16 代码校验

> 你永远无法保证自己的代码是正确的。
> 你只能证明自己的代码存在问题[①]。

让我们先暂停学习语言特性，来了解一些编程的基础知识吧。具体来说，就是如何确保你的代码能正常工作。

16.1 测试

如果没有经过测试，代码就不可能正常工作。

Java（大体上[②]）是一门静态类型语言，程序员通常对这种语言提供的显式安全性过于放心，认为“如果编译器没有提示错误，那就没问题”。但是静态类型检查是一种非常有限的测试类型，它仅仅意味着编译器接受代码的语法和基本类型规则，并不意味着代码满足了程序的目标。当你获得更多编程经验后，会了解到自己的代码几乎永远无法满足这些目

① 你无法证明自己的代码没有问题。——译者注
② 我说“大体上”是因为可以编写出编译器无法检查的代码，如第 19 章中所示。

标。代码验证的第一步就是创建测试，来检查代码行为是否满足你的目标。

16.1.1 单元测试

这是一个将集成测试构建到你创建的所有代码里的过程，并在每次构建系统时运行这些测试。这样，构建过程不仅可以检查语法错误，还可以检查语义错误。

这里的“单元”表明测试的是一小段代码。通常，每个类都有测试，检查它所有方法的行为。而“系统”测试则不同，它检查已完成的程序是否满足了最终要求。

C 风格的编程语言，尤其是 C++，传统上更重视性能而不是编程安全。用 Java 开发程序比用 C++ 快得多（大多数情况下前者速度是后者的两倍），原因在于 Java 的安全网，即垃圾收集和改进的类型检查等功能。通过将单元测试集成到构建过程中，你可以扩展这个安全网，从而加快开发速度。当发现了设计或实现缺陷时，你还可以更轻松、更大胆地重构代码，并且通常还可以更快地推出更好的产品。

为了保证代码的正确性，我需要自动提取本书中的每个程序，然后使用适当的构建系统进行编译。当意识到这一点时，我就开启了自己的测试旅程。本书使用的构建系统是 Gradle，安装 JDK 后，输入 `gradlew compileJava` 即可编译本书的所有代码。自动提取和编译对本书代码质量的影响是如此直接和重大，它很快成为（在我看来）任何编程书的必要条件——你怎么能相信自己没有编译过的代码？我还发现可以使用搜索和替换对整本书进行彻底的更改。我知道如果自己引入了一个缺陷，代码提取器和构建系统会将其清除。

随着程序变得越来越复杂，我发现自己的系统里有一个严重的漏洞。编译程序显然是重要的第一步，而对于一本已出版的书来说，这似乎还是一个相当革命性的步骤（由于出版计划带来的压力，打开一本编程书后，经常可以发现代码缺陷）。但是，我收到了来自读者的消息，报告我的代码中存在的语义问题，而这些问题只有通过运行代码才能发现。虽然我一直知道自己的构建过程有问题，以后肯定会以令人尴尬的错误报告的形式让我难堪，但屈服于出版计划，我只是初步采取了一些不太有效的步骤，实现了一个系统来自动执行测试。

我还经常因为没有显示足够的程序输出而遭受抱怨。我需要验证程序的输出，同时在书中显示经过验证的输出。我以前的态度是，读者应该在阅读本书的同时运行程序，许多读者正是这样做的，并从中受益。然而，这种态度的一个隐藏原因是，我没有办法证明书中显示的输出是正确的。根据经验，我知道随着时间的推移会发生一些事情，导致输出不再正确（或者，一开始我的输出就是不正确的）。为了解决这个问题，我用 Python 语言创建了一个工具（你可以在下载的示例中找到这个工具）。本书中的大多数程序会产生控制

台输出，该工具比较了这个输出和源代码列表末尾注释中的预期输出，这样读者就可以看到预期输出，并知道此输出已通过了构建过程的验证。

JUnit

JUnit 最初发布于 2000 年，大概是基于 Java 1.0 版本的，因此无法使用 Java 的反射机制(请参阅第 19 章)。这就导致了使用旧的 JUnit 来编写单元测试需要做很多冗余的工作。我不喜欢它的设计，因此编写了自己的单元测试框架来作为进阶卷第 4 章的示例。这个框架走向了另一个极端:“尝试最简单且可行的事情。”[这是**极限编程**（XP）中的一句名言。] 从那时起，JUnit 通过使用反射和注解极大地改进了自身，并大大简化了编写单元测试的过程。而使用 Java 8 后，JUnit 甚至添加了对 lambda 表达式的支持。本书使用了 JUnit5，是编写时的最新版本。

在 JUnit 的最简单用法中，可以使用 `@Test` 注解来表示测试的每个方法。JUnit 将这些方法识别为单独的测试，每次设置和运行一个，并采取措施来避免测试之间相互影响。

让我们尝试一个简单的示例。`CountedList` 继承了 `ArrayList`，并添加了一些信息来跟踪 `CountedList` 的创建数量：

```
// validating/CountedList.java
// 跟踪自身创建的数量
package validating;
import java.util.*;

public class CountedList extends ArrayList<String> {
  private static int counter = 0;
  private int id = counter++;
  public CountedList() {
    System.out.println("CountedList #" + id);
  }
  public int getId() { return id; }
}
```

标准做法是将测试代码放在它们自己的子目录中，测试代码也必须放在包里，这样 JUnit 才能识别它们：

```
// validating/tests/CountedListTest.java
// 使用 JUnit 来简单地测试 CountedList
package validating;
import java.util.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class CountedListTest {
```

```
  private CountedList list;
  @BeforeAll
  static void beforeAllMsg() {
    System.out.println(">>> Starting CountedListTest");
  }
  @AfterAll
  static void afterAllMsg() {
    System.out.println(">>> Finished CountedListTest");
  }
  @BeforeEach
  public void initialize() {
    list = new CountedList();
    System.out.println("Set up for " + list.getId());
    for(int i = 0; i < 3; i++)
      list.add(Integer.toString(i));
  }
  @AfterEach
  public void cleanup() {
    System.out.println("Cleaning up " + list.getId());
  }
  @Test
  public void insert() {
    System.out.println("Running testInsert()");
    assertEquals(list.size(), 3);
    list.add(1, "Insert");
    assertEquals(list.size(), 4);
    assertEquals(list.get(1), "Insert");
  }
  @Test
  public void replace() {
    System.out.println("Running testReplace()");
    assertEquals(list.size(), 3);
    list.set(1, "Replace");
    assertEquals(list.size(), 3);
    assertEquals(list.get(1), "Replace");
  }
  // 用于简化代码的辅助方法
  // 只要没有 @Test 注解，JUnit 就不会自动执行
  private
  void compare(List<String> lst, String[] strs) {
    assertArrayEquals(lst.toArray(new String[0]), strs);
  }
  @Test
  public void order() {
    System.out.println("Running testOrder()");
    compare(list, new String[] { "0", "1", "2" });
  }
  @Test
  public void remove() {
    System.out.println("Running testRemove()");
    assertEquals(list.size(), 3);
    list.remove(1);
```

```
/* 输出：
>>> Starting CountedListTest
CountedList #0
Set up for 0
Running testRemove()
Cleaning up 0
CountedList #1
Set up for 1
Running testReplace()
Cleaning up 1
CountedList #2
Set up for 2
Running testAddAll()
Cleaning up 2
CountedList #3
Set up for 3
Running testInsert()
Cleaning up 3
CountedList #4
Set up for 4
Running testOrder()
Cleaning up 4
>>> Finished CountedListTest
*/
```

```
    assertEquals(list.size(), 2);
    compare(list, new String[] { "0", "2" });
  }
  @Test
  public void addAll() {
    System.out.println("Running testAddAll()");
    list.addAll(Arrays.asList(new String[] {
      "An", "African", "Swallow"}));
    assertEquals(list.size(), 6);
    compare(list, new String[] { "0", "1", "2",
       "An", "African", "Swallow" });
  }
}
```

`@BeforeAll` 注解标注的方法会在任何测试执行之前运行一次。`@AfterAll` 注解标注的方法在所有测试执行之后运行一次。两种方法都必须是静态的。

`@BeforeEach` 注解标注的方法通常用于创建和初始化一组公共对象，并在每次测试之前运行。你也可以将这些初始化操作放在测试类的构造器中，不过我认为 `@BeforeEach` 更清晰。JUnit 为每个测试创建一个对象 [①]，以确保运行的测试之间没有副作用。不过，所有测试对应的全部对象都是提前一次性创建的（而不是在测试运行之前创建），所以使用 `@BeforeEach` 和构造器之间的唯一区别是，`@BeforeEach` 在测试之前才被调用。在大多数情况下，这不是问题，如果你愿意，也可以使用构造器方式。

如果在每次测试后必须执行清理（比如需要恢复修改过的 `static` 成员，需要关闭打开的文件、数据库或网络连接等），请使用 `@AfterEach` 注解来标注方法。

上面的示例中，每个测试都生成一个新的 `CountedListTest` 对象，此时会创建所有非静态的成员。然后为该测试调用 `initialize()` 方法，来给 `list` 分配一个新的 `CountedList` 对象，并使用字符串 `"0"`、`"1"` 和 `"2"` 进行初始化。

为了观察 `@BeforeEach` 和 `@AfterEach` 的行为，这些方法在测试初始化和清理时会显示有关测试的信息。

`insert()` 和 `replace()` 演示了如何编写典型的测试方法。JUnit 使用 `@Test` 注解来标注这些方法，并将每个方法作为测试运行。在这些方法中，你可以执行任何所需的操作，并使用 JUnit 的断言方法（均以名称“assert”开头）来验证测试的正确性（在 JUnit 文档中可以找到所有的“assert”语句）。如果断言失败，则会显示导致失败的表达式和值。通常来说这就足够了，但你也可以使用 JUnit 断言的重载版本，提供一个在断言失败时显示的字符串。

① 该对象指的是测试所在类的对象，用来运行测试。——译者注

断言语句不是必需的，也可以在没有断言的情况下运行测试，如果没有异常，就可以认为测试是成功的。

`compare()` 是一个“辅助方法”，它不由 JUnit 执行，而是由类中的其他测试使用。只要没有 `@Test` 注解，JUnit 就不会运行它，或期望它具备特定的方法签名。本例中 `compare()` 是 `private` 的，强调它只在当前测试类中使用，但它也可以是 `public` 的。其余的测试方法通过将重复代码重构到 `compare()` 里来消除冗余。

本书使用 `build.gradle` 文件来控制测试。要运行本章的测试，命令是：

```
gradlew validating:test
```

如果某个测试在本次构建中已经运行，Gradle 不会再次运行它。因此如果没有得到测试结果，请先运行：

```
gradlew validating:clean
```

可以使用以下命令运行本书中的所有测试：

```
gradlew test
```

虽然可以只使用最简单的 JUnit 方法，就像 CountedListTest.java 中演示的那样，但 JUnit 还包含了许多其他的测试结构，你可以在 JUnit 网站上了解它们。

JUnit 是目前最流行的 Java 单元测试框架之一，但也有其他的替代方案。你可以在网上搜索一下，看看有没有更适合自己需求的。

16.1.2 测试覆盖率的幻觉

测试覆盖率（test coverage），也称为**代码覆盖率**（code coverage），是衡量代码库的测试百分比。百分比越高，测试覆盖率越大。[①]

对于没有相关知识但处于控制地位的人来说，很容易做出只接受覆盖率为 100% 的决定。这是有问题的，因为这个数字并不是衡量测试有效性的合理标准。你可能测试了所有需要测试的内容，但只达到 65% 的测试覆盖率。如果有人要求 100% 的覆盖率，那么你就会在其余部分浪费大量时间，并且在以后向项目添加代码时浪费更多时间。

当分析未知的代码库时，测试覆盖率作为粗略的衡量标准非常有用。如果覆盖率工

① 有多种方法用于计算覆盖率，还有一篇描述 Java 代码覆盖率工具的有用文章。请到维基百科网站搜索 code coverage 以及 Java code coverage tools 查看。

具报告的值特别低（例如，小于 40%），则表明覆盖率可能不足。然而，一个非常高的值同样是可疑的，这表明对编程领域知识不足的人强迫团队做出了武断的决定。覆盖工具的最佳用途是发现代码库中未经测试的部分。但是，不要依赖覆盖率来获取测试质量相关的信息。

16.2 前置条件

前置条件（precondition）的概念来自**契约式设计**（Design By Contract, DbC），并使用了基本的**断言**（assertion）机制来实现。本节中我们首先查看 Java 中的断言，然后介绍 DbC，最后以 Google Guava 库为例来讲解。

16.2.1 断言

断言通过验证程序执行期间是否满足某些条件来提高程序的稳健性。

例如，假设在对象中有一个数值字段表示儒略历上的月份。我们知道此值必须始终在 1~12 范围内。断言可以检查这一点，并在它超出该范围时报告错误。如果在方法内部，则可以使用断言来检查参数的有效性。这些都是确保程序正确的重要测试，但它们不能在编译时检查，也不属于单元测试的范围。

1. Java 断言语法

你可以使用其他编程结构来模拟断言的效果，而对于 Java 直接提供的断言来说，它的亮点是易于编写。断言语句有两种形式：

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

两者都表示“我断言这个 boolean-expression 的值是 true”。如果不是这种情况，则断言会产生一个 AssertionError 异常。它是 Throwable 的一个子类，因此不需要指定异常规范。

不幸的是，第一种断言形式产生的异常**不**包含 boolean-expression 的任何信息（这与大多数其他语言的断言机制相反）。下面是使用第一种形式断言的示例：

```
// validating/Assert1.java
// 没有信息提示的断言
// 运行时需要使用 -ea 标志：
// {java -ea Assert1}
// {ThrowsException}
```

```
public class Assert1 {
  public static void main(String[] args) {
    assert false;
  }
}
```

```
/* 输出:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError
        at Assert1.main(Assert1.java:9)
*/
```

如果正常运行程序，不加任何特殊的断言标志，则什么都不会发生。你必须在运行程序时显式地启用断言。最简单的方法是使用 -ea 标志，它也可以拼写为 -enableassertions。这将运行程序并执行任何断言语句。

以上示例的输出中没有太多有用的信息。相较而言，如果使用 information-expression 形式的断言，就可以在异常栈里生成一个有用的消息。最有用的 information-expression 通常是给程序员看的字符串文本：

```
// validating/Assert2.java
// 使用 information-expression 形式的断言
// {java Assert2 -ea}
// {ThrowsException}

public class Assert2 {
  public static void main(String[] args) {
    assert false:
      "Here's a message saying what happened";
  }
}
```

```
/* 输出:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Here's a message saying what happened
        at Assert2.main(Assert2.java:8)
*/
```

information-expression 可以生成任何类型的对象，因此我们通常会构造一个更复杂的字符串，其中包含与失败断言有关的对象的值。

你还可以根据类名或包名打开和关闭断言，也就是说，可以为整个包启用或禁用断言。执行此操作的详细信息在有关断言的 JDK 文档中。在使用断言进行检测的大型项目里，如果想要打开或关闭某些断言，这个功能就非常有用。不过，日志或调试可能是获取这类信息更好的工具，它们分别在 16.4 节和 16.5 节有详细的描述。

还有另一种方法可以控制断言：以编程的方式操作 ClassLoader 对象。ClassLoader 中有几种方法允许动态启用和禁用断言，包括 setDefaultAssertionStatus()，它为之后加载的所有类设置了断言状态。所以你可以像这样静默地开启断言：

```
// validating/LoaderAssertions.java
// 使用类加载器开启断言
// {ThrowsException}

public class LoaderAssertions {
  public static void main(String[] args) {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
    new Loaded().go();
  }
}

class Loaded {
  public void go() {
    assert false: "Loaded.go()";
  }
}
```

```
/* 输出：
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Loaded.go()
        at Loaded.go(LoaderAssertions.java:15)
        at
LoaderAssertions.main(LoaderAssertions.java:9)
*/
```

这消除了运行程序时在命令行上使用 -ea 标志的需要，当然使用 -ea 标志启用断言可能同样简单。在交付独立产品时，你可能需要设置一个执行脚本，来配置其他启动参数，以便用户无论如何都可以启动程序。

不过，在程序运行时再决定是否启用断言也是有道理的。你可以使用以下静态子句完成此操作，该子句放置在系统的主类中：

```
static {
  boolean assertionsEnabled = false;
  // 注意，此处的赋值副作用是故意造成的：
  assert assertionsEnabled = true;
  if(!assertionsEnabled)
    throw new RuntimeException("Assertions disabled");
}
```

如果启用了断言，则会执行 assert 语句，然后 assertionsEnabled 变为 true。这个断

言永远不会失败，因为赋值的返回值是分配的值。如果断言未启用，则不会执行 assert 语句，然后 assertionsEnabled 就保持为 false，从而抛出异常。

2. Guava 里的断言

启用 Java 原生的断言很麻烦，因此 Guava[①] 团队添加了一个 Verify 类，提供了始终启用的替换断言。他们建议静态导入 Verify 方法：

```
// validating/GuavaAssertions.java
// 始终启用的断言
import com.google.common.base.*;
import static com.google.common.base.Verify.*;

public class GuavaAssertions {
  public static void main(String[] args) {
    verify(2 + 2 == 4);
    try {
      verify(1 + 2 == 4);
    } catch(VerifyException e) {
      System.out.println(e);
    }
    try {
      verify(1 + 2 == 4, "Bad math");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
    try {
      verify(1 + 2 == 4, "Bad math: %s", "not 4");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
    String s = "";
    s = verifyNotNull(s);
    s = null;
    try {
      verifyNotNull(s);
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
    try {
      verifyNotNull(
        s, "Shouldn't be null: %s", "arg s");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

```
/* 输出：
com.google.common.base.VerifyException
Bad math
Bad math: not 4
expected a non-null reference
Shouldn't be null: arg s
*/
```

① Google 提供的一个被广泛使用的 Java 第三方库。——译者注

这里有两个方法，`verify()` 和 `verifyNotNull()`，它们各自又有变种方法可以提供错误消息。请注意，`verifyNotNull()` 的内置错误消息通常就足够了，而 `verify()` 则过于笼统，无法提供有用的默认错误消息。

3. 在契约式设计中使用断言

契约式设计（DbC）是由 Eiffel 语言的发明者 Bertrand Meyer 所倡导的一个概念，通过保证对象遵循某些规则来创建稳健的程序[①]。这些规则由要解决问题的性质决定，而这超出了编译器可以验证的范围。

尽管断言没有像 Eiffel 语言那样直接实现 DbC，但它创建了一种非正式的 DbC 编程风格。

DbC 假定服务提供者与该服务的消费者或客户之间存在着明确指定的合同。在面向对象编程中，服务通常由对象提供，对象的边界——提供者和消费者之间的分界——是对象所属类的接口。当客户调用特定的公共方法时，他们期望该调用会产生某些特定的行为：对象中状态的更改，或可预测的返回值。Meyer 对这种行为的设计主旨概括如下。

1. 可以明确规定这种行为，就好像合同一样。
2. 可以通过某些运行时检查来保证这种行为，也就是他所说的**前置条件**、**后置条件**和**不变项**。

无论你是否认可第一点，大部分情况下它确实是正确的，这使得 DbC 成为一种有用的方法。（我相信，就像其他任何解决方案一样，它的有用性也存在局限。但是如果你知道这些局限，也就知道了何时可以尝试使用它。）值得特别关注的是，DbC 如何表达对特定类的约束，这是设计过程中很有价值的部分。如果你无法指定约束，那么你可能对要构建的内容了解得还不够深入。

4. 检查指令

在详细了解 DbC 之前，先考虑断言最简单的用法，Meyer 称之为**检查指令**（check instruction）。检查指令表明你确信代码运行到某一处时已经有了特定的属性。检查指令的思想是在代码中表达并非显而易见的结论，这不仅可以验证测试用例，还可以作为以后阅读代码时的文档。

做化学实验时，你可能会将一种无色透明液体滴入另一种无色透明液体，当滴入达到

① Bertrand Meyer 的著作《面向对象软件构造》（第 2 版）的第 11 章中详细地描述了契约式设计。

一定容量后，所有液体都变成了蓝色。仅凭这两种液体的颜色无法推断出这个结果，因其属于复杂化学反应的一部分。我们可以使用一个有用的检查指令，在滴定过程完成时，来断言所得液体为蓝色。

检查指令是对代码的宝贵补充，只要可以测试和阐明对象或程序的状态，就应该使用它。

5. 前置条件测试

前置条件确保客户（即调用此方法的代码）履行其合同部分。这几乎总是意味着在方法调用的最开始（即在该方法执行任何操作之前）检查参数，以确保它们适合在该方法中使用。你永远不知道客户会给你传递什么参数，所以前置条件检查总是一个好主意。

6. 后置条件

后置条件会测试方法的执行结果。此代码放置在方法调用的末尾，`return` 语句之前（如果有的话）。对于长而复杂的方法，如果需要在返回之前验证计算结果（即由于某种原因，你无法总是信任结果），后置条件检查是必不可少的。不过任何时候你都可以提供对方法结果的约束，在代码中将这些约束表示为后置条件是明智的。

7. 不变项

不变项保证了对象的状态在方法调用之间是不变的。但是，它并不限制方法在执行期间临时偏离这些保证。它只是说对象的状态信息在以下时间段会始终遵守规定的规则：

1. 进入方法后；
2. 离开方法之前。

此外，不变项是对对象构造后状态的保证。

根据这个描述，一个有效的不变项被定义为一个方法，可以命名为 `invariant()`，它在对象构造之后以及每个方法的开始和结束时被调用。该方法可以如此调用：

```
assert invariant();
```

这样，如果出于性能原因禁用断言，就不会产生开销。

8. 放宽 DbC 的限制

尽管 Meyer 强调了前置条件、后置条件和不变项的价值，以及在开发过程中使用这些

工具的重要性，但他也承认在发布的产品中包含所有的 DbC 代码并不总是可行的。你可以根据对特定位置代码的信任程度来放宽 DbC 检查。下面是放宽 DbC 检查的顺序，从最安全到最不安全。

首先禁用每个方法开头的不变项检查，因为每个方法末尾的不变项检查就可以保证：对象的状态在每次方法调用开始时都是有效的。也就是说，你通常可以相信对象的状态不会在方法调用之间发生变化。这是一个非常安全的假设，你可以选择仅在最后编写带有不变项检查的代码。

当有合理的单元测试来验证方法的返回值时，可以禁用后置条件检查。不变项检查会观察对象的状态，因此后置条件检查仅在方法期间验证计算结果，这样的话你可以废弃后置条件检查，而采用单元测试。单元测试不会像运行时后置条件检查那样安全，但它可能已经足够了，尤其是如果你对自己的代码有信心的话。

如果确信方法体不会将对象置于无效状态，则可以禁用方法调用结束时的不变项检查。可以采用**白盒测试**（也就是使用可以访问私有字段的单元测试来验证对象状态）来验证这一点。因此，尽管它可能不如调用 `invariant( )` 有效，但还是可以将不变项检查从运行时测试“迁移”到构建时测试（通过单元测试），就像后置条件那样。

最后，不得已的话，还可以禁用前置条件检查。这是最不安全和最不明智的选择，因为虽然你了解并可以控制自己的代码，但无法控制客户传递给方法的参数。但是，在 (1) 迫切需要提高性能而分析指出前置条件检查是瓶颈的情况下，并且 (2) 你有某种合理的保证，即客户不会违反前置条件（比如客户端代码是你自己编写的），禁用前置条件检查也是可以接受的。

在禁用检查时不应该删除执行此检查的代码（只需将其注释掉）。如果发现错误，你就可以轻松恢复检查以快速发现问题。

16.2.2 DbC + 单元测试

以下示例演示了将契约式设计中的概念与单元测试相结合的效力。它通过“循环”数组实现了一个小型的先进先出（FIFO）队列。所谓的“循环”数组，就是以循环方式使用的数组，当到达数组末尾时，继续访问就又回到了数组的开头。

我们可以为这个队列做一些契约性的定义。

1. **前置条件（对 `put()` 来说）**：不允许将空元素添加到队列中。

2. **前置条件（对 put() 来说）**：将元素放入已满的队列是非法的。
3. **前置条件（对 get() 来说）**：尝试从空队列中获取元素是非法的。
4. **后置条件（对 get() 来说）**：不能从数组中获取空元素。
5. **不变项**：队列中包含对象的区域不能有任何空元素。
6. **不变项**：队列中不包含对象的区域必须只能有空值。

下面是实现这些规则的一种方式：通过显式方法调用来实现每种类型的 DbC 元素。首先，创建一个专用的 Exception：

