Effective-Java-3rd-edition-Chinese-English-bilingual

Effective Java（第 3 版）各章节的中英文学习参考，希望对 Java 技术的提高有所帮助，欢迎通过 issue 或 pr 提出建议和修改意见。

目录（Contents）

Chapter 2. Creating and Destroying Objects（创建和销毁对象）

Chapter 2 Introduction（章节介绍）

This chapter concerns（关注、涉及） creating and destroying objects: when and how to create them, when and how to avoid creating them, how to ensure they are destroyed in a timely manner, and how to manage any cleanup actions that must precede their destruction.

本章涉及创建和销毁对象：何时以及如何创建对象，何时以及如何避免创建对象，如何确保它们被及时销毁，以及如何管理在销毁之前必须执行的清理操作。

Item 1: Consider static factory methods instead of constructors（考虑以静态工厂方法代替构造函数）

客户端获得实例的传统方式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技能树的一部分。一个类可以提供公共静态工厂方法，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 boolean 基本类型转换为 Boolean 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]。本条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。使用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多个构造函数来绕过这个限制，这些构造函数的参数列表仅在参数类型、个数或顺序上有所不同。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不适合的构造函数。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。

译注：`two` 不应是确数，应理解为概数，为绕过这个限制提供的构造可以不止两个；后半句 `...whose parameter lists differ only in the order of their parameter types.` 做了意译。

因为静态工厂方法有确切名称，所以它们没有前一段讨论的局限。如果一个类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，这样的能力允许类在任何时候都能严格控制存在的实例。这样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` 为 true。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形成一个非常紧凑的 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是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，这通常是很好的做法（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。许多公共静态成员应该放在接口本身中，而不是放在类中。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，通常也不应该自己写（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。其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，理解它们相比而言的优点是值得的。通常静态工厂的方式更可取，因此应避免在没有考虑静态工厂的情况下就提供公共构造函数。

Item 2: Consider a builder when faced with many constructor parameters（当构造函数有多个参数时，考虑改用构建器）

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：

若将该案例「构建机制」独立出来，或能广泛适应相似结构的构建需求，详见文末随笔

```java
// 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，这样就可以链式调用，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。前者有一个必需的尺寸大小参数，而后者让你指定酱料应该放在里面还是外面：

```
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，构建器的一个小优点是构建器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。一个构建器可以反复构建多个对象。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。

随笔

每个内部 Builder 类要对每个字段建立相应方法，代码比较冗长。若将「构建机制」独立出来，或能广泛适应相似结构的构建需求。以下是针对原文案例的简要修改，仅供参考：

```
class EntityCreator<T> {

    private Class<T> classInstance;
    private T entityObj;

    public EntityCreator(Class<T> classInstance, Object... initParams) throws
Exception {
        this.classInstance = classInstance;
        Class<?>[] paramTypes = new Class<?>[initParams.length];
        for (int index = 0, length = initParams.length; index < length; index++)
{
            String checkStr = initParams[index].getClass().getSimpleName();
            if (checkStr.contains("Integer")) {
                paramTypes[index] = int.class;
            }
            if (checkStr.contains("Double")) {
                paramTypes[index] = double.class;
            }
            if (checkStr.contains("Boolean")) {
                paramTypes[index] = boolean.class;
            }
            if (checkStr.contains("String")) {
                paramTypes[index] = initParams[index].getClass();
            }
        }
        Constructor<T> constructor =
classInstance.getDeclaredConstructor(paramTypes);
        constructor.setAccessible(true);
        this.entityObj = constructor.newInstance(initParams);
    }
```

```
    public EntityCreator<T> setValue(String paramName, Object paramValue) throws
Exception {
        Field field = classInstance.getDeclaredField(paramName);
        field.setAccessible(true);
        field.set(entityObj, paramValue);
        return this;
    }

    public T build() {
        return entityObj;
    }
}
```

如此，可移除整个内部 Builder 类，NutritionFacts 类私有构造的参数仅包括两个必填的 servingSize、servings 字段：

```
public class NutritionFacts {
    // 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;

    private NutritionFacts(int servingSize, int servings) {
        this.servingSize = servingSize;
        this.servings = servings;
    }
}
```

该案例的客户端代码改为：

```
NutritionFacts cocaCola = new EntityCreator<>(NutritionFacts.class, 240, 8)
        .setValue("calories", 100)
        .setValue("sodium", 35)
        .setValue("carbohydrate", 27).build();
```

Item 3: Enforce the singleton property with a private constructor or an enum type（使用私有构造函数或枚举类型实施单例属性）

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实现可序列化（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。

Item 4: Enforce noninstantiability with a private constructor（用私有构造函数实施不可实例化）

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。

Item 5: Prefer dependency injection to hardwiring resources（依赖注入优于硬连接资源）

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) { ... }
}
```

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、可复用性和可测试性。

Item 6: Avoid creating unnecessary objects（避免创建不必要的对象）

复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。复用可以更快、更流行。如果对象是不可变的，那么它总是可以被复用的（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!
```

The statement creates a new String instance each time it is executed, and none of those object creations is necessary. The argument to the String constructor ("bikini") is itself a String instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly.

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，创建大量 String 实例是不必要的。

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复用该对象，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。除了复用不可变对象之外，如果知道可变对象不会被修改，也可以复用它们。

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复用。不幸的是，当你创建这样一个对象时，这一点并不总是很明显。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。** 问题在于，它在内部为正则表达式创建了一个 Pattern 实例，并且只使用一次，之后就进行垃圾收集了。创建一个 Pattern 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 Pattern 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 方法时复用同一个实例：

```
// 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，初始版本输入 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 Pattern 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，它可以安全地复用，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，它允许程序员混合基本类型和包装类型，根据需要自动装箱和拆箱。**自动装箱模糊了基本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：**基本类型优于包装类，还应提防意外的自动装箱。**

本条目不应该被曲解为是在暗示创建对象是成本昂贵的，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，因此复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：「在应该复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复用现有对象」。请注意，当需要进行防御性复制时，复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；而不必要地创建对象只会影响样式和性能。

Item 7: Eliminate obsolete object references（排除过时的对象引用）

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：

```java
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，一个类管理它自己的内存时，程序员应该警惕内存泄漏。当释放一个元素时，该元素中包含的任何对象引用都应该被置为 null。

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。

Item 8: Avoid finalizers and cleaners（避免使用终结器和清除器）

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。

Item 9: Prefer try-with-resources to try-finally（使用 try-with-resources 优于 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。

Chapter 3. Methods Common to All Objects（对象的通用方法）

Chapter 3 Introduction（章节介绍）

ALTHOUGH Object is a concrete class, it is designed primarily for extension. All of its nonfinal methods (equals, hashCode, toString, clone, and finalize) have explicit general contracts because they are designed to be overridden. It is the responsibility of any class overriding these methods to obey their general contracts; failure to do so will prevent other classes that depend on the contracts (such as HashMap and HashSet) from functioning properly in conjunction with the class.

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。

This chapter tells you when and how to override the nonfinal Object methods. The finalize method is omitted from this chapter because it was discussed in Item 8. While not an Object method,Comparable.compareTo is discussed in this chapter because it has a similar character.

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。

Item 10: Obey the general contract when overriding equals（覆盖 equals 方法时应遵守的约定）

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：

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。

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。

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。

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。

Item 11: Always override hashCode when you override equals（当覆盖 equals 方法时，总要覆盖 hashCode 方法）

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

- 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 方法中用于比较的信息进行修改。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 方法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 方法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散列表的性能。

当你无法覆盖 hashCode 方法时，将违反第二个关键条款：相等的对象必须具有相等的散列码。 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，另一个相等的实例（被试图）用于检索。由于 PhoneNumber 类未能覆盖 hashCode 方法，导致两个相等的实例具有不相等的散列码，这违反了 hashCode 方法约定。因此，get 方法查找电话号码的散列桶可能会与 put 方法存储电话号码的散列桶不同。即使这两个实例碰巧分配在同一个散列桶上，get 方法几乎肯定会返回 null，因为 HashMap 有一个优化，它缓存每个条目相关联的散列码，如果散列码不匹配，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？写一个反面例子很容易。例如，以下方法是合法的，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散列码。同时它也很糟糕，因为它使每个对象都有相同的散列码。因此，每个对象都分配到同一个桶中，散列表退化为链表。这样，原本应该在线性阶 `O(n)` 运行的程序将在平方阶 `O(n^2)` 运行。对于大型散列表，这是工作和不工作的区别。

一个好的散列算法倾向于为不相等的实例生成不相等的散列码。这正是 hashCode 方法约定第三部分的含义。理想情况下，一个散列算法应该在所有 int 值上均匀合理分布所有不相等实例集合。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散列码 c，如步骤 2.a 中计算的那样。（回想一下 Item-10 中的重要字段会对比较产生影响）

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散列码 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 方法来比较字段，则递归调用字段上的 hashCode 方法。如果需要更复杂的比较，则为该字段计算一个「canonical representation」，并在 canonical representation 上调用 hashCode 方法。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，则将其每个重要元素都视为一个单独的字段。也就是说，通过递归地应用这些规则计算每个重要元素的散列码，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 中计算的散列码 c 合并到 result 变量，如下所示：

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

3、返回 result 变量。

当你完成了 hashCode 方法的编写之后，问问自己现在相同的实例是否具有相同的散列码。编写单元测试来验证你的直觉（除非你使用 AutoValue 生成你的 equals 方法和 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散列码，找出原因并修复问题。

可以从散列码计算中排除派生字段。换句话说，你可以忽略任何可以从包含的字段计算其值的字段。你必须排除不用 `equals` 比较的任何字段，否则你可能会违反 hashCode 方法约定的第二个条款。

在步骤 2.b 中使用的乘法将使结果取决于字段的顺序，如果类有多个相似的字段，则会产生一个更好的散列算法。例如，如果字符串散列算法中省略了乘法，那么所有的字母顺序都有相同的散列码。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，因为乘法运算等同于移位。使用素数的好处不太明显，但它是传统用法。31 有一个很好的特性，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的计算结果，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 实例具有相等的散列码。实际上，这个方法是 PhoneNumber 的一个非常好的 hashCode 方法实现，与 Java 库中的 hashCode 方法实现相当。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散列桶中。

虽然本条目中的方法产生了相当不错的散列算法，但它们并不是最先进的。它们的质量可与 Java 库的值类型中的散列算法相媲美，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散列算法，请参阅 Guava 的 com.google.common.hash.Hashing [Guava]。

Objects 类有一个静态方法，它接受任意数量的对象并返回它们的散列码。这个名为 `hash` 的方法允许你编写只有一行代码的 hashCode 方法，这些方法的质量可以与本条目中提供的编写方法媲美。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散列算法。下面是使用这种技术编写的 PhoneNumber 的散列算法：

```
// 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，并且计算散列码的成本非常高，那么你可以考虑在对象中缓存散列码，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散列键，那么你应该在创建实例时计算散列码。否则，你可以选择在第一次调用 hashCode 方法时延迟初始化散列码。在一个延迟初始化的字段（Item-83）的情况下，需要注意以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，如下所示。注意，散列字段的初始值（在本例中为 0）不应该是通常创建的实例的散列码：

```
// 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;
    }
```

不要试图从散列码计算中排除重要字段，以提高性能。 虽然得到的散列算法可能运行得更快，但其糟糕的质量可能会将散列表的性能降低到无法使用的程度。特别是，散列算法可能会遇到大量实例，这些实例在你选择忽略的不同区域。如果发生这种情况，散列算法将把所有这些实例映射很少一部分散列码，使得原本应该在线性阶 `O(n)` 运行的程序将在平方阶 `O(n^2)` 运行。

这不仅仅是一个理论问题。在 Java 2 之前，字符串散列算法在字符串中，以第一个字符开始，最多使用 16 个字符。对于大量且分层次的集合（如 url），该函数完全展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，这样客户端就不能理所应当的依赖它。这（也）给了你更改它的余地。 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提高散列算法的能力。如果你保留了未指定的细节，并且在散列算法中发现了缺陷，或者发现了更好的散列算法，那么你可以在后续版本中更改它。

总之，每次覆盖 equals 方法时都必须覆盖 hashCode 方法，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散列码分配给不相等的实例。这很容易实现，如果有点枯燥，可使用第 51 页的方法。如 Item-10 所述，AutoValue 框架提供了一种能很好的替代手动编写 equals 方法和 hashCode 方法的功能，IDE 也提供了这种功能。

Item 12: Always override toString（始终覆盖 toString 方法）

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。

Item 13: Override clone judiciously（明智地覆盖 clone 方法）

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 接口就能调用 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。尽管存在多种缺陷，但该机制的使用范围相当广泛，因此理解它是值得的。本条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，并提供替代方案。

译注：mixin 接口很可能是指一种带有全部实现或者部分实现的接口，其主要作用是：（1）更好的进行代码复用；（2）间接实现多重继承；（3）扩展功能。与传统接口相比，传统接口中不带实现，而 mixin 接口带有实现。

既然 Cloneable 接口不包含任何方法，用它来做什么呢？它决定了 Object 类受保护的 clone 实现的行为：如果一个类实现了 Cloneable 接口，Object 类的 clone 方法则返回该类实例的逐字段拷贝；否则它会抛出 CloneNotSupportedException。这是接口非常不典型的一种使用方式，不应该效仿。通常，类实现接口可以表明类能够为其客户端做些什么。在本例中，它修改了超类上受保护的方法的行为。

虽然规范没有说明，但是在实践中，实现 Cloneable 接口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。下面的内容是从 Object 规范复制过来的：

Creates and returns a copy of this object. The precise meaning of “copy” may depend on the class of the object. The general intent is that, for any object x,the expression

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

will be true, and the expression

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

will be true, but these are not absolute requirements. While it is typically the case that

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

will be true, this is not an absolute requirement.

---

clone 方法创建并返回对象的副本。「副本」的确切含义可能取决于对象的类别。通常，对于任何对象 x，表达式 `x.clone() != x`、`x.clone().getClass() == x.getClass()` 以及 `x.clone().equals(x)` 的值都将为 true，但都不是绝对的。

译注：以上情况的 equals 方法应覆盖默认实现，改为比较对象中的字段才能得到 true。默认实现是比较两个引用类型的内存地址，结果必然为 false

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

按照约定，clone 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（Object 类除外）都遵守这个约定，在这种情况下，表达式 `x.clone().getClass() == x.getClass()` 则为 true

按照约定，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，可能需要在 super.clone() 返回前，修改对象的一个或多个字段。

这种机制有点类似于构造方法链，只是没有强制执行：

- （1）如果一个类的 clone 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，编译器不会报错，但是如果这个类的一个子类调用 super.clone()，由此产生的对象类型将是错误的，影响子类 clone 方法正常工作。
- （2）如果覆盖 clone 方法的类是 final 修饰的，那么可以安全地忽略这个约定，因为没有子类需要担心。
- （3）如果一个 final 修饰的类不调用 super.clone() 的 clone 方法。类没有理由实现 Cloneable 接口，因为它不依赖于 Object 类的 clone 实现的行为。

译注：本段描述（1）的例子如下，表达式 `x.clone().getClass() == x.getClass()` 值为 false

```
class Base {
    @Override protected Object clone() throws CloneNotSupportedException {
        return new Base(); // ①
    }
}

class BasePro extends Base implements Cloneable {
    @Override protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws Exception {
        BasePro basePro = new BasePro();
        System.out.println(basePro.clone().getClass()); // 输出 class
com.example.demo.Base
        System.out.println(basePro.getClass()); // 输出 class
com.example.demo.BasePro
    }
}
```

可采用两种方式修复

- ① 处改用 super.clone()
- 移除 Base 类整个 clone() 实现

假设你希望在一个类中实现 Cloneable 接口，该类的超类提供了一个表现良好的 clone 方法。首先调用 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本类型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 clone 方法**，because it would merely encourage wasteful copying. 有了这个警告，以下是 PhoneNumber 类的 clone 方法概貌：

```
// 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 类的声明，使之实现 Cloneable 接口。虽然 Object 的 clone 方法返回 Object 类型，但是这个 clone 方法返回 PhoneNumber 类型。这样做是合法的，也是可取的，因为 Java 的返回值类型支持协变。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。我们必须把源自 Object 类的 super.clone() 方法在返回前将结果转换为 PhoneNumber 类型，这类强制转换肯定会成功。

将 super.clone() 包含在 try-catch 块中。这是因为 Object 类声明的 clone 方法会抛出 CloneNotSupportedException，这是一种 checked exception。因为 PhoneNumber 类实现了 Cloneable 接口，所以我们知道对 super.clone() 的调用将会成功。这个样板文件的需求表明 CloneNotSupportedException 应该是 unchecked exception（Item-71）。

如果对象的字段包含可变对象的引用，前面所示 clone 方法的这种简易实现可能引发灾难。例如，考虑 Item-7 中的 Stack 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);
    }
}
```

假设你想让这个类可克隆。如果 clone 方法只返回 super.clone()，得到的 Stack 实例在其 size 字段中会有正确的值，但其 elements 字段将引用与原始 Stack 实例相同的数组。修改初始值将破坏克隆的不变性，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。

调用 Stack 类中唯一构造函数的情况永远不会发生。实际上，clone 方法将充当构造函数；你必须确保它不会对原始对象造成伤害，并且 clone 方法正确地实现了不变性。为了使 Stack 类上的 clone 方法正常工作，它必须复制 Stack 类实例的内部。最简单的做法是在 elements 字段对应的数组递归调用 clone 方法：

```
// 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 方法将返回一个数组，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，复制数组是 clone 机制唯一令人信服的使用场景。

还要注意，如果 elements 字段是 final 修饰的，上述解决方案就无法工作，因为 clone 方法将被禁止为字段分配新值。这是一个基础问题：与序列化一样，可克隆体系结构与使用 final 修饰可变对象引用的常用方式不兼容，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 final 修饰符。

仅仅递归调用 clone 方法并不总是足够的。例如，假设你正在为 HashTable 编写一个 clone 方法，HashTable 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
```

```
    }
```

假设你只是像对 Stack 所做的那样，递归克隆 bucket 数组，如下所示：

```
    // 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 clone 方法有自己的 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
    }
```

私有内部类 HashTable.Entry 已经被增强，它提供了进行「深拷贝」的方法。HashTable 上的 clone 方法分配一个大小合适的新 buckets 数组，并遍历原始 buckets 数组，对每个非空 buckets 元素进行深拷贝。Entry 类的 deepCopy() 方法会被递归调用直至复制完整个链表（该链表以 Entry 类的实例作为头节点）。这种方法虽然很灵活，而且在 buckets 不太长的情况下可以很好地工作，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() 方法的递归调用：

```
    // 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，buckets 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 clone 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，clone 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 修饰或 private 修饰的方法。（如果它是私有的，那么它可能是没有 final 修饰的公共「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）。

用继承（Item-19）方式设计一个类时，你有两种选择，但是无论你选择哪一种，都不应该实现 Cloneable 接口。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 Object 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，你可以选择不实现一个有效的克隆方法，并通过提供以下退化的克隆实现来防止子类实现它：

```
// 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 接口的线程安全类，请记住它的 clone 方法必须正确同步，就像其他任何方法一样（Item-78）。Object 类的 clone 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 clone 方法。

回顾一下，所有实现 Cloneable 接口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。如果类只包含基本数据类型的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 的字段需要修复，即使它是基本数据类型或不可变的。

搞这么复杂真的有必要吗？答案是否定的。如果你扩展了一个已经实现了 Cloneable 接口的类，那么除了实现行为良好的 clone 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) { ... };
```

复制工厂与复制构造函数的静态工厂（Item-1）类似：

```
// 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 接口的危害要小一些，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。

Item 14: Consider implementing Comparable（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.compareTo(x))` （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。

Chapter 4. Classes and Interfaces（类和接口）

Chapter 4 Introduction（章节介绍）

CLASSES and interfaces lie at the heart of the Java programming language. They are its basic units of abstraction. The language provides many powerful elements that you can use to design classes and interfaces. This chapter contains guidelines to help you make the best use of these elements so that your classes and interfaces are usable, robust, and flexible.

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，以便让你的类和接口是可用的、健壮的和灵活的。

Item 15: Minimize the accessibility of classes and members（尽量减少类和成员的可访问性）

要将设计良好的组件与设计糟糕的组件区别开来，最重要的因素是：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，将 API 与实现完全分离。组件之间只通过它们的 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，允许它们被独立开发、测试、优化、使用、理解和修改。这加快了系统开发进度，因为组件可以并行开发。也减轻了维护的负担，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获得良好的性能，但它可以实现有效的性能调优：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复用性，因为没有紧密耦合的组件在其他场景中通常被证明是有用的，除了开发它们时所在的场景之外。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以及声明中出现的访问修饰符（private、protected 和 public）决定。正确使用这些修饰符是信息隐藏的关键。

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，在不影响软件正常功能时，使用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。如果用 public 修饰符声明一个顶级类或接口，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设置为包私有，那么就应该这么做。通过将其设置为包私有，可以使其成为实现的一部分，而不是导出的 API 的一部分，并且可以在后续版本中修改、替换或删除它，而不必担心损害现有的客户端。如果将其公开，就有义务永远提供支持，以保持兼容性。

如果包级私有顶级类或接口只被一个类使用，那么可以考虑：在使用它的这个类中，将顶级类设置为私有静态嵌套类（Item-24）。对于包中的所有类以及使用它的类来说，这降低了它的可访问性。但是，降低公共类的可访问性比减少包级私有顶级类的可访问性重要得多：公共类是包 API 的一部分，而包级私有顶级类已经是实现的一部分。

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，这里按可访问性依次递增的顺序列出：

- **私有**，成员只能从声明它的顶级类内部访问。
- **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，即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**，成员可以通过声明它的类的子类（会受一些限制 [JLS, 6.6.2]）和声明它的包中的任何类访问。

- **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，你应该本能的使所有成员都是私有的。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 private 修饰符，使成员变为包级私有。如果你发现自己经常这样做，那么你应该重新确认系统的设计，看看是否有其他方式能产生更好地相互解耦的类。也就是说，私有成员和包级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，当访问级别从包级私有变为保护时，可访问性会有很大的提高。保护成员是类导出 API 的一部分，必须永远支持。此外，导出类的保护成员表示对实现细节的公开承诺（Item-19）。需要保护成员的场景应该相对少见。

有一个关键规则限制了你降低方法的可访问性。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超类 [JLS, 8.4.8.3] 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同样可用是必要的（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。这个规则的一个特例是，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，你可能会试图使类、接口或成员更容易访问。这在一定程度上是好的。为了测试，将公共类成员由私有变为包私有是可以接受的，但是进一步提高可访问性是不可接受的。换句话说，将类、接口或成员作为包导出 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采用 public 修饰（Item-16）。如果实例字段不是 final 的，或者是对可变对象的引用，那么将其公开，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可变的能力。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，这些字段要么包含基本数据类型，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，这恰恰会导致这个问题。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 List：

```
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();
}
```

如何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。模块是包的分组单位，就像包是类的分组单位一样。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保护成员产生了两个隐式访问级别，它们是正常公共级别和保护级别的类似物。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解决。

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建议级别。如果将模块的 JAR 文件放在应用程序的类路径上，而不是模块路径上，模块中的包将恢复它们的非模块行为：包的公共类的所有公共成员和保护成员都具有正常的可访问性，而不管模块是否导出包 [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，你必须将包以模块分组，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，否则最好还是不使用它们。

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，public 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。

Item 16: In public classes, use accessor methods, not public fields（在公共类中，使用访问器方法，而不是公共字段）

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，不能实现不变量，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，如果一个类是包级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，有时候包级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。

Item 17: Minimize mutability（减少可变性）

不可变类是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、基本类型的包装类、BigInteger 和 BigDecimal。这么做有很好的理由：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**（这类方法也被称为修改器）

2.**确保类不能被继承。** 这可以防止粗心或恶意的通过子类实例对象状态可改变的方式，损害父类的不可变行为。防止子类化通常用 final 修饰父类，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。** 这清楚地表达了意图，并由系统强制执行。同样，如果在没有同步的情况下，引用新创建的实例并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. **所有字段设为私有。** 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本类型或对不可变对象的引用的公共 final 字段，但不建议这样做，因为在以后的版本中无法更改内部表示（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。永远不要向提供对象引用的客户端初始化这样的字段，也不要从访问器返回字段。在构造函数、访问器和 readObject 方法（Item-88）中创建防御性副本（Item-50）。

前面条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，它的每个属性都有访问器，但没有对应的修改器。下面是一个稍微复杂的例子：

```
// 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（包含实部和虚部的数）。除了标准的 Object 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。值得注意的是，算术操作创建和返回一个新的 Complex 实例，而不是修改这个实例。这种模式称为函数式方法，因为方法返回的结果是将函数应用到其操作数，而不是修改它。将其与过程式或命令式方法进行对比，在这种方法中，方法将一个计算过程应用于它们的操作数，从而导致其状态发生变化。注意，方法名是介词（如 plus），而不是动词（如 add）。这强调了这样一个事实，即方法不会改变对象的值。BigInteger 和 BigDecimal 类不遵守这种命名约定，这导致了许多使用错误。

如果不熟悉函数式方法，那么它可能看起来不自然，但它实现了不变性，这么做有很多优势。 **不可变对象很简单。** 不可变对象可以保持它被创建时的状态。如果能够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，而无需你或使用类的程序员做进一步的工作。另一方面，可变对象可以具有任意复杂的状态空间。如果文档没有提供由修改器方法执行的状态转换的精确描述，那么就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。 它们不会因为多线程并发访问而损坏。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。** 同时，不可变类应该鼓励客户端尽可能复用现有的实例。一种简单的方法是为常用值提供公共静态 final 常量。例如，Complex 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，这将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结果之一是，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，即使有，也应该少用（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实现。 例如，BigInteger 类在内部使用符号大小来表示。符号由 int 表示，大小由 int 数组表示。negate 方法产生一个新的 BigInteger，大小相同，符号相反。即使数组是可变的，也不需要复制；新创建的 BigInteger 指向与原始数组相同的内部数组。

不可变对象可以很好的作为其他对象的构建模块， 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内部发生更改，那么维护复杂对象的不变性就会容易得多。这个原则的一个具体的例子是，不可变对象很合适作为 Map 的键和 Set 的元素：你不必担心它们的值在 Map 或 Set 中发生变化，从而破坏 Map 或 Set 的不变性。

不可变对象自带提供故障原子性（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，尤其是对象很大的时候。例如，假设你有一个百万位的 BigInteger，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，也有百万位长，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位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，那么性能问题就会被放大。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，并将它们作为基本数据类型提供。如果将多步操作作为基本数据类型提供，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。可以用 final 修饰以达到目的，但是还有另外一个更灵活的选择，你可以将其所有构造函数变为私有或包私有，并使用公共静态工厂方法来代替公共的构造函数（Item-1）。Complex 类采用这种方式修改后如下所示：

```
// 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
}
```

这种方式通常是最好的选择。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而言，不可变类实际上是 final 类，因为不可能继承自另一个包的类，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，使得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 这一点没有被广泛理解，因此它们的所有方法都可能被重写。遗憾的是，在保留向后兼容性的情况下，这个问题无法得到纠正。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，而不是不受信任的子类实例。如果是后者，你必须防御性的复制它，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());
}
```

这个条目开头的不可变类的规则列表指出，没有方法可以修改对象，它的所有字段必须是 final 的。实际上，这些规则过于严格，可以适当放松来提高性能。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 final 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，这就保证了重复计算会产生相同的结果。

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，以备再次调用。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，应该提出一个警告。如果你选择让不可变类实现 Serializable，并且该类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 或 ObjectInputStream.readUnshared 方法，即使默认的序列化形式是可以接受的。否则攻击者可能创建类的可变实例。Item-88详细讨论了这个主题。

总而言之，不要急于为每个 getter 都编写 setter。**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。你应该始终使小的值对象（如 PhoneNumber 和 Complex）成为不可变的。（Java 库中有几个类，比如 `java.util.Date` 和 `java.awt.Point`，应该是不可改变的，但事实并非如此。）也应该认真考虑将较大的值对象（如 String 和 BigInteger）设置为不可变的。只有确认了实现令人满意的性能是必要的，才应该为不可变类提供一个公共可变伴随类（Item-67）。

对于某些类来说，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更容易地 reason about the object 并减少出错的可能性。因此，除非有令人信服的理由，否则每个字段都应该用 final 修饰。将本条目的建议与 Item-15 的建议结合起来，你自然会倾向于 **声明每个字段为私有 final，除非有很好的理由不这样做。**

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。 除非有充分的理由，否则不要提供与构造函数或静态工厂分离的公共初始化方法。类似地，不要提供「重新初始化」的方法，该方法允许复用对象，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 类体现了这些原则。它是可变的，但是它的状态空间故意保持很小。创建一个实例，使用它一次，它就完成了使命：一旦倒计时锁存器的计数达到零，你可能不会复用它。

关于本条目中 Complex 类的最后一点需要补充的说明。这个例子只是为了说明不变性。它不是一个工业级强度的复数实现。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]。

Item 18: Favor composition over inheritance（优先选择复合而不是继承）

继承是实现代码复用的一种强大方法，但它并不总是最佳的工具。使用不当会导致软件变得脆弱。在同一个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。在对专为扩展而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越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）。本条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，继承破坏了封装 [Snyder86]。换句话说，子类的功能正确与否依赖于它的超类的实现细节。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，即使子类的代码没有被修改过。因此，子类必须与其超类同步发展，除非超类是专门为扩展的目的而设计的，并具有很明确的文档说明。

为了使问题更具体一些，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，我们编写了一个变量，它记录试图插入 HashSet 的元素数量，并为这个计数变量导出一个访问器。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;
    }
```

```
    }
```

这个类看起来是合理的，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 HashSet 内部，addAll 方法是基于 add 方法实现的，尽管 HashSet 理所当然的没有记录这个实现细节。InstrumentedHashSet 中的 addAll 方法使 addCount 变量增加了 3，然后通过 `super.addAll` 调用 HashSet 的 addAll 实现。这相当于反过来调用在 InstrumentedHashSet 中被覆盖的 add 方法，每个元素一次。这三个调用中的每一个都使 addCount 变量增加了 1，总共增加了 6，即使用 addAll 方法添加的每个元素都被重复计数。

我们可以通过移除子类覆盖的 addAll 方法来「修复」。虽然生成的类可以工作，但它的正确功能取决于 HashSet 的 addAll 方法是基于 add 方法实现的事实。这种「自用」是实现细节，不能保证在 Java 平台的所有实现中都保留，并且在不同的版本中可能会有变化。因此，生成的 InstrumentedHashSet 类是脆弱的。

如果通过覆盖 addAll 方法来遍历指定的集合，对每个元素调用一次 add 方法会稍好一些。无论 HashSet 的 addAll 方法是否基于 add 方法实现都能保证得到正确结果，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，这并不总是可行的，如果子类无法访问某些私有字段，这些方法就无法实现。

子类脆弱的一个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断言。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断言。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 被重新改装以加入 Collections 框架时，必须修复几个这种性质的安全漏洞。

这两个问题都源于覆盖方法。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。虽然这种扩展会更安全，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，而子类具有一个相同签名和不同返回类型的方法，那么你的子类将不能编译 [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约定是值得怀疑的，因为在你编写子类方法时，该

约定还没有被写入。

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，因为现有的类是新类的一个组件。新类中的每个实例方法调用现有类实例的对应方法，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。为了说明问题，这里有一个使用复合和转发方法的案例，用以替代 InstrumentedHashSet。注意，实现被分成两部分，类本身和一个可复用的转发类，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，并且有一个参数也是 Set 类型的构造函数。实际上，该类具有「插装」功能，可间接将一种 Set 的转化为另一种 Set。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，包装器类可用于插装任何 Set 实现，并将与任何现有构造函数一起工作：

译注：instrumentation 译为「插装」，类比硬盘（Set）插装到主板（ForwardingSet），无论硬盘如何更新换代，但是用于存储的功能不会变。外设（客户端）通过主板的 USB2.0 口（InstrumentedSet2.0）或 USB3.0 口（InstrumentedSet3.0），与硬盘交互，使用其存储功能。

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

InstrumentedSet 类甚至还可以用来临时配置一个没有「插装」功能的 Set 实例：

```
static void walk(Set<Dog> dogs) {
InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
... // Within this method use iDogs instead of dogs
}
```

InstrumentedSet 类被称为包装器类，因为每个 InstrumentedSet 实例都包含（「包装」）了另一个 Set 实例。这也称为装饰者模式 [Gamma95]，因为 InstrumentedSet 类通过添加「插装」来「装饰」一个集合。有时复合和转发的组合被不当地称为委托。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]。

包装器类的缺点很少。一个需要注意的点是：包装器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对应的包装器，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「自用」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复用的转发类，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]。

只有子类确实是超类的子类型的情况下，继承才合适。换句话说，两个类 A、B 之间只有 B 满足「is-a」关系时才应该扩展 A。如果你想让 B 扩展 A，那就问问自己：每个 B 都是 A 吗？如果不能对这个问题给出肯定回答，B 不应该扩展 A；如果答案是否定的，通常情况下，B 应该包含 A 的私有实例并暴露不同的 API：A 不是 B 的基本组成部分，而仅仅是其实现的一个细节。

在 Java 库中有许多明显违反这一原则的地方。例如，stack 不是 vector，因此 Stack 不应该继承 Vector。类似地，property 列表不是 hash 表，因此 Properties 不应该继承 Hashtable。在这两种情况下，复合都是可取的。

如果在复合适用的地方使用了继承，就会不必要地暴露实现细节。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，通过暴露内部组件，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 Properties 类的实例，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 Properties 类，设计者希望只允许字符串作为键和值，但是直接访问底层 Hashtable 允许违反这个不变性。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，已经来不及纠正了，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，因为它打破了封装。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，如果子类与超类在不同的包中，并且超类不是为继承而设计的，继承也可能导致程序脆弱。为了避免这种缺陷，应使用复合和转发，特别是存在适当接口能实现包装器类时更应如此。包装类不仅比子类更健壮，而且功能更强大。

Item 19: Design and document for inheritance or else prohibit it（继承要设计良好并且具有文档，否则禁止使用）

Item-18 提醒你注意：将不是为继承设计并且缺少文档的「外部」类进行子类化的危险。那么，为继承而设计并且具备文档的类意味着什么呢?

