[第2章 创建和销毁对象 3](#_Toc456980839)

[第1条：考虑用静态工厂方法代替构造器 3](#_Toc456980840)

[第2条：遇到多个构造器参数时要考虑用构建器 3](#_Toc456980841)

[第3条：用私有构造器或者枚举类型强化Singleton属性 6](#_Toc456980842)

[第4条：通过私有构造器强化不可实例化的能力 7](#_Toc456980843)

[第5条：避免创建不必要的对象 8](#_Toc456980844)

[第6条：消除过期的对象引用 10](#_Toc456980845)

[第7条：避免使用终结方法 11](#_Toc456980846)

[第3章 对于所有对象都通用的方法 12](#_Toc456980847)

[第8条：覆盖equals时请遵守通用约定 12](#_Toc456980848)

[第9条：覆盖equals时总要覆盖hashCode 16](#_Toc456980849)

[第10条：始终要覆盖toString 18](#_Toc456980850)

[第11条：谨慎地覆盖clone 18](#_Toc456980851)

[第12条：考虑实现Comparable接口 21](#_Toc456980852)

[第4章 类和接口 23](#_Toc456980853)

[第13条：使类和成员的可访问性最小化 23](#_Toc456980854)

[第14条：在公有类中使用访问方法而非公有域 24](#_Toc456980855)

[第15条：使可变性最小化 24](#_Toc456980856)

[第16条：复合优先于继承 26](#_Toc456980857)

[第17条：要么为继承而设计，并提供文档说明，要么就禁止继承 30](#_Toc456980858)

[第18条：接口优于抽象类 30](#_Toc456980859)

[第19条：接口只用于定义类型 33](#_Toc456980860)

[第20条：类层次优于标签类 34](#_Toc456980861)

[第21条：用函数对象表示策略 34](#_Toc456980862)

[第22条：优先考虑静态成员类 36](#_Toc456980863)

[第5章 泛型 37](#_Toc456980864)

[第23条：请不要在新代码中使用原生态类型 37](#_Toc456980865)

[第24条：消除非受检警告 38](#_Toc456980866)

[第25条：列表优先于数组 39](#_Toc456980867)

[第26条：优先考虑泛型 39](#_Toc456980868)

[第27条：优先考虑泛型方法 40](#_Toc456980869)

[第28条：利用有限制通配符来提升API的灵活性 40](#_Toc456980870)

[第29条：优先考虑类型安全的异构容器 42](#_Toc456980871)

[第6章 枚举和注解 43](#_Toc456980872)

[第30条：用enum代替int常量 43](#_Toc456980873)

[第31条：用实例域代替序数 49](#_Toc456980874)

[第32条：用EnumSet代替位域 49](#_Toc456980875)

[第33条：用EnumMap代替序数索引 50](#_Toc456980876)

[第34条：用接口模拟可伸缩的枚举 52](#_Toc456980877)

[第35条：注解优先于命名模式 52](#_Toc456980878)

[第36条：坚持使用Override注解 53](#_Toc456980879)

[第37条：用标记接口定义类型 54](#_Toc456980880)

[第7章 方法 55](#_Toc456980881)

[第38条：检查参数的有效性 55](#_Toc456980882)

[第39条：必要时进行保护性拷贝 56](#_Toc456980883)

[第40条：谨慎设计方法签名 58](#_Toc456980884)

[第41条：慎用重载 59](#_Toc456980885)

[第42条：慎用可变参数 63](#_Toc456980886)

[第43条：返回零长度的数组或者集合，而不是null 64](#_Toc456980887)

[第44条：为所有导出的API元素编写文档注释 66](#_Toc456980888)

[第8章 通用程序设计 66](#_Toc456980889)

[第45条：将局部变量的作用域最小化 66](#_Toc456980890)

[第46条：for-each循环优先于传统的for循环 67](#_Toc456980891)

[第47条：了解和使用类库 68](#_Toc456980892)

[第48条：如果需要精确的答案，请避免使用float和double 68](#_Toc456980893)

[第49条：基本类型优先于装箱基本类型 69](#_Toc456980894)

[第51条：当心字符串连接的性能 70](#_Toc456980895)

[第52条：通过接口引用对象 70](#_Toc456980896)

[第53条：接口优先于反射机制 71](#_Toc456980897)

[第54条：谨慎地使用本地方法 72](#_Toc456980898)

[第55条：谨慎地进行优化 73](#_Toc456980899)

[第56条：遵守普遍接受的命名惯例 74](#_Toc456980900)

[第9章 异常 74](#_Toc456980901)

[第57条：只针对异常的情况才使用异常 74](#_Toc456980902)

[第58条：对可恢复的情况使用受检异常，对编程错误使用运行时异常 75](#_Toc456980903)

[第59条：避免不必要地使用受检的异常 76](#_Toc456980904)

[第60条：优先使用标准的异常 76](#_Toc456980905)

[第61条：抛出与抽象相对应的异常 77](#_Toc456980906)

[第62条：每个方法抛出的异常都要有文档 78](#_Toc456980907)

[第63条：在细节消息中包含能捕获失败的信息 78](#_Toc456980908)

[第64条：努力使失败保持原子性 79](#_Toc456980909)

[第65条：不要忽略异常 79](#_Toc456980910)

[第10章 并发 79](#_Toc456980911)

[第66条：同步访问共享的可变数据 79](#_Toc456980912)

[第67条：避免过度同步 83](#_Toc456980913)

[第68条：executor和task优先于线程 85](#_Toc456980914)

[第69条：并发工具优先于wait和notify 86](#_Toc456980915)

[第70条：线程安全性的文档化 90](#_Toc456980916)

[第71条：慎用延迟初始化 91](#_Toc456980917)

[第72条：不要依赖于线程调度器 93](#_Toc456980918)

[第73条：避免使用线程组 93](#_Toc456980919)

[第11章 序列化 93](#_Toc456980920)

[第74条：谨慎地实现Serializable接口 94](#_Toc456980921)

[第75条：考虑使用自定义的序列化形式 98](#_Toc456980922)

[第76条：保护性地编写readObject方法 101](#_Toc456980923)

[第77条：对于实例控制，枚举类型优先于readResolve 101](#_Toc456980924)

[第78条：考虑用序列化代理代替序列化实例 102](#_Toc456980925)

第2章 创建和销毁对象

第1条：考虑用静态工厂方法代替构造器

1. 一个类只能有一个带有指定签名的构造器。编程人员通常知道如何避开这一限制：通过提供两个构造器，它们的参数列表只在参数类型的顺序上有所不同。实际上这并不是个好主意。面对这样的API，用户永远也记不住该用哪个构造器，结果常常会调用错误的构造器。并且，读到使用了这些构造器的代码时，如果没有参考类的文档，往往不知所云。
2. 由于静态工厂方法有名称，所以它们不受上述的限制。当一个类需要多个带有相同签名的构造器时，就用静态工厂方法代替构造器，并且慎重地选择名称以便突出它们之间的区别。
3. 静态工厂方法能够为重复的调用返回相同的对象，这样有助于类总能严格控制在某个时刻哪些实例应该存在。这种类被称作实例受控的类（instance-controlled）。编写实例受控的类有几个原因。实例受控使得类可以确保它是一个Singleton或者是不可实例化的。
4. 静态工厂方法与构造器不同的第三大优势在于，它们可以返回原返回类型的任何子类型的对象。

第2条：遇到多个构造器参数时要考虑用构建器

1. 静态工厂和构造器有个共同的局限性：它们都不能很好地扩展到大量的可选参数。
2. 遇到许多构造器参数的时候，还有第二种代替方法，即JavaBeans模式，在这种模式下，调用一个无参构造器来创建对象，然后调用setter方法来设置每个必要的参数，以及每个相关的可选参数。
3. 遗憾的是，JavaBeans模式自身有着很严重的缺点。因为构造过程被分到了几个调用中，在构造过程中JavaBean可能处于不一致的状态。
4. 与此相关的另一点不足在于，JavaBeans模式阻止了把类做成不可变的可能，这就需要程序员付出额外的努力来确保它的线程安全。
5. 注意NutritionFacts是不可变的，所有的默认参数值都单独放在一个地方。builder的setter方法返回builder本身，以便可以把调用链接起来。下面就是客户端代码：

NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).calories(100).sodium(35).carbohydrate(27).build();

builder模式模拟了具名的可选参数，就像Ada和Python中的一样。

// 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 carbohydrate = 0;

private int sodium = 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 carbohydrate(int val) { carbohydrate = val; return this; }

public Builder sodium(int val) { sodium = 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;

}

}

第3条：用私有构造器或者枚举类型强化Singleton属性

//Singleton with public final field

public class Elvis {

public static final Elvis INSTANCE = new Elvis();

private Elvis() { ... }

public void leaveTheBuilding() { ... }

私有构造器仅被调用一次，用来实例化公有的静态final域Elvis.INSTANCE。由于缺少公有的或者受保护的构造器，所以保证了Elvis的全局唯一性：一旦Elvis类被实例化，只会存在一个Elvis实例，不多也不少。

//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() { ... }

}

公有域方法的主要好处在于，组成类的成员的声明很清楚地表明了这个类是一个Singleton：公有的静态域是final的，所以该域将总是包含相同的对象引用。公有域方法在性能上不再有任何优势：现代的JVM实现几乎都能够将静态工厂方法的调用内联化。

为了使利用这其中一种方法实现的Singleton类变成是可序列化的（Serializable），仅仅在声明中加上“implements Serializable”是不够的。为了维护并保证Singleton，必须声明所有实例域都是瞬时（transient）的，并提供一个readResolve方法。

从Java 1.5发行版本起，实现Singleton还有第三种方法。只需编写一个包含单个元素的枚举类型：

//Enum singleton - the preferred approach

public enum Elvis {

INSTANCE;

public void leaveTheBuilding() { ... }

}

这种方法在功能上与公有域方法相近，但是它更加简洁，无偿提供了序列化机制，绝对防止多次实例化，即使是在面对复杂的序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现Singleton的最佳方法。

第4条：通过私有构造器强化不可实例化的能力

1. 有时候，你可能需要编写只包含静态方法和静态域的类。这些类的名声很不好，因为有些人在面向对象的语言中滥用这样的类来编写过程化的程序。尽管如此，它们也确实有它们特有的用处。我们可以利用这种类，以java.lang.Math或者java.util.Arrays的方式，把基本类型的值或者数组类型上的相关方法组织起来。我们也可以通过java.util.Collections的方式，把实现特定接口的对象上的静态方法（包括工厂方法）组织起来。最后，还可以利用这种类把final类上的方法组织起来，以取代扩展该类的做法。
2. 这样的**工具类（utility class）**不希望被实例化，实例对它没有任何意义。然而，在缺少显式构造器的情况下，编译器会自动提供一个公有的、无参的缺省构造器（default constructor）。对于用户而言，这个构造器与其他的构造器没有任何区别。
3. 企图通过将类做成抽象类来强制该类不可被实例化，这是行不通的。该类可以被子类化，并且该子类也可以被实例化。这样做甚至会误导用户，以为这种类是专门为了继承而设计的。

// Noninstantiable utility class

public class UtilityClass {

//Suppress default constructor for noninstantiability

private UtilityClass() {

throw new AssertionError();

}

... //Remainder omitted

}

第5条：避免创建不必要的对象

1. 如果对象是不可变的（immutable），它就始终可以被重用。
2. 传递给String构造器的参数（”stringette”）本身就是一个String实例，功能方面等同于构造器创建的所有对象。
3. 而且，它可以保证，对于所有在同一台虚拟机中运行的代码，只要它们包含相同的字符串字面常量，该对象就会被重用。
4. 对于同时提供了静态工厂方法和构造器的不可变类，通常可以使用静态工厂方法而不是构造器，以避免创建不必要的对象。例如，静态工厂方法Boolean.valueOf(String)几乎总是优先于构造器Boolean(String)。
5. 下面的版本用一个静态的初始化器（initializer），避免了这种效率低下的情况：

class Person {

private final Date birthDate;

// Other fields, methods, and constructor omitted

private static final Date BOOM\_START;

private static final Date BOOM\_END;

static {

Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));

gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);

BOOM\_START = gmtCal.getTime();

gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);

BOOM\_END = gmtCal.getTime();

}

public boolean isBabyBoomer() {

return birthDate.compareTo(BOOM\_START) >= 0 &&

birthDate.compareTo(BOOM\_END) < 0;

}

}

改进后的Person类只在初始化的时候创建Calendar、TimeZone和Date实例一次，而不是在每次调用isBabyBoomer的时候都创建这些实例。

把boomStart和boomEnd从局部变量改为final静态域，这些日期显然是被作为常量对待，从而使得代码更易于理解。

如果改进后的Person类被初始化了，它的isBabyBoomer方法却永远不会被调用，那就没有必要初始化BOOM\_START和BOOM\_END域。通过延迟初始化（lazily initializing），即把对这些域的初始化延迟到isBabyBoomer方法第一次被调用的时候进行，则有可能消除这些不必要的初始化工作，但是不建议这样做。正如延迟初始化中常见的情况一样，这样做会使方法的实现更加复杂，从而无法将性能显著提高到超过已经达到的水平。

考虑适配器（adpater）的情形，有时也叫做视图（view）。适配器是指这样一个对象：它把功能委托给一个后备对象（backing object），从而为后备对象提供一个可以替代的接口。由于适配器除了后备对象之外，没有其他的状态信息，所以针对某个给定对象的特定适配器而言，它不需要创建多个适配器实例。

例如，Map接口的keySet方法返回该Map对象的Set视图，其中包含该Map中所有的键（key）。粗看起来，好像每次调用keySet都应该创建一个新的Set实例，但是，对于一个给定的Map对象，实际上每次调用keySet都返回同样的Set实例。虽然被返回的Set实例一般是可改变的，但是所有返回的对象在功能上是等同的：当其中一个返回对象发生变化的时候，所有其他的返回对象也要发生变化，因为它们是由同一个Map实例支撑的。

变量sum被声明成Long而不是long，意味着程序构造了大约231个多余的Long实例（大约每次往Long sum中增加long时构造一个实例）。结论很明显：要优先使用基本类型而不是装箱基本类型，要当心无意识的自动装箱。

真正正确使用对象池的典型对象示例就是数据库连接池。建立数据库连接的代价是非常昂贵的，因此重用这些对象非常有意义。而且，数据库的许可可能限制你只能使用一定数量的连接。但是，一般而言，维护自己的对象池必定会把代码弄得很乱，同时增加内存占用（footprint），并且还会损害性能。现代的JVM实现具有高度优化的垃圾回收器，其性能很容易就会超过轻量级对象池的性能。

第6条：消除过期的对象引用

1. 如果一个栈先是增长，然后再收缩，那么，从栈中弹出来的对象将不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。这是因为，栈内部维护着对这些对象的过期引用（obsolete reference）。所谓的过期引用，是指永远也不会再被解除的引用。在本例中，凡是在elements数组的”活动部分（active portion）”之外的任何引用都是过期的。活动部分是指elements中下标小于size的那些元素。
2. 在支持垃圾回收的语言中，内存泄漏是很隐蔽的（称这类内存泄漏为“无意识的对象保持（unintentional object retention）”更为恰当。
3. 清空对象引用应该是一种例外，而不是一种规范行为。
4. 一般而言，只要类是自己管理内存，程序员就应该警惕内存泄漏问题。
5. 如果你正好要实现这样的缓存：只要在缓存之外存在对某个项的键的引用，该项就有意义，那么就可以用WeakHashMap代表缓存；当缓存中的项过期之后，它们就会自动被删除。记住只有当所要的缓存项的生命周期是由该键的外部引用而不是由值决定时，WeakHashMap才有用处。
6. 内存泄漏的第三个常见来源是监听器和其他回调。如果你实现了一个API，客户端在这个API中注册回调，却没有显式地取消注册，那么除非你采取某些动作，否则它们就会积聚。确保回调立即被当作垃圾回收的最佳方法是只保存它们的弱引用（weak reference），例如，只将它们保存成WeakHashMap中的键。

第7条：避免使用终结方法

1. 终结方法（finalizer）通常是不可预测的，也是很危险的，一般情况下是不必要的。
2. C++的析构器也可以被用来回收其他的非内存资源。而在Java中，一般用try-finally块来完成类似的工作。
3. 这意味着，注重时间（time-critical）的任务不应该由终结方法来完成。例如，用终结方法来关闭已经打开的文件，这是严重错误，因为打开文件的描述符是一种很有限的资源。
4. 一位同事最近在调试一个长期运行的GUI应用程序的时候，该应用程序莫名其妙地出现OutOfMemoryError错误而死掉。分析表明，该应用程序死掉的时候，其终结方法队列中有数千个图形对象正在等待被终结和回收。
5. 不要被System.gc和System.runFinalization这两个方法所诱惑，它们确实增加了终结方法被执行的机会，但是它们并不保证终结方法一定会被执行。唯一声称保证终结方法被执行的方法是System.runFinalizersOnExit，以及它臭名昭著的孪生兄弟Runtime.runFinalizersOnExit。这两个方法都有致命的缺陷，已经被废弃了[ThreadStop]。
6. 正常情况下，未被捕获的异常将会使线程终止，并打印出轨迹栈（Stack Trace），但是，如果异常发生在终结方法之中，则不会如此，甚至连警告都不会打印出来。
7. 那么，如果类的对象中封装的资源（例如文件或者线程）确实需要终止，应该怎么做才能不用编写终结方法呢？只需提供一个显式的终止方法，并要求该类的客户端在每个实例不再有用的时候调用这个方法。
8. 显示终止方法的典型例子是InputStream、OutputStream和java.sql.Connection上的close方法。另一个例子是java.util.Timer上的cancel方法，它执行必要的状态改变，使得与Timer实例相关联的该线程温和地终止自己。java.awt中的例子还包括Graphics.dispose和Window.dispose。这些方法通常由于性能不好而不被人们关注。一个相关的方法是Image.flush，它会释放所有与Image实例相关联的资源，但是该实例仍然处于可用的状态，如果有必要的话，会重新分配资源。
9. 显式的终止方法通常与try-finally结构结合起来使用，以确保及时终止。在finally子句内部调用显式的终止方法，可以保证即使在使用对象的时候有异常抛出，该终止方法也会执行。
10. 但是如果终结方法发现资源还未被终止，则应该在日志中记录一条警告，因为这表示客户端代码中的一个Bug，应该得到修复。
11. 显式终止方法模式的示例中所示的四个类（FileInputStream、FileOutputStream、Timer和Connection），都具有终结方法，当它们的终止方法未能被调用的情况下，这些终结方法充当了安全网。
12. 本地对等体是一个本地对象（native object），普通对象通过本地方法（native method）委托给一个本地对象。因为本地对等体不是一个普通对象，所以垃圾回收器不会知道它，当它的Java对等体被回收的时候，它不会被回收。在本地对等体并不拥有关键资源的前提下，终结方法正式执行这项任务最合适的工具。

第3章 对于所有对象都通用的方法

1. 尽管Object是一个具体类，但是设计它主要是为了扩展。它所有的非final方法（equals、hashCode、toString、clone和finalize）都有明确的通用约定（general contract），因为它们被设计成是要被覆盖（override）的。任何一个类，它在覆盖这些方法的时候，都有责任遵守这些通用约定；如果不能做到这一点，其他依赖于这些约定的类（例如HashMap和HashSet）就无法结合该类一起正常运作。
2. 而Comparable.compareTo虽然不是Object方法，但是本章也对它进行讨论，因为它具有类似的特征。
3. Object的equals方法：

public boolean equals(Object obj) {

return (this == obj);

