010.NIO2.0即AIO

JDK7在java.nio这个工具包里加入了很多新的元素

我们把这些改变叫NIO2.0(即AIO，Asynchronous IO：异步非阻塞IO)

和NIO相比，同样有Channel和Buffer，但没有Selector。

Java NIO Path

Java Path接口是Java NIO 2.0更新的一部分，同Java NIO一起已经包括在Java6和Java7中。

Java Path接口是在Java7中添加到**java.nio**的。

Path接口位于java.nio.file包中，所以Path接口的完全限定名称为java.nio.file.Path。

Java Path实例表示文件系统中的路径。一个路径可以指向一个文件或一个目录。路径可以是绝对路径，也可以是相对路径。

在许多方面，java.nio.file.Path接口类似于java.io.File类，但是有一些细微的差别。不过，在许多情况下，您可以使用Path接口来替换File类的使用。

创建一个Path实例

为了使用java.nio.file.Path实例必须创建一个Path实例。

您可以使用Paths类(java.nio.file.Paths)中的静态方法来创建路径实例，名为Paths.get()。

下面是一个Java Paths.get()示例:

```
import java.nio.file.Path;
import java.nio.file.Paths;
public class PathExample {
    public static void main(String[] args) {
        Path path = Paths.get("c:\\data\\myfile.txt");
    }
}
```

创建一个绝对路径

创建绝对路径是通过调用Paths.get()工厂方法，给定绝对路径文件作为参数来完成的。

下面是创建一个表示绝对路径的路径实例的例子:

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

上面的路径是一个Windows文件系统路径。

在Unix系统(Linux、MacOS、FreeBSD等)上，上面的绝对路径可能如下:

```
Path path = Paths.get("/home/jakobjenkov/myfile.txt");
```

如果您在Windows机器上使用了这种路径(从/开始的路径)，那么路径将被解释为相对于当前驱动器。

例如,路径

```
/home/jakobjenkov/myfile.txt
```

可以将其解释为位于C盘驱动器上。那么这条路径就会对应这条完整的路径:

```
C:/home/jakobjenkov/myfile.txt
```

创建一个相对路径

一条相对路径的完整路径(绝对路径)是通过将基本路径与相对路径相结合而得到的。

Java NIO `Path`类也可以用于处理相对路径。您可以使用`Paths.get(basePath, relativePath)`方法创建一个相对路径。

下面是Java中的两个相对路径示例:

```
Path projects = Paths.get("d:\\data", "projects");
Path file = Paths.get("d:\\data", "projects\\a-project\\myfile.txt");
```

第一个例子创建了一个Java `Path`的实例，指向路径(目录):`d:\data\projects`，

第二个例子创建了一个`Path`的实例，指向路径(文件):`d:\data\projects\a-project\myfile.txt`

当在工作中使用相对路径时，你可以在你的路径字符串中使用两个特殊代码，它们是：

. 当前目录

```
Path currentDir = Paths.get(".");  //执行上述代码的应用程序的目录
System.out.println(currentDir.toAbsolutePath());
```

在路径字符串的中间使用.，表示同样的目录作为路径指向那个点。

.. 上一层目录

```
Path parentDir = Paths.get("..");  //执行上述代码的应用程序的父目录
```

Java NIO Files

Java NIO `Files`类(`java.nio.file.Files`)提供了几种操作文件系统中的文件的方法。

`java.nio.file.Files`类与`java.nio.file.Path`实例一起工作。

Files.exists()

`Files.exists()`方法检查给定的`Path`在文件系统中是否存在。

可以创建在文件系统中不存在的`Path`实例。

这里是一个Java Files.exists()的例子：

```
Path path = Paths.get("data/logging.properties");
boolean pathExists =
        Files.exists(path,
            new LinkOption[]{ LinkOption.NOFOLLOW_LINKS});
```

这个例子首先创建一个Path实例指向一个路径，我们想要检查这个路径是否存在。
然后，这个例子调用Files.exists()方法，然后将Path实例作为第一个参数。

注意Files.exists()方法的第二个参数。这个参数是一个选项数组，它影响Files.exists()如何确定路径是否存在。
在上面的例子中的数组包含LinkOption.NOFOLLOW_LINKS，
这意味着Files.exists()方法不应该在文件系统中跟踪**符号链接**，以确定文件是否存在。

Files.createDirectory()

Files.createDirectory()方法，用于根据Path实例创建一个新目录，下面是一个Files.createDirectory()例子：

```
Path path = Paths.get("data/subdir");
try {
    Path newDir = Files.createDirectory(path);
} catch(FileAlreadyExistsException e){
    // 目录已经存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
}
```

第一行创建表示要创建的目录的Path实例。
在try-catch块中，用路径作为参数调用Files.createDirectory()方法。
如果创建目录成功，将返回一个Path实例，该实例指向新创建的路径。

如果该目录已经存在，则是抛出一个java.nio.file.FileAlreadyExistsException。
如果出现其他错误，可能会抛出IOException。
例如，如果想要的新目录的父目录不存在，则可能会抛出IOException。

Files.copy()