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性。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所有情况。例如，可能调用来自后台线程或静态初始化器的方法。

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「Implementation Requirements（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)
>
> 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）。
>
> 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用性模式。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)
>
> 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。）
>
> 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。
>
> 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。
>
> 参数
>
> 要删除的第一个元素的 fromIndex 索引。
>
> 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，**你必须在发布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 clone 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：clone 和 readObject 都不能直接或间接调用可覆盖的方法。对于 readObject，覆盖方法将在子类的状态反序列化之前运行。在 clone 的情况下，覆盖方法将在子类的 clone 方法有机会修复 clone 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 clone 的情况下，失败可以破坏原始对象和 clone。例如，如果覆盖方法假设它正在修改对象的深层结构的 clone 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文档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，完全消除类对可重写方法的自使用。这样，您将创建一个对子类而言相当安全的类。重写一个方法不会影响任何其他方法的行为。

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。

Item 20: Prefer interfaces to abstract classes（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，这种限制对抽象类而言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：

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，这种限制对抽象类而言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 JDK 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 2^n 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公共行为。

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。

如果接口方法的实现与其他接口方法类似，那么可以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，这个例子是一个令人印象深刻的演示，体现了骨架实现的强大功能。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基本方法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。如果基本方法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种是简单实现，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。

Item 21: Design interfaces for posterity（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，通常导致现有的实现出现编译时错误，提示缺少该方法。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 Java 使得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 lambda 表达式的使用（Chapter 6）。**但是，并不总是能够编写一个默认方法来维护每个实现所有不变性**

例如，考虑在 Java 8 中被添加到集合接口中的 removeIf 方法。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 predicate 返回 true 的元素。声明大概是这样的：

```
// 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，它在实际使用的一些 Collection 实现中导致了问题。例如，考虑 `org.apache.commons.collections4.collection.SynchronizedCollection`。这个类来自 Apache Commons 库，类似于 `java.util` 提供的静态工厂 `Collections.synchronizedCollection`。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通过编译，但是在运行时出错。虽然这个问题并不常见，但也没有那么罕见。已知 Java 8 中添加到集合接口的少数方法是易受影响的，会影响到现存的一部分实现。

除非别无他法，否则应该避免使用默认方法向现有接口添加新方法，如果非要这么做，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，默认方法非常有助于提供标准方法实现，以减轻实现接口的任务量（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。你不能做出这些更改，除非破坏现有实现。

教训显而易见。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，它可能会永远影响它的使用者；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测试每个接口。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，用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。**

Item 22: Use interfaces only to define types（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使用者可以对类的实例做什么。为其他任何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，以避免用类名修饰常量名。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;
}
```

常量接口模式是使用接口的糟糕方式。类内部会使用一些常量，这是实现细节。然而，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命名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，所有数值包装类，比如 Integer 和 Double，都导出 MIN_VALUE 和 MAX_VALUE 常量。如果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，注意可以在数字字面值中使用下划线（ _ ）。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。无论是不是固定的浮点数，如果它们包含五个或多个连续数字，都可以考虑添加下划线到数字字面值。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。

Item 23: Prefer class hierarchies to tagged classes（类层次结构优于带标签的类）

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来表示实例的样式。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。它们充斥着样板代码，包括 enum 声明、标签字段和 switch 语句。因为多个实现在一个类中混杂，会造成可读性受损。内存占用也增加了，因为实例被其他类型的不相关字段所拖累。除非构造函数初始化不相关的字段，否则不能将字段设置为 final，但这会导致更多的样板文件。构造函数必须设置标签字段并在没有编译器帮助的情况下初始化正确的数据字段：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，那么你必须记住要为每个 switch 语句添加一个 case，否则类将在运行时出错。最后，实例的数据类型没有给出它任何关于样式的提示。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。**标签类只是类层次结构的简易模仿。**

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样式使用，将它们放在这个类中。在 Figure 类中没有这样的独立于样式的方法或字段。

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样式的数据字段。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代码。每种样式的实现都分配有自己的类，这些类没有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抽象方法都有一个实现。这消除了由于缺少 switch case 而导致运行时出错的可能性。多个程序员可以独立地、可互操作地扩展层次结构，而无需查看根类的源代码。每种样式都有一个单独的数据类型，允许程序员指出变量的样式，并将变量和输入参数限制为特定的样式。

类层次结构的另一个优点是，可以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：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使用场景。如果想编写一个带有显式标签字段的类，请考虑是否可以删除标签并用层次结构替换。当遇到具有标签字段的现有类时，请考虑将其重构为层次结构。

Item 24: Favor static member classes over nonstatic（静态成员类优于非静态成员类）

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 Adapter [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：

```java
// 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。更糟糕的是，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。匿名类可以在代码中用在任何一个可以用表达式的地方。当且仅当它们出现在非静态环境中时，匿名类才持有外部类实例。但是，即使它们出现在静态环境中，它们也不能有除常量（final 修饰的基本类型或者初始化为常量表达式的字符串 [JLS, 4.12.4]）以外的任何静态成员。

匿名类的使用有很多限制。除非在声明它们的时候，你不能实例化它们。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。匿名类的使用者除了从超类继承的成员外，不能调用任何成员。因为匿名类出现在表达式中，所以它们必须保持简短——大约 10 行或更短，否则会影响可读性。

在 lambda 表达式被添加到 Java（Chapter 6）之前，匿名类是动态创建小型函数对象和进程对象的首选方法，但 lambda 表达式现在是首选方法（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。除非成员类的每个实例都需要引用其外部类实例，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。

Item 25: Limit source files to a single top-level class（源文件仅限有单个顶层类）

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：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。说得更具体些，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（Utensil 和 Dessert）：

```
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 Utensil 类和 Dessert 类：

```
// Two classes defined in one file. Don't ever do this!
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!
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 Utensil 和 Dessert。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 Utensil 和 Dessert）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），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会拥有多个定义。这反过来保证了编译所生成的类文件，以及程序的行为，与源代码文件传递给编译器的顺序无关。

Chapter 5. Generics（泛型）

Chapter 5 Introduction（章节介绍）

SINCE Java 5, generics have been a part of the language. Before generics, you had to cast every object you read from a collection. If someone accidentally inserted an object of the wrong type, casts could fail at runtime. With generics,you tell the compiler what types of objects are permitted in each collection. The compiler inserts casts for you automatically and tells you at compile time if you try to insert an object of the wrong type. This results in programs that are both safer and clearer, but these benefits, which are not limited to collections, come at a price. This chapter tells you how to maximize the benefits and minimize the complications.

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。

Item 26: Don’t use raw types（不要使用原始类型）

首先，介绍一些术语。声明中具有一个或多个类型参数的类或接口就是泛型类或泛型接口 [JLS, 8.1.2, 9.1.2]。例如，List 接口有一个类型参数 E，用于表示其元素类型。该接口的全名是 `List<E>`（读作「List of E」），但人们通常简称为 List。泛型类和泛型接口统称为泛型。

每个泛型定义了一组参数化类型，这些参数化类型包括类名或接口名，以及带尖括号的参数列表，参数列表是与泛型的形式类型参数相对应的实际类型 [JLS, 4.4, 4.5]。例如，`List<String>`（读作「List of String」）是一个参数化类型，表示元素类型为 String 类型的 List。（String 是与形式类型参数 E 对应的实际类型参数。）

最后，每个泛型都定义了一个原始类型，它是没有任何相关类型参数的泛型的名称 [JLS, 4.8]。例如，`List<E>` 对应的原始类型是 List。原始类型的行为就好像所有泛型信息都从类型声明中删除了一样。它们的存在主要是为了与之前的泛型代码兼容。

在将泛型添加到 Java 之前，这是一个典型的集合声明。就 Java 9 而言，它仍然是合法的，但不应效仿：

```
// Raw collection type - don't do this!
// My stamp collection. Contains only Stamp instances.
private final Collection stamps = ... ;
```

如果你今天使用这个声明，然后意外地将 coin 放入 stamp 集合中，这一错误的插入依然能够编译并没有错误地运行（尽管编译器确实发出了模糊的警告）：

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... )); // Emits "unchecked call" warning
```

直到从 stamp 集合中获取 coin 时才会收到错误提示：

```
// Raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); )
    Stamp stamp = (Stamp) i.next(); // Throws ClassCastException
stamp.cancel();
```

正如在本书中提到的，在出现错误之后尽快发现错误是有价值的，最好是在编译时。在本例这种情况下，直到运行时（在错误发生很久之后）才发现错误，而且报错代码可能与包含错误的代码相距很远。一旦看到 ClassCastException，就必须在代码中搜索将 coin 放进 stamp 集合的方法调用。编译器不能帮助你，因为它不能理解注释「Contains only Stamp instances.」

对于泛型，类型声明应该包含类型信息，而不是注释：

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

从这个声明看出，编译器应该知道 stamps 应该只包含 Stamp 实例，为保证它确实如此，假设你的整个代码库编译没有发出（或抑制；详见 Item-27）任何警告。当 stamps 利用一个参数化的类型进行声明时，错误的插入将生成编译时错误消息，该消息将确切地告诉你哪里出了问题：

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

当从集合中检索元素时，编译器会为你执行不可见的强制类型转换，并确保它们不会失败（再次假设你的所有代码没有产生或抑制任何编译器警告）。虽然不小心将 coin 插入 stamps 集合看起来有些牵强，但这类问题是真实存在的。例如，很容易想象将一个 BigInteger 放入一个只包含 BigDecimal 实例的集合中。

如前所述，使用原始类型（没有类型参数的泛型）是合法的，但是你永远不应该这样做。**如果使用原始类型，就会失去泛型的安全性和表现力。** 既然你不应该使用它们，那么为什么语言设计者一开始就允许原始类型呢？答案是：为了兼容性。Java 即将进入第二个十年，泛型被添加进来时，还存在大量不使用泛型的代码。保持所有这些代码合法并与使用泛型的新代码兼容被认为是关键的。将参数化类型的实例传递给设计用于原始类型的方法必须是合法的，反之亦然。这被称为迁移兼容性的需求，它促使原始类型得到支持并使用擦除实现泛型 （Item-28）。

虽然你不应该使用原始类型（如 List），但是可以使用参数化的类型来允许插入任意对象，如 `List<Object>`。原始类型 List 和参数化类型 `List<Object>` 之间的区别是什么？粗略地说，前者选择了不使用泛型系统，而后者明确地告诉编译器它能够保存任何类型的对象。虽然可以将 `List<String>` 传递给 List 类型的参数，但不能将其传递给类型 `List<Object>` 的参数。泛型有子类型规则，`List<String>` 是原始类型 List 的子类型，而不是参数化类型 `List<Object>` 的子类型（Item-28）。因此，**如果使用原始类型（如 List），就会失去类型安全性，但如果使用参数化类型（如 `List<Object>`）则不会。**

为了使这一点具体些，考虑下面的程序：

```
// 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);
}
```

该程序可以编译，但因为它使用原始类型 List，所以你会得到一个警告：

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

实际上，如果你运行程序，当程序试图将调用 `strings.get(0)` 的结果强制转换为字符串时，你会得到一个 ClassCastException。这是一个由编译器生成的强制类型转换，它通常都能成功，但在本例中，我们忽略了编译器的警告，并为此付出了代价。

如果将 unsafeAdd 声明中的原始类型 List 替换为参数化类型 `List<Object>`，并尝试重新编译程序，你会发现它不再编译，而是发出错误消息：

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

对于元素类型未知且无关紧要的集合，你可能会尝试使用原始类型。例如，假设你希望编写一个方法，该方法接受两个集合并返回它们共有的元素数量。如果你是使用泛型的新手，那么你可以这样编写一个方法：

```
// 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;
}
```

这种方法是可行的，但是它使用的是原始类型，这是很危险的。安全的替代方法是使用无界通配符类型。如果你想使用泛型，但不知道或不关心实际的类型参数是什么，那么可以使用问号代替。例如，泛型集 `Set<E>` 的无界通配符类型是 `Set<?>`（读作「set of some type」）。它是最通用的参数化集合类型，能够容纳任何集合：

```
// Uses unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

无界通配符类型 `Set<?>` 和原始类型 Set 之间的区别是什么？问号真的能起作用吗？我并不是在强调这一点，但是通配符类型是安全的，而原始类型则不是。将任何元素放入具有原始类型的集合中，很容易破坏集合的类型一致性（如上述的 unsafeAdd 方法所示）；你不能将任何元素（除了 null）放入 `Collection<?>` 。尝试这样做将生成这样的编译时错误消息：

```
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 ?
```

无可否认，这个错误消息让人不满意，但是编译器已经完成了它的工作，防止你无视它的元素类型而破坏集合的类型一致性。你不仅不能将任何元素（除 null 之外）放入 `Collection<?>` ，而且不能臆想你得到的对象的类型。如果这些限制是不可接受的，你可以使用泛型方法（Item-30）或有界通配符类型（Item-31）。

对于不应该使用原始类型的规则，有一些小的例外。**必须在类字面量中使用原始类型。** 该规范不允许使用参数化类型（尽管它允许数组类型和基本类型）[JLS, 15.8.2]。换句话说， `List.class` ， `String[].class` 和 `int.class` 都是合法的，但是 `List<String>.class` 和 `List<?>.class` 不是。

规则的第二个例外是 instanceof 运算符。由于泛型信息在运行时被删除，因此在不是无界通配符类型之外的参数化类型上使用 instanceof 操作符是非法的。使用无界通配符类型代替原始类型不会以任何方式影响 instanceof 运算符的行为。在这种情况下，尖括号和问号只是多余的。**下面的例子是使用通用类型 instanceof 运算符的首选方法：**

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

注意，一旦确定 o 是一个 Set，就必须将其强制转换为通配符类型 `Set<?>` ，而不是原始类型 Set。这是一个经过检查的强制类型转换，所以不会引发编译器警告。

总之，使用原始类型可能会在运行时导致异常，所以不要轻易使用它们。它们仅用于与引入泛型之前的遗留代码进行兼容和互操作。快速回顾一下，`Set<Object>` 是一个参数化类型，表示可以包含任何类型的对象的集合，`Set<?>` 是一个通配符类型，表示只能包含某种未知类型的对象的集合，Set 是一个原始类型，它选择了泛型系统。前两个是安全的，后一个就不安全了。

为便于参考，本条目中介绍的术语（以及后面将要介绍的一些术语）总结如下：

Term	Example	Item
Parameterized type	`List<String>`	Item-26
Actual type parameter	`String`	Item-26
Generic type	`List<E>`	Item-26, Item-29
Formal type parameter	`E`	Item-26
Unbounded wildcard type	`List<?>`	Item-26
Raw type	`List`	Item-26
Bounded type parameter	`<E extends Number>`	Item-29
Recursive type bound	`<T extends Comparable<T>>`	Item-30
Bounded wildcard type	`List<? extends Number>`	Item-31
Generic method	`static <E> List<E> asList(E[] a)`	Item-30
Type token	`String.class`	Item-33

Item 27: Eliminate unchecked warnings（消除 unchecked 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。使用泛型的经验越丰富，遇到的警告就越少，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();
```

The compiler will gently remind you what you did wrong:

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。

Item 28: Prefer lists to arrays（list 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。

Item 29: Favor generic types（优先使用泛型）

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。

Item 30: Favor generic methods（优先使用泛型方法）

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）。

Item 31: Use bounded wildcards to increase API flexibility（使用有界通配符增加 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。

Item 32: Combine generics and varargs judiciously（明智地合用泛型和可变参数）

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：

1. 它没有在可变参数数组中存储任何东西，并且
2. 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。

Item 33: Consider typesafe heterogeneous containers（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` 。

Chapter 6. Enums and Annotations（枚举和注解）

Chapter 6 Introduction（章节介绍）

JAVA supports two special-purpose families of reference types: a kind of class called an enum type, and a kind of interface called an annotation type. This chapter discusses best practices for using these type families.

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。

Item 34: Use enums instead of int constants（用枚举类型代替 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，也不能继承它，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 values() 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）。

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复用该枚举，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用 switch 接收枚举值：

```
// 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 throw 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 Operation 枚举的一个版本，它重写 toString 方法来返回与操作相关的符号：

译注：原文 constantspecific data 应修改为 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);
    }
```

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。

Item 35: Use instance fields instead of ordinals（使用实例字段替代序数）

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。

Item 36: Use EnumSet instead of bit fields（用 EnumSet 替代位字段）

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。

Item 37: Use EnumMap instead of ordinal indexing（使用 EnumMap 替换序数索引）

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）。

Item 38: Emulate extensible enums with interfaces（使用接口模拟可扩展枚举）

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。

Item 39: Prefer annotations to naming patterns（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。

译注 1：注解的保留策略

保留策略决定了在什么位置丢弃注解。Java 定义了 3 种策略，它们被封装到 `java.lang.annotation.RetentionPolicy` 枚举中。这 3 种策略分别是 SOURCE、CLASS 和 RUNTIME。

- 使用 SOURCE 保留策略的注解，只在源文件中保留，在编译期间会被抛弃。
- 使用 CLASS 保留策略的注解，在编译时被存储到 `.class` 文件中。但是，在运行时不能通过 JVM 得到这些注解。
- 使用 RUNTIME 保留策略的注解，在编译时被存储到 `.class` 文件中，并且在运行时可以通过 JVM 获取这些注解。因此，RUNTIME 保留策略提供了最永久的注解。

译注 2：ElementType 各常量定义的范围

- ElementType.TYPE
  - Class, interface (including annotation type), or enum declaration（类、接口、注解、枚举）
- ElementType.FIELD
  - Field declaration (includes enum constants)（字段、枚举常量）
- ElementType.METHOD
  - Method declaration（方法）
- ElementType.PARAMETER
  - Formal parameter declaration（方法参数）
- ElementType.CONSTRUCTOR
  - Constructor declaration（构造）
- ElementType.LOCAL_VARIABLE
  - Local variable declaration（局部变量）
- ElementType.ANNOTATION_TYPE
  - Annotation type declaration（注解）
- ElementType.PACKAGE
  - Package declaration（包）
- ElementType.TYPE_PARAMETER
  - Type parameter declaration（泛型参数）
  - Since: 1.8
- ElementType.TYPE_USE
  - Use of a type（任意类型，获取 class 对象和 import 两种情况除外）
  - Since: 1.8
- ElementType.MODULE
  - Module declaration（模块）

- Since: 9

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 工具以命令行方式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(ExceptionTest.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。

进一步修改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。「重复状态」会生成名为「容器注解类型」的合成注解。getAnnotationsByType 方法可忽略这一区别，它可以用于访问可重复注解类型的「重复状态」和「非重复状态」。但是 isAnnotationPresent 明确指出，「重复状态」的情况不属于注解类型，而是「容器注解类型」。如果一个元素是某种类型的「重复状态」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，你将发现它提示不存在。因此，使用此方法检查注解类型的存在与否，将导致你的程序忽略「重复状态」。类似地，使用此方法检查「容器注解类型」将导致程序忽略「非重复状态」。要使用 isAnnotationPresent 检测「重复状态」和「非重复状态」，需要同时检查注解类型及其「容器注解类型」。下面是我们的 RunTests 程序的相关部分修改为使用 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);
    }
}
```

译注：比较原文中提及的 getAnnotationsByType 与 isAnnotationPresent 在可重复注解的「重复状态」和「非重复状态」下的使用差别：

原 doublyBad 方法不变，属于「重复状态」（重复注解大于等于两个的，都属于「重复状态」）；新增一个 doublyBad2 方法，仅使用一个重复注解，属于「非重复状态」

```
class Simple4 {
    // Code containing a repeated annotation
    @ExceptionTest(IndexOutOfBoundsException.class)
    @ExceptionTest(NullPointerException.class)
    public static void doublyBad() {
    }

    @ExceptionTest(ArithmeticException.class)
    public static void doublyBad2() {
    }
}
```

测试代码

```
public static void main(String[] args) throws NoSuchMethodException {
    Class<?> testClass = Simple4.class;
    for (int count = 1; count <= 2; count++) {
        Method m = testClass.getMethod(count == 1 ? "doublyBad" : "doublyBad" +
count);
        System.out.println(m.getName() + "「重复状态」: " +
m.isAnnotationPresent(ExceptionTest.class));
        System.out.println(m.getName() + "「容器注解类型」: " +
m.isAnnotationPresent(ExceptionTestContainer.class));
        System.out.println(m.getName() + "「非重复状态」: " +
m.isAnnotationPresent(ExceptionTest.class));
        System.out.println(m.getName() + "「重复状态」: " +
m.getAnnotationsByType(ExceptionTest.class));
        System.out.println(m.getName() + "「容器注解类型」: " +
m.getAnnotationsByType(ExceptionTestContainer.class));
        System.out.println(m.getName() + "「非重复状态」: " +
m.getAnnotationsByType(ExceptionTest.class));
    }
}
```

结果

```
doublyBad「重复状态」: false
doublyBad「容器注解类型」: true
doublyBad「非重复状态」: false
doublyBad「重复状态」: [LItem_39.ExceptionTest;@1593948d
doublyBad「容器注解类型」: [LItem_39.ExceptionTestContainer;@1b604f19
doublyBad「非重复状态」: [LItem_39.ExceptionTest;@7823a2f9

doublyBad2「重复状态」: true
doublyBad2「容器注解类型」: false
doublyBad2「非重复状态」: true
doublyBad2「重复状态」: [LItem_39.ExceptionTest;@cb5822
doublyBad2「容器注解类型」: [LItem_39.ExceptionTestContainer;@4b9e13df
doublyBad2「非重复状态」: [LItem_39.ExceptionTest;@2b98378d
```

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。

Item 40: Consistently use the Override annotation（坚持使用 @Override 注解）

Java 库包含几种注解类型。对于大多数的程序员来说，其中最重要的是 `@Override` 。此注解只能在方法声明上使用，带有该注解的方法声明将覆盖超类型中的声明。如果你坚持使用这个注解，它将帮助你减少受到有害错误的影响。考虑这个程序，其中类 Bigram 表示一个二元语法，或有序的字母对：

```
// 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());
        }
    }
```

主程序重复地向一个集合中添加 26 个 bigram，每个 bigram 由两个相同的小写字母组成。然后它打印该集合的大小。如果你尝试运行该程序，你会发现它打印的不是 26 而是 260。有什么问题吗?

显然，Bigram 类的作者打算覆盖 equals 方法（Item-10），甚至还记得要一并覆盖 hashCode（Item-11）。不幸的是，我们的程序员没有覆盖 equals，而是重载了它（Item-52）。要覆盖 `Object.equals`，你必须定义一个 equals 方法，它的参数是 Object 类型的，但是 Bigram 的 equals 方法的参数不是 Object 类型的，所以 Bigram 从 Object 继承 equals 方法。这个继承来的 equals 方法只能检测对象同一性，就像 == 操作符一样。每 10 个 bigram 副本为一组，每组中的每个 bigram 副本都不同于其他 9 个，因此 `Object.equals` 认为它们不相等，这就解释了为什么程序最终打印 260。

幸运的是，编译器可以帮助你找到这个错误，但前提是你告诉它你打算覆盖 `Object.equals`。为此，请使用 `@Override` 注解标记 `Bigram.equals`，如下所示:

```
@Override
public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

如果你插入此注解并尝试重新编译程序，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) {
^
```

你会立刻意识到自己做错了什么，拍拍自己的额头，用正确的方式替换不正确的 equals 实现（Item-10）：

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

因此，你应该在 **要覆盖超类声明的每个方法声明上使用 @Override 注解。** 这条规则有一个小小的例外。如果你正在编写一个没有标记为 abstract 的类，并且你认为它覆盖了其超类中的抽象方法，那么你不必费心在这些方法上添加 `@Override` 注解。在未声明为抽象的类中，如果未能覆盖抽象超类方法，编译器将发出错误消息。但是，你可能希望让类中覆盖超类方法的所有方法更加引人注目，在这种情况下，你也可以自由选择是否注解这些方法。大多数 IDE 都可以设置为在选择覆盖方法时自动插入覆盖注解。

大多数 IDE 都提供了一致使用 `@Override` 注解的另一个原因。如果启用适当的检查，如果你的方法没有 `@Override` 注解，但确实覆盖了超类方法，IDE 将生成警告。如果你一致地使用 `@Override` 注解，这些警告将提醒你防止意外覆盖。它们补充编译器的错误消息，这些错误消息会警告你无意的覆盖错误。在 IDE 和编译器的帮助下，你可以确保在任何你想要实施覆盖的地方都覆盖了，而没有遗漏。

`@Override` 注解可用于覆盖接口和类声明的方法声明。随着默认方法的出现，最好对接口方法的具体实现使用 `@Override` 来确保签名是正确的。如果你知道接口没有默认方法，你可以选择忽略接口方法的具体实现上的 `@Override` 注解，以减少混乱。

然而，在抽象类或接口中，标记覆盖超类或超接口方法的所有方法是值得的，无论是具体的还是抽象的。例如，Set 接口不会向 Collection 接口添加任何新方法，因此它的所有方法声明的应该包含 `@Override` 注解，以确保它不会意外地向 Collection 接口添加任何新方法。

总之，如果你在每个方法声明上都使用 `@Override` 注解来覆盖超类型声明（只有一个例外），那么编译器可以帮助你减少受到有害错误的影响。在具体类中，可以不对覆盖抽象方法声明的方法使用该注解（即使这么做也并不会有害）。

Item 41: Use marker interfaces to define types（使用标记接口定义类型）

标记接口是一种不包含任何方法声明的接口，它只是指定（或「标记」）一个类，该类实现了具有某些属性的接口。例如，考虑 Serializable 接口（Chapter 12）。通过实现此接口，表示类的实例可以写入 ObjectOutputStream（或「序列化」）。

你可能听过一个说法：标记接口已经过时，更好的方式是标记注解（Item-39）。这个言论是错误的。与标记注解相比，标记接口有两个优点。首先，**标记接口定义的类型由标记类的实例实现；标记注解不会。** 标记接口类型的存在允许你在编译时捕获错误，如果你使用标记注解，则在运行时才能捕获这些错误。

Java 的序列化工具（Chapter 6）使用 Serializable 标记接口来表明一个类是可序列化的。`ObjectOutputStream.writeObject` 方法序列化传递给它的对象，它要求其参数是可序列化的。假设该方法的参数类型是 Serializable，那么在编译时（通过类型检查）就会检测到对不合适的对象进行序列化的错误。编译时错误检测是使用标记接口的目的，但不幸的是，`ObjectOutputStream.writeObject` 没有利用 Serializable 接口：它的参数被声明为 Object 类型，因此，如果尝试序列化一个不可序列化对象，直到运行时才会提示失败。

译注 1：原文 `ObjectOutputStream.write` 有误，该方法的每种重载仅支持 int 类型和 byte[]，应修改为 `ObjectOutputStream.writeObject`，其源码如下：

```
    public final void writeObject(Object obj) throws IOException {
        if (enableOverride) {
            writeObjectOverride(obj);
            return;
        }
        try {
            writeObject0(obj, false);
        } catch (IOException ex) {
            if (depth == 0) {
                writeFatalException(ex);
```

```
        }
        throw ex;
    }
}
```

译注 2：使用 ObjectOutputStream.writeObject 的例子

```
public class BaseClass implements Serializable {
    private final int id;
    private final String name;

    public BaseClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return "id=" + id + ", name='" + name + '\'';
    }
}

public class Main {
    private void Out() throws IOException {
        BaseClass obj = new BaseClass(1, "Mark");
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(new File("out.txt")))) {
            out.writeObject(obj);
        }
    }

    private void In() throws IOException, ClassNotFoundException {
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(new File("out.txt")))) {
            BaseClass obj = (BaseClass) in.readObject();
            System.out.println(obj);
        }
    }
}
```

标记接口相对于标记注解的另一个优点是可以更精确地定位它们。 如果注解类型使用 `@Target(ElementType.TYPE)` 声明，它可以应用于任何类或接口。假设你有一个只适用于特定接口来实现的标记。如果将其定义为标记接口，则可以让它扩展其适用的惟一接口，确保所有标记的类型也是其适用的惟一接口的子类型。

可以说，Set 接口就是这样一个受限的标记接口。它只适用于 Collection 的子类，但是除了 Collection 定义的方法之外，它不添加任何方法。它通常不被认为是一个标记接口，因为它细化了几个 Collection 方法的约定，包括 add、equals 和 hashCode。但是很容易想象一个标记接口只适用于某些特定接口的子类，而不细化任何接口方法的约定。这样的标记接口可能描述整个对象的某个不变量，或者表明实例能够利用其他类的方法进行处理（就像 Serializable 接口能够利用 ObjectOutputStream 进行处理一样）。

相对于标记接口，标记注解的主要优势是它们可以是其他注解功能的一部分。 因此，标记注解能够与基于使用注解的框架保持一致性。

那么什么时候应该使用标记注解，什么时候应该使用标记接口呢？显然，如果标记应用于类或接口之外的任何程序元素，则必须使用标记注解，因为只有类和接口才能实现或扩展接口。如果标记只适用于类和接口，那么可以问自己这样一个问题：「我是否可以编写一个或多个方法，只接受具有这种标记的对象？」如果是这样，你应该使用标记接口而不是标记注解。这将使你能够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。如果你确信自己永远不会编写只接受带有标记的对象的方法，那么最好使用标记注解。此外，如果框架大量使用注解，那么标记注解就是明确的选择。

总之，标记接口和标记注解都有各自的用途。如果你想要定义一个没有与之关联的新方法的类型，可以使用标记接口。如果你希望标记类和接口之外的程序元素，或者将标记符放入已经大量使用注解类型的框架中，那么标记注解就是正确的选择。如果你发现自己编写的标记注解类型有 `@Target(ElementType.TYPE)` 声明（**译注：意在说明既可以用标记注解，也可以用标记接口的情况**），那么请花时间弄清楚究竟应该用注解类型，还是标记接口更合适。

从某种意义上说，本条目与 Item-22 的说法相反，也就是说，「如果不想定义类型，就不要使用接口。」，与本条目应用场景适应的说法是，「如果你确实想定义类型，那么就要使用接口。」

Chapter 7. Lambdas and Streams（λ 表达式和流）

Chapter 7 Introduction（章节介绍）

In Java 8, functional interfaces, lambdas, and method references were added to make it easier to create function objects. The streams API was added in tandem with these language changes to provide library support for processing sequences of data elements. In this chapter, we discuss how to make best use of these facilities.

在 Java 8 中，为了更容易地创建函数对象，添加了函数式接口、lambda 表达式和方法引用；流 API 也与这些语言特性一并添加进来，为处理数据元素序列提供库支持。在这一章中，我们将讨论如何最好地利用这些工具。

Item 42: Prefer lambdas to anonymous classes（λ 表达式优于匿名类）

在历史上，带有单个抽象方法的接口（或者抽象类，但这种情况很少）被用作函数类型。它们的实例（称为函数对象）表示函数或操作。自从 JDK 1.1 在 1997 年发布以来，创建函数对象的主要方法就是匿名类（Item-24）。下面是一个按长度对字符串列表进行排序的代码片段，使用一个匿名类来创建排序的比较函数（它强制执行排序顺序）：

```
// 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());
    }
});
```

匿名类对于需要函数对象的典型面向对象设计模式来说已经足够了，尤其是策略模式 [Gamma95]。Comparator 接口表示排序的抽象策略；上述匿名类是对字符串排序的一种具体策略。然而，匿名类的冗长使函数式编程在 Java 中变得毫无吸引力。

在 Java 8 中官方化了一个概念，即具有单个抽象方法的接口是特殊的，应该得到特殊处理。这些接口现在被称为函数式接口，允许使用 lambda 表达式创建这些接口的实例。Lambda 表达式在功能上类似于匿名类，但是更加简洁。下面的代码片段，匿名类被 lambda 表达式替换。已经没有了原有刻板的样子，意图非常明显：

```
// Lambda expression as function object (replaces anonymous class)
Collections.sort(words,(s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

注意，lambda 表达式（`Comparator<String>`）、它的参数（s1 和 s2，都是字符串）及其返回值（int）的类型在代码中不存在。编译器使用称为类型推断的过程从上下文中推断这些类型。在某些情况下，编译器无法确定类型，你必须显式指定它们。类型推断的规则很复杂：它们在 JLS 中占了整整一章 [JLS, 18]。很少有程序员能详细理解这些规则，但这没有关系。**省略所有 lambda 表达式参数的类型，除非它们的存在使你的程序更清晰。** 如果编译器生成一个错误，告诉你它不能推断 lambda 表达式参数的类型，那么就显式指定它。有时你可能必须强制转换返回值或整个 lambda 表达式，但这种情况很少见。

关于类型推断，有些警告应该被提及。Item-26 告诉你不要使用原始类型，Item-29 告诉你要优先使用泛型，Item-30 告诉你要优先使用泛型方法。在使用 lambda 表达式时，这些建议尤其重要，因为编译器获得了允许它从泛型中执行类型推断的大部分类型信息。如果不提供此信息，编译器将无法进行类型推断，并且必须在 lambda 表达式中手动指定类型，这将大大增加它们的冗长。举例来说，如果变量声明为原始类型 List 而不是参数化类型 `List<String>`，那么上面的代码片段将无法编译。

顺便说一下，如果使用 comparator 构造方法代替 lambda 表达式（Item-14），那么代码片段可以变得更加简洁：

```
Collections.sort(words, comparingInt(String::length));
```

事实上，通过 Java 8 中添加到 List 接口的 sort 方法，可以使代码片段变得更短：

```
words.sort(comparingInt(String::length));
```

在语言中添加 lambda 表达式使得在以前没有意义的地方使用函数对象变得实际。例如，考虑 Item-34 中的操作枚举类型。因为每个枚举的 apply 方法需要不同的行为，所以我们使用特定于常量的类体并覆盖每个枚举常量中的 apply 方法。为了唤醒你的记忆，以下是代码：

```
// 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 指出，枚举实例字段比特定于常量的类体更可取。Lambda 表达式使得使用前者取代后者来实现特定于常量的行为变得容易。只需将实现每个枚举常量的行为的 lambda 表达式传递给它的构造函数。构造函数将 lambda 表达式存储在实例字段中，apply 方法将调用转发给 lambda 表达式。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);
        }
    }
```

注意，我们对表示枚举常量行为的 lambda 表达式使用了 DoubleBinaryOperator 接口。这是 `java.util.function` （Item-44）中许多预定义的函数式接口之一。它表示一个函数，该函数接收两个 double 类型参数，并且返回值也为 double 类型。

查看基于 lambda 表达式的操作 enum，你可能会认为特定于常量的方法体已经过时了，但事实并非如此。与方法和类不同，**lambda 表达式缺少名称和文档；如果一个算法并非不言自明，或者有很多行代码，不要把它放在 lambda 表达式中。** 一行是理想的，三行是合理的最大值。如果你违反了这一规则，就会严重损害程序的可读性。如果 lambda 表达式很长或者很难读，要么找到一种方法来简化它，要么重构你的程序。此外，传递给 enum 构造函数的参数在静态上下文中计算。因此，enum 构造函数中的 lambda 表达式不能访问枚举的实例成员。如果枚举类型具有难以理解的特定于常量的行为，无法在几行代码中实现，或者需要访问实例字段或方法，则仍然需要特定于常量的类。

同样，你可能认为匿名类在 lambda 表达式时代已经过时了。这更接近事实，但是有一些匿名类可以做的事情是 lambda 表达式不能做的。Lambda 表达式仅限于函数式接口。如果想创建抽象类的实例，可以使用匿名类，但不能使用 lambda 表达式。类似地，你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 表达式无法获得对自身的引用。在 lambda 表达式中，this 关键字指的是封闭实例，这通常是你想要的。在匿名类中，this 关键字引用匿名类实例。如果你需要从函数对象的内部访问它，那么你必须使用一个匿名类。

Lambda 表达式与匿名类共享无法通过实现可靠地序列化和反序列化它们的属性。因此，**很少（如果有的话）序列化 lambda**（或匿名类实例）。如果你有一个想要序列化的函数对象，比如比较器，那么使用私有静态嵌套类的实例（Item-24）。

总之，在 Java 8 中，lambda 表达式是迄今为止表示小函数对象的最佳方式。**不要对函数对象使用匿名类，除非你必须创建非函数式接口类型的实例。** 另外，请记住，lambda 表达式使表示小函数对象变得非常容易，从而为 Java 以前不实用的函数式编程技术打开了大门。

Item 43: Prefer method references to lambdas（方法引用优于 λ 表达式）

lambda 表达式与匿名类相比，主要优势是更简洁。Java 提供了一种方法来生成比 lambda 表达式更简洁的函数对象：方法引用。下面是一个程序的代码片段，该程序维护从任意键到 Integer 类型值的映射。如果该值被解释为键实例数的计数，那么该程序就是一个多集实现。该代码段的功能是，如果数字 1 不在映射中，则将其与键关联，如果键已经存在，则将关联值递增：

```
map.merge(key, 1, (count, incr) -> count + incr);
```

注意，这段代码使用了 merge 方法，它是在 Java 8 中添加到 Map 接口的。如果给定键没有映射，则该方法只插入给定的值；如果已经存在映射，则 merge 将给定的函数应用于当前值和给定值，并用结果覆盖当前值。这段代码代表了 merge 方法的一个典型用例。

代码读起来不错，但是仍然有一些刻板。参数计数和 incr 不会增加太多的价值，而且它们会占用相当大的空间。实际上，lambda 表达式告诉你的是函数返回两个参数的和。在 Java 8 中，Integer（和所有其他基本类型的包装类）提供了一个静态方法 sum，它的作用完全相同。我们可以简单地传递一个引用到这个方法，并得到相同的结果，同时减少视觉混乱：

```
map.merge(key, 1, Integer::sum);
```

一个方法的参数越多，就可以通过一个方法引用消除越多的刻板模式。然而，在某些 lambda 表达式中，您选择的参数名提供了有用的文档，使得 lambda 表达式比方法引用更易于阅读和维护，即使 lambda 表达式更长。

对于方法引用，没有什么是你不能对 lambda 表达式做的（只有一个模糊的例外，如果你好奇的话可参见 [JLS, 9.9-2]）。也就是说，方法引用通常会产生更短、更清晰的代码。如果 lambda 表达式太长或太复杂，它们还会给出一个输出：可以将代码从 lambda 表达式提取到一个新方法中，并以对该方法的引用替换 lambda 表达式。可以为该方法起一个好名字，并将其文档化以满足需要。

如果你使用 IDE 编程，它将在任何可能的地方建议用方法引用替换 lambda 表达式。通常应该（但不总是）接受 IDE 的建议。有时候，lambda 表达式会比方法引用更简洁。当方法与 lambda 表达式在同一个类中时，这种情况最常见。例如，考虑这段代码片段，它假定发生在一个名为 GoshThisClassNameIsHumongous 的类中：

```
service.execute(GoshThisClassNameIsHumongous::action);
```

使用 lambda 表达式是这样的：

```
service.execute(() -> action());
```

使用方法引用的代码片段并不比使用 lambda 表达式的代码片段短，也不清楚，所以选择后者。类似地，函数接口提供了一个通用静态工厂方法来返回标识函数 `Function.identity()`。不使用这个方法，而是一行中编写等价的 lambda 表达式：`x -> x`，通常更短，也更简洁。

许多方法引用引用静态方法，但是有四种方法不引用静态方法。其中两个是绑定和非绑定实例方法引用。在绑定引用中，接收对象在方法引用中指定。绑定引用在本质上与静态引用相似：函数对象接受与引用方法相同的参数。在未绑定引用中，在应用函数对象时通过方法声明参数之前的附加参数指定接收对象。在流管道中，未绑定引用通常用作映射和筛选函数（Item-45）。最后，对于类和数组，有两种构造函数引用。构造函数引用用作工厂对象。五种方法参考文献汇总如下表：

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str ->`
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]`

总之，方法引用通常为 lambda 表达式提供了一种更简洁的选择。**如果方法引用更短、更清晰，则使用它们；如果没有，仍然使用 lambda 表达式。**

Item 44: Favor the use of standard functional interfaces（优先使用标准函数式接口）

现在 Java 已经有了 lambda 表达式，编写 API 的最佳实践已经发生了很大的变化。例如，模板方法模式 [Gamma95]，其中子类覆盖基类方法以专门化其超类的行为，就没有那么有吸引力了。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，你将编写更多以函数对象为参数的构造函数和方法。选择正确的函数参数类型需要谨慎。

考虑 LinkedHashMap。你可以通过覆盖受保护的 removeEldestEntry 方法将该类用作缓存，每当向映射添加新键时，put 都会调用该方法。当该方法返回 true 时，映射将删除传递给该方法的最老条目。下面的覆盖允许映射增长到 100 个条目，然后在每次添加新键时删除最老的条目，维护 100 个最近的条目：

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return size() > 100;
}
```

这种技术工作得很好，但是使用 lambda 表达式可以做得更好。如果 LinkedHashMap 是现在编写的，它将有一个静态工厂或构造函数，它接受一个函数对象。看着 removeEldestEntry 的定义,你可能会认为这个函数对象应该 `Map.Entry<K,V>` 和返回一个布尔值，但不会完全做到：removeEldestEntry 方法调用 `size()` 地图中的条目的数量，这工作，因为 removeEldestEntry 在 Map 上是一个实例方法。传递给构造函数的函数对象不是 Map 上的实例方法，无法捕获它，因为在调用 Map 的工厂或构造函数时，Map 还不存在。因此，Map 必须将自身传递给函数对象，函数对象因此必须在输入端及其最老的条目

上接受 Map。如果要声明这样一个函数式接口，它看起来是这样的：

```
// Unnecessary functional interface; use a standard one instead.
@FunctionalInterface interface EldestEntryRemovalFunction<K,V>{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

这个接口可以很好地工作，但是你不应该使用它，因为你不需要为此声明一个新接口。`java.util.function` 包提供了大量的标准函数接口供你使用。**如果一个标准的函数式接口可以完成这项工作，那么你通常应该优先使用它，而不是使用专门构建的函数式接口。** 通过减少 API 的概念表面积，这将使你的 API 更容易学习，并将提供显著的互操作性优势，因为许多标准函数式接口提供了有用的默认方法。例如，Predicate 接口提供了组合谓词的方法。在我们的 LinkedHashMap 示例中，应该优先使用标准的 `BiPredicate<Map<K,V>` 、`Map.Entry<K,V>>` 接口，而不是定制的 EldestEntryRemovalFunction 接口。

译注：原文笔误，应为 `java.util.function`

`java.util.function` 中有 43 个接口。不能期望你记住所有的接口，但是如果你记住了 6 个基本接口，那么你可以在需要时派生出其余的接口。基本接口操作对象引用类型。Operator 接口表示结果和参数类型相同的函数。Predicate 接口表示接受参数并返回布尔值的函数。Function 接口表示参数和返回类型不同的函数。Supplier 接口表示一个不接受参数并返回（或「供应」）值的函数。最后，Consumer 表示一个函数，该函数接受一个参数，但不返回任何内容，本质上是使用它的参数。六个基本的函数式接口总结如下：

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

还有 6 个基本接口的 3 个变体，用于操作基本类型 int、long 和 double。它们的名称是通过在基本接口前面加上基本类型前缀而派生出来的。例如，一个接受 int 的 Predicate 就是一个 IntPredicate，一个接受两个 long 值并返回一个 long 的二元操作符就是一个 LongBinaryOperator。除了由返回类型参数化的函数变量外，这些变量类型都不是参数化的。例如，`LongFunction<int[]>` 使用 long 并返回一个

int[]。

Function 接口还有 9 个额外的变体，在结果类型为基本数据类型时使用。源类型和结果类型总是不同的，因为不同类型的函数本身都是 UnaryOperator。如果源类型和结果类型都是基本数据类型，则使用带有 SrcToResult 的前缀函数，例如 LongToIntFunction（六个变体）。如果源是一个基本数据类型，而结果是一个对象引用，则使用带前缀 `<Src>ToObj` 的 Function 接口，例如 DoubleToObjFunction（三个变体）。

三个基本函数式接口有两个参数版本，使用它们是有意义的：`BiPredicate<T,U>`、`BiFunction<T,U,R>`、`BiConsumer<T,U>`。也有 BiFunction 变体返回三个相关的基本类型：`ToIntBiFunction<T,U>`、`ToLongBiFunction<T,U>`、`ToDoubleBiFunction<T,U>`。Consumer 有两个参数变体，它们接受一个对象引用和一个基本类型：`ObjDoubleConsumer<T>`、`ObjIntConsumer<T>`、`ObjLongConsumer<T>`。总共有9个基本接口的双参数版本。

最后是 BooleanSupplier 接口，它是 Supplier 的一个变体，返回布尔值。这是在任何标准函数接口名称中唯一显式提到布尔类型的地方，但是通过 Predicate 及其四种变体形式支持布尔返回值。前面描述的 BooleanSupplier 接口和 42 个接口占了全部 43 个标准函数式接口。不可否认，这有很多东西需要消化，而且不是非常直观。另一方面，你将需要的大部分函数式接口都是为你编写的，并且它们的名称足够常规，因此在需要时你应该不会遇到太多麻烦。

大多数标准函数式接口的存在只是为了提供对基本类型的支持。**不要尝试使用带有包装类的基本函数式接口，而不是使用基本类型函数式接口。** 当它工作时，它违反了 Item-61 的建议，“与盒装原语相比，更喜欢原语类型”。在批量操作中使用装箱原语的性能后果可能是致命的。

现在你知道，与编写自己的接口相比，通常应该使用标准的函数式接口。但是你应该什么时候写你自己的呢？当然，如果标准的函数式接口都不能满足你的需要，那么你需要自行编写，例如，如果你需要一个接受三个参数的 Predicate，或者一个抛出已检查异常的 Predicate。但是有时候你应该编写自己的函数接口，即使其中一个标准接口在结构上是相同的。

考虑我们的老朋友 `Comparator<T>`，它在结构上与 `ToIntBiFunction<T,T>` 接口相同。即使后者接口在将前者添加到库时已经存在，使用它也是错误的。有几个原因说明比较器应该有自己的接口。首先，每次在 API 中使用 Comparator 时，它的名称都提供了优秀的文档，而且它的使用非常频繁。通过实现接口，你保证遵守其契约。第三，该接口大量配备了用于转换和组合比较器的有用默认方法。