}

第8条：覆盖equals时请遵守通用约定

1. 类的每个实例本质上都是唯一的。对于代表活动实体而不是值（value）的类来说确实如此，例如Thread。Object提供的equals实现对于这些类来说正是正确的行为。
2. 超类已经覆盖了equals，从超类继承过来的行为对于子类也是合适的。例如，**大多数的Set实现都从AbstractSet继承equals实现，List实现从AbstractList继承equals实现，Map实现从AbstractMap继承equals实现。**

java.util.AbstractSet的equals方法：

public boolean equals(Object o) {

if (o==this)

return true;

if(!(o instanceof Set))

return false;

Collection c = (Collection) o;

if (c.size() != size())

return false;

try {

return containsAll(c);

} catch (ClassCastException unused) {

return false;

} catch (NullPointerException unused) {

return false;

}

}

java.util.AbstractMap的equals方法：

public boolean equals(Object o) {

if (o==this)

return true;

if (!(o instanceof Map))

return false;

Map<K,V> m = (Map<K,V>) o;

if (m.size()!=size())

return false;

try {

Iterator<Entry<K,V>> i = entrySet().iterator();

while(i.hasNext()) {

Entry<K,V> e = i.next();

K key = e.getKey();

V value = e.getValue();

if (value==null) {

if(!(m.get(key)==null && m.containsKey(key)))

return false;

} else {

if(!value.equals(m.get(key)))

return false;

}

}

} catch (ClassCastException unused) {

return false;

} catch (NullPointerException unused) {

return false;

}

return true;

}

类是私有的或是包级私有的，可以确定它的equals方法永远不会被调用。在这种情况下，无疑是应该覆盖equals方法的，以防它被意外调用：

@override public boolean equals(Object o) {

throw new AssertionError(); //Method is never called

}

这通常属于“值类（value class）”的情形。值类仅仅是一个表示值的类，例如Integer或者Date。程序员在利用equals方法来比较值对象的引用时，希望知道它们在逻辑上是否相等，而不是想了解它们是否指向同一个对象。

有一种“值类”不需要覆盖equals方法，即用实例受控确保“每个值至多只存在一个对象”的类。枚举类型就属于这种类。对于这样的类而言，逻辑相同与对象等同是一回事，因此Object的equals方法等同于逻辑意义上的equals方法。

对于既不是float也不是double类型的基本类型域，可以使用==操作符进行比较；对于对象引用域，可以递归地调用equals方法；对于float域，可以使用Float.compare方法；对于double域，则使用Double.compare。对float和double域进行特殊的处理是有必要的，因为存在着Float.NaN、-0.0f以及类似的double常量；详细信息请参考Float.equals的文档。对于数组域，则要把以上这些指导原则应用到每个元素上。如果数组域中的每个元素都很重要，就可以使用发行版本1.5中新增的其中一个Arrays.equals方法。

有些对象引用域包含null可能是合法的，所以，为了避免可能导致NullPointerException异常，则使用下面的习惯用法来比较这样的域：

(field==null ? o.field==null : field.equals(o.field))

域的比较顺序可能会影响到equals方法的性能。为了获得最佳的性能，应该最先比较最有可能不一致的域，或者是开销最低的域，最理想的情况是两个条件同时满足的域。你不应该去比较那些不属于对象逻辑状态的域，例如用于同步操作的Lock域。

覆盖equals时总要覆盖hashCode。

把任何一种别名形式考虑到等价的范围内，往往不会是个好主意。例如，File类不应该试图把指向同一个文件的符号链接（symbolic link）当作相等的对象来看。所幸File类没有这样做。

public boolean equals(MyClass o) {

...

}

问题在于，这个方法并没有覆盖Object.equals，因为它的参数应该是Object类型，相反，它重载（overload）了Object.equals。在原有equals方法的基础上，再提供一个“强类型（strongly typed）”的equals方法，只要这两个方法返回同样的结果，那么这就是可以接受的。在某些特定的情况下，它也许能够稍微改善性能，但是与增加的复杂性相比，这种做法是不值得的。@override注解的用法一致，就如本条目中所示，可以防止犯这种错误。这个equals方法不能编译，错误消息会告诉你到底哪里出了问题：

@Override public boolean equals(MyClass o) {

...

}

第9条：覆盖equals时总要覆盖hashCode

1. 在应用程序执行期间，只要对象的equals方法的比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。
2. 如果两个对象根据equals(Object)方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode方法都必须产生同样的整数结果。
3. 如果两个对象根据equals(Object)方法比较是不相等的，那么调用这两个对象中任意一个对象的hashCode方法，则不一定要产生不同的整数结果。但是程序员应该知道，给不相等的对象产生截然不同的整数结果，有可能提高散列表（hash table）的性能。
4. 由于PhoneNumber类没有覆盖hashCode方法，从而导致两个相等的实例具有不相等的散列码，违反了hashCode的约定。因此，put方法把电话号码对象存放在一个散列桶（hash bucket）中，get方法却在另一个散列桶中查找这个电话号码。即使这两个实例正好被放到同一个散列桶中，get方法也必定会返回null，因为HashMap有一项优化，可以将与每个项相关联的散列码缓存起来，如果散列码不匹配，也不必检验对象的等同性。
   1. 把某个非零的常量值，比如说17，保存在一个名为result的int类型的变量中。
   2. 对于对象中每个关键域f（指equals方法中涉及的每个域），完成以下步骤：   
      a. 为该域计算int类型的散列码c：   
      i. 如果该域是boolean类型，则计算(f ? 1:0)。   
      ii. 如果该域是byte、char、short或者int类型，则计算(int)f。   
      iii. 如果该域是long类型，则计算(int)(f ^ (f>>>32))。   
      iv. 如果该域是float类型，则计算Float.floatToIntBits(f)。   
      v. 如果该域是double类型，则计算Double.doubleToLongBits(f)，然后按照步骤2.a.iii，为得到的long类型值计算散列值。   
      vi. 如果该域是一个对象引用，并且该类的equals方法通过递归地调用equals的方式来比较这个域，则同样为这个域递归地调用hashCode。如果需要更复杂的比较，则为这个域计算一个“范式（canonical representation）”，然后针对这个范式调用hashCode。如果这个域的值为null，则返回0（或者其他某个常数，但通常是0）。   
      vii. 如果该域是一个数组，则要把每一个元素当做单独的域来处理。也就是说，递归地应用上述规则，对每个重要的元素计算一个散列码，然后根据步骤b.2中的做法把这些散列值组合起来。如果数组域中的每个元素都很重要，可以利用发行版本1.5中增加的其中一个Arrays.hashCode方法。   
      b. 按照下面的公式，把步骤2.a中计算得到的散列码c合并到result中：   
      result=31×result+c;   
      返回result。
5. 必须排除equals比较计算中没有用到的任何域，否则很有可能违反hashCode约定的第二条。
6. 步骤2.b中的乘法部分使得散列值依赖于域的顺序，如果一个类包含多个相似的域，这样的乘法运算就会产生一个更好的散列函数。例如，如果String散列函数省略了这个乘法部分，那么只是字母顺序不同的所有字符串都会有相同的散列码。之所以选择31，是因为它是一个奇素数。如果乘数是偶数，并且乘法溢出的话，信息就会丢失，因为与2相乘等价于移位运算。31有个很好的特性，即用移位和减法来代替乘法，可以得到更好的性能：31 \* i == (i << 5) - i。现代的VM可以自动完成这种优化。
7. **如果一个类是不可变的，并且计算散列码的开销也比较大，就应该考虑把散列码缓存在对象内部，而不是每次请求的时候都重新计算散列码。**如果你觉得这种类型的大多数对象会被用作散列键（hash keys），就应该在创建实例的时候计算散列码。否则，可以选择“延迟初始化”散列码，一直到hashCode被第一次调用的时候才初始化。
8. 不要试图从散列码计算中排除掉一个对象的关键部分来提高性能。虽然这样得到的散列函数运行起来可能更快，但是它的效果不见得会好，可能会导致散列表慢到根本无法使用。特别是在实践中，散列函数可能面临大量的实例，在你选择忽略的区域之中，这些实例仍然区别非常大。如果是这样，散列函数就会把所有这些实例映射到极少数的散列码上，基于散列的集合将会显示出平方级的性能指标。这不仅仅是个理论问题。在Java 1.2发行版本之前实现的String散列函数至多只检查16个字符，从第一个字符开始，在整个字符串中均匀选取。对于像URL这种层次状名字的大型集合，该散列函数正好表现出了这里所提到的病态行为。
9. java.lang.String的hashCode方法：

public int hashCode() {

int h = hash; //hash: cache the hash code for the string

if (h==0) { //h==0说明是第一次计算哈希值，hash域还没有缓存

int off = offset; // the offset is the first index of the storage that is used.

char val[] = value; // the value is used for character storage.

int len = count;

for (int i=0;i<len;i++) {

h = 31\*h + val[off++];

}

hash = h;

}

return h;

第10条：始终要覆盖toString

1. 在实现toString的时候，必须要做出一个很重要的决定：是否在文档中指定返回值的格式。
2. 这种表示法可以用于输入和输出，以及用在永久的适合于人类阅读的数据对象中，例如XML文档。
3. 无论你是否决定指定格式，都应该在文档中明确地表明你的意图。如果你要指定格式，则应该严格地这样去做。

第11条：谨慎地覆盖clone

1. 其主要的缺陷在于，它缺少一个clone方法，Object的clone方法是受保护的。如果不借助于反射（reflection），就不能仅仅因为一个对象实现了Cloneable，就可以调用clone方法。即使是反射调用也可能会失败，因为不能保证该对象一定具有可访问的clone方法。
2. 既然Cloneable并没有包含任何方法，那么它到底有什么作用呢？它决定了Object中受保护的clone方法实现的行为：如果一个类实现了Cloneable，Object的clone方法就返回该对象的逐域拷贝，否则就会抛出CloneNotSupportedException异常。这是接口的一种极端非典型的用法，也不值得效仿。通常情况下，实现接口是为了表明类可以为它的客户做些什么。然而，对于Cloneable接口，它改变了超类中受保护的方法的行为。
3. 如果它的clone方法仅仅返回super.clone()，这样得到的Stack实例，在其size域中具有正确的值，但是它的elements域将引用与原始Stack实例相同的数组。
4. 为了使Stack类中的clone方法正常地工作，它必须要拷贝栈的内部信息。最容易的做法是，在elements数组中递归地调用clone：

@Override public Stack clone() {

try {

Stack result = (Stack) super.clone();

result.elements = elements.clone();

return result;

} catch (CloneNotSupportedException e) {

throw new AssertionError();

}

}

虽然被克隆对象有它自己的散列桶数组，但是，这个数组引用的链表与原始对象是一样的，从而很容易引起克隆对象和原始对象中不确定的行为。为了修正这个问题，必须单独地拷贝并组成每个桶的链表。下面是一种常见的做法：

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;

}

// 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;

}

}

@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

}

如果专门为了继承而设计的类覆盖了clone方法，覆盖版本的clone方法就应该模拟Object.clone的行为：它应该被声明为protected、抛出CloneNotSupportedException异常，并且该类不应该实现Cloneable接口。这样做可以使子类具有实现或不实现Cloneable接口的自由，就仿佛它们直接扩展了Object一样。

简而言之，所有实现了Cloneable接口的类都应该用一个公有的方法覆盖clone。此公有方法首先调用super.clone，然后修正任何需要修正的域。一般情况下，这意味着要拷贝任何包含内部“深层结构”的可变对象，并用指向新对象的引用代替原来指向这些对象的引用。

拷贝构造器的做法，以及静态工厂方法的变形，都比Cloneable/clone方法具有更多的优势：它们不依赖于某一种很有风险的、语言之外的对象创建机制；它们不要求遵守尚未制定好文档的规范；它们不会与final域的正常使用发生冲突；它们不会抛出不必要的受检查异常（checked exception）；它们不需要进行类型转换。虽然你不可能把拷贝构造器或者静态工厂放到接口中，但是由于Cloneable接口缺少一个公有的clone方法，所以它也没有提供一个接口该有的功能。因此，使用拷贝构造器或者拷贝工厂来代替clone方法时，并没有放弃接口的功能特性。

第12条：考虑实现Comparable接口

1. 一旦类实现了Comparable接口，它就可以跟许多泛型算法（generic algorithm）以及依赖于该接口的集合实现（collection implementation）进行协作。你付出很小的努力就可以获得非常强大的功能。事实上，Java平台类库中的所有值类（value classes）都实现了Comparable接口。
2. 强烈建议(x.compareTo(y) == 0) == (x.equals(y))，但这并非绝对必要。一般来说，任何实现了Comparable接口的类，若违反了这个条件，都应该明确予以说明。推荐使用这样的说法：“注意：该类具有内在的排序功能，但是与equals不一致。”
3. 与equals不同的是，在跨越不同类的时候，compareTo可以不做比较：如果两个被比较的对象引用不同类的对象，compareTo可以抛出ClassCastException异常。通常，这正是compareTo在这种情况下应该做的事情，如果类设置了正确的参数，这也正是它所要做的事情。
4. 就好像违法了hashCode约定的类会破坏其他依赖于散列做法的类一样，违反compareTo约定的类也会破坏其他依赖于比较关系的类。依赖于比较关系的类包括有序集合类TreeSet和TreeMap，以及工具类Collections和Arrays，它们内部包含有搜索和排序算法。
5. 因此，下面的告诫也同样适用：无法在用新的值组件扩展可实例化的类时，同时保持compareTo约定，除非愿意放弃面向对象的抽象优势。针对equals的权宜之计也同样适用于compareTo方法。如果你想为一个实现了Comparable接口的类增加值组件，请不要扩展这个类；而是要编写一个不相关的类，其中包含第一个类的一个实例。然后提供一个“视图（view）”方法返回这个实例。这样既可以让你自由地在第二个类上实现compareTo方法，同时也允许它的客户端在必要的时候，把第二个类的实例视同第一个类的实例。
6. 例如，考虑BigDecimal类，它的compareTo方法与equals不一致。如果你创建了一个HashSet实例，并且添加new BigDecimal (“1.0”)和new BigDecimal (“1.00”)，这个集合就将包含两个元素，因为新增到集合中的两个BigDecimal实例，通过equals方法来比较时是不相等的。然而，如果你使用TreeSet而不是HashSet来执行同样的过程，集合中将只包含一个元素，因为这两个BigDecimal实例在通过compareTo方法进行比较时是相等的。
7. 因为Comparable接口是参数化的，而且comparable方法是静态的类型（**意思是compareTo方法的参数类型是一定的，就是实现Comparable接口的时候指定的参数类型。而equals方法的参数类型是Object，所以要进行额外的类型检查，instanceof。**），因此不必进行类型检查，也不必对它的参数进行类型转换。如果参数的类型不合适，这个调用甚至无法编译。
8. 比较整数类型基本类型的域，可以使用关系操作符<和>。例如，浮点域用Double.compare或者Float.compare，而不用关系操作符，当应用到浮点值时，它们没有遵守compareTo的通用约定。对于数组域，则要把这些指导原则应用到每个元素上。
9. 如果一个类有多个关键域，那么，按什么样的顺序来比较这些域是非常关键的。你必须从最关键的域开始，逐步进行到所有的重要域。如果某个域的比较产生了非零的结果（零代表相等），则整个比较操作结束，并返回该结果。
10. 这项技巧有时不能正常工作的原因在于，一个有符号的32位的整数还没有大到足以表达任意两个32位整数的差。如果i是一个很大的正整数（int类型），而j是一个很大的负整数（int类型），那么(i-j)将会溢出，并返回一个负值。

第4章 类和接口

第13条：使类和成员的可访问性最小化

1. 信息隐藏之所以非常重要有许多原因，其中大多数理由都源于这样一个事实：它可以有效地解除组成系统的各模块之间的耦合关系，使得这些模块可以独立地开发、测试、优化、使用、理解和修改。
2. 对于顶层的（非嵌套的）类和接口，只有两种可能的访问级别：包级私有的（package-private）和公有的（public）。如果类或者接口能够被做成包级私有的，它就应该被做成包级私有。通过把类或者接口做成包级私有，它实际上成了这个包的实现的一部分，而不是该包导出的API的一部分，在以后的发行版本中，可以对它进行修改、替换，或者删除，而无需担心会影响到现有的客户端程序。
3. 如果一个包级私有的顶层类（或者接口）只是在某一个类的内部被用到，就应该考虑使它成为唯一使用它的那个类的私有嵌套类。然而，降低不必要公有类的可访问性，比降低包级私有的顶层类的更重要得多：因为公有类是包的API的一部分，而包级私有的顶层类则已经是这个包的实现的一部分。
4. 包级私有的（package-private）——声明该成员的包内部的任何类都可以访问这个成员。从技术上讲，它被称为“缺省（default）访问级别”，如果没有为成员指定访问修饰符，就采用这个访问级别。
5. 其实，只有当同一个包内的另一个类真正需要访问一个成员的时候，你才应该删除private修饰符，使该成员变成包级私有的。如果你发现自己经常要做这样的事情，就应该重新检查你的系统设计，看看是否另一种分解方案所得到的类，与其他类之间的耦合度会更小。然而，如果这个类实现了Serializable接口，这些域就有可能被“泄漏（leak）”到导出的API中。
6. 受保护的成员是类的导出的API的一部分，必须永远得到支持。导出的类的受保护成员也代表了该类对于某个实现细节的公开承诺。受保护的成员应该尽量少用。
7. 如果方法覆盖了超类中的一个方法，子类中的访问级别就不允许低于超类中的访问级别。这样可以确保任何可使用超类的实例的地方也都可以使用子类的实例。
8. 为了测试而将一个公有类的私有成员变成包级私有的，这还可以接受，但是要将访问级别提高到超过它，这就无法接受了。换句话说，不能为了测试，而将类、接口或者成员变成包的导出的API的一部分。幸运的是，也没有必要这么做，因为可以让测试作为被测试的包的一部分来运行，从而能够访问它的包级私有的元素。
9. 实例域决不能是公有的。如果域是非final的，或者是一个指向可变对象的final引用，那么一旦使这个域成为公有的，就放弃了对存储在这个域中的值进行限制的能力；这意味着，你也放弃了强制这个域不可变的能力。因此，包含公有可变域的类并不是线程安全的。即使域是final的，并且引用不可变的对象，当把这个域变成公有的时候，也就放弃了“切换到一种新的内部数据表示法”的灵活性。
10. 假设常量构成了类提供的整个抽象中的一部分，可以通过公有的静态final域来暴露这些常量。按惯例，这种域的名称由大写字母组成，单词之间用下划线隔开。很重要的一点是，这些域要么包含基本类型的值，要么包含指向不可变对象的引用。
11. 注意，长度非零的数组总是可变的，所以，类具有公有的静态final数组域，或者返回这种域的访问方法，这几乎总是错误的。
12. 要注意，许多IDE会产生返回指向私有数组域的引用的访问方法，这样就会产生这个问题。修正这个问题有两种方法。可以使公有数组变成私有的，并增加一个公有的不可变列表：

private static final Thing[] PRIVATE\_VALUES = { ... };

public static final List<Thing> VALUES =

Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));

