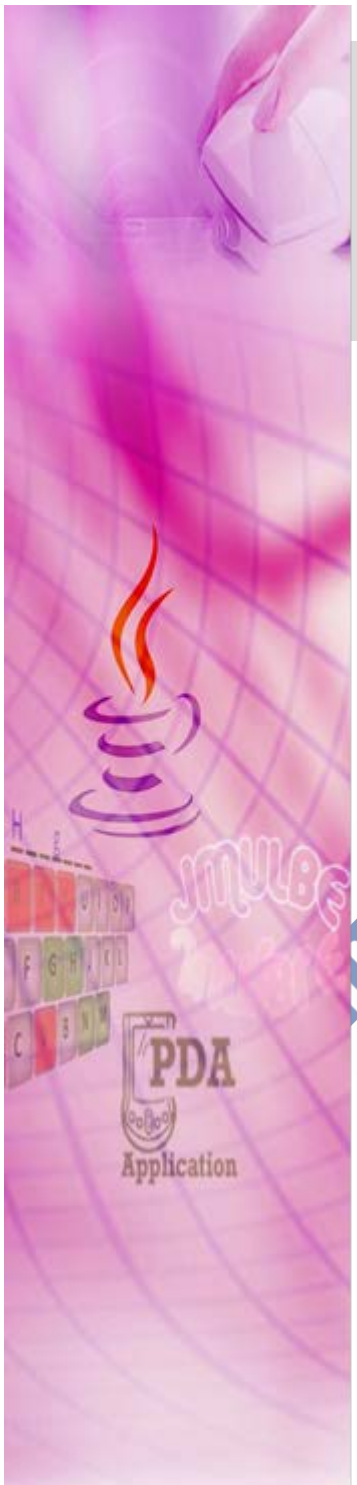


第 8 章

使用 NIO 类和接口

您已学习如何通过 `java.io` 包中的类和接口进行传统的输入/输出 (I/O) 操作。但是，传统 I/O 操作缺少有用的功能，例如文件锁定。因此，为克服此类问题并进一步提高 Java 应用程序的效率，引入了新输入输出 (NIO) 应用程序编程接口 (API) 以提供更简单但高性能的 I/O 操作。

本章重点介绍 NIO 类和接口。此外，您还将学习使用 NIO 类高效地执行文件读取操作。

目标

在本章中，您将学习：

- 熟悉 NIO
- 在文件上执行读写操作


NIIT
training
china
.com

介绍 NIO

在对文件执行 I/O 操作的 Java 应用程序中，您需要存储数据，从文件中检索数据，以及使用目录访问文件。由于 I/O 操作利用系统资源，因此必须高效执行以提高应用程序的性能并达到最大处理生产力。`java.io` 包针对 I/O 操作提供的功能具有性能相关问题，例如在某些数据传输完成之前执行写操作的线程将被阻止。而且，还存在一些效率相关问题，例如没有复制文件或目录的方法。为克服这些问题并进一步简化 I/O 操作，Java 提供了 `java.nio` 包。`java.nio` 包提供用于提高 I/O 操作速度和性能的子包和类。

要使用 NIO API 对文件和目录执行 I/O 操作，您需要利用 `Path` 接口和 `Paths` 类。此外，要使用文件和目录，您需要使用 `Files` 类。而且，NIO 通过实现监视服务支持监视目录。

使用 Path 接口和 Paths 类

`Path` 接口是 `java.nio.file` 包的组件，为识别和使用文件及目录提供支持。`Path` 接口用于表示位于文件系统中的文件或目录的路径。路径包含文件名、目录和磁盘。例如，`D:\Java\Sample\explorer.txt` 表示路径，其中 `explorer.txt` 是文件名，`Java\Sample\` 是目录名，`D:\` 是文件系统中驱动器的名称。

为了使用路径，您需要创建 `Path` 接口的引用。要获取 `Path` 引用，您需要使用 `java.nio.file` 包中 `Paths` 帮助器类的 `get()` 方法。例如，要获取 `Path` 引用，您可以使用以下代码段：

```
Path pathobject = Paths.get("D:\Java\Sample\explorer.txt");
```

在上述代码段中，创建了 `Path` 引用 `pathobject`。该引用表示路径 `D:\Java\Sample\explorer.txt`。

此外，`Path` 接口提供多种方法（例如 `getFileName()`）获取文件或目录路径的有关信息。

下表列出 Path 接口的一些常用方法。

方法名	*描述*	*示例*
Path getFileName()	*返回文件名。*	*Path pathobj = Paths.get("D:/NIODemo/Hello.java");* *System.out.println(pathobj.getFileName());* *上述代码段的输出为：* *Hello.java*
FileSystem getFileSystem()	*返回文件系统名。*	*Path target=Paths.get("D:/Hello.java");* *System.out.println(target.getFileSystem());* *上述代码段的输出为：* *sun.nio.fs.WindowsFileSystem@61316264* *@ 后的数字可变。*
int getNameCount()。	*返回构成路径的元素数量，不包括根盘符，例如 D:/。*	*Path target=Paths.get("D:/NIODemo/Hello.java");* *System.out.println(target.getNameCount());* *上述代码段的输出为：* *2*
Path getName(int index)	*返回索引值指定的路径的 name 元素。索引值 0 指定离根最近的名称。索引值 count-1 指定离根最远的名称。*	*Path target=Paths.get("D:/NIODemo/Hello.java");* *System.out.println(target.getName(0));* *上述代码段的输出为：* *NIODemo*
Path getParent()	*返回文件或目录所在的路径。如果没有指定路径或者路径不包含父级，那么将返回 null。*	*Path target=Paths.get("D:/NIODemo/Hello.java");* *System.out.println(target.getParent());* *上述代码段的输出为：* *D:\NIODemo*

