《Effective Java》第三版中文版

目录

第一章 介绍

本书旨在帮助您有效地使用 Java 编程。

语言及其基本库：Java.Lang.java. UTL 和 ANDJavaIO，以及子包，如 asjav.UTI.CONCURTANTANCE 和 JavaUTILL 函数。其他图书馆也会不时讨论。这本书由九十个项目组成，每一个项目都传达一条规则。规则管理实践通常被最优秀和经验最丰富的程序员认为是有益的。这些项目被松散地分为 11 章，每章涵盖软件设计的一个广泛方面。这本书并不打算从头到尾地阅读：每一项或多或少都是独立的。这些项目被大量引用，因此你可以很容易地在书中绘制出你自己的路线。

自从这本书的最后一版出版以来，许多新的特点被添加到了这个平台上。这本书的大部分内容都在某种程度上使用了这些特性。此表显示了关键功能的主要覆盖范围：

Feature 特征	Items 条目	Release 发行版
Lambdas	Items42–44	Java 8
Streams	Items45–48	Java 8
Optionals	Item55	Java 8
Defaultmethodsininterfaces	Item21	Java 8
try-with-resources	Item9	Java 7
@SafeVarargs	Item32	Java 7
Modules	Item15	Java 9

大多数项目都用程序示例进行说明。这本书的一个主要特点是它包含了说明许多设计模式和习惯用法的代码示例。在适当的情况下，它们与本领域的标准参考工作交叉引用[Gamma95]。

许多项目包含一个或多个程序示例，说明一些需要避免的实践。这样的例子，有时被称为反模式，用注释清晰地标记，例如//neverdothis！.在每种情况下，该项都解释了示例不好的原因，并建议使用另一种方法。

这本书不是针对初学者的：它假设你已经适应了 Java。如果你没有，请考虑许

多很好的介绍性文章中的一篇，比如 Sestoft'sjava（Sestoft16）。虽然有效的 Javais 设计为任何一个有语言工作知识的人都能接触到，但它应该为思想提供食物，即使对于高级程序员也是如此。

这本书中的大多数规则都源于一些基本原则。清晰和简单是最重要的。组件的用户不应该对其行为感到惊讶。组件应该尽可能小，但不能小。（正如本书中所使用的，术语 componentrefer 指任何可重用的软件元素，从单个方法到由多个包组成的复杂框架。）代码应该被重用，而不是被复制。部件之间的偏差应保持在最低限度。错误应该在生成之后尽快检测出来，最好是在编译时。

虽然本书中的规则并不完全适用，但它们在大多数情况下都描述了最佳编程实践。你不应该草率地遵守这些规则，而应该偶尔有理由地违反它们。与大多数其他学科一样，学习编程的艺术包括首先学习规则，然后学习何时打破规则。

在很大程度上，这本书不是关于表演的。它是关于编写清晰、正确、可用、健壮、灵活和可维护的程序。如果你能做到这一点，获得你需要的性能通常是一件相对简单的事情。

（第 67 项）。一些项目确实讨论了性能问题，其中一些项目提供了性能数字。这些数字，用“在我的机器上”这个词来介绍，充其量应该被视为近似值。

值得一提的是，我的机器是一台老化的国产 3.5GHz 四核 Intel Core i7-4770K，16 GB DDR3-1866 CL9 RAM，运行 Azul'szulu 9.0.0.15 版本的 OpenJDK，位于 Microsoft Windows 7 Professional SP1 之上。

（64 位）。

在讨论 Java 编程语言及其库的特性时，有时需要引用特定的版本。为了方便起见，这本书优先使用昵称，而不是正式发行的名字。此表显示发布名称和昵称之间的映射 ping：

Official Release Name	Nickname
JDK 1.0.*x*	Java 1.0
JDK 1.1.*x*	Java 1.1
Java 2 Platform, Standard Edition, v1.2	Java 2
Java 2 Platform, Standard Edition, v1.3	Java 3
Java 2 Platform, Standard Edition, v1.4	Java 4
Java 2 Platform, Standard Edition, v5.0	Java 5
Java Platform, Standard Edition 6	Java 6
Java Platform, Standard Edition 7	Java 7
Java Platform, Standard Edition 8	Java 8
Java Platform, Standard Edition 9	Java 9

这些示例相当完整，但有利于完整性的可读性。他们可以自由使用 packagesjava.utilandjava.io 中的类。为了编写 tocompile 示例，您可能必须添加一个或多个导入声明，或其他类似的样板。该书的网站 http：//joshbloch.com/effectivejava 包含了 eachexample 的扩展版本，您可以编译和运行该版本。

在大多数情况下，本书使用技术术语，因为它们在 Java 语言规范 Java SE 8 版[JLS]中定义。一些术语值得特别提及。该语言支持四种类型：接口（包括

注释），类（包括基因），数组和原语。前三个是已知的参考类型。类实例和数组是对象;原始价值观不是。一个班级的成员包括其领域，方法，成员班级和

成员接口。一个方法的签名由其名称和其形式参数的类型组成;签名不包括方法的返回类型。

本书使用的几个术语与 Java 语言规范不同。与 Java 语言规范不同，本书使用继承作为同义词的子类。本书不是对接口使用术语继承，而是简单地说明了一个类实现接口或一个接口扩展

另一个。为了描述在没有指定时应用的访问级别，本书使用了传统的包 - 私有，而不是技术上正确的包访问[JLS，6.6.1]。

本书使用了 Java 语言规范中未定义的一些技术术语。 termexported API 或简称为 API，指的是程序对类，接口或包进行处理的类，接口，构造函数，成员和序列化形式。 （术语 API，其是短的可编程编程接口，优先于其他优

选术语使用

interface 以避免与该名称的语言结构混淆。）编写使用 API 的程序的编程人员称为 API 的 auser。其实现使用 API 的类是 API 的一部分。

类，接口，构造函数，成员和序列化表单统称为 API 元素。导出的 API 由可在定义 API 的包之外访问的 API 元素组成。这些是任何客户可以使用的 API 元素，并且 API 的作者承诺支持。不可否认，它们也是 Javadoc 实用程序在其默认操作模式下生成文档的元素。简而言之，包的 exportAPI 包含包中每个公共类或接口的 public 和 protected 成员和构造函数。

在 Java 9 中，将 amodule 系统添加到平台中。如果库使用模块系统，则其导出的 API 是库的模块声明导出的所有包的导出 API 的并集。

第二章 创建和销毁对象

本章关注创建和销毁对象：何时及如何创建它们，何时及如何避免创建它们，如何保证它们在适当的时候被销毁，以及如何管理那些必须在销毁前进行的销毁动作。

1 考虑用静态工厂方法替换构造器

允许客户端获得实例的传统方法是由类提供一个公共构造函数。还有一种技术应该成为每个程序员工具包的一部分。一个类可以提供一个公共静态工厂方法，它只是一个返回类实例的静态方法。下面是一个来自 Boolean （boolean 的包装类）的简单示例。该方法将布尔基本类型转换为布尔对象的引用：

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

要注意的是静态工厂方法与来自设计模式的工厂方法模式不同[Gamma95]。本项目中描述的静态工厂方法在设计模式中没有直接等价的方法。

除了公共构造函数，一个类还可以通过静态工厂方法提供它的客户端。提供静态工厂方法而不是公共构造函数的方式既有优点也有缺点。

静态工厂方法与构造函数相比的第一个优点，静态工厂方法有确切名称。 如果构造函数的参数本身并不能描述返回的对象，那么具有确切名称的静态工厂则更容易使用，生成的客户端代码也更容易阅读。例如，返回可能为素数的 BigInteger 类的构造函数 BigInteger(int, int, Random) 最好表示为名为 BigInteger.probableprime 的静态工厂方法。（这个方法是在 Java 4 中添加的）

一个类只能有一个具有给定签名的构造函数。众所周知，程序员可以通过提供两个构造函数来绕过这个限制，这两个构造函数的参数列表仅在参数类型的顺序上有所不同。这真是个坏主意。面对这样一个 API，用户将永远无法记住该用哪个构造函数，并且最终会错误地调用错误的构造函数。如果不参考类文档，阅读使用这些构造函数代码的人就不会知道代码的作用。

因为静态工厂方法有名称，所以它们不受前一段中讨论的限制。如果一个类似乎需要具有相同签名的多个构造函数，那么用静态工厂方法替换构造函数，并仔细选择名称以突出它们的区别。

静态工厂方法与构造函数相比的第二个优点，静态工厂方法不需要在每次调用时创建新对象。 这允许不可变类（Item-17）使用预先构造的实例，或在构造实例时缓存实例，并重复分配它们以避免创建不必要的重复对象。Boolean.valueOf(boolean) 方法说明了这种技术：它从不创建对象。这种技术类似于享元模式[Gamma95]。如果经常请求相同的对象，特别是在创建对象的代价很高时，它可以极大地提高性能。

静态工厂方法在重复调用中能够返回相同对象，这样的能力允许类严格控制任何时候存在的实例。这样做的类被称为实例受控的类。编写实例受控的类有几个原因。实例控制允许一个类来保证它是一个单例（Item-3）或不可实例化的（Item-4）。同时,它允许一个不可变的值类（Item-17）保证不存在两个相同的实例：a.equals(b) 当且仅当 a==b。这是享元模式的基础[Gamma95]。枚举类型（Item-34）提供了这种保证。

译注：原文 noninstantiable 应修改为 non-instantiable ，译为「不可实例化的」

静态工厂方法与构造函数相比的第三个优点，可以通过静态工厂方法获取返回类型的任何子类的对象。 这为选择返回对象的类提供了很大的灵活性。

这种灵活性的一个应用是 API 可以在不公开其类的情况下返回对象。以这种方式隐藏实现类会导致一个非常紧凑的 API。这种技术适用于基于接口的框架（Item-20），其中接口为静态工厂方法提供了自然的返回类型。

在 Java 8 之前，接口不能有静态方法。按照惯例，一个名为 Type 的接口的静态工厂方法被放在一个名为 Types 的不可实例化的伴随类（Item-4）中。例如，Java 的 Collections 框架有 45 个接口实用工具实现，提供了不可修改的集合、同步集合等。几乎所有这些实现都是通过一个非实例化类（java.util.Collections）中的静态工厂方法导出的。返回对象的类都是非公共的。

译注：原文 noninstantiable 应修改为 non-instantiable ，译为「不可实例化的」

Collections 框架 API 比它导出 45 个独立的公共类要小得多，每个公共类对应一个方便的实现。减少的不仅仅是 API 的数量，还有概念上的减少：程序员为了使用 API 必须掌握的概念的数量和难度。程序员知道返回的对象由相关的接口精确地指定的，因此不需要为实现类阅读额外的类文档。此外，使用这种静态工厂方法需要客户端通过接口而不是实现类引用返回的对象，这通常是很好的实际用法（Item-64）。

自 Java 8 起，消除了接口不能包含静态方法的限制，因此通常没有理由为接口提供不可实例化的伴随类。许多本来会在这种级别的公共静态成员应该被放在接口本身中。但是，请注意，仍然有必要将这些静态方法背后的大部分实现代码放到单独的包私有类中。这是因为 Java 8 要求接口的所有静态成员都是公共的。Java 9 允许私有静态方法，但是静态字段和静态成员类仍然需要是公共的。

静态工厂的第四个优点是，返回对象的类可以随调用的不同而变化，作为输入参数的函数。 声明的返回类型的任何子类型都是允许的。返回对象的类也可以因版本而异。

EnumSet 类（Item-36）没有公共构造函数，只有静态工厂。在 OpenJDK 实现中，它们返回两个子类中的一个实例，这取决于底层 enum 类型的大小：如果它有 64 个或更少的元素，就像大多数 enum 类型一样，静态工厂返回一个 long 类型的 RegularEnumSet 实例；如果 enum 类型有 65 个或更多的元素，工厂将返回一个由 long[] 类型的 JumboEnumSet 实例。

客户端看不到这两个实现类的存在。如果 RegularEnumSet 不再为小型 enum 类型提供性能优势，它可能会在未来的版本中被消除，而不会产生不良影响。类似地，如果事实证明 EnumSet 有益于性能，未来的版本可以添加第三或第四个 EnumSet 实现。客户端既不知道也不关心从工厂返回的对象的类；它们只关心它是 EnumSet 的某个子类。

静态工厂的第五个优点是，当编写包含方法的类时，返回对象的类不需要存在。 这种灵活的静态工厂方法构成了服务提供者框架的基础，比如 Java 数据库连接 API（JDBC）。服务提供者框架是一个系统，其中提供者实现一个服务，系统使客户端可以使用这些实现，从而将客户端与实现分离。

服务提供者框架中有三个基本组件：代表实现的服务接口；提供者注册 API，提供者使用它来注册实现，以及服务访问 API，客户端使用它来获取服务的实例。服务访问 API 允许客户端指定选择实现的标准。在没有这些条件的情况下，API 返回一个默认实现的实例，或者允许客户端循环使用所有可用的实现。服务访问 API 是灵活的静态工厂，它构成了服务提供者框架的基础。

服务提供者框架的第四个可选组件是服务提供者接口，它描述了产生服务接口实例的工厂对象。在没有服务提供者接口的情况下，必须以反射的方式实例化实现（Item-65）。在 JDBC 中，连接扮演服务接口 DriverManager 的角色。DriverManager.registerDriver 是提供商注册的 API，DriverManager.getConnection 是服务访问 API，驱动程序是服务提供者接口。

服务提供者框架模式有许多变体。例如，服务访问 API 可以向客户端返回比提供者提供的更丰富的服务接口。这是桥接模式[Gamma95]。依赖注入框架（Item-5）可以看作是强大的服务提供者。由于是 Java 6，该平台包括一个通用服务提供者框架 Java.util.ServiceLoader，所以你不需要，通常也不应该写你自己的（Item-59）。JDBC 不使用 ServiceLoader，因为前者比后者要早。

仅提供静态工厂方法的主要局限是，没有公共或受保护构造函数的类不能被子类化。 例如，不可能在集合框架中子类化任何方便的实现类。这可能是一种因祸得福的做法，因为它鼓励程序员使用组合而不是继承（Item-18），并且对于不可变的类型（Item-17）是必需的。

静态工厂方法的第二个缺点是程序员很难找到它们。 它们在 API 文档中不像构造函数那样引人注目，因此很难弄清楚如何实例化一个只提供静态工厂方法而没有构造函数的类。Javadoc 工具总有一天会关注到静态工厂方法。与此同时，你可以通过在类或接口文档中对静态工厂方法多加留意以及遵守通用命名约定的方式来减少这个困扰。下面是一些静态工厂方法的常用名称。这个列表还远不够详尽：

from，一种型转换方法，该方法接受单个参数并返回该类型的相应实例，例如：

```
Date d = Date.from(instant);
```

of，一个聚合方法，它接受多个参数并返回一个包含这些参数的此类实例，例如：

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

valueOf，一种替代 from 和 of 但更冗长的方法，例如：

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

instance 或 getInstance，返回一个实例，该实例由其参数（如果有的话）描述，但不具有相同的值，例如：

```
StackWalker luke = StackWalker.getInstance(options);
```

create 或 newInstance，与 instance 或 getInstance 类似，只是该方法保证每个调用都返回一个新实例，例如：

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

getType，类似于 getInstance，但如果工厂方法位于不同的类中，则使用此方法。类型是工厂方法返回的对象类型，例如：

```
FileStore fs = Files.getFileStore(path);
```

newType，与 newInstance 类似，但是如果工厂方法在不同的类中使用。类型是工厂方法返回的对象类型，例如：

```
BufferedReader br = Files.newBufferedReader(path);
```

type，一个用来替代 getType 和 newType 的比较简单的方式，例如：

```
List<Complaint> litany = Collections.list(legacyLitany);
```

总之，静态工厂方法和公共构造器都有各自的用途，理解它们的相对优点是值得的。通常静态工厂更可取，因此避免在没有考虑静态工厂的情况下提供公共构造函数。

2 当遇到多个构造器参数时考虑用构建器(建造者)替换

静态工厂和构造函数都有一个局限：它们不能对大量可选参数做很好的扩展。以一个类为例，它表示包装食品上的营养标签。这些标签上有一些字段是必需的，如：净含量、毛重和每单位份量的卡路里，另有超过 20 个可选的字段，如：总脂肪、饱和脂肪、反式脂肪、胆固醇、钠等等。大多数产品只有这些可选字段中的少数，且具有非零值。

应该为这样的类编写什么种类的构造函数或静态工厂呢？传统的方式是使用可伸缩构造函数，在这种模式中，只向构造函数提供必需的参数。即，向第一个构造函数提供单个可选参数，向第二个构造函数提供两个可选参数，以此类推，最后一个构造函数是具有所有可选参数的。这是它在实际应用中的样子。为了简洁起见，只展示具备四个可选字段的情况：

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize; // (mL) required
    private final int servings; // (per container) required
    private final int calories; // (per serving) optional
    private final int fat; // (g/serving) optional
```

```
        private final int sodium; // (mg/serving) optional
        private final int carbohydrate; // (g/serving) optional
        public NutritionFacts(int servingSize, int servings) {
            this(servingSize, servings, 0);
        }
        public NutritionFacts(int servingSize, int servings, int calories) {
            this(servingSize, servings, calories, 0);
        }
        public NutritionFacts(int servingSize, int servings, int calories, int fat) {
            this(servingSize, servings, calories, fat, 0);
        }
        public NutritionFacts(int servingSize, int servings, int calories, int fat, int
sodium) {
            this(servingSize, servings, calories, fat, sodium, 0);
        }

        public NutritionFacts(int servingSize, int servings, int calories, int fat, int
sodium, int carbohydrate) {
            this.servingSize = servingSize;
            this.servings = servings;
            this.calories = calories;
            this.fat = fat;
            this.sodium = sodium;
            this.carbohydrate = carbohydrate;
        }
    }
```

当你想要创建一个实例时，可以使用包含所需的参数的最短参数列表的构造函数：

```
NutritionFacts cocaCola =new NutritionFacts(240, 8, 100, 0, 35, 27);
```

通常，这个构造函数包含许多额外的参数，但是你必须为它们传递一个值。在本例中，我们为 fat 传递了一个值 0。只有六个参数时，这可能看起来不那么糟，但随着参数的增加，它很快就会失控。

简单地说，**可伸缩构造函数模式可以工作，但是当有很多参数时，编写客户端代码是很困难的，而且读起来更困难。**读者想知道所有这些值是什么意思，必须仔细清点参数。相同类型参数的长序列会导致细微的错误。如果客户端不小心倒转了两个这样的参数，编译器不会报错，但是程序会在运行时出错（Item-51）。

当你在构造函数中遇到许多可选参数时，另一种选择是 JavaBean 模式，在这种模式中，你调用一个无参数的构造函数来创建对象，然后调用 setter 方法来设置每个所需的参数和每个感兴趣的可选参数：

```
// JavaBeans Pattern - allows inconsistency, mandates mutability
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize = -1; // Required; no default value
    private int servings = -1; // Required; no default value
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }
    // Setters
    public void setServingSize(int val) { servingSize = val; }
    public void setServings(int val) { servings = val; }
    public void setCalories(int val) { calories = val; }
```

```
        public void setFat(int val) { fat = val; }
        public void setSodium(int val) { sodium = val; }
        public void setCarbohydrate(int val) { carbohydrate = val; }
    }
```

这个模式没有可伸缩构造函数模式的缺点。创建实例很容易，虽然有点冗长，但很容易阅读生成的代码：

```
    NutritionFacts cocaCola = new NutritionFacts();
    cocaCola.setServingSize(240);
    cocaCola.setServings(8);
    cocaCola.setCalories(100);
    cocaCola.setSodium(35);
    cocaCola.setCarbohydrate(27);
```

不幸的是，JavaBean 模式本身有严重的缺点。因为构建是在多个调用之间进行的，所以 JavaBean 可能在构建的过程中处于不一致的状态。该类不能仅通过检查构造函数参数的有效性来强制一致性。在不一致的状态下尝试使用对象可能会导致错误的发生，而包含这些错误的代码很难调试。一个相关的缺点是，JavaBean 模式排除了使类不可变的可能性（Item-17），并且需要程序员额外的努力来确保线程安全。

通过在对象构建完成时手动「冻结」对象，并在冻结之前不允许使用对象，可以减少这些缺陷，但是这种变通方式很笨拙，在实践中很少使用。此外，它可能在运行时导致错误，因为编译器不能确保程序员在使用对象之前调用它的 freeze 方法。

幸运的是，还有第三种选择，它结合了可伸缩构造函数模式的安全性和 JavaBean 模式的可读性。它是建造者模式的一种形式[Gamma95]。客户端不直接生成所需的对象，而是使用所有必需的参数调用构造函数（或静态工厂），并获得一个 builder 对象。然后，客户端在构建器对象上调用像 setter 这样的方法来设置每个感兴趣的可选参数。最后，客户端调用一个无参数的构建方法来生成对象，这通常是不可变的。构建器通常是它构建的类的静态成员类（Item-24）。下面是它在实际应用中的样子：

```
    // Builder Pattern
```

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;
        private int carbohydrate = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }
        public Builder calories(int val) {
            calories = val;
            return this;
        }
```

```
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public Builder sodium(int val) {
            sodium = val;
            return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
```

```
    }
```

NutritionFacts 类是不可变的，所有参数默认值都在一个位置。构建器的 setter 方法返回构建器本身，这样就可以链接调用，从而得到一个流畅的 API。下面是客户端代码的样子：

```
    NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)

    .calories(100).sodium(35).carbohydrate(27).build();
```

该客户端代码易于编写，更重要的是易于阅读。建造者模式模拟 Python 和 Scala 中的可选参数。

为了简洁，省略了有效性检查。为了尽快检测无效的参数，请检查构建器的构造函数和方法中的参数有效性。检查构建方法调用的构造函数中涉及多个参数的不变量。为了确保这些不变量不受攻击，在从构建器复制参数之后检查对象字段（Item-50）。如果检查失败，抛出一个 IllegalArgumentException（Item-72），它的详细消息指示哪些参数无效（Item-75）。

建造者模式非常适合于类层次结构。使用构建器的并行层次结构，每个构建器都嵌套在相应的类中。抽象类有抽象类构建器；具体类有具体类构建器。例如，考虑一个在层次结构处于最低端的抽象类，它代表各种比萨饼：

```
    import java.util.EnumSet;

    import java.util.Objects;

    import java.util.Set;

    // Builder pattern for class hierarchies

    public abstract class Pizza {

        public enum Topping {HAM, MUSHROOM, ONION, PEPPER,
SAUSAGE}

        final Set<Topping> toppings;

        abstract static class Builder<T extends Builder<T>> {

            EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);

            public T addTopping(Topping topping) {

                toppings.add(Objects.requireNonNull(topping));
```

```
            return self();
        }
        abstract Pizza build();
        // Subclasses must override this method to return "this"
        protected abstract T self();
    }
    Pizza(Builder<?> builder) {
        toppings = builder.toppings.clone(); // See Item 50
    }
}
```

请注意，Pizza.Builder 是具有递归类型参数的泛型类型（Item-31）。这与抽象 self 方法一起，允许方法链接在子类中正常工作，而不需要强制转换。对于 Java 缺少自类型这一事实，这种变通方法称为模拟自类型习惯用法。这里有两个具体的比萨子类，一个是标准的纽约风格的比萨，另一个是 calzone。前者有一个所需的大小参数，而后者让你指定酱料应该是内部还是外部：

```
import java.util.Objects;
public class NyPizza extends Pizza {
    public enum Size {SMALL, MEDIUM, LARGE}
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> {
        private final Size size;
        public Builder(Size size) {
            this.size = Objects.requireNonNull(size);
        }
        @Override
```

```
            public NyPizza build() {
                return new NyPizza(this);
            }
            @Override
            protected Builder self() {
                return this;
            }
        }
        private NyPizza(Builder builder) {
            super(builder);
            size = builder.size;
        }
}
public class Calzone extends Pizza {
        private final boolean sauceInside;
        public static class Builder extends Pizza.Builder<Builder> {
            private boolean sauceInside = false; // Default
            public Builder sauceInside() {
                sauceInside = true;
                return this;
            }
            @Override
            public Calzone build() {
                return new Calzone(this);
            }
```

```
        @Override
        protected Builder self() {
            return this;
        }
    }
    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}
```

注意，每个子类的构建器中的构建方法声明为返回正确的子类：构建的方法 NyPizza.Builder 返回 NyPizza，而在 Calzone.Builder 则返回 Calzone。这种技术称为协变返回类型，其中一个子类方法声明为返回超类中声明的返回类型的子类型。它允许客户使用这些构建器，而不需要强制转换。这些「层次构建器」的客户端代码与简单的 NutritionFacts 构建器的代码基本相同。为简洁起见，下面显示的示例客户端代码假定枚举常量上的静态导入：

```
NyPizza pizza = new NyPizza.Builder(SMALL)
.addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
.addTopping(HAM).sauceInside().build();
```

与构造函数相比，构造函数的一个小优点是构造函数可以有多个变量参数，因为每个参数都是在自己的方法中指定的。或者，构建器可以将传递给一个方法的多个调用的参数聚合到单个字段中，如前面的 addTopping 方法中所示。

建造者模式非常灵活。一个构建器可以多次用于构建多个对象。构建器的参数可以在构建方法的调用之间进行调整，以改变创建的对象。构建器可以在创建对象时自动填充某些字段，例如在每次创建对象时增加的序列号。

建造者模式也有缺点。为了创建一个对象，你必须首先创建它的构建器。虽然在实际应用中创建这个构建器的成本可能并不显著，但在以性能为关键的场景下，这可能会是一个问题。而且，建造者模式比可伸缩构造函数模式更冗

长，因此只有在有足够多的参数时才值得使用，比如有 4 个或更多参数时，才应该使用它。但是请记住，你可能希望在将来添加更多的参数。但是，如果你以构造函数或静态工厂开始，直至类扩展到参数数量无法控制的程度时，也会切换到构建器，但是过时的构造函数或静态工厂将很难处理。因此，最好一开始就从构建器开始。

总之，在设计构造函数或静态工厂的类时，建造者模式是一个很好的选择，特别是当许多参数是可选的或具有相同类型时。与可伸缩构造函数相比，使用构建器客户端代码更容易读写，而且构建器比 JavaBean 更安全。

3 用私有构造器或者枚举类型强化单例属性

单例是一个只实例化一次的类 [Gamma95]。单例通常表示无状态对象，比如函数（Item-24）或系统组件，它们在本质上是唯一的。**将一个类设计为单例会使它的客户端测试时变得困难，** 除非它实现了作为其类型的接口，否则无法用模拟实现来代替单例。

实现单例有两种常见的方法。两者都基于保持构造函数私有和导出公共静态成员以提供对唯一实例的访问。在第一种方法中，成员是一个 final 字段：

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public void leaveTheBuilding() { ... }
}
```

私有构造函数只调用一次，用于初始化 public static final 修饰的 Elvis 类型字段 INSTANCE。不使用 public 或 protected 的构造函数保证了「独一无二」的空间：一旦初始化了 Elvis 类，就只会存在一个 Elvis 实例，不多也不少。客户端所做的任何事情都不能改变这一点，但有一点需要注意：拥有特殊权限的客户端可以借助 AccessibleObject.setAccessible 方法利用反射调用私有构造函数（Item-65）如果需要防范这种攻击，请修改构造函数，使其在请求创建第二个实例时抛出异常。

译注：使用 AccessibleObject.setAccessible 方法调用私有构造函数示例：

```
Constructor<?>[] constructors = Elvis.class.getDeclaredConstructors();
```

```
AccessibleObject.setAccessible(constructors, true);


Arrays.stream(constructors).forEach(name -> {
    if (name.toString().contains("Elvis")) {
        Elvis instance = (Elvis) name.newInstance();
        instance.leaveTheBuilding();
    }
});
```

在实现单例的第二种方法中，公共成员是一种静态工厂方法：

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }
    public void leaveTheBuilding() { ... }
}
```

所有对 getInstance() 方法的调用都返回相同的对象引用，并且不会创建其他 Elvis 实例（与前面提到的警告相同）。

译注：这里的警告指拥有特殊权限的客户端可以借助 AccessibleObject.setAccessible 方法利用反射调用私有构造函数

公共字段方法的主要优点是 API 明确了类是单例的：public static 修饰的字段是 final 的，因此它总是包含相同的对象引用。第二个优点是更简单。

译注：static factory approach 等同于 static factory method

静态工厂方法的一个优点是，它可以在不更改 API 的情况下决定类是否是单例。工厂方法返回唯一的实例，但是可以对其进行修改，为调用它的每个线程返回一个单独的实例。第二个优点是，如果应用程序需要的话，可以编写泛型的单例工厂（Item-30）。使用静态工厂的最后一个优点是方法引用能够作为

一个提供者，例如 Elvis::getInstance 是 Supplier<Elvis> 的提供者。除非能够与这些优点沾边，否则使用 public 字段的方式更可取。

译注 1：原文方法引用可能是笔误，修改为 Elvis::getInstance

译注 2：方法引用作为提供者的例子：

```
Supplier<Elvis> sup = Elvis::getInstance;
Elvis obj = sup.get();
obj.leaveTheBuilding();
```

要使单例类使用这两种方法中的任何一种（Chapter 12），仅仅在其声明中添加实现 serializable 是不够的。要维护单例保证，应声明所有实例字段为 transient，并提供 readResolve 方法（Item-89）。否则，每次反序列化实例时，都会创建一个新实例，在我们的示例中，这会导致出现虚假的 Elvis。为了防止这种情况发生，将这个 readResolve 方法添加到 Elvis 类中：

```
// readResolve method to preserve singleton property
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

实现单例的第三种方法是声明一个单元素枚举：

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;
    public void leaveTheBuilding() { ... }
}
```

这种方法类似于 public 字段方法，但是它更简洁，默认提供了序列化机制，提供了对多个实例化的严格保证，即使面对复杂的序列化或反射攻击也是如此。这种方法可能有点不自然，但是**单元素枚举类型通常是实现单例的最佳方法。**注

意，如果你的单例必须扩展一个超类而不是 Enum（尽管你可以声明一个 Enum 来实现接口），你就不能使用这种方法。

4 通过私有构造器强化不可实例化的能力

有时你会想要写一个类，它只是一个静态方法和静态字段的组合。这样的类已经获得了坏名声，因为有些人滥用它们来避免从对象角度思考，但是它们确有用途。它们可以用 java.lang.Math 或 java.util.Arrays 的方式，用于与原始值或数组相关的方法。它们还可以用于对以 java.util.Collections 的方式实现某些接口的对象分组静态方法，包括工厂（Item-1）。（对于 Java 8，你也可以将这些方法放入接口中，假设你可以进行修改。）最后，这些类可用于对 final 类上的方法进行分组，因为你不能将它们放在子类中。

这样的实用程序类不是为实例化而设计的：实例是无意义的。然而，在没有显式构造函数的情况下，编译器提供了一个公共的、无参数的默认构造函数。对于用户来说，这个构造函数与其他构造函数没有区别。在已发布的 API 中看到无意中实例化的类是很常见的。

译注：原文 noninstantiable 应修改为 non-instantiable ，译为「不可实例化的」

试图通过使类抽象来实施不可实例化是行不通的。 可以对类进行子类化，并实例化子类。此外，它误导用户认为类是为继承而设计的（Item-19）。然而，有一个简单的习惯用法来确保不可实例化。只有当类不包含显式构造函数时，才会生成默认构造函数，因此**可以通过包含私有构造函数使类不可实例化：**

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    } ... // Remainder omitted
}
```

因为显式构造函数是私有的，所以在类之外是不可访问的。AssertionError不是严格要求的，但是它提供了保障，以防构造函数意外地被调用。它保证类在任何情况下都不会被实例化。这个习惯用法有点违反常规，因为构造函数是明确提供的，但不能调用它。因此，如上述代码所示，包含注释是明智的做法。

这个习惯用法也防止了类被子类化，这是一个副作用。所有子类构造函数都必须调用超类构造函数，无论是显式的还是隐式的，但这种情况下子类都没有可访问的超类构造函数可调用。

5 依赖注入优先硬连接资源 @

许多类依赖于一个或多个底层资源。例如，拼写检查程序依赖于字典。常见做法是，将这种类实现为静态实用工具类（Item-4）：

```
// Inappropriate use of static utility - inflexible & untestable!
public class SpellChecker {
    private static final Lexicon dictionary = ...;
    private SpellChecker() {} // Noninstantiable
    public static boolean isValid(String word) { ... }
    public static List<String> suggestions(String typo) { ... }
}
```

Similarly, it’s not uncommon to see them implemented as singletons (Item 3):

类似地，我们也经常看到它们的单例实现（Item-3）：

```
// Inappropriate use of singleton - inflexible & untestable!
public class SpellChecker {
    private final Lexicon dictionary = ...;
    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
```

```
    }
```

这两种方法都不令人满意，因为它们假设只使用一个字典。在实际应用中，每种语言都有自己的字典，特殊的字典用于特殊的词汇表。另外，最好使用一个特殊的字典进行测试。认为一本字典就足够了，是一厢情愿的想法。

你可以尝试让 SpellChecker 支持多个字典：首先取消 dictionary 字段的 final 修饰，并在现有的拼写检查器中添加更改 dictionary 的方法。但是在并发环境中这种做法是笨拙的、容易出错的和不可行的。**静态实用工具类和单例不适用于由底层资源参数化的类。**

所需要的是支持类的多个实例的能力（在我们的示例中是 SpellChecker），每个实例都使用客户端需要的资源（在我们的示例中是 dictionary）。满足此要求的一个简单模式是在**创建新实例时将资源传递给构造函数。** 这是依赖注入的一种形式：字典是拼写检查器的依赖项，在创建它时被注入到拼写检查器中。

```
// Dependency injection provides flexibility and testability
public class SpellChecker {
    private final Lexicon dictionary;
    public SpellChecker(Lexicon dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

依赖注入模式非常简单，许多程序员在不知道其名称的情况下使用了多年。虽然拼写检查器示例只有一个资源（字典），但是依赖注入可以处理任意数量的资源和任意依赖路径。它保持了不可变性（Item-17），因此多个客户端可以共享依赖对象（假设客户端需要相同的底层资源）。依赖注入同样适用于构造函数、静态工厂（Item-1）和构建器（Item-2）。

这种模式的一个有用变体是将资源工厂传递给构造函数。工厂是一个对象，可以反复调用它来创建类型的实例。这样的工厂体现了工厂方法模式 [Gamma95]。Java 8 中引入的 Supplier<T> 非常适合表示工厂。在输入中接受 Supplier<T> 的方法通常应该使用有界通配符类型（Item-31）来约束工厂的类

型参数，以允许客户端传入创建指定类型的任何子类型的工厂。例如，这里有一个生产瓷砖方法，每块瓷砖都使用客户提供的工厂来制作马赛克：

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

尽管依赖注入极大地提高了灵活性和可测试性，但它可能会使大型项目变得混乱，这些项目通常包含数千个依赖项。通过使用依赖注入框架（如 Dagger、Guice 或 Spring），几乎可以消除这种混乱。这些框架的使用超出了本书的范围，但是请注意，为手动依赖注入而设计的 API 很容易被这些框架所使用。

总之，不要使用单例或静态实用工具类来实现依赖于一个或多个底层资源的类，这些资源的行为会影响类的行为，也不要让类直接创建这些资源。相反，将创建它们的资源或工厂传递给构造函数（或静态工厂或构建器）。这种操作称为依赖注入，它将大大增强类的灵活性、可重用性和可测试性。

6 避免创建不必要的对象

重用单个对象通常是合适的，不必每次需要时都创建一个新的功能等效对象。重用可以更快、更时尚。如果对象是不可变的，那么它总是可以被重用的（Item-17）。

作为一个不该做的极端例子，请考虑下面的语句：

```
String s = new String("bikini"); // DON'T DO THIS!
```

该语句每次执行时都会创建一个新的 String 实例，而这些对象创建都不是必需的。String 构造函数的参数（"bikini"）本身就是一个 String 实例，在功能上与构造函数创建的所有对象相同。如果这种用法发生在循环或频繁调用的方法中，则不必要创建数百万个 String 实例。

改进后的版本如下：

```
String s = "bikini";
```

这个版本使用单个 String 实例，而不是每次执行时都创建一个新的实例。此外，可以保证在同一虚拟机中运行的其他代码都可以重用该对象，只要恰好包含相同的字符串字面量 [JLS, 3.10.5]。

你通常可以通过使用静态工厂方法（Item-1）来避免创建不必要的对象，而不是在提供这两种方法的不可变类上使用构造函数。例如，工厂方法

Boolean.valueOf(String) 比构造函数 Boolean(String) 更可取，**后者在 Java 9 中被弃用了**。构造函数每次调用时都必须创建一个新对象，而工厂方法从来不需要这样做，在实际应用中也不会这样做。除了重用不可变对象之外，如果知道可变对象不会被修改，也可以重用它们。

有些对象的创建的代价相比而言要昂贵得多。如果你需要重复地使用这样一个「昂贵的对象」，那么最好将其缓存以供重用。不幸的是，当你创建这样一个对象时，并不总是很明显。假设你要编写一个方法来确定字符串是否为有效的罗马数字。下面是使用正则表达式最简单的方法：

```
// Performance can be greatly improved!

static boolean isRomanNumeral(String s) {

    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})" +
"(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

}
```

这个实现的问题是它依赖于 String.matches 方法。**虽然 String.matches 是检查字符串是否与正则表达式匹配的最简单方法，但它不适合在性能关键的情况下重复使用。** 问题是，它在内部为正则表达式创建了一个模式实例，并且只使用一次，之后就可以进行垃圾收集了。创建一个模式实例是很昂贵的因为它需要将正则表达式编译成有限的状态机制。

为了提高性能，将正则表达式显式编译为模式实例（它是不可变的），作为类初始化的一部分，缓存它，并在每次调用 isRomanNumeral 方法时重用同一个实例：

```
// Reusing expensive object for improved performance

public class RomanNumerals {

    private static final Pattern ROMAN =
Pattern.compile("^(?=.)M*(C[MD]|D?C{0,3})" +
"(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s) {

        return ROMAN.matcher(s).matches();

    }

}
```

如果频繁调用 isRomanNumeral，改进版本将提供显著的性能提升。在我的机器上，原始版本输入 8 字符的字符串花费 1.1μs，而改进的版本需要 0.17μs，快 6.5 倍。不仅性能得到了改善，清晰度也得到了提高。为不可见的模式实例创建一个静态终态字段允许我们为它命名，这比正则表达式本身更容易阅读。

如果加载包含改进版 isRomanNumeral 方法的类时，该方法从未被调用过，那么初始化字段 ROMAN 是不必要的。因此，可以用延迟初始化字段（Item-83）的方式在第一次调用 isRomanNumeral 方法时才初始化字段，而不是在类加载时初始化，但不建议这样做。通常情况下，延迟初始化会使实现复杂化，而没有明显的性能改善（Item-67）。

译注：类加载通常指的是类的生命周期中加载、连接、初始化三个阶段。当方法没有在类加载过程中被使用时，可以不初始化与之相关的字段

当一个对象是不可变的，很明显，它可以安全地重用，但在其他情况下，它远不那么明显，甚至违反直觉。考虑适配器的情况 [Gamma95]，也称为视图。适配器是委托给支持对象的对象，提供了一个替代接口。因为适配器的状态不超过其支持对象的状态，所以不需要为给定对象创建一个给定适配器的多个实例。

例如，Map 接口的 keySet 方法返回 Map 对象的 Set 视图，其中包含 Map 中的所有键。天真的是，对 keySet 的每次调用都必须创建一个新的 Set 实例，但是对给定 Map 对象上的 keySet 的每次调用都可能返回相同的 Set 实例。虽然返回的 Set 实例通常是可变的，但所有返回的对象在功能上都是相同的：当返回的对象之一发生更改时，所有其他对象也会发生更改，因为它们都由相同的 Map 实例支持。虽然创建 keySet 视图对象的多个实例基本上是无害的，但这是不必要的，也没有好处。

另一种创建不必要对象的方法是自动装箱，它允许程序员混合原始类型和包装类型，根据需要自动装箱和拆箱。**自动装箱模糊了原始类型和包装类型之间的区别，** 两者有细微的语义差别和不明显的性能差别（Item-61）。考虑下面的方法，它计算所有正整数的和。为了做到这一点，程序必须使用 long，因为 int 值不够大，不足以容纳所有正整数值的和：

```
// Hideously slow! Can you spot the object creation?
private static long sum() {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
```

```
            sum += i;

        return sum;

    }
```

这个程序得到了正确的答案，但是由于一个字符的印刷错误，它的速度比实际要慢得多。变量 sum 被声明为 Long 而不是 long，这意味着程序将构造大约 231 个不必要的 Long 实例（大约每次将 Long i 添加到 Long sum 时都有一个实例）。将 sum 的声明从 Long 更改为 long，机器上的运行时间将从 6.3 秒减少到 0.59 秒。教训很清楚：**基本数据类型优于包装类，还应提防意外的自动装箱。**

这个项目不应该被曲解为是在暗示创建对象是昂贵的，应该避免。相反，创建和回收这些小对象的构造函数成本是很低廉的，尤其是在现代 JVM 实现上。创建额外的对象来增强程序的清晰性、简单性或功能通常是件好事。

相反，通过维护自己的对象池来避免创建对象不是一个好主意，除非池中的对象非常重量级。证明对象池是合理的对象的典型例子是数据库连接。建立连接的成本非常高，因此重用这些对象是有意义的。然而，一般来说，维护自己的对象池会使代码混乱，增加内存占用，并损害性能。现代 JVM 实现具有高度优化的垃圾收集器，在轻量级对象上很容易胜过这样的对象池。

与此项对应的条目是 Item-50（防御性复制）。当前项的描述是：「在应该重用现有对象时不要创建新对象」，而 Item 50 的描述则是：「在应该创建新对象时不要重用现有对象」。请注意，当需要进行防御性复制时，重用对象所受到的惩罚远远大于不必要地创建重复对象所受到的惩罚。在需要时不制作防御性副本可能导致潜在的 bug 和安全漏洞；不必要地创建对象只会影响样式和性能。

7 消除过期的对象引用

如果你从需要手动管理内存的语言（如 C 或 c++）切换到具有垃圾回收机制的语言（如 Java），当你使用完对象后，会感觉程序员工作轻松很多。当你第一次体验它的时候，它几乎就像魔术一样。这很容易让人觉得你不需要考虑内存管理，但这并不完全正确。

考虑以下简单的堆栈实现：

```
import java.util.Arrays;

import java.util.EmptyStackException;
```

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
```

```
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这个程序没有明显的错误（但是通用版本请参阅 Item-29）。你可以对它进行详尽的测试，它会以优异的成绩通过所有的测试，但是有一个潜在的问题。简单地说，该程序有一个「内存泄漏」问题，由于垃圾收集器活动的增加或内存占用的增加，它可以悄无声息地表现为性能的降低。在极端情况下，这种内存泄漏可能导致磁盘分页，甚至出现 OutOfMemoryError 程序故障，但这种故障相对少见。

那么内存泄漏在哪里呢？如果堆栈增长，然后收缩，那么从堆栈中弹出的对象将不会被垃圾收集，即使使用堆栈的程序不再引用它们。这是因为栈保留了这些对象的旧引用。一个过时的引用，是指永远不会被取消的引用。在本例中，元素数组的「活动部分」之外的任何引用都已过时。活动部分由索引小于大小的元素组成。

垃圾收集语言中的内存泄漏（更确切地说是无意的对象保留）是暗藏的风险。如果无意中保留了对象引用，那么对象不仅被排除在垃圾收集之外，该对象引用的任何对象也被排除在外，依此类推。即使只是无意中保留了一些对象引用，许多许多的对象也可能被阻止被垃圾收集，从而对性能产生潜在的巨大影响。

解决这类问题的方法很简单：一旦引用过时，就将置空。在我们的 Stack 类中，对某个项的引用一旦从堆栈中弹出就会过时。pop 方法的正确版本如下：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
```

```
    }
```

用 null 处理过时引用的另一个好处是，如果它们随后被错误地关联引用，程序将立即失败，出现 NullPointerException，而不是悄悄地做错误的事情。尽可能快地检测编程错误总是有益的。

当程序员第一次被这个问题困扰时，他们可能会过度担心，一旦程序使用完它，他们就会取消所有对象引用。这既无必要也不可取；它不必要地搞乱了程序。清除对象引用应该是例外，而不是规范。消除过时引用的最佳方法是让包含引用的变量脱离作用域。如果你在最狭窄的范围内定义每个变量（Item-57），那么这种情况自然会发生。

那么，什么时候应该取消引用呢？Stack 类的哪些方面容易导致内存泄漏？简单地说，它管理自己的内存。存储池包含元素数组的元素（对象引用单元，而不是对象本身）数组的活动部分（如前面所定义的）中的元素被分配，而数组其余部分中的元素是空闲的。垃圾收集器没有办法知道这一点；对于垃圾收集器，元素数组中的所有对象引用都同样有效。只有程序员知道数组的非活动部分不重要。只要数组元素成为非活动部分的一部分，程序员就可以通过手动清空数组元素，有效地将这个事实传递给垃圾收集器。

一般来说，当类管理自己的内存时，程序员应该警惕内存泄漏。每当释放一个元素时，元素中包含的任何对象引用都应该被取消。

另一个常见的内存泄漏源是缓存。一旦将对象引用放入缓存中，就很容易忘记它就在那里，并且在它变得无关紧要之后很久仍将它留在缓存中。有几个解决这个问题的办法。如果你非常幸运地实现了一个缓存，只要缓存外有对其键的引用，那么就将缓存表示为 WeakHashMap；当条目过时后，条目将被自动删除。记住，WeakHashMap 只有在缓存条目的预期生存期由键的外部引用（而不是值）决定时才有用。

更常见的情况是，缓存条目的有效生存期定义不太好，随着时间的推移，条目的价值会越来越低。在这种情况下，缓存偶尔应该清理那些已经停用的条目。这可以通过后台线程（可能是 ScheduledThreadPoolExecutor）或向缓存添加新条目时顺便完成。LinkedHashMap 类通过其 removeEldestEntry 方法简化了后一种方法。对于更复杂的缓存，你可能需要直接使用 java.lang.ref。

内存泄漏的第三个常见来源是侦听器和其他回调。 如果你实现了一个 API，其中客户端注册回调，但不显式取消它们，除非你采取一些行动，否则它们将累积。确保回调被及时地垃圾收集的一种方法是仅存储对它们的弱引用，例如，将它们作为键存储在 WeakHashMap 中。

由于内存泄漏通常不会表现为明显的故障，它们可能会在系统中存在多年。它们通常只能通过仔细的代码检查或借助一种称为堆分析器的调试工具来发现。因此，学会在这样的问题发生之前预测并防止它们发生是非常可取的。

8 避免使用终结方法和清理器

终结器是不可预测的，通常是危险的，也是不必要的。 它们的使用可能导致不稳定的行为、糟糕的性能和可移植性问题。终结器有一些有效的用途，我们将在后面的文章中介绍，但是作为规则，你应该避免使用它们。在 Java 9 中，终结器已经被弃用，但是 Java 库仍然在使用它们。Java 9 替代终结器的是清除器。**清除器的危险比终结器小，但仍然不可预测、缓慢，而且通常是不必要的。**

c++ 程序员被告诫不要把终结器或清除器当成 Java 的 c++ 析构函数。在 c++ 中，析构函数是回收与对象相关联的资源的常用方法，对象是构造函数的必要对等物。在 Java 中，当对象变得不可访问时，垃圾收集器将回收与之关联的存储，无需程序员进行任何特殊工作。c++ 析构函数还用于回收其他非内存资源。在 Java 中，使用带有资源的 try-with-resources 或 try-finally 块用于此目的（Item-9）。

终结器和清除器的一个缺点是不能保证它们会被立即执行[JLS, 12.6]。当对象变得不可访问，终结器或清除器对它进行操作的时间是不确定的。这意味着永远不应该在终结器或清除器中执行任何对时间要求很严格的操作。例如，依赖终结器或清除器关闭文件就是一个严重错误，因为打开的文件描述符是有限的资源。如果由于系统在运行终结器或清除器的延迟导致许多文件处于打开状态，程序可能会运行失败，因为它不能再打开其他文件。

终结器和清除器执行的快速性主要是垃圾收集算法的功能，在不同的实现中存在很大差异。依赖于终结器的及时性或更清晰的执行的程序的行为可能也会发生变化。这样的程序完全有可能在测试它的 JVM 上完美地运行，然后在最重要的客户喜欢的 JVM 上悲惨地失败。

姗姗来迟的定稿不仅仅是一个理论上的问题。为类提供终结器可以任意延迟其实例的回收。一位同事调试了一个长期运行的 GUI 应用程序，该应用程序神秘地终结于 OutOfMemoryError 错误。分析显示，在应用程序终结的时候，终结器队列上有数千个图形对象等待最终完成和回收。不幸的是，终结器线程运行的优先级低于另一个应用程序线程，因此对象不能以适合终结器的速度完成。语言规范没有保证哪个线程将执行终结器，因此除了避免使用终结器之外，没有其他可移植的方法来防止这类问题。在这方面，清洁器比终结器要好一些，

因为类作者可以自己控制是否清理线程，但是清洁器仍然在后台运行，在垃圾收集器的控制下运行，所以不能保证及时清理。

该规范不仅不能保证终结器或清洁剂能及时运行；它并不能保证它们能运行。完全有可能，甚至很有可能，程序在某些不再可访问的对象上运行而终止。因此，永远不应该依赖终结器或清除器来更新持久状态。例如，依赖终结器或清除器来释放共享资源（如数据库）上的持久锁，是让整个分布式系统停止工作的好方法。

不要被 System.gc 和 System.runFinalization 的方法所诱惑。它们可能会增加终结器或清除器被运行的几率，但它们不能保证一定运行。曾经有两种方法声称可以保证这一点：System.runFinalizersOnExit 和它的孪生兄弟 Runtime.runFinalizersOnExit。这些方法存在致命的缺陷，并且已经被废弃了几十年[ThreadStop]。

终结器的另一个问题是，在终结期间抛出的未捕获异常被忽略，该对象的终结终止 [JLS, 12.6]。未捕获的异常可能会使其他对象处于损坏状态。如果另一个线程试图使用这样一个损坏的对象，可能会导致任意的不确定性行为。正常情况下，未捕获的异常将终止线程并打印堆栈跟踪，但如果在终结器中出现，则不会打印警告。清除器没有这个问题，因为使用清除器的库可以控制它的线程。

使用终结器和清除器会严重影响性能。在我的机器上，创建一个简单的 AutoCloseable 对象，使用 try-with-resources 关闭它以及让垃圾收集器回收它的时间大约是 12ns。相反，使用终结器将时间增加到 550ns。换句话说，使用终结器创建和销毁对象大约要慢 50 倍。这主要是因为终结器抑制了有效的垃圾收集。如果使用清除器清除的所有实例（在我的机器上每个实例大约 500ns），那么清除器的速度与终结器相当，但是如果只将它们作为安全网来使用，清除器的速度要快得多，如下所述。在这种情况下，在我的机器上创建、清理和销毁一个对象需要花费 66ns 的时间，这意味着如果你不使用它，你需要多出五倍（而不是五十倍）的保障成本。

终结器有一个严重的安全问题：它们会让你的类受到终结器攻击。终结器攻击背后的思想很简单：如果从构造函数或它的序列化等价物（readObject 和 readResolve 方法（Item-12））抛出一个异常，恶意子类的终结器就可以运行在部分构造的对象上，而这个对象本来应该「胎死腹中」。这个终结器可以在静态字段中记录对对象的引用，防止它被垃圾收集。一旦记录了畸形对象，就很容易在这个对象上调用本来就不应该存在的任意方法。从构造函数抛出异常应该足以防止对象的出现；在有终结器的情况下，就不是这样了。这样的攻击可能会造成可怕的后果。最终类对终结器攻击免疫，因为没有人能够编写最终类

的恶意子类。为了保护非最终类不受终结器攻击，编写一个不执行任何操作的最终终结方法。

那么，如果一个类的对象封装了需要终止的资源，例如文件或线程，那么应该做什么，而不是为它编写终结器或清除器呢？只有你的类实现 AutoCloseable，要求其客户端每个实例在不再需要时调用关闭方法，通常使用 try-with-resources 确保终止，即使面对异常（Item-9）。一个值得一提的细节是实例必须跟踪是否已经关闭：close 方法必须在字段中记录对象不再有效，其他方法必须检查这个字段，如果在对象关闭后调用它们，则必须抛出一个 IllegalStateException。

那么，清除器和终结器有什么用呢？它们可能有两种合法用途。一种是充当一个安全网，以防资源的所有者忽略调用它的 close 方法。虽然不能保证清除器或终结器将立即运行（或根本不运行），但如果客户端没有这样做，最好是延迟释放资源。如果你正在考虑编写这样一个安全网络终结器，那就好好考虑一下这种保护是否值得。一些 Java 库类，如 FileInputStream、FileOutputStream、ThreadPoolExecutor 和 java.sql.Connection，都有终结器作为安全网。

清洁器的第二个合法使用涉及到与本机对等体的对象。本机对等点是普通对象通过本机方法委托给的本机（非 java）对象。因为本机对等点不是一个正常的对象，垃圾收集器不知道它，并且不能在回收 Java 对等点时回收它。如果性能是可接受的，并且本机对等体不持有任何关键资源，那么更清洁或终结器可能是完成这项任务的合适工具。如果性能不可接受，或者本机对等体持有必须立即回收的资源，则类应该具有前面描述的关闭方法。

清除器的使用有些棘手。下面是一个简单的 Room 类，展示了这个设施。让我们假设房间在回收之前必须被清理。Room 类实现了 AutoCloseable；它的自动清洗安全网使用了清除器，这只是一个实现细节。与终结器不同，清除器不会污染类的公共 API：

```
import sun.misc.Cleaner;

// An autocloseable class using a cleaner as a safety net

public class Room implements AutoCloseable {

    private static final Cleaner cleaner = Cleaner.create();


    // Resource that requires cleaning. Must not refer to Room!
```

```
        private static class State implements Runnable {
            int numJunkPiles; // Number of junk piles in this room

            State(int numJunkPiles) {
                this.numJunkPiles = numJunkPiles;
            }

            // Invoked by close method or cleaner
            @Override
            public void run() {
                System.out.println("Cleaning room");
                numJunkPiles = 0;
            }
        }

        // The state of this room, shared with our cleanable
        private final State state;

        // Our cleanable. Cleans the room when it’s eligible for gc
        private final Cleaner.Cleanable cleanable;

        public Room(int numJunkPiles) {
            state = new State(numJunkPiles);
            cleanable = cleaner.register(this, state);
        }

        @Override
        public void close() {
            cleanable.clean();
        }
    }
```

静态嵌套 State 类持有清洁器清洁房间所需的资源。在这种情况下，它仅仅是 numJunkPiles 字段，表示房间的混乱程度。更实际地说，它可能是最后一个包含指向本机对等点的 long 指针。State 实现了 Runnable，它的运行方法最多被调用一次，由我们在 Room 构造器中向 cleaner 实例注册状态实例时得到的 Cleanable 调用。对 run 方法的调用将由以下两种方法之一触发：通常是通过调用 Room 的 close 方法来触发，调用 Cleanable 的 clean 方法。如果当一个 Room 实例有资格进行垃圾收集时，客户端没有调用 close 方法，那么清除器将调用 State 的 run 方法（希望如此）。

状态实例不引用其 Room 实例是非常重要的。如果它这样做了，它将创建一个循环，以防止 Room 实例有资格进行垃圾收集（以及自动清理）。因此，状态必须是一个静态嵌套类，因为非静态嵌套类包含对其封闭实例的引用（Item-24）。同样不建议使用 lambda，因为它们可以很容易地捕获对包围对象的引用。

就像我们之前说的，Room 类的清除器只是用作安全网。如果客户端将所有 Room 实例包围在带有资源的 try 块中，则永远不需要自动清理。这位表现良好的客户端展示了这种做法：

```
public class Adult {
    public static void main(String[] args) {
        try (Room myRoom = new Room(7)) {
            System.out.println("Goodbye");
        }
    }
}
```

如你所料，运行 Adult 程序打印「Goodbye」，然后是打扫房间。但这个从不打扫房间的不守规矩的程序怎么办？

```
public class Teenager {
    public static void main(String[] args) {
        new Room(99);
        System.out.println("Peace out");
    }
```

```
    }
```

你可能期望它打印出「Peace out」，然后打扫房间，但在我的机器上，它从不打扫房间；它只是退出。这就是我们之前提到的不可预测性。Cleaner 规范说：「在 System.exit 中，清洁器的行为是特定于实现的。不保证清理操作是否被调用。」虽然规范没有说明，但对于普通程序退出来说也是一样。在我的机器上，将 System.gc() 添加到 Teenager 的主要方法中就足以让它在退出之前打扫房间，但不能保证在其他机器上看到相同的行为。总之，不要使用清洁器，或者在 Java 9 之前的版本中使用终结器，除非是作为安全网或终止非关键的本机资源。即便如此，也要小心不确定性和性能后果。

9 try-with-resources 优先 try-finally @

Java 库包含许多必须通过调用 close 方法手动关闭的资源。常见的有 InputStream、OutputStream 和 java.sql.Connection。关闭资源常常会被客户端忽略，这会导致可怕的性能后果。虽然这些资源中的许多都使用终结器作为安全网，但终结器并不能很好地工作（Item-8）。

从历史上看，try-finally 语句是确保正确关闭资源的最佳方法，即使在出现异常或返回时也是如此：

```
// try-finally - No longer the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

这可能看起来不坏，但添加第二个资源时，情况会变得更糟：

```
// try-finally is ugly when used with more than one resource!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
```

```
        try {
            OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
            }
        }
        finally {
            in.close();
        }
    }
```

这可能难以置信。在大多数情况下，即使是优秀的程序员也会犯这种错误。首先，我在 Java Puzzlers [Bloch05]的 88 页上做错了，多年来没有人注意到。事实上，2007 年发布的 Java 库中三分之二的 close 方法使用都是错误的。

译注：《Java Puzzlers》的中文译本为《Java 解惑》

使用 try-finally 语句关闭资源的正确代码（如前两个代码示例所示）也有一个细微的缺陷。try 块和 finally 块中的代码都能够抛出异常。例如，在 firstLineOfFile 方法中，由于底层物理设备发生故障，对 readLine 的调用可能会抛出异常，而关闭的调用也可能出于同样的原因而失败。在这种情况下，第二个异常将完全覆盖第一个异常。异常堆栈跟踪中没有第一个异常的记录，这可能会使实际系统中的调试变得非常复杂（而这可能是希望出现的第一个异常，以便诊断问题）。虽然可以通过编写代码来抑制第二个异常而支持第一个异常，但实际上没有人这样做，因为它太过冗长。

当 Java 7 引入 try-with-resources 语句 [JLS, 14.20.3]时，所有这些问题都一次性解决了。要使用这个结构，资源必须实现 AutoCloseable 接口，它由一个单独的 void-return close 方法组成。Java 库和第三方库中的许多类和接口现

在都实现或扩展了 AutoCloseable。如果你编写的类存在必须关闭的资源，那么也应该实现 AutoCloseable。

下面是使用 try-with-resources 的第一个示例：

```
// try-with-resources - the the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

下面是使用 try-with-resources 的第二个示例：

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream in = new FileInputStream(src);OutputStream out = new
FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

和使用 try-finally 的原版代码相比，try-with-resources 为开发者提供了更好的诊断方式。考虑 firstLineOfFile 方法。如果异常是由 readLine 调用和不可见的 close 抛出的，则后一个异常将被抑制，以支持前一个异常。实际上，还可能会抑制多个异常，以保留实际希望看到的异常。这些被抑制的异常不会仅仅被抛弃；它们会被打印在堆栈跟踪中，并标记它们被抑制。可以通过编程方式使用 getSuppressed 方法访问到它们，该方法是在 Java 7 中添加到 Throwable 中的。

可以在带有资源的 try-with-resources 语句中放置 catch 子句，就像在常规的 try-finally 语句上一样。这允许处理异常时不必用另一层嵌套来影响代码。作为一个特指的示例，下面是我们的 firstLineOfFile 方法的一个版本，它不抛

出异常，但如果无法打开文件或从中读取文件，则返回一个默认值：

```
// try-with-resources with a catch clause
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

教训很清楚：在使用必须关闭的资源时，总是优先使用 try-with-resources，而不是 try-finally。前者的代码更短、更清晰，生成的异常更有用。使用 try-with-resources 语句可以很容易地为必须关闭的资源编写正确的代码，而使用 try-finally 几乎是不可能的。

第三章 对所有对象都通用的方法

虽然 Object 是一个具体的类，但它主要是为扩展而设计的。它的所有非 final 方法（equals、hashCode、toString、clone 和 finalize）都有显式的通用约定，因为它们的设计目的是被覆盖。任何类都有责任覆盖这些方法并将之作为一般约定；如果不这样做，将阻止依赖于约定的其他类（如 HashMap 和 HashSet）与之一起正常工作。

本章将告诉你何时以及如何覆盖 Object 类的非 final 方法。finalize 方法在本章中被省略，因为它在 Item-8 中讨论过。虽然 Comparable.compareTo 不是 Object 类的方法，但是由于具有相似的特性，所以本章也对它进行讨论。

10 覆写 equals 时候遵守通用规定

覆盖 equals 方法似乎很简单，但是有很多覆盖的方式会导致出错，而且后果可能非常严重。避免问题的最简单方法是不覆盖 equals 方法，在这种情况下，类的每个实例都只等于它自己。如果符合下列任何条件，就是正确的做法：

类的每个实例本质上都是唯一的。 对于像 Thread 这样表示活动实体类而不是值类来说也是如此。Object 提供的 equals 实现对于这些类具有完全正确的行为。

该类不需要提供「逻辑相等」测试。 例如，java.util.regex.Pattern 可以覆盖 equals 来检查两个 Pattern 实例是否表示完全相同的正则表达式，但设计人员认为客户端不需要或不需要这个功能。在这种情况下，从 Object 继承的 equals 实现是理想的。

超类已经覆盖了 equals，超类行为适合于这个类。 例如，大多数 Set 的实现从 AbstractSet 继承其对等实现，List 从 AbstractList 继承实现，Map 从 AbstractMap 继承实现。

类是私有的或包私有的，并且你确信它的 equals 方法永远不会被调用。 如果你非常厌恶风险，你可以覆盖 equals 方法，以确保它不会意外调用：

```
@Override
public boolean equals(Object o) {
    throw new AssertionError(); // Method is never called
}
```

什么时候覆盖 equals 方法是合适的？当一个类有一个逻辑相等的概念，而这个概念不同于仅判断对象的同一性（相同对象的引用），并且超类还没有覆盖 equals。对于值类通常是这样。值类只是表示值的类，例如 Integer 或 String。使用 equals 方法比较引用和值对象的程序员希望发现它们在逻辑上是否等价，而不是它们是否引用相同的对象。覆盖 equals 方法不仅是为了满足程序员的期望，它还使实例能够作为 Map 的键或 Set 元素时，具有可预测的、理想的行为。

译注 1 ：有一个表示状态的内部类。没有覆盖 equals 方法时，equals 的结果与 s1==s2 相同，为 false，即两者并不是相同对象的引用。

```
public static void main(String[] args) {
    class Status {
        public String status;
    }
    Status s1 = new Status();
```

```
        Status s2 = new Status();



        System.out.println(s1==s2); // false

        System.out.println(s1.equals(s2)); // false
    }
```

译注 2 ：覆盖 equals 方法后，以业务逻辑来判断是否相同，具备相同 status 字段即为相同。在使用去重功能时，也以此作为判断依据。

```
    public static void main(String[] args) {
        class Status {
            public String status;
            @Override
            public boolean equals(Object o) {
                return Objects.equals(status, ((Status) o).status);
            }
        }
        Status s1 = new Status();
        Status s2 = new Status();
        System.out.println(s1==s2); // false
        System.out.println(s1.equals(s2)); // true
    }
```

不需要覆盖 equals 方法的一种值类是使用实例控件（Item-1）来确保每个值最多只存在一个对象的类。枚举类型（Item-34）属于这一类。对于这些类，逻辑相等与对象标识相同，因此对象的 equals 方法函数与逻辑 equals 方法相同。

- 当你覆盖 equals 方法时，你必须遵守它的通用约定。以下是具体内容，来自 Object 规范：

- equals 方法实现了等价关系。它应有这些属性：

- 反身性：对于任何非空的参考值 x，x.equals(x) 必须返回 true。

- 对称性：对于任何非空参考值 x 和 y，x.equals(y) 必须在且仅当 y.equals(x) 返回 true 时返回 true。

- 传递性：对于任何非空的引用值 x, y, z，如果 x.equals(y) 返回 true，y.equals(z) 返回 true，那么 x.equals(z) 必须返回 true。

- 一致性：对于任何非空的引用值 x 和 y, x.equals(y) 的多次调用必须一致地返回 true 或一致地返回 false，前提是不修改 equals 中使用的信息。

- 对于任何非空引用值 x，x.equals(null) 必须返回 false。

除非你有数学方面的倾向，否则这些起来有点可怕，但不要忽略它！如果你违反了它，你的程序很可能会出现行为异常或崩溃，并且很难确定失败的根源。用 John Donne 的话来说，没有一个类是孤立的。一个类的实例经常被传递给另一个类。许多类（包括所有集合类）依赖于传递给它们的对象遵守 equals 约定。

既然你已经意识到了违反 equals 约定的危险，让我们详细讨论一下。好消息是，尽管表面上看起来很复杂，但其实并不复杂。一旦你明白了，就不难坚持下去了。

什么是等价关系？简单地说，它是一个操作符，它将一组元素划分为子集，子集的元素被认为是彼此相等的。这些子集被称为等价类。为了使 equals 方法有用，从用户的角度来看，每个等价类中的所有元素都必须是可互换的。现在让我们依次检查以下五个需求：

反身性，第一个要求仅仅是说一个对象必须等于它自己。很难想象会无意中违反了这条规则。如果你违反了它，然后将类的一个实例添加到集合中，contains 方法很可能会说该集合不包含你刚才添加的实例。

对称性，第二个要求是任何两个对象必须在是否相等的问题上达成一致。与第一个要求不同，无意中违反了这个要求的情况不难想象。例如，考虑下面的类，它实现了不区分大小写的字符串。字符串的情况是保留的 toString，但忽略在 equals 的比较：

```
// Broken - violates symmetry!

public final class CaseInsensitiveString {
```

```
        private final String s;
        public CaseInsensitiveString(String s) {
            this.s = Objects.requireNonNull(s);
    }
    // Broken - violates symmetry!
    @Override
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
        return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
        if (o instanceof String) // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
        } ... // Remainder omitted
    }
```

这个类中的 equals 方法天真地尝试与普通字符串进行互操作。假设我们有一个不区分大小写的字符串和一个普通字符串：

```
    CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
    String s = "polish";
```

正如预期的那样，cis.equals(s) 返回 true。问题是，虽然 CaseInsensitiveString 中的 equals 方法知道普通字符串，但是 String 中的 equals 方法对不区分大小写的字符串不知情。因此，s.equals(cis) 返回 false，这明显违反了对称性。假设你将不区分大小写的字符串放入集合中：

```
    List<CaseInsensitiveString> list = new ArrayList<>();
    list.add(cis);
```

此时 list.contains(s) 返回什么？谁知道呢？在当前的 OpenJDK 实现中，它碰巧返回 false，但这只是一个实现案例。在另一个实现中，它可以很容易地返

回 true 或抛出运行时异常。一旦你违反了 equals 约定，就不知道当其他对象面对你的对象时，会如何表现。

译注：contains 方法在 ArrayList 中的实现源码如下（省略了源码中的多行注释）：

```
// ArrayList 的大小
private int size;
// 保存 ArrayList 元素的容器，一个 Object 数组
transient Object[] elementData; // non-private to simplify nested class access
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

public int indexOf(Object o) {
    return indexOfRange(o, 0, size);
}
int indexOfRange(Object o, int start, int end) {
    Object[] es = elementData;
    if (o == null) {
        for (int i = start; i < end; i++) {
            if (es[i] == null) {
                return i;
            }
        }
    } else {
        for (int i = start; i < end; i++) {
            if (o.equals(es[i])) {
```

```
                    return i;
                }
            }
        }
        return -1;
    }
```

为了消除这个问题，只需从 equals 方法中删除与 String 互操作的错误尝试。一旦你这样做了，你可以重构方法为一个单一的返回语句：

```
    @Override
    public boolean equals(Object o) {
        return o instanceof CaseInsensitiveString && ((CaseInsensitiveString)
o).s.equalsIgnoreCase(s);
    }
```

传递性，equals 约定的第三个要求是，如果一个对象等于第二个对象，而第二个对象等于第三个对象，那么第一个对象必须等于第三个对象。同样，无意中违反了这个要求的情况不难想象。考虑向超类添加新的值组件时，子类的情况。换句话说，子类添加了一条影响 equals 比较的信息。让我们从一个简单的不可变二维整数点类开始：

```
    public class Point {
        private final int x;
        private final int y;
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
        @Override
        public boolean equals(Object o) {
```

```
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ... // Remainder omitted
}
```

假设你想继承这个类，对一个点添加颜色的概念：

```
public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }
    ... // Remainder omitted
}
```

equals 方法应该是什么样子？如果你完全忽略它，则实现将从 Point 类继承而来，在 equals 比较中颜色信息将被忽略。虽然这并不违反 equals 约定，但显然是不可接受的。假设你写了一个 equals 方法，该方法只有当它的参数是另一个颜色点，且位置和颜色相同时才返回 true：

```
// Broken - violates symmetry!
@Override
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
```

```
        return super.equals(o) && ((ColorPoint) o).color == color;
    }
```

这种方法的问题是，当你比较一个点和一个颜色点时，你可能会得到不同的结果，反之亦然。前者比较忽略颜色，而后者比较总是返回 false，因为参数的类型是不正确的。为了使问题更具体，让我们创建一个点和一个颜色点：

```
    Point p = new Point(1, 2);
    ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

然后，p.equals(cp) 返回 true，而 cp.equals(p) 返回 false。当你做「混合比较」的时候，你可以通过让 ColorPoint.equals 忽略颜色来解决这个问题：

```
    // Broken - violates transitivity!
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        // If o is a normal Point, do a color-blind comparison
        if (!(o instanceof ColorPoint))
            return o.equals(this);
        // o is a ColorPoint; do a full comparison
        return super.equals(o) && ((ColorPoint) o).color == color;
    }
```

这种方法确实提供了对称性，但牺牲了传递性：

```
    ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
    Point p2 = new Point(1, 2);
    ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

现在，p1.equals(p2) 和 p2.equals(p3) 返回 true，而 p1.equals(p3) 返回 false，这明显违反了传递性。前两个比较是「色盲」，而第三个比较考虑了颜色。

同样，这种方法会导致无限的递归：假设有两个点的子类，比如 ColorPoint 和 SmellPoint，每个都使用这种 equals 方法。然后调用 myColorPoint.equals(mySmellPoint) 会抛出 StackOverflowError。

那么解决方案是什么？这是面向对象语言中等价关系的一个基本问题。**除非你愿意放弃面向对象的抽象优点，否则无法继承一个可实例化的类并添加一个值组件，同时保留 equals 约定。**

你可能会听到它说你可以继承一个实例化的类并添加一个值组件，同时通过在 equals 方法中使用 getClass 测试来代替 instanceof 测试来保持 equals 约定：

```
// Broken - violates Liskov substitution principle (page 43)
@Override
public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;

    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

只有当对象具有相同的实现类时，才会产生相等的效果。这可能看起来不是很糟糕，但其后果是不可接受的：Point 的子类的实例仍然是一个 Point，并且它仍然需要作为一个函数来工作，但是如果采用这种方法，它就不会这样做！假设我们要写一个方法来判断一个点是否在单位圆上。我们可以这样做：

```
// Initialize unitCircle to contain all Points on the unit circle
private static final Set<Point> unitCircle = Set.of(
        new Point( 1, 0), new Point( 0, 1),
        new Point(-1, 0), new Point( 0, -1)
    );
    public static boolean onUnitCircle(Point p) {
```

```
        return unitCircle.contains(p);
    }
```

虽然这可能不是实现功能的最快方法，但它工作得很好。假设你以一种不添加值组件的简单方式继承 Point，例如，让它的构造函数跟踪创建了多少实例：

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter = new AtomicInteger();
    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated() {
        return counter.get();
    }
}
```

Liskov 替换原则指出，类型的任何重要属性都应该适用于所有子类型，因此为类型编写的任何方法都应该在其子类型上同样有效 [Liskov87]。这是我们先前做的正式声明，即点的子类（如 CounterPoint）仍然是一个 Point，并且必须作为一个 Point。但假设我们传递了一个 CounterPoint 给 onUnitCircle 方法。如果 Point 类使用基于 getclass 的 equals 方法，那么不管 CounterPoint 实例的 x 和 y 坐标如何，onUnitCircle 方法都会返回 false。这是因为大多数集合，包括 onUnitCircle 方法使用的 HashSet，都使用 equals 方法来测试包含性，没有一个 CounterPoint 实例等于任何一个点。但是，如果你在 Point 上使用了正确的基于实例的 equals 方法，那么在提供对位实例时，相同的 onUnitCircle 方法就可以很好地工作。

译注：里氏替换原则（Liskov Substitution Principle，LSP）面向对象设计的基本原则之一。里氏替换原则指出：任何父类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正被复用，而衍生类也能够在父类的基础上增加新的行为。

虽然没有令人满意的方法来继承一个可实例化的类并添加一个值组件，但是有一个很好的解决方案：遵循 Item-18 的建议，「Favor composition over inheritance.」。给 ColorPoint 一个私有的 Point 字段和一个 public 视图方法（Item-6），而不是让 ColorPoint 继承 Point，该方法返回与这个颜色点相同位置的点：

```
// Adds a value component without violating the equals contract
public class ColorPoint {
    private final Point point;
    private final Color color;
    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }
    /**
    * Returns the point-view of this color point.
    */
    public Point asPoint() {
        return point;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }
```

```
        ... // Remainder omitted
    }
```

Java 库中有一些类确实继承了一个可实例化的类并添加了一个值组件。例如，java.sql.Timestamp 继承 java.util.Date 并添加了纳秒字段。如果在同一个集合中使用时间戳和日期对象，或者以其他方式混合使用时间戳和日期对象，那么时间戳的 equals 实现确实违反了对称性，并且可能导致不稳定的行为。Timestamp 类有一个免责声明，警告程序员不要混合使用日期和时间戳。虽然只要将它们分开，就不会遇到麻烦，但是没有什么可以阻止你将它们混合在一起，因此产生的错误可能很难调试。时间戳类的这种行为是错误的，不应该效仿

注意，你可以向抽象类的子类添加一个值组件，而不违反 equals 约定。这对于遵循 Item-23 中的建议而得到的类层次结构很重要，「Prefer class hierarchies to tagged classes.」。例如，可以有一个没有值组件的抽象类形状、一个添加半径字段的子类圆和一个添加长度和宽度字段的子类矩形。只要不可能直接创建超类实例，前面显示的那种问题就不会发生。

一致性，对等约定的第四个要求是，如果两个对象相等，它们必须一直保持相等，除非其中一个（或两个）被修改。换句话说，可变对象可以等于不同时间的不同对象，而不可变对象不能。在编写类时，仔细考虑它是否应该是不可变的（Item-17）。如果你认为应该这样做，那么请确保你的 equals 方法执行了这样的限制，即相等的对象始终是相等的，而不等的对象始终是不等的。

无论一个类是否不可变，都不要编写依赖于不可靠资源的 equals 方法。如果你违反了这个禁令，就很难满足一致性要求。例如，java.net.URL 的 equals 方法依赖于与 url 相关联的主机的 IP 地址的比较。将主机名转换为 IP 地址可能需要网络访问，而且不能保证随着时间的推移产生相同的结果。这可能会导致 URL 的 equals 方法违反约定，并在实践中造成问题。URL 的 equals 方法的行为是一个很大的错误，不应该被模仿。不幸的是，由于兼容性需求，它不能更改。为了避免这种问题，equals 方法应该只对 memoryresident 对象执行确定性计算。

非无效性，最后的要求没有一个正式的名称，所以我冒昧地称之为「非无效性」。它说所有对象都不等于 null。虽然很难想象在响应调用 o.equals(null) 时意外地返回 true，但不难想象意外地抛出 NullPointerException。一般约定中禁止这样做。许多类都有相等的方法，通过显式的 null 测试来防止它：

```
    @Override
```

```
public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

这个测试是不必要的。要测试其参数是否相等，equals 方法必须首先将其参数转换为适当的类型，以便能够调用其访问器或访问其字段。在执行转换之前，方法必须使用 instanceof 运算符来检查其参数的类型是否正确：

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
}
```

如果缺少这个类型检查，并且 equals 方法传递了一个错误类型的参数，equals 方法将抛出 ClassCastException，这违反了 equals 约定。但是，如果 instanceof 操作符的第一个操作数为空，则指定该操作符返回 false，而不管第二个操作数 [JLS, 15.20.2] 中出现的是什么类型。因此，如果传入 null，类型检查将返回 false，因此不需要显式的 null 检查。

综上所述，这里有一个高质量构建 equals 方法的秘诀：

1. **使用 == 运算符检查参数是否是对该对象的引用。** 如果是，返回 true。这只是一种性能优化，但如果比较的代价可能很高，那么这种优化是值得的。

2. **使用 instanceof 运算符检查参数是否具有正确的类型。** 如果不是，返回 false。通常，正确的类型是方法发生的类。有时候，它是由这个类实现的某个接口。如果类实现了一个接口，该接口对 equals 约定进行了改进，以允许跨实现该接口的类进行比较，则使用该接口。集合接口，如 Set、List、Map 和 Map.Entry 具有此属性。

3. **将参数转换为正确的类型。**因为在这个强制类型转换之前有一个实例测试，所以它肯定会成功。

4. **对于类中的每个「重要」字段，检查参数的字段是否与该对象的相应字段匹配。**如果所有这些测试都成功，返回 true；否则返回 false。如果第 2 步中的类型是接口，则必须通过接口方法访问参数的字段；如果是类，你可以根据字段的可访问性直接访问它们。

对于类型不是 float 或 double 的基本类型字段，使用 == 运算符进行比较；对于对象引用字段，递归调用 equals 方法；对于 float 字段，使用 static Float.compare(float,float) 方法；对于 double 字段，使用 Double.compare(double, double)。float 和 double 字段的特殊处理是由于 Float.NaN、-0.0f 和类似的双重值的存在而必须的；请参阅 JLS 15.21.1 或 Float.equals 文档。虽然你可以将 float 和 double 字段与静态方法 Float.equals 和 Double.equals 进行比较，这将需要在每个比较上进行自动装箱，这将有较差的性能。对于数组字段，将这些指导原则应用于每个元素。如果数组字段中的每个元素都很重要，那么使用 Arrays.equals 方法之一。

一些对象引用字段可能合法地包含 null。为了避免可能出现 NullPointerException，请使用静态方法 Objects.equals(Object, Object) 检查这些字段是否相等。

对于某些类，例如上面的 CaseInsensitiveString，字段比较比简单的 equal 测试更复杂。如果是这样，你可能希望存储字段的规范形式，以便 equals 方法可以对规范形式进行廉价的精确比较，而不是更昂贵的非标准比较。这种技术最适合于不可变类（Item-17）；如果对象可以更改，则必须使规范形式保持最新。

equals 方法的性能可能会受到字段比较顺序的影响。为了获得最佳性能，你应该首先比较那些更可能不同、比较成本更低的字段，或者理想情况下两者都比较。不能比较不属于对象逻辑状态的字段，例如用于同步操作的锁字段。你不需要比较派生字段（可以从「重要字段」计算），但是这样做可能会提高 equals 方法的性能。如果派生字段相当于整个对象的摘要描述，那么如果比较失败，比较该字段将节省比较实际数据的开销。例如，假设你有一个多边形类，你缓存这个区域。如果两个多边形的面积不相等，你不需要比较它们的边和顶点。

写完 equals 方法后，问自己三个问题：它具备对称性吗？具备传递性吗？具备一致性吗？不要只问自己，要编写单元测试来检查，除非使用 AutoValue（第 49 页）来生成 equals 方法，在这种情况下，你可以安全地省略测试。如

果属性不能保持，请找出原因，并相应地修改 equals 方法。当然，equals 方法还必须满足其他两个属性（反身性和非无效性），但这两个通常会自己处理。

在这个简单的 PhoneNumber 类中，根据前面的方法构造了一个 equals 方法：

```
// Class with a typical equals method
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;
    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix = rangeCheck(prefix, 999, "prefix");
        this.lineNum = rangeCheck(lineNum, 9999, "line num");
    }
    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }
    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;

        PhoneNumber pn = (PhoneNumber)o;
```

```
                return pn.lineNum == lineNum && pn.prefix == prefix &&
pn.areaCode == areaCode;

            } ... // Remainder omitted

    }
```

以下是一些最后的警告：

- **当你覆盖 equals 时，也覆盖 hashCode。**（Item-11）

- **不要自作聪明。**如果你只是为了判断相等性而测试字段，那么遵循 equals 约定并不困难。如果你在寻求对等方面过于激进，很容易陷入麻烦。一般来说，考虑到任何形式的混叠都不是一个好主意。例如，File 类不应该尝试将引用同一文件的符号链接等同起来。值得庆幸的是，它不是。

- **不要用另一种类型替换 equals 声明中的对象。**对于程序员来说，编写一个类似于这样的 equals 方法，然后花上几个小时思考为什么它不能正常工作是很常见的：

```
    // Broken - parameter type must be Object!

    public boolean equals(MyClass o) {

        ...

    }
```

这里的问题是，这个方法没有覆盖其参数类型为 Object 的 Object.equals，而是重载了它（Item-52）。即使是普通的方法，提供这样一个「强类型的」equals 方法是不可接受的，因为它会导致子类中的重写注释产生误报并提供错误的安全性。

如本条目所示，一致使用 Override 注释将防止你犯此错误（Item-40）。这个 equals 方法不会编译，错误消息会告诉你什么是错误的：

```
    // Still broken, but won't compile

    @Override

    public boolean equals(MyClass o) {

        ...

    }
```

编写和测试 equals （和 hashCode）方法很乏味，生成的代码也很单调。手动编写和测试这些方法的一个很好的替代方法是使用谷歌的开源 AutoValue 框架，它会自动为你生成这些方法，由类上的一个注释触发。在大多数情况下，AutoValue 生成的方法与你自己编写的方法基本相同。

IDE 也有生成 equals 和 hashCode 方法的功能，但是生成的源代码比使用 AutoValue 的代码更冗长，可读性更差，不会自动跟踪类中的变化，因此需要进行测试。也就是说，让 IDE 生成 equals（和 hashCode）方法通常比手动实现更可取，因为 IDE 不会出现粗心的错误，而人会。

总之，除非必须，否则不要覆盖 equals 方法：在许多情况下，从 Object 继承而来的实现正是你想要的。如果你确实覆盖了 equals，那么一定要比较类的所有重要字段，并以保留 equals 约定的所有 5 项规定的方式进行比较。

11 覆写 equals 时候总要覆写 hashCode

在覆盖 equals 的类中，必须覆盖 hashCode。 如果你没有这样做，你的类将违反 hashCode 的一般约定，这将阻止该类在 HashMap 和 HashSet 等集合中正常运行。以下是根据目标规范修改的约定：

当在应用程序执行期间对对象重复调用 hashCode 方法时，它必须一致地返回相同的值，前提是不对 equals 比较中使用的信息进行修改。这个值不需要在应用程序的不同执行之间保持一致。

如果根据 equals(Object) 方法判断出两个对象是相等的，那么在两个对象上调用 hashCode 必须产生相同的整数结果。

如果根据 equals(Object) 方法判断出两个对象不相等，则不需要在每个对象上调用 hashCode 时必须产生不同的结果。但是，程序员应该知道，为不相等的对象生成不同的结果可能会提高 hash 表的性能。

当你无法覆盖 hashCode 时，违反的关键条款是第二个：相等的对象必须具有相等的 hash 代码。根据类的 equals 方法，两个不同的实例在逻辑上可能是相等的，但是对于对象的 hashCode 方法来说，它们只是两个没有什么共同之处的对象。因此，Object 的 hashCode 方法返回两个看似随机的数字，而不是约定要求的两个相等的数字。例如，假设你尝试使用 Item-10 中的 PhoneNumber 类实例作为 HashMap 中的键：

```
Map<PhoneNumber, String> m = new HashMap<>();

m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

此时，你可能期望 m.get(new PhoneNumber(707, 867,5309)) 返回「Jenny」，但是它返回 null。注意，这里涉及到两个 PhoneNumber 实例：一个用于插入到 HashMap 中，另一个 equal 实例用于尝试检索。PhoneNumber 类未能覆盖 hashCode，导致两个相等的实例具有不相等的 hash 代码，这违反了 hashCode 约定。因此，get 方法查找电话号码的 hash 桶可能会在与 put 方法存储电话号码的 hash 桶不同。即使这两个实例碰巧 hash 到同一个 hash 桶上，get 方法几乎肯定会返回 null，因为 HashMap 有一个优化，它缓存与每个条目相关联的 hash 代码，如果 hash 代码不匹配，就不会检查对象是否相等。

解决这个问题就像为 PhoneNumber 编写一个正确的 hashCode 方法一样简单。那么 hashCode 方法应该是什么样的呢？写一个不好的很简单。举个例子，这个方法总是合法的，但是不应该被使用：

```
// The worst possible legal hashCode implementation - never use!