第14条：在公有类中使用访问方法而非公有域

1. Java平台类库中有几个类违反了“公有类不应该直接暴露数据域”的告诫。显著的例子包括java.awt包中的Point和Dimension类。它们是不值得效仿的例子，相反，这些类应该被当作反面警告示例。正如第55条中所讲述的，决定暴露Dimension类的内部数据造成了严重的性能问题，而且这个问题至今依然存在。
2. 总之，公有类永远都不应该暴露可变的域。虽然还是有问题，但是让公有类暴露不可变的域其危害比较小。但是，有时候会需要用包级私有的或者私有的嵌套类来暴露域，无论这个类是可变的还是不可变的。

第15条：使可变性最小化

1. Java平台类库中包含许多不可变的类，其中有String、基本类型的包装类、BigInteger和BigDecimal。
2. 保证类不会被扩展。这样可以防止粗心或者恶意的子类假装对象的状态已经改变，而破坏该类的不可变行为。为了防止子类化，一般做法是使这个类成为final的，但是后面我们还会讨论到其他的做法。
3. 确保对于任何可变组件的互斥访问。如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。并且，永远不要用客户端提供的对象引用来初始化这样的域，也不要从任何访问方法（accessor）中返回该对象的引用。在构造器、访问方法和readObject方法中请使用保护性拷贝（defensive copy）技术。
4. 注意这些算术运算是如何创建并返回新的Complex实例，而不是修改这个实例。大多数重要的不可变类都使用了这种模式。它被称为函数的（functional）做法，因为这些方法返回了一个函数的结果，这些函数对操作数进行运算但并不修改它。与之相对应的更常见的是过程的（procedural）或者命令式的（imperative）做法，使用这些方式时，将一个过程作用在它们的操作数上，会导致它的状态发生改变。
5. “不可变对象可以被自由地共享”导致的结果是，永远也不需要进行保护性拷贝。实际上，你根本无需做任何拷贝，因为这些拷贝始终等于原始的对象。因此，你不需要，也不应该为不可变的类提供clone方法或者拷贝构造器（copy constructor）。这一点在Java平台的早期并不好理解，所以String类仍然具有拷贝构造器，但是应该尽量少用它。
6. 如果能够精确地预测出客户端将要在不可变的类上执行哪些复杂的多阶段操作，这种包级私有的可变配套类的方法就可以工作得很好。如果无法预测，最好的办法是提供一个公有的可变配套类。在Java平台类库中，这种方法的主要例子是String类，它的可变配套类是StringBuilder（和基本上已经废弃的StringBuffer）。可以这样认为，在特定的环境下，相对于BigInteger而言，BitSet同样扮演了可变配套类的角色。
7. 例如，假设你希望提供一种“基于极坐标创建复数”的方式。如果使用构造器来实现这样的功能，可能会使得这个类很零乱，因为这样的构造器与已用的构造器Complex(double, double)具有相同的签名。通过静态工厂，这很容易做到。只需添加第二个静态工厂，并且工厂的名字清楚地表明了它的功能即可：

public static Complex valueOfPolar(double r, double theta) {

return new Complex(r \* Math.cos(theta), r \* Math.sin(theta));

}

当BigInteger和BigDecimal刚被编写出来的时候，对于“不可变的类必须为final的”还没有得到广泛地理解，所以它们的所有方法都有可能会被覆盖。遗憾的是，为了保持向后兼容，这个问题一直无法得以修正。如果你编写一个类，它的安全性依赖于（来自不可信客户端的）BigInteger或者BigDecimal参数的不可变性，就必须进行检查，以确定这个参数是否为“真正的”的BigInteger或者BigDecimal，而不是不可信任子类的实例。如果是后者的话，就必须在假设它可能是可变的前提下对它进行保护性拷贝：

public static BigInteger safeInstance(BigInteger val) {

if (val.getClass() != BigInteger.class)

return new BigInteger(val.toByteArray());

return val;

}

总之，坚决不要为每个get方法编写一个相应的set方法。除非有很好的理由要让类称为可变的类，否则就应该是不可变的。你应该总是使一些小的值对象，比如PhoneNumber和Complex，成为不可变的（在Java平台类库中，有几个类如java.util.Date和java.awt.Point，它们本应该是不可变的，但实际上却不是）。你也应该认真考虑把一些较大的值对象做成不可变的，例如String和BigInteger。只有当你确认有必要实现令人满意的性能时，才应该为不可变的类提供公有的可变配套类。

可以通过TimerTask类来说明这些原则。它是可变的，但是它的状态空间被有意地设计得非常小。你可以创建一个实例，对它进行调度使它执行起来，也可以随意地取消它。一旦一个定时器任务（timer task）已经完成，或者已经被取消，就不可能再对它重新调度。

第16条：复合优先于继承

1. 在包的内部使用继承是非常安全的，在那里，子类和超类的实现都处在同一个程序员的控制之下。对于专门为了继承而设计、并且具有很好的文档说明的类来说，使用继承也是非常安全的。然而，对普通的具体类（concrete class）进行跨越包边界的继承，则是非常危险的。
2. 我们只要去掉被覆盖的addAll方法，就可以“修正”这个子类。虽然这样得到的类可以正常工作，但是，它的功能正确性则需要依赖于这样的事实：HashSet的addAll方法是在它的add方法上实现的。这种“自用性（self-use）”是实现细节，不是承诺，不能保证在Java平台的所有实现中都保持不变，不能保证随着发行版本的不同而不发生变化。因此，这样得到的InstrumentedHashSet类将是非常脆弱的。
3. 上面这两个问题都来源于覆盖（overriding）动作。如果在扩展一个类的时候，仅仅是增加新的方法，而不是覆盖现有的方法，你可能会认为这是安全的。虽然这种扩展方式比较安全一些，但是也并非完全没有风险。如果超类在后续的发行版本中获得了一个新的方法，并且不幸的是，你给子类提供了一个签名相同但返回类型不同的方法，那么这样的子类将无法通过编译。如果给子类提供的方法带有与新的超类方法完全相同的签名和返回类型，实际上就覆盖了超类中的方法，因此又回到上述的两个问题上去了。
4. 不用扩展现有的类，而是在新的类中增加一个私有域，它引用现有类的一个实例。这种设计被称作“复合（composition）”，因为现有的类变成了新类的一个组件。新类中的每个实例方法都可以调用被包含的现有类实例中对应的方法，并返回它的结果。这被称为转发（forwarding），新类中的方法被称为转发方法（forwarding method）。这样得到的类将会非常稳固，它不依赖于现有类的实现细节。即使现有的类添加了新的方法，也不会影响新的类。（**因为新类中的实例方法没有转发这个新的方法的方法。**）

// 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(); }

}

Set接口的存在使得InstrumentedSet类的设计成为可能，因为Set接口保存了HashSet类的功能特性。从本质上讲，这个类把一个Set转变成了另一个Set，同时增加了计数的功能。前面提到的基于继承的方法只适用于单个具体的类，并且**对于超类中所支持的每个构造器都要求有一个单独的构造器**，与此不同的是，这里的包装类（wrapper class）可以被用来包装任何Set实现，并且可以结合任何先前存在的构造器一起工作。例如：

Set<Date> s = new InstrumentedSet<Date>(new TreeSet<Date>(cmp));

Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));

因为每一个InstrumentedSet实例都把另一个Set实例包装起来了，所以InstrumentedSet类被称作包装类（wrapper class）。这也正是Decorator模式，因为InstrumentedSet类对一个Set惊醒了修饰，为它增加了计数特性。有时候，复合和转发的结合也被错误地称为“委托（delegation）”。从技术的角度而言，这不是委托，除非包装对象把自身传递给被包装的对象。

换句话说，对于两个类A和B，只有当两者之间确实存在“is-a”关系的时候，类B才应该扩展类A。如果答案是否定的，通常情况下，B应该包含A的一个私有实例，并且暴露一个较小的、较简单的API：A本质上不是B的一部分，只是它的实现细节而已。

在Java平台类库中，有许多明显违反这条原则的地方。例如，栈（stack）并不是向量（vector），所以Stack不应该扩展Vector。同样地，属性列表也不是散列表，所以Properties不应该扩展Hashtable。在这两种情况下，复合模式才是恰当的。

对于你正试图扩展的类，它的API中有没有缺陷呢？如果有，你是否愿意把那些缺陷传播到类的API中？继承机制会把超类API中的所有缺陷传播到子类中，而复合则允许设计新的API来隐藏这些缺陷。

为了避免这种脆弱性，可以用复合和转发机制来代替继承，尤其是当存在适当的接口可以实现包装类的时候。包装类不仅比子类更加健壮，而且功能也更加强大。

第17条：要么为继承而设计，并提供文档说明，要么就禁止继承

1. 对于为了继承而设计的类，唯一的测试方法就是编写子类。如果遗漏了关键的受保护成员，尝试编写子类就会使遗漏所带来的痛苦变得更加明显。相反，如果编写了多个子类，并且无一使用受保护的成员，或许就应该把它做成私有的。经验表明，3个子类通常就足以测试一个可扩展的类。
2. 构造器决不能调用可被覆盖的方法，无论是直接调用还是间接调用。如果违反了这条规则，很有可能导致程序失败。超类的构造器在子类的构造器之前运行，所以，子类中覆盖版本的方法将会在子类的构造器运行之前就先被调用。
3. 如果你决定在一个为了继承而设计的类中实现Cloneable或者Serializable接口，就应该意识到，因为clone和readObject方法在行为上非常类似于构造器，所以类似的限制规则也是适用的：无论是clone还是readObject，都不可以调用可覆盖的方法，不管是以直接还是间接方式。对于readObject方法，覆盖版本的方法将在子类的状态被反序列化（deserialized）之前先被运行；而对于clone方法，覆盖版本的方法则是在子类的clone方法有机会修正被克隆对象的状态之前先被运行。无论哪种情形，都不可避免地将导致程序失败。
4. 这条建议可能会引来争议，因为许多程序员已经习惯于对普通的具体类进行子类化，以便增加新的功能设施，比如仪表功能（instrumentation，如计数显示等）、通知机制或者同步功能，或者为了限制原有类中的功能。如果类实现了某个能够反映其本质的接口，比如Set、List或者Map，就不应该为了禁止子类化而感到后悔。第16条中介绍的包装类（wrapper class）模式提供了另一种更好的办法，让继承机制实现更多的功能。
5. 如果你认为必须允许从这样的类继承，一种合理的办法是确保这个类永远不会调用它的任何可覆盖的方法，并在文档中说明这一点。

第18条：接口优于抽象类

1. 这两种机制之间最明显的区别在于，抽象类允许包含某些方法的实现，但是接口则不允许。一个更为重要的区别在于，为了实现由抽象类定义的类型，类必须成为抽象类的一个子类。任何一个类，只要它定义了所有必要的方法，并且遵守通用约定，它就被允许实现一个接口，而不管这个类是处于类层次（class hierarchy）的哪个位置。
2. 虽然接口不允许包含方法的实现，但是，使用接口来定义类型并不妨碍你为程序员提供实现上的帮助。通过对你导出的每个重要接口都提供一个抽象的骨架实现（skeletal implementation）类，把接口和抽象类的优点结合起来。接口的作用仍然是定义类型，但是骨架实现类接管了所有与接口实现相关的工作。
3. 按照惯例，骨架实现被称为AbstractInterface，这里的Interface是指所实现的接口的名字。例如，Collections Framework为每个重要的集合接口都提供一个骨架实现，包括AbstractCollection、AbstractSet、AbstractList和AbstractMap。将它们称作SkeletalCollection、SkeletalSet、SkeletalList和SkeletalMap也是有道理的，但是现在Abstract的用法已经根深蒂固。
4. 编写骨架实现类相对比较简单，只是有点单调乏味。首先，必须认真研究接口，并确定哪些方法是最为基本的（primitive），其他的方法则可以根据它们来实现。这些基本方法将成为骨架实现类中的抽象方法。然后，必须为接口中所有其他的方法提供具体的实现。例如，Map.Entry接口的骨架实现类：

// Skeletal Implementation

public abstract class AbstractMapEntry<K,V> implements Map.Entry<K,V> {

// Primitive operations

public abstract K getKey();

public abstract V getValue();

// Entries in modifiable maps must override this method

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<?,?> arg = (Map.Entry) o;

return equals(getKey(), arg.getKey()) && equals(getValue(), arg.getValue());

}

private static boolean equals(Object o1, Object o2) {

return o1 == null ? o2 == null : o1.equals(o2);

}

// Implements the general contract of Map.Entry.hashCode

@Override public int hashCode() {

return hashCode(getKey()) ^ hashCode(getValue());

}

private static int hashCode(Object obj) {

return obj == null ? 0 : obj.hashCode();

}

}

骨架实现上有个小小的不同，就是简单实现（simple implementation），AbstractMap.SimpleEntry就是个例子。简单实现就像个骨架实现，这是因为它实现了接口，并且是为了继承而设计的，但是区别在于它不是抽象的：它是最简单的可能的有效实现。你可以原封不动地使用，也可以看情况将它子类化。

使用抽象类来定义允许多个实现的类型，与使用接口相比有一个明显的优势：抽象类的演变比接口的演变要容易得多。如果在后续的发行版本中，你希望在抽象类中增加新的方法，始终可以增加具体方法，它包含合理的默认实现。然后，该抽象类的所有现有实现都将提供这个新的方法。对于接口，这样做是行不通的。

一般来说，要想在公有接口中增加方法，而不破坏实现这个接口的所有现有的类，这是不可能的。之前实现该接口的类将会漏掉新增加的方法，并且无法再通过编译。在为接口增加新方法的同时，也为骨架实现类增加同样的新方法，这样可以在一定程度上减小由此带来的破坏，但是，这样做并没有真正解决问题。所有不从骨架实现类继承的接口实现仍然会遭到破坏。

简而言之，接口通常是定义允许多个实现的类型的最佳途径。这条规则有个例外，即当演变的容易性比灵活性和功能更为重要的时候。在这种情况下，应该使用抽象类来定义类型，但前提是必须理解并且可以接受这些局限性。如果你导出了一个重要的接口，就应该坚决考虑同时提供骨架实现类。最后，应该尽可能谨慎地设计所有的公有接口，并通过编写多个实现来对它们进行全面的测试。

第19条：接口只用于定义类型

1. 当类实现接口时，接口就充当可以引用这个类的实例的类型（type）。因此，类实现了接口，就表明客户端可以对这个类的实例实施某些动作。为了任何其他目的而定义接口是不恰当的。
2. 常量接口模式是对接口的不良使用。类在内部使用某些常量，这纯粹是实现细节。实现常量接口，会导致把这样的实现细节泄漏到该类的导出API中。类实现常量接口，这对于这个类的用户来讲并没有什么价值。实际上，这样做反而会让他们更加糊涂。更糟糕的是，它代表了一种承诺：如果在将来的发行版本中，这个类被修改了，它不再需要使用这些常量了，它依然必须实现这个接口，以确保二进制兼容性。如果非final类实现了常量接口，它的所有子类的命名空间也会被接口中的常量所“污染”。
3. 在Java平台类库中有几个常量接口，例如java.io.ObjectStreamConstants。这些接口应该被认为是反面的典型，不值得效仿。
4. 如果这些常量与某个现有的类或者接口紧密相关，就应该把这些常量添加到这个类或者接口中。例如，在Java平台类库中所有的数值包装类，如Integer和Double，都导出了MIN\_VALUE和MAX\_VALUE常量。如果这些常量最好被看作枚举类型的成员，就应该用枚举类型（enum type）来导出这些常量。否则，应该使用不可实例化的工具类（utility class）来导出这些常量。下面的例子是前面的PhysicalConstants例子的工具类翻版：

// Constant utility class

public class PhysicalConstants {

private PhysicalConstants() { } // Prevents instantiation

public static final double AVOGADROS\_NUMBER = 6.02214199e23;

public static final double BOLTZMANN\_CONSTANT = 1.3806503e-23;

public static final double ELECTRON\_MASS = 9.10938188e-31;

}

第20条：类层次优于标签类

1. 这种标签类（tagged class）有着许多缺点。它们中充斥着样板代码，包括枚举声明、标签域以及条件语句。由于多个实现乱七八糟地挤在了单个类中，破坏了可读性。内存占用也增加了，因此实例承担着属于其他风格的不相关的域。域不能做成是final的，除非构造器初始化了不相关的域，产生更多的样板代码。
2. 当你遇到一个包含标签域的现有类时，就要考虑将它重构到一个层次结构中去。

第21条：用函数对象表示策略

1. 然而，我们也可能定义这样一种对象，它的方法执行其他对象（other objects）（这些对象被显式传递给这些方法）上的操作。如果一个类仅仅导出这样的一个方法，它的实例实际上就等同于一个指向该方法的指针。这样的实例被称为函数对象（function object）。
2. 作为典型的具体策略类，StringLengthComparator类是无状态的（stateless）：它没有域，所以，这个类的所有实例在功能上都是相互等价的。因此，它作为一个Singleton是非常合适的，可以节省不必要的对象创建开销：

class StringLengthComparator {

private StringLengthComparator() { }

public static final StringLengthComparator INSTANCE = new StringLengthComparator();

public int compare(String s1, String s2) {

return s1.length() - s2.length();

}

}

为了把StringLengthComparator实例传递给方法，需要适当的参数类型。使用StringLengthComparator并不好，因为客户端将无法传递任何其他的比较策略。相反，我们需要定义一个Comparator接口，并修改StringLengthComparator来实现这个接口。换句话说，我们在设计具体的策略类时，还需要定义一个策略接口（strategy interface）。

因为策略接口被用做具体策略实例的类型，所以我们并不需要为了导出具体策略，而把具体策略类做成公有的。相反，“宿主类（host class）”还可以导出公有的静态域（或者静态工厂方法），其类型为策略接口，具体的策略类可以是宿主类的私有嵌套类。下面的例子使用静态成员类，而不是匿名类，以便允许具体的策略类实现第二个接口Seriablizable：

// Exporting a concrete strategy

class Host {

private static class StrLenCmp implements Comparator<String>, Serializable {

public int compare(String s1, String s2) {

return s1.length() - s2.length();

}

}

// Returned comparator is serializable

public static final Comparator<String> STRING\_LENGTH\_COMPARATOR = new StrLenCmp();

... // Bulk of class omitted

String类利用这种模式，通过它的CASE\_INTENSITIVE\_ORDER域，导出一个不区分大小写的字符串比较器。

简而言之，函数指针的主要途径就是实现策略（Strategy）模式。为了在Java中实现这种模式，要声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。当一个具体策略只被使用一次时，通常使用匿名类来声明和实例化这个具体策略类。当一个具体策略是设计用来重复使用的时候，它的类通常要被实现为私有的静态成员类，并通过公有的静态final域被导出，其类型为该策略接口。

第22条：优先考虑静态成员类

1. 嵌套类有四种：静态成员类（static member class）、非静态成员类（nonstatic member class）、匿名类（anonymous class）和局部类（local class）。除了第一种之外，其他三种都被称为内部类（inner class）。
2. 静态成员类的一种常见用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个枚举，它描述了计算器支持的各种操作。Operation枚举应该是Calculator类的公有静态成员类，然后，Calculator类的客户端就可以用诸如Calculator.Operation.PLUS和Calculator.Operation.MINUS这样的名称来引用这些操作。
3. 非静态成员类的每个实例都隐含着与外围类的一个外围实例（enclosing instance）相关联。
4. 如果嵌套类的实例可以在它外围类的实例之外独立存在，这个嵌套类就必须是静态成员类：在没有外围实例的情况下，要想创建非静态成员类的实例是不可能的。
5. 非静态成员类的一种常见用法是定义一个Adapter，它允许外部类的实例被看作是另一个不相关的类的实例。例如，Map接口的实现往往使用非静态成员类来实现它们的集合视图（collection view），这些集合视图是由Map的keySet、entrySet和Values方法返回的。同样地，诸如Set和List这种集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

// Typical use of a nonstatic member class

public class MySet<E> extends AbstractSet<E> {

... // Bulk of the class omitted

public Iterator<E> iterator() {

return new MyIterator();

}

private class MyIterator implements Iterator<E> {

...