方法名	描述	示例
Path getRoot()	返回文件的根盘符组件，例如D:/。	Path target=Paths.get("D:/NIODemo/Hello.java"); System.out.println(target.getRoot()); 上述代码段的输出为: D:\
boolean isAbsolute()	如果路径是绝对路径，返回true。绝对路径表示用于查找路径的整个路径层次结构。	Path target=Paths.get("D:/NIODemo/Hello.java"); System.out.println(target.isAbsolute()); 上述代码段的输出为: True
int compareTo(Path other)	返回两个路径的比较结果。如果两个路径相同，返回0。如果指定的路径大于指定为参数的路径，返回大于0的值。如果指定的路径小于指定为参数的路径，返回小于0的值。	Path target=Paths.get("D:/NIODemo/Hello.java"); Path comparePath=Paths.get("D:/NIODemo/Hello.java"); Path compareNewPath=Paths.get("D:/NIODemo/NIO/Hello.java"); System.out.println(comparePath.compareTo(target)); System.out.println(compareNewPath.compareTo(target)); 上述代码段的输出为: 0 6

Path 接口的方法

注释

为了获得与上表显示相同的输出，确保上述代码段使用的文件在各自位置中创建。

操作文件和目录

思考一个 Java 应用程序的场景，您需要对文件和目录执行多种操作，例如搜索文件，检查文件或目录是否存在，或者创建或删除文件或目录。为满足上述要求，您可以使用 `java.nio.file` 包中的 `Files` 类的方法。此外，为了遍历文件和目录，您需要使用 `FileVisitor` 接口。

可使用 `java.nio.file.Files` 类执行的一些文件操作有：

- 创建文件或目录。
- 复制文件或目录。
- 移动文件或目录。
- 检查文件或目录是否存在。
- 删除文件或目录。
- 遍历文件树。

创建文件或目录

要创建文件，需要使用 `createFile()` 方法。`createFile()` 方法的签名是：

```
static Path createFile(Path path, FileAttribute<?>... attrs)
```

该方法接受 `Path` 引用并在 `Path` 引用变量指定的位置处创建新文件。`FileAttribute` 引用用于指定一些属性，这些属性定义创建文件的目的是为了读和/或写和/或执行。这些属性依赖于文件系统。因此，您需要利用特定于文件系统的文件权限类及其帮助器类。例如，对于兼容 POSIX 的文件系统（例如 Unix），您可以使用 `PosixFilePermission` 枚举和 `PosixFilePermissions` 类，如以下代码段所示：

```
Path target = Paths.get("\Backup\MyStuff.txt");
Set<PosixFilePermission> perms
= PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr
= PosixFilePermissions.asFileAttribute(perms);
Files.createFile(target, attr);
```

在上述代码中，`rw-rw-rw-` 指定在 `MyStuff.txt` 文件创建时向它应用的权限。如果文件位置不存在，将抛出 `java.nio.file.NoSuchFileException` 异常。此外，如果您尝试在相同位置创建已经存在的文件，将发生 `java.nio.file.FileAlreadyExistsException` 异常。但是，不是必须要指定文件属性。以下代码段显示如何创建文件而不指定文件属性：

```
Path pathObject = Paths.get("D:/NIODemo.java");
Files.createFile(pathObject);
```

在上述代码段中，`pathObject` 是 `Path` 接口的引用变量。`pathObject` 存储 `get()` 方法返回的路径。`pathObject` 引用变量传递给 `createFile()` 方法。`createFile()` 方法在 D:\ 盘中创建文件 `NIODemo.java`。

类似地，要创建目录，使用 `createDirectory()` 方法。`createDirectory()` 方法的签名是：

```
static path createDirectory(Path dir, FileAttribute<?>... attrs)
```

该方法接受 Path 引用并在 path 引用变量指定的位置处创建新目录。FileAttribute 引用用于指定一些属性，这些属性定义创建目录的目的是为了读和/或写和/或执行不是必须要指定上述文件属性。如果尝试在相同位置创建已存在的目录，将发生 java.nio.file.FileAlreadyExistsException 异常。

以下代码段显示如何创建目录而不指定文件属性：

```
Path pathObject = Paths.get("D:/NIO");
Files.createDirectory(pathObject);
```

在上述代码段中，pathObject 是 Path 接口的引用变量。pathObject 存储 get() 方法返回的路径。pathObject 引用变量传递给 createDirectory() 方法。createDirectory() 方法在 D:\ 盘中创建目录 NIO。

如果需要创建一层或多层目录（例如 D:\User1\Java\Programs），需要指定相关路径并使用 createDirectories(Path dir, FileAttribute<?>... attrs) 方法。如果目录已经存在，该方法不会抛出异常。

以下代码段显示了如何使用 createDirectories() 方法创建多个目录：

```
Path newdir = Paths.get("D:/User1/Java/Programs");
try
          {
      Files.createDirectories(newdir);
   }
       catch (IOException e)
          {
System.err.println(e);
   }
```