@Override

public int hashCode() { return 42; }
```

它是合法的，因为它确保了相等的对象具有相同的 hash 代码。同时它也很糟糕，因为它使每个对象都有相同的 hash 代码。因此，每个对象都 hash 到同一个桶中， hash 表退化为链表。应以线性时间替代运行的程序。对于大型 hash 表，这是工作和不工作的区别。

一个好的 hash 函数倾向于为不相等的实例生成不相等的 hash 代码。这正是 hashCode 约定的第三部分的含义。理想情况下， hash 函数应该在所有 int 值之间均匀分布所有不相等实例的合理集合。实现这个理想是很困难的。幸运的是，实现一个类似的并不太难。这里有一个简单的方式：

1、声明一个名为 result 的 int 变量，并将其初始化为对象中第一个重要字段的 hash 代码 c，如步骤 2.a 中计算的那样。（回想一下 Item-10 中的重要字段是影响相等比较的字段。）

2、对象中剩余的重要字段 f，执行以下操作：

a.为字段计算一个整数 hash 码 c：

i.如果字段是基本数据类型，计算 Type.hashCode(f)，其中 type 是与 f 类型对应的包装类。

ii.如果字段是对象引用，并且该类的 equals 方法通过递归调用 equals 来比较字段，则递归调用字段上的 hashCode。如果需要更复杂的比较，则为该字段计算一个「规范表示」，并在规范表示上调用 hashCode。如果字段的值为空，则使用 0（或其他常数，但 0 是惯用的）。

iii.如果字段是一个数组，则将其视为每个重要元素都是一个单独的字段。也就是说，通过递归地应用这些规则计算每个重要元素的 hash 代码，并将每个步骤 2.b 的值组合起来。如果数组中没有重要元素，则使用常量，最好不是 0。如果所有元素都很重要，那么使用 Arrays.hashCode。

b. 将步骤 2.a 中计算的 hash 代码 c 合并到结果，如下所示：

```
result = 31 * result + c;
```

3、返回 result。

当你完成了 hashCode 方法的编写之后，问问自己相同的实例是否具有相同的 hash 代码。编写单元测试来验证你的直觉（除非你使用 AutoValue 生成你的 equals 和 hashCode 方法，在这种情况下你可以安全地省略这些测试）。如果相同的实例有不相等的 hash 码，找出原因并修复问题。

可以从 hash 代码计算中排除派生字段。换句话说，你可以忽略任何可以从计算中包含的字段计算其值的字段。你必须排除不用于对等比较的任何字段，否

第二步的乘法。b 使结果取决于字段的顺序，如果类有多个相似的字段，则产生一个更好的 hash 函数。例如，如果字符串 hash 函数中省略了乘法，那么所有的字谜都有相同的 hash 码。选择 31 是因为它是奇素数。如果是偶数，乘法运算就会溢出，信息就会丢失，因为乘法运算等于移位。使用素数的好处不太明显，但它是传统的。31 的一个很好的特性是，可以用移位和减法来代替乘法，从而在某些体系结构上获得更好的性能：31 * i == (i <<5) – i。现代虚拟机自动进行这种优化。

让我们将前面的方法应用到 PhoneNumber 类：

```
// Typical hashCode method
@Override
public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

因为这个方法返回一个简单的确定性计算的结果，它的唯一输入是

PhoneNumber 实例中的三个重要字段，所以很明显，相等的 PhoneNumber 实例具有相等的 hash 码。实际上，这个方法是 PhoneNumber 的一个非常好的 hashCode 实现，与 Java 库中的 hashCode 实现相当。它很简单，速度也相当快，并且合理地将不相等的电话号码分散到不同的 hash 桶中。

虽然本条目中的方法产生了相当不错的 hash 函数，但它们并不是最先进的。它们的质量可与 Java 库的值类型中的 hash 函数相媲美，对于大多数用途来说都是足够的。如果你确实需要不太可能产生冲突的 hash 函数，请参阅 Guava 的 com.google.common.hash.Hashing [Guava]。

对象类有一个静态方法，它接受任意数量的对象并返回它们的 hash 代码。这个名为 hash 的方法允许你编写一行 hash 代码方法，这些方法的质量可以与根据本项中的菜谱编写的方法媲美。不幸的是，它们运行得更慢，因为它们需要创建数组来传递可变数量的参数，如果任何参数是原始类型的，则需要进行装箱和拆箱。推荐只在性能不重要的情况下使用这种 hash 函数。下面是使用这种技术编写的 PhoneNumber 的 hash 函数：

```
// One-line hashCode method - mediocre performance
@Override
public int hashCode() {
    return Objects.hash(lineNum, prefix, areaCode);
}
```

如果一个类是不可变的，并且计算 hash 代码的成本非常高，那么你可以考虑在对象中缓存 hash 代码，而不是在每次请求时重新计算它。如果你认为这种类型的大多数对象都将用作 hash 键，那么你应该在创建实例时计算 hash 代码。否则，你可能选择在第一次调用 hash 代码时延迟初始化 hash 代码。在一个延迟初始化的字段（Item-83）的情况下，需要一些注意来确保该类仍然是线程安全的。我们的 PhoneNumber 类不值得进行这种处理，但只是为了向你展示它是如何实现的，在这里。注意，hashCode 字段的初始值（在本例中为 0）不应该是通常创建的实例的 hash 代码：

```
// hashCode method with lazily initialized cached hash code
private int hashCode; // Automatically initialized to 0
@Override
public int hashCode() {
    int result = hashCode;
```

```
        if (result == 0) {
            result = Short.hashCode(areaCode);
            result = 31 * result + Short.hashCode(prefix);
            result = 31 * result + Short.hashCode(lineNum);
            hashCode = result;
        }
        return result;
    }
```

不要试图从 hash 代码计算中排除重要字段，以提高性能。 虽然得到的 hash 函数可能运行得更快，但其糟糕的质量可能会将 hash 表的性能降低到无法使用的程度。特别是， hash 函数可能会遇到大量实例，这些实例主要在你选择忽略的区域不同。如果发生这种情况， hash 函数将把所有这些实例映射到一些 hash 代码，应该在线性时间内运行的程序将在二次时间内运行。

这不仅仅是一个理论问题。在 Java 2 之前，字符串 hash 函数在字符串中，以第一个字符开始，最多使用 16 个字符。对于大量的分级名称集合（如 url），该函数完全显示了前面描述的病态行为。

不要为 hashCode 返回的值提供详细的规范，这样客户端就不能合理地依赖它。这（也）给了你更改它的灵活性。 Java 库中的许多类，例如 String 和 Integer，都将 hashCode 方法返回的确切值指定为实例值的函数。这不是一个好主意，而是一个我们不得不面对的错误：它阻碍了在未来版本中改进 hash 函数的能力。如果你保留了未指定的细节，并且在 hash 函数中发现了缺陷，或者发现了更好的 hash 函数，那么你可以在后续版本中更改它。

总之，每次覆盖 equals 时都必须覆盖 hashCode，否则程序将无法正确运行。你的 hashCode 方法必须遵守 Object 中指定的通用约定，并且必须合理地将不相等的 hash 代码分配给不相等的实例。这很容易实现，如果有点乏味，可使用第 51 页的方法。如 Item-10 所述，AutoValue 框架提供了一种很好的替代手动编写 equals 和 hashCode 的方法，IDE 也提供了这种功能。

12 始终覆写 toString

虽然 Object 提供 toString 方法的实现，但它返回的字符串通常不是类的用户希望看到的。它由后跟「at」符号（@）的类名和 hash 代码的无符号十六进制表示（例如 PhoneNumber@163b91）组成。toString 的通用约定是这么描

述的，返回的字符串应该是「简洁但信息丰富的表示，易于阅读」。虽然有人认为 PhoneNumber@163b91 简洁易懂，但与 707-867-5309 相比，它的信息量并不大。toString 约定接着描述，「建议所有子类覆盖此方法。」好建议，确实！

虽然它不如遵守 equals 和 hashCode 约定（Item-10 和 Item-11）那么重要，但是 提供一个好的 toString 实现（能）使类更易于使用，使用该类的系统（也）更易于调试。 当对象被传递给 println、printf、字符串连接操作符或断言或由调试器打印时，将自动调用 toString 方法。即使你从来没有调用 toString 对象，其他人也可能（使用）。例如，有对象引用的组件可以在日志错误消息中包含对象的字符串表示。如果你未能覆盖 toString，则该消息可能完全无用。

如果你已经为 PhoneNumber 提供了一个好的 toString 方法，那么生成一个有用的诊断消息就像这样简单：

```
System.out.println("Failed to connect to " + phoneNumber);
```

无论你是否覆盖 toString，程序员都会以这种方式生成诊断消息，但是除非你（覆盖 toString），否则这些消息不会有用。提供好的 toString 方法的好处不仅仅是将类的实例扩展到包含对这些实例的引用的对象，特别是集合。在打印 map 时，你更愿意看到哪个，{Jenny=PhoneNumber@163b91} 还是 {Jenny=707-867-5309}？

当实际使用时，toString 方法应该返回对象中包含的所有有趣信息， 如电话号码示例所示。如果对象很大，或者包含不利于字符串表示的状态，那么这种方法是不切实际的。在这种情况下，toString 应该返回一个摘要，例如曼哈顿住宅电话目录（1487536 号清单）或 Thread[main,5,main]。理想情况下，字符串应该是不言自明的。（线程示例未能通过此测试。）如果没有在字符串表示中包含所有对象的有趣信息，那么一个特别恼人的惩罚就是测试失败报告，如下所示：

```
Assertion failure: expected {abc, 123}, but was {abc, 123}.
```

在实现 toString 方法时，你必须做的一个重要决定是是否在文档中指定返回值的格式。建议你针对值类（如电话号码或矩阵）这样做。指定格式的优点是，它可以作为对象的标准的、明确的、人类可读的表示。这种表示可以用于输入和输出，也可以用于持久的人类可读数据对象，比如 CSV 文件。如果指定了格式，提供一个匹配的静态工厂或构造函数通常是一个好主意，这样程序员就可以轻松地在对象及其字符串表示之间来回转换。Java 库中的许多值类都采用这种方法，包括 BigInteger、BigDecimal 和大多数包装类。

指定 toString 返回值的格式的缺点是，一旦指定了它，就会终生使用它，假设你的类被广泛使用。程序员将编写代码来解析表示、生成表示并将其嵌入

持久数据中。如果你在将来的版本中更改了表示形式，你将破坏它们的代码和数据，它们将发出大量的消息。通过选择不指定格式，你可以保留在后续版本中添加信息或改进格式的灵活性。

无论你是否决定指定格式，你都应该清楚地记录你的意图。 如果指定了格式，则应该精确地指定格式。例如，这里有一个 toString 方法用于 Item-11 中的 PhoneNumber 类：

```
/**
* Returns the string representation of this phone number.
* The string consists of twelve characters whose format is
* "XXX-YYY-ZZZZ", where XXX is the area code, YYY is the
* prefix, and ZZZZ is the line number. Each of the capital
* letters represents a single decimal digit.
**
If any of the three parts of this phone number is too small
* to fill up its field, the field is padded with leading zeros.
* For example, if the value of the line number is 123, the last
* four characters of the string representation will be "0123".
*/
@Override
public String toString() {
    return String.format("%03d-%03d-%04d", areaCode, prefix, lineNum);
}
```

如果你决定不指定一种格式，文档注释应该如下所示：

```
/**
* Returns a brief description of this potion. The exact details
* of the representation are unspecified and subject to change,
* but the following may be regarded as typical:
**
```

```
    "[Potion #9: type=love, smell=turpentine, look=india ink]"
    */
    @Override
    public String toString() { ... }
```

在阅读了这篇文档注释之后，当格式被更改时，生成依赖于格式细节的代码或持久数据的程序员将只能怪他们自己。

无论你是否指定了格式，都要 提供对 toString 返回值中包含的信息的程序性访问。 例如，PhoneNumber 类应该包含区域代码、前缀和行号的访问器。如果做不到这一点，就会迫使需要这些信息的程序员解析字符串。除了降低性能和使程序员不必要的工作之外，这个过程很容易出错，并且会导致脆弱的系统在你更改格式时崩溃。由于没有提供访问器，你可以将字符串格式转换为事实上的 API，即使你已经指定了它可能会发生更改。

在静态实用程序类中编写 toString 方法是没有意义的（Item-4），在大多数 enum 类型中也不应该编写 toString 方法（Item-34），因为 Java 为你提供了一个非常好的方法。但是，你应该在任何抽象类中编写 toString 方法，该类的子类共享公共的字符串表示形式。例如，大多数集合实现上的 toString 方法都继承自抽象集合类。

谷歌的开放源码自动值工具（在 Item-10 中讨论）将为你生成 toString 方法，大多数 IDE 也是如此。这些方法可以很好地告诉你每个字段的内容，但并不专门针对类的含义。因此，例如，对于 PhoneNumber 类使用自动生成的 toString 方法是不合适的（因为电话号码具有标准的字符串表示形式），但是对于 Potion 类来说它是完全可以接受的。也就是说，一个自动生成的 toString 方法要比从对象继承的方法好得多，对象继承的方法不会告诉你对象的值。

回顾一下，在你编写的每个实例化类中覆盖对象的 toString 实现，除非超类已经这样做了。它使类更易于使用，并有助于调试。toString 方法应该以美观的格式返回对象的简明、有用的描述。

13 小心覆写 clone

Cloneable 接口的目的是作为 mixin 接口（Item-20），用于让类来宣称它们允许克隆。不幸的是，它没有达到这个目的。它的主要缺点是缺少 clone 方法，并且 Object 类的 clone 方法是受保护的。如果不求助于反射（Item-65），就不能仅仅因为对象实现了 Cloneable 就能调用 clone 方法。即使反射调用也可能失败，因为不能保证对象具有可访问的 clone 方法。尽管存在这样那样的

缺陷，但该设施的使用范围相当广泛，因此理解它是值得的。本项目将告诉你如何实现行为良好的 clone 方法，讨论什么时候应该这样做，并提供替代方法。

译注：mixin 是掺合，混合，糅合的意思，即可以将任意一个对象的全部或部分属性拷贝到另一个对象上。

如果 Cloneable 不包含任何方法，它会做什么呢？它决定 Object 的受保护克隆实现的行为：如果一个类实现了 Cloneable，对象的克隆方法返回对象的逐域拷贝；否则它会抛出 CloneNotSupportedException。这是接口的一种高度非典型使用，而不是可模仿的。通常，实现接口说明了类可以为其客户做些什么。在本例中，它修改了超类上受保护的方法行为。

虽然规范没有说明，但是在实践中，一个实现 Cloneable 的类应该提供一个功能正常的公共 clone 方法。为了实现这一点，类及其所有超类必须遵守复杂的、不可强制执行的、文档很少的协议。产生的机制是脆弱的、危险的和非语言的：即它创建对象而不调用构造函数。

clone 方法的一般约定很薄弱。这里是从 Object 规范复制过来的：

创建并返回此对象的副本。「复制」的确切含义可能取决于对象的类别。一般的目的是，对于任何对象 x，表达式

```
x.clone() != x
```

值将为 true，并且这个表达式

```
x.clone().getClass() == x.getClass()
```

值将为 true，但这些不是绝对的必要条件。通常情况下

```
x.clone().equals(x)
```

值将为 true，但这些不是绝对的必要条件。

按照惯例，这个方法返回的对象应该通过调用 super.clone 来获得。如果一个类和它的所有超类（对象除外）都遵守这个约定，那么情况就是这样

```
x.clone().getClass() == x.getClass().
```

按照惯例，返回的对象应该独立于被克隆的对象。为了实现这种独立性，可能需要修改 super 返回的对象的一个或多个字段。在返回之前克隆。

这种机制有点类似于构造函数链接，只是没有强制执行：如果一个类的克隆方法返回的实例不是通过调用 super.clone 而是通过调用构造函数获得的，编译器不会抱怨，但是如果这个类的一个子类调用 super.clone,由此产生的对象将有错误的类,防止子类克隆方法从正常工作。如果覆盖克隆的类是 final 的，那

么可以安全地忽略这个约定，因为不需要担心子类。但是如果 final 类有一个不调用 super 的克隆方法。类没有理由实现 Cloneable，因为它不依赖于对象克隆实现的行为。

假设你希望在一个类中实现 Cloneable，该类的超类提供了一个表现良好的克隆方法。第一个叫 super.clone。返回的对象将是原始对象的完整功能副本。类中声明的任何字段都具有与原始字段相同的值。如果每个字段都包含一个基元值或对不可变对象的引用，那么返回的对象可能正是你所需要的，在这种情况下不需要进一步的处理。例如，对于 Item-11 中的 PhoneNumber 类就是这样，但是要注意，不可变类永远不应该提供克隆方法，因为它只会鼓励浪费复制。有了这个警告，以下是 PhoneNumber 的克隆方法的外观：

```
// Clone method for class with no references to mutable state
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError(); // Can't happen
    }
}
```

为了让这个方法工作，必须修改 PhoneNumber 的类声明，以表明它实现了 Cloneable。虽然 Object 的 clone 方法返回 Object，但是这个 clone 方法返回 PhoneNumber。这样做是合法的，也是可取的，因为 Java 支持协变返回类型。换句话说，覆盖方法的返回类型可以是被覆盖方法的返回类型的子类。这样就不需要在客户端中进行强制转换。我们必须打出超级的成绩。在返回对象之前从对象克隆到 PhoneNumber，但强制转换肯定会成功。

对 super.clone 的调用包含在 try-catch 块中。这是因为 Object 声明其克隆方法来抛出 CloneNotSupportedException。因为 PhoneNumber 实现了 Cloneable，所以我们知道对 super.clone 的调用将会成功。这个样板文件的需要表明 CloneNotSupportedException 应该是被选中的（Item-71）。

如果对象包含引用可变对象的字段，前面所示的简单克隆实现可能是灾难性的。例如，考虑 Item-7 中的堆栈类：

```
public class Stack {
    private Object[] elements;
```

```
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() {
       this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

假设你想让这个类是可克隆的。如果克隆方法只返回 super.clone()，则结果堆栈实例在其大小字段中将有正确的值，但其元素字段将引用与原始堆栈实例相同的数组。修改初始值将破坏克隆中的不变量，反之亦然。你将很快发现你的程序产生了无意义的结果或抛出 NullPointerException。

由于调用堆栈类中的唯一构造函数，这种情况永远不会发生。实际上，clone 方法充当构造函数;你必须确保它不会对原始对象造成伤害，并且在克隆上正确地建立不变量。为了使堆栈上的克隆方法正常工作，它必须复制堆栈的内部。最简单的方法是在元素数组上递归地调用 clone：

```
// Clone method for class with references to mutable state
@Override
public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

注意，我们不需要将 elements.clone 的结果强制转换到 Object[]。在数组上调用 clone 将返回一个数组，该数组的运行时和编译时类型与被克隆的数组相同。这是复制数组的首选习惯用法。实际上，数组是 clone 工具唯一引人注目的用途。

还要注意，如果元素字段是 final 的，早期的解决方案就无法工作，因为克隆将被禁止为字段分配新值。这是一个基本问题：与序列化一样，可克隆体系结构与正常使用引用可变对象的 final 字段不兼容，除非在对象与其克隆对象之间可以安全地共享可变对象。为了使类可克隆，可能需要从某些字段中删除最终修饰符。

仅仅递归地调用克隆并不总是足够的。例如，假设你正在为 hash 表编写一个克隆方法， hash 表的内部由一组 bucket 组成，每个 bucket 引用键-值对链表中的第一个条目。为了提高性能，类实现了自己的轻量级单链表，而不是在内部使用 java.util.LinkedList：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
    private static class Entry {
```

```
        final Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    } ... // Remainder omitted
}
```

假设你只是递归地克隆 bucket 数组，就像我们对 Stack 所做的那样：

```
// Broken clone method - results in shared mutable state!
@Override
public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

尽管克隆具有自己的 bucket 数组，但该数组引用的链接列表与原始链表相同，这很容易导致克隆和原始的不确定性行为。要解决这个问题，你必须复制包含每个 bucket 的链表。这里有一个常见的方法：

```
// Recursive clone method for class with complex mutable state
public class HashTable implements Cloneable {
```

```
    private Entry[] buckets = ...;
        private static class Entry {
            final Object key;
            Object value;
            Entry next;
            Entry(Object key, Object value, Entry next) {
                this.key = key;
                this.value = value;
                this.next = next;
            }
            // Recursively copy the linked list headed by this Entry
            Entry deepCopy() {
                return new Entry(key, value,next == null ? null :
next.deepCopy());
            }
        }
        @Override
        public HashTable clone() {
            try {
                HashTable result = (HashTable) super.clone();
                result.buckets = new Entry[buckets.length];
                for (int i = 0; i < buckets.length; i++)
                    if (buckets[i] != null)
                        result.buckets[i] = buckets[i].deepCopy();
                return result;
            } catch (CloneNotSupportedException e) {
                throw new AssertionError();
```

```
            }
        } ... // Remainder omitted
    }
```

私有类 HashTable.Entry 已经被增强为支持「深度复制」方法。HashTable 上的 clone 方法分配一个大小合适的新 bucket 数组，并遍历原始 bucket 数组，深度复制每个非空 bucket。条目上的 deepCopy 方法会递归地调用自己来复制以条目开头的整个链表。虽然这种技术很可爱，而且如果 bucket 不太长也可以很好地工作，但是克隆链表并不是一个好方法，因为它为链表中的每个元素消耗一个堆栈帧。如果列表很长，很容易导致堆栈溢出。为了防止这种情况的发生，你可以用迭代替换 deepCopy 中的递归：

```
    // Iteratively copy the linked list headed by this Entry
    Entry deepCopy() {
        Entry result = new Entry(key, value, next);
        for (Entry p = result; p.next != null; p = p.next)
            p.next = new Entry(p.next.key, p.next.value, p.next.next);
        return result;
    }
```

克隆复杂可变对象的最后一种方法是调用 super.clone，将结果对象中的所有字段设置为初始状态，然后调用更高级别的方法重新生成原始对象的状态。在我们的 HashTable 示例中，bucket 字段将初始化为一个新的 bucket 数组，并且对于克隆的 hash 表中的每个键值映射将调用 put(key, value)方法（未显示）。这种方法通常产生一个简单、相当优雅的克隆方法，它的运行速度不如直接操作克隆的内部的方法快。虽然这种方法很简洁，但它与整个可克隆体系结构是对立的，因为它盲目地覆盖了构成体系结构基础的逐字段对象副本。

与构造函数一样，克隆方法决不能在正在构建的克隆上调用可覆盖方法（Item-19）。如果 clone 调用一个在子类中被重写的方法，这个方法将在子类有机会修复其在克隆中的状态之前执行，很可能导致克隆和原始的破坏。因此，前一段中讨论的 put(key, value)方法应该是 final 或 private 方法。（如果它是私有的，那么它可能是非最终公共方法的「助手方法」。）

对象的 clone 方法被声明为抛出 CloneNotSupportedException，但是重写方法不需要。公共克隆方法应该省略 throw 子句，作为不抛出受控异常的方法更容易使用（Item-71）。

在为继承设计类时，你有两种选择（Item-19），但是无论你选择哪一种，类都不应该实现 Cloneable。你可以选择通过实现一个功能正常的受保护克隆方法来模拟对象的行为，该方法声明为抛出 CloneNotSupportedException。这给子类实现 Cloneable 或不实现 Cloneable 的自由，就像它们直接扩展对象一样。或者，你可以选择不实现工作克隆方法，并通过提供以下简并克隆实现来防止子类实现一个工作克隆方法：

```
// clone method for extendable class not supporting Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

还有一个细节需要注意。如果你编写了一个实现了 Cloneable 的线程安全类，请记住它的克隆方法必须正确同步，就像其他任何方法一样（Item-78）。对象的克隆方法不是同步的，因此即使它的实现在其他方面是令人满意的，你也可能需要编写一个返回 super.clone()的同步克隆方法。

回顾一下，所有实现 Cloneable 的类都应该使用一个返回类型为类本身的公共方法覆盖 clone。这个方法应该首先调用 super.clone，然后修复任何需要修复的字段。通常，这意味着复制任何包含对象内部「深层结构」的可变对象，并将克隆对象对这些对象的引用替换为对其副本的引用。虽然这些内部副本通常可以通过递归调用 clone 来实现，但这并不总是最好的方法。如果类只包含基元字段或对不可变对象的引用，那么很可能不需要修复任何字段。这条规则也有例外。例如，表示序列号或其他唯一 ID 的字段需要固定，即使它是原始的或不可变的。

所有这些复杂性真的有必要吗？很少。如果你扩展了一个已经实现了 Cloneable 的类，那么除了实现行为良好的克隆方法之外，你别无选择。否则，最好提供对象复制的替代方法。一个更好的对象复制方法是提供一个复制构造函数或复制工厂。复制构造函数是一个简单的构造函数，它接受单个参数，其类型是包含构造函数的类，例如，

```
// Copy constructor
public Yum(Yum yum) { ... };
```

复制工厂是复制构造函数的静态工厂（Item-1）类似物：

```
// Copy factory
```

```
public static Yum newInstance(Yum yum) { ... };
```

复制构造函数方法及其静态工厂变体与克隆/克隆相比有许多优点：它们不依赖于易发生风险的语言外对象创建机制；他们不要求无法强制执行的约定；它们与最终字段的正确使用不冲突；它们不会抛出不必要的检查异常；而且不需要强制类型转换。

此外，复制构造函数或工厂可以接受类型为类实现的接口的参数。例如，按照约定，所有通用集合实现都提供一个构造函数，其参数为 collection 或 Map 类型。基于接口的复制构造函数和工厂（更确切地称为转换构造函数和转换工厂）允许客户端选择副本的实现类型，而不是强迫客户端接受原始的实现类型。例如，假设你有一个 HashSet s，并且希望将它复制为 TreeSet。克隆方法不能提供这种功能，但是使用转换构造函数很容易：new TreeSet<>(s)。

考虑到与 Cloneable 相关的所有问题，新的接口不应该扩展它，新的可扩展类不应该实现它。虽然 final 类实现 Cloneable 的危害要小一些，但这应该被视为一种性能优化，仅在极少数情况下（Item-67）是合理的。通常，复制功能最好由构造函数或工厂提供。这个规则的一个明显的例外是数组，最好使用 clone 方法来复制数组。

14 考虑实现 Comparable 接口

与本章讨论的其他方法不同，compareTo 方法不是在 Object 中声明的。相反，它是 Comparable 接口中的唯一方法。它在性质上类似于 Object 的 equals 方法，除了简单的相等比较之外，它还允许顺序比较，而且它是通用的。一个类实现 Comparable，表明实例具有自然顺序。对实现 Comparable 的对象数组进行排序非常简单：

```
Arrays.sort(a);
```

类似地，搜索、计算极值和维护 Comparable 对象的自动排序集合也很容易。例如，下面的程序依赖于 String 实现 Comparable 这一事实，将命令行参数列表按字母顺序打印出来，并消除重复：

```
public class WordList {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        Collections.addAll(s, args);
        System.out.println(s);
```

```
        }
    }
```

通过让类实现 Comparable，就可与依赖于此接口的所有通用算法和集合实现进行互操作。你只需付出一点点努力就能获得强大的功能。实际上，Java 库中的所有值类以及所有枚举类型（Item-34）都实现了 Comparable。如果编写的值类具有明显的自然顺序，如字母顺序、数字顺序或时间顺序，则应实现 Comparable 接口：

```
public interface Comparable<T> {
    int compareTo(T t);
}
```

compareTo 方法的一般约定类似于 equals 方法：

将一个对象与指定的对象进行顺序比较。当该对象小于、等于或大于指定对象时，对应返回一个负整数、零或正整数。如果指定对象的类型阻止它与该对象进行比较，则抛出 ClassCastException。

在下面的描述中，sgn(expression) 表示数学中的符号函数，它被定义为：根据传入表达式的值是负数、零或正数，对应返回 -1、0 或 1。

实现者必须确保所有 x 和 y 满足 sgn(x.compareTo(y)) == -sgn(y)（这意味着 x.compareTo(y) 当且仅当 y.compareTo(x)抛出异常时才抛出异常）。

实现者还必须确保关系是可传递的：(x.compareTo(y) > 0 && y.compareTo(z) > 0) 意味着 x.compareTo(z) > 0。

最后，实现者必须确保 x.compareTo(y) == 0 时，所有的 z 满足 sgn(x.compareTo(z)) == sgn(y.compareTo(z))。

强烈建议 (x.compareTo(y)== 0) == (x.equals(y)) 成立，但不是必需的。一般来说，任何实现 Comparable 接口并违反此条件的类都应该清楚地注明这一事实。推荐使用的表述是「注意：该类的自然顺序与 equals 不一致。」

不要被这些约定的数学性质所影响。就像 equals 约定（Item-10）一样，这个约定并不像看起来那么复杂。与 equals 方法不同，equals 方法对所有对象都施加了全局等价关系，compareTo 不需要跨越不同类型的对象工作：当遇到不同类型的对象时，compareTo 允许抛出 ClassCastException。通常，它就是这么做的。该约定确实允许类型间比较，这种比较通常在被比较对象实现的接口中定义。

就像违反 hashCode 约定的类可以破坏依赖 hash 的其他类一样，违反

compareTo 约定的类也可以破坏依赖 Comparable 的其他类。依赖 Comparable 的类包括排序集合 TreeSet 和 TreeMap，以及实用工具类 Collections 和 Arrays，它们都包含搜索和排序算法。

让我们看一下 compareTo 约定的细节。第一个规定指出，如果你颠倒两个对象引用之间的比较的方向，就应当发生这样的情况：如果第一个对象小于第二个对象，那么第二个对象必须大于第一个；如果第一个对象等于第二个对象，那么第二个对象一定等于第一个对象；如果第一个对象大于第二个对象，那么第二个对象一定小于第一个对象。第二个规定指出，如果一个对象大于第二个，第二个大于第三个，那么第一个对象一定大于第三个对象。最后一个规定指出，所有 compareTo 结果为相等的对象分别与任何其他对象相比，必须产生相同的结果。

这三种规定的一个结果是，由 compareTo 方法进行的相等性检验必须遵守由 equals 约定进行的相同的限制：反身性、对称性和传递性。因此，同样的警告也适用于此：除非你愿意放弃面向对象的抽象优点（Item-10），否则无法在保留 compareTo 约定的同时使用新值组件扩展可实例化类。同样的解决方案也适用。如果要向实现 Comparable 的类中添加值组件，不要继承它；编写一个不相关的类，其中包含第一个类的实例。然后提供返回所包含实例的「视图」方法。这使你可以自由地在包含类上实现你喜欢的任何 compareTo 方法，同时允许它的客户端在需要时将包含类的实例视为包含类的实例。

compareTo 约定的最后一段是一个强烈的建议，而不是一个真正的要求，它只是简单地说明了 compareTo 方法所施加的同等性检验通常应该与 equals 方法返回相同的结果。如果遵守了这一规定，则 compareTo 方法所施加的排序与 equals 方法一致。如果违反这条建议，那么它的顺序就与 equals 不一致。如果一个类的 compareTo 方法强加了一个与 equals 不一致的顺序，那么这个类仍然可以工作，但是包含该类元素的有序集合可能无法遵守集合接口（Collection、Set 或 Map）的一般约定。这是因为这些接口的一般约定是根据 equals 方法定义的，但是有序集合使用 compareTo 代替了 equals 实施同等性检验。如果发生这种情况，这不是一场灾难，但这是需要注意的。

例如，考虑 BigDecimal 类，它的 compareTo 方法与 equals 不一致。如果你创建一个空的 HashSet 实例，然后添加 new BigDecimal("1.0") 和 new BigDecimal("1.00")，那么该 HashSet 将包含两个元素，因为添加到该集合的两个 BigDecimal 实例在使用 equals 方法进行比较时结果是不相等的。但是，如果你使用 TreeSet 而不是 HashSet 执行相同的过程，那么该集合将只包含一个元素，因为使用 compareTo 方法比较两个 BigDecimal 实例时结果是相等的。（有关详细信息，请参阅 BigDecimal 文档。）

编写 compareTo 方法类似于编写 equals 方法，但是有一些关键的区别。

因为 Comparable 接口是参数化的，compareTo 方法是静态类型的，所以不需要进行类型检查或强制转换它的参数。如果参数类型错误，则该调用将不能编译。如果参数为 null，则调用应该抛出 NullPointerException，并且在方法尝试访问其成员时抛出该异常。

在 compareTo 方法中，字段是按顺序而不是按同等性来比较的。要比较对象引用字段，要递归调用 compareTo 方法。如果一个字段没有实现 Comparable，或者需要一个非标准的排序，那么应使用 Comparator。可以编写自定义的比较器，或使用现有的比较器，如 Item-10 中 CaseInsensitiveString 的 compareTo 方法：

```
// Single-field Comparable with object reference field
public final class CaseInsensitiveString implements
Comparable<CaseInsensitiveString> {
    public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    } ... // Remainder omitted
}
```

注意 CaseInsensitiveString 实现了 Comparable<CaseInsensitiveString>。这意味着 CaseInsensitiveString 引用只能与另一个 CaseInsensitiveString 引用进行比较。这是在声明实现 Comparable 的类时要遵循的常规模式。

本书的旧版本建议 compareTo 方法使用关系运算符 < 和 > 来比较整数基本类型字段，使用静态方法 Double.compare 和 Float.compare 来比较浮点基本类型字段。在 Java 7 中，静态比较方法被添加到所有 Java 的包装类中。在 compareTo 方法中使用关系运算符 < 和 > 冗长且容易出错，因此不再推荐使用。

如果一个类有多个重要字段，那么比较它们的顺序非常关键。从最重要的字段开始，一步步往下。如果比较的结果不是 0（用 0 表示相等），那么就完成了；直接返回结果。如果最重要的字段是相等的，就比较下一个最重要的字段，以此类推，直到找到一个不相等的字段或比较到最不重要的字段为止。下面是 Item-11 中 PhoneNumber 类的 compareTo 方法，演示了这种技术：

```
// Multiple-field Comparable with primitive fields
public int compareTo(PhoneNumber pn) {
    int result = Short.compare(areaCode, pn.areaCode);
```

```
        if (result == 0) {
            result = Short.compare(prefix, pn.prefix);
            if (result == 0)
                result = Short.compare(lineNum, pn.lineNum);
        }
        return result;
    }
```

在 Java 8 中，Comparator 接口配备了一组比较器构造方法，可以流畅地构造比较器。然后可以使用这些比较器来实现 Comparator 接口所要求的 compareTo 方法。许多程序员更喜欢这种方法的简明，尽管它存在一些性能成本：在我的机器上，PhoneNumber 实例的数组排序要慢 10% 左右。在使用这种方法时，请考虑使用 Java 的静态导入功能，这样你就可以通过静态比较器构造方法的简单名称来引用它们，以获得清晰和简洁。下面是 PhoneNumber 类的 compareTo 方法改进后的样子：

```
    // Comparable with comparator construction methods
    private static final Comparator<PhoneNumber> COMPARATOR =
comparingInt((PhoneNumber pn) -> pn.areaCode)
            .thenComparingInt(pn -> pn.prefix)
            .thenComparingInt(pn -> pn.lineNum);
    public int compareTo(PhoneNumber pn) {
        return COMPARATOR.compare(this, pn);
    }
```

译注 1：示例代码默认使用了静态导入：import static java.util.Comparator.comparingInt;

译注 2：comparingInt 及 thenComparingInt 的文档描述

```
    static <T> Comparator<T> comparingInt(ToIntFunction<? super T>
keyExtractor)
    Accepts a function that extracts an int sort key from a type T, and returns a
Comparator<T> that compares by that sort key.
    The returned comparator is serializable if the specified function is also
```

serializable.

接受从类型 T 中提取 int 排序 key 的函数，并返回与该排序 key 进行比较的 Comparator<T>。

如果指定的函数是可序列化的，则返回的比较器也是可序列化的。

Type Parameters:

T - the type of element to be compared

Parameters:

keyExtractor - the function used to extract the integer sort key

Returns:

a comparator that compares by an extracted key

Throws:

NullPointerException - if the argument is null

Since:

1.8

default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)

Returns a lexicographic-order comparator with a function that extracts a int sort key.

Implementation Requirements:

This default implementation behaves as if thenComparing(comparingInt(keyExtractor)).

返回具有提取 int 排序 key 的函数的字典顺序比较器。

实现要求：

此默认实现的行为类似于 thenComparing(comparingInt(keyExtractor))。

Parameters:

keyExtractor - the function used to extract the integer sort key

Returns:

a lexicographic-order comparator composed of this and then the int sort key

Throws:

NullPointerException - if the argument is null.

Since:

1.8

这个实现在类初始化时使用两个比较器构造方法构建一个比较器。第一个是 comparingInt。它是一个静态方法，接受一个 key 提取器函数，该函数将对象引用映射到 int 类型的 key ，并返回一个比较器，比较器根据该 key 对实例进行排序。在上述的示例中，comparingInt 使用 lambda 表达式从 PhoneNumber 中提取 areaCode，并返回 Comparator<PhoneNumber>，按区号来排序电话号码。注意，lambda 表达式显式地指定其输入参数的类型为 PhoneNumber。事实证明，在这种情况下，Java 的类型推断并没有强大到足以自己判断类型，因此我们不得不帮助它来编译程序。

如果两个电话号码有相同的区号，我们需要进一步改进比较，这正是第二个 comparator 构造方法 thenComparingInt 所做的。它是 Comparator 上的一个实例方法，它接受一个 int 类型的 key 提取函数，并返回一个比较器，该比较器首先应用原始比较器，然后使用提取的 key 来断开连接。你可以任意堆叠对 thenComparingInt 的调用，从而形成字典顺序。在上面的例子中，我们将两个对 thenComparingInt 的调用叠加起来，得到一个排序，它的第二个 key 是 prefix，而第三个 key 是 lineNum。注意，我们不必指定传递给两个调用 thenComparingInt 的 key 提取器函数的参数类型：Java 的类型推断足够智能，可以自行解决这个问题。

Comparator 类具有完整的构造方法。对于 long 和 double 的基本类型，有类似 comparingInt 和 thenComparingInt 的方法。int 版本还可以用于范围更小的整数类型，如 PhoneNumber 示例中的 short。double 版本也可以用于 float。Comparator 类提供的构造方法覆盖了所有 Java 数值基本类型。

也有对象引用类型的比较器构造方法。静态方法名为 compare，它有两个重载。一个是使用 key 提取器并使用 key 的自然顺序。第二种方法同时使用 key 提取器和比较器对提取的 key 进行比较。实例方法有三种重载，称为 thenComparing。一个重载只需要一个比较器并使用它来提供一个二级顺序。第二个重载只接受一个 key 提取器，并将 key 的自然顺序用作二级顺序。最后的重载需要一个 key 提取器和一个比较器来对提取的 key 进行比较。

有时候，你可能会看到 compareTo 或 compare 方法，它们依赖于以下事

实：如果第一个值小于第二个值，则两个值之间的差为负；如果两个值相等，则为零；如果第一个值大于零，则为正。下面是一个例子：

```
// BROKEN difference-based comparator - violates transitivity!
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return o1.hashCode() - o2.hashCode();
    }
};
```

不要使用这种技术。它充满了来自整数溢出和 IEEE 754 浮点运算构件的危险 [JLS 15.20.1, 15.21.1]。此外，生成的方法不太可能比使用本项目中描述的技术编写的方法快得多。应使用静态比较方法：

```
// Comparator based on static compare method
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};
```

或比较器构造方法：

```
// Comparator based on Comparator construction method
static Comparator<Object> hashCodeOrder = Comparator
    .comparingInt(o -> o.hashCode());
```

总之，无论何时实现具有排序性质的值类，都应该让类实现 Comparable 接口，这样就可以轻松地对实例进行排序、搜索，并与依赖于此接口的集合实现进行互操作。在 compareTo 方法的实现中比较字段值时，避免使用 < 和 > 操作符，应使用包装类中的静态比较方法或 Comparator 接口中的 comparator 构造方法。

第四章 类和接口

类和接口是 Java 编程语言的核心。它们是抽象的基本单位。该语言提供了许多强大的元素，你可以使用它们来设计类和接口。本章包含了帮助你充分利用这些元素的指导原则，以便你的类和接口是可用的、健壮的和灵活的。

15 使类和成员可见性最小

将设计良好的组件与设计糟糕的组件区别开来的最重要的因素是组件对其他组件隐藏内部数据和其他实现细节的程度。设计良好的组件隐藏了所有实现细节，干净地将 API 与实现分离。然后组件只通过它们的 api 进行通信，而不知道彼此的内部工作方式。这个概念被称为信息隐藏或封装，是软件设计的基本原则 [Parnas72]。

由于许多原因，信息隐藏是重要的，其中大部分原因源于这样一个事实：它解耦了组成系统的组件，允许它们被单独开发、测试、优化、使用、理解和修改。这加速了系统开发，因为组件可以并行开发。它减轻了维护的负担，因为组件可以被更快地理解、调试或替换，而不必担心会损害其他组件。虽然信息隐藏本身不会导致良好的性能，但它能够进行有效的性能调优：一旦系统完成，概要分析确定了哪些组件会导致性能问题（Item-67），就可以在不影响其他组件正确性的情况下对这些组件进行优化。信息隐藏增加了软件的重用性，因为没有紧密耦合的组件通常在其他上下文中被证明是有用的，除了开发它们的上下文。最后，信息隐藏降低了构建大型系统的风险，因为即使系统没有成功，单个组件也可能被证明是成功的。

Java 有许多工具来帮助隐藏信息。访问控制机制 [JLS, 6.6] 指定了类、接口和成员的可访问性。实体的可访问性由其声明的位置决定，如果有的话，则由声明中显示的访问修饰符（私有、受保护和公共）决定。正确使用这些修饰词是信息隐藏的关键。

经验法则很简单：让每个类或成员尽可能不可访问。换句话说，使用与你正在编写的软件的适当功能一致的尽可能低的访问级别。

对于顶级（非嵌套）类和接口，只有两个可能的访问级别：包私有和公共。如果声明一个顶级类或与公共修饰符的接口，它将是公共的；否则，它将是包私有的。如果顶级类或接口可以使包私有，那么它应该是私有的。通过使其包私有，你使其成为实现的一部分，而不是导出的 API，你可以在后续版本中修改、替换或消除它，而不必担心损害现有客户端。如果你将其公开，你有义务永远支持它以保持兼容性。

如果包私有顶级类或接口只被一个类使用，那么考虑将顶级类设置为使用它的唯一类的私有静态嵌套类（Item-24）。这降低了它从包中的所有类到使用

它的类的可访问性。但是，减少免费公共类的可访问性比减少包-私有顶级类的可访问性重要得多：公共类是包的 API 的一部分，而包-私有顶级类已经是包的实现的一部分。

对于成员（字段、方法、嵌套类和嵌套接口），有四个可能的访问级别，这里列出了增加可访问性的顺序：

- 私有，成员只能从声明它的顶级类中访问。
- 包级私有，成员可以从包中声明它的任何类访问。技术上称为默认访问，如果没有指定访问修饰符（接口成员除外，默认情况下，接口成员是公共的），就会得到这个访问级别。
- 保护，成员可以从声明它的类的子类（受一些限制 [JLS, 6.6.2])和包中声明它的任何类访问。
- 公共，该成员可以从任何地方访问。

在仔细设计了类的公共 API 之后，你的反射应该是使所有其他成员都是私有的。只有当同一包中的另一个类确实需要访问一个成员时，你才应该删除私有修饰符，使成员包成为私有的。如果你发现自己经常这样做，那么你应该重新检查系统的设计，看看另一个分解是否会产生更好地相互分离的类。也就是说，私有成员和包私有成员都是类实现的一部分，通常不会影响其导出的 API。但是，如果类实现了 Serializable（Item-86 和 Item-87），这些字段可能会「泄漏」到导出的 API 中。

对于公共类的成员来说，当访问级别从包私有到受保护时，可访问性会有很大的提高。受保护的成员是类导出 API 的一部分，必须永远支持。此外，导出类的受保护成员表示对实现细节的公开承诺（Item-19）。对受保护成员的需求应该相对较少。

有一个关键规则限制了你减少方法可访问性的能力。如果一个方法覆盖了超类方法，那么它在子类中的访问级别就不能比在超类 [JLS, 8.4.8.3] 中更严格。这对于确保子类的实例在超类的实例可用的任何地方都可用是必要的（Liskov 替换原则，请参阅 Item-15）。如果违反此规则，编译器将在尝试编译子类时生成错误消息。这个规则的一个特殊情况是，如果一个类实现了一个接口，那么该接口中的所有类方法都必须在类中声明为 public。

为了便于测试代码，你可能会倾向于使类、接口或成员比其他需要更容易访问。这在一定程度上是好的。为了测试一个公共类包的私有成员是可以接受的，但是提高可访问性是不可接受的。换句话说，将类、接口或成员作为包时

代导出的 API 的一部分以方便测试是不可接受的。幸运的是，也没有必要这样做，因为测试可以作为测试包的一部分运行，从而获得对包私有元素的访问权。

公共类的实例字段很少是公共的（Item-16）。如果实例字段是非 final 的，或者是对可变对象的引用，那么通过将其公开，你就放弃了限制字段中可以存储的值的能力。这意味着你放弃了强制包含字段的不变量的能力。此外，你还放弃了在修改字段时采取任何操作的能力，因此带有公共可变字段的 **类通常不是线程安全的。** 即使一个字段是 final 的，并且引用了一个不可变的对象，通过将其公开，你放弃了切换到一个新的内部数据表示的灵活性，而该字段并不存在。

同样的建议也适用于静态字段，只有一个例外。你可以通过公共静态 final 字段公开常量，假设这些常量是类提供的抽象的组成部分。按照惯例，这些字段的名称由大写字母组成，单词以下划线分隔（Item-68）。重要的是，这些字段要么包含原始值，要么包含对不可变对象的引用（Item-17）。包含对可变对象的引用的字段具有非 final 字段的所有缺点。虽然引用不能被修改，但是引用的对象可以被修改——这会导致灾难性的后果。

请注意，非零长度的数组总是可变的，因此对于类来说，拥有一个公共静态 final 数组字段或返回该字段的访问器是错误的。如果一个类具有这样的字段或访问器，客户端将能够修改数组的内容。这是一个常见的安全漏洞来源：

```
// Potential security hole!
public static final Thing[] VALUES = { ... };
```

要注意的是，一些 IDE 生成了返回私有数组字段引用的访问器，这恰恰导致了这个问题。有两种方法可以解决这个问题。你可以将公共数组设置为私有，并添加一个公共不可变列表：

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

或者，你可以将数组设置为私有，并添加一个返回私有数组副本的公共方法：

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values() {
    return PRIVATE_VALUES.clone();
```

```
}
```

要在这些备选方案中进行选择，请考虑客户可能会如何处理结果。哪种返回类型更方便？哪种表现会更好？

对于 Java 9，作为模块系统的一部分，还引入了另外两个隐式访问级别。模块是包的分组，就像包是类的分组一样。模块可以通过模块声明中的导出声明显式地导出它的一些包（按照约定包含在名为 module-info.java 的源文件中）。模块中未导出包的公共成员和受保护成员在模块外不可访问;在模块中，可访问性不受导出声明的影响。通过使用模块系统，你可以在模块内的包之间共享类，而不会让整个世界看到它们。未导出包中的公共类和受保护的成员产生了两个隐式访问级别，它们是正常公共级别和受保护级别的内部类似物。这种共享的需求相对较少，通常可以通过重新安排包中的类来消除。

与四个主要的访问级别不同，这两个基于模块的级别在很大程度上是顾问级别。如果你把一个模块的 JAR 文件在你的应用程序的类路径中,而不是模块路径,包模块恢复到他们 nonmodular 行为：所有的公共成员和保护成员包的公共类正常的可访问性,不管模块导出的包 [Reinhold,1.2]。严格执行新引入的访问级别的一个地方是 JDK 本身：Java 库中未导出的包在其模块之外确实不可访问。

对于典型的 Java 程序员来说，访问保护不仅是有限实用的模块所提供的，而且本质上是建议性的；为了利用它，你必须将包分组到模块中，在模块声明中显式地声明它们的所有依赖项，重新安排源代码树，并采取特殊操作以适应从模块中对非模块化包的任何访问 [Reinhold, 3]。现在说模块能否在 JDK 之外得到广泛使用还为时过早。与此同时，除非你有迫切的需求，否则最好还是避开它们。

总之，你应该尽可能减少程序元素的可访问性（在合理的范围内）。在仔细设计了一个最小的公共 API 之后，你应该防止任何游离的类、接口或成员成为 API 的一部分。除了作为常量的公共静态 final 字段外，公共类应该没有公共字段。确保公共静态 final 字段引用的对象是不可变的。

16 公有类中使用访问方法而非公有域

有时候，可能会编写一些退化类，这些类除了对实例字段进行分组之外，没有其他用途：

```
// Degenerate classes like this should not be public!

class Point {
```

```
        public double x;
        public double y;
    }
```

因为这些类的数据字段是直接访问的，所以这些类没有提供封装的好处（Item-15）。不改变 API 就不能改变表现形式，不能实施不变量，也不能在访问字段时采取辅助操作。坚持面向对象思维的程序员会认为这样的类是令人厌恶的，应该被使用私有字段和公共访问方法 getter 的类所取代，对于可变类，则是赋值方法 setter：

```
    // Encapsulation of data by accessor methods and mutators
    class Point {
        private double x;
        private double y;
        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
        public double getX() { return x; }
        public double getY() { return y; }
        public void setX(double x) { this.x = x; }
        public void setY(double y) { this.y = y; }
    }
```

当然，当涉及到公共类时，强硬派是正确的：如果类可以在包之外访问，那么提供访问器方法来保持更改类内部表示的灵活性。如果一个公共类公开其数据字段，那么改变其表示形式的所有希望都将落空，因为客户端代码可以广泛分发。

但是，如果一个类是包私有的或者是私有嵌套类，那么公开它的数据字段并没有什么本质上的错误——假设它们能够很好地描述类提供的抽象。无论是在类定义还是在使用它的客户端代码中，这种方法产生的视觉混乱都比访问方

法少。虽然客户端代码与类的内部表示绑定在一起，但这段代码仅限于包含该类的包。如果想要对表示形式进行更改，你可以在不接触包外部任何代码的情况下进行更改。对于私有嵌套类，更改的范围进一步限制在封闭类中。

Java 库中的几个类违反了公共类不应该直接公开字段的建议。突出的例子包括 java.awt 包中的 Point 和 Dimension。这些类不应被效仿，而应被视为警示。正如 Item-67 所述，公开 Dimension 类的内部结构导致了严重的性能问题，这种问题至今仍存在。

虽然公共类直接公开字段从来都不是一个好主意，但是如果字段是不可变的，那么危害就会小一些。你不能在不更改该类的 API 的情况下更改该类的表现形式，也不能在读取字段时采取辅助操作，但是你可以实施不变量。例如，这个类保证每个实例代表一个有效的时间：

```
// Public class with exposed immutable fields - questionable
public final class Time {
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOUR = 60;
    public final int hour;
    public final int minute;
    public Time(int hour, int minute) {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);
        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Min: " + minute);
        this.hour = hour;
        this.minute = minute;
    } ... // Remainder omitted
}
```

总之，公共类不应该公开可变字段。对于公共类来说，公开不可变字段的危害要小一些，但仍然存在潜在的问题。然而，有时候包私有或私有嵌套类需要公开字段，无论这个类是可变的还是不可变的。

17 使可变性最小

不可变类就是一个实例不能被修改的类。每个实例中包含的所有信息在对象的生命周期内都是固定的，因此永远不会观察到任何更改。Java 库包含许多不可变的类，包括 String、基本数据类型的包装类、BigInteger 和 BigDecimal。有很多很好的理由：不可变类比可变类更容易设计、实现和使用。它们不太容易出错，而且更安全。

要使类不可变，请遵循以下 5 条规则：

1．**不要提供修改对象状态的方法**（被称为赋值方法）

2．**确保类不能被扩展。**这可以防止粗心或恶意的子类以对象状态发生改变的方式行为，从而损害类的不可变行为。防止子类化通常通过使类成为 final 来完成，但是还有一种替代方法，我们将在后面讨论。

3．**所有字段用 final 修饰。**这清楚地表达了你的意图，方式是由系统执行的。同样，如果引用新创建的实例在没有同步的情况下从一个线程传递到另一个线程，那么就有必要确保正确的行为，就像内存模型中描述的那样 [JLS, 17.5;Goetz06, 16]。

4．**使所有字段为私有。**这将阻止客户端访问字段引用的可变对象并直接修改这些对象。虽然在技术上允许不可变类拥有包含基元值或对不可变对象的引用的公共 final 字段，但不建议这样做，因为它排除了在以后的版本中更改内部表示（Item-15 和 Item-16）。

5．**确保对任何可变组件的独占访问。**如果你的类有任何引用可变对象的字段，请确保该类的客户端无法获得对这些对象的引用。永远不要将这样的字段初始化为 clientprovide 对象引用或从访问器返回字段。在构造函数、访问器和 readObject 方法（Item-88）中创建防御性副本（Item-50）。

前面项目中的许多示例类都是不可变的。其中一个类是 Item-11 中的 PhoneNumber，它对每个属性都有访问器，但没有相应的赋值方法。下面是一个稍微复杂一点的例子：

```
// Immutable complex number class
```

```
public final class Complex {
    private final double re;
    private final double im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex minus(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }
    public Complex times(Complex c) {
        return new Complex(re * c.re - im * c.im,re * c.im + im * c.re);
    }
    public Complex dividedBy(Complex c) {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp, (im * c.re - re *
c.im) / tmp);
    }
    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
```

```
            if (!(o instanceof Complex))
                return false;
            Complex c = (Complex) o;
            // See page 47 to find out why we use compare instead of ==
            return Double.compare(c.re, re) == 0
                    && Double.compare(c.im, im) == 0;
        }
        @Override public int hashCode() {
            return 31 * Double.hashCode(re) + Double.hashCode(im);
        }
        @Override public String toString() {
            return "(" + re + " + " + im + "i)";
        }
    }
```

这个类表示一个复数（一个包含实部和虚部的数）。除了标准的对象方法之外，它还为实部和虚部提供访问器，并提供四种基本的算术运算：加法、减法、乘法和除法。需要注意的是，算术操作如何创建和返回一个新的复杂实例，而不是修改这个实例。这种模式称为泛函方法，因为方法返回对其操作数应用函数的结果，而不修改它。将其与方法将过程应用于其操作数的过程或命令式方法进行对比，使其状态发生变化。注意，方法名是介词（如 plus)，而不是动词（如 add)。这强调了一个事实，方法不会改变对象的值。BigInteger 和 BigDecimal 类不遵守这个命名约定，导致了许多使用错误。

如果你不熟悉函数方法，那么它可能看起来不自然，但是它支持不变性，这有很多优点。**不可变对象很简单。**一个不可变的对象可以恰好处于一种状态，即创建它的状态。如果你确保所有构造函数都建立了类不变量，那么就可以保证这些不变量将一直保持为真，而你和使用该类的程序员无需再做任何努力。另一方面，可变对象可以具有任意复杂的状态空间。如果文档没有提供 mutator 方法执行的状态转换的精确描述，那么可能很难或不可能可靠地使用可变类。

不可变对象本质上是线程安全的；它们不需要同步。它们不能被多线程并发地访问而损坏。这无疑是实现线程安全的最简单方法。由于任何线程都无法观察到另一个线程对不可变对象的任何影响，因此 **可以自由共享不可变对象。** 因此，不可变类应该鼓励客户尽可能重用现有的实例。一种简单的方法是为常用值提供公共静态最终常量。例如，复杂类可能提供以下常量：

```
public static final Complex ZERO = new Complex(0, 0);

public static final Complex ONE = new Complex(1, 0);

public static final Complex I = new Complex(0, 1);
```

这种方法可以更进一步。不可变类可以提供静态工厂（Item-1），这些工厂缓存经常请求的实例，以避免在现有实例可用时创建新实例。所有包装类和 BigInteger 都是这样做的。使用这种静态工厂会导致客户端共享实例而不是创建新实例，从而减少内存占用和垃圾收集成本。在设计新类时，选择静态工厂而不是公共构造函数，这使你能够灵活地在以后添加缓存，而无需修改客户端。

不可变对象可以自由共享这一事实的一个后果是，你永远不需要对它们进行防御性的复制（Item-50）。事实上，你根本不需要做任何拷贝，因为拷贝将永远等同于原件。因此，你不需要也不应该在不可变类上提供克隆方法或复制构造函数（Item-13）。这在 Java 平台的早期并没有得到很好的理解，因此 String 类确实有一个复制构造函数，但是如果有的话，应该很少使用它（Item-6）。

你不仅可以共享不可变对象，而且可以共享它们的内部。 例如，BigInteger 类在内部使用符号幅度表示。符号用 int 表示，星等用 int 数组表示。反求法产生了一个大小相同、符号相反的大整数。即使数组是可变的，它也不需要复制;新创建的 BigInteger 指向与原始的内部数组相同的内部数组。

不可变对象是其他对象的很好的构建模块， 无论是可变的还是不可变的。如果知道复杂对象的组件对象不会在其下面发生更改，那么维护复杂对象的不变性就会容易得多。这个原则的一个特殊情况是，不可变对象会生成很棒的 map 键和 set 元素：你不必担心它们的值在 map 或 set 中发生变化，这会破坏 map 或 set 的不变量。

不可变对象免费提供故障原子性（Item-76）。他们的状态从未改变，所以不可能出现暂时的不一致。

不可变类的主要缺点是每个不同的值都需要一个单独的对象。创建这些对象的成本可能很高，尤其是如果对象很大的话。例如，假设你有一个百万比特的大整数，你想改变它的低阶位：

```
BigInteger moby = ...;

moby = moby.flipBit(0);
```

flipBit 方法创建了一个新的 BigInteger 实例，也有 100 万比特长，只在一个比特上与原始的不同。该操作需要与 BigInteger 的大小成比例的时间和空间。与 java.util.BitSet 形成对比。与 BigInteger 一样，BitSet 表示任意长的位序列，但与 BigInteger 不同，BitSet 是可变的。BitSet 类提供了一种方法，可以让你在固定的时间内改变百万比特实例的单个位的状态：

```
BitSet moby = ...;

moby.flip(0);
```

如果执行多步操作，在每一步生成一个新对象，最终丢弃除最终结果之外的所有对象，那么性能问题就会增大。有两种方法可以解决这个问题。第一种方法是猜测通常需要哪些多步操作，并将它们作为原语提供。如果将多步操作作为基元提供，则不可变类不必在每个步骤中创建单独的对象。在内部，不可变类可以任意聪明。例如，BigInteger 有一个包私有的可变「伴随类」，它使用这个类来加速多步操作，比如模块化求幂。由于前面列出的所有原因，使用可变伴随类要比使用 BigInteger 难得多。幸运的是，你不必使用它：BigInteger 的实现者为你做了艰苦的工作。

如果你能够准确地预测客户端希望在不可变类上执行哪些复杂操作，那么包私有可变伴随类方法就可以很好地工作。如果不是，那么你最好的选择就是提供一个公共可变伴随类。这种方法在 Java 库中的主要示例是 String 类，它的可变伴随类是 StringBuilder（及其过时的前身 StringBuffer)。

既然你已经知道了如何创建不可变类，并且了解了不可变性的优缺点，那么让我们来讨论一些设计方案。回想一下，为了保证不变性，类不允许自己被子类化。这可以通过期末考试来完成，但是还有另外一个更灵活的选择。与使不可变类成为 final 不同，你可以将其所有构造函数变为私有或包-私有，并在公共构造函数的位置添加公共静态工厂（Item-1）。

```
// Immutable class with static factories instead of constructors

public class Complex {

    private final double re;

    private final double im;

    private Complex(double re, double im) {
```

```
            this.re = re;
            this.im = im;
        }
        public static Complex valueOf(double re, double im) {
            return new Complex(re, im);
        }
        ... // Remainder unchanged
    }
```

这种方法通常是最好的选择。它是最灵活的，因为它允许使用多个包私有实现类。对于驻留在包之外的客户端，不可变类实际上是最终类，因为不可能扩展来自另一个包的类，因为它缺少公共或受保护的构造函数。除了允许多实现类的灵活性之外，这种方法还通过改进静态工厂的对象缓存功能，使在后续版本中调优该类的性能成为可能。

当编写 BigInteger 和 BigDecimal 时，不可变类必须是有效的 final，因此可以重写它们的所有方法，这一点没有得到广泛的理解。遗憾的是，在保留向后兼容性的情况下，这一问题无法在事后得到纠正。如果你编写的类的安全性依赖于来自不受信任客户端的 BigInteger 或 BigDecimal 参数的不可变性，那么你必须检查该参数是否是「真正的」BigInteger 或 BigDecimal，而不是不受信任子类的实例。如果是后者，你必须防御地复制它，假设它可能是可变的（Item-50）：

```
    public static BigInteger safeInstance(BigInteger val) {
    return val.getClass() == BigInteger.class ?
    val : new BigInteger(val.toByteArray());
    }
```

这个项目开头的不可变类的规则列表说，没有方法可以修改对象，它的所有字段必须是 final 的。实际上，这些规则比必要的要强大一些，可以通过放松来提高性能。实际上，任何方法都不能在对象的状态中产生外部可见的更改。然而，一些不可变类有一个或多个非最终字段，它们在第一次需要这些字段时，就会在其中缓存昂贵计算的结果。如果再次请求相同的值，则返回缓存的值，

从而节省了重新计算的成本。这个技巧之所以有效，是因为对象是不可变的，这保证了如果重复计算，计算将产生相同的结果。

例如，PhoneNumber 的 hashCode 方法(Item-11，第 53 页）在第一次调用时计算哈希代码，并缓存它，以防再次调用它。这个技术是一个延迟初始化的例子（Item-83），String 也使用这个技术。

关于可序列化性，应该添加一个注意事项。如果你选择让不可变类实现 Serializable，并且它包含一个或多个引用可变对象的字段，那么你必须提供一个显式的 readObject 或 readResolve 方法，或者使用 ObjectOutputStream。writeUnshared ObjectInputStream。readUnshared 方法，即使默认的序列化形式是可

总结一下，抵制为每个 getter 编写 setter 的冲动。类应该是不可变的，除非有很好的理由让它们可变。不可变类提供了许多优点，它们唯一的缺点是在某些情况下可能出现性能问题。你应该始终创建小的值对象，例如 PhoneNumber 和 Complex、stable。（Java 库中有几个类，比如 java.util.Date 和 java.awt.Point，这本来是不可改变的，但事实并非如此。）你应该认真考虑将较大的值对象（如 String 和 BigInteger）设置为不可变的。只有在确认了实现满意性能的必要性之后，才应该为不可变类提供一个公共可变伴随类（Item-67）。

有些类的不变性是不切实际的。如果一个类不能成为不可变的，那么就尽可能地限制它的可变性。 减少对象可能存在的状态数可以使对对象进行推理更容易，并减少出错的可能性。因此，除非有令人信服的理由使每个字段成为 final。将此项目的建议与项目 15 的建议结合起来，你的自然倾向应该是 声明每个字段为私有 final，除非有很好的理由这样做。

构造函数应该创建完全初始化的对象，并建立所有的不变量。 不要提供与构造函数或静态工厂分离的公共初始化方法，除非有充分的理由这样做。类似地，不要提供「重新初始化」的方法，该方法允许重用对象，就好像它是用不同的初始状态构造的一样。这些方法通常只提供很少的性能收益，而代价是增加了复杂性。

CountDownLatch 类演示了这些原理。它是可变的，但它的状态空间故意保持小。你创建一个实例，使用一次，就完成了：一旦倒计时锁的计数达到零，你可能不会重用它。

关于这个项目中的复杂类，应该添加最后一个注意事项。这个例子只是为了说明不变性。它不是一个工业强度的复数实现。它使用了复杂乘法和除法的

标准公式，这些公式没有被正确地四舍五入，并且为复杂的 NaNs 和 infinities 提供了糟糕的语义 [Kahan91, Smith62, Thomas94]。

18 组合优于继承

继承是实现代码重用的一种强大方法，但它并不总是最佳的工具。使用不当会导致软件脆弱。在包中使用继承是安全的，其中子类和超类实现由相同的程序员控制。在扩展专门为扩展设计和文档化的类时使用继承也是安全的（Item-19）。然而，对普通的具体类进行跨包边界的继承是危险的。作为提醒，本书使用「继承」一词来表示实现继承（当一个类扩展另一个类时）。本项目中讨论的问题不适用于接口继承（当类实现接口或一个接口扩展另一个接口时）。

与方法调用不同，继承违反了封装[Snyder86]。换句话说，子类的正确功能依赖于它的父类的实现细节。超类的实现可能在版本之间发生变化，如果发生了变化，子类可能会崩溃，即使它的代码没有被修改过。因此，子类必须与其父类同步发展，除非父类是专门为扩展的目的而设计的，并具有很好的文档说明。

为了使其更具体一些，让我们假设有一个使用 HashSet 的程序。为了优化程序的性能，我们需要查询 HashSet，以确定自创建以来添加了多少元素（不要与当前的大小混淆，当元素被删除时，当前的大小会递减）。为了提供这个功能，我们编写了一个 HashSet 变量，它记录试图插入的元素数量，并为这个计数导出一个访问。HashSet 类包含两个能够添加元素的方法，add 和 addAll，因此我们覆盖这两个方法：

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet() {
    }
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
```

```
        @Override
        public boolean add(E e) {
            addCount++;
            return super.add(e);
        }
        @Override
        public boolean addAll(Collection<? extends E> c) {
            addCount += c.size();
            return super.addAll(c);
        }
        public int getAddCount() {
            return addCount;
        }
    }
```

这个类看起来很合理，但是它不起作用。假设我们创建了一个实例，并使用 addAll 方法添加了三个元素。顺便说一下，我们使用 Java 9 中添加的静态工厂方法 List.of 创建了一个列表；如果你使用的是早期版本，那么使用 Arrays.asList:

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

我们希望 getAddCount 方法此时返回 3，但它返回 6。到底是哪里出了错？在内部，HashSet 的 addAll 方法是在其 add 方法之上实现的，尽管 HashSet 相当合理地没有记录这个实现细节。InstrumentedHashSet 中的 addAll 方法向 addCount 添加了三个元素，然后使用 super.addAll 调用 HashSet 的 addAll 实现。这反过来调用 add 方法（在 InstrumentedHashSet 中被重写过），每个元素一次。这三个调用中的每一个都向 addCount 添加了一个元素，总共增加了 6 个元素：使用 addAll 方法添加的每个元素都被重复计数。

我们可以通过消除 addAll 方法的覆盖来「修复」子类。虽然生成的类可以工作，但它的正确功能取决于 HashSet 的 addAll 方法是在 add 方法之上实现的事实。这种「自用」是实现细节，不能保证在 Java 平台的所有实现中都存在，也不能保证在版本之间进行更改。因此，结果得到的 InstrumentedHashSet 类是脆弱的。

重写 addAll 方法以遍历指定的集合稍微好一些，为每个元素调用一次 add 方法。无论 HashSet 的 addAll 方法是否在其 add 方法之上实现，这都将保证正确的结果，因为 HashSet 的 addAll 实现将不再被调用。然而，这种技术并不能解决我们所有的问题。它相当于重新实现超类方法，这可能会导致自使用，也可能不会，这是困难的、耗时的、容易出错的，并且可能会降低性能。此外，这并不总是可能的，因为如果不访问子类无法访问的私有字段，就无法实现某些方法。

子类脆弱的一个相关原因是他们的超类可以在后续版本中获得新的方法。假设一个程序的安全性取决于插入到某个集合中的所有元素满足某个谓词。这可以通过子类化集合和覆盖每个能够添加元素的方法来确保在添加元素之前满足谓词。这可以很好地工作，直到在后续版本中向超类中添加能够插入元素的新方法。一旦发生这种情况，只需调用新方法就可以添加「非法」元素，而新方法在子类中不会被覆盖。这不是一个纯粹的理论问题。当 Hashtable 和 Vector 被重新安装以加入集合框架时，必须修复这一性质的几个安全漏洞。

这两个问题都源于重写方法。你可能认为，如果只添加新方法，并且不覆盖现有方法，那么扩展类是安全的。虽然这种延长会更安全，但也不是没有风险。如果超类在随后的版本中获得了一个新方法，而你不幸给了子类一个具有相同签名和不同返回类型的方法，那么你的子类将不再编译 [JLS, 8.4.8.3]。如果给子类一个方法，该方法具有与新超类方法相同的签名和返回类型，那么现在要覆盖它，因此你要面对前面描述的问题。此外，你的方法是否能够完成新的超类方法的契约是值得怀疑的，因为在你编写子类方法时，该契约还没有被写入。

幸运的是，有一种方法可以避免上述所有问题。与其扩展现有类，不如为新类提供一个引用现有类实例的私有字段。这种设计称为复合，因为现有的类成为新类的一个组件。新类中的每个实例方法调用现有类的包含实例上的对应方法，并返回结果。这称为转发，新类中的方法称为转发方法。生成的类将非常坚固，不依赖于现有类的实现细节。即使向现有类添加新方法，也不会对新类产生影响。为了使其具体化，这里有一个使用复合和转发方法的 InstrumentedHashSet 的替代方法。注意，实现被分成两部分，类本身和一个可重用的转发类，其中包含所有的转发方法，没有其他内容：

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
        private int addCount = 0;
        public InstrumentedSet(Set<E> s) {
                super(s);
        }
        @Override
        public boolean add(E e) {
                addCount++;
                return super.add(e);
        }
        @Override
        public boolean addAll(Collection<? extends E> c) {
                addCount += c.size();
                return super.addAll(c);
        }
        public int getAddCount() {
                return addCount;
        }
    }
// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
        private final Set<E> s;
        public ForwardingSet(Set<E> s) { this.s = s; }
        public void clear() { s.clear(); }
```

```
        public boolean contains(Object o) { return s.contains(o); }
        public boolean isEmpty() { return s.isEmpty(); }
        public int size() { return s.size(); }
        public Iterator<E> iterator() { return s.iterator(); }
        public boolean add(E e) { return s.add(e); }
        public boolean remove(Object o) { return s.remove(o); }
        public boolean containsAll(Collection<?> c)
        { return s.containsAll(c); }
        public boolean addAll(Collection<? extends E> c)
        { return s.addAll(c); }
        public boolean removeAll(Collection<?> c)
        { return s.removeAll(c); }
        public boolean retainAll(Collection<?> c)
        { return s.retainAll(c); }
        public Object[] toArray() { return s.toArray(); }
        public <T> T[] toArray(T[] a) { return s.toArray(a); }
        @Override
        public boolean equals(Object o){ return s.equals(o); }

        @Override
        public int hashCode() { return s.hashCode(); }
        @Override
        public String toString() { return s.toString(); }
    }
```

InstrumentedSet 类的设计是通过 Set 接口来实现的，这个接口可以捕获 HashSet 类的功能。除了健壮外，这个设计非常灵活。InstrumentedSet 类实现了 Set 接口，有一个构造函数，它的参数也是 Set 类型的。实际上，这个类可以将一个 Set 转换成另一个 Set，添加了 instrumentation 的功能。基于继承的方法只适用于单个具体类，并且需要为超类中每个受支持的构造函数提供单独的构造函数，与此不同的是，包装器类可用于仪器任何集合实现，并将与任何现有构造函数一起工作：

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));

Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

The InstrumentedSet class can even be used to temporarily instrument a set instance that has already been used without instrumentation:

InstrumentedSet 类甚至还可以用来临时配置一个不用插装就可以使用的 set 实例：

```
static void walk(Set<Dog> dogs) {

InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);

... // Within this method use iDogs instead of dogs

}
```

InstrumentedSet 类被称为包装类，因为每个 entedset 实例都包含(「包装」)另一个集合实例。这也称为 Decorator 模式[Gamma95]，因为 InstrumentedSet 类通过添加插装来「修饰」一个集合。有时组合和转发的组合被松散地称为委托。严格来说，除非包装器对象将自身传递给包装对象，否则它不是委托[Lieberman86; Gamma95]。

包装类的缺点很少。一个警告是包装类不适合在回调框架中使用，在回调框架中，对象为后续调用(「回调」)将自定义传递给其他对象。因为包装对象不知道它的包装器，所以它传递一个对它自己的引用（this），回调避开包装器。这就是所谓的自我问题。有些人担心转发方法调用的性能影响或包装器对象的内存占用影响。这两种方法在实践中都没有多大影响。编写转发方法很麻烦，但是你必须只为每个接口编写一次可重用的转发类，而且可能会为你提供转发类。例如，Guava 为所有的集合接口提供了转发类[Guava]。

只有在子类确实是超类的子类型的情况下，继承才合适。换句话说，只有当两个类之间存在「is-a」关系时，类 B 才应该扩展类 a。如果你想让 B 类扩展 a 类，那就问问自己：每个 B 都是 a 吗？如果你不能如实回答是的这个问

题，B 不应该延长 a，如果答案是否定的，通常情况下，B 应该包含一个私人的实例，让不同的 API：不是 B 的一个重要组成部分，只是一个细节的实现。

在 Java 库中有许多明显违反这一原则的地方。例如，堆栈不是向量，因此堆栈不应该扩展向量。类似地，属性列表不是 hash 表，因此属性不应该扩展 hash 表。在这两种情况下，复合都是可取的。

如果在复合合适的地方使用继承，就不必要地公开实现细节。生成的 API 将你与原始实现绑定在一起，永远限制了类的性能。更严重的是，通过公开内部组件，你可以让客户端直接访问它们。至少，它会导致语义混乱。例如，如果 p 引用了一个属性实例，那么 p.getProperty(key) 可能会产生与 p.get(key) 不同的结果：前者考虑了默认值，而后者（从 Hashtable 继承而来）则不会。最严重的是，客户端可以通过直接修改超类来破坏子类的不变量。对于属性，设计者希望只允许字符串作为键和值，但是直接访问底层 hash 表允许违反这个不变量。一旦违反，就不再可能使用 Properties API 的其他部分（加载和存储）。当发现这个问题时，已经太晚了，无法纠正它，因为客户端依赖于非字符串键和值的使用。

在决定使用继承而不是复合之前，你应该问自己最后一组问题。你打算扩展的类在其 API 中有任何缺陷吗？如果是这样，你是否愿意将这些缺陷传播到类的 API 中？继承传播超类 API 中的任何缺陷，而复合允许你设计一个新的 API 来隐藏这些缺陷。

总而言之，继承是强大的，但是它是有问题的，因为它违反了封装。只有当子类和超类之间存在真正的子类型关系时才合适。即使这样，如果子类与超类不在一个不同的包中，并且超类不是为继承而设计的，继承也可能导致脆弱性。为了避免这种脆弱性，使用组合和转发而不是继承，特别是如果存在实现包装器类的适当接口的话。包装类不仅比子类更健壮，而且更强大。

19 要么为继承而设计并提供文档，要么禁止继承

Item-18 提醒你注意子类化不是为继承设计和文档化的「外部」类的危险。那么，为继承而设计和文档化的类意味着什么呢？

首先，类必须精确地在文档中记录覆盖任何方法的效果。换句话说，类必须在文档中记录它对可覆盖方法的自用。对于每个公共或受保护的方法，文档必须指出方法调用的可覆盖方法、调用顺序以及每次调用的结果如何影响后续处理过程。（可覆盖的意思是非 final 的，公共的或受保护的。)更一般地说，类必须记录它可能调用可覆盖方法的任何情况。例如，调用可能来自后台线程或静态初始化器。

调用可覆盖方法的方法在其文档注释末尾应包含这些调用的描述。描述在规范的一个特殊部分中，标记为「实现需求」，它由 Javadoc 标签 @implSpec 生成。本节描述该方法的内部工作方式。下面是一个示例，复制自 java.util.AbstractCollection 规范：

> public boolean remove(Object o)
>
> Removes a single instance of the specified（v. 指定；详细说明，adj. 规定的；） element from this collection, if it is present (optional operation). More formally, removes an element e such that Objects.equals(o, e), if this collection contains one or more such elements. Returns true if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).

从此集合中移除指定元素的单个实例，如果存在（可选操作）。更正式地说，如果此集合包含一个或多个这样的元素，则删除元素 e，使得 Objects.equals(o, e)，如果此 collection 包含指定的元素，则返回 true（或等效地，如果此集合因调用而更改）。

> Implementation Requirements: This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method and this collection contains the specified object.

实现需求：这个实现遍历集合，寻找指定的元素。如果找到元素，则使用迭代器的 remove 方法从集合中删除元素。注意，如果这个集合的迭代器方法返回的迭代器没有实现 remove 方法，并且这个集合包含指定的对象，那么这个实现将抛出 UnsupportedOperationException。

这篇文档无疑说明了重写迭代器方法将影响 remove 方法的行为。它还准确地描述了迭代器方法返回的迭代器的行为将如何影响 remove 方法的行为。与 Item-18 中的情况相反，在 Item-18 中，程序员子类化 HashSet 不能简单地说覆盖 add 方法是否会影响 addAll 方法的行为。

但是，这是否违背了一个格言：好的 API 文档应该描述一个给定的方法做什么，而不是如何做？是的，它确实（违背了）！这是继承违反封装这一事实

的不幸结果。要为一个类编制文档，使其能够安全地子类化，你必须描述实现细节，否则这些细节应该是未指定的。

@implSpec 标记在 Java 8 中添加，在 Java 9 中大量使用。默认情况下应该启用这个标记，但是在 Java 9 中，Javadoc 实用程序仍然忽略它，除非传递命令行开关 -tag "apiNote: a :API Note:"。

为继承而设计不仅仅是记录自使用的模式。为了允许程序员编写高效的子类而不受不必要的痛苦，类可能必须以明智地选择受保护的方法或（在很少的情况下）受保护的字段的形式为其内部工作提供挂钩。例如，考虑来自 java.util.AbstractList 的 removeRange 方法：

> protected void removeRange(int fromIndex, int toIndex)
>
> Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the list by (toIndex - fromIndex) elements. (If toIndex == fromIndex,this operation has no effect.)

从这个列表中删除所有索引位于 fromIndex（包含索引）和 toIndex（独占索引）之间的元素。将任何后续元素移到左边（减少其索引）。这个调用使用 (toIndex - fromIndex) 元素缩短列表。（如果 toIndex == fromIndex，此操作无效。）

> This method is called by the clear operation on this list and its sublists.Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its sublists.

此方法由此列表及其子列表上的 clear 操作调用。重写此方法以利用列表实现的内部特性，可以显著提高对该列表及其子列表的 clear 操作的性能。

> Implementation Requirements: This implementation gets a list iterator positioned before fromIndex and repeatedly calls ListIterator.next followed by ListIterator.remove, until the entire range has been removed. Note: If ListIterator.remove requires linear time, this implementation requires quadratic time.

实现需求：该实现获取位于 fromIndex 之前的列表迭代器，并依次重复调用 ListIterator.next 和 ListIterator.remove，直到删除整个范围的内容。注意：如果 ListIterator.remove 需要线性时间，这个实现需要平方级的时间。

Parameters:

参数

fromIndex index of first element to be removed.

要删除的第一个元素的 fromIndex 索引。

toIndex index after last element to be removed.

要删除的最后一个元素后的索引。

此方法对列表实现的最终用户没有任何兴趣。它的提供只是为了让子类更容易在子列表上提供快速清晰的方法。在没有 removeRange 方法的情况下，当在子列表上调用 clear 方法或从头重写整个子列表机制时，子类将不得不处理二次性能——这不是一项简单的任务!

