

```

String baseDir = props.getProperty("base.dir");
// May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK that baseDir has separators

```

**注意：**路径不必对应着某个实际存在的文件，它仅仅只是一个抽象的名字序列。你在接下来的小节中将要看到，当你想要创建文件时，首先要创建一个路径，然后才调用方法去创建对应的文件。

组合或解析路径是司空见惯的操作，调用 `p.resolve(q)` 将按照下列规则返回一个路径：

- 如果 `q` 是绝对路径，则结果就是 `q`。
- 否则，根据文件系统的规则，将“`p` 后面跟着 `q`”作为结果。

例如，假设你的应用系统需要查找相对于给定基目录的工作目录，其中基目录是从配置文件中读取的，就像前一个例子一样。

```

Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);

```

`resolve` 方法有一种快捷方式，它接受一个字符串而不是路径：

```
Path workPath = basePath.resolve("work");
```

还有一个很方便的方法 `resolveSibling`，它通过解析指定路径的父路径产生其兄弟路径。例如，如果 `workPath` 是 `/opt/myapp/work`，那么下面的调用

```
Path tempPath = workPath.resolveSibling("temp");
```

将创建 `/opt/myapp/temp`。

`resolve` 的对立面是 `relativize`，即调用 `p.relativize(r)` 将产生路径 `q`，而对 `q` 进行解析的结果正是 `r`。例如，以“`/home/cay`”为目标对“`/home/fred/myprog`”进行相对化操作，会产生“`../fred/myprog`”，其中，我们假设 `..` 表示文件系统中的父目录。

`normalize` 方法将移除所有冗余的 `.` 和 `..` 部件（或者文件系统认为冗余的所有部件）。例如，规范化 `/home/cay/..../fred/./myprog` 将产生 `/home/fred/myprog`。

`toAbsolutePath` 方法将产生给定路径的绝对路径，该绝对路径从根部件开始，例如 `/home/fred/input.txt` 或 `c:\Users\fred\input.txt`。

`Path` 类有许多有用的方法用来将路径断开。下面的代码示例展示了其中部分最有用的方法：

```

Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // the path /home/fred
Path file = p.getFileName(); // the path myprog.properties
Path root = p.getRoot(); // the path /

```

正如你已经在卷 I 中看到的，还可以从 `Path` 对象中构建 `Scanner` 对象：

```
Scanner in = new Scanner(Paths.get("/home/fred/input.txt"));
```

**注意：**偶尔，你可能需要与遗留系统的 API 交互，它们使用的是 `File` 类而不是 `Path` 接口。`Path` 接口有一个 `toFile` 方法，而 `File` 类有一个 `toPath` 方法。

**API** java.nio.file.Paths 7

- static Path get(String first, String... more)

通过连接给定的字符串创建一个路径。

**API** java.nio.file.Path 7

- Path resolve(Path other)

- Path resolve(String other)

如果 `other` 是绝对路径，那么就返回 `other`；否则，返回通过连接 `this` 和 `other` 获得的路径。

- Path resolveSibling(Path other)

- Path resolveSibling(String other)

如果 `other` 是绝对路径，那么就返回 `other`；否则，返回通过连接 `this` 的父路径和 `other` 获得的路径。

- Path relativize(Path other)

返回用 `this` 进行解析，相对于 `other` 的相对路径。

- Path normalize()

移除诸如`.`和`..`等冗余的路径元素。

- Path toAbsolutePath()

返回与该路径等价的绝对路径。

- Path getParent()

返回父路径，或者在该路径没有父路径时，返回 `null`。

- Path getFileName()

返回该路径的最后一个部件，或者在该路径没有任何部件时，返回 `null`。

- Path getRoot()

返回该路径的根部件，或者在该路径没有任何根部件时，返回 `null`。

- toFile()

从该路径中创建一个 `File` 对象。

**API** java.io.File.1.0

- Path toPath() 7

从该文件中创建一个 `Path` 对象。

## 2.5.2 读写文件

`Files` 类可以使普通文件操作变得快捷。例如，可以用下面的方式很容易地读取文件的所有内容：

```
byte[] bytes = Files.readAllBytes(path);
```

如果想将文件当作字符串读入，那么可以在调用 `readAllBytes` 之后执行下面的代码：

```
String content = new String(bytes, charset);
```

但是如果希望将文件当作行序列读入，那么可以调用：

```
List<String> lines = Files.readAllLines(path, charset);
```

相反地，如果希望写出一个字符串到文件中，可以调用：

```
Files.write(path, content.getBytes(charset));
```

向指定文件追加内容，可以调用：

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

还可以用下面的语句将一个行的集合写出到文件中：

```
Files.write(path, lines);
```

这些简便方法适用于处理中等长度的文本文件，如果要处理的文件长度比较大，或者是二进制文件，那么还是应该使用所熟知的输入 / 输出流或者读入器 / 写出器：

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

这些便捷方法可以将你从处理 `FileInputStream`、`FileOutputStream`、`BufferedReader` 和 `BufferedWriter` 的繁复操作中解脱出来。



#### java.nio.file.Files 7

- static byte[] readAllBytes(Path path)
- static List<String> readAllLines(Path path, Charset charset)  
读入文件的内容。
- static Path write(Path path, byte[] contents, OpenOption... options)
- static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)  
将给定内容写出到文件中，并返回 `path`。
- static InputStream newInputStream(Path path, OpenOption... options)
- static OutputStream newOutputStream(Path path, OpenOption... options)
- static BufferedReader newBufferedReader(Path path, Charset charset)
- static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)  
打开一个文件，用于读入或写出。

### 2.5.3 创建文件和目录

创建新目录可以调用

```
Files.createDirectory(path);
```

其中，路径中除最后一个部件外，其他部分都必须是已存在的。要创建路径中的中间目录，应该使用

```
Files.createDirectories(path);
```

可以使用下面的语句创建一个空文件：

```
Files.createFile(path);
```

如果文件已经存在了，那么这个调用就会抛出异常。检查文件是否存在和创建文件是原子性的，如果文件不存在，该文件就会被创建，并且其他程序在此过程中是无法执行文件创建操作的。

有些便捷方法可以用来在给定位置或者系统指定位置创建临时文件或临时目录：

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

其中，`dir` 是一个 `Path` 对象，`prefix` 和 `suffix` 是可以为 `null` 的字符串。例如，调用 `Files.createTempFile(null, ".txt")` 可能会返回一个像 `/tmp/1234405522364837194.txt` 这样的路径。

在创建文件或目录时，可以指定属性，例如文件的拥有者和权限。但是，指定属性的细节取决于文件系统，本书在此不做讨论。

#### API java.nio.file.Files 7

- static Path createFile(Path path, FileAttribute<?>... attrs)
- static Path createDirectory(Path path, FileAttribute<?>... attrs)
- static Path createDirectories(Path path, FileAttribute<?>... attrs)
   
创建一个文件或目录，`createDirectories` 方法还会创建路径中所有的中间目录。
- static Path createTempFile(String prefix, String suffix,
 FileAttribute<?>... attrs)
- static Path createTempFile(Path parentDir, String prefix, String
 suffix, FileAttribute<?>... attrs)
- static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
- static Path createTempDirectory(Path parentDir, String prefix,
 FileAttribute<?>... attrs)

在适合临时文件的位置，或者在给定的父目录中，创建一个临时文件或目录。返回所创建的文件或目录的路径。

### 2.5.4 复制、移动和删除文件

将文件从一个位置复制到另一个位置可以直接调用

```
Files.copy(fromPath, toPath);
```

移动文件（即复制并删除原文件）可以调用

```
Files.move(fromPath, toPath);
```

如果目标路径已经存在，那么复制或移动将失败。如果想要覆盖已有的目标路径，可以使用 `REPLACE_EXISTING` 选项。如果想要复制所有的文件属性，可以使用 `COPY_ATTRIBUTES` 选项。也可以像下面这样同时选择这两个选项：