在上述代码段中，createDirectories() 方法创建路径引用 newdir 指定的目录。

复制文件或目录

要复制文件，需要使用 Files 类的 copy() 方法。copy() 方法的签名是：

```
public static Path copy(Path source, Path target, CopyOption... options)
```

该方法接受两个 Path 引用，分别用于要复制的源文件和要创建为源文件副本的目标文件。并且使用 CopyOption... options 指定一个或多个复制选项以确定应如何执行复制过程。不是必须要指定复制选项。这些选项列在 StandardCopyOption 和 LinkOption 枚举下。

下表列出了可使用 copy() 方法指定的常用复制选项。

枚举	*选项*	*描述*
StandardCopyOption	*REPLACE_EXISTING*	*该选项允许即便在指定目标文件已经存在的情况下仍执行复制过程。否则，如果没有指定该选项并且目标文件已经存在，将抛出异常。*
StandardCopyOption	*COPY_ATTRIBUTES*	*该选项允许将源文件关联的文件属性复制到目标文件。文件属性定义文件的特征，例如创建时间和上次修改时间。*
LinkOption	*NOFOLLOW_LINKS*	*该选项允许指定符号链接不跟随到实际的目标文件。符号链接是包含对另一目标文件或目录的引用的文件。*

用于 copy() 方法的复制选项

以下代码段显示如何复制文件：

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardCopyOption.*;

public class NIOdemo
{
public static void main(String[] args)
    {
Path source = Paths.get("D:/Hello.java");
Path target=Paths.get("D:/NIODemo/Hello.java");
try
{
Files.copy(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
}
catch(IOException e)
{
System.out.println(e);
}
    }
}
```

在上述代码段中，要复制的源文件由 source 路径引用指定，要通过复制过程创建的文件在 D:/NIODemo/Hello.java 位置处指定。如果要复制的源文件不存在，将抛出 java.nio.file.NoSuchFileException 异常。此外，如果目标文件已经存在，并且没有指定 REPLACE_EXISTING 选项，将抛出 FileAlreadyExistsException 异常。

移动文件或目录

要移动现有文件或目录，您可以使用 Files 类中指定的 move() 方法。move() 方法的签名是：

```
public static Path move(Path source, Path target, CopyOption... options)
```

该方法接受两个 Path 引用，分别用于要移动的源文件或目录以及应将源文件或目录移动到的目标文件或目录。但是，如果指定的目标文件已经存在，移动过程将失败。并且使用 CopyOption... options 指定一个或多个复制选项以确定应如何执行复制过程。这些选项列在 StandardCopyOption 枚举下。下表列出了可使用 move() 方法指定的复制选项。

选项	*描述*
REPLACE_EXISTING	*该选项允许即便在指定目标文件已经存在的情况下仍执行移动过程。*
ATOMIC_MOVE	*该选项用于将移动过程作为原子操作执行。原子操作执行期间不能中断。*

用于 move() 方法的复制选项

以下代码显示如何将现有文件移到另一位置：

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardCopyOption.*;

public class NIOdemo {
public static void main(String[] args)
{
Path source = Paths.get("D:/NIODemo/Hello.java");
Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
try
{
Files.move(source, target, REPLACE_EXISTING);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

在上述代码中，D:/NIODemo 目录中的 Hello.java 文件移动到 D:\NIODemo\NIO 目录。如果指定位置处没有源文件，或者如果指定了无效目标位置，将抛出 NoSuchFileException 异常。

以下代码段显示如何将现有目录移到另一目录：

```
Path source = Paths.get("D:/Pictures");
Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
```

```
Files.move(source, target, REPLACE_EXISTING);
```

执行上述代码段时，D:\Pictures 路径中指定的内容将移动到 NewEmptyDirectory 目录。移动目录时，必须确保目标目录为空。否则，将抛出 DirectoryNotEmptyException 异常。如果目标目录位置在指定位置处不存在，将抛出 NoSuchFileException 异常。

检查文件或目录是否存在

要检查文件或目录是否存在或者是否不存在，可以分别使用 Files 类的 exists() 和 notExists() 方法。exists() 方法测试文件是否存在。exists() 方法的签名是：

```
public static boolean exists(Path path, LinkOption... options)
```

该方法接受路径引用 path，将验证该路径引用是否存在。此外，使用链接选项指定路径表示符号链接时应怎样处理符号链接。如果文件或目录存在，该方法返回 true。否则，返回 false。以下代码段显示了如何使用 exists() 方法：

```
Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
Boolean pathExists = Files.exists(target, LinkOption.NOFOLLOW_LINKS);
System.out.println(pathExists);
```

在上述代码段中，验证路径 D:/NIODemo/NIO/NewEmptyDirectory 是否存在。

类似地，您可以使用 notExists() 方法检查文件或目录是否不存在。如果文件或目录不存在，该方法返回 true。否则，返回 false。以下代码段显示了如何使用 notExists() 方法：

```
Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
Boolean pathExists = Files.notExists(target, LinkOption.NOFOLLOW_LINKS);
System.out.println(pathExists);
```

在上述代码段中，验证路径 D:/NIODemo/NIO/NewEmptyDirectory 是否不存在。

删除文件或目录

要删除文件、目录或符号链接，可以使用 delete(Path path) 方法或 deleteIfExists(Path path) 方法。delete() 方法的签名是：