那么，在为继承设计类时，如何决定要公开哪些受保护的成员呢？不幸的是，没有灵丹妙药。你能做的最好的事情就是认真思考，做出最好的猜测，然后通过编写子类来测试它。你应该尽可能少地公开受保护的成员，因为每个成员都表示对实现细节的承诺。另一方面，你不能公开太多，因为缺少受保护的成员会导致类实际上无法用于继承。

测试为继承而设计的类的唯一方法是编写子类。 如果你忽略了一个关键的受保护成员，那么尝试编写子类将使遗漏变得非常明显。相反，如果编写了几个子类，而没有一个子类使用受保护的成员，则应该将其设置为私有。经验表明，三个子类通常足以测试一个可扩展类。这些子类中的一个或多个应该由超类作者以外的其他人编写。

当你为继承设计一个可能获得广泛使用的类时，请意识到你将永远致力于你所记录的自使用模式，以及在其受保护的方法和字段中隐含的实现决策。这些承诺会使在后续版本中改进类的性能或功能变得困难或不可能。因此，**你必须在释放类之前通过编写子类来测试类。**

另外，请注意，继承所需的特殊文档会使普通文档变得混乱，这种文档是为那些创建类实例并在其上调用方法的程序员设计的。在撰写本文时，很少有工具能够将普通 API 文档与只对实现子类的程序员感兴趣的信息分离开来。

为了允许继承，类必须遵守更多的限制。**构造函数不能直接或间接调用可重写的方法。** 如果你违反了这个规则，程序就会失败。超类构造函数在子类构造函数之前运行，因此在子类构造函数运行之前将调用子类中的覆盖方法。如

果重写方法依赖于子类构造函数执行的任何初始化，则该方法的行为将不像预期的那样。为了使其具体化，下面是一个违反此规则的类：

```
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }
    public void overrideMe() {
    }
}
```

下面是覆盖 overrideMe 方法的子类，Super 的唯一构造函数错误地调用了 overrideMe 方法：

```
public final class Sub extends Super {
    // Blank final, set by constructor
    private final Instant instant;
    Sub() {
        instant = Instant.now();
    }
    // Overriding method invoked by superclass constructor
    @Override
    public void overrideMe() {
        System.out.println(instant);
    }
    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
```

```
        }

    }
```

你可能希望这个程序打印两次 instant，但是它第一次打印 null，因为在子构造函数有机会初始化 instant 字段之前，超级构造函数调用了 overrideMe。注意，这个程序观察了两个不同状态的最后一个字段!还要注意，如果 overrideMe 立即调用了任何方法，那么当超级构造函数调用 overrideMe 时，它会抛出一个 NullPointerException。这个程序不抛出 NullPointerException 的唯一原因是 println 方法允许空参数。

注意，从构造函数调用私有方法、最终方法和静态方法是安全的，它们都是不可覆盖的。

可克隆和可序列化的接口在设计继承时存在特殊的困难。对于为继承而设计的类来说，实现这两种接口都不是一个好主意，因为它们给扩展类的程序员带来了沉重的负担。但是，你可以采取一些特殊的操作来允许子类实现这些接口，而无需强制它们这样做。Item-13 和 Item-86 叙述了这些行动。

如果你确实决定在为继承而设计的类中实现 Cloneable 或 Serializable，那么你应该知道，由于克隆和 readObject 方法的行为与构造函数非常相似，因此存在类似的限制：克隆和 readObject 都不能直接或间接调用可覆盖的方法。对于 readObject，重写方法将在子类的状态反序列化之前运行。在克隆的情况下，重写方法将在子类的克隆方法有机会修复克隆的状态之前运行。在任何一种情况下，程序失败都可能随之而来。在克隆的情况下，失败可以破坏原始对象和克隆。例如，如果覆盖方法假设它正在修改对象的深层结构的克隆副本，但是复制还没有完成，那么就会发生这种情况。

最后，如果你决定在一个为继承而设计的类中实现 Serializable，并且这个类有一个 readResolve 或 writeReplace 方法，那么你必须使 readResolve 或 writeReplace 方法为 protected，而不是 private。如果这些方法是 private 的，它们将被子类静静地忽略。这是实现细节成为类 API 允许继承的一部分的又一种情况。

到目前为止，显然为继承而设计一个类需要付出很大的努力，并且对类有很大的限制。这不是一个可以轻易作出的决定。在某些情况下，这样做显然是正确的，例如抽象类，包括接口的骨架实现（Item-20）。还有一些情况显然是错误的，比如不可变类（Item-17）。

但是普通的具体类呢？传统上，它们既不是最终的，也不是为子类化而设计和记录的，但这种状态是危险的。每当在这样的类中进行更改时，扩展类的

子类就有可能中断。这不仅仅是一个理论问题。在修改未为继承而设计和记录的非最终具体类的内部结构后，接收与子类相关的 bug 报告并不罕见。

这个问题的最佳解决方案是禁止在没有设计和文档记录的类中进行子类化。有两种方法可以禁止子类化。两者中比较容易的是声明类 final。另一种方法是将所有构造函数变为私有或包私有，并在构造函数的位置添加公共静态工厂。这个替代方案提供了内部使用子类的灵活性，在 Item-17 中进行了讨论。两种方法都可以接受。

这个建议可能有点争议，因为许多程序员已经习惯了子类化普通的具体类，以添加工具、通知和同步等功能或限制功能。如果一个类实现了某个接口，该接口捕获了它的本质，例如 Set、List 或 Map，那么你不应该对禁止子类化感到内疚。在 Item-18 中描述的包装器类模式提供了一种优于继承的方法来增强功能。

如果一个具体的类没有实现一个标准的接口，那么你可能会因为禁止继承而给一些程序员带来不便。如果你认为必须允许继承此类类，那么一种合理的方法是确保该类永远不会调用其任何可重写的方法，并记录这一事实。换句话说，消除类的自用 overridable

你可以在不改变类行为的情况下，机械地消除类对可重写方法的自使用。将每个可覆盖方法的主体移动到一个私有的「助手方法」，并让每个可覆盖方法调用它的私有助手方法。然后，用可覆盖方法的私有助手方法的直接调用替换可覆盖方法的每个自使用。

总之，为继承设计一个类是一项艰苦的工作。你必须记录所有的自用模式，并且一旦你记录了它们，你就必须在整个类的生命周期中都遵守它们。如果没有这样做，子类可能会依赖于超类的实现细节，如果超类的实现发生变化，子类可能会崩溃。为了允许其他人编写高效的子类，你可能还需要导出一个或多个受保护的方法。除非你知道确实需要子类，否则最好通过声明类为 final 或确保没有可访问的构造函数的方式来禁止继承。

20 接口优于抽象类

Java 有两种机制来定义允许多种实现的类型：接口和抽象类。由于 Java 8 [JLS 9.4.3]中引入了接口的默认方法，这两种机制都允许你为一些实例方法提供实现。一个主要区别是，一个类要实现抽象类定义的类型，该类必须是抽象类的子类。因为 Java 只允许单一继承，所以这种对抽象类的限制严重制约了它们作为类型定义的使用。任何定义了所有必需的方法并遵守通用约定的类都允许实现接口，而不管该类驻留在类层次结构中何处。

译注：第一段可拆分出有关抽象类和接口的描述

1、抽象类的局限：一个类要实现抽象类定义的类型，该类必须是抽象类的子类。因为 Java 只允许单一继承，所以这种对抽象类的限制严重制约了它们作为类型定义的使用。

2、接口的优点：任何定义了所有必需的方法并遵守通用约定的类都允许实现接口，而不管该类驻留在类层次结构中何处。

可以很容易地对现有类进行改造，以实现新的接口。 你所要做的就是添加所需的方法（如果它们还不存在的话），并向类声明中添加一个 implements 子句。例如，许多现有的类在添加到平台时进行了修改，以实现 Comparable、Iterable 和 Autocloseable 接口。一般来说，现有的类不能被修改以扩展新的抽象类。如果你想让两个类扩展同一个抽象类，你必须把它放在类型层次结构的高层，作为两个类的祖先。不幸的是，这可能会对类型层次结构造成巨大的附带损害，迫使新抽象类的所有后代对其进行子类化，无论它是否合适。

接口是定义 mixin（混合类型）的理想工具。 粗略地说，mixin 是类除了「基本类型」之外还可以实现的类型，用于声明它提供了一些可选的行为。例如，Comparable 是一个 mixin 接口，它允许类表明它的实例可以与其他的可相互比较的对象进行排序。这样的接口称为 mixin，因为它允许可选功能「混合」到类型的主要功能中。抽象类不能用于定义 mixin，原因与它们不能被修改到现有类相同：一个类不能有多个父类，而且在类层次结构中没有插入 mixin 的合理位置。

接口允许构造非层次化类型框架。 类型层次结构对于组织一些事情很好，但是其他事情不能整齐地归入严格的层次结构。例如，假设我们有一个代表歌手的接口和另一个代表词曲作者的接口：

```
public interface Singer {
AudioClip sing(Song s);
}

public interface Songwriter {
Song compose(int chartPosition);
}
```

在现实生活中，一些歌手也是词曲作者。因为我们使用接口而不是抽象类来定义这些类型，所以完全允许单个类同时实现歌手和词曲作者。事实上，我

们可以定义第三个接口，扩展歌手和词曲作者，并添加适合这种组合的新方法：

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

你并不总是需要这种级别的灵活性，但是当你需要时，接口就是救星。另一种选择是一个臃肿的类层次结构，它为每个受支持的属性组合包含一个单独的类。如果类型系统中有 n 个属性，那么可能需要支持 2n 种组合。这就是所谓的组合爆炸。臃肿的类层次结构可能导致类也臃肿，其中许多方法只在其参数的类型上有所不同，因为类层次结构中没有类型来捕获常见行为。

通过 Item-18 介绍的包装类，接口能够支持安全、强大的功能增强。如果你使用抽象类来定义类型，那么你将让希望添加功能的程序员除了继承之外别无选择。最终生成的类不如包装类强大，也更脆弱。

当接口方法的其他接口方法有明显的实现时，考虑以默认方法的形式为程序员提供实现帮助。有关此技术的示例，请参阅第 104 页的 removeIf 方法。如果你提供了默认方法，请使用 @implSpec 标签，并确保在文档中记录他们的继承关系（Item-19）。

默认方法可以为实现提供的帮助有限。尽管许多接口指定了诸如 equals 和 hashCode 等对象方法的行为，但是不允许为它们提供默认方法。此外，接口不允许包含实例字段或非公共静态成员（私有静态方法除外）。最后，你不能向你不控制的接口添加默认方法。

但是，你可以通过提供一个抽象骨架实现类来结合接口和抽象类的优点。接口定义了类型，可能提供了一些默认方法，而骨架实现类在基本接口方法之上实现了其余的非基本接口方法。扩展骨架实现需要完成实现接口的大部分工作。这是模板方法模式 [Gamma95]。

按照惯例，骨架实现类称为 AbstractInterface，其中 Interface 是它们实现的接口的名称。例如，Collections Framework 提供了一个骨架实现来配合每个主要的集合接口：AbstractCollection、AbstractSet、AbstractList 和 AbstractMap。可以说，将它们称为 SkeletalCollection、SkeletalSet、SkeletalList 和 SkeletalMap 是有意义的，但 Abstract 的用法现在已经根深蒂固。如果设计得当，骨架实现（无论是单独的抽象类，还是仅仅由接口上的默认方法组成）可以使程序员非常容易地提供他们自己的接口实现。例如，这里有一个静态工厂方法，它在 AbstractList 上包含一个完整的、功能完整的 List 实现：

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(int[] a) {
        Objects.requireNonNull(a);
        // The diamond operator is only legal here in Java 9 and later
        // If you're using an earlier release, specify <Integer>
        return new AbstractList<>() {
            @Override
            public Integer get(int i) {
                return a[i]; // Autoboxing (Item 6)
            }

            @Override
            public Integer set(int i, Integer val) {
                int oldVal = a[i];
                a[i] = val; // Auto-unboxing
                return oldVal; // Autoboxing
            }

            @Override
            public int size() {
                return a.length;
            }
        };
}
```

当你考虑到 List 实现为你做的所有事情时，这个例子是骨架实现强大功能的一个令人印象深刻的演示。顺便说一句，这个示例是一个 Adapter（适配器）[Gamma95]，它允许将 int 数组视为 Integer 实例的 list。因为在 int 值和

Integer 实例（装箱和拆箱）之间来回转换，所以它的性能不是很好。注意，实现的形式是匿名类（Item-24）。

骨架实现类的美妙之处在于，它们提供了抽象类的所有实现帮助，而不像抽象类作为类型定义时那样受到严格的约束。对于具有骨架实现类的接口的大多数实现来说，扩展这个类是显而易见的选择，但它并不是必需的。如果不能使类扩展骨架实现，则类总是可以直接实现接口。类仍然受益于接口本身的任何默认方法。此外，骨架实现仍然可以帮助实现人员完成任务。实现接口的类可以将接口方法的调用转发给扩展骨架实现的私有内部类的包含实例。这种技术称为模拟多重继承，与 Item-18 中讨论的包装类密切相关。它提供了多重继承的许多好处，同时避免了缺陷。

编写一个骨架实现是一个相对简单的过程，尽管有点乏味。首先，研究接口并决定哪些方法是基本方法，以便其他方法可以根据它们实现。这些基本方法将是你的骨架实现中的抽象方法。接下来，在接口中为所有可以直接在原语之上实现的方法提供默认方法，但请记住，你可能不会为诸如 equals 和 hashCode 之类的对象方法提供默认方法。如果原语和默认方法覆盖了接口，那么就完成了，不需要一个骨架实现类。否则，编写一个声明为实现接口的类，并实现所有剩余的接口方法。该类可能包含任何适合于任务的非公共字段和方法。

作为一个简单的例子，考虑一下 Map.Entry 接口。最明显的基本方法是 getKey、getValue 和（可选的）setValue。该接口指定了 equals 和 hashCode 的行为，并且在基本方法方面有 toString 的明显实现。由于不允许为对象方法提供默认实现，所有实现都放在骨架实现类中：

```
// Skeletal implementation class
public abstract class AbstractMapEntry<K,V> implements Map.Entry<K,V> {

    // Entries in a modifiable map must override this method
    @Override public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of Map.Entry.equals
    @Override public boolean equals(Object o) {
```

```
            if (o == this)
                return true;
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry<?,?> e = (Map.Entry) o;
            return Objects.equals(e.getKey(), getKey()) &&
Objects.equals(e.getValue(), getValue());
        }

        // Implements the general contract of Map.Entry.hashCode
        @Override public int hashCode() {
            return Objects.hashCode(getKey())^ Objects.hashCode(getValue());
        }

        @Override public String toString() {
            return getKey() + "=" + getValue();
        }
    }
```

注意，这个骨架实现不能在 Map.Entry 接口或子接口中实现，因为不允许默认方法覆盖诸如 equals、hashCode 和 toString 等对象方法。

因为骨架实现是为继承而设计的，所以你应该遵循 Item-19 中的所有设计和文档指南。为了简洁起见，在前面的示例中省略了文档注释，但是优秀的文档对于骨架实现来说是绝对必要的，不管它是由接口上的默认方法还是单独的抽象类组成。

骨架实现的一个小变体是简单实现，例如 AbstractMap.SimpleEntry。一个简单的实现就像一个骨架实现，因为它实现了一个接口，并且是为继承而设计的，但是它的不同之处在于它不是抽象的：它是最简单的工作实现。你可以根据它的状态使用它，也可以根据情况对它进行子类化。

总之，接口通常是定义允许多种实现的类型的最佳方法。如果导出了一个

重要的接口，则应该强烈考虑提供一个骨架实现。尽可能地，你应该通过接口上的默认方法提供骨架实现，以便接口的所有实现者都可以使用它。也就是说，对接口的限制通常要求框架实现采用抽象类的形式。

21 为后代设计接口 @

在 Java 8 之前，在不破坏现有实现的情况下向接口添加方法是不可能的。如果在接口中添加新方法，现有的实现通常会缺少该方法，从而导致编译时错误。在 Java 8 中，添加了默认的方法构造 [JLS 9.4]，目的是允许向现有接口添加方法。但是向现有接口添加新方法充满了风险。

默认方法的声明包括一个默认实现，所有实现接口但不实现默认方法的类都使用这个默认实现。虽然向 Java 添加默认方法使向现有接口添加方法成为可能，但不能保证这些方法在所有现有实现中都能工作。默认方法被「注入」到现有的实现中，而无需实现者的知情或同意。在 Java 8 之前，编写这些实现时都默认它们的接口永远不会获得任何新方法。

Java 8 的核心集合接口增加了许多新的默认方法，主要是为了方便 lambdas 的使用（第 6 章）。**但是，并不总是能够编写一个默认方法来维护每个可想到的实现的所有不变量**

例如，考虑 removeIf 方法，它被添加到 Java 8 中的集合接口中。该方法删除了给定的布尔函数（或 predicate）返回 true 的所有元素。指定默认实现，以使用迭代器遍历集合，在每个元素上调用 predicate，并使用迭代器的 remove 方法删除谓词返回 true 的元素。大概声明是这样的：

```
// Default method added to the Collection interface in Java 8
default boolean removeif(predicate<? super e> filter) {
    objects.requirenonnull(filter);
    boolean result = false;
    for (iterator<e> it = iterator(); it.hasnext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
```

```
            }
        }
        return result;
    }
```

这是为 removeIf 方法编写的最好的通用实现，但遗憾的是，它在一些实际的集合实现中失败了。例如，考虑 org.apache.commons.collections4.collection.SynchronizedCollection。这个类来自 Apache Commons 库，类似于静态工厂集合返回的类。-synchronizedCollection java.util。Apache 版本还提供了使用客户端提供的对象进行锁定的功能，以代替集合。换句话说，它是一个包装器类（Item-18），其所有方法在委托给包装集合之前同步锁定对象。

Apache SynchronizedCollection 类仍然得到了积极的维护，但是在编写本文时，它没有覆盖 removeIf 方法。如果这个类与 Java 8 一起使用，那么它将继承 removeIf 的默认实现，而 removeIf 并不能维护类的基本承诺：自动同步每个方法调用。默认实现对同步一无所知，也无法访问包含锁定对象的字段。如果客户端在 SynchronizedCollection 实例上调用 removeIf 方法，而另一个线程同时修改了集合，那么可能会导致 ConcurrentModificationException 或其他未指定的行为。

为了防止类似的 Java 库实现（例如 Collections.synchronizedCollection 返回的包私有类）中发生这种情况，JDK 维护人员必须覆盖默认的 removeIf 实现和其他类似的方法，以便在调用默认实现之前执行必要的同步。不属于 Java 平台的现有集合实现没有机会与接口更改同步进行类似的更改，有些实现还没有这样做。

在有默认方法的情况下，接口的现有实现可以在没有错误或警告的情况下编译，但是在运行时失败。虽然这个问题并不常见，但也不是孤立的事件。已知 Java 8 中添加到集合接口的少数方法是易受影响的，已知会影响到现有的少数实现。

除非必要，否则应该避免使用默认方法向现有接口添加新方法，在这种情况下，你应该仔细考虑现有接口实现是否可能被默认方法破坏。然而，在创建接口时，默认方法对于提供标准方法实现非常有用，以减轻实现接口的任务（Item-20）。

同样值得注意的是，默认方法的设计并不支持从接口中删除方法或更改现有方法的签名。在不破坏现有客户端的情况下，这些更改都是不可能的。

这个教训很清楚。尽管默认方法现在已经是 Java 平台的一部分，但是谨慎地设计接口仍然是非常重要的。**虽然默认方法使向现有接口添加方法成为可能，但这样做存在很大风险。** 如果一个接口包含一个小缺陷，它可能会永远激怒它的使用者；如果接口有严重缺陷，它可能会毁掉包含它的 API。

因此，在发布每个新接口之前对其进行测试非常重要。多个程序员应该以不同的方式实现每个接口。至少，你应该以三种不同的实现为目标。同样重要的是编写多个客户端程序，这些程序使用每个新接口的实例来执行各种任务。这将大大有助于确保每个接口满足其所有预期用途。这些步骤将允许你在接口被发布之前发现它们的缺陷，而你仍然可以轻松地纠正它们。**虽然在接口被发布之后可以纠正一些接口缺陷，但是你不能指望这种方式。**

22 接口只用于定义类型

当一个类实现了一个接口时，这个接口作为一种类型，可以用来引用类的实例。因此，实现接口的类应该说明客户端可以对类的实例做什么。为任何其他目的定义接口都是不合适的。

不满足上述条件的一种接口是所谓的常量接口。这样的接口不包含任何方法；它仅由静态 final 字段组成，每个字段导出一个常量。使用这些常量的类实现接口，以避免用类名限定常量名。下面是一个例子：

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;
    // Mass of the electron (kg)
    static final double ELECTRON_MASS = 9.109_383_56e-31;
}
```

常量接口模式是对接口的糟糕使用。类内部使用一些常量是实现细节。实现常量接口会导致这个实现细节泄漏到类的导出 API 中。对于类的用户来说，

类实现一个常量接口并没有什么价值。事实上，这甚至会让他们感到困惑。更糟糕的是，它代表了一种承诺：如果在将来的版本中修改了类，使其不再需要使用常量，那么它仍然必须实现接口以确保二进制兼容性。如果一个非 final 类实现了一个常量接口，那么它的所有子类的名称空间都会被接口中的常量所污染。

Java 库中有几个常量接口，例如 java.io.ObjectStreamConstants。这些接口应该被视为反例，不应该被效仿。

如果你想导出常量，有几个合理的选择。如果这些常量与现有的类或接口紧密绑定，则应该将它们添加到类或接口。例如，所有装箱的数值包装类，比如 Integer 和 Double，都导出 MIN_VALUE 和 MAX_VALUE 常量。如果最好将这些常量看作枚举类型的成员，那么应该使用 enum 类型导出它们（Item-34）。否则，你应该使用不可实例化的工具类（Item-4）导出常量。下面是一个之前的 PhysicalConstants 例子的工具类另一个版本：

```
// Constant utility class

package com.effectivejava.science;

public class PhysicalConstants {

    private PhysicalConstants() { } // Prevents instantiation（将构造私有，阻止实例化）

    public static final double AVOGADROS_NUMBER = 6.022_140_857e23;

    public static final double BOLTZMANN_CONST = 1.380_648_52e-23;

    public static final double ELECTRON_MASS = 9.109_383_56e-31;

}
```

顺便说一下，请注意在数字字面值中使用了下划线（ _ ）。下划线自 Java 7 以来一直是合法的，它对数字字面值没有影响，如果谨慎使用，可以使它们更容易阅读。考虑添加下划线到数字字面值，无论是固定的浮点数，如果它们包含五个或多个连续数字。对于以 10 为基数的字面值，无论是整数还是浮点数，都应该使用下划线将字面值分隔为三位数，表示 1000 的正幂和负幂。

通常，工具类要求客户端使用类名来限定常量名，例如 PhysicalConstants.AVOGADROS_NUMBER。如果你大量使用工具类导出的常量，你可以通过使用静态导入机制来避免使用类名限定常量：

```
// Use of static import to avoid qualifying constants
```

```
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    } ...
    // Many more uses of PhysicalConstants justify static import
}
```

总之，接口应该只用于定义类型。它们不应该用于导出常量。

23 类继承优于标签类

有时候，你可能会遇到这样一个类，它的实例有两种或两种以上的样式，并且包含一个标签字段，指示实例的样式。例如，考虑这个类，它能够表示一个圆或一个矩形：

```
// Tagged class - vastly inferior to a class hierarchy!
class Figure {
    enum Shape {RECTANGLE, CIRCLE};
    // Tag field - the shape of this figure
    final Shape shape;
    // These fields are used only if shape is RECTANGLE
    double length;
    double width;
    // This field is used only if shape is CIRCLE
    double radius;
    // Constructor for circle
    Figure(double radius) {
```

```
            shape = Shape.CIRCLE;
            this.radius = radius;
        }
        // Constructor for rectangle
        Figure(double length, double width) {
            shape = Shape.RECTANGLE;
            this.length = length;
            this.width = width;
        }
        double area() {
            switch (shape) {
                case RECTANGLE:
                    return length * width;
                case CIRCLE:
                    return Math.PI * (radius * radius);
                default:
                    throw new AssertionError(shape);
            }
        }
    }
```

这样的标签类有许多缺点。它们充斥着样板文件，包括 enum 声明、标签字段和 switch 语句。可读性会进一步受损，因为多个实现在一个类中混杂在一起。内存占用增加了，因为实例被其他类型的不相关字段所累。除非构造函数初始化不相关的字段，导致更多的样板文件，否则字段不能成为 final。构造函数必须设置标签字段并初始化正确的数据字段，而不需要编译器的帮助：如果初始化了错误的字段，程序将在运行时失败。除非你能够修改它的源文件，否则你不能向标签类添加样式。如果你确实添加了一个样式，那么你必须记住为

每个 switch 语句添加一个 case，否则类将在运行时失败。最后，实例的数据类型没有给出它任何关于风格的线索。简而言之，**标签类冗长、容易出错和低效。**

幸运的是，面向对象的语言（如 Java）提供了一个更好的选择来定义能够表示多种类型对象的单一数据类型：子类型。**标签的类只是类层次结构的（简单）的模仿。**

要将已标签的类转换为类层次结构，首先为标签类中的每个方法定义一个包含抽象方法的抽象类，其行为依赖于标签值。在 Figure 类中，只有一个这样的方法，即 area 方法。这个抽象类是类层次结构的根。如果有任何方法的行为不依赖于标签的值，请将它们放在这个类中。类似地，如果有任何数据字段被所有种类使用，将它们放在这个类中。在 Figure 类中没有这样的独立于味道的方法或字段。

接下来，为原始标签类的每个类型定义根类的具体子类。在我们的例子中，有两个：圆形和矩形。在每个子类中包含特定于其风格的数据字段。在我们的例子中，半径是特定于圆的，长度和宽度是特定于矩形的。还应在每个子类中包含根类中每个抽象方法的适当实现。下面是原 Figure 类对应的类层次结构：

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}
class Circle extends Figure {
    final double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    @Override
    double area() {
        return Math.PI * (radius * radius);
    }
}
```

```
class Rectangle extends Figure {
        final double length;
        final double width;
        Rectangle(double length, double width) {
                this.length = length;
                this.width = width;
        }
        @Override
        double area() {
                return length * width;
        }
}
```

这个类层次结构纠正了前面提到的标签类的所有缺点。代码简单明了，不包含原始代码中的样板文件。每种风格的实现都分配了自己的类，这些类中没有一个被不相关的数据字段拖累。所有字段为 final 字段。编译器确保每个类的构造函数初始化它的数据字段，并且每个类对于根类中声明的每个抽象方法都有一个实现。这消除了由于缺少开关情况而导致运行时失败的可能性。多个程序员可以独立地和互操作地扩展层次结构，而不需要访问根类的源。每种风格都有一个单独的数据类型，允许程序员指示变量的风格，并将变量和输入参数限制为特定的风格。

类层次结构的另一个优点是，可以使它们反映类型之间的自然层次关系，从而提高灵活性和更好的编译时类型检查。假设原始示例中的标签类也允许使用正方形。类层次结构可以反映这样一个事实，即正方形是一种特殊的矩形（假设两者都是不可变的）：

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
```

```
}
```

注意，上面层次结构中的字段是直接访问的，而不是通过访问器方法访问的。这样做是为了简洁，如果层次结构是公共的，那么这将是一个糟糕的设计（Item-16）。

总之，标签类很少有合适的时候。如果你想用显式标签字段编写类，请考虑是否可以消除标签并用层次结构替换类。当你遇到带有标签字段的现有类时，请考虑将其重构为层次结构。

24 优先考虑静态类而不是非静态

嵌套类是在另一个类中定义的类。嵌套类应该只为外部类服务。如果嵌套类在其他环境中有用，那么它应该是顶级类。有四种嵌套类：静态成员类、非静态成员类、匿名类和局部类。除了第一种，所有的类都被称为内部类。本条目会告诉你什么时候使用哪种嵌套类以及原因。

静态成员类是最简单的嵌套类。最好把它看做是一个普通的类，只是碰巧在另一个类中声明而已，并且可以访问外部类的所有成员，甚至那些声明为 private 的成员。静态成员类是其外部类的静态成员，并且遵守与其他静态成员相同的可访问性规则。如果声明为私有，则只能在外部类中访问，等等。

静态成员类的一个常见用法是作为公有的辅助类，只有与它的外部类一起使用时才有意义。例如，考虑一个描述了计算器支持的各种操作的枚举（Item-34）。Operation 枚举应该是 Calculator 类的公有静态成员类，Calculator 类的客户端就可以
用 Calculator.Operation.PLUS 和 Calculator.Operation.MINUS 等名称来引用这些操作。

从语法上讲，静态成员类和非静态成员类之间的唯一区别是静态成员类在其声明中具有修饰符 static。尽管语法相似，但这两种嵌套类有很大不同。非静态成员类的每个实例都隐式地与外部类的外部实例相关联。在非静态成员类的实例方法中，你可以调用外部实例上的方法，或者使用受限制的 this 构造获得对外部实例的引用 [JLS, 15.8.4]。如果嵌套类的实例可以独立于外部类的实例存在，那么嵌套类必须是静态成员类：如果没有外部实例，就不可能创建非静态成员类的实例。

非静态成员类实例与外部实例之间的关联是在创建成员类实例时建立的，之后无法修改。通常，关联是通过从外部类的实例方法中调用非静态成员类构

造函数自动建立的。使用 enclosingInstance.new MemberClass(args) 表达式手动建立关联是可能的，尽管这种情况很少见。正如你所期望的那样，关联占用了非静态成员类实例中的空间，并为其构造增加了时间。

非静态成员类的一个常见用法是定义一个适配器 [Gamma95]，它允许外部类的实例被视为某个不相关类的实例。例如，Map 接口的实现通常使用非静态成员类来实现它们的集合视图，这些视图由 Map 的 keySet、entrySet 和 values 方法返回。类似地，集合接口的实现，例如 Set 和 List，通常使用非静态成员类来实现它们的迭代器：

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
    ... // Bulk of the class omitted
    @Override
    public Iterator<E> iterator() {
        return new MyIterator();
    }
    private class MyIterator implements Iterator<E> {
       ...
    }
}
```

如果声明的成员类不需要访问外部的实例，那么应始终在声明中添加 static 修饰符，使其成为静态的而不是非静态的成员类。如果省略这个修饰符，每个实例都有一个隐藏的对其外部实例的额外引用。如前所述，存储此引用需要时间和空间。更严重的是，它可能会在满足进行垃圾收集条件时仍保留外部类的实例（Item-7）。由于引用是不可见的，因此通常很难检测到。

私有静态成员类的一个常见用法是表示由其外部类表示的对象的组件。例如，考虑一个 Map 实例，它将 key 与 value 关联起来。许多 Map 实现的内部对于映射中的每个 key-value 对都有一个 Entry 对象。虽然每个 entry 都与 Map 关联，但 entry 上的方法（getKey、getValue 和 setValue）不需要访问 Map。因此，使用非静态成员类来表示 entry 是浪费：私有静态成员类是最好的。如

果你不小心在 entry 声明中省略了静态修饰符，那么映射仍然可以工作，但是每个 entry 都包含对 Map 的多余引用，这会浪费空间和时间。

如果所讨论的类是导出类的公共成员或受保护成员，那么在静态成员类和非静态成员类之间正确选择就显得尤为重要。在本例中，成员类是导出的 API 元素，在后续版本中，不能在不违反向后兼容性的情况下将非静态成员类更改为静态成员类。

如你所料，匿名类没有名称。它不是外部类的成员。它不是与其他成员一起声明的，而是在使用时同时声明和实例化。匿名类在代码中任何一个表达式合法的地方都是被允许的。当且仅当它们发生在非静态环境中时，匿名类才具有外部类实例。但是，即使它们发生在静态环境中，它们也不能有除常量变量以外的任何静态成员，常量变量是最终的基本类型或初始化为常量表达式的字符串字段 [JLS, 4.12.4]。

匿名类的适用性有很多限制。你不能实例化它们，除非在声明它们的时候。你不能执行 instanceof 测试，也不能执行任何其他需要命名类的操作。你不能声明一个匿名类来实现多个接口或扩展一个类并同时实现一个接口。匿名类的客户端除了从超类型继承的成员外，不能调用任何成员。因为匿名类发生在表达式的中间，所以它们必须保持简短——大约 10 行或几行，否则可读性会受到影响。

在 lambdas 被添加到 Java（Chapter 6）之前，匿名类是动态创建小函数对象和进程对象的首选方法，但 lambdas 现在是首选方法（Item-42）。匿名类的另一个常见用法是实现静态工厂方法（参见 Item-20 中的 intArrayAsList 类）。

局部类是四种嵌套类中最不常用的。局部类几乎可以在任何能够声明局部变量的地方使用，并且遵守相同的作用域规则。局部类具有与其他嵌套类相同的属性。与成员类一样，它们有名称，可以重复使用。与匿名类一样，它们只有在非静态环境中定义的情况下才具有外部类实例，而且它们不能包含静态成员。和匿名类一样，它们应该保持简短，以免损害可读性。

简单回顾一下，有四种不同类型的嵌套类，每一种都有自己的用途。如果嵌套的类需要在单个方法之外可见，或者太长，不适合放入方法中，则使用成员类。如果成员类的每个实例都需要引用其外部类实例，则使其非静态；否则，让它保持静态。假设嵌套类属于方法内部，如果你只需要从一个位置创建实例，并且存在一个能够描述类的现有类型，那么将其设置为匿名类；否则，将其设置为局部类。

25 将源文件限制为单个顶级类@

虽然 Java 编译器允许你在单个源文件中定义多个顶层类，但这样做没有任何好处，而且存在重大风险。这种风险源于这样一个事实，即在源文件中定义多个顶层类使得为一个类提供多个定义成为可能。所使用的定义受源文件传给编译器的顺序的影响。要使这个问题具体化，请考虑这个源文件，它只包含一个主类，该主类引用另外两个顶层类的成员（餐具和甜点）：

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

现在假设你在一个名为 Utensil.java 的源文件中定义了餐具和甜点：

```
// Two classes defined in one file. Don't ever do this!
// 在一个文件中定义两个类。永远不要这样做!
class Utensil {
    static final String NAME = "pan";
}
class Dessert {
    static final String NAME = "cake";
}
```

当然，main 方法应该输出 pancake。现在假设你意外地制作了另一个名为 Dessert 的源文件。java 定义了相同的两个类：

```
// Two classes defined in one file. Don't ever do this!
// 在一个文件中定义两个类。永远不要这样做!
class Utensil {
    static final String NAME = "pot";
```

```
}

class Dessert {

    static final String NAME = "pie";

}
```

如果你足够幸运，使用 javac Main.java Dessert.java 命令编译程序时，编译将失败，编译器将告诉你多重定义了餐具和甜点。这是因为编译器将首先编译 Main.java，当它看到对 Utensil 的引用（在对 Dessert 的引用之前）时，它将在 Utensil.java 中查找这个类，并找到餐具和甜点。当编译器在命令行上遇到 Dessert.java 时，（编译器）也会载入该文件，导致（编译器）同时遇到 Utensil 和 Dessert 的定义。

如果你使用命令 javac Main.java 或 javac Main.java Utensil.java 编译程序，它的行为将与编写 Dessert.java 文件（打印 pancake）之前一样。但是如果你使用命令 javac Dessert.java Main.java 编译程序，它将打印 potpie。因此，程序的行为受到源文件传递给编译器的顺序的影响，这显然是不可接受的。

修复这个问题非常简单，只需将顶层类（在我们的示例中是餐具和甜点）分割为单独的源文件即可。如果你想将多个顶层类放到一个源文件中，请考虑使用静态成员类（Item-24）作为将类分割为单独的源文件的替代方法。如果（多个顶层类）隶属于另一个类，那么将它们转换成静态成员类通常是更好的选择，因为它增强了可读性，并通过声明它们为私有（Item-15），从而降低了类的可访问性。下面是我们的静态成员类示例的样子：

```
// Static member classes instead of multiple top-level classes

public class Test {

    public static void main(String[] args) {

        System.out.println(Utensil.NAME + Dessert.NAME);

    }

    private static class Utensil {

        static final String NAME = "pan";

    }

    private static class Dessert {
```

```
        static final String NAME = "cake";
    }
}
```

教训很清楚：永远不要将多个顶层类或接口放在一个源文件中。遵循此规则可以确保在编译时单个类不能拥有多个定义。这反过来保证了编译所生成的类文件，以及程序的行为，是独立于源代码文件传递给编译器的顺序的。

第五章 泛型

自 Java 5 以来，泛型一直是 Java 语言的一部分。在泛型出现之前，从集合中读取的每个对象都必须进行强制转换。如果有人不小心插入了错误类型的对象，强制类型转换可能在运行时失败。对于泛型，你可以告知编译器在每个集合中允许哪些类型的对象。编译器会自动为你进行强制转换与插入的操作，如果你试图插入类型错误的对象，编译器会在编译时告诉你。这就产生了更安全、更清晰的程序，但是这些好处不仅仅局限于集合，而且也是有代价的。这一章会告诉你如何最大限度地扬长避短。

26 不要使用原生类型

首先，来介绍几个术语。泛型类或接口是指，声明里有一个或多个类型参数的类或接口[JLS, 8.1.2, 9.1.2]。例如，List 接口就有一个类型参数，E，它表示了 List 的元素类型。接口的全名是 List<E>（读作“E 的列表”），但人们通常简称它为列表。泛型类和接口都被称为泛型类型。

每个泛型类型都定义了一组参数化的类型，它的组成形式为，类名或者接口名后面跟着由尖括号包围的类型参数列表，列表里的每个参数都对应着泛型类型的形式类型参数[JLS, 4.4, 4.5]。例如，List<String>（读作字符串列表）就是一个参数化的类型，代表了元素是 String 类型的列表。（String 是形式类型参数 E 的实际类型参数。）

最后，每个泛型类型定义了一个原始类型，即不带任何类型参数的泛型类型的名称。例如，List<E>对应的原始类型是 List。原始类型看起来就像将所有的泛型类型信息从类型声明了去除了一样。它们的存在主要是为了兼容那些之前在泛型未出现时写的代码。

在泛型被添加进 Java 之前，下面的例子是一个标准的集合声明。对于 Java 9，这么声明仍然是合法的，但就并不是典型的声明了：

```
// Raw collection type - don't do this!
```

```
// My stamp collection. Contains only Stamp instances.

private final Collection stamps = ... ;
```

如果到今天你还是用这种声明然后不小心往 Stamp 集合里放入了一个 Coin 对象，这种错误插入仍然可以编译而且运行也不会出错（虽然编译器会发出一个不明确的警告）：

```
// Erroneous insertion of coin into stamp collection

stamps.add(new Coin( ... )); // Emits "unchecked call" warning
```

在你尝试从这个 Stamp 集合里获取 Coin 对象之前，你都不会遇到程序错误：

```
// Raw iterator type - don't do this!

for (Iterator i = stamps.iterator(); i.hasNext(); )  {

    Stamp stamp = (Stamp) i.next(); // Throws ClassCastException

    stamp.cancel();

}
```

正如本书提到的，出现错误后最好尽快发现，理想情况是在编译时就发现。在上面例子的情况下，直到运行时你才能发现问题，那时问题已经出现很久了，而且真正有问题的代码可能离直接出错的地方很远。在你看到 ClassCastException 后，你不得不搜索整个代码库，找出将 Coin 对象放入 Stamp 集合的方法调用。编译器无法提供帮助，因为它无法理解那句注释：只能包含 Stamp 实例。

而用了泛型后，类型声明里就包含了元素信息，而不仅仅是用注解来说明：

```
// Parameterized collection type - typesafe

private final Collection<Stamp> stamps = ... ;
```

这么声明后，编译器知道了 stamps 应该只包含 Stamp 实例并且会保证这一点，假设你的整个代码库编译时不发出任何警告（或抑制，见条目 27）。若声明 stamps 时用了参数化类型声明，则插入不符类型的元素时将会生成编译时错误信息，告诉你哪里错了：

```
Test.java:9: error: incompatible types: Coin cannot be converted to
Stampc.add(new Coin());


^
```

当你从集合里获取元素时，编译器默默帮你做了强转的工作，保证了取出的元素是符合要求的（假设你所有的代码没有生成或抑制任何编译警告）。虽然不大可能将 Coin 对象插入 Stamp 集合里，但这个问题的确是存在的。例如，不难想象出将一个 BigInteger 放入一个本应该只能包含 BigDecimal 的集合。

如前面所说，使用原始类型（不包含类型参数的泛型类）是合法的，但你永远都不应该这么做。**如果你使用了原始类型，你将会失去泛型所带来的安全性和可读性。**既然不应该使用它们，那为什么 Java 语言设计者还要允许使用它们呢？这么做其实是为了兼容性。在泛型加入 Java 时，Java 即将步入它的第二个十年，那会已经存在了大量没有使用泛型的代码。当时人们认为，让这些代码依旧合法存在而且能与使用了泛型的新代码互用这点是很重要的。将参数类型的实例传入之前那些为了使用原始类型而设计的方法，这必须是合法的，反之亦然。这种需求被称为移植兼容性（migration compatibility），它驱使了支持原始类型和使用擦除来实现泛型的决定（条目 28）。

虽然你不应该使用诸如 List 之类的原始类型，但可以使用参数化类型以允许插入任意对象，例如 List<Object>。那么原始类型 List 和参数化类型 List<Object>之间的区别是什么呢？不严格地说，前者不受泛型类型系统检查，而后者则显示地告诉编译器它可以接受任意类型的对象。虽然你可以将 List<String>传给 List 类型的参数，但你无法将 List<String>传给 List<Object>类型的参数。因此，**如果使用诸如 List 之类的原始类型，你将失去类型安全，但如果你使用了一个像 List<Object>那样的参数化类型，则没有这个问题。**为了更具体地说明，考虑下面这个程序：

```
// Fails at runtime - unsafeAdd method uses a raw type (List)!
public static void main(String[] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // Has compiler-generated cast
}
```

```
    private static void unsafeAdd(List list, Object o) {
        list.add(o);
    }
```

这段程序可以编译通过，但由于它使用了原始类型 List，你会收到一个警告：

```
    Test.java:10: warning: [unchecked] unchecked call to add(E) as a member of the raw type List list.add(o);
                                       ^
```

结果就是，如果你运行这段程序，程序会在试图将 string.get(0)的调用结果从 Integer 强转为 String 时抛出一个 ClassCastException 异常。由于这是编译器生成的强转，所以通常会保证成功，但例子中我们忽略了那条编译器警告，因此而付出了代价。如果你在 unsafeAdd 方法的声明中将原始类型 List 替换为参数化类型 List<Object>，并尝试着重新编译这段程序，你将会发现这段程序无法继续编译，而是出现了下面这条错误信息：

```
    Test.java:5: error: incompatible types: List<String> cannot be converted to List<Object> unsafeAdd(strings, Integer.valueOf(42));
                                                                                              ^
```

对于元素类型未知而且不在乎元素类型的集合，也许你会想用原始类型。例如，假设你想写这么一个方法，它接受两个 Set 参数并返回它们公有元素的个数。如果你刚开始用泛型，你可能会像下面那样去写这个方法：

```
    // Use of raw type for unknown element type - don't do this!
    static int numElementsInCommon(Set s1, Set s2) {
        int result = 0;
        for (Object o1 : s1)
            if (s2.contains(o1))
```

```
                result++;

        return result;

    }
```

这个方法可以运行但它使用了原始类型，这是危险的。安全的方式是使用无限制通配符类型（unbounded wildcard types）。如果你想使用泛型类型，但你不知道或者不关心实际的类型尝试是什么，你可以用一个问好来替代。例如，泛型类型 Set<E>的无限制通配符类型是 Set<?>（读作，某些类型的集合）。这是最通用的参数化 Set 类型，可以接收任何集合。下面是 numElementsInCommon 方法，用无限制通配符类型来声明时，看起来是这样的：

```
    // Uses unbounded wildcard type - typesafe and flexible

    static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

那么，无限制通配符类型 Set<?>和原始类型 Set 之间的区别又是什么呢？这个问号起了什么作用吗？起作用这点是毋庸置疑的，通配符是安全的而原始类型则不是。你可以将任意元素放入一个原始类型集合，轻而易举地破坏了集合的类型约束（就如前面描述的 unsafeAdd 方法），当你无法将任意元素（null 除外）放入一个 Collection<?>。试图这么做的化将产生编译时错误，错误信息就像下面这样子：

```
    WildCard.java:13: error: incompatible types: String cannot be converted to
CAP#1 c.add("verboten");


^

    where CAP#1 is a fresh type-variable: CAP#1 extends Object from capture
of ?
```

的确，这条错误信息不是完全我们想要的，但是编译器已经做了它该做的事，即阻止你破坏集合的类型约束，无论它的元素类型是什么。你不仅无法将任意元素（null 除外）放入一个 Collection<?>，而且还无法猜测取出的对象是什么类型的。如果这些限制你无法接受，那么你可以使用泛型方法（条目 30）或者限制通配符类型（bounded wildcard types，条目 31）。

对于不应该使用原始类型这条规则，有两个小例外。你必须在类字面值中使用原始类型。这条规范不允许使用参数化类型（虽然它允许数组类型和基础类型）[JLS, 15.8.2]。换

句话说，List.class，String[].class 以及 int.class 都是合法的，但 List<String>.class 和 List<?>.class 则不是。第二个例外则与 instanceof 操作符有关。因为泛型类型信息在运行时是被擦除了的，所以在参数化类型而不是无限制通配符类型上用 instanceof 操作符是非法的。用无限制通配符类型来替换原始类型并不影响 instanceof 操作符的行为。在这种情况下，尖括号和问号是多余的。在下面的例子中，泛型类型是使用 instanceof 操作符的首选方式：

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) { // Raw type
    Set<?> s = (Set<?>) o; // Wildcard type
    ...
}
```

注意，一旦你确定了 o 是一个 Set 对象，你必须将它强转为通配符类型 Set<?>，而不是原始类型 Set。这是一个检查的强转，所以它不会引起编译器警告。

总而言之，使用原始类型会导致运行时异常，所以不要使用它们。它们被提供仅是为了兼容性以及能与引入泛型之前的遗留代码互用。下面我们来快速回顾一下，Set<Object>是一个参数化的类型，表示一个可以包含任意类型的集合，Set<?>是一个通配符类型，表示一个只能包含某个未知类型的对象的集合，而 Set 是一个原始类型，不在泛型类型系统之内。前面两种是安全的，而最后一种是不安全的。

为了方便快速参考，本条目介绍的术语（少数几个术语会在本章的后面进行介绍）总结成下面表格的内容：

术语	例子	所在条目
参数化类型（Parameterized type）	List<String>	条目 26
实际类型参数（Actual type parameter）	String	条目 26
泛型类型（Generic type）	List<E>	条目 26

术语	例子	所在条目
形式类型参数（Formal type parameter）	E	条目 26
无限制通配符类型（Unbounded wildcard type）	List<?>	条目 26
原始类型（Raw type）	List	条目 26
有限制类型参数（Bounded type parameter）	<E extends Number>	条目 29
递归类型限制（Recursive type bound）	<T extends Comparable<T>>	条目 30
有限制通配符类型（Bounded wildcard type）	List<? extends Number>	条目 31
泛型方法（Generic method）	static <E> List<E> asList(E[] a)	条目 30
类型令牌（Type token）	String.class	条目 33

27 消除未检查警告

当你使用泛型编程时，你将看到许多编译器警告：unchecked 强制转换警告、unchecked 方法调用警告、unchecked 可变参数类型警告和 unchecked 自动转换警告。使用泛型获得的经验越多，得到的警告就越少，但是不要期望新编写的代码能够完全正确地编译。

许多 unchecked 警告很容易消除。例如，假设你不小心写了这个声明：

```
Set<Lark> exaltation = new HashSet();
```

编译器会精确地提醒你做错了什么：

```
Venery.java:4: warning: [unchecked] unchecked conversion
```

```
Set<Lark> exaltation = new HashSet();

^ required: Set<Lark>

found: HashSet
```

你可以在指定位置进行更正，使警告消失。注意，你实际上不必指定类型参数，只需给出由 Java 7 中引入的 diamond 操作符（<>）。然后编译器将推断出正确的实际类型参数（在本例中为 Lark）：

```
Set<Lark> exaltation = new HashSet<>();
```

一些警告会更难消除。这一章充满这类警告的例子。当你收到需要认真思考的警告时，坚持下去！**力求消除所有 unchecked 警告。**如果你消除了所有警告，你就可以确信你的代码是类型安全的，这是一件非常好的事情。这意味着你在运行时不会得到 ClassCastException，它增加了你的信心，你的程序将按照预期的方式运行。

如果不能消除警告，但是可以证明引发警告的代码是类型安全的，那么（并且只有在那时）使用 SuppressWarnings("unchecked") 注解来抑制警告。如果你在没有首先证明代码是类型安全的情况下禁止警告，那么你是在给自己一种错误的安全感。代码可以在不发出任何警告的情况下编译，但它仍然可以在运行时抛出 ClassCastException。但是，如果你忽略了你知道是安全的 unchecked 警告（而不是抑制它们），那么当出现一个代表真正问题的新警告时，你将不会注意到。新出现的警告就会淹设在所有的错误警告当中。

SuppressWarnings 注解可以用于任何声明中，从单个局部变量声明到整个类。**总是在尽可能小的范围上使用 SuppressWarnings 注解。**通常用在一个变量声明或一个非常短的方法或构造函数。不要在整个类中使用 SuppressWarnings。这样做可能会掩盖关键警告。

如果你发现自己在一个超过一行的方法或构造函数上使用 SuppressWarnings 注解，那么你可以将其移动到局部变量声明中。你可能需要声明一个新的局部变量，但这是值得的。例如，考虑这个 toArray 方法，它来自 ArrayList：

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
```

```
        if (a.length > size)
            a[size] = null;
        return a;
    }
```

如果你编译 ArrayList，这个方法会产生这样的警告：

```
    ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
    ^ required: T[]
    found: Object[]
```

将 SuppressWarnings 注释放在 return 语句上是非法的，因为它不是声明[JLS, 9.7]。你可能想把注释放在整个方法上，但是不要这样做。相反，应该声明一个局部变量来保存返回值并添加注解，如下所示：

```
    // Adding local variable to reduce scope of @SuppressWarnings
    public <T> T[] toArray(T[] a) {
        if (a.length < size) {
            // This cast is correct because the array we're creating
            // is of the same type as the one passed in, which is T[].
            @SuppressWarnings("unchecked") T[] result = (T[])
Arrays.copyOf(elements, size, a.getClass());
            return result;
        }
        System.arraycopy(elements, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }
```

生成的方法编译正确，并将抑制 unchecked 警告的范围减到最小。

每次使用 SuppressWarnings("unchecked") 注解时，要添加一条注释，说明这样做是安全的。 这将帮助他人理解代码，更重要的是，它将降低其他人修改代码而产生不安全事件的几率。如果你觉得写这样的注释很难，那就继续思考合适的方式。你最终可能会发现，unchecked 操作毕竟是不安全的。

总之，unchecked 警告很重要。不要忽视他们。每个 unchecked 警告都代表了在运行时发生 ClassCastException 的可能性。尽最大努力消除这些警告。如果不能消除 unchecked 警告，并且可以证明引发该警告的代码是类型安全的，那么可以在尽可能狭窄的范围内使用 @SuppressWarnings("unchecked") 注释来禁止警告。在注释中记录你决定隐藏警告的理由。

28 list 列表优于数组

数组与泛型有两个重要区别。首先，数组是协变的。这个听起来很吓人的单词的意思很简单，如果 Sub 是 Super 的一个子类型，那么数组类型 Sub[] 就是数组类型 Super[] 的一个子类型。相比之下，泛型是不变的：对于任何两个不同类型 Type1 和 Type2，List<Type1> 既不是 List<Type2> 的子类型，也不是 List<Type2> 的超类型 [JLS, 4.10; Naftalin07, 2.5]。你可能认为这意味着泛型是有缺陷的，但可以说数组才是有缺陷的。这段代码是合法的：

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

但这一段代码就不是：

```
// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

两种方法都不能将 String 放入 Long 容器，但使用数组，你会得到一个运行时错误；使用 list，你可以在编译时发现问题。当然，你更希望在编译时找到问题。

数组和泛型之间的第二个主要区别：数组是具体化的 [JLS, 4.7]。这意味着数组在运行时知道并强制执行他们的元素类型。如前所述，如果试图将 String

元素放入一个 Long 类型的数组中，就会得到 ArrayStoreException。相比之下，泛型是通过擦除来实现的 [JLS, 4.6]。这意味着它们只在编译时执行类型约束，并在运行时丢弃（或擦除）元素类型信息。擦除允许泛型与不使用泛型的遗留代码自由交互操作（Item-26），确保在 Java 5 中平稳地过渡。

由于这些基本差异，数组和泛型不能很好地混合。例如，创建泛型、参数化类型或类型参数的数组是非法的。因此，这些数组创建表达式都不是合法的：new List<E>[]、new List<String>[]、new E[]。所有这些都会在编译时导致泛型数组创建错误。

为什么创建泛型数组是非法的？因为这不是类型安全的。如果合法，编译器在其他正确的程序中生成的强制转换在运行时可能会失败，并导致 ClassCastException。这将违反泛型系统提供的基本保证。

为了更具体，请考虑以下代码片段：

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1]; // (1)
List<Integer> intList = List.of(42); // (2)
Object[] objects = stringLists; // (3)
objects[0] = intList; // (4)
String s = stringLists[0].get(0); // (5)
```

假设创建泛型数组的第 1 行是合法的。第 2 行创建并初始化一个包含单个元素的 List<Integer>。第 3 行将 List<String>数组存储到 Object 类型的数组变量中，这是合法的，因为数组是协变的。第 4 行将 List<Integer> 存储到 Object 类型的数组的唯一元素中，这是成功的，因为泛型是由擦除实现的：List<Integer> 实例的运行时类型是 List，List<String>[] 实例的运行时类型是 List[]，因此这个赋值不会生成 ArrayStoreException。现在我们有麻烦了。我们将一个 List<Integer> 实例存储到一个数组中，该数组声明只保存 List<String> 实例。在第 5 行，我们从这个数组的唯一列表中检索唯一元素。编译器自动将检索到的元素转换为 String 类型，但它是一个 Integer 类型的元素，因此我们在运行时得到一个 ClassCastException。为了防止这种情况发生，第 1 行（创建泛型数组）必须生成编译时错误。

E、List<E> 和 List<string> 等类型在技术上称为不可具体化类型 [JLS, 4.7]。直观地说，非具体化类型的运行时表示包含的信息少于其编译时表示。由于擦

除，唯一可具体化的参数化类型是无限制通配符类型，如 List<?> 和 Map<?,?>（Item-26）。创建无边界通配符类型数组是合法的，但不怎么有用。

禁止创建泛型数组可能很烦人。例如，这意味着泛型集合通常不可能返回其元素类型的数组（部分解决方案请参见 Item-33）。这也意味着在使用 varargs 方法（Item-53）与泛型组合时，你会得到令人困惑的警告。这是因为每次调用 varargs 方法时，都会创建一个数组来保存 varargs 参数。如果该数组的元素类型不可具体化，则会得到警告。SafeVarargs 注解可以用来解决这个问题（Item-32）。

译注：varargs 方法，指带有可变参数的方法。

当你在转换为数组类型时遇到泛型数组创建错误或 unchecked 强制转换警告时，通常最好的解决方案是使用集合类型 List<E>，而不是数组类型 E[]。你可能会牺牲一些简洁性或性能，但作为交换，你可以获得更好的类型安全性和互操作性。

例如，假设你希望编写一个 Chooser 类，该类的构造函数接受一个集合，而单个方法返回随机选择的集合元素。根据传递给构造函数的集合，可以将选择器用作游戏骰子、魔术 8 球或蒙特卡洛模拟的数据源。下面是一个没有泛型的简单实现：

```
// Chooser - a class badly in need of generics!
public class Chooser {
    private final Object[] choiceArray;
    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
}
    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

要使用这个类，每次使用方法调用时，必须将 choose 方法的返回值从对象转换为所需的类型，如果类型错误，转换将在运行时失败。我们认真考虑了 Item-29 的建议，试图对 Chooser 进行修改，使其具有通用性。变化以粗体显示：

```
// A first cut at making Chooser generic - won't compile
public class Chooser<T> {
    private final T[] choiceArray;
    public Chooser(Collection<T> choices) {
        choiceArray = choices.toArray();
    }
    // choose method unchanged
}
```

如果你尝试编译这个类，你将得到这样的错误消息：

```
Chooser.java:9: error: incompatible types: Object[] cannot be converted to T[]
choiceArray = choices.toArray();
^ where T is a type-variable:
T extends Object declared in class Chooser
```

没什么大不了的，你会说，我把对象数组转换成 T 数组：

```
choiceArray = (T[]) choices.toArray();
```

这样就消除了错误，但你得到一个警告：

```
Chooser.java:9: warning: [unchecked] unchecked cast choiceArray = (T[]) choices.toArray();
^ required: T[], found: Object[]
where T is a type-variable:
T extends Object declared in class Chooser
```

编译器告诉你，它不能保证在运行时转换的安全性，因为程序不知道类型 T 代表什么。记住，元素类型信息在运行时从泛型中删除。这个计划会奏效吗？

是的，但是编译器不能证明它。你可以向自己证明这一点，但是你最好将证据放在注释中，指出消除警告的原因（Item-27），并使用注解隐藏警告。

若要消除 unchecked 强制转换警告，请使用 list 而不是数组。下面是编译时没有错误或警告的 Chooser 类的一个版本：

```
// List-based Chooser - typesafe
public class Chooser<T> {
    private final List<T> choiceList;
    public Chooser(Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
    }
    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

这个版本稍微有点冗长，可能稍微慢一些，但是为了让你安心，在运行时不会得到 ClassCastException 是值得的。

总之，数组和泛型有非常不同的类型规则。数组是协变的、具体化的；泛型是不变的和可被擦除的。因此，数组提供了运行时类型安全，而不是编译时类型安全，对于泛型反之亦然。一般来说，数组和泛型不能很好地混合。如果你发现将它们混合在一起并得到编译时错误或警告，那么你的第一个反应该是将数组替换为 list。

29 优先考虑泛型

通常，对声明进行参数化并使用 JDK 提供的泛型和方法并不太难。编写自己的泛型有点困难，但是值得努力学习。

考虑 Item-7 中简单的堆栈实现：

```
// Object-based collection - a prime candidate for generics
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    public boolean isEmpty() {
        return size == 0;
    }
    private void ensureCapacity() {
        if (elements.length == size)
```

```
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
```

这个类一开始就应该是参数化的，但是因为它不是参数化的，所以我们可以在事后对它进行泛化。换句话说，我们可以对它进行参数化，而不会损害原始非参数化版本的客户端。按照目前的情况，客户端必须转换从堆栈中弹出的对象，而这些转换可能在运行时失败。生成类的第一步是向其声明中添加一个或多个类型参数。在这种情况下，有一个类型参数，表示堆栈的元素类型，这个类型参数的常规名称是 E（Item-68）。

下一步是用适当的类型参数替换所有的 Object 类型，然后尝试编译修改后的程序：

```
// Initial attempt to generify Stack - won't compile!
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E result = elements[--size];
```

```
            elements[size] = null; // Eliminate obsolete reference

            return result;

        } ... // no changes in isEmpty or ensureCapacity

    }
```

通常至少会得到一个错误或警告，这个类也不例外。幸运的是，这个类只生成一个错误：

```
    Stack.java:8: generic array creation

    elements = new E[DEFAULT_INITIAL_CAPACITY];

    ^
```

正如 Item-28 中所解释的，你不能创建非具体化类型的数组，例如 E。每当你编写由数组支持的泛型时，就会出现这个问题。有两种合理的方法来解决它。第一个解决方案直接绕过了创建泛型数组的禁令：创建对象数组并将其强制转换为泛型数组类型。现在，编译器将发出一个警告来代替错误。这种用法是合法的，但（一般而言）它不是类型安全的：

```
    Stack.java:8: warning: [unchecked] unchecked cast

    found: Object[], required: E[]

    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];

    ^
```

编译器可能无法证明你的程序是类型安全的，但你可以。你必须说服自己，unchecked 的转换不会损害程序的类型安全性。所涉及的数组（元素）存储在私有字段中，从未返回给客户端或传递给任何其他方法。数组中存储的惟一元素是传递给 push 方法的元素，它们属于 E 类型，因此 unchecked 的转换不会造成任何损害。

一旦你证明了 unchecked 的转换是安全的，就将警告限制在尽可能小的范围内（Item-27）。在这种情况下，构造函数只包含 unchecked 的数组创建，因此在整个构造函数中取消警告是合适的。通过添加注解来实现这一点，Stack 可以干净地编译，而且你可以使用它而无需显式强制转换或担心 ClassCastException：

```
    // The elements array will contain only E instances from push(E).
```

```
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

消除 Stack 中泛型数组创建错误的第二种方法是将字段元素的类型从 E[] 更改为 Object[]。如果你这样做，你会得到一个不同的错误：

```
Stack.java:19: incompatible types
found: Object, required: E
E result = elements[--size];
^
```

通过将从数组中检索到的元素转换为 E，可以将此错误转换为警告，但你将得到警告：

```
Stack.java:19: warning: [unchecked] unchecked cast
found: Object, required: E
E result = (E) elements[--size];
^
```

因为 E 是不可具体化的类型，编译器无法在运行时检查强制转换。同样，你可以很容易地向自己证明 unchecked 的强制转换是安全的，因此可以适当地抑制警告。根据 Item-27 的建议，我们仅对包含 unchecked 强制转换的赋值禁用警告，而不是对整个 pop 方法禁用警告：

```
// Appropriate suppression of unchecked warning
public E pop() {
    if (size == 0)
        throw new EmptyStackException();
    // push requires elements to be of type E, so cast is correct
```

```
        @SuppressWarnings("unchecked")
        E result =(E) elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
```

消除泛型数组创建的两种技术都有其追随者。第一个更容易读：数组声明为 E[] 类型，这清楚地表明它只包含 E 的实例。它也更简洁：在一个典型的泛型类中，从数组中读取代码中的许多点；第一种技术只需要一次转换（在创建数组的地方），而第二种技术在每次读取数组元素时都需要单独的转换。因此，第一种技术是可取的，在实践中更常用。但是，它确实会造成堆污染（Item-32）：数组的运行时类型与其编译时类型不匹配（除非 E 恰好是 Object）。尽管堆污染在这种情况下是无害的，但这使得一些程序员感到非常不安，因此他们选择了第二种技术。

下面的程序演示了通用 Stack 的使用。程序以相反的顺序打印它的命令行参数并转换为大写。在从堆栈弹出的元素上调用 String 的 toUpperCase 方法不需要显式转换，自动生成的转换保证成功：

```
    // Little program to exercise our generic Stack
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        for (String arg : args)
            stack.push(arg);
        while (!stack.isEmpty())
            System.out.println(stack.pop().toUpperCase());
    }
```

前面的例子可能与 Item-28 相矛盾，Item-28 鼓励优先使用列表而不是数组。在泛型中使用列表并不总是可能的或可取的。Java 本身不支持列表，因此一些泛型（如 ArrayList）必须在数组之上实现。其他泛型（如 HashMap）是在数组之上实现的，以提高性能。大多数泛型与我们的 Stack 示例相似，因为它们的类型参数没有限制：你可以创建 Stack<Object>、Stack<int[]>、Stack<List> 或任

何其他对象引用类型的堆栈。注意，不能创建基本类型的 Stack：试图创建 Stack<int> 或 Stack<double>将导致编译时错误。

这是 Java 泛型系统的一个基本限制。你可以通过使用装箱的基本类型（Item-61）来绕过这一限制。有一些泛型限制了其类型参数的允许值。例如，考虑 java.util.concurrent.DelayQueue，其声明如下：

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>
```

类型参数列表（）要求实际的类型参数 E 是 java.util.concurrent.Delayed 的一个子类型。这允许 DelayQueue 实现及其客户端利用 DelayQueue 元素上的 Delayed 方法，而不需要显式转换或 ClassCastException 的风险。类型参数 E 称为有界类型参数。注意，子类型关系的定义使得每个类型都是它自己的子类型 [JLS, 4.10]，所以创建 DelayQueue<Delayed> 是合法的。

总之，泛型比需要在客户端代码中转换的类型更安全、更容易使用。在设计新类型时，请确保可以在不使用此类类型转换的情况下使用它们。这通常意味着使类型具有通用性。如果你有任何应该是泛型但不是泛型的现有类型，请对它们进行泛型。这将使这些类型的新用户在不破坏现有客户端（Item-26）的情况下更容易使用。

30 优先考虑泛型方法

类可以是泛型的，方法也可以是泛型的。操作参数化类型的静态实用程序方法通常是泛型的。Collections 类中的所有「算法」方法（如 binarySearch 和 sort）都是泛型的。

编写泛型方法类似于编写泛型类型。考虑这个有缺陷的方法，它返回两个集合的并集：

```
// Uses raw types - unacceptable! (Item 26)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

该方法可进行编译，但有两个警告：

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
        Set result = new HashSet(s1);
                     ^
Union.java:6: warning: [
unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
        result.addAll(s2);
                      ^
```

要修复这些警告并使方法类型安全，请修改其声明，以声明表示三个集合（两个参数和返回值）的元素类型的类型参数，并在整个方法中使用该类型参数。类型参数列表声明类型参数，它位于方法的修饰符与其返回类型之间。在本例中，类型参数列表为 <E>，返回类型为 Set<E>。类型参数的命名约定与泛型方法和泛型类型的命名约定相同（Item-29、Item-68）：

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

至少对于简单的泛型方法，这就是（要注意细节的）全部。该方法编译时不生成任何警告，并且提供了类型安全性和易用性。这里有一个简单的程序来演示。这个程序不包含转换，编译时没有错误或警告：

```
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = Set.of("Tom", "Dick", "Harry");
```

```
        Set<String> stooges = Set.of("Larry", "Moe", "Curly");
        Set<String> aflCio = union(guys, stooges);
        System.out.println(aflCio);
    }
```

当你运行程序时，它会打印出 [Moe, Tom, Harry, Larry, Curly, Dick]。（输出元素的顺序可能不同）。

union 方法的一个限制是，所有三个集合（输入参数和返回值）的类型必须完全相同。你可以通过使用有界通配符类型（Item-31）使方法更加灵活。

有时，你需要创建一个对象，该对象是不可变的，但适用于许多不同类型。因为泛型是由擦除（Item-28）实现的，所以你可以为所有需要的类型参数化使用单个对象，但是你需要编写一个静态工厂方法，为每个请求的类型参数化重复分配对象。这种模式称为泛型单例工厂，可用于函数对象（Item-42），如 Collections.reverseOrder，偶尔也用于集合，如 Collections.emptySet。

假设你想要编写一个恒等函数分发器。这些库提供 Function.identity，所以没有理由编写自己的库（Item-59），但是它很有指导意义。在请求标识函数对象时创建一个新的标识函数对象是浪费时间的，因为它是无状态的。如果 Java 的泛型被具体化了，那么每个类型都需要一个标识函数，但是由于它们已经被擦除，一个泛型单例就足够了。它是这样的：

```
    // Generic singleton factory pattern
    private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;
    @SuppressWarnings("unchecked")
    public static <T> UnaryOperator<T> identityFunction() {
        return (UnaryOperator<T>) IDENTITY_FN;
    }
```

IDENTITY_FN 到（UnaryFunction<T>）的转换会生成一个 unchecked 转换警告，因为 UnaryOperator<Object> 并不是每个 T 都是 UnaryOperator<T>，但是恒等函数是特殊的：它会返回未修改的参数，所以我们知道，无论 T 的值是多少，都可以将其作为 UnaryFunction<T> 使用，这是类型安全的。一旦我们这样做了，代码编译就不会出现错误或警告。

下面是一个示例程序，它使用我们的泛型单例作为 UnaryOperator<String> 和 UnaryOperator<Number>。像往常一样，它不包含类型转换和编译，没有错误或警告：

```
// Sample program to exercise generic singleton

public static void main(String[] args) {

    String[] strings = { "jute", "hemp", "nylon" };

    UnaryOperator<String> sameString = identityFunction();

    for (String s : strings)

        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };

    UnaryOperator<Number> sameNumber = identityFunction();

    for (Number n : numbers)

        System.out.println(sameNumber.apply(n));

}
```

允许类型参数被包含该类型参数本身的表达式限制，尽管这种情况比较少见。这就是所谓的递归类型限定。递归类型边界的一个常见用法是与 Comparable 接口相关联，后者定义了类型的自然顺序（Item-14）。该界面如下图所示：

```
public interface Comparable<T> {

    int compareTo(T o);

}
```

类型参数 T 定义了实现 Comparable<T> 的类型的元素可以与之进行比较的类型。在实践中，几乎所有类型都只能与它们自己类型的元素进行比较。例如，String 实现 Comparable<String>， Integer 实现 Comparable<Integer>，等等。

许多方法采用实现 Comparable 的元素集合，在其中进行搜索，计算其最小值或最大值，等等。要做到这些，需要集合中的每个元素与集合中的每个其他元素相比较，换句话说，就是列表中的元素相互比较。下面是如何表达这种约束（的示例）：

```
// Using a recursive type bound to express mutual comparability
public static <E extends Comparable<E>> E max(Collection<E> c);
```

类型限定 <E extends Comparable<E>> 可以被理解为「可以与自身进行比较的任何类型 E」，这或多或少与相互可比性的概念相对应。

下面是一个与前面声明相同的方法。它根据元素的自然顺序计算集合中的最大值，编译时没有错误或警告：

```
// Returns max value in a collection - uses recursive type bound
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");
    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
    result = Objects.requireNonNull(e);
    return result;
}
```

注意，如果列表为空，该方法将抛出 IllegalArgumentException。更好的选择是返回一个 Optional<E>（Item-55）。

递归类型限定可能会变得复杂得多，但幸运的是，这种情况很少。如果你理解这个习惯用法、它的通配符变量（Item-31）和模拟的自类型习惯用法（Item-2），你就能够处理在实践中遇到的大多数递归类型限定。

总之，与要求客户端对输入参数和返回值进行显式转换的方法相比，泛型方法与泛型一样，更安全、更容易使用。与类型一样，你应该确保你的方法可以在不使用类型转换的情况下使用，这通常意味着要使它们具有通用性。与类型类似，你应该将需要强制类型转换的现有方法泛型化。这使得新用户在不破坏现有客户端的情况下更容易使用（Item-26）。

31 使用有界通配符提升 API 的灵活性

如 Item-28 所示，参数化类型是不可变的。换句话说，对于任意两种不同类型 Type1 和 Type2，List<Type1> 既不是 List<Type2> 的子类型，也不是它的父类。虽然 List<String> 不是 List<Object> 的子类型，这和习惯的直觉不符，但它确实有意义。你可以将任何对象放入 List<Object>，但只能将字符串放入 List<String>。因为 List<String> 不能做 List<Object> 能做的所有事情，所以它不是子类型（可通过 Liskov 替换原则来理解这一点，Item-10）。

译注：里氏替换原则（Liskov Substitution Principle，LSP）面向对象设计的基本原则之一。里氏替换原则指出：任何父类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正被复用，而衍生类也能够在父类的基础上增加新的行为。

有时你需要获得比不可变类型更多的灵活性。考虑 Item-29 中的堆栈类。以下是它的公共 API：

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

假设我们想添加一个方法，该方法接受一系列元素并将它们全部推入堆栈。这是第一次尝试：

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

该方法能够正确编译，但并不完全令人满意。如果 Iterable src 的元素类型与堆栈的元素类型完全匹配，那么它正常工作。但是假设你有一个 Stack<Number>，并且调用 push(intVal)，其中 intVal 的类型是 Integer。这是可行的，因为 Integer 是 Number 的子类型。因此，从逻辑上讲，这似乎也应该奏效：

```
Stack<Number> numberStack = new Stack<>();

Iterable<Integer> integers = ... ;

numberStack.pushAll(integers);
```

但是，如果你尝试一下，将会得到这个错误消息，因为参数化类型是不可变的：

```
StackTest.java:7: error: incompatible types: Iterable<Integer>

cannot be converted to Iterable<Number>

        numberStack.pushAll(integers);

                            ^
```

幸运的是，有一种解决方法。Java 提供了一种特殊的参数化类型，有界通配符类型来处理这种情况。pushAll 的输入参数的类型不应该是「E 的 Iterable 接口」，而应该是「E 的某个子类型的 Iterable 接口」，并且有一个通配符类型，它的确切含义是：Iterable<? extends E>（关键字 extends 的使用稍微有些误导：回想一下 Item-29，定义了子类型，以便每个类型都是其本身的子类型，即使它没有扩展自己。）让我们修改 pushAll 来使用这种类型：

```
// Wildcard type for a parameter that serves as an E producer

public void pushAll(Iterable<? extends E> src) {

    for (E e : src)

        push(e);

}
```

更改之后，不仅 Stack 可以正确编译，而且不能用原始 pushAll 声明编译的客户端代码也可以正确编译。因为 Stack 和它的客户端可以正确编译，所以你知道所有东西都是类型安全的。现在假设你想编写一个与 pushAll 一起使用的 popAll 方法。popAll 方法将每个元素从堆栈中弹出，并将这些元素添加到给定的集合中。下面是编写 popAll 方法的第一次尝试：

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

同样，如果目标集合的元素类型与堆栈的元素类型完全匹配，那么这种方法可以很好地编译。但这也不是完全令人满意。假设你有一个 Stack<Number> 和 Object 类型的变量。如果从堆栈中取出一个元素并将其存储在变量中，那么它将编译并运行，不会出错。所以你不能也这样做吗？

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

如果你尝试根据前面显示的 popAll 版本编译此客户端代码，你将得到一个与第一个版本的 pushAll 非常相似的错误：Collection<Object>不是 Collection<Number> 的子类型。同样，通配符类型提供解决方法。popAll 的输入参数的类型不应该是「E 的集合」，而应该是「E 的某个超类型的集合」（其中的超类型定义为 E 本身是一个超类型[JLS, 4.10]）。同样，有一个通配符类型，它的确切含义是：Collection<? super E>。让我们修改 popAll 来使用它：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
   while (!isEmpty())
      dst.add(pop());
}
```

通过此更改，Stack 类和客户端代码都可以正确编译。

教训是清楚的。为了获得最大的灵活性，应在表示生产者或消费者的输入参数上使用通配符类型。如果输入参数既是生产者又是消费者，那么通配符类型对你没有任何好处：你需要一个精确的类型匹配，这就是在没有通配符的情况下得到的结果。这里有一个助记符帮助你记住使用哪种通配符类型：

PECS 表示生产者应使用 extends，消费者应使用 super。

换句话说，如果参数化类型表示 T 生成器，则使用 <? extends T>；如果它表示一个 T 消费者，则使用 <? super T>。在我们的 Stack 示例中，pushAll 的 src 参数生成 E 的实例供 Stack 使用，因此 src 的适当类型是 Iterable<? extends E>；popAll 的 dst 参数使用 Stack 中的 E 实例，因此适合 dst 的类型是 Collection<? super E>。PECS 助记符捕获了指导通配符类型使用的基本原则。Naftalin 和 Wadler 称之为 Get and Put 原则[Naftalin07, 2.4]。

记住这个助记符后，再让我们看一看本章前面提及的一些方法和构造函数声明。Item-28 中的 Chooser 构造函数有如下声明：

```
public Chooser(Collection<T> choices)
```

这个构造函数只使用集合选项来生成类型 T 的值（并存储它们以供以后使用），因此它的声明应该使用扩展 T 的通配符类型 **extends T**。下面是生成的构造函数声明：

```
// Wildcard type for parameter that serves as an T producer

public Chooser(Collection<? extends T> choices)
```

这种改变在实践中会有什么不同吗？是的，它会。假设你有一个 List<Integer>，并且希望将其传递给 Chooser<Number> 的构造函数。这不会与原始声明一起编译，但是一旦你将有界通配符类型添加到声明中，它就会编译。

现在让我们看看 Item-30 中的 union 方法。以下是声明：

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

参数 s1 和 s2 都是 E 的生产者，因此 PECS 助记符告诉我们声明应该如下：

```
public static <E> Set<E> union(Set<? extends E> s1,Set<? extends E> s2)
```

注意，返回类型仍然设置为 Set<E>。**不要使用有界通配符类型作为返回类型。** 它将强制用户在客户端代码中使用通配符类型，而不是为用户提供额外的灵活性。经修订后的声明可正确编译以下代码：

```
Set<Integer> integers = Set.of(1, 3, 5);

Set<Double> doubles = Set.of(2.0, 4.0, 6.0);

Set<Number> numbers = union(integers, doubles);
```

如果使用得当，通配符类型对于类的用户几乎是不可见的。它们让方法接受它们应该接受的参数，拒绝应该拒绝的参数。**如果类的用户必须考虑通配符类型，那么它的 API 可能有问题。**

在 Java 8 之前，类型推断规则还不足以处理前面的代码片段，这要求编译器使用上下文指定的返回类型（或目标类型）来推断 E 的类型。前面显示的 union 调用的目标类型设置为 Set<Number> 如果你尝试在 Java 的早期版本中编译该片段（使用 Set.of factory 的适当替代），你将得到一条长而复杂的错误消息，如下所示：

```
Union.java:14: error: incompatible types

Set<Number> numbers = union(integers, doubles);

^ required: Set<Number>

found: Set<INT#1>

where INT#1,INT#2 are intersection types:

INT#1 extends Number,Comparable<? extends INT#2>

INT#2 extends Number,Comparable<?>
```

幸运的是，有一种方法可以处理这种错误。如果编译器没有推断出正确的类型，你总是可以告诉它使用显式类型参数[JLS, 15.12]使用什么类型。即使在 Java 8 中引入目标类型之前，这也不是必须经常做的事情，这很好，因为显式类型参数不是很漂亮。通过添加显式类型参数，如下所示，代码片段可以在 Java 8 之前的版本中正确编译：

```
// Explicit type parameter - required prior to Java 8

Set<Number> numbers = Union.<Number>union(integers, doubles);
```

接下来让我们将注意力转到 Item-30 中的 max 方法。以下是原始声明：

```
public static <T extends Comparable<T>> T max(List<T> list)
```

下面是使用通配符类型的修正声明：

```
public static <T extends Comparable<? super T>> T max(List<? extends T>
list)
```

为了从原始声明中得到修改后的声明，我们两次应用了 PECS 启发式。直接的应用程序是参数列表。它生成 T 的实例，所以我们将类型从 List<T> 更改

为 List<? extends T>。复杂的应用是类型参数 T。这是我们第一次看到通配符应用于类型参数。最初，T 被指定为扩展 Comparable<T>，但是 T 的 Comparable 消费 T 实例（并生成指示顺序关系的整数）。因此，将参数化类型 Comparable<T> 替换为有界通配符类型 Comparable<? super T>，Comparables 始终是消费者，所以一般应**优先使用 Comparable<? super T> 而不是 Comparable<T>**，比较器也是如此；因此，通常应该**优先使用 Comparator<? super T> 而不是 Comparator<T>**。

修订后的 max 声明可能是本书中最复杂的方法声明。增加的复杂性真的能给你带来什么好处吗？是的，它再次生效。下面是一个简单的列表案例，它在原来的声明中不允许使用，但经订正的声明允许：

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

不能将原始方法声明应用于此列表的原因是 ScheduledFuture 没有实现 Comparable<ScheduledFuture>。相反，它是 Delayed 的一个子接口，扩展了 Comparable<Delayed>。换句话说，ScheduledFuture 的实例不仅仅可以与其他 ScheduledFuture 实例进行比较；它可以与任何 Delayed 实例相比较，这足以导致初始声明时被拒绝。更通俗来说，通配符用于支持不直接实现 Comparable（或 Comparator）但扩展了实现 Comparable（或 Comparator）的类型的类型。

还有一个与通配符相关的主题值得讨论。类型参数和通配符之间存在对偶性，可以使用其中一种方法声明许多方法。例如，下面是静态方法的两种可能声明，用于交换列表中的两个索引项。第一个使用无界类型参数（Item-30），第二个使用无界通配符：

```
// Two possible declarations for the swap method

public static <E> void swap(List<E> list, int i, int j);

public static void swap(List<?> list, int i, int j);
```

这两个声明中哪个更好，为什么？在公共 API 中第二个更好，因为它更简单。传入一个列表（任意列表），该方法交换索引元素。不需要担心类型参数。通常，如果类型参数在方法声明中只出现一次，则用通配符替换它。**如果它是一个无界类型参数，用一个无界通配符替换它；** 如果它是有界类型参数，则用有界通配符替换它。

交换的第二个声明有一个问题。这个简单的实现无法编译：

```
public static void swap(List<?> list, int i, int j) {

   list.set(i, list.set(j, list.get(i)));
```

```
}
```

试图编译它会产生一个不太有用的错误消息：

```
Swap.java:5: error: incompatible types: Object cannot be

converted to CAP#1

list.set(i, list.set(j, list.get(i)));

^ where CAP#1

is a fresh type-variable: CAP#1 extends Object from capture of ?
```

我们不能把一个元素放回刚刚取出的列表中，这看起来是不正确的。问题是 list 的类型是 List<?>，你不能在 List<?> 中放入除 null 以外的任何值。幸运的是，有一种方法可以实现，而无需求助于不安全的强制转换或原始类型。其思想是编写一个私有助手方法来捕获通配符类型。为了捕获类型，helper 方法必须是泛型方法。它看起来是这样的：

```
public static void swap(List<?> list, int i, int j) {

    swapHelper(list, i, j);

}

// Private helper method for wildcard capture

private static <E> void swapHelper(List<E> list, int i, int j) {

    list.set(i, list.set(j, list.get(i)));

}
```

swapHelper 方法知道 list 是一个 List<E>。因此，它知道它从这个列表中得到的任何值都是 E 类型的，并且将 E 类型的任何值放入这个列表中都是安全的。这个稍微复杂的实现能够正确编译。它允许我们导出基于 通配符的声明，同时在内部利用更复杂的泛型方法。swap 方法的客户端不必面对更复杂的 swapHelper 声明，但它们确实从中受益。值得注意的是，helper 方法具有我们认为对于公共方法过于复杂而忽略的签名。

总之，在 API 中使用通配符类型虽然很棘手，但可以使其更加灵活。如果你编写的库将被广泛使用，则必须考虑通配符类型的正确使用。记住基本规则：生产者使用 extends，消费者使用 super（PECS）。还要记住，所有的 comparable 和 comparator 都是消费者。

32 小心组合泛型和可变参数 @

可变参数方法（Item-53）和泛型都是在 Java 5 中添加的，因此你可能认为它们能够优雅地交互；可悲的是，他们并不能。可变参数的目的是允许客户端向方法传递可变数量的参数，但这是一个漏洞百出的抽象概念：当你调用可变参数方法时，将创建一个数组来保存参数；该数组的实现细节应该是可见的。因此，当可变参数具有泛型或参数化类型时，会出现令人困惑的编译器警告。

回想一下 Item-28，非具体化类型是指其运行时表示的信息少于其编译时表示的信息，并且几乎所有泛型和参数化类型都是不可具体化的。如果方法声明其可变参数为不可具体化类型，编译器将在声明上生成警告。如果方法是在其推断类型不可具体化的可变参数上调用的，编译器也会在调用时生成警告。生成的警告就像这样：

```
warning: [unchecked] Possible heap pollution from parameterized vararg type List<String>
```

当参数化类型的变量引用不属于该类型的对象时，就会发生堆污染[JLS, 4.12.2]。它会导致编译器自动生成的强制类型转换失败，违反泛型类型系统的基本保证。

例如，考虑这个方法，它摘自 127 页（Item-26）的代码片段，但做了些修改：

```
// Mixing generics and varargs can violate type safety!
// 泛型和可变参数混合使用可能违反类型安全原则！
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList; // Heap pollution
    String s = stringLists[0].get(0); // ClassCastException
}
```

此方法没有显式的强制类型转换，但在使用一个或多个参数调用时抛出 ClassCastException。它的最后一行有一个由编译器生成的隐式强制转换。此转

换失败，表明类型安全性受到了影响，并且**在泛型可变参数数组中存储值是不安全的。**

这个例子提出了一个有趣的问题：为什么使用泛型可变参数声明方法是合法的，而显式创建泛型数组是非法的？换句话说，为什么前面显示的方法只生成警告，而 127 页上的代码片段发生错误？答案是，带有泛型或参数化类型的可变参数的方法在实际开发中非常有用，因此语言设计人员选择忍受这种不一致性。事实上，Java 库导出了几个这样的方法，包括 Arrays.asList(T... a)、Collections.addAll(Collection<? super T> c, T... elements) 以及 EnumSet.of(E first, E... rest)。它们与前面显示的危险方法不同，这些库方法是类型安全的。

在 Java 7 之前，使用泛型可变参数的方法的作者对调用点上产生的警告无能为力。使得这些 API 难以使用。用户必须忍受这些警告，或者在每个调用点（Item-27）使用 @SuppressWarnings("unchecked") 注释消除这些警告。这种做法乏善可陈，既损害了可读性，也忽略了标记实际问题的警告。

在 Java 7 中添加了 SafeVarargs 注释，以允许使用泛型可变参数的方法的作者自动抑制客户端警告。本质上，**SafeVarargs 注释构成了方法作者的一个承诺，即该方法是类型安全的。** 作为这个承诺的交换条件，编译器同意不对调用可能不安全的方法的用户发出警告。

关键问题是，使用 @SafeVarargs 注释方法，该方法实际上应该是安全的。那么怎样才能确保这一点呢？回想一下，在调用该方法时创建了一个泛型数组来保存可变参数。如果方法没有将任何内容存储到数组中（这会覆盖参数），并且不允许对数组的引用进行转义（这会使不受信任的代码能够访问数组），那么它就是安全的。换句话说，如果可变参数数组仅用于将可变数量的参数从调用方传输到方法（毕竟这是可变参数的目的），那么该方法是安全的。

值得注意的是，在可变参数数组中不存储任何东西就可能违反类型安全性。考虑下面的通用可变参数方法，它返回一个包含参数的数组。乍一看，它似乎是一个方便的小实用程序：

```
// UNSAFE - Exposes a reference to its generic parameter array!

static <T> T[] toArray(T... args) {

    return args;

}
```

这个方法只是返回它的可变参数数组。这种方法看起来并不危险，但确实危险！这个数组的类型由传递给方法的参数的编译时类型决定，编译器可能没

有足够的信息来做出准确的决定。因为这个方法返回它的可变参数数组，所以它可以将堆污染传播到调用堆栈上。

为了使其具体化，请考虑下面的泛型方法，该方法接受三个类型为 T 的参数，并返回一个包含随机选择的两个参数的数组：

```
static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
        case 0: return toArray(a, b);
        case 1: return toArray(a, c);
        case 2: return toArray(b, c);
    }
    throw new AssertionError(); // Can't get here
}
```

这个方法本身并不危险，并且不会生成警告，除非它调用 toArray 方法，该方法有一个通用的可变参数。

编译此方法时，编译器生成代码来创建一个可变参数数组，在该数组中向 toArray 传递两个 T 实例。这段代码分配了 type Object[] 的一个数组，这是保证保存这些实例的最特定的类型，无论调用站点上传递给 pickTwo 的是什么类型的对象。toArray 方法只是将这个数组返回给 pickTwo，而 pickTwo 又将这个数组返回给它的调用者，所以 pickTwo 总是返回一个 Object[] 类型的数组。

现在考虑这个主要方法，练习 pickTwo：

```
public static void main(String[] args) {
    String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

这个方法没有任何错误，因此它在编译时不会生成任何警告。但是当你运行它时，它会抛出 ClassCastException，尽管它不包含可见的强制类型转换。你没有看到的是，编译器在 pickTwo 返回的值上生成了一个隐藏的 String[] 转换，这样它就可以存储在属性中。转换失败，因为 Object[] 不是 String[] 的子类型。这个失败非常令人不安，因为它是从方法中删除了两个导致堆污染的级别（toArray），并且可变参数数组在实际参数存储在其中之后不会被修改。

这个示例的目的是让人明白，**让另一个方法访问泛型可变参数数组是不安全的**，只有两个例外：将数组传递给另一个使用 @SafeVarargs 正确注释的可变参数方法是安全的，将数组传递给仅计算数组内容的某个函数的非可变方法也是安全的。

下面是一个安全使用泛型可变参数的典型示例。该方法接受任意数量的列表作为参数，并返回一个包含所有输入列表的元素的序列列表。因为该方法是用 @SafeVarargs 注释的，所以它不会在声明或调用点上生成任何警告：

```
// Safe method with a generic varargs parameter
@SafeVarargs
static <T> List<T> flatten(List<? extends T>... lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

决定何时使用 SafeVarargs 注释的规则很简单：**在每个带有泛型或参数化类型的可变参数的方法上使用 @SafeVarargs**，这样它的用户就不会被不必要的和令人困惑的编译器警告所困扰。这意味着你永远不应该编写像 dangerous 或 toArray 这样不安全的可变参数方法。每当编译器警告你控制的方法中的泛型可变参数可能造成堆污染时，请检查该方法是否安全。提醒一下，一个通用的可变参数方法是安全的，如果：

它没有在可变参数数组中存储任何东西，并且

它不会让数组（或者其副本）出现在不可信的代码中。如果违反了这些禁令中的任何一条，就纠正它。

请注意，SafeVarargs 注释仅对不能覆盖的方法合法，因为不可能保证所有可能覆盖的方法都是安全的。在 Java 8 中，注释仅对静态方法和最终实例方法合法；在 Java 9 中，它在私有实例方法上也成为合法的。

使用 SafeVarargs 注释的另一种选择是接受 Item-28 的建议，并用 List 参数替换可变参数（它是一个伪装的数组）。下面是将这种方法应用到我们的 flatten 方法时的效果。注意，只有参数声明发生了更改：

```
// List as a typesafe alternative to a generic varargs parameter
static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

然后可以将此方法与静态工厂方法 List.of 一起使用，以允许可变数量的参数。注意，这种方法依赖于 List.of 声明是用 @SafeVarargs 注释的：

```
audience = flatten(List.of(friends, romans, countrymen));
```

这种方法的优点是编译器可以证明该方法是类型安全的。你不必使用 SafeVarargs 注释来保证它的安全性，也不必担心在确定它的安全性时可能出错。主要的缺点是客户端代码比较冗长，可能会比较慢。

这种技巧也可用于无法编写安全的可变参数方法的情况，如第 147 页中的 toArray 方法。它的列表类似于 List.of 方法，我们甚至不用写；Java 库的作者为我们做了这些工作。pickTwo 方法变成这样：

```
static <T> List<T> pickTwo(T a, T b, T c) {
    switch(rnd.nextInt(3)) {
        case 0: return List.of(a, b);
        case 1: return List.of(a, c);
        case 2: return List.of(b, c);
    }
    throw new AssertionError();
}
```

main 方法是这样的：

```
public static void main(String[] args) {
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
```

```
}
```

.