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
          StandardCopyOption.COPY_ATTRIBUTES);
```

你可以将移动操作定义为原子性的，这样就可以保证要么移动操作成功完成，要么源文件继续保持在原来位置。具体可以使用 `ATOMIC_MOVE` 选项来实现：

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

你还可以将一个输入流复制到 `Path` 中，这表示你想要将该输入流存储到硬盘上。类似地，你可以将一个 `Path` 复制到输出流中。可以使用下面的调用：

```
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

至于其他对 `copy` 的调用，可以根据需要提供相应的复制选项。

最后，删除文件可以调用：

```
Files.delete(path);
```

如果要删除的文件不存在，这个方法就会抛出异常。因此，可转而使用下面的方法：

```
boolean deleted = Files.deleteIfExists(path);
```

该删除方法还可以用来移除空目录。

请查阅表 2-3 以了解对文件操作而言可用的选项。

表 2-3 用于文件操作的标准选项

选 项	描 述
<code>StandardOpenOption</code> ; 与 <code>newBufferedWriter</code> , <code>newInputStream</code> , <code>newOutputStream</code> , <code>write</code> 一起使用	
<code>READ</code>	用于读取而打开
<code>WRITE</code>	用于写入而打开
<code>APPEND</code>	如果用于写入而打开，那么在文件末尾追加
<code>TRUNCATE_EXISTING</code>	如果用于写入而打开，那么移除已有内容
<code>CREATE_NEW</code>	创建新文件并且在文件已存在的情况下会创建失败
<code>CREATE</code>	自动在文件不存在的情况下创建新文件
<code>DELETE_ON_CLOSE</code>	当文件被关闭时，尽“可能”地删除该文件

(续)

选 项	描 述
SPARSE	给文件系统一个提示，表示该文件是稀疏的
DSYN SYN	要求对文件数据   数据和元数据的每次更新都必须同步地写入到存储设备中
StandardCopyOption; 与 copy, move 一起使用	
ATOMIC_MOVE	原子性地移动文件
COPY_ATTRIBUTES	复制文件的属性
REPLACE_EXISTING	如果目标已存在，则替换它
LinkOption; 与上面所有方法以及 exists, isDirectory, isRegularFile 等一起使用	
NOFOLLOW_LINKS	不要跟踪符号链接
FileVisitOption; 与 find, walk, walkFileTree 一起使用	
FOLLOW_LINKS	跟踪符号链接

**API** java.nio.file.Files 7

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)
 

将 from 复制或移动到给定位置，并返回 to。
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)
 

从输入流复制到文件中，或者从文件复制到输出流中，返回复制的字节数。
- static void delete(Path path)
- static boolean deleteIfExists(Path path)
 

删除给定文件或空目录。第一个方法在文件或目录不存在情况下抛出异常，而第二个方法在这种情况下会返回 false。

### 2.5.5 获取文件信息

下面的静态方法都将返回一个 boolean 值，表示检查路径的某个属性的结果：

- exists
- isHidden
- isReadable, isWritable, isExecutable
- isRegularFile, isDirectory, isSymbolicLink

size 方法将返回文件的字节数：

```
long fileSize = Files.size(path);
```

getOwner 方法将文件的拥有者作为 java.nio.file.attribute.UserPrincipal 的一个实例返回。

所有的文件系统都会报告一个基本属性集，它们被封装在 BasicFileAttributes 接口

中，这些属性与上述信息有部分重叠。基本文件属性包括：

- 创建文件、最后一次访问以及最后一次修改文件的时间，这些时间都表示成 `java.nio.file.attribute.FileTime`
- 文件是常规文件、目录还是符号链接，抑或这三者都不是。
- 文件尺寸。
- 文件主键，这是某种类的对象，具体所属类与文件系统相关，有可能是文件的唯一标识符，也可能不是。

要获取这些属性，可以调用

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

如果你了解到用户的文件系统兼容 POSIX，那么你可以获取一个 `PosixFileAttributes` 实例：

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

然后从中找到组拥有者，以及文件的拥有者、组和访问权限。我们不会详细讨论其细节，因为这种信息中很多内容在操作系统之间并不具备可移植性。

#### API `java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`

检查由路径指定的文件的给定属性。

- `static long size(Path path)`

获取文件按字节数度量的尺寸。

- `A readAttributes(Path path, Class<A> type, LinkOption... options)`  
读取类型为 A 的文件属性。

#### API `java.nio.file.attribute.BasicFileAttributes` 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`

- `boolean isSymbolicLink()`

- `long size()`

- `Object fileKey()`

获取所请求的属性。

### 2.5.6 访问目录中的项

静态的 `Files.list` 方法会返回一个可以读取目录中各个项的 `Stream<Path>` 对象。目录是被惰性读取的，这使得处理具有大量项的目录可以变得更高效。

因为读取目录涉及需要关闭的系统资源，所以应该使用 `try` 块：

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
    ...
}
```

`list` 方法不会进入子目录。为了处理目录中的所有子目录，需要使用 `File.walk` 方法。

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
    // Contains all descendants, visited in depth-first order
}
```

下面是加压后的 `src.zip` 树的遍历样例：

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

正如你所见，无论何时，只要遍历的项是目录，那么在进入它之前，会继续访问它的兄弟项。

可以通过调用 `File.walk(pathToRoot, depth)` 来限制想要访问的树的深度。两种 `walk` 方法都具有 `FileVisitOption...` 的可变长参数，但是你只能提供一种选项：`FOLLOW_LINKS`，即跟踪符号链接。



**注意：**如果要过滤 `walk` 返回的路径，并且你的过滤标准涉及与目录存储相关的文件属性，例如尺寸、创建时间和类型（文件、目录、符号链接），那么应该使用 `find` 方法来替代 `walk` 方法。可以用某个谓词函数来调用这个方法，该函数接受一个路径和一个 `BasicFileAttributes` 对象。这样做唯一的优势就是效率高。因为路径总是会被读入，所以这些属性很容易获取。

这段代码使用了 `Files.walk` 方法来将一个目录复制到另一个目录：

```
Files.walk(source).forEach(p ->
{
    try
    {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    }
    catch (IOException ex)
    {
        throw new UncheckedIOException(ex);
    }
});
}
```

遗憾的是，你无法很容易地使用 `Files.walk` 方法来删除目录树，因为你需要在删除父目录之前必须先删除子目录。下一节将展示如何克服此问题。

### 2.5.7 使用目录流

正如在前一节中所看到的，`Files.walk` 方法会产生一个可以遍历目录中所有子孙的 `Stream<Path>` 对象。有时，你需要对遍历过程进行更加细粒度的控制。在这种情况下，应该使用 `File.newDirectoryStream` 对象，它会产生一个 `DirectoryStream`。注意，它不是 `java.util.stream.Stream` 的子接口，而是专门用于目录遍历的接口。它是 `Iterable` 的子接口，因此你可以在增强的 `for` 循环中使用目录流。下面是其使用模式：