```
public static void delete(Path path)
```

上述方法接受指定要删除路径的 Path 引用。该方法可能会抛出以下异常：

- NoSuchFileException：路径不存在时抛出。
- DirectoryNotEmptyException：路径指定不为空的目录时抛出。
- IOException：发生 I/O 错误时抛出。
- SecurityException：拒绝删除文件的权限时抛出。

以下代码段显示如何删除文件：

```
Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
Files.delete(target);
```

执行上述代码段后，将删除 D:\NIODemo\NIO 目录中的 Hello.java 文件。

类似地，您可以使用 deleteIfExists() 方法删除文件或目录。该方法的签名是：

```
public static boolean deleteIfExists(Path path)
```

上述方法接受指定要删除路径的 path 引用。但是指定路径不存在时，该方法不会抛出异常。在这种情况下，它将返回布尔值 false。以下代码段显示了如何使用 deleteIfExists() 方法：

```
Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
System.out.println(Files.deleteIfExists(target));
```

执行上述代码段后，如果 D:\NIODemo\NIO 目录中存在 Hello.java 文件，将删除该文件并打印相应的布尔值。

遍历文件树

思考场景：您需要创建一个应遍历目录结构中所有文件夹和文件的 Java 应用程序以执行诸如创建、查找或删除文件之类的操作。并且希望将某目录内容以递归方式复制到另一目录。要满足上述需求，您可以使用 NIO 提供的 FileVisitor 接口。该接口位于 java.nio.file 包中。

FileVisitor 接口针对以递归方式遍历文件树提供支持。还允许获取对遍历过程的控制并在遍历过程中执行操作。为在遍历过程中执行操作，该接口提供了一些方法。您需要重写以下方法以在遍历过程中执行操作：

- FileVisitResult preVisitDirectory()：该方法用于目录，并在访问该目录内容前调用。
- FileVisitResult preVisitDirectory()：该方法用于目录，并在访问该目录所有内容后调用。
- FileVisitResult visitFile()：该方法用于文件，并在遍历过程中访问文件时调用。
- FileVisitResult visitFileFailed()：该方法用于文件，并在要查看而无法访问该文件时调用。

上述每种方法都返回类型 FileVisitResult 枚举的值。该值用于确定如何继续遍历过程。FileVisitResult 枚举中的一些常量值有：

- CONTINUE：用于指示遍历必须继续。
- SKIP_SUBTREE：用于指示遍历必须继续而不访问目录内部的内容。
- TERMINATE：用于指示遍历必须终止。

此外，要开始遍历过程，您需要使用 Files 类中的 walkFileTree() 方法。该方法接受需要开始遍历过程的起始路径和实现 FileVisitor 接口的类的实例。该方法的签名是：

```
public static Path walkFileTree(Path startpoint, FileVisitor<? super Path>
visitor)
```

在上述签名中，startpoint 表示应开始遍历的路径引用。FileVisitor<? super Path> visitor 表示实现 FileVisitor 接口的类的实例。

以下代码显示如何通过 FileVisitor 接口实现文件树遍历过程：

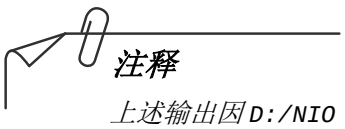
```
import java.io.IOException;
import java.nio.file.FileVisitResult;
import java.nio.file.FileVisitor;
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;

class MyFileVisitor implements FileVisitor<Path> {

public FileVisitResult postVisitDirectory(Path dir, IOException e) throws
IOException {
System.out.println("Just Visited " + dir);
return FileVisitResult.CONTINUE;
}


public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
throws IOException {

System.out.println("About to visit " + dir);
return FileVisitResult.CONTINUE;

}

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

System.out.println("Currently visiting "+file);
System.out.println("Is this file a directory:"+ Files.isDirectory(file));
System.out.println("Checking done..!!");
return FileVisitResult.CONTINUE;

}

public FileVisitResult visitFileFailed(Path file, IOException e) throws
IOException {
System.err.println(e.getMessage());
return FileVisitResult.CONTINUE;
}
}
public class NIODemo
{
public static void main(String a[]) throws IOException
{
Path listDir = Paths.get("D:/NIO");
MyFileVisitor obj = new MyFileVisitor();
Files.walkFileTree(listDir, obj);
}
}
```

上述代码的输出是：

```
About to visit D:\NIO
About to visit D:\NIO\Hello
Just Visited D:\NIO\Hello
Currently visiting D:\NIO\Hello.txt
Is this file a directory: false
```

```
Checking done..!!
Just Visited D:\NIO
```



注释

上述输出因 D:/NIO 目录的内容而异。

在上述代码中，listDir 表示应开始遍历的路径引用。然后，使用 walkFileTree() 方法和 MyFileVisitor 类的 obj 实例从 listDir 指定的路径开始遍历。在该类中，重写 FileVisitor 接口的方法。postVisitDirectory() 方法打印 Just Visited 消息和文件名或目录名。preVisitDirectory() 方法打印 About to visit 消息和文件名或目录名。visitFile() 方法打印当前访问文件的名称，并且还使用 Files.isDirectory() 方法检查访问的路径是否是目录。此外，visitFileFailed() 方法用于在文件访问失败时打印消息。

但是，可能要求您无需提供 FileVisitor 接口的所有方法的实现，但只需提供几个方法的实现。在这种情况下，您可以扩展 SimpleFileVisitor 类。例如，只需要在查看文件树中的文件时打印文件名的情况。SimpleFileVisitor 类实现 FileVisitor 接口，因此可根据应用程序的需求重写其方法。

思考场景：您需要执行模式匹配以获取与某个模式（例如 *.java）匹配的文件名的列表和计数。因此，要将文件名与模式名匹配，可以使用 PathMatcher 类。以下代码演示如何在计算机的 D:\ 盘中搜索所有 .java 文件。

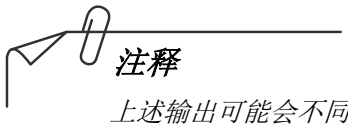
```
import java.io.IOException;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.FileVisitOption;
import java.nio.file.FileVisitResult;
import java.nio.file.FileVisitor;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.EnumSet;