```
// validating/CircularQueueException.java
package validating;

public class
CircularQueueException extends RuntimeException {
  public CircularQueueException(String why) {
    super(why);
  }
}
```

它用于报告 CircularQueue 类的错误：

```
// validating/CircularQueue.java
// 契约式设计（DbC）的演示
package validating;
import java.util.*;

public class CircularQueue {
  private Object[] data;
  private int
    in = 0, // 下一个可用的存储空间
    out = 0; // 下一个可以获取的对象
  // 是否已经回到了循环队列的开头?
  private boolean wrapped = false;
  public CircularQueue(int size) {
    data = new Object[size];
    // 构造后必须为真:
    assert invariant();
  }
  public boolean empty() {
    return !wrapped && in == out;
  }
  public boolean full() {
    return wrapped && in == out;
  }
  public boolean isWrapped() { return wrapped; }
  public void put(Object item) {
    precondition(item != null, "put() null item");
    precondition(!full(),
```

```
      "put() into full CircularQueue");
    assert invariant();
    data[in++] = item;
    if(in >= data.length) {
      in = 0;
      wrapped = true;
    }
    assert invariant();
  }
  public Object get() {
    precondition(!empty(),
      "get() from empty CircularQueue");
    assert invariant();
    Object returnVal = data[out];
    data[out] = null;
    out++;
    if(out >= data.length) {
      out = 0;
      wrapped = false;
    }
    assert postcondition(
      returnVal != null,
        "Null item in CircularQueue");
    assert invariant();
    return returnVal;
  }
  // 契约式设计的相关方法
  private static void
  precondition(boolean cond, String msg) {
    if(!cond) throw new CircularQueueException(msg);
  }
  private static boolean
  postcondition(boolean cond, String msg) {
    if(!cond) throw new CircularQueueException(msg);
    return true;
  }
  private boolean invariant() {
    // 保证在保存了对象的 data 区域不会有空值:
    for(int i = out; i != in; i = (i + 1) % data.length)
      if(data[i] == null)
        throw new CircularQueueException(
          "null in CircularQueue");
    // 保证在保存了对象的 data 区域之外只会有空值:
    if(full()) return true;
    for(int i = in; i != out; i = (i + 1) % data.length)
      if(data[i] != null)
        throw new CircularQueueException(
          "non-null outside of CircularQueue range: "
          + dump());
    return true;
  }
  public String dump() {
```

```
      return "in = " + in +
        ", out = " + out +
        ", full() = " + full() +
        ", empty() = " + empty() +
        ", CircularQueue = " + Arrays.asList(data);
    }
  }
```

计数器 in 表示在数组中存储下一个对象时的位置。计数器 out 表示要获取的下一个对象的位置。wrapped 标志表示 in 已经回到了循环队列的开头，现在在 out 后面移动。当 in 和 out 重合时，队列为空（如果 wrapped 为 false）或满（如果 wrapped 为 true）。

put() 和 get() 方法调用了 precondition()、postcondition() 和 invariant()，它们是定义在类的下面部分的 private 方法。precondition() 和 postcondition() 是辅助方法，可以让代码更清晰。注意 precondition() 返回 void，因为它不与 assert 一起使用。如前所述，你通常要在代码中保留前置条件。通过将它们包装在一个 precondition() 方法调用中，可以很方便地减少或关闭这些前置条件。

postcondition() 和 invariant() 都返回一个布尔值，因此它们可以在 assert 语句中使用。之后如果出于性能原因禁用断言，那么就根本不会有方法调用。

invariant() 在对象上执行内部有效性检查。如果像 Meyer 建议的那样，在每个方法调用的开始和结束时都执行此操作，那么付出的代价会很高。不过它的价值在代码中很清楚地展示了出来，它能帮助我调试自己的实现代码。此外，如果你对实现有任何更改，invariant() 可以确保你没有破坏自己的代码。将不变项测试从方法调用移到单元测试中相当简单，因此如果你的单元测试很完善，那么就有理由相信不变项会得到遵守。

dump() 辅助方法返回一个包含所有数据的字符串，而不是直接打印数据。这为返回信息的使用提供了更多的选项。

现在可以为该类创建 JUnit 测试了：

```
// validating/tests/CircularQueueTest.java
package validating;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class CircularQueueTest {
  private CircularQueue queue = new CircularQueue(10);
  private int i = 0;
  @BeforeEach
  public void initialize() {
```

```
    while(i < 5) // 提前加载一些数据
      queue.put(Integer.toString(i++));
  }
  // 支撑方法：
  private void showFullness() {
    assertTrue(queue.full());
    assertFalse(queue.empty());
    System.out.println(queue.dump());
  }
  private void showEmptiness() {
    assertFalse(queue.full());
    assertTrue(queue.empty());
    System.out.println(queue.dump());
  }
  @Test
  public void full() {
    System.out.println("testFull");
    System.out.println(queue.dump());
    System.out.println(queue.get());
    System.out.println(queue.get());
    while(!queue.full())
      queue.put(Integer.toString(i++));
    String msg = "";
    try {
      queue.put("");
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "put() into full CircularQueue");
    showFullness();
  }
  @Test
  public void empty() {
    System.out.println("testEmpty");
    while(!queue.empty())
      System.out.println(queue.get());
    String msg = "";
    try {
      queue.get();
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "get() from empty CircularQueue");
    showEmptiness();
  }
  @Test
  public void nullPut() {
    System.out.println("testNullPut");
    String msg = "";
    try {
```

```
      queue.put(null);
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "put() null item");
  }
  @Test
  public void circularity() {
    System.out.println("testCircularity");
    while(!queue.full())
      queue.put(Integer.toString(i++));
    showFullness();
    assertTrue(queue.isWrapped());
    while(!queue.empty())
      System.out.println(queue.get());
    showEmptiness();
    while(!queue.full())
      queue.put(Integer.toString(i++));
    showFullness();
    while(!queue.empty())
      System.out.println(queue.get());
    showEmptiness();
  }
}
```

```
/* 输出:
testNullPut
put() null item
testCircularity
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
1
2
3
4
5
6
7
8
9
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
10
11
12
13
14
15
16
17
18
19
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
testFull
in = 5, out = 0, full() = false, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, null, null, null, null, null]
0
1
put() into full CircularQueue
in = 2, out = 2, full() = true, empty() = false,
CircularQueue =
[10, 11, 2, 3, 4, 5, 6, 7, 8, 9]
testEmpty
0
1
2
3
4
get() from empty CircularQueue
in = 5, out = 5, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
*/
```

`initialize()` 方法加载了一些数据，因此每个测试里的 `CircularQueue` 都有数据。支撑方法 `showFullness()` 和 `showEmptiness()` 分别表示 `CircularQueue` 是满还是空。四个测试方法分别确保 `CircularQueue` 的某个功能正常。

注意，通过将 DbC 与单元测试相结合，你不仅可以利用它们各自的优点，而且还有一条迁移路径——你可以将一些 DbC 测试移动到单元测试中，而不是简单地禁用它们，这样你仍然能保证有某些层次的测试。

16.2.3 使用 Guava 里的前置条件

在 16.2.1 节“放宽 DbC 的限制”中，我指出前置条件是 DbC 中不应该删除的部分，因为它会检查方法参数的有效性。这是你无法控制的，因此你确实需要检查它们。由于 Java 默认禁用断言，因此最好还是使用始终验证方法参数的其他库。

Google 的 Guava 库包含了一组很好的前置条件测试，这些测试不仅易于使用，而且提供了具有描述性的良好命名。在下面的示例中，你可以看到所有这些测试的简单用法。库设计者建议静态导入这些前置条件：

```
// validating/GuavaPreconditions.java
// 演示 Guava 的前置条件测试
import java.util.function.*;
import static com.google.common.base.Preconditions.*;

public class GuavaPreconditions {
  static void test(Consumer<String> c, String s) {
    try {
      System.out.println(s);
      c.accept(s);
      System.out.println("Success");
    } catch(Exception e) {
      String type = e.getClass().getSimpleName();
      String msg = e.getMessage();
      System.out.println(type +
        (msg == null ? "" : ": " + msg));
    }
  }
  public static void main(String[] args) {
    test(s -> s = checkNotNull(s), "X");
    test(s -> s = checkNotNull(s), null);
    test(s -> s = checkNotNull(s, "s was null"), null);
    test(s -> s = checkNotNull(
      s, "s was null, %s %s", "arg2", "arg3"), null);

    test(s -> checkArgument(s == "Fozzie"), "Fozzie");
    test(s -> checkArgument(s == "Fozzie"), "X");
    test(s -> checkArgument(s == "Fozzie"), null);
    test(s -> checkArgument(
      s == "Fozzie", "Bear Left!"), null);
    test(s -> checkArgument(
      s == "Fozzie", "Bear Left! %s Right!", "Frog"),
      null);

    test(s -> checkState(s.length() > 6), "Mortimer");
    test(s -> checkState(s.length() > 6), "Mort");
    test(s -> checkState(s.length() > 6), null);

    test(s ->
```

```
        checkElementIndex(6, s.length()), "Robert");
      test(s ->
        checkElementIndex(6, s.length()), "Bob");
      test(s ->
        checkElementIndex(6, s.length()), null);

      test(s ->
        checkPositionIndex(6, s.length()), "Robert");
      test(s ->
        checkPositionIndex(6, s.length()), "Bob");
      test(s ->
        checkPositionIndex(6, s.length()), null);

      test(s -> checkPositionIndexes(
        0, 6, s.length()), "Hieronymus");
      test(s -> checkPositionIndexes(
        0, 10, s.length()), "Hieronymus");
      test(s -> checkPositionIndexes(
        0, 11, s.length()), "Hieronymus");
      test(s -> checkPositionIndexes(
        -1, 6, s.length()), "Hieronymus");
      test(s -> checkPositionIndexes(
        7, 6, s.length()), "Hieronymus");
      test(s -> checkPositionIndexes(
        0, 6, s.length()), null);
    }
}
```

```
/* 输出:
X
Success
null
NullPointerException
null
NullPointerException: s was null
null
NullPointerException: s was null, arg2 arg3
Fozzie
Success
X
IllegalArgumentException
null
IllegalArgumentException
null
IllegalArgumentException: Bear Left!
null
IllegalArgumentException: Bear Left! Frog Right!
Mortimer
Success
Mort
```

```
IllegalStateException
null
NullPointerException
Robert
IndexOutOfBoundsException: index (6) must be less than
size (6)
Bob
IndexOutOfBoundsException: index (6) must be less than
size (3)
null
NullPointerException
Robert
Success
Bob
IndexOutOfBoundsException: index (6) must not be
greater than size (3)
null
NullPointerException
Hieronymus
Success
Hieronymus
Success
Hieronymus
IndexOutOfBoundsException: end index (11) must not be
greater than size (10)
Hieronymus
IndexOutOfBoundsException: start index (-1) must not be
negative
Hieronymus
IndexOutOfBoundsException: end index (6) must not be
less than start index (7)
null
NullPointerException
*/
```

尽管 Guava 的前置条件适用于所有类型，但我在这里只演示了 `String` 类型。`test()` 方法需要一个 `Consumer<String>`，因此可以将 lambda 表达式作为第一个参数传递，并将 `String` 作为第二个参数传递给 lambda 表达式。它先打印了 `String`，方便你在查看输出时定位对应的代码，然后将 `String` 传递给 lambda 表达式。`try` 块中的第二个 `println()` 方法仅在 lambda 表达式成功时才显示，否则会执行 `catch` 子句，然后显示错误信息。注意我们通过 `test()` 方法消除了多少重复代码。

每个前置条件都有三种不同的重载形式：没有消息的测试，带有简单 `String` 消息的测试，以及带有 `String` 及替换值的可变参数列表的测试。出于效率原因，只允许使用 `%s`（`String` 类型）替换标签。在上面的示例中，我们仅在 `checkNotNull()` 和 `checkArgument()`

里演示了这两种形式的 String 消息，但它们对所有其余的前置条件方法都是适用的。

请注意，checkNotNull() 会返回其参数，因此你可以在表达式中通过内联的方式使用它。下面的示例在构造器中使用它来防止构造包含 null 值的对象：

```
// validating/NonNullConstruction.java
import static com.google.common.base.Preconditions.*;

public class NonNullConstruction {
  private Integer n;
  private String s;
  NonNullConstruction(Integer n, String s) {
    this.n = checkNotNull(n);
    this.s = checkNotNull(s);
  }
  public static void main(String[] args) {
    NonNullConstruction nnc =
      new NonNullConstruction(3, "Trousers");
  }
}
```

checkArgument() 使用布尔表达式来对参数进行更具体的测试，并在失败时抛出 IllegalArgumentException。checkState() 会测试对象的状态（例如，不变项检查），而不是检查参数，并在失败时抛出 IllegalStateException。

最后三个方法在失败时都抛出 IndexOutOfBoundsException。checkElementIndex() 保证其第一个参数是一个 List、String 或数组的有效元素索引，这个 List、String 或数组的大小由第二个参数指定。checkPositionIndex() 确定它的第一个参数是否在 0 和第二个参数（包括）的范围内。checkPositionIndexes() 保证 [first_arg, second_arg) 是一个 List、String 或数组的有效子范围，而这个 List、String 或数组的大小是由第三个参数指定的。

Guava 的所有前置条件方法都被重载，包括基本类型和 Object 类型。

16.3 测试驱动开发

测试驱动开发（TDD）的前提是，如果在设计和编写代码时考虑到测试，你不仅会创建可测试的代码，而且代码的设计会变得更好。总的来说，这似乎是正确的。如果在开发时想着“我将如何测试这段代码”，这会使代码变得不一样，而通常来说，“可测试”的代码“可用性”也更高。

TDD 纯粹主义者在实现新功能之前会为该功能编写测试，这称为**测试优先开发**（Test-First Development）。为了演示，考虑一个小小的实用程序示例，它反转字符串中字符的

大小写。我们随意添加一些约束：字符串必须小于或等于 30 个字符，并且只能包含字母、空格、逗号和句点。

这个例子与标准的 TDD 不同，它被设计为能接受 StringInverter 的不同实现，这样可以显示当我们逐步满足测试时类的演变。为了实现这一点，StringInverter 被定义为一个 interface：

```
// validating/StringInverter.java
package validating;

interface StringInverter {
  String invert(String str);
}
```

现在编写测试来表达我们的需求。通常情况下并不是这样编写测试的，但这里有一个特殊限制：我们想要测试 StringInverter 的多个版本的实现。为了能做到这一点，我们利用了 JUnit5 中最复杂的新特性之一：**动态测试生成**（dynamic test generation）。顾名思义，你可以编写代码在运行时生成测试，而不是自己手动编写每个测试。这带来了许多新的可能性，特别是在显式编写完整测试集可能会令人望而却步的情况下。

JUnit5 提供了多种动态生成测试的方法，但这里使用的方法可能是最复杂的。DynamicTest.stream() 方法的参数包括下列组成部分。

- 一组对象的迭代器，每组测试的对象都是不同的。该迭代器生成的对象可以是任何类型，但每次只生成一个对象，因此对于不同的多个项目，你必须人为地将它们打包成一个类型。
- 一个 Function，它从迭代器中获取对象并生成一个字符串来描述这个测试。
- 一个 Consumer，它接受来自迭代器的对象，并包含了基于该对象的测试代码。

在这个例子中，所有本来会被复制的代码都被合并到了 testVersions() 中。迭代器生成的对象是 StringInverter 的不同实现，这些不同实现演示了我们是如何一步步添加新功能，最终满足所有测试要求的：

```
// validating/tests/DynamicStringInverterTests.java
package validating;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.DynamicTest.*;
```

```
class DynamicStringInverterTests {
  // 组合操作来防止重复代码:
  Stream<DynamicTest> testVersions(String id,
    Function<StringInverter, String> test) {
    List<StringInverter> versions = Arrays.asList(
      new Inverter1(), new Inverter2(),
      new Inverter3(), new Inverter4());
    return DynamicTest.stream(
      versions.iterator(),
      inverter -> inverter.getClass().getSimpleName(),
      inverter -> {
        System.out.println(
          inverter.getClass().getSimpleName() +
            ": " + id);
        try {
          if(test.apply(inverter) != "fail")
            System.out.println("Success");
        } catch(Exception | Error e) {
          System.out.println(
            "Exception: " + e.getMessage());
        }
      }
    );
  }
  String isEqual(String lval, String rval) {
    if(lval.equals(rval))
      return "success";
    System.out.println("FAIL: " + lval + " != " + rval);
    return "fail";
  }
  @BeforeAll
  static void startMsg() {
    System.out.println(
      ">>> Starting DynamicStringInverterTests <<<");
  }
  @AfterAll
  static void endMsg() {
    System.out.println(
      ">>> Finished DynamicStringInverterTests <<<");
  }
  @TestFactory
  Stream<DynamicTest> basicInversion1() {
    String in =  "Exit, Pursued by a Bear.";
    String out = "eXIT, pURSUED BY A bEAR.";
    return testVersions(
      "Basic inversion (should succeed)",
      inverter -> isEqual(inverter.invert(in), out)
    );
  }
  @TestFactory
  Stream<DynamicTest> basicInversion2() {
```

```
    return testVersions(
      "Basic inversion (should fail)",
      inverter -> isEqual(inverter.invert("X"), "X"));
  }
  @TestFactory
  Stream<DynamicTest> disallowedCharacters() {
    String disallowed = ";-_()*&^%$#@!~`0123456789";
    return testVersions(
      "Disallowed characters",
      inverter -> {
        String result = disallowed.chars()
          .mapToObj(c -> {
            String cc = Character.toString((char)c);
            try {
              inverter.invert(cc);
              return "";
            } catch(RuntimeException e) {
              return cc;
            }
          }).collect(Collectors.joining(""));
        if(result.length() == 0)
          return "success";
        System.out.println("Bad characters: " + result);
        return "fail";
      }
    );
  }
  @TestFactory
  Stream<DynamicTest> allowedCharacters() {
    String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
    String upcase =  "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
    return testVersions(
      "Allowed characters (should succeed)",
      inverter -> {
        assertEquals(inverter.invert(lowcase), upcase);
        assertEquals(inverter.invert(upcase), lowcase);
        return "success";
      }
    );
  }
  @TestFactory
  Stream<DynamicTest> lengthNoGreaterThan30() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() > 30);
    return testVersions(
      "Length must be less than 31 (throws exception)",
      inverter -> inverter.invert(str)
    );
  }
  @TestFactory
  Stream<DynamicTest> lengthLessThan31() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
```

```
        assertTrue(str.length() < 31);
        return testVersions(
          "Length must be less than 31 (should succeed)",
          inverter -> inverter.invert(str)
        );
      }
    }
```

在一般的测试中，我们遇到一个失败的测试后就会停止构建。然而在这里，我们希望系统只报告问题，但测试仍然继续，以便看到不同版本的 `StringInverter` 的效果。

用 `@TestFactory` 注解标注过的每个方法都会生成一个 `DynamicTest` 对象的流（通过 `testVersions()`），JUnit 会像执行常规的 `@Test` 方法一样执行流里的每个测试。

现在测试已经写好，可以开始实现 `StringInverter` 了。我们从一个没什么功能的类开始，它直接返回传入的参数：

```
// validating/Inverter1.java
package validating;

public class Inverter1 implements StringInverter {
  @Override
  public String invert(String str) { return str; }
}
```

接下来实现反转操作：

```
// validating/Inverter2.java
package validating;
import static java.lang.Character.*;

public class Inverter2 implements StringInverter {
  @Override public String invert(String str) {
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

现在添加代码来确保字符串长度不超过 30 个字符：

```
// validating/Inverter3.java
package validating;
```

```
import static java.lang.Character.*;

public class Inverter3 implements StringInverter {
  @Override public String invert(String str) {
    if(str.length() > 30)
      throw new RuntimeException("argument too long!");
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

最后，排除不允许的字符：

```
// validating/Inverter4.java
package validating;
import static java.lang.Character.*;

public class Inverter4 implements StringInverter {
  static final String ALLOWED =
    "abcdefghijklmnopqrstuvwxyz ,." +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  @Override public String invert(String str) {
    if(str.length() > 30)
      throw new RuntimeException("argument too long!");
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      if(ALLOWED.indexOf(c) == -1)
        throw new RuntimeException(c + " Not allowed");
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

从测试用例的输出中可以看到，不同版本的 `Inverter` 一个比一个更接近于通过所有的测试。这提升了你在执行**测试优先开发**时的体验。

DynamicStringInverterTests.java 用于展示在 TDD 中 `StringInverter` 不同实现版本的开发过程。通常来说，只需要编写如下所示的测试，并修改单个 `StringInverter` 类，直到它满足所有测试为止：

```
// validating/tests/StringInverterTests.java
package validating;
import java.util.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class StringInverterTests {
  StringInverter inverter = new Inverter4();
  @BeforeAll
  static void startMsg() {
    System.out.println(">>> StringInverterTests <<<");
  }
  @Test
  void basicInversion1() {
    String in = "Exit, Pursued by a Bear.";
    String out = "eXIT, pURSUED BY A bEAR.";
    assertEquals(inverter.invert(in), out);
  }
  @Test
  void basicInversion2() {
    assertThrows(Error.class, () -> {
      assertEquals(inverter.invert("X"), "X");
    });
  }
  @Test
  void disallowedCharacters() {
    String disallowed = ";-_()*&^%$#@!~`0123456789";
    String result = disallowed.chars()
      .mapToObj(c -> {
        String cc = Character.toString((char)c);
        try {
          inverter.invert(cc);
          return "";
        } catch(RuntimeException e) {
          return cc;
        }
      }).collect(Collectors.joining(""));
    assertEquals(result, disallowed);
  }
  @Test
  void allowedCharacters() {
    String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
    String upcase =  "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
    assertEquals(inverter.invert(lowcase), upcase);
    assertEquals(inverter.invert(upcase), lowcase);
  }
  @Test
  void lengthNoGreaterThan30() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() > 30);
    assertThrows(RuntimeException.class, () -> {
```

```
        inverter.invert(str);
      });
    }
    @Test
    void lengthLessThan31() {
      String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
      assertTrue(str.length() < 31);
      inverter.invert(str);
    }
  }
```

你可以先在测试用例里指明所有要实现的功能，然后以此为起点，在代码中实现相关功能，直到所有测试都通过。如果后续需要修复错误或添加新功能，万一代码被破坏，你还可以继续使用这些测试来提示自己（或其他任何人）。TDD 可以产生更好、更周到的测试，而试图在事后实现完整的测试覆盖率通常会产生仓促或无意义的测试。

测试驱动与测试优先

虽然我自己还没有形成测试优先的意识，但对于来自测试优先社区的“将未通过的测试作为书签”的这一概念，我十分感兴趣。当你暂时离开工作后，再回来时可能很难重新回到最佳状态，甚至很难找到上次工作暂停的地方。然而，一个失败的测试会让你回到你停下来的地方。这似乎可以让你更轻松地离开，而不必担心失去掌控。