如果你需要与 Comparator 共享以下一个或多个特性的函数式接口，那么你应该认真考虑编写一个专用的函数式接口，而不是使用标准接口：

- 它将被广泛使用，并且可以从描述性名称中获益。

- 它有一个强有力的约定。
- 它将受益于自定义默认方法。

如果你选择编写自己的函数式接口，请记住这是一个接口，因此应该非常小心地设计它（Item-21）。

注意 EldestEntryRemovalFunction 接口(第199页)使用 `@FunctionalInterface` 注释进行标记。这种注释类型在本质上类似于 `@Override`。它是程序员意图的声明，有三个目的：它告诉类及其文档的读者，接口的设计是为了启用 lambda 表达式；它使你保持诚实，因为接口不会编译，除非它只有一个抽象方法；它还可以防止维护者在接口发展过程中意外地向接口添加抽象方法。**总是用 `@FunctionalInterface` 注释你的函数接口。**

最后一点应该是关于 API 中函数式接口的使用。不要提供具有多个重载的方法，这些方法采用相同参数位置的不同函数式接口，否则会在客户机中造成可能的歧义。这不仅仅是一个理论问题。ExecutorService 的 submit 方法可以是 `Callable<T>` 级的，也可以是 Runnable 的，并且可以编写一个客户端程序，它需要一个类型转换来指示正确的重载(Item 52)。避免此问题的最简单方法是不要编写将不同函数式接口放在相同参数位置的重载。这是 Item-52 「明智地使用过载」建议的一个特例。

总之，既然 Java 已经有了 lambda 表达式，你必须在设计 API 时考虑 lambda 表达式。在输入时接受函数式接口类型，在输出时返回它们。一般情况下，最好使用 `java.util.function` 中提供的标准函数式接口，但请注意比较少见的一些情况，在这种情况下，你最好编写自己的函数式接口。

Item 45: Use streams judiciously（明智地使用流）

在 Java 8 中添加了流 API，以简化序列或并行执行批量操作的任务。这个 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）和流管道（表示对这些元素的多阶段计算）。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、正则表达式的 Pattern 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的引用或基本数据类型。支持三种基本数据类型：int、long 和 double。

流管道由源流、零个或多个 Intermediate 操作和一个 Terminal 操作组成。每个 Intermediate 操作以某种方式转换流，例如将每个元素映射到该元素的一个函数，或者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，其元素类型可能与输入流相同，也可能与输入流不同。Terminal 操作对最后一次 Intermediate 操作所产生的流进行最终计算，例如将其元素存储到集合中、返回特定元素、或打印其所有元素。

流管道的计算是惰性的：直到调用 Terminal 操作时才开始计算，并且对完成 Terminal 操作不需要的数据元素永远不会计算。这种惰性的求值机制使得处理无限流成为可能。请注意，没有 Terminal 操作的流管道是无动作的，因此不要忘记包含一个 Terminal 操作。

流 API 是流畅的：它被设计成允许使用链式调用将组成管道的所有调用写到单个表达式中。实际上，可以将多个管道链接到一个表达式中。

默认情况下，流管道按顺序运行。让管道并行执行与在管道中的任何流上调用并行方法一样简单，但是这样做不一定合适（Item-48）。

流 API 非常通用，实际上任何计算都可以使用流来执行，但这并不意味着你就应该这样做。如果使用得当，流可以使程序更短、更清晰；如果使用不当，它们会使程序难以读取和维护。对于何时使用流没有硬性的规则，但是有一些启发式的规则。

考虑下面的程序，它从字典文件中读取单词并打印所有大小满足用户指定最小值的变位组。回想一下，如果两个单词以不同的顺序由相同的字母组成，那么它们就是字谜。该程序从用户指定的字典文件中读取每个单词，并将这些单词放入一个 Map 中。Map 的键是按字母顺序排列的单词，因此「staple」的键是「aelpst」，而「petals」的键也是「aelpst」：这两个单词是字谜，所有的字谜都有相同的字母排列形式（有时称为字母图）。Map 的值是一个列表，其中包含共享按字母顺序排列的表单的所有单词。在字典被处理之后，每个列表都是一个完整的字谜组。然后，该程序遍历 Map 的 values() 视图，并打印大小满足阈值的每个列表：

```
// 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);
        }
    }
```

这个程序中的一个步骤值得注意。将每个单词插入到 Map 中（以粗体显示）使用 computeIfAbsent 方法，该方法是在 Java 8 中添加的。此方法在 Map 中查找键：如果键存在，则该方法仅返回与其关联的值。若不存在，则该方法通过将给定的函数对象应用于键来计算一个值，将该值与键关联，并返回计算的值。computeIfAbsent 方法简化了将多个值与每个键关联的 Map 的实现。

现在考虑下面的程序，它解决了相同的问题，但是大量使用了流。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，该语句确保字典文件是关闭的：

```
    // 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);
            }
        }
    }
```

如果你发现这段代码难以阅读，不要担心；不单是你有这样的感觉。它虽然更短，但可读性也更差，特别是对于不擅长流使用的程序员来说。过度使用流会使得程序难以读取和维护。幸运的是，有一个折衷的办法。下面的程序解决了相同的问题，在不过度使用流的情况下使用流。结果，这个程序比原来的程序更短，也更清晰：

```
    // 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
    }
```

即使你以前很少接触流，这个程序也不难理解。它在带有资源的 try 块中打开字典文件，获得由文件中所有行组成的流。流变量名为 words，表示流中的每个元素都是一个单词。此流上的管道没有 Intermediate 操作；它的 Terminal 操作将所有单词收集到一个 Map 中，该 Map 按字母顺序将单词分组（Item-46）。这与在程序的前两个版本中构造的 Map 完全相同。然后在 Map 的 values() 视图上打开一个新的 `Stream<List<String>>`。这个流中的元素当然是字谜组。对流进行过滤，以便忽略所有大小小于 minGroupSize 的组，最后，Terminal 操作 forEach 打印其余组。

注意，lambda 表达式参数名称是经过仔细选择的。参数 g 实际上应该命名为 group，但是生成的代码行对于本书排版来说太宽了。**在没有显式类型的情况下，lambda 表达式参数的谨慎命名对于流管道的可读性至关重要。**

还要注意，单词的字母化是在一个单独的字母化方法中完成的。这通过为操作提供一个名称并将实现细节排除在主程序之外，从而增强了可读性。**在流管道中使用 helper 方法比在迭代代码中更重要**，因为管道缺少显式类型信息和命名的临时变量。

本来可以重新实现字母顺序方法来使用流，但是基于流的字母顺序方法就不那么清晰了，更难于正确地编写，而且可能会更慢。这些缺陷是由于 Java 不支持基本字符流（这并不意味着 Java 应该支持字符流；这样做是不可行的）。要演示使用流处理 char 值的危害，请考虑以下代码：

```
    "Hello world!".chars().forEach(System.out::print);
```

你可能希望它打印 Hello world!，但如果运行它，你会发现它打印 721011081081113211911111410810033。这是因为 `"Hello world!".chars()` 返回的流元素不是 char 值，而是 int 值，因此调用了 print 的 int 重载。无可否认，一个名为 chars 的方法返回一个 int 值流是令人困惑的。你可以通过强制调用正确的重载来修复程序：

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

但理想情况下，你应该避免使用流来处理 char 值。当你开始使用流时，你可能会有将所有循环转换为流的冲动，但是要抵制这种冲动。虽然这是可实施的，但它可能会损害代码库的可读性和可维护性。通常，即使是中等复杂的任务，也最好使用流和迭代的某种组合来完成，如上面的 Anagrams 程序所示。因此，**重构现有代码或是在新代码中，都应该在合适的场景使用流。**

如本项中的程序所示，流管道使用函数对象（通常是 lambda 表达式或方法引用）表示重复计算，而迭代代码使用代码块表示重复计算。有些事情你可以对代码块做，而你不能对函数对象做：

从代码块中，可以读取或修改作用域中的任何局部变量；在 lambda 表达式中，只能读取 final 或有效的 final 变量 [JLS 4.12.4]，不能修改任何局部变量。

- 从代码块中，可以从封闭方法返回、中断或继续封闭循环，或抛出声明要抛出的任何已检查异常；在 lambda 表达式中，你不能做这些事情。

如果使用这些技术最好地表达计算，那么它可能不适合流。相反，流使做一些事情变得非常容易：

- 元素序列的一致变换
- 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）
- 将元素序列累积到一个集合中，可能是按某个公共属性对它们进行分组
- 在元素序列中搜索满足某些条件的元素

如果使用这些技术能够最好地表达计算，那么它就是流的一个很好的使用场景。

使用流很难做到的一件事是从管道的多个阶段同时访问相应的元素：一旦你将一个值映射到另一个值，原始值就会丢失。一个解决方案是将每个值映射到包含原始值和新值的 pair 对象，但这不是一个令人满意的解决方案，特别是如果管道的多个阶段都需要 pair 对象的话。生成的代码混乱而冗长，这违背了流的主要目的。当它适用时，更好的解决方案是在需要访问早期阶段值时反转映射。

例如，让我们编写一个程序来打印前 20 个 Mersenne 素数。刷新你的记忆,一个 Mersenne 素数的数量是一个数字形式 2p – 1。如果 p 是素数，对应的 Mersenne 数可以是素数；如果是的话，这就是 Mersenne 素数。作为管道中的初始流，我们需要所有质数。这里有一个返回（无限）流的方法。我们假设已经使用静态导入来方便地访问 BigInteger 的静态成员：

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

方法的名称（素数）是描述流元素的复数名词。强烈推荐所有返回流的方法使用这种命名约定，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：流中的第一个元素和一个函数，用于从前一个元素生成流中的下一个元素。下面是打印前 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);
    }
```

这个程序是上述散文描述的一个简单编码：它从素数开始，计算相应的 Mersenne 数，过滤掉除素数以外的所有素数（魔法数字 50 控制概率素数测试），将结果流限制为 20 个元素，并打印出来。

现在假设我们想要在每个 Mersenne 素数之前加上它的指数 (p)，这个值只在初始流中存在，因此在输出结果的终端操作中是不可访问的。幸运的是，通过对第一个中间操作中发生的映射求逆，可以很容易地计算出 Mersenne 数的指数。指数仅仅是二进制表示的比特数，因此这个终端操作产生了想要的结果：

```
    .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

在许多任务中，使用流还是迭代并不明显。例如，考虑初始化一副新纸牌的任务。假设 Card 是一个不可变的值类，它封装了 Rank 和 Suit，它们都是 enum 类型。此任务代表需要计算可从两个集合中选择的所有元素对的任何任务。数学家称之为这两个集合的笛卡尔积。下面是一个嵌套 for-each 循环的迭代实现，你应该非常熟悉它：

```
    // 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;
    }
```

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到单个流中（或将其扁平化）。注意，这个实现包含一个嵌套 lambda 表达式，用粗体显示：

```
// 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());
}
```

两个版本的 newDeck 哪个更好？这可以归结为个人偏好和编程环境。第一个版本更简单，可能感觉更自然。大部分 Java 程序员将能够理解和维护它，但是一些程序员将对第二个（基于流的）版本感到更舒服。如果你相当精通流和函数式编程，那么它会更简洁，也不会太难理解。如果你不确定你更喜欢哪个版本，迭代版本可能是更安全的选择。如果你更喜欢流版本，并且相信与代码一起工作的其他程序员也会分享你的偏好，那么你应该使用它。

总之，有些任务最好使用流来完成，有些任务最好使用迭代来完成。许多任务最好通过结合这两种方法来完成。对于选择任务使用哪种方法，没有硬性的规则，但是有一些有用的启发。在许多情况下，使用哪种方法是清楚的；在某些情况下很难决定。如果你不确定一个任务是通过流还是通过迭代更好地完成，那么同时尝试这两种方法，看看哪一种更有效。

Item 46: Prefer side-effect-free functions in streams（在流中使用无副作用的函数）

如果你是流的新手，可能很难掌握它们。仅仅将计算表示为流管道是困难的。当你成功时，你的程序可以运行，但你可能意识不到什么好处。流不仅仅是一个 API，它是一个基于函数式编程的范式。为了获得流提供的可表达性、速度以及在某些情况下的并行性，你必须采纳范式和 API。

流范式中最重要的部分是将计算构造为一系列转换，其中每个阶段的结果都尽可能地接近上一阶段结果的纯函数。纯函数的结果只依赖于它的输入：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，传递到流操作（包括 Intermediate 操作和 Terminal 操作）中的任何函数对象都应该没有副作用。

译注：流的操作类型分为以下几种：

1、Intermediate

- 一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是惰性化的（lazy），就是说，仅仅调用到这类方法，并没有真正开始流的遍历。常见的操作：map（mapToInt、flatMap 等）、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered

2、Terminal

- 一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用「光」了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果，或者一个 side effect。常见的操作：forEach、forEachOrdered、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst、findAny、iterator
- 在对于一个流进行多次转换操作 (Intermediate 操作)，每次都对流的每个元素进行转换，而且是执行多次，这样时间复杂度就是 N（转换次数）个 for 循环里把所有操作都做掉的总和吗？其实不是这样的，转换操作都是 lazy 的，多个转换操作只会在 Terminal 操作的时候融合起来，一次循环完成。我们可以这样简单的理解，流里有个操作函数的集合，每次转换操作就是把转换函数放入这个集合中，在 Terminal 操作的时候循环流对应的集合，然后对每个元素执行所有的函数。

3、short-circuiting

- 对于一个 intermediate 操作，如果它接受的是一个无限大（infinite/unbounded）的流，但返回一个有限的新流。
- 对于一个 terminal 操作，如果它接受的是一个无限大的流，但能在有限的时间计算出结果。当操作一个无限大的流，而又希望在有限时间内完成操作，则在管道内拥有一个 short-circuiting 操作是必要非充分条件。常见的操作：anyMatch、allMatch、 noneMatch、findFirst、findAny、limit

偶尔，你可能会看到如下使用流的代码片段，它用于构建文本文件中单词的频率表：

```
// 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);
    });
}
```

这段代码有什么问题？毕竟，它使用了流、lambda 表达式和方法引用，并得到了正确的答案。简单地说，它根本不是流代码，而是伪装成流代码的迭代代码。它没有从流 API 中获得任何好处，而且它（稍微）比相应的迭代代码更长、更难于阅读和更难以维护。这个问题源于这样一个事实：这段代码在一个 Terminal 操作中（forEach）执行它的所有工作，使用一个会改变外部状态的 lambda 表达式（频率表）。forEach 操作除了显示流执行的计算结果之外，还会执行其他操作，这是一种「代码中的不良习惯」，就像 lambda 表达式会改变状态一样。那么这段代码应该是什么样的呢？

```
// 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()));
}
```

这个代码片段与前面的代码片段做了相同的事情，但是正确地使用了流 API。它更短更清晰。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，并且与 forEach 操作是类似的。但是 forEach 操作是 Terminal 操作中功能最弱的操作之一，对流最不友好。它是显式迭代的，因此不适合并行化。**forEach 操作应该只用于报告流计算的结果，而不是执行计算。** 有时候，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现有集合中。

改进后的代码使用了 collector，这是使用流必须学习的新概念。Collectors 的 API 令人生畏：它有 39 个方法，其中一些方法有多达 5 个类型参数。好消息是，你可以从这个 API 中获得大部分好处，而不必深入研究它的全部复杂性。对于初学者，可以忽略 Collector 接口，将 collector 视为封装了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。collector 生成的对象通常是一个集合（这也解释了为何命名为 collector）。

将流的元素收集到一个真正的 Collection 中的 collector 非常简单。这样的 collector 有三种：`toList()`、`toSet()` 和 `toCollection(collectionFactory)`。它们分别返回 List、Set 和程序员指定的集合类型。有了这些知识，我们就可以编写一个流管道来从 freq 表中提取前 10 个元素来构成一个新 List。

```
// 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());
```

注意，我们还没有用它的类 Collectors 对 toList 方法进行限定。**静态导入 Collectors 的所有成员是习惯用法，也是明智的，因为这使流管道更具可读性。**

这段代码中唯一棘手的部分是我们传递给 sorted 的 `comparing(freq::get).reversed()`。comparing 方法是 comparator 的一种构造方法（Item-14），它具有键提取功能。函数接受一个单词，而「提取」实际上是一个表查找：绑定方法引用 `freq::get` 在 freq 表中查找该单词，并返回该单词在文件中出现的次数。最后，我们在比较器上调用 reverse 函数，我们将单词从最频繁排序到最不频繁进行排序。然后，将流限制为 10 个单词并将它们收集到一个列表中。

前面的代码片段使用 Scanner 的流方法在扫描器上获取流。这个方法是在 Java 9 中添加的。如果使用的是较早的版本，则可以使用类似于 Item-47（`streamOf(Iterable<E>)`）中的适配器将实现 Iterator 的扫描程序转换为流。

那么 Collectors 中的其他 36 个方法呢？它们中的大多数都允许你将流收集到 Map 中，这比将它们收集到真正的集合要复杂得多。每个流元素与一个键和一个值相关联，多个流元素可以与同一个键相关联。

最简单的 Map 收集器是 `toMap(keyMapper，valueMapper)`，它接受两个函数，一个将流元素映射到键，另一个映射到值。我们在 Item-34 中的 fromString 实现中使用了这个收集器来创建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));
```

如果流中的每个元素映射到唯一的键，那么这种简单的 toMap 形式就是完美的。如果多个流元素映射到同一个键，管道将以 IllegalStateException 结束。

toMap 更为复杂的形式，以及 groupingBy 方法，提供了各种方法来提供处理此类冲突的策略。一种方法是为 toMap 方法提供一个 merge 函数，以及它的键和值映射器。merge 函数是一个 `BinaryOperator<V>`，其中 V 是 Map 的值类型。与键关联的任何附加值都将使用 merge 函数与现有值组合，因此，例如，如果 merge 函数是乘法，那么你将得到一个值，该值是 value mapper 与键关联的所有值的乘积。

toMap 的三参数形式对于从键到与该键关联的所选元素的映射也很有用。例如，假设我们有一个由不同艺术家录制的唱片流，并且我们想要一个从唱片艺术家到畅销唱片的映射。这个 collector 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)
    )
));
```

注意，比较器使用静态工厂方法 maxBy，该方法从 BinaryOperator 静态导入。此方法将 `Comparator<T>` 转换为 `BinaryOperator<T>`，该操作符计算指定比较器所隐含的最大值。在这种情况下，比较器是通过比较器构造方法返回的，比较器构造方法取 `Album::sales`。这看起来有点复杂，但是代码可读性很好。粗略地说，代码是这样描述的：「将专辑流转换为 Map，将每个艺人映射到销量最好的专辑。」这与问题的文字陈述惊人地接近。

toMap 的三参数形式的另一个用途是生成一个 collector，当发生冲突时，它强制执行 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是相同的，或者它们都是可接受的，那么这个 collector 的行为可能正是你想要的：

```
// Collector to impose last-write-wins policy
toMap(keyMapper, valueMapper, (v1, v2) -> v2)
```

toMap 的第三个也是最后一个版本采用了第四个参数，这是一个 Map 工厂，当你想要指定一个特定的 Map 实现（如 EnumMap 或 TreeMap）时，可以使用它。

还有前三个版本的 toMap 的变体形式，名为 toConcurrentMap，它们可以有效地并行运行，同时生成 ConcurrentHashMap 实例。

除了 toMap 方法之外，collector API 还提供 groupingBy 方法，该方法返回 collector，以生成基于分类器函数将元素分组为类别的映射。分类器函数接受一个元素并返回它所属的类别。这个类别用作元素的 Map 键。groupingBy 方法的最简单版本只接受一个分类器并返回一个 Map，其值是每个类别中所有元素的列表。这是我们在 Item-45 的字谜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映射：

```
words.collect(groupingBy(word -> alphabetize(word)))
```

如果你希望 groupingBy 返回一个使用列表之外的值生成映射的收集器，你可以指定一个下游收集器和一个分类器。下游收集器从包含类别中的所有元素的流中生成一个值。这个参数最简单的用法是传递 toSet()，这会生成一个 Map，其值是 Set，而不是 List。

或者，你可以传递 `toCollection(collectionFactory)`，它允许你创建集合，将每个类别的元素放入其中。这使你可以灵活地选择所需的任何集合类型。groupingBy 的两参数形式的另一个简单用法是将 `counting()` 作为下游收集器传递。这将生成一个 Map，该 Map 将每个类别与类别中的元素数量相关联，而不是包含元素的集合。这是你在这一项开始的 freq 表例子中看到的：

```
Map<String, Long> freq = words.collect(groupingBy(String::toLowerCase,
counting()));
```

groupingBy 的第三个版本允许你指定除了下游收集器之外的 Map 工厂。注意，这个方法违反了标准的可伸缩参数列表模式：mapFactory 参数位于下游参数之前，而不是之后。groupingBy 的这个版本允许你控制包含的 Map 和包含的集合，因此，例如，你可以指定一个收集器，该收集器返回一个 TreeMap，其值为 TreeSet。

groupingByConcurrent 方法提供了 groupingBy 的所有三种重载的变体。这些变体可以有效地并行运行，并生成 ConcurrentHashMap 实例。还有一个与 groupingBy 关系不大的词，叫做 partitioningBy 。代替分类器方法，它接受一个 Predicate 并返回一个键为布尔值的 Map。此方法有两个重载，其中一个除了 Predicate 外还接受下游收集器。

计数方法返回的收集器仅用于作为下游收集器。相同的功能可以通过 count 方法直接在流上使用，**所以永远没有理由说 `collect(counting())`。** 还有 15 个具有此属性的收集器方法。它们包括 9 个方法，它们的名称以求和、平均和汇总开头（它们的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，以及过滤、映射、平面映射和 collectingAndThen 方法。大多数程序员可以安全地忽略这些方法中的大多数。从设计的角度来看，这些收集器试图部分复制收集器中的流的功能，以便下游收集器可以充当「迷你存储器」。

我们还没有提到三种 Collectors 方法。虽然它们是在 Collectors 中，但它们不涉及收集。前两个是 minBy 和 maxBy，它们接受 comparator 并返回由 comparator 确定的流中的最小或最大元素。它们是流接口中最小和最大方法的一些小泛化，是 BinaryOperator 中同名方法返回的二进制操作符的 collector 类似物。回想一下，在我们最畅销的专辑示例中，我们使用了 `BinaryOperator.maxBy`。

最后一个 Collectors 方法是 join，它只对 CharSequence 实例流（如字符串）执行操作。在其无参数形式中，它返回一个收集器，该收集器只是将元素连接起来。它的一个参数形式接受一个名为 delimiter 的 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，收集器将返回逗号分隔的值字符串（但是要注意，如果流中的任何元素包含逗号，该字符串将是不明确的）。除了分隔符外，三参数形式还接受前缀和后缀。生成的收集器生成的字符串与打印集合时得到的字符串类似，例如 `[came, saw, conquer]`。

总之，流管道编程的本质是无副作用的函数对象。这适用于传递给流和相关对象的所有函数对象。Terminal 操作 forEach 只应用于报告由流执行的计算结果，而不应用于执行计算。为了正确使用流，你必须了解 collector。最重要的 collector 工厂是 toList、toSet、toMap、groupingBy 和 join。

Item 47: Prefer Collection to Stream as a return type（优先选择 Collection 而不是流作为返回类型）

许多方法都返回元素序列。在 Java 8 之前，此类方法常见的返回类型是 Collection 集合接口，如 Set 和 List，另外还有 Iterable 以及数组类型。通常，很容易决定使用哪一种类型。标准是一个集合接口。如果方法的存在仅仅是为了支持 for-each 循环，或者无法使返回的序列实现某个集合方法（通常是 `contains(Object)`），则使用 Iterable 接口。如果返回的元素是基本数据类型或有严格的性能要求，则使用数组。在 Java 8 中，流被添加进来，这大大增加了为序列返回方法选择适当返回类型的复杂性。

你可能听说现在流是返回元素序列的明显选择，但是正如 Item-45 中所讨论的，流不会让迭代过时：编写好的代码需要明智地将流和迭代结合起来。如果一个 API 只返回一个流，而一些用户希望使用 for-each 循环遍历返回的序列，那么这些用户将会感到不适。这尤其令人沮丧，因为流接口包含 Iterable 接口中惟一的抽象方法，而且流对该方法的规范与 Iterable 的规范兼容。唯一阻止程序员使用 for-each 循环在流上迭代的是流不能扩展 Iterable。

遗憾的是，这个问题没有好的解决办法。乍一看，似乎将方法引用传递给流的 iterator 方法是可行的。生成的代码可能有点繁琐，不易理解，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
}
```

不幸的是，如果你试图编译这段代码，你会得到一个错误消息：

```
Test.java:6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
^
```

为了编译代码，你必须将方法引用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)
```

这个客户端代码可以工作，但是它太过繁琐并不易理解，无法在实践中使用。更好的解决方案是使用适配器方法。JDK 没有提供这样的方法，但是使用上面代码片段中使用的内联技术编写方法很容易。注意，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工作正常：

```
// Adapter from Stream<E> to Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}
```

使用此适配器，你可以使用 for-each 语句遍历任何流：

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
    // Process the process
}
```

注意，Item-34 中 Anagrams 程序的流版本使用 `Files.lines` 读取字典，而迭代版本使用扫描器。`Files.lines` 方法优于扫描器，扫描器在读取文件时静默地接收任何异常。理想情况下，我们在 `Files.lines` 的迭代版本也应该如此。如果一个 API 只提供对一个序列的流访问，而程序员希望用 for-each 语句遍历该序列，那么这是程序员会做出的一种妥协。

相反，如果程序员希望使用流管道来处理序列，那么只提供可迭代的 API 就会有理由让他心烦。JDK 同样没有提供适配器，但是编写适配器非常简单：

```
// Adapter from Iterable<E> to Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

如果你正在编写一个返回对象序列的方法，并且你知道它只会在流管道中使用，那么你当然应该可以随意返回流。类似地，返回仅用于迭代的序列的方法应该返回一个 Iterable。但是如果你写一个公共 API，它返回一个序列，你应该兼顾想写流管道以及想写 for-each 语句的用户，除非你有充分的理由相信大多数用户想要使用相同的机制。

Collection 接口是 Iterable 的一个子类型，它有一个流方法，因此它提供了迭代和流两种访问方式。因此，**Collection 或其适当的子类通常是公共序列返回方法的最佳返回类型。** 数组还提供了使用 `Arrays.asList` 和 `Stream.of` 方法进行简单迭代和流访问。如果返回的序列足够小，可以轻松地装入内存，那么最好返回标准集合实现之一，例如 ArrayList 或 HashSet。但是 **不要将一个大的序列存储在内存中，只是为了将它作为一个集合返回。**

如果返回的序列比较大，但是可以有规律地表示，那么可以考虑实现一个特殊用途的集合。例如，假设你想要返回给定集合的幂集，该集合由它的所有子集组成。`{a, b, c}` 的排列组合有 `{{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}`。如果一个集合有 n 个元素，它的幂集有 2^n。因此，你甚至不应该考虑在标准集合实现中存储全部排列组合。然而，在 AbstractList 的帮助下，可以很容易实现这个需求的自定义集合。

诀窍是使用索引幂集的每个元素设置一个位向量，在该指数的 n 位表示第 n 个元素的存在与否从源。在本质上，之间有一个自然的映射二进制数字从 0 到 2n–1 和一组 n 元的幂集。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;
            }
        };
    }
}
```

注意，如果输入集包含超过 30 个元素，`PowerSet.of` 将抛出异常。这突出的缺点使用 Collection 作为返回类型而不是流或 Iterable：收集 int-returning 大小的方法,这限制了 Integer.MAX_VALUE 返回序列的长度，或 231–1。收集规范允许大小方法返回 231–1 如果集合更大，甚至是无限的,但这不是一个完全令人满意的解决方案。

为了在 AbstractCollection 之上编写 Collection 实现，除了 Iterable 所需的方法外，只需要实现两个方法：contains 和 size。通常很容易编写这些方法的有效实现。如果它是不可行的，可能是因为序列的内容在迭代发生之前没有预先确定，那么返回一个流或 iterable，以感觉更自然的方式返回。如果你选择，你可以使用两个不同的方法返回这两个值。

有时，你将仅根据实现的易用性来选择返回类型。例如，假设你想编写一个返回输入列表的所有（连续

的）子列表的方法。生成这些子列表并将它们放入标准集合中只需要三行代码，但是保存该集合所需的内存是源列表大小的二次方。虽然这没有幂集那么糟糕，幂集是指数的，但显然是不可接受的。实现自定义集合（就像我们为 power 集所做的那样）将会非常繁琐，因为 JDK 缺少一个框架迭代器实现来帮助我们。

然而，实现一个输入列表的所有子列表的流是很简单的，尽管它确实需要一些深入的了解。让我们将包含列表的第一个元素的子列表称为列表的前缀。例如，`(a,b,c)` 的前缀 `(a)`、`(a、b)` 和 `(a,b,c)`。类似地，让我们调用包含最后一个元素后缀的子列表，因此 `(a，b，c)` 的后缀是 `(a，b，c)`、`(b，c)` 和 `(c)`。我们的理解是，列表的子列表仅仅是前缀的后缀（或后缀的前缀相同）和空列表。这个观察直接导致了一个清晰、合理、简洁的实现：

```
// 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()));
    }
}
```

注意 `Stream.concat` 方法将空列表添加到返回的流中。还要注意，flatMap 方法（Item-45）用于生成由所有前缀的所有后缀组成的单一流。最后，请注意，我们通过映射由 `IntStream.range` 和 `IntStream.rangeClosed` 返回的连续 int 值流来生成前缀和后缀。因此，我们的子列表实现在本质上类似于嵌套的 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));
```

可以将这个 for 循环直接转换为流。结果比我们以前的实现更简洁，但可读性可能稍差。它在形态上类似于 Item-45 中 Cartesian 的 streams 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);
}
```

与前面的 for 循环一样，该代码不发出空列表。为了修复这个缺陷，你可以使用 concat，就像我们在上一个版本中所做的那样，或者在 rangeClosed 调用中将 1 替换为 `(int) Math.signum(start)` 。

子列表的这两种流实现都可以，但是都需要一些用户使用流到迭代的适配器，或者在迭代更自然的地方使用流。流到迭代适配器不仅打乱了客户机代码，而且在我的机器上，它还将循环速度降低了 2.3 倍。专门构建的集合实现（这里没有显示）非常冗长，但是运行速度是我的机器上基于流的实现的 1.4 倍。

总之，在编写返回元素序列的方法时，请记住，有些用户可能希望将它们作为流处理，而有些用户可能希望对它们进行迭代。试着适应这两个群体。如果可以返回集合，那么就这样做。如果你已经在一个集合中拥有了元素，或者序列中的元素数量足够小，可以创建一个新的元素，那么返回一个标准集合，例如 ArrayList 。否则，请考虑像对 power 集那样实现自定义集合。如果返回集合不可行，则返回流或 iterable，以看起来更自然的方式返回。如果在未来的 Java 版本中，流接口声明被修改为可迭代的，那么你应该可以随意返回流，因为它们将允许流处理和迭代。

Item 48: Use caution when making streams parallel（谨慎使用并行流）

在主流语言中，Java 一直走在提供简化并发编程任务工具的前列。当 Java 在 1996 年发布时，它内置了对线程的支持，支持同步和 wait/notify。Java 5 引入了 `java.util.concurrent` 。具有并发集合和执行器框架的并发库。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，它可以通过对 parallel 方法的一次调用来并行化。用 Java 编写并发程序变得越来越容易，但是编写正确且快速的并发程序却和以前一样困难。在并发编程中，安全性和活性的违反是不可避免的，并行流管道也不例外。

考虑 Item-45 的程序：

```
// 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);
}
```

在我的机器上，这个程序立即开始打印素数，运行 12.5 秒完成。假设我天真地尝试通过向流管道添加对 `parallel()` 的调用来加速它。你认为它的性能会怎么样？它会快几个百分点吗？慢了几个百分点？遗憾的是，它不会打印任何东西，但是 CPU 使用率会飙升到 90%，并且会无限期地停留在那里（活跃性失败）。这个项目最终可能会终止，但我不愿意知道；半小时后我强行停了下来。

这是怎么回事？简单地说，stream 库不知道如何并行化这个管道，因此启发式会失败。即使在最好的情况下，**如果源来自 `Stream.iterate` 或使用 Intermediate 操作限制，并行化管道也不太可能提高其性能。** 这条管道必须解决这两个问题。更糟糕的是，默认的并行化策略通过假设处理一些额外的元素和丢弃任何不需要的结果没有害处来处理极限的不可预测性。在这种情况下，找到每一个 Mersenne 素数所需的时间大约是找到上一个 Mersenne 素数所需时间的两倍。因此，计算单个额外元素的成本大致等于计算之前所有元素的总和，而这条看上去毫无问题的管道将自动并行化算法推到了极致。这个故事的寓意很简单：**不要不加区别地将流管道并行化。** 性能后果可能是灾难性的。

通常，**并行性带来的性能提升在 ArrayList、HashMap、HashSet 和 ConcurrentHashMap 实例上的流效果最好；int 数组和 long 数组也在其中。** 这些数据结构的共同之处在于，它们都可以被精确且廉价地分割成任意大小的子程序，这使得在并行线程之间划分工作变得很容易。stream 库用于执行此任务的抽象是 spliterator，它由流上的 spliterator 方法返回并可迭代。

所有这些数据结构的另一个重要共同点是，当按顺序处理时，它们提供了从优秀到优秀的引用位置：顺序元素引用一起存储在内存中。这些引用引用的对象在内存中可能彼此不太接近，这降低了引用的位置。引用位置对于并行化批量操作非常重要：如果没有它，线程将花费大量时间空闲，等待数据从内存传输到处理器的缓存中。具有最佳引用位置的数据结构是基本数组，因为数据本身是连续存储在内存中的。

流管道 Terminal 操作的性质也会影响并行执行的有效性。如果与管道的总体工作相比，在 Terminal 操作中完成了大量的工作，并且该操作本质上是顺序的，那么管道的并行化将具有有限的有效性。并行性的最佳 Terminal 操作是缩减，其中来自管道的所有元素都使用流的缩减方法之一进行组合，或者使用预先打包的缩减，如最小、最大、计数和和。anyMatch、allMatch 和 noneMatch 的短路操作也适用于并行性。流的 collect 方法执行的操作称为可变缩减，它们不是并行性的好候选，因为组合集合的开销是昂贵的。

如果你编写自己的流、Iterable 或 Collection 实现，并且希望获得良好的并行性能，则必须重写 spliterator 方法，并广泛地测试结果流的并行性能。编写高质量的 spliterator 是困难的，超出了本书的范围。

并行化流不仅会导致糟糕的性能，包括活动失败；它会导致不正确的结果和不可预知的行为（安全故障）。 如果管道使用映射器、过滤器和其他程序员提供的函数对象，而这些对象没有遵守其规范，则并行化管道可能导致安全故障。流规范对这些功能对象提出了严格的要求。例如，传递给流的 reduce 操作的累加器和组合器函数必须是关联的、不干扰的和无状态的。如果你违反了这些要求（其中一些要求在 Item-46 中讨论），但是按顺序运行管道，则可能会产生正确的结果；如果你并行化它，它很可能会失败，可能是灾难性的。沿着这些思路，值得注意的是，即使并行化的 Mersenne 素数程序运行到完成，它也不会以正确的（升序）顺序打印素数。为了保留序列版本所显示的顺序，你必须将 forEach 这一 Terminal 操作替换为 forEachOrdered，它保证按顺序遍历并行流。

即使假设你正在使用一个高效的可分割源流、一个可并行化的或廉价的 Terminal 操作，以及不受干扰的函数对象，你也不会从并行化中获得良好的加速，除非管道正在做足够的实际工作来抵消与并行性相关的成本。作为一个非常粗略的估计，流中的元素数量乘以每个元素执行的代码行数至少应该是 100000 [Lea14]。

重要的是要记住，并行化流严格来说是一种性能优化。与任何优化一样，你必须在更改之前和之后测试性能，以确保它值得进行（Item-67）。理想情况下，你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。一个行为不当的管道可能会损害系统中不相关部分的其他管道的性能。

如果在并行化流管道时，听起来你的胜算非常大，那是因为它们确实如此。一位熟悉的人维护着大量使用流的数百万在线代码库，他发现只有少数几个地方并行流是有效的。这并不意味着你应该避免并行化流。在适当的情况下，可以通过向流管道添加并行调用来实现处理器内核数量的近乎线性的加速。某些领域，如机器学习和数据处理，特别适合于这些加速。

作为一个简单的例子，一个流管道并行性是有效的，考虑这个函数计算 `π(n)`，质数数目小于或等于 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，需要 31 秒计算 `π(108)` 使用这个函数。简单地添加 `parallel()` 调用将时间缩短到 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，在我的四核计算机上，并行化的计算速度提高了 3.7 倍。值得注意的是，这不是你如何计算 `π(n)` 为大 n 的值。有更有效的算法，特别是 Lehmer 公式。

如果要并行化一个随机数流，可以从一个 SplittableRandom 实例开始，而不是从一个 ThreadLocalRandom（或者本质上已经过时的 random）开始。SplittableRandom 正是为这种用途而设计的，它具有线性加速的潜力。ThreadLocalRandom 是为单个线程设计的，它将自适应为并行流源，但速度没有 SplittableRandom 快。随机同步每个操作，因此它将导致过度的并行争用。

总之，甚至不要尝试并行化流管道，除非你有充分的理由相信它将保持计算的正确性以及提高速度。不适当地并行化流的代价可能是程序失败或性能灾难。如果你认为并行性是合理的，那么请确保你的代码在并行运行时保持正确，并在实际情况下进行仔细的性能度量。如果你的代码保持正确，并且这些实验证实了你对提高性能的怀疑，那么，并且只有这样，才能在生产代码中并行化流。

Chapter 8. Methods（方法）

Chapter 8 Introduction（章节介绍）

THIS chapter discusses several aspects of method design: how to treat parameters and return values, how to design method signatures, and how to document methods. Much of the material in this chapter applies to constructors as well as to methods. Like Chapter 4, this chapter focuses on usability, robustness, and flexibility.

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，以及如何编写方法文档。本章的大部分内容不仅适用于方法，也适用于构造函数。与第四章一样，本章重点讨论可用性、健壮性和灵活性。

Item 49: Check parameters for validity（检查参数的有效性）

大多数方法和构造函数都对传递给它们的参数值有一些限制。例如，索引值必须是非负的，对象引用必须是非空的，这种情况并不少见。你应该清楚地在文档中记录所有这些限制，并在方法主体的开头使用检查来实施它们。你应该在错误发生后尽快找到它们，这是一般原则。如果不这样做，就不太可能检测到错误，而且即使检测到错误，确定错误的来源也很难。

如果一个无效的参数值被传递给一个方法，如果该方法在执行之前会检查它的参数，那么这个过程将迅速失败，并引发适当的异常。如果方法未能检查其参数，可能会发生以下几件事。该方法可能会在处理过程中出现令人困惑的异常而失败。更糟的是，该方法可以正常返回，但会静默计算错误的结果。最糟糕的是，该方法可以正常返回，但会使某个对象处于隐患状态，从而在将来某个不确定的时间在代码中某个不相关的位置上导致错误。换句话说，如果没有验证参数，可能会违反故障原子性（Item-76）。

对于公共方法和受保护的方法，如果在方法说明使用 Javadoc 的 `@throws` 标签记录异常，表明如果违反了对参数值的限制，将会引发该异常（Item-74）。通常，生成的异常将是 IllegalArgumentException、IndexOutOfBoundsException 或 NullPointerException（Item-72）。一旦你在文档中记录了方法参数上的限制，并且记录了如果违反这些限制将引发的异常，那么实施这些限制就很简单了。这里有一个典型的例子：

```
/**
* 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
}
```

注意，文档注释并没有说「如果 m 为空，mod 将抛出NullPointerException」，尽管方法确实是这样做的，这是调用 `m.signum()` 的副产品。这个异常记录在类级别的文档注释中，用于包含 BigInteger 类。类级别注释适用于类的所有公共方法中的所有参数。这是避免在每个方法上分别记录每个 NullPointerException 而造成混乱的好方法。它可以与 `@Nullable` 或类似的注释结合使用，以指示某个特定参数可能为 null，但这种做法并不标准，为此使用了多个注释。

在 Java 7 中添加的 `Objects.requireNonNull` 方法非常灵活和方便，因此不再需要手动执行空检查。 如果愿意，可以指定自己的异常详细信息。该方法返回它的输入，所以你可以执行一个空检查，同时你使用一个值：