class SearchDemo implements FileVisitor {

private final PathMatcher matcher;
int counter = 0;

public SearchDemo(String pattern) {
FileSystem fs=FileSystems.getDefault();
matcher = fs.getPathMatcher("glob:"+ pattern);
}

void search(Path file) throws IOException {
Path name = file.getFileName();
```

```
if (name != null && matcher.matches(name)) {

System.out.println("Searched file located:" + name + " in " +
file.getParent().toAbsolutePath());
counter++;
}

}


public FileVisitResult postVisitDirectory(Object dir, IOException exc)
throws IOException {

return FileVisitResult.CONTINUE;
}


public FileVisitResult preVisitDirectory(Object dir, BasicFileAttributes
attrs)
throws IOException {
return FileVisitResult.CONTINUE;
}


public FileVisitResult visitFile(Object file, BasicFileAttributes attrs)
throws IOException {
search((Path) file);
return FileVisitResult.CONTINUE;
}


public FileVisitResult postVisitDirectory(Object dir, IOException exc)
throws IOException {
return FileVisitResult.CONTINUE;
}
}

class MainClass {

public static void main(String[] args) throws IOException {
String pattern = "*.java";
Path fileTree = Paths.get("D:/");
SearchDemo walk = new SearchDemo(pattern);
EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
Files.walkFileTree(fileTree, opts, Integer.MAX_VALUE, walk);
System.out.println("Total files found:" + walk.counter);
}
}
```

上述代码的输出是：

```
Searched file located:Hello.java in D:\NIODemo
Searched file located:NIODemo - Copy.java in D:\
Searched file located:NIODemo.java in D:\
```

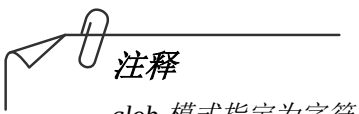
```
Searched file located:ThreadOutput.java in D:\PPS
Searched file located:Puzzle.java in D:\
Total files found:5
```



注释

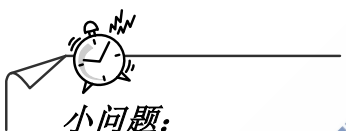
上述输出可能会不同。

在上述代码的 `main()` 方法中，模式指定为 `*.java`。在构造函数中，使用 `getPathMatcher()` 方法。该方法用于创建 `PathMatcher` 以对路径执行匹配操作。`FileSystems.getDefault()` 方法返回 `FileSystem` 类的对象。`getPathMatcher()` 方法用于获取 `PathMatcher` 类的对象。该方法接受 glob 模式，例如 glob:*.java。此外，还使用 `matches()` 方法针对 glob 模式比较已浏览文件的名称。通过使用 `counter` 变量，与该模式匹配的总文件数将显示在 `main()` 方法中。



注释

glob 模式指定为字符串并用于指定要针对字符串（例如文件名）进行匹配的模式。



小问题：

`FileVisitResult` 枚举的 _____ 常量用于指示遍历必须结束。

答案：

`TERMINATE`

实现监视服务

思考场景：您需要开发一个 Java 应用程序以监视目录并跟踪对该目录执行的更改。例如，您可能想要跟踪目录并在该目录中创建、删除或修改任何文件时收到相关通知。要完成上述任务，您可以使用监视服务 API，这是 `java.nio.file` 的一部分。该 API 用于实现文件更改通知功能以使 Java 应用程序能够监视和检测对目录执行的更改。

监视服务可通过执行以下步骤实现：

1. 创建监视服务。

2. 向监视服务注册要监视的对象。
3. 等待事件发生。
4. 检索监视键。
5. 检索键的未决事件。
6. 重置键。
7. 关闭监视服务。

创建监视服务

监视服务用于监视对象并检测修改对象的时间。您可以通过使用以下代码段获取 WatchService 的引用：

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

向监视服务注册要监视的对象

创建监视服务后，您需要向监视服务注册要监视其修改情况的对象。可监视实现 Watchable 接口的任意对象的更改和事件。Path 类实现 Watchable 接口，允许您注册目录以监视更改。为注册对象，Watchable 接口提供了 register() 方法。register() 方法的签名是：

```
WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)
```

在上述签名中，watcher 引用通过 newWatchService() 方法创建的监视服务。WatchEvent.Kind<?>... events 指定监视服务应监视并提供通知的事件。这些事件可以使用 StandardWatchEventKinds 类中定义的常量值指定。下表列出了 StandardWatchEventKinds 类的一些常量。

方法	*描述*
StandardWatchEventKinds.ENTRY_CREATE	*创建目录条目。*
StandardWatchEventKinds.ENTRY_DELETE	*删除目录条目。*
StandardWatchEventKinds.ENTRY_MODIFY	*修改目录条目。*

StandardWatchEventKinds 类常量

register() 方法返回 WatchKey 对象作为每个已注册目录的注册标记。向监视服务注册 Watchable 对象时将创建一个监视键。注册后，目录的对应 WatchKey 应处于 ready 状态。在就绪状态下，WatchKey 可以接受任何事件。

以下代码段用于接受注册标记：