纯测试优先编程的主要问题是，它假设你预先了解正在解决的问题的一切。根据我自己的经验，我通常从试验开始，对这个问题已经研究了一段时间后，我才能很好地理解它并编写测试。当然，偶尔会有一些问题在你开始解决之前就已经能完整地定义，但我个人并不经常遇到此类问题。实际上，可能值得创造一个像“面向测试开发”这样的词，来描述编写测试良好的代码的做法。

16.4 日志

日志会报告正在运行的程序的相关信息。

在可调试的程序中，日志可以是显示程序进度的普通状态数据（例如，安装程序可能会记录安装过程中采取的步骤、存储文件的目录、程序的启动值等）。

日志在调试过程中也很有帮助。如果没有日志，你可能会尝试通过插入 `println()` 语句来破译程序的行为。本书中的一些示例就使用了这种技术。而在没有调试器（一个即将介绍的主题）的情况下，这就是你所能做的一切。但是，一旦确定程序可以正常工作，你

可能就要删除 println() 语句。然后，如果遇到更多错误，你可能又需要将它们放回原处。如果能包含仅在必要时使用的输出语句，那该有多好啊。

Java 编译器会对未调用的代码进行优化，在日志包（logging package）可用之前，程序员可以基于这一点进行编程。如果 debug 是 static final boolean 的，你可以这样做：

```
if(debug) {
  System.out.println("Debug info");
}
```

当 debug 为 false 时，编译器会删除大括号内的代码。因此，代码不使用的话就对运行时没有影响。通过这种方法，你就可以在整个程序中放置跟踪代码，然后轻松地打开或关闭它。但该技术的一个缺点是，你必须重新编译代码以打开或关闭跟踪语句。如果能通过更改配置文件来修改日志记录属性，无须重新编译程序即可打开跟踪语句的话，那会方便很多。

标准 Java 发行版日志包（java.util.logging）的设计被普遍认为很糟糕。大多数人会选择一个替代的日志包。SLF4J（Simple Logging Facade for Java）为多个日志框架提供了一个统一的门面（facade），例如 java.util.logging、logback 和 log4j。SLF4J 允许最终用户在部署时再插入所需的日志框架。

SLF4J 提供了一个成熟的工具来报告有关程序的信息，其效率几乎与前面示例中的技术相同。对于非常简单的信息记录，可以执行以下操作：

```
// validating/SLF4JLogging.java
import org.slf4j.*;

public class SLF4JLogging {
  private static Logger log =
    LoggerFactory.getLogger(SLF4JLogging.class);
  public static void main(String[] args) {
    log.info("hello logging");
  }
}
/* 输出:
2021-01-24T08:49:38.496
[main] INFO  SLF4JLogging - hello logging
*/
```

输出中的格式和信息，甚至输出的内容是正常信息还是“错误”信息，都取决于连接到 SLF4J 的后端包。在上面的示例中，它连接到了 logback 库（通过本书的 build.gradle 文件），并作为标准输出展示。

如果我们修改 build.gradle 来使用 JDK 内置的日志包作为后端，输出将显示为错误输出，如下所示：

```
Aug 16, 2016 5:40:31 PM InfoLogging main
INFO: hello logging
```

日志系统会检测日志消息来源的类名和方法名。不过它不能保证这些名称是正确的，所以不要依赖于它们的准确性。

日志级别

SLF4J 提供了多个级别的报告。下面的示例按“严重性”的递增顺序显示了所有的级别：

```
// validating/SLF4JLevels.java
import org.slf4j.*;

public class SLF4JLevels {
  private static Logger log =
    LoggerFactory.getLogger(SLF4JLevels.class);
  public static void main(String[] args) {
    log.trace("Hello");
    log.debug("Logging");
    log.info("Using");
    log.warn("the SLF4J");
    log.error("Facade");
  }
}
```

```
/* 输出：
2021-01-24T08:49:37.658
[main] TRACE SLF4JLevels - Hello
2021-01-24T08:49:37.661
[main] DEBUG SLF4JLevels - Logging
2021-01-24T08:49:37.661
[main] INFO  SLF4JLevels - Using
2021-01-24T08:49:37.661
[main] WARN  SLF4JLevels - the SLF4J
2021-01-24T08:49:37.661
[main] ERROR SLF4JLevels - Facade
*/
```

这些不同的级别设置可以让你查看某个级别的消息。这些级别通常设置在单独的配置文件中，因此无须重新编译即可重新配置。配置文件格式取决于使用的后端日志实现。下面是 logback 使用的 XML 配置：

```
<!-- validating/logback.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
```

```
    %d{yyyy-MM-dd'T'HH:mm:ss.SSS}
    [%thread] %-5level %logger - %msg%n
          </pattern>
        </encoder>
      </appender>
      <root level="TRACE">
        <appender-ref ref="STDOUT" />
      </root>
    </configuration>
```

试着将 `<root level="TRACE">` 这一行更改为不同的级别，然后重新运行程序来查看输出是如何变化的。如果你不提供 `logback.xml` 文件，就会使用默认的配置。

这只是对 SLF4J 和日志的一个简短介绍，足够让你了解日志的基础知识了——你还有很长的路要走。可以访问 SLF4J 文档来更深入地了解相关内容。

16.5 调试

如果能明智地使用 `System.out` 语句或日志信息，也能深入洞察程序的行为，但对于复杂的问题来说，这种方法变得烦琐且耗时。

和打印语句相比，你可能还需要更深入地查看程序。为此，你需要一个**调试器**（debugger）。

除了比使用打印语句生成的信息更快、更容易显示之外，调试器还可以设置**断点**（breakpoint），并在程序到达这些断点时停止。调试器可以随时显示程序的状态，查看变量的值，逐行执行程序，连接到远程运行的程序，等等。尤其是开始构建更大的系统时（其中的错误很容易被忽视），熟悉调试器是值得的。

16.5.1 使用 JDB 进行调试

Java 调试器（JDB）是 JDK 附带的命令行工具。就调试指令及其命令行接口而言，JDB 在概念上可以说是继承自 Gnu 调试器（GDB，受原始 UNIX DB 的启发）。JDB 对于学习调试和执行简单的调试任务很有用，并且只要安装了 JDK，它就是可用的，了解这一点也很重要。但较大的项目需要图形调试器，稍后会对此进行介绍。

假设你编写了如下程序：

```
// validating/SimpleDebugging.java
// {ThrowsException}
```

```
public class SimpleDebugging {
  private static void foo1() {
    System.out.println("In foo1");
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    int j = 1;
    j--;
    int i = 5 / j;
  }
  public static void main(String[] args) {
    foo1();
  }
}
```

如果你看一下 foo3()，会发现问题很明显：发生了除零错误。但是假设这段代码隐藏在一个大程序中（正如这里的调用序列所暗示的那样），并且你不知道从哪里开始寻找问题。在这里，异常为你提供了足够的信息来定位问题。但是如果事情比这更困难，你就需要更深入地钻研它，这样获得的信息才能比异常提供的更多。

要运行 JDB，首先要让编译器使用 -g 标志来编译 SimpleDebugging.java，这样才会生成调试信息。然后开始使用命令行调试程序：

```
jdb SimpleDebugging
```

这将打开 JDB 并为你提供一个命令提示符。可以通过在提示符下键入 ? 来查看可用的 JDB 命令列表。

下面是一个交互式调试跟踪，演示了如何追查问题：

```
Initializing jdb ...
> catch Exception
```

> 表示 JDB 正在等待命令输入。命令 catch Exception 会在任何抛出异常的地方设置一个断点（但是，即使你没有明确给出这个指令，调试器也会停止——异常似乎是 JDB 中的默认断点）。

```
Deferring exception catch Exception.
It will be set after the class is loaded.
> run
```

现在程序将运行到下一个断点，在这个示例里就是发生异常的地方。下面是 run 命令的运行结果：

```
run SimpleDebugging
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: In foo1
In foo2
In foo3
Exception occurred: java.lang.ArithmeticException
(uncaught)"thread=main",
SimpleDebugging.foo3(), line=17 bci=15
17        int i = 5 / j;
```

程序一直运行到发生异常的第 17 行，但遇到异常时 JDB 并没有退出。调试器还会显示导致异常的代码行。你可以使用 list 命令列出程序源码中停止运行的地方：

```
main[1] list
13      private static void foo3() {
14        System.out.println("In foo3");
15        int j = 1;
16        j--;
17 =>     int i = 5 / j;
18      }
19      public static void main(String[] args) {
20        foo1();
21      }
22    }
```

此清单中的指针（“=>”）显示了程序执行到的位置，恢复执行后将从此处继续。你可以使用 cont（ continue ）命令恢复执行，但这会使 JDB 在异常处退出，然后打印栈信息。

locals 命令会转储所有的局部变量的值：

```
main[1] locals
Method arguments:
Local variables:
j = 0
```

wherei 命令会打印当前线程的方法栈里压入的栈帧：

```
main[1] wherei
[1] SimpleDebugging.foo3 (SimpleDebugging.java:17), pc = 15
[2] SimpleDebugging.foo2 (SimpleDebugging.java:11), pc = 8
[3] SimpleDebugging.foo1 (SimpleDebugging.java:7), pc = 8
[4] SimpleDebugging.main (SimpleDebugging.java:20), pc = 0
```

wherei 之后的每一行代表一个方法调用和调用返回的点 [由**程序计数器**（program counter）pc 的值表示]。这里的调用顺序是 main()、foo1()、foo2() 和 foo3()。

list 命令能显示运行停止的位置，所以你通常可以很容易地了解发生的情况并修复它。help 命令会告诉你 jdb 的更多用处，但在你花大量时间学习它之前，请记住命令行调试器往往需要花费更多精力才能获得结果。我们应该使用 jdb 学习调试的基础知识，然后转向图形调试器。

16.5.2 图形调试器

使用像 JDB 这样的命令行调试器可能很不方便。它需要显式命令来查看变量的状态（locals、dump），在源码中列出执行点（list），找出系统中的线程（threads），设置断点（stop in、stop at），等等。图形调试器只需单击几下鼠标即可提供这些功能，并且在不使用显式命令的情况下，还可以显示正在调试的程序的最新详细信息。

因此，尽管你可能是通过体验 JDB 开始的，但你会发现，学习使用图形调试器来快速跟踪错误会更有效率。像 JetBrains 公司的 IntelliJ IDEA 这样的 IDE，都包含了很好的 Java 图形调试器。

16.6 基准测试

> “忘记那些微不足道的性能调整吧，在 97% 的情况下，过早优化是万恶之源。”
>
> ——Donald Knuth

如果你发现自己处于过早优化的滑坡上，并且野心勃勃，那你可能会浪费数月的时间。通常，简单直接的编程方法就足够了。如果进行了不必要的优化，会使你的代码变得过于复杂和难以理解。

基准测试（Benchmarking）意味着对代码或算法进行计时，以查看哪些运行得更快。这和 16.7 节介绍的分析和优化不一样，后者查看整个程序并找到该程序最耗时的部分。

能不能简单地计算一段代码的执行时间？在像 C 这样直观的语言中，这种方法确实有效。而对于像 Java 这样具有复杂运行时系统的语言，基准测试变得更具挑战性。为了产生可靠的数据，实验设置必须控制各种变量，例如 CPU 频率、节能功能、在同一台机器上运行的其他进程、优化器选项，等等。

16.6.1 微基准测试

我们可以编写一个计时实用程序来比较不同代码段的运行速度，这个想法很诱人，看起来好像能生成一些有用的数据。

例如，我们有一个简单的 Timer 类，它可以通过如下两种方式使用。

1. 创建一个 Timer 对象，执行你想要的操作，然后在这个 Timer 对象上调用 duration() 方法，来生成以毫秒为单位的用时。

2. 将 Runnable 传递给静态方法 duration()。符合 Runnable 接口的类都会有一个函数式方法 run()，这个函数式方法有一个不带参数也不返回任何东西的函数签名。

```
// onjava/Timer.java
package onjava;
import static java.util.concurrent.TimeUnit.*;

public class Timer {
  private long start = System.nanoTime();
  public long duration() {
    return NANOSECONDS.toMillis(
      System.nanoTime() - start);
  }
  public static long duration(Runnable test) {
    Timer timer = new Timer();
    test.run();
    return timer.duration();
  }
}
```

这是一个简单的计时方法。那我们能不能直接运行一下代码，看看需要多长时间？

有许多因素会影响运行的结果，甚至会产生相反的指标。下面是一个使用标准 Java 里的 Arrays 库的示例（在第 21 章中有更完整的描述），它看起来好像没什么问题：

```
// validating/BadMicroBenchmark.java
// {ExcludeFromTravisCI}
import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark {
  static final int SIZE = 250_000_000;
  public static void main(String[] args) {
    try { // 对于内存不足的机器
      long[] la = new long[SIZE];
      System.out.println("setAll: " +
        Timer.duration(() ->
          Arrays.setAll(la, n -> n)));
      System.out.println("parallelSetAll: " +
```

```
          Timer.duration(() ->
            Arrays.parallelSetAll(la, n -> n)));
      } catch(OutOfMemoryError e) {
        System.out.println("Insufficient memory");
        System.exit(0);
      }
    }
  }
```

```
/* 输出:
Insufficient memory
*/
```

main() 的主体位于 try 块内，这样的话，如果一台机器[①]内存不足，就会停止构建。

对于 2.5 亿个 long 的数组（在大多数机器上几乎不会产生“内存不足”的异常），我们“比较”了 Arrays.setAll() 和 Arrays.parallelSetAll() 的性能。并行版本尝试使用多个处理器来更快地完成工作。（虽然我在本节中引入了一些并行的思想，但直到进阶卷第 5 章才会详细解释这些概念）。尽管如此，非并行版本似乎运行得更快，不过结果也可能因机器而异。

BadMicroBenchmark.java 中的每个操作都是独立的，但如果你的操作依赖于公共资源，并行版本最终可能会慢得多，因为多个任务会争用该资源：

```
// validating/BadMicroBenchmark2.java
// 依赖于某个公共资源
import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark2 {
  // 减小了 SIZE 的值来运行得更快一点:
  static final int SIZE = 5_000_000;
  public static void main(String[] args) {
    long[] la = new long[SIZE];
    Random r = new Random();
    System.out.println("parallelSetAll: " +
      Timer.duration(() ->
        Arrays.parallelSetAll(la, n -> r.nextLong())));
    System.out.println("setAll: " +
      Timer.duration(() ->
        Arrays.setAll(la, n -> r.nextLong())));
    SplittableRandom sr = new SplittableRandom();
    System.out.println("parallelSetAll: " +
      Timer.duration(() ->
        Arrays.parallelSetAll(la, n -> sr.nextLong())));
    System.out.println("setAll: " +
      Timer.duration(() ->
        Arrays.setAll(la, n -> sr.nextLong())));
  }
}
```

```
/* 输出:
parallelSetAll: 1008
setAll: 294
parallelSetAll: 78
setAll: 88
*/
```

① 比如具有 8GB 内存的 Mac Mini 整机。

SplittableRandom 是为并行算法设计的，它确实比 parallelSetAll() 中的普通 Random 运行得更快。但它似乎仍然比非并行的 setAll() 花费更长的时间，看上去不太可能（但也许就是真的。我们只是无法通过糟糕的微基准测试来判断）。

这里仅仅讨论了微基准测试的问题，JVM 的 Hotspot 技术对性能影响也很大。如果在运行测试之前不先运行代码来“预热”JVM，则可能会得到“冷”结果，这些结果并没有反映出程序运行一段时间后的速度（如果正在运行的应用程序使用并不频繁，结果最终没有触发 JVM 的“预热”呢？你将无法获得预期的性能，甚至可能还会降低速度）。

优化器有时可以检测到你创建了某些东西而没有使用，或者某些代码的运行对程序没有影响。如果它优化掉了你的测试代码，那么你就会得到不准确的结果。

一个好的微基准测试系统会自动修复此类问题（以及许多其他问题），从而产生合理的结果，但创建这样的系统非常棘手，需要深厚的知识。

16.6.2 介绍 JMH

在撰写本书时，唯一可以产生不错结果的 Java 微基准测试系统是 **Java 微基准测试工具**（Java Microbenchmarking Harness, JMH）。本书的 build.gradle 自动化了 JMH 设置，因此你可以轻松地使用它。

你可以编写 JMH 代码，然后通过命令行运行它，但推荐的方法是让 JMH 系统为你运行测试。build.gradle 文件对此进行了配置，这样就可以使用单个命令来运行 JMH 测试。

JMH 试图使基准测试尽可能简单。举例来说，我们将通过重写 BadMicroBenchmark.java 来使用 JMH。这里唯一需要的注解是 @State 和 @Benchmark。其余的注解只是为了生成更易于理解的输出，或让该示例的基准测试运行得更快（JMH 基准测试通常需要很长时间才能运行完）：

```
// validating/jmh/JMH1.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
// 包括下面三个注解来提高精确度：
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
```

```
public class JMH1 {
  private long[] la;
  @Setup
  public void setup() {
    la = new long[250_000_000];
  }
  @Benchmark
  public void setAll() {
    Arrays.setAll(la, n -> n);
  }
  @Benchmark
  public void parallelSetAll() {
    Arrays.parallelSetAll(la, n -> n);
  }
}
```

“分支”（fork）的默认数量是 10，这意味着每个测试集运行 10 次。为了加快速度，我使用了 @Fork 注解将其减为 1。我还使用 @Warmup 和 @Measurement 注解将预热迭代和测量迭代的数量从默认的 20 次减少到 5 次。尽管这会降低整体的准确度，但与使用默认值的结果几乎相同。你可以试着注释掉 @Warmup、@Measurement 和 @Fork 等注解，看一下使用默认值运行测试是否有明显变化。通常，使用更长时间运行测试时，你应该只会看到误差因子下降，而不会看到结果的变化。

运行基准测试需要一个显式的 gradle 命令（从示例代码的根目录执行）。这可以防止触发其他 gradlew 命令的耗时基准测试：

```
gradlew validating:jmh
```

这需要运行几分钟，具体时间取决于你的机器（如果没有使用注解调整，则可能需要数小时）。控制台输出显示了一个 results.txt 文件的路径，该文件聚合了运行结果。注意，results.txt 包含了本章中所有 jmh 测试的结果：JMH1.java、JMH2.java 和 JMH3.java。

因为输出的是绝对时间，所以结果会因机器和操作系统而异。这里重要的因素不是绝对时间。我们真正想要知道的是一种算法与另一种算法比较起来怎么样，特别是它会快多少或慢多少。如果在自己的机器上运行测试，你会看到不同的数值，但模式是相同的。

我已经在多台机器上测试过这段代码，虽然绝对值因机器而异，但相对值还是相当一致的。我只展示了 results.txt 的相关片段，并对输出进行了编辑，使其更易于理解和在页面上展示。所有测试的 Mode（模式）显示为 avgt，表示“平均时间”。Cnt（测试数量）为 200，不过如果你按照上面的配置运行示例，会看到 Cnt 为 5。Units 的单位是 us/op，表示“每次运行花费的微秒数”，数字越小就表示性能越高。

我还显示了默认的预热次数、测量次数和分支数目的输出。我从示例中删除了相应的注解，以便更准确地运行我的测试（这需要几小时）。无论你如何运行测试，数值的模式看起来应该相同。

下面是 JMH1.java 的结果：

```
Benchmark              Score
JMH1.setAll            196280.2
JMH1.parallelSetAll    195412.9
```

即使对于像 JMH 这样成熟的基准测试工具，基准测试的过程也很重要，你必须小心谨慎地对待。在这里，测试产生了违反直觉的结果：并行版本与非并行版本的 `setAll()` 花费的时间大致相同，而且两者似乎都花费了相当长的时间。

我创建示例时的假设是，如果我们测试数组的初始化，那么使用非常大的数组是有意义的。所以我选择了最大的数组；如果你进行实验，可能会发现当数组大于 2.5 亿 [①] 时，你开始看到内存不足所导致的异常。不过，在如此大的数组上执行大量操作，这可能会破坏内存系统，从而产生意想不到的结果。不管这个假设是否正确，看起来我们**确实**没有测试到想要测试的内容。

考虑一下其他因素。

- C：执行操作的客户线程数。
- P：并行算法使用的并行量。
- N：数组的大小：`10^(2*k)`，其中 `k=1..7` 通常足以覆盖不同的缓存占用场景。
- Q：`setter` 操作的成本。

这个 C/P/N/Q 模型在早期 JDK 8 Lambda 开发期间浮出水面，`Stream` 里的大多数并行操作（与 `parallelSetAll()` 非常相似）符合以下结论。

- `N*Q`（基本上就是工作量）对并行性能至关重要。如果工作量减少，并行算法实际上可能运行得更慢。
- 如果操作对资源的竞争很激烈的话，无论 `N*Q` 有多大，并行性能都不会高。
- 当 `C` 较高时，`P` 的相关性要低得多（大量的外部并行性使得内部并行性变得多余）。此外，在某些情况下，对于 `C` 大小相同的同一客户来说，并行分解带来的成本使得它运行并行算法比运行顺序算法更慢。

① 如果机器配置有限，这个数值可能小得多。

基于这些信息，我们使用不同大小（即 N 的值）的数组重新运行测试：

```
// validating/jmh/JMH2.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH2 {
  private long[] la;
  @Param({
    "1",
    "10",
    "100",
    "1000",
    "10000",
    "100000",
    "1000000",
    "10000000",
    "100000000",
    "250000000"
  })
  int size;

  @Setup
  public void setup() {
    la = new long[size];
  }
  @Benchmark
  public void setAll() {
    Arrays.setAll(la, n -> n);
  }
  @Benchmark
  public void parallelSetAll() {
    Arrays.parallelSetAll(la, n -> n);
  }
}
```

@Param 自动将它的每个值插入到它注解的变量中。值必须是字符串类型，然后会被转换为适当的类型，本例中是 int。

以下是编辑后的结果以及计算出的速度提升（Speedup）：

```
JMH2 Benchmark        Size    Score       % Speedup
setAll                   1    0.001
parallelSetAll           1    0.036       0.028
```

```
setAll                  10    0.005
parallelSetAll          10    3.965       0.001
setAll                 100    0.031
parallelSetAll         100    3.145       0.010
setAll                1000    0.302
parallelSetAll        1000    3.285       0.092
setAll               10000    3.152
parallelSetAll       10000    9.669       0.326
setAll              100000    34.971
parallelSetAll      100000    20.153      1.735
setAll             1000000    420.581
parallelSetAll     1000000    165.388     2.543
setAll            10000000    8160.054
parallelSetAll    10000000    7610.190    1.072
setAll           100000000    79128.752
parallelSetAll   100000000    76734.671   1.031
setAll           250000000    199552.121
parallelSetAll   250000000    191791.927  1.040
```

大约 100 000 个元素的时候，`parallelSetAll()` 开始领先，但随后回落到平均标准。即使它赢了，提升的程度似乎也不足以证明它的存在是合理的。

`setAll()`/`parallelSetAll()` 里运行的计算工作量大小对结果影响会不会很大？在前面的示例中，我们所做的只是将索引的值分配到数组对应的位置，这是最简单的任务之一。所以即使 `N` 变大，`N*Q` 仍然不是那么好，看起来我们没有提供足够的并行机会。（JMH 提供了一种模拟变量 `Q` 的方法，要了解更多信息，请搜索 `Blackhole.consumeCPU`。）

