Руководство по WatchService в Java NIO2

Непонятная «фигня». Для своей работы должна постоянно занимать один поток процессора, т.к. нет памяти на последнее состояние!!!

**1. Обзор**

В этой статье мы собираемся изучить интерфейс *WatchService*API-интерфейсов файловой системы Java NIO.2. Это одна из менее известных функций новых API ввода-вывода, которые были представлены в Java 7 вместе с интерфейсом *FileVisitor*.

Чтобы использовать интерфейс *WatchService*в своих приложениях, вам необходимо импортировать соответствующие классы:

import java.nio.file.\*\* ;

**2. Зачем использовать *WatchService***

Новые фреймворки, такие как Play, также выполняют горячую перезагрузку кода приложения по умолчанию - всякий раз, когда вы выполняете редактирование из любого редактора. Эти приложения используют функцию под названием ***file change messages***, которая доступна во всех файловых системах.

По сути, **мы можем написать код для опроса файловой системы на предмет изменений в конкретных файлах и каталогах**. Однако это решение не масштабируемо, особенно если файлы и каталоги достигают сотен и тысяч.

В Java 7 NIO.2 API *WatchService*предоставляет масштабируемое решение для мониторинга каталогов на предмет изменений. Он имеет чистый API и настолько хорошо оптимизирован для производительности, что нам не нужно реализовывать собственное решение.

**3. Как работает *WatchService*?**

Чтобы использовать функции *WatchService*, первым шагом является создание экземпляра *WatchService*с использованием класса *java.nio.file.FileSystems*:

WatchService watchService = FileSystems.getDefault().newWatchService();

Далее нам нужно создать путь к каталогу, который мы хотим отслеживать:

Path path = Paths.get("pathToDir");

После этого шага мы должны зарегистрировать путь с помощью службы часов. На этом этапе нужно понять две важные концепции. Класс *StandardWatchEventKinds*и класс *WatchKey*. Взгляните на следующий регистрационный код, чтобы понять, где каждый упал. Мы последуем этому с объяснениями того же самого:

WatchKey watchKey = path.register(

watchService, StandardWatchEventKinds...);

Здесь обратите внимание только на две важные вещи: во-первых, вызов API регистрации пути принимает экземпляр службы наблюдения в качестве первого параметра, за которым следуют переменные аргументы *StandardWatchEventKinds*. Во-вторых, возвращаемый тип процесса регистрации - это экземпляр *WatchKey*.

**3.1. *StandardWatchEventKinds***

Это класс, экземпляры которого сообщают службе часов о видах событий, которые нужно отслеживать в зарегистрированном каталоге. В настоящее время есть четыре возможных события для просмотра:

* ***StandardWatchEventKinds.ENTRY***CREATE \_\_ - срабатывает при появлении новой записи

сделано в наблюдаемом каталоге. Это может быть связано с созданием нового файла или переименованием существующего файла.

* ***StandardWatchEventKinds.ENTRY***MODIFY \_\_ - срабатывает, когда существующий

запись в наблюдаемом каталоге изменена. Все редактирование файла вызывает это событие. На некоторых платформах даже изменение атрибутов файла вызовет это.

* ***StandardWatchEventKinds.ENTRY***DELETE \_\_ - срабатывает, когда запись

удален, перемещен или переименован в наблюдаемом каталоге.

* ***StandardWatchEventKinds.OVERFLOW***- срабатывает, чтобы указать потерянный или

отброшенные события. Мы не будем заострять на этом внимание

**3.2. *WatchKey***

Этот класс представляет собой регистрацию каталога с помощью службы часов. Его экземпляр возвращается нам службой наблюдения, когда мы регистрируем каталог и когда мы спрашиваем службу наблюдения, произошли ли какие-либо события, для которых мы зарегистрировались.

Сервис Watch не предлагает нам методов обратного вызова, которые вызываются при возникновении события. Мы можем только опросить его несколькими способами, чтобы найти эту информацию.

Мы можем использовать API *poll*:

WatchKey watchKey = watchService.poll();

Этот вызов API возвращается сразу. Он возвращает следующий ключ наблюдения в очереди, любое из событий которого произошло, или ноль, если не было зарегистрировано ни одного события.