```
WatchKey key = path.register(watchService,
StandardWatchEventKinds.ENTRY_CREATE,StandardWatchEventKinds.ENTRY_MODIFY,
StandardWatchEventKinds.ENTRY_DELETE);
```

在上述代码段中，向监视服务注册 path 时在 key 中接受注册标记。

等待事件发生

由于目录中的事件随时可能发生，因此有必要一直坚持该目录。这要求在无限循环中监视目录，如以下代码段所示：

```
while(true)
{
//retrieve and process the incoming events
…
}
```

检索监视键

当目录中发生事件时，其对应的监视键的状态将更改为 signaled。对应的监视键被置于可从中检索该键的监视服务队列中。

下表列出可用于检索已排队键的 WatchService 类的方法。

方法	*描述*
WatchKey take()	*该方法返回已排队键并等到键进入队列或无限循环停止。*
WatchKey poll()	*该方法在已排队键可用时返回该键，否则返回 null 值。*
WatchKey poll(long timeNum, TimeUnit)	*该方法在已排队键可用时返回该键，否则等待 timeNum 指示的时间段然后重试。TimeUnit 指定时间单位，例如纳秒或毫秒。*

检索已排队键的方法

以下代码段显示如何在无限循环中检索已排队键：

```
while (true)
{
    //retrieve and remove the next watch key
    final WatchKey key = watchService.poll();
    //the thread flow gets here immediately with an available key or a null
    value
}
```

检索键的未决事件

键处于 signaled 状态并发生其他事件时，事件将排队等待对应键。对应键的未决事件可以使用 WatchKey 接口的 pollEvents() 方法检索和移动。该方法返回类型 WatchEvent 对象的未决事件列表。该列表可以进一步迭代以单独处理各个事件。列表类型为 WatchEvent<T>。

以下代码段显示如何检索监视键的未决事件：

```
while (true)
          {
                final WatchKey key = watchService.poll();
                //   process events of the watch key
                for (WatchEvent<?> watchEvent : key.pollEvents())
                      {
                      // code for processing the events
                      }
          }
```

在上述代码段中，for 循环用于检索 key 监视键的未决事件。

此外，如果想要获取事件的更多信息，可以使用 WatchEvent<T> 接口的方法。

下表列出了 WatchEvent<T> 接口的方法。

方法	*描述*
WatchEvent.Kind<T> kind()	*该方法返回已发生事件的类型。*
T context()	*该方法返回事件的内容。例如，创建、删除或修改目录条目时，上下文是创建、删除或修改的条目。*

WatchEvent<T> 接口的方法

以下代码段显示了如何使用 kind() 方法：

```
for (WatchEvent<?> watchEvent : key.pollEvents())
{
Kind<?> eventKind = watchEvent.kind(); //determine the kind of the event
}
```

在上述代码段中，eventKind 存储 kind() 方法返回的值。

重置键

要确保监视键可用于监视事件，您需要重置处于 signaled 状态的监视键。reset() 方法将键恢复为 ready 状态。如果键有效，该方法返回 true，否则返回 false。以下代码段显示如何重置键：

```
while(true)
{
    final WatchKey key = watchService.poll();

// code to get event kind
    boolean valid = key.reset();
// code to break
}
```

关闭监视服务

最后，不再需要监视服务时，您可以使用 `WatchService` 接口的 `close()` 方法将其关闭，或确保在 `try-` 资源块中创建监视服务以便在不再需要监视服务时它能够自动关闭。以下代码段显示了如何使用 `createDirectories()` 方法关闭监视服务：

```
watchService.close();
```

以下代码显示如何实现监视服务：

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.WatchEvent.*;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import static java.nio.file.StandardWatchEventKinds.*;
import java.nio.file.WatchEvent;

public class WatchServiceDemo {
private Path path = null;
WatchService watchService;
private void initializeService() {
path = Paths.get("D:/NIO");  // the path to the directory to be monitored
try {
watchService = FileSystems.getDefault().newWatchService();
path.register(watchService, ENTRY_CREATE, ENTRY_DELETE,ENTRY_MODIFY);   //
register the watch service for the path        }
catch (IOException e) {
System.out.println("IOException"+ e.getMessage());
}
}

private void monitorDirectory() {
WatchKey key = null;
while(true) {  // infinite loop to monitor changes
try {
key = watchService.take(); // get watch key
for (WatchEvent event : key.pollEvents()) {
Kind kind = event.kind(); // get event kind
System.out.println("The event that occurred on " + event.context().toString()
+ " is " + kind);
}
} catch (InterruptedException e) {
System.out.println("InterruptedException:"+e.getMessage());
}
boolean reset = key.reset();
if(!reset)
break;
}
}
```

```
public static void main(String[] args) {
WatchServiceDemo watchservicedemo = new WatchServiceDemo();
watchservicedemo.initializeService();
watchservicedemo.monitorDirectory();
}
}
```

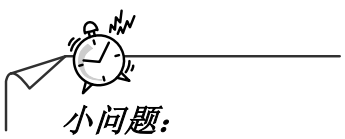
在上述代码中，创建了监视服务 `watchService`。要监视其更改的路径是 `D:/NIO`。然后，向 `watchService` 注册该目录路径监视其创建、删除和修改事件。之后，在 `monitorDirectory()` 方法中创建无限循环以时刻监视目录的任何修改情况。在该方法中，检索事件并打印与已发生事件类型有关的信息。