通过使用下面的方法 `f()`，我们让任务更加复杂，从而提高了并行的可能性：

```
// validating/jmh/JMH3.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH3 {
  private long[] la;
  @Param({
    "1",
    "10",
    "100",
    "1000",
```

```
      "10000",
      "100000",
      "1000000",
      "10000000",
      "100000000",
      "250000000"
    })
    int size;

    @Setup
    public void setup() {
      la = new long[size];
    }
    public static long f(long x) {
      long quadratic = 42 * x * x + 19 * x + 47;
      return Long.divideUnsigned(quadratic, x + 1);
    }
    @Benchmark
    public void setAll() {
      Arrays.setAll(la, n -> f(n));
    }
    @Benchmark
    public void parallelSetAll() {
      Arrays.parallelSetAll(la, n -> f(n));
    }
  }
```

`f()` 提供了更复杂和更耗时的操作。现在，我们并不是简单地将索引赋值到其相应的位置，`setAll()` 和 `parallelSetAll()` 都有更多的工作要做，这对结果肯定有影响：

```
JMH3 Benchmark        Size    Score        % Speedup
setAll                   1    0.012
parallelSetAll           1    0.047        0.255
setAll                  10    0.107
parallelSetAll          10    3.894        0.027
setAll                 100    0.990
parallelSetAll         100    3.708        0.267
setAll                1000    133.814
parallelSetAll        1000    11.747       11.391
setAll               10000    97.954
parallelSetAll       10000    37.259       2.629
setAll              100000    988.475
parallelSetAll      100000    276.264      3.578
setAll             1000000    9203.103
parallelSetAll     1000000    2826.974     3.255
setAll            10000000    92144.951
parallelSetAll    10000000    28126.202    3.276
setAll           100000000    921701.863
parallelSetAll   100000000    266750.543   3.455
setAll           250000000    2299127.273
parallelSetAll   250000000    538173.425   4.272
```

可以看到在数组大小为 1000 左右的时候，`parallelSetAll()` 领先于 `setAll()`。看来 `parallelSetAll()` 的结果在很大程度上取决于计算的复杂性和数组的大小。这正是基准测试的价值，只有使用它们之后，我们才能了解 `setAll()` 和 `parallelSetAll()` 是如何工作的，以及使用场景有哪些等微妙的细节。而通过研究 Javadoc 是看不出来这些的。

大多数时候，简单应用一下 JMH 就能产生良好的结果（你将在本书后面的示例中看到），但在这里可以看到，事情并不总是如此。JMH 网站提供了一些示例来帮助你入门。

16.7 分析与优化

有时你需要检测自己程序的运行时间都花在哪里，查看哪部分的性能可以提高。**分析器**（profiler）可以找到耗时的部分，这样你就可以通过最简单、最明显的方法来加快速度。

分析器会收集各种信息，比如程序的哪些部分消耗内存，以及哪些方法消耗了最多的时间。一些分析器甚至会禁用垃圾收集器来帮助确定内存的分配模式。

分析器对于检测程序中的线程死锁也很有用。

请注意分析和基准测试之间的区别。分析着眼于处理实际数据的完整程序，而基准测试着眼于程序的一个独立片段，通常是为了优化算法。

Java 开发工具包（JDK）安装时附带了一个名为 VisualVM 的可视化分析器。它自动安装在与 javac 相同的目录下，在你的执行路径中应该可以找到它。要启动 VisualVM，控制台命令是：

```
> jvisualvm
```

此命令会打开一个窗口，其中包含了指向帮助信息的链接。

优化指南

- 避免为了性能而牺牲代码可读性。
- 不要孤立地看待性能。权衡付出的努力与获得的好处。
- 程序的大小很重要。性能优化通常只对长时间运行的大型项目有价值。小型项目通常不需要关心性能。
- 让程序先正常工作比努力提高其性能更重要。一旦有了一个能运行的程序，你就可以在必要时使用分析器来提高它的效率。仅当性能是关键因素时，才应该在初始设计 / 开发阶段就考虑性能。

- 不要猜测性能瓶颈在哪里。运行分析器来获取该数据。
- JVM 会优化 `static final` 变量来提高程序速度。因此，程序常量应该声明为 `static` 和 `final` 的。

16.8 样式检查

当整个团队一起在某个项目（特别是开源项目）上工作时，如果每个人都遵循相同的编程风格，这会很有用。这样的话，阅读和理解项目代码时就不会被风格差异所干扰。但是，如果你习惯于不同的样式，则可能很难记住特定项目的所有样式指南。幸运的是，有一些工具可以指出代码中不符合项目风格指南的地方。

比较流行的样式检查器是 Checkstyle。

16.9 静态错误分析

尽管 Java 的静态类型检查会发现基本的语法错误，但额外的分析工具可以发现更复杂的错误，而这些错误是 javac 无法发现的。其中一种工具是 Findbugs。

Findbugs 可能会有许多误报，指出实际上正常工作的代码存在问题。最初查看本书的 Findbugs 输出时，我发现了一些技术上没问题，但能够使我改进代码的报告。如果你正在查找错误，那么在启动调试器之前运行一下 Findbugs 是值得的，因为它可能很快就会找到一些问题，而发现这些问题原本可能需要花费数小时。

16.10 代码审查

单元测试（参见 16.1.1 节）能发现一些重要类别的错误。Checkstyle 和 Findbugs 能执行自动代码审查（code review），从而发现其他问题。最终，我们还需要将人工代码审查添加到这个组合中。代码审查有多种方式，一般由一个人或一组人编写代码，然后让其他人或其他组来阅读和评估。这起初看起来令人生畏，它确实需要情感上的信任，但代码审查的目标绝对不是羞辱或嘲笑任何人，而是发现程序错误。在这方面，代码审查是最成功的方法之一。唉，不过它们通常也被认为“太贵了”（有时这可能是程序员的一个借口，避免审查带来的尴尬）。

代码审查可以作为**结对编程**的一部分，置于代码签入过程中（另一个程序员被自动分配审查新代码的任务），或者与一个小组一起，使用**演练**（walkthrough）的方式，每个人

都阅读代码并讨论它。后一种方法具有共享知识与编程文化的显著优势。

16.11 结对编程

结对编程（pair programming）是两个程序员一起编程的一种实践。通常，一个程序员编写代码，另一个程序员（“观察者”或“导航者”）审查和分析代码，并考虑策略。这就是一种实时代码审查。通常程序员会定期转换角色。

结对编程有很多好处，但最引人注目的两个好处是共享知识和防止信息阻塞。传递信息的最佳方式之一是共同解决问题，我在许多研讨会中使用结对编程，取得了很好的效果（而且，研讨会上的人们也是通过这种方式相互了解的）。两个人结伴工作，继续前进要容易得多，而孤军作战很容易陷入困境。

结对程序员通常对他们的工作表示出了更高的满意度。

结对编程有时很难打动管理人员，他们可能会立即认为两个程序员处理一个问题的效率低于各自处理自己的项目。虽然短期来看这通常是正确的，但结对编程生成的代码质量更高。除了结对编程的其他好处之外，如果长远来看，这会产生更高的生产力。

16.12 重构

技术债务（technical debt）是那些在软件中积累的快速而肮脏的解决方案，它使设计无法理解，代码无法阅读。当你必须进行更改或添加功能时，这尤其是个问题。

重构（refactoring）是技术债务的解毒剂。重构的关键在于，它改进了代码设计、结构和可读性（从而减少了技术债务），但它**不会改变代码的行为**。

因此，这样就很难说服管理层：“我们将投入大量工作，但不会添加任何功能，而且从外部看，完成后也不会有明显变化。但请相信我们，情况会好很多。”不幸的是，管理层意识到重构价值的时候，可能为时已晚：当他们要求“再增加一个功能”时，你只能告诉他们这是不可能的，因为代码库已经积累了太多丑陋的临时解决方案（hack）。如果尝试添加另一个功能，它可能会崩溃，即使你**可以**弄清楚如何去做。

重构的基础

在开始重构代码之前，必须具备以下三个支持系统。

1. 测试（通常来说，最低要求是要有 JUnit 测试），因此你可以确保自己的重构不会

改变代码的行为。

2. 构建自动化，从而容易构建代码并运行所有测试。这样就可以轻松进行小的改动，并验证是否破坏了任何东西。本书使用 Gradle 构建系统，你可以在本书示例中找到 build.gradle 文件。

3. 版本控制，这样就可以随时提供或回退到可工作的代码版本，并跟踪这个过程中的所有操作。本书的示例代码托管在 Github 上，并使用了 `git` 版本控制系统。

没有这三个系统，重构几乎是不可能的。事实上，如果没有这些，构建、维护和添加代码就会立刻成为一个巨大的挑战。令人惊讶的是，许多成功的公司在不使用这三个系统的情况下支撑了很长时间。但是此类公司或早或晚都会遇到严重问题。

16.13 持续集成

在软件开发的早期，人们一次只能管理一个步骤，因此他们相信自己是在“快乐通道”中前进，每个开发阶段都会无缝衔接。这种错觉通常被称为软件开发的“瀑布模型”（The Waterfall Model）。有人告诉我，瀑布模型是他们选择的方法，就好像它是一个可选的工具，而不仅仅是一厢情愿的想法。

在个童话王国里，每一步都会按照制订好的时间表完美、准时地完成，然后下一步就可以开始了。当你到达终点时，所有的部分都会无缝地衔接在一起。瞧！一个可发布产品！

当然，实际上，没有任何事情按计划或安排正常进行。相信它可以正常进行，然后在出问题时更加相信，只会让整个事情变得更糟。否认现实不会产生什么好的结果。

最重要的是，产品本身通常对客户没有价值。有时，一大堆功能完全是在浪费时间，因为这些功能的需求并非来自客户，而是来自其他人。

根据来自流水线的思维方式，每个开发阶段都有自己的团队。上游团队的进度表被传递给下游团队，当开始测试和整合时，这些团队被期望以某种方式赶上进度，当他们不可避免地失败时，就会被认为是“不合格的团队成员”。不可能的时间表和团队间不顺畅的交流相结合，共同创造了一个自我实现的预言：只有最敢拼命的开发人员才愿意做这些工作。

更重要的是，商学院继续培训管理人员如何顺应既有流程，而这个流程源于工业时代的制造理念。培养管理人员的创造力而非鼓励他们从众的商学院仍然非常罕见。

最终，编程队伍里的人们再也无法忍受了，他们开始进行实验。其中一些最初的实验称为“极限编程”，因为它们与工业时代的思维非常不同。当实验结出果实后，这些想法开始看起来像是常识。这些实验演变出了现在显而易见的观点，即把可工作的产品——尽管功能非常少——交付到客户手中，并询问他们：(1) 这是不是他们想要的，(2) 他们是否喜欢这个产品的工作方式，以及 (3) 他们还觉得哪些新功能会有用。然后重新回到开发阶段来迭代新版本。一个版本接一个版本，项目最终发展成真正为客户创造价值的产品。

这完全颠覆了瀑布模型的理念。你不再将产品测试和部署等环节放到“最后一步”。相反，即使对于一开始就几乎没有任何功能的产品（在这种情况下，你可能只是测试安装），从头到尾都必须使用这个流程。这样做可以在开发早期发现更多问题。此外，你不需要做大量的前期整体规划，也不必在无用的功能上浪费时间和金钱，而是会持续与客户沟通反馈。当客户不想要更多功能时，产品就完成了。这节省了大量的时间和金钱，并极大提升了客户满意度。

这种开发方式有许多不同的部分和想法，但当前最重要的术语是**持续集成**（continuous integration, CI）。CI 和产生 CI 的想法之间的区别在于，CI 是一个独特的机械过程，它包含这些想法，是一种定义明确的工作方式。事实上，它的定义如此明确，以至于整个过程都是自动化的。

当前 CI 技术的顶点是**持续集成服务器**（continuous integration server）。这是一台单独的机器或虚拟机，通常是第三方公司托管的完全独立的服务。这些公司通常免费提供基本服务，你如果需要更多的处理器、内存、专用工具或系统等附加功能，则需要付费。CI 服务器一开始是一个完全空白的平台，只有最精简的可用操作系统。这很重要，因为如果你在开发机器上安装了某个东西，可能会很容易就忘记将其包含在你的构建和部署系统中。

就像重构一样，持续集成也有一些基本要求，包括分布式版本控制、构建自动化和自动化测试。CI 服务器通常会绑定到你的版本控制存储库。当 CI 服务器发现存储库有变更时，就会检出最新版本，并开始运行 CI 脚本中指定的过程。这包括安装所有必要的工具和库（请记住，CI 服务器是从一个干净的基本操作系统开始的），因此如果在该过程中有任何问题，你可以发现它们。然后它会执行脚本中指定的任何构建和测试，该脚本使用的命令通常和安装测试过程中手工使用的命令完全相同。无论成功还是失败，CI 服务器都有多种方式向你报告，包括出现在代码存储库中的小徽章。

使用持续集成时，你签入存储库的每个变更都会被从头到尾自动验证。通过这种方式，你可以立即发现是否有问题。更好的一点是，当你准备发布产品的新版本时，不会有任何延迟或任何额外的必要步骤 [能够随时交付，也就是**持续交付**（Continuous Delivery）]。

16.14 总结

“它在我的机器上能正常运行。”“我们不会把你的机器连同它一起发布！”

代码验证不是单一的过程或技术。任何一种方法都只能找到特定类别的错误。作为一名程序员，随着你的持续成长，你了解到的每一种额外的技术都会增加代码的可靠性和稳健性。当你向应用程序添加功能时，代码验证不仅可以在开发过程中发现更多错误，在整个项目生命周期中也能如此。现代开发不仅仅意味着编写代码，而且还意味着融入开发过程中的每一种测试技术——尤其是为适应特定应用程序而创建的自定义工具——可以带来更好、更快、更愉快的开发过程以及更高的价值，并为客户提供更满意的体验。

17

文件

> “在非常难用的文件 I/O 编程存在多年之后，Java 终于简化了读写文件的基本操作。”

进阶卷第 7 章中详细介绍了 I/O 编程那种难用的方式。读过那一章之后，你可能会得出这样的结论：Java 的设计者真是不注重用户体验。打开和读取文件，在大多数语言中是相当常见的操作，但在 Java 中需要编写特别笨拙的代码，而且每次都得查一下，否则根本没人能记住怎么打开一个文件。

Java 7 带来了巨大的改进，似乎设计者们终于听到了用户多年来的呼声。这些新元素被打包放在了 `java.nio.file` 之下，其中 `nio` 中的 `n` 以前是指“new”（新的），现在是指“non-blocking”（非阻塞），`io` 是指“input/output”（**输入 / 输出**）。`java.nio.file` 库最终将 Java 的文件操作提升到了可以与其他编程语言媲美的程度。除此之外，Java 8 还增加了流的功能，使得文件编程更好用了。

本章将研究操作文件的两个基本组件：

1. 文件或目录的路径；
2. 文件本身。

17.1 文件和目录路径

Path 对象代表的是一个文件或目录的路径，它是在不同的操作系统和文件系统之上的抽象。它的目的是，在构建路径时，我们不必注意底层的操作系统，我们的代码不需要重写就能在不同的操作系统上工作。

java.nio.file.Paths 类包含了重载的 static get() 方法，可以接受一个 String 序列，或一个**统一资源标识符**（Uniform Resource Identifier, URI），然后将其转化为一个 Path 对象：

```
// files/PathInfo.java
import java.nio.file.*;
import java.net.URI;
import java.io.File;
import java.io.IOException;

public class PathInfo {
  static void show(String id, Object p) {
    System.out.println(id + p);
  }
  static void info(Path p) {
    show("toString:\n ", p);
    show("Exists: ", Files.exists(p));
    show("RegularFile: ", Files.isRegularFile(p));
    show("Directory: ", Files.isDirectory(p));
    show("Absolute: ", p.isAbsolute());
    show("FileName: ", p.getFileName());
    show("Parent: ", p.getParent());
    show("Root: ", p.getRoot());
    System.out.println("********************");
  }
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    info(Paths.get(
      "C:", "path", "to", "nowhere", "NoFile.txt"));
    Path p = Paths.get("PathInfo.java");
    info(p);
    Path ap = p.toAbsolutePath();
    info(ap);
    info(ap.getParent());
    try {
      info(p.toRealPath());
    } catch(IOException e) {
      System.out.println(e);
```

```
    }
    URI u = p.toUri();
    System.out.println("URI:\n" + u);
    Path puri = Paths.get(u);
    System.out.println(Files.exists(puri));
    File f = ap.toFile(); // 不要被骗了
  }
}
```

```
/* 输出:
Windows 10
toString:
 C:\path\to\nowhere\NoFile.txt
Exists: false
RegularFile: false
Directory: false
Absolute: true
FileName: NoFile.txt
Parent: C:\path\to\nowhere
Root: C:\
******************
toString:
 PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: false
FileName: PathInfo.java
Parent: null
Root: null
******************
toString:
 C:\Git\OnJava8\ExtractedExamples\files\PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Git\OnJava8\ExtractedExamples\files
Root: C:\
******************
toString:
 C:\Git\OnJava8\ExtractedExamples\files
Exists: true
RegularFile: false
Directory: true
Absolute: true
FileName: files
Parent: C:\Git\OnJava8\ExtractedExamples
Root: C:\
```

```
******************
toString:
 C:\Git\OnJava8\ExtractedExamples\files\PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Git\OnJava8\ExtractedExamples\files
Root: C:\
******************
URI:
file:///C:/Git/OnJava8/ExtractedExamples/files/PathInfo.java
true
*/
```

我把这里的 `main()` 中的第一行加到了本章中适合的程序中，以显示操作系统的名称，这样就可以看到不同的操作系统之间的差异了。理想情况下，这些差异相对较少，并被隔离在预期的位置，比如路径分隔符是 / 还是 \。从输出中可以看到，我是在 Windows 上做开发的。

虽然 `toString()` 生成的是路径的完整表示，但是可以看到 `getFileName()` 总是会生成文件的名字。使用 `Files` 工具类(后面会看到更多),我们可以测试文件是否存在,是否为“普通”文件，是否为目录，等等。“Nofile.txt”这个示例表明，可以描述一个并不存在的文件，这允许我们创建一个新的路径。“PathInfo.java”存在于当前目录中，最初它只是没有路径的文件名，但检查其状态得到的结果仍然是存在。一旦将其转换为绝对路径，我们就会得到从“C:”盘开始的完整路径 (这是在安装了 Windows 系统的机器上测试的)。现在它也有一个父目录。

文档中对“真实”路径的定义有点模糊，因为它取决于特定的文件系统。例如，如果文件名的比较不区分大小写，即使路径因为大小写的原因看起来不是完全一样，匹配也会成功。在这样的平台上，`toRealPath()` 返回的 `Path` 会使用实际的大小写。它还会删除任何多余的元素。

在这里，我们看到了文件的 URI 是什么样子的，但 URI 可以用来描述大多数事物，不仅限于文件。然后，我们成功地将 URI 转回到了一个 `Path` 之中。

最后，我们看到的是一个略带欺骗性的东西，那就是调用 `toFile()` 来生成一个 `File` 对象。这看起来可能会得到一个类似文件的东西 (毕竟称为 `File`)，但这个方法之所以存

在，只是为了向后兼容旧的做事方式。在那个世界里，File 实际上意味着一个文件或一个目录——听起来它应该被称为“路径”（path）。这个命名非常草率，也令人困惑，但现在忽略它也没什么问题，因为 java.nio.file 已经存在。

17.1.1 选择 Path 的片段

我们可以轻松获得 Path 对象路径的各个部分。

```
// files/PartsOfPaths.java
import java.nio.file.*;

public class PartsOfPaths {
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    Path p =
      Paths.get("PartsOfPaths.java").toAbsolutePath();
    for(int i = 0; i < p.getNameCount(); i++)
      System.out.println(p.getName(i));
    System.out.println("ends with '.java': " +
      p.endsWith(".java"));
    for(Path pp : p) {
      System.out.print(pp + ": ");
      System.out.print(p.startsWith(pp) + " : ");
      System.out.println(p.endsWith(pp));
    }
    System.out.println("Starts with " + p.getRoot() +
      " " + p.startsWith(p.getRoot()));
  }
}
```

```
/* 输出:
Windows 10
Git
OnJava8
ExtractedExamples
files
PartsOfPaths.java
ends with '.java': false
Git: false : false
OnJava8: false : false
ExtractedExamples: false : false
files: false : false
PartsOfPaths.java: false : true
Starts with C:\ true
*/
```

在 getNameCount() 界定的上限之内，我们可以结合索引使用 getName()，得到一个 Path 的各个部分。Path 也可以生成 Iterator，所以我们也可以使用 for-in 来遍历。请注意，尽管这里的路径确实是以 .java 结尾的，但 endsWith() 的结果是 false。这是因为

endsWith() 比较的是整个路径组件，而不是名字中的一个子串。在 for-in 的代码体内，使用 startsWith() 和 endsWith() 来检查路径的当前片段时，这一点就可以显示出来了。然而，我们看到在对 Path 进行遍历时，并没有包含根目录，只有当我们用根目录来检查 startsWith() 时，才会得到 true。

17.1.2 分析 Path

Files 工具类中包含了一整套用于检查 Path 的各种信息的方法。

```
// files/PathAnalysis.java
import java.nio.file.*;
import java.io.IOException;

public class PathAnalysis {
  static void say(String id, Object result) {
    System.out.print(id + ": ");
    System.out.println(result);
  }
  public static void
  main(String[] args) throws IOException {
    System.out.println(System.getProperty("os.name"));
    Path p =
      Paths.get("PathAnalysis.java").toAbsolutePath();
    say("Exists", Files.exists(p));
    say("Directory", Files.isDirectory(p));
    say("Executable", Files.isExecutable(p));
    say("Readable", Files.isReadable(p));
    say("RegularFile", Files.isRegularFile(p));
    say("Writable", Files.isWritable(p));
    say("notExists", Files.notExists(p));
    say("Hidden", Files.isHidden(p));
    say("size", Files.size(p));
    say("FileStore", Files.getFileStore(p));
    say("LastModified: ", Files.getLastModifiedTime(p));
    say("Owner", Files.getOwner(p));
    say("ContentType", Files.probeContentType(p));
    say("SymbolicLink", Files.isSymbolicLink(p));
    if(Files.isSymbolicLink(p))
      say("SymbolicLink", Files.readSymbolicLink(p));
    if(FileSystems.getDefault()
       .supportedFileAttributeViews().contains("posix"))
      say("PosixFilePermissions",
        Files.getPosixFilePermissions(p));
  }
}
```

```
/* 输出:
Windows 10
Exists: true
Directory: false
Executable: true
Readable: true
RegularFile: true
Writable: true
notExists: false
Hidden: false
size: 1617
FileStore: (C:)
LastModified: :
2021-11-08T00:34:52.693768Z
Owner: GROOT\Bruce (User)
ContentType: text/plain
SymbolicLink: false
*/
```

对于最后的这项测试，在调用 getPosixFilePermissions() 之前必须弄清楚当前的文件系统是否支持 Posix，否则会产生一个运行时异常。

17.1.3 添加或删除路径片段

我们必须能够通过对自己的 Path 对象添加和删除某些路径片段来构建 Path 对象。要去掉 Path 这个基准路径，应该使用 relativize()。要在一个 Path 对象的后面增加路径片段，则应该使用 resolve()（这些方法名让人很难“顾名思义”）。

在下面的示例中，我使用 relativize() 从所有的输出中删除了基准路径，也就是 base 所表示的路径。之所以这么做，一方面是为了演示，另一方面是为了简化输出。只有这个 Path 是绝对路径时，才能将其用作 relativize() 的参数。