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
{
    for (Path entry : entries)
        Process entries
}
```

`try` 语句块用来确保目录流可以被正确关闭。访问目录中的项并没有具体的顺序。可以用 `glob` 模式来过滤文件：

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

表 2-4 展示了所有的 `glob` 模式。

表 2-4 Glob 模式

模式	描述	示例
*	匹配路径组成部分中 0 个或多个字符	*.java 匹配当前目录中的所有 Java 文件
**	匹配跨目录边界的 0 个或多个字符	**.java 匹配在所有子目录中的 Java 文件
?	匹配一个字符	????.java 匹配所有四个字符的 Java 文件(不包括扩展名)
[...]	匹配一个字符集合, 可以使用连线符 [0-9] 和取反符 [!0-9]	Test[0-9A-F].java 匹配 Testx.java, 其中 x 是一个十六进制数字
{...}	匹配由逗号隔开的多个可选项之一	*.{java,class} 匹配所有的 Java 文件和类 class 文件
\	转义上述任意模式中的字符以及 \ 字符	*\** 匹配所有文件名中包含 * 的文件

◆ 警告: 如果使用 Windows 的 glob 语法, 则必须对反斜杠转义两次: 一次为 glob 语法转义, 一次为 Java 字符串转义: `Files.newDirectoryStream(dir, "C:\\\\")`

如果想要访问某个目录的所有子孙成员, 可以转而调用 `walkFileTree` 方法, 并向其传递一个 `FileVisitor` 类型的对象, 这个对象会得到下列通知:

- 在遇到一个文件或目录时: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`
  - 在一个目录被处理前: `FileVisitResult preVisitDirectory(T dir, IOException ex)`
  - 在一个目录被处理后: `FileVisitResult postVisitDirectory(T dir, IOException ex)`
  - 在试图访问文件或目录时发生错误, 例如没有权限打开目录: `FileVisitResult visitFileFailed(path, IOException)`
- 对于上述每种情况, 都可以指定是否希望执行下面的操作:
- 继续访问下一个文件: `FileVisitResult.CONTINUE`
  - 继续访问, 但是不再访问这个目录下的任何项了: `FileVisitResult.SKIP_SUBTREE`
  - 继续访问, 但是不再访问这个文件的兄弟文(和该文件在同一个目录下的文件)了: `FileVisitResult.SKIP_SIBLINGS`
  - 终止访问: `FileVisitResult.TERMINATE`

当有任何方法抛出异常时, 就会终止访问, 而这个异常会从 `walkFileTree` 方法中抛出。

◆ 注意: `FileVisitor` 接口是泛化类型, 但是你也太可能会使用除 `FileVisitor<Path>` 之外的东西。`walkFileTree` 方法可以接受 `FileVisitor<? Super Path>` 类型的参数, 但是 `Path` 并没有多少超类型。

便捷类 `SimpleFileVisitor` 实现了 `FileVisitor` 接口, 但是其除 `visitFileFailed` 方法之外的所有方法并不做任何处理而是直接继续访问, 而 `visitFileFailed` 方法会抛出由失败导致的异常, 并进而终止访问。

例如, 下面的代码展示了如何打印出给定目录下的所有子目录:



```

Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes attrs)
        throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }

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

    public FileVisitResult visitFileFailed(Path path, IOException exc) throws IOException
    {
        return FileVisitResult.SKIP_SUBTREE;
    }
});

```

值得注意的是，我们需要覆盖 `postVisitDirectory` 方法和 `visitFileFailed` 方法，否则，访问会在遇到不允许打开的目录或不允许访问的文件时立即失败。

还应该注意的是，路径的众多属性是作为 `preVisitDirectory` 和 `visitFile` 方法的参数传递的。访问者不得不通过操作系统调用来获得这些属性，因为它需要区分文件和目录。因此，你就不需要再次执行系统调用了。

如果你需要在进入或离开一个目录时执行某些操作，那么 `FileVisitor` 接口的其他方法就显得非常有用了。例如，在删除目录树时，需要在移除当前目录的所有文件之后，才能移除该目录。下面是删除目录树的完整代码：

```

// Delete the directory tree starting at root
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult postVisitDirectory(Path dir, IOException e) throws IOException
    {
        if (e != null) throw e;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});

```

#### API java.nio.File.Files 7

- static DirectoryStream<Path> newDirectoryStream(Path path)
- static DirectoryStream<Path> newDirectoryStream(Path path, String glob)

获取给定目录中可以遍历所有文件和目录的迭代器。第二个方法只接受那些与给定的 glob 模式匹配的项。

- static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)

遍历给定路径的所有子孙，并将访问器应用于这些子孙之上。

#### java.nio.file.SimpleFileVisitor<T> 7

- static FileVisitResult visitFile(T path, BasicFileAttributes attrs) 在访问文件或目录时被调用，返回 CONTINUE、SKIP\_SUBTREE、SKIP\_SIBLINGS 和 TERMINATE 之一，默认实现是不做任何操作而继续访问。

- static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
- static FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)

在访问目录之前和之后被调用，默认实现是不做任何操作而继续访问。

- static FileVisitResult visitFileFailed(T path, IOException exc)

如果在试图获取给定文件的信息时抛出异常，则该方法被调用。默认实现是重新抛出异常，这会导致访问操作以这个异常而终止。如果你想自己访问，可以覆盖这个方法。

## 2.5.8 ZIP 文件系统

Paths 类会在默认文件系统中查找路径，即在用户本地磁盘中的文件。你也可以有别的文件系统，其中最有用的之一是 ZIP 文件系统。如果 `zipname` 是某个 ZIP 文件的名字，那么下面的调用

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
```

将建立一个文件系统，它包含 ZIP 文档中的所有文件。如果知道文件名，那么从 ZIP 文档中复制出这个文件就会变得很容易：

```
Files.copy(fs.getPath(sourceName), targetPath);
```

其中，`fs.getPath` 对于任意文件系统来说，都与 `Paths.get` 类似。

要列出 ZIP 文档中的所有文件，可以遍历文件树：

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
    {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

这比 2.3.3 节中描述的 API 要好用，它使用的是多个专门处理 ZIP 文档的新类。

**API** java.nio.file.FileSystems 7

- **static FileSystem newFileSystem(Path path, ClassLoader loader)**

对所安装的文件系统提供者进行迭代，并且如果 `loader` 不为 `null`，那么就还迭代给定的类加载器能够加载的文件系统，返回由第一个可以接受给定路径的文件系统提供者创建的文件系统。默认情况下，对于 ZIP 文件系统是有一个提供者的，它接受名字以 `.zip` 或 `.jar` 结尾的文件。

**API** java.nio.file.FileSystem 7

- **static Path getPath(String first, String... more)**

将给定的字符串连接起来创建一个路径。

## 2.6 内存映射文件

大多数操作系统都可以利用虚拟内存实现来将一个文件或者文件的一部分“映射”到内存中。然后，这个文件就可以当作是内存数组一样地访问，这比传统的文件操作要快得多。

### 2.6.1 内存映射文件的性能

在本节的末尾，你可以发现一个计算传统的文件输入和内存映射文件的 CRC32 校验和的程序。在同一台机器上，我们对 JDK 的 `jre/lib` 目录中的 37MB 的 `rt.jar` 文件用不同的方式来计算校验和，记录下来的时间数据如表 2-5 所示。

正如你所见，在这台特定的机器上，内存映射比使用带缓冲的顺序输入要稍微快一点，但是比使用 `RandomAccessFile` 快很多。

当然，精确的值因机器不同会产生很大的差异，但是很明显，与随机访问相比，性能提高总是很显著的。另一方面，对于中等尺寸文件的顺序读入则没有必要使用内存映射。

`java.nio` 包使内存映射变得十分简单，下面就是我们需要做的。

首先，从文件中获得一个通道（channel），通道是用于磁盘文件的一种抽象，它使我们可以访问诸如内存映射、文件加锁机制以及文件间快速数据传递等操作系统特性。

```
FileChannel channel = FileChannel.open(path, options);
```

然后，通过调用 `FileChannel` 类的 `map` 方法从这个通道中获得一个 `ByteBuffer`。你可以指定想要映射的文件区域与映射模式，支持的模式有三种：

- `FileChannel.MapMode.READ_ONLY`：所产生的缓冲区是只读的，任何对该缓冲区写入的尝试都会导致 `ReadOnlyBufferException` 异常。

表 2-5 文件操作的处理时间数据

方 法	时 间
普通输入流	110 秒
带缓冲的输入流	9.9 秒
随机访问文件	162 秒
内存映射文件	7.2 秒

- `FileChannel.MapMode.READ_WRITE`：所产生的缓冲区是可写的，任何修改都会在某个时刻写回到文件中。注意，其他映射同一个文件的程序可能不能立即看到这些修改，多个程序同时进行文件映射的确切行为是依赖于操作系统的。
- `FileChannel.MapMode.PRIVATE`：所产生的缓冲区是可写的，但是任何修改对这个缓冲区来说都是私有的，不会传播到文件中。

