http://wiki.jikexueyuan.com/project/java-nio-zh/java-nio-files.html

Java NIO中的Files类（java.nio.file.Files）提供了多种操作文件系统中文件的方法。本节教程将覆盖大部分方法。Files类包含了很多方法，所以如果本文没有提到的你也可以直接查询JavaDoc文档。

java.nio.file.Files类是和java.nio.file.Path相结合使用的，所以在用Files之前确保你已经理解了Path类。

Files.exists()

Files.exits()方法用来检查给定的Path在文件系统中是否存在。 在文件系统中创建一个原本不存在的Path是可行的。例如，你想新建一个目录，那么先创建对应的Path实例，然后创建目录。

由于Path实例可能指向文件系统中的不存在的路径，所以需要用Files.exists()来确认。

下面是一个使用Files.exists()的示例：

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

这个示例中，我们首先创建了一个Path对象，然后利用Files.exists()来检查这个路径是否真实存在。

注意Files.exists()的的第二个参数。他是一个数组，这个参数直接影响到Files.exists()如何确定一个路径是否存在。在本例中，这个数组内包含了LinkOptions.NOFOLLOW_LINKS，表示检测时不包含符号链接文件。

Files.createDirectory()

Files.createDirectory()会创建Path表示的路径，下面是一个示例：

```
Path path = Paths.get("data/subdir");
try {
    Path newDir = Files.createDirectory(path);
} catch(FileAlreadyExistsException e){
    // the directory already exists.
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

第一行创建了一个Path实例，表示需要创建的目录。接着用try-catch把

Files.createDirectory()的调用捕获住。如果创建成功，那么返回值就是新创建的路径。

如果目录已经存在了，那么会抛出java.nio.file.FileAlreadyExistException异常。

如果出现其他问题，会抛出一个IOException。比如说，要创建的目录的父目录不存在，那么就会抛出IOException。父目录指的是你要创建的目录所在的位置。也就是新创建的目录的上一级父目录。

Files.copy()

Files.copy()方法可以吧一个文件从一个地址复制到另一个位置。例如：

```
Path sourcePath = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");
try {
    Files.copy(sourcePath, destinationPath);
} catch(FileAlreadyExistsException e) {
    //destination file already exists
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

这个例子当中，首先创建了原文件和目标文件的Path实例。然后把它们作为参数，传递给Files.copy(),接着就会进行文件拷贝。

如果目标文件已经存在，就会抛出java.nio.file.FileAlreadyExistsException异常。类似的吐过中间出错了，也会抛出IOException。

覆盖已经存在的文件(Overwriting Existing Files)

copy操作可以强制覆盖已经存在的目标文件。下面是具体的示例：

```
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) {
    //destination file already exists
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

注意copy方法的第三个参数，这个参数决定了是否可以覆盖文件。

Files.move()

Java NIO的Files类也包含了移动的文件的接口。移动文件和重命名是一样的，但是

还会改变文件的目录位置。java.io.File类中的renameTo()方法与之功能是一样的。

```
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) {
    //moving file failed.
    e.printStackTrace();
}
```

首先创建源路径和目标路径的，原路径指的是需要移动的文件的初始路径，目标路径是指需要移动到的位置。

这里move的第三个参数也允许我们覆盖已有的文件。

Files.delete()

Files.delete()方法可以删除一个文件或目录：

```
Path path = Paths.get("data/subdir/logging-moved.properties");
try {
    Files.delete(path);
} catch (IOException e) {
    //deleting file failed
    e.printStackTrace();
}
```

首先创建需要删除的文件的path对象。接着就可以调用delete了。

Files.walkFileTree()

Files.walkFileTree()方法具有递归遍历目录的功能。walkFileTree接受一个Path和FileVisitor作为参数。Path对象是需要遍历的目录，FileVistor则会在每次遍历中被调用。

下面先来看一下FileVisitor这个接口的定义：

```
public interface FileVisitor {
    public FileVisitResult preVisitDirectory(
        Path dir, BasicFileAttributes attrs) throws IOException;

    public FileVisitResult visitFile(
        Path file, BasicFileAttributes attrs) throws IOException;

    public FileVisitResult visitFileFailed(
        Path file, IOException exc) throws IOException;
```

```
    public FileVisitResult postVisitDirectory(
        Path dir, IOException exc) throws IOException {

}
```

FileVisitor需要调用方自行实现，然后作为参数传入walkFileTree().FileVisitor的每个方法会在遍历过程中被调用多次。如果不需要处理每个方法，那么可以继承他的默认实现类SimpleFileVisitor，它将所有的接口做了空实现。

下面看一个walkFileTree()的示例：

```
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的方法会在不同时机被调用： preVisitDirectory()在访问目录前被调用。postVisitDirectory()在访问后调用。

visitFile()会在整个遍历过程中的每次访问文件都被调用。他不是针对目录的，而是

针对文件的。visitFileFailed()调用则是在文件访问失败的时候。例如，当缺少合适的权限或者其他错误。

上述四个方法都返回一个FileVisitResult枚举对象。具体的可选枚举项包括：

- CONTINUE
- TERMINATE
- SKIP_SIBLINGS
- SKIP_SUBTREE

返回这个枚举值可以让调用方决定文件遍历是否需要继续。 CONTINE表示文件遍历和正常情况下一样继续。

TERMINATE表示文件访问需要终止。

SKIP_SIBLINGS表示文件访问继续，但是不需要访问其他同级文件或目录。

SKIP_SUBTREE表示继续访问，但是不需要访问该目录下的子目录。这个枚举值仅在preVisitDirectory()中返回才有效。如果在另外几个方法中返回，那么会被理解为CONTINE。

Searching For Files

下面看一个例子，我们通过walkFileTree()来寻找一个README.txt文件：

```
Path rootPath = Paths.get("data");
String fileToFind = File.separator + "README.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();
}
```

Deleting Directies Recursively

Files.walkFileTree()也可以用来删除一个目录以及内部的所有文件和子目。

Files.delete()只用用于删除一个空目录。我们通过遍历目录，然后在visitFile()接口中删除所有文件，最后在postVisitDirectory()内删除目录本身。

```
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();
}
```

Additional Methods in the Files Class

java.nio.file.Files类还有其他一些很有用的方法，比如创建符号链接，确定文件大小以及设置文件权限等。具体用法可以查阅JavaDoc中的API说明。