Files.copy()方法从一个路径拷贝一个文件到另外一个目录：

```
Path sourcePath = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");
try {
```

```
    Files.copy(sourcePath, destinationPath);
} catch(FileAlreadyExistsException e) {
    // 目标文件已经存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
}
```

首先，该示例创建一个源和目标Path实例。然后，这个例子调用Files.copy()，将两个Path实例作为参数传递。这可以让源路径引用的文件被复制到目标路径引用的文件中。

如果目标文件已经存在，则抛出一个java.nio.file.FileAlreadyExistsException异常。如果有其他错误，则会抛出一个IOException。例如，如果将该文件复制到不存在的目录，则会抛出IOException。

覆盖已存在的文件

可以强制Files.copy()覆盖现有的文件。

这里有一个示例，演示如何使用Files.copy()覆盖现有文件。

```
Path sourcePath      = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");
try {
    Files.copy(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch(FileAlreadyExistsException e) {
    // 目标文件已存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
}
```

请注意Files.copy()方法的第三个参数。如果目标文件已经存在，这个参数指示copy()方法覆盖现有的文件。

Files.move()

Java NIO Files还包含一个函数，用于将文件从一个路径移动到另一个路径。

移动文件与重命名相同，但是移动文件既可以移动到不同的目录，也可以在相同的操作中更改它的名称。

java.io.File类也可以使用它的renameTo()方法来完成这个操作，但是现在已经在java.nio.file.Files中有了文件移动功能。

```
Path sourcePath = Paths.get("data/logging-copy.properties");
Path destinationPath = Paths.get("data/subdir/logging-moved.properties");
```

```
try {
    Files.move(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    //移动文件失败
    e.printStackTrace();
}
```

Files.delete()

Files.delete()方法可以删除一个文件或者目录。

```
Path path = Paths.get("data/subdir/logging-moved.properties");
try {
    Files.delete(path);
} catch (IOException e) {
    // 删除文件失败
    e.printStackTrace();
}
```

首先，创建指向要删除的文件的Path。然后调用Files.delete()方法。

如果Files.delete()由于某种原因不能删除文件(例如，文件或目录不存在)，会抛出一个IOException。

Files.walkFileTree()

Files.walkFileTree()方法包含递归遍历目录树的功能。

walkFileTree()方法将Path实例和FileVisitor作为参数。

Path实例指向您想要遍历的目录。

FileVisitor在遍历期间被调用。

FileVisitor接口必须自己实现，并将实现的实例传递给walkFileTree()方法。

在目录遍历过程中，FileVisitor实现的每个方法都将被调用。

如果不需要实现所有这些方法，那么可以扩展SimpleFileVisitor类，它包含FileVisitor接口中所有方法的默认实现。

walkFileTree()的例子：

```
Files.walkFileTree(path, new FileVisitor<Path>() {
  @Override
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
    System.out.println("pre visit dir:" + dir);
    return FileVisitResult.CONTINUE;
  }
  @Override
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
```

```
    System.out.println("visit file: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {
    System.out.println("visit file failed: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
    System.out.println("post visit directory: " + dir);
    return FileVisitResult.CONTINUE;
  }
});
```

FileVisitor实现中的每个方法在遍历过程中的不同时间都被调用:
在访问任何目录之前调用preVisitDirectory()方法。
在访问一个目录之后调用postVisitDirectory()方法。

调用visitFile()在文件遍历过程中访问的每一个文件。
它不会访问目录-只会访问文件。
在访问文件失败时调用visitFileFailed()方法。

这四个方法中的每个都返回一个FileVisitResult枚举实例。
FileVisitResult枚举包含以下四个选项:
CONTINUE 继续，意味着文件的执行应该像正常一样继续。
TERMINATE 终止，意味着文件遍历现在应该终止。
SKIP_SIBLING 跳过同级，意味着文件遍历应该继续，但不需要访问该文件或目录的任何同级。
SKIP_SUBTREE 跳过子级，意味着文件遍历应该继续，但是不需要访问这个目录中的子目录。

这个值只有从preVisitDirectory()返回时才是一个函数。如果从任何其他方法返回，

它将被解释为一个CONTINUE继续。

通过返回其中一个值，调用方法可以决定如何继续执行文件。

文件搜索

这里是一个用于扩展SimpleFileVisitor的walkFileTree()，以查找一个名为aaaa.txt的文件：

```
Path rootPath = Paths.get("data");
String fileToFind = File.separator + "aaaa.txt";
try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
{
      String fileString = file.toAbsolutePath().toString();
      //System.out.println("pathString = " + fileString);
      if(fileString.endsWith(fileToFind)){
        System.out.println("file found at path: " + file.toAbsolutePath());
        return FileVisitResult.TERMINATE;
      }
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
    e.printStackTrace();
}
```

递归删除目录

Files.walkFileTree()也可以用来删除包含所有文件和子目录的目录。

Files.delete()方法只会删除一个目录，如果它是空的。

通过遍历所有目录并删除每个目录中的所有文件(在visitFile())中，

然后删除目录本身(在postVisitDirectory()中)，您可以删除包含所有子目录和文件的目录。

```
Path rootPath = Paths.get("data/to-delete");
try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
{
      System.out.println("delete file: " + file.toString());
      Files.delete(file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
      Files.delete(dir);
```

```
            System.out.println("delete dir: " + dir.toString());
            return FileVisitResult.CONTINUE;
        }
    });
} catch(IOException e){
    e.printStackTrace();
}
```

