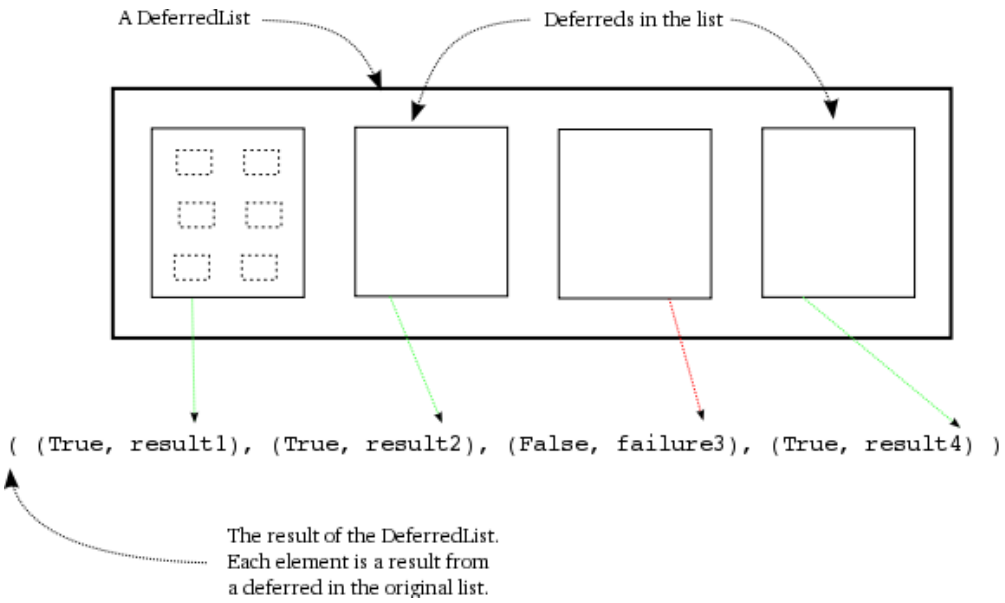

Рис. 37: Результат DeferredList'a

18.5. Упражнения

1. Прочитайте исходники DeferredList.
2. Поменяйте примеры в deferred-list для экспериментирования с опциями конструктора fireOnOneCallback и fireOnOneErrback. Продумайте ситуации, когда вы бы использовали один или другой аргумент (или оба).
3. Можете ли вы создать DeferredList, используя список DeferredList'ов? Если да, как бы выглядел результат?
4. Поменяйте клиент 8.0 так, чтобы он не ничего не печатал до тех пор, пока все поэмы не скачались. На этот раз вы будете использовать результат из DeferredList.
5. Определите семантику DeferredDict и реализуйте этот класс.

19. Отмена deferred'ов

19.1. Введение

Twisted развивающийся проект и разработчики Twisted регулярно добавляют новые свойства и расширяют старые. В релиз Twisted 10.1.0, разработчики добавили новое свойство аннулирования в классе Deffered, которое мы собираемся изучить в этой главе.

Асинхронное программирование отделяет запросы от ответов, и таким образом генерирует новую возможность: между запросом результата и получением его обратно, вы можете решить больше не ждать. Рассмотрим поэтический прокси сервер из главы 14. Далее то, как работает прокси, по меньшей мере для первого запроса поэмы:

1. Приходит запрос поэмы.
2. Прокси соединяется с реальным сервером для получения поэмы.
3. После того, как вся поэма получена, отправляет ее запрашивающему клиенту.

Все достаточно хорошо, но что, если клиент завис до получения поэмы? Может сначала клиент запросил одну поэму, потом передумал и решил, что ему надо другую. Теперь наш прокси застрял, скачивая первую поэму, и нашему медленному серверу понадобится некоторое время, чтобы ее передать. Лучше закрыть соединение и позволить медленному серверу опять ожидать нового запроса.

Вспомним рисунок 15, на котором изображена диаграмма, которая показывает поток выполнения в синхронной программе. На этом рисунке мы видим низходящий поток вызовов функций и восходящий поток исключений. Если мы хотим аннулировать синхронный вызов фукнкции, направление потока выполнения было такое же как и в случае вызова функции, от высокоуровнего кодо до низкоуровнего, как это показано на рисунке 38.

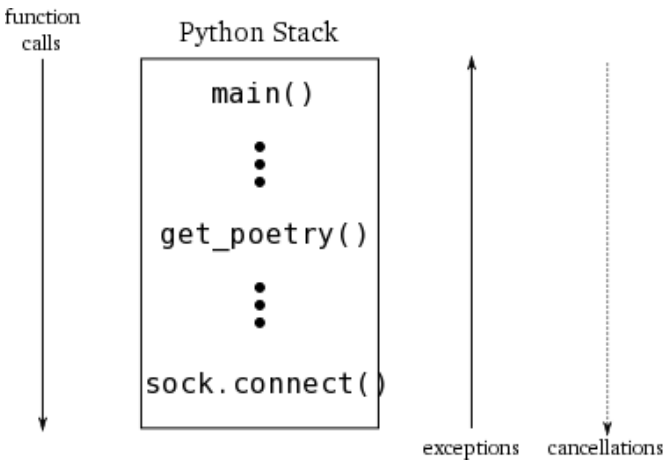

Рис. 38: Поток выполнения в синхронной программе в случае гипотетического аннулирования

Конечно же, в синхронной программе это невозможно, поскольку высокоуровневый код не может приостановиться до того момента, как низкоуровневые операции не завершатся, но с этого момента уже нечего аннулировать. Но в асинхронной программе высокоуровневый код получает контроль до того, как завершится низкоуровневый код, который может сгенерировать отмену низкоуровневого кода до момента его завершения.

В Twisted программе, низкоуровневый запрос реализуется объектом Deferred, про который вы можете думать как об обработчике невыполненной асинхронной операции.

Обычный поток информации в deferred'е низходящий, от низкоуровневого кода до высокоуровнего, который сопоставляет поток возвращенной информации в синхронной программе. Начиная с Twisted 10.1.0, высокоуровневый код может отправить информацию обратно в другом направлении: он может сказать низкоуровневому коду, что не надо больше возобновляться. Посмотрите на рисунок 39.

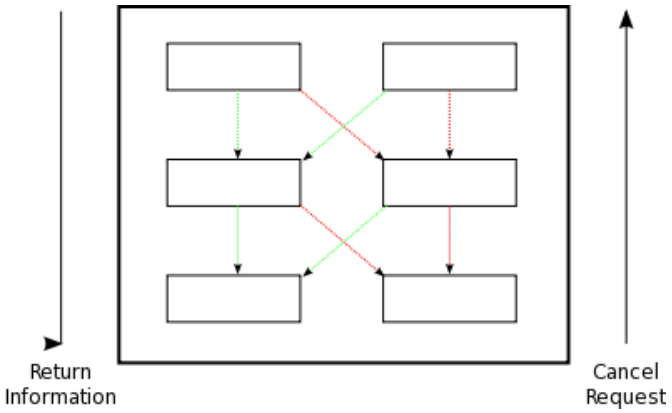

Рис. 39: Информационный поток в deferred'е, включая аннулирование

19.2. Аннулирующиеся deferred'ы

Давайте посмотрим на несколько примеров программ, чтобы увидеть как в действительности работает аннулирование deferred'ов. Чтобы запускать примеры из этой главы, нужно иметь Twisted версии 10.1.0 или выше. Рассмотрим deferred-cancel/defer-cancel-1.py:

```
from twisted.internet import defer

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

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

С новым свойством аннулирования, класс Deferred имеет новый метод с названием cancel. В примере создается deferred, добавляется callback и затем вызывается функция cancel без активизации deferred'а. Далее вывод:

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

Таким образом аннулирование deferred'а вызывает запуск цепочки errback, и обычный callback никогда не вызывается. Заметим, что ошибка типа twisted.internet.defer.CancelledError, является пользовательским Exception, означающим, что deferred был аннулирован. Давайте добавим errback в deferred-cancel/defer-cancel-2.py:

```
from twisted.internet import defer

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

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

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

Теперь мы получаем следующий вывод:

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

То есть мы можем поймать аннулирование в errback'е подобно тому, как возникновление обычной ошибки в deferred'е.

Давайте попытаемся активизировать deferred и затем его дективизируем так, как в примере deferred-cancel/defer-cancel-3.py:

```
from twisted.internet import defer

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

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

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

Здесь мы активизировали deferred обычным образом методом callback и затем его деактивизировали. Далее вывод:

```
callback got: result
done
```

Наш callback был вызван (так как мы и ожидали), и программа нормально завершилась, так будто cancel никогда не вызывался. Таким образом, аннулирование уже активизиврованного deferred'а не оказывает влияния.

Что если мы активизируем deferred после его аннулирования так, как это сделано в deferred-cancel/defer-cancel-4.py?