}

}

如果声明成员类不要求访问外围实例，就要始终把static修饰符放在它的声明中，使它成为静态成员类，而不是非静态成员类。

私有静态成员类的一种常见用法是用来代表外围类所代表的对象的组件。例如，考虑一个Map实例，它把键（key）和值（value）关联起来。许多Map实现的内部都有一个Entry对象，对应于Map中的每个键-值对。虽然每个entry都与一个Map关联，但是entry上的方法（getKey、getValue和setValue）并不需要访问该Map。因此，使用非静态成员来表示entry是很浪费的：私有的静态成员类是最佳的选择。如果不小心漏掉了entry声明中的static修饰符，该Map仍然可以工作，但是每个entry中将会包含一个指向该Map的引用，这样就浪费了空间和时间。

正如你所想象的，匿名类没有名字。它不是外围类的一个成员。

匿名类的另一种常见用法是创建过程对象（process object），比如Runnable、Thread或者TimerTask实例。

第5章 泛型

在没有泛型之前，从集合中读取到的每一个对象都必须进行转换。如果有人不小心插入了类型错误的对象，在运行时的转换处理就会出错。有了泛型之后，可以告诉编译器每个集合中接受哪些对象类型。编译器自动地为你的插入进行转化，并在编译时告知是否插入了类型错误的对象。

第23条：请不要在新代码中使用原生态类型

1. 例如，从Java 1.5发行版本起，List接口就只有单个类型参数E，表示列表的元素类型。从技术角度来看，这个接口的名称应该是指现在的List<E>（读作“E的列表”），但是人们经常把它简称为List。泛型类和接口统称为泛型（generic type）。
2. 虽然假设不小心将coin插入到stamp集合中可能显得有点牵强，但这类问题却是真实的。例如，很容易想象有人会不小心将一个java.util.Date实例放进一个原本只包含java.sql.Date实例的集合中。
3. 虽然你可以将List<String>传递给类型List的参数，但是不能将它传给类型List<Object>的参数。泛型有子类型化（subtyping）的规则，List<String>是原生态类型List的一个子类型，而不是参数化类型List<Object>的子类型。因此，如果使用像List这样的原生态类型，就会失掉类型安全性，但是如果使用像List<Object>这样的参数化类型，则不会。
4. 从Java 1.5发行版本开始，Java就提供了一种安全的替代方法，称作无限制的通配符类型（unbounded wildcard type）。如果要使用泛型，但不确定或者不关心实际的类型参数，就可以使用一个问号代替。例如，泛型Set<E>的无限制通配符类型为Set<?>（读作“某个类型的集合”）。
5. 你不仅无法将任何元素（除null之外）放进Collection<?>中，而且根本无法猜测你会得到哪种类型的对象。要是无法接受这些限制，就可以使用泛型方法（generic method）或者有限制的通配符类型（bounded wildcard type）。
6. 在类文字（class literal）中必须使用原生态类型。规范不允许使用参数化类型（虽然允许数组类型和基本类型）。换句话说，List.class，String[].class和int.class都合法，但是List<String>.class和List<?>.class则不合法。
7. 由于泛型信息可以在运行时被擦除，因此在参数化类型而非无限制通配符类型上使用instanceof操作符是非法的。用无限制通配符类型代替原生态类型，对instanceof操作符的行为不会产生任何影响。
8. Set<?>则是一个通配符类型，表示只能包含某种未知对象类型的一个集合。

 有限制类型参数 <E extends Number>   
递归类型限制 <T extends Comparable<T>>   
有限制通配符类型 List<? extends Number>   
泛型方法 static <E> List<E> asList(E[] a)   
类型令牌 String.class

第24条：消除非受检警告

1. 如果无法消除警告，同时可以证明引起警告的代码是类型安全的，（只有在这种情况下才）可以用一个@SuppressWarnings(“unchecked”)注解来禁止这条警告。
2. SuppressWarnings注解可以用在任何粒度的级别中，从单独的局部变量声明到整个类都可以。应该始终在尽可能小的范围中使用SuppressWarnings注解。它通常是个变量声明，或是非常简短的方法或者构造器。永远不要在整个类上使用SuppressWarnings，这么做可能会掩盖了重要的警告。
3. 每当使用SuppressWarnings(“unchecked”)注解时，都要添加一条注释，说明为什么这么做是安全的。

第25条：列表优先于数组

1. 首先，数组是协变的（covariant）。这个词听起来有点吓人，其实只是表示如果Sub为Super的子类型，那么数组类型Sub[]就是Super[]的子类型。相反，泛型则是不可变的（invariant）：对于任意两个不同的类型Type1和Type2，List<Type1>既不是List<Type2>的子类型，也不是List<Type2>的超类型。你可能认为，这意味着泛型是有缺陷的，但实际上可以说数组才是有缺陷的。
2. 相比之下，泛型则是通过擦除（erasure）来实现的。因此泛型只在编译时强化它们的类型信息，并在运行时丢弃（或者擦除）它们的元素类型信息。
3. 由于上述这些根本的区别，因此数组和泛型不能很好地混合使用。例如，创建泛型、参数化类型或者类型参数的数组是非法的。这些数组创建表达式没有一个是合法的：new List<E>[]、new List<String>[]和new E[]。这些在编译时都会导致一个generic array creation（泛型数组创建）错误。
4. 从技术的角度来说，像E、List<E>和List<String>这样的类型应称作不可具体化的（non-reifiable）类型。直观地说，不可具体化的（non-reifiable）类型是指其运行时表示法包含的信息比它的编译时表示法包含的信息更少的类型。
5. 这也意味着在结合使用可变参数（varargs）方法和泛型时会出现令人费解的警告。这是由于每当调用可变参数方法时，就会创建一个数组来存放varargs参数。如果这个数组的元素类型不是可具体化的（reifiable），就会得到一条警告。
6. 编译器告诉你，它无法在运行时检查转换的安全性，因为它在运行时还不知道E是什么——记住，元素类型信息会在运行时从泛型中被擦除。

第26条：优先考虑泛型

1. 如第25条中所述，你不能创建不可具体化的（non-reifiable）类型的数组，如E。每当编写用数组支持的泛型时，都会出现这个问题。解决这个问题有两种方法。第一种，直接绕过创建泛型数组的禁令：创建一个Object的数组，并将它转换成泛型数组类型。现在错误是消除了，但是编译器会产生一条警告。这种用法是合法的，但（整体上而言）不是类型安全的。
2. 编译器不可能证明你的程序是类型安全的，但是你可以证明。你自己必须确保未受检的转换不会危及到程序的类型安全性。相关的数组（即elements变量）保存在一个私有的域中，永远不会被返回到客户端，或者传给任何其他方法。这个数组中保存的唯一元素，是传给push方法的那些元素，它们的类型为E，因此未受检的转换不会有任何危害。
3. 看来上述的示例与第25条相矛盾了，第25条鼓励优先使用列表而非数组。实际上并不可能总是或者总想在泛型中使用列表。Java并不是生来就支持列表，因此有些泛型如ArrayList，则必须在数组上实现。为了提升性能，其他泛型如HashMap也是数组上实现。

第27条：优先考虑泛型方法

1. 静态工厂方法尤其适合于泛型化。Collections中的所有“算法”方法（例如binarySearch和sort）都泛型化了。
2. 声明类型参数的类型参数列表，处在方法的修饰符及其返回类型之间。
3. 类型限制<T extends Comparable<T>>，可以读作“针对可以与自身进行比较的每个类型T”，这与互比性的概念或多或少有些一致。

第28条：利用有限制通配符来提升API的灵活性

1. 为了获得最大限度地灵活性，要在表示生产者或者消费者的输入参数上使用通配符类型。如果某个输入参数既是生产者，又是消费者，那么通配符类型对你就没有什么好处了：因为你需要的是严格的类型匹配，这是不用任何通配符而得到的。
2. PECS表示producer-extends, consumer-super。
3. 换句话说，如果参数化类型表示一个T生产者，就使用<? extends T>；如果它表示一个T消费者，就使用<? super T>。在我们的Stack示例中，pushAll的src参数产生E实例供Stack使用，因此src相应的类型为Iterable<? extends E>；popAll的dst参数通过Stack消费E实例，因此dst相应的类型为Collection<? super E>。
4. 不要用通配符类型作为返回类型。
5. 最初T被指定用来扩展Comparable<T>，但是T的comparable消费T实例（并产生表示顺序关系的整值）。因此，参数化类型Comparable<T>被有限制通配符类型Comparable<? super T>取代。comparable始终是消费者，因此使用时始终应该是Comparable<? super T>优先于Comparable<T>。对于comparator也一样，因此使用时始终应该是Comparator<? super T>优先于Comparator<T>。

// Won't compile - wildcards can require change in method body!

public static <T extends Comparable<? super T>> T max(List<? extends T> list) {

Iterator<T> i = list.iterator();

T result = i.next();

while(i.hasNext()) {

T t = i.next();

if(t.compareTo(result) > 0)

result = t;

}

return result;

}

它意味着list不是一个List<T>，因此它的iterator方法没有返回Iterator<T>。它返回T的某个子类型的一个iterator，因此我们用它代替iterator声明，它使用了一个有限制的通配符类型：

Iterator<? extends T> i = list.iterator();

问题在于list的类型为List<?>，你不能把null之外的任何值放到List<?>中。幸运的是，有一种方式可以实现这个方法，无需求助于不安全的转换或者原生态类型（raw type）。这种想法就是编写一个私有的辅助方法来捕捉通配符类型。为了捕捉类型，辅助方法必须是泛型方法，像下面这样：

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)));

}

**还要记住所有的comparable和comparator都是消费者。**

第29条：优先考虑类型安全的异构容器

1. 泛型最常用于集合，如Set和Map，以及单元素的容器，如ThreadLocal和AtomicReference。在这些用法中，它都充当被参数化了的容器。
2. 例如，数据库行可以有任意多的列，如果能以类型安全的方式访问所有列就好了。幸运的是，有一种方法可以很容易地做到这一点。这种想法就是将键（key）进行参数化而不是将容器（container）参数化。然后将参数化的键提交给容器，来插入或者获取值。用泛型系统来确保值的类型与它的键相符。
3. Class对象充当参数化键的部分。之所以可以这样，是因为类Class在Java 1.5版本中被泛型化了。类的类型从字面上来看不再只是简单的Class，而是Class<T>。例如，String.class属于Class<String>类型，Integer.class属于Class<Integer>类型。
4. Favorites实例是类型安全的（typesafe）的：当你向它请求String的时候，它从来不会返回一个Integer给你。同时它也是异构的（heterogeneous）：不像普通的map，它的所有键都是不同类型的。因此，我们将Favorites称作类型安全的异构容器（typesafe heterogeneous container）。

// Typesafe heterogeneous container pattern - implementation

public class Favorites {

private Map<Class<?>, Object> favorites = new HashMap<Class<?>, Object>();

public <T> void putFavorite(Class<T> type, T instance) {

if (type==null)

throw new NullPointerException("Type is null");

favorites.put(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实例都得到一个称作favorites的私有Map<Class<?>, Object>的支持。你可能认为由于无限制通配符类型的关系，将不能把任何东西放进这个Map中，但事实正好相反。要注意的是通配符类型是嵌套的：它不是属于通配符类型的Map的类型，而是它的键的类型。由此可见，每个键都可以有一个不同的参数化类型：一个可以是Class<String>，接下来是Class<Integer>等等。异构就是从这里来的。**

它先从favorites映射中获得与指定Class对象相对应的值。这正是要返回的对象引用，但它的编译时类型是错误的。它的类型只是Object（favorites映射的值类型），我们需要返回一个T。因此，getFavorite方法的实现利用Class的cast方法，将对象引用动态地转换（dynamically cast）成了Class对象所表示的类型。

cast方法是Java的cast操作符的动态模拟。它只检验它的参数是否为Class对象所表示的类型的实例。如果是，就返回参数；否则就抛出ClassCastException异常。

换句话说，你可以保存最喜爱的String或者String[]，但不能保存最喜爱的List<String>。如果试图保存最喜爱的List<String>，程序就不能进行编译。原因在于你无法为List<String>获得一个Class对象：List<String>.Class是个语法错误，这也是件好事。List<String>和List<Integer>共用一个Class对象，即List.class。

第6章 枚举和注解

Java 1.5发行版本中增加了两个新的引用类型家族：一种新的类称作枚举类型（enum type），一种新的接口称作注解类型（annotation type）。

第30条：用enum代替int常量

1. 这种方法称作int枚举模式（int enum pattern），存在着诸多不足。它在类型安全性和使用方便性方面没有任何帮助。如果你将apple传到想要orange的方法中，编译器也不会出现警告，还会用==操作符将apple与orange进行对比，甚至更糟糕。
2. 采用int枚举模式的程序是十分脆弱的。因为int枚举是编译时常量，被编译到使用它们的客户端中。如果与枚举常量关联的int发生了变化，客户端就必须重新编译。如果没有重新编译，程序还是可以运行，但是它们的行为就是不确定的。
3. 将int枚举常量翻译成可打印的字符串，并没有很便利的方法。
4. Java的枚举类型是功能十分齐全的类，功能比其他语言中的对等物要更强大得多，Java的枚举本质上是int值。
5. Java枚举类型背后的基本想法非常简单：它们就是通过公有的静态final域为每个枚举常量导出实例的类。因为没有可以访问的构造器，枚举类型是真正的final。
6. 它们是单例（Singleton）的泛型化，本质上是单元素的枚举。
7. 你可以增加或者重新排列枚举类型中的常量，而无需重新编译它的客户端代码，因为导出常量的域在枚举类型和它的客户端之间提供了一个隔离层：常量值并没有被编译到客户端代码中，而是在int枚举模式之中。最终，可以通过调用toString方法，将枚举转换成可打印的字符串。

// 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

}

}

为了将数据与枚举常量关联起来，得声明实例域，并编写一个带有数据并将数据保存在域中的构造器。枚举天生就是不可变的，因此所有的域都应该为final的。

注意Plant就像所有的枚举一样，它有一个静态的values方法，按照声明顺序返回它的值数组。还要注意toString方法返回每个枚举值的声明名称，使得println和printf的打印变得更加容易。如果你不满意这种字符串表示法，可以通过覆盖toString方法对它进行修改。

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，并在特定于常量的类主体（constant-specific class body）中，用具体的方法覆盖每个常量的抽象apply方法。这种方法被称作特定于常量的方法实现（constant-specific method implementation）：

// Enum type with constant-specific method implementations

public enum Operation {

PLUS { double apply(double x, double y){return x+y;}},

MINUS { double apply(double x, double y){return x-y;}},

TIMES { double apply(double x, double y){return x\*y;}},

DIVIDE { double apply(double x, double y){return x/y;}};

abstract double apply(double x, double y);

}

即使你真的忘记了，编译器也会提醒你，因为枚举类型中的抽象方法必须被它所有常量中的具体方法所覆盖。

特定于常量的方法实现可以与特定于常量的数据结合起来。例如，下面的Operation覆盖了toString来返回通常与该操作关联的符号：

// Enum type with constant-specific class bodies and data

public enum Operation {

PLUS("+") {

double apply(double x, double y) { return x+y; }

},

MINUS("-") {

double apply(double x, double y) { return x-y; }

},

TIMES("\*") {

double apply(double x, double y) { return x\*y; }

},

DIVIDE("/") {

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; }

abstract double apply(double x, double y);

}

枚举类型有一个自动产生的valueOf(String)方法，它将常量的名字转变成常量本身。

枚举构造器不可以访问枚举的静态域，除了编译时常量域之外。这一限制是有必要的，因为构造器运行的时候，这些静态域还没有被初始化。

你真正想要的就是每当添加一个枚举常量时，就强制选择一种加班报酬策略。幸运的是，有一种很好的方法可以实现这一点。这种想法就是将加班工资计算移到一个私有的嵌套枚举中，将这个策略枚举（strategy enum）的实例传到PayrollDay枚举的构造器中。之后PayrollDay枚举将加班工资计算委托给策略枚举，PayrollDay中就不需要switch语句或者特定于常量的方法实现了。虽然这种模式没有switch语句那么简洁，但更加安全，也更加灵活：

// The strategy enum pattern

enum PayrollDay {

MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),

WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),

FRIDAY(PayType.WEEKDAY),

SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

private final PayType payType;

PayrollDay(PayType payType) { this.payType = payType; }

double pay(double hoursWorked, double payRate) {

return payType.pay(hoursWorked, payRate);

}

// The strategy enum type

private enum PayType {

WEEKDAY {

double overtimePay(double hours, double payRate) {

return hours<=HOURS\_PER\_SHIFT ? 0 :

(hours - HOURS\_PER\_SHIFT) \* payRate / 2;

}

},

WEEKEND {

double overtimePay(double hours, double payRate) {

return hours \* payRate / 2;

}

};

private static final int HOURS\_PER\_SHIFT = 8;

abstract double overtimePay(double hrs, double payRate);

double pay(double hoursWorked, double payRate) {

double basePay = hoursWorked \* payRate;

return basePay + overtimePay(hoursWorked, payRate);

}

}

}

枚举中的switch语句适合于给外部的枚举类型增加特定于常量的行为。

那么什么时候应该使用枚举呢？每当需要一组固定常量的时候。当然，这包括“天然的枚举类型”，例如行星、一周的天数以及棋子的数目等等。但它也包括你在编译时就知道其所有可能值得其他集合，例如菜单的选项、操作代码以及命令行标记等。枚举类型中的常量集并不一定要始终保持不变。专门设计枚举特性是考虑到枚举类型的二进制兼容演变。

第31条：用实例域代替序数

1. 永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中：

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; }

}

Enum规范中谈到ordinal时这么写道：“大多数程序员都不需要这个方法。它是设计成用于像EnumSet和EnumMap这种基于枚举的通用数据结构的。”除非你在编写的是这种数据结构，否则最好完全避免使用ordinal方法。

第32条：用EnumSet代替位域

1. java.util包提供了EnumSet类来有效地表示从单个枚举类型中提取的多个值的多个集合。这个类实现Set接口，提供了丰富的功能、类型安全性，以及可以从任何其他Set实现中得到的互用性。但是在内部具体的实现上，每个EnumSet内容都表示为位矢量。如果底层的枚举类型有64个或者更少的元素——大多如此——整个EnumSet就是单个long来表示，因此它的性能比得上位域的性能。批处理，如removeAll和retainAll，都是利用位算法来实现的，就像手工替位域实现得那样。但是可以避免手工位操作时容易出现的错误以及不太雅观的代码，因为EnumSet替你完成了这项艰巨的工作。