一旦有了缓冲区，就可以使用 `ByteBuffer` 类和 `Buffer` 超类的方法读写数据了。

缓冲区支持顺序和随机数据访问，它有一个可以通过 `get` 和 `put` 操作来移动的位置。例如，可以像下面这样顺序遍历缓冲区中的所有字节：

```
while (buffer.hasRemaining())
{
    byte b = buffer.get();
    ...
}
```

或者，像下面这样进行随机访问：

```
for (int i = 0; i < buffer.limit(); i++)
{
    byte b = buffer.get(i);
    ...
}
```

你可以用下面的方法来读写字节数组：

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

最后，还有下面的方法：

```
getInt
getLong
getShort
getChar
getFloat
getDouble
```

用来读入在文件中存储为二进制值的基本类型值。正如我们提到的，Java 对二进制数据使用高位在前的排序机制，但是，如果需要以低位在前的排序方式处理包含二进制数字的文件，那么只需调用

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

要查询缓冲区内当前的字节顺序，可以调用：

```
ByteOrder b = buffer.order()
```

**◆ 警告：**这一对方法没有使用 `set/get` 命名惯例。

要向缓冲区写数字，可以使用下列的方法：

```
putInt
putLong
```

```
putShort
putChar
putFloat
putDouble
```

在恰当的时机，以及当通道关闭时，会将这些修改写回到文件中。

程序清单 2-5 用于计算文件的 32 位的循环冗余校验和 (CRC32)，这个数值就是经常用来判断一个文件是否已损坏的校验和，因为文件损坏极有可能导致校验和改变。java.util.zip 包中包含一个 CRC32 类，可以使用下面的循环来计算一个字节序列的校验和：

```
CRC32 crc = new CRC32();
while (more bytes)
    crc.update(next byte)
long checksum = crc.getValue();
```

**■ 注意：**对 CRC 算法有一个很精细的解释，请查看 <http://www.relishsoft.com/Science/CrcMath.html>。

CRC 计算的细节并不重要，我们只是将它作为一个有用的文件操作的实例来使用。(在实践中，每次会以更大的工夫而不是一个字节为单位来读取和更新数据，而它们的速度差异并不明显。)

应该像下面这样运行程序：

```
java memoryMap.MemoryMapTest filename
```

程序清单 2-5 memoryMap/MemoryMapTest.java

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
7 import java.util.zip.*;
8
9 /**
10 * This program computes the CRC checksum of a file in four ways. <br>
11 * Usage: java memoryMap.MemoryMapTest filename
12 * @version 1.01 2012-05-30
13 * @author Cay Horstmann
14 */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename) throws IOException
18     {
19         try (InputStream in = Files.newInputStream(filename))
20         {
21             CRC32 crc = new CRC32();
22
23             int c;
24             while ((c = in.read()) != -1)
```



```

25         crc.update(c);
26     return crc.getValue();
27   }
28 }
29
30 public static long checksumBufferedInputStream(Path filename) throws IOException
31 {
32     try (InputStream in = new BufferedInputStream(Files.newInputStream(filename)))
33     {
34         CRC32 crc = new CRC32();
35
36         int c;
37         while ((c = in.read()) != -1)
38             crc.update(c);
39         return crc.getValue();
40     }
41 }
42
43 public static long checksumRandomAccessFile(Path filename) throws IOException
44 {
45     try (RandomAccessFile file = new RandomAccessFile(filename.toFile(), "r"))
46     {
47         long length = file.length();
48         CRC32 crc = new CRC32();
49
50         for (long p = 0; p < length; p++)
51         {
52             file.seek(p);
53             int c = file.readByte();
54             crc.update(c);
55         }
56         return crc.getValue();
57     }
58 }
59
60 public static long checksumMappedFile(Path filename) throws IOException
61 {
62     try (FileChannel channel = FileChannel.open(filename))
63     {
64         CRC32 crc = new CRC32();
65         int length = (int) channel.size();
66         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
67
68         for (int p = 0; p < length; p++)
69         {
70             int c = buffer.get(p);
71             crc.update(c);
72         }
73         return crc.getValue();
74     }
75 }
76
77 public static void main(String[] args) throws IOException
78 {

```