```
from twisted.internet import defer

def callback(res):
```

```
    print 'callback got:', res

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

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

В этом случае мы получаем следующий вывод:

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

Та же выдача, что и во втором примере, где мы никогда не активизировали deferred. Но почему вызов d.callback('result') не спровоцировал ошибку, из предположения, что deferred не может быть активизирован более одного раза и из предположения, что цепочка errback явно была запущена?

Посмотрим снова на рисунок 39. Активизация deferred'a с результатом или ошибкой - низкоуровневая задача, в то время как сброс deferred'a - действие, производимое высокоуровневым кодом. Активизация deferred'a означает "Вот Ваш результат", в то время как сброс deferred'a означает "Я больше этого не хочу". Запомните, что сброс - новое свойство, поэтому большая часть Twisted кода не написана для управления операциями сброса. Но разработчики Twisted сделали возможным для нас сбрасывать любые deferred's, даже, если имеющийся код был написан до Twisted 10.1.0.

Для того, чтобы сделать это возможным, метод cancel в действительности делает две вещи:

1. Скажи объекту Deferred, что ты не хочешь получить результат, если он еще не сформировался (например, deferred еще не был активизирован), и проигнорируй любые последовательные вызовы callback и errback.

2. Опционально: скажи низкоуровневому коду, который производит результат, принять любые меры, требующиеся для отмены операции.

Для совместимости со старыми версиями Twisted отмененный deferred все равно активизируется, шаг 1 гарантирует, что наша программа не будет блокироваться, если мы отменим deferred из более старой версии библиотеки.

Это означает, что мы всегда можем отменить deferred, и мы будем уверены, что не получим результат, если он еще не прибыл (даже, если прибудет позже). Но отмена deferred'a может в действительности не отменить асинхронную операцию. Сброс асинхронной опрации требует контекстно-специфичного действия. вам нужно закрыть сетевое соединение, откатить транзакцию к базе данных, убить подпроцесс и т. д. Поскольку deferred - просто callback организатор общего назначения, как он узнает, что нужно выполнить какое-то специфичное дейтсвие при его отмене? Или, иначе, как он мог бы перенаправить запрос на отмену низкоуровневому коду, который создал и возвратил deferred? Ответ - с помощью callback'a.

19.3. Действительно отмененные Deferred'ы

Взглянем на deferred-cancel/defer-cancel-5.py:

```
from twisted.internet import defer

def canceller(d):
    print "I need to cancel this deferred:", d

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

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

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

Этот код в основном подобен второму примеру, за тем лишь исключением, что существует третий callback (canceller), который подставляется в Deferred, когда мы его создаем, а не после создания. Этот callback несет ответсвенность за выполнение контекстно-специфичных действий, требующихся для прерывания асинхронной операции (только, если deferred действительно отменяется, конечно же). callback canceller - необходимая часть низкоуровневого кода, который возвращает deferred, это не высокоуровневый код, который получает deferred и добавляет свои callback'и и errback'и.

Запущенный пример производит вывод:

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

Как вы видите, callback canceller вызывается deferred'ом при отмене. В эту функцию помещаются любые действия, которые нужны для отмены асинхронной операции. Заметим, что canceller вызывается до того, как активизируется цепочка errback. Фактически, мы можем выбрать активизировать самим deferred в этом месте с любым результатом или ошибкой на наш выбор (таким образом, упреждая ошибку CancelledError). Обе возможности проиллюстрированы в deferred-cancel/defer-cancel-6.py и deferred-cancel/defer-cancel-7.py.

Давайте сделаем еще один простой тест, до того как мы запустим reactor. Создадим deferred с callback'ом canceller, активизируем его и отменим. Пример кода в deferred-cancel/defer-cancel-8.py. Изучая вывод этого скрипта, мы можем понять, что отмена deferred'а после его активизации не вызывает callback функцию canceller. И это то, что мы и ожидали, поскольку нечего отменять.

Примеры, на которые мы смотрели, в действительности не содержат асинхронных операций. Давайте сделаем простую программу, которая вызывает одну асинхронную операцию, затем мы выясним, как сделать возможность отменять эту операцию. Рассмотрим код deferred-cancel/defer-cancel-9.py:

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

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

def get_poem():
    """Return a poem 5 seconds later."""
    from twisted.internet import reactor
    d = Deferred()
    reactor.callLater(5, send_poem, d)
    return d

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

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

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

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

    reactor.run()

main()
```

Этот пример включает функцию get_poem, которая использует метод rector'a callLater, для того, чтобы возвратить поэму через 5 секунд после вызова get_poem. Функция main вызывает get_poem, добавляет пару callback/errback, затем запускает reactor. Мы также останавливаем reactor через 10 секунд (снова используя callLater). Обычно мы бы делали это добавлением callback'a в deferred, но позже мы увидим, почему мы делаем это таким способом.

Запуск примера производит следующий вывод (после соответсвующей задержки):

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

После 10 секунд наша маленькая программа завершается. Теперь давайте попробуем отменить deferred до того, как отправится поэма. Мы добавим небольшой код для отмены deferred'a через 2 минуты:

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

Исходники нового примера находятся в deferred-cancel/defer-cancel-10.py, при запуске получаем вывод:

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

Этот пример ясно иллюстрирует, что отмена deferred'f не обязательно отменяет асинхронный вызов. После 2 секунд мы увидим вывод нашего errback'a, печатающего CancelledError как мы и ожидали. Затем, после 5 секунд, мы увидим вывод из send_poem (но callback в deferred'е не будет активизироваться).

В этом месте , мы в той же ситуации, что и в примере deferred-cancel/defer-cancel-4.py. Отмена deferred'a вызывает то, что окончательный результат игнорируется, но не вызывает прерывания операции. Как мы изучили выше, чтобы сделать реальную возможность отмены deferred'a, мы должны добавить отменяющий callback при создании deferred'a.

Что должен делать новый callback? Посмотрим в документацию для метода callLater. Возвращаемое значение callLater - еще один объект, реализующий IDelayedCall, с методом cancel, который мы можем использовать для того, чтобы предотвратить выполнение отложенных вызовов.

Обновленный код находится в deferred-cancel/defer-cancel-11.py. Значимые изменения находятся в функции get_poem:

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

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

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

    d = Deferred(canceler)

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

    return d
```

В этой новой версии, мы сохраняем возвращаемое значение фукции callLater, поэтому мы можем использовать его в нашем cancel callback'е. Единственное, что наш callback должен сделать - вызвать delayed_call.cancel(). Как мы обсудили ранее, мы могли бы также выбрать активизацию deferred'a. Последняя версия нашего примера выводит:

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

Как видите, deferred отменился и асинхронная операция была дейтсвительно прервана (например, мы больше не видим вывод функции send_poem).

19.4. Поэтический прокси 3.0

Как мы обсуждали во введении, поэтический прокси сервер - хороший кандидат для реализации отмены, так как это позволяет нам прерывать скачивание поэмы, если так получилась, что она никому не нужна (например, клиент закрывает соединение до того, как мы отправили поэму). Версия прокси 3.0 находится twisted-server-4/poetry-proxy.py и реализует отмену deferred'ов. Первое изменение в PoetryProxyProtocol:

```
class PoetryProxyProtocol(Protocol):

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

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

Вы можете сравнить со старой версией. Два основных изменения:

1. Сохраняет deferred, который мы получили из get_poem так, чтобы мы могли позже его отменить, если понадобится.
2. Отменяет deferred при закрытии соединения. Заметим, что deferred'ы также отменяются, после того, как мы действительно получили поэму, но, как мы поняли из примеров, отмена активизированного deferred'а не оказывает никакого воздействсвия.