// 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) { ... }

}

text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));

注意applyStyles方法采用的是Set<Style>而非EnumSet<Style>。虽然看起来好像所有的客户端都可以将EnumSet传到这个方法，但是最好还是接受接口类型而非接受实现类型。这是考虑到可能会有特殊的客户端要传递一些其他的Set实现，并且没有什么明显的缺点。

第33条：用EnumMap代替序数索引

// Using an EnumMap to associate data with an enum

Map<Herb.Type, Set<Herb>> herbsByType =

new EnumType<Herb.Type, Set<Herb>>(Herb.Type.class);

for (Herb.Type t : Herb.Type.values())

herbsByType.put(t, new HashSet<Herb>());

for (Herb h : garden)

herbsByType.get(h.type).add(h);

System.out.println(herbsByType);

EnumMap在运行速度方面之所以能与通过序数索引的数组相媲美，是因为EnumMap在内部使用了这种数组。

注意EnumMap构造器采用键类型的Class对象：这是一个有限制的类型令牌（bounded type token），它提供了运行时的泛型信息。

// 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 src;

private final Phase dst;

Transition(Phase src, Phase dst) {

this.src = src;

this.dst = dst;

}

// Initialize the phase transition map

private staic 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);

}

}

}

现在假设想要给系统添加一个新的阶段：plasma（离子）或者电离气体。只有两个过渡与这个阶段关联：电离化，它将气体变成离子；以及消电离化，将离子变成气体。

为了更新基于EnumMap的版本，所要做的就是必须将PLASMA添加到Phase列表，并将IONIZE(GAS, PLASMA)和DEIONIZE(PLASMA, GAS)添加到Phase.Transition的列表中。程序会自行处理所有其他的事情，你几乎没有机会出错。从内部来看，Map的Map被实现成了数组的数组，因此在提升了清楚性、安全性和易维护性的同时，在空间或者时间上还几乎不用任何开销。

总而言之，最好不要用序数来索引数组，而要使用EnumMap。如果你所表示的这种关系是多维的，就使用EnumMap<…, EnumMap<…>>。

第34条：用接口模拟可伸缩的枚举

1. 枚举的可伸缩性最后证明基本上都不是什么好点子。
2. 总而言之，虽然无法编写可扩展的枚举类型，却可以通过编写接口以及实现该接口的基础枚举类型，对它进行模拟。这样允许客户端编写自己的枚举来实现接口。如果API是根据接口编写的，那么在可以使用基础枚举类型的任何地方，也都可以使用这些枚举。

第35条：注解优先于命名模式

1. Java 1.5发行版本之前，一般使用命名模式（naming pattern）表明有些程序元素需要通过某种工具或者框架进行特殊处理。例如，JUnit测试框架原本要求它的用户一定要用test作为测试方法名称的开头。
2. 测试运行工具在命令行上使用完全匹配的类名，并通过调用Method.invoke反射式地运行类中所有标注了Test的方法。isAnnotationPresent方法告知该工具要运行哪些方法。如果测试方法抛出异常，反射机制就会将它封装在InvocationTargetException中。该工具捕捉到了这个异常，并打印失败报告，包含测试方法抛出的原始异常，这些信息是通过getCause方法从InvocationTargetException中提取出来的。

第36条：坚持使用Override注解

// 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<Bigram>();

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());

}

}

如果你试着运行程序，会发现它打印的不是26而是260。哪里出错了呢？

很显然，Bigram类的创建者原本想要覆盖equals方法，同时还记得覆盖了hashCode。遗憾的是，不幸的程序员没能覆盖equals，而是将它重载了。

@Override public boolean equals(Object o) {

if(!(o instanceof Bigram))

return false;

Bigram b = (Bigram) o;

return b.first == first && b.second == second;

}

现代的IDE提供了坚持使用Override注解的另一种理由。这种IDE具有自动检查功能，称作代码检验（code inspection）。如果启用相应的代码检验功能，当有一个方法没有Override注解，却覆盖了超类方法时，IDE就会产生一条警告。

第37条：用标记接口定义类型

1. 标记接口（marker interface）是没有包含声明方法的接口，而只是指明（或者“标明”）一个类实现了具有某种属性的接口。例如，考虑Serializable接口。通过实现这个接口，类表明它的实例可以被写到ObjectOutputStream（或者“被序列化”）。
2. 你可能听说过标记注解使得标记接口过时了。这种断言是不正确的。标记接口有两点胜过标记注解。首先，也是最重要的一点是，标记接口定义的类型是由被标记类的实例实现的；标记注解则没有定义这样的类型。这个类型允许你在编译时捕捉在使用标记注解的情况下要到运行时才能捕捉到的错误。
3. 如果标记只应用给类和接口，就要问问自己：我要编写一个还是多个只接受有这种标记的方法呢？如果是这种情况，就应该优先使用标记接口而非注解。这样你就可以用接口作为相关方法的参数类型，它真正可以为你提供编译时进行类型检查的好处。
4. 如果你对第一个问题的回答是否定的，就要再问问自己：我要永远限制这个标记只用于特殊接口的元素吗？如果是，最好将标记定义成该接口的一个子接口。如果这两个问题的答案都是否定的，或许就应该使用标记注解。

第7章 方法

第38条：检查参数的有效性

1. 对于公有的方法，要用Javadoc的@throws标签（tag）在文档中说明违反参数值限制时会抛出的异常。这样的异常通常为IllegalArgumentException、IndexOutOfBoundsException或NullPointerException。
2. 对于未被导出的方法（unexported method），作为包的创建者，你可以控制这个方法将在哪些情况下被调用，因此你可以，也应该确保只将有效的参数值传递进来。因此，非公有的方法通常应该使用断言（assertion）来检查它们的参数，具体做法如下：

// 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

}

如前所述，有些参数被方法保存起来供以后使用，构造器正是代表了这种原则的一种特殊情形。检查构造器参数的有效性是非常重要的，这样可以避免构造出来的对象违反了这个类的约束条件。

不要从本条目的内容中得出这样的结论：对参数的任何限制都是件好事。相反，在设计方法时，应该使他们尽可能地通用，并符合实际的需要。假如方法对于它能接受的所有参数值都能够完成合理的工作，对参数的限制就应该是越少越好。然而，通常情况下，有些限制对于被实现的抽象来说是固有的。

第39条：必要时进行保护性拷贝

// Broken "immutable" time period class

public final class Period {

private final Date start;

private final Date end;

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;

}

}

然而，因为Date类本身是可变的，因此很容易违反这个约束条件：

// 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!

为了保护Period实例的内部信息避免受到这种攻击，对于构造器的每个可变参数进行保护性拷贝（defensive copy）是必要的，并且使用备份对象作为Period实例的组件，而不使用原始的对象：

// 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(start + " after " + end);

}

注意，保护性拷贝是在检查参数的有效性之前进行的，并且有效性检查是针对拷贝之后的对象，而不是针对原始的对象。这样做可以避免在“危险阶段（window of vulnerability）”期间从另一个线程改变类的参数，这里的危险阶段是指从检查参数开始，直到拷贝参数之间的时间段。（在计算机安全社区中，这被称作Time-Of-Check/Time-Of-Use或者TOCTOU攻击。）

我们没有用Date的clone方法来进行保护性拷贝。因为Date是非final的，不能保证clone方法一定返回类为java.util.Date的对象：它有可能返回专门出于恶意的目的而设计的不可信子类的实例。

// Second attack on the internals of a Period instance

Date start = new Date();

Date end = new Date();

Period p = new Period(start, end);

p.end().setYear(78); // Modifies internals of p!

为了防御这第二种攻击，只需修改这两个访问方法，使它返回可变内部域的保护性拷贝即可：

// Repaired accessors - make defensive copies of internal fields

public Date start() {

return new Date(start.getTime());

}

public Date end() {

return new Date(end.getTime());

}

每当编写方法或者构造器时，如果它要允许客户提供的对象进入到内部数据结构中，则有必要考虑一下，客户提供的对象是否有可能是可变的。如果是，就要考虑你的类是否能够容忍对象进入数据结构之后发生变化。如果答案是否定的，就必须对该对象进行保护性拷贝，并且让拷贝之后的对象而不是原始对象进入到数据结构中。例如，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（key），就应该意识到，如果这个对象在插入之后再被修改，Set或者Map的约束条件就会遭到破坏。

在内部组件被返回给客户端之前，对它们进行保护性拷贝也是同样的道理。不管类是否为不可变的，在把一个指向内部可变组件的引用返回给客户端之前，也应该加倍认真地考虑。解决方案是，应该返回保护性拷贝。记住长度非零的数组总是可变的。因此，在把内部数组返回客户端之前，应该总要进行保护性拷贝。另一种解决方案是，给客户端返回该数组的不可变视图（immutable view）。

可以肯定地说，上述的真正启示在于，只要有可能，都应该使用不可变的对象作为对象内部的组件，这样就不必再为保护性拷贝操心。在前面的Period例子中，值得一提的是，有经验的程序员通常使用Date.getTime()返回的long基本类型作为内部的时间表示法，而不是使用Date对象引用。他们之所以这样做，主要因为Date是可变的。

第40条：谨慎设计方法签名

1. 避免过长的参数列表。目标是四个参数，或者更少。大多数程序员都无法记住更长的参数列表。
2. 相同类型的长参数序列格外有害。
3. 但是通过提升它们的正交性（orthogonality），还可以减少（reduce）方法的数目。例如，考虑java.util.List接口。它并没有提供“在子列表（sublist）中查找元素的第一个索引和最后一个索引”的方法，这两个方法都需要三个参数。相反，它提供了subList方法，这个方法带有两个参数，并返回子列表的一个视图（view）。这个方法可以与indexOf或者lastIndexOf方法结合起来，获得期望的功能，而这两个方法都分别只有一个参数。而且，subList方法也可以与其他任何“针对List实例进行操作”的方法结合起来，在子列表上执行任意的计算。这样得到的API就有很高的“功能-重量”（power-to-weight）比。
4. 缩短长参数列表的第二种方法是创建辅助类（helper class），用来保存参数的分组。这些辅助类一般为静态成员类。如果一个频繁出现的参数序列可以被看作是代表了某个独特的实体，则建议使用这种方法。例如，假设你正在编写一个表示纸牌游戏的类，你会发现，经常要传递一个两参数的序列来表示纸牌的点数和花色。如果增加辅助类来表示一张纸牌，并且把每个参数序列都换成这个辅助类的单个参数，那么这个纸牌游戏类的API以及它的内部表示都可能会得到改进。
5. 对于参数类型，要优先使用接口而不是类。只要有适当的接口可用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。例如，没有理由在编写方法时使用HashMap类来作为输入，相反，应当使用Map接口作为参数。这使你可以传入一个Hashtable、HashMap、TreeMap、TreeMap的子映射表（submap），或者任何有待于将来编写的Map实现。
6. 对于boolean参数，要优先使用两个元素的枚举类型。它使代码更易于阅读和编写，尤其当你在使用支持自动完成功能的IDE的时候。它也使以后更易于添加更多的选项。例如，你可能会有一个Thermometer类型，它带有一个静态工厂方法，而这个静态工厂方法的签名需要传入这个枚举的值：

public enum TemperatureScale { FAHRENHEIT, CELSIUS }

Thermometer.newInstance(TemperatureScale.CELSIUS)不仅比Thermometer.newInstance(true)更有用，而且你还可以在未来的发行版本中将KELVIN添加到TemperatureScale中，无需非得给Thermometer添加新的静态工厂。你还可以将依赖于温度刻度单位的代码重构到枚举常量的方法中。例如，每个刻度单位都可以有一个方法，它带有一个double值，并将它规格化成摄氏度。

第41条：慎用重载

// 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));

}

}

因为classify方法被重载（overloaded）了，而要调用哪个重载（overloading）方法是在编译时做出决定的。对于for循环中的全部三次迭代，参数的编译时类型都是相同的：Collection<?>。

这个程序的行为有悖常理，因为对于重载方法（overloaded method）的选择是静态的，而对于被覆盖的方法（overridden method）的选择则是动态的。

在CollectionClassifier这个示例中，该程序的意图是：期望编译器根据参数的运行时类型自动将调用分发给适当的重载方法，以此来识别出参数的类型，就好像Wine的例子中的name方法所做的那样。方法重载机制完全没有提供这样的功能。假设需要有个静态方法，这个程序的最佳修正方案是，用单个方法来替换这三个重载的classify方法，并在这个方法中做一个显式的instanceof测试：

public static String classify(Collection<?> c) {

return c instanceof Set ? "Set" :

c instanceof List ? "List" : "Unknown Collection";

}

因为覆盖机制是规范，而重载机制是例外，所以，覆盖机制满足了人们对于方法调用行为的期望。

安全而保守的策略是，永远不要导出两个具有相同参数数目的重载方法。如果方法使用可变参数（varargs），保守的策略是根本不要重载它，除第42条中所述的情形之外。

这项限制并不麻烦，因为你始终可以给方法起不同的名字，而不使用重载机制。

例如，考虑ObjectOutputStream类。对于每个基本类型，以及几种引用类型，它的write方法都有一种变形。这些变形方法并不是重载write方法，而是具有诸如writeBoolean(boolean)、writeInt(int)和writeLong(long)这样的签名。与重载方案相比较，这种命名模式带来的好处是，有可能提供相应名称的读方法，比如readBoolean()、readInt()和readLong()。实际上，ObjectInputStream类正是提供了这样的读方法。

对于构造器，你没有选择使用不同名称的机会；一个类的多个构造器总是重载的。在许多情况下，可以选择导出静态工厂，而不是构造器。对于构造器，还不用担心重载和覆盖的相互影响，因为构造器不可能被覆盖。或许你有可能导出多个具有相同参数数目的构造器，所以有必要了解一下如何安全地做到这一点。

如果对于每一对重载方法，至少有一个对应的参数在两个重载方法中具有“根本不同（radically different）”的类型，就属于这种情形。如果显然不可能把一种类型的实例转换为另一种类型，这两种类型就是根本不同的。

public class SetList {

public static void main(String[] args) {

Set<Integer> set = new TreeSet<Integer>();

List<Integer> list = new ArrayList<Integer>();

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);

}

}

实际发生的情况是：set.remove(i)调用选择重载方法remove(E)，这里的E是集合（Integer）的元素类型，将i从int自动装箱到Integer中。这是你所期待的行为，因此程序会从集合中去除正值。另一方面，list.remove(i)调用选择重载方法remove(int i)，它从列表的指定位置上去除元素。为了解决这个问题，要将list.remove的参数转换成Integer，迫使选择正确的重载方法。另一种方法是，可以调用Integer.valueOf(i)，并将结果传给list.remove。这两种方法都如我们所料，打印出[-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))

}

当它在Java 1.5发行版本中被泛型化之前，List接口有一个remove(Object)而不是remove(E)，相应的参数类型：Object和int，则根本不同。但是自从有了泛型和自动装箱之后，这两种参数类型就不再根本不同了。换句话说，Java语言中添加了泛型和自动装箱之后，破坏了List接口。幸运的是，Java类库中几乎再没有API受到同样的破坏，但是这种情形清楚地说明了，自动装箱和泛型成了Java语言的一部分之后，谨慎重载显得更加重要了。

数组类型和Object之外的类截然不同。数组类型和Serializable与Cloneable之外的接口也截然不同。如果两个类都不是对方的后代，这两个独特的类就是不相关的（unrelated）。例如，String和Throwable就是不相关的。任何对象都不可能是两个不相关的类的实例，因此不相关的类是根本不同的。

有时候，尤其在更新现有类的时候，可能会被迫违反本条目的指导原则。例如，自从Java 1.4发行版本以来，String类就已经有一个contentEquals（StringBuffer）方法。在Java 1.5发行版本中，新增了一个称作CharSequence的接口，用来为StringBuffer、StringBuilder、String、CharBuffer以及其他类似的类型提供公共接口，为实现这个接口，对它们全都进行了改造。在Java平台中增加CharSequence的同时，String也配备了重载的contentEquals方法，即contentEquals(CharSequence)方法，这个方法表示，当且仅当此String表示与CharSequence序列相同的char值时，返回true。

尽管这样的重载很显然违反了本条目的指导原则，但是只要当这两个重载方法在同样的参数上被调用时，它们执行相同的功能，重载就不会带来危害。程序员可能并不知道哪个重载函数会被调用，但只要这两个方法返回相同的结果就行。确保这种行为的标准做法是，让更具体化的重载方法把调用转发给更一般化的重载方法：

public boolean contentEquals(StringBuffer sb) {

return contentEquals((CharSequence) sb);

}

虽然Java平台类库很大程度上遵循了本条目中的建议，但是也有诸多的类违背了。例如，String类导出两个重载的静态工厂方法：valueOf(char[])和valueOf(Object)，当着两个方法被传递了同样的对象引用时，它们所做的事情完全不同。没有正当的理由可以解释这一点，它应该被看作是一种反常行为，有可能会造成真正的混淆。

第42条：慎用可变参数

// 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;

}

1. 可变参数是为printf而设计的，它是在Java 1.5发行版本中添加到平台中的，为了核心的反射机制，在该发行版本中被改造成利用可变参数。printf和反射机制都从可变参数中极大地受益。
2. 从好的方面看，将数组转变成字符串的Arrays.asList做法现在是过时的，当前的做法要健壮得多。也是在Java 1.5发行版本中，Arrays类得到了补充完整的Arrays.toString方法（不是可变参数方法！），专门为了将任何类型的数组转变成字符串而设计的。
3. 不必改造具有final数组参数的每个方法；只当确实是在数量不定的值上执行调用时才使用可变参数。
4. 可变参数方法的每次调用都会导致进行一次数组分配和初始化。如果凭经验确定无法承受这一成本，但又需要可变参数的灵活性，还有一种模式可以让你如愿以偿。假设确定对某个方法95%的调用会有3个或者更少的参数，就声明该方法的5个重载，每个重载方法带有0至3个普通参数，当参数的数目超过3个时，就使用一个可变参数方法：

public void foo() { }

public void foo(int a1) { }

public void foo(int a1, int a2) { }

public void foo(int a1, int a2, int a3) { }

public void foo(int a1, int a2, int a3, int... rest) { }

EnumSet类对它的静态工厂使用这种方法，最大限度地减少创建枚举集合的成本。当时这么做是有必要的，因为枚举集合为位域提供在性能方面有竞争力的替代方法，这是很重要的。

第43条：返回零长度的数组或者集合，而不是null

1. 把没有奶酪可买的情况当作是一种特例，这是不合常理的。这样做会要求客户端中必须有额外的代码来处理null返回值。
2. 对于一个返回null而不是零长度数组或者集合的方法，几乎每次用到该方法时都需要这种曲折的处理方式。这样做很容易出错，因为编写客户端程序的程序员可能会忘记写这种专门的代码来处理null返回值。
3. 有时候会有人认为：null返回值比零长度数组更好，因为它避免了分配数组所需要的开销。这种观点是站不住脚的，原因有两点。第一，在这个级别上担心性能问题是不明智的，除非分析表明这个方法正是造成性能问题的真正源头。第二，对于不返回任何元素的调用，每次都返回同一个零长度数组是有可能的，因为零长度数组是不可变的，而不可变对象有可能被自由地分享。实际上，当你使用标准做法（standard idiom）把一些元素从一个集合转存到一个类型化的数组（typed array）中时，它正是这样做的：