```
// Inline use of Java's null-checking facility
this.strategy = Objects.requireNonNull(strategy, "strategy");
```

你还可以忽略返回值并使用 `Objects.requireNonNull` 作为一个独立的 null 检查来满足你的需要。

在 Java 9 中，范围检查功能被添加到 `java.util.Objects` 中。这个功能由三个方法组成：checkFromIndexSize、checkFromToIndex 和 checkIndex。这个工具不如空检查方法灵活。它不允许你指定自己的异常详细信息，而且它仅用于 List 和数组索引。它不处理封闭范围（包含两个端点）。但如果它满足你的需求，它仍是一个有用的工具。

对于未导出的方法，作为包的作者，你应该定制方法调用的环境，因此你可以并且应该确保只传递有效的参数值。因此，非公共方法可以使用断言检查它们的参数，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
}
```

从本质上说，这些断言在声明时，条件将为 true，而不管其客户端如何使用所包含的包。与普通的有效性检查不同，如果断言失败，则会抛出 AssertionError。与普通的有效性检查不同，如果断言没有起到

作用，本质上不存在成本，除非你启用它们，你可以通过将 `-ea`（或 `-enableassertion`）标志传递给 java 命令来启用它们。有关断言的更多信息，请参见教程 [Asserts]。

特别重要的是，应检查那些不是由方法使用，而是存储起来供以后使用的参数的有效性。例如，考虑第 101 页中的静态工厂方法，它接受一个 int 数组并返回数组的 List 视图。如果客户端传入 null，该方法将抛出 NullPointerException，因为该方法具有显式检查(调用 `Objects.requireNonNull`)。如果省略了检查，该方法将返回对新创建的 List 实例的引用，该实例将在客户端试图使用它时抛出 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变得非常复杂。

构造函数代表了一种特殊的情况，即，你应该检查要存储起来供以后使用的参数的有效性。检查构造函数参数的有效性对于防止构造生成实例对象时，违背类的对象的不变性非常重要。

在执行方法的计算任务之前，应该显式地检查方法的参数，这条规则也有例外。一个重要的例外是有效性检查成本较高或不切实际，或者检查是在计算过程中隐式执行了。例如，考虑一个为对象 List 排序的方法，比如 `Collections.sort(List)`。List 中的所有对象必须相互比较。在对 List 排序的过程中，List 中的每个对象都会与列表中的其他对象进行比较。如果对象不能相互比较，将抛出 ClassCastException，这正是 sort 方法应该做的。因此，没有必要预先检查列表中的元素是否具有可比性。但是，请注意，不加区别地依赖隐式有效性检查可能导致失败原子性的丢失（Item-76）。

有时，计算任务会隐式地执行所需的有效性检查，但如果检查失败，则抛出错误的异常。换句话说，计算任务由于无效参数值抛出的异常，与文档中记录的方法要抛出的异常不匹配。在这种情况下，你应该使用 Item-73 中描述的异常转译技术来将计算任务抛出的异常转换为正确的异常。

不要从本条目推断出：对参数的任意限制都是一件好事。相反，你应该把方法设计得既通用又实用。对参数施加的限制越少越好，假设该方法可以对它所接受的所有参数值进行合理的处理。然而，一些限制常常是实现抽象的内在限制。

总而言之，每次编写方法或构造函数时，都应该考虑参数存在哪些限制。你应该在文档中记录这些限制，并在方法主体的开头显式地检查。养成这样的习惯是很重要的。它所涉及的这一少量工作及其所花费的时间，将在有效性检查出现第一次失败时连本带利地偿还。

Item 50: Make defensive copies when needed（在需要时制作防御性副本）

Java 是一种安全的语言，这是它的一大优点。这意味着在没有本地方法的情况下，它不受缓冲区溢出、数组溢出、非法指针和其他内存损坏错误的影响，这些错误困扰着 C 和 c++ 等不安全语言。在一种安全的语言中，可以编写一个类并确定它们的不变量将保持不变，而不管在系统的任何其他部分发生了什么。在将所有内存视为一个巨大数组的语言中，这是不可能的。

即使使用一种安全的语言，如果你不付出一些努力，也无法与其他类隔离。**你必须进行防御性的设计，并假定你的类的客户端会尽最大努力破坏它的不变量。** 随着人们越来越多地尝试破坏系统的安全性，这个观点越来越正确，但更常见的情况是，你的类将不得不处理程序员的无意错误所导致的意外行为。无论哪种方式，都值得花时间编写一个健壮的类来面对行为不轨的客户端。

虽然如果没有对象的帮助，另一个类是不可能修改对象的内部状态的，但是要提供这样的帮助却出奇地容易。例如，考虑下面的类，它表示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，并且要求一个时间段的开始时间不能在结束时间之后。然而，利用 Date 是可变的这一事实很容易绕过这个约束：

```
// 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，解决这个问题的典型方法就是使用 Instant（或 Local-DateTime 或 ZonedDateTime）来代替 Date，因为 Instant（和其他时间类）类是不可变的（Item-17）。**Date 已过时，不应在新代码中使用。** 尽管如此，问题仍然存在：有时你必须在 API 和内部表示中使用可变值类型，本项目中讨论的技术适用于这些情形。

为了保护 Period 实例的内部不受此类攻击，**必须将每个可变参数的防御性副本复制给构造函数**，并将副本用作 Period 实例的组件，而不是原始组件：

```
// 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);
}
```

有了新的构造函数，之前的攻击将不会对 Period 实例产生影响。注意，**防御性副本是在检查参数的有效性之前制作的（Item-49），有效性检查是在副本上而不是在正本上执行的。** 虽然这看起来不自然，但却是必要的。在检查参数和复制参数之间的空窗期，它保护类不受来自其他线程的参数更改的影响。在计算机安全社区里，这被称为 time-of-check/time-of-use 或 TOCTOU 攻击 [Viega01]。

还要注意，我们没有使用 Date 的 clone 方法来创建防御性副本。因为 Date 不是 final 的，所以不能保证 clone 方法返回一个 java.util.Date 的实例对象：它可以返回一个不受信任子类的实例，这个子类是专门为恶意破坏而设计的。例如，这样的子类可以在创建时在私有静态列表中记录对每个实例的引用，并允许攻击者访问这个列表。这将使攻击者可以自由控制所有实例。为防止此类攻击，**对可被不受信任方子类化的参数类型，不要使用 clone 方法进行防御性复制。**

虽然替换构造函数成功地防御了之前的攻击，但是仍然可以修改 Period 实例，因为它的访问器提供了对其可变内部结构的访问：

```
// 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!
```

要防御第二次攻击，只需修改访问器，返回可变内部字段的防御副本：

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

有了新的构造函数和新的访问器，Period 实际上是不可变的。无论程序员多么恶毒或无能，都不可能违背一个时间段的开始时间不能在结束时间之后这一不变条件（除非使用诸如本地方法和反射之类的外部语言手段）。这是真的，因为除了 Period 本身之外，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，可以使用 clone 方法进行防御性复制。这是因为我们知道 Period 的内部 Date 对象的类是 `java.util.Date`，而不是某个不可信的子类。也就是说，基于 Item-13 中列出的原因，一般情况下，最好使用构造函数或静态工厂来复制实例。

参数的防御性复制不仅适用于不可变类。在编写方法或构造函数时，如果要在内部数据结构中存储对客户端提供的对象的引用，请考虑客户端提供的对象是否可能是可变的。如果是，请考虑该对象进入数据结构之后，你的类是否能够容忍该对象发生更改。如果答案是否定的，则必须防御性地复制对象，并将副本输入到数据结构中，而不是原始正本。举个例子，如果你正在考虑使用由客户提供的对象引用作为内部 Set 实例的元素，或者作为内部 Map 实例的键，就应该意识到如果这个对象在插入之后发生改变，Set 或者 Map 的约束条件就会遭到破坏。

在将内部组件返回给客户端之前应对其进行防御性复制也是如此。无论你的类是否是不可变的，在返回对可变内部组件的引用之前，你都应该三思。很有可能，你应该返回一个防御性副本。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，应该始终创建一个防御性的副本。或者，你可以返回数组的不可变视图。这两种技术都已经在 Item-15 中演示过。

可以说，所有这些教训体现了，在可能的情况下，应该使用不可变对象作为对象的组件，这样就不必操心防御性复制（Item-17）。在我们的 Period 示例中，使用 Instant（或 LocalDateTime 或 ZonedDateTime），除非你使用的是 Java 8 之前的版本。如果使用较早的版本，一个选项是存储 `Date.getTime()` 返回的 long 基本数据类型，而不是 Date 引用。

防御性复制可能会带来性能损失，而且并不总是合理的。如果一个类信任它的调用者不会去修改内部组件，可能是因为类和它的客户端都是同一个包的一部分，那么就应该避免防御性复制。在这种情况下，类文档应该表明调用者不能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对其进行防御性复制也并不总是合适的。有一些方法和构造函数，它们的调用要求参数引用的对象要进行显式切换。当调用这样一个方法时，客户端承诺不再直接修改对象。希望拥有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这一点。

包含方法或构造函数的类，如果其方法或构造函数的调用需要移交对象的控制权，就不能保护自己免受恶意客户端的攻击。只有当一个类和它的客户端之间相互信任，或者对类的不变量的破坏只会对客户端造成伤害时，这样的类才是可接受的。后一种情况的一个例子是包装类模式（Item-18）。根据包装类的性质，客户端可以在包装对象之后直接访问对象，从而破坏类的不变量，但这通常只会损害客户端。

总而言之，如果一个类具有从客户端获取或返回给客户端的可变组件，则该类必须防御性地复制这些组件。如果复制的成本过高，并且类信任它的客户端不会不适当地修改组件，那么可以不进行防御性的复制，取而代之的是在文档中指明客户端的职责是不得修改受到影响的组件。

Item 51: Design method signatures carefully（仔细设计方法签名）

本条目是一个 API 设计提示的大杂烩，它们还不完全值得拥有独立的条目。总之，它们将帮助你使 API 更容易学习和使用，并且更不容易出错。

仔细选择方法名称。 名称应始终遵守标准的命名约定（Item-68）。你的主要目标应该是选择可理解的、与同一包中的其他名称风格一致的名称。你的第二个目标应该是选择被广泛认可的名字。避免长方法名。如果有疑问，可以参考 Java 库 API，尽管其中也存在大量的矛盾（考虑到这些库的规模和范围，这是不可避免的）但也达成了相当多的共识。

不要提供过于便利的方法。 每种方法都应该各司其职。太多的方法使得类难以学习、使用、记录、测试和维护。对于接口来说更是如此，在接口中，太多的方法使实现者和用户的工作变得复杂。对于类或接口支持的每个操作，请提供一个功能齐全的方法。只有在经常使用时才考虑提供便捷方式。**但如果有疑问，就不要提供。**

避免长参数列表。 设定四个或更少的参数。大多数程序员记不住更长的参数列表。如果你的许多方法超过了这个限制，而用户没有对文档的不断查看，你的 API 将无法使用。现代 IDE 会有所帮助，但是使用简短的参数列表仍然会让情况好得多。**长序列的同类型参数尤其有害**。用户不仅不能记住参数的顺序，而且当他们不小心转置参数时，他们的程序仍然会编译和运行。它们只是不会按照作者的意图去做。

有三种方法可以缩短过长的参数列表。一种方法是将方法分解为多个方法，每个方法只需要参数的一个子集。如果操作不当，这可能导致产生太多的方法，但它也可以通过增加正交性来帮助减少方法数量。例如，考虑 `java.util.List` 接口。它不提供查找子列表中元素的第一个或最后一个索引的方法，这两个方法都需要三个参数。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 方法组合使用以达到所需的功能，其中每个方法都有一个参数。此外，subList 方法可以与操作 List 实例的任何方法组合使用，以执行子列表上的任意操作。这样的 API 就具有非常高的 power-to-weight 比。

缩短长参数列表的第二种技术是创建 helper 类来保存参数组。通常，这些 helper 类是静态成员类（Item-42）。如果经常出现的参数序列表示一些不同的实体，则推荐使用这种技术。例如，假设你正在编写一个表示纸牌游戏的类，你发现会不断地传递一个序列，其中包含两个参数，分别表示纸牌的等级和花色。如果你添加一个 helper 类来表示一张卡片，并用 helper 类的一个参数替换参数序列中的每个出现的参数，那么你的 API 以及类的内部结构都可能受益。

结合前两个方面讨论的第三种技术是，从对象构建到方法调用都采用建造者模式（Item-2）。如果你有一个方法带有许多参数，特别是其中一些参数是可选的，最好定义一个对象来表示所有参数，并允许客户端多次调用「setter」来使用这个对象，每一次都设置一个参数或较小相关的组。一旦设置了所需的参数，客户机就调用对象的「execute」方法，该方法对参数进行最终有效性检查并执行实际操作。

对于参数类型，优先选择接口而不是类（Item-64）。如果有合适的接口来定义参数，那么使用它来支持实现该接口的类。例如，没有理由编写一个输入使用 HashMap 的方法，而应该使用 Map。这允许你传入 HashMap、TreeMap、ConcurrentHashMap、TreeMap 的子映射或任何尚未编写的 Map 实现。通过使用类而不是接口，你可以将客户端限制在特定的实现中，如果输入数据碰巧以某种其他形式存在，则会强制执行不必要的、可能代价很高的复制操作。

双元素枚举类型优于 boolean 参数， 除非布尔值的含义在方法名中明确。枚举使代码更容易读和写。此外，它们使以后添加更多选项变得更加容易。例如，你可能有一个 Thermometer 类型与静态工厂，采用枚举：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

`Thermometer.newInstance(TemperatureScale.CELSIUS)` 不仅比 `Thermometer.newInstance(true)` 更有意义，而且你可以在将来的版本中向 TemperatureScale 添加 KELVIN，而不必向 Thermometer 添加新的静态工厂。此外，你还可以将 TemperatureScale 依赖项重构为 enum 常量（Item-34）上的方法。例如，每个刻度单位都有一个方法，该方法带有 double 值并将其转换为摄氏度。

Item 52: Use overloading judiciously（明智地使用重载）

下面的程序是一个善意的尝试，根据一个 Collection 是 Set、List 还是其他的集合类型来进行分类：

```
// 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));
    }
}
```

你可能期望这个程序打印 Set，然后是 List 和 Unknown Collection，但是它没有这样做。它打印 Unknown Collection 三次。为什么会这样？因为 classify 方法被重载，并且 **在编译时就决定了要调用哪个重载。** 对于循环的三个迭代过程，参数的编译时类型是相同的： `Collection<?>` 。运行时类型在每个迭代中是不同的，但这并不影响重载的选择。因为参数的编译时类型是 `Collection<?>` ，唯一适

用的重载是第三个，`classify(Collection<?>)`，这个重载在循环的每个迭代过程中都会调用。

这个程序的行为违反常规，因为 **重载方法的选择是静态的，而覆盖法的选择是动态的。** 在运行时根据调用方法的对象的运行时类型选择覆盖方法的正确版本。提醒一下，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，方法将被覆盖。如果在子类中覆盖实例方法，并且在子类的实例上调用此方法，则执行子类的覆盖方法，而不管子类实例的编译时类型如何。为了更具体的说明，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 SparklingWine 和 Champagne 子类中重写。正如你所期望的，这个程序打印出 wine、sparkling 和 champagne，即使实例的编译时类型是循环每次迭代中的 wine。对象的编译时类型对调用覆盖方法时执行的方法没有影响；「最特定的」覆盖方法总是被执行。将此与重载进行比较，在重载中，对象的运行时类型对执行重载没有影响；选择是在编译时进行的，完全基于参数的编译时类型。

在 CollectionClassifier 示例中，程序的目的是通过根据参数的运行时类型自动分派到适当的方法重载来识别参数的类型，就像 Wine 示例中的 name 方法所做的那样。方法重载不提供此功能。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 classification 的所有三个重载：

```
public static String classify(Collection<?> c) {
    return c instanceof Set ? "Set" :c instanceof List ? "List" : "Unknown
Collection";
}
```

因为覆盖是规范，而重载是例外，所以覆盖满足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。编写可能使程序员感到困惑的代码是不好的行为。对于 API 尤其如此。如果 API 的用户不知道一组参数应该调用哪一种方法重载，那么使用 API 时很可能会导致错误。这些错误很可能在运行时表现为不稳定的行为，许多程序员将很难诊断它们。因此，**应该避免混淆重载的用法。**

究竟是什么构成了混淆重载的用法还有待商榷。**安全、保守的策略是永远不导出具有相同数量参数的两个重载。** 如果一个方法使用了可变参数，保守策略是根本不重载它，除非如 Item-53 所述。如果遵守这些限制，程序员就不会怀疑一组参数应该调用哪一种方法重载。这些限制并不十分繁琐，因为 **你总是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。对于每个基本类型和几种引用类型，其 write 方法都有变体。这些变体都有不同的名称，而不是重载 write 方法，例如 `writeBoolean(boolean)`、`writeInt(int)` 和 `writeLong(long)`。与重载相比，这种命名模式的另一个好处是，可以为 read 方法提供相应的名称，例如 `readBoolean()`、`readInt()` 和 `readLong()`。ObjectInputStream 类实际上也提供了这样的读方法。

对于构造函数，你没有使用不同名称的机会：一个类的多个构造函数只能重载。在很多情况下，你可以选择导出静态工厂而不是构造函数（Item-1）。你可能会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是有必要的。

如果总是清楚一组参数应该调用哪一种方法重载，那么用相同数量的参数导出多个重载不太可能让程序员感到困惑。在这种情况下，每对重载中至少有一个对应的形式参数在这两个重载中具有「完全不同的」类型。如果不可能将任何非空表达式强制转换为这两种类型，那么这两种类型是完全不同的。在这些情况下，应用于给定实际参数集的重载完全由参数的运行时类型决定，且不受其编译时类型的影响，因此消除了一个主要的混淆源。例如，ArrayList 有一个接受 int 的构造函数和第二个接受 Collection 的构造函数。很难想象在什么情况下会不知道这两个构造函数中哪个会被调用。

在 Java 5 之前，所有原始类型都与所有引用类型完全不同，但在自动装箱时并非如此，这造成了真正的麻烦。考虑以下方案：

```
    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（包括）添加到已排序的 Set 和 List 中。然后，它执行三个相同的调用来删除集合和列表。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 Set 中删除非负值，从 List 中删除奇数值，并输出 `[-3,2,1][-2,0,2]`。把这种行为称为混乱，只是一种保守的说法。

实际情况如下：调用 `set.remove(i)` 选择重载 `remove(E)`，其中 E 是 set （Integer）的元素类型，而将从 int 自动装箱到 Integer 中。这是你期望的行为，因此程序最终会从 Set 中删除正值。另一方面，对 `list.remove(i)` 的调用选择重载 `remove(int i)`，它将删除 List 中指定位置的元素。如果从 List `[-3, -2, -1,0,1,2]` 开始，移除第 0 个元素，然后是第 1 个，然后是第 2 个，就只剩下 `[-2,0,2]`，谜底就解开了。若要修复此问题，要将 `list.remove` 的参数转换成 Integer，强制选择正确的重载。或者，你可以调用 `Integer.valuef()`，然后将结果传递给 `list.remove`。无论哪种方式，程序都会按预期打印 `[-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))
    }
```

前一个示例所演示的令人困惑的行为是由于 List 接口对 remove 方法有两个重载：`remove(E)` 和 `remove(int)`。在 Java 5 之前，当 List 接口被「泛化」时，它有一个 `remove(Object)` 方法代替 `remove(E)`，而相应的参数类型 Object 和 int 则完全不同。但是，在泛型和自动装箱的存在下，这两种参数类型不再完全不同。换句话说，在语言中添加泛型和自动装箱破坏了 List 接口。幸运的是，Java 库中的其他 API 几乎没有受到类似的破坏，但是这个故事清楚地表明，自动装箱和泛型出现后，在重载时就应更加谨慎。Java 8 中添加的 lambda 表达式和方法引用进一步增加了重载中混淆的可能性。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 方法调用看起来很相似，但是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重载，该重载接受 Runnable。这是怎么回事？令人惊讶的答案是，submit 方法有一个重载，它接受一个 `Callable<T>`，而线程构造函数没有。你可能认为这不会有什么区别，因为 println 的所有重载都会返回 void，所以方法引用不可能是 Callable。这很有道理，但重载解析算法不是这样工作的。也许同样令人惊讶的是，如果 println 方法没有被重载，那么 submit 方法调用将是合法的。正是被引用的方法 println 和被调用的方法 submit 的重载相结合，阻止了重载解析算法按照你所期望的那样运行。

从技术上讲，问题出在 System.out::println 上，它是一个不准确的方法引用 [JLS, 15.13.1]，并且「某些包含隐式类型化 lambda 表达式或不准确的方法引用的参数表达式会被适用性测试忽略，因为它们的含义在选择目标类型之前无法确定 [JLS, 15.12.2]。」如果你不明白这段话，不要担心；它的目标是编译器编写器。关键是在相同的参数位置上重载具有不同功能接口的方法或构造函数会导致混淆。因此，**不要重载方法来在相同的参数位置上使用不同的函数式接口。** 用本项目的话说，不同的函数式接口并没有根本的不同。如果你通过命令行开关 `Xlint:overloads`，Java 编译器将对这种有问题的重载发出警告。

数组类型和 Object 以外的类类型是完全不同的。此外，数组类型和 Serializable 和 Cloneable 之外的接口类型也完全不同。如果两个不同的类都不是另一个类的后代 [JLS, 5.5]，则称它们是不相关的。例如，String 和 Throwable 是不相关的。任何对象都不可能是两个不相关类的实例，所以不相关的类也是完全不同的。

还有其他成对的类不能在任何方向相互转换 [JLS, 5.1.12]，但是一旦超出上面描述的简单情况，大多数程序员就很难辨别一组参数应该调用哪一种方法重载。决定选择哪个重载的规则非常复杂，并且随着每个版本的发布而变得越来越复杂。很少有程序员能理解它们所有的微妙之处。

有时候，你可能觉得会被迫违反本条目中的指导原则，特别是在更新现有类时。例如，考虑 String，它从 Java 4 开始就有一个 `contentEquals(StringBuffer)` 方法。在 Java 5 中，添加了 CharSequence 来为 StringBuffer、StringBuilder、String、CharBuffer 和其他类似类型提供公共接口。在添加 CharSequence 的同时，String 还配备了一个重载的 contentEquals 方法，该方法接受 CharSequence。

虽然这样的重载明显违反了此项中的指导原则，但它不会造成任何危害，因为当在同一个对象引用上调用这两个重载方法时，它们做的是完全相同的事情。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，就没有什么不良后果。确保这种行为的标准方法是将更具体的重载转发给更一般的重载：

```
// Ensuring that 2 methods have identical behavior by forwarding
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

虽然 Java 库在很大程度上遵循了本条目中的主旨精神，但是有一些类违反了它。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执行完全不同的操作。这样做没有真正的理由，它应该被视为一种异常行为，有可能造成真正的混乱。

总而言之，方法可以重载，但并不意味着就应该这样做。通常，最好避免重载具有相同数量参数的多个签名的方法。在某些情况下，特别是涉及构造函数的情况下，可能难以遵循这个建议。在这些情况下，你至少应该避免同一组参数只需经过类型转换就可以被传递给不同的重载方法。如果这是无法避免的，例如，因为要对现有类进行改造以实现新接口，那么应该确保在传递相同的参数时，所有重载的行为都是相同的。如果你做不到这一点，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能工作。

Item 53: Use varargs judiciously（明智地使用可变参数）

可变参数方法的正式名称是 variable arity methods [JLS, 8.4.1]，它接受指定类型的零个或多个参数。可变参数首先创建一个数组，其大小是在调用点上传递的参数数量，然后将参数值放入数组，最后将数组传递给方法。

例如，这里有一个可变参数方法，它接受一系列 int 参数并返回它们的和。如你所料，`sum(1, 2, 3)` 的值为 6，`sum()` 的值为 0：

```java
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

有时，编写一个方法需要一个或多个某种类型的参数，而不是零个或多个参数，这是合适的。例如，假设你想编写一个函数来计算其参数的最小值。如果客户端不传递参数，则此函数定义得不好。你可以在运行时检查数组长度：

```java
// 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;
}
```

这个解决方案有几个问题。最严重的情况是，如果客户端不带参数调用此方法，那么它将在运行时而不是编译时失败。另一个问题是它不美观。必须包含对 args 的显式有效性检查，并且不能使用 for-each 循环，除非将 min 初始化为 Integer.MAX_VALUE，也很不美观。

幸运的是，有一种更好的方法可以达到预期的效果。声明方法获取两个参数，一个指定类型的常规参数和一个该类型的可变参数。这个解决方案弥补了前一个解决方案的所有不足：

```java
// 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;
}
```

从这个例子中可以看出，在方法需要参数数量可变的情况下，可变参数是有效的。可变参数是为 printf 和经过改造的核心反射机制（Item-65）而设计的，它们与可变参数同时被添加到 JDK，printf 和 reflection 都从可变参数中受益匪浅。

在性能关键的情况下使用可变参数时要小心。每次调用可变参数方法都会导致数组分配和初始化。如果你已经从经验上确定你负担不起这个成本，但是你仍需要可变参数的灵活性，那么有一种模式可以让你鱼与熊掌兼得。假设你已经确定对方法 95% 的调用只需要三个或更少的参数。可以声明该方法的 5 个重载，每个重载 0 到 3 个普通参数，当参数数量超过 3 个时引入可变参数：

```
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) { }
```

现在你知道，在所有参数数量超过 3 的调用中，只有 5% 的调用需要付出创建数组的成本。与大多数性能优化一样，这种技术使用并不广泛，但当它合适出现时，就是一个救星。

EnumSet 的静态工厂使用这种技术将创建枚举集的成本降到最低。这是适当的，因为 enum 集合为位字段提供具有性能竞争力的替代方法是至关重要的（Item-36）。

总之，当你需要定义具有不确定数量参数的方法时，可变参数是非常有用的。在可变参数之前加上任何必需的参数，并注意使用可变参数可能会引发的性能后果。

Item 54: Return empty collections or arrays, not nulls（返回空集合或数组，而不是 null）

如下的方法很常见：

```
// 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);
}
```

没有理由对没有奶酪可供购买的情况进行特殊处理。如果这样做，在客户端需要额外的代码处理可能为空的返回值，例如：

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

在几乎每次使用返回 null 来代替空集合或数组的方法时，都需要使用这种权宜之计。它很容易出错，因为编写客户端的程序员可能忘记编写特殊情况的代码来处理 null 返回。这样的错误可能会被忽略多年，因为这样的方法通常返回一个或多个对象。此外，在空容器中返回 null 会使返回容器的方法的实现复杂化。

有时有人认为，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开销。这个论点有两点是不成立的。首先，在这个级别上担心性能是不明智的，除非分析表明这个方法正是造成性能问题的真正源头（Item-67）。第二，返回空集合和数组而不分配它们是可能的。下面是返回可能为空的集合的典型代码。通常，这就是你所需要的：

```
//The right way to return a possibly empty collection
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

在不太可能的情况下，你有证据表明分配空集合会损害性能，你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（Item-17）。下面是使用 `Collections.emptyList` 完成此任务的代码。如果你要返回一个 Set，你会使用 `Collections.emptySet`；如果要返回 Map，则使用 `Collections.emptyMap`。但是请记住，这是一个优化，很少真正需要它。如果你认为你需要它，测试一下前后的表现，确保它确实有帮助：

```
// 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，而应该返回零长度的数组。通常，你应该简单地返回一个正确长度的数组，它可能是零长度。注意，我们将一个零长度的数组传递到 toArray 方法中，以指示所需的返回类型，即 Cheese[0]：

```
//The right way to return a possibly empty array
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(new Cheese[0]);
}
```

如果你认为分配零长度数组会损害性能，你可以重复返回相同的零长度数组，因为所有的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，当 cheesesInStock 为空时，这个数组将从 getCheeses 返回。不要为了提高性能而预先分配传递给 toArray 的数组。研究表明，这样做会适得其反 [Shipilev16]：

```
// Don't do this - preallocating the array harms performance!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

总之，永远不要用 null 来代替空数组或集合。它使你的 API 更难以使用，更容易出错，并且没有性能优势。

Item 55: Return optionals judiciously（明智地的返回 Optional）

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。要么抛出异常，要么返回 null（假设返回类型是对象引用类型）。这两种方法都不完美。应该为异常条件保留异常（Item-69），并且抛出异常代价高昂，因为在创建异常时捕获整个堆栈跟踪。返回 null 没有这些缺点，但是它有自己的缺点。如果方法返回 null，客户端必须包含特殊情况代码来处理 null 返回的可能性，除非程序员能够证明 null 返回是不可能的。如果客户端忽略检查 null 返回并将 null 返回值存储在某个数据结构中，那么 NullPointerException 可能会在将来的某个时间，在代码中的某个与该问题无关的位置产生。

在 Java 8 中，还有第三种方法来编写可能无法返回值的方法。`Optional<T>` 类表示一个不可变的容器，它可以包含一个非空的 T 引用，也可以什么都不包含。不包含任何内容的 Optional 被称为空。一个值被认为存在于一个非空的 Optional 中。Optional 的本质上是一个不可变的集合，它最多可以容纳一个元素。`Optional<T>` 不实现 `Collection<T>`，但原则上可以。

理论上应返回 T，但在某些情况下可能无法返回 T 的方法可以将返回值声明为 `Optional<T>` 。这允许该方法返回一个空结果来表明它不能返回有效的结果。具备 Optional 返回值的方法比抛出异常的方法更灵活、更容易使用，并且比返回 null 的方法更不容易出错。

在 Item-30 中，我们展示了根据集合元素的自然顺序计算集合最大值的方法。

```
// 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，此方法将抛出 IllegalArgumentException。我们在 Item-30 中提到，更好的替代方法是返回 `Optional<E>` 。

```
// 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);
}
```

如你所见，返回一个 Optional 是很简单的。你所要做的就是使用适当的静态工厂创建。在这个程序中，我们使用了两个静态工厂： `Optional.empty()` 返回一个空的 Optional， `Optional.of(value)` 返回一个包含给定非空值的可选值。将 null 传递给 `Optional.of(value)` 是一个编程错误。如果你这样做，该方法将通过抛出 NullPointerException 来响应。 `Optional.ofNullable(value)` 方法接受一个可能为空的值，如果传入 null，则返回一个空的 Optional。**永远不要从具备 Optional 返回值的方法返回空值:** 它违背了这个功能的设计初衷。

许多流上的 Terminal 操作返回 Optional。如果我们使用一个流来重写 max 方法，那么流版本的 max 操作会为我们生成一个 Optional（尽管我们必须传递一个显式的 comparator）：

```
// 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，如何选择是返回 Optional 而不是返回 null 或抛出异常呢？Optional 在本质上类似于已检查异常（Item-71），因为它们迫使 API 的用户面对可能没有返回值的事实。抛出未检查的异常或返回 null 允许用户忽略这种可能性，从而带来潜在的可怕后果。但是，抛出一个已检查的异常需要在客户端中添加额外的样板代码。

如果一个方法返回一个 Optional，客户端可以选择如果该方法不能返回值该采取什么操作。你可以指定一个默认值：

```
// Using an optional to provide a chosen default value
String lastWordInLexicon = max(words).orElse("No words...");
```

或者你可以抛出任何适当的异常。注意，我们传递的是异常工厂，而不是实际的异常。这避免了创建异常的开销，除非它实际被抛出：

```
// Using an optional to throw a chosen exception
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

如果你能证明一个 Optional 非空，你可以从 Optional 获取值，而不需要指定一个操作来执行，如果 Optional 是空的，但是如果你错了，你的代码会抛出一个 NoSuchElementException：

```
// Using optional when you know there's a return value
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

有时候，你可能会遇到这样一种情况：获取默认值的代价很高，除非必要，否则你希望避免这种代价。对于这些情况，Optional 提供了一个方法，该方法接受 `Supplier<T>`，并仅在必要时调用它。这个方法被称为 orElseGet，但是它可能应该被称为 orElseCompute，因为它与以 compute 开头的三个 Map 方法密切相关。有几个 Optional 的方法来处理更特殊的用例：filter、map、flatMap 和 ifPresent。在 Java 9 中，又添加了两个这样的方法：or 和 ifPresentOrElse。如果上面描述的基本方法与你的实例不太匹配，请查看这些更高级方法的文档，确认它们是否能够完成任务。

如果这些方法都不能满足你的需要，Optional 提供 `isPresent()` 方法，可以将其视为安全阀。如果 Optional 包含值，则返回 true；如果为空，则返回 false。你可以使用此方法对 Optional 结果执行任何你希望进行的处理，但请确保明智地使用它。`isPresent()` 的许多用途都可以被上面提到的方法所替代，如此生成的代码可以更短、更清晰、更符合习惯。

例如，考虑这段代码，它打印一个进程的父进程的 ID，如果进程没有父进程，则打印 N/A。该代码段使用了在 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如下形式，它使用了 Optional 的 map 函数：

```
System.out.println("Parent PID: " + ph.parent().map(h ->
String.valueOf(h.pid())).orElse("N/A"));
```

当使用流进行编程时，通常会发现你经常使用 `Stream<Optional<T>>`，并且需要一个 `Stream<T>`，其中包含非空 Optional 中的所有元素，以便继续。如果你正在使用 Java 8，下面的语句演示了如何弥补这个不足：

```
streamOfOptionals.filter(Optional::isPresent).map(Optional::get)
```

在 Java 9 中，Optional 配备了一个 `stream()` 方法。这个方法是一个适配器，它将一个 Optional 元素转换成一个包含元素的流（如果一个元素出现在 Optional 元素中），如果一个元素是空的，则一个元素都没有。与 Stream 的 flatMap 方法（Item-45）相结合，这个方法为上面的代码段提供了一个简洁的替换版本：

```
streamOfOptionals..flatMap(Optional::stream)
```

并不是所有的返回类型都能从 Optional 处理中获益。**容器类型，包括集合、Map、流、数组和 Optional，不应该封装在 Optional 中。** 你应该简单的返回一个空的 `List<T>`，而不是一个空的 `Optional<List<T>>`（Item-54）。返回空容器将消除客户端代码处理 Optional 容器的需要。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种特例，不应该被仿效。

那么，什么时候应该声明一个方法来返回 `Optional<T>` 而不是 T 呢？作为规则，**你应该声明一个方法来返回 `Optional<T>`（如果它可能无法返回结果），如果没有返回结果，客户端将不得不执行特殊处**

理。 也就是说，返回 `Optional<T>` 并不是没有代价的。Optional 对象必须分配和初始化，从 Optional 对象中读取值需要额外的间接操作。这使得 Optional 不适合在某些性能关键的情况下使用。某一特定方法是否属于这一情况只能通过仔细衡量来确定（Item-67）。

与返回基本数据类型相比，返回包含包装类的 Optional 类型的代价高得惊人，因为 Optional 类型有两个装箱级别，而不是零。因此，库设计人员认为应该为基本类型 int、long 和 double 提供类似的 `Optional<T>`。这些可选类型是 OptionalInt、OptionalLong 和 OptionalDouble。它们包含 `Optional<T>` 上的大多数方法，但不是所有方法。因此，**永远不应该返包装类的 Optional**，可能除了「次基本数据类型」，如 Boolean、Byte、Character、Short 和 Float 之外。

到目前为止，我们已经讨论了返回 Optional 并在返回后如何处理它们。我们还没有讨论其他可能的用法，这是因为大多数其他 Optional 用法都是值得疑的。例如，永远不要将 Optional 用作 Map 的值。如果这样做，则有两种方法可以表示键在 Map 中逻辑上的缺失：键可以不在 Map 中，也可以存在并映射到空的 Optional。这代表了不必要的复杂性，很有可能导致混淆和错误。更一般地说，**在集合或数组中使用 Optional 作为键、值或元素几乎都是不合适的。**

这留下了一个悬而未决的大问题。在实例字段中存储 Optional 字段是否合适？通常这是一种「代码中的不良习惯」：建议你可能应该有一个包含 Optional 字段的子类。但有时这可能是合理的。考虑 Item-2 中的 NutritionFacts 类的情况。NutritionFacts 实例包含许多不需要的字段。不能为这些字段的所有可能组合提供子类。此外，字段具有原始类型，这使得直接表示缺少非常困难。对于 NutritionFacts 最好的 API 将为每个可选字段从 getter 返回一个 Optional，因此将这些 Optional 作为字段存储在对象中是很有意义的。

总之，如果你发现自己编写的方法不能总是返回确定值，并且你认为该方法的用户在每次调用时应该考虑这种可能性，那么你可能应该让方法返回一个 Optional。但是，你应该意识到，返回 Optional 会带来实际的性能后果；对于性能关键的方法，最好返回 null 或抛出异常。最后，除了作为返回值之外，你几乎不应该以任何其他方式使用 Optional。

Item 56: Write doc comments for all exposed API elements（为所有公开的 API 元素编写文档注释）

如果 API 要可用，就必须对其进行文档化。传统上，API 文档是手工生成的，保持与代码的同步是一件苦差事。Java 编程环境使用 Javadoc 实用程序简化了这一任务。Javadoc 使用特殊格式的文档注释（通常称为文档注释）从源代码自动生成 API 文档。

虽然文档注释约定不是正式语言的一部分，但它们实际上构成了每个 Java 程序员都应该知道的 API。这些约定在如何编写文档注释的 web 页面 [Javadoc-guide] 中进行了描述。虽然自 Java 4 发布以来这个页面没有更新，但它仍然是一个非常宝贵的资源。在 Java 9 中添加了一个重要的文档标签，`{@index}`；Java 8 有一个重要标签，`{@implSpec}`；Java 5 中有两个重要标签，`{@literal}`

和 `{@code}`。上述 web 页面中缺少这些标签，但将在本项目中讨论。

要正确地编写 API 文档，必须在每个公开的类、接口、构造函数、方法和字段声明之前加上文档注释。如果一个类是可序列化的，还应该记录它的序列化形式（Item-87）。在缺少文档注释的情况下，Javadoc 所能做的最好的事情就是重新生成该声明，作为受影响的 API 元素的唯一文档。使用缺少文档注释的 API 是令人沮丧和容易出错的。公共类不应该使用默认构造函数，因为无法为它们提供文档注释。要编写可维护的代码，还应该为大多数未公开的类、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公开 API 元素那样完整。

方法的文档注释应该简洁地描述方法与其客户端之间的约定。除了为继承而设计的类中的方法（Item-19），约定应该说明方法做什么，而不是它如何做它的工作。文档注释应该列举方法的所有前置条件（这些条件必须为真，以便客户端调用它们）和后置条件（这些条件是在调用成功完成后才为真）。通常，对于 unchecked 的异常，前置条件由 `@throw` 标记隐式地描述；每个 unchecked 异常对应于一个先决条件反例。此外，可以在前置条件及其 `@param` 标记中指定受影响的参数。

除了前置条件和后置条件外，方法还应该文档中描述产生的任何副作用。副作用是系统状态的一个可观察到的变化，它不是实现后置条件所明显需要的。例如，如果一个方法启动了一个后台线程，文档应该说明。

要完整地描述方法的约定，文档注释应该为每个参数设置一个 `@param` 标记和一个 `@return` 标记（除非方法返回类型是 void），以及一个 `@throw` 标记（对于方法抛出的每个异常，无论 checked 或 unchecked）（Item-74。如果 `@return` 标记中的文本与方法的描述相同，则可以忽略它，这取决于你所遵循的标准。

按照惯例， `@param` 标记或 `@return` 标记后面的文本应该是一个名词短语，描述参数或返回值所表示的值。算术表达式很少用来代替名词短语；有关示例，请参见 BigInteger。`@throw` 标记后面的文本应该包含单词「if」，后面跟着一个描述抛出异常的条件的子句。按照惯例， `@param` 、 `@return` 或 `@throw` 标记后面的短语或子句不以句号结束。以下的文档注释展示了所有这些约定：

```
/**
* 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);
```