Теперь нам нужно убедиться в том, что отмена deferred'а действительно прерывает скачивание поэмы. Для этого мы должны изменить ProxyService:

```
class ProxyService(object):

    poem = None # the cached poem

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

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

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

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

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

Можно сравнить со старой версией. В этом классе несколько больше изменений:

1. Мы сохраняем возвращаемое значение из reactor.connectTCP, объект IConnector. Мы можем использовать метод для сохраненного объекта для закрытия соединения.

2. Мы создаем deferred с callback'ом canceler. Этот callback является замыканием, которое использует connector для закрытия соединения. Но сначала атрибут factory.deferred выставляется в None. Иначе, factory может активизировать errback deferred'а с ошибкой "закрытое соединение" до того, как deferred сам активизирует errback с ошибкой CancelledError. После того, как deferred был отменен, иметь активизированный deferred с CancelledError, кажется более явным.

вы можете заметить, что теперь мы создаем deferred в ProxyService вместо PoetryClientFacto
Поскольку callback canceler нуждается в доступе к объекту IConnector, ProxyService - наиболее удобное место для создания deferred'а.

Также, как в одном из наших ранних примерах, наш callback canceler реализован как замыкание. Замыкания кажутся очень полезными при реализации отменяющих callback'ов!

Давайте испытаем наш проки сервер. Сначала запустим медленный сервер. Он должен быть медленным для того, чтобы мы действительно имели время отменить:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
```

Теперь мы можем запустить наш прокси (потребуется Twisted 10.1.0 или выше):

```
python twisted-server-4/poetry-proxy.py --port 10000 10001
```

Теперь мы можем начать скачивать поэму из сервера, используя любой клиент, или даже curl:

```
curl localhost:10000
```

Через несколько секунд нажмите Ctrl-C для того, чтобы остановить клиент или curl процесс. В терминале, где запущен прокси, вы должны увидеть следующий вывод:

```
Fetching poem from server.
Canceling poem download.
```

Также вы должны увидеть, что медленный сервер остановил печатать вывод поэмы, которую он отправляет, поскольку наш прокси завершил соединение. вы можете начинать и останавливать клиент несколько раз для того, чтобы проверить, что каждое скачивание каждый раз отменяется. Но если вы позволите поэме завершиться до конца, затем прокси закеширует поэму и будет отправлять сразу же.

19.5. Еще один полезный совет

Выше мы говорили несколько раз о том, что при отмене уже активизированных deferred'ов ничего не происходит. Но это не совсем правда. В главе 13 мы изучили, что callback'и и errback'и, присоединенные к deferred'у, могут сами возвращать deferred. И в этом случае, первоначальный (внешний) deferred приостанавливает выполенение своей цепочки callback и ожидает активизации внутреннего deferred'а.

Таким образом, даже хотя deferred активизировал высокоуровневый код, который делает асинхронный запрос, который мог не получить еще результат, поскольку цепочка

callback приостановлена в ожидании завершения внутреннего deferred'а. Что происходит, если высокоуровневый код отменяет внешний deferred? В этом случае внешний deferred не отменяет сам себя (он уже активизирован); вместо этого, внешний deferred отменяет внутренний deferred.

Когда вы отменяете deferred, вы можете не отменить основную асинхронную операцию, а некоторую другую асинхронную операцию, срабатываемую как результат первой.

Мы можем проиллюстрировать это еще одним примером. Рассмотрим код в deferred-cancel/defer-cancel-12.py:

```
from twisted.internet import defer

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

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

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

def second_outer_callback(res):
    print 'second outer callback got:', res

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

outer_d = defer.Deferred(cancel_outer)
inner_d = defer.Deferred(cancel_inner)

outer_d.addCallback(first_outer_callback)
outer_d.addCallbacks(second_outer_callback, outer_errback)

outer_d.callback('result')

# at this point the outer deferred has fired, but is paused
# on the inner deferred.

print 'canceling outer deferred.'
outer_d.cancel()

print 'done'
```

В этом примере мы создали два deferred'а, внешний и внутренний, один внешний callback возвращает внутренний deferred. Сначала активизируется внешний deferred, затем он отменяется. Пример печатает следующий вывод:

```
first outer callback, returning inner deferred
canceling outer deferred.
inner cancel callback.
outer errback got: [Failure instance: Traceback (failure with no frames):
<class 'twisted.internet.defer.CancelledError'>:]
done
```

Как вы можете видеть, отменяющийся внешний deferred не вызывает активизацию callback'а cancel. Вместо этого, он отменяет внутренний deferred так, что активизируется внутренний cancel callback, затем внешний errback получает CancelledError (из внутреннего deferred'а).

19.6. Обсуждение

Отмена deferred'а может быть очень полезной операцией, позволяющей нашим программам избежать работы, которую им больше не надо делать. И, как мы увидели, есть несколько хитростей.

Один очень важный факт, который надо помнить, - то, что отмена deferred'а не обязательно отменяет асинхронные операции. Фактически, как было написано, большинство deferred'ов не будут реально "отменяться", поскольку большая часть кода Twisted написана до версии Twisted 10.1.0, включая многие API самого Twisted. Прочитайте документацию по исходным кода для того, чтобы понять будет ли отменный deferred действительно отменять запрос или будет игнорировать отмену.

Второй важный факт - возвращая deferred из вашего асинхронного API не будет обязательно делать отмену в полном смысле этого слова. Если вы хотите реализовать отмену в ваших программах, вы должны изучить исходные коды Twisted, чтобы найти больше примеров. Отмена - новой свойство и как лучше использовать это свойство находится на стадии изучения.

19.7. Заглядывая в будущее

С этого момента мы изучили все о Deferred, основной концепцией Twisted. Оставшееся в Twisted состоит в основном из определенных приложений, подобно web программированию или асинхронному доступу к базе данных. Таким образом, в следующей главе мы собираемся немного изучить две других системы, которые используют асинхронный ввод-вывод, для того чтобы посмотреть как их идеи связаны с идеями в Twisted. Затем, мы предложим пути дальнейшего изучения Twisted.

19.8. Упражнения

1. Знали ли вы, что можно написать "cancel" с одной или двумя l? Это правда. Все зависит от настроения.
2. Внимательно прочитайте исходный код класса Deferred, уделяя особое внимание реализации отмены.
3. Поищите исходники Twisted 10.10 для примеров deferred'ов с cancel callback'ми. Изучите их реализацию.
4. Сделайте deferred, возвращаемый методом get_poetry одного из наших поэтических клиентов, отменяемым.
5. Сделайте пример, основанный на реакторе, который иллюстрирует отмену внешнего deferred'а, который был приостановлен внутренним deferred'ом. Если вы будете использовать callLater, вам нужно будет осторожно выбирать задержки для того, чтобы гарантировать, что внешний deferred отменяется в нужный момент.
6. Найдите асинхронный API в Twisted, который не поддерживает отмену и реализуйте отмену для него. Предложите патч в проект Twisted. Не забудьте про unit тесты!

20. Колеса внутри колес: Twisted и Erlang

20.1. Введение

Один факт, который мы еще не рассмотрели это то, что смешивание обычного синхронного Python кода с асинхронным Twisted кодом - задача непростая, поскольку блокирование на непродолжительный период времени в Twisted программе может испортить все то, что вы пытались достичь, используя асинхронную модель.

Если это ваше первое введение в асинхронное программирование, то может показаться, что приобретенное знание имеет ограниченную применимость. Вы можете использовать все эти новые техники внутри Twisted, но не в большинстве Python программ. При работе с Twisted, в основном, вы ограничены библиотеками, написанными специально для использования как части Twisted программы, поменьшей мере, если вы хотите их вызывать напрямую из потока, в котором запущен реактор.

Но техника асинхронного программирования используется много где и довольно давно и едва ли ограничена Twisted. Для Python'а существует много других асинхронных систем. Немного поискав, вы найдете достаточно много реализаций. Детально они отличаются от Twisted, но основные идеи (асинхронный ввод-вывод, обработка данных маленькими порциями среди множество потоков данных) остаются те же. Поэтому, если вам нужно использовать другую систему, то вы уже будете иметь преимущество, изучив Twisted.

Перемещаясь за пределы Python'а, существует много других языков и систем, которые основаны или используют асинхронную модель программирования. Ваше знание Twisted будет служить вам при открытии более широкой области этого предмета.

В этой главе мы очень коротко посмотрим на Erlang[1] - язык программирования и исполняющая система (runtime system), которая активно использует концепции асинхронного программирования, но делает это уникальным образом. Обратите внимание, что далее не следует общее введение в Erlang. Скорее, это короткое раскрытие некоторых идей, заложенных в Erlang, и как они связаны с идеями в Twisted. Основная тема - знания, которые вы приобрели, изучая Twisted, могут быть применены при изучении других технологий.

20.2. Переосмысленные обратные вызовы

Рассмотрим графическое представление обратного вызова (callback) на рисунке 6. Основной callback в поэтической клиенте 3.0, представленном в главе 6, и во всех последующих поэтических клиентах - метод dataReceived. Этот callback вызывается каждый раз, когда мы получаем часть поэзии из поэтических серверов, к которым мы присоединились.

Допустим, наш клиент скачивает три поэмы из трех различных серверов. Посмотрим на вещи с точки зрения реактора (на этой точки зрения мы акцентировали внимание во многих главах), мы имеем один большой цикл, который делает один или более обратных вызовов при каждой проходе. Посмотрим на рисунок .

Этот рисунок показывает, что реактор успешно вращается вокруг, вызывая *dataReceived* по приходу поэзии. Каждый вызов *dataReceived* применяется к одному из определенных экземпляров нашего класса *PoetryProtocol*. И мы знаем, что существует три экземпляра, поскольку мы скачиваем три поэмы (и поэтому должно быть три соединения).