```

79 System.out.println("Input Stream:");
80 long start = System.currentTimeMillis();
81 Path filename = Paths.get(args[0]);
82 long crcValue = checksumInputStream(filename);
83 long end = System.currentTimeMillis();
84 System.out.println(Long.toHexString(crcValue));
85 System.out.println((end - start) + " milliseconds");
86
87 System.out.println("Buffered Input Stream:");
88 start = System.currentTimeMillis();
89 crcValue = checksumBufferedInputStream(filename);
90 end = System.currentTimeMillis();
91 System.out.println(Long.toHexString(crcValue));
92 System.out.println((end - start) + " milliseconds");
93
94 System.out.println("Random Access File:");
95 start = System.currentTimeMillis();
96 crcValue = checksumRandomAccessFile(filename);
97 end = System.currentTimeMillis();
98 System.out.println(Long.toHexString(crcValue));
99 System.out.println((end - start) + " milliseconds");
100
101 System.out.println("Mapped File:");
102 start = System.currentTimeMillis();
103 crcValue = checksumMappedFile(filename);
104 end = System.currentTimeMillis();
105 System.out.println(Long.toHexString(crcValue));
106 System.out.println((end - start) + " milliseconds");
107 }
108 }
```

**API** **java.io.FileInputStream 1.0**

- **FileChannel getChannel() 1.4**

返回用于访问这个输入流的通道。

**API** **java.io.FileOutputStream 1.0**

- **FileChannel getChannel() 1.4**

返回用于访问这个输出流的通道。

**API** **java.io.RandomAccessFile 1.0**

- **FileChannel getChannel() 1.4**

返回用于访问这个文件的通道。

**API** **java.nio.channels.FileChannel 1.4**

- **static FileChannel open(Path path, OpenOption... options) 7**

打开指定路径的文件通道，默认情况下，通道打开时用于读入。

- 参数: **path** 打开通道的文件所在的路径  
**options** StandardOpenOption 枚举中的 WRITE、APPEND、TRUNCATE\_EXISTING、CREATE 值
- **MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)**  
将文件的一个区域映射到内存中。  
参数: **mode** FileChannel.MapMode 类中的常量 READ\_ONLY、READ\_WRITE、或 PRIVATE 之一  
**position** 映射区域的起始位置  
**size** 映射区域的大小

**API** java.nio.Buffer 1.4

- **boolean hasRemaining()**

如果当前的缓冲区位置没有到达这个缓冲区的界限位置，则返回 true。

- **int limit()**

返回这个缓冲区的界限位置，即没有任何值可用的第一个位置。

**API** java.nio.ByteBuffer 1.4

- **byte get()**

从当前位置获得一个字节，并将当前位置移动到下一个字节。

- **byte get(int index)**

从指定索引处获得一个字节。

- **ByteBuffer put(byte b)**

向当前位置推入一个字节，并将当前位置移动到下一个字节。返回对这个缓冲区的引用。

- **ByteBuffer put(int index, byte b)**

向指定索引处推入一个字节。返回对这个缓冲区的引用。

- **ByteBuffer get(byte[] destination)**

- **ByteBuffer get(byte[] destination, int offset, int length)**

用缓冲区中的字节来填充字节数组，或者字节数组的某个区域，并将当前位置向前移动读入的字节数个位置。如果缓冲区不够大，那么就不会读入任何字节，并抛出 BufferUnderflow Exception。返回对这个缓冲区的引用。

参数: **destination** 要填充的字节数组

**offset** 要填充区域的偏移量

**length** 要填充区域的长度

- **ByteBuffer put(byte[] source)**

- `ByteBuffer put(byte[] source, int offset, int length)`

将字节数组中的所有字节或者给定区域的字节都推入缓冲区中，并将当前位置向前移动写出的字节数个位置。如果缓冲区不够大，那么就不会读入任何字节，并抛出 `BufferUnderflowException`。返回对这个缓冲区的引用。

参数：`source` 要写出的数组

`offset` 要写出区域的偏移量

`length` 要写出区域的长度

- `Xxx getXxx()`

- `Xxx getXxx(int index)`

- `ByteBuffer putXxx(Xxx value)`

- `ByteBuffer putXxx(int index, Xxx value)`

获得或放置一个二进制数。`Xxx` 是 `Int`、`Long`、`Short`、`Char`、`Float` 或 `Double` 中的一个。

- `ByteBuffer order(ByteOrder order)`

- `ByteOrder order()`

设置或获得字节顺序，`order` 的值是 `ByteOrder` 类的常量 `BIG_ENDIAN` 或 `LITTLE_ENDIAN` 中的一个。

- `static ByteBuffer allocate(int capacity)`

构建具有给定容量的缓冲区。

- `static ByteBuffer wrap(byte[] values)`

构建具有指定容量的缓冲区，该缓冲区是对给定数组的包装。

- `CharBuffer asCharBuffer()`

构建字符缓冲区，它是对这个缓冲区的包装。对该字符缓冲区的变更将在这个缓冲区中反映出来，但是该字符缓冲区有自己的位置、界限和标记。

**API**

`java.nio.CharBuffer 1.4`

- `char get()`

- `CharBuffer get(char[] destination)`

- `CharBuffer get(char[] destination, int offset, int length)`

从这个缓冲区的当前位置开始，获取一个 `char` 值，或者一个范围内的所有 `char` 值，然后将位置向前移动越过所有读入的字符。最后两个方法将返回 `this`。

- `CharBuffer put(char c)`

- `CharBuffer put(char[] source)`

- `CharBuffer put(char[] source, int offset, int length)`

- `CharBuffer put(String source)`

- `CharBuffer put(CharBuffer source)`



从这个缓冲区的当前位置开始，放置一个 `char` 值，或者一个范围内的所有 `char` 值，然后将位置向前移动越过所有被写出的字符。当放置的值是从 `CharBuffer` 读入时，将读入所有剩余字符。所有方法将返回 `this`。

## 2.6.2 缓冲区数据结构

在使用内存映射时，我们创建了单一的缓冲区横跨整个文件或我们感兴趣的文件区域。我们还可以使用更多的缓冲区来读写大小适度的信息块。

本节将简要地介绍 `Buffer` 对象上的基本操作。缓冲区是由具有相同类型的数值构成的数组，`Buffer` 类是一个抽象类，它有众多的具体子类，包括 `ByteBuffer`、`CharBuffer`、`DoubleBuffer`、`IntBuffer`、`LongBuffer` 和 `ShortBuffer`。

**注意：**`StringBuffer` 类与这些缓冲区没有关系。

在实践中，最常用的将是 `ByteBuffer` 和 `CharBuffer`。如图 2-10 所示，每个缓冲区都具有：

- 一个容量，它永远不能改变。
- 一个读写位置，下一个值将在此进行读写。
- 一个界限，超过它进行读写是没有意义的。
- 一个可选的标记，用于重复一个读入或写出操作。

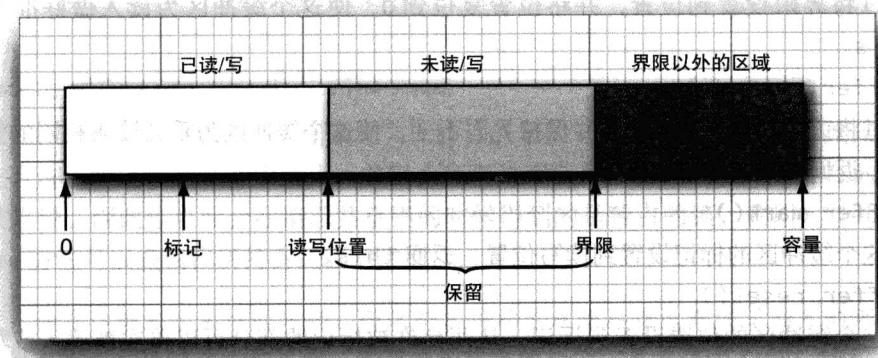


图 2-10 一个缓冲区

这些值满足下面的条件：

$$0 \leq \text{标记} \leq \text{位置} \leq \text{界限} \leq \text{容量}$$

使用缓冲区的主要目的是执行“写，然后读入”循环。假设我们有一个缓冲区，在一开始，它的位置为 0，界限等于容量。我们不断地调用 `put` 将值添加到这个缓冲区中，当我们耗尽所有的数据或者写出的数据量达到容量大小时，就该切换到读入操作了。

这时调用 `flip` 方法将界限设置到当前位置，并把位置复位到 0。现在在 `remaining` 方

法返回正数时（它返回的值是“界限 - 位置”），不断地调用 `get`。在我们将缓冲区中所有的值都读入之后，调用 `clear` 使缓冲区为下一次写循环做好准备。`clear` 方法将位置复位到 0，并将界限复位到容量。

如果你想重读缓冲区，可以使用 `rewind` 或 `mark/reset` 方法，详细内容请查看 API 注释。

要获取缓冲区，可以调用诸如 `ByteBuffer.allocate` 或 `ByteBuffer.wrap` 这样的静态方法。

然后，可以用来自某个通道的数据填充缓冲区，或者将缓冲区的内容写出通道中。例如：

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

这是一种非常有用的方法，可以替代随机访问文件。

#### java.nio.Buffer 1.4

- **Buffer clear()**

通过将位置复位到 0，并将界限设置到容量，使这个缓冲区为写出做好准备。返回 `this`。

- **Buffer flip()**

通过将界限设置到位置，并将位置复位到 0，使这个缓冲区为读入做好准备。返回 `this`。

- **Buffer rewind()**

通过将读写位置复位到 0，并保持界限不变，使这个缓冲区为重新读入相同的值做好准备。返回 `this`。

- **Buffer mark()**

将这个缓冲区的标记设置到读写位置，返回 `this`。

- **Buffer reset()**

将这个缓冲区的位置设置到标记，从而允许被标记的部分可以再次被读入或写出，返回 `this`。

- **int remaining()**

返回剩余可读入或可写出的值的数量，即界限与位置之间的差异。

- **int position()**

- **void position(int newValue)**

返回这个缓冲区的位置。

- **int capacity()**

返回这个缓冲区的容量。

### 2.6.3 文件加锁机制

考虑一下多个同时执行的程序需要修改同一个文件的情形，很明显，这些程序需要以某种方式进行通信，不然这个文件很容易被损坏。文件锁可以解决这个问题，它可以控制对文件或文件中某个范围的字节的访问。

假设你的应用程序将用户的偏好存储在一个配置文件中，当用户调用这个应用的两个实例时，这两个实例就有可能会同时希望写这个配置文件。在这种情况下，第一个实例应该锁定这个文件，当第二个实例发现这个文件被锁定时，它必须决策是等待直至这个文件解锁，还是直接跳过这个写操作过程。

要锁定一个文件，可以调用 `FileChannel` 类的 `lock` 或 `tryLock` 方法：

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

或

```
FileLock lock = channel.tryLock();
```

第一个调用会阻塞直至可获得锁，而第二个调用将立即返回，要么返回锁，要么在锁不可获得的情况下返回 `null`。这个文件将保持锁定状态，直至这个通道关闭，或者在锁上调用了 `release` 方法。

你还可以通过下面的调用锁定文件的一部分：

```
FileLock lock(long start, long size, boolean shared)
```

或

```
FileLock tryLock(long start, long size, boolean shared)
```

如果 `shared` 标志为 `false`，则锁定文件的目的是读写，而如果为 `true`，则这是一个共享锁，它允许多个进程从文件中读入，并阻止任何进程获得独占的锁。并非所有的操作系统都支持共享锁，因此你可能会在请求共享锁的时候得到的是独占的锁。调用 `FileLock` 类的 `isShared` 方法可以查询你所持有的锁的类型。

**■ 注意：**如果你锁定了文件的尾部，而这个文件的长度随后增长超过了锁定的部分，那么增长出来的额外区域是未锁定的，要想锁定所有的字节，可以使用 `Long.MAX_VALUE` 来表示尺寸。

要确保在操作完成时释放锁，与往常一样，最好在一个 `try` 语句中执行释放锁的操作：