总结：上面的方法只是对常用方法一个小总结，其他还有很多其他方法，可以查看文档。

AIO在channel方面主要引入了三个组件：

AsynchronousFileChannel

AsynchronousSocketChannel （网络编程里我们在玩儿）

AsynchronousServerSocketChannel（网络编程里我们在玩儿）

AsynchronousFileChannel

在Java 7中，`AsynchronousFileChannel`被添加到Java NIO。

`AsynchronousFileChannel`让读取数据，并异步地将数据写入文件成为可能。

创建一个AsynchronousFileChannel

可以通过它的静态方法`open()`创建一个`AsynchronousFileChannel`。

```
Path path = Paths.get("data/test.xml");
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
```

`open()`方法的第一个参数是指向与`AsynchronousFileChannel`相关联的文件的`Path`实例。

第二个参数是一个或多个打开选项，它告诉`AsynchronousFileChannel`在底层文件上执行哪些操作。

在本例中，我们使用了`StandardOpenOption.READ`选项。意味着该文件将以只读的方式被打开。

读取数据

可以通过两种方式从`AsynchronousFileChannel`读取数据。

读取数据的每一种方法都调用`AsynchronousFileChannel`的`read()`方法之一。

通过Future阅读数据

从`AsynchronousFileChannel`读取数据的第一种方法是调用返回`Future`的read()方法。

```
Future<Integer> operation = fileChannel.read(buffer, 0);
```

read()方法的第一个参数，从AsynchronousFileChannel读取的数据放入这个ByteBuffer。
第二个参数是文件中开始读取的字节位置。

read()方法会立即返回，即使读操作还没有完成。
通过调用read()方法返回的Future实例的isDone()方法，您可以检查读取操作是否完成。

```
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;
Future<Integer> operation = fileChannel.read(buffer, position);
while(!operation.isDone());
buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

这个例子创建了一个AsynchronousFileChannel，然后创建一个ByteBuffer，
在调用read()之后，一直循环，直到返回的isDone()方法返回true。
当然，这不是非常有效地使用CPU，但是需要等到读取操作完成之后才会执行。
读取操作完成后，数据读取到ByteBuffer中，然后把缓冲区中的字符串打印到System.out中。

通过一个CompletionHandler读取数据

从AsynchronousFileChannel读取数据的第二种方法
调用read()方法，让CompletionHandler作为第四个参数。

```
fileChannel.read(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("result = " + result);
        attachment.flip();
        byte[] data = new byte[attachment.limit()];
        attachment.get(data);
        System.out.println(new String(data));
        attachment.clear();
    }
    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {

    }
```

```
});
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

一旦读取操作完成，将调用CompletionHandler的completed()方法。

对于completed()方法的参数传递一个整数，它告诉我们读取了多少字节，以及传递给read()方法的“附件”。

“附件”是read()方法的第三个参数。

在本例中，它是ByteBuffer，数据也被读取。您可以自由选择要附加的对象。

如果读取操作失败，则将调用CompletionHandler的failed()方法。

写数据

就像阅读一样，您可以通过两种方式将数据写入一个AsynchronousFileChannel。

写入数据的每一种方法都调用异步文件通道的write()方法之一。

通过Future写数据

AsynchronousFileChannel还允许您异步地写数据。

下面是一个完整的Java AsynchronousFileChannel示例:

```
Path path = Paths.get("data/test-write.txt");
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;
buffer.put("test data".getBytes());
buffer.flip();
Future<Integer> operation = fileChannel.write(buffer, position);
buffer.clear();
while(!operation.isDone());
System.out.println("Write done");
```

首先，AsynchronousFileChannel以写模式打开。

然后创建一个ByteBuffer，并将一些数据写入其中。

然后，ByteBuffer中的数据被写入到文件中。

最后，示例检查返回的Future，以查看写操作完成时的情况。

注意，在此代码生效之前，文件必须已经存在。

如果该文件不存在，那么write()方法将抛出一个java.nio.file.NoSuchFileException。

您可以确保该Path指向的文件具有以下代码：

```
if(!Files.exists(path)){
    Files.createFile(path);
}
```

通过一个CompletionHandler写入数据

您还可以使用一个CompletionHandler将数据写入到AsynchronousFileChannel中，以告诉您何时完成写入，而不是Future。

下面是一个将数据写入到AsynchronousFileChannel的示例，该通道有一个CompletionHandler：

```
Path path = Paths.get("data/test-write.txt");
if(!Files.exists(path)){
    Files.createFile(path);
}
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("bytes written: " + result);
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        exc.printStackTrace();
    }
});
```

当写操作完成时，将会调用CompletionHandler的completed()方法。

如果由于某种原因而写失败，则会调用failed()方法。

注意如何将ByteBuffer用作附件——该对象被传递给CompletionHandler的方法。