注意，在这个文档注释中使用 HTML 标签（`<p>` 和 `<i>`）。Javadoc 实用程序将文档注释转换为 HTML，文档注释中的任意 HTML 元素最终会出现在生成的 HTML 文档中。有时候，程序员甚至会在他们的文档注释中嵌入 HTML 表，尽管这种情况很少见。

还要注意在 `@throw` 子句中的代码片段周围使用 Javadoc 的 `{@code}` 标记。这个标记有两个目的：它使代码片段以代码字体呈现，并且它抑制了代码片段中 HTML 标记和嵌套 Javadoc 标记的处理。后一个属性允许我们在代码片段中使用小于号（`<`），即使它是一个 HTML 元字符。要在文档注释中包含多行代码，请将其包装在 `<pre>` 标签中。换句话说，在代码示例之前加上字符 `<pre>{@code}` `</pre>`。这保留了代码中的换行符，并消除了转义 HTML 元字符的需要，但不需要转义 at 符号（`@`），如果代码示例使用注释，则必须转义 at 符号（`@`）。

最后，请注意文档注释中使用的单词「this list」。按照惯例，「this」指的是调用实例方法的对象。

正如 Item-15 中提到的，当你为继承设计一个类时，你必须记录它的自用模式，以便程序员知道覆盖它的方法的语义。这些自用模式应该使用在 Java 8 中添加的 `@implSpec` 标记来记录。回想一下，普通的文档注释描述了方法与其客户机之间的约定；相反，`@implSpec` 注释描述了方法与其子类之间的约定，允许子类依赖于实现行为（如果它们继承了方法或通过 super 调用方法）。下面是它在实际使用时的样子：

```
    /**
    * 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，Javadoc 实用程序仍然忽略 `@implSpec` 标记，除非你通过命令行开关 `-tag "implSpec: a :Implementation Requirements:"`。希望在后续的版本中可以纠正这个错误。

不要忘记，你必须采取特殊的操作来生成包含 HTML 元字符的文档，比如小于号（<）、大于号（>）、与号（&）。将这些字符放到文档中最好的方法是用 {@literal} 标记包围它们，这将抑制 HTML 标记和嵌套 Javadoc 标记的处理。它类似于 {@code} 标记，只是它不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| < 1，则几何级数收敛。」 {@literal} 标签可以放在小于号周围，而不是整个不等式周围，得到的文档是相同的，但是文档注释在源代码中可读性会更差。这说明了 **一条原则，文档注释应该在源代码和生成的文档中都具备可读性。** 如果不能同时实现这两个目标，要保证生成的文档的可读性超过源代码的可读性。

每个文档注释的第一个「句子」（定义如下）成为注释所属元素的摘要描述。例如，255 页文档注释中的摘要描述是「返回列表中指定位置的元素」。摘要描述必须独立地描述它总结的元素的功能。为了避免混淆，**类或接口中的任何两个成员或构造函数都不应该具有相同的摘要描述。** 特别注意重载，对于重载，使用相同的摘要描述是很正常的（但在文档注释中是不可接受的）。

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「A college degree, such as B.S., M.S. or Ph.D.」开头的文档注释，会产生这样的概要描述「A college degree, such as B.S., M.S.」，问题在于，摘要描述在第一个句点结束，然后是空格、制表符或行结束符（或第一个块标记）[Javadoc-ref]。在这种情况下，缩写「M.S.」中的第二个句点就要接着用一个空格。最好的解决方案是用 {@literal} 标记来包围不当的句点和任何相关的文本，这样源代码中的句点后面就不会有空格了：

```
/**
* A college degree, such as B.S., {@literal M.S.} or Ph.D.
*/
public class Degree { ... }
```

将摘要描述说成是是文档注释中的第一句话有点误导人。按照惯例，它通常不是一个完整的句子。对于方法和构造函数，摘要描述应该是一个动词短语（包括任何对象），描述方法执行的动作。例如：

- 构造具有指定初始容量的空 List。
- 返回此集合中的元素数量。

如这些例子所示，应使用第三人称陈述句时态「returns the number」而不是第二人称祈使句「return the number」。

对于类、接口和字段，摘要描述应该是一个名词短语，描述由类或接口的实例或字段本身表示的事物。例如：

- 时间线上的瞬时点。
- 这个 double 类型的值比任何其它值的都更接近于圆周率（圆周长与直径之比）。

在 Java 9 中，客户端索引被添加到 Javadoc 生成的 HTML 中。这个索引以页面右上角的搜索框的形式出现，它简化了导航大型 API 文档集的任务。当你在框中键入时，你将得到一个匹配页面的下拉菜单。API 元素（如类、方法和字段）是自动索引的。有时，你可能希望索引 API 中很重要的术语。为此添加了 `{@index}` 标记。对文档注释中出现的术语进行索引，就像将其包装在这个标签中一样简单，如下面的片段所示：

```
    * This method complies with the {@index IEEE 754} standard.
```

泛型、枚举和注解在文档注释中需要特别注意。**为泛型类型或方法编写文档时，请确保说明所有类型参数：**

```
    /**
    * 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> { ... }
```

编写枚举类型的文档时，一定要说明常量 以及类型、任何公共方法。注意，如果文档很短，你可以把整个文档注释放在一行：

```
/**
* 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;
}
```

为注释类型的文档时，请确保说明全部成员 以及类型本身。用名词短语描述成员，就当它们是字段一样。对于类型的摘要描述，请使用动词短语，它表示当程序元素具有此类注解时的含义：

```
/**
* 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();
}
```

包级别的文档注释应该放在名为 package info.java 的文件中。除了这些注释之外，package info.java 必须包含一个包声明，并且可能包含关于这个声明的注释。类似地，如果你选择使用模块系统（Item-15），模块级别的注释应该放在 module-info.java 文件中。

在文档中经常忽略的 API 的两个方面是线程安全性和可序列化性。**无论类或静态方法是否线程安全，你都应该说明它的线程安全级别**，如 Item-82 所述。如果一个类是可序列化的，你应该说明它的序列化形式，如 Item-87 中所述。

Javadoc 具有「继承」方法注释的能力。如果 API 元素没有文档注释，Javadoc 将搜索最适用的文档注释，优先选择接口而不是超类。搜索算法的详细信息可以在《The Javadoc Reference Guide》[Javadoc-ref] 中找到。你还可以使用 `{@inheritDoc}` 标记从超类型继承部分文档注释。这意味着类可以复用它们实现的接口中的文档注释，而不是复制这些注释。这个工具有能力减少维护多个几乎相同的文档注释集的负担，但是它使用起来很棘手，并且有一些限制。这些细节超出了这本书的范围。

关于文档注释，有一点需要特别注意。虽然有必要为所有公开的 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，通常需要用描述 API 总体架构的外部文档来补充文档注释。如果存在这样的文档，相关的类或包文档注释应该包含到它的链接。

Javadoc 会自动检查文档是否符合本项目中提及的许多建议。在 Java 7 中，需要命令行开关 `-Xdoclint` 来获得这种特性。在 Java 8 和 Java 9 中，默认情况下启用了该机制。诸如 checkstyle 之类的 IDE 插件在检查是否符合这些建议方面做得更好 [Burn01]。还可以通过 HTML 有效性检查器运行 Javadoc 生成的 HTML 文件来降低文档注释中出现错误的可能性。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，你可以使用 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 就能够生成 HTML 5 和 HTML 4.01，尽管默认情况下它仍然生成 HTML 4.01。如果希望 Javadoc 生成 HTML5，请使用 `-html5` 命令行开关。

本条目中描述的约定涵盖了基本内容。尽管撰写本文时已经有 15 年的历史，但编写文档注释的最终指南仍然是《How to Write Doc Comments》[Javadoc-guide]。如果你遵循本条目中的指导原则，生成的文档应该提供对 API 的清晰描述。然而，唯一确定的方法是 **读取 Javadoc 实用程序生成的 web 页面。** 对于其他人将使用的每个 API 都值得这样做。正如测试程序几乎不可避免地会导致对代码的一些更改一样，阅读文档通常也会导致对文档注释的一些较小更改。

总之，文档注释是记录API的最佳、最有效的方法。应该认为，所有公开的 API 元素都必须使用文档注释，并采用符合标准约定的统一样式。请记住，在文档注释中允许使用任意 HTML 标签，并且必须转义 HTML 元字符。

Chapter 9. General Programming（通用程序设计）

Chapter 9 Introduction（章节介绍）

THIS chapter is devoted to the nuts and bolts of the language. It discusses local variables, control structures, libraries, data types, and two extralinguistic facilities: reflection and native methods. Finally, it discusses optimization and naming conventions.

本章主要讨论了 Java 语言的具体细节，包括局部变量、控制结构、类库、数据结构和两种不是由语言本身提供的机制：反射和本地方法。最后，讨论了优化和命名惯例。

Item 57: Minimize the scope of local variables（将局部变量的作用域最小化）

本条目在性质上类似于 Item-15，即「最小化类和成员的可访问性」。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。

较老的编程语言，如 C 语言，强制要求必须在代码块的头部声明局部变量，一些程序员出于习惯目前继续这样做。这是一个应改变的习惯。温馨提醒，Java 允许你在任何能够合法使用语句的地方声明变量（这与 C99 标准后 C 语言一样）。

将局部变量的作用域最小化，最具说服力的方式就是在第一次使用它的地方声明。 如果一个变量在使用之前声明了，代码会变得很混乱，这是另一件分散读者注意力的事情，因为读者正在试图弄清楚程序的功能。在使用到该变量时，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会导致其作用域开始得太早，而且结束得过晚。局部变量的范围应该从声明它的地方直到封闭块的末尾。如果变量在使用它的代码块外部声明，则在程序退出该块之后它仍然可见。如果一个变量在其预期使用区域之前或之后意外使用，其后果可能是灾难性的。

每个局部变量声明都应该包含一个初始化表达式。 如果你还没有足够的信息来合理地初始化一个变量，你应该推迟声明，直到条件满足。这个规则的一个例外是 try-catch 语句。如果一个变量被初始化为一个表达式，该表达式的计算结果可以抛出一个 checked 异常，那么该变量必须在 try 块中初始化（除非所包含的方法可以传播异常）。如果该值必须在 try 块之外使用，那么它必须在 try 块之前声明，此时它还不能「合理地初始化」。例子可参见 283 页。

循环提供了一个特殊的机会来最小化变量的范围。for 循环的传统形式和 for-each 形式都允许声明循环变量，将它们的作用域精确限制在需要它们的区域。（这个区域由循环的主体以及 for 关键字和主体之间括号中的代码组成。）因此，假设循环结束后不再需要循环变量，for 循环就优于 while 循环。

例如，下面是遍历集合的首选习惯用法（Item-58）：

```
// Preferred idiom for iterating over a collection or array
for (Element e : c) {
    ... // Do Something with e
}
```

如果你需要访问 iterator，或者调用它的 remove 方法，首选的习惯用法是使用传统的 for 循环来代替 for-each 循环：

```
// 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
}
```

要弄清楚为什么 for 循环比 while 循环更好，请考虑下面的代码片段，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粘贴错误：它计划初始化一个新的循环变量 i2，却误用了旧的变量 i，不幸的是，i 仍然在作用域中。生成的代码编译时没有错误，运行时没有抛出异常，但是它做了错误的事情。第二个循环并没有遍历 c2，而是立即终止，从而产生 c2 为空的假象。因为程序会静默地出错，所以很长一段时间内都无法检测到错误。

如果将类似的复制粘贴错误发生在 for 循环（for-each 循环或传统循环），则生成的代码甚至无法编译。对于第二个循环，第一个循环中的（或 iterator）变量已经不在作用域中。下面是它与传统 for 循环的样子：

```
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，那么发生复制粘贴错误的可能性要小得多，因为这两种循环中没有使用不同变量名称的动机。循环是完全独立的，所以复用循环（或 iterator）变量名没有害处。事实上，这样做通常很流行。for 循环相比 while 循环还有一个优点：它更短，这增强了可读性。下面是另一个循环习惯用法，它也最小化了局部变量的范围：

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    ... // Do something with i;
}
```

关于这个用法需要注意的重要一点是，它有两个循环变量，i 和 n，它们都具有完全正确的作用域。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。作为一个规则，如果循环测试涉及一个方法调用，并且保证在每次迭代中返回相同的结果，那么应该使用这个习惯用法。

最小化局部变量范围的最后一种技术是保持方法小而集中。如果在同一方法中合并两个操作，与一个操作相关的局部变量可能位于执行另一个操作的代码的范围内。为了防止这种情况发生，只需将方法分成两个部分：每个操作一个。

Item 58: Prefer for-each loops to traditional for loops（for-each 循环优于传统的 for 循环）

正如在 Item-45 中所讨论的，一些任务最好使用流来完成，其他任务最好使用 iteration。下面是使用一个传统的 for 循环来遍历一个集合：

```
// 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
}
```

这是使用传统的 for 循环来遍历数组：

```
// Not the best way to iterate over an array!
for (int i = 0; i < a.length; i++) {
    ... // Do something with a[i]
}
```

这些习惯用法比 while 循环更好（Item-57），但是它们并不完美。迭代器和索引变量都很混乱（你只需要元素）。此外，它们有出错的可能。迭代器在每个循环中出现三次，索引变量出现四次，这使得有很多机会使用到错误的变量。如果这样做，就不能保证编译器会捕捉到问题。最后，这两个循环区别很大，（第一个例子）还需要额外注意容器类型，并给类型转换增加小麻烦。

for-each 循环（官方称为「enhanced for 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出错的机会。由此产生的习惯用法同样适用于集合和数组，从而简化了将容器的实现类型从一种转换为另一种的过程：

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    ... // Do something with e
}
```

当你看到冒号 `(:)` 时，请将其读作「in」。因此，上面的循环读作「对元素集的每个元素 e 进行操作」。使用 for-each 循环不会降低性能，对于数组也是如此：它们生成的代码本质上与你手工编写的 for 循环代码相同。

当涉及到嵌套迭代时，for-each 循环相对于传统 for 循环的优势甚至更大。下面是人们在进行嵌套迭代时经常犯的一个错误：

```
// 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，不要感到难过。许多专业程序员都曾犯过这样的错误。问题是，迭代器对外部的集合 suits 调用了太多次 next 方法。它应该从外部循环调用，因此每种花色调用一次，但它是从内部循环调用的，因此每一张牌调用一次。在用完所有花色之后，循环抛出 NoSuchElementException。

如果真的很不幸，外部集合的大小是内部集合大小的几倍（可能因为它们是相同的集合），循环将正常终止，但是它不会执行你想要的操作。例如，考虑一个打印一对骰子所有可能的组合值的错误尝试：

```
// 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());
```

程序不会抛出异常，但它只打印 6 个重复数值（从「ONE ONE」到「SIX SIX」），而不是预期的 36 个组合。

要修复这些例子中的错误，必须在外部循环的作用域内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，如果使用嵌套 for-each 循环，问题就会消失。生成的代码更简洁：

```
// 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，有三种常见的情况你不应使用 for-each：

- **破坏性过滤**，如果需要遍历一个集合并删除选定元素，则需要使用显式的迭代器，以便调用其 remove 方法。通过使用 Collection 在 Java 8 中添加的 removeIf 方法，通常可以避免显式遍历。
- **转换**，如果需要遍历一个 List 或数组并替换其中部分或全部元素的值，那么需要 List 迭代器或数组索引来替换元素的值。
- **并行迭代**，如果需要并行遍历多个集合，那么需要显式地控制迭代器或索引变量，以便所有迭代器或索引变量都可以同步执行（如上述牌和骰子示例中无意中演示的错误那样）。如果发现自己处于这些情况中的任何一种，请使用普通的 for 循环，并警惕本条目中提到的陷阱。

for-each 循环不仅允许遍历集合和数组，还允许遍历实现 Iterable 接口的任何对象，该接口由一个方法组成。如下所示：

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

如果必须从头开始编写自己的 Iterator 实现，确实有点棘手，但是如果正在编写的类型表示一组元素，即使选择不让它实现 Collection，那么也应该强烈考虑让它实现 Iterable。这将允许用户使用 foreach 循环遍历类型，他们将永远感激不尽。

总之，for-each 循环在清晰度、灵活性和 bug 预防方面比传统的 for 循环更有优势，并且没有性能损失。尽可能使用 for-each 循环而不是 for 循环。

Item 59: Know and use the libraries（了解并使用库）

假设你想要生成 0 到某个上界之间的随机整数。面对这个常见任务，许多程序员会编写一个类似这样的小方法：

```
// Common but deeply flawed!
static Random rnd = new Random();
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个方法看起来不错，但它有三个缺点。首先，如果 n 是小的平方数，随机数序列会在相当短的时间内重复。第二个缺陷是，如果 n 不是 2 的幂，那么平均而言，一些数字将比其他数字更频繁地返回。如果 n 很大，这种效果会很明显。下面的程序有力地证明了这一点，它在一个精心选择的范围内生成 100 万个随机数，然后打印出有多少个数字落在范围的下半部分：

```
    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 random 方法工作正常，程序将输出一个接近 50 万的数字，但是如果运行它，你将发现它输出一个接近 666666 的数字。随机方法生成的数字中有三分之二落在其范围的下半部分！

random 方法的第三个缺陷是，在极少数情况下会返回超出指定范围的数字，这是灾难性的结果。这是因为该方法试图通过调用 `Math.abs` 将 `rnd.nextInt()` 返回的值映射到非负整数。如果 `nextInt()` 返回整数。`Integer.MIN_VALUE` 、`Math.abs` 也将返回整数。假设 n 不是 2 的幂，那么 `Integer.MIN_VALUE` 和求模运算符 `(%)` 将返回一个负数。几乎肯定的是，这会导致你的程序失败，并且这种失败可能难以重现。

要编写一个 random 方法来纠正这些缺陷，你必须对伪随机数生成器、数论和 2 的补码算法有一定的了解。幸运的是，你不必这样做（这是为你而做的成果）。它被称为 `Random.nextInt(int)`。你不必关心它如何工作的（尽管如果你感兴趣，可以研究文档或源代码）。一位具有算法背景的高级工程师花了大量时间设计、实现和测试这种方法，然后将其展示给该领域的几位专家，以确保它是正确的。然后，这个库经过 beta 测试、发布，并被数百万程序员广泛使用了近 20 年。该方法还没有发现任何缺陷，但是如果发现了缺陷，将在下一个版本中进行修复。**通过使用标准库，你可以利用编写它的专家的知识和以前使用它的人的经验。**

从 Java 7 开始，就不应该再使用 Random。在大多数情况下，**选择的随机数生成器现在是 ThreadLocalRandom。** 它能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，使用 SplittableRandom。

使用这些库的第二个好处是，你不必浪费时间为那些与你的工作无关的问题编写专门的解决方案。如果你像大多数程序员一样，那么你宁愿将时间花在应用程序上，而不是底层管道上。

使用标准库的第三个优点是，随着时间的推移，它们的性能会不断提高，而你无需付出任何努力。由于许多人使用它们，而且它们是在行业标准基准中使用的，所以提供这些库的组织有很强的动机使它们运行得更快。多年来，许多 Java 平台库都被重新编写过，有时甚至是反复编写，从而带来了显著的性能改进。使用库的第四个好处是，随着时间的推移，它们往往会获得新功能。如果一个库丢失了一些东西，开发人员社区会将其公布于众，并且丢失的功能可能会在后续版本中添加。

使用标准库的最后一个好处是，可以将代码放在主干中。这样的代码更容易被开发人员阅读、维护和复用。

考虑到所有这些优点，使用库工具而不选择专门的实现似乎是合乎逻辑的，但许多程序员并不这样做。为什么不呢？也许他们不知道库的存在。**在每个主要版本中，都会向库中添加许多特性，了解这些新增特性是值得的。** 每次发布 Java 平台的主要版本时，都会发布一个描述其新特性的 web 页面。这些页面非常值得一读 [Java8-feat, Java9-feat]。为了强调这一点，假设你想编写一个程序来打印命令行中指定的 URL 的内容（这大致是 Linux curl 命令所做的）。在 Java 9 之前，这段代码有点乏味，但是在 Java 9 中，transferTo 方法被添加到 InputStream 中。这是一个使用这个新方法执行这项任务的完整程序：

```
// 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);
    }
}
```

库太大，无法学习所有文档 [Java9-api]，但是 **每个程序员都应该熟悉 `java.lang`、`java.util` 和 `java.io` 的基础知识及其子包。** 其他库的知识可以根据需要获得。概述库中的工具超出了本项目的范围，这些工具多年来已经发展得非常庞大。

有几个图书馆值得一提。collections 框架和 streams 库（可参看 Item 45-48）应该是每个程序员的基本工具包的一部分，`java.util.concurrent` 中的并发实用程序也应该是其中的一部分。这个包既包含高级的并发工具来简化多线程的编程任务，还包含低级别的并发基本类型，允许专家们自己编写更高级的并发抽象。`java.util.concurrent` 的高级部分，在 Item-80 和 Item-81 中讨论。

有时，类库工具可能无法满足你的需求。你的需求越专门化，发生这种情况的可能性就越大。虽然你的第一个思路应该是使用这些库，但是如果你已经了解了它们在某些领域提供的功能，而这些功能不能满足你的需求，那么可以使用另一种实现。任何有限的库集所提供的功能总是存在漏洞。如果你在 Java 平台库中找不到你需要的东西，你的下一个选择应该是寻找高质量的第三方库，比如谷歌的优秀的开源 Guava 库 [Guava]。如果你无法在任何适当的库中找到所需的功能，你可能别无选择，只能自己实现它。

总而言之，不要白费力气重新发明轮子。如果你需要做一些看起来相当常见的事情，那么库中可能已经有一个工具可以做你想做的事情。如果有，使用它；如果你不知道，检查一下。一般来说，库代码可能比你自己编写的代码更好，并且随着时间的推移可能会得到改进。这并不反映你作为一个程序员的能力。规模经济决定了库代码得到的关注要远远超过大多数开发人员所能承担的相同功能。

Item 60: Avoid float and double if exact answers are required（若需要精确答案就应避免使用 float 和 double 类型）

float 和 double 类型主要用于科学计算和工程计算。它们执行二进制浮点运算，该算法经过精心设计，能够在很大范围内快速提供精确的近似值。但是，它们不能提供准确的结果，也不应该在需要精确结果的地方使用。**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 10 的任意负次幂）精确地表示为 float 或 double。

例如，假设你口袋里有 1.03 美元，你消费了 42 美分。你还剩下多少钱？下面是一个简单的程序片段，试图回答这个问题：

```
System.out.println(1.03 - 0.42);
```

不幸的是，它输出了 0.6100000000000001。这不是一个特例。假设你口袋里有一美元，你买了 9 台洗衣机，每台 10 美分。你能得到多少零钱？

```
System.out.println(1.00 - 9 * 0.10);
```

根据这个程序片段，可以得到 0.09999999999999998 美元。

你可能认为，只需在打印之前将结果四舍五入就可以解决这个问题，但不幸的是，这种方法并不总是有效。例如，假设你口袋里有一美元，你看到一个架子上有一排好吃的糖果，它们的价格仅仅是 10 美分，20 美分，30 美分，以此类推，直到 1 美元。你每买一颗糖，从 10 美分的那颗开始，直到你买不起货架上的下一颗糖。你买了多少糖果，换了多少零钱？这里有一个简单的程序来解决这个问题：

```
// 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);
}
```

如果你运行这个程序，你会发现你可以买得起三块糖，你还有 0.3999999999999999 美元。这是错误的答案！解决这个问题的正确方法是 **使用 BigDecimal、int 或 long 进行货币计算。**

这里是前一个程序的一个简单改版，使用 BigDecimal 类型代替 double。注意，使用 BigDecimal 的 String 构造函数而不是它的 double 构造函数。这是为了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，你会发现你可以买四颗糖，最终剩下 0 美元。这是正确答案。

然而，使用 BigDecimal 有两个缺点：它与原始算术类型相比很不方便，而且速度要慢得多。如果你只解决一个简单的问题，后一种缺点是无关紧要的，但前者可能会让你烦恼。

除了使用 BigDecimal，另一种方法是使用 int 或 long，这取决于涉及的数值大小，还要自己处理十进制小数点。在这个例子中，最明显的方法是用美分而不是美元来计算。下面是一个采用这种方法的简单改版：

```
    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，对于任何需要精确答案的计算，不要使用 float 或 double 类型。如果希望系统来处理十进制小数点，并且不介意不使用基本类型带来的不便和成本，请使用 BigDecimal。使用 BigDecimal 的另一个好处是，它可以完全控制舍入，当执行需要舍入的操作时，可以从八种舍入模式中进行选择。如果你使用合法的舍入行为执行业务计算，这将非常方便。如果性能是最重要的，那么你不介意自己处理十进制小数点，而且数值不是太大，可以使用 int 或 long。如果数值不超过 9 位小数，可以使用 int；如果不超过 18 位，可以使用 long。如果数量可能超过 18 位，则使用 BigDecimal。

Item 61: Prefer primitive types to boxed primitives（基本数据类型优于包装类）

Java 有一个由两部分组成的类型系统，包括基本类型（如 int、double 和 boolean）和引用类型（如 String 和 List）。每个基本类型都有一个对应的引用类型，称为包装类型。与 int、double 和 boolean 对应的包装类是 Integer、Double 和 Boolean。

正如 Item-6 中提到的，自动装箱和自动拆箱模糊了基本类型和包装类型之间的区别，但不会消除它们。这两者之间有真正的区别，重要的是你要始终意识到正在使用的是哪一种，并在它们之间仔细选择。

基本类型和包装类型之间有三个主要区别。首先，基本类型只有它们的值，而包装类型具有与其值不同的标识。换句话说，两个包装类型实例可以具有相同的值和不同的标识。第二，基本类型只有全功能值，而每个包装类型除了对应的基本类型的所有功能值外，还有一个非功能值，即 null。最后，基本类型比包装类型更节省时间和空间。如果你不小心的话，这三种差异都会给你带来真正的麻烦。

考虑下面的比较器，它的设计目的是表示 Integer 值上的升序数字排序。（回想一下，比较器的 compare 方法返回一个负数、零或正数，这取决于它的第一个参数是小于、等于还是大于第二个参数。）你不需要在实际使用中编写这个比较器，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可以工作，它将通过许多测试。例如，它可以与 `Collections.sort` 一起使用，以正确地排序一个百万元素的 List，无论该 List 是否包含重复的元素。但这个比较存在严重缺陷。要使自己相信这一点，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)`，所以这个表达式的值应该是 0，但它是 1，这表明第一个 Integer 值大于第二个！

那么问题出在哪里呢？naturalOrder 中的第一个测试工作得很好。计算表达式 `i < j` 会使 i 和 j 引用的 Integer 实例自动拆箱；也就是说，它提取它们的基本类型值。计算的目的是检查得到的第一个 int 值是否小于第二个 int 值。但假设它不是。然后，下一个测试计算表达式 `i==j`，该表达式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，这个比较将返回 false，比较器将错误地返回 1，表明第一个整型值大于第二个整型值。**将 `==` 操作符应用于包装类型几乎都是错误的。**

在实际使用中，如果你需要一个比较器来描述类型的自然顺序，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，或者对基本类型使用静态比较方法（Item-14）。也就是说，你可以通过添加两个局部变量来存储基本类型 int 值，并对这些变量执行所有的比较，从而修复损坏的比较器中的问题。这避免了错误的标识比较：

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

接下来，考虑一下这个有趣的小程序：

```
public class Unbelievable {
static Integer i;
public static void main(String[] args) {
    if (i == 42)
        System.out.println("Unbelievable");
    }
}
```

不，它不会打印出令人难以置信的东西，但它的行为很奇怪。它在计算表达式 `i==42` 时抛出 NullPointerException。问题是，i 是 Integer，而不是 int 数，而且像所有非常量对象引用字段一样，它的初值为 null。当程序计算表达式 `i==42` 时，它是在比较 Integer 与 int。**在操作中混合使用基本类型和包装类型时，包装类型就会自动拆箱**，这种情况无一例外。如果一个空对象引用自动拆箱，那么你将得到一个 NullPointerException。正如这个程序所演示的，它几乎可以在任何地方发生。修复这个问题非常简单，只需将 i 声明为 int 而不是 Integer。

最后，考虑 Item-6 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);
}
```

这个程序比它预期的速度慢得多，因为它意外地声明了一个局部变量（sum），它是包装类型 Long，而不是基本类型 long。程序在没有错误或警告的情况下编译，变量被反复装箱和拆箱，导致产生明显的性能下降。

在本条目中讨论的所有三个程序中，问题都是一样的：程序员忽略了基本类型和包装类型之间的区别，并承担了恶果。在前两个项目中，结果是彻底的失败；第三个例子还产生了严重的性能问题。

那么，什么时候应该使用包装类型呢？它们有几个合法的用途。第一个是作为集合中的元素、键和值。不能将基本类型放在集合中，因此必须使用包装类型。这是一般情况下的特例。在参数化类型和方法（Chapter 5）中，必须使用包装类型作为类型参数，因为 Java 不允许使用基本类型。例如，不能将变量声明为 `ThreadLocal<int>` 类型，因此必须使用 `ThreadLocal<Integer>` 。最后，在进行反射方法调用时，必须使用包装类型（Item-65）。

总之，只要有选择，就应该优先使用基本类型，而不是包装类型。基本类型更简单、更快。如果必须使用包装类型，请小心！**自动装箱减少了使用包装类型的冗长，但没有减少危险。** 当你的程序使用 `==` 操作符比较两个包装类型时，它会执行标识比较，这几乎肯定不是你想要的。当你的程序执行包含包装类型和基本类型的混合类型计算时，它将进行拆箱，**当你的程序执行拆箱时，将抛出 NullPointerException。** 最后，当你的程序将基本类型装箱时，可能会导致代价高昂且不必要的对象创建。

Item 62: Avoid strings where other types are more appropriate（其他类型更合适时应避免使用字符串）

字符串被设计用来表示文本，它们在这方面做得很好。因为字符串是如此常见，并且受到 Java 的良好支持，所以很自然地会将字符串用于其他目的，而不是它们适用的场景。本条目讨论了一些不应该使用字符串的场景。

字符串是其他值类型的糟糕替代品。当一段数据从文件、网络或键盘输入到程序时，它通常是字符串形式的。有一种很自然的倾向是保持这种格式不变，但是这种倾向只有在数据本质上是文本的情况下才合理。如果是数值类型，则应将其转换为适当的数值类型，如 int、float 或 BigInteger。如果是问题的答

案，如「是」或「否」这类形式，则应将其转换为适当的枚举类型或布尔值。更一般地说，如果有合适的值类型，无论是基本类型还是对象引用，都应该使用它；如果没有，你应该写一个。虽然这条建议似乎很多余，但经常被违反。

字符串是枚举类型的糟糕替代品。 正如 Item-34 中所讨论的，枚举类型常量比字符串更适合于枚举类型常量。

字符串是聚合类型的糟糕替代品。 如果一个实体有多个组件，将其表示为单个字符串通常是一个坏主意。例如，下面这行代码来自一个真实的系统标识符，它的名称已经被更改，以免引发罪责：

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

这种方法有很多缺点。如果用于分隔字段的字符出现在其中一个字段中，可能会导致混乱。要访问各个字段，你必须解析字符串，这是缓慢的、冗长的、容易出错的过程。你不能提供 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，通常是一个私有静态成员类（Item-24）。

字符串不能很好地替代 capabilities。 有时，字符串用于授予对某些功能的访问权。例如，考虑线程本地变量机制的设计。这样的机制提供了每个线程都有自己的变量值。自 1.2 版以来，Java 库就有了一个线程本地变量机制，但在此之前，程序员必须自己设计。许多年前，当面临设计这样一个机制的任务时，有人提出了相同的设计，其中客户端提供的字符串键，用于标识每个线程本地变量：

```
// 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);
}
```

这种方法的问题在于，字符串键表示线程本地变量的共享全局名称空间。为了使这种方法有效，客户端提供的字符串键必须是惟一的：如果两个客户端各自决定为它们的线程本地变量使用相同的名称，它们无意中就会共享一个变量，这通常会导致两个客户端都失败。而且，安全性很差。恶意客户端可以故意使用与另一个客户端相同的字符串密钥来非法访问另一个客户端的数据。

这个 API 可以通过用一个不可伪造的键（有时称为 capability）替换字符串来修复：

```
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 API 中基于字符串的两个问题，但是你可以做得更好。你不再真正需要静态方法。它们可以变成键上的实例方法，此时键不再是线程局部变量：而是线程局部变量。此时，顶层类不再为你做任何事情，所以你可以删除它，并将嵌套类重命名为 ThreadLocal：

```
public final class ThreadLocal {
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

这个 API 不是类型安全的，因为在从线程本地变量检索值时，必须将值从 Object 转换为它的实际类型。原始的基于 String 类型 API 的类型安全是不可能实现的，基于键的 API 的类型安全也是很难实现的，但是通过将 ThreadLocal 作为一个参数化的类来实现这个 API 的类型安全很简单（Item-29）：

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

粗略地说，这就是 `java.lang.ThreadLocal` 提供的 API，除了解决基于字符串的问题之外，它比任何基于键的 API 都更快、更优雅。

总之，当存在或可以编写更好的数据类型时，应避免将字符串用来表示对象。如果使用不当，字符串比其他类型更麻烦、灵活性更差、速度更慢、更容易出错。字符串经常被误用的类型包括基本类型、枚举和聚合类型。

Item 63: Beware the performance of string concatenation（当心字符串连接引起的性能问题）

字符串连接操作符 `(+)` 是将几个字符串组合成一个字符串的简便方法。对于生成单行输出或构造一个小的、固定大小的对象的字符串表示形式，它是可以的，但是它不能伸缩。使用 **字符串串联运算符重复串联 n 个字符串需要 n 的平方级时间。** 这是字符串不可变这一事实导致的结果（Item-17）。当连接两个字符串时，将复制这两个字符串的内容。

例如，考虑这个方法，它通过将每个账单项目重复连接到一行来构造账单语句的字符串表示：

```
// 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;
}
```

如果项的数量很大，则该方法的性能非常糟糕。**要获得能接受的性能，请使用 StringBuilder 代替 String** 来存储正在构建的语句：

```
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，为了使字符串连接更快，已经做了大量工作，但是这两个方法在性能上的差异仍然很大：如果 numItems 返回 100，lineForItem 返回 80 个字符串，那么第二个方法在我的机器上运行的速度是第一个方法的 6.5 倍。由于第一种方法在项目数量上是平方级的，而第二种方法是线性的，所以随着项目数量的增加，性能差异会变得越来越大。注意，第二个方法预先分配了一个足够大的 StringBuilder 来保存整个结果，从而消除了自动增长的需要。即使使用默认大小的 StringBuilder，它仍然比第一个方法快 5.5 倍。

道理很简单：**不要使用字符串连接操作符合并多个字符串**，除非性能无关紧要。否则使用 StringBuilder 的 append 方法。或者，使用字符数组，再或者一次只处理一个字符串，而不是组合它们。

Item 64: Refer to objects by their interfaces（通过接口引用对象）

Item-51 指出，应该使用接口而不是类作为参数类型。更一般地说，你应该优先使用接口而不是类来引用对象。**如果存在合适的接口类型，那么应该使用接口类型声明参数、返回值、变量和字段。** 惟一真正需要引用对象的类的时候是使用构造函数创建它的时候。为了具体说明这一点，考虑 LinkedHashSet 的情况，它是 Set 接口的一个实现。声明时应养成这样的习惯：

```
// Good - uses interface as type
Set<Son> sonSet = new LinkedHashSet<>();
```

而不是这样：

```
// Bad - uses class as type!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

如果你养成了使用接口作为类型的习惯，那么你的程序将更加灵活。 如果你决定要切换实现，只需在构造函数中更改类名（或使用不同的静态工厂）。例如，第一个声明可以改为：

```
Set<Son> sonSet = new HashSet<>();
```

所有的代码都会继续工作。周围的代码不知道旧的实现类型，所以它不会在意更改。

有一点值得注意：如果原实现提供了接口的通用约定不需要的一些特殊功能，并且代码依赖于该功能，那么新实现提供相同的功能就非常重要。例如，如果围绕第一个声明的代码依赖于 LinkedHashSet 的排序策略，那么在声明中将 HashSet 替换为 LinkedHashSet 将是不正确的，因为 HashSet 不保证迭代顺序。

那么，为什么要更改实现类型呢？因为第二个实现比原来的实现提供了更好的性能，或者因为它提供了原来的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 将为迭代提供更好的性能和与键的自然顺序，但是你只能在键类型为 enum 类型的情况下使用 EnumMap。将 HashMap 更改为 LinkedHashMap 将提供可预测的迭代顺序，性能与 HashMap 相当，而不需要对键类型作出任何特殊要求。

你可能认为使用变量的实现类型声明变量是可以的，因为你可以同时更改声明类型和实现类型，但是不能保证这种更改会正确编译程序。如果客户端代码对原实现类型使用了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实现类型的方法，那么在进行此更改之后，代码将不再编译。使用接口类型声明变量可以保持一致。

如果没有合适的接口存在，那么用类引用对象是完全合适的。 例如，考虑值类，如 String 和 BigInteger。值类很少在编写时考虑到多个实现。它们通常是 final 的，很少有相应的接口。使用这样的值类作为参数、变量、字段或返回类型非常合适。

没有合适接口类型的第二种情况是属于框架的对象，框架的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，那么最好使用相关的基类来引用它，这通常是抽象的，而不是使用它的实现类。在 java.io 类中许多诸如 OutputStream 之类的就属于这种情况。

没有合适接口类型的最后一种情况是，实现接口但同时提供接口中不存在的额外方法的类，例如，PriorityQueue 有一个在 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，这种情况应该非常少见。

这三种情况并不是面面俱到的，而仅仅是为了传达适合通过类引用对象的情况。在实际应用中，给定对象是否具有适当的接口应该是显而易见的。如果是这样，如果使用接口引用对象，程序将更加灵活和流行。**如果没有合适的接口，就使用类层次结构中提供所需功能的最底层的类**

Item 65: Prefer interfaces to reflection（接口优于反射）

核心反射机制 `java.lang.reflect` 提供对任意类的编程访问。给定一个 Class 对象，你可以获得 Constructor、Method 和 Field 实例，分别代表了该 Class 实例所表示的类的构造器、方法和字段。这些对象提供对类的成员名、字段类型、方法签名等的编程访问。

此外，Constructor、Method 和 Field 实例允许你反射性地操作它们的底层对应项：你可以通过调用 Constructor、Method 和 Field 实例上的方法，可以构造底层类的实例、调用底层类的方法，并访问底层类中的字段。例如，Method.invoke 允许你在任何类的任何对象上调用任何方法（受默认的安全约束）。反射允许一个类使用另一个类，即使在编译前者时后者并不存在。然而，这种能力是有代价的：

- 你失去了编译时类型检查的所有好处，包括异常检查。如果一个程序试图反射性地调用一个不存在

的或不可访问的方法，它将在运行时失败，除非你采取了特殊的预防措施。
- 执行反射访问所需的代码既笨拙又冗长。写起来很乏味，读起来也很困难。
- 性能降低。反射方法调用比普通方法调用慢得多。到底慢了多少还很难说，因为有很多因素在起作用。在我的机器上，调用一个没有输入参数和返回 int 类型的方法时，用反射执行要慢 11 倍。

有一些复杂的应用程序需要反射。包括代码分析工具和依赖注入框架。即使是这样的工具，随着它的缺点变得越来越明显，人们也在逐渐远离并反思这种用法。如果你对应用程序是否需要反射有任何疑问，那么它可能不需要。

通过非常有限的形式使用反射，你可以获得反射的许多好处，同时花费的代价很少。 对于许多程序，它们必须用到在编译时无法获取的类，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，**可以用反射方式创建实例，并通过它们的接口或超类正常地访问它们。**