```
try (FileLock lock = channel.lock())
{
    access the locked file or segment
}
```

请记住，文件加锁机制是依赖于操作系统的，下面是需要注意的几点：

- 在某些系统中，文件加锁仅仅是建议性的，如果一个应用未能得到锁，它仍旧可以向被另一个应用并发锁定的文件执行写操作。

- 在某些系统中，不能在锁定一个文件的同时将其映射到内存中。
- 文件锁是由整个 Java 虚拟机持有的。如果有两个程序是由同一个虚拟机启动的（例如 Applet 和应用程序启动器），那么它们不可能每一个都获得一个在同一个文件上的锁。当调用 `lock` 和 `tryLock` 方法时，如果虚拟机已经在同一个文件上持有了另一个重叠的锁，那么这两个方法将抛出 `OverlappingFileLockException`。
- 在一些系统中，关闭一个通道会释放由 Java 虚拟机持有的底层文件上的所有锁。因此，在同一个锁定文件上应避免使用多个通道。
- 在网络文件系统上锁定文件是高度依赖于系统的，因此应该尽量避免。

#### java.nio.channels.FileChannel 1.4

- `FileLock lock()`

在整个文件上获得一个独占的锁，这个方法将阻塞直至获得锁。

- `FileLock tryLock()`

在整个文件上获得一个独占的锁，或者在无法获得锁的情况下返回 `null`。

- `FileLock lock(long position, long size, boolean shared)`

- `FileLock tryLock(long position, long size, boolean shared)`

在文件的一个区域上获得锁。第一个方法将阻塞直至获得锁，而第二个方法将在无法获得锁时返回 `null`。

参数: `position`

要锁定区域的起始位置

`size`

要锁定区域的尺寸

`shared`

`true` 为共享锁, `false` 为独占锁

#### java.nio.channels.FileLock 1.4

- `void close() 1.7`

释放这个锁。

## 2.7 正则表达式

正则表达式 (regular expression) 用于指定字符串的模式，你可以在任何需要定位匹配某种特定模式的字符串的情况下使用正则表达式。例如，我们有一个示例程序就是用来定位 HTML 文件中的所有超链接的，它是通过查找 `<a href = "...>` 模式的字符串来实现此目的的。

当然，在指定模式时，`...` 标记法并不够精确。你需要精确地指定什么样的字符序列才是合法的匹配，这就要求无论何时，当你要描述一个模式时，都需要使用某种特定的语法。

下面是一个简单的示例，正则表达式

`[Jj]ava.+`

匹配下列形式的所有字符串：

- 第一个字母是 J 或 j。
- 接下来的三个字母是 ava。
- 字符串的其余部分由一个或多个任意的字符构成。

例如，字符串 “javanese” 就匹配这个特定的正则表达式，但是字符串 “core java” 就不匹配。

正如你所见，你需要了解一点这种语法，以理解正则表达式的含义。幸运的是，对于大多数情况，一小部分很直观的语法结构就足够用了。

- 字符类 (character class) 是一个括在括号中的可选择的字符集，例如，[Jj]、[0-9]、[A-Za-z] 或 [^0-9]。这里 “-” 表示是一个范围（所有 Unicode 值落在两个边界范围之内的字符），而 ^ 表示补集（除了指定字符之外的所有字符）。
- 如果字符类中包含 “-”，那么它必须是第一项或最后一项；如果要包含 “[”，那么它必须是第一项；如果要包含 “^”，那么它可以是除开始位置之外的任何位置。其中，你只需要转义 “[” 和 “\”。
- 有许多预定的字符类，例如 \d (数字) 和 \p{Sc} (Unicode 货币符号)。请查看表 2-6 和表 2-7。

表 2-6 正则表达式语法

表达式	描述	示例
<b>字符</b>		
c, 除 .*+?{ () [\^\$] 之外	字符 c	]
	任何除行终止符之外的字符，或者在 DOTALL 标志被设置时表示任何字符	
\x{p}	十六进制码为 p 的 Unicode 码点	\x{1D546}
\uhhhh, \xhh, \0o, \ooo, \oooo	具有给定十六进制或八进制值的码元	\uFEFF
\a, \e, \f, \n, \r, \t	响铃符 (\x{7})、转义符 (\x{18})、换页符 (\x{8})、换行符 (\x{A})、回车符 (\x{D})、指标符 (\x{9})	\n
\cc, 其中 c 在 [A,Z] 的范围内，或者是 @[\]^_? 之一	对应于字符 c 的控制字符	\cH 是退格符 (\x{8})
\c, 其中 c 不在 [A-Za-z0-9] 的范围内	字符 c	\\"
\Q... \E	在左引号和右引号之间的所有字符	\Q(...) \E 匹配字符串 (...)
<b>字符类</b>		
[C <sub>1</sub> C <sub>2</sub> ...], 其中 C <sub>i</sub> 是多个字符，范围从 c-d，或者是字符类	任何由 C <sub>1</sub> , C <sub>2</sub> , ... 表示的字符	[0-9+-]
[^...]	某个字符类的补集	[^\d\s]
[...&&...]	字符集的交集	[^\p{L}&&[^A-Za-z]]
\p{...}, \P{...}	某个预定义字符类 (参阅表 2-7)；它的补集	\p{L} 匹配一个 Unicode 字母，而 \p{L} 也匹配这个字母，可以忽略单个字母情况下的括号

(续)

表达式	描述	示例
\d, \D	数字([0-9]，或者在 <b>UNICODE_CHARACTER_CLASS</b> 标志被设置时表示 \p{Digit})；它的补集	\d+ 是一个数字序列
\w, \W	单词字符([a-zA-Z0-9]，或者在 <b>UNICODE_CHARACTER_CLASS</b> 标志被设置时表示 Unicode 单词字符)；它的补集	
\s, \S	空格([\v\r\t\f\x{8}])，或者在 <b>UNICODE_CHARACTER_CLASS</b> 标志被设置时表示 \p{IsWhite_Space})；它的补集	\s*, \s* 是由可选的空格字符包围的逗号
\h, \v, \H, \V	水平空白字符、垂直空白字符，它们的补集	
<b>序列和选择</b>		
XY	任何 X 中的字符串，后面跟随任何 Y 中的字符串	[1-9][0-9]* 表示没有前导零的正整数
X Y	任何 X 或 Y 中的字符串	
<b>群组</b>		
(X)	捕获 X 的匹配	'([^\"]*)' 捕获的是被引用的文本
\n	第 n 组	(["\"]).*\1 可以匹配 'Fred' 和 "Fred"，但是不能匹配 "Fred"
(?<name>X)	捕获与给定名字匹配的 X	'(?<id>[A-Za-z0-9]+)' 可以捕获名字为 id 的匹配
\k<name>	具有给定名字的组	\k<id> 可以匹配名字为 id 的组
(?:X)	使用括号但是不捕获 X	在(?:http ftp)://(.*) 中，在 :// 之后的匹配是 \1
(?f <sub>1</sub> f <sub>2</sub> ...:X) (?f <sub>1</sub> ...-f <sub>k</sub> :X)，其中 f <sub>i</sub> 在 [dimsuUx] 的范围内	匹配但是不捕获给定标志开或关(在 - 之后)的 X	(?i:jpe?g) 是大小写不敏感的匹配
其他(?)	请参阅 Pattern API 文档	
<b>量词</b>		
X?	可选 X	\+? 是可选的 + 号
X*, X+	0 或多个 X, 1 或多个 X	[1-9][0-9]+ 是大于 10 的整数
X{n}, X{n,}, X{m, n}	n 个 X, 至少 n 个 X, m 到 n 个 X	[0-7]{1,3} 是一位到三位的八进制数
Q?, 其中 Q 是一个量词表达式	勉强量词，在尝试最长匹配之前先尝试最短匹配	.*(<.+?>).* 捕获尖括号括起来的最短序列
Q+, 其中 Q 是一个量词表达式	占有量词，在不回溯的情况下获取最长匹配	'[^']*+' 匹配单引号引起的字符串，并且在字符串中没有右单引号的情况下立即匹配失败

(续)

表达式	描述	示例
<b>边界匹配</b>		
<code>^, \$</code>	输入的开头和结尾(或者多行模式中的开头和结尾行)	<code>^Java\$</code> 匹配输入中的 Java 或 Java 构成的行
<code>\A, \Z, \z</code>	输入的开头, 输入的结尾、输入的绝对结尾(在多行模式中不会发生变化)	
<code>\b, \B</code>	单词边界, 非单词边界	<code>\bJava\b</code> 匹配单词 Java
<code>\R</code>	Unicode 行分隔符	
<code>\G</code>	前一个匹配的结尾	

表 2-7 与 \p 一起使用的预定义字符类名字

字符类名字	解 释
<code>posixClass</code>	<code>posixClass</code> 是 Lower、Upper、Alpha、Digit、Alnum、Punct、Graph、Print、Cntrl、XDigit、Space、Blank、ASCII 之一, 它会依 UNICODE_CHARACTER_CLASS 标志的值而被解释为 POSIX 或 Unicode 类
<code>IsScript, sc=Script,</code> <code>script=Script</code>	<code>Character.UnicodeScript.forName</code> 可以接受的脚本
<code>InBlock, blk=Block,</code> <code>block=Block</code>	<code>Character.UnicodeScript.forName</code> 可以接受的块
<code>Category, InCategory,</code> <code>gc=Category, general_category=Category</code>	Unicode 通用分类的单字母或双字母名字
<code>IsProperty</code>	<code>Property</code> 是 Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned 之一
<code>javaMethod</code>	调用 <code>Character.isMethod</code> 方法(必须不是过时的方法)

- 大部分字符都可以与它们自身匹配, 例如在前面示例中的 `ava` 字符。
- 符号可以匹配任何字符(有可能不包括行终止符, 这取决于标志的设置)。
- 使用 \ 作为转义字符, 例如, `\.` 匹配句号而 `\\"` 匹配反斜线。
- `^` 和 `$` 分别匹配一行的开头和结尾。
- 如果 X 和 Y 是正则表达式, 那么 XY 表示“任何 X 的匹配后面跟随 Y 的匹配”, `X | Y` 表示“任何 X 或 Y 的匹配”。
- 你可以将量词运用到表达式 X: `X+(1个或多个)`、`X*(0个或多个)` 与 `X? (0个或1个)`。
- 默认情况下, 量词要匹配能够使整个匹配成功的最大可能的重复次数。你可以修改这种行为, 方法是使用后缀 ? (使用勉强或吝啬匹配, 也就是匹配最小的重复次数) 或使用后缀 + (使用占有或贪婪匹配, 也就是即使让整个匹配失败, 也要匹配最大的重复次数)。

例如，字符串 `cab` 匹配 `[a-z]*ab`，但是不匹配 `[a-z]*+ab`。在第一种情况中，表达式 `[a-z]*` 只匹配字符 `c`，使得字符 `ab` 匹配该模式的剩余部分；但是贪婪版本 `[a-z]*+` 将匹配字符 `cab`，模式的剩余部分将无法匹配。

- 我们使用群组来定义子表达式，其中群组用括号 `()` 括起来。例如，`([+-]?)([0-9]+)`。

然后你可以询问模式匹配器，让其返回每个组的匹配，或者用 `\n` 来引用某个群组，其中 `n` 是群组号（从 `\1` 开始）。

例如，下面是一个有些复杂但是却可能很有用的正则表达式，它描述了十进制和十六进制整数：