Давайте думать о картинке с точки зрения одного из наших экземпляров Protocol. Этот экземпляр "видит" поток при вызове метода, который каждый раз приносит следующую порцию поэмы:

[1] http://erlang.org/

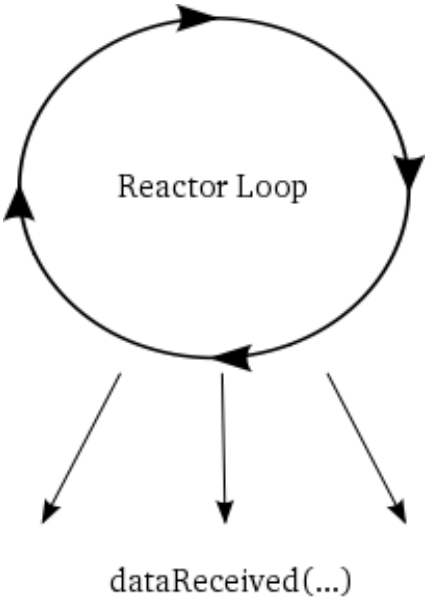

Рис. 40: обратные вызовы с точки зрения реактора

```
dataReceived(self, "When I have fears")
dataReceived(self, " that I may cease to be")
dataReceived(self, "Before my pen has glea")
dataReceived(self, "n'd my teeming brain")
...
```

Поскольку это похоже на цикл в Python'е, мы можем концептуализировать это так:

```
for data in poetry_stream(): # pseudo-code
    dataReceived(data)
```

Мы можем представить себе это "цикл обратных вызовов" на рисунке 41.

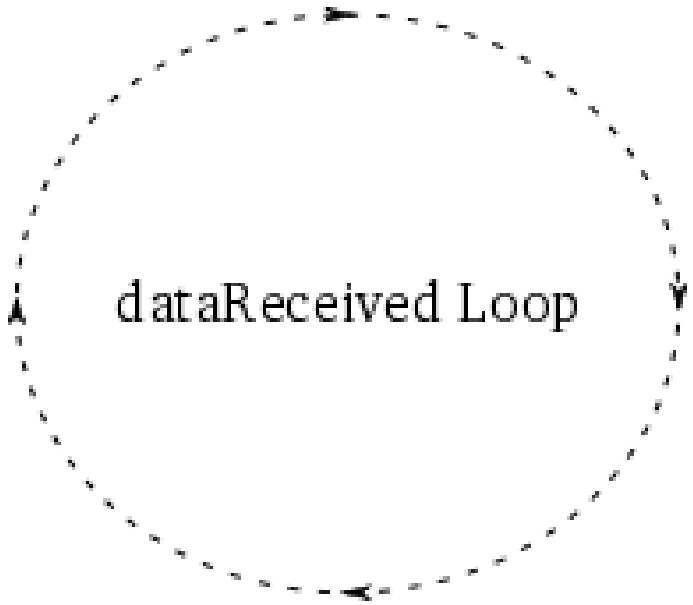

Рис. 41: Виртуальный цикл обратные вызовов

И снова, это не цикл *for* и не цикл *while*. Единственно существенным циклом в наших поэтических клиентах является реактор. Но мы можем думать о каждом Protocol'е, как о виртуальном цикле, который прокручивается каждый раз при появлении поэмы. Имея это в виду, мы можем представить заново весь клиент на рисунке 42.

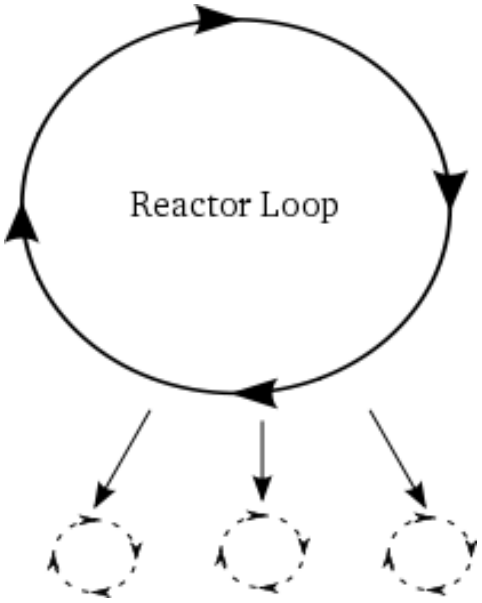

Рис. 42: Реактор, вращающий некоторые виртуальные циклы

На этом рисунке изображен один большой цикл (реактор) и три виртуальных цикла (отдельные экземпляры поэтического протокола). Большой цикл крутится и, крутясь, вызывает движение виртуальных циклов как будто это множество сцепленных шестеренок.

20.3. Беремся за Erlang

Erlang[1] подобно Python'у - динамически типизированный язык общего назначения, созданный в 80'х. В отличии от Python'а, Erlang больше функциональный, чем объектно-ориентированный, и имеет синтаксис, напоминающий Prolog[2], язык в котором Erlang был первоначально реализован. Erlang был спроектирован для построения высоко надежных распределенных телефонных систем, поэтому Erlang поддерживает множество вещей, связанных с сетью.

Одна из наиболее заметных особенностей Erlang - это параллелизм, основанный на легковесных процессах. Процесс в Erlang - это не процесс и не тред в операционной системе. Скорее, это независимо выполняющиеся функции внутри среды выполнения Erlang'а со своим собственным стеком. Erlang процессы это не легковесные треды, поскольку Erlang процессы не могут разделять состояние (и большинство типов данных неизменяемы (immutable), поскольку Erlang - функциональный язык программирования). Процессы Erlang'а могут взаимодействовать с другими процессами Erlang'а только отправлением сообщений, и сообщения всегда, по меньшей мере концептуально, копируются и никогда не разделяются).

Таким образом Erlang программа могла бы выглядеть так как рисунке 43.

На этом рисунке отдельные процессы стали «реальными», поскольку процессы - конструкции первого класса в Erlang'е, подобно объектам в Python'е. Среда исполнения становится «виртуальной» не потому, что ее нет, а потому что нет необходимости в цикле.

[1] http://erlang.org/

[2] http://ru.wikipedia.org/wiki/Prolog

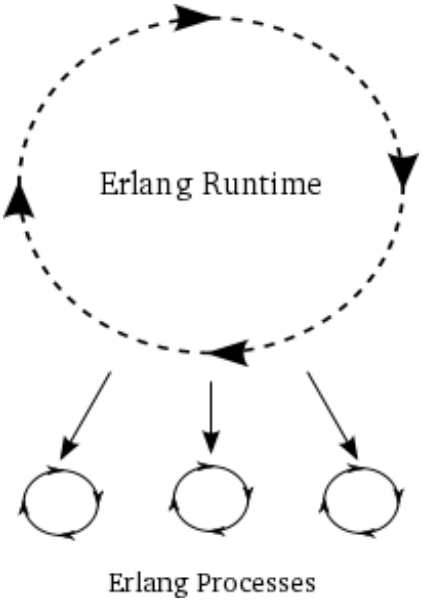

Рис. 43: Erlang программа с тремя процессами

Среда Erlang может быть многотредовой, и она несет ответственность не только за управление асинхронным вводом-выводом. Более того, языковая среда - это не дополнительная конструкция, подобно реактору в Twisted, это среда, в которой Erlang обрабатывает и исполняет код.

Поэтому, рисунок 44 лучше соответсвует Erlang программе.

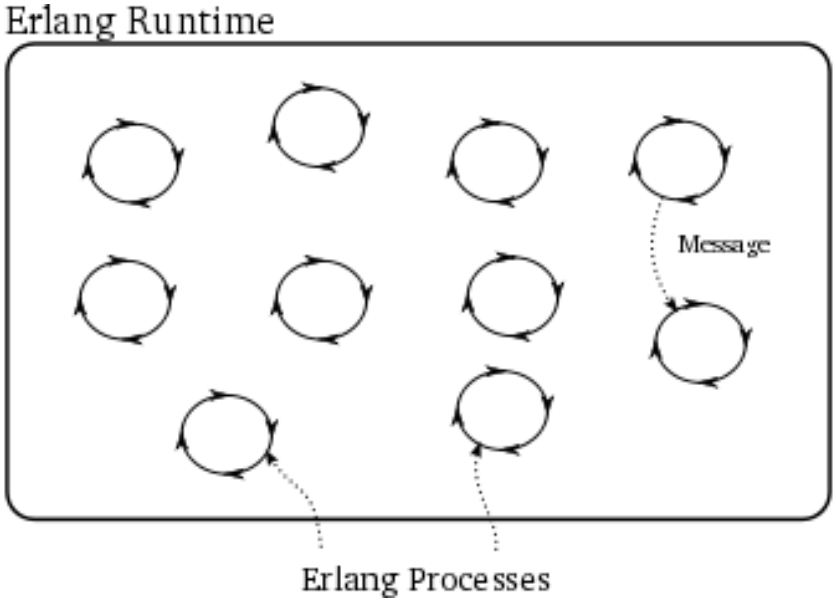

Рис. 44: Erlang программа с несколькими процессами

Конечно, среда Erlang должна использовать асинхронный ввод-вывод и один или более select-циклов, поскольку Erlang позволяет создавать большое количество процессов. Большие Erlang программы могут запустить десятки или сотни Erlang процессов, поэтому создание реального потока операционной системы для каждого из них просто вне обсуждения. Если Erlang собирается позволить множеству процессов выполнить ввод-вывод, и позволить другим процессам выполняться при блокировании на вводе-выводе, то понятна необходимость использования асинхронного ввода-вывода.