Мы также можем использовать перегруженную версию, которая принимает аргумент *timeout*:

WatchKey watchKey = watchService.poll(long timeout, TimeUnit units);

Этот вызов API похож на предыдущий в возвращаемом значении. Однако он блокируется на *timeout*единицы времени, чтобы дать больше времени, в течение которого может произойти событие, вместо того, чтобы сразу возвращать ноль.

Наконец, мы можем использовать API *take*:

WatchKey watchKey = watchService.take();

Этот последний подход просто блокирует, пока не произойдет событие.

Здесь мы должны отметить кое-что очень важное: **когда экземпляр *WatchKey*возвращается API-интерфейсом *poll*или *take*, он не будет регистрировать больше событий, если не был вызван API сброса:**

watchKey.reset();

Это означает, что экземпляр ключа наблюдения удаляется из очереди службы наблюдения каждый раз, когда его возвращает операция опроса. Вызов API *reset*помещает его обратно в очередь для ожидания новых событий.

Наиболее практичное применение службы наблюдателя требует цикла, в рамках которого мы постоянно проверяем изменения в наблюдаемом каталоге и обрабатываем соответствующим образом. Мы можем использовать следующую идиому для реализации этого:

WatchKey key;

while ((key = watchService.take()) != null) {

for (WatchEvent<?> event : key.pollEvents()) {

//process

}

key.reset();

}

Мы создаем ключ наблюдения для хранения возвращаемого значения операции poll.

Цикл while будет блокироваться до тех пор, пока условный оператор не вернется либо с ключом наблюдения, либо с нулем.

Когда мы получаем ключ наблюдения, тогда цикл while выполняет код внутри него. Мы используем API *WatchKey.pollEvents*, чтобы вернуть список событий, которые произошли. Затем мы используем цикл *for each*, чтобы обработать их один за другим.

После обработки всех событий мы должны вызвать API *reset*, чтобы снова поставить в очередь ключ наблюдения.

**4. Пример просмотра каталога**

Поскольку мы рассмотрели API-интерфейс *WatchService*в предыдущем подразделе и его внутреннюю работу, а также способы ее использования, теперь мы можем перейти к полному и практическому примеру.

По причинам переносимости мы будем следить за активностью в домашнем каталоге пользователя, который должен быть доступен во всех современных операционных системах.

Код содержит всего несколько строк кода, поэтому мы просто оставим его в методе main:

public class DirectoryWatcherExample {

public static void main(String[]args) {

WatchService watchService

= FileSystems.getDefault().newWatchService();

Path path = Paths.get(System.getProperty("user.home"));

path.register(

watchService,

StandardWatchEventKinds.ENTRY\_\_CREATE,

StandardWatchEventKinds.ENTRY\_\_DELETE,

StandardWatchEventKinds.ENTRY\_\_MODIFY);

WatchKey key;

while ((key = watchService.take()) != null) {

for (WatchEvent<?> event : key.pollEvents()) {

System.out.println(

"Event kind:" + event.kind()

+ ". File affected: " + event.context() + ".");

}

key.reset();

}

}

}

И это все, что нам действительно нужно сделать. Теперь вы можете запустить класс, чтобы начать просмотр каталога.

Когда вы переходите к домашнему каталогу пользователя и выполняете какие-либо действия с файлами, такие как создание файла или каталога, изменение содержимого файла или даже удаление файла, все это будет зарегистрировано на консоли.

Например, если вы идете к пользователю домой, щелкните правой кнопкой мыши в пространстве, выберите ` \_new → file` , чтобы создать новый файл, а затем назовите его testFile\_ .

Затем вы добавляете контент и сохраняете. Вывод на консоли будет выглядеть так:

Event kind:ENTRY\_\_CREATE. File affected: New Text Document.txt.

Event kind:ENTRY\_\_DELETE. File affected: New Text Document.txt.

Event kind:ENTRY\_\_CREATE. File affected: testFile.txt.

Event kind:ENTRY\_\_MODIFY. File affected: testFile.txt.

Event kind:ENTRY\_\_MODIFY. File affected: testFile.txt.

Не стесняйтесь редактировать путь, чтобы указать на любой каталог, который вы хотите посмотреть.