```
[+-]?[0-9]+|[0[Xx][0-9A-Fa-f]+
```

遗憾的是，在使用正则表达式的各种程序和类库之间，表达式语法并未完全标准化。尽管在基本结构上达成了一致，但是它们在细节上仍旧存在着许多令人抓狂的差异。Java 正则表达式类使用的语法与 Perl 语言使用的语法十分相似，但是并不完全一样。表 2-6 展示的是 Java 语法中的所有结构。关于正则表达式语法的更多信息，可以求教于 `Pattern` 类的 API 文档和 Jeffrey E. F. Friedl 的《Mastering Regular Expressions》(O'Reilly and Associates, 2006)。

正则表达式的最简单用法就是测试某个特定的字符串是否与它匹配。下面展示了如何用 Java 来编写这种测试，首先用表示正则表达式的字符串构建一个 `Pattern` 对象。然后从这个模式中获得一个 `Matcher`，并调用它的 `matches` 方法：

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

这个匹配器的输入可以是任何实现了 `CharSequence` 接口的类的对象，例如 `String`、`StringBuilder` 和 `CharBuffer`。

在编译这个模式时，你可以设置一个或多个标志，例如：

```
Pattern pattern = Pattern.compile(expression,
    Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

或者可以在模式中指定它们：

```
String regex = "(?iU:expression)";
```

下面是各个标志。

- `Pattern.CASE_INSENSITIVE` 或 `r`：匹配字符时忽略字母的大小写，默认情况下，这个标志只考虑 US ASCII 字符。
- `Pattern.UNICODE_CASE` 或 `u`：当与 `CASE_INSENSITIVE` 组合使用时，用 Unicode 字母的大小写来匹配。
- `Pattern.UNICODE_CHARACTER_CLASS` 或 `U`：选择 Unicode 字符类代替 POSIX，其中蕴含了 `UNICODE_CASE`。
- `Pattern.MULTILINE` 或 `m`：`^` 和 `$` 匹配行的开头和结尾，而不是整个输入的开头和结尾。



- **Pattern.UNIX\_LINES** 或 **d**：在多行模式中匹配 ^ 和 \$ 时，只有 '\n' 被识别成行终止符。
- **Pattern.DOTALL** 或 **s**：当使用这个标志时，. 符号匹配所有字符，包括行终止符。
- **Pattern.COMMENTS** 或 **x**：空白字符和注释（从 # 到行末尾）将被忽略。
- **Pattern.LITERAL**：该模式将被逐字地采纳，必须精确匹配，因字母大小写而造成的差异除外。
- **Pattern.CANON\_EQ**：考虑 Unicode 字符规范的等价性，例如，u 后面跟随 “（分音符号）匹配 ü。

最后两个标志不能在正则表达式内部指定。

如果想要在集合或流中匹配元素，那么可以将模式转换为谓词：

```
Stream<String> strings = . . .;
Stream<String> result = strings.filter(pattern.asPredicate());
```

其结果中包含了匹配正则表达式的所有字符串。

如果正则表达式包含群组，那么 **Matcher** 对象可以揭示群组的边界。下面的方法

```
int start(int groupIndex)
int end(int groupIndex)
```

将产生指定群组的开始索引和结束之后的索引。

可以直接通过调用下面的方法抽取匹配的字符串：

```
String group(int groupIndex)
```

群组 0 是整个输入，而用于第一个实际群组的群组索引是 1。调用 **groupCount** 方法可以获得全部群组的数量。对于具名的组，使用下面的方法

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

嵌套群组是按照前括号排序的，例如，假设我们有下面的模式

```
(([1-9]|1[0-2]):([0-5][0-9]))[ap]m
```

和下面的输出

11:59am

那么，匹配器会报告下面的群组：

群组索引	开始	结束	字符串
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

程序清单 2-6 的程序提示输入一个模式，然后提示输入用于匹配的字符串，随后将打印出输入是否与模式相匹配。如果输入匹配模式，并且模式包含群组，那么这个程序将用括号