这个版本的演示包含了 id，这样就更方便跟踪输出了。

```
// files/AddAndSubtractPaths.java
import java.nio.file.*;
import java.io.IOException;

public class AddAndSubtractPaths {
  static Path base = Paths.get("..", "..", "..")
    .toAbsolutePath()
    .normalize();
  static void show(int id, Path result) {
    if(result.isAbsolute())
      System.out.println("(" + id + ")r " +
        base.relativize(result));
    else
      System.out.println("(" + id + ")  " + result);
    try {
      System.out.println("RealPath: "
        + result.toRealPath());
    } catch(IOException e) {
      System.out.println(e);
    }
  }
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    System.out.println(base);
    Path p = Paths.get("AddAndSubtractPaths.java")
      .toAbsolutePath();
    show(1, p);
    Path convoluted = p.getParent().getParent()
      .resolve("strings")
      .resolve("..")
      .resolve(p.getParent().getFileName());
    show(2, convoluted);
    show(3, convoluted.normalize());

    Path p2 = Paths.get("..", "..");
    show(4, p2);
    show(5, p2.normalize());
```

```
    show(6, p2.toAbsolutePath().normalize());

    Path p3 = Paths.get(".").toAbsolutePath();
    Path p4 = p3.resolve(p2);
    show(7, p4);
    show(8, p4.normalize());

    Path p5 = Paths.get("").toAbsolutePath();
    show(9, p5);
    show(10, p5.resolveSibling("strings"));
    show(11, Paths.get("nonexistent"));
  }
}
```

```
/* 输出:
Windows 10
C:\Git
(1)r OnJava8\ExtractedExamples\files\AddAndSubtractPaths.java
RealPath:
C:\Git\OnJava8\ExtractedExamples\files\AddAndSubtractPaths.java
(2)r OnJava8\ExtractedExamples\files
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(3)r OnJava8\ExtractedExamples\files
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(4)  ..\..
RealPath: C:\Git\OnJava8
(5)  ..\..
RealPath: C:\Git\OnJava8
(6)r OnJava8
RealPath: C:\Git\OnJava8
(7)r OnJava8
RealPath: C:\Git\OnJava8
(8)r OnJava8
RealPath: C:\Git\OnJava8
(9)r OnJava8\ExtractedExamples\files
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(10)r OnJava8\ExtractedExamples\strings
RealPath: C:\Git\OnJava8\ExtractedExamples\strings
(11)  nonexistent
java.nio.file.NoSuchFileException:
C:\Git\OnJava8\ExtractedExamples\files\nonexistent
*/
```

我还增加了对 `toRealPath()` 的进一步测试。除了路径不存在的情况下会抛出异常，它总是会对 `Path` 进行扩展和规范化。

17.2 目录

`Files` 工具类包含了操作目录和文件所需的大部分操作。然而由于某些原因，其中并

没有包括用于删除目录树的工具，所以我们会创建一个，并将其添加到 onjava 库中。

```
// onjava/RmDir.java
package onjava;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.io.IOException;

public class RmDir {
  public static void rmdir(Path dir)
  throws IOException {
    Files.walkFileTree(dir,
      new SimpleFileVisitor<Path>() {
      @Override public FileVisitResult
      visitFile(Path file, BasicFileAttributes attrs)
      throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
      }
      @Override public FileVisitResult
      postVisitDirectory(Path dir, IOException exc)
      throws IOException {
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
      }
    });
  }
}
```

这依赖于 Files.walkFileTree()，这里“walk”的意思是查找每个子目录和文件，也就是遍历。**访问者**（Visitor）设计模式提供了一个访问集合中的每个对象的标准机制，我们需要提供想在每个对象上执行的动作。这个动作取决于我们如何实现 FileVisitor 参数，其中包括如下方法。

- preVisitDirectory()：先在当前目录上运行，然后进入这个目录下的文件和目录。
- visitFile()：在这个目录下的每个文件上运行。
- visitFileFailed()：当文件无法访问时调用。
- postVisitDirectory()：先进入当前目录下的文件和目录（包括所有的子目录），最后在当前目录上运行。

为了让事情更简单，java.nio.file.SimpleFileVisitor 为所有这些方法提供了默认的定义。这样，在匿名内部类中，我们只是用非标准的行为重写了这些方法：visitFile() 删除文件，postVisitDirectory() 删除目录。这两个方法的返回标志都表示应该继续遍历（直到找到我们要找的东西为止）。

作为我们探索创建和填充目录的一部分，现在可以有条件地删除某个现有的目录了。在下面的示例中，makeVariant() 接受了一个基准目录 test，然后通过旋转 parts 列表生成了不同的子目录路径。使用 String.join() 将旋转后的 parts 通过路径分隔符 sep 连接到一起，然后将结果作为 Path 返回。

```
// files/Directories.java
import java.util.*;
import java.nio.file.*;
import onjava.RmDir;

public class Directories {
  static Path test = Paths.get("test");
  static String sep =
    FileSystems.getDefault().getSeparator();
  static List<String> parts =
    Arrays.asList("foo", "bar", "baz", "bag");
  static Path makeVariant() {
    Collections.rotate(parts, 1);
    return Paths.get("test", String.join(sep, parts));
  }
  static void refreshTestDir() throws Exception {
    if(Files.exists(test))
      RmDir.rmdir(test);
    if(!Files.exists(test))
      Files.createDirectory(test);
  }
  public static void
  main(String[] args) throws Exception {
    refreshTestDir();
    Files.createFile(test.resolve("Hello.txt"));
    Path variant = makeVariant();
    // 抛出异常（层次太多了）：
    try {
      Files.createDirectory(variant);
    } catch(Exception e) {
      System.out.println("Nope, that doesn't work.");
    }
    populateTestDir();
    Path tempdir =
      Files.createTempDirectory(test, "DIR_");
    Files.createTempFile(tempdir, "pre", ".non");
    Files.newDirectoryStream(test)
      .forEach(System.out::println);
    System.out.println("*********");
    Files.walk(test).forEach(System.out::println);
  }
  static void populateTestDir() throws Exception {
    for(int i = 0; i < parts.size(); i++) {
      Path variant = makeVariant();
      if(!Files.exists(variant)) {
```

```
          Files.createDirectories(variant);
          Files.copy(Paths.get("Directories.java"),
            variant.resolve("File.txt"));
          Files.createTempFile(variant, null, null);
        }
      }
    }
  }
```

```
/* 输出:
Nope, that doesn't work.
test\bag
test\bar
test\baz
test\DIR_8683707748599240459
test\foo
test\Hello.txt
*********
test
test\bag
test\bag\foo
test\bag\foo\bar
test\bag\foo\bar\baz
test\bag\foo\bar\baz\4316127347949967230.tmp
test\bag\foo\bar\baz\File.txt
test\bar
test\bar\baz
test\bar\baz\bag
test\bar\baz\bag\foo
test\bar\baz\bag\foo\1223263495976065729.tmp
test\bar\baz\bag\foo\File.txt
test\baz
test\baz\bag
test\baz\bag\foo
test\baz\bag\foo\bar
test\baz\bag\foo\bar\6666183168609095028.tmp
test\baz\bag\foo\bar\File.txt
test\DIR_8683707748599240459
test\DIR_8683707748599240459\pre6366626804787365549.non
test\foo
test\foo\bar
test\foo\bar\baz
test\foo\bar\baz\bag
test\foo\bar\baz\bag\4712324129011589115.tmp
test\foo\bar\baz\bag\File.txt
test\Hello.txt
*/
```

首先，`refreshTestDir()` 会检查 `test` 是不是已经存在了。如果是的话，就使用新的 `rmdir()` 工具删除整个目录。后面再检查它是否存在，看上去是多余的步骤，但我想说明

的是，如果我们对一个已经存在的目录调用了 createDirectory()，则会产生异常。

createFile() 用参数 Path 创建了一个空文件，resolve() 将文件名添加到了 test 这个 Path 的末尾。

我们尝试使用 createDirectory() 来创建一个不止一层的路径，但这里抛出了一个异常，因为这个方法只能创建单层目录。

这里把 populateTestDir() 设计成了一个单独的方法，因为它在后面的示例中还会复用。对于每一个变种（即 variant），我们使用 createDirectories() 创建了完整的目录路径，然后将这个文件（即 Directories.java）的副本放到了最后一层目录中，只不过更换了文件名称。之后又添加了一个用 createTempFile() 生成的临时文件。这里通过将最后两个参数设置为 null，让这个方法来生成整个临时文件名。

在调用了 populateTestDir() 之后，我们在 test 下面创建了一个临时目录。注意 createTempDirectory() 只有一个名字前缀选项，这点和 createTempFile() 不同，后者可以同时指定名字的前缀和后缀。我们再次使用 createTempFile() 将一个临时文件放到了新的临时目录下。从输出中可以看到，如果我们没有指定后缀，所生成的文件会自动带上“.tmp”后缀。

为了显示结果，我们首先尝试了 newDirectoryStream()，它似乎有希望，但结果是流中只有 test 目录下的内容，而没有进入更下层的目录。要获得包含整个目录树内容的流，请使用 Files.walk()。

17.3 文件系统

为了完整起见，我们需要一种方式来找出文件系统的其他信息。在这里，我们可以使用静态的 FileSystems 工具来获得“默认”的文件系统，但也可以在一个 Path 对象上调用 getFileSystem() 来获得创建这个路径对象的文件系统。我们可以通过给定的 URI 获得一个文件系统，也可以构建一个新的文件系统（如果操作系统支持的话）。

```
// files/FileSystemDemo.java
import java.nio.file.*;

public class FileSystemDemo {
  static void show(String id, Object o) {
    System.out.println(id + ": " + o);
  }
  public static void main(String[] args) {
```

```
    System.out.println(System.getProperty("os.name"));
    FileSystem fsys = FileSystems.getDefault();
    for(FileStore fs : fsys.getFileStores())
      show("File Store", fs);
    for(Path rd : fsys.getRootDirectories())
      show("Root Directory", rd);
    show("Separator", fsys.getSeparator());
    show("UserPrincipalLookupService",
      fsys.getUserPrincipalLookupService());
    show("isOpen", fsys.isOpen());
    show("isReadOnly", fsys.isReadOnly());
    show("FileSystemProvider", fsys.provider());
    show("File Attribute Views",
      fsys.supportedFileAttributeViews());
  }
}
```

```
/* 输出:
Windows 10
File Store: (C:)
File Store: System Reserved (E:)
File Store: (F:)
File Store: Google Drive (G:)
Root Directory: C:\
Root Directory: D:\
Root Directory: E:\
Root Directory: F:\
Root Directory: G:\
Separator: \
UserPrincipalLookupService:
sun.nio.fs.WindowsFileSystem$LookupService$1@1bd4fdd
isOpen: true
isReadOnly: false
FileSystemProvider:
sun.nio.fs.WindowsFileSystemProvider@55183b20
File Attribute Views: [owner, dos, acl, basic, user]
*/
```

FileSystem 还可以生成一个 WatchService 和一个 PathMatcher。

17.4 监听 Path

WatchService 使我们能够设置一个进程，对某个目录中的变化做出反应。在下面的示例中，delTxtFiles() 作为一个独立的任务运行，它会遍历整个目录树，删除所有名字以 .txt 结尾的文件，WatchService 会对文件的删除做出反应。

```
// files/PathWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
```

```
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class PathWatcher {
  static Path test = Paths.get("test");
  static void delTxtFiles() {
    try {
      Files.walk(test)
        .filter(f ->
          f.toString().endsWith(".txt"))
        .forEach(f -> {
          try {
            System.out.println("deleting " + f);
            Files.delete(f);
          } catch(IOException e) {
            throw new RuntimeException(e);
          }
        });
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public static void
  main(String[] args) throws Exception {
    Directories.refreshTestDir();
    Directories.populateTestDir();
    Files.createFile(test.resolve("Hello.txt"));
    WatchService watcher =
      FileSystems.getDefault().newWatchService();
    test.register(watcher, ENTRY_DELETE);
    Executors.newSingleThreadScheduledExecutor()
      .schedule(
        PathWatcher::delTxtFiles,
        250, TimeUnit.MILLISECONDS);
    WatchKey key = watcher.take();
    for(WatchEvent evt : key.pollEvents()) {
      System.out.println(
        "evt.context(): " + evt.context() +
        "\nevt.count(): " + evt.count() +
        "\nevt.kind(): " + evt.kind());
      System.exit(0);
    }
  }
}
```

```
/* 输出:
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
deleting test\baz\bag\foo\bar\File.txt
deleting test\foo\bar\baz\bag\File.txt
deleting test\Hello.txt
evt.context(): Hello.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

`delTxtFiles()` 中的 `try` 块看上去是冗余的，因为它们都在捕捉相同类型的异常，所以似乎有外部的 `try` 就足够了。然而，由于某些原因（可能是个 bug），Java 要求两者都存在。还要注意，在 `filter()` 中，必须显式地调用 `f.toString()`，否则 `endsWith()` 会比较整个 `Path` 对象，而不是其用字符串表示的名字部分。

一旦从 FileSystem 得到一个 WatchService，我们就将它和由我们感兴趣的事件组成的可变参数列表一起注册给 test 这个 Path。感兴趣的事件可以从 ENTRY_CREATE、ENTRY_DELETE 或 ENTRY_MODIFY 中选择（创建和删除不属于修改）。

因为即将开始的对 watcher.take() 的调用会停掉一切工作，直到某个事情发生才恢复，所以我想让 delTxtFiles() 以并行方式开始运行，以便它能产生我们感兴趣的事件。为了做到这一点，我首先通过调用 Executors.newSingleThreadScheduledExecutor() 获得一个 ScheduledExecutorService，然后调用 schedule()，将所需函数的方法引用以及运行之前应该等待的时间交给它。

然后我们调用 watcher.take()，主线程会在这里等待。当有符合我们目标模式的事情发生时，会返回一个包含 WatchEvent 的 WatchKey。这里演示的三个方法是我们能对 WatchEvent 做的所有事情了。

通过输出看看会发生什么。尽管我们正在删除名字是以 .txt 结尾的文件，但在 Hello.txt 被删除之前，WatchService 不会触发。你可能会**认为**，如果我们说“监听这个目录”，它自然会包括整个子树，但这里就要按字面意思理解：它只监听**这个**目录，而**不是**它下面的一切。如果想监听整个目录树，则必须在整个树的每个子目录上设置一个 WatchService。

```
// files/TreeWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class TreeWatcher {
  static void watchDir(Path dir) {
    try {
      WatchService watcher =
        FileSystems.getDefault().newWatchService();
      dir.register(watcher, ENTRY_DELETE);
      Executors.newSingleThreadExecutor().submit(() -> {
        try {
          WatchKey key = watcher.take();
          for(WatchEvent evt : key.pollEvents()) {
            System.out.println(
              "evt.context(): " + evt.context() +
              "\nevt.count(): " + evt.count() +
              "\nevt.kind(): " + evt.kind());
            System.exit(0);
          }
```

```
          } catch(InterruptedException e) {
            return;
          }
        });
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
    public static void
    main(String[] args) throws Exception {
      Directories.refreshTestDir();
      Directories.populateTestDir();
      Files.walk(Paths.get("test"))
        .filter(Files::isDirectory)
        .forEach(TreeWatcher::watchDir);
      PathWatcher.delTxtFiles();
    }
  }
```

```
/* 输出:
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
evt.context(): File.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

watchDir() 方法在其参数上放了一个关注 ENTRY_DELETE 事件的 WatchService，同时启动了一个独立的进程来监控这个 WatchService。这里没有通过 schedule() 方法让任务推迟到以后再运行，而是通过 submit() 让它现在就运行。我们遍历整个目录树，并在每个子目录上应用 watchDir()。现在当我们运行 delTxtFiles() 时，其中一个 WatchService 检测到了第一个删除操作。

17.5 查找文件

到目前为止，要查找文件的话，我们一直在使用相当笨拙的方法，即在 Path 上调用 toString()，然后使用 String 的各种操作来查看结果。其实 java.nio.file 有一个更好的解决方案：PathMatcher。可以通过在 FileSystem 对象上调用 getPathMatcher() 来获得一个 PathMatcher，并传入我们感兴趣的模式。模式有两个选项：glob 和 regex。glob 更简单，但实际上非常强大，可以解决很多问题。如果问题更为复杂，可以使用 regex，下一章会解释。

这里使用 glob 来查找所有文件名以 .tmp 或 .txt 结尾的 Path。

```
// files/Find.java
// {ExcludeFromGradle}
import java.nio.file.*;

public class Find {
  public static void
  main(String[] args) throws Exception {
```

```
    Path test = Paths.get("test");
    Directories.refreshTestDir();
    Directories.populateTestDir();
    // 创建一个目录，而不是文件：
    Files.createDirectory(test.resolve("dir.tmp"));

    PathMatcher matcher = FileSystems.getDefault()
      .getPathMatcher("glob:**/*.{tmp,txt}");
    Files.walk(test)
      .filter(matcher::matches)
      .forEach(System.out::println);
    System.out.println("***************");

    PathMatcher matcher2 = FileSystems.getDefault()
      .getPathMatcher("glob:*.tmp");
    Files.walk(test)
      .map(Path::getFileName)
      .filter(matcher2::matches)
      .forEach(System.out::println);
    System.out.println("***************");

    Files.walk(test) // 只查找文件
      .filter(Files::isRegularFile)
      .map(Path::getFileName)
      .filter(matcher2::matches)
      .forEach(System.out::println);
  }
}
```

```
/* 输出：
test\bag\foo\bar\baz\5208762845883213974.tmp
test\bag\foo\bar\baz\File.txt
test\bar\baz\bag\foo\7918367201207778677.tmp
test\bar\baz\bag\foo\File.txt
test\baz\bag\foo\bar\8016595521026696632.tmp
test\baz\bag\foo\bar\File.txt
test\dir.tmp
test\foo\bar\baz\bag\5832319279813617280.tmp
test\foo\bar\baz\bag\File.txt
***************
5208762845883213974.tmp
7918367201207778677.tmp
8016595521026696632.tmp
dir.tmp
5832319279813617280.tmp
***************
5208762845883213974.tmp
7918367201207778677.tmp
8016595521026696632.tmp
5832319279813617280.tmp
*/
```

在 matcher 中，glob 表达式开头的 **/ 表示“所有子目录”，如果你想匹配的不仅仅是以基准目录为结尾的 Path，那么它是必不可少的，因为它匹配的是完整路径，直到找到你想要的结果。单个的 * 代表的是“任何东西”，然后是一个英文句点，再后面的花括号表示的是一系列的可能性——我们正在查找任何以 .tmp 或 .txt 结尾的东西。我们可以在 getPathMatcher() 的文档中找到更多细节。

matcher2 只是使用了 *.tmp，通常不会匹配到任何东西，但添加 map() 操作后会将完整路径减少到只剩最后的名字。

注意在这两种情况下，dir.tmp 都出现在了输出中，尽管它是目录而非文件。如果只想寻找文件，必须像最后的 Files.walk() 那样对它们进行过滤。

17.6 读写文件

目前为止，我们可以做的只是对路径和目录的操作。现在来看看如何操作文件本身的内容。

如果一个文件是“小”的，针对“小”的某种定义（这只是意味着“对你来说运行得足够快，并且不会耗尽内存”），java.nio.file.Files 类包含了方便读写文本文件和二进制文件的工具函数。

Files.readAllLines() 可以一次性读入整个文件（这也是“小”文件的重要性），生成一个 List<String>。我们将再次使用 streams/Cheese.dat 作为示例文件。

```
// files/ListOfLines.java
import java.util.*;
import java.nio.file.*;

public class ListOfLines {
  public static void
  main(String[] args) throws Exception {
    Files.readAllLines(
      Paths.get("../streams/Cheese.dat"))
      .stream()
      .filter(line -> !line.startsWith("//"))
      .map(line ->
        line.substring(0, line.length()/2))
      .forEach(System.out::println);
  }
}
```

```
/* 输出:
Not much of a cheese
Finest in the
And what leads you
Well, it's
It's certainly uncon
*/
```

注释行被跳过了，其余的内容只打印了一半。看看这有多简单：只需要把一个 Path 对象交给 readAllLines()（过去要凌乱得多）。readAllLines() 有一个重载的版本，还包含一个 Charset 参数，用来确定文件的 Unicode 编码。

Files.write() 也被重载了，可以将 byte 数组或任何实现了 Iterable 接口的类的对象（还包括一个 Charset 选项）写入文件。

```
// files/Writing.java
import java.util.*;
import java.nio.file.*;

public class Writing {
  static Random rand = new Random(47);
  static final int SIZE = 1000;
  public static void
  main(String[] args) throws Exception {
    // 将字节写入一个文件：
    byte[] bytes = new byte[SIZE];
    rand.nextBytes(bytes);
    Files.write(Paths.get("bytes.dat"), bytes);
    System.out.println("bytes.dat: " +
      Files.size(Paths.get("bytes.dat")));

    // 将实现了 Iterable 接口的类的对象写入一个文件：
    List<String> lines = Files.readAllLines(
      Paths.get("../streams/Cheese.dat"));
    Files.write(Paths.get("Cheese.txt"), lines);
    System.out.println("Cheese.txt: " +
      Files.size(Paths.get("Cheese.txt")));
  }
}
```

```
/* 输出：
bytes.dat: 1000
Cheese.txt: 199
*/
```

我们使用 Random 创建了 1000 个随机的 byte，可以看到生成的文件大小是 1000。

这里是将一个 List 对象写到了文件中，但是任何实现了 Iterable 接口的类的对象都是可以的。

如果文件大小是个问题怎么办？可能是以下情况之一：

1. 这个文件非常大，如果一次性读取整个文件，可能会耗尽内存；
2. 我们只需要文件的一部分就能得到想要的结果，所以读取整个文件是在浪费时间。

Files.lines() 可以很方便地将一个文件变为一个由行组成的 Stream。

```
// files/ReadLineStream.java
import java.nio.file.*;
```

```
public class ReadLineStream {
  public static void
  main(String[] args) throws Exception {
    Files.lines(Paths.get("PathInfo.java"))
      .skip(13)
      .findFirst()
      .ifPresent(System.out::println);
  }
}
```

```
/* 输出：
    show("RegularFile", Files.isRegularFile(p));
*/
```

这是将本章的第一个示例流化了，跳过了 13 行，取得下一行并打印。

如果把文件当作一个由行组成的**输入**流来处理，那么 `Files.lines()` 非常有用，但是如果我们想在一个流中完成读取、处理和写入，那该怎么办呢？这就需要稍微复杂些的代码了。

```
// files/StreamInAndOut.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class StreamInAndOut {
  public static void main(String[] args) {
    try(
      Stream<String> input =
        Files.lines(Paths.get("StreamInAndOut.java"));
      PrintWriter output =
        new PrintWriter("StreamInAndOut.txt")
    ) {
      input
        .map(String::toUpperCase)
        .forEachOrdered(output::println);
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

因为我们是在同一个块中执行的所有操作，所以两个文件可以在相同的 `try-with-resources` 块中打开。`PrintWriter` 是一个旧式的 `java.io` 类，允许我们“打印”到一个文件，所以它是这个应用的理想选择。如果看一下 StreamInAndOut.txt，会发现里面的内容确实是全部大写的。

17.7 小结

Java 中的文件和目录操作，本章已做了相当全面的介绍，但库中仍然有一些我们没讲到的特性，请务必研究一下 `java.nio.file` 的文档，特别是 `java.nio.file.Files` 的。

Java 7 和 Java 8 大幅改进了处理文件和目录的库。如果你刚开始使用 Java，那你很幸运。过去它的使用体验是如此不友好，以至于我都确信 Java 的设计者只是认为文件操作还没有重要到需要简化的地步。这个问题不仅使初学者头疼，也使指导初学者学习这门语言的人头疼。我不明白为什么要花这么长时间来解决这个明显的问题，但无论如何，Java 在这方面终于改进了，我很高兴。现在处理文件很容易，甚至很有趣，以前我们是从来不会这样说的。

18

字符串

> “字符串操作可以说是计算机编程中最常见的行为之一。”

在 Java 大展身手的 Web 系统中更是如此。String 类可以说是该语言中使用最频繁的，在本章中，我们将更深入地研究它，以及一些和它相关的类和工具。

18.1 不可变的字符串

String 类的对象是不可变的。如果查看它的 JDK 文档你就会发现，该类中每个看起来似乎会修改 String 值的方法，实际上都创建并返回了一个全新的 String 对象，该对象包含了修改的内容。而原始的 String 则保持不变。

看看下面的代码：