例如，这是一个创建 `Set<String>` 实例的程序，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出来。不管第一个参数是什么，程序都会打印剩余的参数，并去掉重复项。然而，打印这些参数的顺序取决于第一个参数中指定的类。如果你指定 `java.util.HashSet`，它们显然是随机排列的；如果你指定 `java.util.TreeSet`，它们是按字母顺序打印的，因为 TreeSet 中的元素是有序的：

```
// 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小把戏，但它演示的技术非常强大。这个程序可以很容易地转换成一个通用的集合测试器，通过积极地操作一个或多个实例并检查它们是否遵守 Set 接口约定来验证指定的 Set 实现。类似地，它可以变成一个通用的集合性能分析工具。事实上，该技术足够强大，可以实现一个成熟的服务提供者框架（Item-1）。

这个例子也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，如果没有使用反射实例化，所有这些异常都将是编译时错误。（有趣的是，你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，根据类的名称生成类的实例需要 25 行冗长的代码，而构造函数调用只需要一行。通过捕获 ReflectiveOperationException（Java 7 中引入的各种反射异常的超类），可以减少程序的长度。这两个缺点都只限于实例化对象的程序部分。实例化后，与任何其他 Set 实例将难以区分。在实际的程序中，通过这种限定使用反射的方法，大部分代码可以免受影响。

如果编译此程序，将得到 unchecked 的强制转换警告。这个警告是合法的，即使指定的类不是 Set 实现，`Class<? extends Set<String>>` 也会成功，在这种情况下，程序在实例化类时抛出 ClassCastException。要了解如何抑制警告，请阅读 Item-27。

反射的合法用途（很少）是管理类对运行时可能不存在的其他类、方法或字段的依赖关系。如果你正在编写一个包，并且必须针对其他包的多个版本运行，此时反射将非常有用。该技术是根据支持包所需的最小环境（通常是最老的版本）编译包，并反射性地访问任何较新的类或方法。如果你试图访问的新类或方法在运行时不存在，要使此工作正常进行，则必须采取适当的操作。适当的操作可能包括使用一些替代方法来完成相同的目标，或者使用简化的功能进行操作。

总之，反射是一种功能强大的工具，对于某些复杂的系统编程任务是必需的，但是它有很多缺点。如果编写的程序必须在编译时处理未知的类，则应该尽可能只使用反射实例化对象，并使用在编译时已知的接口或超类访问对象。

Item 66: Use native methods judiciously（明智地使用本地方法）

Java 本地接口（JNI）允许 Java 程序调用本地方法，这些方法是用 C 或 C++ 等本地编程语言编写的。从历史上看，本地方法主要有三种用途。它们提供对特定于平台的设施（如注册中心）的访问。它们提供对现有本地代码库的访问，包括提供对遗留数据访问。最后，本地方法可以通过本地语言编写应用程序中注重性能的部分，以提高性能。

使用本地方法访问特定于平台的机制是合法的，但是很少有必要：随着 Java 平台的成熟，它提供了对许多以前只能在宿主平台中上找到的特性。例如，Java 9 中添加的流 API 提供了对 OS 流程的访问。在 Java 中没有等效库时，使用本地方法来使用本地库也是合法的。

为了提高性能，很少建议使用本地方法。 在早期版本（Java 3 之前），这通常是必要的，但是从那时起 JVM 变得更快了。对于大多数任务，现在可以在 Java 中获得类似的性能。例如，在版本 1.1 中添加了 `java.math`，BigInteger 是在一个用 C 编写的快速多精度运算库的基础上实现的。在当时，为了获得足够的性能这样做是必要的。在 Java 3 中，BigInteger 则完全用 Java 重写了，并且进行了性能调优，新的版本比原来的版本更快。

这个故事的一个可悲的结尾是，除了在 Java 8 中对大数进行更快的乘法运算之外，BigInteger 此后几乎没有发生什么变化。在此期间，对本地库的工作继续快速进行，尤其是 GNU 多精度算术库（GMP）。需要真正高性能多精度算法的 Java 程序员现在可以通过本地方法使用 GMP [Blum14]。

使用本地方法有严重的缺点。由于本地语言不安全（Item-50），使用本地方法的应用程序不再能免受内存毁坏错误的影响。由于本地语言比 Java 更依赖于平台，因此使用本地方法的程序的可移植性较差。它们也更难调试。如果不小心，本地方法可能会降低性能，因为垃圾收集器无法自动跟踪本地内存使用情况（Item-8），而且进出本地代码会产生相关的成本。最后，本地方法需要「粘合代码」，这很难阅读，而且编写起来很乏味。

总之，在使用本地方法之前要三思。一般很少需要使用它们来提高性能。如果必须使用本地方法来访问底层资源或本地库，请尽可能少地使用本地代码，并对其进行彻底的测试。本地代码中的一个错误就可以破坏整个应用程序。

Item 67: Optimize judiciously（明智地进行优化）

有三条关于优化的格言是每个人都应该知道的：

比起其他任何单一的原因（包括盲目的愚蠢），很多计算上的过失都被归咎于效率（不一定能实现）。

—William A. Wulf [Wulf72]

不要去计较效率上的一些小小的得失，在 97% 的情况下，不成熟的优化才是一切问题的根源。

—Donald E. Knuth [Knuth74]

在优化方面，我们应该遵守两条规则：

规则 1：不要进行优化。

规则 2 （仅针对专家）：还是不要进行优化，也就是说，在你还没有绝对清晰的未优化方案之前，请不要进行优化。

—M. A. Jackson [Jackson75]

所有这些格言都比 Java 编程语言早了 20 年。它们告诉我们关于优化的一个深刻的事实：很容易弊大于利，尤其是如果过早地进行优化。在此过程中，你可能会生成既不快速也不正确且无法轻松修复的软件。

不要为了性能而牺牲合理的架构。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，它的架构将允许它被优化。好的程序体现了信息隐藏的原则：在可能的情况下，它们在单个组件中本地化设计决策，因此可以在不影响系统其余部分的情况下更改单个决策（Item-15）。

这并不意味着在程序完成之前可以忽略性能问题。实现上的问题可以通过以后的优化来解决，但是对于架构缺陷，如果不重写系统，就不可能解决限制性能的问题。在系统完成之后再改变设计的某个基本方面可能导致结构不良的系统难以维护和进化。因此，你必须在设计过程中考虑性能。

尽量避免限制性能的设计决策。 设计中最难以更改的组件是那些指定组件之间以及与外部世界的交互的组件。这些设计组件中最主要的是 API、线路层协议和持久数据格式。这些设计组件不仅难以或不可能在事后更改，而且所有这些组件都可能对系统能够达到的性能造成重大限制。

考虑API设计决策的性能结果。 使公共类型转化为可变，可能需要大量不必要的防御性复制（Item-50）。类似地，在一个公共类中使用继承（在这个类中组合将是合适的）将该类永远绑定到它的超类，这会人为地限制子类的性能（Item-18）。最后一个例子是，在 API 中使用实现类而不是接口将你绑定到特定的实现，即使将来可能会编写更快的实现也无法使用（Item-64）。

API 设计对性能的影响是非常实际的。考虑 `java.awt.Component` 中的 getSize 方法。该性能很关键方法返回 Dimension 实例的决定，加上维度实例是可变的决定，强制该方法的任何实现在每次调用时分配一个新的 Dimension 实例。尽管在现代 VM 上分配小对象并不昂贵，但不必要地分配数百万个对象也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（Item-17）；或者，getSize 可以被返回 Dimension 对象的原始组件的两个方法所替代。事实上，出于性能原因，在 Java 2 的组件中添加了两个这样的方法。然而，现有的客户端代码仍然使用 getSize 方法，并且仍然受到原始 API 设计决策的性能影响。

幸运的是，通常情况下，好的 API 设计与好的性能是一致的。**为了获得良好的性能而改变 API 是一个非常糟糕的想法。** 导致你改变 API 的性能问题，可能在平台或其他底层软件的未来版本中消失，但是改变的 API 和随之而来的问题将永远伴随着你。

一旦你仔细地设计了你的程序，成了一个清晰、简洁、结构良好的实现，那么可能是时候考虑优化了，假设此时你还不满意程序的性能。

记得 Jackson 的两条优化规则是「不要做」和「（只针对专家）」。先别这么做。他本可以再加一个：**在每次尝试优化之前和之后测量性能。** 你可能会对你的发现感到惊讶。通常，试图做的优化通常对于性能并没有明显的影响；有时候，还让事情变得更糟。主要原因是很难猜测程序将时间花费在哪里。程序中你认为很慢的部分可能并没有问题，在这种情况下，你是在浪费时间来优化它。一般认为，程序将 90% 的时间花费在了 10% 的代码上。

分析工具可以帮助你决定将优化工作的重点放在哪里。这些工具提供了运行时信息，比如每个方法大约花费多少时间以及调用了多少次。除了关注你的调优工作之外，这还可以提醒你是否需要改变算法。如果程序中潜伏着平方级（或更差）的算法，那么再多的调优也无法解决这个问题。你必须用一个更有效的算法来代替这个算法。系统中的代码越多，使用分析器就越重要。这就像大海捞针：大海越大，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，提供了对 Java 代码性能无与伦比的预测性。

与 C 和 C++ 等更传统的语言相比，Java 甚至更需要度量尝试优化的效果，因为 Java 的性能模型更弱：各种基本操作的相对成本没有得到很好的定义。程序员编写的内容和 CPU 执行的内容之间的「抽象鸿沟」更大，这使得可靠地预测优化的性能结果变得更加困难。有很多关于性能的传说流传开来，但最终被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清，而且在不同的实现、不同的发布版本、不同的处理器之间都有所不同。如果你要在多个实现或多个硬件平台上运行程序，那么度量优化对每个平台的效果是很重要的。有时候，你可能会被迫在不同实现或硬件平台上的性能之间进行权衡。

自本条目首次编写以来的近 20 年里，Java 软件栈的每个组件都变得越来越复杂，从处理器到 vm 再到库，Java 运行的各种硬件都有了极大的增长。所有这些加在一起，使得 Java 程序的性能比 2001 年更难以预测，而对它进行度量的需求也相应增加。

总而言之，不要努力写快的程序，要努力写好程序；速度自然会提高。但是在设计系统时一定要考虑性能，特别是在设计API、线路层协议和持久数据格式时。当你完成了系统的构建之后，请度量它的性能。如果足够快，就完成了。如果没有，利用分析器找到问题的根源，并对系统的相关部分进行优化。第一步是检查算法的选择：再多的底层优化也不能弥补算法选择的不足。根据需要重复这个过程，在每次更改之后测量性能，直到你满意为止。

Item 68: Adhere to generally accepted naming conventions（遵守被广泛认可的命名约定）

Java 平台有一组完善的命名约定，其中许多约定包含在《The Java Language Specification》[JLS, 6.1]。不严格地讲，命名约定分为两类：排版和语法。

有少量的与排版有关的命名约定，包括包、类、接口、方法、字段和类型变量。如果没有很好的理由，你不应该违反它们。如果 API 违反了这些约定，那么它可能很难使用。如果实现违反了这些规则，可能很难维护。在这两种情况下，违规都有可能使其他使用代码的程序员感到困惑和恼怒，并使他们做出错误的假设，从而导致错误。本条目概述了各项约定。

包名和模块名应该是分层的，组件之间用句点分隔。组件应该由小写字母组成，很少使用数字。任何在你的组织外部使用的包，名称都应该以你的组织的 Internet 域名开头，并将组件颠倒过来，例如，edu.cmu、com.google、org.eff。以 java 和 javax 开头的标准库和可选包是这个规则的例外。用户不能创建名称以 java 或 javax 开头的包或模块。将 Internet 域名转换为包名前缀的详细规则可以在《The Java Language Specification》[JLS, 6.1] 中找到。

包名的其余部分应该由描述包的一个或多个组件组成。组件应该很短，通常为 8 个或更少的字符。鼓励使用有意义的缩写，例如 util 而不是 utilities。缩写词是可以接受的，例如 awt。组件通常应该由一个单词或缩写组成。

除了 Internet 域名之外，许多包的名称只有一个组件。附加组件适用于大型工具包，这些工具包的大小要求将其分解为非正式的层次结构。例如 `javax.util` 包具有丰富的包层次结构，包的名称如 `java.util.concurrent.atomic`。这样的包称为子包，尽管 Java 几乎不支持包层次结构。

类和接口名称，包括枚举和注释类型名称，应该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。除了缩略语和某些常见的缩略语，如 max 和 min，缩略语应该避免使用。缩略语应该全部大写，还是只有首字母大写，存在一些分歧。虽然有些程序员仍然使用大写字母，但支持只将第一个字母大写的理由很充分：即使多个首字母缩写连续出现，你仍然可以知道一个单词从哪里开始，下一个单词从哪里结束。你希望看到哪个类名，HTTPURL 还是 HttpUrl?

方法和字段名遵循与类和接口名相同的排版约定，除了方法或字段名的第一个字母应该是小写，例如 remove 或 ensureCapacity。如果方法或字段名的首字母缩写出现在第一个单词中，那么它应该是小写的。

前面规则的唯一例外是「常量字段」，它的名称应该由一个或多个大写单词组成，由下划线分隔，例如 VALUES 或 NEGATIVE_INFINITY。常量字段是一个静态的 final 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)，那么它就是常量字段。例如，枚举常量是常量字段。如果静态 final 字段有一个可变的引用类型，那么如果所引用的对象是不可变的，那么它仍然可以是一个常量字段。注意，常量字段是唯一推荐使用下划线用法的。

局部变量名与成员名具有类似的排版命名约定，但允许使用缩写，也允许使用单个字符和短字符序列，它们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。它们的命名应该比普通的局部变量谨慎得多，因为它们的名称是方法文档的组成部分。

类型参数名通常由单个字母组成。最常见的是以下五种类型之一：T 表示任意类型，E 表示集合的元素类型，K 和 V 表示 Map 的键和值类型，X 表示异常。函数的返回类型通常为 R。任意类型的序列可以是 T、U、V 或 T1、T2、T3。

为了快速参考，下表显示了排版约定的示例。

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

语法命名约定比排版约定更灵活，也更有争议。包没有语法命名约定。可实例化的类，包括枚举类型，通常使用一个或多个名词短语来命名，例如 Thread、PriorityQueue 或 ChessPiece。不可实例化的实用程序类（Item-4）通常使用复数名词来命名，例如 collector 或 Collections。接口的名称类似于类，例如集合或比较器，或者以 able 或 ible 结尾的形容词，例如 Runnable、Iterable 或 Accessible。因为注解类型有很多的用途，所以没有哪部分占主导地位。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。返回布尔值的方法的名称通常以单词 is 或 has（通常很少用）开头，后面跟一个名词、一个名词短语，或者任何用作形容词的单词或短语，例如 isDigit、isProbablePrime、isEmpty、isEnabled 或 hasSiblings。

返回被调用对象的非布尔函数或属性的方法通常使用以 get 开头的名词、名词短语或动词短语来命名，例如 size、hashCode 或 getTime。有一种说法是，只有第三种形式（以 get 开头）才是可接受的，但这种说法几乎没有根据。前两种形式的代码通常可读性更强，例如：

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

以 get 开头的表单起源于基本过时的 Java bean 规范，该规范构成了早期可复用组件体系结构的基础。有一些现代工具仍然依赖于 bean 命名约定，你应该可以在任何与这些工具一起使用的代码中随意使用它。如果类同时包含相同属性的 setter 和 getter，则遵循这种命名约定也有很好的先例。在本例中，这两个方法通常被命名为 getAttribute 和 setAttribute。

一些方法名称值得特别注意。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。返回与接收对象类型不同的视图（Item-6）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基本类型的方法通常称为类型值，例如 intValue。静态工厂的常见名称包括 from、of、valueOf、instance、getInstance、newInstance、getType 和 newType（Item-1，第 9 页）。

字段名的语法约定没有类、接口和方法名的语法约定建立得好，也不那么重要，因为设计良好的 API 包含很少的公开字段。类型为 boolean 的字段的名称通常类似于 boolean 访问器方法，省略了初始值「is」，例如 initialized、composite。其他类型的字段通常用名词或名词短语来命名，如 height、digits 和 bodyStyle。局部变量的语法约定类似于字段的语法约定，但要求更少。

总之，将标准命名约定内在化，并将其作为第二性征来使用。排版习惯是直接的，而且在很大程度上是明确的；语法惯例更加复杂和松散。引用《The JavaLanguage Specification》[JLS, 6.1] 中的话说，「如果长期以来的传统用法要求不遵循这些约定，就不应该盲目地遵循这些约定。」，应使用常识判断。

Chapter 10. Exceptions（异常）

Chapter 10 Introduction（章节介绍）

WHEN used to best advantage, exceptions can improve a program's readability, reliability, and maintainability. When used improperly, they can have the opposite effect. This chapter provides guidelines for using exceptions effectively.

当充分利用好异常时，可以提高程序的可读性、可靠性和可维护性。如果使用不当，则会产生负面效果。本章提供了有效使用异常的指南。

Item 69: Use exceptions only for exceptional conditions（仅在确有异常条件下使用异常）

你可能会偶然发现这样一段代码：

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        range[i++].climb();
    }
    catch (ArrayIndexOutOfBoundsException e) {
}
```

这段代码是做什么的？从表面上看，一点也不明显，这足以成为不使用它的充分理由（Item-67）。事实证明，这是一个用于遍历数组的元素的非常糟糕的习惯用法。当试图访问数组边界之外的数组元素时，通过抛出、捕获和忽略 ArrayIndexOutOfBoundsException 来终止无限循环。如下循环遍历数组的标准习惯用法，任何 Java 程序员都可以立即识别它：

```
for (Mountain m : range)
    m.climb();
```

那么，为什么会有人使用基于异常的循环而不使用习惯的循环模式呢？由于 VM 检查所有数组访问的边界，所以误认为正常的循环终止测试被编译器隐藏了，但在 for-each 循环中仍然可见，这无疑是多余的，应该避免，因此利用错误判断机制来提高性能是错误的。这种思路有三点误区：

- 因为异常是为特殊情况设计的，所以 JVM 实现几乎不会让它们像显式测试一样快。

- 将代码放在 try-catch 块中会抑制 JVM 可能执行的某些优化。
- 遍历数组的标准习惯用法不一定会导致冗余检查。许多 JVM 实现对它们进行了优化。

事实上，基于异常的用法比标准用法慢得多。在我的机器上，用 100 个元素的数组测试，基于异常的用法与标准用法相比速度大约慢了两倍。

基于异常的循环不仅混淆了代码的目的，降低了代码的性能，而且不能保证它能正常工作。如果循环中存在 bug，使用异常进行流程控制会掩盖该 bug，从而大大增加调试过程的复杂性。假设循环体中的计算步骤调用一个方法，该方法对一些不相关的数组执行越界访问。如果使用合理的循环习惯用法，该 bug 将生成一个未捕获的异常，导致线程立即终止，并带有完整的堆栈跟踪。相反，如果使用了基于异常的循环，当捕获与 bug 相关的异常时，会将其误判为正常的循环终止条件。

这个案例的寓意很简单：**顾名思义，异常只适用于确有异常的情况；它们不应该用于一般的控制流程。**更进一步说，使用标准的、易于识别的习惯用法，而不是声称能够提供更好性能的过于抖机灵的技术。即使性能优势是真实存在的，但在稳步改进平台实现的前提下，这种优势也并不可靠。而且，来自抖机灵的技术存在的细微缺陷和维护问题肯定会继续存在。

这个原则对 API 设计也有影响。一个设计良好的 API 不能迫使其客户端为一般的控制流程使用异常。只有在某些不可预知的条件下才能调用具有「状态依赖」方法的类，通常应该有一个单独的「状态测试」方法，表明是否适合调用「状态依赖」方法。例如，Iterator 接口具有「状态依赖」的 next 方法和对应的「状态测试」方法 hasNext。这使得传统 for 循环（在 for-each 循环内部也使用了 hasNext 方法）在集合上进行迭代成为标准习惯用法：

```
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，客户端将被迫这样做：

```
// 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) {
}
```

这与一开始举例的对数组进行迭代的例子非常相似，除了冗长和误导之外，基于异常的循环执行效果可能很差，并且会掩盖系统中不相关部分的 bug。

提供单独的「状态测试」方法的另一种方式，就是让「状态依赖」方法返回一个空的 Optional 对象（Item-55），或者在它不能执行所需的计算时返回一个可识别的值，比如 null。

有一些指导原则，帮助你在「状态测试」方法、Optional、可识别的返回值之间进行选择。（1）如果要在没有外部同步的情况下并发地访问对象，或者受制于外部条件的状态转换，则必须使用 Optional 或可识别的返回值，因为对象的状态可能在调用「状态测试」方法与「状态依赖」方法的间隔中发生变化。（2）如果一个单独的「状态测试」方法重复「状态依赖」方法的工作，从性能问题考虑，可能要求使用 Optional 或可识别的返回值。（3）在所有其他条件相同的情况下，「状态测试」方法略优于可识别的返回值。它提供了较好的可读性，而且不正确的使用可能更容易被检测：如果你忘记调用「状态测试」方法，「状态依赖」方法将抛出异常，使错误显而易见；（4）如果你忘记检查一个可识别的返回值，那么这个 bug 可能很难发现。但是这对于返回 Optional 对象的方式来说不是问题。

总之，异常是为确有异常的情况设计的。不要将它们用于一般的控制流程，也不要编写强制其他人这样做的 API。

Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors（对可恢复情况使用 checked 异常，对编程错误使用运行时异常）

Java 提供了三种可抛出项：checked 异常、运行时异常和错误。程序员们对什么时候使用这些可抛出项比较困惑。虽然决策并不总是明确的，但是有一些通用规则可以提供强有力的指导。

决定是使用 checked 异常还是 unchecked 异常的基本规则是：**使用 checked 异常的情况是为了合理地期望调用者能够从中恢复。** 通过抛出一个 checked 的异常，你可以强制调用者在 catch 子句中处理异常，或者将其传播出去。因此，方法中声明的要抛出的每个 checked 异常，都清楚的向 API 用户表明：the associated condition is a possible outcome of invoking the method.

通过向用户提供 checked 异常，API 设计者提供了从条件中恢复的要求。用户为了无视强制要求，可以捕获异常并忽略，但这通常不是一个好主意（Item-77）
。

有两种 unchecked 的可抛出项：运行时异常和错误。它们在行为上是一样的：都是可抛出的，通常不需要也不应该被捕获。如果程序抛出 unchecked 异常或错误，通常情况下是不可能恢复的，如果继续执行，弊大于利。如果程序没有捕获到这样的可抛出项，它将导致当前线程停止，并发出适当的错误消息。

使用运行时异常来指示编程错误。 绝大多数运行时异常都表示操作违反了先决条件。违反先决条件是指使用 API 的客户端未能遵守 API 规范所建立的约定。例如，数组访问约定指定数组索引必须大于等于 0 并且小于等于 length-1 （length：数组长度）。ArrayIndexOutOfBoundsException 表示违反了此先决条件。

这个建议存在的问题是，并不总能清楚是在处理可恢复的条件还是编程错误。例如，考虑资源耗尽的情况，这可能是由编程错误（如分配一个不合理的大数组）或真正的资源短缺造成的。如果资源枯竭是由于暂时短缺或暂时需求增加造成的，这种情况很可能是可以恢复的。对于 API 设计人员来说，判断给定的资源耗尽实例是否允许恢复是一个问题。如果你认为某个条件可能允许恢复，请使用 checked 异常；如果没有，则使用运行时异常。如果不清楚是否可以恢复，最好使用 unchecked 异常，原因将在 Item-71 中讨论。

虽然 Java 语言规范没有要求，但有一个约定俗成的约定，即错误保留给 JVM 使用，以指示：资源不足、不可恢复故障或其他导致无法继续执行的条件。考虑到这种约定被大众认可，所以最好不要实现任何新的 Error 子类。因此，**你实现的所有 unchecked 可抛出项都应该继承 RuntimeException**（直接或间接）。不仅不应该定义 Error 子类，而且除了 AssertionError 之外，不应该抛出它们。

可以自定义一种可抛出的异常，它不是 Exception、RuntimeException 或 Error 的子类。JLS 不直接处理这些可抛出项，而是隐式地指定它们作为普通 checked 异常（普通 checked 异常是 Exception 的子类，但不是 RuntimeException 的子类）。那么，什么时候应该使用这样的「猛兽」呢？总之，永远不要。与普通 checked 异常相比，它们没有任何好处，只会让 API 的用户感到困惑。

API 设计人员常常忘记异常是成熟对象，可以为其定义任意方法。此类方法的主要用途是提供捕获异常的代码，并提供有关引发异常的附加信息。如果缺乏此类方法，程序员需要自行解析异常的字符串表示以获取更多信息。这是极坏的做法（Item-12）。这种类很少指定其字符串表示的细节，因此字符串表示可能因实现而异，也可能因版本而异。因此，解析异常的字符串表示形式的代码可能是不可移植且脆

弱的。

Because checked exceptions generally indicate recoverable conditions, it's especially important for them to provide methods that furnish information to help the caller recover from the exceptional condition. For example, suppose a checked exception is thrown when an attempt to make a purchase with a gift card fails due to insufficient funds. The exception should provide an accessor method to query the amount of the shortfall. This will enable the caller to relay the amount to the shopper. See Item 75 for more on this topic.

因为 checked 异常通常表示可恢复的条件，所以这类异常来说，设计能够提供信息的方法来帮助调用者从异常条件中恢复尤为重要。例如，假设当使用礼品卡购物由于资金不足而失败时，抛出一个 checked 异常。该异常应提供一个访问器方法来查询差额。这将使调用者能够将金额传递给购物者。有关此主题的更多信息，请参见 Item-75。

To summarize, throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions. Don't define any throwables that are neither checked exceptions nor runtime exceptions. Provide methods on your checked exceptions to aid in recovery.

总而言之，为可恢复条件抛出 checked 异常，为编程错误抛出 unchecked 异常。当有疑问时，抛出 unchecked 异常。不要定义任何既不是 checked 异常也不是运行时异常的自定义异常。应该为 checked 异常设计相关的方法，如提供异常信息，以帮助恢复。

Item 71: Avoid unnecessary use of checked exceptions（避免不必要地使用 checked 异常）

许多 Java 程序员不喜欢 checked 异常，但是如果使用得当，它们可以有利于 API 和程序。与返回代码和 unchecked 异常不同，它们强制程序员处理问题，提高了可靠性。相反，在 API 中过度使用 checked 异常会变得不那么令人愉快。如果一个方法抛出 checked 异常，调用它的代码必须在一个或多个 catch 块中处理它们；或者通过声明抛出，让它们向外传播。无论哪种方式，它都给 API 的用户带来了负担。Java 8 中，这一负担增加得更多，因为会抛出 checked 异常的方法不能直接在流（Item 45-48）中使用。

如果（1）正确使用 API 也不能防止异常情况，（2）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，那么这种负担是合理的。除非满足这两个条件，否则可以使用 unchecked 异常。作为程序能否成功的试金石，程序员应该问问自己将如何处理异常。这是最好的办法吗?

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Can't happen!
}
```

或者这样?

```
} catch (TheCheckedException e) {
    e.printStackTrace(); // Oh well, we lose.
    System.exit(1);
}
```

如果程序员不能做得更好，则需要一个 unchecked 异常。

如果 checked 异常是方法抛出的唯一 checked 异常，那么 checked 异常给程序员带来的额外负担就会大得多。如果还有其他方法，则该方法必须已经出现在 try 块中，并且此异常最多需要另一个 catch 块。如果一个方法抛出一个 checked 异常，那么这个异常就是该方法必须出现在 try 块中而不能直接在流中使用的唯一原因。在这种情况下，有必要问问自己是否有办法避免 checked 异常。

消除 checked 异常的最简单方法是返回所需结果类型的 Optional 对象（Item-55）。该方法只返回一个空的 Optional 对象，而不是抛出一个 checked 异常。这种技术的缺点是，该方法不能返回任何详细说明其无法执行所需计算的附加信息。相反，异常具有描述性类型，并且可以导出方法来提供附加信息（Item-70）。

你还可以通过将抛出异常的方法拆分为两个方法，从而将 checked 异常转换为 unchecked 异常，第一个方法返回一个布尔值，指示是否将抛出异常。这个 API 重构将调用序列转换为：

```
// Invocation with checked exception
try {
    obj.action(args);
}
catch (TheCheckedException e) {
    ... // Handle exceptional condition
}
```

转换为这种形式：

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
}
else {
    ... // Handle exceptional condition
}
```

这种重构并不总是适当的，但是只要在适当的地方，它就可以使 API 更易于使用。虽然后一种调用序列并不比前一种调用序列漂亮，但是经过重构的 API 更加灵活。如果程序员知道调用会成功，或者不介意由于调用失败而导致的线程终止，那么该重构还可以接受更简单的调用序列：

```
obj.action(args);
```

If you suspect that the trivial calling sequence will be the norm，那么 API 重构可能是合适的。重构之后的 API 在本质上等同于 Item-69 中的「状态测试」方法，并且，也有同样的告诫：如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，这种重构就是不恰当的，因为在 actionPermitted 和 action 这两个调用的间隔，对象的状态有可能会发生变化。如果单独的 actionPermitted 方法必须重复 action 方法的工作，出于性能的考虑，这种 API 重构就不值得去做。

总之，如果谨慎使用，checked 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。如果调用者不应从失败中恢复，则抛出 unchecked 异常。如果恢复是可能的，并且你希望强制调用者处理异常条件，那么首先考虑返回一个 Optional 对象。只有当在失败的情况下，提供的信息不充分时，你才应该抛出一个 checked 异常。

Item 72: Favor the use of standard exceptions（鼓励复用标准异常）

专家程序员与经验较少的程序员之间的一个区别是，专家力求实现高度的代码复用。代码复用是一件好事，异常也不例外。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。

复用标准异常有几个好处。其中最主要的是，它使你的 API 更容易学习和使用，因为它符合程序员已经熟悉的既定约定。其次，使用你的 API 的程序更容易阅读，因为它们不会因为不熟悉的异常而混乱。最后（也是最不重要的），更少的异常类意味着更小的内存占用和更少的加载类的时间。

最常见的复用异常类型是 IllegalArgumentException（Item-49）。这通常是调用者传入不合适的参数时抛出的异常。例如，如果调用者在表示某个操作要重复多少次的参数中传递了一个负数，则抛出这个异常。

另一个常被复用异常是 IllegalStateException。如果接收对象的状态导致调用非法，则通常会抛出此异常。例如，如果调用者试图在对象被正确初始化之前使用它，那么这将是抛出的异常。

可以说，每个错误的方法调用都归结为参数非法或状态非法，但是有一些异常通常用于某些特定的参数非法和状态非法。如果调用者在禁止空值的参数中传递 null，那么按照惯例，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，则应该抛出 IndexOutOfBoundsException，而不是 IllegalArgumentException。

另一个可复用异常是 ConcurrentModificationException。如果一个对象被设计为由单个线程使用（或与外部同步），并且检测到它正在被并发地修改，则应该抛出该异常。因为不可能可靠地检测并发修改，所以该异常充其量只是一个提示。

最后一个需要注意的标准异常是 UnsupportedOperationException。如果对象不支持尝试的操作，则抛出此异常。它很少使用，因为大多数对象都支持它们的所有方法。此异常用于一个类没有实现由其实现的接口定义的一个或多个可选操作。例如，对于只支持追加操作的 List 实现，试图从中删除元素时就会抛出这个异常。

不要直接复用 Exception、RuntimeException、Throwable 或 Error。 应当将这些类视为抽象类。你不能对这些异常进行可靠的测试，因为它们是方法可能抛出的异常的超类。

此表总结了最常见的可复用异常：

Exception	Occasion for Use
IllegalArgumentException	Non-null parameter value is inappropriate（非空参数值不合适）
IllegalStateException	Object state is inappropriate for method invocation（对象状态不适用于方法调用）
NullPointerException	Parameter value is null where prohibited（禁止参数为空时仍传入 null）
IndexOutOfBoundsException	Index parameter value is out of range（索引参数值超出范围）
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited（在禁止并发修改对象的地方检测到该动作）
UnsupportedOperationException	Object does not support method（对象不支持该方法调用）

虽然到目前为止，这些是最常见的复用异常，但是在环境允许的情况下也可以复用其他异常。例如，如果你正在实现诸如复数或有理数之类的算术对象，那么复用 ArithmeticException 和 NumberFormatException 是合适的。如果一个异常符合你的需要，那么继续使用它，但前提是你抛出它的条件与异常的文档描述一致：复用必须基于文档化的语义，而不仅仅是基于名称。另外，如果你想添加更多的细节，可以随意子类化标准异常(第75项)，但是请记住，异常是可序列化的（Chapter 12）。如果没有充分的理由，不要编写自己的异常类。

选择复用哪个异常可能比较棘手，因为上表中的「使用场合」似乎并不相互排斥。考虑一个对象，表示一副牌，假设有一个方法代表发牌操作，该方法将手牌多少作为参数。如果调用者传递的值大于牌堆中剩余的牌的数量，则可以将其解释为 IllegalArgumentException （handSize 参数值太大）或 IllegalStateException（牌堆中包含的牌太少）。在这种情况下，规则是：如果没有参数值，抛出 IllegalStateException，否则抛出 IllegalArgumentException。

Item 73: Throw exceptions appropriate to the abstraction（抛出能用抽象解释的异常）

当一个方法抛出一个与它所执行的任务没有明显关联的异常时，这是令人不安的。这种情况经常发生在由方法传播自低层抽象抛出的异常。它不仅令人不安，而且让实现细节污染了上层的 API。如果高层实现在以后的版本中发生变化，那么它抛出的异常也会发生变化，可能会破坏现有的客户端程序。

为了避免这个问题，**高层应该捕获低层异常，并确保抛出的异常可以用高层抽象解释。** 这个习惯用法称为异常转换：

```
// Exception Translation
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

下面是来自 AbstractSequentialList 类的异常转换示例，该类是 List 接口的一个框架实现（Item-20）。在本例中，异常转换是由 `List<E>` 接口中的 get 方法规范强制执行的：

```
/**
* 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);
        }
    }
```

如果低层异常可能有助于调试高层异常的问题，则需要一种称为链式异常的特殊异常转换形式。低层异常（作为原因）传递给高层异常，高层异常提供一个访问器方法（Throwable 的 getCause 方法）来检索低层异常：

```
    // Exception Chaining
    try {
        ... // Use lower-level abstraction to do our bidding
    }
    catch (LowerLevelException cause) {
        throw new HigherLevelException(cause);
    }
```

高层异常的构造函数将原因传递给能够接收链式异常的超类构造函数，因此它最终被传递给 Throwable 的一个接收链式异常的构造函数，比如 `Throwable(Throwable)`：

```
    // Exception with chaining-aware constructor
    class HigherLevelException extends Exception {
        HigherLevelException(Throwable cause) {
            super(cause);
        }
    }
```

大多数标准异常都有接收链式异常的构造函数。对于不支持链式异常的异常，可以使用 Throwable 的 initCause 方法设置原因。异常链接不仅允许你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层异常的堆栈跟踪中。

虽然异常转换优于底层异常的盲目传播，但它不应该被过度使用。在可能的情况下，处理低层异常的最佳方法是确保低层方法避免异常。有时，你可以在将高层方法的参数传递到低层之前检查它们的有效性。

如果不可能从低层防止异常，那么下一个最好的方法就是让高层静默处理这些异常，使较高层方法的调用者免受低层问题的影响。在这种情况下，可以使用一些适当的日志工具（如 `java.util.logging`）来记录异常。这允许程序员研究问题，同时将客户端代码和用户与之隔离。

总之，如果无法防止或处理来自低层的异常，则使用异常转换，但要保证低层方法的所有异常都适用于较高层。链式异常提供了兼顾两方面的最佳服务：允许抛出适当的高层异常，同时捕获并分析失败的潜在原因（Item-75）。

Item 74: Document all exceptions thrown by each method（为每个方法记录会抛出的所有异常）

描述方法抛出的异常，是该方法文档的重要部分。因此，花时间仔细记录每个方法抛出的所有异常是非常重要的（Item-56）。

始终单独声明 checked 异常，并使用 Javadoc 的 `@throw` 标记精确记录每次抛出异常的条件。如果一个方法抛出多个异常，不要使用快捷方式声明这些异常的超类。作为一个极端的例子，即不要在公共方法声明 `throws Exception`，或者更糟，甚至 `throws Throwable`。除了不能向方法的用户提供会抛出哪些异常的任何消息之外，这样的声明还极大地阻碍了方法的使用，因为它掩盖了在相同上下文中可能抛出的任何其他异常。这个建议的一个特例是 main 方法，它可以安全地声明 `throw Exception`，因为它只被 VM 调用。

虽然 Java 不要求程序员为方法声明会抛出的 unchecked 异常，但明智的做法是，应该像 checked 异常一样仔细地记录它们。unchecked 异常通常表示编程错误（Item-70），让程序员熟悉他们可能犯的所有错误可以帮助他们避免犯这些错误。将方法可以抛出的 unchecked 异常形成良好文档，可以有效描述方法成功执行的先决条件。每个公共方法的文档都必须描述它的先决条件（Item-56），记录它的 unchecked 异常是满足这个需求的最佳方法。

特别重要的是，接口中的方法要记录它们可能抛出的 unchecked 异常。此文档构成接口通用约定的一部分，并指明接口的多个实现之间应该遵守的公共行为。

使用 Javadoc 的 `@throw` 标记记录方法会抛出的每个异常，但是不要对 unchecked 异常使用 throws 关键字。 让使用你的 API 的程序员知道哪些异常是 checked 异常，哪些是 unchecked 异常是很重要的，因为程序员在这两种情况下的职责是不同的。Javadoc 的 `@throw` 标记生成的文档在方法声明中没有对应的 throws 子句，这向程序员提供了一个强烈的视觉提示，这是 unchecked 异常。

应该注意的是，记录每个方法会抛出的所有 unchecked 异常是理想的，但在实际中并不总是可以做到。当类进行修订时，如果将导出的方法修改，使其抛出额外的 unchecked 异常，这并不违反源代码或二进制兼容性。假设一个类 A 从另一个独立的类 B 中调用一个方法。A 类的作者可能会仔细记录每个方法抛出的 unchecked 异常，如果 B 类被修改了，使其抛出额外的 unchecked 异常，很可能 A 类（未经修改）将传播新的 unchecked 异常，尽管它没有在文档中声明。

如果一个类中的许多方法都因为相同的原因抛出异常，你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异常。一个常见的例子是 NullPointerException。类的文档注释可以这样描述：「如果在任何参数中传递了 null 对象引用，该类中的所有方法都会抛出 NullPointerException」，或者类似意思的话。

总之，记录你所编写的每个方法可能引发的每个异常。对于 unchecked 异常、checked 异常、抽象方法、实例方法都是如此。应该在文档注释中采用 `@throw` 标记的形式。在方法的 throws 子句中分别声明每个 checked 异常，但不要声明 unchecked 异常。如果你不记录方法可能抛出的异常，其他人将很难或不可能有效地使用你的类和接口。

Item 75: Include failure capture information in detail messages（异常详细消息中应包含捕获失败的信息）

当程序由于未捕获异常而失败时，系统可以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，这是调用其 toString 方法的结果。这通常包括异常的类名及其详细信息。通常，这是程序员或管理员在调查软件故障时所掌握的唯一信息。如果失败不容易重现，想获得更多的信息会非常困难。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非常重要的。换句话说，由失败导致的异常的详细信息应该被捕获，以便后续分析。

要捕获失败，异常的详细消息应该包含导致异常的所有参数和字段的值。 例如，IndexOutOfBoundsException 的详细消息应该包含下界、上界和未能位于下界之间的索引值。这些信息说明了很多关于失败的信息。这三个值中的任何一个或所有值都可能是错误的。索引可以小于或等于上界（「越界错误」），也可以是一个无效值，太小或太大。下界可能大于上界（严重的内部故障）。每一种情况都指向一个不同的问题，如果你知道你在寻找什么样的错误，这对诊断有很大的帮助。

提及一个与安全敏感信息有关的警告。因为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，所以 **不应包含密码、加密密钥等详细信息。**

虽然在异常的详细信息中包含所有相关数据非常重要，但通常不需要包含大量的描述。堆栈跟踪将与文档一起分析，如果需要，还将与源代码一起分析。它通常包含抛出异常的确切文件和行号，以及堆栈上所有方法调用的文件和行号。冗长的描述对一个失败问题而言是多余的；可以通过阅读文档和源代码来收集信息。