打印出群组边界，例如

((11):(59))am

### 程序清单 2-6 regex/RegexTest.java

```

1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7 * This program tests regular expression matching. Enter a pattern and strings to match,
8 * or hit Cancel to exit. If the pattern contains groups, the group boundaries are displayed
9 * in the match.
10 * @version 1.02 2012-06-02
11 * @author Cay Horstmann
12 */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         Scanner in = new Scanner(System.in);
18         System.out.println("Enter pattern: ");
19         String patternString = in.nextLine();
20
21         Pattern pattern = Pattern.compile(patternString);
22
23         while (true)
24         {
25             System.out.println("Enter string to match: ");
26             String input = in.nextLine();
27             if (input == null || input.equals("")) return;
28             Matcher matcher = pattern.matcher(input);
29             if (matcher.matches())
30             {
31                 System.out.println("Match");
32                 int g = matcher.groupCount();
33                 if (g > 0)
34                 {
35                     for (int i = 0; i < input.length(); i++)
36                     {
37                         // Print any empty groups
38                         for (int j = 1; j <= g; j++)
39                             if (i == matcher.start(j) && i == matcher.end(j))
40                                 System.out.print("()");
41                         // Print ( for non-empty groups starting here
42                         for (int j = 1; j <= g; j++)
43                             if (i == matcher.start(j) && i != matcher.end(j))
44                                 System.out.print('(');
45                         System.out.print(input.charAt(i));
46                         // Print ) for non-empty groups ending here
47                         for (int j = 1; j <= g; j++)
48                             if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
49                                 System.out.print(")");
50
51             }
52         }
53     }
54 }
```



```

49         System.out.print(')');
50     }
51     System.out.println();
52   }
53 } else
54   System.out.println("No match");
55 }
56 }
57 }
58 }

```

通常，你不希望用正则表达式来匹配全部输入，而只是想找出输入中一个或多个匹配的子字符串。这时可以使用 **Matcher** 类的 **find** 方法来查找匹配内容，如果返回 **true**，再使用 **start** 和 **end** 方法来查找匹配的内容，或使用不带引元的 **group** 方法来获取匹配的字符串。

```

while (matcher.find())
{
    int start = matcher.start();
    int end = matcher.end();
    String match = input.group();
    ...
}

```

程序清单 2-7 对这种机制进行了应用，它定位一个 Web 页面上的所有超文本引用，并打印它们。为了运行这个程序，你需要在命令行中提供一个 URL，例如

```
java match.HrefMatch http://horstmann.com
```

### 程序清单 2-7 match/HrefMatch.java

```

1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
8 /**
9  * This program displays all URLs in a web page by matching a regular expression that describes
10 * the <a href=...> HTML tag. Start the program as <br>
11 * java match.HrefMatch URL
12 * @version 1.02 2016-07-14
13 * @author Cay Horstmann
14 */
15 public class HrefMatch
16 {
17     public static void main(String[] args)
18     {
19         try
20         {
21             // get URL string from command line or use default

```



```

22     String urlString;
23     if (args.length > 0) urlString = args[0];
24     else urlString = "http://java.sun.com";
25
26     // open reader for URL
27     InputStreamReader in = new InputStreamReader(new URL(urlString).openStream(),
28             StandardCharsets.UTF_8);
29
30     // read contents into string builder
31     StringBuilder input = new StringBuilder();
32     int ch;
33     while ((ch = in.read()) != -1)
34         input.append((char) ch);
35
36     // search for all occurrences of pattern
37     String patternString = "<a\\>\\s+href\\s*=\\s*(\"[^\"\\n]*|[\\n\\r\\t]*\")\\s*";
38     Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
39     Matcher matcher = pattern.matcher(input);
40
41     while (matcher.find())
42     {
43         String match = matcher.group();
44         System.out.println(match);
45     }
46 }
47 catch (IOException | PatternSyntaxException e)
48 {
49     e.printStackTrace();
50 }
51 }
52 }

```

Matcher 类的 `replaceAll` 方法将正则表达式出现的所有地方都用替换字符串来替换。例如，下面的指令将所有的数字序列都替换成 # 字符。

```

Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");

```

替换字符串可以包含对模式中群组的引用：\$n 表示替换成第 n 个群组，\${name} 被替换成具有给定名字的组，因此我们需要用 \\$ 来表示在替换文本中包含一个 \$ 字符。

如果字符串中包含 \$ 和 \，但是又不希望它们被解释成群组的替换符，那么就可以调用 `matcher.replaceFirst(Matcher.quoteReplacement(str))`。

`replaceFirst` 方法将只替换模式的第一次出现。

最后，`Pattern` 类有一个 `split` 方法，它可以用正则表达式来匹配边界，从而将输入分割成字符串数组。例如，下面的指令可以将输入分割成标记，其中分隔符是由可选的空白字符包围的标点符号。

```

Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);

```

如果有多个标记，那么可以惰性地获取它们：

```
Stream<String> tokens = commas.splitAsStream(input);
```

如果不关心预编译模式和惰性获取，那么可以使用 `String.split` 方法：

```
String[] tokens = input.split("\s*,\s*");
```

#### **API** java.util.regex.Pattern 1.4

- `static Pattern compile(String expression)`
- `static Pattern compile(String expression, int flags)`

把正则表达式字符串编译到一个用于快速处理匹配的模式对象中。

参数: `expression` 正则表达式

`flags`      `CASE_INSENSITIVE`、`UNICODE_CASE`、`MULTILINE`、  
                       `UNIX_LINES`、`DOTALL` 和 `CANON_EQ` 标志中的一个

- `Matcher matcher(CharSequence input)`

返回一个 `Matcher` 对象，你可以用它在输入中定位模式的匹配。

- `String[] split(CharSequence input)`

- `String[] split(CharSequence input, int limit)`

- `Stream<String> splitAsStream(CharSequence input)` 8

将输入分割成标记，其中模式指定了分隔符的形式。返回标记数组，分隔符并非标记的一部分。

参数: `input` 要分割成标记的字符串

`limit` 所产生的字符串的最大数量。如果已经发现了 `limit-1` 个匹配的分隔符，那么返回的数组中的最后一项就包含所有剩余未分割的输入。如果 `limit <= 0`，那么整个输入都被分割；如果 `limit` 为 0，那么坠尾的空字符串将不会置于返回的数组中。

#### **API** java.util.regex.Matcher 1.4

- `boolean matches()`

如果输入匹配模式，则返回 `true`。

- `boolean lookingAt()`

如果输入的开头匹配模式，则返回 `true`。

- `boolean find()`

- `boolean find(int start)`

尝试查找下一个匹配，如果找到了另一个匹配，则返回 `true`。

参数: `start` 开始查找的索引位置

- `int start()`



- **int end()**

返回当前匹配的开始索引和结尾之后的索引位置。

- **String group()**

返回当前的匹配。

- **int groupCount()**

返回输入模式中的群组数量。

- **int start(int groupIndex)**

- **int end(int groupIndex)**

返回当前匹配中给定群组的开始和结尾之后的位置。

参数: **groupIndex** 群组索引 (从 1 开始), 或者表示整个匹配的 0

- **String group(int groupIndex)**

返回匹配给定群组的字符串。

参数: **groupIndex** 群组索引 (从 1 开始), 或者表示整个匹配的 0

- **String replaceAll(String replacement)**

- **String replaceFirst(String replacement)**

返回从匹配器输入获得的通过将所有匹配或第一个匹配用替换字符串替换之后的字符串。

参数: **replacement** 替换字符串, 它可以包含用 \$n 表示的对群组的引用, 这时需要用 \\$ 来表示字符串中包含一个 \$ 符号

- **static String quoteReplacement(String str) 5.0**

引用 str 中的所有 \ 和 \$。

- **Matcher reset()**

- **Matcher reset(CharSequence input)**

复位匹配器的状态。第二个方法将使匹配器作用于另一个不同的输入。这两个方法都返回 this。

你现在已经看到了在 Java 中输入输出操作是如何实现的, 也对正则表达式有了概略的了解。在下一章中, 我们将转而研究对 XML 数据的处理。