执行时，创建新文本文档 `WatchDemo` 时，检测该创建操作并显示以下输出：

```
The event that occurred on WatchDemo.txt is ENTRY_CREATE
```

接下来，修改 `WatchDemo.txt` 文件的内容并保存时，检测该修改操作并显示以下输出：

```
The event that occurred on WatchDemo.txt is ENTRY_MODIFY
```



小问题：

_____ *状态表示注册目录时的监视键的状态。*

答案：

ready

在文件上执行读写操作

思考频繁读/写大量数据的应用程序的场景。由于读写操作需要利用系统资源，例如处理器和存储介质（比如磁盘），因此应用程序使用有效方法执行这些读写操作很关键。此外，最小化上述操作所耗时间也很重要。实现高效性能并提高读写操作速度的其中一种方法是使用缓冲区。*缓冲区*是指内存中可用于写或读取数据的存储区域。与流相比，缓冲区可用于提供更快的 I/O 操作。

跟传统的 I/O 操作相比，您可以提高应用程序执行的读写操作的速度，例如使用 NIO 在计算机硬盘的文件中存储文本。此外，使用 NIO 会降低代码复杂性。

读取文件

思考场景：您需要创建一个 Java 应用程序以高效读取文件内容并在控制台窗口中显示这些内容。要读取文件，您可以使用 `java.nio.file.Files` 类中定义的 `newBufferedReader()` 方法。

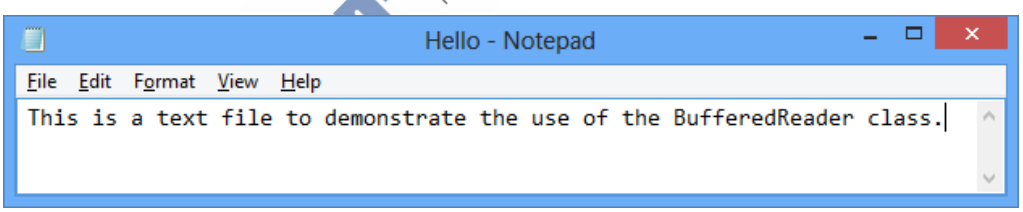
该方法返回可用于高效读取文件文本目的的 `BufferedReader` 对象。

以下代码段显示如何创建 `BufferedReader` 引用变量以读取文件：

```
BufferedReader name = Files.newBufferedReader(path, charset);
```

在上述代码段中，`name` 指定 `BufferedReader` 类的引用，`path` 是 `Path` 接口的引用变量，用于指定存储的文件路径，`charset` 是 `Charset` 类的引用变量，用于指定字符集的类型（例如 `US-ASCII`）以用于将文件字节转换为 Unicode 字符。如果发生 I/O 错误，`newBufferedReader()` 方法将抛出 `IOException`。

思考场景：您需要读取文件 `Hello.txt` 的内容，该文件位于计算机的 `D:\` 盘中。`Hello.txt` 文件的内容如下图所示。



Hello.txt 文件的内容

您需要开发一个程序，通过 `BufferedReader` 类读取上述文件中的字符并显示。您可以使用以下代码实现上述功能：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedReaderDemo {
public static void main(String a[]) {
```

```
Path pathObject = Paths.get("D:/Hello.java");
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(pathObject, charset)) {
String line = null;
while ((line = reader.readLine()) != null) {
System.out.println(line);
}
} catch (IOException x) {
System.out.println(x);
}
}
}
```

上述代码的输出是：

```
This is a text file to demonstrate the use of the BufferedReader class.
```

在上述代码中，创建表示要读取文件所在位置的 `pathObject`。然后，创建 `BufferedReader` 引用变量 `reader`。在 `while` 循环中，`readLine()` 方法读取一行文本并将文本赋值给 `String` 引用变量 `line`，它将一直显示到到达文件末尾为止。

写入文件

思考场景：您需要开发 Java 应用程序以在文件中写入内容。要执行上述任务，您可以使用 `java.nio.file.Files` 类中定义的 `newBufferedWriter()` 方法。`newBufferedWriter()` 方法创建 `BufferedWriter` 对象。如果该文件已经存在，那么将打开文件以进行写入。对象的 `write()` 方法用于创建文件（如果文件不存在）以进行写入。

以下代码段显示如何创建 `BufferedWriter` 引用变量以写文件：