// The right way to return an array from a collection

private final List<Cheese> cheesesInStock = ...;

private static final Cheese[] EMPTY\_CHEESE\_ARRAY = new Cheese[0];

public Cheese[] getCheeses() {

return cheesesInStock.toArray(EMPTY\_CHEESE\_ARRAY);

}

在这种习惯用法中，零长度数组常量被传递给toArray方法，以指明所期望的返回类型。正常情况下，toArray方法分配了返回的数组，但是，如果集合是空的，它将使用零长度的输入数组，Collection.toArray(T[])的规范保证：如果输入数组大到足够容纳这个集合，它就将返回这个输入数组。因此，这种做法永远也不会分配零长度的数组。   
**List有**Object[] toArray()**和**<T> T[] toArray(T[]a)**两个方法。它对a的说明是：the array into which the elements of this list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.**

同样的，集合值的方法也可以做成在每当需要返回空集合时都返回同一个不可变的空集合。Collections.emptySet、emptyList和emptyMap方法提供的正是你所需要的，如下所示：

// The right way to return a copy of a collection

public List<Cheese> getCheeseList() {

if(cheesesInStock.isEmpty())

return Collections.emptyList(); // Always returns same list

else

return new ArrayList<Cheese>(cheesesInStock);

}

第44条：为所有导出的API元素编写文档注释

1. 除了前提条件和后置条件之外，每个方法还应该在文档中描述它的副作用（side effect）。所谓副作用是指系统状态中可以观察到的变化，它不是为了获得后置条件而明确要求的变化。例如，如果方法启动了后台线程，文档中就应该说明这一点。最后，文档注释也应该描述类或者方法的线程安全性（thread safety）。
2. 在Java 1.5发行版本之前，是通过使用HTML标签和HTML转义，将代码片段包含在文档注释中。现在再也没有必要在文档注释中使用HTML<code>或者<tt>标签了：Javadoc{@code}标签更好，因为它避免了转义HTML元字符。
3. 不要忘记，为了产生包含HTML元字符的文档，比如小于号（<）、大于号（>）以及“与”号（&），必须采取特殊的动作。让这些字符出现在文档中的最佳办法是用{@literal}标签将它们包围起来，这样就限制了HTML标记和嵌套的Javadoc标签的处理。除了它不以代码字体渲染文本之外，其余方面就像{@code}标签一样。

第8章 通用程序设计

第45条：将局部变量的作用域最小化

1. 要使局部变量的作用域最小化，最有力的方法就是在第一次使用它的地方声明。如果变量在使用之前进行声明，这只会造成混乱——对于试图理解程序功能的读者来说，这又多了一种只会分散他们注意力的因素。
2. 几乎每个局部变量的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对一个变量进行有意义的初始化，就应该推迟这个声明，直到可以初始化为止。这条规则有个例外的情况与try-catch语句有关。如果一个变量被一个方法初始化，而这个方法可能会抛出一个受检的异常（checked exception），该变量就必须在try块的内部被初始化。如果变量的值必须在try块的外部被使用到，它就必须在try块之前被声明，但是在try块之前，它还不能被“有意义地初始化”。
3. 循环中提供了特殊的机会来将变量的作用域最小化。（无论是传统的还是for-each形式的）for循环，都允许声明循环变量（loop variable），它们的作用域被限定在正好需要的范围之内。（这个范围包括循环体，以及循环体之前的初始化、测试、更新部分。）因此，如果在循环终止之后不再需要循环变量的内容，for循环就优先于while循环。

第46条：for-each循环优先于传统的for循环

1. 注意，利用for-each循环不会有性能损失，甚至用于数组也一样。实际上，在某些情况下，比起普通的for循环，它还稍有些性能优势，因为它对数组索引的边界值只计算一次。虽然可以手工完成这项工作，但程序员并不总会这么做。
2. 为了修正这些示例中的Bug，必须在外部循环的作用域中添加一个变量来保存外部元素：

// Fixed, but ugly - you can do better!

for (Iterator<Suit> i=suits.iterator(); i.hasNext(); ) {

Suit suit = i.next();

for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )

deck.add(new Card(suit, j.next()));

}

如果使用的是嵌套的for-each循环，这个问题就会完全消失。产生的代码就如你所希望得那样简洁。

// Preferred idiom for nested iteration on collections and arrays

for (Suit suit : suits)

for (Rank rank : ranks)

deck.add(new Card(suit, rank));

实现Iterable接口并不难。如果你在编写的类型表示的是一组元素，即使你选择不让它实现Collection，也要让它实现Iterable。这样可以允许用户利用for-each循环遍历你的类型，会令用户永远感激不尽的。

遗憾的是，有三种常见的情况无法使用for-each循环：

过滤——如果需要遍历集合，并删除选定的元素，就需要使用显式的迭代器，以便可以调用它的remove方法。   
转换——如果需要遍历列表或者数组，并取代它部分或者全部的元素值，就需要列表迭代器或者数组索引，以便设定元素的值。   
平行迭代——如果需要并行地遍历多个集合，就需要显式地控制迭代器或者索引变量，以便所有迭代器或者索引变量都可以得到同步前移（就如上述关于有问题的牌和头子的示例中无意中所示范的那样）。

第47条：了解和使用类库

1. 在1.2发行版本中，Collections Framework（集合框架）被加入到了java.util包中。它应该成为每个程序员基本工具箱中的一部分。Collections Framework是一个统一的体系结构，用来表示和操作集合，允许它们对集合进行独立于表示细节的操作。
2. 1.5发行版本中，在java.util.concurrent包中增加了一组并发实用工具。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，也应该是每个程序员基本工具箱中的一部分。
3. 一般而言，类库的代码可能比你自己编写的代码更好一些，并且会随着时间的推移而不断改进。这并不是在影射你作为一个程序员的能力。从经济角度的分析表明：类库代码受到的关注远远超过大多数普通程序员在同样的功能上所能够给予的投入。

第48条：如果需要精确的答案，请避免使用float和double

1. float和double类型尤其不适合用于货币计算，因为要让一个float或者double精确地表示0.1（或者10的任何其他负数次方值）是不可能的。
2. 解决这个问题的正确办法是使用BigDecimal、int或者long进行货币计算。

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)) {

itemsBought++;

funds = funds.substract(price);

}

System.out.println(itemsBought + " items bought.");

System.out.println("Money left over: $" + funds);

}

如果你想让系统来记录十进制小数点，并且不介意因为不使用基本类型而带来的不便，就请使用BigDecimal。使用BigDecimal还有一些额外的好处，它允许你完全控制舍入，每当一个操作涉及舍入的时候，它允许你从8种舍入模式中选择其一。如果正通过法定要求的舍入行为进行业务计算，使用BigDecimal是非常方便的。

第49条：基本类型优先于装箱基本类型

1. 第一，基本类型只有值，而装箱基本类型则具有与它们的值不同的同一性。换句话说，两个装箱基本类型可以具有相同的值和不同的同一性。第二，基本类型只有功能完备的值，而每个装箱基本类型除了它对应基本类型的所有功能值之外，还有个非功能值：null。
2. 它在计算表达式（i==42）的时候抛出NullPointerException异常。问题在于，i是个Integer，而不是int，就像所有的对象引用域一样，它的初始值为null。当程序计算表达式（i==42）时，它会将Integer与int进行比较。几乎在任何一种情况下，当在一项操作中混合使用基本类型和装箱基本类型时，装箱基本类型就会自动拆箱，这种情况无一例外。如果null对象引用被自动拆箱，就会得到一个NullPointerException。
3. 那么什么时候应该使用装箱基本类型呢？它们有几个合理的用处。第一个是作为集合中的元素、键和值。你不能将基本类型放在集合中，因此必须使用装箱基本类型。在参数化类型中，必须使用装箱基本类型作为类型参数，因为Java不允许使用基本类型。例如，你不能将变量声明为ThreadLocal<int>类型，因此必须使用ThreadLocal<Integer>代替。最后，在进行反射的方法调用时，必须使用装箱基本类型。

第51条：当心字符串连接的性能

1. 为了获得可以接受的性能，请使用StringBuilder替代String，来存储建造中的对账单。（Java 1.5发行版本中增加了非同步StringBuilder类，代替了现在已经过时的StringBuffer类。）：

public String statement() {

StringBuilder b = new StringBuilder(numItems() \* LINE\_WIDTH);

for (int i=0; i<numItems(); i++)

b.append(lineForItem(i));

return b.toString();

}

第52条：通过接口引用对象

1. 更一般地讲，应该优先使用接口而不是类来引用对象。如果有合适的接口类型存在，那么对于参数、返回值、变量和域来说，就都应该使用接口类型进行声明。
2. 有一点值得注意：如果原来的实现提供了某种特殊的功能，而这种功能并不是这个接口的通用约定所要求的，并且周围的代码又依赖于这种功能，那么很关键的一点是，新的实现也要提供同样的功能。例如，如果第一个声明周围的代码依赖于Vector的同步策略，在声明中用ArrayList代替Vector就是不正确的。如果依赖于实现的任何特殊属性，就要在声明变量的地方给这些需求建立相应的文档说明。
3. 那么，为什么要改变实现呢？因为新的实现提供了更好的性能，或者因为它提供了期望得到的额外功能。有个真实的例子与ThreadLocal类有关。在内部，这个类在Thread中使用了一个包级私有的Map域，将每个线程的值（per-thread values）与ThreadLocal实例关联起来。在1.3发行版本中，这个域被初始化为HashMap实例。在1.4发行版本中，Java平台增加了一个新的、被称为IdentityHashMap的专用Map实现。只需将初始化域的那一行代码改变为IdentityHashMap，代替原来的HashMap，ThreadLocal机制就会变快许多。
4. 如果没有合适的接口存在，完全可以用类而不是接口来引用对象。例如，考虑值类（value class），比如String和BigInteger。记住，值类很少会用多个实现编写。它们通常是final的，并且很少有对应的接口。使用这种值类作为参数、变量、域或者返回类型是再合适不过的了。更一般地讲，如果具体类没有相关联的接口，不管它是否表示一个值，你都没有别的选择，只有通过它的类来引用它的对象。Random类就属于这种情形。

第53条：接口优先于反射机制

1. 执行反射访问所需要的代码非常笨拙和冗长。编写这样的代码非常乏味，阅读起来也很困难。
2. 核心反射机制最初是为了基于组件的应用创建工具而设计的。这类工具通常要根据需要装载类，并且用反射功能找出它们支持哪些方法和构造器。这些工具允许用户交互式地构建出访问这些类的应用程序，但是所产生出来的这些应用程序能够以正常的方式访问这些类，而不是以反射的方式。反射功能只是在设计时（design time）被用到。通常，普通应用程序在运行时不应该以反射方式访问对象。
3. 有一些复杂的应用程序需要使用反射机制。这些示例包括类浏览器、对象监视器、代码分析工具、解释型的内嵌式系统。在RPC（远程过程调用）系统中使用反射机制也是非常合适的，这样可以不再需要存根编译器（stub compiler）。如果你对自己的应用程序是否也属于这一类应用程序而感到怀疑，它很有可能就不属于这一类。
4. 第一，这个例子会产生3个运行时错误，如果不使用反射方式的实例化，这3个错误都会成为编译时错误。第二，根据类名生成它的实例需要20行冗长的代码，而调用一个构造器可以非常简洁地只使用一行代码。

// Translate the class name into a Class object

Class<?> cl = null;

try {

cl = Class.forName(args[0]);

} catch(ClassNotFoundException e) {

System.err.println("Class not found.");

System.exit(1);

}

// Instantiate the class

Set<String> s = null;

try {

s = (Set<String>) cl.newInstance();

} catch(IllegalAccessException e) {

System.err.println("Class not accessible.");

System.exit(1);

} catch(InstantiationException e) {

System.err.println("Class not instantiable.");

System.exit(1);

}

如果要编写一个包，并且它运行的时候必须依赖其他某个包的多个版本，这种做法可能就非常有用。这种做法就是，在支持包所需要的最小环境下对它进行编译，通常是最老的版本，然后以反射方式访问任何更加新的类或者方法。

如果你编写的程序必须要与编译时未知的类一起工作，如有可能，就应该仅仅使用反射机制来实例化对象，而访问对象时则使用编译时已知的某个接口或者超类。

第54条：谨慎地使用本地方法

1. 从历史上看，本地方法主要有三种用途。它们提供了“访问特定于平台的机制”的能力，比如访问注册表（registry）和文件锁（file lock）。它们还提供了访问遗留代码库的能力，从而可以访问遗留数据（legacy data）。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，以提高系统的性能。
2. 使用本地方法来访问特定于平台的机制是合法的，但是随着Java平台的不断成熟，它提供了越来越多以前只有在宿主平台上才拥有的特性。例如，1.4发行版本中新增加的java.util.prefs包，提供了注册表的功能，1.6发行版本中增加了java.awt.SystemTray，提供了访问桌面系统托盘区的能力。
3. **使用本地方法来提供性能的做法不值得提倡。**在早期的发行版本中（1.3发行版本之前），这样做往往是很有必要的，但是JVM实现变得越来越快了。对于大多数任务，现在即使不使用本地方法也可以获得与之相当的性能。举例来说，当Java 1.1发行版本中增加了java.math时，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1.3发行版本中，BigInteger则完全用Java重写了，并且进行了精心的性能调优。即便如此，新的版本还是比原来的版本更快，在这些年间，VM也已经变得更快了。
4. 因为本地语言不是安全的，所以，使用本地方法的应用程序也不再能免受内存毁坏错误的影响。

第55条：谨慎地进行优化

1. 它们讲述了一个关于优化的深刻真理：优化的弊大于利，特别是不成熟的优化。
2. 不要因为性能而牺牲合理的结构。要努力编写好的程序而不是快的程序。如果好的程序不够快，它的结构将使它可以得到优化。好的程序体现了信息隐藏（information hiding）的原则：只要有可能，它们就会把设计决策集中在单个模块中，因此，可以改变单个决策，而不会影响到系统的其他部分。
3. 实现上的问题可以通过后期的优化而得到修正，但是，遍布全局并且限制性能的结构缺陷几乎是不可能被改正的，除非重新编写系统。
4. 当一个系统设计完成之后，其中最难以更改的组件是那些指定了模块之间交互关系以及模块与外界交互关系的组件。在这些设计组件之中，最主要的是API、线路层（wire-level）协议以及永久数据格式。这些设计组件不仅在事后难以甚至不可能改变，而且它们都有可能对系统本该达到的性能产生严重的限制。
5. 要考虑API设计决策的性能后果。使公有的类型成为可变的（mutable），这可能会导致大量不必要的保护性拷贝。同样地，在适合使用复合模式的公有类中使用继承，会把这个类与它的超类永远地束缚在一起，从而人为地限制了子类的性能。最后一个例子，在API中使用实现类型而不是接口，会把你束缚在一个具体的实现上，即使将来出现更快的实现你也无法使用。
6. API设计对于性能的影响是非常实际的。考虑java.awt.Component类中的getSize方法。这个决定就是，这个注重性能的方法返回Dimension实例，与此密切相关的决定是，Dimension实例是可变的，迫使这个方法的任何实现都必须为每个调用分配一个新的Dimension实例。尽管在现代VM上分配小对象的开销并不大，但是分配数百万个不必要的对象仍然会严重地损害性能。
7. 在Java平台上对优化的结果进行测量，比在其他的传统平台上更有必要，因为Java程序设计语言没有很强的性能模型（performance model）。各种基本操作的相对开销也没有明确定义。程序员所编写的代码与CPU执行的代码之间存在“语义沟（semantic gap）”，而且这条语义沟比传统编译语言中的更大，这使得要想可靠地预测出任何优化的性能结果都非常困难。大量流传的关于性能的说法最终都被证明为半真半假，或者根本就不正确。
8. 第一个步骤是检查所选择的算法：再多的低层优化也无法弥补算法的选择不当。

第56条：遵守普遍接受的命名惯例

1. 常量域是个静态final域，它的值是不可变的。如果静态final域有基本类型，或者有不可变的引用类型，它就是个常量域。例如，枚举常量是常量域。如果静态final域有个可变的引用类型，若被引用的对象是不可变的，它也仍然可以是个常量域。注意，常量域是唯一推荐使用下划线的情形。
2. 类型参数名称通常由单个字母组成。这个字母通常是以下五种类型之一：T表示任意的类型，E表示集合的元素类型，K和V表示映射的键和值类型，X表示异常。任何类型的序列可以是T、U、V或者T1、T2、T3。
3. 转换对象类型的方法、返回不同类型的独立对象的方法，通常被称为toType，例如toString和toArray。返回视图（view，视图的类型不同于接收对象的类型）的方法通常被称为asType，例如asList。返回一个与被调用对象同值得基本类型的方法，通常被称为typeValue，例如intValue。静态工厂的常用名称为valueOf、of、getInstance、newInstance、getType和NewType。

第9章 异常

第57条：只针对异常的情况才使用异常

1. 因为异常机制的设计初衷是用于不正常的情形，所以很少会有JVM实现试图对它们进行优化，使得与显式的测试一样快速。
2. 把代码放在try-catch块中反而阻止了现代JVM实现本来可能要执行的某些特定优化。
3. 对数组进行遍历的标准模式并不会导致冗余的检查。有些现代的JVM实现会将它们优化掉。
4. 设计良好的API不应该强迫它的客户端为了正常的控制流而使用异常。如果类具有“状态相关（state-dependent）”的方法，即只有在特定的不可预知的条件下才可以被调用的方法，这个类往往也应该有个单独的“状态测试（state-testing）”方法，即指示是否可以调用这个状态相关的方法。例如，Iterator接口有一个“状态相关”的next方法，和相应的状态测试方法hasNext。
5. 另一种提供单独的状态测试方法的做法是，如果“状态相关的”方法被调用时，该对象处于不适当的状态之中，它就会返回一个可识别的值，比如null。
6. 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，使用可被识别的返回值可能是很有必要的，因为在调用“状态测试”方法和调用对应的“状态相关”方法的时间间隔之中，对象的状态有可能会发生变化。如果单独的“状态测试”方法必须重复“状态相关”方法的工作，从性能的角度考虑，就应该使用可被识别的返回值。如果所有其他方面都是等同的，那么“状态测试”方法则略优于可被识别的返回值。它提供了更好的可读性，对于使用不当的情形，可能更加易于检测和改正：如果忘了去调用状态测试方法，状态相关的方法就会抛出异常，使这个Bug变得很明显；如果忘了去检查可识别的返回值，这个Bug就很难会被发现。