异常的详细信息不应该与用户层的错误消息混淆，因为用户层错误消息最终必须被用户理解。与用户层错误消息不同，详细消息主要是为程序员或管理员在分析故障时提供的。因此，信息内容远比可读性重要。用户层错误消息通常是本地化的，而异常详细信息消息很少本地化。确保异常在其详细信息中包含足够的故障捕获信息的一种方法是，在其构造函数中配置，而不是以传入字符串方式引入这些信息。之后可以自动生成详细信息来包含细节。例如，IndexOutOfBoundsException 构造函数不包含 String 参数，而是像这样：

```
    /**
    * 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 最终获得了一个接受 int 值索引参数的构造函数，但遗憾的是它忽略了下界和上界参数。更一般地说，Java 库不会大量使用这个习惯用法，但是强烈推荐使用它。它使程序员很容易通过抛出异常来捕获失败。事实上，它使程序员不想捕获失败都难！实际上，这个习惯用法将集中在异常类中生成高质量的详细信息，而不是要求该类的每个用户都生成冗余的详细信息。

译注：IndexOutOfBoundsException 有关 int 参数的构造函数源码

```java
    /**
     * Constructs a new {@code IndexOutOfBoundsException} class with an
     * argument indicating the illegal index.
     *
     * <p>The index is included in this exception's detail message.  The
     * exact presentation format of the detail message is unspecified.
     *
     * @param index the illegal index.
     * @since 9
     */
    public IndexOutOfBoundsException(int index) {
        super("Index out of range: " + index);
    }
```

正如 Item-70 中建议的，异常为其故障捕获信息提供访问器方法是适合的（上面示例中的下界、上界和索引）。在 checked 异常上提供此类访问器方法比 unchecked 异常上提供此类访问器方法更为重要，因为故障捕获信息可能有助于程序从故障中恢复。程序员可能希望通过编程访问 unchecked 异常的详细信息，但这是很少见的（尽管是可以想象的）。然而，即使对于 unchecked 异常，根据一般原则，提供这些访问器也是可以的（Item-12，第 57 页）。

Item 76: Strive for failure atomicity（尽力保证故障原子性）

在对象抛出异常之后，通常希望对象仍然处于定义良好的可用状态，即使在执行操作时发生了故障。对于 checked 异常尤其如此，调用者希望从异常中恢复。**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。** 具有此属性的方法称为具备故障原子性。

有几种方式可以达到这种效果。最简单的方法是设计不可变对象（Item-17）。如果对象是不可变的，则故障原子性是必然的。如果一个操作失败，它可能会阻止创建一个新对象，但是它不会让一个现有对象处于不一致的状态，因为每个对象的状态在创建时是一致的，并且在创建后不能修改。

对于操作可变对象的方法，实现故障原子性的最常见方法是在执行操作之前检查参数的有效性（Item-49）。这使得大多数异常在对象修改开始之前被抛出。例如，考虑 `Stack.pop` 方法（Item-7）：

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

如果取消了初始大小检查，当该方法试图从空堆栈中弹出元素时，仍然会抛出异常。但是，这会使 size 字段处于不一致的（负值）状态，导致以后该对象的任何方法调用都会失败。此外，pop 方法抛出的 ArrayIndexOutOfBoundsException 也不适于高层抽象解释（Item-73）。

实现故障原子性的另一种方式是对计算进行排序，以便可能发生故障的部分都先于修改对象的部分发生。当执行某部分计算才能检查参数时，这种方法是前一种方法的自然扩展。例如，考虑 TreeMap 的情况，它的元素按照一定的顺序排序。为了向 TreeMap 中添加元素，元素的类型必须能够使用 TreeMap 的顺序进行比较。在以任何方式修改「树」之前，由于在「树」中搜索元素，试图添加类型不正确的元素自然会失败，并导致 ClassCastException 异常。

实现故障原子性的第三种方法是以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。当数据存储在临时数据结构中后，计算过程会更加迅速，这种办法就是很自然的。例如，一些排序函数在排序之前将其输入 list 复制到数组中，以降低访问排序内循环中的元素的成本。这样做是为了提高性能，但是作为一个额外的好处，它确保如果排序失败，输入 list 将保持不变。

实现故障原子性的最后一种不太常见的方法是编写恢复代码，拦截在操作过程中发生的故障，并使对象回滚到操作开始之前的状态。这种方法主要用于持久的（基于磁盘的）数据结构。

虽然故障原子性通常是可取的，但它并不总是可以实现的。例如，如果两个线程试图在没有适当同步的情况下并发地修改同一个对象，那么该对象可能会处于不一致的状态。因此，如果假定在捕捉到 ConcurrentModificationException 之后对象仍然可用，那就错了。该错误是不可恢复的，所以在抛出 AssertionError 时，你甚至不需要尝试保存故障原子性。

即使在可以实现故障原子性的情况下，也并不总是可取的。对于某些操作，它将显著增加成本或复杂性。也就是说，一旦意识到这个问题，就可以轻松地实现故障原子性。

总之，作为规则，也作为方法规范的一部分，生成的任何异常都应该使对象保持在方法调用之前的状态。如果违反了这条规则，API 文档应该清楚地指出对象将处于什么状态。不幸的是，许多现有的 API 文档都没有做到。

Item 77: Don't ignore exceptions（不要忽略异常）

虽然这一建议似乎显而易见，但它经常被违反，因此值得强调。当 API 的设计人员声明一个抛出异常的方法时，他们试图告诉你一些事情。不要忽略它！如果在方法调用的周围加上一条 try 语句，其 catch 块为空，可以很容易忽略异常：

```
// Empty catch block ignores exception - Highly suspect!
try {
    ...
}
catch (SomeException e) {
}
```

空 catch 块违背了异常的目的， 它的存在是为了强制你处理异常情况。忽略异常类似于忽略火灾警报一样，关掉它之后，其他人就没有机会看到是否真的发生了火灾。你可能侥幸逃脱，但结果可能是灾难性的。每当你看到一个空的 catch 块，你的脑海中应该响起警报。

在某些情况下，忽略异常是合适的。例如，在关闭 FileInputStream 时，忽略异常可能是合适的。你没有更改文件的状态，因此不需要执行任何恢复操作，并且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，这样如果这些异常经常发生，你应该研究起因。**如果你选择忽略异常，catch 块应该包含一条注释，解释为什么这样做是合适的，并且应该将变量命名为 ignore：**

```
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
}
```

本条目中的建议同样适用于 checked 异常和 unchecked异常。不管异常是表示可预测的异常条件还是编程错误，用空 catch 块忽略它将导致程序在错误面前保持静默。然后，程序可能会在未来的任意时间点，在与问题源没有明显关系的代码中失败。正确处理异常可以完全避免失败。仅仅让异常向外传播，可能会导致程序走向失败，保留信息有利于调试。

Chapter 11. Concurrency（并发）

Chapter 11 Introduction（章节介绍）

THREADS allow multiple activities to proceed concurrently. Concurrent programming is harder than single-threaded programming, because more things can go wrong, and failures can be hard to reproduce. You can't avoid concurrency. It is inherent in the platform and a requirement if you are to obtain good performance from multicore processors, which are now ubiquitous. This chapter contains advice to help you write clear, correct, well-documented concurrent programs.

线程允许多个活动并发进行。并发编程比单线程编程更困难，容易出错的地方更多，而且失败很难重现。你无法避开并发。它是平台中固有的，并且多核处理器现在也是无处不在，而你会有从多核处理器获得良好的性能的需求。本章包含一些建议，帮助你编写清晰、正确、文档良好的并发程序。

Item 78: Synchronize access to shared mutable data（对共享可变数据的同步访问）

synchronized 关键字确保一次只有一个线程可以执行一个方法或块。许多程序员认为同步只是一种互斥的方法，是为防止一个线程在另一个线程修改对象时使对象处于不一致的状态。这样看来，对象以一致的状态创建（Item-17），并由访问它的方法锁定。这些方法可以察觉当前状态，并引起状态转换，将对象从一致的状态转换为另一个一致的状态。正确使用同步可以保证没有方法会让对象处于不一致状态。

这种观点是正确的，但它只是冰山一角。没有同步，一个线程所做的的更改可能对其他线程不可见。同步不仅阻止线程察觉到处于不一致状态的对象，而且确保每个进入同步方法或块的线程都能察觉由同一把锁保护的所有已修改的效果。

语言规范保证读取或写入变量是原子性的，除非变量的类型是 long 或 double [JLS, 17.4, 17.7]。换句话说，读取 long 或 double 之外的变量将保证返回某个线程存储在该变量中的值，即使多个线程同时修改该变量，并且没有同步时也是如此。

你可能听说过，为了提高性能，在读取或写入具有原子性的数据时应该避免同步。这种建议大错特错。虽然语言规范保证线程在读取字段时不会觉察任意值，但它不保证由一个线程编写的值对另一个线程可见。**线程之间能可靠通信以及实施互斥，同步是所必需的。** 这是由于语言规范中，称为内存模型的部分指定了一个线程所做的更改何时以及如何对其他线程可见 [JLS, 17.4; Goetz06, 16]。

即使数据是原子可读和可写的，无法同步访问共享可变数据的后果也可能是可怕的。考虑从一个线程中使另一个线程停止的任务。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，它的使用可能导致数据损坏。**不要使用 `Thread.stop`。** 一个建议的方法是让第一个线程轮询一个 boolean 字段，该字段最初为 false，但第二个线程可以将其设置为 true，以指示第

一个线程要停止它自己。由于读写布尔字段是原子性的，一些程序员在访问该字段时不需要同步：

```
// 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;
    }
}
```

你可能认为这个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。然而，在我的机器上，程序永远不会终止：后台线程永远循环！

问题在于在缺乏同步的情况下，无法保证后台线程何时（如果有的话）看到主线程所做的 stopRequested 值的更改。在缺乏同步的情况下，虚拟机可以很好地转换这段代码：

```
while (!stopRequested)
    i++;
into this code:
if (!stopRequested)
    while (true)
        i++;
```

这种优化称为提升，这正是 OpenJDK 服务器 VM 所做的。结果是活性失败：程序无法取得进展。解决此问题的一种方法是同步对 stopRequested 字段的访问。程序在大约一秒内结束，正如预期：

```
// 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();
    }
}
```

注意，写方法（requestStop）和读方法（stopRequested）都是同步的。仅同步写方法是不够的！**除非读和写操作都同步，否则不能保证同步工作。** 有时，只同步写（或读）的程序可能在某些机器上显示有效，但在这种情况下，不能这么做。

即使没有同步，StopThread 中同步方法的操作也是原子性的。换句话说，这些方法的同步仅用于其通信效果，而不是互斥。虽然在循环的每个迭代上同步的成本很小，但是有一种正确的替代方法，它不那么冗长，而且性能可能更好。如果 stopRequested 声明为 volatile，则可以省略 StopThread 的第二个版本中的锁。虽然 volatile 修饰符不执行互斥，但它保证任何读取字段的线程都会看到最近写入的值：

```
// 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;
        }
    }
```

在使用 volatile 时一定要小心。考虑下面的方法，它应该生成序列号：

```
    // Broken - requires synchronization!
    private static volatile int nextSerialNumber = 0;

    public static int generateSerialNumber() {
        return nextSerialNumber++;
    }
```

该方法的目的是确保每次调用返回一个唯一的值（只要不超过 2^{32} 次调用）。方法的状态由一个原子可访问的字段 nextSerialNumber 组成，该字段的所有可能值都是合法的。因此，不需要同步来保护它的不变性。不过，如果没有同步，该方法将无法正常工作。

问题在于增量运算符 `(++)` 不是原子性的。它对 nextSerialNumber 字段执行两个操作：首先读取值，然后返回一个新值，旧值再加 1。如果第二个线程在读取旧值和写入新值之间读取字段，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。这是一个安全故障：使程序计算错误的原因。

修复 generateSerialNumber 的一种方法是将 synchronized 修饰符添加到它的声明中。这确保了多个调用不会交叉，并且该方法的每次调用都将看到之前所有调用的效果。一旦你这样做了，你就可以并且应该从 nextSerialNumber 中删除 volatile 修饰符。为了使方法更可靠，应使用 long 而不是 int，或者在 nextSerialNumber 即将超限时抛出异常。

更好的方法是，遵循 Item-59 中的建议并使用 AtomicLong 类，它是 `java.util.concurrent.atomic` 的一部分。这个包为单变量的无锁、线程安全编程提供了基本类型。虽然 volatile 只提供同步的通信效果，但是这个包提供原子性。这正是我们想要的 generateSerialNumber，它很可能优于同步版本：

```
    // Lock-free synchronization with java.util.concurrent.atomic
    private static final AtomicLong nextSerialNum = new AtomicLong();

    public static long generateSerialNumber() {
        return nextSerialNum.getAndIncrement();
    }
```

为避免出现本条目中讨论的问题，最佳方法是不共享可变数据。要么共享不可变数据（Item-17），要么完全不共享。换句话说，**应当将可变数据限制在一个线程中。** 如果采用此策略，重要的是对其进行文档化，以便随着程序的发展维护该策略。深入了解你正在使用的框架和库也很重要，因为它们可能会引入你不知道的线程。

一个线程可以暂时修改一个数据对象，然后与其他线程共享，并且只同步共享对象引用的操作。然后，其他线程可以在没有进一步同步的情况下读取对象，只要不再次修改该对象。这些对象被认为是有效不可变的 [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]。安全地发布对象引用的方法有很多：可以将它存储在静态字段中，作为类初始化的一部分；你可以将其存储在易失性字段、final 字段或使用普通锁定访问的字段中；或者你可以将其放入并发集合中（Item-81）。

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。** 在缺乏同步的情况下，不能保证一个线程的更改对另一个线程可见。同步共享可变数据失败的代价是活性失败和安全失败。这些故障是最难调试的故障之一。它们可能是间歇性的，并与时间相关，而且程序行为可能在不同 VM 之间发生根本的变化。如果只需要线程间通信，而不需要互斥，那么 volatile 修饰符是一种可接受的同步形式，但是要想正确使用它可能会比较棘手。

Item 79: Avoid excessive synchronization（避免过度同步）

Item-78 警告我们同步不到位的危险。本条目涉及相反的问题。根据不同的情况，过度的同步可能导致性能下降、死锁甚至不确定行为。

为避免活性失败和安全故障，永远不要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，不要调用一个设计为被覆盖的方法，或者一个由客户端以函数对象的形式提供的方法（Item-24）。从具有同步区域的类的角度来看，这种方法是不一样的。类不知道该方法做什么，也无法控制它。Depending on what an alien method does，从同步区域调用它可能会导致异常、死锁或数据损坏。

要使这个问题具体化，请考虑下面的类，它实现了一个可视 Set 包装器。当元素被添加到集合中时，它允许客户端订阅通知。这是观察者模式 [Gamma95]。为了简单起见，当元素从集合中删除时，该类不提供通知，即使要提供通知也很简单。这个类是在 Item-18（第 90 页）的可复用 ForwardingSet 上实现的：

```
// 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，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方法。

```
    @FunctionalInterface
    public interface SetObserver<E> {
        // Invoked when an element is added to the observable set
        void added(ObservableSet<E> set, E element);
    }
```

这个接口在结构上与 `BiConsumer<ObservableSet<E>,E>` 相同。我们选择定义一个自定义函数式接口，因为接口和方法名称使代码更具可读性，而且接口可以演化为包含多个回调。也就是说，使用 BiConsumer 也是合理的（Item-44）。

粗略地检查一下，ObservableSet 似乎工作得很好。例如，下面的程序打印从 0 到 99 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);
    }
```

现在让我们尝试一些更奇特的东西。假设我们将 addObserver 调用替换为一个传递观察者的调用，该观察者打印添加到集合中的整数值，如果该值为 23，则该调用将删除自身：

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});
```

注意，这个调用使用一个匿名类实例来代替前面调用中使用的 lambda 表达式。这是因为函数对象需要将自己传递给 `s.removeObserver`，而 lambda 表达式不能访问自身（Item-42）。

你可能希望程序打印数字 0 到 23，然后观察者将取消订阅，程序将无声地终止。实际上，它打印这些数字，然后抛出 ConcurrentModificationException。问题在于 notifyElementAdded 在调用观察者的 added 方法时，正在遍历 observers 列表。added 方法调用可观察集的 removeObserver 方法，该方法反过来调用方法 `observers.remove`。现在我们有麻烦了。我们试图在遍历列表的过程中从列表中删除一个元素，这是非法的。notifyElementAdded 方法中的迭代位于一个同步块中，以防止并发修改，但是无法防止迭代线程本身回调到可观察的集合中，也无法防止修改它的 observers 列表。

现在让我们尝试一些奇怪的事情：让我们编写一个观察者来尝试取消订阅，但是它没有直接调用 removeObserver，而是使用另一个线程的服务来执行这个操作。该观察者使用 executor 服务（Item-80）：

```
// 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，注意这个程序在一个 catch 子句中捕获了两种不同的异常类型。这个功能在 Java 7 中添加了，非正式名称为 multi-catch。它可以极大地提高清晰度，并减少在响应多种异常类型时表现相同的程序的大小。

当我们运行这个程序时，我们不会得到异常；而是遭遇了死锁。后台线程调用 `s.removeObserver`，它试图锁定观察者，但无法获取锁，因为主线程已经拥有锁。一直以来，主线程都在等待后台线程完成删除观察者的操作，这就解释了死锁的原因。

这个例子是人为设计的，因为观察者没有理由使用后台线程来取消订阅本身，但是问题是真实的。在实际系统中，从同步区域内调用外来方法会导致许多死锁，比如 GUI 工具包。

在前面的两个例子中（异常和死锁），我们都很幸运。调用外来方法（added）时，由同步区域（观察者）保护的资源处于一致状态。假设你要从同步区域调用一个外来方法，而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。与第一个导致异常的示例一样，调用线程已经持有锁，所以当它试图重新获得锁时，线程将成功，即使另一个概念上不相关的操作正在对锁保护的数据进行中。这种失败的后果可能是灾难性的。从本质上说，这把锁没能发挥它的作用。可重入锁简化了多线程面向对象程序的构造，但它们可以将活动故障转化为安全故障。

幸运的是，通过将外来方法调用移出同步块来解决这类问题通常并不难。对于 notifyElementAdded 方法，这涉及到获取观察者列表的「快照」，然后可以在没有锁的情况下安全地遍历该列表。有了这个改变，前面的两个例子都可以再也不会出现异常或者死锁了：

```
// 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);
}
```

实际上，有一种更好的方法可以将外来方法调用移出同步块。库提供了一个名为 CopyOnWriteArrayList 的并发集合（Item-81），该集合是为此目的量身定制的。此列表实现是 ArrayList 的变体，其中所有修改操作都是通过复制整个底层数组来实现的。因为从不修改内部数组，所以迭代不需要锁定，而且速度非常快。如果大量使用，CopyOnWriteArrayList 的性能会很差，但是对于很少修改和经常遍历的观察者列表来说，它是完美的。

如果将 list 修改为使用 CopyOnWriteArrayList，则不需要更改 ObservableSet 的 add 和 addAll 方法。下面是类的其余部分。请注意，没有任何显式同步：

```
// 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 open call [Goetz06, 10.1.4]。除了防止失败之外，开放调用还可以极大地提高并发性。一个陌生的方法可以运行任意长的时间。如果从同步区域调用了外来方法，其他线程对受保护资源的访问就会遭到不必要的拒绝。

作为规则，你应该在同步区域内做尽可能少的工作。 获取锁，检查共享数据，根据需要进行转换，然后删除锁。如果你必须执行一些耗时的活动，请设法将其移出同步区域，而不违反 Item-78 中的指导原则。

本条目的第一部分是关于正确性的。现在让我们简要地看一下性能。虽然自 Java 早期以来，同步的成本已经大幅下降，但比以往任何时候都更重要的是：不要过度同步。在多核世界中，过度同步的真正代价不是获得锁所花费的 CPU 时间；这是一种争论：而是失去了并行化的机会，以及由于需要确保每个核心都有一个一致的内存视图而造成的延迟。过度同步的另一个隐藏成本是，它可能限制 VM 优化代码执行的能力。

如果你正在编写一个可变的类，你有两个选择：你可以省略所有同步并允许客户端在需要并发使用时在外部进行同步，或者你可以在内部进行同步，从而使类是线程安全的（Item-82）。只有当你能够通过内部同步实现比通过让客户端在外部锁定整个对象获得高得多的并发性时，才应该选择后者。`java.util` 中的集合（废弃的 Vector 和 Hashtable 除外）采用前一种方法，而 `java.util.concurrent` 中的方法则采用后者（Item-81）。

在 Java 的早期，许多类违反了这些准则。例如，StringBuffer 实例几乎总是由一个线程使用，但是它们执行内部同步。正是由于这个原因，StringBuffer 被 StringBuilder 取代，而 StringBuilder 只是一个未同步的 StringBuffer。类似地，同样，`java.util.Random` 中的线程安全伪随机数生成器被 `java.util.concurrent.ThreadLocalRandom` 中的非同步实现所取代，这也是原因之一。如果有疑问，不要同步你的类，但要记录它不是线程安全的。

如果你在内部同步你的类，你可以使用各种技术来实现高并发性，例如分拆锁、分离锁和非阻塞并发控制。这些技术超出了本书的范围，但是在其他地方也有讨论 [Goetz06, Herlihy08]。

如果一个方法修改了一个静态字段，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（除非该类能够容忍不确定性行为）。多线程客户端不可能对这样的方法执行外部同步，因为不相关的客户端可以在不同步的情况下调用该方法。字段本质上是一个全局变量，即使它是私有的，因为它可以被不相关的客户端读取和修改。Item-78 中的 generateSerialNumber 方法使用的 nextSerialNumber 字段演示了这种情况。

总之，为了避免死锁和数据损坏，永远不要从同步区域内调用外来方法。更一般地说，将你在同步区域内所做的工作量保持在最小。在设计可变类时，请考虑它是否应该执行自己的同步。在多核时代，比以往任何时候都更重要的是不要过度同步。只有在有充分理由时，才在内部同步类，并清楚地记录你的决定（Item-82）。

Item 80: Prefer executors, tasks, and streams to threads（Executor、task、流优于直接使用线程）

本书的第一版包含一个简单工作队列的代码 [Bloch01, Item 49]。这个类允许客户端通过后台线程为异步处理排队。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上的任何工作后优雅地终止自己。这个实现只不过是一个玩具，但即便如此，它也需要一整页的代码，如果你做得不对，就很容易出现安全和活性失败。幸运的是，没有理由再编写这种代码了。

当这本书的第二版出版时，`java.util.concurrent` 已经添加到 Java 中。这个包有一个 Executor 框架，它是一个灵活的基于接口的任务执行工具。创建一个工作队列，它在任何方面都比在这本书的第一版更好，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();
```

你可以使用 executor 服务做更多的事情。例如，你可以等待一个特定任务完成（使用 get 方法，参见 Item-79，319 页），你可以等待任务集合中任何或全部任务完成（使用 invokeAny 或 invokeAll 方法），你可以等待 executor 服务终止（使用 awaitTermination 方法），你可以一个接一个检索任务，获取他们完成的结果（使用一个 ExecutorCompletionService），还可以安排任务在特定时间运行或定期运行（使用 ScheduledThreadPoolExecutor），等等。

如果希望多个线程处理来自队列的请求，只需调用一个不同的静态工厂，该工厂创建一种称为线程池的不同类型的 executor 服务。你可以使用固定或可变数量的线程创建线程池。`java.util.concurrent.Executors` 类包含静态工厂，它们提供你需要的大多数 executor。但是，如果你想要一些不同寻常的东西，你可以直接使用 ThreadPoolExecutor 类。这个类允许你配置线程池操作的几乎每个方面。

为特定的应用程序选择 executor 服务可能比较棘手。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，而且通常「做正确的事情」。但是对于负载沉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，则创建一个新的线程。如果服务器负载过重，所有 CPU 都被充分利用，并且有更多的任务到达，就会创建更多的线程，这只会使情况变得更糟。因此，在负载沉重的生产服务器中，最好使用 `Executors.newFixedThreadPool`，它为你提供一个线程数量固定的池，或者直接使用 ThreadPoolExecutor 类来实现最大限度的控制。

你不仅应该避免编写自己的工作队列，而且通常还应该避免直接使用线程。当你直接使用线程时，线程既是工作单元，又是执行它的机制。在 executor 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，即任务。有两种任务：Runnable 和它的近亲 Callable（与 Runnable 类似，只是它返回一个值并可以抛出任意异常）。执行任务的一般机制是 executor 服务。如果你从任务的角度考虑问题，并让 executor 服务为你执行这些任务，那么你就可以灵活地选择合适的执行策略来满足你的需求，并在你的需求发生变化时更改策略。本质上，Executor 框架执行的功能与 Collections 框架聚合的功能相同。

在 Java 7 中，Executor 框架被扩展为支持 fork-join 任务，这些任务由一种特殊的 Executor 服务（称为 fork-join 池）运行。由 ForkJoinTask 实例表示的 fork-join 任务可以划分为更小的子任务，由 ForkJoinPool 组成的线程不仅处理这些任务，而且还从其他线程「窃取」任务，以确保所有线程都处于繁忙状态，从而提高 CPU 利用率、更高的吞吐量和更低的延迟。编写和调优 fork-join 任务非常棘手。并行流（Item-48） 是在 fork 连接池之上编写的，假设它们适合当前的任务，那么你可以轻松地利用它们的性能优势。

对 Executor 框架的完整处理超出了本书的范围，但是感兴趣的读者可以在实践中可以参阅《Java Concurrency in Practice》 [Goetz06]。

Item 81: Prefer concurrency utilities to wait and notify（并发实用工具优于 wait 和 notify）

这本书的第一版专门介绍了 wait 和 notify 的正确用法 [Bloch01, item 50]。这些建议仍然有效，并在本条目末尾作了总结，但这一建议已远不如从前重要。这是因为使用 wait 和 notify 的理由要少得多。自 Java 5 以来，该平台提供了更高级别的并发实用工具，可以执行以前必须在 wait 和 notify 上手工编写代码的操作。**考虑到正确使用 wait 和 notify 的困难，你应该使用更高级别的并发实用工具。**

`java.util.concurrent` 中级别较高的实用工具可分为三类：Executor 框架，Item-80 简要介绍了该框架；并发集合；同步器。本条目简要介绍并发集合和同步器。

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它们自己的同步（Item-79）。因此，**不可能从并发集合中排除并发活动；锁定它只会使程序变慢。**

因为不能排除并发集合上的并发活动，所以也不能原子地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，这些操作将多个基本操作组合成单个原子操作。这些操作在并发集合上非常有用，因此使用默认方法（Item-21）将它们添加到 Java 8 中相应的集合接口。

例如，Map 的 `putIfAbsent(key, value)` 方法为一个没有映射的键插入一个映射，并返回与键关联的前一个值，如果没有，则返回 null。这使得实现线程安全的规范化 Map 变得很容易。这个方法模拟了 `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 get 等检索操作进行了优化。因此，只有在 get 表明有必要时，才值得首先调用 get 再调用 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;
}
```

除了提供优秀的并发性，ConcurrentHashMap 还非常快。在我的机器上，上面的 intern 方法比 `String.intern` 快六倍多（但是请记住， `String.intern` 必须使用一些策略来防止在长时间运行的应用程序中内存泄漏）。并发集合使同步集合在很大程度上过时。例如，**使用 ConcurrentHashMap 而不是 `Collections.synchronizedMap`。** 只要用并发 Map 替换同步 Map 就可以显著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了扩展，这些操作将等待（或阻塞）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首个元素，如果队列为空，则等待。这允许将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添加到该工作队列中，一个或多个消费者线程将工作项从该工作队列中取出并在这些工作项可用时处理它们。正如你所期望的，大多数 ExecutorService 实现，包括 ThreadPoolExecutor，都使用 BlockingQueue（Item-80）。

同步器是允许线程彼此等待的对象，允许它们协调各自的活动。最常用的同步器是 CountDownLatch 和 Semaphore。较不常用的是 CyclicBarrier 和 Exchanger。最强大的同步器是 Phaser。

Countdown latches are single-use barriers，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 的惟一构造函数接受一个 int，这个 int 是在允许所有等待的线程继续之前，必须在锁存器上调用倒计时方法的次数。

在这个简单的基本类型上构建有用的东西非常容易。例如，假设你想要构建一个简单的框架来为一个操作的并发执行计时。这个框架由一个方法组成，该方法使用一个 executor 来执行操作，一个并发级别表示要并发执行的操作的数量，一个 runnable 表示操作。所有工作线程都准备在 timer 线程启动时钟之前运行操作。当最后一个工作线程准备好运行该操作时，计时器线程「发令枪」，允许工作线程执行该操作。一旦最后一个工作线程完成该操作，计时器线程就停止时钟。在 wait 和 notify 的基础上直接实现这种逻辑至少会有点麻烦，但是在 CountDownLatch 的基础上实现起来却非常简单：

```
// 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;
}
```

```
// 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 习惯用法，即循环来调用 wait 方法；永远不要在循环之外调用它。 循环用于在等待之前和之后测试条件。

在等待之前测试条件，如果条件已经存在，则跳过等待，以确保活性。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，则不能保证线程将从等待中唤醒。

为了确保安全，需要在等待之后再测试条件，如果条件不成立，则再次等待。如果线程在条件不成立的情况下继续执行该操作，它可能会破坏由锁保护的不变性。当条件不成立时，有一些理由唤醒线程：

- 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更改保护状态。
- 当条件不成立时，另一个线程可能意外地或恶意地调用 notify。类通过等待公共可访问的对象来暴露自己。公共可访问对象的同步方法中的任何 wait 都容易受到这个问题的影响。
- 通知线程在唤醒等待线程时可能过于「慷慨」。例如，即使只有一些等待线程的条件得到满足，通知线程也可能调用 notifyAll。
- 在没有通知的情况下，等待的线程可能（很少）醒来。这被称为伪唤醒 [POSIX, 11.4.3.6.1; Java9-api]。

一个相关的问题是，是使用 notify 还是 notifyAll 来唤醒等待的线程。（回想一下 notify 唤醒一个等待线程，假设存在这样一个线程，notifyAll 唤醒所有等待线程)。有时人们会说，应该始终使用 notifyAll。这是合理的、保守的建议。它总是会产生正确的结果，因为它保证你将唤醒需要唤醒的线程。你可能还会唤醒其他一些线程，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，如果发现为条件不满足，将继续等待。

作为一种优化，如果在等待状态的所有线程都在等待相同的条件，并且每次只有一个线程可以从条件中获益，那么你可以选择调用 notify 而不是 notifyAll。

即使满足了这些先决条件，也可能有理由使用 notifyAll 来代替 notify。正如将 wait 调用放在循环中可以防止公共访问对象上的意外或恶意通知一样，使用 notifyAll 代替 notify 可以防止不相关线程的意外或恶意等待。否则，这样的等待可能会「吞下」一个关键通知，让预期的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 就像使用「并发汇编语言」编程一样原始。**在新代码中很少有理由使用 wait 和 notify。** 如果维护使用 wait 和 notify 的代码，请确保它始终使用标准的习惯用法，即在 while 循环中调用 wait。另外，notifyAll 方法通常应该优

先于 notify。如果使用 notify，则必须非常小心以确保其活性。

Item 82: Document thread safety（文档应包含线程安全属性）

类在其方法并发使用时的行为是其与客户端约定的重要组成部分。如果你没有记录类在这一方面的行为，那么它的用户将被迫做出假设。如果这些假设是错误的，生成的程序可能缺少足够的同步（Item-78）或过度的同步（Item-79）。无论哪种情况，都可能导致严重的错误。

你可能听说过，可以通过在方法的文档中查找 synchronized 修饰符来判断方法是否线程安全。这个观点有好些方面是错误的。在正常操作中，Javadoc 的输出中没有包含同步修饰符，这是有原因的。方法声明中 synchronized 修饰符的存在是实现细节，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 synchronized 修饰符的存在就足以记录线程安全性，这个观点是对线程安全性属性的误解，认为要么全有要么全无。实际上，线程安全有几个级别。**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。** 下面的列表总结了线程安全级别。它并非详尽无遗，但涵盖以下常见情况：

- 不可变的。这个类的实例看起来是常量。不需要外部同步。示例包括 String、Long 和 BigInteger（Item-17）。
- 无条件线程安全。该类的实例是可变的，但是该类具有足够的内部同步，因此无需任何外部同步即可并发地使用该类的实例。例如 AtomicLong 和 ConcurrentHashMap。
- 有条件的线程安全。与无条件线程安全类似，只是有些方法需要外部同步才能安全并发使用。示例包括 Collections.synchronized 包装器返回的集合，其迭代器需要外部同步。
- 非线程安全。该类的实例是可变的。要并发地使用它们，客户端必须使用外部同步来包围每个方法调用（或调用序列）。这样的例子包括通用的集合实现，例如 ArrayList 和 HashMap。
- 线程对立。即使每个方法调用都被外部同步包围，该类对于并发使用也是不安全的。线程对立通常是由于在不同步的情况下修改静态数据而导致的。没有人故意编写线程对立类；此类通常是由于没有考虑并发性而导致的。当发现类或方法与线程不相容时，通常将其修复或弃用。Item-78 中的 generateSerialNumber 方法在没有内部同步的情况下是线程对立的，如第 322 页所述。

这些类别（不包括线程对立类）大致对应于《Java Concurrency in Practice》中的线程安全注解，分别为 Immutable、ThreadSafe 和 NotThreadSafe [Goetz06, Appendix A]。上面分类中的无条件线程安全和有条件的线程安全都包含在 ThreadSafe 注解中。

在文档中记录一个有条件的线程安全类需要小心。你必须指出哪些调用序列需要外部同步，以及执行这些序列必须获得哪些锁（在极少数情况下是锁）。通常是实例本身的锁，但也有例外。例如，`Collections.synchronizedMap` 的文档提到：

当用户遍历其集合视图时，必须手动同步返回的 Map：

```
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();
}
```

不遵循这个建议可能会导致不确定的行为。

类的线程安全的描述通常属于该类的文档注释，但是具有特殊线程安全属性的方法应该在它们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。除非从返回类型可以明显看出，否则静态工厂必须记录返回对象的线程安全性，正如 `Collections.synchronizedMap` 所演示的那样。

当一个类使用公共可访问锁时，它允许客户端自动执行一系列方法调用，但是这种灵活性是有代价的。它与诸如 ConcurrentHashMap 之类的并发集合所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持有可公开访问的锁来发起拒绝服务攻击。这可以是无意的，也可以是有意的。

为了防止这种拒绝服务攻击，你可以使用一个私有锁对象，而不是使用同步方法（隐含一个公共可访问的锁）：

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();
public void foo() {
    synchronized(lock) {
        ...
    }
}
```

因为私有锁对象在类之外是不可访问的，所以客户端不可能干扰对象的同步。实际上，我们通过将锁对象封装在它同步的对象中，是在应用 Item-15 的建议。

注意，lock 字段被声明为 final。这可以防止你无意中更改它的内容，这可能导致灾难性的非同步访问（Item-78）。我们正在应用 Item-17 的建议，最小化锁字段的可变性。**Lock 字段应该始终声明为 final。** 无论使用普通的监视器锁（如上所示）还是 `java.util.concurrent` 包中的锁，都是这样。

私有锁对象用法只能在无条件的线程安全类上使用。有条件的线程安全类不能使用这种用法，因为它们必须在文档中记录，在执行某些方法调用序列时要获取哪些锁。

私有锁对象用法特别适合为继承而设计的类（Item-19）。如果这样一个类要使用它的实例进行锁定，那么子类很容易在无意中干扰基类的操作，反之亦然。通过为不同的目的使用相同的锁，子类和基类最终可能「踩到对方的脚趾头」。这不仅仅是一个理论问题，它就发生在 Thread 类中 [Bloch05, Puzzle 77]。

总之，每个类都应该措辞严谨的描述或使用线程安全注解清楚地记录其线程安全属性。synchronized 修饰符在文档中没有任何作用。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，以及在执行这些序列时需要获取哪些锁。如果你编写一个无条件线程安全的类，请考虑使用一个私有锁对象来代替同步方法。这将保护你免受客户端和子类的同步干扰，并为你提供更大的灵活性，以便在后续的版本中采用复杂的并发控制方式。

Item 83: Use lazy initialization judiciously（明智地使用延迟初始化）

延迟初始化是延迟字段的初始化，直到需要它的值。如果不需要该值，则不会初始化字段。这种技术既适用于静态字段，也适用于实例字段。虽然延迟初始化主要是一种优化，it can also be used to break harmful circularities in class and instance initialization [Bloch05, Puzzle 51]。

与大多数优化一样，延迟初始化的最佳建议是「除非需要，否则不要这样做」(第67项)。延迟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，代价是增加了访问延迟初始化字段的成本。根据这些字段中最终需要初始化的部分、初始化它们的开销以及初始化后访问每个字段的频率，延迟初始化实际上会损害性能（就像许多「优化」一样）。

延迟初始化也有它的用途。如果一个字段只在类的一小部分实例上访问，并且初始化该字段的代价很高，那么延迟初始化可能是值得的。唯一确定的方法是以使用和不使用延迟初始化的效果对比来度量类的性能。

在存在多个线程的情况下，使用延迟初始化很棘手。如果两个或多个线程共享一个延迟初始化的字段，那么必须使用某种形式的同步，否则会导致严重的错误（Item-78）。本条目讨论的所有初始化技术都是线程安全的。

在大多数情况下，常规初始化优于延迟初始化。 下面是一个使用常规初始化的实例字段的典型声明。注意 final 修饰符的使用（Item-17）：

```
// Normal initialization of an instance field
private final FieldType field = computeFieldValue();
```

如果您使用延迟初始化来取代初始化 circularity，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;
}
```

这两种习惯用法（使用同步访问器进行常规初始化和延迟初始化）在应用于静态字段时都没有改变，只是在字段和访问器声明中添加了 static 修饰符。

如果需要在静态字段上使用延迟初始化来提高性能，use the lazy initialization holder class idiom. 这个用法可保证一个类在使用之前不会被初始化 [JLS, 12.4.1]。它是这样的：

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}
private static FieldType getField() { return FieldHolder.field; }
```

第一次调用 getField 时，它执行 FieldHolder.field，导致初始化 FieldHolder 类。这个习惯用法的优点是 getField 方法不是同步的，只执行字段访问，所以延迟初始化实际上不会增加访问成本。典型的 VM 只会同步字段访问来初始化类。初始化类之后，VM 会对代码进行修补，这样对字段的后续访问就不会涉及任何测试或同步。

如果需要使用延迟初始化来提高实例字段的性能，请使用双重检查模式。这个模式避免了初始化后访问字段时的锁定成本（Item-79）。这个模式背后的思想是两次检查字段的值（因此得名 double check）：一次没有锁定，然后，如果字段没有初始化，第二次使用锁定。只有当第二次检查指示字段未初始化时，调用才初始化字段。由于初始化字段后没有锁定，因此将字段声明为 volatile 非常重要（Item-78）。下面是这个模式的示例：

```
// 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;
}
```

这段代码可能看起来有点复杂。特别是不清楚是否需要局部变量（result）。该变量的作用是确保 field 在已经初始化的情况下只读取一次。

虽然不是严格必需的，但这可能会提高性能，而且与低级并发编程相比，这更优雅。在我的机器上，上述方法的速度大约是没有局部变量版本的 1.4 倍。虽然您也可以将双重检查模式应用于静态字段，但是没有理由这样做：the lazy initialization holder class idiom is a better choice.