```
// strings/Immutable.java

public class Immutable {
  public static String upcase(String s) {
    return s.toUpperCase();
```

```
  }
  public static void main(String[] args) {
    String q = "howdy";
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
  }
}
```

```
/* 输出:
howdy
HOWDY
howdy
*/
```

当 q 被传递给 upcase() 时，实际上传递的是 q 对象引用的一个副本。此引用所指向的对象只存在于单一的物理位置中。在传递时被复制的只是引用。

在 upcase() 里，参数 s 只存活于这个方法的方法体里。当 upcase() 运行完成后，局部引用 s 就会消失。upcase() 返回执行的结果：一个指向新字符串的引用。我们通过将原来字符串的每个字符设置为大写，从而得到了这个新字符串。传递进来的原始字符串对象则原封不动地保留了下来。

这种行为一般来说正是我们想要的。例如：

```
String s = "asdf";
String x = Immutable.upcase(s);
```

你真的想让 upcase() 方法**修改**传入的参数吗？对于代码的读者来说，参数一般是给方法提供信息的，而不是要被修改的。这种不变性是一个重要的保证，因为它使代码更易于编写和理解。

18.2 重载 + 与 StringBuilder

String 对象是不可变的，因此我们可以根据需要为特定的 String 设置多个别名。因为 String 是只读的，指向它的任何引用都不可能改变它的值，所以引用之间不会相互影响。

不变性可能会带来效率问题。一个典型的例子是操作符 +，它针对 String 对象做了重载。操作符重载意味着在与特定类一起使用时，相应的操作具有额外的意义。（应用于 String 的 + 和 += 是 Java 中仅有的被重载的操作符，Java 不允许程序员重载其他操作符 [①]。）

① C++ 允许程序员随意重载操作符。一般来说这个过程很复杂 [请参阅《C++ 编程思想》（第 2 版）第 10 章]，因此 Java 设计者认为操作符重载是一个“糟糕”的特性，不应该包含在 Java 中。他们最终没有实现这个功能，不过这个决定也没有多么糟糕。不过具有讽刺意味的是，Java 中使用操作符重载会比 C++ 中容易很多。在 Python 和 C# 中可以看到这一点，它们都具有垃圾收集和简单的操作符重载。

+ 操作符可以用来拼接字符串：

```
// strings/Concatenation.java

public class Concatenation {
  public static void main(String[] args) {
    String mango = "mango";
    String s = "abc" + mango + "def" + 47;
    System.out.println(s);
  }
}
```

```
/* 输出：
abcmangodef47
*/
```

想象一下这段代码**可能的**工作原理。字符串 "abc" 可以有一个方法 append()，它创建了一个新的 String 对象，其中包含 "abc" 和 mango 拼接后的内容。随后新的 String 对象添加了 "def" 后，会创建另一个新的 String，依此类推。

这当然行得通，但它需要创建许多 String 对象来组合这个新的 String，这样就有了一堆 String 类型的中间对象需要被垃圾收集。我怀疑 Java 设计者一开始尝试过这种方法（这也是软件设计中的一个教训——除非在代码中尝试并完成了一些工作，否则你对这个系统一无所知）。他们肯定也发现了这种实现在性能上令人难以接受。

要想知道真正发生了什么，可以使用 JDK 自带的 javap 工具反编译上述代码。命令如下：

```
javap -c Concatenation
```

-c 标志表示生成 JVM 字节码。在去掉我们不感兴趣的部分并进行了一些修改后，以下是相关的字节码：

```
public static void main(java.lang.String[]);
 Code:
  Stack=2, Locals=3, Args_size=1
  0:  ldc #2; //String mango
  2:  astore_1
  3:  new #3; //class StringBuilder
  6:  dup
  7:  invokespecial #4; //StringBuilder."<init>":()
  10: ldc #5; //String abc
  12: invokevirtual #6; //StringBuilder.append:(String)
  15: aload_1
  16: invokevirtual #6; //StringBuilder.append:(String)
  19: ldc #7; //String def
  21: invokevirtual #6; //StringBuilder.append:(String)
  24: bipush 47
  26: invokevirtual #8; //StringBuilder.append:(I)
  29: invokevirtual #9; //StringBuilder.toString:()
```

```
32: astore_2
33: getstatic #10; //Field System.out:PrintStream;
36: aload_2
37: invokevirtual #11; //PrintStream.println:(String)
40: return
```

如果你有汇编语言的经验，以上代码看起来可能很熟悉——像 dup 和 invokevirtual 这样的语句相当于 JVM 的汇编语言。如果你从未见过汇编语言，也不要担心——需要注意的重点部分是编译器对 java.lang.StringBuilder 类的引入。我们的代码中没有用到 StringBuilder，但编译器还是决定使用它，因为它的效率更高。

在这里，编译器创建了一个 StringBuilder 对象来构建字符串，并为每个字符串调用了一次 append()，总共四次。最后，它调用了 toString() 来生成结果，并将其存为 s（使用 astore_2 实现）。

你或许会认为可以随意使用 String，反正编译器会对字符串的使用进行优化。在这样想之前，先让我们更仔细地看看编译器在做什么。下面是一个以两种不同方式生成 String 对象的示例：直接使用 String，以及手动使用 StringBuilder 编码。

```
// strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
  public String implicit(String[] fields) {
    String result = "";
    for(String field : fields) {
      result += field;
    }
    return result;
  }
  public String explicit(String[] fields) {
    StringBuilder result = new StringBuilder();
    for(String field : fields) {
      result.append(field);
    }
    return result.toString();
  }
}
```

现在运行 javap -c WhitherStringBuilder，你会看到这两个方法对应的字节码（我已经删除了不必要的细节）。首先是 implicit() 方法：

```
public java.lang.String implicit(java.lang.String[]);
0: ldc           #2  // String
2: astore_2
3: aload_1
```

```
4: astore_3
5: aload_3
6: arraylength
7: istore        4
9: iconst_0
10: istore        5
12: iload         5
14: iload         4
16: if_icmpge     51
19: aload_3
20: iload         5
22: aaload
23: astore        6
25: new           #3  // StringBuilder
28: dup
29: invokespecial #4  // StringBuilder."<init>"
32: aload_2
33: invokevirtual #5  // StringBuilder.append:(String)
36: aload         6
38: invokevirtual #5  // StringBuilder.append:(String;)
41: invokevirtual #6  // StringBuilder.toString:()
44: astore_2
45: iinc          5, 1
48: goto          12
51: aload_2
52: areturn
```

注意 16: 和 48:，它们一起形成了一个循环。16: 对栈上的操作数进行“大于或等于的整数比较”，并在循环完成时跳转到 51:。48: 会返回到循环的开头 12:。注意，StringBuilder 构造发生在这个循环的**内部**，这意味着每次循环时，你都会得到一个新的 StringBuilder 对象。

下面是 explicit() 的字节码：

```
public java.lang.String explicit(java.lang.String[]);
0: new           #3  // StringBuilder
3: dup
4: invokespecial #4  // StringBuilder."<init>"
7: astore_2
8: aload_1
9: astore_3
10: aload_3
11: arraylength
12: istore        4
14: iconst_0
15: istore        5
17: iload         5
19: iload         4
21: if_icmpge     43
```

```
24: aload_3
25: iload         5
27: aaload
28: astore        6
30: aload_2
31: aload         6
33: invokevirtual #5  // StringBuilder.append:(String)
36: pop
37: iinc          5, 1
40: goto          17
43: aload_2
44: invokevirtual #6  // StringBuilder.toString:()
47: areturn
```

不仅循环的代码更短更简单，而且该方法只创建了一个 StringBuilder 对象。显式使用 StringBuilder 时，如果知道字符串可能有多大，你还可以预先分配它的大小，这样就不会不断地重新分配缓冲区了。

因此，当创建 toString() 方法时，如果操作很简单，通常可以依赖编译器，让它以合理的方式自行构建结果。但是如果涉及循环，**并且对性能也有一定要求**，那就需要在 toString() 中显式使用 StringBuilder 了，如下所示：

```
// strings/UsingStringBuilder.java
import java.util.*;
import java.util.stream.*;

public class UsingStringBuilder {
  public static String string1() {
    Random rand = new Random(47);
    StringBuilder result = new StringBuilder("[");
    for(int i = 0; i < 25; i++) {
      result.append(rand.nextInt(100));
      result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("]");
    return result.toString();
  }
  public static String string2() {
    String result = new Random(47)
      .ints(25, 0, 100)
      .mapToObj(Integer::toString)
      .collect(Collectors.joining(", "));
    return "[" + result + "]";
  }
  public static void main(String[] args) {
    System.out.println(string1());
    System.out.println(string2());
```

```
  }
}
```

```
/* 输出:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*/
```

在 string1() 中，最终的结果是用 append() 语句对每一部分进行拼接而成的。如果你想走捷径，执行诸如 append(a + ": " + c) 之类的操作，编译器就会介入，并再次开始创建更多的 StringBuilder 对象。如果不确定使用哪种方法，你可以随时运行 javap 来仔细斟酌。

StringBuilder 提供了丰富而全面的方法，包括 insert()、replace()、substring() 甚至 reverse()，但我们通常使用的只有 append() 和 toString()。注意，在添加右方括号之前，可以调用 delete() 来删除最后一个逗号和空格。

string2() 使用了 Stream，生成的代码更加赏心悦目。实际上，Collectors.joining() 内部使用 StringBuilder 实现，所以使用这种方式不会有任何损失！

StringBuilder 是在 Java 5 中引入的。在此之前，Java 使用 StringBuffer，它是线程安全的（参考进阶卷第 5 章），因此成本也明显更高。使用 StringBuilder 进行字符串操作会更快。

18.3 无意识的递归

和其他类一样，Java 的标准集合最终也是从 Object 继承而来的，所以它们也包含了一个 toString() 方法。这个方法在集合中被重写，这样它生成的结果字符串就能表示容器自身，以及该容器持有的所有对象。以 ArrayList.toString() 为例，它会遍历 ArrayList 的元素并为每个元素调用 toString() 方法：

```
// strings/ArrayListDisplay.java
import java.util.*;
import java.util.stream.*;
import generics.coffee.*;

public class ArrayListDisplay {
  public static void main(String[] args) {
    List<Coffee> coffees =
      Stream.generate(new CoffeeSupplier())
```

```
        .limit(10)
        .collect(Collectors.toList());
    System.out.println(coffees);
  }
}
```

```
/* 输出:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4,
Breve 5, Americano 6, Latte 7, Cappuccino 8, Cappuccino
9]
*/
```

如果你希望 `toString()` 打印对象的内存地址，使用 `this` 来实现似乎是合情合理的：

```
// strings/InfiniteRecursion.java
// 意外的递归
// {ThrowsException}
// {VisuallyInspectOutput} 抛出很长的异常栈
import java.util.*;
import java.util.stream.*;

public class InfiniteRecursion {
  @Override public String toString() {
    return
      " InfiniteRecursion address: " + this + "\n";
  }
  public static void main(String[] args) {
    Stream.generate(InfiniteRecursion::new)
      .limit(10)
      .forEach(System.out::println);
  }
}
```

如果创建一个 `InfiniteRecursion` 对象，然后将其打印出来，你会得到一个很长的异常栈。如果将 `InfiniteRecursion` 对象放在 `ArrayList` 中，然后如上所示的那样打印这个 `ArrayList`，也会有同样的结果。之所以这样，是因为字符串的**自动类型转换**。当如下代码运行时：

```
"InfiniteRecursion address: " + this
```

编译器看到一个 `String` 后面跟着一个 `+` 和一个不是 `String` 的东西，它就试图将这个 `this` 转换为一个 `String`。这个转换是通过调用 `toString()` 来完成的，而这样就产生了一个递归调用。

如果真的想打印对象的地址，可以直接调用 `Object` 的 `toString()` 方法来实现。因此，这里不应该使用 `this`，而应该使用 `super.toString()`。

18.4 对字符串的操作

下面是可操作 String 对象的大多数方法。重载的方法单独汇总在一行中（见表 18-1）：

表 18-1

方　法	参数，重载	用　途
构造器	重载版本包括：默认构造器；参数分别为 String、StringBuilder、StringBuffer、char 数组、byte 数组的构造器	创建 String 对象
length()	—	String 中的 Unicode 代码单元（code units）个数
charAt()	int 索引	String 中某个位置的 char
getChars()、getBytes()	要复制的开始和结束索引，要复制到的目标数组，以及目标数组的起始索引	将 char 或 byte 复制到外部数组中
toCharArray()	—	生成一个 char[]，包含了 String 中的所有字符
equals()、equalsIgnoreCase()	要与之比较的 String	对两个 String 的内容进行相等性检查。如果内容相等，则返回 true
compareTo()、compareToIgnoreCase()	要与之比较的 String	按字典顺序比较 String 的内容，结果可能为负数、零或正数。注意大写和小写不相等
contains()	要查找的 CharSequence	如果参数包含在 String 中，则结果为 true
contentEquals()	用来比较的 CharSequence 或 StringBuffer	如果该 String 与参数的内容完全匹配，则结果为 true
isEmpty()	—	返回一个 boolean 值，表明该 String 的长度是否为 0
regionMatches()	该 String 的索引偏移量，参数 String 和它的索引偏移量，以及要比较的长度。重载方法添加了“忽略大小写”功能	返回一个 boolean 值，表明该区域是否匹配
startsWith()	该字符串可能的前缀 String。重载方法在参数列表中增加了偏移量	返回一个 boolean 值，表明该 String 是否以参数字符串开头
endsWith()	该字符串可能的后缀 String	返回一个 boolean 值，表明参数字符串是否为后缀
indexOf()、lastIndexOf()	重载版本包括：char、char 和起始索引；String、String 和起始索引	如果在此 String 中找不到该参数，则返回 -1；否则返回参数开始的索引。lastIndexOf() 则从后向前搜索
matches()	一个正则表达式	返回一个 boolean 值，表明此 String 是否与给定的正则表达式匹配
split()	一个正则表达式。可选的第二个参数是要进行的最大分割数	根据正则表达式拆分 String。返回结果数组
join()（在 Java 8 中引入）	分隔符以及要合并的元素。通过将元素与分隔符连接在一起，生成一个新的 String	将片段合并成一个由分隔符分隔的新 String
substring()（还有 subSequence()）	重载版本包括：起始索引；起始索引 + 结束索引	返回一个 String 对象，包含了指定的字符集合

（续）

方　法	参数，重载	用　途
concat()	要拼接的 String	返回一个新的 String 对象，其中包含了原始 String 的字符，后跟参数的字符
replace()	要搜索的旧字符，以及用来替换的新字符。也可以用来在 CharSequence 之间进行替换	返回一个替换后的新 String 对象。如果没有匹配，则使用旧的 String
replaceFirst()	用来进行搜索的正则表达式，以及用来替换的新 String	返回替换后的新 String 对象
replaceAll()	用来进行搜索的正则表达式，以及用来替换的新 String	返回替换后的新 String 对象
toLowerCase()、toUpperCase()	—	返回一个新的 String 对象，所有字母的大小写都发生了相应的变化。如果没有任何更改，则使用旧的 String
trim()	—	返回一个删除了两端空白字符的新 String 对象。如果没有任何更改，则使用旧的 String
valueOf()（静态）	重载版本包括：Object、char[]、char[] 和偏移量还有计数、boolean、char、int、long、float、double	返回一个 String，里面包含了参数的字符表示
intern()	—	为每个唯一的字符序列生成一个独一无二的 String 引用
format()	格式字符串（内含要被替换的格式说明符）、参数	生成格式化后的结果 String

当需要更改内容时，每个 String 方法都会小心地返回一个新的 String 对象。如果不需要更改内容，该方法就返回一个对原始 String 的引用。这节省了存储和开销。

本章后面将讲解涉及**正则表达式**的 String 方法。

18.5　格式化输出

Java 5 提供了类似 C 语言中 printf() 语句风格的格式化输出，这是一个用户期待已久的特性。它不仅简化了控制输出功能的代码，而且还为 Java 开发人员提供了对输出格式和对齐的强大控制。

18.5.1　printf()

C 语言的 printf() 并不像 Java 中那样组装字符串，而是采用单个**格式化字符串**，然后将值插入其中，并进行格式化。printf() 方法没有使用重载的 + 操作符（C 没有重载）来拼接引号内的文本和变量，而是使用特殊的占位符来表示数据的位置。要插入到格式化字符串的参数用逗号分隔排列。例如：

```
System.out.printf("Row 1: [%d %f]%n", x, y);
```

在运行时，x 的值会插入到 %d 的位置，y 的值会插入到 %f 的位置。这些占位符称为**格式说明符**，除了表明插入值的位置外，它们还说明了插入变量的类型以及如何对其进行格式化。例如，上面的 %d 表示 x 是一个整数，%f 表示 y 是一个浮点数（float 或 double）。

18.5.2 System.out.format()

Java 5 引入的 format() 方法可用于 PrintStream 或 PrintWriter 对象（可以在进阶卷第 7 章中了解更多信息），因此也可直接用于 System.out。format() 方法模仿了 C 语言的 printf() 方法。如果你比较怀旧的话，也可以直接使用 printf() 方法，它用起来很方便，内部直接调用了 format() 来实现。下面是一个简单的例子：

```
// strings/SimpleFormat.java

public class SimpleFormat {
  public static void main(String[] args) {
    int x = 5;
    double y = 5.332542;
    // 旧的方式:
    System.out.println("Row 1: [" + x + " " + y + "]");
    // 新的方式:
    System.out.format("Row 1: [%d %f]%n", x, y);
    // 或者:
    System.out.printf("Row 1: [%d %f]%n", x, y);
  }
}
```

```
/* 输出:
Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
*/
```

format() 和 printf() 是等价的。它们都只需要一个格式化字符串，后面跟着参数，其中每个参数都对应一个格式说明符。

String 类也有一个静态的 format() 方法，它会产生一个格式化字符串。

18.5.3 Formatter 类

Java 中所有的格式化功能都由 java.util 包里的 Formatter 类处理。你可以将 Formatter 视为一个转换器，将格式化字符串和数据转换为想要的结果。当创建一个 Formatter 对象时，你可以将信息传递给构造器，来表明希望将结果输出到哪里：

```
// strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
```

```
  private String name;
  private Formatter f;
  public Turtle(String name, Formatter f) {
    this.name = name;
    this.f = f;
  }
  public void move(int x, int y) {
    f.format("%s The Turtle is at (%d,%d)%n",
      name, x, y);
  }
  public static void main(String[] args) {
    PrintStream outAlias = System.out;
    Turtle tommy = new Turtle("Tommy",
      new Formatter(System.out));
    Turtle terry = new Turtle("Terry",
      new Formatter(outAlias));
    tommy.move(0,0);
    terry.move(4,8);
    tommy.move(3,4);
    terry.move(2,5);
    tommy.move(3,3);
    terry.move(3,3);
  }
}
```

```
/* 输出:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*/
```

%s 格式说明符表示这是一个 String 参数。

tommy 相关的输出都转到了 System.out，而 terry 相关的输出则转到 System.out 的别名。构造器被重载以获取一系列的输出位置，但最有用的是 PrintStream（如上例所示）、OutputStream 和 File。你将在进阶卷第 7 章中了解到更多相关信息。

18.5.4 格式说明符

如果想要在插入数据时控制间距和对齐方式，你需要更详细的格式说明符。下面是它的通用语法：

```
%[argument_index$][flags][width][.precision]conversion
```

一般来说，你必须控制一个字段的最小长度。这可以通过指定 width 来实现。Formatter 会确保这个字段至少达到一定数量的字符宽度，必要时会使用空格来填充。默认情况下，数据是右对齐的，但这可以通过使用一个 - 标记来改变。

和 width 相对的是 precision（精度），用于指定字段长度的最大值。width 适用于所有进行转换的数据类型，并且对每种类型来说其行为方式都一样，而 precision 对不同的

类型则有不同的含义。对字符串而言，precision 指定了字符串的最大输出字符数。对浮点数而言，precision 指定了要显示的小数位数（默认为 6 位），小数位数如果太多则舍入，如果太少则末尾补零。整数没有小数部分，因此 precision 不适用于此。如果对整数应用 precision，则会抛出异常。

在下面这个简单示例中，我们使用格式说明符来打印购物收据。它使用了生成器模式，你可以创建一个起始对象，然后向其中添加内容，最后使用 build() 方法来生成最终结果：

```
// strings/ReceiptBuilder.java
import java.util.*;

public class ReceiptBuilder {
  private double total = 0;
  private Formatter f =
    new Formatter(new StringBuilder());
  public ReceiptBuilder() {
    f.format(
      "%-15s %5s %10s%n", "Item", "Qty", "Price");
    f.format(
      "%-15s %5s %10s%n", "----", "---", "-----");
  }
  public void add(String name, int qty, double price) {
    f.format("%-15.15s %5d %10.2f%n", name, qty, price);
    total += price * qty;
  }
  public String build() {
    f.format("%-15s %5s %10.2f%n", "Tax", "",
      total * 0.06);
    f.format("%-15s %5s %10s%n", "", "", "-----");
    f.format("%-15s %5s %10.2f%n", "Total", "",
      total * 1.06);
    return f.toString();
  }
  public static void main(String[] args) {
    ReceiptBuilder receiptBuilder =
      new ReceiptBuilder();
    receiptBuilder.add("Jack's Magic Beans", 4, 4.25);
    receiptBuilder.add("Princess Peas", 3, 5.1);
    receiptBuilder.add(
      "Three Bears Porridge", 1, 14.29);
    System.out.println(receiptBuilder.build());
  }
}
```

```
/* 输出：
Item             Qty      Price
----             ---      -----
Jack's Magic Be    4       4.25
Princess Peas      3       5.10
Three Bears Por    1      14.29
Tax                        2.80
                           -----
Total                     49.39
*/
```

将 StringBuilder 传递给 Formatter 构造器后，它就有了一个构建 String 的地方。还可以使用构造器参数将其发送到标准输出甚至文件里。

Formatter 用相当简洁的符号提供了对间距和对齐的强大控制。在这个程序中，为了生成适当的间距，格式化字符串被重复利用了多次。

18.5.5 Formatter 转换

表 18-2 中展示了一些最常见的转换字符。

表 18-2

字 符	效 果
d	整数类型（十进制）
c	Unicode 字符
b	Boolean 值
s	字符串
f	浮点数（十进制）
e	浮点数（科学记数法）
x	整数类型（十六进制）
h	哈希码（十六进制）
%	字面量 "%"

下面的示例显示了这些转换的实际效果：

```
// strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
  public static void main(String[] args) {
    Formatter f = new Formatter(System.out);

    char u = 'a';
    System.out.println("u = 'a'");
    f.format("s: %s%n", u);
    // f.format("d: %d%n", u);
    f.format("c: %c%n", u);
    f.format("b: %b%n", u);
    // f.format("f: %f%n", u);
    // f.format("e: %e%n", u);
    // f.format("x: %x%n", u);
    f.format("h: %h%n", u);

    int v = 121;
    System.out.println("v = 121");
    f.format("d: %d%n", v);
    f.format("c: %c%n", v);
    f.format("b: %b%n", v);
    f.format("s: %s%n", v);
    // f.format("f: %f%n", v);
    // f.format("e: %e%n", v);
    f.format("x: %x%n", v);
    f.format("h: %h%n", v);