第58条：对可恢复的情况使用受检异常，对编程错误使用运行时异常

1. 在决定使用受检的异常或是未受检的异常时，主要的原则是：如果期望调用者能够适当地恢复，对于这种情况就应该使用受检的异常。通过抛出受检的异常，强迫调用者在一个catch子句中处理该异常，或者将它传播出去。因此，方法中声明要抛出的每个受检的异常，都是对API用户的一种潜在指示：与异常相关联的条件是调用这个方法的一种可能的结果。
2. 有两种未受检的可抛出结构：运行时异常和错误。在行为上两者是等同的：它们都是不需要也不应该被捕获的可抛出结构。如果程序抛出未受检的异常或者错误，往往就属于不可恢复的情形，继续执行下去有害无益。如果程序没有捕捉到这样的可抛出结构，将会导致当前线程停止（halt），并出现适当的错误消息。
3. 虽然JLS（Java语言规范）并没有要求，但是按照惯例，错误往往被JVM保留用于表示资源不足、约束失败，或者其他使程序无法继续执行的条件。由于这已经是个几乎被普遍接受的惯例，因此最好不要再实现任何新的Error子类。因此，你实现的所有未受检的抛出结构都应该是RuntimeException的子类（直接的或者间接的）。
4. 因为受检的异常往往指明了可恢复的条件，所以，对于这样的异常，提供一些辅助方法尤其重要，通过这些方法，调用者可以获得一些有助于恢复的信息。例如，假设因为用户没有储存足够数量的钱，他企图在一个收费电话上进行呼叫就会失败，于是抛出受检的异常。这个异常应该提供一个访问方法，以便允许客户查询所缺的费用金额，从而可以将这个数值传递给电话用户。

第59条：避免不必要地使用受检的异常

1. 如果正确地使用API并不能阻止这种异常条件的产生，并且一旦产生异常，使用API的程序员可以立即采取有用的动作，这种负担就被认为是正当的。除非这两个条件都成立，否则更适合于使用未受检的异常。
2. 如果使用API的程序员无法做得比这更好，那么未受检的异常可能更为合适。这种例子就是CloneNotSupportedException。它是被Object.clone抛出来的，而Object.clone应该只是在实现了Cloneable的对象上才可以被调用。在实践中，catch块几乎总是具有断言（assertion）失败的特征。异常受检的本质并没有为程序员提供任何好处，它反而需要付出努力，还使程序更为复杂。

第60条：优先使用标准的异常

1. 最经常被重用的异常是IllegalArgumentException。当调用者传递的参数值不合适的时候，往往就会抛出这个异常。例如，假设一个参数代表了“某个动作的重复次数”，如果程序员给这个参数传递了一个负数，就会抛出这个异常。
2. 另一个经常被重用的异常是IllegalStateException。如果因为接收对象的状态而使调用非法，通常就会抛出这个异常。例如，如果在某个对象被正确地初始化之前，调用者就企图使用这个对象，就会抛出这个异常。
3. 可以这么说，所有错误的方法调用都可以被归结为非法参数或者非法状态，但是，其他还有一些标准异常也被用于某些特定情况下的非法参数和非法状态。如果调用者在某个不允许null值的参数中传递了null，习惯的做法就是抛出NullPointerException，而不是IllegalArgumentException。同样地，如果调用者在表示序列下标的参数中传递了越界的值，应该抛出的就是IndexOutOfBoundsException，而不是IllegalArgumentException。
4. 另一个值得了解的通用异常是ConcurrentModificationException。如果一个对象被设计为专用于单线程或者与外部同步机制配合使用，一旦发现它正在（或已经）被并发地修改，就应该抛出这个异常。
5. 最后一个值得注意的通用异常是UnsupportedOperationException。如果对象不支持所请求的操作，就会抛出这个异常。与本条目中讨论的其他异常相比，它很少用到，因为绝大多数对象都会支持它们实现的所有方法。如果接口的具体实现没有实现该接口所定义的一个或者多个可选操作，它就可以使用这个异常。
6. 例如，如果要实现诸如复数或者有理数之类的算术对象，也可以重用ArithmeticException和NumberFormatException。
7. 这种重用必须建立在语义的基础上，而不是建立在名称的基础之上。而且，如果希望稍微增加更多的失败-捕获（failure-capture）信息，可以放心地把现有的异常进行子类化。

第61条：抛出与抽象相对应的异常

1. 如果方法抛出的异常与它所执行的任务没有明显的联系，这种情形将会使人不知所措。当方法传递由低层抽象抛出的异常时，往往会发生这种情况。除了使人感到困惑之外，这也让实现细节污染了更高层的API。
2. 为了避免这个问题，更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解释的异常。这种做法被称为异常转译（exception translation），如下所示：

// Exception Translation

try {

// Use lower-level abstraction to do our bidding

...

} catch(LowerLevelException e) {

throw new HigherLevelException(...);

}

一种特殊的异常转译形式称为异常链（exception chaining），如果低层的异常对于调试导致高层异常的问题非常有帮助，使用异常链就很合适。低层的异常（原因）被传到高层的异常，高层的异常提供访问方法（Throwable.getCause）来获得低层的异常：

// Exception Chaining

try {

... // Use lower-level abstraction to do our bidding

} catch(LowerLevelException cause) {

throw new HigherLevelException(cause);

}

如有可能，处理来自低层异常的最好做法是，在调用低层方法之前确保它们会成功执行，从而避免它们抛出异常。有时候，可以在给低层传递参数之前，检查更高层方法的参数的有效性，从而避免低层方法抛出异常。

如果无法避免低层异常，次选方案是，让更高层来悄悄地绕开这些异常，从而将高层方法的调用者与低层的问题隔离开来。在这种情况下，可以用某种适当的记录机制（如java.util.logging）将异常记录下来。这样有助于管理员调查问题，同时又将客户端代码和最终用户与问题隔离开来。

第62条：每个方法抛出的异常都要有文档

1. 总而言之，要为你编写的每个方法所能抛出的每个异常建立文档。对于未受检和受检的异常，以及对于抽象的和具体的方法也都一样。要为每个受检异常提供单独的throws子句，不要为未受检的异常提供throws子句。

第63条：在细节消息中包含能捕获失败的信息

1. 虽然在异常的细节消息中包含所有相关的“硬数据（hard data）”是非常重要的，但是包含大量的描述信息往往没有什么意义。
2. 异常的细节消息不应该与“用户层次的错误消息”混为一谈，后者对于最终用户而言必须是可理解的。与用户层次的错误消息不同，异常的字符串表示法主要是让程序员或者域服务人员用来分析失败的原因。因此，信息的内容比可理解性要重要得多。
3. 为了确保在异常的细节消息中包含足够的能捕获失败的信息，一种办法是在异常的构造器而不是字符串细节消息中引入这些信息。然后，有了这些信息，只要把它们放到消息描述中，就可以自动产生细节消息。例如，IndexOutOfBoundsException并不是有个String构造器，而是有个这样的构造器：

public IndexOutOfBoundsException(int lowerBound, int upperBound, int index) {

// Generate a detail message that captures the failure

super("Lower bound: " + lowerBound +

", Upper bound: " + upperBound +

", Index: " + index);

// Save failure information for programmatic access

this.lowerBound = lowerBound;

this.upperBound = upperBound;

this.index = index;

}

第64条：努力使失败保持原子性

1. 一般而言，失败的方法调用应该使对象保持在被调用之前的状态。具有这种属性的方法被称为具有失败原子性（failure atomic）。
2. 对于在可变对象上执行操作的方法，获得失败原子性最常见的办法是，在执行操作之前检查参数的有效性。这可以使得在对象的状态被修改之前，先抛出适当的异常。例如，Stack.pop方法：

public Object pop() {

if (size==0)

throw new EmptyStackException();

Object result = elements[--size];

elements[size] = null; // Eliminate obsolete reference

return result;

}

虽然一般情况下都希望实现失败原子性，但并非总是可以做到。例如，如果两个线程企图在没有适当的同步机制的情况下，并发地修改同一个对象，这个对象就有可能被留在不一致的状态之中。因此，在捕获了ConcurrentModificationException异常之后再假设对象仍然是可用的，这就是不正确的。错误（相对于异常）通常是不可恢复的，当方法抛出错误时，它们不需要努力保持失败原子性。

第65条：不要忽略异常

第10章 并发

第66条：同步访问共享的可变数据

1. 如果没有同步，一个线程的变化就不能被其他线程看到。同步不仅可以阻止一个线程看到对象处于不一致的状态之中，它还可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护的之前所有的修改效果。
2. 你可能听说过，为了提高性能，在读或写原子数据的时候，应该避免使用同步。这个建议是非常危险而错误的。虽然语言规范保证了线程在读取原子数据的时候，不会看到任意的数值，但是它并不保证一个线程写入的值对于另一个线程将是可见的。为了在线程之间进行可靠的通信，也为了互斥访问，同步是必要的。这归因于Java语言规范中的内存模型（memory model），它规定了一个线程所做的变化何时以及如何变成对其他线程可见。

// 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(new Runnable() {

public void run() {

int i=0;

while(!stopRequested)

i++;

}

});

backgroundThread.start();

TimeUnit.SECONDS.sleep(1);

stopRequested = true;

}

}

问题在于，由于没有同步，就不能保证后台线程何时“看到”主线程对stopRequested的值所做的改变。没有同步，虚拟机将这个代码：

while(!done)

i++;

转变成这样：

if (!done)

while(true)

i++;

这是可以接受的。这种优化称作提升（hoisting），正是HotSpot Server VM的工作。结果是个活性失败（liveness failure）：这个程序无法前进。修正这个问题的一种方式是同步访问stopRequested域。这个程序会如预期般在大约一秒钟之内终止：

// Properly synchronized cooperative thread termination

public class StopThread {

private static boolean stopRequested;

private static synchronized void requestStop() {

stopRequested = true;

}

private static synchronized boolean stopRequested() {

return stopRequested;

}

public static void main(String[] args) throws InterruptedException {

Thread backgroundThread = new Thread(new Runnable() {

public void run() {

int i=0;

while(!stopRequested())

i++;

}

});

backgroundThread.start();

TimeUnit.SECONDS.sleep(1);

requestStop();

}

}

StopThread中被同步方法的动作即使没有同步也是原子的。换句话说，这些方法的同步只是为了它的通信效果。

如果stopRequested被声明为volatile，第二种版本的StopThread中的锁就可以省略。虽然**volatile修饰符不执行互斥访问**，但它可以保证任何一个线程在读取该域的时候都将看到最近刚刚被写入的值：

// 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(new Runnable() {

public void run() {

int i=0;

while(!stopRequested)

i++;

}

});

backgroundThread.start();

TimeUnit.SECONDS.sleep(1);

stopRequested = true;

}

}

最好还是遵循第47条中的建议，使用类AtomicLong，它是java.util.concurrent.atomic的一部分。它所做的工作正是你想要的，并且有可能比同步版的generateSerialNumber执行得更好：

private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {

return nextSerialNum.getAndIncrement();

}

让一个线程在短时间内修改一个数据对象，然后与其他线程共享，这是可以接受的，只同步共享对象引用的动作。然后其他线程没有进一步的同步也可以读取对象，只要它没有再被修改。这种对象称作事实上不可变的（effectively immutable）。将这种对象引用从一个线程传递到其他的线程被称作安全发布（safe publication）。安全发布对象引用有许多种方法：可以将它保存在静态域中，作为类初始化的一部分；可以将它保存在volatile域、final域或者通过正常锁定访问的域中；或者可以将它放进并发的集合中。

如果只需要线程之间的交互通信，而不需要互斥，volatile修饰符就是一种可以接受的同步形式，但要正确地使用它可能需要一些技巧。

第67条：避免过度同步

1. 换句话说，在一个被同步的区域内部，不要调用设计成要被覆盖的方法，或者是由客户端以函数对象的形式提供的方法。从包含该同步区域的类的角度来看，这样的方法是外来的（alien）。这个类不知道该方法会做什么事情，也无法控制它。根据外来方法的作用，从同步区域中调用它会导致异常、死锁或者数据损坏。
2. notifyElementAdded方法中的迭代是在一个同步的块中，可以防止并发的修改，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。
3. 从同步区域中调用外来方法，在真实的系统中已经造成了许多死锁，例如GUI工具箱。
4. 幸运的是，通过将外来方法的调用移出同步的代码块来解决这个问题通常并不太困难。对于notifyElementAdded方法，这还涉及给observers列表拍张“快照”，然后没有锁也可以安全地遍历这个列表了。经过这一修改，前两个例子运行起来便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<SetObserver<E>>(observers);

}

for (SetObserver<E> observer : snapshot)

observer.added(this, element);

}

事实上，要将外来方法的调用移出同步的代码块，还有一种更好的方法。自从Java 1.5发行版本以来，Java类库就提供了一个并发集合（concurrent collection），称作CopyOnWriteArrayList，这是专门为此定制的。这是ArrayList的一种变体，通过重新拷贝整个底层数组，在这里实现所有的写操作。由于内部数组永远不改变，因此迭代不需要锁定，速度也非常快。如果大量使用，CopyOnWriteArrayList的性能将大受影响，但是对于观察者列表来说却是很好的，因为它们几乎不改动，并且经常被遍历。

// Thread-safe observable set with CopyOnWriteArrayList

private final List<SetObserver<E>> observers =

new CopyOnWriteArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {

observers.add(observer);

}

public boolean removeObserver(SerObserver<E> observer) {

return observers.remove(observer);

}

private void notifyElementAdded(E element) {

for (SetObserver<E> observer : observers)

observer.added(this, element);

}

在这个多核的时代，过度同步的实际成本并不是指获取锁所花费的CPU时间，而是指失去了并行的机会，以及因为需要确保每个核都有一个一致的内存视图而导致的延迟。过度同步的另一项潜在开销在于，它会限制VM优化代码执行的能力。

**如果一个可变的类要并发使用，应该使这个类变成是线程安全的，通过内部同步，你还可以获得明显比从外部锁定整个对象更高的并发性。否则，就不要在内部同步。让客户在必要的时候从外部同步。在Java平台出现的早期，许多类都违背了这些指导方针。例如，StringBuffer实例几乎总是被用于单个线程之中，而它们执行的却是内部同步。为此，StringBuffer基本上都由StringBuilder代替，它在Java 1.5发行版本中是个非同步的StringBuffer。当你不确定的时候，就不要同步你的类，而是应该建立文档，注明它不是线程安全的。**

如果你在内部同步了类，就可以使用不同的方法来实现高并发性，例如分拆锁（lock splitting）、分离锁（lock striping）和非阻塞（nonblocking）并发控制。

第68条：executor和task优先于线程

1. 在Java 1.5发行版本中，Java平台中增加了java.util.concurrent。这个包中包含了一个Executor Framework，这是一个很灵活的基于接口的任务执行工具。它创建了一个在各方面都比本书第一版更好的工作队列，却只需要这一行代码：

ExecutorService executor = Executors.newSingleThreadExecutor();

下面是为执行提交一个runnable的方法：

executor.execute(runnable);

下面是告诉executor如何优雅地终止（如果做不到这一点，虚拟机可能将不会退出）：

executor.shutdown();

例如，可以等待完成一项特殊的任务，你可以等待一个任务集合中的任何任务或者所有任务完成（利用invokeAny或者invokeAll方法），你可以等待executor service优雅地完成终止（利用awaitTermination方法），你可以在任务完成时逐个地获取这些任务的结果（利用ExecutorCompletionService），等等。

如果编写的是小程序，或者是轻载的服务器，使用Executors.newCachedThreadPool通常是个不错的选择，因为它不需要配置，并且一般情况下能够正确地完成工作。

因此，在大负载的产品服务器中，最好使用Executors.newFixedThreadPool，它为你提供了一个包含固定线程数目的线程池，或者为了最大限度地控制它，就直接使用ThreadPoolExecutor类。

**现在关键的抽象不再是Thread了，它以前可是既充当工作单元，又是执行机制。现在工作单元和执行机制是分开的。现在关键的抽象是工作单元，称作任务（task）。任务有两种：Runnable及其近亲Callable。执行任务的通用机制是executor service。**

从本质上讲，Executor Framework所做的工作是执行，犹如Collections Framework所做的工作室聚集（aggregation）一样。

Executor Framework也有一个可以代替java.util.Timer的东西，即ScheduledThreadPoolExecutor。虽然timer使用起来更加容易，但是被调度的线程池executor更加灵活。timer只用一个线程来执行任务，这在面对长期运行的任务时，会影响到定时的准确性。如果timer唯一的线程抛出未被捕获的异常，timer就会停止执行。被调度的线程池executor支持多个线程，并且优雅地从抛出未受检异常的任务中恢复。

第69条：并发工具优先于wait和notify

1. 这是因为几乎没有理由再使用wait和notify了。自从Java 1.5发行版本开始，Java平台就提供了更高级的并发工具，它们可以完成以前必须在wait和notify上手写代码来完成的各项工作。既然正确地使用wait和notify比较困难，就应该用更高级的并发工具来代替。
2. java.util.concurrent中更高级的工具分成三类：Executor Framework、并发集合（Concurrent Collection）以及同步器（Synchronizer）。
3. 因此有些集合接口已经通过依赖状态的修改操作（state-dependent modify operation）进行了扩展，它将几个基本操作合并到了单个原子操作中。例如，ConcurrentMap扩展了Map接口，并添加了几个方法，包括putIfAbsent(key, value)，当键没有映射时会替它插入一个映射，并返回与键关联的前一个值，如果没有这样的值，则返回null。例如，下面这个方法模拟了String.intern的行为（ConcurrentHashMap对获取操作（如get）进行了优化。因此，只有当get表明有必要的时候，才值得先调用get，再调用putIfAbsent）：

// Concurrent canonicalizing map atop ConcurrentMap - faster!

private static final ConcurrentMap<String, String> map =

new ConcurrentHashMap<String, String>();

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;

}

除非不得已，否则应该优先使用ConcurrentHashMap，而不是使用Collections.synchronizedMap或者Hashtable。

有些集合接口已经通过阻塞操作（blocking operation）进行了扩展，它们会一直等待（或者阻塞）到可以成功执行为止。例如，BlockingQueue扩展了Queue接口，并添加了包括take在内的方法，它从队列中删除并返回了头元素，如果队列为空，就等待。这样就允许将阻塞队列用于工作队列（work queue），也称作生产者-消费者队列（producer-consumer queue），一个或者多个生产者线程（producer thread）在工作队列中添加工作项目，并且当工作项目可用时，一个或者多个消费者线程（consumer thread）则从工作队列中取出并处理工作项目。不出所料，大多数ExecutorService实现（包括ThreadPoolExecutor）都使用BlockingQueue。

倒计数锁存器（Countdown Latch）是一次性的障碍，允许一个或者多个线程等待一个或者多个其他线程来做某些事情。CountDownLatch的唯一构造器带有一个int类型的参数，这个int参数是指允许所有在等待的线程被处理之前，必须在锁存器上调用countDown方法的次数。

// Simple framework for timing concurrent execution

public static long time(Executor executor, int concurrency,

final Runnable action) throws InterruptedException {

final CountDownLatch ready = new CountDownLatch(concurrency);

final CountDownLatch start = new CountDownLatch(1);

final CountDownLatch done = new CountDownLatch(concurrency);

for(int i=0; i<concurrency; i++) {

executor.execute(new Runnable() {

public void run() {

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;

}

wait方法被用来使线程等待某个对象。它必须在同步区域内部被调用，这个同步区域将对象锁定在了调用wait方法的对象上。下面是使用wait方法的标准模式：

// 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

}

始终应该使用wait循环模式来调用wait方法；永远不要在循环之外调用wait方法。循环会在等待之前和之后测试条件。

一种常见的说法是，你总是应该使用notifyAll。这是合理而保守的建议。它总会产生正确的结果，因为它可以保证你将会唤醒所有需要被唤醒的线程。你可能也会唤醒其他一些线程，但是这不会影响程序的正确性。这些线程醒来之后，会检查它们正在等待的条件，如果发现条件并不满足，就会继续等待。

就好像把wait调用放在一个循环中，以避免在公有可访问对象上的意外或恶意的通知一样，与此类似，使用notifyAll代替notify可以避免来自不相关线程的意外或恶意的等待。否则，这样的等待会“吞掉”一个关键的通知，使真正的接收线程无限地等待下去。

第70条：线程安全性的文档化

1. 当一个类的实例或者静态方法被并发使用的时候，这个类的行为如何，是该类与其客户端程序建立的约定的重要组成部分。
2. 无条件的线程安全（unconditionally thread-safe）——这个类的实例是可变的，但是这个类有着足够的内部同步，所以，它的实例可以被并发使用，无需任何外部同步。其例子包括Random和ConcurrentHashMap。
3. 有条件的线程安全（conditionally thread-safe）——除了有些方法为进行安全的并发使用而需要外部同步之外，这种线程安全级别与无条件的线程安全相同。这样的例子包括Collections.synchronized包装返回的集合，它们的迭代器（iterator）要求外部同步。
4. 没有必要说明枚举类型的不可变性。除非从返回类型来看已经很明显，否则静态工厂必须在文档中说明被返回对象的线程安全性，如Collections.synchronizedMap所示。
5. 当一个类承诺了“使用一个公有可访问的锁对象”时，就意味着允许客户端以原子的方式执行一个方法调用序列，但是，这种灵活性是要付出代价的。并发集合（如ConcurrentHashMap和ConcurrentLinkedQueue）使用的那种并发控制，并不能与高性能的内部并发控制相兼容。客户端还可以发起拒绝服务攻击，他只需超时地保持公有可访问锁即可。
6. 为了避免这种拒绝服务攻击，应该使用一个私有锁对象（private lock object）来代替同步的方法（隐含着一个公有可访问锁）：

\\ Private lock object idiom - thwarts denial-of-service attack

private final Object lock = new Object();

public void foo() {

synchronized(lock) {

...