**Еще один пример использования**

Отслеживание изменений в каталоге

Предположим, что необходимо написать некое приложение, работающее с файлами (IDE, файловый менеджер или какой-либо редактор). Допустим какой-либо файл, важный для приложения был либо создан, либо изменён, либо удалён из вне и надо сообщить об этом пользователю.

Java 7 предоставляет сервис для слежения за каталогами. Вы можете зарегистрировать в нём каталог, чтобы получать уведомления о любом изменении в каталоге (создание, изменение и удаление файла).

Рассмотрим API этого сервиса на примере. Для этого примера используйте каталог в котором при выполнении программы в бесконечном цикле Вы будете создавать, изменять и удалять файлы — действия должны выводится на консоли. Код ниже:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**public** **class** Test14 {   
    **public** **static** **void** main(String[] args) {   
        Path path = Paths.get("/Введите сюда путь к каталогу, изменения в котором будут отслеживаться");   
        WatchService watchService = **null**;   
        **try** {   
            watchService = path.getFileSystem().newWatchService();   
            path.register(watchService,   
                    StandardWatchEventKinds.ENTRY\_CREATE,   
                    StandardWatchEventKinds.ENTRY\_DELETE,   
                    StandardWatchEventKinds.ENTRY\_MODIFY);   
        } **catch** (IOException e1) {   
            e1.printStackTrace();   
        }   
        // Бесконечный цикл   
        **for** (;;) {   
            WatchKey key = **null**;   
            **try** {   
                key = watchService.take();   
            } **catch** (InterruptedException e) {   
                e.printStackTrace();   
            }   
            // Итерации для каждого события   
            **for** (WatchEvent event : key.pollEvents()) {   
                **switch** (event.kind().name()) {   
                **case** "OVERFLOW":   
                    System.out.println("We lost some events");   
                    **break**;   
                **case** "ENTRY\_CREATE":   
                    System.out.println("File " + event.context()   
                            + " is created!");   
                    **break**;   
                **case** "ENTRY\_MODIFY":   
                    System.out.println("File " + event.context()   
                            + " is modified!");   
                    **break**;   
                **case** "ENTRY\_DELETE":   
                    System.out.println("File " + event.context()   
                            + " is deleted!");   
                    **break**;   
                }   
            }   
            // Сброс ключа важен для получения последующих уведомлений   
            key.reset();   
        }   
    }   
}

*Пояснения к коду:*

* Получение экземпляра WatchService осуществляется через вызов метода newWatchService() из класса FileSystem, который в свою очередь был получен из объекта класса Path через вызов метода getFileSystem(). Но его (экземпляр FileSystem) можно получить и через вызов метода getDefault() из класса FileSystems, т.е. можно заменить watchService = path.getFileSystem().newWatchService();   
  на   
  watchService = FileSystems.getDefault().newWatchService();
* Далее происходит регистрация каталога на данном сервисе. Из объекта Path можно вызвать два метода register(), которые различаются принимаемыми параметрами. В нашем случае — это метод принимающий объект сервиса и переменное число параметров, определяющих отслеживаемые события.

path.register(watchService,   
StandardWatchEventKinds.ENTRY\_CREATE,   
StandardWatchEventKinds.ENTRY\_DELETE,   
StandardWatchEventKinds.ENTRY\_MODIFY);

* OVERFLOW указывает, что несколько уведомлений о событиях были отброшены или пропущены. ENTRY\_CREATE, ENTRY\_DELETE, ENTRY\_MODIFY соответствуют созданию, редактированию и удалению.
* В цикле происходит ожидание события. Здесь можно использовать три метода для уведомления:
  + Метод poll() возвращает ключи очереди, если они есть, в противном случае сразу завершается.
  + Метод poll(long, TimeUnit) возвращает ключи очереди, если они есть, в противном случае ждёт указанное число единиц времени.
  + Метод take() возвращает ключ очереди, если он доступен, иначе ждёт, пока он не будет доступен.
* Получить вид события можно с помощью метода kind(), а имя файла, для которого произошло событие - с помощью метода context().

Основное различие между poll() и take() в том, что poll() это неблокирующий вызов, а take() — блокирующий.

Когда ключ возвращается, то одно или более событий могут быть помещены в очередь, именно поэтому используется ещё один цикл (для перебора всех событий).