双重检查模式的两个变体值得注意。有时候，您可能需要延迟初始化一个实例字段，该字段可以容忍重复初始化。如果您发现自己处于这种情况，您可以使用双重检查模式的变体来避免第二个检查。毫无疑问，这就是所谓的「单检查」模式。它是这样的。注意，field 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;
}
```

本条目中讨论的所有初始化技术都适用于基本字段和对象引用字段。当双检查或单检查模式应用于数值基本类型字段时，将根据 0（数值基本类型变量的默认值）而不是 null 检查字段的值。

如果您不关心每个线程是否都会重新计算字段的值，并且字段的类型是 long 或 double 之外的基本类型，那么您可以选择在单检查模式中从字段声明中删除 volatile 修饰符。这种变体称为原生单检查模式。它加快了某些架构上的字段访问速度，代价是需要额外的初始化（每个访问该字段的线程最多需要一个初始化）。这绝对是一种奇特的技术，不是日常使用的。

总之，您应该正常初始化大多数字段，而不是延迟初始化。如果必须延迟初始化字段以实现性能目标或 break a harmful initialization circularity，则使用适当的延迟初始化技术。对于字段，使用双重检查模式；对于静态字段，the lazy initialization holder class idiom. 例如，可以容忍重复初始化的实例字段，您还可以考虑单检查模式。

Item 84: Don’t depend on the thread scheduler（不要依赖线程调度器）

当许多线程可以运行时，线程调度器决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，但是策略可能会有所不同。因此，编写良好的程序不应该依赖于此策略的细节。**任何依赖线程调度器来保证正确性或性能的程序都可能是不可移植的。**

编写健壮、响应快、可移植程序的最佳方法是确保可运行线程的平均数量不显著大于处理器的数量。这使得线程调度器几乎没有选择：它只运行可运行线程，直到它们不再可运行为止。即使在完全不同的线程调度策略下，程序的行为也没有太大的变化。注意，可运行线程的数量与线程总数不相同，后者可能更高。正在等待的线程不可运行。

保持可运行线程数量低的主要技术是让每个线程做一些有用的工作，然后等待更多的工作。**如果线程没有做有用的工作，它们就不应该运行。** 对于 Executor 框架（Item-80），这意味着适当调整线程池的大小 [Goetz06, 8.2]，并保持任务短小（但不要太短），否则分派开销依然会损害性能。

线程不应该处于循环检查共享对象状态变化。除了使程序容易受到线程调度器变化无常的影响之外，循环检查状态变化还大大增加了处理器的负载，还影响其他线程获取处理器进行工作。作为反面的极端例子，考虑一下 CountDownLatch 的不正确的重构实现：

```
// 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 个线程等待一个锁存器时，SlowCountDownLatch 的速度大约是 Java 的 CountDownLatch 的 10 倍。虽然这个例子看起来有点牵强，但是具有一个或多个不必要运行的线程的系统并不少见。性能和可移植性可能会受到影响。

当面对一个几乎不能工作的程序时，而原因是由于某些线程相对于其他线程没有获得足够的 CPU 时间，那么 **通过调用 `Thread.yield` 来「修复」程序** 你也许能勉强让程序运行起来，但它是不可移植的。在一个 JVM 实现上提高性能的相同的 yield 调用，在一些JVM 实现上可能会使性能变差，而在其他 JVM 实现上可能没有任何影响。 **`Thread.yield` 没有可测试的语义。** 更好的做法是重构应用程序，以减少并发运行线程的数量。

一个相关的技术是调整线程优先级，类似的警告也适用于此技术，即，线程优先级是 Java 中最不可移植的特性之一。通过调整线程优先级来调优应用程序的响应性并非不合理，但很少情况下是必要的，而且不可移植。试图通过调整线程优先级来解决严重的活性问题是不合理的。在找到并修复潜在原因之前，问题很可能会再次出现。

总之，不要依赖线程调度器来判断程序的正确性。生成的程序既不健壮也不可移植。因此，不要依赖 `Thread.yield` 或线程优先级。这些工具只是对调度器的提示。线程优先级可以少量地用于提高已经工作的程序的服务质量，但绝不应该用于「修复」几乎不能工作的程序。

Chapter 12. Serialization（序列化）

Chapter 12 Introduction（章节介绍）

THIS chapter concerns object serialization, which is Java's framework for encoding objects as byte streams (serializing) and reconstructing objects from their encodings (deserializing). Once an object has been serialized, its encoding can be sent from one VM to another or stored on disk for later deserialization. This chapter focuses on the dangers of serialization and how to minimize them.

本章关注对象序列化，它是 Java 的框架，用于将对象编码为字节流（序列化），并从对象的编码中重构对象（反序列化）。对象序列化后，可以将其编码从一个 VM 发送到另一个 VM，或者存储在磁盘上，以便今后反序列化。本章主要讨论序列化的风险以及如何将其最小化。

Item 85: Prefer alternatives to Java serialization（Java 序列化的替代方案）

当序列化在 1997 年添加到 Java 中时，它被认为有一定的风险。这种方法曾在研究语言（Modula-3）中尝试过，但从未在生产语言中使用过。虽然程序员不费什么力气就能实现分布式对象，这一点很吸引人，但代价也不小，如：不可见的构造函数、API 与实现之间模糊的界线，还可能会出现正确性、性能、安全性和维护方面的问题。支持者认为收益大于风险，但历史证明并非如此。

在本书之前的版本中描述的安全问题，和人们担心的一样严重。21 世纪初仅停留在讨论的漏洞在接下来的 10 年间变成了真实严重的漏洞，其中最著名的包括 2016 年 11 月对旧金山大都会运输署市政铁路（SFMTA Muni）的勒索软件攻击，导致整个收费系统关闭了两天 [Gallagher16]。

序列化的一个根本问题是它的可攻击范围太大，且难以保护，而且问题还在不断增多：通过调用 ObjectInputStream 上的 readObject 方法反序列化对象图。这个方法本质上是一个神奇的构造函数，可以用来实例化类路径上几乎任何类型的对象，只要该类型实现 Serializable 接口。在反序列化字节流的过程中，此方法可以执行来自任何这些类型的代码，因此所有这些类型的代码都在攻击范围内。

攻击可涉及 Java 平台库、第三方库（如 Apache Commons collection）和应用程序本身中的类。即使坚持履行实践了所有相关的最佳建议，并成功地编写了不受攻击的可序列化类，应用程序仍然可能是脆弱的。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真实的危险源，因为它被应用程序直接和间接地广泛使用，比如 RMI（远程方法调用）、JMX（Java 管理扩展）和 JMS（Java 消息传递系统）。不可信流的反序列化可能导致远程代码执行（RCE）、拒绝服务（DoS）和一系列其他攻击。应用程序很容易受到这些攻击，即使它们本身没有错误。[Seacord17]

攻击者和安全研究人员研究 Java 库和常用的第三方库中的可序列化类型，寻找在反序列化过程中调用的潜在危险活动的方法称为 gadget。多个小工具可以同时使用，形成一个小工具链。偶尔会发现一个小部件链，它的功能足够强大，允许攻击者在底层硬件上执行任意的本机代码，允许提交精心设计的字节流进行反序列化。这正是 SFMTA Muni 袭击中发生的事情。这次袭击并不是孤立的。不仅已经存在，而且还会有更多。

不使用任何 gadget，你都可以通过对需要很长时间才能反序列化的短流进行反序列化，轻松地发起拒绝服务攻击。这种流被称为反序列化炸弹 [Svoboda16]。下面是 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 5744 字节，但是在你对其进行反序列化之前，资源就已经耗尽了。问题在于，反序列化 HashSet 实例需要计算其元素的哈希码。根哈希集的 2 个元素本身就是包含 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，以此类推，深度为 100。因此，反序列化 Set 会导致 hashCode 方法被调用超过 2100 次。除了反序列化会持续很长时间之外，反序列化器没有任何错误的迹象。生成的对象很少，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？当你反序列化一个你不信任的字节流时，你就会受到攻击。**避免序列化利用的最好方法是永远不要反序列化任何东西。** 用 1983 年电影《战争游戏》（WarGames）中名为约书亚（Joshua）的电脑的话来说，「唯一的制胜绝招就是不玩。」**没有理由在你编写的任何新系统中使用 Java 序列化。** 还有其他一些机制可以在对象和字节序列之间进行转换，从而避免了 Java 序列化的许多危险，同时还提供了许多优势，比如跨平台支持、高性能、大量工具和广泛的专家社区。在

本书中，我们将这些机制称为跨平台结构数据表示。虽然其他人有时将它们称为序列化系统，但本书避免使用这种说法，以免与 Java 序列化混淆。

以上所述技术的共同点是它们比 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，它们支持简单的结构化数据对象，由一组「属性-值」对组成。只有少数基本数据类型和数组数据类型得到支持。事实证明，这个简单的抽象足以构建功能极其强大的分布式系统，而且足够简单，可以避免 Java 序列化从一开始就存在的严重问题。

领先的跨平台结构化数据表示是 JSON 和 Protocol Buffers，也称为 protobuf。JSON 由 Douglas Crockford 设计用于浏览器与服务器通信，Protocol Buffers 由谷歌设计用于在其服务器之间存储和交换结构化数据。尽管这些技术有时被称为「中性语言」，但 JSON 最初是为 JavaScript 开发的，而 protobuf 是为 c++ 开发的；这两种技术都保留了其起源的痕迹。

JSON 和 protobuf 之间最显著的区别是 JSON 是基于文本的，并且是人类可读的，而 protobuf 是二进制的，但效率更高；JSON 是一种专门的数据表示，而 protobuf 提供模式（类型）来记录和执行适当的用法。虽然 protobuf 比 JSON 更有效，但是 JSON 对于基于文本的表示非常有效。虽然 protobuf 是一种二进制表示，但它确实提供了另一种文本表示，可用于需要具备人类可读性的场景（pbtxt）。

如果你不能完全避免 Java 序列化，可能是因为你需要在遗留系统环境中工作，那么你的下一个最佳选择是 **永远不要反序列化不可信的数据。** 特别要注意，你不应该接受来自不可信来源的 RMI 流量。Java 的官方安全编码指南说：「反序列化不可信的数据本质上是危险的，应该避免。」这句话是用大号、粗体、斜体和红色字体设置的，它是整个文档中唯一得到这种格式处理的文本。[Java-secure]

如果无法避免序列化，并且不能绝对确定反序列化数据的安全性，那么可以使用 Java 9 中添加的对象反序列化筛选，并将其移植到早期版本（java.io.ObjectInputFilter）。该工具允许你指定一个过滤器，该过滤器在反序列化数据流之前应用于数据流。它在类粒度上运行，允许你接受或拒绝某些类。默认接受所有类，并拒绝已知潜在危险类的列表称为黑名单；在默认情况下拒绝其他类，并接受假定安全的类的列表称为白名单。**优先选择白名单而不是黑名单，** 因为黑名单只保护你免受已知的威胁。一个名为 Serial Whitelist Application Trainer（SWAT）的工具可用于为你的应用程序自动准备一个白名单 [Schneider16]。过滤工具还将保护你免受过度内存使用和过于深入的对象图的影响，但它不能保护你免受如上面所示的序列化炸弹的影响。

不幸的是，序列化在 Java 生态系统中仍然很普遍。如果你正在维护一个基于 Java 序列化的系统，请认真考虑迁移到跨平台的结构化数据，尽管这可能是一项耗时的工作。实际上，你可能仍然需要编写或维护一个可序列化的类。编写一个正确、安全、高效的可序列化类需要非常小心。本章的其余部分将提供何时以及如何进行此操作的建议。

总之，序列化是危险的，应该避免。如果你从头开始设计一个系统，可以使用跨平台的结构化数据，如 JSON 或 protobuf。不要反序列化不可信的数据。如果必须这样做，请使用对象反序列化过滤，但要注意，它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，一定要非常小心。

Item 86: Implement Serializable with great caution（非常谨慎地实现 Serializable）

使类的实例可序列化非常简单，只需实现 Serializable 接口即可。因为这很容易做到，所以有一个普遍的误解，认为序列化只需要程序员付出很少的努力。而事实上要复杂得多。虽然使类可序列化的即时代价可以忽略不计，但长期代价通常是巨大的。

实现 Serializable 接口的一个主要代价是，一旦类的实现被发布，它就会降低更改该类实现的灵活性。 当类实现 Serializable 时，其字节流编码（或序列化形式）成为其导出 API 的一部分。一旦广泛分发了一个类，通常就需要永远支持序列化的形式，就像需要支持导出 API 的所有其他部分一样。如果你不努力设计自定义序列化形式，而只是接受默认形式，则序列化形式将永远绑定在类的原始内部实现上。换句话说，如果你接受默认的序列化形式，类中私有的包以及私有实例字段将成为其导出 API 的一部分，此时最小化字段作用域（Item-15）作为信息隐藏的工具，将失去其有效性。

如果你接受默认的序列化形式，然后更改了类的内部实现，则会导致与序列化形式不兼容。试图使用类的旧版本序列化实例，再使用新版本反序列化实例的客户端（反之亦然）程序将会失败。当然，可以在维护原始序列化形式的同时更改内部实现（使用 ObjectOutputStream.putFields 或 ObjectInputStream.readFields），但这可能会很困难，并在源代码中留下明显的缺陷。如果你选择使类可序列化，你应该仔细设计一个高质量的序列化形式，以便长期使用（Item-87、Item-90）。这样做会增加开发的初始成本，但是这样做是值得的。即使是设计良好的序列化形式，也会限制类的演化；而设计不良的序列化形式，则可能会造成严重后果。

可序列化会使类的演变受到限制，施加这种约束的一个简单示例涉及流的唯一标识符，通常称其为串行版本 UID。每个可序列化的类都有一个与之关联的唯一标识符。如果你没有通过声明一个名为 serialVersionUID 的静态 final long 字段来指定这个标识符，那么系统将在运行时对类应用加密散列函数（SHA-1）自动生成它。这个值受到类的名称、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。如果你更改了其中任何一项，例如，通过添加一个临时的方法，生成的序列版本 UID 就会更改。如果你未能声明序列版本 UID，兼容性将被破坏，从而在运行时导致 InvalidClassException。

实现 Serializable 接口的第二个代价是，增加了出现 bug 和安全漏洞的可能性(第85项)。 通常，对象是用构造函数创建的；序列化是一种用于创建对象的超语言机制。无论你接受默认行为还是无视它，反序列化都是一个「隐藏构造函数」，其他构造函数具有的所有问题它都有。由于没有与反序列化关联的显式构造函数，因此很容易忘记必须让它能够保证所有的不变量都是由构造函数建立的，并且不允许攻击者访问正在构造的对象内部。依赖于默认的反序列化机制，会让对象轻易地遭受不变性破坏和非法访问（Item-88）。

实现 Serializable 接口的第三个代价是，它增加了与发布类的新版本相关的测试负担。 当一个可序列化的类被修改时，重要的是检查是否可以在新版本中序列化一个实例，并在旧版本中反序列化它，反之亦然。因此，所需的测试量与可序列化类的数量及版本的数量成正比，工作量可能很大。你必须确保「序列化-反序列化」过程成功，并确保它生成原始对象的无差错副本。如果在第一次编写类时精心设计了

自定义序列化形式，那么测试的工作量就会减少（Item-87、Item-90）。

实现 Serializable 接口并不是一个轻松的决定。 如果一个类要参与一个框架，该框架依赖于 Java 序列化来进行对象传输或持久化，这对于类来说实现 Serializable 接口就是非常重要的。此外，如果类 A 要成为另一个类 B 的一个组件，类 B 必须实现 Serializable 接口，若类 A 可序列化，它就会更易于被使用。然而，与实现 Serializable 相关的代价很多。每次设计一个类时，都要权衡利弊。历史上，像 BigInteger 和 Instant 这样的值类实现了 Serializable 接口，集合类也实现了 Serializable 接口。表示活动实体（如线程池）的类很少情况适合实现 Serializable 接口。

为继承而设计的类（Item-19）很少情况适合实现 Serializable 接口，接口也很少情况适合扩展它。 违反此规则会给扩展类或实现接口的任何人带来很大的负担。有时，违反规则是恰当的。例如，如果一个类或接口的存在主要是为了参与一个要求所有参与者都实现 Serializable 接口的框架，那么类或接口实现或扩展 Serializable 可能是有意义的。

在为了继承而设计的类中，Throwable 类和 Component 类都实现了 Serializable 接口。正是因为 Throwable 实现了 Serializable 接口，RMI 可以将异常从服务器发送到客户端；Component 类实现了 Serializable 接口，因此可以发送、保存和恢复 GUI，但即使在 Swing 和 AWT 的鼎盛时期，这个工具在实践中也很少使用。

如果你实现了一个带有实例字段的类，它同时是可序列化和可扩展的，那么需要注意几个风险。如果实例字段值上有任何不变量，关键是要防止子类覆盖 finalize 方法，可以通过覆盖 finalize 并声明它为 final 来做到。最后，如果类的实例字段初始化为默认值（整数类型为 0，布尔值为 false，对象引用类型为 null），那么必须添加 readObjectNoData 方法：

```
// readObjectNoData for stateful extendable serializable classes
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

这个方法是在 Java 4 中添加的，涉及将可序列化超类添加到现有可序列化类 [Serialization, 3.5] 的特殊情况。

关于不实现 Serializable 的决定，有一个警告。如果为继承而设计的类不可序列化，则可能需要额外的工作来编写可序列化的子类。子类的常规反序列化，要求超类具有可访问的无参数构造函数 [Serialization, 1.10]。如果不提供这样的构造函数，子类将被迫使用序列化代理模式（Item-90）。

内部类（Item-24）不应该实现 Serializable。 它们使用编译器生成的合成字段存储对外围实例的引用，并存储来自外围的局部变量的值。这些字段与类定义的对应关系，就和没有指定匿名类和局部类的名称一样。因此，内部类的默认序列化形式是不确定的。但是，静态成员类可以实现 Serializable 接口。

总而言之，认为实现 Serializable 接口很简单这个观点似是而非。除非类只在受保护的环境中使用，在这种环境中，版本永远不必互操作，服务器永远不会暴露不可信的数据，否则实现 Serializable 接口是一项严肃的事情，应该非常小心。如果类允许继承，则更加需要格外小心。

Item 87: Consider using a custom serialized form（考虑使用自定义序列化形式）

当你在时间紧迫的情况下编写类时，通常应该将精力集中在设计最佳 API 上。有时，这意味着发布一个「一次性」实现，你也知道在将来的版本中会替换它。通常这不是一个问题，但是如果类实现 Serializable 接口并使用默认的序列化形式，你将永远无法完全摆脱这个「一次性」的实现。它将永远影响序列化的形式。这不仅仅是一个理论问题。这种情况发生在 Java 库中的几个类上，包括 BigInteger。

在没有考虑默认序列化形式是否合适之前，不要接受它。 接受默认的序列化形式应该是一个三思而后行的决定，即从灵活性、性能和正确性的角度综合来看，这种编码是合理的。一般来说，设计自定义序列化形式时，只有与默认序列化形式所选择的编码在很大程度上相同时，才应该接受默认的序列化形式。

对象的默认序列化形式，相对于它的物理表示法而言是一种比较有效的编码形式。换句话说，它描述了对象中包含的数据以及从该对象可以访问的每个对象的数据。它还描述了所有这些对象相互关联的拓扑结构。理想的对象序列化形式只包含对象所表示的逻辑数据。它独立于物理表征。

如果对象的物理表示与其逻辑内容相同，则默认的序列化形式可能是合适的。 例如，默认的序列化形式对于下面的类来说是合理的，它简单地表示一个人的名字：

```
// 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，名字由三个字符串组成，分别表示姓、名和中间名。Name 的实例字段精确地反映了这个逻辑内容。

即使你认为默认的序列化形式是合适的，你通常也必须提供 readObject 方法来确保不变性和安全性。对于 Name 类而言, readObject 方法必须确保字段 lastName 和 firstName 是非空的。Item-88 和 Item-90 详细讨论了这个问题。

注意，虽然 lastName、firstName 和 middleName 字段是私有的，但是它们都有文档注释。这是因为这些私有字段定义了一个公共 API，它是类的序列化形式，并且必须对这个公共 API 进行文档化。`@serial` 标记的存在告诉 Javadoc 将此文档放在一个特殊的页面上，该页面记录序列化的形式。

与 Name 类不同，考虑下面的类，它是另一个极端。它表示一个字符串列表（使用标准 List 实现可能更好，但此时暂不这么做）：

```
    // 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，这个类表示字符串序列。在物理上，它将序列表示为双向链表。如果接受默认的序列化形式，该序列化形式将不遗余力地镜像出链表中的所有项，以及这些项之间的所有双向链接。

当对象的物理表示与其逻辑数据内容有很大差异时，使用默认的序列化形式有四个缺点：

- **它将导出的 API 永久地绑定到当前的内部实现。** 在上面的例子中，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实现，StringList 类仍然需要接受链表形式的输

出，并产生链表形式的输出。这个类永远也摆脱不掉处理链表项所需要的所有代码，即使不再使用链表作为内部数据结构。

- **它会占用过多的空间。** 在上面的示例中，序列化的形式不必要地表示链表中的每个条目和所有链接关系。这些链表项以及链接只不过是实现细节，不值得记录在序列化形式中。因为这样的序列化形式过于庞大，将其写入磁盘或通过网络发送将非常慢。
- **它会消耗过多的时间。** 序列化逻辑不知道对象图的拓扑结构，因此必须遍历开销很大的图。在上面的例子中，只要遵循 next 的引用就足够了。
- **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，即使对于中等规模的对象图，这也可能导致堆栈溢出。用 1000-1800 个元素序列化 StringList 实例会在我的机器上生成一个 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；有些实现可能根本没有这个问题。

StringList 的合理序列化形式就是列表中的字符串数量，然后是字符串本身。这构成了由 StringList 表示的逻辑数据，去掉了其物理表示的细节。下面是修改后的 StringList 版本，带有实现此序列化形式的 writeObject 和 readObject 方法。提醒一下，transient 修饰符表示要从类的默认序列化表单中省略该实例字段：

```
// 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 做的第一件事是调用 defaultWriteObject, readObject 做的第一件事是调用 defaultReadObject，即使 StringList 的所有字段都是 transient 的。你可能听说过，如果一个类的所有实例字段都是 transient 的，那么你可以不调用 defaultWriteObject 和 defaultReadObject，但是序列化规范要求你无论如何都要调用它们。这些调用的存在使得在以后的版本中添加非瞬态实例字段成为可能，同时保留了向后和向前兼容性。如果实例在较晚的版本中序列化，在较早的版本中反序列化，则会忽略添加的字段。如果早期版本的 readObject 方法调用 defaultReadObject 失败，反序列化将失败，并出现 StreamCorruptedException。

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。这个私有方法定义了一个公共 API，它是序列化的形式，并且应该对该公共 API 进行文档化。与字段的 `@serial` 标记一样，方法的 `@serialData` 标记告诉 Javadoc 实用工具将此文档放在序列化形式页面上。

为了给前面的性能讨论提供一定的伸缩性，如果平均字符串长度是 10 个字符，那么经过修改的 StringList 的序列化形式占用的空间大约是原始字符串序列化形式的一半。在我的机器上，序列化修订后的 StringList 的速度是序列化原始版本的两倍多，列表长度为 10。最后，在修改后的形式中没有堆栈溢出问题，因此对于可序列化的 StringList 的大小没有实际的上限。

虽然默认的序列化形式对 StringList 不好，但是对于某些类来说，情况会更糟。对于 StringList，默认的序列化形式是不灵活的，并且执行得很糟糕，但是它是正确的，因为序列化和反序列化 StringList 实例会生成原始对象的无差错副本，而所有不变量都是完整的。对于任何不变量绑定到特定于实现的细节的对象，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」项的哈希桶序列。一个项所在的桶是其键的散列代码的函数，通常情况下，不能保证从一个实现到另一个实现是相同的。事实上，它甚至不能保证每次运行都是相同的。因此，接受哈希表的默认序列化形式将构成严重的 bug。对哈希表进行序列化和反序列化可能会产生一个不变量严重损坏的对象。

无论你是否接受默认的序列化形式，当调用 defaultWriteObject 方法时，没有标记为 transient 的每个实例字段都会被序列化。因此，可以声明为 transient 的每个实例字段都应该做这个声明。这包括派生字段，其值可以从主数据字段（如缓存的哈希值）计算。它还包括一些字段，这些字段的值与 JVM 的一个特定运行相关联，比如表示指向本机数据结构指针的 long 字段。**在决定使字段非 transient 之前，请确信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，大多数或所有实例字段都应该标记为 transient，如上面的 StringList 示例所示。

如果使用默认的序列化形式，并且标记了一个或多个字段为 transient，请记住，当反序列化实例时，这些字段将初始化为默认值：对象引用字段为 null，数字基本类型字段为 0，布尔字段为 false [JLS, 4.12.5]。如果这些值对于任何 transient 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 transient 字段恢复为可接受的值（Item-88）。或者，可以采用延迟初始化（Item-83），在第一次使用这些字段时初始化它们。

无论你是否使用默认的序列化形式，**必须对对象序列化强制执行任何同步操作，就像对读取对象的整个状态的任何其他方法强制执行的那样。** 例如，如果你有一个线程安全的对象（Item-82），它通过同步每个方法来实现线程安全，并且你选择使用默认的序列化形式，那么使用以下 write-Object 方法：

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
}
```

如果将同步放在 writeObject 方法中，则必须确保它遵守与其他活动相同的锁排序约束，否则将面临资源排序死锁的风险 [Goetz06, 10.1.5]。

无论选择哪种序列化形式，都要在编写的每个可序列化类中声明显式的序列版本 UID。 这消除了序列版本 UID 成为不兼容性的潜在来源（Item-86）。这么做还能获得一个小的性能优势。如果没有提供序列版本 UID，则需要执行高开销的计算在运行时生成一个 UID。

声明序列版本 UID 很简单，只要在你的类中增加这一行：

```
private static final long serialVersionUID = randomLongValue;
```

如果你编写一个新类，为 randomLongValue 选择什么值并不重要。你可以通过在类上运行 serialver 实用工具来生成该值，但是也可以凭空选择一个数字。串行版本 UID 不需要是唯一的。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。你可以通过在类的旧版本上运行 serialver 实用工具（序列化实例存在于旧版本上）来获得这个数字。

如果你希望创建一个新版本的类，它与现有版本不兼容，如果更改序列版本 UID 声明中的值，这将导致反序列化旧版本的序列化实例的操作引发 InvalidClassException。**不要更改序列版本 UID，除非你想破坏与现有序列化所有实例的兼容性。**

总而言之，如果你已经决定一个类应该是可序列化的（Item-86），那么请仔细考虑一下序列化的形式应该是什么。只有在合理描述对象的逻辑状态时，才使用默认的序列化形式；否则，设计一个适合描述对象的自定义序列化形式。设计类的序列化形式应该和设计导出方法花的时间应该一样多，都应该严谨对待（Item-51）。正如不能从未来版本中删除导出的方法一样，也不能从序列化形式中删除字段；必须永远保存它们，以确保序列化兼容性。选择错误的序列化形式可能会对类的复杂性和性能产生永久性的负面影响。

Item 88: Write readObject methods defensively（防御性地编写 readObject 方法）

Item-50 包含一个具有可变私有 Date 字段的不可变日期范围类。该类通过在构造函数和访问器中防御性地复制 Date 对象，不遗余力地保持其不变性和不可变性。它是这样的：

```
// 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
    }
```

假设你决定让这个类可序列化。由于 Period 对象的物理表示精确地反映了它的逻辑数据内容，所以使用默认的序列化形式是合理的（Item-87）。因此，要使类可序列化，似乎只需将实现 Serializable 接口。但是，如果这样做，该类将不再保证它的临界不变量。

问题是 readObject 方法实际上是另一个公共构造函数，它与任何其他构造函数有相同的注意事项。如，构造函数必须检查其参数的有效性（Item-49）并在适当的地方制作防御性副本（Item-50）一样，readObject 方法也必须这样做。如果 readObject 方法没有做到这两件事中的任何一件，那么攻击者就很容易违反类的不变性。

不严格地说，readObject 是一个构造函数，它唯一的参数是字节流。在正常使用中，字节流是通过序列化一个正常构造的实例生成的。当 readObject 呈现一个字节流时，问题就出现了，这个字节流是人为构造的，用来生成一个违反类不变性的对象。这样的字节流可用于创建一个不可思议的对象，而该对象不能使用普通构造函数创建。

假设我们只是简单地让 Period 实现 Serializable 接口。然后，这个有问题的程序将生成一个 Period 实例，其结束比起始时间还要早。对其高位位设置的字节值进行强制转换，这是由于 Java 缺少字节字面值，再加上让字节类型签名的错误决定导致的：

```
    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 的字节数组文本是通过序列化一个普通 Period 实例并手工编辑得到的字节流生成的。流的细节对示例并不重要，但是如果你感兴趣，可以在《JavaTM Object Serialization Specification》[serialization, 6]中查到序列化字节流的格式描述。如果你运行这个程序，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 可序列化，就可以创建一个违反其类不变性的对象。

要解决此问题，请为 Period 提供一个 readObject 方法，该方法调用 defaultReadObject，然后检查反序列化对象的有效性。如果有效性检查失败，readObject 方法抛出 InvalidObjectException，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，但还有一个更微妙的问题仍然潜伏着。可以通过字节流来创建一个可变的 Period 实例，该字节流以一个有效的 Period 实例开始，然后向 Period 实例内部的私有日期字段追加额外的引用。攻击者从 ObjectInputStream 中读取 Period 实例，然后读取附加到流中的「恶意对象引用」。这些引用使攻击者能够访问 Period 对象中的私有日期字段引用的对象。通过修改这些日期实例，攻击者可以修改 Period 实例。下面的类演示了这种攻击：

```
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);
        }
    }
}
```

要查看攻击的实际效果，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，运行这个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 Period 实例时保留了它的不变性，但是可以随意修改它的内部组件。一旦拥有一个可变的 Period 实例，攻击者可能会将实例传递给一个依赖于 Period 的不变性来保证其安全性的类，从而造成极大的危害。这并不是牵强附会的：有些类依赖于 String 的不变性来保证其安全。

问题的根源在于 Period 的 readObject 方法没有进行足够的防御性复制。**当对象被反序列化时，对任何客户端不能拥有的对象引用的字段进行防御性地复制至关重要。** 因此，对于每个可序列化的不可变类，如果它包含了私有的可变组件，那么在它的 readObjec 方法中，必须要对这些组件进行防御性地复制。下面的 readObject 方法足以保证周期的不变性，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);
}
```

注意，防御副本是在有效性检查之前执行的，我们没有使用 Date 的 clone 方法来执行防御副本。这两个细节对于保护 Period 免受攻击是必要的(第50项)。还要注意，防御性复制不可能用于 final 字段。要使用 readObject 方法，必须使 start 和 end 字段非 final。这是不幸的，但却是权衡利弊后的方案。使用新的 readObject 方法，并从 start 和 end 字段中删除 final 修饰符，MutablePeriod 类将无效。上面的攻击程序现在生成这个输出：

```
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
```

下面是一个简单的测试，用于判断默认 readObject 方法是否可用于类：你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 transient 字段的值作为参数，并在没有任何验证的情况下将值存储在字段中？如果没有，则必须提供 readObject 方法，并且它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，你可以使用序列化代理模式（Item-90）。强烈推荐使用这种模式，否则会在安全反序列化方面花费大量精力。

readObject 方法和构造函数之间还有一个相似之处，适用于非 final 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（Item-19）。如果违反了这条规则，并且涉及的方法被覆盖，则覆盖方法将在子类的状态反序列化之前运行。很可能导致程序失败 [Bloch05, Puzzle 91]。

总而言之，无论何时编写 readObject 方法，都要采用这样的思维方式，即编写一个公共构造函数，该构造函数必须生成一个有效的实例，而不管给定的是什么字节流。不要假设字节流表示实际的序列化实例。虽然本条目中的示例涉及使用默认序列化形式的类，但是所引发的所有问题都同样适用于具有自定义序列化形式的类。下面是编写 readObject 方法的指导原则：

- 对象引用字段必须保持私有的的类，应防御性地复制该字段中的每个对象。不可变类的可变组件属于这一类。
- 检查任何不变量，如果检查失败，则抛出 InvalidObjectException。检查动作应该跟在任何防御性复制之后。
- 如果必须在反序列化后验证整个对象图，那么使用 ObjectInputValidation 接口（在本书中没有讨论）。
- 不要直接或间接地调用类中任何可被覆盖的方法。

Item 89: For instance control, prefer enum types to readResolve（对于实例控制，枚举类型优于 readResolve）

Item-3 描述了单例模式，并给出了下面的单例类示例。该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 Item-3 所述，如果实现 Serializable 接口，该类将不再是单例的。类使用默认序列化形式还是自定义序列化形式并不重要（Item-87），类是否提供显式 readObject 方法也不重要（Item-88）。任何 readObject 方法，不管是显式的还是默认的，都会返回一个新创建的实例，这个实例与类初始化时创建的实例不同。

readResolve 特性允许你用另一个实例替换 readObject[Serialization, 3.7] 创建的实例。如果正在反序列化的对象的类定义了 readResolve 方法，新创建的对象反序列化之后，将在该对象上调用该方法。该方法返回的对象引用将代替新创建的对象返回。在该特性的大多数使用中，不保留对新创建对象的引用，因此它立即就有资格进行垃圾收集。

如果 Elvis 类要实现 Serializable 接口，下面的 readResolve 方法就足以保证其单例属性：

```
// 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，返回初始化类时创建的特殊 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；所有实例字段都应该声明为 transient。事实上，**如果你依赖 readResolve 进行实例控制，那么所有具有对象引用类型的实例字段都必须声明为 transient。** 否则，有的攻击者有可能在运行反序列化对象的 readResolve 方法之前保护对该对象的引用，使用的技术有点类似于 Item-88 中的 MutablePeriod 攻击。

攻击有点复杂，但其基本思想很简单。如果单例包含一个非 transient 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。这允许一个精心设计的流在对象引用字段的内容被反序列化时「窃取」对原来反序列化的单例对象的引用。

下面是它的工作原理。首先，编写一个 stealer 类，该类具有 readResolve 方法和一个实例字段，该实例字段引用序列化的单例，其中 stealer 「隐藏」在其中。在序列化流中，用一个 stealer 实例替换单例的非 transient 字段。现在你有了一个循环：单例包含了 stealer，而 stealer 引用了单例。

因为单例包含 stealer，所以当反序列化单例时，窃取器的 readResolve 方法首先运行。因此，当 stealer 的 readResolve 方法运行时，它的实例字段仍然引用部分反序列化（且尚未解析）的单例。

stealer 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，该方法为其隐藏的字段返回正确类型的值。如果不这样做，当序列化系统试图将 stealer 引用存储到该字段时，VM 将抛出 ClassCastException。

要使问题具体化，请考虑以下被破坏的单例：

```
// 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;
    }
}
```

这里是一个 stealer 类，按照上面的描述构造：

```
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，这是一个有问题的程序，它反序列化了一个手工制作的流，以生成有缺陷的单例的两个不同实例。这个程序省略了反序列化方法，因为它与第 354 页的方法相同：

```
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();
        }
    }
```

运行此程序将生成以下输出，最终证明可以创建两个不同的 Elvis 实例（具有不同的音乐品味）：

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

通过将 favorites 字段声明为 transient 可以解决这个问题，但是最好把 Elvis 做成是一个单元素的枚举类型（Item-3）。ElvisStealer 所示的攻击表名，使用 readResolve 方法防止「temporary」反序列化实例被攻击者访问的方式是脆弱的，需要非常小心。

如果你将可序列化的实例控制类编写为枚举类型, Java 保证除了声明的常量之外不会有任何实例，除非攻击者滥用了特权方法，如 `AccessibleObject.setAccessible` 。任何能够做到这一点的攻击者都已经拥有足够的特权来执行任意的本地代码，all bets are off。以下是把 Elvis 写成枚举的例子：

```
// 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 进行实例控制并不过时。如果必须编写一个可序列化的实例控制类，而该类的实例在编译时是未知的，则不能将该类表示为枚举类型。

readResolve 的可访问性非常重要。 如果你将 readResolve 方法放在 final 类上，那么它应该是私有的。如果将 readResolve 方法放在非 final 类上，必须仔细考虑其可访问性。如果它是私有的，它将不应用于任何子类。如果它是包级私有的，它将只适用于同一包中的子类。如果它是受保护的或公共的，它将应用于不覆盖它的所有子类。如果 readResolve 方法是受保护的或公共的，而子类没有覆盖它，反序列化子类实例将生成超类实例，这可能会导致 ClassCastException。

总之，在可能的情况下，使用枚举类型强制实例控制不变量。如果这是不可能的，并且你需要一个既可序列化又实例控制的类，那么你必须提供一个 readResolve 方法，并确保该类的所有实例字段都是基本类型，或使用 transient 修饰。

Item 90: Consider serialization proxies instead of serialized instances（考虑以序列化代理代替序列化实例）

正如在 Item-85 和 Item-86 中提到的贯穿本章的问题：实现 Serializable 接口的决定增加了出现 bug 和安全问题的可能性，因为它允许使用一种超语言机制来创建实例，而不是使用普通的构造函数。然而，有一种技术可以大大降低这些风险。这种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，它简洁地表示外围类实例的逻辑状态。这个嵌套类称为外围类的序列化代理。它应该有一个构造函数，其参数类型是外围类。这个构造函数只是从它的参数复制数据：它不需要做任何一致性检查或防御性复制。按照设计，序列化代理的默认序列化形式是外围类的完美序列化形式。外围类及其序列代理都必须声明实现 Serializable 接口。

例如，考虑 Item-50 中编写的不可变 Period 类，并在 Item-88 中使其可序列化。这是该类的序列化代理。Period 非常简单，它的序列化代理具有与类完全相同的字段：

```
// 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 方法添加到外围类中。通过序列化代理，这个方法可以被逐字地复制到任何类中：

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

该方法存在于外围类，导致序列化系统产生 SerializationProxy 实例，而不是外围类的实例。换句话说，writeReplace 方法在序列化之前将外围类的实例转换为其序列化代理。

有了这个 writeReplace 方法，序列化系统将永远不会生成外围类的序列化实例，但是攻击者可能会创建一个实例，试图违反类的不变性。为了保证这样的攻击会失败，只需将这个 readObject 方法添加到外围类中：

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream) throws InvalidObjectException
{
    throw new InvalidObjectException("Proxy required");
}
```

最后，在 SerializationProxy 类上提供一个 readResolve 方法，该方法返回外围类的逻辑等效实例。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外围类的实例。

这个 readResolve 方法仅使用其公共 API 创建了一个外围类的实例，这就是该模式的美妙之处。它在很大程度上消除了序列化的语言外特性，因为反序列化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。这使你不必单独确保反序列化的实例遵从类的不变性。如果类的静态工厂或构造函数建立了这些不变性，而它的实例方法维护它们，那么你就确保了这些不变性也将通过序列化来维护。

以下是上述 `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）类似，序列化代理方法阻止伪字节流攻击（第 354 页）和内部字段盗窃攻击（第 356 页）。与前两种方法不同，这种方法允许 Period 的字段为 final，这是 Period 类真正不可变所必需的（Item-17）。与前两种方法不同，这一种方法不需要太多的思考。你不必指出哪些字段可能会受到狡猾的序列化攻击的危害，也不必显式地执行有效性检查作为反序列化的一部分。

序列化代理模式还有另一种方式比 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初始序列化实例不同的类。你可能不认为这在实践中有用，但它确实有用。

考虑 EnumSet 的情况（Item-36）。该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，但是在当前 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。如果底层枚举类型有 64 个或更少的元素，则静态工厂返回一个 RegularEnumSet；否则，它们返回一个 JumboEnumSet。

现在考虑，如果序列化一个枚举集合，它的枚举类型有 60 个元素，然后给这个枚举类型再增加 5 个元素，之后反序列化这个枚举集合。当它被序列化的时候，返回 RegularEnumSet 实例，但最好是 JumboEnumSet 实例。事实上正是这样，因为 EnumSet 使用序列化代理模式。如果你好奇，这里是 EnumSet 的序列化代理。其实很简单：

```
// 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。它与客户端可扩展的类不兼容（Item-19）。而且，它也不能与对象图中包含循环的某些类兼容：如果你试图从对象的序列化代理的 readResolve 方法中调用对象上的方法，你将得到一个 ClassCastException，因为你还没有对象，只有对象的序列化代理。

最后，序列化代理模式所增强的功能和安全性并不是没有代价的。在我的机器上，通过序列化代理来序列化和反序列化 Period 实例的开销，比用保护性拷贝进行的开销增加了14%。

总之，当你发现必须在客户端不可扩展的类上编写 readObject 或 writeObject 方法时，请考虑序列化代理模式。要想稳健地将带有重要约束条件的对象序列化时，这种模式可能是最容易的方法。