生成的代码是类型安全的，因为它只使用泛型，而不使用数组。

总之，可变参数方法和泛型不能很好地交互，因为可变参数工具是构建在数组之上的漏洞抽象，并且数组具有与泛型不同的类型规则。虽然泛型可变参数不是类型安全的，但它们是合法的。如果选择使用泛型（或参数化）可变参数编写方法，首先要确保该方法是类型安全的，然后使用 @SafeVarargs 对其进行注释，这样使用起来就不会令人不愉快。

33 优先考虑类型安全的异构容器

集合是泛型的常见应用之一，如 Set<E> 和 Map<K,V>，以及单元素容器，如 ThreadLocal<T> 和 AtomicReference<T>。在所有这些应用中，都是参数化的容器。这将每个容器的类型参数限制为固定数量。通常这正是你想要的。Set 只有一个类型参数，表示其元素类型；Map 有两个，表示键和值的类型；如此等等。

然而，有时你需要更大的灵活性。例如，一个数据库行可以有任意多列，能够以类型安全的方式访问所有这些列是很好的。幸运的是，有一种简单的方法可以达到这种效果。其思想是参数化键而不是容器。然后向容器提供参数化键以插入或检索值。泛型类型系统用于确保值的类型与键一致。

作为这种方法的一个简单示例，考虑一个 Favorites 类，它允许客户端存储和检索任意多种类型的 Favorites 实例。Class 类的对象将扮演参数化键的角色。这样做的原因是 Class 类是泛型的。Class 对象的类型不仅仅是 Class，而是 Class<T>。例如，String.class 的类型为 Class<String>、Integer.class 的类型为 Class<Integer>。在传递编译时和运行时类型信息的方法之间传递类 Class 对象时，它被称为类型标记[Bracha04]。

Favorites 类的 API 很简单。它看起来就像一个简单的 Map，只不过键是参数化的，而不是整个 Map。客户端在设置和获取 Favorites 实例时显示一个 Class 对象。以下是 API:

```
// Typesafe heterogeneous container pattern - API

public class Favorites {

    public <T> void putFavorite(Class<T> type, T instance);
```

```
        public <T> T getFavorite(Class<T> type);
    }
```

下面是一个示例程序，它演示了 Favorites 类、存储、检索和打印 Favorites 字符串、整数和 Class 实例：

```
    // Typesafe heterogeneous container pattern - client
    public static void main(String[] args) {
        Favorites f = new Favorites();
        f.putFavorite(String.class, "Java");
        f.putFavorite(Integer.class, 0xcafebabe);
        f.putFavorite(Class.class, Favorites.class);
        String favoriteString = f.getFavorite(String.class);
        int favoriteInteger = f.getFavorite(Integer.class);
        Class<?> favoriteClass = f.getFavorite(Class.class);
        System.out.printf("%s %x %s%n", favoriteString,favoriteInteger,
favoriteClass.getName());
    }
```

如你所料，这个程序打印 Java cafebabe Favorites。顺便提醒一下，Java 的 printf 方法与 C 的不同之处在于，你应该在 C 中使用 \n 的地方改用 %n。

译注：favoriteClass.getName() 的打印结果与 Favorites 类所在包名有关，结果应为：包名.Favorites

Favorites 的实例是类型安全的：当你向它请求一个 String 类型时，它永远不会返回一个 Integer 类型。它也是异构的：与普通 Map 不同，所有键都是不同类型的。因此，我们将 Favorites 称为一个类型安全异构容器。

Favorites 的实现非常简短。下面是全部内容：

```
    // Typesafe heterogeneous container pattern - implementation
    public class Favorites {
       private Map<Class<?>, Object> favorites = new HashMap<>();
```

```
    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instance);
    }
    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

这里发生了一些微妙的事情。每个 Favorites 实例都由一个名为 favorites 的私有 Map<Class<?>, Object> 支持。你可能认为由于通配符类型是无界的，所以无法将任何内容放入此映射中，但事实恰恰相反。需要注意的是，通配符类型是嵌套的：通配符类型不是 Map 的类型，而是键的类型。这意味着每个键都可以有不同的参数化类型：一个可以是 Class<String>，下一个是 Class<Integer>，等等。这就是异构的原理。

接下来要注意的是 favorites 的值类型仅仅是 Object。换句话说，Map 不保证键和值之间的类型关系，即每个值都是其键所表示的类型。实际上，Java 的类型系统还没有强大到足以表达这一点。但是我们知道这是事实，当需要检索一个 favorite 时，我们会利用它。

putFavorite 的实现很简单：它只是将从给定 Class 对象到给定 Favorites 实例的放入 favorites 中。如前所述，这将丢弃键和值之间的「类型关联」；将无法确定值是键的实例。但这没关系，因为 getFavorites 方法可以重新建立这个关联。

getFavorite 的实现比 putFavorite 的实现更复杂。首先，它从 favorites 中获取与给定 Class 对象对应的值。这是正确的对象引用返回，但它有错误的编译时类型：它是 Object（favorites 的值类型），我们需要返回一个 T。因此，getFavorite 的实现通过使用 Class 的 cast 方法，将对象引用类型动态转化为所代表的 Class 对象。

cast 方法是 Java 的 cast 运算符的动态模拟。它只是检查它的参数是否是类对象表示的类型的实例。如果是，则返回参数；否则它将抛出 ClassCastException。我们知道 getFavorite 中的强制转换调用不会抛出 ClassCastException，假设客户端代码已正确地编译。也就是说，我们知道 favorites 中的值总是与其键的类型匹配。

如果 cast 方法只是返回它的参数，那么它会为我们做什么呢？cast 方法的签名充分利用了 Class 类是泛型的这一事实。其返回类型为 Class 对象的类型参数：

```
public class Class<T> {
    T cast(Object obj);
}
```

这正是 getFavorite 方法所需要的。它使我们能够使 Favorites 类型安全，而不需要对 T 进行 unchecked 的转换。

Favorites 类有两个值得注意的限制。首先，恶意客户端很容易通过使用原始形式的类对象破坏 Favorites 实例的类型安全。但是生成的客户端代码在编译时将生成一个 unchecked 警告。这与普通的集合实现（如 HashSet 和 HashMap）没有什么不同。通过使用原始类型 HashSet（Item-26），可以轻松地将 String 类型放入 HashSet<Integer> 中。也就是说，如果你愿意付出代价的话，你可以拥有运行时类型安全。确保 Favorites 不会违反其类型不变量的方法是让 putFavorite 方法检查实例是否是 type 表示的类型的实例，我们已经知道如何做到这一点。只需使用动态转换：

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

java.util.Collections 中的集合包装器也具有相同的功能。它们被称为 checkedSet、checkedList、checkedMap，等等。除了集合（或 Map）外，它们的静态工厂还接受一个（或两个）Class 对象。静态工厂是通用方法，确保 Class 对象和集合的编译时类型匹配。包装器将具体化添加到它们包装的集合中。例如，如果有人试图将 Coin 放入 Collection<Stamp> 中，包装器将在运行时抛出 ClassCastException。在混合了泛型类型和原始类型的应用程序中，这些包装器对跟踪将类型错误的元素添加到集合中的客户端代码非常有用。

Favorites 类的第二个限制是它不能用于不可具体化的类型（Item-28）。换句话说，你可以存储的 Favorites 实例类型为 String 类型或 String[]，但不能存储 List<String>。原因是你不能为 List<String> 获取 Class 对象，List<String>.class 是一个语法错误，这也是一件好事。
List<String> 和 List<Integer> 共享一个 Class 对象，即 List.class。如果「字面类

型」List<String>.class 和 List<Integer>.class 是合法的，并且返回相同的对象引用，那么它将严重破坏 Favorites 对象的内部结构。对于这个限制，没有完全令人满意的解决方案。

Favorites 使用的类型标记是无界的：getFavorite 和 put-Favorite 接受任何 Class 对象。有时你可能需要限制可以传递给方法的类型。这可以通过有界类型标记来实现，它只是一个类型标记，使用有界类型参数（Item-30）或有界通配符（Item-31）对可以表示的类型进行绑定。

annotation API（Item-39）广泛使用了有界类型标记。例如，下面是在运行时读取注释的方法。这个方法来自 AnnotatedElement 接口，它是由表示类、方法、字段和其他程序元素的反射类型实现的：

```
public <T extends Annotation>

    T getAnnotation(Class<T> annotationType);
```

参数 annotationType 是表示注释类型的有界类型标记。该方法返回该类型的元素注释（如果有的话），或者返回 null（如果没有的话）。本质上，带注释的元素是一个类型安全的异构容器，其键是注释类型。

假设你有一个 Class<?> 类型的对象，并且希望将其传递给一个需要有界类型令牌（例如 getAnnotation）的方法。你可以将对象强制转换为 Class<? extends Annotation>，但是这个强制转换是未选中的，因此它将生成一个编译时警告（Item-27）。幸运的是，class 类提供了一个实例方法，可以安全地（动态地）执行这种类型的强制转换。该方法称为 asSubclass，它将类对象强制转换为它所调用的类对象，以表示由其参数表示的类的子类。如果转换成功，则该方法返回其参数；如果失败，则抛出 ClassCastException。

下面是如何使用 asSubclass 方法读取在编译时类型未知的注释。这个方法编译没有错误或警告：

```
// Use of asSubclass to safely cast to a bounded type token

static Annotation getAnnotation(AnnotatedElement element,String
annotationTypeName) {

    Class<?> annotationType = null; // Unbounded type token

    try {

        annotationType = Class.forName(annotationTypeName);

    } catch (Exception ex) {
```

```
                throw new IllegalArgumentException(ex);

            }

            return
element.getAnnotation(annotationType.asSubclass(Annotation.class));

    }
```

总之，以集合的 API 为例的泛型在正常使用时将每个容器的类型参数限制为固定数量。你可以通过将类型参数放置在键上而不是容器上来绕过这个限制。你可以使用 Class 对象作为此类类型安全异构容器的键。以这种方式使用的 Class 对象称为类型标记。还可以使用自定义键类型。例如，可以使用 DatabaseRow 类型表示数据库行（容器），并使用泛型类型 Column<T> 作为它的键。

第六章 枚举和注解

JAVA 支持两种特殊用途的引用类型：一种称为枚举类型的类，以及一种称为注解类型的接口。本章将讨论这些类型在实际使用时的最佳方式。

34 用枚举 enum 代替 int 常量

枚举类型是这样一种类型：它合法的值由一组固定的常量组成，如：一年中的季节、太阳系中的行星或扑克牌中的花色。在枚举类型被添加到 JAVA 之前，表示枚举类型的一种常见模式是声明一组 int 的常量，每个类型的成员都有一个：

```
    // The int enum pattern - severely deficient!

    public static final int APPLE_FUJI = 0;

    public static final int APPLE_PIPPIN = 1;

    public static final int APPLE_GRANNY_SMITH = 2;

    public static final int ORANGE_NAVEL = 0;

    public static final int ORANGE_TEMPLE = 1;

    public static final int ORANGE_BLOOD = 2;
```

这种技术称为 int 枚举模式，它有许多缺点。它没有提供任何类型安全性，并且几乎不具备表现力。如果你传递一个苹果给方法，希望得到一个橘子，使用 == 操作符比较苹果和橘子时编译器并不会提示错误，或更糟的情况：

```
// Tasty citrus flavored applesauce!

int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

注意，每个 apple 常量的名称都以 APPLE_ 为前缀，每个 orange 常量的名称都以 ORANGE_ 为前缀。这是因为 Java 不为这些 int 枚举提供名称空间。当两组 int 枚举具有相同的命名常量时，前缀可以防止名称冲突，例如 ELEMENT_MERCURY 和 PLANET_MERCURY 之间的冲突。

使用 int 枚举的程序很脆弱。因为 int 枚举是常量变量 [JLS, 4.12.4]，所以它们的值被编译到使用它们的客户端中 [JLS, 13.1]。如果与 int 枚举关联的值发生了更改，则必须重新编译客户端。如果不重新编译，客户端仍然可以运行，但是他们的行为将是错误的。

没有一种简单的方法可以将 int 枚举常量转换为可打印的字符串。如果你打印这样的常量或从调试器中显示它，你所看到的只是一个数字，这不是很有帮助。没有可靠的方法可以遍历组中的所有 int 枚举常量，甚至无法获得组的大小。

可能会遇到这种模式的另一种形式：使用 String 常量代替 int 常量。这种称为 String 枚举模式的变体甚至更不可取。虽然它确实为常量提供了可打印的字符串，但是它可能会导致不知情的用户将字符串常量硬编码到客户端代码中，而不是使用字段名。如果这样一个硬编码的 String 常量包含一个排版错误，它将在编译时躲过检测，并在运行时导致错误。此外，它可能会导致性能问题，因为它依赖于字符串比较。

幸运的是，Java 提供了一种替代方案，它避免了 int 和 String 枚举模式的所有缺点，并提供了许多额外的好处。它就是枚举类型 [JLS, 8.9]。下面是它最简单的形式：

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }

public enum Orange { NAVEL, TEMPLE, BLOOD }
```

从表面上看，这些枚举类型可能与其他语言（如 C、c++ 和 c#）的枚举类型类似，但不能只看表象。Java 的枚举类型是成熟的类，比其他语言中的枚举类型功能强大得多，在其他语言中的枚举本质上是 int 值。

Java 枚举类型背后的基本思想很简单：它们是通过 public static final 修饰的字段为每个枚举常量导出一个实例的类。枚举类型实际上是 final 类型，因为没有可访问的构造函数。客户端既不能创建枚举类型的实例，也不能扩展它，所以除了声明的枚举常量之外，不能有任何实例。换句话说，枚举类型是实例受控的类（参阅第 6 页，Item-1）。它们是单例（Item-3）的推广应用，单例本质上是单元素的枚举。

枚举提供编译时类型的安全性。如果将参数声明为 Apple 枚举类型，则可以保证传递给该参数的任何非空对象引用都是三个有效 Apple 枚举值之一。尝试传递错误类型的值将导致编译时错误，将一个枚举类型的表达式赋值给另一个枚举类型的变量，或者使用 == 运算符比较不同枚举类型的值同样会导致错误。

名称相同的枚举类型常量能和平共存，因为每种类型都有自己的名称空间。你可以在枚举类型中添加或重新排序常量，而无需重新编译其客户端，因为导出常量的字段在枚举类型及其客户端之间提供了一层隔离：常量值不会像在 int 枚举模式中那样编译到客户端中。最后，你可以通过调用枚举的 toString 方法将其转换为可打印的字符串。

除了纠正 int 枚举的不足之外，枚举类型还允许添加任意方法和字段并实现任意接口。它们提供了所有 Object 方法的高质量实现（参阅 Chapter 3），还实现了 Comparable（Item-14）和 Serializable（参阅 Chapter 12），并且它们的序列化形式被设计成能够适应枚举类型的可变性。

那么，为什么要向枚举类型添加方法或字段呢？首先，你可能希望将数据与其常量关联起来。例如，我们的 Apple 和 Orange 类型可能受益于返回水果颜色的方法，或者返回水果图像的方法。你可以使用任何适当的方法来扩充枚举类型。枚举类型可以从枚举常量的简单集合开始，并随着时间的推移演变为功能齐全的抽象。

对于富枚举类型来说，有个很好的例子，考虑我们太阳系的八颗行星。每颗行星都有质量和半径，通过这两个属性你可以计算出它的表面引力。反过来，可以给定物体的质量，让你计算出一个物体在行星表面的重量。这个枚举是这样的。每个枚举常量后括号中的数字是传递给其构造函数的参数。在本例中，它们是行星的质量和半径：

```
// Enum type with data and behavior

public enum Planet {

    MERCURY(3.302e+23, 2.439e6),
```

```
        VENUS (4.869e+24, 6.052e6),
        EARTH (5.975e+24, 6.378e6),
        MARS (6.419e+23, 3.393e6),
        JUPITER(1.899e+27, 7.149e7),
        SATURN (5.685e+26, 6.027e7),
        URANUS (8.683e+25, 2.556e7),
        NEPTUNE(1.024e+26, 2.477e7);
        private final double mass; // In kilograms
        private final double radius; // In meters
        private final double surfaceGravity; // In m / s^2
        // Universal gravitational constant in m^3 / kg s^2
        private static final double G = 6.67300E-11;
        // Constructor
        Planet(double mass, double radius) {
            this.mass = mass;
            this.radius = radius;
            surfaceGravity = G * mass / (radius * radius);
        }
        public double mass() { return mass; }
        public double radius() { return radius; }
        public double surfaceGravity() { return surfaceGravity; }
        public double surfaceWeight(double mass) {
            return mass * surfaceGravity; // F = ma
        }
}
```

编写一个富枚举类型很容易，如上述的 Planet。**要将数据与枚举常量关联，可声明实例字段并编写一个构造函数，该构造函数接受数据并将其存储在字段中。** 枚举本质上是不可变的，因此所有字段都应该是 final（Item-17）。字段可以是公共的，但是最好将它们设置为私有并提供公共访问器（Item-16）。在 Planet 的例子中，构造函数还计算和存储表面重力，但这只是一个优化。每一次使用 surfaceWeight 方法时，都可以通过质量和半径重新计算重力。surfaceWeight 方法获取一个物体的质量，并返回其在该常数所表示的行星上的重量。虽然 Planet 枚举很简单，但它的力量惊人。下面是一个简短的程序，它获取一个物体的地球重量（以任何单位表示），并打印一个漂亮的表格，显示该物体在所有 8 个行星上的重量（以相同的单位表示）：

```
public class WeightTable {

    public static void main(String[] args) {

        double earthWeight = Double.parseDouble(args[0]);

        double mass = earthWeight / Planet.EARTH.surfaceGravity();

    for (Planet p : Planet.values())

        System.out.printf("Weight on %s is %f%n",p,
p.surfaceWeight(mass));

    }

}
```

请注意，Planet 和所有枚举一样，有一个静态值方法，该方法按照声明值的顺序返回其值的数组。还要注意的是，toString 方法返回每个枚举值的声明名称，这样就可以通过 println 和 printf 轻松打印。如果你对这个字符串表示不满意，可以通过重写 toString 方法来更改它。下面是用命令行运行我们的 WeightTable 程序（未覆盖 toString）的结果：

```
Weight on MERCURY is 69.912739

Weight on VENUS is 167.434436

Weight on EARTH is 185.000000

Weight on MARS is 70.226739

Weight on JUPITER is 467.990696

Weight on SATURN is 197.120111
```

```
Weight on URANUS is 167.398264

Weight on NEPTUNE is 210.208751
```

直到 2006 年，也就是枚举被添加到 Java 的两年后，冥王星还是一颗行星。这就提出了一个问题：「从枚举类型中删除元素时会发生什么?」答案是，任何不引用被删除元素的客户端程序将继续正常工作。例如，我们的 WeightTable 程序只需打印一个少一行的表。那么引用被删除元素（在本例中是 Planet.Pluto）的客户端程序又如何呢？如果重新编译客户端程序，编译将失败，并在引用该「过时」行星的行中显示一条有用的错误消息；如果你未能重新编译客户端，它将在运行时从这行抛出一个有用的异常。这是你所希望的最佳行为，比 int 枚举模式要好得多。

与枚举常量相关的一些行为可能只需要在定义枚举的类或包中使用。此类行为最好以私有或包私有方法来实现。然后，每个常量都带有一个隐藏的行为集合，允许包含枚举的类或包在使用该常量时做出适当的反应。与其他类一样，除非你有充分的理由向其客户端公开枚举方法，否则将其声明为私有的，或者在必要时声明为包私有（Item-15）。

译注：Java 中访问级别规则如下：

- **类访问级别：public（公共）、无修饰符（package-private，包私有）**
- **成员访问级别：public（公共）、protected（保护）、private（私有）、无修饰符（package-private，包私有）**

通常，如果一个枚举用途广泛，那么它应该是顶级类；如果它被绑定到一个特定的顶级类使用，那么它应该是这个顶级类（Item-24）的成员类。例如，java.math.RoundingMode 枚举表示小数部分的舍入模式。BigDecimal 类使用这些四舍五入模式，但是它们提供了一个有用的抽象，这个抽象与 BigDecimal 没有本质上的联系。通过使 RoundingMode 成为顶级枚举，库设计人员支持任何需要舍入模式的程序员重用该枚举，从而提高 API 之间的一致性。

Planet 示例中演示的技术对于大多数枚举类型来说已经足够了，但有时还需要更多。每个行星常数都有不同的数据，但有时你需要将基本不同的行为与每个常数联系起来。例如，假设你正在编写一个枚举类型来表示一个基本的四则运算计算器上的操作，并且你希望提供一个方法来执行由每个常量表示的算术操作。实现这一点的一种方式是切换枚举的值：

```
// Enum type that switches on its own value - questionable

public enum Operation {
```

```
        PLUS, MINUS, TIMES, DIVIDE;
        // Do the arithmetic operation represented by this constant
        public double apply(double x, double y) {
            switch(this) {
                case PLUS: return x + y;
                case MINUS: return x - y;
                case TIMES: return x * y;
                case DIVIDE: return x / y;
            }
        throw new AssertionError("Unknown op: "+ this);
        }
    }
```

这段代码可以工作，但不是很漂亮。如果没有抛出语句，它将无法编译，因为从理论上讲，方法的结尾是可到达的，尽管它确实永远不会到达 [JLS, 14.21]。更糟糕的是，代码很脆弱。如果你添加了一个新的枚举常量，但忘记向 switch 添加相应的 case，则枚举仍将编译，但在运行时尝试应用新操作时将失败。

幸运的是，有一种更好的方法可以将不同的行为与每个枚举常量关联起来：在枚举类型中声明一个抽象的 apply 方法，并用一个特定于常量的类体中的每个常量的具体方法覆盖它。这些方法称为特定常量方法实现：

```
    // Enum type with constant-specific method implementations
    public enum Operation {
        PLUS {public double apply(double x, double y){return x + y;}},
        MINUS {public double apply(double x, double y){return x - y;}},
        TIMES {public double apply(double x, double y){return x * y;}},
        DIVIDE{public double apply(double x, double y){return x / y;}};
        public abstract double apply(double x, double y);
```

```
}
```

如果你在 Operation 枚举的第二个版本中添加一个新常量，那么你不太可能忘记提供一个 apply 方法，因为该方法紧跟每个常量声明。在不太可能忘记的情况下，编译器会提醒你，因为枚举类型中的抽象方法必须用其所有常量中的具体方法覆盖。

特定常量方法实现可以与特定于常量的数据相结合。例如，下面是一个 Operation 枚举的版本，它重写 toString 方法来返回与操作相关的符号：

译注：原文 constantspecific data 修改为 constant-specific data ，译为「特定常量数据」

```
// Enum type with constant-specific class bodies and data
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override
    public String toString() { return symbol; }
```

```
        public abstract double apply(double x, double y);
    }
```

The toString implementation shown makes it easy to print arithmetic expressions, as demonstrated by this little program:

重写的 toString 实现使得打印算术表达式变得很容易，如下面的小程序所示：

```
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.printf("%f %s %f = %f%n",x, op, y, op.apply(x, y));
    }
```

以 2 和 4 作为命令行参数运行这个程序将产生以下输出：

```
    2.000000 + 4.000000 = 6.000000
    2.000000 - 4.000000 = -2.000000
    2.000000 * 4.000000 = 8.000000
    2.000000 / 4.000000 = 0.500000
```

枚举类型有一个自动生成的 valueOf(String) 方法，该方法将常量的名称转换为常量本身。如果在枚举类型中重写 toString 方法，可以考虑编写 fromString 方法将自定义字符串表示形式转换回相应的枚举。只要每个常量都有唯一的字符串表示形式，下面的代码（类型名称适当更改）就可以用于任何枚举：

```
    // Implementing a fromString method on an enum type
    private static final Map<String, Operation> stringToEnum
=Stream.of(values()).collect(toMap(Object::toString, e -> e));
    // Returns Operation for string, if any
    public static Optional<Operation> fromString(String symbol) {
        return Optional.ofNullable(stringToEnum.get(symbol));
```

```
    }
```

注意，Operation 枚举的常量是从创建枚举常量之后运行的静态字段初始化中放入 stringToEnum 的。上述代码在 values() 方法返回的数组上使用流（参阅第 7 章）；在 Java 8 之前，我们将创建一个空 HashMap，并遍历值数组，将自定义字符串与枚举的映射插入到 HashMap 中，如果你愿意，你仍然可以这样做。但是请注意，试图让每个常量通过构造函数将自身放入 HashMap 中是行不通的。它会导致编译错误，这是好事，因为如果合法，它会在运行时导致 NullPointerException。枚举构造函数不允许访问枚举的静态字段，常量变量除外（Item-34）。这个限制是必要的，因为在枚举构造函数运行时静态字段还没有初始化。这种限制的一个特殊情况是枚举常量不能从它们的构造函数中相互访问。

还要注意 fromString 方法返回一个 Optional<String>。这允许该方法提示传入的字符串并非有效操作，并强制客户端处理这种可能（Item-55）。

特定常量方法实现的一个缺点是，它们使得在枚举常量之间共享代码变得更加困难。例如，考虑一个表示一周当中计算工资发放的枚举。枚举有一个方法，该方法根据工人的基本工资（每小时）和当天的工作分钟数计算工人当天的工资。在五个工作日内，任何超过正常轮班时间的工作都会产生加班费；在两个周末，所有的工作都会产生加班费。使用 switch 语句，通过多个 case 标签应用于每一类情况，可以很容易地进行计算：

```
    // Enum that switches on its value to share code - questionable

    enum PayrollDay {

        MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY,SATURDAY, SUNDAY;

        private static final int MINS_PER_SHIFT = 8 * 60;

        int pay(int minutesWorked, int payRate) {

            int basePay = minutesWorked * payRate;

            int overtimePay;

            switch(this) {

                case SATURDAY:

                case SUNDAY: // Weekend
```

```
                        overtimePay = basePay / 2;

                        break;

                    default: // Weekday

                        overtimePay = minutesWorked <= MINS_PER_SHIFT ?0 :
(minutesWorked - MINS_PER_SHIFT) * payRate / 2;

                }

                return basePay + overtimePay;

            }

        }
```

译注 1：该例子中，加班的每分钟工资为工作日每分钟工资（payRate）的一半

译注 2：原文中 pay 方法存在问题，说明如下：

```
    // 基本工资 basePay 不应该直接将工作时间参与计算，如果工作日存在加班的情况，会将加班时间也计入基本工资计算。假设在周一工作 10 小时，假设每分钟 1 元：

    /*

    修改前：

        基本工资 basePay = minutesWorked * payRate=10*60*1=600（不应该将 2 小时加班也计入正常工作时间）

        加班工资 overtimePay = (minutesWorked - MINS_PER_SHIFT) * payRate / 2=2*60*1/2=60

        合计= basePay + overtimePay=660

    修改后：

        基本工资 basePay = MINS_PER_SHIFT * payRate=8*60*1=480（基本工资最高只能按照 8 小时计算）

        加班工资 overtimePay = (minutesWorked - MINS_PER_SHIFT) * payRate / 2=2*60*1/2=60

        合计= basePay + overtimePay=540
```

```
    */
    // 修改后代码：
    int pay(int minutesWorked, int payRate) {
        int basePay = 0;
        int overtimePay;
        switch (this) {
            case SATURDAY:
            case SUNDAY: // Weekend
                overtimePay = minutesWorked * payRate / 2;
                break;
            default: // Weekday
                basePay = minutesWorked <= MINS_PER_SHIFT ?
minutesWorked * payRate : MINS_PER_SHIFT * payRate;
                overtimePay = minutesWorked <= MINS_PER_SHIFT ? 0 :
(minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }
        return basePay + overtimePay;
    }
```

不可否认，这段代码非常简洁，但是从维护的角度来看，它是危险的。假设你向枚举中添加了一个元素，可能是一个表示假期的特殊值，但是忘记向 switch 语句添加相应的 case。这个程序仍然会被编译，但是 pay 方法会把假期默认当做普通工作日并支付工资。

为了使用特定常量方法实现安全地执行工资计算，你必须为每个常量复制加班费计算，或者将计算移动到两个辅助方法中，一个用于工作日，一个用于周末，再从每个常量调用适当的辅助方法。任何一种方法都会导致相当数量的样板代码，极大地降低可读性并增加出错的机会。

用工作日加班计算的具体方法代替发薪日的抽象加班法，可以减少样板。那么只有周末才需要重写该方法。但是这与 switch 语句具有相同的缺点：如果

你在不覆盖 overtimePay 方法的情况下添加了另一天，那么你将默默地继承工作日的计算。

你真正想要的是在每次添加枚举常量时被迫选择加班费策略。幸运的是，有一个很好的方法可以实现这一点。其思想是将加班费计算移到私有嵌套枚举中，并将此策略枚举的实例传递给 PayrollDay 枚举的构造函数。然后 PayrollDay 枚举将加班费计算委托给策略枚举，从而消除了在 PayrollDay 中使用 switch 语句或特定于常量的方法实现的需要。虽然这种模式不如 switch 语句简洁，但它更安全，也更灵活：

```
// The strategy enum pattern
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY,SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);
    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }
    PayrollDay() { this(PayType.WEEKDAY); } // Default

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }
    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ?
0 :(minsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
```

```
                int overtimePay(int minsWorked, int payRate) {
                    return minsWorked * payRate / 2;
                }
            };
            abstract int overtimePay(int mins, int payRate);
            private static final int MINS_PER_SHIFT = 8 * 60;
            int pay(int minsWorked, int payRate) {
                int basePay = minsWorked * payRate;
                return basePay + overtimePay(minsWorked, payRate);
            }
        }
    }
```

译注：上述代码 pay 方法也存将加班时间计入基本工资计算的问题，修改如下：

```
    int pay(int minsWorked, int payRate) {
        int basePay = minsWorked <= MINS_PER_SHIFT ? minsWorked *
payRate : MINS_PER_SHIFT * payRate;
        return basePay + overtimePay(minsWorked, payRate);
    }
```

如果在枚举上实现特定常量的行为时 switch 语句不是一个好的选择，那么它们有什么用呢？**枚举中的 switch 有利于扩展具有特定常量行为的枚举类型。** 例如，假设 Operation 枚举不在你的控制之下，你希望它有一个实例方法来返回每个操作的逆操作。你可以用以下静态方法模拟效果：

```
    // Switch on an enum to simulate a missing method
    public static Operation inverse(Operation op) {
        switch(op) {
            case PLUS: return Operation.MINUS;
```

```
            case MINUS: return Operation.PLUS;
            case TIMES: return Operation.DIVIDE;
            case DIVIDE: return Operation.TIMES;
            default: throw new AssertionError("Unknown op: " + op);
        }
    }
```

如果一个方法不属于枚举类型，那么还应该在你控制的枚举类型上使用这种技术。该方法可能适用于某些特殊用途，但通常如果没有足够的好处，就不值得包含在枚举类型中。

一般来说，枚举在性能上可与 int 常量相比。枚举在性能上有一个小缺点，加载和初始化枚举类型需要花费空间和时间，但是在实际应用中这一点可能不太明显。

那么什么时候应该使用枚举呢？**在需要一组常量时使用枚举，这些常量的成员在编译时是已知的。** 当然，这包括「自然枚举类型」，如行星、星期几和棋子。但是它还包括其他在编译时已知所有可能值的集合，例如菜单上的选项、操作代码和命令行标志。**枚举类型中的常量集没有必要一直保持固定。** 枚举的特性是专门为枚举类型的二进制兼容进化而设计的。

总之，枚举类型相对于 int 常量的优势是毋庸置疑的。枚举更易于阅读、更安全、更强大。许多枚举不需要显式构造函数或成员，但有些枚举则受益于将数据与每个常量关联，并提供行为受数据影响的方法。将多个行为与一个方法关联起来，这样的枚举更少。在这种相对少见的情况下，相对于使用 switch 的枚举，特定常量方法更好。如果枚举常量有一些（但不是全部）共享公共行为，请考虑策略枚举模式。

35 用实例域代替序数

许多枚举天然地与单个 int 值相关联。所有枚举都有一个 ordinal 方法，该方法返回枚举类型中每个枚举常数的数值位置。你可能想从序号中获得一个关联的 int 值：

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
```

```
        SOLO, DUET, TRIO, QUARTET, QUINTET,SEXTET, SEPTET, OCTET,
NONET, DECTET;

        public int numberOfMusicians() { return ordinal() + 1; }

    }
```

虽然这个枚举可以工作，但维护却是噩梦。如果常量被重新排序，numberOfMusicians 方法将被破坏。或者你想添加一个与已经使用过的 int 值相关联的第二个枚举常量，那么你就没有那么幸运了。例如，为双四重奏增加一个常量可能会很好，就像八重奏一样，由八个音乐家组成，但是没有办法做到。

译注：「If you want to add a second enum constant associated with an int value that you've already used」是指每个常量如果不用实例字段的方式，就只能有一个序号值。实例字段可以将自定义的值对应多个常量，例如：SOLO(3), DUET(3), TRIO(3)，可以都设置为序号 3

此外，如果不为所有插入的 int 值添加常量，就不能为 int 值添加常量。例如，假设你想添加一个常量来表示一个由 12 位音乐家组成的三重四重奏。对于 11 位音乐家组成的合奏，由于没有标准术语，因此你必须为未使用的 int 值（11）添加一个虚拟常量。往好的说，这仅仅是丑陋的。如果许多 int 值未使用，则不切实际。幸运的是，这些问题有一个简单的解决方案。**不要从枚举的序数派生与枚举关联的值；而是将其存储在实例字段中：**

```
    public enum Ensemble {

        SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),SEXTET(6),
SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),NONET(9),
DECTET(10),TRIPLE_QUARTET(12);



        private final int numberOfMusicians

        Ensemble(int size) { this.numberOfMusicians = size; }

        public int numberOfMusicians() { return numberOfMusicians; }

    }
```

枚举规范对 ordinal 方法的评价是这样的：「大多数程序员都不会去使用这个方法。它是为基于枚举的通用数据结构（如 EnumSet 和 EnumMap）而设计的」。除非你使用这个数据结构编写代码，否则最好完全避免使用这个方法。

36 用 EnumSet 代替 Bit 位域

如果枚举类型的元素主要在 Set 中使用，传统上使用 int 枚举模式（Item-34），通过不同的 2 平方数为每个常量赋值：

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD = 1 << 0; // 1
    public static final int STYLE_ITALIC = 1 << 1; // 2
    public static final int STYLE_UNDERLINE = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8
    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

这种表示方式称为位字段，允许你使用位运算的 OR 操作将几个常量组合成一个 Set：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位字段表示方式允许使用位运算高效地执行 Set 操作，如并集和交集。但是位字段具有 int 枚举常量所有缺点，甚至更多。当位字段被打印为数字时，它比简单的 int 枚举常量更难理解。没有一种简单的方法可以遍历由位字段表示的所有元素。最后，你必须预测在编写 API 时需要的最大位数，并相应地为位字段（通常是 int 或 long）选择一种类型。一旦选择了一种类型，在不更改 API 的情况下，不能超过它的宽度（32 或 64 位）。

一些使用枚举而不是 int 常量的程序员在需要传递常量集时仍然坚持使用位字段。没有理由这样做，因为存在更好的选择。java.util 包提供 EnumSet 类来有效地表示从单个枚举类型中提取的值集。这个类实现了 Set 接口，提供了所有其他 Set 实现所具有的丰富性、类型安全性和互操作性。但在内部，每个 EnumSet 都表示为一个位向量。如果底层枚举类型有 64 个或更少的元素（大多数都是），则整个 EnumSet 用一个 long 表示，因此其性能与位字段的性能相当。批量操作（如 removeAll 和 retainAll）是使用逐位算法实现的，就像手

动处理位字段一样。但是，你可以避免因手工修改导致产生不良代码和潜在错误：EnumSet 为你完成了这些繁重的工作。

当之前的示例修改为使用枚举和 EnumSet 而不是位字段时。它更短，更清晰，更安全：

```
// EnumSet - a modern replacement for bit fields

public class Text {

    public enum Style { BOLD, ITALIC, UNDERLINE,
STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best

    public void applyStyles(Set<Style> styles) { ... }

}
```

下面是将 EnumSet 实例传递给 applyStyles 方法的客户端代码。EnumSet 类提供了一组丰富的静态工厂，可以方便地创建 Set，下面的代码演示了其中的一个：

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

请注意，applyStyles 方法采用 Set<Style> 而不是 EnumSet<Style>。虽然似乎所有客户端都可能将 EnumSet 传递给该方法，但通常较好的做法是接受接口类型而不是实现类型（Item-64）。这允许特殊的客户端传入其他 Set 实现的可能性。

总之，**因为枚举类型将在 Set 中使用，没有理由用位字段表示它。**EnumSet 类结合了位字段的简洁性和性能，以及 Item-34 中描述的枚举类型的许多优点。EnumSet 的一个真正的缺点是，从 Java 9 开始，它不能创建不可变的 EnumSet，在未来发布的版本中可能会纠正这一点。同时，可以用 Collections.unmodifiableSet 包装 EnumSet，但简洁性和性能将受到影响。

37 用 EnumMap 代替序数索引

偶尔你可能会看到使用 ordinal() 的返回值（Item-35）作为数组或 list 索引的代码。例如，考虑这个简单的类，它表示一种植物：

```
class Plant {
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }
    final String name;
    final LifeCycle lifeCycle;
    Plant(String name, LifeCycle lifeCycle) {
        this.name = name;
        this.lifeCycle = lifeCycle;
    }
    @Override public String toString() {
        return name;
    }
}
```

现在假设你有一个代表花园全部植物的 Plant 数组，你想要列出按生命周期（一年生、多年生或两年生）排列的植物。要做到这一点，你需要构造三个集合，每个生命周期一个，然后遍历整个数组，将每个植物放入适当的集合中：

```
// Using ordinal() to index into an array - DON'T DO THIS!
Set<Plant>[] plantsByLifeCycle =(Set<Plant>[]) new
Set[Plant.LifeCycle.values().length];
for (int i = 0; i < plantsByLifeCycle.length; i++)
    plantsByLifeCycle[i] = new HashSet<>();
for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);

// Print the results
for (int i = 0; i < plantsByLifeCycle.length; i++) {
    System.out.printf("%s: %s%n",
```

```
            Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
    }
```

译注：假设 Plant 数组如下：

```
    Plant[] garden = new Plant[]{
                new Plant("A", LifeCycle.ANNUAL),
                new Plant("B", LifeCycle.BIENNIAL),
                new Plant("C", LifeCycle.PERENNIAL),
                new Plant("D", LifeCycle.BIENNIAL),
                new Plant("E", LifeCycle.PERENNIAL),
    };
```

输出结果为：

```
    ANNUAL: [A]
    PERENNIAL: [E, C]
    BIENNIAL: [B, D]
```

这种技术是有效的，但它充满了问题。因为数组与泛型不兼容（Item-28），所以该程序需要 unchecked 的转换，否则不能顺利地编译。因为数组不知道它的索引表示什么，所以必须手动标记输出。但是这种技术最严重的问题是，当你访问一个由枚举序数索引的数组时，你有责任使用正确的 int 值；int 不提供枚举的类型安全性。如果你使用了错误的值，程序将静默执行错误的操作，如果幸运的话，才会抛出 ArrayIndexOutOfBoundsException。

有一种更好的方法可以达到同样的效果。该数组有效地充当从枚举到值的映射，因此你不妨使用 Map。更具体地说，有一种非常快速的 Map 实现，用于枚举键，称为 java.util.EnumMap。以下就是这个程序在使用 EnumMap 时的样子：

```
    // Using an EnumMap to associate data with an enum
    Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =new
EnumMap<>(Plant.LifeCycle.class);
    for (Plant.LifeCycle lc : Plant.LifeCycle.values())
```

```
            plantsByLifeCycle.put(lc, new HashSet<>());
        for (Plant p : garden)
            plantsByLifeCycle.get(p.lifeCycle).add(p);
        System.out.println(plantsByLifeCycle);
```

这个程序比原来的版本更短，更清晰，更安全，速度也差不多。没有不安全的转换；不需要手动标记输出，因为 Map 的键是能转换为可打印字符串的枚举；在计算数组索引时不可能出错。EnumMap 在速度上与有序索引数组相当的原因是，EnumMap 在内部使用这样的数组，但是它向程序员隐藏了实现细节，将 Map 的丰富的功能和类型安全性与数组的速度结合起来。注意，EnumMap 构造函数接受键类型的 Class 对象：这是一个有界类型标记，它提供运行时泛型类型信息（Item-33）。

通过使用流（Item-45）来管理映射，可以进一步缩短前面的程序。下面是基于流的最简单的代码，它在很大程度上复制了前一个示例的行为：

```
        // Naive stream-based approach - unlikely to produce an EnumMap!
        System.out.println(Arrays.stream(garden).collect(groupingBy(p -> p.lifeCycle)));
```

译注：以上代码需要引入 java.util.stream.Collectors.groupingBy，输出结果如下：

```
        {BIENNIAL=[B, D], ANNUAL=[A], PERENNIAL=[C, E]}
```

这段代码的问题在于它选择了自己的 Map 实现，而实际上它不是 EnumMap，所以它的空间和时间性能与显式 EnumMap 不匹配。要纠正这个问题，可以使用 Collectors.groupingBy 的三参数形式，它允许调用者使用 mapFactory 参数指定 Map 实现：

```
        // Using a stream and an EnumMap to associate data with an enum
        System.out.println(
                Arrays.stream(garden).collect(groupingBy(p -> p.lifeCycle,() -> new EnumMap<>(LifeCycle.class), toSet()))
        );
```

译注：以上代码需要引入 java.util.stream.Collectors.toSet

这种优化在示例程序中不值得去做，但在大量使用 Map 的程序中可能非常重要。

基于流的版本的行为与 EmumMap 版本略有不同。EnumMap 版本总是为每个植物生命周期生成一个嵌套 Map，而基于流的版本只在花园包含具有该生命周期的一个或多个植物时才生成嵌套 Map。例如，如果花园包含一年生和多年生植物，但没有两年生植物，plantsByLifeCycle 的大小在 EnumMap 版本中为 3，在基于流的版本中为 2。

你可能会看到被序数索引（两次！）的数组，序数用于表示两个枚举值的映射。例如，这个程序使用这样的一个数组来映射两个状态到一个状态的转换过程（液体到固体是冻结的，液体到气体是沸腾的，等等）：

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase {
    SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by from-ordinal, cols by to-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null, MELT, SUBLIME },
            { FREEZE, null, BOIL },
            { DEPOSIT, CONDENSE, null }
        };
        // Returns the phase transition from one phase to another
        public static Transition from(Phase from, Phase to) {
            return TRANSITIONS[from.ordinal()][to.ordinal()];
        }
    }
}
```

译注：固体、液体、气体三态，对应的三组变化：融化 MELT，冻结 FREEZE（固态与液态）；沸腾 BOIL，凝固 CONDENSE（液态与气态）；升华 SUBLIME，凝华 DEPOSIT（固态与气态）。

这个程序可以工作，甚至可能看起来很优雅，但外表可能具有欺骗性。就像前面展示的更简单的 garden 示例一样，编译器无法知道序数和数组索引之间的关系。如果你在转换表中出错，或者在修改 Phase 或 Phase.Transition 枚举类型时忘记更新，你的程序将在运行时失败。失败可能是抛出 ArrayIndexOutOfBoundsException、NullPointerException 或（更糟糕的）静默错误行为。并且即使非空项的数目更小，该表的大小也为状态数量的二次方。

同样，使用 EnumMap 可以做得更好。因为每个阶段转换都由一对阶段枚举索引，所以最好将这个关系用 Map 表示，从一个枚举（起始阶段）到第二个枚举（结束阶段）到结果（转换阶段）。与阶段转换相关联的两个阶段最容易捕捉到的是将它们与阶段过渡的 enum 联系起来，这样就可以用来初始化嵌套的 EnumMap：

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
    SOLID, LIQUID, GAS;
    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);
        private final Phase from;
        private final Phase to;
        Transition(Phase from, Phase to) {
            this.from = from;
            this.to = to;
        }
        // Initialize the phase transition map
```

```
        private static final Map<Phase_new, Map<Phase_new, Transition>>
m = Stream.of(values())
                .collect(groupingBy(
                        t -> t.from,
                        () -> new EnumMap<>(Phase_new.class),
                        toMap(t -> t.to, t -> t, (x, y) -> y, () -> new
EnumMap<>(Phase_new.class))
                        )
                );
        public static Transition from(Phase from, Phase to) {
            return m.get(from).get(to);
        }
    }
}
```

初始化阶段变化 Map 的代码有点复杂。Map 的类型是 Map<Phase, Map<Phase, Transition>>，这意味着「从（源）阶段 Map 到（目标）阶段 Map 的转换过程」。这个 Map 嵌套是使用两个收集器的级联序列初始化的。第一个收集器按源阶段对转换进行分组，第二个收集器使用从目标阶段到转换的映射创建一个 EnumMap。第二个收集器 ((x, y) -> y) 中的 merge 函数未使用；之所以需要它，只是因为我们需要指定一个 Map 工厂来获得 EnumMap，而 Collector 提供了伸缩工厂。本书的上一个版本使用显式迭代来初始化阶段转换映射。代码更冗长，但也更容易理解。

译注：第二版中的实现代码如下：

```
// Initialize the phase transition map
private static final Map<Phase, Map<Phase,Transition> m =
    new EnumMap<Phase, Map<Phase ,Transition>>(Phase.class);
    static{
        for (Phase p : Phase. values())
```

```
                m.put(p,new EnumMap<Phase,Transition (Phase.class));
            for (Transition trans : Transition.values() )
                m.get(trans. src).put(trans.dst, trans) ;
        }
    public static Transition from(Phase src, Phase dst) {
        return m.get(src).get(dst);
    }
```

现在假设你想向系统中加入一种新阶段：等离子体，或电离气体。这个阶段只有两个变化：电离，它把气体转为等离子体；去离子作用，把等离子体变成气体。假设要更新基于数组版本的程序，必须向 Phase 添加一个新常量，向 Phase.Transition 添加两个新常量，并用一个新的 16 个元素版本替换原来的数组中的 9 个元素数组。如果你向数组中添加了太多或太少的元素，或者打乱了元素的顺序，那么你就麻烦了：程序将编译，但在运行时将失败。相比之下，要更新基于 EnumMap 的版本，只需将 PLASMA 添加到 Phase 列表中，将 IONIZE(GAS, PLASMA) 和 DEIONIZE(PLASMA, GAS) 添加到 Phase.Transition 中：

```
    // Adding a new phase using the nested EnumMap implementation
    public enum Phase {
        SOLID, LIQUID, GAS, PLASMA;
        public enum Transition {
            MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
            BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
            SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
            IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
            ... // Remainder unchanged
        }
    }
```

这个程序会处理所有其他事情，实际上不会给你留下任何出错的机会。在内部，Map 的映射是用一个数组来实现的，因此你只需花费很少的空间或时间成本就可以获得更好的清晰度、安全性并易于维护。

为了简洁起见，最初的示例使用 null 表示没有状态更改（其中 to 和 from 是相同的）。这不是一个好的方式，可能会在运行时导致 NullPointerException。针对这个问题设计一个干净、优雅的解决方案是非常棘手的，并且生成的程序冗长，以至于它们会偏离条目中的主要内容。

总之，**用普通的序数索引数组是非常不合适的：应使用 EnumMap 代替。** 如果所表示的关系是多维的，则使用 EnumMap<..., EnumMap<...>>。这是一种特殊的基本原则，程序员很少（即使有的话）使用 Enum.ordinal（Item-35）。

38 用接口模拟可扩展的枚举

枚举类型几乎在所有方面都优于本书第一版 [Bloch01] 中描述的 typesafe 枚举模式。从表面上看，有一个与可扩展性有关的例外，它在字节码模式下是可能的，但是语言构造不支持。换句话说，使用字节码模式，可以让一个枚举类型扩展另一个枚举类型；但使用语言特性，则不能这样。这并非偶然。因为在大多数情况下，枚举的可扩展性被证明是一个坏主意，主要在于：扩展类型的元素是基类的实例，而基类的实例却不是扩展类型的元素。而且没有一种好方法可以枚举基类及其扩展的所有元素。最后，可扩展性会使设计和实现的许多方面变得复杂。

也就是说，对于可扩展枚举类型，至少有一个令人信服的用例，即操作码，也称为 opcodes。操作码是一种枚举类型，其元素表示某些机器上的操作，例如 Item-34 中的 Operation 类，它表示简单计算器上的函数。有时候，我们希望 API 的用户提供自己的操作，从而有效地扩展 API 提供的操作集。

幸运的是，有一种很好的方法可以使用枚举类型来实现这种效果。其基本思想是利用枚举类型可以实现任意接口这一事实，为 opcode 类型定义一个接口，并为接口的标准实现定义一个枚举。例如，下面是 Item-34 Operation 类的可扩展版本：

```
// Emulated extensible enum using an interface

public interface Operation {

    double apply(double x, double y);
```

```
}
public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override
    public String toString() {
        return symbol;
    }
}
```

枚举类型（BasicOperation）是不可扩展的，而接口类型（Operation）是可扩展的，它是用于在 API 中表示操作的接口类型。你可以定义另一个实现此接口的枚举类型，并使用此新类型的实例代替基类型。例如，假设你想定义前面

显示的操作类型的扩展，包括求幂和余数操作。你所要做的就是写一个枚举类型，实现操作接口：

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };
    private final String symbol;
    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override
    public String toString() {
        return symbol;
    }
}
```

现在可以在任何可以使用 Operation 的地方使用新 Operation，前提是编写的 API 采用接口类型（Operation），而不是实现（BasicOperation）。注意，不必像在具有特定于实例的方法实现的非可扩展枚举中那样在枚举中声明抽象

apply 方法（第 162 页）。这是因为抽象方法（apply）是接口（Operation）的成员。

译注：示例如下

```
public static void main(String[] args) {
    Operation op = BasicOperation.DIVIDE;
    System.out.println(op.apply(15, 3));
    op=ExtendedOperation.EXP;
    System.out.println(op.apply(2,5));
}
```

不仅可以在需要「基枚举」的任何地方传递「扩展枚举」的单个实例，还可以传入整个扩展枚举类型，并在基类型的元素之外使用或替代基类型的元素。例如，这里是 163 页测试程序的一个版本，它执行了前面定义的所有扩展操作：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}
private static <T extends Enum<T> & Operation> void test(Class<T>
opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",x, op, y, op.apply(x, y));
}
```

注意，扩展 Operation 类型（ExtendedOperation.class）的 class 字面量是从 main 传递到 test 的，以描述扩展 Operation 类型的 Set。class 字面量用作有界类型标记（Item-33）。诚然，opEnumType 参数的复杂声明（<T extends Enum<T> & Operation> Class<T>）确保类对象同时表示枚举和 Operation 的子类型，而这正是遍历元素并执行与每个元素相关的操作所必需的。

第二个选择是传递一个 Collection<? extends Operation>，它是一个有界通配符类型（Item-31），而不是传递一个类对象：

```
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        test(Arrays.asList(ExtendedOperation.values()), x, y);
    }
    private static void test(Collection<? extends Operation> opSet,double x, double y) {
        for (Operation op : opSet)
            System.out.printf("%f %s %f = %f%n",x, op, y, op.apply(x, y));
    }
```

生成的代码稍微不那么复杂，test 方法稍微灵活一些：它允许调用者组合来自多个实现类型的操作。另一方面，放弃了在指定操作上使用 EnumSet（Item-36）和 EnumMap（Item-37）的能力。

在运行命令行参数 4 和 2 时，前面显示的两个程序都将产生这个输出：

```
    4.000000 ^ 2.000000 = 16.000000
    4.000000 % 2.000000 = 0.000000
```

使用接口来模拟可扩展枚举的一个小缺点是实现不能从一个枚举类型继承到另一个枚举类型。如果实现代码不依赖于任何状态，则可以使用默认实现（Item-20）将其放置在接口中。在我们的 Operation 示例中，存储和检索与操作相关的符号的逻辑必须在 BasicOperation 和 ExtendedOperation 中复制。在这种情况下，这并不重要，因为复制的代码非常少。如果有大量的共享功能，可以将其封装在 helper 类或静态 helper 方法中，以消除代码重复。

此项中描述的模式在 Java 库中使用。例如，java.nio.file.LinkOption 枚举类型实现 CopyOption 和 OpenOption 接口。

总之，虽然你不能编写可扩展枚举类型，但是你可以通过编写接口来模拟它，以便与实现该接口的基本枚举类型一起使用。这允许客户端编写自己的枚

举（或其他类型）来实现接口。假设 API 是根据接口编写的，那么这些类型的实例可以在任何可以使用基本枚举类型的实例的地方使用。

39 注解优于命名模式

从历史上看，使用命名模式来标明某些程序元素需要工具或框架特殊处理的方式是很常见的。例如，在版本 4 之前，JUnit 测试框架要求其用户通过以字符 test [Beck04] 开头的名称来指定测试方法。这种技术是有效的，但是它有几个很大的缺点。首先，排版错误会导致没有提示的失败。例如，假设你意外地将一个测试方法命名为 tsetSafetyOverride，而不是 testSafetyOverride。JUnit 3 不会报错，但它也不会执行测试，这导致一种正确执行了测试的假象。

命名模式的第二个缺点是，无法确保只在相应的程序元素上使用它们。例如，假设你调用了一个类 TestSafetyMechanisms，希望 JUnit 3 能够自动测试它的所有方法，而不管它们的名称是什么。同样，JUnit 3 不会报错，但它也不会执行测试。

命名模式的第三个缺点是，它们没有提供将参数值与程序元素关联的好方法。例如，假设你希望支持只有在抛出特定异常时才成功的测试类别。异常类型本质上是测试的一个参数。你可以使用一些精心设计的命名模式，将异常类型名称编码到测试方法名称中，但这样的代码将不好看且脆弱（Item-62）。编译器将无法检查这些用于命名异常的字符串是否确实执行了。如果指定的类不存在或不是异常，则在运行测试之前不会被发现。

注解 [JLS, 9.7] 很好地解决了所有这些问题，JUnit 从版本 4 开始就采用了它们。在本条目中，我们将编写自己的示例测试框架来展示注解是如何工作的。假设你希望定义注解类型，以指定自动运行的简单测试，并在抛出异常时失败。下面是这种名为 Test 的注解类型的概貌：

```
// Marker annotation type declaration

import java.lang.annotation.*;

/**

* Indicates that the annotated method is a test method.

* Use only on parameterless static methods.

*/

@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)

public @interface Test {

}
```

Test 注解类型的声明本身带有 Retention 注解和 Target 注解。这种注解类型声明上的注解称为元注解。@Retention(RetentionPolicy.RUNTIME) 元注解表明测试注解应该在运行时保留。没有它，测试工具将无法识别测试注解。@Target.get(ElementType.METHOD) 元注解表明测试注解仅对方法声明合法：它不能应用于类声明、字段声明或其他程序元素。

Test 注解声明之前的代码注释是这么描述的:「Use only on parameterless static methods.（只对无参数的静态方法使用）」如果编译器能够强制执行这一点，那就太好了，但是它不能，除非你编写代码注释处理器来执行。有关此主题的更多信息，请参阅 javax.annotation.processing 的文档。在没有这样的代码注释处理程序的情况下，如果你将 Test 注解放在实例方法的声明上，或者放在带有一个或多个参数的方法上，测试程序仍然会编译，让测试工具在运行时处理。

下面是 Test 注解实际使用时的样子。它被称为标记注解，因为它没有参数，只是对带注解的元素进行「标记」。如果程序员拼错 Test 或将 Test 注解应用于除方法声明之外的程序元素，程序将无法编译：

```
// Program containing marker annotations

public class Sample {

    @Test

    public static void m1() { } // Test should pass

    public static void m2() { }

    @Test

    public static void m3() { // Test should fail

        throw new RuntimeException("Boom");

    }

    public static void m4() { }

    @Test

    public void m5() { } // INVALID USE: nonstatic method
```

```
    public static void m6() { }
    @Test
    public static void m7() { // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

Sample 类有 7 个静态方法，其中 4 个被注解为 Test。其中两个方法 m3 和 m7 抛出异常，另外两个 m1 和 m5 没有抛出异常。但是，不抛出异常的带注解的方法 m5 是一个实例方法，因此它不是注解的有效使用。总之，Sample 包含四个测试：一个通过，两个失败，一个无效。没有使用 Test 注释的四个方法将被测试工具忽略。

Test 注解对 Sample 类的语义没有直接影响。它们仅用于向相关程序提供信息。更普遍的是，注解不会改变被注解代码的语义，而是通过工具（就像如下这个简单的 RunTests 类）对其进行特殊处理：

```
// Program to process marker annotations
import java.lang.reflect.*;
public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
```

```
                        m.invoke(null);
                        passed++;
                    } catch (InvocationTargetException wrappedExc) {
                        Throwable exc = wrappedExc.getCause();
                        System.out.println(m + " failed: " + exc);
                    } catch (Exception exc) {
                        System.out.println("Invalid @Test: " + m);
                    }
                }
            }
            System.out.printf("Passed: %d, Failed: %d%n",passed, tests - passed);
            }
    }
```

test runner 工具在命令行上接受一个完全限定的类名，并通过调用 Method.invoke 以反射方式运行类的所有带测试注解的方法。isAnnotationPresent 方法告诉工具要运行哪些方法。如果测试方法抛出异常，反射工具将其封装在 InvocationTargetException 中。该工具捕获这个异常并打印一个失败报告，其中包含测试方法抛出的原始异常，该异常是用 getCause 方法从 InvocationTargetException 提取的。

如果通过反射调用测试方法时抛出除 InvocationTargetException 之外的任何异常，则表明在编译时存在未捕获的 Test 注解的无效用法。这些用途包括实例方法的注解、带有一个或多个参数的方法的注解或不可访问方法的注解。测试运行程序中的第二个 catch 块捕获这些 Test 使用错误并打印对应的错误消息。如果在 Sample 上运行 RunTests，输出如下：

```
public static void Sample.m3() failed: RuntimeException: Boom
Invalid @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

现在让我们添加一个只在抛出特定异常时才成功的测试支持。我们需要一个新的注解类型：

```
// Annotation type with a parameter

import java.lang.annotation.*;

/**

* Indicates that the annotated method is a test method that

* must throw the designated exception to succeed.

*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface ExceptionTest {

    Class<? extends Throwable> value();

}
```

这个注解的参数类型是 Class<? extends Throwable>，这个通配符类型确实很复杂。在英语中，它的意思是「某个扩展自 Throwable 的类的 Class 对象」，它允许注解的用户指定任何异常（或错误）类型。这种用法是有界类型令牌（Item-33）的一个示例。下面是这个注解在实际应用时的样子。注意，类的字面量被用作注解参数的值：

```
// Program containing annotations with a parameter

public class Sample2 {

    @ExceptionTest(ArithmeticException.class)

    public static void m1() { // Test should pass

        int i = 0;

        i = i / i;

    }
```

```
        @ExceptionTest(ArithmeticException.class)
        public static void m2() { // Should fail (wrong exception)
            int[] a = new int[0];
            int i = a[1];
        }

        @ExceptionTest(ArithmeticException.class)
        public static void m3() { } // Should fail (no exception)
    }
```

现在让我们修改 test runner 工具来处理新的注解。向 main 方法添加以下代码：

```
    if (m.isAnnotationPresent(ExceptionTest.class)) {
        tests++;
        try {
            m.invoke(null);
            System.out.printf("Test %s failed: no exception%n", m);
        } catch (InvocationTargetException wrappedEx) {
            Throwable exc = wrappedEx.getCause();
            Class<? extends Throwable> excType =m.getAnnotation(Exception
Test.class).value();
            if (excType.isInstance(exc)) {
                passed++;
            } else {
                System.out.printf("Test %s failed: expected %s, got %s%n",
m, excType.getName(), exc);
```

```
                }
            }
            catch (Exception exc) {
                System.out.println("Invalid @Test: " + m);
            }
        }
```

这段代码与我们用来处理 Test 注解的代码类似，只有一个不同：这段代码提取注解参数的值，并使用它来检查测试抛出的异常是否是正确的类型。这里没有显式的强制类型转换，因此没有 ClassCastException 的危险。编译的测试程序保证其注解参数表示有效的异常类型，但有一点需要注意：如果注解参数在编译时有效，但表示指定异常类型的类文件在运行时不再存在，那么测试运行程序将抛出 TypeNotPresentException。

进一步来看我们的异常测试示例，如果它抛出几个指定异常中的任意一个，那么可以认为测试通过了。注解机制具有一种工具，可以轻松地支持这种用法。假设我们将 ExceptionTest 注解的参数类型更改为一个 Class 对象数组：

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

注解中数组参数的语法是灵活的。它针对单元素数组进行了优化。前面的 ExceptionTest 注解对于 ExceptionTest 的新数组参数版本仍然有效，并且可以生成单元素数组。要指定一个多元素数组，用花括号包围元素，并用逗号分隔它们：

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,NullPointerException.class })
public static void doublyBad() {
```

```
        List<String> list = new ArrayList<>();
        // The spec permits this method to throw either
        // IndexOutOfBoundsException or NullPointerException
        list.addAll(5, null);
    }
```

修改测试运行器工具来处理 ExceptionTest 的新版本是相当简单的。这段代码替换了原来的版本：

```
    if (m.isAnnotationPresent(ExceptionTest.class)) {
        tests++;
        try {
            m.invoke(null);
            System.out.printf("Test %s failed: no exception%n", m);
        } catch (Throwable wrappedExc) {
            Throwable exc = wrappedExc.getCause();
            int oldPassed = passed;
            Class<? extends Exception>[] excTypes =m.getAnnotation(ExceptionTest.class).value();
            for (Class<? extends Exception> excType : excTypes) {
                if (excType.isInstance(exc)) {
                    passed++;
                    break;
                }
            }
            if (passed == oldPassed)
```

```
                System.out.printf("Test %s failed: %s %n", m, exc);
            }
        }
```

在 Java 8 中，还有另一种方法可以执行多值注解。你可以在注解声明上使用 @Repeatable 元注解，以表明注解可以重复地应用于单个元素，而不是使用数组参数来声明注解类型。这个元注解只接受一个参数，这个参数是包含注解类型的类对象，它的唯一参数是注解类型的数组 [JLS, 9.6.3]。如果我们对 ExceptionTest 注解采用这种方法，那么注解声明是这样的。注意，包含的注解类型必须使用适当的 Retention 注解和 Target 注解，否则声明将无法编译：

```
// Repeatable annotation type
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer {
    ExceptionTest[] value();
}
```

下面是使用重复注解代替数组值注解的 doublyBad 测试：

```
// Code containing a repeated annotation
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

处理可重复注解需要小心。重复的注解生成包含注解类型的合成注解。getAnnotationsByType 方法掩盖了这一事实，可以用于访问可重复注解类型的重复和非重复注解。但是 isAnnotationPresent 明确指出，重复的注解不是注解类型，而是包含注解的类型。如果一个元素具有某种类型的重复注解，并且你使用 isAnnotationPresent 方法检查该元素是否具有该类型的注解，你将发现它没有。因此，使用此方法检查注解类型的存在将导致你的程序无声地忽略重复的注解。类似地，使用此方法检查包含的注解类型将导致程序无声地忽略不重复的注解。要使用 isAnnotationPresent 检测重复和非重复注解，需要同时检查注解类型及其包含的注解类型。下面是我们的 RunTests 程序的相关部分在修改为使用 ExceptionTest 注解的可重复版本时的样子：

```
// Processing repeatable annotations

if (m.isAnnotationPresent(ExceptionTest.class)|| m.isAnnotationPresent(ExceptionTestContainer.class)) {

    tests++;

    try {

        m.invoke(null);

        System.out.printf("Test %s failed: no exception%n", m);

    } catch (Throwable wrappedExc) {

        Throwable exc = wrappedExc.getCause();

        int oldPassed = passed;

        ExceptionTest[] excTests =m.getAnnotationsByType(ExceptionTest.class);

        for (ExceptionTest excTest : excTests) {

            if (excTest.value().isInstance(exc)) {

                passed++;

                break;

            }

        }

        if (passed == oldPassed)
```

```
                System.out.printf("Test %s failed: %s %n", m, exc);
            }
        }
```

添加可重复注解是为了提高源代码的可读性，源代码在逻辑上将同一注解类型的多个实例应用于给定的程序元素。如果你觉得它们增强了源代码的可读性，那么就使用它们，但是请记住，在声明和处理可重复注解方面有更多的样板，并且处理可重复注解很容易出错。

本条目中的测试框架只是一个示例，但是它清楚地展示了注解相对于命名模式的优势，并且它只涉及到你可以使用它们做什么。如果你编写的工具要求程序员向源代码中添加信息，请定义适当的注解类型。如果可以使用注解，那么就没有理由使用命名模式。

也就是说，除了 toolsmiths 之外，大多数程序员不需要定义注解类型。但是所有程序员都应该使用 Java 提供的预定义注解类型（Item-40 和 Item-27）。另外，考虑使用 IDE 或静态分析工具提供的注解。这些注解可以提高这些工具提供的诊断信息的质量。但是，请注意，这些注解还没有标准化，因此，如果你切换了工具或出现了标准，那么你可能需要做一些工作。

40 统一使用 Override 注解

The Java libraries contain several annotation types. For the typical programmer, the most important of these is @Override. This annotation can be used only on method declarations, and it indicates that the annotated method declaration overrides a declaration in a supertype. If you consistently use this annotation, it will protect you from a large class of nefarious bugs. Consider this program, in which the class Bigram represents a bigram, or ordered pair of letters:

Java 库包含多种注释类型。对于典型的程序员，其中最重要的是@Override。此注释只能用于方法声明，它表示带注释的方法声明覆盖了超类型中的声明。 如果你一直使用这个注释，它将保护你免受一大堆邪恶的错误。 考虑这个程序，其中类 Bigram 代表一个二元组或一对有序的字母：：

```
// Can you spot the bug?
public class Bigram {
private final char first;
```

```
private final char second;

public Bigram(char first, char second) {

this.first = first;

this.second = second;

} public boolean equals(Bigram b) {

return b.first == first && b.second == second;

} public int hashCode() {

return 31 * first + second;

}

public static void main(String[] args) {

Set<Bigram> s = new HashSet<>();

for (int i = 0; i < 10; i++)

for (char ch = 'a'; ch <= 'z'; ch++)

s.add(new Bigram(ch, ch));

System.out.println(s.size());

}}
```

The main program repeatedly adds twenty-six bigrams, each consisting of two identical lowercase letters, to a set. Then it prints the size of the set. You might expect the program to print 26, as sets cannot contain duplicates. If you try running the program, you'll find that it prints not 26 but 260. What is wrong with it?

主程序重复添加二十六个双字母组，每个双字母组由两个相同的小写字母组成。 然后它打印集的大小。 您可能希望程序打印 26，因为集合不能包含重复项。 如果您尝试运行该程序，您会发现它打印的不是 26 而是 260.它有什么问题？

Clearly, the author of the Bigram class intended to override the equals method (Item 10) and even remembered to override hashCode in tandem (Item 11). Unfortunately, our hapless programmer failed to override equals, overloading it instead (Item 52). To override Object.equals, you must define an equals method whose parameter is of

type Object, but the parameter of Bigram's equals method is not of type Object, so Bigram inherits the equals method from Object. This equals method tests for object identity, just like the == operator. Each of the ten copies of each bigram is distinct from the other nine, so they are deemed unequal by Object.equals, which explains why the program prints 260.

Luckily, the compiler can help you find this error, but only if you help it by telling it that you intend to override Object.equals. To do this, annotate Bigram.equals with @Override, as shown here:

显然，Bigram 类的作者打算覆盖 equals 方法（第 10 项），甚至记得在串联中重写 hashCode（第 11 项）。 不幸的是，我们倒霉的程序员未能覆盖等于，而是重载它（第 52 项）。 要覆盖 Object.equals，必须定义一个 equals 方法，其参数类型为 Object，但 Bigram 的 equals 方法的参数不是 Object 类型，因此 Bigram 从 Object 继承 equals 方法。 这等于对象标识的方法测试，就像==运算符一样。每个二元组的十个副本中的每一个都与其他九个不同，因此它们被 Object.equals 视为不相等，这解释了程序打印 260 的原因。

幸运的是，编译器可以帮助您找到此错误，但只有在您通过告诉它您打算覆盖 Object.equals 来帮助它时。 为此，使用@Override 注释 Bigram.equals，如下所示：

```
@Override public boolean equals(Bigram b) {

return b.first == first && b.second == second;

}
```

If you insert this annotation and try to recompile the program, the compiler will generate an error message like this:

如果插入此批注并尝试重新编译该程序，编译器将生成如下错误消息：

```
Bigram.java:10: method does not override or implement a method from a supertype

@Override public boolean equals(Bigram b) {

^
```

You will immediately realize what you did wrong, slap yourself on the forehead, and replace the broken equals implementation with a correct one (Item 10):

你会立即意识到你做错了什么，把自己拍在额头上，用正确的方法替换破碎的等于实现（第 10 项）：

```
@Override public boolean equals(Object o) {

if (!(o instanceof Bigram))

return false;

Bigram b = (Bigram) o;

return b.first == first && b.second == second;

}
```

Therefore, you should **use the Override annotation on every method declaration that you believe to override a superclass declaration.** There is one minor exception to this rule. If you are writing a class that is not labeled abstract and you believe that it overrides an abstract method in its superclass, you needn't bother putting the Override annotation on that method. In a class that is not declared abstract, the compiler will emit an error message if you fail to override an abstract superclass method. However, you might wish to draw attention to all of the methods in your class that override superclass methods, in which case you should feel free to annotate these methods too. Most IDEs can be set to insert Override annotations automatically when you elect to override a method.

因此，您应该在您认为覆盖超类声明的每个方法声明上使用覆盖注释。 这条规则有一个小例外。 如果您正在编写一个未标记为抽象的类，并且您认为它覆盖了其超类中的抽象方法，则无需在该方法上放置 Override 注释。 在未声明为 abstract 的类中，如果未能覆盖抽象超类方法，编译器将发出错误消息。 但是，您可能希望引起对类中覆盖超类方法的所有方法的注意，在这种情况下，您也可以随意注释这些方法。 当您选择覆盖方法时，可以将大多数 IDE 设置为自动插入覆盖注释。

Most IDEs provide another reason to use the Override annotation consistently. If you enable the appropriate check, the IDE will generate a warning if you have a method that doesn't have an Override annotation but does override a superclass method. If you use the Override annotation consistently, these warnings will alert you to unintentional overriding. They complement the compiler's error messages, which alert you to unintentional failure to override. Between the IDE and the compiler, you can be sure that you're overriding methods everywhere you want to and nowhere else.

大多数 IDE 提供了一致使用覆盖注释的另一个原因。 如果启用相应的检查，如果您的方法没有覆盖注释但覆盖超类方法，则 IDE 将生成警告。 如果您始终使用“覆盖”注释，则这些警告将提醒您无意覆盖。 它们补充了编译器的错误消息，提醒您无意中无法覆盖。 在 IDE 和编译器之间，您可以确保在任何地方覆盖您想要的方法，而不是其他任何方法。

The Override annotation may be used on method declarations that override declarations from interfaces as well as classes. With the advent of default methods, it is good practice to use Override on concrete implementations of interface methods to ensure that the signature is correct. If you know that an interface does not have default methods, you may choose to omit Override annotations on concrete implementations of interface methods to reduce clutter.

覆盖注释可用于覆盖接口和类的声明的方法声明。 随着默认方法的出现，优良作法是在接口方法的具体实现上使用 Override 来确保签名是正确的。 如果您知道接口没有默认方法，则可以选择在接口方法的具体实现上省略 Override annotations 以减少混乱。

In an abstract class or an interface, however, it is worth annotating all methods that you believe to override superclass or superinterface methods, whether concrete or abstract. For example, the Set interface adds no new methods to the Collection interface, so it should include Override annotations on all of its method declarations to ensure that it does not accidentally add any new methods to the Collection interface.

但是，在抽象类或接口中，值得注释您认为覆盖超类或超接口方法的所有方法，无论是具体还是抽象。 例如，Set 接口不向 Collection 接口添加新方法，因此它应该在其所有方法声明中包含 Override annotations，以确保它不会意外地向 Collection 接口添加任何新方法。

In summary, the compiler can protect you from a great many errors if you use the Override annotation on every method declaration that you believe to override a supertype declaration, with one exception. In concrete classes, you need not annotate methods that you believe to override abstract method declarations (though it is not harmful to do so).

总之，如果在您认为覆盖超类型声明的每个方法声明上使用覆盖注释，编译器可以保护您免受大量错误的影响，但有一个例外。 在具体的类中，您不需要注释您认为覆盖抽象方法声明的方法（尽管这样做没有害处）。

41 用标记接口定义类型

A marker interface is an interface that contains no method declarations but merely designates (or “marks”) a class that implements the interface as having some property. For example, consider the Serializable interface (Chapter 12). By implementing this interface, a class indicates that its instances can be written to an ObjectOutputStream (or “serialized”).

标记接口是一个接口，它不包含任何方法声明，只是指定（或“标记”）一个实现接口具有某些属性的类。 例如，考虑 Serializable 接口（第 12 章）。 通过实现此接口，类指示其实例可以写入 ObjectOutputStream（或“序列化”）。

You may hear it said that marker annotations (Item 39) make marker interfaces obsolete. This assertion is incorrect. Marker interfaces have two advantages over marker annotations. First and foremost, **marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not.** The existence of a marker interface type allows you to catch errors at compile time that you couldn’t catch until runtime if you used a marker annotation.

您可能会听到它说标记注释（第 39 项）使标记接口过时。 这个断言是不正确的。 标记接口与标记注释相比具有两个优点。 首先，标记接口定义由标记类的实例实现的类型; 标记注释没有。 标记接口类型的存在允许您在编译时捕获错误，如果使用标记注释，则在运行时无法捕获这些错误。

Java’s serialization facility (Chapter 6) uses the Serializable marker interface to indicate that a type is serializable. The ObjectOutputStream.writeObject method, which serializes the object that is passed to it, requires that its argument be serializable. Had the argument of this method been of type Serializable, an attempt to serialize an inappropriate object would have been detected at compile time (by type checking). Compile-time error detection is the intent of marker interfaces, but unfortunately, the ObjectOutputStream.write API does not take advantage of the Serializable interface: its argument is declared to be of type Object, so attempts to serialize an unserializable object won’t fail until runtime.

Java 的序列化工具（第 6 章）使用 Serializable 标记接口来指示类型是可序列化的。 ObjectOutputStream.writeObject 方法（序列化传递给它的对象）要求其参数可序列化。 如果此方法的参数是 Serializable 类型，则在编译时（通过类型检查）将检测到序列化不适当对象的尝试。 编译时错误检测是标记接口的目的，但不幸的是，ObjectOutputStream.write API 没有利用 Serializable 接口：它

的参数被声明为 Object 类型，因此尝试序列化不可序列化的对象不会失败 直到运行时

Another advantage of marker interfaces over marker annotations is that they can be targeted more precisely. If an annotation type is declared with target ElementType.TYPE, it can be applied to any class or interface. Suppose you have a marker that is applicable only to implementations of a particular interface. If you define it as a marker interface, you can have it extend the sole interface to which it is applicable, guaranteeing that all marked types are also subtypes of the sole interface to which it is applicable.

标记接口相对于标记注释的另一个优点是可以更精确地定位它们。 如果使用目标 ElementType.TYPE 声明注释类型，则可以将其应用于任何类或接口。 假设您有一个仅适用于特定接口实现的标记。 如果将其定义为标记接口，则可以使其扩展到适用的唯一接口，从而保证所有标记类型也是适用的唯一接口的子类型。

Arguably, the Set interface is just such a restricted marker interface. It is applicable only to Collection subtypes, but it adds no methods beyond those defined by Collection. It is not generally considered to be a marker interface because it refines the contracts of several Collection methods, including add, equals, and hashCode. But it is easy to imagine a marker interface that is applicable only to subtypes of some particular interface and does not refine the contracts of any of the interface's methods. Such a marker interface might describe some invariant of the entire object or indicate that instances are eligible for processing by a method of some other class (in the way that the Serializable interface indicates that instances are eligible for processing by ObjectOutputStream).

可以说，Set 接口就是这样一个受限制的标记接口。 它仅适用于 Collection 子类型，但它不会添加除 Collection 定义的方法之外的任何方法。 它通常不被认为是标记接口，因为它改进了几个 Collection 方法的契约，包括 add，equals 和 hashCode。 但很容易想象一个标记接口只适用于某些特定接口的子类型，并且不会细化任何接口方法的契约。 这样的标记接口可能描述整个对象的某些不变量，或者指示实例有资格通过某个其他类的方法进行处理（以 Serializable 接口指示实例有资格由 ObjectOutputStream 处理的方式）。

The chief advantage of marker annotations over marker interfaces is that they are part of the larger annotation facility. Therefore, marker annotations allow for consistency in annotation-based frameworks.

标记注释优于标记接口的主要优点是它们是较大注释工具的一部分。 因此，标记注释允许基于注释的框架的一致性。

So when should you use a marker annotation and when should you use a marker interface? Clearly you must use an annotation if the marker applies to any program element other than a class or interface, because only classes and interfaces can be made to implement or extend an interface. If the marker applies only to classes and interfaces, ask yourself the question “Might I want to write one or more methods that accept only objects that have this marking?” If so, you should use a marker interface in preference to an annotation. This will make it possible for you to use the interface as a parameter type for the methods in question, which will result in the benefit of compile-time type checking. If you can convince yourself that you’ll never want to write a method that accepts only objects with the marking, then you’re probably better off using a marker annotation. If, additionally, the marking is part of a framework that makes heavy use of annotations, then a marker annotation is the clear choice.