Заметьте, что наш рисунок Erlang программы имеет каждый процесс, запущенный «своей собственной силой», а не незапущенный обратными вызовами. И это очень важно. С задачей реактора, включенной в среду Erlang, обратный вызов не имеет больше такого ключевого значения. Что решалось бы в Twisted использованием обратных вызовов, в Erlang решалось бы отправлением асинхронного сообщения из одного Erlang процесса в другой.

20.4. Erlang поэтический клиент

Давайте посмотрим на поэтический клиент, написанный на Erlang'е. Мы сразу же перейдем к работающей версии вместо медленного построения подобно тому, как мы это делали в Twisted. И снова напоминаю, что это не полноценное введение в Erlang. В конце главы приведены ссылки на дополнительный материал.

Исходный код клиента находится в erlang-client-1/get-poetry. Для того, чтобы его запустить, нужно, чтобы был установлен Erlang. Далее код функции main, которая служит тем же целям, что и функция main для наших клиентов, написанных на Python'е:

```
main([]) ->
    usage();

main(Args) ->
    Addresses = parse_args(Args),
    Main = self(),
    [erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
     || {TaskNum, Addr} <- enumerate(Addresses)],
    collect_poems(length(Addresses), []).
```

Если вы никогда раньше не видели Prolog или подобный язык, то синтаксис Erlang'а вам покажется странным. Но многие люди тоже самое говорят о Python'е. В функции main определены два отдельных условия, которые разделены точкой с запятой. Erlang выбирает какое условие запустить, сравнивая аргументы, так что первое условие запускается только, если выполняется клиент без аргументов командной строки, и он печатает только сообщение с помощью (usage). Второе условие - это то, где выполняются все действия.

Отдельные операторы в Erlang функции разделяются запятыми, и все функции завершаются точкой. Давайте рассмотрим каждую строку во втором операторе. Первая строка анализирует аргументы командной строки и привязвает их к переменной (все переменные в Erlang должны начинаться с заглавной буквы). Вторая строка использует функцию *self* Erlang'а для того, чтобы получить идентификатор процесса текущего запущенного Erlang процесса (не процесса операционной системы). Поскольку это функция main, то вы можете подумать о ней как об эквиваленте модуля __main__ в Python'е. Третья строчка гораздо интересней:

```
[erlang:spawn_monitor(fun () -> get_poetry(TaskNum, Addr, Main) end)
     || {TaskNum, Addr} <- enumerate(Addresses)],
```

Этот оператор - список в Erlang'е с синтаксисом, похожим на синтаксис в Python'е. Он создает новые Erlang процессы, один на каждый поэтический сервер, к которому нам надо присоединиться. И каждый процесс запустит одну и ту же функцию (*get_poetry*), но с различными аргументами, специфичными только для того сервера. Мы также передаем идентификатор главного процесса так, чтобы новые процессы могли отправить поэзию обратно (идентификатор процесса нужен, чтобы отправить ему сообщение).

Последний оператор в функции main вызывает функцию *collect_poems*, которая ожидает возврата поэзии, и завершения процессов *get_poetry*. Мы кратко рассмотрим другие функции, но сначала сравните функцию main, написанную на Erlang, с эквивалентом, написанным на Twisted для одного из наших поэтических клиентов.

Теперь давайте посмотрим на Erlang функцию *get_poetry*. В действительности у нас есть две функции с названием *get_poetry*. В Erlang функция идентифицируется названием и арностью (количеством аргументов), так что наш скрипт содержит две разных функции: *get_poetry/3* и *get_poetry/4*, которые соответственно принимают три и четыре аргумента. Далее функция *get_poetry/3*, которая порождается *main*:

```
get_poetry(Tasknum, Addr, Main) ->
    {Host, Port} = Addr,
    {ok, Socket} = gen_tcp:connect(Host, Port,
                                   [binary, {active, false}, {packet, 0}]),
    get_poetry(Tasknum, Socket, Main, []).
```

Эта функция сначала делает TCP соединение, подобно функции *get_poetry* Twisted клиента. Затем, TCP соединение продолжает использоваться в вызове *get_poetry/4*, показанной ниже:

```
get_poetry(Tasknum, Socket, Main, Packets) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Packet} ->
            io:format("Task ~w: got ~w bytes of poetry from ~s\n",
                      [Tasknum, size(Packet), peername(Socket)]),
            get_poetry(Tasknum, Socket, Main, [Packet|Packets]);
        {error, _} ->
            Main ! {poem, list_to_binary(lists:reverse(Packets))}
    end.
```

Эта Erlang функция выполняет работу *PoetryProtocol* из Twisted клиента, за тем исключением, что она продолжает использование блокирующих вызовов функций. Функция *gen_tcp:recv* ожидает до тех пор, пока порция данных не прибудет в сокет (или сокет не закроется), однако ожидать она может долго. Но ”блокирование” функции в Erlang только блокирует процесс, запускающий функцию, а не всю среду Erlang. Рассмотренный TCP сокет в действительности не блокирующийся сокет (вы не можете создать реально блокирующийся сокет, используя чистый Erlang). Для каждого из Erlang сокетов существует где-то внутри среды Erlang «настоящий» TCP сокет, установленный в неблокирующее состояние и использованный как часть select-цикла.

Но Erlang процесс не должен про все это знать. Он просто ожидает некоторых данных, и если он блокируется, то выполняется другой Erlang процесс. И даже, если процесс никогда не блокируется, среда Erlang в произвольный момент времени может переключиться на выполнение другого процесса. Другими словами, Erlang основано на некооперативной модели параллелизма (non-cooperative concurrency model).

Заметьте, что *get_poetry/4* после получения порции поэмы продолжает рекурсивно себя вызывать. В императивном языке это подобно рецепту для переполнения стека и выхода за пределы доступной памяти, но Erlang компилятор может оптимизировать хвостовые вызовы (функциональные вызовы, которые являются последними операторами в функции) и превращать их в циклы. И это подчеркивает еще одну любопытную параллель между Erlang и Twisted клиентами. В Twisted клиенте, «виртуальные» циклы создаются реактором многократным вызовом одной и той же функции (dataReceived) снова и снова. А в Erlang клиенте «действительные» процессы (*get_poetry/4*) образуют цикл путем вызова самих себя снова и снова с использованием оптимизации хвостового вызова.

Если соединение закрывается, последнее, что делает *get_poetry* - это отправляет поэму обратно процессу main. Это также завершает выполнение *get_poetry*, поскольку делать уже нечего.

Оставшаяся ключевая функция нашего Erlang клиента это collect_poems:

```
collect_poems(0, Poems) ->
    [io:format("~s\n", [P]) || P <- Poems];
collect_poems(N, Poems) ->
    receive
        {'DOWN', _, _, _, _} ->
            collect_poems(N-1, Poems);
        {poem, Poem} ->
            collect_poems(N, [Poem|Poems])
    end.
```

Эта функция запускается процессом main и, подобно *get_poetry*, она рекурсивно вызывает саму себя. Она также блокируется. Оператор *receive* говорит процессу ждать прибытия сообщения, которое соответсвует одному из заданных шаблонов, затем извлечь сообщение из его слота (mailbox).

Функция *collect_poems* ожидает двух видов сообщений: поэм и уведомлений типа «DOWN». Уведомление типа «DOWN» - сообщение, посылаемое процессу main, когда один из процессов *get_poetry* по какой-то причине обрушился (это часть мониторига *spawn_monitor*). Подсчитывая ”DOWN” сообщения, мы узнаем, когда вся поэзия завершилась. Шаблон - сообщение от одного из *get_poetry* процессов, содержащих одну завершенную поэму.

Хорошо, давайте запустим Erlang клиент. Сначала запустим три медленных поэтических сервера:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30
```

Теперь вы можете запустить Erlang клиент, который имеет похожий синтаксис командной строки, что и Python клиенты. Если вы используете Linux или другую UNIX-подобную систему, то вам следует запустить клиент напрямую (предполагается, что вы установили Erlang и он доступен по вашей переменной окружения *PATH*). В Windows вам нужно будет запустить программу escript с путем к Erlang клиенту в качестве первого аргумента (и с остальными аргументами для самого Erlang клиента).