```
BufferedWriter name = Files.newBufferedWriter(path, charset, options);
```

在上述代码段中，`name` 是 `BufferedWriter` 引用变量。`path` 是 `Path` 接口的引用变量，用于指定存储的文件。`charset` 是 `Charset` 类的引用变量，用于指定字符集的类型（例如 `US-ASCII`）以将文件字节转换为 Unicode 字符。`options` 参数用于指定如何将文件用于写操作。您可以使用 `StandardOpenOption` 枚举指定选项。

下表列出了可针对写操作指定的一些标准选项。

选项	*描述*
StandardOpenOption.WRITE	*该选项打开文件进行写访问。*
StandardOpenOption.APPEND	*如果打开文件进行写访问，该选项写到文件末尾。*
StandardOpenOption.CREATE	*如果文件不存在，该选项创建新文件。*
StandardOpenOption.CREATE_NEW	*如果存在同名文件，该选项创建新文件但抛出异常。*
StandardOpenOption.DELETE_ON_CLOSE	*该选项尝试在文件关闭后将其删除。*
StandardOpenOption.TRUNCATE_EXISTING	*如果文件已经存在并打开进行写访问，那么其长度将截断到零。*

StandardOpenOption 枚举的各种选项

如果在打开或创建文件时发生 I/O 错误，`newBufferedWriter()` 将抛出 `IOException`。如果在打开文件时指定不受支持的选项，它还会抛出 `UnsupportedOperationException` 异常。

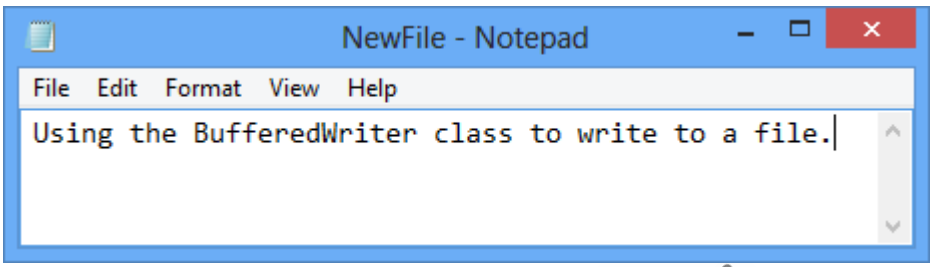
思考场景：您需要开发一个 Java 程序以在计算机的 D:\ 盘创建文件 NewFile.txt，其中包含使用 `BufferedWriter` 类写入文件的文本。您可以使用以下代码实现上述需求：

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class BufferedWriterDemo
{
public static void main(String[] args)
   {
String content = "Using the BufferedWriter class to write to a file.";
Path file = Paths.get("D:/NewFile.txt");
Charset charset = Charset.forName("US-ASCII");
try (BufferedWriter writer = Files.newBufferedWriter(file, charset,
StandardOpenOption.CREATE))
{
writer.write(content);
System.out.println("Done");
} catch (IOException e)
{
System.out.println(e);
```

```
    }
    }
    }
```

在上述代码中，创建表示要写入文件所在位置的 `file` 引用变量。然后，创建 `BufferedWriter` 引用变量 `writer`。`write()` 方法将 `String` 引用变量 `content` 的值写入 `NewFile.txt`。

执行上述代码后，将创建 `NewFile.txt` 文件。`NewFile.txt` 文件的内容如下图所示。



NewFile.txt 文件的内容

在上述代码中，如果 `try (BufferedWriter writer = Files.newBufferedWriter(file, charset, StandardOpenOption.CREATE))` 语句替换为 `try (BufferedWriter writer = Files.newBufferedWriter(file, charset, StandardOpenOption.CREATE_NEW))` 语句，那么将生成以下异常：

```
java.nio.file.FileAlreadyExistsException:D:\NewFile.txt
```

由于指定位置已经存在名为 `NewFile.txt` 的文件，因此无法创建新文件，从而导致发生上述异常。

活动 8.1：使用 NIO 执行文件操作

练习问题

1. 以下哪个 `Paths` 帮助器类的方法用于获取文件路径引用？
   a. `get()`
   b. `getFileName()`
   c. `getName()`
   d. `getRoot()`

2. 指出返回文件系统名称的 `Path` 接口方法。
   a. `getFileName()`
   b. `getFileSystem()`
   c. `FileSystem()`
   d. `getParent()`

3. 指出用于确定路径是否表示整个路径层次结构的 `Path` 接口方法。
   a. `getFileName()`
   b. `getFileSystem()`
   c. `getRoot()`
   d. `isAbsolute()`

4. 指出尝试使用 `createDirectory()` 方法在指定位置创建已经存在的目录时抛出的异常。
   a. `FileAlreadyExistsException`
   b. `DirectoryNotEmptyException`
   c. `NoSuchFileException`
   d. `IOException`

5. 确定针对递归遍历文件结构提供支持的接口。
   a. `FileVisitor`
   b. `FileReader`
   c. `Watchable`
   d. `Path`

小结

在本章中，您学习了：

- java.nio 包提供用于提高 I/O 操作速度和性能的子包和类。
- Path 接口是 java.nio.file 包的组件，为识别和使用文件及目录提供支持。
- 可使用 java.nio.file.Files 类执行的一些文件操作有：
  - 创建文件或目录
  - 复制文件或目录
  - 移动文件或目录
  - 检查文件或目录是否存在
  - 删除文件或目录
  - 遍历文件树
- 要创建文件，需要使用 createFile() 方法。
- 要复制文件，需要使用 Files 类的 copy() 方法。
- 要移动现有文件或目录，您可以使用 Files 类中指定的 move() 方法。
- 要检查文件或目录是否存在或者是否不存在，可以分别使用 Files 类的 exists() 和 notExists() 方法。
- 要删除文件、目录或符号链接，可以使用 delete(Path path) 方法或 deleteIfExists(Path path) 方法。
- FileVisitor 接口针对以递归方式遍历文件树提供支持。
- 监视服务用于监视对象并检测修改对象的时间。
- 可监视实现 Watchable 接口的任意对象的更改和事件。
- 缓冲区是指内存中可用于写或读取数据的存储区域。
- 要读取文件，您可以使用 java.nio.file.Files 类中定义的 newBufferedReader() 方法。