Отслеживание путей

Одним из наиболее полезных компонентов нового пакета `java.nio.file` является интерфейс `WatchService`, который позволяет следить за изменениями в файлах и директориях по заданному пути `Path`. Мы можем получать события, связанные с созданием, изменением и удалением элементов файловой системы. Следующий фрагмент кода следит за изменениями в каталоге */Users/pat*:

```
Path watchPath = fs.getPath("/Users/pat");
WatchService watchService = fs.newWatchService();
watchPath.register( watchService, ENTRY_CREATE, ENTRY_MODIFY, ENTRY_DELETE
);
while( true )
{
   WatchKey changeKey = watchService.take();
   List<WatchEvent<?>> watchEvents = changeKey.pollEvents();
   for ( WatchEvent<?> watchEvent : watchEvents )
   {
      // Все наши события имеют тип Path:
      WatchEvent<Path> pathEvent = (WatchEvent<Path>)watchEvent;
      Path path = pathEvent.context();
      WatchEvent.Kind<Path> eventKind = pathEvent.kind();
      System.out.println( eventKind + " for path: " + path );
   }
   changeKey.reset(); // Важно!
}
```

С помощью метода `newWatchService()` из объекта `FileSystem` мы создаем сервис `WatchService`. Далее мы можем зарегистрировать для этого сервиса реализацию интерфейса `Watchable` (пока что в этом качестве выступает объект `Path`), события которого будут отслеживаться. Как видно, в API-интерфейсе все устроено наоборот: метод `register()` вызывается именно из объекта `Watchable`, а аргументами для него служат сам сервис и список параметров переменной длины, которые представляют интересующие нас типы событий (`ENTRY_CREATE`, `ENTRY_MODIFY` или `ENTRY_DELETE`). Есть еще один дополнительный тип, `OVERFLOW`; события, которые можно получать с его помощью, сигнализируют о том, что механизм мониторинга файловой системы слишком медленный для того, чтобы обработать все изменения, поэтому часть событий может до нас не дойти.

Закончив с приготовлениями, мы можем начать запрашивать информацию об изменениях с помощью метода `take()`, который возвращает объект `WatchKey`. Метод `take()` блокирует поток выполнения до тех пор, пока не появится событие; существует также неблокирующая версия этой операции под названием `poll()`. Мы можем извлечь события из объекта `WatchKey`, воспользовавшись методом `pollEvents()`. Здесь мы снова сталкиваемся с некоторой неочевидностью API-интерфейса, так как `WatchEvent` — это обобщенный тип, параметризированный для работы с объектом `Watchable`. В нашем случае в качестве событий всегда выступают объекты `Path`, которые при необходимости можно привести к нужному нам типу. Вид события (создание, изменение, удаление) можно узнать при помощи метода `WatchEventkind()`, а метод `context()` возвращает измененный путь. В конце важно не забыть вызвать метод `reset()` из объекта `WatchKey`, чтобы очистить список событий и иметь возможность дальше получать информацию об изменениях.

Производительность сервиса `WatchService` во многом зависит от его реализации. Часто механизм мониторинга файловой системы встроен в ОС, и это позволяет получать события практически мгновенно. Но нередко платформа Java полагается на собственную универсальную реализацию, основанную на фоновых потоках, которая очень долго обнаруживает изменения. Например, на момент написания этой книги версия платформы Java 7 для OS X не использует преимущества системного мониторинга файлов, получая данные из медленного универсального сервиса.

Сериализация

С помощью потока `DataOutputStream` вы можете написать приложение, которое последовательно преобразует содержимое ваших объектов в значения простых типов. Однако платформа Java предоставляет еще более мощный механизм, который способен выполнить за вас всю работу. Речь идет о *сериализации* — автоматическом способе сохранения и загрузки состояния объекта. Это очень обширная тема, которую мы не в состоянии полностью охватить в рамках данной книги. К примеру, вы не найдете здесь некоторые интересные приемы вроде ведения версий, равно как и сведения об управлении процессом сериализации.

В сущности, экземпляр любого класса, который реализует интерфейс `Serializable`, может быть сохранен в поток и восстановлен обратно. Для сериализации простых типов и объектов используются разновидности