```
./erlang-client-1/get-poetry 10001 10002 10003
```

После запуска вы увидите примерно следующее:

```
Task 3: got 30 bytes of poetry from 127:0:0:1:10003
Task 2: got 10 bytes of poetry from 127:0:0:1:10002
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...
```

Это похоже на вывод одного из наших ранних Python клиентов, где вы печатаем сообщение для каждого полученного небольшого куска поэзии. Когда все поэмы скачаются, клиент напишет полный текст каждой из поэм. Заметьте, что клиент переключается между всеми серверами взависимости от того, какой сервер отправил кусок поэзии.

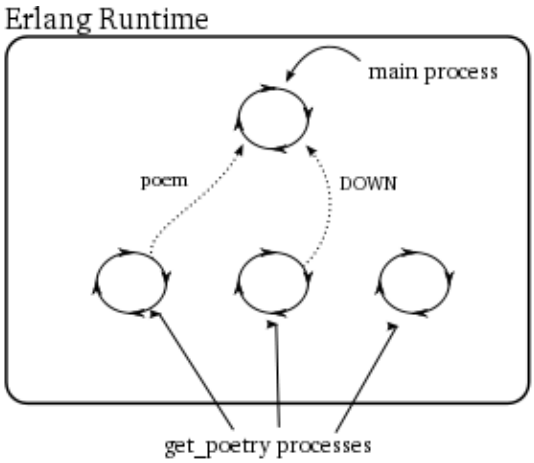

Рис. 45: Поэтический клиент на Erlang

На рисунке 45 изображено три процесса *get_poetry* (один на каждый сервер) и один main процесс. Вы также можете увидеть сообщения, идущие от поэтических процессов к main процессу.

Так что же случается, если один из этих серверов не доступен? Давайте попробуем следующее:

```
./erlang-client-1/get-poetry 10001 10005
```

Команда выше содержит один действующий порт (предполагается, что оставили все ранее запущенные поэтические серверы), и один - недействующий (предполагается, что вы не запустили никакого сервера на порту 10005). И мы получаем некоторый вывод подобный приведенному ниже:

```
Task 1: got 10 bytes of poetry from 127:0:0:1:10001

=ERROR REPORT==== 25-Sep-2010::21:02:10 ===
Error in process <0.33.0> with exit value: {{badmatch,{error,econnrefused}},[{erl_eval,exp

Task 1: got 10 bytes of poetry from 127:0:0:1:10001
Task 1: got 10 bytes of poetry from 127:0:0:1:10001
...
```

Под конец клиент завершает скачивание поэмы из действующего сервера, выводит поэму и выходит. Так как же функция main узнает, что оба процесса завершились? Сообщение об ошибке выше - подсказка. Ошибка происходит, когда *get_poetry* пытается присоединиться к серверу и получает ошибку отказа в соединении («connection refused») вместо ожидаемого значения (ok, Socket). Результирующее исключение называется ошибка сопоставления, поскольку операторы «присваивания» в Erlang являются в действительности операторами сопоставления шаблону.

Необработанное исключение в Erlang процессе вызывает «разрушение» процесса, что означает, что процесс останавливает выполнение и все его ресурсы утилизируются сборщиком мусора. Но процесс main, который контролирует все *get_poetry* процессы, будет получать все DOWN-сообщения, когда любые из этих процессов завершат выполнение по какой-то причине. Таким образом, наш клиент завершается, а не выполняется бесконечно.

20.5. Обсуждение

Давайте выявим некоторые параллели между Twisted и Erlang клиентами:

1. Оба клиента соединяются (или пытаются соединиться) со всеми поэтическими серверами одновременно.

2. Оба клиента получают данные из серверов как только они приходят, независимо от того, какие сервера доставили данные.

3. Оба клиента обрабатывают поэзию маленькими порциями, поэтому должны сохранять порцию полученных поэм.

4. Оба клиента создают "объект" (либо Python объект, либо Erlang процесс) для работы с одним определенным сервером.

5. Оба клиента должны аккуратно определять, когда вся поэзия скачалась, независимо от того успешно или неуспешно произошло определенное скачивание.

И наконец, функции main в обоих клиентах асинхронно получают поэмы и уведомления о завершении задачи. В Twisted клиенте эта информация доставляется через Deferred, в то время как Erlang получает межпроцессорные сообщения (inter-process messages).

Заметьте как похожи оба клиента, в своих общих стратегиях и структуре их кода. Механизмы немного отличаются: с объектами, deferred'ми, callback'ми с одной стороны; процессами и сообщениями с другой стороны. Но высокоуровневые модели обоих клиентов практически одинаковые, и достаточно просто понять один, зная другой.

Даже шаблон проектирования reactor появляется в миниатюре в Erlang клиенте. Каждый Erlang процесс в нашем поэтическом клиенте превращается в рекурсивный цикл, который:

1. Ожидает что что-то произойдет (пришел кусок поэзии, поэма доставлена, другой процесс завершился).

2. Производит определенное действие.

Вы можете думать об Erlang программе как о большой коллекции маленьких реакторов, каждый из которых выполняется в цикле и отправляет сообщение другому маленькому реактору (который обработает это сообщение как еще одно событие).

И если вы копнете глубже в Erlang, вы обнаружите обратные вызовы. Erlang процесс *gen_server* - это общий цикл реактора, экземпляр которого вы создаете, предоставляя фиксированное множество callback-функций, а также это шаблон, повторямый везде в Erlang системе.

Поэтому, изучив Twisted, вы даже можете решить попробовать Erlang и, возможно, обнаружите близкую к вам среду.

20.6. Для дальнейшего ознакомления

В этой главе мы сфокусировались на схожих стороных между Twisted и Erlang, но есть много отличий. Одно особенное уникальное свойство Erlang - это его метод управления ошибками. Большие Erlang программы структурированы как дерево процессов, с «супервизорами» («supervisors») в ветвях и «рабочими» («workers») в листьях. Если рабочий процесс рушится, процесс-супервизор сообщит об этом и сделать некоторые дейтсвия (обычно - перезапустит рухнувший рабочий процесс).

Если вас заинтересовало изучениие Erlang'а, то вы можете почитать приведенные далее книги по Erlang'у, которые были недавно опубликованы или которые скоро будут опубликованы:

- Programming Erlang[1]. Книга написана одним из создателей Erlang'а. Прекрасное введение в язык.
- Erlang Programming[2]. Это дополнение к книге Армстронга и содержит больше деталей по нескольким ключевым темам.
- Erlang and OTP in Action[3]. Эта книга, возможно еще не опубликована. В ней, в отличии от двух предыдущих, рассматривается OTP: Erlang система для построения больших приложений.

На этом мы завершаем рассмотрение Erlang'а. В следующей главе мы посмотрим на Haskell, еще один функциональный язык, очень отличающийся от Python'а и Erlang'а. Тем не менее, мы попытаемся найти что-то общее.

20.7. Упражнения для особо мотивированных

1. Просмотрите Erlang и Python клиенты и идентифицируйте, где они похожи, и где отличаются. Как каждый из них обрабатывает ошибки (например, ошибкой соединения с поэтическим сервером)?
2. Упроситите Erlang клиент так, чтобы он больше не печатал каждый полученный кусок поэзии (также вам не нужно хранить номера задач).
3. Модифицируйте Erlang клиент, чтобы замерить время, которое он тратит на скачивание каждой поэмы.
4. Модифицируйте Erlang клиент, чтобы он выводил поэмы в том же порядке, в котором они были даны в командной строке.
5. Модифицируйте Erlang клиент, чтобы он вывол более читабельные сообщения с ошибкой, когда мы не можем соединиться с поэтическим сервером.
6. Напишите Erlang версии поэтических серверов, которые мы сделали с помощью Twisted.

[1] http://www.amazon.com/exec/obidos/ASIN/193435600X/krondonet-20
[2] http://www.amazon.com/exec/obidos/ASIN/0596518188/krondonet-20
[3] http://www.amazon.com/exec/obidos/ASIN/1933988789/krondonet-20

21. Twisted и Haskell

21.1. Введение

В предыдущей главе мы сравнивали Twisted с Erlang'ом, уделяя внимание некоторым идеям, являющимися для них общими. В результате все оказалось достаточно простым, поскольку асинхронный ввод-вывод и реативное программирование - ключевые компоненты среды и модели процесса Erlang.

Сегодня мы собираемся побродить вдалеке и посмотреть на Haskell[1], еще один функциональный язык, который отличается от Erlang'а (и, конечно, от Python'а). Здесь не будет так много сходств, но мы, тем не менее, найдем спрятый под оболочкой асинхронный ввод-вывод.

21.2. Функциональность с заглавной Ф

Хотя Erlang также является функциональным языком, он в основном сфокусирован на надежной модели согласованности. Haskell наоборот - функциональный во всех отношениях, что делает его неотъемлемой частью концепции теории категорий[2], подобной функторам[3] и монадам[4].

Не беспокойтесь, мы не собираемся сейчас в это погружаться. Вместо этого мы сфокусируемся на одной наиболее традиционной функциональной особенности: ленивости. Подобно большинству функциональных языков (но в отличии от Erlang'а), Haskell поддерживает ленивые вычисления[5]. В языке ленивых вычислений текст программы не описывает то как что-то вычислять, а больше - что вычислять. Детали действительной обработки вычислений обычно скрываются за компилятором и исполняющей системой.

И, нужно отметить, что по мере обработки ленивых вычислений, исполняющая система может вычислять выражения только частично (лениво), а не полностью. В целом, исполняющая система обрабатывает так много от выражения, сколько нужно для того, чтобы сдвинуться в текущих вычислениях.

Далее простой оператор Haskell'а, применяющий функцию *head*, которая получает первый элемент из списка *[1,2,3]* (Haskell и Python имеют общий синтаксис для списка):