那么什么时候应该使用标记注释？何时应该使用标记界面？显然，如果标记适用于除类或接口之外的任何程序元素，则必须使用注释，因为只能使用类和接口来实现或扩展接口。如果标记仅适用于类和接口，请问自己“我是否要编写一个或多个只接受具有此标记的对象的方法？”的问题。如果是这样，您应该优先使用标记接口作为注释。这将使您可以将接口用作相关方法的参数类型，这将带来编译时类型检查的好处。如果你可以说服自己，你永远不想写一个只接受带有标记的对象的方法，那么你最好使用标记注释。此外，如果标记是大量使用注释的框架的一部分，那么标记注释是明确的选择。

In summary, marker interfaces and marker annotations both have their uses. If you want to define a type that does not have any new methods associated with it, a marker interface is the way to go. If you want to mark program elements other than classes and interfaces or to fit the marker into a framework that already makes heavy use of annotation types, then a marker annotation is the correct choice. **If you find yourself writing a marker annotation type whose target is ElementType.TYPE, take the time to figure out whether it really should be an annotation type or whether a marker interface would be more appropriate.**

总之，标记接口和标记注释都有其用途。 如果要定义一个没有与之关联的新方法的类型，则可以使用标记接口。 如果要标记类和接口以外的程序元素，或者将标记放入已经大量使用注释类型的框架中，则标记注释是正确的选择。 如果

您发现自己编写了一个标记注释类型，其目标是 ElementType.TYPE，请花点时间确定它是否真的应该是注释类型，或者标记接口是否更合适。

In a sense, this item is the inverse of Item 22, which says, “If you don’t want to define a type, don’t use an interface.” To a first approximation, this item says, “If you do want to define a type, do use an interface.”

从某种意义上说，这个项目与第 22 项相反，它说：“如果你不想定义一个类型，就不要使用界面。”对于第一个近似值，这个项目说：“如果你想要 要定义一个类型，请使用一个接口。“

第七章 Lambdas 表达式 and 流 Streams

在 Java 8 中，添加了函数接口，lambdas 和方法引用，以便更容易地创建函数对象。 流 API 与这些语言更改一起添加，以便为处理数据元素序列提供库支持。在本章中，我们将讨论如何充分利用这些设施。

42 Lambda 表达式优于匿名类

Historically, interfaces (or, rarely, abstract classes) with a single abstract method were used as function types. Their instances, known as function objects, represent functions or actions. Since JDK 1.1 was released in 1997, the primary means of creating a function object was the anonymous class (Item 24). Here’s a code snippet to sort a list of strings in order of length, using an anonymous class to create the sort’s comparison function (which imposes the sort order):

从历史上看，使用单个抽象方法的接口（或很少是抽象类）被用作函数类型。 它们的实例称为函数对象，代表函数或动作。 自 JDK 1.1 于 1997 年发布以来，创建函数对象的主要方法是匿名类（第 24 项）。 这是一个代码片段，用于按长度顺序对字符串列表进行排序，使用匿名类创建排序的比较函数（强制排序顺序）：

```
// Anonymous class instance as a function object - obsolete!
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
```

```
            return Integer.compare(s1.length(), s2.length());
        }
    });
```

Anonymous classes were adequate for the classic objected-oriented design patterns requiring function objects, notably the Strategy pattern [Gamma95]. The Comparator interface represents an abstract strategy for sorting; the anonymous class above is a concrete strategy for sorting strings. The verbosity of anonymous classes, however, made functional programming in Java an unappealing prospect.

匿名类适用于需要功能对象的经典的面向对象的设计模式，特别是策略模式[Gamma95]。 Comparator 接口表示用于排序的抽象策略；上面的匿名类是排序字符串的具体策略。 然而，匿名类的冗长使得 Java 中的函数式编程成为一个没有吸引力的前景。

In Java 8, the language formalized the notion that interfaces with a single abstract method are special and deserve special treatment. These interfaces are now known as functional interfaces, and the language allows you to create instances of these interfaces using lambda expressions, or lambdas for short. Lambdas are similar in function to anonymous classes, but far more concise. Here's how the code snippet above looks with the anonymous class replaced by a lambda. The boilerplate is gone, and the behavior is clearly evident:

在 Java 8 中，该语言形式化了这样一种概念，即使用单一抽象方法的接口是特殊的，值得特别对待。 这些接口现在称为功能接口，该语言允许您使用 lambda 表达式或简称 lambdas 创建这些接口的实例。 Lambdas 在功能上与匿名类相似，但更加简洁。 以下是上面的代码片段如何将匿名类替换为 lambda。 样板消失了，行为很明显：

```
// Lambda expression as function object (replaces anonymous class)
Collections.sort(words,(s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Note that the types of the lambda (Comparator), of its parameters (s1 and s2, both String), and of its return value (int) are not present in the code. The compiler deduces these types from context, using a process known as type inference. In some cases, the compiler won't be able to determine the types, and you'll have to specify them. The rules for type inference are complex: they take up an entire chapter in the JLS [JLS, 18]. Few programmers understand these rules in detail, but that's OK. **Omit the types of all lambda parameters unless their presence makes your**

program clearer. If the compiler generates an error telling you it can't infer the type of a lambda parameter, then specify it. Sometimes you may have to cast the return value or the entire lambda expression, but this is rare.

请注意，lambda（Comparator）的类型，其参数（s1 和 s2，两个 String）及其返回值（int）的类型不在代码中。 编译器使用称为类型推断的过程从上下文中推导出这些类型。 在某些情况下，编译器将无法确定类型，您必须指定它们。 类型推断的规则很复杂：它们占据了 JLS 的整个章节[JLS，18]。 很少有程序员详细了解这些规则，但这没关系。 省略所有 lambda 参数的类型，除非它们的存在使您的程序更清晰。 如果编译器生成错误，告诉您无法推断 lambda 参数的类型，请指定它。 有时您可能必须转换返回值或整个 lambda 表达式，但这种情况很少见。

One caveat should be added concerning type inference. Item 26 tells you not to use raw types, Item 29 tells you to favor generic types, and Item 30 tells you to favor generic methods. This advice is doubly important when you're using lambdas, because the compiler obtains most of the type information that allows it to perform type inference from generics. If you don't provide this information, the compiler will be unable to do type inference, and you'll have to specify types manually in your lambdas, which will greatly increase their verbosity. By way of example, the code snippet above won't compile if the variable words is declared to be of the raw type List instead of the parameterized type List.

关于类型推断，应该添加一个警告。 第 26 项告诉您不要使用原始类型，第 29 项告诉您支持泛型类型，第 30 项告诉您支持通用方法。 当你使用 lambdas 时，这个建议是非常重要的，因为编译器获得了允许它从泛型执行类型推断的大多数类型信息。 如果您不提供此信息，编译器将无法进行类型推断，您必须在 lambdas 中手动指定类型，这将大大增加它们的详细程度。 举例来说，如果变量词被声明为原始类型 List 而不是参数化类型 List，则上面的代码片段将不会编译。

Incidentally, the comparator in the snippet can be made even more succinct if a comparator construction method is used in place of a lambda (Items 14. 43):

顺便提一下，如果使用比较器构造方法代替 lambda，则片段中的比较器可以更简洁（第 14. 43 项）：

```
Collections.sort(words, comparingInt(String::length));
```

In fact, the snippet can be made still shorter by taking advantage of the sort method that was added to the List interface in Java 8:

实际上，通过利用 Java 8 中添加到 List 接口的 sort 方法，可以使代码段更短：

```
words.sort(comparingInt(String::length));
```

The addition of lambdas to the language makes it practical to use function objects where it would not previously have made sense. For example, consider the Operation enum type in Item 34. Because each enum required different behavior for its apply method, we used constant-specific class bodies and overrode the apply method in each enum constant. To refresh your memory, here is the code:

将 lambda 添加到语言中使得使用函数对象变得切实可行。 例如，考虑第 34 项中的 Operation 枚举类型。因为每个枚举对其 apply 方法需要不同的行为，所以我们使用常量特定的类主体并覆盖每个枚举常量中的 apply 方法。 为了刷新你的记忆，这里是代码：

```
// Enum type with constant-specific class bodies & data (Item 34)
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
```

```
        @Override public String toString() { return symbol; }
        public abstract double apply(double x, double y);
    }
```

Item 34 says that enum instance fields are preferable to constant-specific class bodies. Lambdas make it easy to implement constant-specific behavior using the former instead of the latter. Merely pass a lambda implementing each enum constant's behavior to its constructor. The constructor stores the lambda in an instance field, and the apply method forwards invocations to the lambda. The resulting code is simpler and clearer than the original version:

第 34 项说 enum 实例字段比常量特定的类体更可取。使用前者而不是后者，Lambdas 可以轻松实现常量特定的行为。 只需将实现每个枚举常量行为的 lambda 传递给它的构造函数。 构造函数将 lambda 存储在实例字段中，apply 方法将调用转发给 lambda。 生成的代码比原始版本更简单，更清晰：

```
    // Enum with function object fields & constant-specific behavior
    public enum Operation {
        PLUS ("+", (x, y) -> x + y),
        MINUS ("-", (x, y) -> x - y),
        TIMES ("*", (x, y) -> x * y),
        DIVIDE("/", (x, y) -> x / y);
        private final String symbol;
        private final DoubleBinaryOperator op;
        Operation(String symbol, DoubleBinaryOperator op) {
            this.symbol = symbol;
            this.op = op;
        }
        @Override public String toString() { return symbol; }
        public double apply(double x, double y) {
            return op.applyAsDouble(x, y);
```

```
        }

    }
```

Note that we're using the DoubleBinaryOperator interface for the lambdas that represent the enum constant's behavior. This is one of the many predefined functional interfaces in java.util.function (Item 44). It represents a function that takes two double arguments and returns a double result.

请注意，我们使用 DoubleBinaryOperator 接口来表示枚举常量行为的 lambdas。 这是 java.util.function（Item 44）中许多预定义的功能接口之一。 它表示一个函数，它接受两个双参数并返回一个 double 结果。

Looking at the lambda-based Operation enum, you might think constantspecific method bodies have outlived their usefulness, but this is not the case. Unlike methods and classes, **lambdas lack names and documentation; if a computation isn't self-explanatory, or exceeds a few lines, don't put it in a lambda.** One line is ideal for a lambda, and three lines is a reasonable maximum. If you violate this rule, it can cause serious harm to the readability of your programs. If a lambda is long or difficult to read, either find a way to simplify it or refactor your program to eliminate it. Also, the arguments passed to enum constructors are evaluated in a static context. Thus, lambdas in enum constructors can't access instance members of the enum. Constant-specific class bodies are still the way to go if an enum type has constant-specific behavior that is difficult to understand, that can't be implemented in a few lines, or that requires access to instance fields or methods.

查看基于 lambda 的 Operation 枚举，您可能会认为常量特定的方法体已经过时了，但事实并非如此。 与方法和类不同，lambdas 缺少名称和文档; 如果计算不是自我解释，或超过几行，请不要将它放在 lambda 中。 一条线对于 lambda 是理想的，三条线是合理的最大值。 如果违反此规则，可能会严重损害程序的可读性。 如果 lambda 很长或难以阅读，要么找到简化它的方法，要么重构你的程序以消除它。 此外，传递给枚举构造函数的参数在静态上下文中进行计算。 因此，枚举构造函数中的 lambdas 无法访问枚举的实例成员。 如果枚举类型具有难以理解的常量特定行为，无法在几行中实现，或者需要访问实例字段或方法，则仍然可以使用特定于常量的类主体。

Likewise, you might think that anonymous classes are obsolete in the era of lambdas. This is closer to the truth, but there are a few things you can do with anonymous classes that you can't do with lambdas. Lambdas are limited to functional interfaces. If you want to create an instance of an abstract class, you can do it with an

anonymous class, but not a lambda. Similarly, you can use anonymous classes to create instances of interfaces with multiple abstract methods. Finally, a lambda cannot obtain a reference to itself. In a lambda, the this keyword refers to the enclosing instance, which is typically what you want. In an anonymous class, the this keyword refers to the anonymous class instance. If you need access to the function object from within its body, then you must use an anonymous class.

同样，您可能会认为匿名类在 lambdas 时代已经过时了。 这更接近事实，但是你可以用匿名类做一些事情，而你无法用 lambdas 做。 Lambdas 仅限于功能接口。 如果要创建抽象类的实例，可以使用匿名类，但不能使用 lambda。 同样，您可以使用匿名类来创建具有多个抽象方法的接口实例。 最后，lambda 无法获得对自身的引用。 在 lambda 中，this 关键字引用封闭实例，这通常是您想要的。 在匿名类中，this 关键字引用匿名类实例。 如果需要从其体内访问函数对象，则必须使用匿名类。

Lambdas share with anonymous classes the property that you can't reliably serialize and deserialize them across implementations. Therefore, **you should rarely, if ever, serialize a lambda** (or an anonymous class instance). If you have a function object that you want to make serializable, such as a Comparator, use an instance of a private static nested class (Item 24).

Lambdas 与匿名类共享您无法在实现中可靠地序列化和反序列化它们的属性。 因此，您应该很少（如果有的话）序列化 lambda（或匿名类实例）。 如果您有一个要进行序列化的函数对象，例如 Comparator，请使用私有静态嵌套类的实例（Item 24）。

In summary, as of Java 8, lambdas are by far the best way to represent small function objects. **Don't use anonymous classes for function objects unless you have to create instances of types that aren't functional interfaces. Also, remember that lambdas make it so easy to represent small function objects that it opens the door to functional programming techniques that were not previously practical in Java.

总之，从 Java 8 开始，lambda 是迄今为止表示小函数对象的最佳方式。 ** 除非必须创建非功能接口类型的实例，否则不要对函数对象使用匿名类。 另外，请记住 lambda 使代表小函数对象变得如此容易，以至于它打开了以前在 Java 中不实用的函数式编程技术的大门。

43 方法引用优于 Lambda 表达式

The primary advantage of lambdas over anonymous classes is that they are more succinct. Java provides a way to generate function objects even more succinct than lambdas: method references. Here is a code snippet from a program that maintains a map from arbitrary keys to Integer values. If the value is interpreted as a count of the number of instances of the key, then the program is a multiset implementation. The function of the code snippet is to associate the number 1 with the key if it is not in the map and to increment the associated value if the key is already present:

lambdas 优于匿名类的主要优点是它们更简洁。 Java 提供了一种生成函数对象的方法，它比 lambdas：方法引用更简洁。 这是一个程序的代码片段，它维护从任意键到 Integer 值的映射。 如果该值被解释为密钥实例数的计数，则该程序是多集实现。代码段的功能是将数字 1 与密钥相关联（如果它不在映射中），并在密钥已存在时增加相关值：

```
map.merge(key, 1, (count, incr) -> count + incr);
```

Note that this code uses the merge method, which was added to the Map interface in Java 8. If no mapping is present for the given key, the method simply inserts the given value; if a mapping is already present, merge applies the given function to the current value and the given value and overwrites the current value with the result. This code represents a typical use case for the merge method.

请注意，此代码使用 merge 方法，该方法已添加到 Java 8 中的 Map 接口。如果给定键没有映射，则该方法只是插入给定值；如果已存在映射，则 merge 将给定函数应用于当前值和给定值，并使用结果覆盖当前值。 此代码表示合并方法的典型用例。

The code reads nicely, but there’s still some boilerplate. The parameters count and incr don’t add much value, and they take up a fair amount of space. Really, all the lambda tells you is that the function returns the sum of its two arguments. As of Java 8, Integer (and all the other boxed numerical primitive types) provides a static method sum that does exactly the same thing. We can simply pass a reference to this method and get the same result with less visual clutter:

代码读得很好，但仍然有一些样板。 参数 count 和 incr 不会增加太多值，并且占用相当大的空间。 实际上，所有 lambda 告诉你的是该函数返回其两个参数的总和。 从 Java 8 开始，Integer（以及所有其他盒装数字基元类型）提供

了一个完全相同的静态方法求和。 我们可以简单地传递对此方法的引用，并以较少的视觉混乱获得相同的结果：

```
map.merge(key, 1, Integer::sum);
```

The more parameters a method has, the more boilerplate you can eliminate with a method reference. In some lambdas, however, the parameter names you choose provide useful documentation, making the lambda more readable and maintainable than a method reference, even if the lambda is longer.

There's nothing you can do with a method reference that you can't also do with a lambda (with one obscure exception—see JLS, 9.9-2 if you're curious). That said, method references usually result in shorter, clearer code. They also give you an out if a lambda gets too long or complex: You can extract the code from the lambda into a new method and replace the lambda with a reference to that method. You can give the method a good name and document it to your heart's content.

If you're programming with an IDE, it will offer to replace a lambda with a method reference wherever it can. You should usually, but not always, take the IDE up on the offer. Occasionally, a lambda will be more succinct than a method reference. This happens most often when the method is in the same class as the lambda. For example, consider this snippet, which is presumed to occur in a class named GoshThisClassNameIsHumongous:

方法具有的参数越多，使用方法引用就可以消除的样板越多。但是，在某些 lambdas 中，您选择的参数名称提供了有用的文档，使得 lambda 比方法引用更易读和可维护，即使 lambda 更长。

对于一个你不能用 lambda 做的方法引用，你无能为力（有一个模糊的例外 - 如果你很好奇，请参阅 JLS，9.9-2）。也就是说，方法引用通常会导致更短，更清晰的代码。如果 lambda 变得太长或太复杂，它们也会给你一个 out：你可以将 lambda 中的代码提取到一个新方法中，并用对该方法的引用替换 lambda。您可以为该方法提供一个好名称，并将其记录在心脏的内容中。

如果您使用 IDE 进行编程，它将提供用方法引用替换 lambda，只要它可以。您通常（但不总是）应该在优惠中使用 IDE。偶尔，lambda 将比方法引用更简洁。当方法与 lambda 属于同一类时，这种情况最常发生。例如，考虑这个片段，假定它出现在名为 GoshThisClassNameIsHumongous 的类中：

```
service.execute(GoshThisClassNameIsHumongous::action);
```

The lambda equivalent looks like this:

lambda 等价物看起来像这样：(lambda 相当于这样)

```
service.execute(() -> action());
```

The snippet using the method reference is neither shorter nor clearer than the snippet using the lambda, so prefer the latter. Along similar lines, the Function interface provides a generic static factory method to return the identity function, Function.identity(). It's typically shorter and cleaner not to use this method but to code the equivalent lambda inline: x -> x.

Many method references refer to static methods, but there are four kinds that do not. Two of them are bound and unbound instance method references. In bound references, the receiving object is specified in the method reference. Bound references are similar in nature to static references: the function object takes the same arguments as the referenced method. In unbound references, the receiving object is specified when the function object is applied, via an additional parameter before the method's declared parameters. Unbound references are often used as mapping and filter functions in stream pipelines (Item 45). Finally, there are two kinds of constructor references, for classes and arrays. Constructor references serve as factory objects. All five kinds of method references are summarized in the table below:

使用方法引用的代码段既不比使用 lambda 的代码段更短也更清晰，所以更喜欢后者。类似地，Function 接口提供了一个通用的静态工厂方法来返回 Identity 函数 Function.identity（）。它通常更短更清洁，不使用此方法，而是编写等效的 lambda 内联：x　- > x。

许多方法引用引用静态方法，但有四种方法引用不引用静态方法。其中两个是绑定和未绑定的实例方法引用。在绑定引用中，接收对象在方法引用中指定。绑定引用在本质上类似于静态引用：函数对象采用与引用方法相同的参数。在未绑定的引用中，在应用函数对象时，通过方法声明的参数之前的附加参数指定接收对象。未绑定引用通常用作流管道中的映射和过滤功能（第 45 项）。最后，对于类和数组，有两种构造函数引用。构造函数引用充当工厂对象。所有五种方法参考总结在下表中：

方法参考类型	**例子**	**Lambda 等效**
Static	Integer::parseInt	str ->

方法参考类型	例子	**Lambda 等效**
Bound	Instant.now()::isAfter	Instant then =Instant.now(); t ->then.isAfter(t)
Unbound	String::toLowerCase	str ->str.toLowerCase()
Class Constructor	TreeMap<K,V>::new	() -> new TreeMap<K,V>
Array Constructor	int[]::new	len -> new int[len]

In summary, method references often provide a more succinct alternative to lambdas. **Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.**

总之，方法引用通常提供了一种更简洁的 lambdas 替代方法。 方法参考更短更清晰的地方，使用它们; 他们不在的地方，坚持使用 lambdas。

44 优先使用标准的函数式接口

Now that Java has lambdas, best practices for writing APIs have changed considerably. For example, the Template Method pattern [Gamma95], wherein a subclass overrides a primitive method to specialize the behavior of its superclass, is far less attractive. The modern alternative is to provide a static factory or constructor that accepts a function object to achieve the same effect. More generally, you'll be writing more constructors and methods that take function objects as parameters. Choosing the right functional parameter type demands care.

既然 Java 有 lambda，那么编写 API 的最佳实践已经发生了很大变化。 例如，模板方法模式[Gamma95]，其中子类重写基本方法以专门化其超类的行为，远没那么有吸引力。 现代的替代方法是提供一个静态工厂或构造函数，它接受一个函数对象来实现相同的效果。 更一般地说，您将编写更多以函数对象作为参数的构造函数和方法。 选择正确的功能参数类型需要谨慎。

Consider LinkedHashMap. You can use this class as a cache by overriding its protected removeEldestEntry method, which is invoked by put each time a new key

is added to the map. When this method returns true, the map removes its eldest entry, which is passed to the method. The following override allows the map to grow to one hundred entries and then deletes the eldest entry each time a new key is added, maintaining the hundred most recent entries:

考虑 LinkedHashMap。 您可以通过覆盖其受保护的 removeEldestEntry 方法将此类用作缓存，该方法每次将新键添加到地图时都会调用。当此方法返回 true 时，映射将删除其最旧的条目，该条目将传递给该方法。 以下覆盖允许地图增长到一百个条目，然后在每次添加新密钥时删除最旧的条目，保留最近的一百个条目：

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return size() > 100;
}
```

This technique works fine, but you can do much better with lambdas. If LinkedHashMap were written today, it would have a static factory or constructor that took a function object. Looking at the declaration for removeEldestEntry, you might think that the function object should take a Map.Entry<K,V> and return a boolean, but that wouldn’t quite do it: The removeEldestEntry method calls size() to get the number of entries in the map, which works because removeEldestEntry is an instance method on the map. The function object that you pass to the constructor is not an instance method on the map and can’t capture it because the map doesn’t exist yet when its factory or constructor is invoked. Thus, the map must pass itself to the function object, which must therefore take the map on input as well as its eldest entry. If you were to declare such a functional interface, it would look something like this:

这种技术很好，但你可以用 lambdas 做得更好。 如果今天写了 LinkedHashMap，它将有一个带有函数对象的静态工厂或构造函数。 查看 removeEldestEntry 的声明，你可能会认为函数对象应该采用 Map.Entry <K，V> 并返回一个布尔值，但是不会这样做：removeEldestEntry 方法调用 size（）来获取数字 地图中的条目，因为 removeEldestEntry 是地图上的实例方法。 传递给构造函数的函数对象不是地图上的实例方法，并且无法捕获它，因为在调用其工厂或构造函数时映射尚不存在。 因此，映射必须将自身传递给函数对象，因此函数对象必须在输入及其最旧条目上获取映射。 如果你要声明这样一个功能界面，它看起来像这样：

```
// Unnecessary functional interface; use a standard one instead.
```

```
@FunctionalInterface interface EldestEntryRemovalFunction<K,V>{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

This interface would work fine, but you shouldn't use it, because you don't need to declare a new interface for this purpose. The java.util.function package provides a large collection of standard functional interfaces for your use. **If one of the standard functional interfaces does the job, you should generally use it in preference to a purpose-built functional interface.** This will make your API easier to learn, by reducing its conceptual surface area, and will provide significant interoperability benefits, as many of the standard functional interfaces provide useful default methods. The Predicate interface, for instance, provides methods to combine predicates. In the case of our LinkedHashMap example, the standard BiPredicate<Map<K,V>, Map.Entry<K,V>> interface should be used in preference to a custom EldestEntryRemovalFunction interface.

此接口可以正常工作，但您不应该使用它，因为您不需要为此目的声明新接口。 java.util.function 包提供了大量标准功能接口供您使用。 如果其中一个标准功能接口完成了这项工作，您通常应该优先使用它，而不是专门构建的功能接口。 这将使您的 API 更容易学习，通过减少其概念表面积，并将提供显着的互操作性优势，因为许多标准功能接口提供有用的默认方法。 例如，Predicate 接口提供了组合谓词的方法。 对于 LinkedHashMap 示例，应优先使用标准 BiPredicate <Map <K，V>，Map.Entry <K，V >>接口，而不是自定义 EldestEntryRemovalFunction 接口。

There are forty-three interfaces in java.util.Function. You can't be expected to remember them all, but if you remember six basic interfaces, you can derive the rest when you need them. The basic interfaces operate on object reference types. The Operator interfaces represent functions whose result and argument types are the same. The Predicate interface represents a function that takes an argument and returns a boolean. The Function interface represents a function whose argument and return types differ. The Supplier interface represents a function that takes no arguments and returns (or "supplies") a value. Finally, Consumer represents a function that takes an argument and returns nothing, essentially consuming its argument. The six basic functional interfaces are summarized below:

java.util.Function 中有四十三个接口。 不能指望你记住它们，但如果你记得六个基本接口，你可以在需要时得到其余的接口。 基本接口对对象引用类型进

行操作。 Operator 接口表示结果和参数类型相同的函数。 Predicate 接口表示一个接受参数并返回布尔值的函数。 Function 接口表示其参数和返回类型不同的函数。Supplier 接口表示不带参数并返回（或“提供”）值的函数。最后，Consumer 表示一个函数，它接受一个参数并且什么都不返回，基本上消耗它的参数。 六个基本功能接口总结如下：

接口	**功能签名 Function Signature**	**Example**
UnaryOperator	T apply(T t)	String::toLowerCase
BinaryOperator	T apply(T t1, T t2)	BigInteger::add
Predicate	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier	T get()	Instant::now
Consumer	void accept(T t)	System.out::println

There are also three variants of each of the six basic interfaces to operate on the primitive types int, long, and double. Their names are derived from the basic interfaces by prefixing them with a primitive type. So, for example, a predicate that takes an int is an IntPredicate, and a binary operator that takes two long values and returns a long is a LongBinaryOperator. None of these variant types is parameterized except for the Function variants, which are parameterized by return type. For example, LongFunction<int[]> takes a long and returns an int[].

六种基本接口中的每一种都有三种变体可以对基本类型 int，long 和 double 进行操作。 它们的名称来源于基本接口，前缀为基本类型。 因此，例如，带有 int 的谓词是 IntPredicate，带有两个 long 值并返回 long 的二元运算符是 LongBinaryOperator。 除函数变量外，这些变量类型都不参数化，函数变量由返回类型参数化。 例如，LongFunction <int []>接受一个 long 并返回一个 int []。

There are nine additional variants of the Function interface, for use when the result type is primitive. The source and result types always differ, because a function from a type to itself is a UnaryOperator. If both the source and result types are primitive, prefix Function with SrcToResult, for example LongToIntFunction (six

variants). If the source is a primitive and the result is an object reference, prefix Function with ToObj, for example DoubleToObjFunction (three variants).

Function 接口有九个附加变体，供结果类型为原始时使用。 源和结果类型总是不同，因为从类型到自身的函数是 UnaryOperator。 如果源类型和结果类型都是原始类型，则使用 SrcToResult 作为前缀 Function，例如 LongToIntFunction（六个变体）。 如果源是基元并且结果是对象引用，则使用 ToObj 作为前缀 Function，例如 DoubleToObjFunction（三个变体）。

There are two-argument versions of the three basic functional interfaces for which it makes sense to have them: BiPredicate<T,U>, BiFunction<T,U,R>, and BiConsumer<T,U>. There are also BiFunction variants returning the three relevant primitive types: ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, and ToDoubleBiFunction<T,U>. There are two-argument variants of Consumer that take one object reference and one primitive type: ObjDoubleConsumer, ObjIntConsumer, and ObjLongConsumer. In total, there are nine two-argument versions of the basic interfaces.

有三个基本功能接口的两个参数版本，使用它们是有意义的：BiPredicate <T，U>，BiFunction <T，U，R>和 BiConsumer <T，U>。 还有 BiFunction 变体返回三种相关的基本类型：ToIntBiFunction <T，U>，ToLongBiFunction <T，U>和 ToDoubleBiFunction <T，U>。 Consumer 的两个参数变体采用一个对象引用和一个基本类型：ObjDoubleConsumer，ObjIntConsumer 和 ObjLongConsumer。总共有九个基本接口的双参数版本。

Finally, there is the BooleanSupplier interface, a variant of Supplier that returns boolean values. This is the only explicit mention of the boolean type in any of the standard functional interface names, but boolean return values are supported via Predicate and its four variant forms. The BooleanSupplier interface and the forty-two interfaces described in the previous paragraphs account for all forty-three standard functional interfaces. Admittedly, this is a lot to swallow, and not terribly orthogonal. On the other hand, the bulk of the functional interfaces that you'll need have been written for you and their names are regular enough that you shouldn't have too much trouble coming up with one when you need it.

最后，还有 BooleanSupplier 接口，这是 Supplier 的一个变量，它返回布尔值。这是任何标准功能接口名称中唯一明确提到的布尔类型，但是通过 Predicate 及其四种变体形式支持布尔返回值。 BooleanSupplier 接口和前面段落中描述的四十二个接口占所有四十三个标准功能接口。 不可否认，这是一个很大的吞并，

而不是非常正交。 另一方面，您需要的大部分功能接口都是为您编写的，并且它们的名称足够常规，以便您在需要时不会遇到太多麻烦。

Most of the standard functional interfaces exist only to provide support for primitive types. **Don't be tempted to use basic functional interfaces with boxed primitives instead of primitive functional interfaces.** While it works, it violates the advice of Item 61, "prefer primitive types to boxed primitives." The performance consequences of using boxed primitives for bulk operations can be deadly.

Now you know that you should typically use standard functional interfaces in preference to writing your own. But when should you write your own? Of course you need to write your own if none of the standard ones does what you need, for example if you require a predicate that takes three parameters, or one that throws a checked exception. But there are times you should write your own functional interface even when one of the standard ones is structurally identical.

大多数标准功能接口仅用于提供对原始类型的支持。 不要试图使用基本功能接口与盒装基元而不是原始功能接口。 虽然它有效但它违反了第 61 条的建议，"更喜欢原始类型到盒装基元。"使用盒装基元进行批量操作的性能后果可能是致命的。

现在您知道通常应该使用标准功能接口而不是编写自己的接口。 但你应该什么时候写自己的？ 当然，如果没有标准的那些符合您的需要，您需要自己编写，例如，如果您需要一个带有三个参数的谓词，或者一个抛出已检查异常的谓词。 但有时您应该编写自己的功能界面，即使其中一个标准结构完全相同。

Consider our old friend Comparator, which is structurally identical to the ToIntBiFunction<T,T> interface. Even if the latter interface had existed when the former was added to the libraries, it would have been wrong to use it. There are several reasons that Comparator deserves its own interface. First, its name provides excellent documentation every time it is used in an API, and it's used a lot. Second, the Comparator interface has strong requirements on what constitutes a valid instance, which comprise its general contract. By implementing the interface, you are pledging to adhere to its contract. Third, the interface is heavily outfitted with useful default methods to transform and combine comparators.

考虑我们的老朋友 Comparator，它在结构上与 ToIntBiFunction <T，T>接口相同。 即使后者接口已经存在，当前者被添加到库中时，使用它也是错误的。 Comparator 有几个原因值得拥有自己的界面。 首先，它的名称在每次在 API 中使用时都提供了出色的文档，并且它被大量使用。 其次，Comparator 接口对构

成有效实例的内容有很强的要求，有效实例包含其一般合同。 通过实施界面，您承诺遵守其合同。 第三，接口配备了大量有用的默认方法来转换和组合比较器。

You should seriously consider writing a purpose-built functional interface in preference to using a standard one if you need a functional interface that shares one or more of the following characteristics with Comparator:

如果您需要一个与 Comparator 共享以下一个或多个特性的功能接口，您应该认真考虑编写专用的功能接口而不是使用标准接口：

It will be commonly used and could benefit from a descriptive name.

It has a strong contract associated with it.

It would benefit from custom default methods.

- 它将被普遍使用，并可从描述性名称中受益。
- 它与之相关的合同很强。
- 它将受益于自定义默认方法。

If you elect to write your own functional interface, remember that it’s an interface and hence should be designed with great care (Item 21).

Notice that the EldestEntryRemovalFunction interface (page 199) is labeled with the @FunctionalInterface annotation. This annotation type is similar in spirit to @Override. It is a statement of programmer intent that serves three purposes: it tells readers of the class and its documentation that the interface was designed to enable lambdas; it keeps you honest because the interface won’t compile unless it has exactly one abstract method; and it prevents maintainers from accidentally adding abstract methods to the interface as it evolves. **Always annotate your functional interfaces with the @FunctionalInterface annotation.**

如果您选择编写自己的功能界面，请记住它是一个界面，因此应该非常谨慎地设计（第 21 项）。

请注意，EldestEntryRemovalFunction 接口（第 199 页）标有@FunctionalInterface 注释。 此注释类型在精神上与@Override 类似。 它是程序员意图的声明，有三个目的：它告诉读者该类及其文档，该接口旨在启用 lambdas; 它保持诚实，因为除非它只有一个抽象方法，否则接口不会编译; 并且它可以防

止维护者在接口发生时意外地将抽象方法添加到接口。 始终使用 @FunctionalInterface 注释来注释您的功能接口。

A final point should be made concerning the use of functional interfaces in APIs. Do not provide a method with multiple overloadings that take different functional interfaces in the same argument position if it could create a possible ambiguity in the client. This is not just a theoretical problem. The submit method of ExecutorService can take either a Callable or a Runnable, and it is possible to write a client program that requires a cast to indicate the correct overloading (Item 52). The easiest way to avoid this problem is not to write overloadings that take different functional interfaces in the same argument position. This is a special case of the advice in Item 52, “use overloading judiciously.”

In summary, now that Java has lambdas, it is imperative that you design your APIs with lambdas in mind. Accept functional interface types on input and return them on output. It is generally best to use the standard interfaces provided in java.util.function.Function, but keep your eyes open for the relatively rare cases where you would be better off writing your own functional interface.

关于 API 中功能接口的使用，应该最后一点。如果可能在客户端中产生可能的歧义，则不要提供具有多个重载的方法，这些方法在相同的参数位置采用不同的功能接口。这不仅仅是一个理论问题。 ExecutorService 的 submit 方法可以采用 Callable 或 Runnable，并且可以编写一个客户端程序，需要使用强制转换来指示正确的重载（第 52 项）。避免此问题的最简单方法是不要编写在同一参数位置使用不同功能接口的重载。这是第 52 项建议中的一个特例，“明智地使用重载”。

总而言之，既然 Java 已经有了 lambdas，那么在设计 API 时必须考虑到 lambdas。接受输入上的功能接口类型并在输出上返回它们。通常最好使用 java.util.function.Function 中提供的标准接口，但请注意相对罕见的情况，即最好编写自己的功能接口。

45 小心使用流

The streams API was added in Java 8 to ease the task of performing bulk operations, sequentially or in parallel. This API provides two key abstractions: the stream, which represents a finite or infinite sequence of data elements, and the stream pipeline, which represents a multistage computation on these elements. The elements in a stream can come from anywhere. Common sources include collections, arrays, files, regular expression pattern matchers, pseudorandom number generators, and

other streams. The data elements in a stream can be object references or primitive values. Three primitive types are supported: int, long, and double.

在 Java 8 中添加了流 API，以简化顺序或并行执行批量操作的任务。该 API 提供了两个关键的抽象：流，表示有限或无限的数据元素序列，以及流管道，表示对这些元素的多级计算。 流中的元素可以来自任何地方。 常见的源包括集合，数组，文件，正则表达式模式匹配器，伪随机数生成器和其他流。 流中的数据元素可以是对象引用或原始值。 支持三种基本类型：int，long 和 double。

A stream pipeline consists of a source stream followed by zero or more intermediate operations and one terminal operation. Each intermediate operation transforms the stream in some way, such as mapping each element to a function of that element or filtering out all elements that do not satisfy some condition. Intermediate operations all transform one stream into another, whose element type may be the same as the input stream or different from it. The terminal operation performs a final computation on the stream resulting from the last intermediate operation, such as storing its elements into a collection, returning a certain element, or printing all of its elements.

流管道由源流和零个或多个中间操作以及一个终端操作组成。 每个中间操作以某种方式转换流，例如将每个元素映射到该元素的函数或过滤掉不满足某些条件的所有元素。 中间操作都将一个流转换为另一个流，其元素类型可以与输入流相同或与之不同。 终端操作对从最后的中间操作产生的流执行最终计算，例如将其元素存储到集合中，返回某个元素或打印其所有元素。

Stream pipelines are evaluated lazily: evaluation doesn’t start until the terminal operation is invoked, and data elements that aren’t required in order to complete the terminal operation are never computed. This lazy evaluation is what makes it possible to work with infinite streams. Note that a stream pipeline without a terminal operation is a silent no-op, so don’t forget to include one.

流管道被懒惰地评估：在调用终端操作之前不开始评估，并且从不计算为完成终端操作而不需要的数据元素。 这种懒惰的评估使得可以使用无限流。 请注意，没有终端操作的流管道是静默无操作，因此不要忘记包含一个。

The streams API is fluent: it is designed to allow all of the calls that comprise a pipeline to be chained into a single expression. In fact, multiple pipelines can be chained together into a single expression.

流 API 非常流畅：它旨在允许将构成管道的所有调用链接到单个表达式中。实际上，多个管道可以链接在一起形成一个表达式。

By default, stream pipelines run sequentially. Making a pipeline execute in parallel is as simple as invoking the parallel method on any stream in the pipeline, but it is seldom appropriate to do so (Item 48).

默认情况下，流管道按顺序运行。 使管道并行执行就像在管道中的任何流上调用并行方法一样简单，但很少这样做（第 48 项）。

The streams API is sufficiently versatile that practically any computation can be performed using streams, but just because you can doesn't mean you should. When used appropriately, streams can make programs shorter and clearer; when used inappropriately, they can make programs difficult to read and maintain. There are no hard and fast rules for when to use streams, but there are heuristics.

流 API 具有足够的通用性，几乎任何计算都可以使用流来执行，但仅仅因为你并不意味着你应该这样做。 如果使用得当，流可以使程序更短更清晰；如果使用不当，可能会使程序难以阅读和维护。 什么时候使用流没有硬性规定，但有启发式方法。

Consider the following program, which reads the words from a dictionary file and prints all the anagram groups whose size meets a user-specified minimum. Recall that two words are anagrams if they consist of the same letters in a different order. The program reads each word from a user-specified dictionary file and places the words into a map. The map key is the word with its letters alphabetized, so the key for "staple" is "aelpst", and the key for "petals" is also "aelpst": the two words are anagrams, and all anagrams share the same alphabetized form (or alphagram, as it is sometimes known). The map value is a list containing all of the words that share an alphabetized form. After the dictionary has been processed, each list is a complete anagram group. The program then iterates through the map's values() view and prints each list whose size meets the threshold:

考虑以下程序，该程序从字典文件中读取单词并打印其大小符合用户指定的最小值的所有 anagram 组。 回想一下，如果两个单词由不同顺序的相同字母组成，则它们是字谜。 程序从用户指定的字典文件中读取每个单词并将单词放入地图中。 地图键是用字母按字母顺序排列的单词，因此“staple”的键是“aelpst”，“花瓣”的键也是“aelpst”：两个单词是 anagrams，所有的 anagrams 共享相同的字母形式 （或 alphagram，因为它有时是已知的）。 地图值是包含共享按字母顺序排列的形式的所有单词的列表。 字典处理完毕后，每个列表都是一

个完整的字谜组。 然后程序遍历 map 的 values（）视图并打印每个大小符合阈值的列表：

```
// Prints all large anagram groups in a dictionary iteratively

public class Anagrams {

    public static void main(String[] args) throws IOException {

        File dictionary = new File(args[0]);

        int minGroupSize = Integer.parseInt(args[1]);

        Map<String, Set<String>> groups = new HashMap<>();

        try (Scanner s = new Scanner(dictionary)) {

            while (s.hasNext()) {

                String word = s.next();

                groups.computeIfAbsent(alphabetize(word),(unused) -> new
TreeSet<>()).add(word);

            }

        }

        for (Set<String> group : groups.values())

        if (group.size() >= minGroupSize)

            System.out.println(group.size() + ": " + group);

    }

    private static String alphabetize(String s) {

        char[] a = s.toCharArray();

        Arrays.sort(a);

        return new String(a);

    }

}
```

One step in this program is worthy of note. The insertion of each word into the map, which is shown in bold, uses the computeIfAbsent method, which was added in Java 8. This method looks up a key in the map: If the key is present, the method simply returns the value associated with it. If not, the method computes a value by applying the given function object to the key, associates this value with the key, and returns the computed value. The computeIfAbsent method simplifies the implementation of maps that associate multiple values with each key.

该计划的一个步骤值得注意。 将每个单词插入到地图中（以粗体显示）使用在 Java 8 中添加的 computeIfAbsent 方法。此方法在地图中查找键：如果键存在，则该方法仅返回关联的值 用它。 如果不是，则该方法通过将给定的函数对象应用于键来计算值，将该值与键相关联，并返回计算的值。 computeIfAbsent 方法简化了将多个值与每个键相关联的映射的实现。

Now consider the following program, which solves the same problem, but makes heavy use of streams. Note that the entire program, with the exception of the code that opens the dictionary file, is contained in a single expression. The only reason the dictionary is opened in a separate expression is to allow the use of the try-with-resources statement, which ensures that the dictionary file is closed:

现在考虑以下程序，它解决了同样的问题，但大量使用了流。 请注意，除了打开字典文件的代码之外，整个程序都包含在一个表达式中。 在单独的表达式中打开字典的唯一原因是允许使用 try-with-resources 语句，这可确保关闭字典文件：

```
// Overuse of streams - don't do this!

public class Anagrams {

    public static void main(String[] args) throws IOException {

        Path dictionary = Paths.get(args[0]);

        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {

            words.collect(

            groupingBy(word -> word.chars().sorted()

            .collect(StringBuilder::new,(sb, c) -> sb.append((char) c),

            StringBuilder::append).toString()))
```

```
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .map(group -> group.size() + ": " + group)
                .forEach(System.out::println);
        }
    }
}
```

If you find this code hard to read, don't worry; you're not alone. It is shorter, but it is also less readable, especially to programmers who are not experts in the use of streams. Overusing streams makes programs hard to read and maintain. Luckily, there is a happy medium. The following program solves the same problem, using streams without overusing them. The result is a program that's both shorter and clearer than the original:

如果您发现此代码难以阅读，请不要担心; 你不是一个人。 它更短，但也不太可读，特别是对于不是使用流的专家的程序员。 过度使用流程会使程序难以阅读和维护。 幸运的是，有一个幸福的媒介。 以下程序使用流而不过度使用流来解决相同的问题。 结果是一个比原始程序更短更清晰的程序：

```
// Tasteful use of streams enhances clarity and conciseness
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);
        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .forEach(g -> System.out.println(g.size() + ": " + g));
        }
```

```
        }

        // alphabetize method is the same as in original version

    }
```

Even if you have little previous exposure to streams, this program is not hard to understand. It opens the dictionary file in a try-with-resources block, obtaining a stream consisting of all the lines in the file. The stream variable is named words to suggest that each element in the stream is a word. The pipeline on this stream has no intermediate operations; its terminal operation collects all the words into a map that groups the words by their alphabetized form (Item 46). This is exactly the same map that was constructed in both previous versions of the program. Then a new Stream<List> is opened on the values() view of the map. The elements in this stream are, of course, the anagram groups. The stream is filtered so that all of the groups whose size is less than minGroupSize are ignored, and finally, the remaining groups are printed by the terminal operation forEach.

即使你以前很少接触过流，这个程序也不难理解。 它在 try-with-resources 块中打开字典文件，获取包含文件中所有行的流。 stream 变量被命名为单词，表示流中的每个元素都是一个单词。 此流上的管道没有中间操作; 它的终端操作将所有单词收集到一个地图中，该地图按字母顺序排列单词（第 46 项）。 这与在以前版本的程序中构建的地图完全相同。 然后在地图的 values（）视图上打开一个新的 Stream <List>。 当然，这个流中的元素是 anagram 组。 过滤流以便忽略大小小于 minGroupSize 的所有组，最后，通过终端操作 forEach 打印剩余的组。

Note that the lambda parameter names were chosen carefully. The parameter g should really be named group, but the resulting line of code would be too wide for the book. **In the absence of explicit types, careful naming of lambda parameters is essential to the readability of stream pipelines.**

请注意，小心选择了 lambda 参数名称。 参数 g 应该真正命名为 group，但是生成的代码行对于本书来说太宽了。 在没有显式类型的情况下，仔细命名 lambda 参数对于流管道的可读性至关重要。

Note also that word alphabetization is done in a separate alphabetize method. This enhances readability by providing a name for the operation and keeping implementation details out of the main program. **Using helper methods is even more important for readability in stream pipelines than in iterative code** because pipelines lack explicit type information and named temporary variables.

The alphabetize method could have been reimplemented to use streams, but a stream-based alphabetize method would have been less clear, more difficult to write correctly, and probably slower. These deficiencies result from Java's lack of support for primitive char streams (which is not to imply that Java should have supported char streams; it would have been infeasible to do so). To demonstrate the hazards of processing char values with streams, consider the following code:

另请注意，单词字母化是在单独的字母顺序排列方法中完成的。 这通过提供操作的名称并将实现细节保留在主程序之外来增强可读性。 使用辅助方法对于流管道中的可读性比在迭代代码中更为重要，因为管道缺少显式类型信息和命名临时变量。

可以重新实现字母顺序排列方法以使用流，但是基于流的字母顺序排列方法不太清晰，更难以正确编写，并且可能更慢。 这些缺陷是由于 Java 缺乏对原始字符串流的支持（这并不意味着 Java 应该支持 char 流;这样做是不可行的）。 要演示使用流处理 char 值的危险，请考虑以下代码：

```
"Hello world!".chars().forEach(System.out::print);
```

You might expect it to print Hello world!, but if you run it, you'll find that it prints 72101108108111321191111141081003 3. This happens because the elements of the stream returned by "Hello world!".chars() are not char values but int values, so the int overloading of print is invoked. It is admittedly confusing that a method named chars returns a stream of int values. You could fix the program by using a cast to force the invocation of the correct overloading:

您可能希望它打印 Hello world！，但如果您运行它，您会发现它打印 72101108108111321191111141081003 3。这是因为“Hello world！”。chars（）返回的流的元素不是 char 值而是 int 值，因此调用 print 的 int 重载。 令人遗憾的是，名为 chars 的方法返回一个 int 值流。 您可以通过使用强制转换来强制调用正确的重载来修复程序：

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

but ideally you should refrain from using streams to process char values. When you start using streams, you may feel the urge to convert all your loops into streams, but resist the urge. While it may be possible, it will likely harm the readability and maintainability of your code base. As a rule, even moderately complex tasks are best accomplished using some combination of streams and iteration, as illustrated by the Anagrams programs above. So **refactor existing code to use streams and use them in new code only where it makes sense to do so.**

但理想情况下，您应该避免使用流来处理 char 值。 当您开始使用流时，您可能会感觉到将所有循环转换为流的冲动，但抵制冲动。 尽管有可能，但可能会损害代码库的可读性和可维护性。 通常，使用流和迭代的某种组合可以最好地完成中等复杂的任务，如上面的 Anagrams 程序所示。 因此，重构现有代码以使用流，并仅在有意义的情况下在新代码中使用它们。

As shown in the programs in this item, stream pipelines express repeated computation using function objects (typically lambdas or method references), while iterative code expresses repeated computation using code blocks. There are some things you can do from code blocks that you can’t do from function objects:

如该项目中的程序所示，流管道使用函数对象（通常是 lambdas 或方法引用）表示重复计算，而迭代代码使用代码块表示重复计算。 您可以从函数对象无法执行的代码块中执行以下操作：

> From a code block, you can read or modify any local variable in scope; from a lambda, you can only read final or effectively final variables [JLS 4.12.4], and you can’t modify any local variables.
>
> From a code block, you can return from the enclosing method, break or continue an enclosing loop, or throw any checked exception that this method is declared to throw; from a lambda you can do none of these things.

If a computation is best expressed using these techniques, then it’s probably not a good match for streams. Conversely, streams make it very easy to do some things:

> Uniformly transform sequences of elements
>
> Filter sequences of elements
>
> Combine sequences of elements using a single operation (for example to add them, concatenate them, or compute their minimum)
>
> Accumulate sequences of elements into a collection, perhaps grouping them by some common attribute
>
> Search a sequence of elements for an element satisfying some criterion

If a computation is best expressed using these techniques, then it is a good candidate for streams.

- 从代码块中，您可以读取或修改范围内的任何局部变量;从 lambda 中，您只能读取最终或有效的最终变量[JLS 4.12.4]，并且您无法修改任何局部变量。
- 从代码块，您可以从封闭方法返回，中断或继续封闭循环，或抛出声明此方法被抛出的任何已检查异常;从一个 lambda 你不能做这些事情。
- 如果使用这些技术最好地表达计算，那么它可能不是流的良好匹配。相反，流可以很容易地做一些事情：
- 均匀地转换元素序列
- 过滤元素序列
- 使用单个操作组合元素序列（例如，添加它们，连接它们或计算它们的最小值）
- 将元素序列累积到集合中，或者通过一些常见属性对它们进行分组
- 在元素序列中搜索满足某个条件的元素
- 如果使用这些技术最好地表达计算，那么它是流的良好候选者。

One thing that is hard to do with streams is to access corresponding elements from multiple stages of a pipeline simultaneously: once you map a value to some other value, the original value is lost. One workaround is to map each value to a pair object containing the original value and the new value, but this is not a satisfying solution, especially if the pair objects are required for multiple stages of a pipeline. The resulting code is messy and verbose, which defeats a primary purpose of streams. When it is applicable, a better workaround is to invert the mapping when you need access to the earlier-stage value.

使用流很难做的一件事是同时从管道的多个阶段访问相应的元素：一旦将值映射到某个其他值，原始值就会丢失。 一种解决方法是将每个值映射到包含原始值和新值的对对象，但这不是一个令人满意的解决方案，尤其是如果管道的多个阶段需要对对象。 由此产生的代码是混乱和冗长的，这破坏了流的主要目的。 如果适用，更好的解决方法是在需要访问早期阶段值时反转映射。

For example, let’s write a program to print the first twenty Mersenne primes. To refresh your memory, a Mersenne number is a number of the form $2^p - 1$. If p is prime, the corresponding Mersenne number may be prime; if so, it’s a Mersenne prime. As the initial stream in our pipeline, we want all the prime numbers. Here’s a

method to return that (infinite) stream. We assume a static import has been used for easy access to the static members of BigInteger:

例如，让我们编写一个程序来打印前 20 个 Mersenne 素数。 为了刷新你的记忆，梅森数是一个 2p - 1 的数字。如果 p 是素数，相应的梅森数可能是素数; 如果是这样的话，那就是梅森素数。 作为我们管道中的初始流，我们需要所有素数。 这是一种返回该（无限）流的方法。 我们假设使用静态导入来轻松访问 BigInteger 的静态成员：

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

The name of the method (primes) is a plural noun describing the elements of the stream. This naming convention is highly recommended for all methods that return streams because it enhances the readability of stream pipelines. The method uses the static factory Stream.iterate, which takes two parameters: the first element in the stream, and a function to generate the next element in the stream from the previous one. Here is the program to print the first twenty Mersenne primes:

方法（primes）的名称是描述流的元素的复数名词。 强烈建议所有返回流的方法使用此命名约定，因为它增强了流管道的可读性。 该方法使用静态工厂 Stream.iterate，它接受两个参数：流中的第一个元素，以及从前一个元素生成流中的下一个元素的函数。 这是打印前 20 个 Mersenne 素数的程序：

```
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
    .filter(mersenne -> mersenne.isProbablePrime(50))
    .limit(20)
    .forEach(System.out::println);
}
```

This program is a straightforward encoding of the prose description above: it starts with the primes, computes the corresponding Mersenne numbers, filters out all but the primes (the magic number 50 controls the probabilistic primality test), limits the resulting stream to twenty elements, and prints them out.

这个程序是上面的散文描述的直接编码：它从素数开始，计算相应的梅森数，过滤掉除素数之外的所有数字（幻数 50 控制概率素性测试），将得到的流限制为 20 个元素， 并打印出来。

Now suppose that we want to precede each Mersenne prime with its exponent (p). This value is present only in the initial stream, so it is inaccessible in the terminal operation, which prints the results. Luckily, it's easy to compute the exponent of a Mersenne number by inverting the mapping that took place in the first intermediate operation. The exponent is simply the number of bits in the binary representation, so this terminal operation generates the desired result:

现在假设我们想要在每个 Mersenne 素数之前加上它的指数（p）。 该值仅出现在初始流中，因此在终端操作中无法访问，从而打印结果。 幸运的是，通过反转第一个中间操作中发生的映射，可以很容易地计算出 Mersenne 数的指数。指数只是二进制表示中的位数，因此该终端操作会生成所需的结果：

```
    .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

There are plenty of tasks where it is not obvious whether to use streams or iteration. For example, consider the task of initializing a new deck of cards. Assume that Card is an immutable value class that encapsulates a Rank and a Suit, both of which are enum types. This task is representative of any task that requires computing all the pairs of elements that can be chosen from two sets. Mathematicians call this the Cartesian product of the two sets. Here's an iterative implementation with a nested for-each loop that should look very familiar to you:

有很多任务，无论是使用流还是迭代都不明显。 例如，考虑初始化一副新牌的任务。 假设 Card 是一个不可变的值类，它封装了 Rank 和 Suit，两者都是枚举类型。 此任务代表任何需要计算可从两组中选择的所有元素对的任务。 数学家称之为两组的笛卡尔积。 这是一个带有嵌套 for-each 循环的迭代实现，对你来说应该非常熟悉：

```
    // Iterative Cartesian product computation

    private static List<Card> newDeck() {

        List<Card> result = new ArrayList<>();

        for (Suit suit : Suit.values())

        for (Rank rank : Rank.values())

        result.add(new Card(suit, rank));
```

```
        return result;
    }
```

And here is a stream-based implementation that makes use of the intermediate operation flatMap. This operation maps each element in a stream to a stream and then concatenates all of these new streams into a single stream (or flattens them). Note that this implementation contains a nested lambda, shown in boldface:

这是一个基于流的实现，它使用了中间操作 flatMap。 此操作将流中的每个元素映射到流，然后将所有这些新流连接成单个流（或展平它们）。 请注意，此实现包含嵌套的 lambda，以粗体显示：

```
    // Stream-based Cartesian product computation
    private static List<Card> newDeck() {
        return Stream.of(Suit.values())
        .flatMap(suit ->Stream.of(Rank.values())
        .map(rank -> new Card(suit, rank)))
        .collect(toList());
    }
```

Which of the two versions of newDeck is better? It boils down to personal preference and the environment in which you're programming. The first version is simpler and perhaps feels more natural. A larger fraction of Java programmers will be able to understand and maintain it, but some programmers will feel more comfortable with the second (stream-based) version. It's a bit more concise and not too difficult to understand if you're reasonably well-versed in streams and functional programming. If you're not sure which version you prefer, the iterative version is probably the safer choice. If you prefer the stream version and you believe that other programmers who will work with the code will share your preference, then you should use it.

newDeck 的两个版本中哪一个更好？ 它归结为个人偏好和您编程的环境。第一个版本更简单，也许感觉更自然。大部分 Java 程序员将能够理解和维护它，但是一些程序员会对第二个（基于流的）版本感觉更舒服。 如果您对流和函数式编程有相当的精通，那么它会更简洁，也不会太难理解。 如果您不确定自己

喜欢哪个版本，则迭代版本可能是更安全的选择。 如果你更喜欢流版本，并且你相信其他使用代码的程序员会分享你的偏好，那么你应该使用它。

In summary, some tasks are best accomplished with streams, and others with iteration. Many tasks are best accomplished by combining the two approaches. There are no hard and fast rules for choosing which approach to use for a task, but there are some useful heuristics. In many cases, it will be clear which approach to use; in some cases, it won't. If you're not sure whether a task is better served by streams or iteration, try both and see which works better.

总之，一些任务最好用流完成，其他任务最好用迭代完成。 通过组合这两种方法可以最好地完成许多任务。 选择哪种方法用于任务没有硬性规定，但有一些有用的启发式方法。 在许多情况下，将清楚使用哪种方法; 在某些情况下，它不会。 如果您不确定某个任务是否更适合流或迭代，请尝试两者并查看哪个更好。

46 流中优先使用无副作用的函数

If you're new to streams, it can be difficult to get the hang of them. Merely expressing your computation as a stream pipeline can be hard. When you succeed, your program will run, but you may realize little if any benefit. Streams isn't just an API, it's a paradigm based on functional programming. In order to obtain the expressiveness, speed, and in some cases parallelizability that streams have to offer, you have to adopt the paradigm as well as the API.

如果你对于流是新手，可能很难掌握它们。 仅仅将您的计算表示为流管道可能很难。 当你成功的时候，你的程序就会运行，但你可能几乎没有任何好处。 Streams 不仅仅是一个 API，它还是一个基于函数式编程的范例。 为了获得流必须提供的表现力，速度和某些情况下的并行性，您必须采用范式以及 API。

The most important part of the streams paradigm is to structure your computation as a sequence of transformations where the result of each stage is as close as possible to a pure function of the result of the previous stage. A pure function is one whose result depends only on its input: it does not depend on any mutable state, nor does it update any state. In order to achieve this, any function objects that you pass into stream operations, both intermediate and terminal, should be free of side-effects.

流范例中最重要的部分是将计算结构化为一系列转换，其中每个阶段的结果尽可能接近前一阶段结果的纯函数。 纯函数的结果仅取决于其输入：它不依

赖于任何可变状态，也不更新任何状态。 为了实现这一点，您传递给流操作的任何函数对象（中间和终端）都应该没有副作用。

Occasionally, you may see streams code that looks like this snippet, which builds a frequency table of the words in a text file:

有时，您可能会看到类似于此代码段的流代码，它会在文本文件中构建单词的频率表：

```
// Uses the streams API but not the paradigm--Don't do this!
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
      words.forEach(word -> {
            freq.merge(word.toLowerCase(), 1L, Long::sum);
      });
}
```

What's wrong with this code? After all, it uses streams, lambdas, and method references, and gets the right answer. Simply put, it's not streams code at all; it's iterative code masquerading as streams code. It derives no benefits from the streams API, and it's (a bit) longer, harder to read, and less maintainable than the corresponding iterative code. The problem stems from the fact that this code is doing all its work in a terminal forEach operation, using a lambda that mutates external state (the frequency table). A forEach operation that does anything more than present the result of the computation performed by a stream is a "bad smell in code," as is a lambda that mutates state. So how should this code look?

这段代码出了什么问题？ 毕竟，它使用流，lambdas 和方法引用，并得到正确的答案。 简单地说，它根本不是流代码; 它的迭代代码伪装成流代码。 它没有从流 API 中获益，并且它比相应的迭代代码更长，更难以阅读，并且维护更少。 问题源于这样一个事实：这个代码在一个终端 forEach 操作中完成所有工作，使用一个变异外部状态的 lambda（频率表）。 执行除了呈现流执行的计算结果之外的任何操作的 forEach 操作都是“代码中的难闻气味”，因为是一个变异状态的 lambda。 那么这段代码应该怎么样？

```
// Proper use of streams to initialize a frequency table
Map<String, Long> freq;
```

```
try (Stream<String> words = new Scanner(file).tokens()) {
    freq = words.collect(groupingBy(String::toLowerCase, counting()));
}
```

This snippet does the same thing as the previous one but makes proper use of the streams API. It's shorter and clearer. So why would anyone write it the other way? Because it uses tools they're already familiar with. Java programmers know how to use for-each loops, and the forEach terminal operation is similar. But the forEach operation is among the least powerful of the terminal operations and the least stream-friendly. It's explicitly iterative, and hence not amenable to parallelization. **The forEach operation should be used only to report the result of a stream computation, not to perform the computation.**Occasionally, it makes sense to use forEach for some other purpose, such as adding the results of a stream computation to a preexisting collection.

此代码段与前一代码相同，但正确使用了流 API。 它更短更清晰。 那么为什么有人会用另一种方式写呢？ 因为它使用了他们已经熟悉的工具。 Java 程序员知道如何使用 for-each 循环，而 forEach 终端操作是类似的。 但 forEach 操作是终端操作中最不强大的操作之一，也是最不友好的流操作。 它是明确的迭代，因此不适合并行化。 forEach 操作应该仅用于报告流计算的结果，而不是用于执行计算。有时，将 forEach 用于其他目的是有意义的，例如将流计算的结果添加到预先存在的集合中。

The improved code uses a collector, which is a new concept that you have to learn in order to use streams. The Collectors API is intimidating: it has thirty-nine methods, some of which have as many as five type parameters. The good news is that you can derive most of the benefit from this API without delving into its full complexity. For starters, you can ignore the Collector interface and think of a collector as an opaque object that encapsulates a reduction strategy. In this context, reduction means combining the elements of a stream into a single object. The object produced by a collector is typically a collection (which accounts for the name collector).

改进的代码使用了一个收集器，这是一个新概念，您必须学习才能使用流。 Collectors API 令人生畏：它有三十九种方法，其中一些方法有多达五种类型参数。 好消息是，您可以从这个 API 中获得大部分好处，而无需深入研究其完整的复杂性。 对于初学者，您可以忽略 Collector 接口，并将收集器视为封装缩减

策略的不透明对象。 在这种情况下，缩减意味着将流的元素组合成单个对象。 收集器生成的对象通常是一个集合（它代表名称收集器）。

The collectors for gathering the elements of a stream into a true Collection are straightforward. There are three such collectors: toList(), toSet(), and toCollection(collectionFactory). They return, respectively, a set, a list, and a programmer-specified collection type. Armed with this knowledge, we can write a stream pipeline to extract a top-ten list from our frequency table.

用于将流的元素收集到真正的集合中的收集器是直截了当的。 有三个这样的收集器：toList（），toSet（）和 toCollection（collectionFactory）。 它们分别返回一个集合，一个列表和一个程序员指定的集合类型。 有了这些知识，我们可以编写一个流管道来从频率表中提取前十个列表。

```
// Pipeline to get a top-ten list of words from a frequency table

List<String> topTen = freq.keySet().stream()

.sorted(comparing(freq::get).reversed())

.limit(10)

.collect(toList());
```

Note that we haven’t qualified the toList method with its class, Collectors. **It is customary and wise to statically import all members of Collectors because it makes stream pipelines more readable.**

请注意，我们没有使用其类 Collectors 限定 toList 方法。 静态导入收集器的所有成员是习惯和明智的，因为它使流管道更具可读性。

The only tricky part of this code is the comparator that we pass to sorted, comparing(freq::get).reversed(). The comparing method is a comparator construction method (Item 14) that takes a key extraction function. The function takes a word, and the “extraction” is actually a table lookup: the bound method reference freq::get looks up the word in the frequency table and returns the number of times the word appears in the file. Finally, we call reversed on the comparator, so we’re sorting the words from most frequent to least frequent. Then it’s a simple matter to limit the stream to ten words and collect them into a list.

这段代码中唯一棘手的部分是我们传递给 sorted，compare（freq :: get）.reversed（）的比较器。 比较方法是采用密钥提取功能的比较器构造方法

（第 14 项）。 该函数接受一个单词，“extract”实际上是一个表查找：绑定方法引用 freq :: get 在频率表中查找单词并返回单词在文件中出现的次数。 最后，我们在比较器上调用 reverse，因此我们将单词从最频繁到最不频繁地排序。 然后将流限制为十个单词并将它们收集到一个列表中是一件简单的事情。

The previous code snippets use Scanner’s stream method to get a stream over the scanner. This method was added in Java 9. If you’re using an earlier release, you can translate the scanner, which implements Iterator, into a stream using an adapter similar to the one in Item 47 (streamOf(Iterable)).

之前的代码片段使用 Scanner 的流方法在扫描程序上获取流。 在 Java 9 中添加了此方法。如果您使用的是早期版本，则可以使用类似于第 47 项（streamOf（Iterable））的适配器将实现 Iterator 的扫描程序转换为流。

So what about the other thirty-six methods in Collectors? Most of them exist to let you collect streams into maps, which is far more complicated than collecting them into true collections. Each stream element is associated with a key and a value, and multiple stream elements can be associated with the same key.

那么收藏家的其他 36 种方法呢？ 它们中的大多数存在是为了让您将流收集到地图中，这比将它们收集到真实集合中要复杂得多。 每个流元素与键和值相关联，并且多个流元素可以与相同的键相关联。

The simplest map collector is toMap(keyMapper, valueMapper), which takes two functions, one of which maps a stream element to a key, the other, to a value. We used this collector in our fromString implementation in Item 34 to make a map from the string form of an enum to the enum itself:

最简单的映射收集器是 toMap（keyMapper，valueMapper），它接受两个函数，其中一个函数将一个流元素映射到一个键，另一个函数映射到一个值。 我们在第 34 项的 fromString 实现中使用了这个收集器来创建从枚举的字符串形式到枚举本身的映射：

```
// Using a toMap collector to make a map from string to enum

private static final Map<String, Operation> stringToEnum
=Stream.of(values()).collect(toMap(Object::toString, e -> e));
```

This simple form of toMap is perfect if each element in the stream maps to a unique key. If multiple stream elements map to the same key, the pipeline will terminate with an IllegalStateException.

如果流中的每个元素都映射到唯一键，则这种简单的 toMap 形式是完美的。 如果多个流元素映射到同一个键，则管道将以 IllegalStateException 终止。

The more complicated forms of toMap, as well as the groupingBy method, give you various ways to provide strategies for dealing with such collisions. One way is to provide the toMap method with a merge function in addition to its key and value mappers. The merge function is a BinaryOperator, where V is the value type of the map. Any additional values associated with a key are combined with the existing value using the merge function, so, for example, if the merge function is multiplication, you end up with a value that is the product of all the values associated with the key by the value mapper.

更复杂的 toMap 形式以及 groupingBy 方法为您提供了各种方法来提供处理此类冲突的策略。 一种方法是除了键和值映射器之外，还为 toMap 方法提供合并函数。 合并函数是 BinaryOperator，其中 V 是映射的值类型。 使用合并函数将与键关联的任何其他值与现有值组合，因此，例如，如果合并函数是乘法，则最终得到的值是与键关联的所有值的乘积。 价值映射器。

The three-argument form of toMap is also useful to make a map from a key to a chosen element associated with that key. For example, suppose we have a stream of record albums by various artists, and we want a map from recording artist to best-selling album. This collector will do the job.

toMap 的三参数形式对于创建从键到与该键关联的所选元素的映射也很有用。 例如，假设我们有各种艺术家的唱片专辑流，我们想要一张从录音艺术家到最畅销专辑的地图。 这个收藏家将完成这项工作。

```
// Collector to generate a map from key to chosen element for key
Map<Artist, Album> topHits = albums.collect(
        toMap(Album::artist, a->a, maxBy(comparing(Album::sales)
    )
));
```

Note that the comparator uses the static factory method maxBy, which is statically imported from BinaryOperator. This method converts a Comparator into a BinaryOperator that computes the maximum implied by the specified comparator. In this case, the comparator is returned by the comparator construction method comparing, which takes the key extractor function Album::sales. This may seem a bit

convoluted, but the code reads nicely. Loosely speaking, it says, “convert the stream of albums to a map, mapping each artist to the album that has the best album by sales.” This is surprisingly close to the problem statement.

请注意，比较器使用静态工厂方法 maxBy，它是从 BinaryOperator 静态导入的。 此方法将 Comparator 转换为 BinaryOperator，用于计算指定比较器隐含的最大值。 在这种情况下，比较器由比较器构造方法比较返回，它采用密钥提取器功能 Album :: sales。 这可能看起来有点复杂，但代码读得很好。 简而言之，它说，“将专辑流转换为地图，将每位艺术家映射到销售量最佳专辑的专辑。”这令人惊讶地接近问题陈述。

Another use of the three-argument form of toMap is to produce a collector that imposes a last-write-wins policy when there are collisions. For many streams, the results will be nondeterministic, but if all the values that may be associated with a key by the mapping functions are identical, or if they are all acceptable, this collector’s s behavior may be just what you want:

toMap 的三参数形式的另一个用途是产生一个收集器，当发生冲突时强制执行 last-write-wins 策略。 对于许多流，结果将是不确定的，但如果映射函数可能与键关联的所有值都相同，或者它们都是可接受的，则此收集器的行为可能正是您想要的：

```
// Collector to impose last-write-wins policy

toMap(keyMapper, valueMapper, (v1, v2) -> v2)
```

The third and final version of toMap takes a fourth argument, which is a map factory, for use when you want to specify a particular map implementation such as an EnumMap or a TreeMap.

toMap 的第三个也是最后一个版本采用第四个参数，即一个地图工厂，用于指定特定的地图实现，例如 EnumMap 或 TreeMap。

There are also variant forms of the first three versions of toMap, named toConcurrentMap, that run efficiently in parallel and produce ConcurrentHashMap instances.

toMap 的前三个版本也有变体形式，名为 toConcurrentMap，它们并行高效运行并生成 ConcurrentHashMap 实例。

In addition to the toMap method, the Collectors API provides the groupingBy method, which returns collectors to produce maps that group elements into categories

based on a classifier function. The classifier function takes an element and returns the category into which it falls. This category serves as the element's map key. The simplest version of the groupingBy method takes only a classifier and returns a map whose values are lists of all the elements in each category. This is the collector that we used in the Anagram program in Item 45 to generate a map from alphabetized word to a list of the words sharing the alphabetization:

除了 toMap 方法之外，Collectors API 还提供了 groupingBy 方法，该方法返回收集器以生成基于分类器函数将元素分组到类别中的映射。 分类器函数接受一个元素并返回它所属的类别。 此类别用作元素的地图键。 groupingBy 方法的最简单版本仅采用分类器并返回一个映射，其值是每个类别中所有元素的列表。 这是我们在第 45 项中的 Anagram 程序中使用的收集器，用于生成从按字母顺序排列的单词到共享字母顺序的单词列表的地图：

```
words.collect(groupingBy(word -> alphabetize(word)))
```

If you want groupingBy to return a collector that produces a map with values other than lists, you can specify a downstream collector in addition to a classifier. A downstream collector produces a value from a stream containing all the elements in a category. The simplest use of this parameter is to pass toSet(), which results in a map whose values are sets of elements rather than lists.

如果希望 groupingBy 返回一个生成带有除列表之外的值的映射的收集器，则除了分类器之外，还可以指定下游收集器。 下游收集器从包含类别中所有元素的流生成值。 此参数的最简单用法是传递 toSet（），这将生成一个映射，其值是元素集而不是列表。

Alternatively, you can pass toCollection(collectionFactory), which lets you create the collections into which each category of elements is placed. This gives you the flexibility to choose any collection type you want. Another simple use of the two-argument form of groupingBy is to pass counting() as the downstream collector. This results in a map that associates each category with the number of elements in the category, rather than a collection containing the elements. That's what you saw in the frequency table example at the beginning of this item:

或者，您可以传递 toCollection（collectionFactory），它允许您创建放置每个元素类别的集合。 这使您可以灵活地选择所需的任何集合类型。 另一种简单使用 groupingBy 的双参数形式的方法是将 counting（）作为下游收集器传递。这会生成一个映射，该映射将每个类别与类别中的元素数相关联，而不是包含元素的集合。 这就是您在本项目开头的频率表示例中看到的内容：

```
Map<String, Long> freq = words.collect(groupingBy(String::toLowerCase,
counting()));
```

The third version of groupingBy lets you specify a map factory in addition to a downstream collector. Note that this method violates the standard telescoping argument list pattern: the mapFactory parameter precedes, rather than follows, the downStream parameter. This version of groupingBy gives you control over the containing map as well as the contained collections, so, for example, you can specify a collector that returns a TreeMap whose values are TreeSets.

groupingBy 的第三个版本允许您指定除下游收集器之外的地图工厂。 请注意，此方法违反了标准的 telescoping 参数列表模式：mapFactory 参数位于 downStream 参数之前，而不是之后。 此版本的 groupingBy 使您可以控制包含的映射以及包含的集合，因此，例如，您可以指定一个收集器，该收集器返回值为 TreeSet 的 TreeMap。

The groupingByConcurrent method provides variants of all three overloadings of groupingBy. These variants run efficiently in parallel and produce ConcurrentHashMap instances. There is also a rarely used relative of groupingBy called partitioningBy. In lieu of a classifier method, it takes a predicate and returns a map whose key is a Boolean. There are two overloadings of this method, one of which takes a downstream collector in addition to a predicate.

groupingByConcurrent 方法提供了 groupingBy 的所有三个重载的变体。 这些变体并行高效运行并生成 ConcurrentHashMap 实例。 还有一个很少使用的 grouping 的亲戚叫做 partitioningBy。 代替分类器方法，它接受谓词并返回其键为布尔值的映射。 此方法有两个重载，其中一个除谓词之外还包含下游收集器。

The collectors returned by the counting method are intended only for use as downstream collectors. The same functionality is available directly on Stream, via the count method, so **there is never a reason to say collect(counting()).** There are fifteen more Collectors methods with this property. They include the nine methods whose names begin with summing, averaging, and summarizing (whose functionality is available on the corresponding primitive stream types). They also include all overloadings of the reducing method, and the filtering, mapping, flatMapping, and collectingAndThen methods. Most programmers can safely ignore the majority of these methods. From a design perspective, these collectors represent an attempt to partially duplicate the functionality of streams in collectors so that downstream collectors can act as “ministreams.”

通过计数方法返回的收集器仅用作下游收集器。 通过 count 方法直接在 Stream 上提供相同的功能，因此没有理由说 collect（counting（））。 此属性还有十五种收集器方法。 它们包括九个方法，其名称以求和，平均和汇总开头（其功能在相应的原始流类型上可用）。 它们还包括 reduce 方法的所有重载，以及 filter，mapping，flatMapping 和 collectingAndThen 方法。 大多数程序员可以安全地忽略大多数这些方法。 从设计角度来看，这些收集器代表了尝试在收集器中部分复制流的功能，以便下游收集器可以充当“迷你流”。

There are three Collectors methods we have yet to mention. Though they are in Collectors, they don’t involve collections. The first two are minBy and maxBy, which take a comparator and return the minimum or maximum element in the stream as determined by the comparator. They are minor generalizations of the min and max methods in the Stream interface and are the collector analogues of the binary operators returned by the like-named methods in BinaryOperator. Recall that we used BinaryOperator.maxBy in our best-selling album example.

我们还有三种收藏家方法尚未提及。 虽然他们在收藏家，但他们不涉及收藏。 前两个是 minBy 和 maxBy，它们取比较器并返回由比较器确定的流中的最小或最大元素。 它们是 Stream 接口中 min 和 max 方法的次要推广，是 BinaryOperator 中类似命名方法返回的二元运算符的收集器类似物。 回想一下，我们在最畅销的专辑中使用了 BinaryOperator.maxBy。

The final Collectors method is joining, which operates only on streams of CharSequence instances such as strings. In its parameterless form, it returns a collector that simply concatenates the elements. Its one argument form takes a single CharSequence parameter named delimiter and returns a collector that joins the stream elements, inserting the delimiter between adjacent elements. If you pass in a comma as the delimiter, the collector returns a comma-separated values string (but beware that the string will be ambiguous if any of the elements in the stream contain commas). The three argument form takes a prefix and suffix in addition to the delimiter. The resulting collector generates strings like the ones that you get when you print a collection, for example [came, saw, conquered].

最后的 Collectors 方法是 join，它只对 CharSequence 实例的流进行操作，例如字符串。 在其无参数形式中，它返回一个简单地连接元素的收集器。 它的一个参数形式采用名为 delimiter 的单个 CharSequence 参数，并返回一个连接流元素的收集器，在相邻元素之间插入分隔符。 如果传入逗号作为分隔符，则收集器将返回逗号分隔值字符串（但请注意，如果流中的任何元素包含逗号，则

字符串将不明确）。 除了分隔符之外，三个参数形式还带有前缀和后缀。 生成的收集器会生成类似于打印集合时获得的字符串，例如[来，看到，征服]。

In summary, the essence of programming stream pipelines is side-effect-free function objects. This applies to all of the many function objects passed to streams and related objects. The terminal operation forEach should only be used to report the result of a computation performed by a stream, not to perform the computation. In order to use streams properly, you have to know about collectors. The most important collector factories are toList, toSet, toMap, groupingBy, and joining.

总之，编程流管道的本质是无副作用的功能对象。 这适用于传递给流和相关对象的所有许多函数对象。 终端操作 forEach 仅应用于报告流执行的计算结果，而不是用于执行计算。 为了正确使用流，您必须了解收集器。 最重要的收集器工厂是 toList，toSet，toMap，groupingBy 和 join。

47 返回类型流优先 Collection

Many methods return sequences of elements. Prior to Java 8, the obvious return types for such methods were the collection interfaces Collection, Set, and List; Iterable; and the array types. Usually, it was easy to decide which of these types to return. The norm was a collection interface. If the method existed solely to enable for-each loops or the returned sequence couldn’t be made to implement some Collection method (typically, contains(Object)), the Iterable interface was used. If the returned elements were primitive values or there were stringent performance requirements, arrays were used. In Java 8, streams were added to the platform, substantially complicating the task of choosing the appropriate return type for a sequence-returning method.

许多方法返回元素序列。 在 Java 8 之前，这些方法的明显返回类型是集合接口 Collection，Set 和 List;可迭代; 和数组类型。 通常，很容易决定返回哪些类型。 规范是一个集合界面。 如果该方法仅用于启用 for-each 循环或返回的序列无法实现某些 Collection 方法（通常为 contains（Object）），则使用 Iterable 接口。 如果返回的元素是原始值或者存在严格的性能要求，则使用数组。 在 Java 8 中，流被添加到平台中，这使得为序列返回方法选择适当的返回类型的任务变得非常复杂。

You may hear it said that streams are now the obvious choice to return a sequence of elements, but as discussed in Item 45, streams do not make iteration obsolete: writing good code requires combining streams and iteration judiciously. If

an API returns only a stream and some users want to iterate over the returned sequence with a for-each loop, those users will be justifiably upset. It is especially frustrating because the Stream interface contains the sole abstract method in the Iterable interface, and Stream's specification for this method is compatible with Iterable's. The only thing preventing programmers from using a for-each loop to iterate over a stream is Stream's failure to extend Iterable.

您可能听说过，流现在是返回一系列元素的明显选择，但如第 45 项所述，流不会使迭代过时：编写好的代码需要明智地组合流和迭代。 如果 API 只返回一个流，而某些用户想要使用 for-each 循环迭代返回的序列，那么这些用户将有理由感到不安。 特别令人沮丧的是，Stream 接口包含 Iterable 接口中唯一的抽象方法，Stream 的此方法规范与 Iterable 兼容。 阻止程序员使用 for-each 循环迭代流的唯一因素是 Stream 无法扩展 Iterable。

Sadly, there is no good workaround for this problem. At first glance, it might appear that passing a method reference to Stream's iterator method would work. The resulting code is perhaps a bit noisy and opaque, but not unreasonable:

可悲的是，这个问题没有好的解决方法。 乍一看，似乎可以将方法引用传递给 Stream 的迭代器方法。 结果代码可能有点嘈杂和不透明，但并非不合理：

```
// Won't compile, due to limitations on Java's type inference
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    // Process the process
}
```

Unfortunately, if you attempt to compile this code, you'll get an error message:

不幸的是，如果您尝试编译此代码，您将收到一条错误消息：

```
Test.java:6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
^
```

In order to make the code compile, you have to cast the method reference to an appropriately parameterized Iterable:

为了使代码编译，您必须将方法引用强制转换为适当参数化的 Iterable：

```
// Hideous workaround to iterate over a stream
for (ProcessHandle ph : (Iterable<ProcessHandle>)
        ProcessHandle.allProcesses()::iterator)
```

This client code works, but it is too noisy and opaque to use in practice. A better workaround is to use an adapter method. The JDK does not provide such a method, but it's easy to write one, using the same technique used in-line in the snippets above. Note that no cast is necessary in the adapter method because Java's type inference works properly in this context:

此客户端代码有效，但在实践中使用它太嘈杂和不透明。 更好的解决方法是使用适配器方法。 JDK 没有提供这样的方法，但是使用上面的代码片段中使用的相同技术，可以很容易地编写一个方法。 请注意，在适配器方法中不需要强制转换，因为 Java 的类型推断在此上下文中正常工作：

```
// Adapter from Stream<E> to Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}
```

With this adapter, you can iterate over any stream with a for-each statement:

使用此适配器，您可以使用 for-each 语句迭代任何流：

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
    // Process the process
}
```

Note that the stream versions of the Anagrams program in Item 34 use the Files.lines method to read the dictionary, while the iterative version uses a scanner. The Files.lines method is superior to a scanner, which silently swallows any exceptions encountered while reading the file. Ideally, we would have used Files.lines in the iterative version too. This is the sort of compromise that programmers will make if an API provides only stream access to a sequence and they want to iterate over the sequence with a for-each statement.

请注意，第 34 项中的 Anagrams 程序的流版本使用 Files.lines 方法读取字典，而迭代版本使用扫描程序。 Files.lines 方法优于扫描程序，它可以在读取文件时静默吞下任何异常。 理想情况下，我们也会在迭代版本中使用 Files.lines。 如果 API 仅提供对序列的流访问并且他们希望使用 for-each 语句迭代序列，那么程序员将会做出这种妥协。

Conversely, a programmer who wants to process a sequence using a stream pipeline will be justifiably upset by an API that provides only an Iterable. Again the JDK does not provide an adapter, but it's easy enough to write one:

相反，想要使用流管道处理序列的程序员将完全被仅提供 Iterable 的 API 所打乱。 JDK 再次没有提供适配器，但写一个很容易：

```
// Adapter from Iterable<E> to Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

If you're writing a method that returns a sequence of objects and you know that it will only be used in a stream pipeline, then of course you should feel free to return a stream. Similarly, a method returning a sequence that will only be used for iteration should return an Iterable. But if you're writing a public API that returns a sequence, you should provide for users who want to write stream pipelines as well as those who want to write for-each statements, unless you have a good reason to believe that most of your users will want to use the same mechanism.

如果您正在编写一个返回一系列对象的方法，并且您知道它只会在流管道中使用，那么您当然可以随意返回一个流。 类似地，返回仅用于迭代的序列的方法应返回 Iterable。 但是，如果您正在编写一个返回序列的公共 API，那么您应该为想要编写流管道的用户以及想要为每个语句编写的用户提供，除非您有充分的理由相信大多数 用户希望使用相同的机制。

The Collection interface is a subtype of Iterable and has a stream method, so it provides for both iteration and stream access. Therefore, **Collection or an appropriate subtype is generally the best return type for a public, sequence-returning method.** Arrays also provide for easy iteration and stream access with the Arrays.asList and Stream.of methods. If the sequence you're returning is small enough to fit easily in memory, you're probably best off returning

one of the standard collection implementations, such as ArrayList or HashSet. But **do not store a large sequence in memory just to return it as a collection.**

Collection 接口是 Iterable 的子类型，并且具有 stream 方法，因此它提供迭代和流访问。 因此，Collection 或适当的子类型通常是公共序列返回方法的最佳返回类型。 Arrays 还提供了 Arrays.asList 和 Stream.of 方法的简单迭代和流访问。 如果您返回的序列小到足以容易地放入内存中，那么最好返回一个标准集合实现，例如 ArrayList 或 HashSet。 但是不要在内存中存储大的序列只是为了将它作为集合返回。

If the sequence you're returning is large but can be represented concisely, consider implementing a special-purpose collection. For example, suppose you want to return the power set of a given set, which consists of all of its subsets. The power set of {a, b, c} is {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}. If a set has n elements, its power set has 2n. Therefore, you shouldn't even consider storing the power set in a standard collection implementation. It is, however, easy to implement a custom collection for the job with the help of AbstractList.