    BigInteger w = new BigInteger("50000000000000");
    System.out.println(
      "w = new BigInteger(\"50000000000000\")");
    f.format("d: %d%n", w);
```

```
        // f.format("c: %c%n", w);
        f.format("b: %b%n", w);
        f.format("s: %s%n", w);
        // f.format("f: %f%n", w);
        // f.format("e: %e%n", w);
        f.format("x: %x%n", w);
        f.format("h: %h%n", w);

        double x = 179.543;
        System.out.println("x = 179.543");
        // f.format("d: %d%n", x);
        // f.format("c: %c%n", x);
        f.format("b: %b%n", x);
        f.format("s: %s%n", x);
        f.format("f: %f%n", x);
        f.format("e: %e%n", x);
        // f.format("x: %x%n", x);
        f.format("h: %h%n", x);

        Conversion y = new Conversion();
        System.out.println("y = new Conversion()");
        // f.format("d: %d%n", y);
        // f.format("c: %c%n", y);
        f.format("b: %b%n", y);
        f.format("s: %s%n", y);
        // f.format("f: %f%n", y);
        // f.format("e: %e%n", y);
        // f.format("x: %x%n", y);
        f.format("h: %h%n", y);

        boolean z = false;
        System.out.println("z = false");
        // f.format("d: %d%n", z);
        // f.format("c: %c%n", z);
        f.format("b: %b%n", z);
        f.format("s: %s%n", z);
        // f.format("f: %f%n", z);
        // f.format("e: %e%n", z);
        // f.format("x: %x%n", z);
        f.format("h: %h%n", z);
    }
}
```

```
/* 输出:
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("50000000000000")
d: 50000000000000
b: true
s: 50000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@19e0bfd
h: 19e0bfd
z = false
b: false
s: false
h: 4d5
*/
```

对于特定的变量类型而言，被注释掉的代码行是一个无效的转换。如果执行它们的话，会触发异常。

请注意，转换字符 b 适用于上述的每个变量。尽管对所有参数类型都有效，但它的行为可能与预期的不同。对于 boolean 基本类型或 Boolean 对象来说，相应的结果就是 true 或 false。但是，对于任何其他参数，只要参数类型不是 null，结果总是 true。即使是数值 0，

其转换结果依然是 true，而 0 在许多语言（包括 C）中是与 false 同义的，因此在对非布尔类型使用这种转换时一定要小心。

Formatter 类还有很多晦涩的转换类型和其他格式说明符选项，你可以在它的 JDK 文档中阅读这些内容。

18.5.6 String.format()

Java 5 还借鉴了 C 语言中用来创建字符串的 sprintf()，提供了 String.format() 方法。它是一个静态方法，参数与 Formatter 类的 format() 方法完全相同，但返回一个 String。当只调用一次 format() 时，这个方法用起来就很方便：

```
// strings/DatabaseException.java

public class DatabaseException extends Exception {
  public DatabaseException(int transactionID,
    int queryID, String message) {
    super(String.format("(t%d, q%d) %s", transactionID,
        queryID, message));
  }
  public static void main(String[] args) {
    try {
      throw new DatabaseException(3, 7, "Write failed");
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

```
/* 输出：
DatabaseException: (t3, q7) Write failed
*/
```

String.format() 的方法内部所做的，其实就是实例化一个 Formatter，并将传入的参数直接传递给它。和手动来做这些相比，使用这个方法通常更方便，代码也更清晰易读。

十六进制转储工具

作为 String.format() 的第二个示例，让我们将二进制文件中的字节格式化为十六进制。下面这个小工具通过使用 String.format()，将一个二进制字节数组按可读的十六进制格式打印出来：

```
// strings/Hex.java
// {java onjava.Hex}
package onjava;
import java.io.*;
import java.nio.file.*;
```

```
public class Hex {
  public static String format(byte[] data) {
    StringBuilder result = new StringBuilder();
    int n = 0;
    for(byte b : data) {
      if(n % 16 == 0)
        result.append(String.format("%05X: ", n));
      result.append(String.format("%02X ", b));
      n++;
      if(n % 16 == 0) result.append("\n");
    }
    result.append("\n");
    return result.toString();
  }
  public static void
  main(String[] args) throws Exception {
    if(args.length == 0)
      // 通过输出这个类文件来测试
      System.out.println(format(
        Files.readAllBytes(Paths.get(
          "build/classes/java/main/onjava/Hex.class"))));
    else
      System.out.println(format(
        Files.readAllBytes(Paths.get(args[0]))));
  }
}
```

```
/* 输出（前 6 行）：
00000: CA FE BA BE 00 00 00 34 00 61 0A 00 05 00 31 07
00010: 00 32 0A 00 02 00 31 08 00 33 07 00 34 0A 00 35
00020: 00 36 0A 00 0F 00 37 0A 00 02 00 38 08 00 39 0A
00030: 00 3A 00 3B 08 00 3C 0A 00 02 00 3D 09 00 3E 00
00040: 3F 08 00 40 07 00 41 0A 00 42 00 43 0A 00 44 00
00050: 45 0A 00 14 00 46 0A 00 47 00 48 07 00 49 01 00
                    ...
*/
```

为了打开和读取二进制文件，我们使用了第 17 章中介绍的另一个实用工具：`Files.readAllBytes()`，它以 byte 数组的形式返回了整个文件。

18.6 新特性：文本块

JDK 15 最终添加了**文本块**（text block），这是从 Python 语言借鉴而来的一个特性。我们使用三引号来表示包含换行符的文本块。文本块可以让我们更轻松地创建多行文本：

```
// strings/TextBlocks.java
// {NewFeature} 从 JDK 15 开始
// 诗歌：Antigonish 作者：Hughes Mearns
```

```
public class TextBlocks {
  public static final String OLD =
    "Yesterday, upon the stair,\n" +
    "I met a man who wasn't there\n" +
    "He wasn't there again today\n" +
    "I wish, I wish he'd go away...\n" +
    "\n" +
    "When I came home last night at three\n" +
    "The man was waiting there for me\n" +
    "But when I looked around the hall\n" +
    "I couldn't see him there at all!\n";

  public static final String NEW = """
    Yesterday, upon the stair,
    I met a man who wasn't there
    He wasn't there again today
    I wish, I wish he'd go away...

    When I came home last night at three
    The man was waiting there for me
    But when I looked around the hall
    I couldn't see him there at all!
    """;
  public static void main(String[] args) {
    System.out.println(OLD.equals(NEW));
  }
}
```

```
/* 输出:
true
*/
```

OLD 展示了处理多行字符串的传统方式，里面有很多换行符 \n 和符号 +。NEW 消除了这些符号，提供了更好、更易读的语法。

注意开头的 """ 后面的换行符会被自动去掉，块中的公用缩进也会被去掉，所以 NEW 的结果没有缩进。如果想要保留缩进，那就移动最后的 """ 来产生所需的缩进，如下所示：

```
// strings/Indentation.java
// {NewFeature} 从 JDK 15 开始

public class Indentation {
  public static final String NONE = """
          XXX
          YYY
          """; // 没有缩进
  public static final String TWO = """
          XXX
          YYY
        """;   // 产生 2 个缩进
  public static final String EIGHT = """
          XXX
          YYY
```

```
/* 输出:
XXX
YYY
  XXX
  YYY
        XXX
        YYY
*/
```

```
    """;          // 产生 8 个缩进
    public static void main(String[] args) {
      System.out.print(NONE);
      System.out.print(TWO);
      System.out.print(EIGHT);
    }
  }
```

为了支持文本块，String 类里添加了一个新的 formatted() 方法：

```
// strings/DataPoint.java
// {NewFeature} 从 JDK 15 开始

public class DataPoint {
  private String location;
  private Double temperature;
  public DataPoint(String loc, Double temp) {
    location = loc;
    temperature = temp;
  }
  @Override public String toString() {
    return """
    Location: %s
    Temperature: %.2f
    """.formatted(location, temperature);
  }
  public static void main(String[] args) {
    var hill = new DataPoint("Hill", 45.2);
    var dale = new DataPoint("Dale", 65.2);
    System.out.print(hill);
    System.out.print(dale);
  }
}
```

```
/* 输出：
Location: Hill
Temperature: 45.20
Location: Dale
Temperature: 65.20
*/
```

formatted() 是一个成员方法，而不是一个像 String.format() 那样的单独的静态函数，所以除了文本块之外，也可以把它用于普通字符串，它用起来更好、更清晰，因为可以将它直接添加到字符串的后面。

文本块的结果是一个常规的字符串，所以其他字符串能做的事情，它也可以做。

18.7 正则表达式

很久以前，**正则表达式**就已经整合进 sed 和 awk 等标准的 UNIX 工具集里，以及 Python 和 Perl 等语言中（有些人认为这也是 Perl 能获得成功的主要原因）。Java 中的字符串操作以前委托给了 String、StringBuffer 和 StringTokenizer 类，与正则表达式相比，它们的功能相对简单。

正则表达式是强大而灵活的文本处理工具。利用正则表达式，我们可以通过编程的方式，构建复杂的文本模式，从而在输入的字符串中进行查找。一旦发现了这些匹配的模式，你就可以随心所欲地对它们进行处理。尽管刚开始接触正则表达式时，其语法可能令人生畏，但它提供了一种紧凑且动态的语言，可以用完全通用的方式来解决字符串处理、匹配和选择，以及编辑、验证等各种问题。

18.7.1 基础

正则表达式用通用术语来描述字符串，因此你可以这样说："如果字符串中包含这些内容，那么它就符合我的搜索条件。"例如，要表示一个数前面可能有也可能没有减号，可以在减号后面加上一个问号，如下所示：

```
-?
```

如果想描述一个整数，你可以说它是一个或多个数字。在正则表达式中，数字用 \d 来描述。在 Java 中，\\ 的意思是"我正在插入一个正则表达式反斜杠，所以后面的字符有特殊含义"。例如，要表示一个数字，你的正则表达式字符串应该是 \\d。要插入普通的反斜杠，你可以用 \\\\。但是，换行符和制表符之类的符号只使用一个反斜杠：\n\t[①]。

为了显示普通字符串反斜杠和正则表达式反斜杠之间的区别，我们将使用简单的 String.matches() 函数。matches() 的参数是一个正则表达式，它会作用于调用 matches() 的字符串。我们首先定义普通的字符串反斜杠 one、two 和 three。可以看到在一个普通的字符串中你需要两个反斜杠来生成一个反斜杠：

```
// strings/BackSlashes.java

public class BackSlashes {
  public static void main(String[] args) {
    String one = "\\";
    String two = "\\\\";
    String three = "\\\\\\";
    System.out.println(one);
    System.out.println(two);
    System.out.println(three);
    System.out.println(one.matches("\\\\"));
    System.out.println(two.matches("\\\\\\\\"));
    System.out.println(three.matches("\\\\\\\\\\\\"));
  }
}
```

```
/* 输出：
\
\\
\\\
true
true
true
*/
```

① Java 一开始设计的时候并没有考虑到正则表达式，所以后来只能硬塞进这种笨拙的语法。

而在正则表达式中，我们需要使用**四个**反斜杠才能与单个反斜杠匹配。因此，要匹配字符串中的 3 个反斜杠，我们需要在正则表达式中使用 12 个反斜杠。

如果要在表达式里表示“前面有一个或多个”，请使用 +。因此，如果想说“前面可能有一个减号，后面跟着一个或多个数字”，对应的表达式是这样的：

```
-?\\d+
```

使用正则表达式的最简单方式，就是直接使用内置在 String 类中的功能。例如，我们可以查看一个 String 是否与上面的这个正则表达式匹配：

```
// strings/IntegerMatch.java

public class IntegerMatch {
  public static void main(String[] args) {
    System.out.println("-1234".matches("-?\\d+"));
    System.out.println("5678".matches("-?\\d+"));
    System.out.println("+911".matches("-?\\d+"));
    System.out.println("+911".matches("(-|\\+)?\\d+"));
  }
}
```

```
/* 输出：
true
true
false
true
*/
```

前两个字符串匹配，但第三个字符串以 + 开头。这是一个合法的符号，但这样一来，字符串里的数字就与正则表达式不匹配了。所以我们需要一种方式来表达“可以以 + 或 - 开头”。在正则表达式中，括号可以将表达式分组，竖线 | 表示“或”操作。因此有：

```
(-|\\+)?
```

这个正则表达式表示对应部分的字符串可以是 -、+ 或什么都没有（这是因为后面跟着 ?）。+ 字符在正则表达式中具有特殊意义，所以在表达式中必须用 \\ 转义成普通字符。

String 类中内置了一个很有用的正则表达式工具 split()，它可以“围绕给定正则表达式的匹配项来拆分字符串”。

```
// strings/Splitting.java
import java.util.*;

public class Splitting {
  public static String knights =
    "Then, when you have found the shrubbery, " +
    "you must cut down the mightiest tree in the " +
    "forest...with... a herring!";
```

```
  public static void split(String regex) {
    System.out.println(
      Arrays.toString(knights.split(regex)));
  }
  public static void main(String[] args) {
    split(" "); // 参数里不一定要有正则字符
    split("\\W+"); // 不是单词的字符
    split("n\\W+"); // n 后面跟着一个不是单词的字符
  }
}
```

```
/* 输出:
[Then,, when, you, have, found, the, shrubbery,, you,
must, cut, down, the, mightiest, tree, in, the,
forest...with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you,
must, cut, down, the, mightiest, tree, in, the, forest,
with, a, herring]
[The, whe, you have found the shrubbery, you must cut
dow, the mightiest tree i, the forest...with... a
herring!]
*/
```

首先，请注意可以使用普通字符作为正则表达式——正则表达式不必包含特殊字符，如 `split()` 的第一次调用所示，它只使用空格进行了拆分。

第二次和第三次调用的 `split()` 使用了 `\\W`，表示非单词字符（小写版本的 `\\w` 表示单词字符）。在第二种情况下标点符号被删除了。第三次调用的 `split()` 表示“字母 n 后跟一个或多个非单词字符”。用来拆分的模式，即字符串中与正则表达式匹配的部分，并不会出现在结果中。

`String.split()` 有一个重载版本可以限制拆分发生的次数。

还可以使用正则表达式进行替换，你可以只替换第一个匹配的项，也可以全部替换：

```
// strings/Replacing.java

public class Replacing {
  static String s = Splitting.knights;
  public static void main(String[] args) {
    System.out.println(
      s.replaceFirst("f\\w+", "located"));
    System.out.println(
      s.replaceAll("shrubbery|tree|herring","banana"));
  }
}
```

```
/* 输出：
Then, when you have located the shrubbery, you must cut
down the mightiest tree in the forest...with... a
herring!
Then, when you have found the banana, you must cut down
the mightiest banana in the forest...with... a banana!
*/
```

第一个表达式要匹配的是，字母 f 后跟一个或多个单词字符（注意这次 w 是小写的）。它只替换找到的第一个匹配项，因此单词“found”被替换为“located”。

第二个表达式是竖线分隔的三个单词，竖线表示“或”操作，因此它会匹配这三个单词中的任意一个，并替换找到的所有匹配项。

接下来你将会看到，非字符串的正则表达式具有更强大的替换工具——例如，可以调用方法来执行替换。如果正则表达式会被多次使用，非字符串正则表达式会具有更高的效率。

18.7.2 创建正则表达式

你可以从正则表达式的所有构造项中，选取一个子集开始学习（见表 18-3）。完整的列表可以在 Pattern 类的 JDK 文档中找到，它属于 java.util.regex 包。

表 18-3

构 造 项	生成结果
B	指定字符 B
\xhh	具有十六进制值 0xhh 的字符
\uhhhh	十六进制表示为 0xhhhh 的 Unicode 字符
\t	制表符（Tab）
\n	换行
\r	回车
\f	换页
\e	转义（escape）

当你想要定义符合某种模式的字符时，正则表达式的威力才真正开始显现。表 18-4 中是一些定义某种模式字符的典型方式，以及部分预先定义好的字符模式。

表 18-4

符 号	结 果
.	任何字符
[abc]	a、b 或 c 中的任何一个字符（与 a|b|c 相同）
[^abc]	a、b 或 c 之外的任何字符（否定）
[a-zA-Z]	a~z 或 A~Z 的任何字符（范围）
[abc[hij]]	a、b、c、h、i、j 中的任何一个字符（与 a|b|c|h|i|j 相同，求并集）
[a-z&&[hij]]	h、i 或 j 中的任何一个字符（求交集）
\s	一个空白字符（空格、制表符、换行符、换页、回车）
\S	非空白字符 ([^\s])
\d	数字 ([0-9])
\D	非数字 ([^0-9])
\w	一个单词字符 ([a-zA-Z_0-9])
\W	一个非单词字符 ([^\w])

这里的示例只列出了部分常用的模式，你可以把 java.util.regex.Pattern 的 JDK 文档页面添加为书签，这样就能轻松访问所有可能的正则表达式模式了。表 18-5 中列出了一些逻辑操作符。

表 18-5

逻辑操作符	含 义
XY	X 后跟 Y
X|Y	X 或 Y
(X)	一个捕获组（capturing group）。你可以在后面的表达式中用 \i 来引用第 *i* 个捕获组

表 18-6 中是不同的边界匹配器。

表 18-6

边界匹配器	含 义
^	行首
$	行尾
\b	单词的边界
\B	非单词的边界
\G	前一个匹配的结束

例如，下面的每个正则表达式都能成功匹配字符序列 "Rudolph"：

```
// strings/Rudolph.java

public class Rudolph {
  public static void main(String[] args) {
    for(String pattern : new String[]{
```

```
        ,
        "[rR]udolph",
        "[rR][aeiou][a-z]ol.*",
        "R.*" })
        System.out.println("Rudolph".matches(pattern));
    }
}
```

```
/* 输出:
true
true
true
true
*/
```

你的目标不应该是创建最难理解的正则表达式，而应该是创建能完成工作的最简单的正则表达式。在编写新的正则表达式时，你会发现自己经常需要参考旧的代码。

18.7.3 量词

量词（quantifier）描述了一个正则表达式模式处理输入文本的方式（见表 18-7）。

贪婪型

量词默认是贪婪的，除非另有设置。贪婪型表达式会为模式找到尽可能多的匹配项，这可能会导致问题。我们经常会犯的一个错误就是，认为自己的模式只会匹配第一个可能的字符组，但实际上它是贪婪的，会一直持续执行，直到找到最大的匹配字符串。

勉强型

用问号来指定，这个量词会匹配满足模式所需的最少字符数。也称为**惰性匹配**、**最小匹配**、**非贪婪匹配**或**不贪婪匹配**。

占有型

目前这种类型仅适用于 Java（不适用于其他语言），这是一个更高级的功能，因此你可能不会立即使用它。当正则表达式应用于字符串时，它会生成许多状态，以便在匹配失败时可以回溯。占有型量词不会保留这些中间状态，因此可以防止回溯。这还可以防止正则表达式运行时失控，并使其执行更有效。

表 18-7

贪 婪 型	勉 强 型	占 有 型	匹 配
X?	X??	X?+	X，一个或一个都没有
X*	X*?	X*+	X，零个或多个
X+	X+?	X++	X，一个或多个
X{n}	X{n}?	X{n}+	X，正好 n 个
X{n,}	X{n,}?	X{n,}+	X，至少 n 个
X{n,m}	X{n,m}?	X{n,m}+	X，至少 n 个但不超过 m 个

请记住，表达式 X 经常需要用括号括起来，才能按你希望的方式工作。例如：

```
abc+
```

这个表达式看起来会匹配一个或多个 abc 序列，如果将它应用到输入字符串 abcabcabc，也的确会得到三个匹配。但是，这个表达式**实际上**表示“ab 后跟一个或多个 c”。要匹配一个或多个完整字符串 abc，就必须这样表示：

```
(abc)+
```

使用正则表达式时很容易上当。它是位于 Java 之上的一门正交语言。

CharSequence

CharSequence 接口对 CharBuffer、String、StringBuffer 或 StringBuilder 等类进行了抽象，给出了一个字符序列的通用定义：

```
interface CharSequence {
  char charAt(int i);
  int length();
  CharSequence subSequence(int start, int end);
  String toString();
}
```

上述类实现了这个接口。许多正则表达式操作使用了 CharSequence 参数。

18.7.4 Pattern 和 Matcher

通常我们会编译正则表达式对象，而不是使用功能相当有限的 String。为此，只需要导入 java.util.regex 包，然后使用静态方法 Pattern.compile() 编译正则表达式即可。它会根据自己的字符串参数来生成一个 Pattern 对象。你可以通过调用 matcher() 方法来使用这个 Pattern，对传递的 String 进行搜索。matcher() 方法会产生一个 Matcher 对象，它有一组可供选择的操作（所有这些都记录在了 java.util.regex.Matcher 的 JDK 文档中）。例如，replaceAll( ) 会用其参数替换所有的匹配项。

作为第一个示例，下面的类可以针对输入字符串来测试正则表达式。第一个命令行参数是要匹配的输入字符串，然后是一个或多个应用于输入字符串的正则表达式。在 UNIX/Linux 环境下，命令行中的正则表达式必须用引号包围起来。当构建正则表达式时，可以使用该程序来测试，以查看它是否能够产生预期的匹配行为 [①]。

① 互联网上有更多有用和复杂的正则表达式辅助工具。

```
// strings/TestRegularExpression.java
// 简单的正则表达式演示
// {java TestRegularExpression
// abcabcabcdefabc "abc+" "(abc)+" }
import java.util.regex.*;

public class TestRegularExpression {
  public static void main(String[] args) {
    if(args.length < 2) {
      System.out.println(
        "Usage:\njava TestRegularExpression " +
        "characterSequence regularExpression+");
      System.exit(0);
    }
    System.out.println("Input: \"" + args[0] + "\"");
    for(String arg : args) {
      System.out.println(
        "Regular expression: \"" + arg + "\"");
      Pattern p = Pattern.compile(arg);
      Matcher m = p.matcher(args[0]);
      while(m.find()) {
        System.out.println(
          "Match \"" + m.group() + "\" at positions " +
          m.start() + "-" + (m.end() - 1));
      }
    }
  }
}
```

```
/* 输出：
Input: "abcabcabcdefabc"
Regular expression: "abcabcabcdefabc"
Match "abcabcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
*/
```

还可以尝试在命令行中添加 `"(abc){2,}"`。

`Pattern` 对象表示正则表达式的编译版本。如前面的示例所示，你可以使用编译后的 `Pattern` 对象的 `matcher()` 方法，加上输入的字符串作为参数，来生成一个 `Matcher` 对象。`Pattern` 还提供了一个静态方法：

```
static boolean matches(String regex, CharSequence input)
```

它会检查正则表达式 `regex` 与整个 `CharSequence` 类型的输入参数 `input` 是否匹配。

Pattern 里还有一个 split() 方法，它根据 regex 的匹配结果来分割字符串，并返回分割后的字符串数组。

我们通常调用 Pattern.matcher() 方法，并传入一个输入字符串作为参数，来生成一个 Matcher 对象。然后就可以使用 Matcher 对象提供的方法，来评估不同类型的匹配，从而获得匹配成功与否的结果：

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

如果模式能匹配整个输入字符串，则 matches() 方法返回匹配成功；如果输入字符串的起始部分与模式匹配，则 lookingAt() 方法返回匹配成功。

1. find()