}

}

因为这个私有锁对象不能被这个类的客户端程序所访问，所以它们不可能妨碍对象的同步。

如果你编写的是无条件的线程安全类，就应该考虑使用私有锁对象来代替同步的方法。这样可以防止客户端程序和子类的不同步干扰，让你能够在后续的版本中灵活地并发控制采用更加复杂的方法。

第71条：慎用延迟初始化

1. 延迟初始化就像一把双刃剑。它降低了初始化类或者创建实例的开销，却增加了访问被延迟初始化的域的开销。根据延迟初始化的域最终需要初始化的比例、初始化这些域要多少开销，以及每个域多久被访问一次，延迟初始化（就像其他的许多优化一样）实际上降低了性能。
2. **如果出于性能的考虑而需要对静态域使用延迟初始化，就使用lazy initialization holder class模式。**这种模式（也称作initialize-on-demand holder class idiom）保证了类要到被用到的时候才会被初始化。

// Lazy initialization holder class idiom for static fields

private static class FieldHolder {

static final FieldType field = computeFieldValue();

}

static FieldType getField() { return FieldHolder.field; }

* 1
* 2
* 3
* 4
* 5

当getField方法第一次被调用时，它第一次读取FieldHolder.field，导致FieldHolder类得到初始化。这种模式的魅力在于，getField方法没有被同步，并且只执行一个域访问，因此延迟初始化实际上并没有增加任何访问成本。现代的VM将在初始化该类的时候，同步域的访问。一旦这个类被初始化，VM将修补代码，以便后续对该域的访问不会导致任何测试或者同步。

**如果出于性能的考虑需要对实例域使用延迟初始化，就使用双重检查模式（double-check idiom）。这种模式避免了在域被初始化之后访问这个域时的锁定开销。**这种模式背后的思想是：两次检查域的值，第一次检查时没有锁定，看看这个域是否被初始化了；第二次检查时有锁定。只有当第二次检查时表明这个域没有被初始化，才会调用computeFieldValue方法对这个域进行初始化。因为如果域已经被初始化就不会有锁定，域被声明为volatile很重要。

// Double-check idiom for lazy initialization of instance fields

private volatile FieldType field;

FieldType getField() {

FieldType result = field;

if (result==null) { // First check (no locking)

synchronized(this) {

result = field;

if(result==null) // Second check (with locking)

field = result = computeFieldValue();

}

}

return result;

}

尤其对于需要用到局部变量result可能有点不解。这个变量的作用是确保field只在已经被初始化的情况下读取一次。（**我觉得像是快照的功能。**）

有时候，你可能需要延迟初始化一个可以接受重复初始化的实例域。如果处于这种情况，这可以使用双重检查惯用法的一个变形，它省去了第二次检查。没错，它就是单重检查模式（single-check idiom）。

如果你不在意是否每个线程都重新计算域的值，并且域的类型为基本类型，而不是long或者double类型，就可以选择从单重检查模式的域声明中删除volatile修饰符。这种变体称之为racy single-check idiom。它加快了某些架构上的域访问，代价是增加了额外的初始化（直到访问该域的每个线程都进行一次初始化）。这显然是一种特殊的方法，不适合日常的使用。然而，String实例却用它来缓存它们的散列码。

第72条：不要依赖于线程调度器

1. 要编写健壮的、响应良好的、可移植的多线程应用程序，最好的办法是确保可运行线程的平均数量不明显多于处理器的数量。
2. 如果某一个程序不能工作，是因为某些线程无法像其他线程那样获得足够的CPU时间，那么，不要企图通过调用Thread.yield来“修正”该程序。Thread.yield没有可测试的语义（testable semantic）。更好的解决办法是重新构造应用程序，以减少可并发运行的线程数量。
3. 有一种相关的方法是调整线程优先级，同样有类似的警告。线程优先级是Java平台上最不可移植的特征了。
4. 在本书第一版中说过，对于大多数程序员来说，Thread.yield的唯一用途是在测试期间人为地增加程序的并发性。应该使用Thread.sleep(1)代替Thread.yield来进行并发测试。

第73条：避免使用线程组

1. 线程组的初衷是作为一种隔离applet（小程序）的机制，当然是出于安全的考虑。但是它们从来没有真正履行这个承诺，它们的安全价值已经差到根本不在Java安全模型的标准工作中提及的地步。
2. 总而言之，线程组并没有提供太多有用的功能，而且它们提供的许多功能还都是有缺陷的。我们最好把线程组看作是一个不成功的试验，你可以忽略掉它们，就当它们根本不存在一样。如果你正在设计的一个类需要处理线程的逻辑组，或许就应该使用线程池executor。

第11章 序列化

序列化技术为远程通信提供了标准的线路级（wire-level）对象表示法，也为JavaBeans组件结构提供了标准的持久化数据格式。

第74条：谨慎地实现Serializable接口

1. 如果一个类实现了Serializable接口，它的字节流编码（或者说序列化形式）就变成了它的导出的API的一部分。一旦这个类被广泛使用，往往必须永远支持这种序列化形式，就好像你必须要支持导出的API的所有其他部分一样。如果你不努力设计一种自定义的序列化形式，而仅仅接受了默认的序列化形式，这种序列化形式将被永远地束缚在该类最初的内部表示法上。换句话说，如果你接受了默认的序列化形式，这个类中私有的和包级私有的实例域将都变成导出的API的一部分，这不符合“最低限度地访问域”的实践准则，从而它就失去了作为信息隐藏工具的有效性。
2. 序列化会使类的演变受到限制，这种限制的一个例子与流的唯一标识符（stream unique identifier）有关，通常它也被称为序列版本UID。每个可序列化的类都有一个唯一标识号与它相关联。如果你没有在一个名为serialVersionUID的私有静态final的long域中显式地指定该标识号，系统就会自动地根据这个类来调用一个复杂的运算过程，从而在运行时产生该标识号。这个自动产生的值会受到类名称、它所实现的接口的名称、以及所有公有的和受保护的成员的名称所影响。如果你通过任何方式改变了这些信息，比如，增加了一个不很重要的工具方法，自动产生的序列版本UID也会发生变化。因此，如果你没有声明一个显式的序列版本UID，兼容性将会遭到破坏，在运行时导致InvalidClassException异常。
3. 因为反序列化机制中没有显式的构造器，所以你很容易忘记要确保：反序列化过程必须也要保证所有“由真正的构造器建立起来的约束关系”，并且不允许攻击者访问正在构造过程中的对象的内部信息。
4. 根据经验，比如Date和BigInteger这样的值类应该实现Serializable，大多数的集合类也应该如此。代表活动实体的类，比如线程池（thread pool），一般不应该实现Serializable。
5. 为了继承而设计的类应该尽可能少地去实现Serializable接口，用户的接口也应该尽可能少地继承Serializable接口。
6. 在为了继承而设计的类中，真正实现了Serializable接口的有Throwable类、Component和HttpServlet抽象类。因为Throwable类实现了Serializable接口，所以RMI的异常可以从服务器端传到客户端。Component实现了Serializable接口，因此GUI可以被发送、保存和恢复。HttpServlet实现了Serializable接口，因此会话状态（session state）可以被缓存。
7. 下面的转换增加了一个受保护的无参构造器，和一个初始化方法。初始化方法与正常的构造器具有相同的参数，并且也建立起同样的约束关系。注意保存对象状态（x和y）的变量不能是final的，因为它们是由initialize方法设置的：

// Nonserializable stateful class allowing serializable subclass

public abstract class AbstractFoo {

private int x, y; // Our state

// This enum and field are used to track initialization

private enum State { NEW, INITIALIZING, INITIALIZED };

private final AtomicReference<State> init = new AtomicReference<State>(State.NEW);

public AbstractFoo(int x, int y) { initialize(x, y); }

// This constructor and the following method allow subclass's

// readObject method to initialize our state.

protected AbstractFoo() { }

protected final void initialize(int x, int y) {

if(!init.compareAndSet(State.NEW, State.INITIALIZING))

throw new IllegalStateException("Already initialized");

this.x = x;

this.y = y;

... // Do anything else the original constructor did

init.set(State.INITIALIZED);

}

// These methods provide access to internal state so it can

// be manually serialized by subclass's writeObject method.

protected final int getX() { checkInit(); return x; }

protected final int getY() { checkInit(); return y; }

// Must call from all public and protected instance methods

private void checkInit() {

if(init.get()!=State.INITIALIZED)

throw new IllegalStateException("Uninitialized");

}

... // Remainder omitted

}

AbstractFoo中所有公有的和受保护的实例方法在开始做任何其他工作之前都必须先调用checkInit。这样可以确保如果有编写不好的子类没有初始化实例，该方法调用就可以快速而干净地失败。注意init域是一个原子引用（atomic reference）。在遇到特定的情况时，确保对象的完整性是很有必要的。如果没有这样的防范机制，万一有个线程要在某一个实例上调用initialize，而另一个线程又要企图使用这个实例，第二个线程就有可能看到这个实例处于不一致的状态。这种模式利用compareAndSet方法来操作枚举的原子引用，这是一个很好的线程安全状态机（thread-safe state machine）的通用实现。如果有了这样的机制做保证，实现一个可序列化的子类就非常简单明了：

// Serializable subclass of nonserializable stateful class

public class Foo extends AbstractFoo implements Seriablizable {

private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {

s.defaultReadObject();

// Manually deserialize and initialize superclass state

int x = s.readInt();

int y = s.readInt();

initialize(x, y);

}

private void writeObject(ObjectOutputStream s) throws IOException {

s.defaulteWriteObject();

// Manually serialize superclass state

s.writeInt(getX());

s.writeInt(getY());

}

// Constructor does not use the fancy mechanism

public Foo(int x, int y) { super(x, y); }

private static final long serialVersionUID = 1856835860954L;

}

内部类（inner class）不应该实现Serializable。它们使用编译器产生的合成域（synthetic field）来保存指向外围实例（enclosing instance）的引用，以及保存来自外围作用域的局部变量的值。“这些域如何对应到类定义中”并没有明确的规定，就好像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定义不清楚的。然而，静态成员类（static member class）却可以实现Serializable接口。

第75条：考虑使用自定义的序列化形式

1. 换句话说，默认的序列化形式描述了该对象内部所包含的数据，以及每一个可以从这个对象到达的其他对象的内部数据。它也描述了所有这些对象被链接起来后的拓扑结构。对于一个对象来说，理想的序列化形式应该只包含该对象所表示的逻辑数据，而逻辑数据与物理表示法应该是各自独立的。
2. 如果一个对象的物理表示法等同于它的逻辑内容，可能就适合于使用默认的序列化形式。
3. 即使你确定了默认的序列化形式是合适的，通常还必须提供一个readObject方法以保证约束关系和安全性。

// 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

}

从逻辑意义上讲，这个类表示了一个字符串序列。但是从物理意义上讲，它把该序列表示成一个双向链表。如果你接受了默认的序列化形式，该序列化形式将不遗余力地镜像出（mirror）链表中的所有项，以及这些项之间的所有双向链接。

它使这个类的导出API永远地束缚在该类的内部表示法上。在上面的例子中，私有的String.Entry类变成了公有API的一部分。如果在将来的版本中，内部表示法发生了变化，StringList类仍将需要接受链表形式的输入，并产生链表形式的输出。

它会消耗过多的空间。在上面的例子中，序列化形式既表示了链表中的每个项，也表示了所有的链接关系，这是不必要的。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。

对于StringList类，合理的序列化形式可以非常简单，只需先包含链表中字符串的数目，然后紧跟着这些字符串即可。这样就构成了StringList所表示的逻辑数据，与它的物理表示细节脱离。

// 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) { ... }

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

}

如果某一个实例将在未来的版本中被序列化，然后在前一个版本中被反序列化，那么，后增加的域将被忽略掉。如果旧版本的readObject方法没有调用defaultReadObject，反序列化过程将失败，引发StreamCorruptedException异常。

因此，每一个可以被标记为transient的实例域都应该做上这样的标记。这包括那些冗余的域，即它们的值可以根据其他“基本数据域”计算而得到的域，比如缓存起来的散列值。它也包括那些“其值依赖于JVM的某一次运行”的域，比如一个long域代表了一个指向本地数据结构的指针。

如果你正在使用一种自定义的序列化形式，大多数实例域，或者所有的实例域则都应该被标记为transient，就像上面例子中的StringList那样。

只有当默认的序列化形式能够合理地描述对象的逻辑状态时，才能使用默认的序列化形式；否则就要设计一个自定义的序列化形式，通过它合理地描述对象的状态。

第76条：保护性地编写readObject方法

1. 为了修正这个问题，你可以为Period提供一个readObject方法，该方法首先调用defaultReadObject，然后检查被反序列化之后的对象的有效性。如果有效性检查失败，readObject方法就抛出一个InvalidObjectException异常，使反序列化过程不能成功地完成。
2. 当一个对象被反序列化的时候，对于客户端不应该拥有的对象引用，如果哪个域包含了这样的对象引用，就必须要做保护性拷贝，这是非常重要的。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readObject方法中，必须要对这些组件进行保护性拷贝。
3. 有一个简单的“石蕊”测试，可以用来确定默认的readObject方法是否可以被接受。测试方法：增加一个公有的构造器，其参数对应于该对象中每个非transient的域，并且无论参数的值是什么，都是不进行检查就可以保存到相应的域中的。对于这样的做法，你是否会感到很舒适？如果你对这个问题的回答是否定的，就必须提供一个显式的readObject方法，并且它必须执行构造器所要求的所有有效性检查和保护性拷贝。另一种方法是，可以使用序列化代理模式（serialization proxy pattern）。
4. 对于对象引用域必须保持为私有的类，要保护性地拷贝这些域中的每个对象。不可变类的可变组件就属于这一类别。
5. 对于任何约束条件，如果检查失败，则抛出一个InvalidObjectException异常。这些检查动作应该跟在所有的保护性拷贝之后。

第77条：对于实例控制，枚举类型优先于readResolve

public class Elvis {

public static final Elvis INSTANCE = new Elvis();

private Elvis() { ... }

public void leaveTheBuilding() { ... }

}

如果这个类的声明中加上了“implements Serializable”的字样，它就不再是一个Singleton。无论该类使用了默认的序列化形式，还是自定义的序列化形式，都没有关系；也跟它是否提供了显式的readObject方法无关。任何一个readObject方法，不管是显式的还是默认的，它都会返回一个新建的实例，这个新建的实例不同于该类初始化时创建的实例。

事实上，如果依赖readResolve进行实例控制，带有对象引用类型的所有实例域都必须声明为transient的。否则，那种破釜沉舟式的攻击者，就有可能在readResolve方法被运行之前，保护指向反序列化对象的引用。

如果反过来，你将一个可序列化的实例受控的类编写成枚举，就可以绝对保证除了所声明的常量之外，不会有别的实例。JVM对此提供了保障，这一点你可以确信无疑。从你这方面来讲，并不需要特别注意什么。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));

}

}

总而言之，你应该尽可能地使用枚举类型来实施实例控制的约束条件。如果做不到，同时又需要一个既可序列化又是实例受控（instance-controlled）的类，就必须提供一个readResolve方法，并确保该类的所有实例域都为基本类型，或者是transient的。

第78条：考虑用序列化代理代替序列化实例

1. 序列化代理模式相当简单。首先，为可序列化的类设计一个私有的静态嵌套类，精确地表示外围类的实例的逻辑状态。这个嵌套类被称作序列化代理（serialization proxy），它应该有一个单独的构造器，其参数类型就是那个外围类。这个构造器只从它的参数中复制数据：它不需要进行任何一致性检查或者保护性拷贝。从设计的角度来看，序列化代理的默认序列化形式是外围类最好的序列化形式。外围类及其序列代理都必须声明实现Serializable接口。

// 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

}

接下来，将下面的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);

}

有了这个writeReplace方法之后，序列化系统永远不会产生外围类的序列化实例，但是攻击者有可能伪造，企图违反该类的约束条件。为了确保这种攻击无法得逞，只要在外围类中添加这个readObject方法即可：

// readObject method for the serialization proxy pattern

private void readObject(ObjectInputStream stream) throws InvalidObjectException {

throw new InvalidObjectException("Proxy required");

}

最后，在SerializationProxy类中提供一个readResolve方法，它返回一个逻辑上相当的外围类的实例。这个方法的出现，导致序列化系统在反序列化时将序列化代理转变回外围类的实例。

这个readResolve方法仅仅利用它的公有API创建外围类的一个实例，这正是该模式的魅力之所在。它极大地消除了序列化机制中语言本身之外的特征，因为反序列化实例是利用与任何其他实例相同的构造器、静态工厂和方法而创建的。这样你就不必单独确保被反序列化的实例一定要遵守类的约束条件。如果该类的静态工厂或者构造器建立了这些约束条件，并且它的实例方法在维持着这些约束条件，你就可以确信序列化也会维持这些约束条件。

// readResolve method for Period.SerializationProxy

private Object readResolve() {

return new Period(start, end); // Uses public constructor

}

总而言之，每当你发现自己必须在一个不能被客户端扩展的类上编写readObject或者writeObject方法的时候，就应该考虑使用序列化代理模式。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。