如果您返回的序列很大但可以简洁地表示，请考虑实现一个特殊用途的集合。 例如，假设您要返回给定集的幂集，该集包含其所有子集。 {a，b，c}的幂集为{{}，{a}，{b}，{c}，{a，b}，{a，c}，{b，c}，{a，b ， C}}。 如果一个集合具有 n 个元素，则其功率集具有 2n。 因此，您甚至不应考虑将电源设置存储在标准集合实现中。 但是，在 AbstractList 的帮助下，很容易为作业实现自定义集合。

The trick is to use the index of each element in the power set as a bit vector, where the nth bit in the index indicates the presence or absence of the nth element from the source set. In essence, there is a natural mapping between the binary numbers from 0 to 2n − 1 and the power set of an n-element set. Here's the code:

技巧是使用功率集中每个元素的索引作为位向量，其中索引中的第 n 位指示源集合中是否存在第 n 个元素。 本质上，从 0 到 2n - 1 的二进制数和 n 元素集的幂集之间存在自然映射。 这是代码：

```
// Returns the power set of an input set as custom collection
public class PowerSet {
    public static final <E> Collection<Set<E>> of(Set<E> s) {
        List<E> src = new ArrayList<>(s);
```

```
            if (src.size() > 30)
                throw new IllegalArgumentException("Set too big " + s);
            return new AbstractList<Set<E>>() {
                @Override public int size() {
                    return 1 << src.size(); // 2 to the power srcSize
                }
                @Override public boolean contains(Object o) {
                    return o instanceof Set && src.containsAll((Set)o);
                }
                @Override public Set<E> get(int index) {
                    Set<E> result = new HashSet<>();
                    for (int i = 0; index != 0; i++, index >>= 1)
                        if ((index & 1) == 1)
                            result.add(src.get(i));
                    return result;
                }
            };
        }
    }
```

Note that PowerSet.of throws an exception if the input set has more than 30 elements. This highlights a disadvantage of using Collection as a return type rather than Stream or Iterable: Collection has an int-returning size method, which limits the length of the returned sequence to Integer.MAX_VALUE, or 231 − 1. The Collection specification does allow the size method to return 231 − 1 if the collection is larger, even infinite, but this is not a wholly satisfying solution.

请注意，如果输入集具有超过 30 个元素，则 PowerSet.of 会抛出异常。 这突出了使用 Collection 作为返回类型而不是 Stream 或 Iterable 的缺点：Collection

具有 int 返回大小方法，该方法将返回序列的长度限制为 Integer.MAX_VALUE 或 231-1。Collection 规范允许 size 方法返回 231 - 1 如果集合更大，甚至无限，但这不是一个完全令人满意的解决方案。

In order to write a Collection implementation atop AbstractCollection, you need implement only two methods beyond the one required for Iterable: contains and size. Often it's easy to write efficient implementations of these methods. If it isn't feasible, perhaps because the contents of the sequence aren't predetermined before iteration takes place, return a stream or iterable, whichever feels more natural. If you choose, you can return both using two separate methods.

为了在 AbstractCollection 上编写 Collection 实现，您只需要实现 Iterable 所需的两个方法：contains 和 size。 通常，编写这些方法的有效实现很容易。 如果不可行，可能是因为在迭代发生之前未预先确定序列的内容，返回流或可迭代的，无论哪种感觉更自然。 如果选择，您可以使用两种不同的方法返回。

There are times when you'll choose the return type based solely on ease of implementation. For example, suppose you want to write a method that returns all of the (contiguous) sublists of an input list. It takes only three lines of code to generate these sublists and put them in a standard collection, but the memory required to hold this collection is quadratic in the size of the source list. While this is not as bad as the power set, which is exponential, it is clearly unacceptable. Implementing a custom collection, as we did for the power set, would be tedious, more so because the JDK lacks a skeletal Iterator implementation to help us.

有时您会根据易于实施的方式选择退货类型。 例如，假设您要编写一个返回输入列表的所有（连续）子列表的方法。 生成这些子列表只需要三行代码并将它们放在标准集合中，但保存此集合所需的内存是源列表大小的二次方。 虽然这并不像指数的功率集那么糟糕，但显然是不可接受的。 正如我们为电源设置所做的那样，实现自定义集合将是乏味的，因为 JDK 缺乏骨架迭代器实现来帮助我们。

It is, however, straightforward to implement a stream of all the sublists of an input list, though it does require a minor insight. Let's call a sublist that contains the first element of a list a prefix of the list. For example, the prefixes of (a, b, c) are (a), (a, b), and (a, b, c). Similarly, let's call a sublist that contains the last element a suffix, so the suffixes of (a, b, c) are (a, b, c), (b, c), and (c). The insight is that the sublists of a list are simply the suffixes of the prefixes (or identically, the prefixes of the suffixes) and the empty list. This observation leads directly to a clear, reasonably concise implementation:

但是，直接实现输入列表的所有子列表的流，尽管它确实需要一个小的洞察力。 让我们调用一个子列表，该子列表包含列表的第一个元素和列表的前缀。 例如，（a，b，c）的前缀是（a），（a，b）和（a，b，c）。 类似地，让我们调用包含后缀的最后一个元素的子列表，因此（a，b，c）的后缀是（a，b，c），（b，c）和（c）。 洞察力是列表的子列表只是前缀的后缀（或相同的后缀的前缀）和空列表。 这一观察直接导致了一个清晰，合理简洁的实施：

```
// Returns a stream of all the sublists of its input list

public class SubLists {

    public static <E> Stream<List<E>> of(List<E> list) {

        return
Stream.concat(Stream.of(Collections.emptyList()),prefixes(list).flatMap(SubLists::suffixes));

    }

    private static <E> Stream<List<E>> prefixes(List<E> list) {

        return IntStream.rangeClosed(1, list.size()).mapToObj(end ->
list.subList(0, end));

    }

    private static <E> Stream<List<E>> suffixes(List<E> list) {

        return IntStream.range(0, list.size()).mapToObj(start ->
list.subList(start, list.size()));

    }

}
```

Note that the Stream.concat method is used to add the empty list into the returned stream. Also note that the flatMap method (Item 45) is used to generate a single stream consisting of all the suffixes of all the prefixes. Finally, note that we generate the prefixes and suffixes by mapping a stream of consecutive int values returned by IntStream.range and IntStream.rangeClosed. This idiom is, roughly speaking, the stream equivalent of the standard for-loop on integer indices. Thus, our sublist implementation is similar in spirit to the obvious nested for-loop:

请注意，Stream.concat 方法用于将空列表添加到返回的流中。 另请注意，flatMap 方法（第 45 项）用于生成由所有前缀的所有后缀组成的单个流。 最后，

请注意我们通过映射 IntStream.range 和 IntStream.rangeClosed 返回的连续 int 值流来生成前缀和后缀。 粗略地说，这个成语是整数索引上标准 for 循环的流等价物。 因此，我们的子列表实现在精神上类似于明显的嵌套 for 循环：

```
for (int start = 0; start < src.size(); start++)
    for (int end = start + 1; end <= src.size(); end++)
        System.out.println(src.subList(start, end));
```

It is possible to translate this for-loop directly into a stream. The result is more concise than our previous implementation, but perhaps a bit less readable. It is similar in spirit to the streams code for the Cartesian product in Item 45:

可以将此 for 循环直接转换为流。 结果比我们之前的实现更简洁，但可能性稍差。 它在精神上类似于第 45 项中笛卡尔积的流代码：

```
// Returns a stream of all the sublists of its input list
public static <E> Stream<List<E>> of(List<E> list) {
    return IntStream.range(0, list.size())
    .mapToObj(start ->
    IntStream.rangeClosed(start + 1, list.size())
    .mapToObj(end -> list.subList(start, end)))
    .flatMap(x -> x);
}
```

Like the for-loop that precedes it, this code does not emit the empty list. In order to fix this deficiency, you could either use concat, as we did in the previous version, or replace 1 by (int) Math.signum(start) in the rangeClosed call.

与之前的 for 循环一样，此代码不会发出空列表。 为了解决这个问题，您可以使用 concat，就像我们在之前版本中所做的那样，或者在 rangeClosed 调用中用（int）Math.signum（start）替换 1。

Either of these stream implementations of sublists is fine, but both will require some users to employ a Stream-to-Iterable adapter or to use a stream in places where iteration would be more natural. Not only does the Stream-to- Iterable adapter clutter up client code, but it slows down the loop by a factor of 2.3 on my machine. A

purpose-built Collection implementation (not shown here) is considerably more verbose but runs about 1.4 times as fast as our stream-based implementation on my machine.

这些子列表的流实现中的任何一个都很好，但两者都需要一些用户使用 Stream-to-Iterable 适配器或在迭代更自然的地方使用流。 Stream-to-Iterable 适配器不仅使客户端代码混乱，而且还会使我的机器上的循环速度降低 2.3 倍。 专用的 Collection 实现（此处未显示）相当冗长，但运行速度是我机器上基于流的实现的 1.4 倍。

In summary, when writing a method that returns a sequence of elements, remember that some of your users may want to process them as a stream while others may want to iterate over them. Try to accommodate both groups. If it's feasible to return a collection, do so. If you already have the elements in a collection or the number of elements in the sequence is small enough to justify creating a new one, return a standard collection such as ArrayList. Otherwise, consider implementing a custom collection as we did for the power set. If it isn't feasible to return a collection, return a stream or iterable, whichever seems more natural. If, in a future Java release, the Stream interface declaration is modified to extend Iterable, then you should feel free to return streams because they will allow for both stream processing and iteration.

总之，在编写返回元素序列的方法时，请记住，您的某些用户可能希望将它们作为流处理，而其他用户可能希望迭代它们。 尽量适应两个群体。 如果返回集合是可行的，请执行此操作。 如果您已经拥有集合中的元素，或者序列中的元素数量足够小以证明创建新元素，则返回标准集合，例如 ArrayList。 否则，请考虑实施自定义集合，就像我们为电源设置所做的那样。 如果返回集合是不可行的，则返回一个流或可迭代的，无论哪个看起来更自然。 如果在将来的 Java 版本中，Stream 接口声明被修改为扩展 Iterable，那么您应该随意返回流，因为它们将允许流处理和迭代。

48 当创建并行流的时候小心些

Among mainstream languages, Java has always been at the forefront of providing facilities to ease the task of concurrent programming. When Java was released in 1996, it had built-in support for threads, with synchronization and wait/notify. Java 5 introduced the java.util.concurrent library, with concurrent

collections and the executor framework. Java 7 introduced the fork-join package, a high-performance framework for parallel decomposition. Java 8 introduced streams, which can be parallelized with a single call to the parallel method. Writing concurrent programs in Java keeps getting easier, but writing concurrent programs that are correct and fast is as difficult as it ever was. Safety and liveness violations are a fact of life in concurrent programming, and parallel stream pipelines are no exception.

在主流语言中，Java 始终处于提供便于并发编程任务的设施的最前沿。 当 Java 于 1996 年发布时，它内置了对线程的支持，具有同步和等待/通知。 Java 5 引入了 java.util.concurrent 库，包含并发集合和执行器框架。 Java 7 引入了 fork-join 包，这是一个用于并行分解的高性能框架。 Java 8 引入了流，可以通过对并行方法的单个调用来并行化。 用 Java 编写并发程序变得越来越容易，但编写正确快速的并发程序就像以前一样困难。 安全性和活性违规是并发编程中的事实，并行流管道也不例外。

Consider this program from Item 45:

考虑第 45 项中的这个程序：

```
// Stream-based program to generate the first 20 Mersenne primes
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
    .filter(mersenne -> mersenne.isProbablePrime(50))
    .limit(20)
    .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

On my machine, this program immediately starts printing primes and takes 12.5 seconds to run to completion. Suppose I naively try to speed it up by adding a call to parallel() to the stream pipeline. What do you think will happen to its performance?

Will it get a few percent faster? A few percent slower? Sadly, what happens is that it doesn't print anything, but CPU usage spikes to 90 percent and stays there indefinitely (a liveness failure). The program might terminate eventually, but I was unwilling to find out; I stopped it forcibly after half an hour.

在我的机器上，该程序立即开始打印质数，并需要 12.5 秒才能完成运行。假设我天真地试图通过向流管道添加对 parallel（）的调用来加速它。 您认为它的表现会怎样？ 它会加快几个百分点吗？ 几个百分点慢？ 可悲的是，发生的事情是它没有打印任何东西，但是 CPU 使用率飙升至 90％并且无限期地停留在那里（活跃度失败）。 该计划最终可能会终止，但我不愿意发现; 半小时后我强行停了下来。

这里发生了什么？简而言之，流库不知道如何并行化此管道并且启发式失败。即使在最好的情况下，如果源来自 Stream.iterate，或者使用中间操作限制，并行化管道也不太可能提高其性能。这条管道必须应对这两个问题。更糟糕的是，默认的并行化策略通过假设处理一些额外元素并丢弃任何不需要的结果没有任何损害来处理限制的不可预测性。在这种情况下，找到每个 Mersenne prime 需要大约两倍的时间才能找到前一个。因此，计算单个额外元素的成本大致等于计算所有先前元素组合的成本，并且这种无害的管道使自动并行化算法瘫痪。这个故事的寓意很简单：不要无差别地并行化流水线。绩效后果可能是灾难性的。

作为一项规则，并行性的性能增益最好是在 ArrayList，HashMap，HashSet 和 ConcurrentHashMap 实例上的流上;阵列; int 范围; 和远程。 这些数据结构的共同之处在于它们都可以准确且廉价地分成任何所需大小的子范围，这使得在并行线程之间划分工作变得容易。 流库用于执行此任务的抽象是 spliterator，它由 Stream 和 Iterable 上的 spliterator 方法返回。

所有这些数据结构的另一个重要因素是它们在顺序处理时提供了良好到极好的参考局部性：顺序元素引用一起存储在存储器中。 这些引用所引用的对象在存储器中可能彼此不接近，这减少了引用的局部性。 对于并行化批量操作而言，参考位置对于非常重要：如果没有它，线程会将大部分时间用在空闲状态，等待数据从内存传输到处理器的缓存中。 具有最佳参考局部性的数据结构是原始阵列，因为数据本身连续存储在存储器中。

流管道终端操作的性质也会影响并行执行的有效性。 如果与管道的整体工作相比在终端操作中完成了大量工作并且该操作本质上是顺序的，那么并行化管道将具有有限的有效性。 并行性的最佳终端操作是减少，其中从管道中出现的所有元素使用 Stream 的 reduce 方法或预先打包的减少（例如 min，max，count 和 sum）进行组合。 anyMatch，allMatch 和 noneMatch 的短路操作也适用于并

行操作。 Stream 的 collect 方法执行的操作（称为可变约简）不是并行性的良好候选者，因为组合集合的开销很昂贵。

如果您编写自己的 Stream，Iterable 或 Collection 实现并且希望获得良好的并行性能，则必须覆盖 spliterator 方法并广泛测试生成的流的并行性能。 编写高质量的分裂器很困难，超出了本书的范围。

并行化流不仅会导致性能不佳，包括活动失败;它可能导致不正确的结果和不可预测的行为（安全故障）。使用映射器，过滤器和其他程序员提供的不符合其规范的功能对象的管道并行化可能会导致安全故障。 Stream 规范对这些功能对象提出了严格的要求。例如，传递给 Stream 的 reduce 操作的累加器和组合器函数必须是关联的，非干扰的和无状态的。如果您违反了这些要求（其中一些在第 46 项中讨论过），但按顺序运行您的管道，则可能会产生正确的结果;如果你将它并行化，它可能会失败，也许是灾难性的。沿着这些思路，值得注意的是，即使并行化的 Mersenne 素数程序已经完成，它也不会以正确的（升序）顺序打印素数。要保留顺序版本显示的顺序，您必须使用 forEachOrdered 替换 forEach 终端操作，该操作保证以相遇顺序遍历并行流。

即使假设您正在使用有效可拆分的源流，可并行化或廉价的终端操作以及非干扰功能对象，除非管道正在做足够的实际工作以抵消相关成本，否则您将无法从并行化获得良好的加速并行性。作为一个非常粗略的估计，流中元素的数量乘以每个元素执行的代码行数应该至少为十万[Lea14]。

重要的是要记住并行化流是严格的性能优化。与任何优化一样，您必须在更改之前和之后测试性能，以确保它值得做（第 67 项）。理想情况下，您应该在实际的系统设置中执行测试。通常，程序中的所有并行流管道都在公共 fork-join 池中运行。单个行为不当的管道可能会损害系统中不相关部分的其他行为。

如果在并行化流水线时听起来像是堆积在你身上的几率，那是因为它们是。维持数百万行代码库的熟人大量使用流，只发现了平行流有效的少数几个地方。这并不意味着您应该避免并行化流。在适当的情况下，只需通过向流管道添加并行调用，就可以实现处理器内核数量的近线性加速。某些领域，例如机器学习和数据处理，特别适合这些加速。

作为并行性有效的流管道的简单示例，请考虑此函数来计算 π（n），素数小于或等于 n：

```
// Prime-counting stream pipeline - benefits from parallelization

static long pi(long n) {
```

```
        return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
    }
```

在我的机器上，使用此功能计算 π（108）需要 31 秒。 只需添加 parallel（）调用即可将时间缩短为 9.2 秒：

```
    // Prime-counting stream pipeline - parallel version
    static long pi(long n) {
        return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
    }
```

换句话说，并行化计算可以在我的四核机器上将其加速 3.7 倍。值得注意的是，这并不是你在实践中如何计算大 n 值的 π（n）。有更高效的算法，特别是 Lehmer 的公式。

如果要并行化随机数流，请从 SplittableRandom 实例开始，而不是 ThreadLocalRandom（或基本上过时的 Random）。 SplittableRandom 专为此用途而设计，具有线性加速的潜力。 ThreadLocalRandom 设计用于单个线程，并将自身适应作为并行流源，但不会像 SplittableRandom 一样快。随机同步每个操作，因此会导致过度的并行杀戮争用。

总之，除非您有充分的理由相信它将保持计算的正确性并提高其速度，否则甚至不尝试并行化流管道。不恰当地并行化流的成本可能是程序失败或性能灾难。如果您认为并行性可能是合理的，请确保在并行运行时代码保持正确，并在实际条件下进行仔细的性能测量。如果您的代码仍然正确并且这些实验表明您怀疑性能提升，那么只有在生产代码中并行化流。

第八章 方法

本章讨论了方法设计的几个方面：如何处理参数和返回值，如何设计方法签名以及如何记录方法。 本章中的大部分内容适用于构造函数和方法。 与第 4 章一样，本章重点介绍可用性，健壮性和灵活性。

49 检查参数的有效性

大多数方法和构造函数对可以将哪些值传递到其参数中有一些限制。 例如，索引值必须是非负数并且对象引用必须为非 null 并不罕见。 您应该清楚地记录所有这些限制，并在方法主体的开头使用检查来强制执行。 这是一般原则的特例，您应该在发生错误后尽快检测错误。 如果不这样做，则不太可能检测到错误，并且一旦检测到错误就更难确定错误的来源。

如果将无效参数值传递给方法，并且该方法在执行之前检查其参数，则它将以适当的异常快速且干净地失败。 如果方法无法检查其参数，可能会发生一些事情。 在处理过程中，该方法可能会出现令人困惑的异常。 更糟糕的是，该方法可以正常返回，但默默地计算错误的结果。 最糟糕的是，该方法可以正常返回，但是将某个对象置于受损状态，在将来某个未确定的时间在代码中的某些不相关点处导致错误。 换句话说，验证参数失败可能导致违反故障原子性（第 76 项）。

对于公共方法和受保护方法，请使用 Javadoc @throws 标记来记录在违反参数值限制时将引发的异常（第 74 项）。 通常，生成的异常将是 IllegalArgumentException，IndexOutOfBoundsException 或 NullPointerException（Item 72）。 一旦您记录了对方法参数的限制，并且您已经记录了违反这些限制时将抛出的异常，那么强制执行这些限制就很简单了。 这是一个典型的例子：

```
/**
* Returns a BigInteger whose value is (this mod m). This method
* differs from the remainder method in that it always returns a
* non-negative BigInteger.
**
@param m the modulus, which must be positive
* @return this mod m
```

```
    * @throws ArithmeticException if m is less than or equal to 0
    */
    public BigInteger mod(BigInteger m) {
        if (m.signum() <= 0)
            throw new ArithmeticException("Modulus <= 0: " + m);
        ... // Do the computation
    }
```

请注意，如果 m 为 null，则 doc 注释不会说“mod 抛出 NullPointerException”，即使该方法确实如此，也可以作为调用 m.signum（）的副产品。 封闭 BigInteger 类的类级 doc 注释中记录了此异常。 classlevel 注释适用于所有类的公共方法中的所有参数。 这是避免在每个方法上单独记录每个 NullPointerException 的混乱的好方法。 它可以与@Nullable 或类似注释的使用相结合来指示特定参数可以为空，但是这种做法不是标准的，并且为此目的使用了多个注释。

在 Java 7 中添加的 Objects.requireNonNull 方法非常灵活方便，因此没有理由再手动执行空值检查。 如果您愿意，可以指定自己的异常详细消息。 该方法返回其输入，因此您可以在使用值的同时执行空检查：

```
    // Inline use of Java's null-checking facility
    this.strategy = Objects.requireNonNull(strategy, "strategy");
```

您也可以忽略返回值，并使用 Objects.requireNonNull 作为满足您需求的独立空值检查。

在 Java 9 中，范围检查工具被添加到 java.util.Objects 中。 该工具由三个方法组成：checkFromIndexSize，checkFromToIndex 和 checkIndex。 此工具不如空检查方法灵活。 它不允许您指定自己的异常详细消息，它仅用于列表和数组索引。 它不处理闭合范围（包含两个端点）。 但如果它能满足您的需求，那将是一种非常有用的便利。

对于未导出的方法，作为包作者，您可以控制调用该方法的环境，因此您可以而且应该确保只传入有效的参数值。因此，非公共方法可以使用断言检查其参数，如 如下所示：

```
    // Private helper function for a recursive sort
    private static void sort(long a[], int offset, int length) {
```

```
        assert a != null;
        assert offset >= 0 && offset <= a.length;
        assert length >= 0 && length <= a.length - offset;
        ... // Do the computation
    }
```

本质上，这些断言声称断言条件将成立，无论其客户端如何使用封闭包。 与正常的有效性检查不同，断言如果失败则抛出 AssertionError。 与正常的有效性检查不同，它们没有效果，基本上没有成本，除非你启用它们，你可以通过将-ea（或-enableassertions）标志传递给 java 命令来实现。 有关断言的更多信息，请参阅教程[断言]。

检查方法未使用但存储以供以后使用的参数的有效性尤为重要。 例如，考虑第 101 页的静态工厂方法，该方法采用 int 数组并返回数组的 List 视图。 如果客户端传入 null，则该方法将抛出 NullPointerException，因为该方法具有显式检查（对 Objects.requireNonNull 的调用）。 如果省略了检查，则该方法将返回对新创建的 List 实例的引用，该实例将在客户端尝试使用它时立即抛出 NullPointerException。 到那时，List 实例的起源可能很难确定，这可能会使调试任务大大复杂化。

构造函数代表了一个特殊情况，即您应该检查要存储的参数的有效性以供以后使用。 检查构造函数参数的有效性以防止构造违反其不变量的对象至关重要。

规则有例外，您应该在执行计算之前显式检查方法的参数。 一个重要的例外是有效性检查昂贵或不切实际的情况，并且在进行计算的过程中隐式执行检查。 例如，考虑一种对对象列表进行排序的方法，例如 Collections.sort（List）。 列表中的所有对象必须是可相互比较的。 在对列表进行排序的过程中，列表中的每个对象都将与列表中的其他对象进行比较。 如果对象不可相互比较，则这些比较之一将抛出 ClassCastException，这正是 sort 方法应该做的事情。 因此，提前检查列表中的要素是否具有可比性是没有意义的。 但请注意，不加选择地依赖隐式有效性检查会导致失败原子性的丢失（第 76 项）。

有时，计算会隐式执行必需的有效性检查，但如果检查失败则会抛出错误的异常。 换句话说，计算由于无效参数值而自然抛出的异常与记录方法抛出的异常不匹配。 在这些情况下，您应该使用第 73 项中描述的异常翻译惯用法将自然异常转换为正确的异常。

不要从这个项目推断出对参数的任意限制是一件好事。 相反，你应该设计方法，使其尽可能通用。 假设方法可以对它接受的所有参数值做一些合理的操作，那么对参数的限制越少越好。 但是，通常，一些限制是实现抽象的固有特性。

总而言之，每次编写方法或构造函数时，都应该考虑其参数存在哪些限制。您应记录这些限制，并在方法主体的开头使用显式检查来强制执行这些限制。养成这样做的习惯很重要。 它所带来的适度工作将在第一次有效性检查失败时以利息回报。

50 需要时进行保护性拷贝

让 Java 愉快使用的一件事是它是一种安全的语言。这意味着在没有本机方法的情况下，它不受缓冲区溢出，数组溢出，野指针以及其他困扰 C 和 C ++等不安全语言的内存损坏错误的影响。 在一种安全的语言中，无论系统的任何其他部分发生什么，都可以编写类并确切地知道它们的不变量将保持不变。 在将所有内存视为一个巨型阵列的语言中，这是不可能的。

即使是安全的语言，如果没有您的努力，您也不会与其他课程隔离。 你必须采取防御性的方案，假设你们班级的客户会尽力摧毁它的不变量。 随着人们更加努力地破坏系统的安全性，这种情况越来越正确，但更常见的是，您的班级将不得不应对善意程序员诚实错误导致的意外行为。 无论哪种方式，值得花时间编写一些在不良行为客户面前强大的类。

虽然没有对象的帮助，另一个类不可能修改对象的内部状态，但是提供这样的帮助却没有意义，这是非常容易的。 例如，考虑以下类，它声称代表一个不可变的时间段：

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;
    /**
    * @param start the beginning of the period
```

```
        * @param end the end of the period; must not precede start
        * @throws IllegalArgumentException if start is after end
        * @throws NullPointerException if start or end is null
        */
        public Period(Date start, Date end) {
            if (start.compareTo(end) > 0)
                throw new IllegalArgumentException(start + " after " + end);
            this.start = start;
            this.end = end;
        }
        public Date start() {
            return start;
        }
        public Date end() {
            return end;
        }
        ... // Remainder omitted
    }
```

乍一看，这个类似乎是不可变的，并强制执行一个句点的开始不跟随其结束的不变量。 但是，通过利用 Date 是可变的这一事实，很容易违反这个不变量：

```
    // Attack the internals of a Period instance
    Date start = new Date();
    Date end = new Date();
    Period p = new Period(start, end);
    end.setYear(78); // Modifies internals of p!
```

从 Java 8 开始，解决此问题的显而易见的方法是使用 Instant（或 Local-DateTime 或 ZonedDateTime）代替 Date，因为 Instant（和其他 java.time 类）是不可变的（第 17 项）。 日期已过时，不应再在新代码中使用。 也就是说，问题仍然存在：有时您必须在 API 和内部表示中使用可变值类型，并且此项中讨论的技术适用于那些时间。

为了保护 Period 实例的内部免受此类攻击，必须将每个可变参数的防御性副本制作到构造函数，并将副本用作 Period 实例的组件来代替原始文件：

```
// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(this.start + " after " + this.end);
}
```

使用新的构造函数，先前的攻击对 Period 实例没有影响。请注意，在检查参数的有效性之前进行防御性复制（第 49 项），并且对副本而不是原件执行有效性检查。虽然这看似不自然，但这是必要的。它可以在检查参数的时间和复制时间之间的漏洞窗口期间保护类不受其他线程参数的更改。在计算机安全社区，这被称为检查时间/使用时间或 TOCTOU 攻击[Viega01]。

另请注意，我们没有使用 Date 的克隆方法来制作防御性副本。因为 Date 是非最终的，所以不能保证 clone 方法返回一个类为 java.util.Date 的对象：它可以返回一个专门为恶意恶作剧设计的不可信子类的实例。例如，这样的子类可以在创建时记录对私有静态列表中每个实例的引用，并允许攻击者访问该列表。这将使攻击者可以自由控制所有实例。要防止此类攻击，请不要使用 clone 方法制作类型可由不信任方进行子类化的参数的防御副本。

当替换构造函数成功抵御先前的攻击时，仍然可以改变 Period 实例，因为它的访问器提供对其可变内部的访问：

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
```

```
Period p = new Period(start, end);
p.end().setYear(78); // Modifies internals of p!
```

为了抵御第二次攻击，只需修改访问者以返回可变内部字段的防御性副本：

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

使用新的构造函数和新的访问器，Period 是真正不可变的。无论程序员多么恶意或无能，根本没有办法违反一个句号的开头不跟随其结束的不变量（不使用诸如本机方法和反射之类的语言学手段）。这是正确的，因为除了 Period 本身之外的任何类都无法访问 Period 实例中的任何可变字段。这些字段真正封装在对象中。

在访问器中，与构造函数不同，允许使用克隆方法来制作防御性副本。这是因为我们知道 Period 的内部 Date 对象的类是 java.util.Date，而不是一些不受信任的子类。也就是说，由于第 13 项中概述的原因，通常最好使用构造函数或静态工厂来复制实例。

防御性复制参数不仅适用于不可变类。每当您编写一个方法或构造函数来存储对内部数据结构中客户端提供的对象的引用时，请考虑客户端提供的对象是否可能是可变的。如果是，请考虑在将对象输入数据结构后，您的类是否可以容忍对象的更改。如果答案为否，则必须防御性地复制对象并将副本输入数据结构中以代替原始对象。例如，如果您正在考虑使用客户端提供的对象引用作为内部 Set 实例中的元素或作为内部 Map 实例中的键，您应该知道如果对象的集合或映射的不变量将被破坏插入后进行了修改。

内部组件在将其返回给客户端之前进行防御性复制也是如此。无论您的类是否是不可变的，在返回对可变内部组件的引用之前，您应该三思而后行。机会是，你应该返回防御性副本。请记住，非零长度数组总是可变的。因此，在

将内部数组返回给客户端之前，应始终制作内部数组的防御副本。或者，您可以返回数组的不可变视图。这两种技术都在第 15 项中示出。

可以说，所有这一切的真正教训是，在可能的情况下，您应该使用不可变对象作为对象的组件，这样您就不必担心防御性复制（第 17 项）。在我们的 Period 示例的情况下，使用 Instant（或 LocalDateTime 或 ZonedDateTime），除非您使用的是 Java 8 之前的版本。如果您使用的是早期版本，则一个选项是存储 Date.getTime 返回的原始 long。 （）代替 Date 引用。

可能存在与防御性复制相关的性能损失，并且它并不总是合理的。如果一个类信任其调用者不修改内部组件，可能是因为该类及其客户端都是同一个包的一部分，那么放弃防御性复制可能是适当的。在这些情况下，类文档应明确调用者不得修改受影响的参数或返回值。

即使跨越包边界，在将可变参数集成到对象之前制作可变参数的防御副本并不总是合适的。有一些方法和构造函数，其调用指示参数引用的对象的显式切换。在调用这样的方法时，客户端承诺它将不再直接修改对象。希望获得客户端提供的可变对象所有权的方法或构造函数必须在其文档中明确说明。

包含调用指示控制转移的方法或构造函数的类不能防御恶意客户端。只有当一个类与其客户之间存在相互信任，或者当对类的不变量造成损害时，除了客户之外，任何人都不会受到损害。后一种情况的一个例子是包装类模式（第 18 项）。根据包装类的性质，客户端可以通过在包装后直接访问对象来销毁类的不变量，但这通常只会损害客户端。

总之，如果一个类具有从其客户端获取或返回其客户端的可变组件，则该类必须防御性地复制这些组件。如果副本的成本过高并且该类信任其客户不要不恰当地修改组件，则可以用文档替换防御性副本，该文档概述了客户不负责修改受影响组件的责任。

51 小心设计方法签名

这个项目是一个 API 设计提示的抓包，不值得他们自己的项目。 总之，它们将有助于使您的 API 更易于学习和使用，并且不易出错。

仔细选择方法名称。名称应始终遵守标准命名约定（第 68 项）。您的主要目标应该是选择可理解且与同一包中的其他名称一致的名称。您的次要目标应该是选择与更广泛的共识一致的名称。避免使用长方法名称。如有疑问，请查

看 Java 库 API 以获取指导。虽然存在许多不一致 - 不可避免的，考虑到这些图书馆的规模和范围 - 还有相当多的共识。

不要过分提供便利方法。每种方法都应该“拉动它的重量。”太多的方法使得课程难以学习，使用，记录，测试和维护。对于接口而言，这是双重的，因为太多的方法使实现者和用户的生活变得复杂。对于您的类或接口支持的每个操作，请提供功能完备的方法。只有在经常使用时才考虑提供“速记”。如有疑问，请将其删除。

避免使用长参数列表。瞄准四个或更少的参数。大多数程序员都记不起更长的参数列表。如果您的许多方法超出此限制，则在未经常引用其文档的情况下，您的 API 将无法使用。现代 IDE 有所帮助，但使用简短的参数列表仍然会有所改善。长序列的相同类型的参数尤其有害。用户不仅不能记住参数的顺序，而且当他们意外地转换参数时，他们的程序仍然可以编译和运行。他们只是不会做他们的作者的意图。

有三种技术可以缩短过长的参数列表。一种方法是将方法分解为多种方法，每种方法只需要一部分参数。如果不小心完成，这可能导致太多方法，但它也可以通过增加正交性来帮助减少方法计数。例如，考虑 java.util.List 接口。它没有提供查找子列表中元素的第一个或最后一个索引的方法，这两个索引都需要三个参数。相反，它提供了 subList 方法，该方法接受两个参数并返回子列表的视图。此方法可以与 indexOf 或 lastIndexOf 方法结合使用，每个方法都有一个参数，以产生所需的功能。此外，subList 方法可以与在 List 实例上操作的任何方法组合，以对子列表执行任意计算。得到的 API 具有非常高的功率重量比。

缩短长参数列表的第二种技术是创建辅助类来保存参数组。通常，这些辅助类是静态成员类（第 24 项）。如果看到频繁出现的参数序列代表某个不同的实体，则建议使用此技术。例如，假设您正在编写一个代表纸牌游戏的类，并且您发现自己经常传递一系列两个参数来表示卡的等级及其套装。如果添加了一个辅助类来表示一个卡并用一个辅助类的参数替换参数序列的每一个参数，那么您的 API 以及类的内部结构可能会受益。

结合前两个方面的第三种技术是使 Builder 模式（第 2 项）从对象构造适应方法调用。如果你有一个包含许多参数的方法，特别是如果它们中的一些是可选的，那么定义一个表示所有参数的对象并允许客户端对该对象进行多次“setter”调用是有益的，每个参数都是设置单个参数或小的相关组。一旦设置了所需的参数，客户端就会调用对象的“执行”方法，该方法对参数进行任何最终有效性检查并执行实际计算。

对于参数类型，优先于类的接口（Item 64）。 如果有适当的接口来定义参数，请使用它来支持实现接口的类。 例如，没有理由编写一个在输入使用 Map 上使用 HashMap 的方法。 这使您可以传入 HashMap，TreeMap，ConcurrentHashMap，TreeMap 的子图或任何尚未编写的 Map 实现。 通过使用类而不是接口，可以将客户端限制为特定的实现，并在输入数据恰好以其他形式存在时强制执行不必要且可能很昂贵的复制操作。

除非从方法名称中清除布尔值的含义，否则首选两元素枚举类型为布尔参数。 枚举使您的代码更易于阅读和编写。 此外，它们可以让以后轻松添加更多选项。 例如，您可能有一个具有静态工厂的温度计类型，该工厂采用此枚举：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Thermometer.newInstance（TemperatureScale.CELSIUS）不仅比 Thermometer.newInstance（true）更有意义，而且您可以在将来的版本中将 KELVIN 添加到 TemperatureScale，而无需向 Thermometer 添加新的静态工厂。 此外，您可以将温度计依赖关系重构为枚举常量的方法（第 34 项）。 例如，每个缩放常量可以有一个采用 double 值并将其转换为 Celsius 的方法。

52 谨慎使用重载

以下程序是一种善意的尝试，根据集合是集合，列表还是其他类型的集合对集合进行分类：

```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }
    public static String classify(List<?> lst) {
        return "List";
    }
    public static String classify(Collection<?> c) {
```

```
            return "Unknown Collection";
        }
        public static void main(String[] args) {
            Collection<?>[] collections = {
                new HashSet<String>(),new ArrayList<BigInteger>(),new
HashMap<String, String>().values()
            };
            for (Collection<?> c : collections)
                System.out.println(classify(c));
        }
    }
```

您可能希望此程序打印 Set，然后是 List 和 Unknown Collection，但它不会。它打印三次 Unknown Collection。 为什么会这样？ 因为 classify 方法被重载，所以在编译时选择要调用的重载。 对于循环的所有三次迭代，参数的编译时类型是相同的：集合。 运行时类型在每次迭代中都不同，但这不会影响重载的选择。因为参数的编译时类型是Collection，所以唯一适用的重载是第三个，classify（Collection <？ >），并且在循环的每次迭代中调用这个重载。

此程序的行为是违反直觉的，因为重载方法之间的选择是静态的，而重写方法之间的选择是动态的。 根据调用方法的对象的运行时类型，在运行时选择正确版本的重写方法。 作为提醒，当子类包含与祖先中的方法声明具有相同签名的方法声明时，将覆盖方法。 如果在子类中重写实例方法并且在子类的实例上调用此方法，则无论子类实例的编译时类型如何，都会执行子类的重写方法。为了具体，请考虑以下程序：

```
    class Wine {
        String name() { return "wine"; }
    }
    class SparklingWine extends Wine {
        @Override
        String name() { return "sparkling wine"; }
```

```
    }
    class Champagne extends SparklingWine {
        @Override
        String name() { return "champagne"; }
    }
    public class Overriding {
        public static void main(String[] args) {
            List<Wine> wineList = List.of(new Wine(), new SparklingWine(),
new Champagne());
        for (Wine wine : wineList)
            System.out.println(wine.name());
        }
    }
```

name 方法在 Wine 类中声明，并在子类 SparklingWine 和 Champagne 中重写。正如您所料，此程序打印出葡萄酒，起泡酒和香槟，即使实例的编译时类型在循环的每次迭代中都是 Wine。当调用重写方法时，对象的编译时类型对执行哪个方法没有影响;总是会执行“最具体”的重写方法。将此与重载进行比较，其中对象的运行时类型对执行的重载没有影响;选择是在编译时完成的，完全基于参数的编译时类型。

在 CollectionClassifier 示例中，程序的目的是通过基于参数的运行时类型自动调度到适当的方法重载来辨别参数的类型，就像 Wine 方法中的名称方法一样。方法重载根本不提供此功能。假设需要一个静态方法，修复 CollectionClassifier 程序的最佳方法是用一个执行显式 instanceof 测试的方法替换 classify 的所有三个重载：

```
    public static String classify(Collection<?> c) {
        return c instanceof Set ? "Set" :c instanceof List ? "List" : "Unknown
Collection";
    }
```

因为覆盖是常态，而重载是例外，所以覆盖设置了人们对方法调用行为的期望。正如 CollectionClassifier 示例所示，重载很容易混淆这些期望。编写行为可能会使程序员感到困惑的代码是不好的做法。对于 API 尤其如此。如果 API 的典型用户不知道将为给定的参数集调用多个方法重载中的哪一个，则使用 API 可能会导致错误。这些错误很可能表现为运行时的不稳定行为，许多程序员很难诊断它们。因此，您应该避免混淆使用重载。

对于一些混淆使用重载的确切原因是一些争论。安全，保守的策略永远不会导出具有相同数量参数的两个过载。如果方法使用 varargs，保守策略根本不会使其超载，除非在第 53 项中描述。如果遵守这些限制，程序员将永远不会怀疑哪些重载适用于任何实际参数集。这些限制并不是非常繁重，因为您始终可以为方法提供不同的名称而不是重载它们。

例如，考虑 ObjectOutputStream 类。它为每种基本类型和几种引用类型提供了 write 方法的变体。这些变体不是重载 write 方法，而是具有不同的名称，例如 writeBoolean（boolean），writeInt（int）和 writeLong（long）。与重载相比，此命名模式的另一个好处是可以为读取方法提供相应的名称，例如 readBoolean（），readInt（）和 readLong（）。事实上，ObjectInputStream 类提供了这样的读取方法。

对于构造函数，您无法使用不同的名称：类的多个构造函数总是被重载。在许多情况下，您可以选择导出静态工厂而不是构造函数（第 1 项）。此外，使用构造函数，您不必担心重载和重写之间的交互，因为构造函数不能被覆盖。您可能有机会导出具有相同数量参数的多个构造函数，因此知道如何安全地执行它是值得的。

如果始终清楚哪些重载将适用于任何给定的实际参数集，则使用相同数量的参数导出多个过载不太可能使程序员感到困惑。当两对过载中的至少一个相应的形式参数在两个过载中具有“完全不同”类型时就是这种情况。如果显然不可能将任何非空表达式转换为两种类型，则两种类型完全不同。在这些情况下，哪些重载适用于给定的一组实际参数完全由参数的运行时类型决定，并且不受其编译时类型的影响，因此混淆的主要原因消失了。例如，ArrayList 有一个带有 int 的构造函数和带有 Collection 的第二个构造函数。很难想象在任何情况下都会混淆这两个构造函数中的哪一个。

在 Java 5 之前，所有原始类型都与所有引用类型完全不同，但在自动装箱存在的情况下并非如此，并且它已经造成了真正的麻烦。考虑以下程序：

```
public class SetList {

public static void main(String[] args) {
```

```
        Set<Integer> set = new TreeSet<>();
        List<Integer> list = new ArrayList<>();
        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set +""+list);
        }
    }
```

首先，程序将从-3 到 2 的整数添加到有序集和列表中。然后，它在集合和列表上进行三次相同的调用。如果你和大多数人一样，你希望程序从集合和列表中删除非负值（0,1 和 2）并打印[-3，-2，-1] [- 3，-2，-1]。实际上，程序从集合中删除非负值，从列表中删除奇数值并打印[-3，-2，-1] [-2,0,2]。称这种行为令人困惑是一种保守的说法。

这是正在发生的事情：调用 set.remove（i）选择重载 remove（E），其中 E 是集合的元素类型（Integer），以及从 int 到 Integer 的 autobox。这是您期望的行为，因此程序最终会从集合中删除正值。另一方面，对 list.remove（i）的调用选择重载 remove（int i），它删除列表中指定位置的元素。如果你从列表[-3，-2，-1,0,1,2]开始并删除第 0 个元素，那么第一个，然后是第二个，你留下[-2,0,2] ，这个谜就解决了。要解决此问题，请将 list.remove 的参数强制转换为 Integer，强制选择正确的重载。或者，您可以在 i 上调用 Integer.valueOf 并将结果传递给 list.remove。无论哪种方式，程序按预期打印[-3，-2，-1] [-3，-2，-1]：

```
    for (int i = 0; i < 3; i++) {
        set.remove(i);
        list.remove((Integer) i); // or remove(Integer.valueOf(i))
```

```
}
```

上一个示例演示的混乱行为是因为List接口有两个remove方法：remove（E）和remove（int）。 在Java 5之前，当List接口被“生成”时，它有一个remove（Object）方法代替remove（E），相应的参数类型Object和int完全不同。 但是在存在泛型和自动装箱的情况下，这两种参数类型不再完全不同。 换句话说，向语言添加泛型和自动装箱会损坏List界面。 幸运的是，很少有Java库中的其他API被类似地损坏，但是这个故事清楚地表明自动装箱和泛型增加了重载时注意的重要性。 在Java 8中添加lambda和方法引用进一步增加了重载混淆的可能性。 例如，考虑以下两个片段：

```
new Thread(System.out::println).start();

ExecutorService exec = Executors.newCachedThreadPool();

exec.submit(System.out::println);
```

虽然Thread构造函数调用和submit方法调用看起来类似，但前者编译而后者不编译。参数是相同的（System.out :: println），构造函数和方法都有一个带有Runnable的重载。这里发生了什么？令人惊讶的答案是，submit方法有一个带有Callable的重载，而Thread构造函数却没有。您可能认为这不应该有任何区别，因为println的所有重载都返回void，因此方法引用不可能是Callable。这很有道理，但这不是重载解析算法的工作方式。也许同样令人惊讶的是，如果println方法也没有重载，则submit方法调用将是合法的。它是引用方法（println）和调用方法（submit）的重载的组合，它可以防止重载决策算法按照您的预期运行。

从技术上讲，问题是System.out :: println是一个不精确的方法引用[JLS，15.13.1]，并且“包含隐式类型的lambda表达式或不精确的方法引用的某些参数表达式被适用性测试忽略，因为它们的在选择目标类型之前无法确定意义[JLS，15.12.2]。“如果你不理解这段经文，请不要担心;它针对的是编译器编写者。关键是在同一参数位置中具有不同功能接口的重载方法或构造函数会导致混淆。因此，不要重载方法在同一参数位置采用不同的功能接口。在这个项目的说法中，不同的功能接口并没有根本不同。如果你传递命令行开关--Xlint：overloads，Java编译器会警告你这种有问题的重载。

除Object之外的数组类型和类类型完全不同。此外，Serializable和Cloneable以外的数组类型和接口类型完全不同。如果两个类都不是另一个类的后代，那么两个不同的类被认为是无关的[JLS，5.5]。例如，String和Throwable是无关的。任何对象都不可能是两个不相关的类的实例，因此不相关的类也是根本不同的。

还有其他一对类型无法在任何一个方向上进行转换[JLS，5.1.12]，但是一旦超出上述简单情况，大多数程序员就很难辨别哪些（如果有的话）重载适用一组实际参数。确定选择哪个重载的规则非常复杂，并且每个版本都会变得更加复杂。很少有程序员能够理解他们所有的微妙之处。

有时您可能觉得有必要违反本条款中的指导原则，特别是在发展现有课程时。例如，考虑 String，它具有自 Java 4 以来的 contentEquals（StringBuffer）方法。在 Java 5 中，添加了 CharSequence 以提供 StringBuffer，StringBuilder，String，CharBuffer 和其他类似类型的公共接口。在添加 CharSequence 的同时，String 配备了带有 CharSequence 的 contentEquals 方法的重载。

虽然产生的重载明显违反了此项中的指导原则，但它不会造成任何损害，因为重载方法在同一对象引用上调用时会执行完全相同的操作。程序员可能不知道将调用哪个重载，但只要它们的行为相同，它就没有任何意义。确保此行为的标准方法是将更具体的重载转发到更一般的：

```
// Ensuring that 2 methods have identical behavior by forwarding

public boolean contentEquals(StringBuffer sb) {

    return contentEquals((CharSequence) sb);

}
```

虽然 Java 库很大程度上遵循了这个项目中的建议精神，但是有许多类违反了它。例如，String 导出两个重载的静态工厂方法 valueOf（char []）和 valueOf（Object），它们在传递相同的对象引用时会执行完全不同的操作。对此没有任何正当理由，它应被视为可能存在真正混淆的异常现象。

总结一下，仅仅因为你可以重载方法并不意味着你应该。通常最好不要使用具有相同参数数量的多个签名来重载方法。在某些情况下，尤其是涉及构造函数的情况下，可能无法遵循此建议。在这些情况下，您应该至少避免通过添加强制转换将相同的参数集传递给不同的重载的情况。如果无法避免这种情况，例如，因为您正在改进现有类以实现新接口，则应确保在传递相同参数时所有重载的行为都相同。如果你不这样做，程序员将很难有效地使用重载方法或构造函数，他们将无法理解为什么它不起作用。

53 谨慎可变参数

Varargs 方法，正式称为变量 arity 方法[JLS，8.4.1]，接受零个或多个指定类型的参数。 varargs 工具首先创建一个数组，其大小是在调用站点传递的参数数量，然后将参数值放入数组中，最后将数组传递给方法。

例如，这是一个 varargs 方法，它接受一系列 int 参数并返回它们的总和。 正如您所料，sum（1,2,3）的值为 6，sum（）的值为 0：

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

有时，编写一个需要某种类型的一个或多个参数的方法是合适的，而不是零或更多。 例如，假设您要编写一个计算其参数最小值的函数。 如果客户端不传递任何参数，则此函数定义不明确。 您可以在运行时检查数组长度：

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
    min = args[i];
    return min;
}
```

该解决方案存在几个问题。 最严重的是，如果客户端在没有参数的情况下调用此方法，则它在运行时而不是编译时失败。 另一个问题是它很难看。 您必须在 args 上包含显式有效性检查，除非将 min 初始化为 Integer.MAX_VALUE，否则不能使用 for-each 循环，这也很难看。

幸运的是，有一种更好的方法可以达到预期的效果。 声明方法采用两个参数，一个指定类型的普通参数和一个此类型的 varargs 参数。 该解决方案纠正了前一个方面的所有缺陷：

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
    min = arg;
    return min;
}
```

正如您在此示例中所看到的，varargs 在您需要具有可变数量参数的方法的情况下非常有效。 Varargs 专为 printf 设计，与 varargs 同时添加到平台，以及改装后的核心反射设施（项目 65）。 printf 和反射都从 varargs 中获益匪浅。

在性能危急情况下使用 varargs 时要小心。 每次调用 varargs 方法都会导致数组分配和初始化。 如果你根据经验确定你不能承担这笔费用，但你需要 varargs 的灵活性，有一种模式可以让你吃蛋糕并吃掉它。 假设您已确定 95％的方法调用具有三个或更少的参数。 然后声明方法的五个重载，一个用零到三个普通参数，并且当参数个数超过三个时使用单个 varargs 方法：

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

现在您知道，只有在参数数量超过 3 的所有调用的 5%中，您才需要支付数组创建的成本。 像大多数性能优化一样，这种技术通常是不合适的，但是当它成功时，它就是救星。

EnumSet 的静态工厂使用此技术将创建枚举集的成本降至最低。 这是合适的，因为枚举集为比特字段提供了具有性能竞争力的替代品至关重要（第 36 项）。

总之，当您需要使用可变数量的参数定义方法时，varargs 非常有用。 在 varargs 参数前加上任何必需的参数，并注意使用 varargs 的性能后果。

54 返回空集合或者数组，而不是 null

It is not uncommon to see methods that look something like this:

看到类似这样的方法并不罕见：

```
// Returns null to indicate an empty collection. Don't do this!
private final List<Cheese> cheesesInStock = ...;
/**
* @return a list containing all of the cheeses in the shop,
* or null if no cheeses are available for purchase.
*/
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null: new ArrayList<>(cheesesInStock);
}
```

在没有奶酪可供购买的情况下，没有理由特殊情况。 这样做需要客户端中的额外代码来处理可能为 null 的返回值，例如：

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

几乎每次使用返回 null 代替空集合或数组的方法都需要这种绕行。它容易出错，因为编写客户端的程序员可能忘记编写特殊情况代码来处理 null 返回。多年来这种错误可能会被忽视，因为这种方法通常会返回一个或多个对象。此外，返回 null 代替空容器会使返回容器的方法的实现变得复杂。

有时候认为空返回值比空集合或数组更可取，因为它避免了分配空容器的费用。这个论点在两个方面失败了。首先，除非测量结果表明所讨论的分配是性能问题的真正原因，否则不宜担心此级别的性能（第 67 项）。其次，可以在不分配空集合和数组的情况下返回它们。以下是返回可能为空的集合的典型代码。通常，这就是您所需要的：

```
//The right way to return a possibly empty collection
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

如果您有证据表明分配空集合会损害性能，则可以通过重复返回相同的不可变空集合来避免分配，因为不可变对象可以自由共享（第 17 项）。 下面是使用 Collections.emptyList 方法执行此操作的代码。 如果你要返回一个集合，你将使用 Collections.emptySet; 如果您要返回地图，则使用 Collections.emptyMap。但请记住，这是一种优化，而且很少需要它。 如果您认为需要它，请测量前后的性能，以确保它实际上有所帮助：

```
// Optimization - avoids allocating empty collections
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? Collections.emptyList(): new
ArrayList<>(cheesesInStock);
}
```

数组的情况与集合的情况相同。 永远不要返回 null 而不是零长度数组。 通常，您应该只返回一个正确长度的数组，该数组可能为零。 请注意，我们将一个零长度数组传递给 toArray 方法以指示所需的返回类型，即 Cheese []：

```
//The right way to return a possibly empty array
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(new Cheese[0]);
```

```
}
```

如果您认为分配零长度数组会损害性能，则可以重复返回相同的零长度数组，因为所有零长度数组都是不可变的：

```
// Optimization - avoids allocating empty arrays
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

在优化版本中，我们将相同的空数组传递到每个 toArray 调用中，并且每当 cheesesInStock 为空时，此数组将从 getCheeses 返回。 不要预先分配传递给 toArray 的数组，以期提高性能。 研究表明它适得其反[Shipilëv16]：

```
// Don’t do this - preallocating the array harms performance!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

总之，永远不会返回 null 来代替空数组或集合。 它使您的 API 更难以使用并且更容易出错，并且它没有性能优势。

55 谨慎返回 Optionals @

在 Java 8 之前，在编写在某些情况下无法返回值的方法时，可以采用两种方法。 您可以抛出异常，也可以返回 null（假设返回类型是对象引用类型）。 这些方法都不完美。 应该为异常条件保留异常（第 69 项），抛出异常是很昂贵的，因为在创建异常时会捕获整个堆栈跟踪。 返回 null 没有这些缺点，但它有自己的缺点。 如果方法返回 null，则客户端必须包含特殊情况代码以处理返回 null 的可能性，除非程序员可以证明无法返回 null。 如果客户端忽略检查空返回并在某些数据结构中存储空返回值，则 NullPointerException 可能在将来的某个任意时间导致，在代码中与该问题无关的某个位置。

在 Java 8 中，有第三种方法来编写可能无法返回值的方法。 Optional 类表示一个不可变容器，它可以包含单个非空 T 引用，也可以不包含任何内容。 一个包含任何内容的可选项被认为是空的。 据说一个值存在于非空的可选项中。 可选的本质上是一个不可变的集合，最多可以容纳一个元素。 可选不实现 Collection，但原则上可以。

在某些情况下，概念上返回 T 但可能无法执行此操作的方法可以声明为返回 Optional。 这允许该方法返回空结果以指示它无法返回有效结果。 Optional-returning 方法比抛出异常的方法更灵活，更易于使用，并且比返回 null 的方法更不容易出错。

在第 30 项中，我们展示了这种方法，根据元素的自然顺序计算集合中的最大值。

```
// Returns maximum value in collection - throws exception if empty
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");
    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
    result = Objects.requireNonNull(e);
    return result;
}
```

如果给定集合为空，则此方法抛出 IllegalArgumentException。 我们在第 30 项中提到，更好的选择是返回 Optional。 以下是修改它时方法的外观：

```
// Returns maximum value in collection as an Optional<E>
public static <E extends Comparable<E>> Optional<E> max(Collection<E> c)
{
    if (c.isEmpty())
        return Optional.empty();
    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
    result = Objects.requireNonNull(e);
```

```
        return Optional.of(result);
    }
```

如您所见，返回可选项很简单。 您所要做的就是使用适当的静态工厂创建可选项。 在这个程序中，我们使用两个：Optional.empty（）返回一个空的可选项，Optional.of（value）返回一个包含给定非 null 值的可选项。 将 null 传递给 Optional.of（value）是一个编程错误。 如果这样做，该方法通过抛出 NullPointerException 来响应。 Optional.ofNullable（value）方法接受一个可能为 null 的值，如果传入 null 则返回一个空的可选项。永远不要从 Optional-returning 方法返回一个空值：它会破坏工具的整个目的。

流上的许多终端操作返回选项。 如果我们重写 max 方法以使用流，Stream 的 max 操作会为我们生成一个可选项（尽管我们必须传入一个显式比较器）：

```
    // Returns max val in collection as Optional<E> - uses stream
    public static <E extends Comparable<E>> Optional<E> max(Collection<E> c)
{
        return c.stream().max(Comparator.naturalOrder());
    }
```

那么如何选择返回可选而不是返回 null 或抛出异常？ Optionals 在精神上类似于检查异常（第 71 项），因为它们迫使 API 的用户面对可能没有返回值的事实。 抛出未经检查的异常或返回 null 允许用户忽略此可能性，并可能产生可怕的后果。 但是，抛出已检查的异常需要在客户端中添加额外的样板代码。

如果方法返回一个可选项，则客户端可以选择在方法无法返回值时要采取的操作。 您可以指定默认值：

```
    // Using an optional to provide a chosen default value
    String lastWordInLexicon = max(words).orElse("No words...");
```

或者您可以抛出任何适当的异常。 请注意，我们传入异常工厂而不是实际异常。 除非实际抛出异常，否则这将避免创建异常的开销：

```
    // Using an optional to throw a chosen exception
    Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

如果你可以证明一个 optional 是非空的，那么你可以从 Optional 中获取值，而不指定当 optional 是空的时要采取的操作，但是如果你错了，你的代码将抛出 NoSuchElementException：

```
// Using optional when you know there's a return value

Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

有时您可能会遇到这样的情况，即获取默认值很昂贵，并且除非必要，否则您希望避免这种成本。 对于这些情况，Optional 提供了一种方法，该方法接受供应商并仅在必要时调用它。 这个方法叫做 orElseGet，但也许它应该被称为 orElseCompute，因为它与名称以 compute 开头的三个 Map 方法密切相关。 有几种可选方法可用于处理更专业的用例：filter，map，flatMap 和 ifPresent。 在 Java 9 中，添加了另外两个方法：或者 ifPresentOrElse。 如果上述基本方法与您的用例不匹配，请查看这些更高级方法的文档，看看它们是否能完成这项工作。

如果这些方法都不符合您的需求，Optional 提供了 isPresent（）方法，可以将其视为安全阀。 如果可选项包含值，则返回 true;如果为空，则返回 false。 您可以使用此方法在可选结果上执行您喜欢的任何处理，但请确保明智地使用它。 isPresent 的许多用途可以有利地被上面提到的方法之一取代。 生成的代码通常更短，更清晰，更具惯用性。

例如，请考虑此代码段，它打印进程父进程的进程 ID，如果进程没有父进程，则为 N / A. 该代码段使用 Java 9 中引入的 ProcessHandle 类：

```
Optional<ProcessHandle> parentProcess = ph.parent();

System.out.println("Parent PID: " + (parentProcess.isPresent() ?

String.valueOf(parentProcess.get().pid()) : "N/A"));
```

上面的代码片段可以替换为使用 Optional 的 map 函数的代码片段：

```
System.out.println("Parent PID: " + ph.parent().map(h ->
String.valueOf(h.pid())).orElse("N/A"));
```

使用流进行编程时，发现自己使用 Stream <Optional>并要求包含非空选项中的所有元素的 Stream 以继续进行并不罕见。 如果你正在使用 Java 8，那么这里是如何弥合差距：

```
streamOfOptionals.filter(Optional::isPresent).map(Optional::get)
```

在 Java 9 中，Optional 配备了 stream（）方法。 此方法是一个适配器，它将 Optional 变为包含元素的 Stream（如果在可选项中存在，或者如果它为空则为 none）。 结合 Stream 的 flatMap 方法（第 45 项），此方法为上面的代码片段提供了简洁的替代：

```
streamOfOptionals..flatMap(Optional::stream)
```

并非所有返回类型都受益于可选的治疗。 容器类型（包括集合，映射，流，数组和选项）不应包含在选项中。 您应该只返回一个空的 List（Item 54），而不是返回一个空的 Optional <List>。 返回空容器将消除客户端代码处理可选项的需要。 ProcessHandle 类确实有 arguments 方法，它返回 Optional <String []>，但是这个方法应该被视为一个不能被模拟的异常。

那么什么时候应该声明一个方法来返回 Optional 而不是 T？ 作为规则，如果可能无法返回结果，则应声明返回 Optional 的方法，如果未返回结果，则客户端必须执行特殊处理。 也就是说，返回一个 Optional 并非没有成本。 Optional 是必须分配和初始化的对象，从可选项中读取值需要额外的间接。 这使得选项不适合在某些性能关键的情况下使用。 特定方法是否属于此类别只能通过仔细测量来确定（第 67 项）。

返回包含盒装基元类型的可选项与返回基元类型相比非常昂贵，因为可选项具有两个级别的装箱而不是零。因此，库设计者认为适合为基本类型 int，long 和 double 提供 Optional 的类似物。 这些可选类型是 OptionalInt，OptionalLong 和 OptionalDouble。 它们包含大多数但不是全部的可选方法。 因此，您永远不应该返回可选的盒装基元类型，可能的例外是“次要基元类型”，布尔，字节，字符，短和浮点数。

到目前为止，我们已经讨论了返回选项并在返回后处理它们。 我们还没有讨论其他可能的用途，这是因为大多数其他选项的使用都是可疑的。 例如，您不应该使用选项作为地图值。 如果这样做，您有两种方法可以从地图中表示键的逻辑缺失：键可以不在地图中，也可以存在并映射到空的可选项。 这代表了不必要的复杂性，具有很大的混淆和错误的可能性。 更一般地说，在集合或数组中使用可选项作为键，值或元素几乎是不合适的。

这留下了一个无法回答的大问题。 是否适合在实例字段中存储可选项？ 通常它是一种“难闻的气味”：它表明你可能应该有一个包含可选字段的子类。 但有时它可能是合理的。 考虑第 2 项中我们的 NutritionFacts 类的情况.AdamitionFacts 实例包含许多不需要的字段。对于这些字段的每个可能组合，您不能拥有子类。 此外，这些字段具有原始类型，这使得直接表达缺席变得尴

尬。 NutritionFacts 的最佳 API 将为每个可选字段从 getter 返回一个可选项，因此将这些选项作为字段存储在对象中是很有意义的。

总之，如果您发现自己编写的方法无法始终返回值，并且您认为方法的用户每次调用它时都考虑这种可能性，那么您应该返回一个可选项。 但是，您应该意识到返回选项会产生真正的性能影响; 对于性能关键的方法，最好返回 null 或抛出异常。 最后，您应该很少使用任何其他容量的可选项而不是返回值。

56 为所有导出的 API 元素写文档注释

如果 API 可用，则必须记录。 传统上，API 文档是手动生成的，并且与代码保持同步是件苦差事。 Java 编程环境使用 Javadoc 实用程序简化了此任务。 Javadoc 使用特殊格式的文档注释（通常称为 doc 注释）从源代码自动生成 API 文档。

虽然文档注释约定不是语言的正式部分，但它们构成了每个 Java 程序员都应该知道的事实上的 API。“如何编写文档注释”网页[Javadoc-guide]中介绍了这些约定。 虽然自 Java 4 发布以来该页面尚未更新，但它仍然是一个非常宝贵的资源。 Java 9 中添加了一个重要的 doc 标签，{@ index}; Java 8 中的一个，{@implSpec}; Java 5 中有两个，{@literal}和{@code}。 上述网页中缺少这些标签，但在此项目中进行了讨论。

要正确记录 API，必须在每个导出的类，接口，构造函数，方法和字段声明之前加上 doc 注释。如果一个类是可序列化的，你还应该记录它的序列化表格（第 87 项）。在没有文档注释的情况下，Javadoc 可以做的最好的事情是将声明重现为受影响的 API 元素的唯一文档。使用缺少文档注释的 API 令人沮丧且容易出错。公共类不应使用默认构造函数，因为无法为它们提供文档注释。要编写可维护的代码，您还应该为大多数未导出的类，接口，构造函数，方法和字段编写文档注释，尽管这些注释不需要像导出的 API 元素那样彻底。

方法的 doc 注释应该简洁地描述方法与其客户之间的契约。除了为继承而设计的类中的方法（第 19 项）之外，合同应该说明方法的作用而不是它的工作方式。 doc 注释应该枚举所有方法的前提条件，这些条件是客户端调用它的必要条件，以及它的后置条件，这些条件是在调用成功完成后将成立的事情。通常，@throws 标记隐含地描述了前提条件，用于未经检查的异常;每个未经检查的异常对应于前提条件违规。此外，可以在@param 标记中指定前提条件以及受影响的参数。

除了先决条件和后置条件之外，方法还应记录任何副作用。副作用是系统状态的可观察到的变化，这对于实现后置条件而言显然不是必需的。例如，如果方法启动后台线程，则文档应记录它。

要完全描述方法的合约，doc 注释应该为每个参数都有一个@param 标记，@return 标记除非方法具有 void 返回类型，并且对于方法抛出的每个异常都应该有@throws 标记，无论是选中还是未选中（第 74 项）。如果@return 标记中的文本与方法的描述相同，则可以允许省略它，具体取决于您遵循的编码标准。

按照惯例，@ param 标记或@return 标记后面的文本应该是描述参数或返回值表示的值的名词短语。很少使用算术表达式代替名词短语;请参阅 BigInteger 的示例。 @throws 标记后面的文本应该包含单词“if”，后跟一个描述抛出异常的条件的子句。按照惯例，@ param，@ return 或@throws 标记之后的短语或子句不会被句点终止。以下文档评论说明了所有这些约定：

```
/**
* Returns the element at the specified position in this list.
**
<p>This method is <i>not</i> guaranteed to run in constant
* time. In some implementations it may run in time proportional
* to the element position.
**
@param index index of element to return; must be
* non-negative and less than the size of this list
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException if the index is out of range
* ({@code index < 0 || index >= this.size()})
*/
E get(int index);
```

请注意在此 doc 注释（<p>和<i>）中使用 HTML 标记。 Javadoc 实用程序将 doc 注释转换为 HTML，文档注释中的任意 HTML 元素最终都会生成 HTML

文档。有时候，程序员甚至会在他们的文档评论中嵌入 HTML 表格，尽管这种情况很少见。

还要注意在@throws 子句中围绕代码片段使用 Javadoc {@code}标记。此标记有两个用途：它使代码片段以代码字体呈现，并禁止在代码片段中处理 HTML 标记和嵌套的 Javadoc 标记。后一个属性允许我们在代码片段中使用小于号（<），即使它是 HTML 元字符。要在文档注释中包含多行代码示例，请使用包含在 HTML <pre>标记内的 Javadoc {@code}标记。换句话说，在代码示例之前加上字符<pre> {@ code 并跟随它} </ pre>。这样可以保留代码中的换行符，并且不需要转义 HTML 元字符，而不需要转义符号（@），如果代码示例使用注释，则必须对其进行转义。

最后，请注意在 doc 评论中使用“this list”。按照惯例，单词“this”指的是在实例方法的 doc 注释中使用方法时调用方法的对象。

如第 15 项所述，当你设计一个继承类时，你必须记录它的自用模式，因此程序员知道覆盖其方法的语义。应使用在 Java 8 中添加的@implSpec 标记来记录这些自用模式。回想一下，普通文档注释描述了方法与其客户之间的契约;相反，@ implSpec 注释描述了方法及其子类之间的契约，如果子类继承方法或通过 super 调用它，则允许子类依赖于实现行为。以下是它在实践中的表现：

```
/**
* Returns true if this collection is empty.
**
@implSpec
* This implementation returns {@code this.size() == 0}.
**
@return true if this collection is empty
*/
public boolean isEmpty() { ... }
```

从 Java 9 开始，除非您传递命令行开关-tag“implSpec：a：Implementation Requirements：”，否则 Javadoc 实用程序仍会忽略@implSpec 标记。 希望这将在随后的版本中得到补救。

不要忘记，您必须采取特殊操作来生成包含 HTML 元字符的文档，例如小于号（<），大于号（>）和符号（＆）。 将这些字符放入文档的最佳方法是使用{@literal}标记将它们包围起来，该标记禁止处理 HTML 标记和嵌套的 Javadoc 标记。它就像{@code}标记，但它不会以代码字体呈现文本。例如，这个 Javadoc 片段：

```
* A geometric series converges if {@literal |r| < 1}.
```

生成文档：“如果| r |，几何系列会收敛<1.“{@literal}标记可能只放在小于号的位置，而不是整个不等式，并且使用相同的结果文档，但是文档注释在源代码中的可读性较差。这说明了 doc 注释在源代码和生成的文档中都应该是可读的一般原则。如果您无法实现这两者，则生成的文档的可读性将胜过源代码的可读性。

每个文档注释的第一个“句子”（如下定义）成为注释所属元素的摘要描述。例如，第 255 页上的 doc 注释中的摘要描述是“返回此列表中指定位置的元素。”摘要描述必须单独用于描述其汇总的元素的功能。为避免混淆，类或接口中的两个成员或构造函数不应具有相同的摘要描述。要特别注意过载，为此通常使用相同的第一句话是自然的（但在文档评论中是不可接受的）。

如果预期的摘要描述包含句点，请小心，因为句点可能会提前终止描述。例如，以“大学学位”开头的文档评论，例如 B.S.，M.S。或者博士“将导致摘要描述”大学学位，例如 BS，MS“问题是摘要描述在第一个句点结束，后面跟着一个空格，制表符或行终止符（或者第一个块标签）[Javadoc-ref]。这里，缩写“M.S.”中的第二个句点后跟一个空格。最好的解决方案是使用{@literal}标记包围违规期和任何相关文本，因此源代码中的空格后面不再是空格：

```
/**
* A college degree, such as B.S., {@literal M.S.} or Ph.D.
*/
public class Degree { ... }
```

说摘要描述是文档评论中的第一句话有点误导。 公约规定它很少应该是一个完整的句子。 对于方法和构造函数，摘要描述应该是描述方法执行的操作的动词短语（包括任何对象）。 例如：

> ArrayList(int initialCapacity)—Constructs an empty list with the specified initial capacity.

Collection.size()—Returns the number of elements in this collection.

As shown in these examples, use the third person declarative tense (“returns the number”) rather than the second person imperative (“return the number”).

For classes, interfaces, and fields, the summary description should be a noun phrase describing the thing represented by an instance of the class or interface or by the field itself. For example:

Instant—An instantaneous point on the time-line.

Math.PI—The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

- ArrayList（int initialCapacity） - 使用指定的初始容量构造一个空列表。
- Collection.size（） - 返回此集合中的元素数。
- 如这些示例所示，使用第三人称声明时（“返回数字”）而不是第二人命令（“返回数字”）。
- 对于类，接口和字段，摘要描述应该是描述由类或接口的实例或字段本身表示的事物的名词短语。 例如：
- 瞬间 - 时间线上的瞬时点。
- Math.PI-比 pi 更接近 pi 的双值，即圆周长与直径的比值。

在 Java 9 中，客户端索引被添加到 Javadoc 生成的 HTML 中。 该索引简化了导航大型 API 文档集的任务，采用页面右上角的搜索框形式。 当您在框中键入内容时，您会看到一个匹配页面的下拉菜单。 API 元素（例如类，方法和字段）会自动编入索引。 有时，您可能希望索引对 API 很重要的其他术语。 为此目的添加了{@index}标记。 索引 doc 注释中出现的术语就像将其包装在此标记中一样简单，如此片段所示：

```
* This method complies with the {@index IEEE 754} standard.
```

泛型，枚举和注释需要特别注意文档注释。 记录泛型类型或方法时，请务必记录所有类型参数：

```
/**
* An object that maps keys to values. A map cannot contain
```

```
* duplicate keys; each key can map to at most one value.

**

(Remainder omitted)

**

@param <K> the type of keys maintained by this map

* @param <V> the type of mapped values

*/

public interface Map<K, V> { ... }
```

记录枚举类型时，请务必记录常量以及类型和任何公共方法。 请注意，如果简短，您可以将整个文档注释放在一行上：

```
/**

* An instrument section of a symphony orchestra.

*/

public enum OrchestraSection {

/** Woodwinds, such as flute, clarinet, and oboe. */

WOODWIND,

/** Brass instruments, such as french horn and trumpet. */

BRASS,

/** Percussion instruments, such as timpani and cymbals. */

PERCUSSION,

/** Stringed instruments, such as violin and cello. */

STRING;

}
```

记录注释类型时，请务必记录任何成员以及类型本身。 使用名词短语记录成员，就好像他们是字段一样。 对于类型的摘要描述，请使用动词短语，该动词短语说明当程序元素具有此类型的注释时它的含义：

```
/**
* Indicates that the annotated method is a test method that
* must throw the designated exception to pass.
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
/**
* The exception that the annotated test method must throw
* in order to pass. (The test is permitted to throw any
* subtype of the type described by this class object.)
*/
Class<? extends Throwable> value();
}
```

包级文档注释应放在名为 packageinfo 的文件中。java 的。 除了这些注释之外，package-info.java 还必须包含一个包声明，并且可以在此声明中包含注释。同样，如果您选择使用模块系统（第 15 项），则应将模块级注释放在 module-info.java 文件中。

在文档中经常被忽略的 API 的两个方面是线程安全性和可序列化。无论类或静态方法是否为线程安全，您都应记录其线程安全级别，如第 82 项中所述。如果类是可序列化的，则应记录其序列化形式，如第 87 项中所述。

Javadoc 具有“继承”方法注释的能力。如果 API 元素没有 doc 注释，Javadoc 将搜索最具体的适用 doc 注释，优先考虑超类上的接口。搜索算法的详细信息可以在 The Javadoc Reference Guide [Javadoc-ref]中找到。您还可以使用 {@inheritDoc}标记从超类型继承部分文档注释。这意味着，除其他外，类可以重用它们实现的接口的 doc 注释，而不是复制这些注释。该工具有可能减轻维护多组几乎相同的文档注释的负担，但使用起来很棘手并且有一些限制。详细信息超出了本书的范围。

关于文档注释，应该添加一个警告。虽然有必要为所有导出的 API 元素提供文档注释，但这并不总是足够的。对于由多个相互关联的类组成的复杂 API，通常需要使用描述 API 总体体系结构的外部文档来补充文档注释。如果存在此类文档，则相关的类或包文档注释应包含指向它的链接。

Javadoc 会自动检查是否符合此项中的许多建议。在 Java 7 中，需要命令行开关-Xdoclint 才能获得此行为。在 Java 8 和 9 中，默认情况下启用检查。诸如 checkstyle 之类的 IDE 插件会进一步检查是否符合这些建议[Burn01]。您还可以通过 HTML 有效性检查器运行 Javadoc 生成的 HTML 文件，从而降低文档注释中出错的可能性。这将检测 HTML 标记的许多不正确用法。有几个这样的检查器可供下载，您可以使用 W3C 标记验证服务[W3C-validator]在 Web 上验证 HTML。在验证生成的 HTML 时，请记住，从 Java 9 开始，Javadoc 能够生成 HTML5 以及 HTML 4.01，但默认情况下仍会生成 HTML 4.01。如果希望 Javadoc 生成 HTML5，请使用-html5 命令行开关。

本项中描述的约定涵盖了基础知识。虽然在撰写本文时已有十五岁，但撰写文档评论的权威指南仍然是如何撰写文档评论[Javadoc-guide]。如果您遵守此项目中的指南，生成的文档应提供 API 的清晰描述。但是，唯一可以确定的方法是读取 Javadoc 实用程序生成的网页。对于其他人将使用的 API，值得这样做。正如测试程序几乎不可避免地导致代码发生一些变化一样，阅读文档通常会导致对文档注释进行至少一些小的更改。

总而言之，文档注释是记录 API 的最佳，最有效的方法。对于所有导出的 API 元素，它们的使用应被视为必需的。采用符合标准惯例的一致风格。请记住，文档注释中允许使用任意 HTML，并且必须转义 HTML 元字符。

第九章 通用程序设计

本章专门讨论该语言的细节。 它讨论了局部变量，控制结构，库，数据类型和两个语言设施：反射和本机方法。 最后，它讨论了优化和命名约定。

57 最小化局部变量作用域

此项与第 15 项“最小化类和成员的可访问性”类似。通过最小化局部变量的范围，可以提高代码的可读性和可维护性，并降低出错的可能性。

较旧的编程语言（如 C）要求必须在块的头部声明局部变量，并且一些程序员继续习惯这样做。这是一个值得打破的习惯。作为一个温和的提醒，Java 允许您在声明合法的任何地方声明变量（C，自 C99 以来）。

用于最小化局部变量范围的最强大的技术是将其声明为首次使用的位置。如果变量在使用之前被声明，那就更加混乱了 - 还有一件事要分散那些试图弄清楚程序运行情况的读者的注意力。到使用变量时，读者可能不记得变量的类型或初始值。

过早地声明局部变量可能导致其范围不仅过早开始而且结束太晚。局部变量的范围从声明它的位置延伸到封闭块的末尾。如果变量在使用它的块之外声明，则在程序退出该块后它仍然可见。如果在其预定用途区域之前或之后意外使用变量，则后果可能是灾难性的。

几乎每个局部变量声明都应该包含一个初始化器。如果您还没有足够的信息来合理地初始化变量，那么您应该推迟声明直到您这样做。此规则的一个例外是 try-catch 语句。如果将变量初始化为其求值可以抛出已检查异常的表达式，则必须在 try 块内初始化该变量（除非封闭方法可以传播异常）。如果必须在 try 块之外使用该值，则必须在 try 块之前声明它，它不能“合理地初始化”。例如，请参见第 283 页。

循环提供了一个特殊的机会来最小化变量的范围。 for 循环以其传统形式和 for-each 形式允许您声明循环变量，将其范围限制在需要它们的确切区域。（该区域由循环体和 for 关键字与正文之间的括号中的代码组成。）因此，优先选择循环到 while 循环，假设循环终止后不需要循环变量的内容。

例如，这是迭代集合的首选习语（第 58 项）：

```
// Preferred idiom for iterating over a collection or array
for (Element e : c) {
    ... // Do Something with e
}
// Idiom for iterating when you need the iterator
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e and i
```

```
}
```

要了解为什么这些 for 循环优于 while 循环，请考虑以下代码片段，其中包含两个 while 循环和一个 bug：

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) { // BUG!
    doSomethingElse(i2.next());
}
```

第二个循环包含一个复制和粘贴错误：它初始化一个新的循环变量 i2，但是使用旧的变量 i，不幸的是，它仍在范围内。 生成的代码编译时没有错误，并且在不抛出异常的情况下运行，但它做错了。 第二个循环不是在 c2 上迭代，而是立即终止，给出了 c2 为空的错误印象。 由于程序无声地错误，因此错误可能会长时间未被检测到。

如果与 for 循环（for-each 或 traditional）中的任何一个一起进行类似的复制和粘贴错误，则生成的代码甚至不会编译。 第一个循环中的元素（或迭代器）变量不在第二个循环的范围内。 以下是传统 for 循环的外观：

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
Element e = i.next();
... // Do something with e and i
}
...
// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
```

```
    Element e2 = i2.next();
    ... // Do something with e2 and i2
}
```

此外，如果你使用 for 循环，那么你制作复制和粘贴错误的可能性要小得多，因为没有动机在两个循环中使用不同的变量名。 循环是完全独立的，因此重用元素（或迭代器）变量名称没有坏处。 事实上，这样做通常很时尚。 for 循环比 while 循环还有一个优点：它更短，增强了可读性。 这是另一个循环习惯用法，它最小化了局部变量的范围：

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    ... // Do something with i;
}
```

关于这个习语的重要注意事项是它有两个循环变量 i 和 n，它们都具有完全正确的范围。 第二个变量 n 用于存储第一个变量的极限，从而避免了每次迭代中冗余计算的成本。 通常，如果循环测试涉及一个方法调用，保证在每次迭代时返回相同的结果，则应该使用此习惯用法。

最小化局部变量范围的最终技术是保持方法小而集中。 如果在同一方法中组合两个活动，则与一个活动相关的局部变量可能位于执行另一个活动的代码范围内。 为了防止这种情况发生，只需将方法分为两个：每个活动一个。

58 for each 优于传统 for 循环

如第 45 项所述，某些任务最好用流完成，其他任务最好用迭代完成。 这是一个传统的 for 循环迭代集合：

```
// Not the best way to iterate over a collection!
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // Do something with e
}
```

这里是迭代数组的传统 for 循环：

```
// Not the best way to iterate over an array!
```

```
for (int i = 0; i < a.length; i++) {
    ... // Do something with a[i]
}
```

这些成语比 while 循环更好（第 57 项），但它们并不完美。 迭代器和索引变量都只是杂乱无章 - 你需要的只是元素。 此外，它们代表了错误的机会。 迭代器在每个循环中出现三次，索引变量为四，这使您有很多机会使用错误的变量。 如果这样做，则无法保证编译器能够解决问题。 最后，两个循环完全不同，不必要地注意容器的类型，并添加（轻微）麻烦来改变这种类型。

for-each 循环（官方称为“增强语句”）解决了所有这些问题。 它通过隐藏迭代器或索引变量来消除混乱和错误的机会。 由此产生的习惯同样适用于集合和数组，简化了将容器的实现类型从一个切换到另一个的过程：

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    ... // Do something with e
}
```

当您看到冒号（:)时，将其读作“in”。因此，上面的循环读作“对于元素中的每个元素 e”。使用 for-each 循环没有性能损失，即使对于数组：代码 它们生成的内容基本上与您手动编写的代码相同。

与嵌套迭代相比，for-each 循环优于传统 for 循环的优势更大。 这是人们在进行嵌套迭代时常犯的错误：

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
EIGHT,NINE, TEN, JACK, QUEEN, KING }
...
static Collection<Suit> suits = Arrays.asList(Suit.values());
static Collection<Rank> ranks = Arrays.asList(Rank.values());
List<Card> deck = new ArrayList<>();
```

```
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )

for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )

deck.add(new Card(i.next(), j.next()));
```

如果你没有发现这个 bug，不要心疼。 许多专家程序员在某个时候犯过这个错误。 问题是在外部集合（诉讼）的迭代器上调用下一个方法的次数太多。 它应该从外部循环调用，以便每个套装调用一次，而是从内部循环调用它，因此每个卡调用一次。 在用完套装后，循环抛出 NoSuchElementException。

如果你真的不走运，外部集合的大小是内部集合大小的倍数 - 也许是因为它们是相同的集合 - 循环将正常终止，但它不会做你想要的。 例如，考虑这种错误的尝试打印一对骰子的所有可能的卷：

```
// Same bug, different symptom!

enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }

...

Collection<Face> faces = EnumSet.allOf(Face.class);

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )

for (Iterator<Face> j = faces.iterator(); j.hasNext(); )

System.out.println(i.next() + " " + j.next());
```

该程序不会抛出异常，但它只打印六个“双打”（从“ONE ONE”到“SIX SIX”），而不是预期的三十六个组合。

要修复这些示例中的错误，必须在外部循环的范围中添加一个变量来保存外部元素：

```
// Fixed, but ugly - you can do better!