```
head [1,2,3]
```

Если вы установили среду исполнения GHC Haskell, вы можете попробовать это сами:

```
[~] ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  : ? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> head [1,2,3]
1
Prelude>
```

Результат, как и ожидалось, - число 1.

[1] http://haskell.org/
[2] http://en.wikipedia.org/wiki/Category_theory
[3] http://en.wikipedia.org/wiki/Functor
[4] http://en.wikipedia.org/wiki/Monad_%28category_theory%29
[5] http://en.wikipedia.org/wiki/Lazy_evaluation

Синтаксис списка в Haskell имеет удобное свойство определять список через свои первые несколько аргументов. Например, список [2,4 ..] - последовательность четных чисел, начинающихся с 2. Где же конец? А его и нет. В Haskell список [2,4 ..] и другие подобные представляют (концептуально) бесконечные списки.

```
Prelude> [2,4 ..]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,
32,34,36,38,40,42,44,46,48,50,52,54,56,58,
60,62,64,66,68,70,72,74,76,78,80,82,84,86,
88,90,92,94,96,98,100,102,104,106,108,110,
...
```

Вы должны нажать Ctrl-C для того, чтобы остановить вычисление, так как оно само не завершится. Ввиду ленивости вычислений, то можно без проблем использовать эти бесконечные циклы в Haskell:

```
Prelude> head [2,4 ..]
2
Prelude> head (tail [2,4 ..])
4
Prelude> head (tail (tail [2,4 ..]))
6
```

Здесь мы сначала берем соответственно первый, второй и третий элементы бесконечного списка без бесконечного зацикливания. В этом и есть сущность ленивых вычислений. Вместо того, чтобы сначада вычислить весь список (который может привести к бесконечному циклу), и затем дать этот список функции head, исполняющая система Haskell только конструирует столько элементов в списке, сколько нужно для того, чтобы head завершила свою работу. Остальные элементы в списке никогда не конструируются, поскольку вычисления не нужно продолжать.

Когда мы вводим в игру функцию tail, то Haskell вынужден продолжить создание списка, но снова только до того момента, когда достаточно для вычисления следующего шага. И как только вычисления сделаны, (незаконченный) список может быть сброшен.

Далее некоторый код на Haskell, который частично отбрасывает три различных бесконечных списка:

```
Prelude> let x = [1..]
Prelude> let y = [2,4 ..]
Prelude> let z = [3,6 ..]
Prelude> head (tail (tail (zip3 x y z)))
(3,6,9)
```

Здесь мы соединяем все списки, затем берем голову хвоста от хвоста. И снова, Haskell не имеет с этим проблем и только конструирует столько элементов в списке, сколько нужно, чтобы завершить выполнение вашего кода. Мы можем визуализировать исполняющую среду Haskell'а «потребляющую» эти бесконечные списки на рисунке 46:

Хотя мы изобразили исполняющую среду Haskell как простой цикл, он мог бы быть реализован несколькими потоками (и, вероятно, это так и есть, если вы используете GHC версию Haskell). Но основная мысль, в том, что рисунок выглядит как реакторный цикл, потребляющий данных по мере их поступления на сетевые сокеты.

Выможете думать об асинхронном вводе-выводе и реакторном шаблоне как об очень ограниченной форме ленивых вычислений. Лозунг асинхронного программирования: "Обработать столько данных, сколько имеется". А лозунг ленивых вычислений: "Обработать

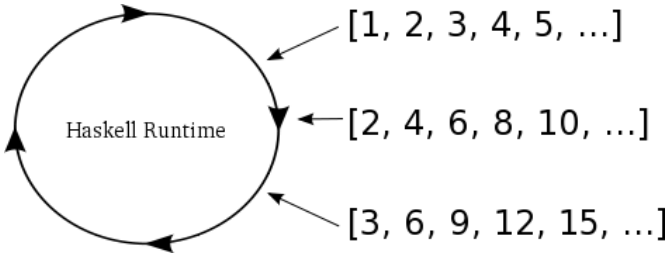

Рис. 46: Haskell, потребляющий некоторые бесконечные списки

столько данных, сколько нужно". Более того, язык ленивых вычислений применяет этот лозунг практически везде, не ограничиваясь только вводом-выводом.

Но основная мысль в том, что язык ленивых вычислений позволяет несложно реализовывать асинхронный ввод-вывод.

Компилятор и исполняющая среда уже спроектированы так, чтобы обрабатывать структуры данных бит за битом, что равносильно ленивой обработке приходящих кусков асинхронного потока.

И таким образом исполняющая среда Haskell, подобно исполняющей среде Erlang, просто включает асинхронный ввод-вывод как часть своих сокетных абстракций. И мы можем продемонстрировать это, реализуя поэтический клиент на Haskell.

21.3. Haskell поэзия

Наш первый поэтический клиент на Haskell'е расположен в haskell-client-1/get-poetry.hs. Также как и с Erlang, мы собираемся сразу же перейти к окончательной версии клиента, и затем предложить ссылки для дополнительного чтения, если вы хотите изучить больше.

Haskell также поддерживает легковесные потоки и процессы, хотя они не являются центральным местом в Haskell, в отличие от Erlang, и наши Haskell клиенты создают один процесс для каждой поэмы, которую мы хотим выкачать. Ключевая функция здесь - это runTask, которая соединяется с сокетом и запускает функцию getPoetry в легковесном потоке.

В коде вы обнаружите много объявлений типов. Haskell, в отличие от Python'а или Erlang'а, ститически типизированный. Мы не объявляем тип для каждой переменной, потому что Haskell автоматически определяет типы для неявно определенных переменных (или сообщает ошибку, если он не может определить). Ряд функций включает тип IO (технически это монада), поскольку Haskell требует от нас четко разделять код со сторонними эффектами (например, код, выполняющий ввод-вывод) от чистых функций.

Функция *getPoetry* включает следующую строчку:

```
poem <- hGetContents h
```

В этой строке происходит чтение всей поэмы из handle'а (например, TCP сокета) за раз. Но Haskell, как обычно, ленивый. И исполняющая среда Haskell'а включает один или более работающих потоков, которые выполняют асинхронный ввод-вывод в select-цикле, таким образом сохраняя возможность ленивых вычислений на потоках ввода-вывода.

Только для иллюстрации того, что асинхронный ввод-вывод в действительности работает, мы включили функцию "обратного вызова" gotLine, которая выводит некоторую информацию по каждой строке в поэме. Но в действительности это вовсе не callback-функция, и программа использовала бы асинхронный ввод-вывод в любом случае. Даже

вызов gotLine отражающщий мышление императивного языка, неуместен в Haskell программе. Но это не важно, мы почистим это немного, давайте запустим Haskell клиент. Запустите медленные серверы:

```
python blocking-server/slowpoetry.py --port 10001 poetry/fascination.txt
python blocking-server/slowpoetry.py --port 10002 poetry/science.txt
python blocking-server/slowpoetry.py --port 10003 poetry/ecstasy.txt --num-bytes 30
```

Теперь скомпилируйте Haskell клиент:

```
cd haskell-client-1/
ghc --make get-poetry.hs
```

Появится исполняемый файл с названием get-poetry. И наконец, запустите клиент для наших серверов:

```
./get-poetry 10001 10002 1000
```

И вы увидите следующий вывод:

```
Task 3: got 12 bytes of poetry from localhost:10003
Task 3: got 1 bytes of poetry from localhost:10003
Task 3: got 30 bytes of poetry from localhost:10003
Task 2: got 20 bytes of poetry from localhost:10002
Task 3: got 44 bytes of poetry from localhost:10003
Task 2: got 1 bytes of poetry from localhost:10002
Task 3: got 29 bytes of poetry from localhost:10003
Task 1: got 36 bytes of poetry from localhost:10001
Task 1: got 1 bytes of poetry from localhost:10001
...
```

Вывод будет немного отличаться от выводов предыдущих наших клиентов, поскольку в предыдущих случаях мы печатали полностью сформированную строку поэмы. Сейчас на печать выводится строка с информацией для каждого куска данных. Но, как вы видите, клиент обрабатывает данные со всех серверов одновременно, а не последовательно. Вы также заметите, что клиент печатает первую поэму сразу же после того, как он ее скачал, не ожидая других, которые продолжают в том же темпе.

Хорошо, давайте очистим наш клиент от оставшихся кусков императивного хлама и представим версию, которая только скачивает поэзию без возни с номерами задач. Вы можете исходный код в haskell-client-2/get-poetry.hs. Заметьте, что он намного короче и для каждого сервера просто соединяется с сокетом, скачивает все данные и отправляет их обратно.

Хорошо, давайте скомпилируем новый клиент:

```
cd haskell-client-2/
ghc --make get-poetry.hs
```

И запустим для того же множества поэтических серверов:

```
./get-poetry 10001 10002 10003
```

В конечном итоге, вы увидите текст каждой поэмы на экране.

Из вывода серверов можно понять, что сервера отправляют данные одновременно. Более того, клиент печатает каждую строку первой поэмы как можно скорее, не ожидая пока будет скачана вся поэма, даже того, когда он работает над двумя оставшимися поэмами. Затем клиент быстро печатает вторую поэму, которая аккумулировалась все это время.

И все это происходит без нашего вмешательства. Здесь нету ни обратных вызовов, ни сообщений передаваемых туда и обратно, только сжатое описание того, что мы хотим, чтобы делала программа, и совсем немного о способе как следует это делать. Остальное берет на себя компилятор и исполняющая среда Haskell.

21.4. Обсуждение и дальнейшее чтение

В движении Twisted-Erlang-Haskell мы можем увидеть движение от внешней реализации к более внутренней идей асинхронного программирования. В Twisted асинхронное программирование - центральная мотивирующая идея существования Twisted. И реализация Twisted как среды, отделенной от Python'а (и отсутствием в Python'е встроенных абстракций, подобных легковесным потокам), сохраняет асинхронную модель в качестве центральной при программировании с использованием Twisted.

В Erlang асинхронность все еще очень видна программисту, но детали являются частью языка и исполняющей системы, делающей доступной асбстракцию, в которой синхронные процессы обмениваются асинхронными сообщениями.

И наконец, в Haskell асинхронный ввод-вывод - одна из техника внутри исполняющей среды, по большому счету невидимая программисту, для обеспечения ленивых вычислений, которые являются одной из центральных идей Haskell'а.

Мы не имеем никакого глубокого видения этой ситуации, мы просто указали на некоторые интересные места проявления асинхронной модели и на различные способы, которыми эта модель может быть выражена.

Если что-то из вышесказанного вызвало у вас особый интерес, то рекомендуется прочитать Real World Haskell[1] для продолжения изучения. Книга - это модель того, каким должно быть хорошее введение в язык. Также можно порекомендовать книгу «Learn You a Haskell»[2].

На этом мы заканчиваем тур по асинхронным системам вне Twisted и предпоследнюю главу. Следующая глава будет завершающая, и в ней будет рассказано про дальнейшие варианты изучения Twisted.

21.5. Упражнения для поразительно мотивированных

1. Сравните Twisted, Erlang и Haskell клиенты друг с другом.

2. Модифицируйте Haskell клиенты так, чтобы они обрабатывали ошибки соединения с поэтическим сервером, при этом они скачивали бы все поэмы, которые могли, и выводили разумные сообщения об ошибках для поэм, которые они не могут скачать.

3. Напишите Haskell версии поэтических серверов, которые мы сделали с помощью Twisted.

[1] http://www.amazon.com/exec/obidos/ASIN/0596514980/krondonet-20

[2] http://learnyouahaskell.com/

22. Конец

22.1. Все сделано

На создание всех глав было потрачено много времени, но надеюсь, что вам понравилось.

Далее несколько предложений по поводу дальнейшего изучения Twisted.

22.2. Дальнейшее чтение

Во-первых, рекомендуется почитать документацию на сайте[1].

Если вы хотите использовать Twisted для web программирования, то есть хорошая книга Jean-Paul Calderone «Twisted Web in 60 Seconds»[2]. Хотя, думается, что на прочтение вам потребуется немного больше времени.

Также есть Twisted Book[3].

Но важнее, чем все остальное, - это сами исходники Twisted. Поскольку код написан людьми, которые очень хорошо знают Twisted, и это отличный источник примеров создания программ «способом Twisted».

22.3. Упражнения

1. Портируйте какую-нибудь вашу синхронную программу на Twisted.
2. Напиши новую Twisted программу с нуля.
3. Выберите ошибку из базы данных Twisted багов[4] и исправьте его. Предоставьте patch разработчикам Twisted. Не забудьте прочитать про то, как можно посодействовать проекту[5].

22.4. Действительно конец

Удачного программирования!

[1] http://twistedmatrix.com/trac/wiki/Documentation
[2] http://jcalderone.livejournal.com/50562.html
[3] http://www.amazon.com/gp/product/0596100329?ie=UTF8&tag=jpcalsjou-20&linkCode=as2&camp=1789&creative
[4] http://twistedmatrix.com/trac/report
[5] http://twistedmatrix.com/trac/wiki/ContributingToTwistedLabs

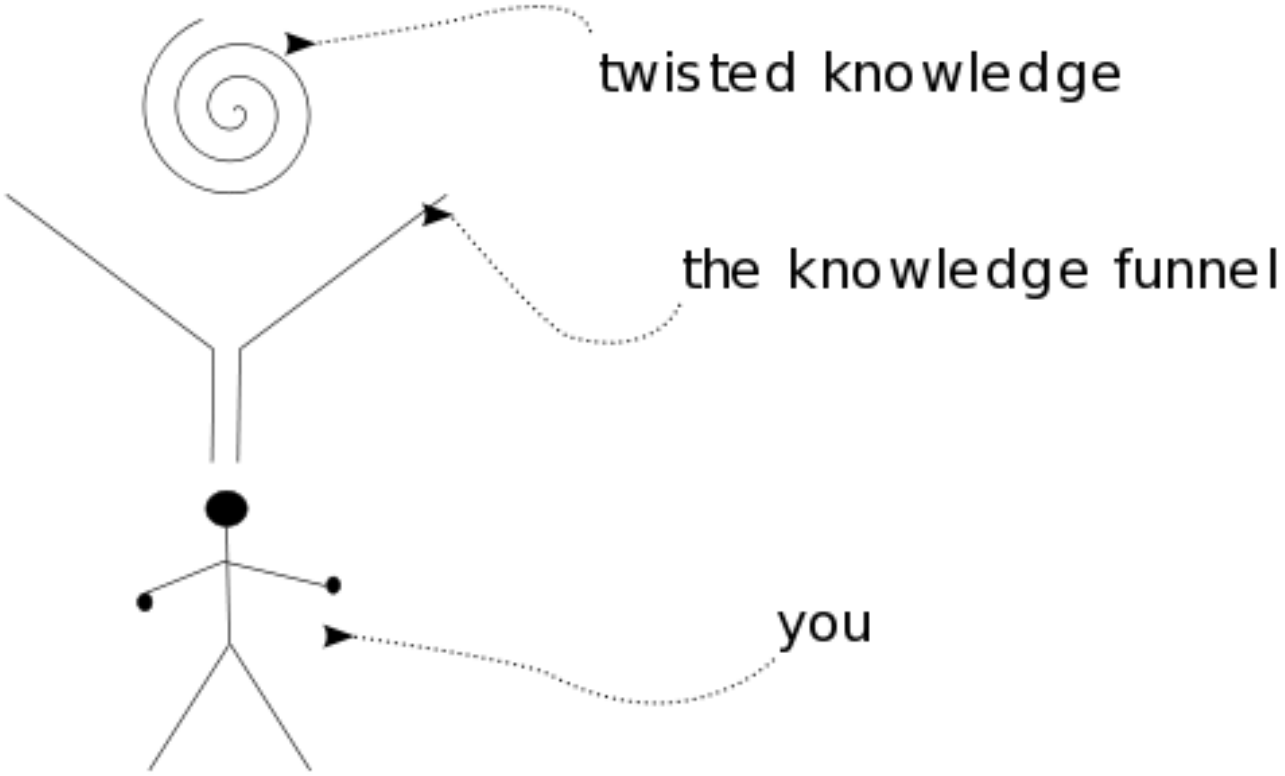

Рис. 47: Конец