Matcher.find() 可以在应用它的 CharSequence 中查找多个匹配。例如：

```
// strings/Finding.java
import java.util.regex.*;

public class Finding {
  public static void main(String[] args) {
    Matcher m = Pattern.compile("\\w+")
      .matcher(
        "Evening is full of the linnet's wings");
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    int i = 0;
    while(m.find(i)) {
      System.out.print(m.group() + " ");
      i++;
    }
  }
}
```

```
/* 输出：
Evening is full of the linnet s wings
Evening vening ening ning ing ng g is is s full full
ull ll l of of f the the he e linnet linnet innet nnet
net et t s s wings wings ings ngs gs s
*/
```

模式 \\w+ 会将输入的字符串拆分为单词。find() 就像一个迭代器，会向前遍历输入的字符串。而另一个版本的 find() 可以接收一个整数参数，来表示搜索开始的字符位置——这个版本的 find() 会将起始搜索位置重置为参数的值，如输出所示。

2. 分组

分组（group）是用括号括起来的正则表达式，后续代码里可以用分组号来调用它们。分组 0 表示整个表达式，分组 1 是第一个带括号的分组，以此类推。因此，下面这个表达式中共有 3 个分组：

```
A(B(C))D
```

分组 0 是 ABCD，分组 1 是 BC，分组 2 是 C。

Matcher 对象提供了一些方法，可以获取与分组相关的信息。

- `public int groupCount()` 返回该匹配器模式中的分组数目。分组 0 不包括在此计数中。
- `public String group()` 返回前一次匹配操作（例如 `find()`）的第 0 个分组（即整个匹配）。
- `public String group(int i)` 返回前一次匹配操作期间给定的分组号。如果匹配成功，但指定的分组未能匹配输入字符串的任何部分，则返回 `null`。
- `public int start(int group)` 返回在前一次匹配操作中找到的分组的起始索引。
- `public int end(int group)` 返回在前一次匹配操作中找到的分组的最后一个字符的索引加 1 的值。

下面是一个示例：

```
// strings/Groups.java
import java.util.regex.*;

public class Groups {
  public static final String POEM =
    "Twas brillig, and the slithy toves\n" +
    "Did gyre and gimble in the wabe.\n" +
    "All mimsy were the borogoves,\n" +
    "And the mome raths outgrabe.\n\n" +
    "Beware the Jabberwock, my son,\n" +
    "The jaws that bite, the claws that catch.\n" +
    "Beware the Jubjub bird, and shun\n" +
    "The frumious Bandersnatch.";
  public static void main(String[] args) {
    Matcher m = Pattern.compile(
      "(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$")
      .matcher(POEM);
    while(m.find()) {
      for(int j = 0; j <= m.groupCount(); j++)
        System.out.print("[" + m.group(j) + "]");
```

```
        System.out.println();
      }
    }
  }
```

```
/* 输出:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the
borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths
outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun]
[The frumious Bandersnatch.][The][frumious
Bandersnatch.][frumious][Bandersnatch.]
*/
```

这里采用了“Jabberwocky”这首诗的第一部分，选自刘易斯·卡罗尔的《爱丽丝镜中奇遇记》。正则表达式模式里有几个带括号的分组，由任意数量的非空白字符（\\S+）和任意数量的空白字符（\\s+）组成。目标是捕获每行的最后三个单词，行尾由 $ 分隔。但在正常情况下是将 $ 与整个输入序列的结尾进行匹配，因此我们必须明确告诉正则表达式注意输入中的换行符。这是通过序列开头的模式标记 (?m) 来完成的（模式标记很快就会介绍）。

3. start() 和 end()

在匹配成功之后，start() 返回本次匹配结果的起始索引，而把本次匹配结果最后一个字符的索引加上 1，就是 end() 的返回值。如果匹配不成功（或在尝试匹配操作之前），这时调用 start() 或 end() 会产生一个 IllegalStateException。下面的示例同时展示了 matches() 和 lookingAt() 的用法 [①]：

```
// strings/StartEnd.java
import java.util.regex.*;

public class StartEnd {
  public static String input =
    "As long as there is injustice, whenever a\n" +
    "Targathian baby cries out, wherever a distress\n" +
    "signal sounds among the stars " +
    "... We'll be there.\n"+
    "This fine ship, and this fine crew ...\n" +
    "Never give up! Never surrender!";
```

① 这里的 input 来自科幻电视剧集 *Galaxy Quest* 中指挥官 Taggart 的演讲。

```
  private static class Display {
    private boolean regexPrinted = false;
    private String regex;
    Display(String regex) { this.regex = regex; }
    void display(String message) {
      if(!regexPrinted) {
        System.out.println(regex);
        regexPrinted = true;
      }
      System.out.println(message);
    }
  }
  static void examine(String s, String regex) {
    Display d = new Display(regex);
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while(m.find())
      d.display("find() '" + m.group() +
        "' start = "+ m.start() + " end = " + m.end());
    if(m.lookingAt()) // 不需要 reset()
      d.display("lookingAt() start = "
        + m.start() + " end = " + m.end());
    if(m.matches()) // 不需要 reset()
      d.display("matches() start = "
        + m.start() + " end = " + m.end());
  }
  public static void main(String[] args) {
    for(String in : input.split("\n")) {
      System.out.println("input : " + in);
      for(String regex : new String[]{"\\w*ere\\w*",
        "\\w*ever", "T\\w+", "Never.*?!"})
        examine(in, regex);
    }
  }
}
```

```
/* 输出:
input : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' start = 11 end = 16
\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' start = 27 end = 35
\w*ever
find() 'wherever' start = 27 end = 35
T\w+
find() 'Targathian' start = 0 end = 10
lookingAt() start = 0 end = 10
```

```
input : signal sounds among the stars ... We'll be
there.
\w*ere\w*
find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+
find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20
lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*/
```

`find()` 会在输入字符串中的任何位置匹配正则表达式，但是对 `lookingAt()` 和 `matches()` 来说，只有正则表达式和输入字符串的开始位置匹配时它们才会成功。`matches()` 仅在**整个**输入字符串都与正则表达式匹配时才会成功，而 `lookingAt()` 则仅在输入字符串的开始部分匹配时才成功。（我不知道 `lookingAt()` 这个方法名称是怎么想出来的，也不知道它指的是什么。这就是为什么代码审查这么重要。）

4. Pattern 标记

`Pattern` 类的 `compile()` 方法还有一个重载版本，它可以接受一个标记参数，来影响匹配行为：

```
Pattern Pattern.compile(String regex, int flag)
```

其中，`flag` 来自 `Pattern` 类中的常量（见表 18-8）。

表 18-8

编译标记	效　果
`Pattern.CANON_EQ`	当且仅当两个字符的完全正则分解匹配时，才认为它们匹配。例如，当指定此标记时，表达式 `\u003F` 将匹配字符串 `?`。默认情况下，匹配不考虑正则的等价性
`Pattern.CASE_INSENSITIVE (?i)`	默认情况下，匹配仅在 US-ASCII 字符集中进行时才不区分大小写。这个标记允许模式匹配时不考虑大小写。可以通过指定 `UNICODE_CASE` 标记，并结合这个标记来在 Unicode 字符集里启用不区分大小写的匹配

（续）

编译标记	效　果
Pattern.COMMENTS (?x)	在这种模式下，空白符被忽略，并且以 # 开头的嵌入注释也会被忽略，直到行尾。UNIX 的行模式也可以通过嵌入的标记表达式来启用
Pattern.DOTALL (?s)	在 dotall 模式下，表达式 . 匹配任何字符，包括换行符。默认情况下，. 表达式不匹配换行符
Pattern.MULTILINE (?m)	在多行模式下，表达式 ^ 和 $ 分别匹配一行的开头和结尾。此外，^ 匹配输入字符串的开头，$ 匹配输入字符串的结尾。默认情况下，这些表达式仅匹配整个输入字符串的开头和结尾
Pattern.UNICODE_CASE (?u)	当指定了这个标记，并且同时启用了 CASE_INSENSITIVE 标记时，不区分大小写的匹配将以符合 Unicode 标准的方式完成。默认情况下，匹配仅在 US-ASCII 字符集中进行时才不区分大小写
Pattern.UNIX_LINES (?d)	这种模式下，在 .、^ 和 $ 的行为里，只有换行符 \n 被识别

上表中列举的标记中，特别有用的是以下几种：

- Pattern.CASE_INSENSITIVE
- Pattern.MULTILINE
- Pattern.COMMENTS（有助于清晰度和 / 或文档记录）

注意，你也可以在正则表达式中直接使用上表中的大多数标记，只需要将左栏括号中的字符插入希望模式生效的位置之前即可。

可以通过“或”操作（|）来组合这些标记，实现多种效果：

```
// strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
  public static void main(String[] args) {
    Pattern p =  Pattern.compile("^java",
      Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
    Matcher m = p.matcher(
      "java has regex\nJava has regex\n" +
      "JAVA has pretty good regular expressions\n" +
      "Regular expressions are in Java");
    while(m.find())
      System.out.println(m.group());
  }
}
```

```
/* 输出：
java
Java
JAVA
*/
```

这将创建一个模式来匹配以“java”“Java”和“JAVA”等字符串开头的行，并且在一个多行集合下尝试匹配其中的每一行（从字符序列的开头直到该字符序列中的换行符为止）。注意，group() 方法只返回匹配的部分。

18.7.5 split()

split() 根据输入的正则表达式来拆分字符串，然后返回拆分后的字符串对象数组。

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

这是一种在通用边界上拆分输入文本的便捷方法：

```
// strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;

public class SplitDemo {
  public static void main(String[] args) {
    String input =
      "This!!unusual use!!of exclamation!!points";
    System.out.println(Arrays.toString(
      Pattern.compile("!!").split(input)));
    // 只执行前 3 次：
    System.out.println(Arrays.toString(
      Pattern.compile("!!").split(input, 3)));
  }
}
```

```
/* 输出：
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*/
```

split() 还提供了另外一种形式，可以限制拆分的次数。

18.7.6 替换操作

正则表达式对于替换文本特别有用。下面是一些可用的方法。

- replaceFirst(String replacement) 用参数 replacement 替换输入字符串的第一个匹配的部分。
- replaceAll(String replacement) 用参数 replacement 替换输入字符串的每个匹配的部分。
- appendReplacement(StringBuffer sbuf, String replacement) 执行逐步替换，并保存到 sbuf 中，而不是像 replaceFirst() 和 replaceAll() 那样分别替换第一个匹配和全部匹配。这是一个**非常**重要的方法，因为你可以调用其他方法来处理或生成 replacement（ replaceFirst() 和 replaceAll() 只能放入固定的字符串 ）。使用此方法，你能够以编程的方式进行分组，从而创建更强大的替换功能。
- 在调用了一次或多次 appendReplacement() 方法后，可以再调用 appendTail(StringBuffer sbuf) 方法，将输入字符串的剩余部分复制到 sbuf。

下面是一个包含了以上所有替换操作的示例。开头的注释文本块就是示例中要被正则表达式提取并处理的输入字符串：

```
// strings/TheReplacements.java
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

/*! Here's a block of text to use as input to
    the regular expression matcher. Note that we
    first extract the block of text by looking for
    the special delimiters, then process the
    extracted block. !*/

public class TheReplacements {
  public static void
  main(String[] args) throws Exception {
    String s = Files.lines(
      Paths.get("TheReplacements.java"))
      .collect(Collectors.joining("\n"));
    // 匹配上面被特地注释掉的文本块：
    Matcher mInput = Pattern.compile(
      "/\\*!(.*)!\\*/", Pattern.DOTALL).matcher(s);
    if(mInput.find())
      s = mInput.group(1); // 被括号捕获的
    // 用一个空格替换两个或多个空格：
    s = s.replaceAll(" {2,}", " ");
    // 删除每行开头的一个或多个空格。必须启用多行模式：
    s = s.replaceAll("(?m)^ +", "");
    System.out.println(s);
    s = s.replaceFirst("[aeiou]", "(VOWEL1)");
    StringBuffer sbuf = new StringBuffer();
    Pattern p = Pattern.compile("[aeiou]");
    Matcher m = p.matcher(s);
    // 一边查找一边替换：
    while(m.find())
      m.appendReplacement(
        sbuf, m.group().toUpperCase());
    // 插入文本的剩余部分：
    m.appendTail(sbuf);
    System.out.println(sbuf);
  }
}
```

```
/* 输出：
Here's a block of text to use as input to
the regular expression matcher. Note that we
first extract the block of text by looking for
the special delimiters, then process the
extracted block.
H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO
thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE
fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr
thE spEcIAl dElImItErs, thEn prOcEss thE
ExtrActEd blOck.
*/
```

我们使用第 17 章中介绍的 Files 类来打开和读取文件。Files.lines() 会生成一个包含多行字符串的 Stream，而 Collectors.joining() 则将参数附加到每行的末尾，然后将它们合并成一个字符串。

mInput 匹配了 /*! 和 !*/ 之间的所有文本（注意用来分组的括号）。然后，将两个或两个以上的空格缩减为一个空格，并删除了每行开头的任何空格（要在所有的行上都执行此操作，而不仅仅是输入的开头，则必须启用多行模式）。这两个替换操作使用的是 String 类自带的 replaceAll()，它和 Matcher 里的 replaceAll() 等效，而且在此处使用起来更方便。注意，因为每个替换操作在程序中都只执行了一次，所以并不需要将它预编译为 Pattern，这样直接使用不会有额外的成本。

replaceFirst() 只对它找到的第一个匹配进行替换。此外，replaceFirst() 和 replaceAll() 中用来替换的字符串只能是固定的文本，因此如果想在每次替换时进行一些处理，它们是无能为力的。此时，可以使用 appendReplacement() 方法，它允许你在执行替换的过程中添加任意数量的代码。在前面的示例中，我们先获取一个 group()，在这里它表示正则表达式匹配到的元音，然后将其设置为大写，最后将结果写入 sbuf。通常，我们逐步执行并替换所有的匹配项，然后调用 appendTail()，但是为了模拟 replaceFirst()（或替换 *n* 次），你可以只执行一次替换，然后调用 appendTail() 将剩余部分放入 sbuf。

在 appendReplacement() 方法的替换字符串中，你还可以通过 \$g 直接引用匹配的某个组，其中 g 就是组号。不过它只能应付简单的处理，无法实现前面的示例中你想要的类似功能。

18.7.7 reset()

可以使用 reset() 方法将现有的 Matcher 对象应用于新的字符序列：

```
// strings/Resetting.java
import java.util.regex.*;

public class Resetting {
  public static void
  main(String[] args) throws Exception {
    Matcher m = Pattern.compile("[frb][aiu][gx]")
      .matcher("fix the rug with bags");
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    m.reset("fix the rig with rags");
    while(m.find())
      System.out.print(m.group() + " ");
```

```
/* 输出：
fix rug bag
fix rig rag
*/
```

```
  }
}
```

没有任何参数的 reset() 会将 Matcher 对象设置到当前序列的起始位置。

18.7.8 正则表达式和 Java I/O

到目前为止，大多数示例将正则表达式应用于静态的字符串。下面这个示例演示了如何用正则表达式来搜索文件中的匹配项。受 UNIX 的 grep 启发，JGrep.java 接受了两个参数：文件名和用来匹配的正则表达式。输出的是每个匹配项及其在该行中的位置。

```
// strings/JGrep.java
// grep 程序的一个简化版
// {java JGrep
// WhitherStringBuilder.java "return|for|String"}
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

public class JGrep {
  public static void
  main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println(
        "Usage: java JGrep file regex");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    Matcher m = p.matcher("");
    // 遍历输入文件的每一行
    Files.readAllLines(Paths.get(args[0])).forEach(
      line -> {
        m.reset(line);
        while(m.find())
          System.out.println(
            m.group() + ": " + m.start());
      }
    );
  }
}
```

```
/* 输出:
String: 18
String: 20
String: 9
String: 25
String: 4
for: 4
String: 8
return: 4
String: 9
String: 25
String: 4
String: 31
for: 4
String: 8
return: 4
String: 20
*/
```

虽然可以为每一行创建一个新的 Matcher 对象，但最好还是创建一个空的 Matcher 对象，然后在遍历时使用 reset() 方法来为 Matcher 对象加载对应的行，最后用 find() 来查找结果。

这里的测试参数是 WhitherStringBuilder.java，程序打开并读取该文件作为输入，然后搜索单词 return、for 或 String。

如果想要更深入地学习正则表达式，可以阅读 Jeffrey E. F. Friedl 的《精通正则表达式》

（第 2 版）。网上也有许多对正则表达式的介绍，在 Perl 和 Python 等语言的文档中通常也可以找到有用的信息。

18.8 扫描输入

到目前为止，从人类可读的文件或标准输入中读取数据还是比较痛苦的。一般的解决方案是读入一行文本，对其进行分词解析，然后使用 Integer、Double 等类里的各种方法来解析数据：

```
// strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
  public static BufferedReader input =
    new BufferedReader(new StringReader(
    "Sir Robin of Camelot\n22 1.61803"));
  public static void main(String[] args) {
    try {
      System.out.println("What is your name?");
      String name = input.readLine();
      System.out.println(name);
      System.out.println("How old are you? " +
        "What is your favorite double?");
      System.out.println("(input: <age> <double>)");
      String numbers = input.readLine();
      System.out.println(numbers);
      String[] numArray = numbers.split(" ");
      int age = Integer.parseInt(numArray[0]);
      double favorite = Double.parseDouble(numArray[1]);
      System.out.format("Hi %s.%n", name);
      System.out.format("In 5 years you will be %d.%n",
        age + 5);
      System.out.format("My favorite double is %f.",
        favorite / 2);
    } catch(IOException e) {
      System.err.println("I/O exception");
    }
  }
}
```

```
/* 输出:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

input 字段使用了来自 java.io 的类，这些类在进阶卷第 7 章中有描述。StringReader 将字符串转换为可读流对象，该对象用于创建 BufferedReader，因为 BufferedReader 有一个 readLine() 方法。这样我们就可以每次从 input 对象里读取一行，就好像它是来自控制台的标准输入一样。

readLine() 方法将获取的每行输入转为 String 对象。当一行数据只对应一个输入时，处理起来还是很简单的，但如果两个输入值在一行上，事情就会变得混乱——我们必须拆分该行，才能分别解析每个输入。在这个例子中，拆分发生在创建 numArray 时。

Java 5 中添加的 Scanner 类大大减轻了扫描输入的负担：

```
// strings/BetterRead.java
import java.util.*;

public class BetterRead {
  public static void main(String[] args) {
    Scanner stdin = new Scanner(SimpleRead.input);
    System.out.println("What is your name?");
    String name = stdin.nextLine();
    System.out.println(name);
    System.out.println(
      "How old are you? What is your favorite double?");
    System.out.println("(input: <age> <double>)");
    int age = stdin.nextInt();
    double favorite = stdin.nextDouble();
    System.out.println(age);
    System.out.println(favorite);
    System.out.format("Hi %s.%n", name);
    System.out.format("In 5 years you will be %d.%n",
      age + 5);
    System.out.format("My favorite double is %f.",
      favorite / 2);
  }
}
```

```
/* 输出:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

Scanner 的构造器可以接受任何类型的输入对象，包括 File 对象、InputStream、String，或者此例里的 Readable（Java 5 引入的一个接口，用于描述“具有 read() 方法的

东西”）。以上示例中的 BufferedReader 就归于这一类。

在 Scanner 中，输入、分词和解析这些操作都被包含在各种不同类型的“next”方法中。一个普通的 next() 返回下一个 String，所有的基本类型（char 除外）以及 BigDecimal 和 BigInteger 都有对应的“next”方法。所有的“next”方法都是**阻塞的**，这意味着它们只有在输入流能提供一个完整可用的数据分词时才会返回。你也可以根据相应的“hasNext”方法是否返回 true，来判断下一个输入分词的类型是否正确。

在 BetterRead.java 中没有针对 IOException 的 try 块。这是因为 Scanner 会假设 IOException 表示输入结束，因此 Scanner 会把 IOException 隐藏起来。不过最近发生的异常可以通过它的 ioException() 方法获得，因此你可以在必要时检查它。

18.8.1 Scanner 分隔符

默认情况下，Scanner 通过空格分割输入数据，但也可以用正则表达式的形式来指定自己的分隔符模式：

```
// strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
  public static void main(String[] args) {
    Scanner scanner = new Scanner("12, 42, 78, 99, 42");
    scanner.useDelimiter("\\s*,\\s*");
    while(scanner.hasNextInt())
      System.out.println(scanner.nextInt());
  }
}
```

```
/* 输出：
12
42
78
99
42
*/
```

此示例使用逗号（由任意数量的空白符包围）作为分隔符，来处理读取的给定字符串。同样的技术也可以用来读取逗号分隔的文件。除了用于设置分隔符模式的 useDelimiter()，还有 delimiter() 方法，它返回了当前用作分隔符的 Pattern 对象。

18.8.2 使用正则表达式扫描

除了扫描预定义的基本类型，你还可以用自己定义的正则表达式模式来扫描，这在扫描更复杂的数据时非常有用。下面这个示例扫描防火墙日志中的威胁数据：

```
// strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
```

```
  static String threatData =
    "58.27.82.161@08/10/2015\n" +
    "204.45.234.40@08/11/2015\n" +
    "58.27.82.161@08/11/2015\n" +
    "58.27.82.161@08/12/2015\n" +
    "58.27.82.161@08/12/2015\n" +
    "[Next log section with different data format]";
  public static void main(String[] args) {
    Scanner scanner = new Scanner(threatData);
    String pattern = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@" +
      "(\\d{2}/\\d{2}/\\d{4})";
    while(scanner.hasNext(pattern)) {
      scanner.next(pattern);
      MatchResult match = scanner.match();
      String ip = match.group(1);
      String date = match.group(2);
      System.out.format(
        "Threat on %s from %s%n", date,ip);
    }
  }
}
```

```
/* 输出:
Threat on 08/10/2015 from 58.27.82.161
Threat on 08/11/2015 from 204.45.234.40
Threat on 08/11/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
*/
```

next() 与特定模式一起使用时，该模式会和下一个输入分词进行匹配。结果由 match() 方法提供，正如上面的示例所示，它的工作方式与之前看到的正则表达式匹配相似。

使用正则表达式扫描时，有一点要注意：该模式仅与下一个输入的分词进行匹配。因此，如果你的模式里包含了分隔符，那就永远不会匹配成功。

18.9 StringTokenizer

在正则表达式（Java 1.4 引入）或 Scanner 类（Java 5 引入）之前，对字符串进行拆分的方式是使用 StringTokenizer 对其分词。但是现在有了正则表达式和 Scanner 类，对于同样的功能，它们实现起来更容易，也更简洁。下面是 StringTokenizer 与其他两种技术的简单比较：

```
// strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
  public static void main(String[] args) {
    String input =
      "But I'm not dead yet! I feel happy!";
    StringTokenizer stoke = new StringTokenizer(input);
    while(stoke.hasMoreElements())
```

```
      System.out.print(stoke.nextToken() + " ");
    System.out.println();
    System.out.println(
      Arrays.toString(input.split(" ")));
    Scanner scanner = new Scanner(input);
    while(scanner.hasNext())
      System.out.print(scanner.next() + " ");
  }
}
```

```
/* 输出：
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*/
```

使用正则表达式或 Scanner 对象，你还可以使用更复杂的模式来拆分字符串，而这对 StringTokenizer 来说就很困难了。我们应该可以放心地说，StringTokenizer 已经过时了。

18.10 总结

过去，Java 对字符串操作的支持还很初级，但在该语言的后续版本中，我们可以看到，Java 从其他语言中吸取了许多经验，对字符串提供了更复杂的操作。现在 Java 对字符串的支持已经相当完善，不过有时还是要在一些细节上注意效率的问题，比如合理地使用 StringBuilder。