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {

    Suit suit = i.next();

    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )

        deck.add(new Card(suit, j.next()));

}
```

相反，如果你使用嵌套的 for-each 循环，问题就会消失。 生成的代码简洁如您所愿：

```
// Preferred idiom for nested iteration on collections and arrays

for (Suit suit : suits)

for (Rank rank : ranks)

deck.add(new Card(suit, rank));
```

- 不幸的是，有三种情况你不能使用 foreach：
- 破坏性过滤 - 如果需要遍历删除所选元素的集合，则需要使用显式迭代器，以便可以调用其 remove 方法。您通常可以使用在 Java 8 中添加的 Collection 的 removeIf 方法来避免显式遍历。
- 转换 - 如果需要遍历列表或数组并替换其元素的部分或全部值，则需要列表迭代器或数组索引才能替换元素的值。
- 并行迭代 - 如果您需要并行遍历多个集合，那么您需要对迭代器或索引变量进行显式控制，以便所有迭代器或索引变量都可以在锁步中前进（如上面的有缺陷的卡和骰子示例中无意中所示）。如果您发现自己处于上述任何一种情况，请使用普通的 for 循环并警惕此项中提到的陷阱。
- for-each 循环不仅可以迭代集合和数组，还可以迭代实现 Iterable 接口的任何对象，该接口由单个方法组成。以下是界面的外观：

```
public interface Iterable<E> {

// Returns an iterator over the elements in this iterable

Iterator<E> iterator();

}
```

如果你必须从头开始编写自己的 Iterator 实现，那么实现 Iterable 有点棘手，但是如果你正在编写一个代表一组元素的类型，你应该强烈考虑让它实现 Iterable，即使你选择不拥有它也是如此。 它实现了 Collection。 这将允许您的用户使用 foreach 循环迭代您的类型，他们将永远感激。

总之，for-each 循环在清晰度，灵活性和错误预防方面提供了超越传统 for 循环的引人注目的优势，而且没有性能损失。 尽可能使用 for-each 循环优先于 for 循环。

59 了解和使用类库

假设您要生成零和某个上限之间的随机整数。 面对这个常见的任务，许多程序员会编写一个看起来像这样的小方法：

```
// Common but deeply flawed!
static Random rnd = new Random();
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这种方法可能看起来不错，但它有三个缺陷。 首先，如果 n 是 2 的小幂，则随机数的序列将在相当短的时间段后重复。 第二个缺陷是，如果 n 不是 2 的幂，平均而言，某些数字将比其他数字更频繁地返回。 如果 n 很大，这种效果可能非常明显。 以下程序有力地证明了这一点，该程序在精心选择的范围内生成了一百万个随机数，然后打印出在该范围的下半部分中有多少个数字：

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
    low++;
    System.out.println(low);
}
```

如果随机方法正常工作，程序将打印接近 50 万的数字，但如果你运行它，你会发现它打印的数字接近 666,666。随机方法生成的数字的三分之二落在其范围的下半部分！

随机方法的第三个缺陷是，在极少数情况下，它可能会灾难性地失败，返回指定范围之外的数字。这是因为该方法尝试通过调用 Math.abs 将 rnd.nextInt（）返回的值映射到非负 int。如果 nextInt（）返回 Integer.MIN_VALUE，则

Math.abs 也将返回 Integer.MIN_VALUE，余数运算符（％）将返回负数，假设 n 不是 2 的幂。这几乎肯定会导致程序失败，并且可能难以重现失败。

要编写一个纠正这些缺陷的随机方法的版本，你必须知道关于伪随机数生成器，数论和二进制补码算法的相当数量。幸运的是，你不必这样做 - 它已经为你完成了。它叫做 Random.nextInt（int）。您不必关心它如何完成其工作的细节（尽管您可以研究文档或源代码，如果您很好奇）。一位具有算法背景的高级工程师花费了大量时间来设计，实现和测试这种方法，然后向该领域的几位专家展示，以确保它是正确的。然后，这个库经过了 beta 测试，发布，并被数百万程序员广泛使用了近二十年。该方法尚未发现任何缺陷，但如果要发现缺陷，将在下一个版本中修复。通过使用标准库，您可以利用编写它的专家的知识以及在您之前使用它的人的经验。

从 Java 7 开始，您不应再使用 Random。对于大多数用途，选择的随机数生成器现在是 ThreadLocalRandom。它产生更高质量的随机数，而且速度非常快。在我的机器上，它比 Random 快 3.6 倍。对于 fork 连接池和并行流，请使用 SplittableRandom。

使用这些库的第二个好处是，您不必浪费时间编写临时解决方案来解决与您的工作仅有轻微关系的问题。如果你像大多数程序员一样，你宁愿花时间在应用程序上而不是在底层管道上工作。

使用标准库的第三个优点是，它们的性能会随着时间的推移而不断提高，而您无需付出任何努力。因为许多人使用它们并且因为它们被用于行业标准基准测试，所以提供这些库的组织有强烈的动力使它们运行得更快。多年来，许多 Java 平台库都经过重写，有时会反复重复，从而显着提升性能。使用库的第四个优点是它们倾向于随着时间的推移获得功能。如果某个库遗失了某些东西，开发人员社区就会知道它，并且可能会在后续版本中添加缺少的功能。

使用标准库的最后一个优点是您可以将代码置于主流中。这样的代码更容易被大量开发人员读取，维护和重用。

鉴于所有这些优点，使用库设施优先于临时实现似乎是合乎逻辑的，但许多程序员却没有。为什么不？也许他们不知道图书馆设施存在。每个主要版本的库中都添加了许多功能，并且可以随时了解这些新增内容。每次有 Java 平台的主要版本时，都会发布一个描述其新功能的网页。这些页面非常值得一读[Java8-feat，Java9-feat]。为了强调这一点，假设您想编写一个程序来打印命令行中指定的 URL 的内容（这大致与 Linux curl 命令相同）。在 Java 9 之前，这段代码有点乏味，但在 Java 9 中，transferTo 方法被添加到 InputStream 中。以下是使用此新方法执行此任务的完整程序：

```
// Printing the contents of a URL with transferTo, added in Java 9
public static void main(String[] args) throws IOException {
    try (InputStream in = new URL(args[0]).openStream()) {
        in.transferTo(System.out);
    }
}
```

这些库太大了，无法学习所有文档[Java9-api]，但每个程序员都应该熟悉 java.lang，java.util 和 java.io 及其子包的基础知识。可以根据需要获取其他图书馆的知识。总结图书馆的设施超出了本项目的范围，这些设施多年来一直在增长。

几个图书馆特别值得一提。集合框架和流库（项目 45-48）应该是每个程序员的基本工具包的一部分，java.util.concurrent 中的并发实用程序的一部分也应如此。该软件包包含用于简化多线程编程任务的高级实用程序和低级原语，以允许专家编写自己的高级并发抽象。 java.util.concurrent 的高级部分在第 80 和 81 项中讨论。

有时，图书馆设施可能无法满足您的需求。您的需求越专业化，就越有可能发生这种情况。虽然您的第一个冲动应该是使用库，但如果您已经查看了它们在某些区域提供的内容并且它不能满足您的需求，那么请使用备用实现。任何有限的库集提供的功能总是存在漏洞。如果您无法在 Java 平台库中找到所需内容，那么您的下一个选择应该是查看高质量的第三方库，例如 Google 优秀的开源 Guava 库[Guava]。如果您在任何适当的库中找不到所需的功能，您可能别无选择，只能自己实现。

总而言之，不要重新发明轮子。如果您需要做一些似乎应该相当普遍的事情，那么库中可能已经有了一个可以满足您需求的工具。如果有，请使用它;如果您不知道，请检查。一般来说，库代码可能比您自己编写的代码更好，并且可能会随着时间的推移而改进。这并不反映你作为程序员的能力。规模经济决定了图书馆代码得到的关注远远超过大多数开发人员可以承担的相同功能。

60 如果需要精确答案，避免使用 float 和 double

float 和 double 类型主要用于科学和工程计算。 它们执行二进制浮点运算，经过精心设计，可在很宽的范围内快速提供准确的近似值。 但是，它们不能提

供准确的结果，不应在需要确切结果的地方使用。 float 和 double 类型特别不适合进行货币计算，因为不可能将 0.1（或任何其他 10 的负幂）表示为 float 或 double。

例如，假设你的口袋里有 1.03 美元，你花费 42 美分。 你还剩多少钱？ 这是一个试图回答这个问题的天真程序片段：

```
System.out.println(1.03 - 0.42);
```

不幸的是，它打印出 0.6100000000000001。 这不是一个孤立的案例。 假设你口袋里有一美元，你买九个洗衣机，每个洗衣机价格为 10 美分。 你得到多少变化？

```
System.out.println(1.00 - 9 * 0.10);
```

根据这个程序片段，你得到$ 0.09999999999999998。

您可能认为问题只能通过在打印前舍入结果来解决，但不幸的是，这并不总是有效。 例如，假设你的口袋里有一块钱，你看到一个架子上有一排美味的糖果，价格分别为 10 美分，20 美分，30 美分等等，最高可达 1 美元。 你买一个糖果，从一个 10 美分的糖果开始，直到你买不起货架上的下一个糖果。 你买了多少个糖果，你有多少变化？ 这是一个旨在解决这个问题的天真程序：

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = 0.10; funds >= price; price += 0.10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought +"items bought.");
    System.out.println("Change: $" + funds);
}
```

如果你运行该程序，你会发现你可以买三块糖果，剩下 $ 0.3999999999999999。 这是错误的答案！ 解决此问题的正确方法是使用 BigDecimal，int 或 long 进行货币计算。

这是对前一个程序的直接转换，使用 BigDecimal 类型代替 double。请注意，使用 BigDecimal 的 String 构造函数而不是其双构造函数。 这是必要的，以避免在计算中引入不准确的值[Bloch05，Puzzle 2]：

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal(".10");
    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;funds.compareTo(price) >= 0;price
= price.add(TEN_CENTS)) {
        funds = funds.subtract(price);
    itemsBought++;
    }
    System.out.println(itemsBought +"items bought.");
    System.out.println("Money left over: $" + funds);
}
```

如果你运行修改后的程序，你会发现你可以买到四块糖果，剩下 0.00 美元。这是正确的答案。

但是，使用 BigDecimal 有两个缺点：它比使用原始算术类型方便得多，而且速度要慢得多。 如果你解决一个短暂的问题，后一个缺点是无关紧要的，但前者可能会惹恼你。

使用 BigDecimal 的另一种方法是使用 int 或 long，具体取决于所涉及的数量，并自己跟踪小数点。 在这个例子中，显而易见的方法是以美分而不是美元进行所有计算。 这是采用这种方法的直接转换：

```
public static void main(String[] args) {
    int itemsBought = 0;
```

```
        int funds = 100;
        for (int price = 10; funds >= price; price += 10) {
            funds -= price;
            itemsBought++;
        }
        System.out.println(itemsBought +"items bought.");
        System.out.println("Cash left over: " + funds + " cents");
    }
```

总之，不要对任何需要精确答案的计算使用 float 或 double。 如果您希望系统跟踪小数点，请使用 BigDecimal，并且不介意不使用基本类型的不便和成本。使用 BigDecimal 具有额外的优势，它可以让您完全控制舍入，只要执行需要舍入的操作，就可以从八种舍入模式中进行选择。 如果您使用法律规定的舍入行为执行业务计算，这会派上用场。 如果性能至关重要，您不介意自己跟踪小数点，并且数量不是太大，请使用 int 或 long。 如果数量不超过九位十进制数，则可以使用 int; 如果他们不超过十八位数，你可以使用长。 如果数量可能超过十八位，请使用 BigDecimal。

61 基本类型优于装箱类型

Java 有一个由两部分组成的类型系统，由基元组成，如 int，double 和 boolean，以及引用类型，如 String 和 List。每个基本类型都有一个相应的引用类型，称为盒装基元。对应于 int，double 和 boolean 的盒装基元是 Integer，Double 和 Boolean。

如第 6 项所述，自动装箱和自动拆箱模糊，但不删除基元和盒装基元类型之间的区别。两者之间存在真正的差异，重要的是您要清楚自己正在使用哪种以及在它们之间仔细选择。

基元和盒装基元之间存在三个主要差异。首先，基元只有它们的值，而盒装基元具有与它们的值不同的标识。换句话说，两个盒装原始实例可以具有相同的值和不同的身份。其次，原始类型只有完全功能的值，而每个盒装基元类型除了相应基元类型的所有功能值外，还有一个非功能值，即 null。最后，基元比盒装基元更具时间和空间效率。如果你不小心，所有这三个差异都会让你陷入困境。

请考虑以下比较器，该比较器旨在表示整数值的升序数字顺序。 （回想一下，比较器的 compare 方法返回一个负数，零或正数，具体取决于它的第一个参数是小于，等于还是大于它的第二个参数。）你不需要在实践中编写这个比较器因为它实现了 Integer 的自然排序，但它提供了一个有趣的例子：

```
// Broken comparator - can you spot the flaw?

Comparator<Integer> naturalOrder =(i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

这个比较器看起来应该工作，它将通过许多测试。例如，它可以与 Collections.sort 一起使用，以正确排序百万元素列表，无论列表是否包含重复元素。但比较国存在严重缺陷。为了说服自己，只需打印 naturalOrder.compare（new Integer（42），new Integer（42））的值。两个 Integer 实例都表示相同的值（42），因此该表达式的值应为 0，但它为 1，表示第一个 Integer 值大于第二个值！

所以有什么问题？ naturalOrder 中的第一个测试工作正常。评估表达式 i <j 会导致 i 和 j 引用的 Integer 实例自动取消装箱;也就是说，它提取原始值。进行评估以检查结果 int 值中的第一个是否小于第二个。但是假设它不是。然后，下一个测试将计算表达式 i == j，它对两个对象引用执行标识比较。如果 i 和 j 引用表示相同 int 值的不同 Integer 实例，则此比较将返回 false，并且比较器将错误地返回 1，表示第一个 Integer 值大于第二个。将==运算符应用于盒装基元几乎总是错误的。

实际上，如果你需要一个比较器来描述一个类型的自然顺序，你应该简单地调用 Comparator.naturalOrder（），如果你自己编写一个比较器，你应该使用比较器构造方法，或原始类型的静态比较方法（项目 14）。也就是说，您可以通过添加两个局部变量来存储与盒装 Integer 参数对应的原始 int 值，并对这些变量执行所有比较，从而解决损坏的比较器中的问题。这避免了错误的身份比较：

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

接下来，考虑这个令人愉快的小程序：

```
public class Unbelievable {
static Integer i;
```

```
    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
        }
    }
```

不，它不打印难以置信 - 但它的作用几乎一样奇怪。 在计算表达式 i == 42 时，它会抛出 NullPointerException。 问题是我是一个 Integer，而不是一个 int，和所有非常量对象引用字段一样，它的初始值为 null。当程序计算表达式 i == 42 时，它将 Integer 与 int 进行比较。 几乎在每种情况下，当您在操作中混合基元和盒装基元时，盒装基元将自动取消装箱。 如果空对象引用是自动取消装箱，则会出现 NullPointerException。 正如该计划所示，它几乎可以在任何地方发生。解决问题就像声明我是一个 int 而不是一个 Integer 一样简单。

最后，请考虑第 6 项中第 24 页的程序：

```
    // Hideously slow program! Can you spot the object creation?
    public static void main(String[] args) {
        Long sum = 0L;
        for (long i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
```

这个程序比它应该慢得多，因为它意外地声明一个局部变量（sum）是盒装基元类型 Long 而不是基本类型 long。程序编译时没有错误或警告，并且变量被重复加框和取消装箱，导致观察到的性能下降。

在本项目讨论的所有三个程序中，问题都是相同的：程序员忽略了原语和盒装原语之间的区别，并承受了后果。在前两个计划中，后果是完全失败;在第三，严重的性能问题。

那么什么时候应该使用盒装原语？它们有几种合法用途。第一个是集合中的元素，键和值。您不能将基元放在集合中，因此您不得不使用盒装基元。这

是一个更普遍的特例。您必须使用盒装基元作为参数化类型和方法（第 5 章）中的类型参数，因为该语言不允许您使用基元。例如，您不能将变量声明为 ThreadLocal 类型，因此您必须使用 ThreadLocal。最后，在进行反射方法调用时必须使用盒装基元（第 65 项）。

总之，只要有选择，就可以优先使用基元而不是盒装基元。原始类型更简单，更快捷。如果你必须使用盒装基元，小心！自动装箱减少了使用盒装基元的冗长，但没有降低危险。当你的程序将两个盒装基元与==运算符进行比较时，它会进行身份比较，这几乎肯定不是你想要的。当您的程序执行涉及盒装和未装箱原语的混合类型计算时，它会进行拆箱，当您的程序进行拆箱时，它会抛出 NullPointerException。最后，当您的程序框原始值时，它可能导致代价高昂且不必要的对象创建。

62 如果其他类型更合适，避免使用 String

字符串旨在表示文本，并且它们可以很好地完成它。由于字符串非常常见并且语言得到很好的支持，因此将字符串用于除设计字符串之外的其他目的的自然倾向。这个项目讨论了一些你不应该用字符串做的事情。

字符串是其他值类型的不良替代品。当一段数据从文件，网络或键盘输入进入程序时，它通常是字符串形式。有一种自然倾向，就是这样，但只有当数据本质上是文本性的时候，这种趋势才是合理的。如果它是数字，则应将其转换为适当的数字类型，例如 int，float 或 BigInteger。如果它是一个是或否的问题的答案，它应该被翻译成适当的枚举类型或布尔值。更一般地说，如果存在适当的值类型，无论是原始值还是对象引用，都应该使用它;如果没有，你应该写一个。虽然这个建议似乎很明显，但它经常被违反。

字符串是枚举类型的不良替代品。正如第 34 项中所讨论的，枚举使得枚举类型常量比字符串好得多。

字符串是聚合类型的不良替代品。如果实体具有多个组件，则将其表示为单个字符串通常是个坏主意。例如，这里是来自真实系统的一行代码 - 标识符名称已被更改以保护有罪：

```
// Inappropriate use of string as aggregate type

String compoundKey = className + "#" + i.next();
```

这种方法有许多缺点。如果用于分隔字段的字符出现在其中一个字段中，则可能会产生混乱。要访问单个字段，您必须解析字符串，这很慢，很乏味且容易出错。您不能提供 equals，toString 或 compareTo 方法，但必须接受 String

提供的行为。更好的方法是编写一个类来表示聚合，通常是私有静态成员类（第 24 项）。

字符串是功能的不良替代品。有时，字符串用于授予对某些功能的访问权限。例如，考虑线程局部变量工具的设计。这样的工具提供了每个线程都有自己值的变量。从版本 1.2 开始，Java 库就有了一个线程局部变量工具，但在此之前，程序员必须自己动手。当多年前遇到设计这样一个设施的任务时，有几个人独立地提出了相同的设计，其中客户提供的字符串密钥用于识别每个线程局部变量：

```
// Broken - inappropriate use of string as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    // Sets the current thread's value for the named variable.
    public static void set(String key, Object value);

    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

这种方法的问题是字符串键表示线程局部变量的共享全局命名空间。 为了使方法起作用，客户端提供的字符串键必须是唯一的：如果两个客户端独立决定对其线程局部变量使用相同的名称，则它们无意中共享一个变量，这通常会导致两个客户端 失败。 而且，安全性很差。 恶意客户端可能故意使用与另一个客户端相同的字符串密钥来获取对其他客户端数据的非法访问权限。

可以通过使用不可伪造的密钥（有时称为功能）替换字符串来修复此 API：

```
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable
    public static class Key { // (Capability)
        Key() { }
```

```
    }

    // Generates a unique, unforgeable key

    public static Key getKey() {

        return new Key();

    }

    public static void set(Key key, Object value);

    public static Object get(Key key);

    }
```

虽然这解决了基于字符串的 API 的两个问题，但您可以做得更好。 你不再需要静态方法了。 它们可以成为键上的实例方法，此时键不再是线程局部变量的键：它是线程局部变量。 此时，toplevel 类不再为你做任何事情，所以你不妨去除它并将嵌套类重命名为 ThreadLocal：

```
    public final class ThreadLocal {

    public ThreadLocal();

    public void set(Object value);

    public Object get();

    }
```

此 API 不是类型安全的，因为当您从线程局部变量中检索它时，必须将值从 Object 转换为其实际类型。 不可能使原始的基于 String 的 API 类型安全且难以使基于 Keybased 的 API 类型安全，但通过使 ThreadLocal 成为参数化类（第 29 项）来使这种 API 类型安全是一件简单的事情：

```
    public final class ThreadLocal<T> {

    public ThreadLocal();

    public void set(T value);

    public T get();

    }
```

粗略地说，这是 java.lang.ThreadLocal 提供的 API。 除了解决基于字符串的 API 的问题之外，它还比任何基于密钥的 API 更快，更优雅。

总而言之，当存在或可以编写更好的数据类型时，避免将对象表示为字符串的自然倾向。 使用不当，字符串比其他类型更麻烦，更灵活，更慢，更容易出错。 字符串通常被滥用的类型包括基本类型，枚举和聚合类型。

63 小心 String 连接性能

字符串连接运算符（+）是将几个字符串合并为一个的便捷方式。 它可以生成单行输出或构造一个小的固定大小对象的字符串表示，但它不能缩放。 重复使用字符串连接运算符来连接 n 个字符串需要 n 中的时间二次方。 这是字符串不可变这一事实的不幸后果（第 17 项）。 当连接两个字符串时，将复制两者的内容。

例如，考虑这种方法，它通过重复连接每个项目的一行来构造一个记帐语句的字符串表示：

```
// Inappropriate use of string concatenation - Performs poorly!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i); // String concatenation
    return result;
}
```

如果项目数量很大，则该方法执行得非常糟糕。 要获得可接受的性能，请使用 StringBuilder 代替 String 来存储正在构造的语句：

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
```

```
}
```

自 Java 6 以来，许多工作已经开始使字符串连接更快，但两种方法的性能差异仍然很大：如果 numItems 返回 100 而 lineForItem 返回 80 个字符的字符串，则第二个方法的运行速度比 我机器上的第一个。 因为第一种方法是项目数量的二次方，而第二种方法是线性的，所以随着项目数量的增长，性能差异会变得更大。 请注意，第二种方法预分配一个足够大的 StringBuilder 来保存整个结果，从而无需自动增长。 即使使用默认大小的 StringBuilder 失谐，它仍然比第一种方法快 5.5 倍。

道德很简单：除非性能无关紧要，否则不要使用字符串连接运算符来组合多个字符串。 请改用 StringBuilder 的 append 方法。 或者，使用字符数组，或一次处理一个字符串而不是组合它们。

64 通过接口引用对象

第 51 项说你应该使用接口而不是类作为参数类型。 更一般地说，您应该倾向于使用类上的接口来引用对象。 如果存在适当的接口类型，则应使用接口类型声明参数，返回值，变量和字段。 您真正需要引用对象类的唯一时间是使用构造函数创建它。 为了使这个具体，请考虑 LinkedHashSet 的情况，它是 Set 接口的一个实现。 养成打字的习惯：

```
// Good - uses interface as type

Set<Son> sonSet = new LinkedHashSet<>();
```

不是这样:

```
// Bad - uses class as type!

LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

如果您养成使用接口作为类型的习惯，您的程序将更加灵活。 如果您决定要切换实现，那么您所要做的就是更改构造函数中的类名（或使用不同的静态工厂）。 例如，第一个声明可以更改为：

```
Set<Son> sonSet = new HashSet<>();
```

并且所有周围的代码将继续工作。周围的代码不知道旧的实现类型，因此它将无视这一变化。

有一点需要注意：如果原始实现提供了接口的常规合约不需要的某些特殊功能，并且代码依赖于该功能，那么新实现提供相同的功能至关重要。例如，

如果围绕第一个声明的代码依赖于 LinkedHashSet 的排序策略，那么在声明中用 HashSet 替换 LinkedHashSet 是不正确的，因为 HashSet 不保证迭代顺序。

那你为什么要改变一种实现类型呢？因为第二个实现提供了比原始实现更好的性能，或者因为它提供了原始实现所缺乏的理想功能。例如，假设一个字段包含一个 HashMap 实例。将其更改为 EnumMap 将提供更好的性能和迭代顺序与键的自然顺序一致，但如果键类型是枚举类型，则只能使用 EnumMap。将 HashMap 更改为 LinkedHashMap 将提供可预测的迭代顺序，其性能可与 HashMap 相媲美，而不会对密钥类型提出任何特殊要求。

您可能认为使用其实现类型声明变量是可以的，因为您可以同时更改声明类型和实现类型，但无法保证此更改将导致编译的程序。如果客户端代码使用原始实现类型上的替换时不存在的方法，或者客户端代码将实例传递给需要原始实现类型的方法，则在进行此更改后代码将不再编译。使用接口类型声明变量可以保证您的诚实。

如果不存在适当的接口，则通过类而不是接口引用对象是完全合适的。例如，考虑值类，例如 String 和 BigInteger。值类很少用多个实现来编写。它们通常是最终的，很少有相应的接口。将这样的值类用作参数，变量，字段或返回类型是完全合适的。

没有适当接口类型的第二种情况是属于框架的对象，其基本类型是类而不是接口。如果一个对象属于这样一个基于类的框架，最好用相关的基类来引用它，它通常是抽象的，而不是它的实现类。许多 java.io 类（如 OutputStream）都属于此类。

没有适当接口类型的最后一种情况是实现接口的类，但也提供了在接口中找不到的额外方法 - 例如，PriorityQueue 具有 Queue 接口上不存在的比较器方法。只有当程序依赖于额外的方法时，才应该使用这样的类来引用它的实例，这应该是非常罕见的。

这三种情况并不是详尽无遗的，而仅仅是为了传达适合通过其类别引用对象的情况的味道。在实践中，应该明确给定对象是否具有适当的接口。如果是这样，如果您使用界面来引用对象，您的程序将更加灵活和时尚。如果没有适当的接口，只需使用提供所需功能的类层次结构中最不具体的类。

65 接口优于反射

核心反射工具 java.lang.reflect 提供对任意类的编程访问。给定一个 Class 对象，您可以获得 Constructor，Method 和 Field 实例，这些实例表示由 Class 实例

表示的类的构造函数，方法和字段。这些对象提供对类的成员名称，字段类型，方法签名等的编程访问。

此外，Constructor，Method 和 Field 实例允许您反思性地操作它们的底层对应物：您可以通过在 Constructor，Method 和 Field 实例上调用方法来构造实例，调用方法和访问底层类的字段。例如，Method.invoke 允许您在任何类的任何对象上调用任何方法（受通常的安全性约束）。反射允许一个类使用另一个类，即使在编译前者时后一个类不存在。然而，这种力量是有代价的：

您将失去编译时类型检查的所有好处，包括异常检查。如果程序试图反射性地调用不存在或不可访问的方法，则除非您采取了特殊的预防措施，否则它将在运行时失败。

执行反射访问所需的代码是笨拙和冗长的。写作和阅读困难是单调乏味的。

性能受损。反射方法调用比普通方法调用慢得多。究竟要慢多少，因为有很多因素在起作用。在我的机器上，当反射完成时，调用没有输入参数和 int 返回的方法会慢 11 倍。

有一些复杂的应用程序需要反射。示例包括代码分析工具和依赖注入框架。即便是这样的工具也已经远离了最近的反思，因为它的缺点变得更加清晰。如果您对应用程序是否需要反射有任何疑问，则可能不会。

通过仅以非常有限的形式使用反射，您可以获得许多反射的好处，同时产生很少的成本。对于许多必须使用在编译时不可用的类的程序，在编译时存在一个适当的接口或超类来引用该类（Item 64）。如果是这种情况，您可以反射创建实例并通过其接口或超类正常访问它们。

例如，这是一个程序，它创建一个 Set 实例，其类由第一个命令行参数指定。程序将剩余的命令行参数插入到集合中并打印它。无论第一个参数如何，程序都会打印剩余的参数，并删除重复项。但是，打印这些参数的顺序取决于第一个参数中指定的类。如果指定 java.util.HashSet，则它们以明显随机的顺序打印; 如果指定 java.util.TreeSet，则它们按字母顺序打印，因为 TreeSet 中的元素是按顺序排序的：

```
// Reflective instantiation with interface access

public static void main(String[] args) {

    // Translate the class name into a Class object

    Class<? extends Set<String>> cl = null;
```

```
try {
        cl = (Class<? extends Set<String>>) // Unchecked cast!
        Class.forName(args[0]);
} catch (ClassNotFoundException e) {
        fatalError("Class not found.");
}

// Get the constructor
Constructor<? extends Set<String>> cons = null;
try {
        cons = cl.getDeclaredConstructor();
} catch (NoSuchMethodException e) {
        fatalError("No parameterless constructor");
}
// Instantiate the set
Set<String> s = null;
try {
        s = cons.newInstance();
} catch (IllegalAccessException e) {
        fatalError("Constructor not accessible");
} catch (InstantiationException e) {
        fatalError("Class not instantiable.");
} catch (InvocationTargetException e) {
        fatalError("Constructor threw " + e.getCause());
} catch (ClassCastException e) {
```

```
            fatalError("Class doesn't implement Set");
        }
        // Exercise the set
        s.addAll(Arrays.asList(args).subList(1, args.length));
        System.out.println(s);
    }

    private static void fatalError(String msg) {
        System.err.println(msg);
        System.exit(1);
    }
```

虽然这个程序只是一个玩具，但它演示的技术非常强大。玩具程序可以很容易地变成通用集测试程序，通过积极地操纵一个或多个实例并检查它们是否遵守 Set 契约来验证指定的 Set 实现。同样，它可以变成通用的集合性能分析工具。事实上，这种技术足以实现一个成熟的服务提供者框架（第 1 项）。通常，这种技术就是你在反思中所需要的。

这个例子说明了反射的两个缺点。首先，该示例可以在运行时生成六个不同的异常，如果不使用反射实例化，则所有这些异常都是编译时错误。 （为了好玩，您可以通过传入适当的命令行参数使程序生成六个异常中的每一个。）第二个缺点是需要二十五行繁琐的代码才能从其名称生成类的实例，而构造函数调用可以整齐地放在一行上。可以通过捕获 ReflectiveOperationException 来减少程序的长度，ReflectiveOperationException 是 Java 7 中引入的各种反射异常的超类。这两个缺点仅限于实例化对象的程序部分。实例化后，该集与任何其他 Set 实例无法区分。在真实的程序中，大量的代码因此不受这种有限的反射使用的影响。

如果您编译此程序，您将获得未经检查的强制转换警告。这个警告是合法的，因为演员阵容<？即使命名类不是 Set 实现，扩展 Set>也会成功，在这种情况下，程序在实例化类时抛出 ClassCastException。要了解有关抑制警告的信息，请阅读第 27 项。

合法（如果罕见）使用反射是管理类对运行时可能不存在的其他类，方法或字段的依赖性。如果您正在编写必须针对某些其他软件包的多个版本运行的

软件包，这将非常有用。该技术是针对支持它所需的最小环境（通常是最旧的版本）编译您的包，并反复访问任何更新的类或方法。要使其工作，如果在运行时不存在您尝试访问的较新类或方法，则必须采取适当的操作。适当的行动可能包括使用一些替代手段来实现相同的目标或以减少的功能运行。

总之，反射是某些复杂系统编程任务所需的强大工具，但它有许多缺点。如果您正在编写一个必须在编译时使用未知类的程序，那么您应该尽可能使用反射来实例化对象，并使用编译时已知的某个接口或超类来访问这些对象。

66 谨慎使用本地方法

Java Native Interface（JNI）允许 Java 程序调用本机方法，这些方法是用本机编程语言（如 C 或 C ++）编写的方法。从历史上看，本机方法有三个主要用途。它们提供对特定于平台的设施（如注册表）的访问。它们提供对现有本机代码库的访问，包括提供对旧数据的访问的旧库。最后，本机方法用于以本机语言编写应用程序的性能关键部分，以提高性能。

使用本机方法访问特定于平台的工具是合法的，但很少需要：随着 Java 平台的成熟，它提供了对以前仅在主机平台中发现的许多功能的访问。例如，Java 9 中添加的流程 API 提供对 OS 流程的访问。当 Java 中没有可用的等效库时，使用本机方法来使用本机库也是合法的。

很难建议使用本机方法来提高性能。在早期版本（Java 3 之前）中，它通常是必需的，但从那时起 JVM 就变得更快了。对于大多数任务，现在可以在 Java 中获得可比较的性能。例如，当在版本 1.1 中添加 java.math 时，BigInteger 依赖于用 C 编写的一个快速的多精度算术库。在 Java 3 中，BigInteger 在 Java 中重新实现，并仔细调整到比原始运行速度快的程度。本机实现。

这个故事的一个令人遗憾的结论是 BigInteger 从那以后变化不大，除了 Java 8 中大数字的快速乘法。在那个时候，本地库继续快速工作，特别是 GNU 多精度算术库（GMP）。现在需要真正高性能多精度算术的 Java 程序员通过本机方法[Blum14]使用 GMP 是合理的。

使用本机方法具有严重的缺点。由于本机语言不安全（第 50 项），使用本机方法的应用程序不再免受内存损坏错误的影响。由于本机语言比 Java 更依赖于平台，因此使用本机方法的程序不太可移植。它们也更难调试。如果您不小心，原生方法可能会降低性能，因为垃圾收集器无法自动化甚至跟踪本机内存使用情况（第 8 项），并且存在与进出本机代码相关的成本。最后，本机方法需要“粘合代码”，难以阅读和编写繁琐。

总之，在使用本机方法之前要三思而后行。您很少需要使用它们来提高性能。如果必须使用本机方法来访问低级资源或本机库，请尽可能使用本机代码并对其进行彻底测试。本机代码中的单个错误可能会破坏整个应用程序。

67 谨慎优化

关于优化有三个格言，每个人都应该知道：

更多的计算罪是以效率的名义（不一定实现它）而不是任何其他单一原因 - 包括盲目的愚蠢。

-William A. Wulf [Wulf72]

我们应该忘记小的效率，大约 97％的时间说：过早的优化是所有邪恶的根源。

-Donald E. Knuth [Knuth74]

我们在优化问题上遵循两条规则：规则 1.不要这样做。 规则 2（仅限专家）。不要这样做 - 也就是说，直到你有一个完全清晰和未经优化的解决方案。

-M。 A.杰克逊[杰克逊 75]

所有这些格言都早于 Java 编程语言二十年。 他们讲述了优化的深层真理：弊大于利，特别是如果你过早优化的话。 在此过程中，您可能会生成既不快又不正确且无法轻松修复的软件。

不要为了表现而牺牲合理的建筑原则。努力编写好的程序而不是快速的程序。如果一个好的程序不够快，它的架构将允许它进行优化。好的程序体现了信息隐藏的原则：在可能的情况下，它们将设计决策本地化为单个组件，因此可以在不影响系统其余部分的情况下更改个别决策（第 15 项）。

这并不意味着您可以在程序完成之前忽略性能问题。实现问题可以通过以后的优化来解决，但是如果不重写系统，就无法修复限制性能的普遍存在的架构缺陷。事后改变设计的基本方面可能导致结构不良的系统难以维护和发展。因此，您必须在设计过程中考虑性能。

努力避免限制性能的设计决策。事后最难改变的设计组件是指定组件之间和外部世界之间的交互。这些设计组件中最主要的是 API，线级协议和持久数据

格式。事实上，这些设计组件不仅难以或不可能改变，而且所有这些都可能对系统可以实现的性能产生重大限制。

考虑 API 设计决策的性能影响。使公共类型可变可能需要大量不必要的防御性复制（第 50 项）。类似地，在公共类中使用继承，其中组合适当地将类永远地绑定到其超类，这可以对子类的性能施加人为限制（第 18 项）。作为最后一个示例，使用实现类型而不是 API 中的接口将您与特定实现联系起来，即使将来可能会编写更快的实现（第 64 项）。

API 设计对性能的影响是非常真实的。考虑 java.awt.Component 类中的 getSize 方法。这个性能关键方法返回 Dimension 实例的决定，加上 Dimension 实例可变的决定，强制此方法的任何实现都在每次调用时分配一个新的 Dimension 实例。尽管在现代 VM 上分配小对象的成本很低，但是不必要地分配数百万个对象会对性能造成实际损害。

存在几种 API 设计替代方案。理想情况下，Dimension 应该是不可变的（第 17 项）;或者，getSize 可能已被两个返回 Dimension 对象的各个基本组件的方法所取代。实际上，出于性能原因，在 Java 2 中将两个这样的方法添加到 Component 中。但是，预先存在的客户端代码仍然使用 getSize 方法，并且仍然会受到原始 API 设计决策的性能影响。

幸运的是，通常情况下，良好的 API 设计与良好的性能一致。扭曲 API 以获得良好性能是一个非常糟糕的主意。导致您变形 API 的性能问题可能会在未来版本的平台或其他底层软件中消失，但扭曲的 API 及其附带的支持头痛将永远伴随着您。

一旦您仔细设计了程序并生成了清晰，简洁且结构良好的实现，那么可能是时候考虑优化，假设您对程序的性能不满意。

回想一下杰克逊的两个优化规则是“不要这样做”和“（仅限专家）。不要这样做。“他本可以再添加一个：在每次尝试优化之前和之后测量性能。您可能会对所发现的内容感到惊讶。通常，尝试优化对性能没有可测量的影响;有时，他们会使情况变得更糟。主要原因是很难猜出你的程序在哪里花费时间。您认为程序部分很慢可能没有错，在这种情况下，您将浪费时间尝试优化它。普遍的看法是，程序将 90％的时间花在 10％的代码上。

分析工具可以帮助您确定优化工作的重点。这些工具为您提供运行时信息，例如每个方法消耗的大致时间以及调用的次数。除了重点调整工作之外，这还可以提醒您需要进行算法更改。如果一个二次（或更差）算法潜伏在你的程序内，那么没有多少调整可以解决问题。您必须将算法替换为更有效的算法。系

统中的代码越多，使用分析器就越重要。就像在大海捞针一样：大海捞针越大，金属探测器就越有用。另一个值得特别提及的工具是 jmh，它不是一个分析器，而是一个微基准测试框架，它提供了对 Java 代码详细性能的无与伦比的可视性[JMH]。

测量优化尝试效果的需求在 Java 中比在 C 和 C ++等更传统的语言中更大，因为 Java 具有较弱的性能模型：各种原始操作的相对成本定义不太明确。程序员编写的内容与 CPU 执行的内容之间的“抽象差距”更大，这使得更可靠地预测优化的性能结果变得更加困难。有很多表演神话浮出水面，结果证明是半真半假或彻头彻尾的谎言。

Java 的性能模型不仅定义不明确，而且从实现到实现，从发布到发布，从处理器到处理器都有所不同。如果您将在多个实现或多个硬件平台上运行程序，那么衡量优化对每个实现的影响非常重要。有时，您可能不得不在不同实现或硬件平台上的性能之间进行权衡。

自该项目首次编写以来近二十年，Java 软件堆栈的每个组件都变得越来越复杂，从处理器到虚拟机再到库，以及 Java 运行的各种硬件都在不断增长。所有这些结合在一起，使得 Java 程序的性能现在比 2001 年更难以预测，并且相应地增加了测量它的需求。

总而言之，不要努力写出快速的程序 - 努力写出好的程序; 速度将随之而来。 但是在设计系统时要考虑性能，尤其是在设计 API，线级协议和持久数据格式时。 完成系统构建后，请测量其性能。 如果它足够快，你就完成了。 如果没有，请借助分析器找到问题的根源，然后开始优化系统的相关部分。 第一步是检查您的算法选择：没有多少低级优化可以弥补差的算法选择。 根据需要重复此过程，在每次更改后测量性能，直到您满意为止。

68 遵守普遍的命名规范

Java 平台有一套完善的命名约定，其中许多都包含在 Java 语言规范[JLS，6.1]中。简而言之，命名约定分为两类：印刷和语法。

只有少数印刷命名约定，包括包，类，接口，方法，字段和类型变量。你应该很少违反它们，也绝不会没有充分的理由。如果 API 违反这些约定，则可能难以使用。如果实施违反了它们，则可能难以维护。在这两种情况下，违规都有可能混淆和激怒使用代码的其他程序员，并可能导致导致错误的错误假设。这些公约在本项目中进行了总结。

包和模块名称应该是分层的，组件以句点分隔。组件应包含小写字母字符，很少包含数字。将在您的组织外部使用的任何程序包的名称应以您组织的 Internet 域名开头，其组件相反，例如，edu.cmu，com.google，org.eff。名称以 java 和 javax 开头的标准库和可选包是此规则的例外。用户不得创建名称以 java 或 javax 开头的包或模块。可以在 JLS [JLS，6.1]中找到将 Internet 域名转换为包名称前缀的详细规则。

包名称的其余部分应由描述包的一个或多个组件组成。组件应该很短，通常是八个或更少的字符。鼓励使用有意义的缩写，例如，util 而不是实用程序。缩略语是可以接受的，例如，awt。组件通常应由单个单词或缩写组成。

除了 Internet 域名之外，许多包的名称只包含一个组件。其他组件适用于大型设施，其大小要求将其分解为非正式层次结构。例如，javax.util 包具有丰富的包层次结构，其名称如 java.util.concurrent.atomic。这样的包被称为子包，尽管对包层次结构几乎没有语言支持。

类和接口名称（包括枚举和注释类型名称）应由一个或多个单词组成，每个单词的首字母大写，例如 List 或 FutureTask。除了首字母缩略词和某些常用缩写（如 max 和 min）之外，应避免使用缩写。关于首字母缩略词是大写还是仅首字母大写，存在一些分歧。虽然一些程序员仍然使用大写字母，但是可以做出强有力的论证，只支持大写第一个字母：即使多个首字母缩略词背靠背出现，你仍然可以知道一个单词的起始位置和下一个单词的结尾。您更喜欢看哪个类名，HTTPURL 或 HttpUrl？

方法和字段名称遵循与类和接口名称相同的排版约定，但方法或字段名称的第一个字母应为小写，例如 remove 或 ensureCapacity。如果首字母缩略词作为方法或字段名称的第一个单词出现，则它应该是小写的。

以前规则的唯一例外是“常量字段”，其名称应由一个或多个由下划线字符分隔的大写单词组成，例如 VALUES 或 NEGATIVE_INFINITY。常量字段是静态最终字段，其值是不可变的。如果静态 final 字段具有基本类型或不可变引用类型（第 17 项），则它是常量字段。例如，枚举常量是常量字段。如果静态 final 字段具有可变引用类型，则如果引用的对象是不可变的，则它仍然可以是常量字段。请注意，常量字段构成了下划线的唯一推荐用法。

局部变量名称与成员名称具有相似的排版命名约定，但允许使用缩写除外，单个字符和短字符序列的含义取决于它们出现的上下文，例如 i，denom，houseNum。输入参数是一种特殊的局部变量。它们的名称应该比普通的局部变量更加仔细，因为它们的名称是其方法文档中不可或缺的一部分。

类型参数名称通常由单个字母组成。最常见的是它是以下五种中的一种：T 代表任意类型，E 代表集合的元素类型，K 代表 V 和 V 代表地图的值类型，X 代表异常。函数的返回类型通常是 R.任意类型的序列可以是 T，U，V 或 T1，T2，T3。

为便于快速参考，下表显示了印刷约定的示例。

Identifier Type	Example
Package or module	org.junit.jupiter.api, com.google.common.collect
Class or Interface	Stream, FutureTask, LinkedHashMap,HttpClient
Method or Field	remove, groupingBy, getCrc
Constant Field	MIN_VALUE, NEGATIVE_INFINITY
Local Variable	i, denom, houseNum
Type Parameter	T, E, K, V, X, R, U, V, T1, T2

语法命名约定比印刷约定更灵活，更具争议性。对于包而言，没有语法命名约定。可实例化的类（包括枚举类型）通常以单数名词或名词短语命名，例如 Thread，PriorityQueue 或 ChessPiece。不可实例化的实用程序类（第 4 项）通常以复数名词命名，例如收集器或集合。接口被命名为类，例如，Collection 或 Comparator，或者以能够或者为结尾的形容词，例如，Runnable，Iterable 或 Accessible。因为注释类型具有如此多的用途，所以没有词性占主导地位。名词，动词，介词和形容词都很常见，例如 BindingAnnotation，Inject，ImplementedBy 或 Singleton。

执行某些操作的方法通常使用动词或动词短语（包括对象）命名，例如 append 或 drawImage。返回布尔值的方法通常具有以单词 is 或不常见的单词开头的名称，后跟名词，名词短语或任何用作形容词的单词或短语，例如 isDigit，isProbablePrime，isEmpty， isEnabled，或 hasSiblings。

返回非布尔函数或调用它们的对象的属性的方法通常以名词，名词短语或以动词 get 开头的动词短语命名，例如 size，hashCode 或 getTime。有一个声乐

队伍声称只有第三种形式（以 get 开头）是可以接受的，但这种说法几乎没有基础。前两种形式通常会产生更易读的代码，例如：

```
if (car.speed() > 2 * SPEED_LIMIT)
generateAudibleAlert("Watch out for cops!");
```

以 get 开头的表单源于大部分过时的 Java Beans 规范，该规范构成了早期可重用组件体系结构的基础。有一些现代工具继续依赖于 Beans 命名约定，您可以随意在任何与这些工具结合使用的代码中使用它。如果一个类包含同一属性的 setter 和 getter，那么遵循这个命名约定也有一个很好的先例。在这种情况下，这两个方法通常命名为 getAttribute 和 setAttribute。

一些方法名称值得特别提及。转换对象类型，返回不同类型的独立对象的实例方法通常称为 toType，例如 toString 或 toArray。返回类型与接收对象类型不同的视图（第 6 项）的方法通常称为 asType，例如 asList。返回与调用它们的对象具有相同值的原语的方法通常称为 typeValue，例如 intValue。静态工厂的通用名称包括 from，of，valueOf，instance，getInstance，newInstance，getType 和 newType（第 1 项，第 9 页）。

字段名称的语法约定不太完善，并且不如类，接口和方法名称那么重要，因为设计良好的 API 包含很少（如果有）暴露字段。 boolean 类型的字段通常被命名为 boolean accessor 方法，省略了 initial，例如，initialized，composite。其他类型的字段通常以名词或名词短语命名，例如 height，digits 或 bodyStyle。局部变量的语法约定类似于字段，但甚至更弱。

总而言之，内化标准命名约定并学习使用它们作为第二天性。印刷惯例很简单，很明确;语法惯例更复杂，更宽松。引用 Java 语言规范[JLS，6.1]，“如果长期存在的常规用法另有规定，则不应盲目遵循这些约定。”使用常识。

第十章 异常

在最常见的情况下，异常可以提高程序的可读性，可靠性和可维护性。 如果使用不当，可能会产生相反的效果。 本章提供有效使用异常的指南。

69 只针对异常情况才使用异常

Someday, if you are unlucky, you may stumble across a piece of code that looks something like this:

总有一天，如果你运气不好，你可能偶然发现一段看起来像这样的代码：

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        range[i++].climb();
    }
    catch (ArrayIndexOutOfBoundsException e) {
}
```

这段代码有什么作用？ 从检查来看，这一点并不明显，这就是不使用它的原因（第 67 项）。 事实证明，这是一种用于循环遍历数组元素的非常错误的习惯用语。 无限循环在尝试访问数组边界外的第一个数组元素时抛出，捕获并忽略 ArrayIndexOutOfBoundsException，从而终止。 它应该等同于循环数组的标准习惯用法，任何 Java 程序员都可以立即识别：

```
for (Mountain m : range)
    m.climb();
```

那么为什么有人会使用基于异常的循环而不是尝试和真实？根据错误推理提高性能是一种错误的尝试，因为 VM 检查所有数组访问的边界，由编译器隐藏但仍然存在于 for-each 循环中的正常循环终止测试是多余的，应该是避免。这个推理有三个问题：

因为异常是针对特殊情况而设计的，所以 JVM 实现者很少有动力使它们像显式测试一样快。

将代码放在 try-catch 块中会禁止 JVM 实现可能执行的某些优化。

循环数组的标准习惯用法不一定会导致冗余检查。许多 JVM 实现会优化它们。

事实上，基于异常的习语远比标准习惯慢。在我的机器上，基于异常的习惯用法的速度大约是 100 个元素数组的标准速度的两倍。

基于异常的循环不仅会混淆代码的目的并降低其性能，而且不能保证它能够正常工作。如果循环中存在错误，则使用流控制异常可以掩盖错误，从而使调试过程变得非常复杂。假设循环体中的计算调用一个方法，该方法对一些不

相关的数组执行越界访问。如果使用了合理的循环习惯用法，则该错误将生成未捕获的异常，从而导致使用完整堆栈跟踪立即终止线程。如果使用了误导的基于异常的循环，则会捕获与错误相关的异常并将其误解为正常的循环终止。

这个故事的寓意很简单：顾名思义，例外仅用于特殊情况;它们永远不应该用于普通的控制流程。更一般地说，使用标准的，易于识别的习语，而不是过于聪明的技术，旨在提供更好的性能。即使性能优势是真实的，它也可能不会继续面对稳定改进的平台实施。然而，过于聪明的技术带来的微妙错误和维护难题肯定会存在。

这个原则也对 API 设计有影响。精心设计的 API 不得强制其客户端使用普通控制流的异常。 具有“状态依赖”方法的类只能在某些不可预测的条件下调用，通常应该有一个单独的“状态测试”方法，指示调用依赖于状态的方法是否合适。例如，Iterator 接口接下来是状态依赖方法，相应的状态测试方法 hasNext。这使得标准习惯用于使用传统的 for 循环遍历集合（以及 for-each 循环，其中 hasNext 方法在内部使用）：

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
    ...
}
```

If Iterator lacked the hasNext method, clients would be forced to do this instead:

如果 Iterator 缺少 hasNext 方法，则客户端将被迫执行此操作：

```
// Do not use this hideous code for iteration over a collection!
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next();
    ...
    }
}
catch (NoSuchElementException e) {
```

```
}
```

在开始此项的数组迭代示例之后，这看起来应该非常熟悉。除了冗长和误导之外，基于异常的循环可能表现不佳并且可以掩盖系统中不相关部分中的错误。

提供单独的状态测试方法的替代方法是使 statedependent 方法返回空的可选项（项 55）或者如果它不能执行所需的计算则返回诸如 null 的区别值。

以下是一些指导原则，可帮助您在状态测试方法和可选或可区分的返回值之间进行选择。如果要在没有外部同步的情况下同时访问对象或者受外部引发的状态转换，则必须使用可选或可区分的返回值，因为对象的状态可能会在调用 statetesting 方法与其状态之间的间隔内发生变化。依赖方法。如果单独的状态测试方法将复制与状态相关的方法的工作，则性能问题可能要求使用可选的或区分的返回值。在所有其他条件相同的情况下，状态测试方法稍微优于不同的返回值。它提供了稍好的可读性，并且可能更容易检测到错误使用：如果您忘记调用状态测试方法，则 statedependent 方法将抛出异常，使错误变得明显；如果您忘记检查区分返回值，则错误可能很微妙。这不是可选返回值的问题。

总之，例外是针对特殊情况而设计的。不要将它们用于普通的控制流程，也不要编写强制其他人这样做的 API。

70 对可恢复的情况使用受检异常，对编程错误使用运行时异常

Java 提供了三种 throwable：checked exception，runtime exceptions 和 errors。程序员之间存在一些混淆，即何时适合使用各种 throwable。虽然决定并不总是明确，但有一些一般规则可以提供强有力的指导。

决定是否使用已检查或未经检查的异常的基本规则是：对可以合理预期调用者恢复的条件使用已检查的异常。通过抛出已检查的异常，可以强制调用者在 catch 子句中处理异常或向外传播它。因此，声明抛出方法的每个已检查异常都是 API 用户的强有力指示，即关联条件是调用该方法的可能结果。

通过将用户与检查的异常对话，API 设计者提出了从条件中恢复的权限。用户可以通过捕获异常并忽略它来忽略任务，但这通常是个坏主意（第 77 项）。

有两种未经检查的 throwable：运行时异常和错误。它们的行为是相同的：两者都是不需要的，通常不应该被捕获的抛弃物。如果程序抛出未经检查的异

常或错误，通常情况下无法进行恢复，并且继续执行会带来弊大于利。如果程序没有捕获这样的 throwable，它将导致当前线程停止并显示相应的错误消息。

使用运行时异常来指示编程错误。绝大多数运行时异常表示违反前提条件。违反前提条件的原因仅仅是客户端 API 无法遵守 API 规范建立的合同。例如，数组访问的契约指定数组索引必须介于 0 和数组长度减去 1 之间（包括 1 和 1）。ArrayIndexOutOfBoundsException 指示违反了此前提条件。

这个建议的一个问题是，您是否正在处理可恢复的条件或编程错误并不总是很清楚。例如，考虑资源耗尽的情况，这可能是由编程错误引起的，例如分配不合理的大型阵列，或者是由于资源的真正短缺。如果资源耗尽是由于暂时短缺或暂时需求增加造成的，那么这种情况很可能是可以恢复的。 API 设计人员判断资源耗尽的给定实例是否可能允许恢复是一个问题。如果您认为某个条件可能允许恢复，请使用已检查的异常;如果不是，请使用运行时异常。如果不清楚是否可以进行恢复，则可能最好使用未经检查的异常，原因如第 71 项中所述。

虽然 Java 语言规范不要求它，但有一个强烈的约定，即保留错误以供 JVM 使用以指示资源缺陷，不变故障或其他使其无法继续执行的条件。鉴于几乎普遍接受这种约定，最好不要实现任何新的 Error 子类。因此，您实现的所有未经检查的 throwable 应该是 RuntimeException 的子类（直接或间接）。您不仅不应该定义 Error 子类，而且除了 AssertionError 之外，您也不应该抛出它们。

可以定义一个不是 Exception，RuntimeException 或 Error 的子类的 throwable。JLS 没有直接处理这样的 throwable，而是隐式指定它们表现为普通的检查异常（它们是 Exception 的子类，但不是 RuntimeException）。所以你什么时候应该使用这样的野兽？总之，永远不会。与普通的已检查异常相比，它们没有任何好处，只会使您的 API 用户感到困惑。

API 设计者经常忘记异常是完全成熟的对象，可以在其上定义任意方法。此类方法的主要用途是提供捕获异常的代码，其中包含有关导致抛出异常的条件的其他信息。在没有这样的方法的情况下，已知程序员解析异常的字符串表示以发现附加信息。这是非常糟糕的做法（第 12 项）。可抛出的类很少指定其字符串表示的细节，因此字符串表示可以从实现到实现和发布到发布不同。因此，解析异常的字符串表示的代码可能是不可移植且易碎的。

由于经过检查的例外通常表明可恢复的条件，因此对他们来说提供提供信息以帮助呼叫者从异常情况中恢复的方法尤为重要。例如，假设由于资金不足而尝试使用礼品卡进行购买时，会抛出已检查的异常。该异常应该提供一种访问器方法来查询不足量。这将使呼叫者能够将金额转发给购物者。有关此主题的更多信息，请参阅第 75 项。

总而言之，抛出可恢复条件的已检查异常和编程错误的未经检查的异常。如有疑问，请抛出未经检查的异常。不要定义既不是检查异常也不是运行时异常的任何 throwable。提供已检查异常的方法以帮助恢复。

71 避免不必要使用受检异常

许多 Java 程序员不喜欢检查异常，但如果使用得当，他们可以改进 API 和程序。与返回代码和未经检查的异常不同，它们迫使程序员处理问题，增强可靠性。也就是说，在 API 中过度使用已检查的异常会使它们使用起来不那么令人愉快。如果方法抛出已检查的异常，则调用它的代码必须在一个或多个 catch 块中处理它们，或者声明它抛出它们并让它们向外传播。无论哪种方式，它都会给 API 的用户带来负担。 Java 8 中的负担增加了，因为抛出已检查异常的方法不能直接在流中使用（项目 45-48）。

如果通过正确使用 API 无法阻止异常情况，并且使用 API 的程序员在遇到异常时可以采取一些有用的操作，则这种负担可能是合理的。除非满足这两个条件，否则未经检查的异常是合适的。作为试金石，请问自己程序员将如何处理异常。这是最好的吗？

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Can't happen!
}
```

Or this? 或这个？

```
} catch (TheCheckedException e) {
    e.printStackTrace(); // Oh well, we lose.
    System.exit(1);
}
```

如果程序员不能做得更好，则需要调用未经检查的异常。

如果它是由方法抛出的唯一检查异常，则由检查异常引起的程序员的额外负担要大得多。如果还有其他方法，则该方法必须已出现在 try 块中，并且此异常最多需要另一个 catch 块。如果方法抛出单个已检查的异常，则此异常是该方

法必须出现在 try 块中且不能直接在流中使用的唯一原因。在这种情况下，问问自己是否有办法避免检查异常是值得的。

消除已检查异常的最简单方法是返回所需结果类型的可选项（第 55 项）。该方法只返回一个空的可选项，而不是抛出一个已检查的异常。该技术的缺点在于该方法不能返回任何附加信息，详细说明其无法执行所需的计算。相反，例外具有描述性类型，并且可以导出方法以提供附加信息（项目 70）。

您还可以通过将抛出异常的方法分解为两个方法来将已检查的异常转换为未经检查的异常，第一个方法返回一个布尔值，指示是否抛出异常。此 API 重构会从以下内容转换调用序列：

```
// Invocation with checked exception
try {
    obj.action(args);
}
catch (TheCheckedException e) {
    ... // Handle exceptional condition
}
```

添加这里:

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
}
else {
    ... // Handle exceptional condition
}
```

这种重构并不总是合适的，但是它可以使 API 更加舒适。 虽然后一个调用序列并不比前者更漂亮，但重构的 API 更灵活。 如果程序员知道调用将成功，或者满足于让线程在失败时终止，那么重构也允许这个简单的调用序列：

```
    obj.action(args);
```

如果您怀疑普通的调用序列是常态，那么 API 重构可能是合适的。生成的 API 本质上是第 69 项中的状态测试方法 API，并且同样的警告适用：如果要在没有外部同步的情况下同时访问对象，或者它受到外部诱导的状态转换，则此重构是不合适的，因为对象的状态可能是在对 actionPermitted 和 action 的调用之间进行更改。如果单独的 actionPermitted 方法会复制 action 方法的工作，则可能会因性能原因而排除重构。

总之，在谨慎使用时，检查异常可以提高程序的可靠性;当过度使用时，它们会使 API 难以使用。如果调用者无法从失败中恢复，则抛出未经检查的异常。如果可能进行恢复并且您希望强制调用者处理异常情况，请首先考虑返回可选项。只有在失败的情况下提供的信息不足时才应该抛出一个检查过的异常。

72 优先使用标准异常

将专家程序员与经验不足的程序员区分开来的一个属性是专家努力并且通常实现高度的代码重用。代码重用是一件好事的规则也不例外。 Java 库提供了一组异常，涵盖了大多数 API 的大多数异常抛出需求。

重用标准异常有几个好处。其中最主要的是它使您的 API 更容易学习和使用，因为它符合程序员已经熟悉的既定惯例。紧接其后的是，使用您的 API 的程序更容易阅读，因为它们不会被不熟悉的异常所混淆。最后（和最少），更少的异常类意味着更小的内存占用和更少的加载类所花费的时间。

最常用的异常类型是 IllegalArgumentException（Item 49）。当调用者传入一个值不合适的参数时，这通常是抛出的异常。例如，如果调用者在表示某个操作要重复的次数的参数中传递了一个负数，则抛出此异常。

另一个常用的异常是 IllegalStateException。如果由于接收对象的状态而调用是非法的，则通常会抛出异常。例如，如果调用者在正确初始化之前尝试使用某个对象，则抛出此异常。

可以说，每个错误的方法调用都归结为非法的参数或状态，但是其他例外标准地用于某些非法的参数和状态。如果调用者在某些禁止空值的参数中传递 null，则约定表示抛出 NullPointerException 而不是 IllegalArgumentException。类似地，如果调用者将表示索引的参数中的超出范围的值传递给序列，则应抛出 IndexOutOfBoundsException 而不是 IllegalArgumentException。

另一个可重用的异常是 ConcurrentModificationException。如果设计为由单个线程（或外部同步）使用的对象检测到它正在被同时修改，则应该抛出它。此异常充其量只是一个提示，因为无法可靠地检测并发修改。

注意的最后一个标准例外是 UnsupportedOperationException。如果对象不支持尝试的操作，则抛出异常。它的使用很少见，因为大多数对象都支持它们的所有方法。此异常由未能实现由其实现的接口定义的一个或多个可选操作的类使用。例如，如果有人试图从列表中删除元素，则仅附加列表实现会抛出此异常。

不要直接重用 Exception，RuntimeException，Throwable 或 Error。将这些类看作是抽象的。您无法可靠地测试这些异常，因为它们是方法可能抛出的其他异常的超类。

此表总结了最常用的异常情况：

Exception	**Occasion for Use**
IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected
UnsupportedOperationException	Object does not support method

虽然这些是迄今为止最常用的例外情况，但其他情况可能会在情况允许的情况下重复使用。例如，如果要实现算术对象（如复数或有理数），则重用 ArithmeticException 和 NumberFormatException 是合适的。如果异常符合您的需求，请继续使用它，但前提是您抛出它的条件与异常文档一致：重用必须基于文档化的语义，而不仅仅是名称。此外，如果要添加更多细节（第 75 项），请随意为标准异常创建子类，但请记住，异常是可序列化的（第 12 章）。仅此就是没有充分理由不编写自己的异常类的理由。

选择要重用的异常可能很棘手，因为上表中的“使用场合”似乎并不相互排斥。考虑一个代表一副牌的对象的情况，并假设有一种方法可以从牌组中处理一手牌作为参数的手牌。如果调用者传递的值大于牌组中剩余的牌数，则可能被解释为IllegalArgumentException（handSize参数值太高）或IllegalStateException（牌组包含的牌数太少）。在这些情况下，如果没有参数值可行，则规则是抛出 IllegalStateException，否则抛出 IllegalArgumentException。

73 抛出与抽象对应的异常

当一个方法抛出一个与它执行的任务没有明显联系的异常时，这是令人不安的。 当一个方法传播由低级抽象抛出的异常时，通常会发生这种情况。 它不仅令人不安，而且还通过实现细节污染了更高层的 API。 如果更高层的实现在以后的版本中发生更改，则它抛出的异常也会发生更改，从而可能会破坏现有的客户端程序。

为了避免这个问题，更高层应该捕获较低级别的异常，并且在它们的位置抛出可以用更高级别的抽象来解释的异常。 这个成语被称为异常翻译：

```
// Exception Translation
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

以下是从 AbstractSequentialList 类获取的异常转换示例，该类是 List 接口的骨干实现（第 20 项）。 在此示例中，异常转换由 List <E>接口中的 get 方法规范强制要求：

```
/**
* Returns the element at the specified position in this list.
* @throws IndexOutOfBoundsException if the index is out of range
* ({@code index < 0 || index >= size()}).
*/
```

```
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    }
    catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

如果较低级别的异常可能对调试导致更高级别异常的问题的某人有帮助，则需要一种称为异常链接的特殊形式的异常链接。 较低级别的异常（原因）被传递给更高级别的异常，它提供了一个访问器方法（Throwable 的 getCause 方法）来检索较低级别的异常：

```
// Exception Chaining
try {
    ... // Use lower-level abstraction to do our bidding
}
catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

高级异常的构造函数将原因传递给链接感知超类构造函数，因此它最终传递给 Throwable 的一个链接感知构造函数，例如 Throwable（Throwable）：

```
// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
```

```
        }

    }
```

大多数标准异常都具有链接感知构造函数。对于没有的异常，可以使用 Throwable 的 initCause 方法设置原因。异常链接不仅允许您以编程方式访问原因（使用 getCause），而且还将原因的堆栈跟踪集成到更高级别异常的跟踪中。

虽然异常转换优于低层异常的无意识传播，但不应过度使用。在可能的情况下，处理较低层异常的最佳方法是通过确保较低级别的方法成功来避免它们。有时您可以通过检查更高级别方法的参数的有效性，然后再将它们传递到较低层来完成此操作。

如果不可能防止来自较低层的异常，那么下一个最好的事情就是让较高层静默地解决这些异常，从而使较高级别方法的调用者与较低级别的问题隔离开来。在这些情况下，使用某些适当的日志记录工具（如 java.util.logging）记录异常可能是适当的。这允许程序员调查问题，同时隔离客户端代码和用户。

总之，如果阻止或处理较低层的异常是不可行的，请使用异常转换，除非较低级别的方法恰好保证其所有异常都适用于较高级别。 链接提供了两全其美：它允许您抛出适当的更高级别异常，同时捕获失败分析的根本原因（第 75 项）。

74 每个方法抛出异常要有文档

方法抛出的异常的描述是正确使用该方法所需的文档的重要部分。因此，花时间仔细记录每种方法抛出的所有异常（第 56 项）至关重要。

始终单独声明已检查的异常，并准确记录使用 Javadoc @throws 标记抛出每个异常的条件。不要采用声明方法抛出它可以抛出的多个异常类的超类的快捷方式。作为一个极端的例子，不要声明公共方法抛出异常，或者更糟糕的是，抛出 Throwable。除了拒绝对方法的用户关于它能够抛出的异常的任何指导之外，这样的声明极大地阻碍了该方法的使用，因为它有效地模糊了可能在相同上下文中抛出的任何其他异常。这个建议的一个例外是 main 方法，它可以安全地声明为抛出 Exception，因为它只由 VM 调用。

虽然该语言不要求程序员声明方法能够抛出的未经检查的异常，但明智地将它们作为已检查的异常进行仔细记录是明智的。未经检查的异常通常表示编程错误（第 70 项），并且让程序员熟悉他们可以做出的所有错误有助于他们避

免出现这些错误。一个详细记录的方法可以有效抛出的未经检查的异常列表描述了其成功执行的前提条件。每个公共方法的文档都必须描述其前提条件（第56项），并记录其未经检查的异常是满足此要求的最佳方法。

接口中的方法记录它们可能抛出的未经检查的异常尤为重要。该文档构成了接口的一般合同的一部分，并实现了接口的多个实现之间的共同行为。

使用 Javadoc @throws 标记来记录方法可以抛出的每个异常，但不要在未经检查的异常上使用 throws 关键字。使用您的 API 的程序员必须知道检查了哪些异常以及哪些异常未经检查，因为程序员的责任在这两种情况下有所不同。由 Javadoc @throws 标记生成的文档在方法声明中没有相应的 throws 子句，为程序员提供了一个强大的视觉提示，即未选中异常。

应该注意的是，记录每个方法可以抛出的所有未经检查的异常是理想的，在现实世界中并不总是可以实现的。当类经历修订时，如果修改导出的方法以抛出其他未经检查的异常，则不会违反源或二进制兼容性。假设一个类从另一个独立编写的类调用一个方法。前一类的作者可能会仔细记录每个方法抛出的所有未经检查的异常，但如果后一个类被修改为抛出额外的未经检查的异常，那么前一类（未经过修订）很可能会传播新的未经检查的例外，即使它没有记录它们。

如果由于同样的原因，类中的许多方法抛出异常，则可以在类的文档注释中记录异常，而不是为每个方法单独记录它。一个常见的例子是 NullPointerException。对于类的文档注释，可以说“如果在任何参数中传递空对象引用，则此类中的所有方法都会抛出 NullPointerException”，或者说是该效果的单词。

总之，记录您编写的每个方法可能抛出的每个异常。对于未经检查和已检查的异常，以及抽象和具体方法，都是如此。此文档应采用 doc 注释中的@throws 标记的形式。在方法的 throws 子句中单独声明每个已检查的异常，但不声明未经检查的异常。如果您未能记录方法可能引发的异常，则其他人很难或不可能有效地使用您的类和接口。

75 在细节信息中包含捕获失败的信息

当程序因未捕获的异常而失败时，系统会自动打印出异常的堆栈跟踪。堆栈跟踪包含异常的字符串表示形式，即调用其 toString 方法的结果。这通常包含异常的类名，后跟其详细消息。通常，这是程序员或站点可靠性工程师在调查软件故障时将获得的唯一信息。如果故障不易再现，则可能很难或不可能获得

更多信息。因此，异常的 toString 方法返回尽可能多的关于失败原因的信息是至关重要的。换句话说，异常的详细消息应该捕获后续分析的失败。

要捕获失败，异常的详细消息应包含导致异常的所有参数和字段的值。例如，IndexOutOfBoundsException 的详细消息应包含下限，上限和未能在边界之间的索引值。这些信息告诉了很多关于失败的信息。三个值中的任何一个或全部都可能是错误的。索引可能比下限小一个或等于上限（“fencepost error”），或者它可能是一个狂野值，太低或太高。下限可能大于上限（严重的内部不变失败）。这些情况中的每一种都指向一个不同的问题，如果您知道您正在寻找什么样的错误，它将极大地帮助您进行诊断。

一个警告涉及安全敏感信息。由于许多人在诊断和修复软件问题的过程中可能会看到堆栈跟踪，因此请不要在详细消息中包含密码，加密密钥等。

虽然将所有相关数据包含在例外的详细信息中至关重要，但通常包含大量散文并不重要。堆栈跟踪旨在与文档以及必要时的源代码一起进行分析。它通常包含引发异常的确切文件和行号，以及堆栈上所有其他方法调用的文件和行号。冗长的散文描述失败是多余的;通过阅读文档和源代码可以收集信息。

不应将异常的详细消息与用户级错误消息混淆，后者必须能够为最终用户理解。与用户级错误消息不同，详细消息主要是为了程序员或站点可靠性工程师在分析故障时的利益。因此，信息内容远比可读性重要。用户级错误消息通常是本地化的，而异常详细消息很少。确保异常在其详细消息中包含足够的故障捕获信息的一种方法是在其构造函数中而不是字符串详细消息中要求此信息。然后可以自动生成详细消息以包括该信息。例如，IndexOutOfBoundsException 可能有一个如下所示的构造函数，而不是 String 构造函数：

```
/**
* Constructs an IndexOutOfBoundsException.
**
@param lowerBound the lowest legal index value
* @param upperBound the highest legal index value plus one
* @param index the actual index value
*/
```

```
    public IndexOutOfBoundsException(int lowerBound, int upperBound,int index)
{

        // Generate a detail message that captures the failure

        super(String.format("Lower bound: %d, Upper bound: %d,
Index: %d",lowerBound, upperBound, index));

        // Save failure information for programmatic access

        this.lowerBound = lowerBound;

        this.upperBound = upperBound;

        this.index = index;

    }
```

从 Java 9 开始，IndexOutOfBoundsException 最终获得了一个带有 int 值索引参数的构造函数，但遗憾的是它省略了 lowerBound 和 upperBound 参数。更一般地说，Java 库并没有大量使用这个习惯用法，但强烈建议使用它。它使程序员很容易抛出异常来捕获失败。事实上，它使程序员难以捕获失败！实际上，成语集中了代码以在异常类中生成高质量的详细消息，而不是要求类的每个用户冗余地生成详细消息。

如第 70 项所示，异常可能适合为其失败捕获信息（上例中的 lowerBound，upperBound 和 index）提供访问器方法。在已检查的异常上提供此类访问器方法比未选中更为重要，因为故障捕获信息可用于从故障中恢复。程序员可能希望以编程方式访问未经检查的异常的细节，这种情况很少见（尽管不是不可思议）。但是，即使对于未经检查的例外情况，最好根据一般原则提供这些访问者（第 12 项，第 57 页）。

76 努力使失败保持原子性

在对象抛出异常之后，通常希望对象仍处于明确定义的可用状态，即使故障发生在执行操作中。对于已检查的异常尤其如此，预期调用者将从中恢复。一般来说，失败的方法调用应该使对象处于调用之前的状态。具有此属性的方法被称为失败原子。

有几种方法可以达到这种效果。最简单的是设计不可变对象（第 17 项）。如果对象是不可变的，则失败原子性是免费的。如果操作失败，则可能会阻止

创建新对象，但它永远不会使现有对象处于不一致状态，因为每个对象的状态在创建时都是一致的，之后无法修改。

对于对可变对象进行操作的方法，实现失败原子性的最常用方法是在执行操作之前检查参数的有效性（第 49 项）。这导致在对象修改开始之前抛出大多数异常。例如，考虑第 7 项中的 Stack.pop 方法：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

如果消除了初始大小检查，则该方法在尝试从空堆栈中弹出元素时仍会抛出异常。但是，它会使 size 字段处于不一致（负）状态，从而导致对象的任何未来方法调用失败。另外，pop 方法抛出的 ArrayIndexOutOfBoundsException 对抽象是不合适的（Item 73）。

实现失败原子性的一种密切相关的方法是对计算进行排序，以便任何可能失败的部分发生在修改对象的任何部分之前。当不执行部分计算时无法检查参数时，此方法是前一个方法的自然扩展。例如，考虑 TreeMap 的情况，其元素按照某种顺序排序。为了向 TreeMap 添加元素，元素必须是可以使用 TreeMap 的顺序进行比较的类型。在以任何方式修改树之前，尝试添加错误键入的元素将自然会因为在树中搜索元素而导致 ClassCastException 失败。

实现故障原子性的第三种方法是对对象的临时副本执行操作，并在操作完成后用临时副本替换对象的内容。当数据已经存储在临时数据结构中时，可以更快地执行计算，这种方法自然发生。例如，某些排序函数在排序之前将其输入列表复制到数组中，以降低访问排序内部循环中元素的成本。这样做是为了提高性能，但作为额外的好处，它确保在排序失败时输入列表不会受到影响。

实现故障原子性的最后且不太常见的方法是编写恢复代码，该代码拦截在操作中发生的故障，并使对象将其状态回滚到操作开始之前的点。此方法主要用于持久（基于磁盘）的数据结构。

虽然通常需要失效原子性，但并不总是可以实现。例如，如果两个线程尝试在没有适当同步的情况下同时修改同一对象，则该对象可能处于不一致状态。因此，假设在捕获 ConcurrentModificationException 之后对象仍然可用，那将是错误的。错误是不可恢复的，因此在抛出 AssertionError 时甚至不需要尝试保留失败原子性。

即使在可能存在故障原子性的情况下，也并非总是如此。对于某些操作，它会显着增加成本或复杂性。也就是说，一旦你意识到这个问题，通常都可以自由而轻松地实现故障原子性。

总之，作为规则，任何生成的异常都是方法规范的一部分，应该使对象处于方法调用之前的状态。违反此规则的地方，API 文档应清楚地指出该对象将保留在哪种状态。遗憾的是，许多现有的 API 文档无法实现这一理想。

77 不要忽略异常

虽然这个建议可能看起来很明显，但它经常被违反而且需要重复。 当 API 的设计者声明一个抛出异常的方法时，他们会试图告诉你一些事情。 不要忽视它！ 通过使用 catch 块为空的 try 语句包围方法调用，可以轻松忽略异常：

```
// Empty catch block ignores exception - Highly suspect!
try {
        ...
}
catch (SomeException e) {
}
```

空的捕获块会破坏异常的目的，这会迫使您处理异常情况。 忽略一个例外类似于忽略一个火灾警报 - 并将其关闭，这样就没有其他人有机会看到是否有真正的火灾。 你可能会逃脱它，或者结果可能是灾难性的。 每当你看到一个空的挡块时，你的头上就会响起警铃。

在某些情况下，忽略异常是合适的。 例如，关闭 FileInputStream 可能是合适的。 您尚未更改文件的状态，因此无需执行任何恢复操作，并且您已经从文件中读取了所需的信息，因此没有理由中止正在进行的操作。 记录异常可能是明智的，这样如果经常发生这些异常，您就可以调查此事。 如果您选择忽略异

常，catch 块应该包含一个注释，解释为什么这样做是合适的，并且该变量应该被命名为 ignored：

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // Default; guaranteed sufficient for any map
try {
    numColors = f.get(1L, TimeUnit.SECONDS);
}
catch (TimeoutException | ExecutionException ignored) {
    // Use default: minimal coloring is desirable, not required
}
```

此项目中的建议同样适用于已检查和未检查的异常。 无论异常是代表可预测的异常情况还是编程错误，使用空 catch 块忽略它都会导致程序在出现错误时以静默方式继续运行。 然后，程序可能在将来的任意时间失败，代码中的某个点与问题的根源没有明显的关系。 正确处理异常可以完全避免失败。 仅仅让异常向外传播至少会导致程序迅速失败，保留信息以帮助调试失败。

第十一章 并发

THREADS 允许多个活动同时进行。 并发编程比单线程编程更难，因为更多的事情可能会出错，并且失败很难重现。 你无法避免并发。 它是平台中固有的，也是您要从多核处理器获得良好性能的要求，现在无处不在。 本章包含的建议可帮助您编写清晰，正确，记录完备的并发程序。

78 同步访问共享可变数据

synchronized 关键字确保只有一个线程可以一次执行方法或块。许多程序员认为同步只是作为一种互斥的手段，以防止一个对象在被另一个线程修改时被一个线程看到处于不一致状态。在此视图中，对象以一致状态（第 17 项）创建，并由访问它的方法锁定。这些方法观察状态并可选地导致状态转换，将对象从

一个一致状态转换为另一个状态。正确使用同步可确保任何方法都不会在不一致的状态下观察对象。

这个观点是正确的，但它只是故事的一半。如果没有同步，其他线程可能看不到一个线程的更改。同步不仅阻止线程观察处于不一致状态的对象，而且还确保进入同步方法或块的每个线程都能看到由同一个锁保护的所有先前修改的效果。

语言规范保证读取或写入变量是原子的，除非变量的类型为 long 或 double [JLS，17.4,17.7]。换句话说，读取 long 或 double 以外的变量可以保证返回某个线程存储到该变量中的值，即使多个线程同时修改变量而没有同步也是如此。

您可能听说它说要提高性能，在读取或写入原子数据时应该省去同步。这个建议是危险的错误。虽然语言规范保证线程在读取字段时不会看到任意值，但它不能保证一个线程写入的值对另一个线程可见。线程之间的可靠通信以及互斥是必需的。这是由于语言规范的一部分称为内存模型，它指定了一个线程所做的更改何时以及如何变得对其他人可见[JLS，17.4; Goetz06,16]。

即使数据是原子可读和可写的，未能同步对共享可变数据的访问的后果也是可怕的。 考虑从另一个线程停止一个线程的任务。 这些库提供了 Thread.stop 方法，但是这个方法很久以前就被弃用了，因为它本质上是不安全的 - 使用它会导致数据损坏。 不要使用 Thread.stop。 从另一个线程中停止一个线程的推荐方法是让第一个线程轮询一个最初为 false 的布尔字段，但是第二个线程可以设置为 true 以指示第一个线程要自行停止。因为读取和写入布尔字段是原子的，所以一些程序员在访问字段时不需要同步：

```
// Broken! - How long would you expect this program to run?
public class StopThread {
    private static boolean stopRequested;
    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
        int i = 0;
        while (!stopRequested)
            i++;
        });
```

```
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
        }
    }
```

您可能希望此程序运行大约一秒钟，之后主线程将 stopRequested 设置为 true，从而导致后台线程的循环终止。 但是，在我的机器上，程序永远不会终止：后台线程永远循环！

问题是在没有同步的情况下，无法确保后台线程何时（如果有的话）将看到主线程所做的 stopRequested 值的变化。 在没有同步的情况下，虚拟机转换此代码是完全可以接受的：

```
    while (!stopRequested)
    i++;
    into this code:
    if (!stopRequested)
    while (true)
    i++;
```

这种优化称为提升，它正是 OpenJDK Server VM 所做的。结果是活泼失败：程序无法取得进展。 解决问题的一种方法是同步对 stopRequested 字段的访问。正如预期的那样，该程序大约一秒钟终止：

```
    // Properly synchronized cooperative thread termination
    public class StopThread {
        private static boolean stopRequested;
        private static synchronized void requestStop() {
            stopRequested = true;
        }
```

```
        private static synchronized boolean stopRequested() {
            return stopRequested;
        }
        public static void main(String[] args) throws InterruptedException {
            Thread backgroundThread = new Thread(() -> {
                int i = 0;
                while (!stopRequested())
                i++;
            });
            backgroundThread.start();
            TimeUnit.SECONDS.sleep(1);
            requestStop();
        }
    }
```

请注意，write 方法（requestStop）和 read 方法（stop-Requested）都是同步的。仅同步 write 方法是不够的！除非读取和写入操作同步，否则不保证同步有效。偶尔只能同步写入（或读取）的程序似乎可以在某些机器上运行，但在这种情况下，外观是欺骗性的。

即使没有同步，StopThread 中的 synchronized 方法的操作也将是原子的。换句话说，这些方法的同步仅用于其通信效果，而不是用于互斥。虽然在循环的每次迭代中同步的成本很小，但是有一个正确的替代方案，其不那么冗长且性能可能更好。如果将 stopRequested 声明为 volatile，则可以省略第二版 StopThread 中的锁定。虽然 volatile 修饰符不执行互斥，但它保证读取该字段的任何线程都将看到最近写入的值：

```
    // Cooperative thread termination with a volatile field
    public class StopThread {
        private static volatile boolean stopRequested;
        public static void main(String[] args) throws InterruptedException {
```

```
        Thread backgroundThread = new Thread(() -> {
        int i = 0;
        while (!stopRequested)
            i++;
    });
    backgroundThread.start();
    TimeUnit.SECONDS.sleep(1);
    stopRequested = true;
    }
}
```

使用 volatile 时你必须要小心。 考虑以下方法，该方法应该生成序列号：

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;
public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

该方法的目的是保证每次调用都返回一个唯一值（只要调用次数不超过 232 次）。方法的状态由单个可原子访问的字段 nextSerialNumber 组成，该字段的所有可能值都是合法的。因此，不需要同步来保护其不变量。但是，如果没有同步，该方法将无法正常工作。

问题是增量运算符（++）不是原子的。它对 nextSerialNumber 字段执行两个操作：首先它读取值，然后它写回一个新值，等于旧值加 1。如果第二个线程在线程读取旧值并写回新值之间读取字段，则第二个线程将看到与第一个线程相同的值并返回相同的序列号。这是安全故障：程序计算错误的结果。

修复 generateSerialNumber 的一种方法是将 synchronized 修饰符添加到其声明中。这确保了多个调用不会交错，并且每次调用该方法都会看到所有先前调用的效果。完成后，您可以并且应该从 nextSerialNumber 中删除 volatile 修饰符。

要防止该方法，请使用 long 而不是 int，或者如果 nextSerialNumber 即将换行则抛出异常。

更好的是，遵循第 59 条中的建议并使用类 AtomicLong，它是 java.util.concurrent.atomic 的一部分。该软件包为单个变量提供了无锁，线程安全编程的原语。虽然 volatile 只提供同步的通信效果，但这个包也提供了原子性。这正是我们想要的 generateSerialNumber，它可能胜过同步版本：

```
// Lock-free synchronization with java.util.concurrent.atomic

private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {

    return nextSerialNum.getAndIncrement();

}
```

避免此项中讨论的问题的最佳方法是不共享可变数据。共享不可变数据（第 17 项）或根本不共享。换句话说，将可变数据限制在单个线程中。如果您采用此策略，则必须对其进行记录，以便在程序发展时维护策略。深入了解您正在使用的框架和库也很重要，因为它们可能会引入您不知道的线程。

一个线程可以修改数据对象一段时间然后与其他线程共享它，只同步共享对象引用的行为。然后，其他线程可以在不进一步同步的情况下读取对象，只要它不再被修改即可。据说这些物体实际上是不可改变的[Goetz06,3.5.4]。将这样的对象引用从一个线程转移到其他线程称为安全发布[Goetz06,3.5.3]。有许多方法可以安全地发布对象引用：您可以将它作为类初始化的一部分存储在静态字段中;您可以将它存储在易失性字段，最终字段或通过正常锁定访问的字段中;或者你可以将它放入并发集合中（第 81 项）。

总之，当多个线程共享可变数据时，每个读取或写入数据的线程都必须执行同步。在没有同步的情况下，无法保证一个线程的更改对另一个线程可见。未能同步共享可变数据的处罚是活跃性和安全性故障。这些失败是最难调试的。它们可以是间歇性的和时间相关的，并且程序行为可以从一个 VM 到另一个 VM 发生根本变化。如果您只需要线程间通信，而不是互斥，则 volatile 修饰符是可接受的同步形式，但正确使用可能很棘手。

79 避免过度同步

项目 78 警告同步不足的危险。这个项目涉及相反的问题。根据具体情况，过度同步可能会导致性能降低，死锁或甚至出现不确定行为。

为避免活动和安全故障，请勿在同步方法或块中将控制权交给客户端。换句话说，在同步区域内，不要调用设计为被覆盖的方法，也不要调用客户端以函数对象的形式提供的方法（第 24 项）。从具有同步区域的类的角度来看，这种方法是陌生的。该类不知道该方法的作用，也无法控制它。根据外来方法的作用，从同步区域调用它可能会导致异常，死锁或数据损坏。

为了使这个具体，请考虑以下类，它实现了一个可观察的集合包装器。它允许客户端在元素添加到集合时订阅通知。这是观察者模式[Gamma95]。为简洁起见，当从集合中删除元素时，类不提供通知，但提供它们将是一件简单的事情。此类在第 18 项（第 90 页）中可重用的 ForwardingSet 顶部实现：

```
// Broken - invokes alien method from synchronized block!
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers= new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }

    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
```

```
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }

    @Override
    public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element); // Calls notifyElementAdded
        return result;
    }
}
```

观察者通过调用 addObserver 方法订阅通知，并通过调用 removeObserver 方法取消订阅。 在这两种情况下，都会将此回调接口的实例传递给该方法。

```
@FunctionalInterface public interface SetObserver<E> {
// Invoked when an element is added to the observable set
void added(ObservableSet<E> set, E element);
}
```

该接口在结构上与 BiConsumer <ObservableSet <E>，E>相同。 我们选择定义自定义功能接口，因为接口和方法名称使代码更具可读性，并且因为接口可以演变为包含多个回调。 也就是说，使用 BiConsumer 也可以做出合理的论证（议题 44）。

在粗略检查中，ObservableSet 似乎工作正常。 例如，以下程序打印 0 到 99 之间的数字：

```
public static void main(String[] args) {
    ObservableSet<Integer> set =new ObservableSet<>(new HashSet<>());
    set.addObserver((s, e) -> System.out.println(e));
    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

现在让我们尝试更有趣的东西。 假设我们将一个 addObserver 调用替换为一个传递一个观察者的调用，该观察者打印添加到集合中的 Integer 值，如果值为 23 则自行删除：

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
```

```
});
```

请注意，此调用使用匿名类实例代替上一次调用中使用的 lambda。这是因为函数对象需要将自身传递给 s.removeObserver，而 lambdas 不能自己访问（第 42 项）。

您可能希望程序打印 0 到 23 的数字，之后观察者将取消订阅并且程序将以静默方式终止。实际上，它打印这些数字然后抛出 ConcurrentModificationException。问题是 notifyElementAdded 在调用观察者添加的方法时正在迭代观察者列表。添加的方法调用 observable set 的 removeObserver 方法，该方法又调用方法 observers.remove。现在我们遇到了麻烦。我们试图在迭代它的过程中从列表中删除一个元素，这是非法的。 notifyElementAdded 方法中的迭代在同步块中以防止并发修改，但它不会阻止迭代线程本身回调到可观察集并修改其观察者列表。

现在让我们尝试一些奇怪的事情：让我们编写一个尝试取消订阅的观察者，但不是直接调用 removeObserver，而是使用另一个线程的服务来执行契约。该观察者使用执行者服务（第 80 项）：

```
// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService exec =Executors.newSingleThreadExecutor();
            try {
                exec.submit(() -> s.removeObserver(this)).get();
            } catch (ExecutionException | InterruptedException ex) {
                throw new AssertionError(ex);
            } finally {
                exec.shutdown();
            }
        }
```

```
        }
    });
```

顺便提一下，请注意，此程序在一个 catch 子句中捕获两种不同的异常类型。Java 7 中添加了这种非常称为 multi-catch 的工具。它可以极大地提高清晰度并减小程序的大小，这些程序在响应多种异常类型时的行为方式相同。

当我们运行这个程序时，我们没有得到例外;我们陷入僵局。后台线程调用 s.removeObserver，它试图锁定观察者，但它无法获取锁，因为主线程已经有锁。一直以来，主线程都在等待后台线程完成删除观察者，这解释了死锁。

这个例子是设计的，因为观察者没有理由使用后台线程来取消订阅，但问题是真实的。从同步区域内调用外来方法已在实际系统中引起许多死锁，例如 GUI 工具包。

在前面的两个例子中（异常和死锁）我们很幸运。当调用外来方法（添加）时，由同步区域（观察者）保护的资源处于一致状态。假设您要从同步区域调用外来方法，而受同步区域保护的不变量暂时无效。因为 Java 编程语言中的锁是可重入的，所以这样的调用不会死锁。与导致异常的第一个示例一样，调用线程已经保持锁定，因此线程在尝试重新获取锁定时将成功，即使锁定保护的数据正在进行另一个概念上不相关的操作。这种失败的后果可能是灾难性的。从本质上讲，锁定未能完成其工作。重入锁简化了多线程面向对象程序的构建，但它们可以将活动失败转化为安全故障。

幸运的是，通过将异步方法调用移出同步块来解决此类问题通常并不困难。对于 notifyElementAdded 方法，这涉及获取观察者列表的“快照”，然后可以在没有锁定的情况下安全地遍历。通过此更改，前面的两个示例都运行无异常或死锁：

```
    // Alien method moved outside of synchronized block - open calls
    private void notifyElementAdded(E element) {
        List<SetObserver<E>> snapshot = null;
        synchronized(observers) {
            snapshot = new ArrayList<>(observers);
        }
        for (SetObserver<E> observer :snapshot)
```

```
            observer.added(this, element);
    }
```

The add and addAll methods of ObservableSet need not be changed if the list is modified to use CopyOnWriteArrayList. Here is how the remainder of the class looks. Notice that there is no explicit synchronization whatsoever:

实际上，有一种更好的方法可以将异类方法调用移出 synchronized 块。 这些库提供了一个名为 CopyOnWriteArrayList 的并发集合（Item 81），它是为此目的而量身定制的。 此 List 实现是 ArrayList 的一种变体，其中所有修改操作都是通过制作整个底层数组的新副本来实现的。 因为内部数组永远不会被修改，所以迭代不需要锁定并且非常快。 对于大多数用途，CopyOnWriteArrayList 的性能会很糟糕，但它非常适合观察者列表，这些列表很少被修改并经常遍历。

如果修改列表以使用 CopyOnWriteArrayList，则无需更改 ObservableSet 的 add 和 addAll 方法。 以下是该类其余部分的外观。 请注意，没有任何明确的同步：

```
    // Thread-safe observable set with CopyOnWriteArrayList
    private final List<SetObserver<E>> observers =new
CopyOnWriteArrayList<>();
    public void addObserver(SetObserver<E> observer) {
        observers.add(observer);
    }
    public boolean removeObserver(SetObserver<E> observer) {
        return observers.remove(observer);
    }
    private void notifyElementAdded(E element) {
        for (SetObserver<E> observer : observers)
            observer.added(this, element);
    }
```

在同步区域之外调用的外来方法称为开放调用[Goetz06,10.1.4]。除了防止失败，开放调用可以大大增加并发性。外来方法可能会持续任意长时间。如果从同步区域调用 alien 方法，则将不允许其他线程访问受保护资源。

通常，您应该在同步区域内尽可能少地工作。获取锁，检查共享数据，根据需要进行转换，然后取消锁定。如果您必须执行一些耗时的活动，请找到一种方法将其移出同步区域，而不违反第 78 项中的准则。

这个项目的第一部分是关于正确性。现在让我们简要介绍一下性能。虽然自 Java 早期以来同步成本已经大幅下降，但重要的是不要过度同步。在多核世界中，过度同步的实际成本不是获得锁定所花费的 CPU 时间;这是争论：并行性失去的机会以及确保每个核心都有一致的记忆观点的需要所造成的延迟。过度同步的另一个隐藏成本是它可以限制 VM 优化代码执行的能力。

如果您正在编写可变类，则有两个选项：您可以省略所有同步并允许客户端在需要并发使用时从外部进行同步，或者您可以在内部进行同步，从而使类具有线程安全性（第 82 项）。只有当您通过内部同步实现显着更高的并发性时，才应选择后一个选项，而不是让客户端在外部锁定整个对象。 java.util 中的集合（过时的 Vector 和 Hashtable 除外）采用前一种方法，而 java.util.concurrent 中的集合采用后者（项目 81）。

在 Java 的早期，许多类违反了这些准则。例如，StringBuffer 实例几乎总是由单个线程使用，但它们执行内部同步。正是由于这个原因，StringBuffer 被 StringBuilder 取代，而 StringBuilder 只是一个不同步的 StringBuffer。同样，java.util.Random 中的线程安全伪随机数生成器被 java.util.concurrent.ThreadLocalRandom 中的非同步实现取代也是很大一部分原因。如有疑问，请不要同步您的类，但要记录它不是线程安全的。

如果在内部同步类，则可以使用各种技术来实现高并发性，例如锁定拆分，锁定条带化和非阻塞并发控制。这些技术超出了本书的范围，但在其他地方也有讨论[Goetz06，Herlihy08]。

如果方法修改了静态字段，并且有可能从多个线程调用该方法，则必须在内部同步对该字段的访问（除非该类可以容忍非确定性行为）。多线程客户端无法在此类方法上执行外部同步，因为不相关的客户端可以在不同步的情况下调用该方法。该字段本质上是一个全局变量，即使它是私有的，因为它可以由不相关的客户端读取和修改。第 78 项中方法 generateSerialNumber 使用的 nextSerialNumber 字段举例说明了这种情况。

总之，为避免死锁和数据损坏，请勿在同步区域内调用外来方法。更一般地说，将您在同步区域内完成的工作量保持在最低水平。在设计可变类时，请考虑是否应该进行自己的同步。在多核时代，不要过度同步比以往任何时候都

重要。只有在有充分理由的情况下才能在内部同步您的课程，并清楚地记录您的决定（第 82 项）。

80 executors，task，stream 优于线程 @

本书的第一版包含一个简单工作队列的代码[Bloch01，第 49 项]。 此类允许客户端将后台线程的异步处理工作排入队列。 当不再需要工作队列时，客户端可以调用一个方法，要求后台线程在完成队列中已有的任何工作后正常终止自身。 实现只不过是一个玩具，但即便如此，它还需要一整页精细，细致的代码，如果你没有恰到好处的话，这种代码很容易出现安全和活动失败。 幸运的是，没有理由再编写这种代码了。

到本书第二版出版时，java.util.concurrent 已添加到 Java 中。 该软件包包含一个 Executor Framework，它是一个灵活的基于接口的任务执行工具。 创建一个比本书第一版更好的工作队列只需要一行代码：

```
ExecutorService exec = Executors.newSingleThreadExecutor();

Here is how to submit a runnable for execution:

exec.execute(runnable);

And here is how to tell the executor to terminate gracefully (if you fail to do this,it is likely that your VM will not exit):

exec.shutdown();
```

您可以使用执行程序服务执行更多操作。例如，您可以等待特定任务完成（使用 get 方法，如第 79 页的第 79 项所示），您可以等待任何或所有任务集合完成（使用 invokeAny 或 invokeAll 方法），您可以等待执行程序服务终止（使用 awaitTermination 方法），您可以在完成任务时逐个检索任务结果（使用 ExecutorCompletionService），您可以安排任务在特定时间运行或定期运行（使用 ScheduledThreadPoolExecutor），依此类推。

如果您希望多个线程处理来自队列的请求，只需调用另一个静态工厂，该工厂创建一种称为线程池的不同类型的执行器服务。您可以创建具有固定或可变数量线程的线程池。 java.util.concurrent.Executors 类包含静态工厂，它们提供了您需要的大多数执行程序。但是，如果您想要一些与众不同的东西，可以直接使用 ThreadPoolExecutor 类。此类允许您配置线程池操作的几乎每个方面。

为特定应用程序选择执行程序服务可能很棘手。对于小程序或负载较轻的服务器，Executors.newCachedThreadPool 通常是一个不错的选择，因为它不需要配置，通常“做正确的事情。”但是对于负载很重的生产服务器来说，缓存的线程池不是一个好的选择！在缓存的线程池中，提交的任务不会排队，而是立即传递给线程执行。如果没有可用的线程，则创建一个新线程。如果服务器负载过重以至于所有 CPU 都被充分利用并且更多任务到达，则会创建更多线程，这只会使事情变得更糟。因此，在负载很重的生产服务器中，最好使用 Executors.newFixedThreadPool，它为您提供具有固定线程数的池，或直接使用 ThreadPoolExecutor 类，以实现最大程度的控制。

您不仅应该避免编写自己的工作队列，而且通常应该避免直接使用线程。当您直接使用线程时，线程既可以作为工作单元，也可以作为执行它的机制。在执行程序框架中，工作单元和执行机制是分开的。关键的抽象是工作单元，这是任务。有两种任务：Runnable 及其近亲，Callable（类似于 Runnable，除了它返回一个值并且可以抛出任意异常）。执行任务的一般机制是执行程序服务。如果您考虑任务并让执行程序服务为您执行它们，您可以灵活地选择适当的执行策略以满足您的需求，并在需求发生变化时更改策略。本质上，Executor Framework 执行 Collections Framework 为聚合所做的工作。

在 Java 7 中，Executor Framework 被扩展为支持 fork-join 任务，这些任务由称为 fork-join 池的特殊执行器服务运行。由 ForkJoinTask 实例表示的 fork-join 任务可以拆分为较小的子任务，而包含 ForkJoinPool 的线程不仅处理这些任务，而且还“彼此”窃取“任务”以确保所有线程都保持忙碌，从而导致更高的任务 CPU 利用率，更高的吞吐量和更低的延迟。编写和调优 fork-join 任务很棘手。并行流（第 48 项）写在 fork 连接池的顶部，允许您轻松地利用它们的性能优势，假设它们适合于手头的任务。

对 Executor Framework 的完整处理超出了本书的范围，但感兴趣的读者可以参考 Java Concurrency in Practice [Goetz06]。

81 并发工具优于 wait 和 notify

本书的第一版专门用一个项目来正确使用等待和通知[Bloch01，Item 50]。它的建议仍然有效，并在本项末尾进行了总结，但这个建议远不如以前那么重要。这是因为使用 wait 和 notify 的原因要少得多。从 Java 5 开始，该平台提供了更高级别的并发实用程序，可以执行以前必须在等待和通知时手动编写代码的各种操作。鉴于正确使用 wait 和 notify 的困难，您应该使用更高级别的并发实用程序。

java.util.concurrent 中的高级实用程序分为三类：Executor Framework，在 Item 80 中简要介绍了它;并发集合;和同步器。本项简要介绍了并发集合和同步器。

并发集合是标准集合接口（如 List，Queue 和 Map）的高性能并发实现。为了提供高并发性，这些实现在内部管理自己的同步（第 79 项）。因此，不可能从并发集合中排除并发活动;锁定它只会减慢程序。

因为您不能排除并发集合上的并发活动，所以您也不能以原子方式组合对它们的方法调用。因此，并发集合接口配备了依赖于状态的修改操作，这些操作将几个原语组合成单个原子操作。事实证明，这些操作对并发集合非常有用，它们使用默认方法（第 21 项）添加到 Java 8 中相应的集合接口中。

例如，Map 的 putIfAbsent（key，value）方法插入键的映射（如果不存在）并返回与键关联的先前值，如果没有则返回 null。这样可以轻松实现线程安全的规范化映射。此方法模拟 String.intern 的行为：

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal
private static final ConcurrentMap<String, String> map =new
ConcurrentHashMap<>();
public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

事实上，你可以做得更好。 ConcurrentHashMap 针对检索操作进行了优化，例如 get。 因此，如果 get 表明有必要，最初只需调用 get 并调用 putIfAbsent：

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
        result = s;
```

```
        }

        return result;

    }
```

除了提供出色的并发性外，ConcurrentHashMap 非常快。在我的机器上，上面的实习方法比 String.intern 快 6 倍（但请记住，String.intern 必须采用一些策略来防止在长期存在的应用程序中泄漏内存）。并发集合使同步集合在很大程度上已经过时。例如，使用 ConcurrentHashMap 优先于 Collections.synchronizedMap。简单地用并发映射替换同步映射可以显着提高并发应用程序的性能。

一些集合接口使用阻塞操作进行扩展，这些操作等待（或阻塞）直到可以成功执行。例如，BlockingQueue 扩展了 Queue 并添加了几个方法，包括 take，它从队列中删除并返回 head 元素，等待队列为空。这允许阻塞队列用于工作队列（也称为生产者 - 消费者队列），一个或多个生产者线程将工作项排入其中，并且一个或多个消费者线程从哪个队列变为可用时出列并处理项目。正如您所期望的那样，大多数 ExecutorService 实现（包括 ThreadPoolExecutor）都使用 BlockingQueue（Item 80）。

同步器是使线程能够彼此等待的对象，允许它们协调它们的活动。最常用的同步器是 CountDownLatch 和 Semaphore。不太常用的是 CyclicBarrier 和 Exchanger。最强大的同步器是 Phaser。

倒计时锁存器是一次性使用的屏障，允许一个或多个线程等待一个或多个其他线程执行某些操作。 CountDownLatch 的唯一构造函数接受一个 int，它是在允许所有等待的线程继续之前必须在 latch 上调用 countDown 方法的次数。

在这个简单的原语上构建有用的东西是非常容易的。例如，假设您要构建一个简单的框架来计算操作的并发执行时间。该框架由一个执行器执行操作的单个方法，一个表示要并发执行的操作数的并发级别以及一个表示该操作的 runnable 组成。在计时器线程启动时钟之前，所有工作线程都准备好自己运行操作。当最后一个工作线程准备好运行该操作时，计时器线程“触发起始枪”，允许工作线程执行操作。一旦最后一个工作线程完成执行操作，计时器线程就会停止计时。直接在等待和通知的基础上实现这个逻辑至少可以说是混乱，但在 CountDownLatch 之上它是令人惊讶的直截了当：

```
    // Simple framework for timing concurrent execution

    public static long time(Executor executor, int concurrency,Runnable action)
throws InterruptedException {
```

```
        CountDownLatch ready = new CountDownLatch(concurrency);
        CountDownLatch start = new CountDownLatch(1);
        CountDownLatch done = new CountDownLatch(concurrency);
        for (int i = 0; i < concurrency; i++) {
            executor.execute(() -> {
                ready.countDown(); // Tell timer we're ready
                try {
                    start.await(); // Wait till peers are ready
                    action.run();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    done.countDown(); // Tell timer we're done
                }
            });
        }
        ready.await(); // Wait for all workers to be ready
        long startNanos = System.nanoTime();
        start.countDown(); // And they're off!
        done.await(); // Wait for all workers to finish
        return System.nanoTime() - startNanos;
    }
```

请注意，该方法使用三个倒计时锁存器。第一个就绪，由工作线程用来告诉计时器线程何时准备就绪。工作线程然后等待第二个锁存器，这是开始。当最后一个工作线程调用 ready.countDown 时，计时器线程记录开始时间并调用 start.countDown，允许所有工作线程继续。然后，计时器线程等待第三个锁存器

完成，直到最后一个工作线程完成运行并调用 done.countDown。一旦发生这种情况，计时器线程就会唤醒并记录结束时间。

还有一些细节需要注意。传递给 time 方法的执行程序必须允许创建至少与给定并发级别一样多的线程，否则测试将永远不会完成。这被称为线程饥饿僵局[Goetz06,8.1.1]。如果一个工作线程捕获一个 InterruptedException，它会使用成语 Thread.currentThread（）。interrupt（）重新断言该中断并从其 run 方法返回。这允许执行程序在其认为合适时处理中断。请注意，System.nanoTime 用于计算活动的时间。对于间隔计时，始终使用 System.nanoTime 而不是 System.currentTimeMillis.System.nanoTime 更准确，更精确，并且不受系统实时时钟调整的影响。最后，请注意，此示例中的代码不会产生准确的计时，除非操作执行了大量工作，例如一秒钟或更长时间。准确的微基准测试是非常困难的，最好借助于 jmh [JMH]等专用框架来完成。

这个项目只涉及使用并发实用程序可以做的事情的表面。例如，前一个示例中的三个倒计时锁存器可以由单个 CyclicBarrier 或 Phaser 实例替换。结果代码会更简洁，但可能更难理解。

虽然您应该始终优先使用并发实用程序来等待并通知，但您可能必须维护使用 wait 和 notify 的旧代码。 wait 方法用于使线程等待某些条件。必须在同步区域内调用它，该区域锁定调用它的对象。这是使用 wait 方法的标准习惯用法：

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(); // (Releases lock, and reacquires on wakeup)
    ... // Perform action appropriate to condition
}
```

始终使用 wait 循环习惯来调用 wait 方法; 永远不要在循环之外调用它。 循环用于测试等待前后的状况。

如果条件已经存在，则在等待之前测试条件并跳过等待以确保活跃。 如果条件已经存在并且在线程等待之前已经调用了 notify（或 notifyAll）方法，则无法保证线程将从等待中唤醒。

如果条件不成立，等待和等待后再次测试条件是必要的，以确保安全。 如果线程在条件不成立时继续执行操作，则可以销毁由锁保护的不变量。 当条件不成立时，线程可能会唤醒的原因有多种：

另一个线程可以获得锁并在线程调用 notify 和等待线程醒来之间改变了保护状态。

当条件不成立时，另一个线程可能会意外或恶意地调用通知。 类通过等待可公开访问的对象来暴露自己这种恶作剧。 在公共可访问对象的同步方法中的任何等待都容易受到此问题的影响。

通知线程在唤醒等待线程时可能过于“慷慨”。 例如，即使只有一些等待的线程满足条件，通知线程也可以调用 notifyAll。

等待线程可以（很少）在没有通知的情况下唤醒。 这被称为虚假唤醒[POSIX，11.4.3.6.1;Java9-API]。

相关问题是是否使用 notify 或 notifyAll 来唤醒等待线程。 （回想一下，如果存在这样一个线程，notify 会唤醒一个等待的线程，并且 notifyAll 会唤醒所有等待的线程。）有时候你应该总是使用 notifyAll。这是合理的，保守的建议。它总是会产生正确的结果，因为它可以保证你唤醒需要被唤醒的线程。您也可以唤醒其他一些线程，但这不会影响程序的正确性。这些线程将检查它们正在等待的条件，并且发现它为假，将继续等待。

作为优化，如果可能在等待集中的所有线程都在等待相同的条件并且一次只有一个线程可以从条件变为 true 中受益，则可以选择调用 notify 而不是 notifyAll。

即使满足这些先决条件，也可能有理由使用 notifyAll 代替通知。正如将循环中的等待调用置于可公开访问的对象上的意外或恶意通知一样，使用 notifyAll 代替通知可以防止不相关的线程发生意外或恶意等待。否则，这样的等待可以“吞下”关键通知，使其预期接收者无限期地等待。

总之，与 java.util.concurrent 提供的高级语言相比，直接使用 wait 和 notify 就像在“并发汇编语言”中编程一样。很少，如果有的话，在新代码中使用 wait 和 notify 的理由。如果您维护使用 wait 和 notify 的代码，请确保它始终使用标准惯用法在 while 循环内调用 wait。通常应优先使用 notifyAll 方法进行通知。如果使用通知，必须非常小心以确保活跃。

82 线程安全文档化

当同时使用其方法时，类的行为方式是其与客户签订合同的重要部分。如果您未能记录某个类行为的这一方面，其用户将被迫做出假设。如果这些假设是错误的，则生成的程序可能执行不充分的同步（项目 78）或过度同步（项目 79）。无论哪种情况，都可能导致严重错误。

您可能会听到它说您可以通过在其文档中查找 synchronized 修饰符来判断方法是否是线程安全的。这有几点是错误的。在正常操作中，Javadoc 在其输出中不包含 synchronized 修饰符，并且有充分的理由。方法声明中 synchronized 修饰符的存在是一个实现细节，而不是其 API 的一部分。它不能可靠地表明方法是线程安全的。

此外，声称同步修改器的存在足以记录线程安全性的说法体现了线程安全是全有或全无的属性的误解。实际上，有几个级别的线程安全性。要启用安全的并发使用，类必须清楚地记录它支持的线程安全级别。以下列表总结了线程安全级别。它并非详尽无遗，但涵盖了常见情况：

不可变的 - 这个类的实例看起来是不变的。无需外部同步。示例包括 String，Long 和 BigInteger（第 17 项）。

无条件线程安全 - 此类的实例是可变的，但该类具有足够的内部同步，可以同时使用其实例而无需任何外部同步。示例包括 AtomicLong 和 ConcurrentHashMap。

有条件地线程安全 - 类似于无条件线程安全，除了某些方法需要外部同步以便安全并发使用。示例包括 Collections.synchronized 包装器返回的集合，其迭代器需要外部同步。

不是线程安全的 - 这个类的实例是可变的。要同时使用它们，客户端必须使用客户端选择的外部同步来包围每个方法调用（或调用序列）。示例包括通用集合实现，例如 ArrayList 和 HashMap。

线程恶意 - 即使每个方法调用都被外部同步包围，这个类对于并发使用也是不安全的。线程敌意通常是在没有同步的情况下修改静态数据。没有人故意写一个线索敌对的类;此类通常是由于未考虑并发而导致的。当发现类或方法是线程敌对的时，通常会修复或弃用它。如第 322 页所述，在没有内部同步的情况下，第 78 项中的 generateSerialNumber 方法将是线程不可靠的。

这些类别（除了线程恶意）大致对应于 Java Concurrency in Practice 中的线程安全注释，它们是 Immutable，ThreadSafe 和 NotThreadSafe [Goetz06，附录 A]。 上述分类中的无条件和条件线程安全类别都包含在 ThreadSafe 注释中。

记录有条件的线程安全类需要小心。 您必须指明哪些调用序列需要外部同步，以及必须获取哪个锁（或在极少数情况下，锁）才能执行这些序列。 通常它是实例本身的锁，但也有例外。 例如，Collections.synchronizedMap 的文档说明了这一点：

当迭代任何集合视图时，用户必须手动同步返回的地图：

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<>());

Set<K> s = m.keySet(); // Needn't be in synchronized block

...

synchronized(m) { // Synchronizing on m, not s!

    for (K key : s)

        key.f();

}
```

不遵循此建议可能会导致非确定性行为。

类的线程安全性的描述通常属于类的 doc 注释，但具有特殊线程安全属性的方法应在其自己的文档注释中描述这些属性。没有必要记录枚举类型的不变性。除非从返回类型中显而易见，否则静态工厂必须记录返回对象的线程安全性，如 Collections.synchronizedMap（上文）所示。

当类承诺使用可公开访问的锁时，它允许客户端以原子方式执行一系列方法调用，但这种灵活性需要付出代价。它与并发集合（如 ConcurrentHashMap）使用的高性能内部并发控制不兼容。此外，客户端可以通过长时间保持可公开访问的锁来发起拒绝服务攻击。这可以是偶然或有意的。

要防止此拒绝服务攻击，您可以使用私有锁对象而不是使用 synchronized 方法（这意味着可公开访问的锁）：

```
// Private lock object idiom - thwarts denial-of-service attack

private final Object lock = new Object();

public void foo() {
```

```
        synchronized(lock) {
            ...
        }
    }
```

由于私有锁对象在类外是不可访问的，因此客户端不可能干扰对象的同步。实际上，我们通过将锁定对象封装在它同步的对象中来应用第 15 项的建议。

请注意，锁定字段被声明为 final。这可以防止您无意中更改其内容，从而导致灾难性的非同步访问（第 78 项）。我们通过最小化锁定字段的可变性来应用第 17 项的建议。锁定字段应始终声明为 final。无论您使用普通的监视器锁（如上所示）还是使用 java.util.concurrent.locks 包中的锁，都是如此。

私有锁对象习惯用法只能用于无条件的线程安全类。有条件的线程安全类不能使用这个习惯用法，因为它们必须记录在执行某些方法调用序列时客户端要获取的锁。

私有锁对象习惯用法特别适合用于继承的类（第 19 项）。如果这样的类要使用其实例进行锁定，则子类可能容易且无意地干扰基类的操作，反之亦然。通过为不同的目的使用相同的锁，子类和基类可能最终“踩到彼此的脚趾。”这不仅仅是一个理论问题;它发生在 Thread 类[Bloch05，Puzzle 77]中。

总而言之，每个类都应该用一个措辞谨慎的散文描述或线程安全注释清楚地记录其线程安全属性。 synchronized 修饰符在本文档中不起作用。有条件的线程安全类必须记录哪些方法调用序列需要外部同步，哪些锁定在执行这些序列时获取。如果编写无条件的线程安全类，请考虑使用私有锁对象代替同步方法。这可以保护您免受客户端和子类的同步干扰，并使您可以更灵活地在以后的版本中采用复杂的并发控制方法。

83 慎用延迟初始化

延迟初始化是延迟字段初始化直到需要其值的行为。如果永远不需要该值，则永远不会初始化该字段。此技术适用于静态和实例字段。虽然延迟初始化主要是一种优化，但它也可以用来打破类和实例初始化中的有害循环[Bloch05，Puzzle 51]。

与大多数优化一样，延迟初始化的最佳建议是“除非你需要，否则不要这样做”（第 67 项）。懒惰的初始化是一把双刃剑。它降低了初始化类或创建实例的成本，但代价是增加了访问延迟初始化字段的成本。取决于这些字段最终需

要初始化的部分，初始化它们的成本是多少，以及初始化后每个字段的访问频率，延迟初始化（如许多“优化”）实际上会损害性能。

也就是说，延迟初始化有其用途。如果仅在类的一小部分实例上访问字段，并且初始化字段的成本很高，则延迟初始化可能是值得的。确切知道的唯一方法是使用和不使用延迟初始化来测量类的性能。

在存在多个线程的情况下，延迟初始化很棘手。如果两个或多个线程共享一个延迟初始化的字段，则必须采用某种形式的同步，否则可能导致严重的错误（第 78 项）。此项中讨论的所有初始化技术都是线程安全的。

在大多数情况下，正常初始化优于延迟初始化。以下是通常初始化的实例字段的典型声明。注意使用 final 修饰符（Item 17）：

```
// Normal initialization of an instance field
private final FieldType field = computeFieldValue();
```

如果使用延迟初始化来破坏初始化循环，请使用同步访问器，因为它是最简单，最清晰的替代方法：

```
// Lazy initialization of instance field - synchronized accessor
private FieldType field;
private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

当应用于静态字段时，这两个习惯用法（正常初始化和使用同步访问器的延迟初始化）都不会更改，除了您将 static 修饰符添加到字段和访问器声明。

如果需要在静态字段上使用延迟初始化来提高性能，请使用延迟初始化持有者类习惯用法。 这个成语利用了在使用类之前不会初始化类的保证[JLS，12.4.1]。 以下是它的外观：

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
```

```
        static final FieldType field = computeFieldValue();
    }
    private static FieldType getField() { return FieldHolder.field; }
```

当第一次调用 getField 时，它首次读取 FieldHolder.field，导致 FieldHolder 类的初始化。这个习惯用法的优点在于 getField 方法不是同步的，只执行字段访问，因此延迟初始化几乎不会增加访问成本。典型的 VM 将仅同步字段访问以初始化类。初始化类后，VM 会对代码进行修补，以便后续访问该字段不涉及任何测试或同步。

如果需要在实例字段上使用延迟初始化来提高性能，请使用双重检查惯用法。这个习惯用法避免了初始化后访问字段时锁定的成本（第 79 项）。成语背后的想法是检查字段的值两次（因此名称仔细检查）：一次没有锁定，然后，如果字段看起来是未初始化的，则第二次锁定。仅当第二次检查表明该字段未初始化时，该呼叫才会初始化该字段。因为字段初始化后没有锁定，所以将字段声明为 volatile 是非常重要的（第 78 项）。这是成语：

```
    // Double-check idiom for lazy initialization of instance fields
    private volatile FieldType field;
    private FieldType getField() {
        FieldType result = field;
        if (result == null) { // First check (no locking)
            synchronized(this) {
                if (field == null) // Second check (with locking)
                    field = result = computeFieldValue();
            }
        }
        return result;
    }
```

此代码可能看起来有点复杂。特别是，对局部变量（结果）的需求可能不清楚。这个变量的作用是确保该字段在已经初始化的常见情况下只读一次。

虽然不是绝对必要，但这可以提高性能，并且通过应用于低级并发编程的标准更加优雅。在我的机器上，上面的方法大约是没有局部变量的明显版本的 1.4 倍。虽然您也可以将双重检查成语应用于静态字段，但没有理由这样做：延迟初始化持有者类习惯用法是更好的选择。

双重检查成语的两个变种熊注意到。有时，您可能需要懒惰地初始化一个可以容忍重复初始化的实例字段。如果你发现自己处于这种情况，你可以使用复核的双重检查成语，省去第二次检查。毫不奇怪，它被称为单一检查成语。这是它的外观。请注意，该字段仍然声明为 volatile：

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;
private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}
```

本项中讨论的所有初始化技术都适用于原始字段以及对象引用字段。当将双重检查或单一检查惯用法应用于数字原始字段时，将针对 0（数字原始变量的默认值）而不是空来检查字段的值。

如果你不关心每个线程是否重新计算字段的值，并且字段的类型是 long 或 double 以外的原语，那么你可以选择从单一检查成语中的字段声明中删除 volatile 修饰符。这种变体被称为生动的单一检查成语。它加速了某些体系结构上的字段访问，但代价是额外的初始化（每个访问该字段的线程最多一个）。这绝对是一种奇特的技术，不适合日常使用。

总之，您应该正常初始化大多数字段，而不是懒惰。如果必须懒惰地初始化字段以实现性能目标或打破有害的初始化循环，则使用适当的延迟初始化技术。例如字段，它是双重检查成语;对于静态字段，惰性初始化持有者类成语。例如，可以容忍重复初始化的字段，您也可以考虑单一检查习语。

84 不要依赖线程调度器

当许多线程可以运行时，线程调度程序会确定哪些线程可以运行以及运行多长时间。任何合理的操作系统都会尝试公平地做出这个决定，但政策可能会有所不同。因此，精心编写的程序不应该依赖于本政策的细节。任何依赖于线程调度程序以获得正确性或性能的程序都可能是不可移植的。

编写健壮，响应迅速的可移植程序的最佳方法是确保可运行线程的平均数量不会明显大于处理器数量。这使得线程调度程序几乎没有选择：它只是运行可运行的线程，直到它们不再可运行。即使在完全不同的线程调度策略下，程序的行为也不会有太大变化。请注意，可运行线程的数量与线程总数不同，后者可能要高得多。等待的线程不可运行。

保持可运行线程数量较少的主要技术是让每个线程做一些有用的工作，然后等待更多。如果线程没有做有用的工作，它们就不应该运行。就 Executor Framework 而言（第 80 项），这意味着适当地调整线程池[Goetz06,8.2]并保持任务简短但不会太短，或者调度开销会损害性能。

线程不应该忙等待，反复检查等待其状态改变的共享对象。除了使程序容易受到线程调度程序的变幻莫测之外，忙等待大大增加了处理器的负担，减少了其他人可以完成的有用工作量。作为不该做的极端例子，请考虑 CountDownLatch 的这种反常重新实现：

```
// Awful CountDownLatch implementation - busy-waits incessantly!
public class SlowCountDownLatch {
    private int count;
    public SlowCountDownLatch(int count) {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }
    public void await() {
        while (true) {
            synchronized(this) {
```

```
                    if (count == 0)
                    return;
                }
            }
        }
        public synchronized void countDown() {
            if (count != 0)
                count--;
        }
    }
```

在我的机器上，当 1000 个线程在锁存器上等待时，SlowCountDownLatch 比 Java 的 CountDownLatch 慢大约十倍。虽然这个例子看起来有点牵强，但是看到一个或多个线程不必要地运行的系统并不罕见。性能和可移植性可能会受到影响。

当面对一个几乎无法工作的程序，因为某些线程没有相对于其他线程获得足够的 CPU 时间时，可以通过调用 Thread.yield 来抵制“修复”程序的诱惑。您可能会成功地使程序在时尚之后工作，但它不会是可移植的。在一个 JVM 实现上提高性能的相同 yield 调用可能会使其在一秒钟内变得更糟，对第三个没有影响。 Thread.yield 没有可测试的语义。更好的做法是重构应用程序以减少可并发运行的线程数。

类似警告适用的相关技术是调整线程优先级。线程优先级是 Java 中最不便携的功能之一。通过调整一些线程优先级来调整应用程序的响应性并不是不合理的，但它很少是必需的并且不可移植。尝试通过调整线程优先级来解决严重的活跃度问题是不合理的。在您找到并解决根本原因之前，问题可能会重新出现。

总之，不要依赖线程调度程序来确定程序的正确性。由此产生的程序既不健壮也不便携。作为推论，不要依赖 Thread.yield 或线程优先级。这些设施仅仅是调度程序的提示。可以谨慎地使用线程优先级来提高已经工作的程序的服务质量，但是它们永远不应该用于“修复”几乎不起作用的程序。

第十二章 序列化

本章涉及对象序列化，它是 Java 的框架，用于将对象编码为字节流（序列化）并从其编码中重构对象（反序列化）。 一旦对象被序列化，其编码可以从一个 VM 发送到另一个 VM 或存储在磁盘上以便以后反序列化。 本章重点介绍序列化的危险以及如何最小化序列化。

85 考虑其他可选择优于 Java 序列化 @

当序列化在 1997 年被添加到 Java 时，它被认为有点风险。该方法已经在一种研究语言（Modula-3）中尝试过，但从未用过生产语言。虽然程序员很少花费分布式对象的承诺是有吸引力的，但价格是隐形的构造函数和 API 与实现之间模糊的界限，可能存在正确性，性能，安全性和维护方面的问题。支持者认为这些好处超过了风险，但历史已经证明不是这样。

本书前几版中描述的安全问题与一些人担心的一样严重。在未来十年中讨论的漏洞在未来十年被转化为严重漏洞，其中包括对旧金山都市交通局市政铁路（SFMTA Muni）的勒索软件攻击，该铁路在 2016 年 11 月关闭整个收费系统两天[Gallagher16。

序列化的一个基本问题是它的攻击面太大而无法保护并且不断增长：通过在 ObjectInputStream 上调用 readObject 方法来反序列化对象图。这个方法本质上是一个神奇的构造函数，只要类型实现了 Serializable 接口，就可以在类路径上实例化几乎任何类型的对象。在反序列化字节流的过程中，此方法可以从任何这些类型执行代码，因此所有这些类型的代码都是攻击面的一部分。

攻击面包括 Java 平台库中的类，第二方库（如 Apache Commons Collections）和应用程序本身。即使您遵守所有相关的最佳实践并成功编写无法攻击的可序列化类，您的应用程序仍可能容易受到攻击。引用 CERT 协调中心技术经理 Robert Seacord 的话：

Java 反序列化是一个明显且存在的危险，因为它直接被应用程序广泛使用，并间接地由 Java 子系统（如 RMI（远程方法调用），JMX（Java 管理扩展）和 JMS（Java 消息系统））广泛使用。不受信任的流的反序列化可能导致远程代码执行（RCE），拒绝服务（DoS）以及一系列其他漏洞利用。应用程序即使没有做错也容易受到这些攻击。 [Seacord17]

攻击者和安全研究人员研究 Java 库和常用第三方库中的可序列化类型，查找在反序列化期间调用的执行潜在危险活动的方法。这种方法称为小工具。可以一起使用多个小工具来形成小工具链。有时会发现一个足够强大的小工具链，

允许攻击者在底层硬件上执行任意本机代码，只要有机会提交精心设计的字节流进行反序列化。这正是 SFMTA Muni 攻击中发生的事情。这次袭击没有孤立。还有其他人，会有更多。

在不使用任何小工具的情况下，您可以通过导致需要很长时间反序列化的短流的反序列化来轻松地发起拒绝服务攻击。这种流被称为反序列化炸弹[Svoboda16]。这是 Wouter Coekaerts 的一个例子，它只使用哈希集和字符串[Coekaerts15]：

```
// Deserialization bomb - deserializing this stream takes forever
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // Make t1 unequal to t2
        s1.add(t1); s1.add(t2);
        s2.add(t1); s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // Method omitted for brevity
}
```

对象图由 201 个 HashSet 实例组成，每个实例包含 3 个或更少的对象引用。整个流的长度为 5,744 字节，但是在你将其反序列化之前很久就会烧掉太阳。问题是反序列化 HashSet 实例需要计算其元素的哈希码。根哈希集的 2 个元素本身是包含 2 个哈希集元素的哈希集，每个哈希集元素包含 2 个哈希集元素，依此类推，深度为 100 个级别。因此，反序列化集会导致 hashCode 方法被调用超过

2100 次。除了反序列化永远存在的事实之外，解串器没有任何迹象表明任何问题。产生的对象很少，并且堆栈深度是有界的。

那么你能做些什么来抵御这些问题呢？每当您反序列化您不信任的字节流时，您就会打开攻击。避免序列化漏洞利用的最佳方法是永远不要反序列化任何东西。用 1983 年电影“战争游戏”中名为约书亚的电脑的话来说，“唯一的胜利就是不玩。”没有理由在你编写的任何新系统中使用 Java 序列化。还有其他在对象和字节序列之间进行转换的机制，可以避免 Java 序列化的许多危险，同时提供许多优势，例如跨平台支持，高性能，大型工具生态系统以及广泛的专业知识社区。在本书中，我们将这些机制称为跨平台结构化数据表示。虽然其他人有时将它们称为序列化系统，但本书避免了这种用法，以防止与 Java 序列化混淆。

这些表示的共同之处在于它们比 Java 序列化简单得多。它们不支持任意对象图的自动序列化和反序列化。相反，它们支持由一组属性 - 值对组成的简单结构化数据对象。仅支持少数原始数组和数组数据类型。这种简单的抽象结果足以构建极其强大的分布式系统，并且足够简单，可以避免从一开始就困扰 Java 序列化的严重问题。

领先的跨平台结构化数据表示是 JSON [JSON]和 Protocol Buffers，也称为 protobuf [Protobuf]。JSON 由 Douglas Crockford 设计用于浏览器 - 服务器通信，并且协议缓冲器由 Google 设计用于在其服务器之间存储和交换结构化数据。即使这些表示有时被称为语言中性，JSON 最初是为 JavaScript 开发的，而 forobobf 是为 C ++开发的;这两种表述都保留了其起源的痕迹。

JSON 和 protobuf 之间最显着的区别是 JSON 是基于文本的，人类可读的，而 protobuf 是二元的，效率更高;并且 JSON 完全是数据表示，而 protobuf 提供模式（类型）来记录和实施适当的用法。尽管 protobuf 比 JSON 更有效，但 JSON 对于基于文本的表示非常有效。虽然 protobuf 是二进制表示，但它确实提供了一种替代文本表示，用于需要人类可读性的用途（pbtxt）。

如果您无法完全避免 Java 序列化，可能是因为您在需要它的遗留系统的上下文中工作，那么您的下一个最佳选择是永远不会反序列化不受信任的数据。特别是，您永远不应接受来自不受信任来源的 RMI 流量。 Java 的官方安全编码指南说“不受信任的数据的反序列化本质上是危险的，应该避免。”这个句子设置为大，粗体，斜体，红色类型，它是整个文档中唯一获得此处理的文本[Java 的安全。

如果您无法避免序列化，并且您不确定要反序列化的数据的安全性，请使用 Java 9 中添加的对象反序列化过滤并向后移植到早期版本（java.io.ObjectInputFilter）。此工具允许您指定在反序列化之前应用于数据流

的过滤器。它以类粒度运行，允许您接受或拒绝某些类。默认接受类并拒绝潜在危险类列表称为黑名单;默认情况下拒绝类并接受假定安全的列表称为白名单。喜欢将白名单列入黑名单，因为黑名单只能保护您免受已知威胁。名为 Serial Whitelist Application Trainer（SWAT）的工具可用于为您的应用程序[Schneider16]自动准备白名单。过滤工具还可以保护您免受过多的内存使用和过深的对象图，但它不会保护您免受如上所示的序列化炸弹的攻击。

不幸的是，序列化在 Java 生态系统中仍然普遍存在。如果您要维护基于 Java 序列化的系统，请认真考虑迁移到跨平台的结构化数据表示，即使这可能是一项耗时的工作。实际上，您可能仍然发现自己必须编写或维护可序列化的类。编写一个正确，安全，高效的可序列化类需要非常小心。本章的其余部分提供了有关何时以及如何执行此操作的建议。

总之，序列化是危险的，应该避免。如果您从头开始设计系统，请使用跨平台的结构化数据表示，例如 JSON 或 protobuf。不要反序列化不受信任的数据。如果必须这样做，请使用对象反序列化过滤，但请注意，不能保证阻止所有攻击。避免编写可序列化的类。如果你必须这样做，请谨慎行事。

86 考虑使用自定义序列化形式

允许序列化类的实例可以像在其声明中添加实现 Serializable 的单词一样简单。 因为这很容易做到，所以有一种常见的误解，即序列化对程序员来说需要很少的努力。 事实要复杂得多。 虽然使类可序列化的直接成本可以忽略不计，但长期成本通常很高。

实现 Serializable 的一个主要成本是它降低了一旦发布后更改类的实现的灵活性。当类实现 Serializable 时，其字节流编码（或序列化形式）将成为其导出 API 的一部分。在广泛分发类之后，通常需要永久支持序列化表单，就像您需要支持导出的 API 的所有其他部分一样。如果您没有努力设计自定义序列化表单但仅接受默认值，则序列化表单将永远与类的原始内部表示相关联。换句话说，如果您接受默认的序列化表单，则该类的私有和包私有实例字段将成为其导出 API 的一部分，并且最小化对字段的访问（第 15 项）的做法将失去其作为信息隐藏工具的有效性。

如果接受默认的序列化表单并稍后更改类的内部表示，则将导致序列化表单中的不兼容更改。尝试使用旧版本的类序列化实例并使用新版本对其进行反序列化（反之亦然）的客户端将遇到程序失败。可以在保持原始序列化形式（使用 ObjectOutputStream.putFields 和 ObjectInputStream.readFields）的同时更改内部表示，但这可能很困难并且在源代码中留下可见的瑕疵。如果您选择将类序列化，您应该仔细设计一个您愿意长期生活的高质量序列化表单（第 87,90 项）。

这样做会增加开发的初始成本，但值得付出努力。即使是精心设计的序列化形式也会限制一个类的演变;一个设计不良的序列化形式可能会瘫痪。

可串行化所强加的进化约束的一个简单示例涉及流唯一标识符，通常称为串行版本 UID。每个可序列化的类都有一个与之关联的唯一标识号。如果未通过声明名为 serialVersionUID 的静态最终长字段来指定此数字，则系统会在运行时通过将加密哈希函数（SHA-1）应用于类的结构来自动生成它。此值受类的名称，它实现的接口及其大多数成员（包括编译器生成的合成成员）的影响。如果您更改任何这些内容，例如，通过添加便捷方法，生成的串行版本 UID 会更改。如果未能声明串行版本 UID，则兼容性将被破坏，从而导致运行时出现 InvalidClassException。

实现 Serializable 的第二个成本是它增加了错误和安全漏洞的可能性（第 85 项）。通常，使用构造函数创建对象;序列化是一种创建对象的语言机制。无论您是接受默认行为还是覆盖默认行为，反序列化都是一个“隐藏的构造函数”，其中包含与其他构造函数相同的所有问题。因为没有与反序列化相关联的显式构造函数，所以很容易忘记必须确保它保证构造函数建立的所有不变量，并且它不允许攻击者访问构造中的对象的内部。依赖于默认的反序列化机制，可以轻松地将对象置于不变的损坏和非法访问之外（第 88 项）。

实现 Serializable 的第三个成本是它增加了与发布新版本类相关的测试负担。修改可序列化类时，重要的是检查是否可以序列化新版本中的实例并在旧版本中反序列化，反之亦然。因此，所需的测试量与可序列化类的数量和可能很大的发布数量的乘积成比例。您必须确保序列化 - 反序列化过程成功并且它会导致原始对象的忠实副本。如果在首次编写类时仔细设计自定义序列化表单，则需要进行测试（第 87,90 项）。

实施 Serializable 不是一个轻率的决定。如果一个类要参与依赖于 Java 序列化进行对象传输或持久化的框架，那么这一点至关重要。此外，它极大地简化了将类用作另一个必须实现 Serializable 的类的组件。但是，实现 Serializable 会产生许多成本。每次设计课程时，都要权衡成本和收益。从历史上看，BigInteger 和 Instant 实现的 Serializable 等值类以及集合类也是如此。表示活动实体（如线程池）的类应该很少实现 Serializable。

为继承而设计的类（第 19 项）应该很少实现 Serializable，接口应该很少扩展它。违反此规则会给扩展类或实现接口的任何人带来沉重的负担。有时候违反规则是合适的。例如，如果一个类或接口主要存在于要求所有参与者实现 Serializable 的框架中，那么实现或扩展 Serializable 可能对类或接口有意义。

专为实现 Serializable 的继承而设计的类包括 Throwable 和 Component。Throwable 实现 Serializable，因此 RMI 可以从服务器向客户端发送异常。组件实现 Serializable，因此可以发送，保存和恢复 GUI，但即使在 Swing 和 AWT 的全盛时期，这个设施在实践中很少使用。

如果实现具有可序列化和可扩展的实例字段的类，则需要注意几个风险。如果实例字段值上存在任何不变量，则防止子类覆盖 finalize 方法至关重要，该类可以通过重写 finalize 并将其声明为 final 来完成。 否则，该类将容易受到终结者攻击（第 8 项）。 最后，如果类的实例字段初始化为其默认值（整数类型为零，布尔值为 false，对象引用类型为 null），则会违反不变量，必须添加此 readObjectNoData 方法：

```
// readObjectNoData for stateful extendable serializable classes

private void readObjectNoData() throws InvalidObjectException {

    throw new InvalidObjectException("Stream data required");

}
```

在 Java 4 中添加了此方法，以涵盖涉及向现有可序列化类[Serialization，3.5]添加可序列化超类的极端情况。

关于不实施 Serializable 的决定有一点需要注意。如果为继承而设计的类不可序列化，则可能需要额外的努力才能编写可序列化的子类。这种类的正常反序列化要求超类具有可访问的无参数构造函数[Serialization，1.10]。如果您不提供这样的构造函数，则强制子类使用序列化代理模式（Item 90）。

内部类（第 24 项）不应实现 Serializable。它们使用编译器生成的合成字段来存储对封闭实例的引用，并存储来自封闭范围的局部变量的值。这些字段如何对应于类定义是未指定的，匿名和本地类的名称也是如此。因此，内部类的默认序列化形式是未定义的。但是，静态成员类可以实现 Serializable。

总而言之，实现 Serializable 的难易程度。除非只在受保护的环境中使用类，其中版本永远不必进行互操作，并且服务器永远不会暴露给不受信任的数据，否则实现 Serializable 是一项严肃的承诺，应该非常谨慎。如果一个类允许继承，则需要格外小心。

87 谨慎实现 Serializable 接口

当您在时间压力下编写课程时，通常应该集中精力设计最佳 API。有时这意味着发布一个“一次性”实现，你知道你将在未来的版本中替换它。通常这不是

问题，但如果类实现了 Serializable 并使用了默认的序列化形式，那么您将无法完全逃离一次性实现。它将永远决定序列化的形式。这不仅仅是一个理论问题。它发生在 Java 库中的几个类中，包括 BigInteger。

如果没有考虑是否合适，请不要接受默认的序列化表格。接受默认的序列化形式应该有意识地决定从灵活性，性能和正确性的角度来看这种编码是合理的。一般来说，只有在与设计自定义序列化表单时所选择的编码大致相同的情况下，才应接受默认的序列化表单。

对象的默认序列化形式是以对象为根的对象图的物理表示的合理有效编码。换句话说，它描述了对象中以及可从此对象访问的每个对象中包含的数据。它还描述了所有这些对象相互链接的拓扑。对象的理想序列化形式仅包含对象表示的逻辑数据。它独立于物理表示。

如果对象的物理表示与其逻辑内容相同，则默认的序列化表单可能是合适的。例如，对于以下类，默认的序列化形式是合理的，这简单地表示一个人的姓名：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
    * Last name. Must be non-null.
    * @serial
    */
    private final String lastName;
    /**
    * First name. Must be non-null.
    * @serial
    */
    private final String firstName;
    /**
    * Middle name, or null if there is none.
```

```
        * @serial
        */
        private final String middleName;
        ... // Remainder omitted
    }
```

从逻辑上讲，名称由三个字符串组成，这三个字符串代表姓氏，名字和中间名。 Name 中的实例字段精确地镜像了这个逻辑内容。

即使您确定默认的序列化表单是合适的，您通常也必须提供 readObject 方法以确保不变量和安全性。对于 Name，readObject 方法必须确保字段 lastName 和 firstName 为非 null。第 88 和 90 项详细讨论了这个问题。

请注意，对 lastName，firstName 和 middleName 字段有文档注释，即使它们是私有的。这是因为这些私有字段定义了一个公共 API，它是该类的序列化形式，并且必须记录此公共 API。 @serial 标记的存在告诉 Javadoc 将此文档放在一个记录序列化表单的特殊页面上。

在 Name 的频谱的另一端附近，考虑下面的类，它代表一个字符串列表（忽略你可能最好使用一个标准的 List 实现）：

```
    // Awful candidate for default serialized form
    public final class StringList implements Serializable {
        private int size = 0;
        private Entry head = null;
        private static class Entry implements Serializable {
            String data;
            Entry next;
            Entry previous;
        }
        ... // Remainder omitted
    }
```

从逻辑上讲，这个类代表一系列字符串。在物理上，它将序列表示为双向链表。如果您接受默认的序列化表单，则序列化表单将精心镜像链接列表中的每个条目以及两个方向上条目之间的所有链接。

当对象的物理表示与其逻辑数据内容显着不同时，使用默认的序列化表单有四个缺点：

它将导出的 API 永久绑定到当前内部表示。在上面的示例中，私有 StringList.Entry 类成为公共 API 的一部分。如果在将来的版本中更改了表示，则 StringList 类仍需要接受输入上的链表表示并在输出时生成它。该类永远不会消除处理链表条目的所有代码，即使它不再使用它们。

它可以消耗过多的空间。在上面的示例中，序列化表单不必要地表示链接列表中的每个条目和所有链接。这些条目和链接仅仅是实现细节，不值得包含在序列化形式中。由于序列化表单过大，将其写入磁盘或通过网络发送将非常慢。

它可能会消耗过多的时间。序列化逻辑不了解对象图的拓扑结构，因此必须经历昂贵的图遍历。在上面的例子中，仅仅遵循下一个引用就足够了。

它可能导致堆栈溢出。默认的序列化过程执行对象图的递归遍历，即使对于中等大小的对象图，也可能导致堆栈溢出。使用 1,000-1,800 个元素序列化 StringList 实例会在我的机器上生成 StackOverflowError。令人惊讶的是，序列化导致堆栈溢出的最小列表大小因运行而异（在我的机器上）。显示此问题的最小列表大小可能取决于平台实现和命令行标志;某些实现可能根本没有这个问题。

StringList 的合理序列化形式只是列表中的字符串数，后跟字符串本身。这构成了由 StringList 表示的逻辑数据，剥离了其物理表示的细节。这是 StringList 的修订版本，其中包含实现此序列化形式的 writeObject 和 readObject 方法。提醒一下，transient 修饰符指示要从类的默认序列化形式中省略实例字段：

```
// StringList with a reasonable custom serialized form

public final class StringList implements Serializable {

    private transient int size = 0;

    private transient Entry head = null;

    // No longer Serializable!
```

```
        private static class Entry {
            String data;
            Entry next;
            Entry previous;
        }
        // Appends the specified string to the list
        public final void add(String s) { ... }


        /**
        * Serialize this {@code StringList} instance.
        **
        @serialData The size of the list (the number of strings
        * it contains) is emitted ({@code int}), followed by all of
        * its elements (each a {@code String}), in the proper
        * sequence.
        */
        private void writeObject(ObjectOutputStream s) throws IOException {
            s.defaultWriteObject();
            s.writeInt(size);
            // Write out all elements in the proper order.
            for (Entry e = head; e != null; e = e.next)
                s.writeObject(e.data);
        }
        private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {
            s.defaultReadObject();
```

```
            int numElements = s.readInt();

            // Read in all elements and insert them in list

            for (int i = 0; i < numElements; i++)

                add((String) s.readObject());

        }

        ... // Remainder omitted

    }
```

writeObject 做的第一件事就是调用 defaultWriteObject，而 readObject 做的第一件事就是调用 defaultReadObject，即使所有 StringList 的字段都是瞬态的。您可能会听到它说如果所有类的实例字段都是瞬态的，您可以省去调用 defaultWriteObject 和 defaultReadObject，但序列化规范要求您无论如何都要调用它们。这些调用的存在使得可以在以后的版本中添加非瞬态实例字段，同时保持向后和向前兼容性。如果实例在更高版本中序列化并在早期版本中反序列化，则添加的字段将被忽略。如果早期版本的 readObject 方法无法调用 defaultReadObject，则反序列化将因 StreamCorruptedException 而失败。

请注意，writeObject 方法有一个文档注释，即使它是私有的。这类似于 Name 类中私有字段的文档注释。此私有方法定义了一个公共 API，它是序列化形式，并且应该记录公共 API。与字段的@serial 标记一样，方法的@serialData 标记告诉 Javadoc 实用程序将此文档放在序列化表单页面上。

为了给早期的性能讨论带来一定的规模感，如果平均字符串长度为 10 个字符，则 StringList 的修订版本的序列化形式占用原始序列化形式的大约一半的空间。在我的机器上，序列化 StringList 的修订版本的速度是序列化原始版本的两倍，列表长度为 10。最后，修订后的表单中没有堆栈溢出问题，因此没有可串行化的 StringList 大小的实际上限。

虽然默认的序列化形式对于 StringList 来说是不好的，但是有些类会更糟糕。对于 StringList，默认的序列化形式是不灵活的并且执行得很糟糕，但是在序列化和反序列化 StringList 实例的意义上，它产生了原始对象的忠实副本，其所有不变量都是完整的。对于其不变量与特定于实现的详细信息相关联的任何对象，情况并非如此。

例如，考虑哈希表的情况。物理表示是包含键值条目的一系列散列桶。条目所在的桶是其密钥的哈希码的函数，通常，从实现到实现，它通常不保证是相同的。实际上，从运行到运行甚至都不能保证相同。因此，接受哈希表的默

认序列化表单将构成严重错误。序列化和反序列化哈希表可能会产生一个不变量严重损坏的对象。

无论您是否接受默认的序列化表单，当调用 defaultWriteObject 方法时，每个未标记为 transient 的实例字段都将被序列化。因此，每个可以声明为瞬态的实例字段都应该是。这包括派生字段，其值可以从主数据字段计算，例如缓存的哈希值。它还包括其值与 JVM 的一个特定运行相关联的字段，例如表示指向本机数据结构的指针的长字段。在决定使字段不变量之前，请说服自己它的值是对象逻辑状态的一部分。如果使用自定义序列化表单，则大多数或所有实例字段都应标记为瞬态，如上面的 StringList 示例中所示。

如果您使用的是默认序列化表单并且已将一个或多个字段标记为瞬态，请记住，在反序列化实例时，这些字段将初始化为其默认值：对象引用字段为 null，数字基本字段为零，false 为布尔字段[JLS，4.12.5]。如果这些值对于任何瞬态字段都是不可接受的，则必须提供一个 readObject 方法，该方法调用 defaultReadObject 方法，然后将瞬态字段恢复为可接受的值（第 88 项）。或者，这些字段可以在第一次使用时进行延迟初始化（第 83 项）。

无论是否使用默认的序列化表单，都必须在对象序列化上强制执行任何同步，这些同步将强加到读取对象的整个状态的任何其他方法上。 因此，例如，如果您有一个线程安全的对象（Item 82）通过同步每个方法来实现其线程安全，并且您选择使用默认的序列化表单，请使用以下 write-Object 方法：

```
// writeObject for synchronized class with default serialized form

private synchronized void writeObject(ObjectOutputStream s) throws
IOException {

    s.defaultWriteObject();

}
```

如果在 writeObject 方法中放置同步，则必须确保它遵循与其他活动相同的锁定排序约束，否则就会冒着资源排序死锁的风险[Goetz06,10.1.5]。

无论选择哪种序列化形式，在您编写的每个可序列化类中声明显式串行版本 UID。 这消除了串行版本 UID 作为不兼容的潜在来源（第 86 项）。 还有一个小的性能优势。 如果没有提供串行版本 UID，则执行昂贵的计算以在运行时生成一个。

声明串行版 UID 很简单。 只需将此行添加到您的班级：

```
private static final long serialVersionUID = randomLongValue;
```

如果您编写一个新类，则为 randomLongValue 选择的值无关紧要。您可以通过在类上运行 serialver 实用程序来生成值，但也可以凭空挑选一个数字。串行版本 UID 不是唯一的。如果修改缺少串行版本 UID 的现有类，并且希望新版本接受现有的序列化实例，则必须使用为旧版本自动生成的值。您可以通过在类的旧版本上运行 serialver 实用程序来获取此编号，该类型是存在序列化实例的类。

如果您想要创建与现有版本不兼容的类的新版本，只需更改串行版本 UID 声明中的值。这将导致尝试反序列化先前版本的序列化实例以抛出 InvalidClassException。除非您要破坏与类的所有现有序列化实例的兼容性，否则请勿更改串行版本 UID。

总而言之，如果您已确定某个类应该可序列化（第 86 项），请仔细考虑序列化表单应该是什么。仅当它是对象逻辑状态的合理描述时，才使用默认的序列化形式;否则设计一个适当描述对象的自定义序列化表单。在分配设计导出方法时，您应该分配尽可能多的时间来设计类的序列化形式（第 51 项）。正如您无法从将来的版本中删除导出的方法一样，您无法从序列化表单中删除字段;必须永久保存它们以确保序列化兼容性。选择错误的序列化表单会对类的复杂性和性能产生永久性的负面影响。

88 保护性编写 readObject 方法

Item 50 包含一个带有可变私有 Date 字段的不可变日期范围类。 该类通过在其构造函数和访问器中防御性地复制 Date 对象，竭尽全力保留其不变量和不变性。 这是班级：

```
// Immutable class that uses defensive copying

public final class Period {

    private final Date start;

    private final Date end;

    /**

    * @param start the beginning of the period

    * @param end the end of the period; must not precede start

    * @throws IllegalArgumentException if start is after end

    * @throws NullPointerException if start or end is null
```

```
        */
        public Period(Date start, Date end) {
            this.start = new Date(start.getTime());
            this.end = new Date(end.getTime());
            if (this.start.compareTo(this.end) > 0)
                throw new IllegalArgumentException(start + " after " + end);
        }

        public Date start () { return new Date(start.getTime()); }

        public Date end () { return new Date(end.getTime()); }

        public String toString() { return start + " - " + end; }
        ... // Remainder omitted
    }
```

假设您决定要将此类序列化。由于 Period 对象的物理表示形式恰好反映了其逻辑数据内容，因此使用默认的序列化表单并不合理（第 87 项）。因此，似乎要使类可序列化所需要做的就是将实现 Serializable 的单词添加到类声明中。但是，如果你这样做了，那么这个类将不再保证它的关键不变量。

问题是 readObject 方法实际上是另一个公共构造函数，它需要与任何其他构造函数一样的小心。正如构造函数必须检查其参数的有效性（第 49 项）并在适当的地方制作参数的防御性副本（第 50 项），因此必须使用 readObject 方法。如果 readObject 方法无法执行这些操作中的任何一个，则攻击者违反类的不变量是相对简单的事情。

简而言之，readObject 是一个构造函数，它将字节流作为唯一参数。在正常使用中，字节流是通过序列化正常构造的实例生成的。当 readObject 被呈现为字节流时，问题出现了，该字节流被人工构造以生成违反其类的不变量的对象。这样的字节流可用于创建一个不可能的对象，该对象无法使用普通构造函数创建。

假设我们只是将工具 Serializable 添加到 Period 的类声明中。然后，这个丑陋的程序将生成一个 Period 实例，其结束在其开始之前。设置高顺序位的字节值的强制转换是 Java 缺少字节文字的结果，并且不幸的决定使字节类型符号化：

```
public class BogusPeriod {
// Byte stream couldn't have come from a real Period instance!
    private static final byte[] serializedForm = {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
        0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
        0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
        (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
        0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
        0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
        0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
        0x00, 0x78
    };

    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
        System.out.println(p);
    }
```

```
        // Returns the object with the specified serialized form
        static Object deserialize(byte[] sf) {
            try {
                return new ObjectInputStream(new
ByteArrayInputStream(sf)).readObject();
            } catch (IOException | ClassNotFoundException e) {
                throw new IllegalArgumentException(e);
            }
        }
    }
```

用于初始化 serializedForm 的字节数组文字是通过序列化正常的 Period 实例并手动编辑生成的字节流生成的。 流的细节对于该示例并不重要，但是如果您好奇，则在 Java 对象序列化规范[序列化，6]中描述了序列化字节流格式。 如果您运行此程序，它将打印 Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984.只需声明 Period serializable，我们就可以创建一个违反其类不变量的对象。

要解决此问题，请为 Period 调用 defaultReadObject 提供 readObject 方法，然后检查反序列化对象的有效性。 如果有效性检查失败，则 readObject 方法将抛出 InvalidObjectException，从而阻止反序列化完成：

```
    // readObject method with validity checking - insufficient!
    private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {
        s.defaultReadObject();
        // Check that our invariants are satisfied
        if (start.compareTo(end) > 0)
            throw new InvalidObjectException(start +" after "+ end);
    }
```

虽然这可以防止攻击者创建无效的 Period 实例，但仍然存在潜在的更微妙的问题。 可以通过构造以有效 Period 实例开头的字节流来创建可变 Period 周期

实例，然后将额外引用附加到 Period 实例内部的私有 Date 字段。 攻击者从 ObjectInputStream 中读取 Period 实例，然后读取附加到流的“恶意对象引用”。这些引用使攻击者可以访问 Period 对象中私有 Date 字段引用的对象。 通过改变这些 Date 实例，攻击者可以改变 Period 实例。 以下类演示了此攻击：

```
public class MutablePeriod {
    // A period instance
    public final Period period;
    // period's start field, to which we shouldn't have access
    public final Date start;
    // period's end field, to which we shouldn't have access
    public final Date end;
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(bos);
            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));
            /*
            * Append rogue "previous object refs" for internal
            * Date fields in Period. For details, see "Java
            * Object Serialization Specification," Section 6.4.
            */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4; // Ref # 4
            bos.write(ref); // The end field
```

```
                        // Deserialize Period and "stolen" Date references
                        ObjectInputStream in = new ObjectInputStream(new
ByteArrayInputStream(bos.toByteArray()));
                        period = (Period) in.readObject();
                        start = (Date) in.readObject();
                        end = (Date) in.readObject();
                } catch (IOException | ClassNotFoundException e) {
                        throw new AssertionError(e);
                }
        }
    }
```

要查看正在进行的攻击，请运行以下程序：

```
    public static void main(String[] args) {
        MutablePeriod mp = new MutablePeriod();
        Period p = mp.period;
        Date pEnd = mp.end;
        // Let's turn back the clock
        pEnd.setYear(78);
        System.out.println(p);
        // Bring back the 60s!
        pEnd.setYear(69);
        System.out.println(p);
    }
```

在我的语言环境中，运行此程序会产生以下输出：

```
    Wed Nov 22 00:21:29 PST 2017 - Wed Nov 22 00:21:29 PST 1978
```

```
Wed Nov 22 00:21:29 PST 2017 - Sat Nov 22 00:21:29 PST 1969
```

虽然创建了 Period 实例且其不变量保持不变，但可以随意修改其内部组件。一旦拥有可变的 Period 实例，攻击者可能会通过将实例传递给依赖于 Period 的安全性不变性的类来造成巨大的伤害。 这不是那么牵强：有些类依赖于 String 的安全性不变性。

问题的根源是 Period 的 readObject 方法没有做足够的防御性复制。 对象反序列化时，防御性地复制包含客户端不得拥有的对象引用的任何字段至关重要。因此，每个包含私有可变组件的可序列化不可变类必须在其 readObject 方法中防御性地复制这些组件。 以下 readObject 方法足以确保 Period 的不变量并保持其不变性：

```
// readObject method with defensive copying and validity checking

private void readObject(ObjectInputStream s) throws IOException,
ClassNotFoundException {

    s.defaultReadObject();

    // Defensively copy our mutable components

    start = new Date(start.getTime());

    end = new Date(end.getTime());

    // Check that our invariants are satisfied

    if (start.compareTo(end) > 0)

        throw new InvalidObjectException(start +" after "+ end);

}
```

请注意，防御性副本在有效性检查之前执行，并且我们没有使用 Date 的克隆方法来执行防御性副本。需要这两个细节来保护 Period 免受攻击（第 50 项）。另请注意，最终字段无法进行防御性复制。 要使用 readObject 方法，我们必须使 start 和 end 字段为非最终字段。这是不幸的，但它是两个邪恶中较小的一个。使用新的 readObject 方法并从开始和结束字段中删除最终修饰符后，MutablePeriod 类将呈现无效。 上面的攻击程序现在生成此输出：

```
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017

Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
```

这是一个简单的试金石，用于判断默认的 readObject 方法是否适用于某个类：您是否愿意添加一个公共构造函数，该构造函数将对象中每个非瞬态字段的值作为参数，并将值存储在字段中而不进行验证任何？如果没有，则必须提供 readObject 方法，并且必须执行构造函数所需的所有有效性检查和防御性复制。或者，您可以使用序列化代理模式（项目 90）。强烈建议使用此模式，因为它需要花费大量精力进行安全反序列化。

readObject 方法和构造函数之间还有一个相似之处，它们适用于非最终可序列化类。与构造函数一样，readObject 方法不能直接或间接调用可覆盖的方法（第 19 项）。如果违反此规则并且重写了相关方法，则重写方法将在子类的状态被反序列化之前运行。程序失败可能会导致[Bloch05，Puzzle 91]。

总而言之，无论何时编写 readObject 方法，都要采用您正在编写公共构造函数的思维模式，该构造函数必须生成有效的实例，而不管它给出了什么字节流。不要假设字节流表示实际的序列化实例。虽然此项中的示例涉及使用默认序列化表单的类，但所有引发的问题同样适用于具有自定义序列化表单的类。这里，以摘要形式，是编写 readObject 方法的指南：

- 对于具有必须保持私有的对象引用字段的类，在这样的字段中防御性地复制每个对象。 不可变类的可变组件属于此类。
- 检查任何不变量，如果检查失败则抛出 InvalidObjectException。 检查应遵循任何防御性复制。
- 如果在反序列化后必须验证整个对象图，请使用 ObjectInputValidation 接口（本书未讨论）。
- 不要直接或间接地在课堂上调用任何可覆盖的方法。

89 对于实例控制，枚举优于 readResolve

Item 3 describes the Singleton pattern and gives the following example of a singleton class. This class restricts access to its constructor to ensure that only a single instance is ever created:

第 3 项描述了 Singleton 模式，并给出了单例类的以下示例。 此类限制对其构造函数的访问，以确保只创建一个实例：

```
public class Elvis {
```

```
        public static final Elvis INSTANCE = new Elvis();

        private Elvis() { ... }

        public void leaveTheBuilding() { ... }
    }
```

如第 3 项所述，如果将实现 Serializable 的单词添加到其声明中，则此类将不再是单例。类是否使用默认的序列化表单或自定义序列化表单（第 87 项）并不重要，该类是否提供显式的 readObject 方法（第 88 项）也无关紧要。任何 readObject 方法，无论是显式方法还是默认方法，都会返回一个新创建的实例，该实例与在类初始化时创建的实例不同。

readResolve 功能允许您将另一个实例替换为 readObject [Serialization，3.7] 创建的实例。如果被反序列化的对象的类使用适当的声明定义了 readResolve 方法，则在反序列化后对新创建的对象调用此方法。然后返回此方法返回的对象引用来代替新创建的对象。在此功能的大多数用途中，不保留对新创建的对象的引用，因此它立即有资格进行垃圾回收。

如果 Elvis 类用于实现 Serializable，则以下 read-Resolve 方法足以保证 singleton 属性：

```
    // readResolve for instance control - you can do better!
    private Object readResolve() {
        // Return the one true Elvis and let the garbage collector
        // take care of the Elvis impersonator.
        return INSTANCE;
    }
```

此方法忽略反序列化对象，返回在初始化类时创建的区分 Elvis 实例。因此，Elvis 实例的序列化形式不需要包含任何实际数据;应将所有实例字段声明为瞬态。实际上，如果您依赖 readResolve 进行实例控制，则必须将具有对象引用类型的所有实例字段声明为 transient。否则，确定的攻击者有可能在运行 readResolve 方法之前使用类似于第 88 项中的 MutablePeriod 攻击的技术来保护对反序列化对象的引用。

攻击有点复杂，但潜在的想法很简单。如果单例包含非瞬态对象引用字段，则在运行单例的 readResolve 方法之前，将对该字段的内容进行反序列化。这允

许精心设计的流在反序列化对象引用字段的内容时“窃取”对原始反序列化单例的引用。

以下是它如何更详细地工作。首先，编写一个“stealer”类，它同时具有readResolve 方法和一个实例字段，该字段引用窃取程序“隐藏”的序列化单例。在序列化流中，将单例的非瞬态字段替换为窃取程序的实例。你现在有一个圆形：单身包含窃取者，偷窃者指的是单身人士。

因为单例包含窃取程序，所以当单例被反序列化时，窃取程序的 readResolve 方法首先运行。因此，当窃取程序的 readResolve 方法运行时，其实例字段仍然引用部分反序列化（并且尚未解析）的单例。

窃取程序的 readResolve 方法将引用从其实例字段复制到静态字段，以便在readResolve 方法运行后访问引用。然后，该方法返回其隐藏的字段的正确类型的值。如果它没有这样做，当序列化系统试图将窃取者引用存储到该字段时，VM 将抛出 ClassCastException。

为了使这个具体，请考虑以下破碎的单身人士：

```
// Broken singleton - has nontransient object reference field!
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }
    private String[] favoriteSongs ={ "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
    private Object readResolve() {
    return INSTANCE;
    }
}
```

这是一个“窃取者”类，按照上面的描述构建：

```
public class ElvisStealer implements Serializable {
```

```
        static Elvis impersonator;
        private Elvis payload;

        private Object readResolve() {
            // Save a reference to the "unresolved" Elvis instance
            impersonator = payload;
            // Return object of correct type for favoriteSongs field
            return new String[] { "A Fool Such as I" };
        }
        private static final long serialVersionUID =0;
    }
```

最后，这里是一个简单的程序，它反序列化一个手工制作的流，以生成有缺陷的单例的两个不同实例。此程序中省略了反序列化方法，因为它与第 354 页上的方法相同：public class ElvisImpersonator {

```
    // Byte stream couldn't have come from a real Elvis instance!
        private static final byte[] serializedForm = {
            (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
            0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
            (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
            0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
            0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
            0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
            0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
            0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
            0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
```

```
            0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
            0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
            0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
        };
        public static void main(String[] args) {
            // Initializes ElvisStealer.impersonator and returns
            // the real Elvis (which is Elvis.INSTANCE)
            Elvis elvis = (Elvis) deserialize(serializedForm);
            Elvis impersonator = ElvisStealer.impersonator;
            elvis.printFavorites();
            impersonator.printFavorites();
        }
    }
```

运行此程序会产生以下输出，最终证明可以创建两个不同的 Elvis 实例（音乐中具有不同的品味）：

```
    [Hound Dog, Heartbreak Hotel]
    [A Fool Such as I]
```

您可以通过声明 favoriteSongs 字段瞬态来解决问题，但最好通过使 Elvis 成为单元素枚举类型来修复它（第 3 项）。 正如 ElvisStealer 攻击所证明的那样，使用 readResolve 方法来防止攻击者访问“临时”反序列化实例是非常脆弱的，需要非常小心。

如果将可序列化的实例控制类编写为枚举，Java 会保证除了声明的常量之外不能有任何实例，除非攻击者滥用 AccessibleObject.setAccessible 等特权方法。任何能够做到这一点的攻击者已经拥有足够的权限来执行任意本机代码，并且所有的赌注都已关闭。 以下是我们的 Elvis 示例如何看作枚举：

```
    // Enum singleton - the preferred approach
    public enum Elvis {
```

```
        INSTANCE;
        private String[] favoriteSongs ={ "Hound Dog", "Heartbreak Hotel" };
        public void printFavorites() {
            System.out.println(Arrays.toString(favoriteSongs));
        }
    }
```

使用 readResolve 进行实例控制并不是过时的。如果必须编写一个可序列化的实例控制类，其实例在编译时是未知的，那么您将无法将该类表示为枚举类型。

readResolve 的可访问性非常重要。如果在最终类上放置 readResolve 方法，它应该是私有的。如果将 readResolve 方法放在非最终类上，则必须仔细考虑其可访问性。如果它是私有的，则不适用于任何子类。如果是 packageprivate，它将仅适用于同一包中的子类。如果它是受保护的或公共的，它将适用于所有不覆盖它的子类。如果 readResolve 方法受保护或公共，并且子类不覆盖它，则反序列化子类实例将生成一个超类实例，这可能会导致 ClassCastException。

总而言之，使用枚举类型尽可能强制实例控制不变量。如果这是不可能的，并且您需要一个类可序列化和实例控制，则必须提供 readResolve 方法并确保所有类的实例字段都是原始的或瞬态的。

90 考虑序列化代理替换序列化实例

如第 85 和第 86 项所述并在本章中讨论过，实现 Serializable 的决定增加了错误和安全问题的可能性，因为它允许使用 extralinguistic 机制代替普通构造函数创建实例。然而，有一种技术可以大大降低这些风险。此技术称为序列化代理模式。

序列化代理模式相当简单。首先，设计一个私有静态嵌套类，它简洁地表示封闭类的实例的逻辑状态。这个嵌套类称为封闭类的序列化代理。它应该有一个构造函数，其参数类型是封闭类。此构造函数仅复制其参数中的数据：它不需要执行任何一致性检查或防御性复制。根据设计，序列化代理的默认序列化形式是封闭类的完美序列化形式。必须声明封闭类及其序列化代理以实现 Serializable。

例如，考虑在 Item 50 中编写的不可变 Period 类，并在 Item 88 中进行序列化。这是该类的序列化代理。 Period 非常简单，其序列化代理与该字段具有完全相同的字段：

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;
    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }
    private static final long serialVersionUID =234098243823485285L; // Any number will do (Item 87)
}
```

接下来，将以下 writeReplace 方法添加到封闭类中。 可以将此方法逐字复制到具有序列化代理的任何类中：

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

封闭类上存在此方法会导致序列化系统发出 SerializationProxy 实例而不是封闭类的实例。 换句话说，writeReplace 方法在序列化之前将封闭类的实例转换为其序列化代理。

使用此 writeReplace 方法，序列化系统将永远不会生成封闭类的序列化实例，但攻击者可能会构造一个试图违反类不变量的实例。 要确保此类攻击失败，只需将此 readObject 方法添加到封闭类：

```
// readObject method for the serialization proxy pattern
```

```
    private void readObject(ObjectInputStream stream) throws
InvalidObjectException {

        throw new InvalidObjectException("Proxy required");

    }
```

最后，在 SerializationProxy 类上提供 readResolve 方法，该方法返回封闭类的逻辑等效实例。此方法的存在导致序列化系统在反序列化时将序列化代理转换回封闭类的实例。

这个 readResolve 方法只使用它的公共 API 创建一个封闭类的实例，其中就是模式的美妙之处。它在很大程度上消除了序列化的语言特征，因为反序列化的实例是使用与任何其他实例相同的构造函数，静态工厂和方法创建的。这使您不必单独确保反序列化的实例服从类的不变量。如果类的静态工厂或构造函数建立这些不变量并且其实例方法维护它们，那么您已确保不变量也将通过序列化来维护。

以下是 Period.SerializationProxy 的 readResolve 方法：

```
    // readResolve method for Period.SerializationProxy

    private Object readResolve() {

        return new Period(start, end); // Uses public constructor

    }
```

与防御性复制方法（第 357 页）一样，序列化代理方法可以阻止伪造的字节流攻击（第 354 页）和内部字段盗窃攻击（第 356 页）。与前两种方法不同，这一方法允许 Period 的字段为 final，这是 Period 类真正不可变所必需的（第 17 项）。与之前的两种方法不同，这一方法不涉及很多想法。您不必弄清楚哪些字段可能会被狡猾的序列化攻击所破坏，也不必在反序列化过程中明确执行有效性检查。

还有另一种方法，序列化代理模式比 readObject 中的防御性复制更强大。序列化代理模式允许反序列化实例具有与最初序列化实例不同的类。您可能不认为这在实践中有用，但确实如此。

考虑 EnumSet 的情况（第 36 项）。这个类没有公共构造函数，只有静态工厂。从客户端的角度来看，它们返回 EnumSet 实例，但在当前的 OpenJDK 实现中，它们返回两个子类中的一个，具体取决于底层枚举类型的大小。如果底层

枚举类型包含 64 个或更少的元素，则静态工厂返回 RegularEnumSet;否则，他们返回一个 JumboEnumSet。

现在考虑如果序列化其枚举类型具有六十个元素的枚举集，然后再向枚举类型添加五个元素，然后反序列化枚举集，会发生什么。它是序列化时的 RegularEnumSet 实例，但最好是反序列化后的 JumboEnumSet 实例。实际上，这正是发生的事情，因为 EnumSet 使用序列化代理模式。如果你很好奇，这里是 EnumSet 的序列化代理。这真的很简单：

```
    // EnumSet's serialization proxy
    private static class SerializationProxy <E extends Enum<E>> implements
Serializable {
        // The element type of this enum set.
        private final Class<E> elementType;
        // The elements contained in this enum set.
        private final Enum<?>[] elements;
        SerializationProxy(EnumSet<E> set) {
            elementType = set.elementType;
            elements = set.toArray(new Enum<?>[0]);
        }
        private Object readResolve() {
            EnumSet<E> result = EnumSet.noneOf(elementType);
            for (Enum<?> e : elements)
                result.add((E)e);
            return result;
        }
        private static final long serialVersionUID =362491234563181265L;
    }
```

序列化代理模式有两个限制。 它与用户可扩展的类不兼容（第 19 项）。 此外，它与某些对象图包含圆形的类不兼容：如果您尝试从其序列化代理的 readResolve 方法中调用此类对象上的方法，您将获得 ClassCastException，因为您还没有该对象， 只有它的序列化代理。

最后，序列化代理模式的附加功能和安全性不是免费的。 在我的机器上，使用序列化代理序列化和反序列化 Period 实例比使用防御性复制更加昂贵 14％。

总之，只要您发现自己必须在不能由其客户端扩展的类上编写 readObject 或 writeObject 方法，请考虑序列化代理模式。 这种模式可能是使用非平凡不变量强健序列化对象的最简单方法。

完结 2019-02-20