《EffectiveJava》学习笔记

**1.考虑静态工厂方法代替构造器：**

优势：

1）.有名称；

2）.不必在每次调用它们的时候都创建一个新对象；

3）.可以返回原返回类型的任何子类型的对象；

4）.在创建参数化类型实例的时候，它们使代码变得更加简洁。

缺点：

1）.类如果不含公有的或者受保护的构造器，就不能被子类化；

2）.它们与其他静态方法实际上没有区别。















**2.遇到多个构造器参数时要考虑用构建器：**

1）、重叠构造器模式可行，但是当有许多参数的时候，客户端代码会很难编写，并且难以阅读；

2）、JavaBean模式也可行，但是有个很严重的缺点，JavaBean模式阻止了把类做成不可变的可能，因此线程不安全；

3）、因此第三种方式，既可以保证像重叠构造器模式那样的安全性，也能保证像JavaBean模式那么好的可读性，也就是Builder模式。

Builder模式：

不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器，得到一个builder对象，然后客户端在builder对象上调用类似setter的方法，来设置每个相关的可选参数，最后客户端调用无参的builder方法生成不可变的对象。







**3.用私有构造器或者枚举类型强化Singleton属性：**

Singleton指仅仅被实例化一次的类。在jdk1.5前有多中方式实现单例模式，例如静态工厂方法、静态内部类等，但是它们各有弊端，比如线程问题或者反射问题，当然，抵御反射攻击时，可以修改构造器，让它在被要求创建第二个实例的时候抛出异常。



jdk1.5之后，单元素的枚举类型已经成为实现Singleton的最佳方法，既线程安全，又可避免反射攻击，同时代码也很优雅。

**4.通过私有构造器强化不可实例化的能力**

一些只包含静态方法或者静态域的类，比如工具类等，我们并不希望实例化该类，因为实例化并没有意义，此时就可以显示的私有化构造器来禁止类被实例化。但是也有副作用就是不能被子类继承了。

**5.避免创建不必要的对象**

一般来说，最好能重用对象而不是在每次需要的时候就创建一个相同功能的新对象。重用方式既快速又流行。如果对象是不可变的，它就始终可以被重用。

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。

除了重用不可变的对象之外，也可以重用那些已知不会被修改的可变对象。

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。

由于小对象的构造器只做很少量的显示工作，所以，小对象的创建和回收动作是非常廉价，特别是现代的JVM实现上更是如此。

**6.消除过期的对象引用**

过期引用：指永远也不会被解除的引用。

消除对象引用应该是一种例外，而不是一种规范行为。消除过期引用最好的方法是让包含该引用的变量结束其生命周期。

只要类是自己管理内存，程序猿就应该警惕内存泄漏问题。

内存泄漏的另一个常见来源是缓存。





内存泄漏的第三个常见来源是监听器和其他回调。

**7.避免使用终结方法**

终结方法通常是不可预测的，也是很危险的，一般情况下是不必要的。使用终结方法会导致行为不稳定，降低性能，以及可移植性问题。

**8.覆盖equals（）方法时要遵守通用约定**

如果满足了以下任一条件，就可不用覆盖equals（）方法：

1）、类的每个实例本质上是唯一的。

2）、不关心类是否提供了“逻辑相等”的测试功能。

3）、超类已经覆盖了equals，从超类继承过来的行为对于子类是合适的。

4）、类是私有的或是包私有的，可以确定它的equals方法永远不会被调用。

如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时就需要覆盖equals方法。这通常属于“值类”的情形。值类仅仅是一个表示值的类，例如Integer和Date。

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，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。

equals方法实现了等价关系：

1）、自反性：对于任何非null的引用值x，x.equals（x）必须返回true。

2）、对称性：对于任何非null的引用值x、y，当且仅当y.equals（x）返回true时，x.equals（y）必须返回true。



3）、传递性：对于任何非null的引用值x、y和z，如果x.equals（y）返回true，并且y.equals（z），那么x.equals（z）也必须返回true。

4）、一致性：对于任何非null的引用值x、y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals（y）就会一致地返回true，或者一致的返回false。

5）、非空性：对于任何非null的引用值x，x.equals（null）必须返回false。

里氏替换原则：一个类型的任何重要属性也将适用于它的子类型，因此为该类型编写任何方法，在它的子类型上也应该同样运行得很好。

实现高质量equals方法的诀窍：

1）、使用==操作符检查“参数是否为这个对象的引用”；

2）、使用instanceof操作符检查“参数是否为正确的类型”；

3）、把参数转化为正确的类型；

4）、对于该类中的每个“关键”域，检查参数中的域是否与该对象中对应的域相匹配；

5）、当编写完成了equals方法之后，应该问自己三个问题：它是否是对称的、传递的、一致的？并编写单元测试进行检验。

下面是String类的equals方法：



实现equals方法时要注意的问题：

1）、覆盖equals方法时总要覆盖hashCode方法；

2）、不要企图让equals方法过于智能；

3）、不要将equals声明中的Object对象替换为其他的类型。

**9.覆盖equals（）方法时总要覆盖hashCode方法**

在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。

**hashCode规范：**

**1）、在应用程序的执行期间，只要对象的equals方法的比较操作所用的的信息没有被修改，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。在同一个应用程序的多次执行过程中，每次执行所返回的整数可以不一致。（也就是说在应用程序一次的执行期间，相同对象的hashCode值一直相同）**

**2）、如果两个对象根据equals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。（也就是说相同对象的hashCode值一直相等。但是不同对象的hashCode值不一定不相等，比如String类型的相同值。）**

**3）、如果两个对象根据equals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方法时，则不一定要产生不同的散列码。但是，给不同的对象产生不同的整数结果，有可能提高散列表的性能。（比如HashMap类）**

**hashCode方法的解决方案：**

**1）、把某个非零的常数值，例如17，保存在一个名为result的int类型的常量中；**

**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”，然后针对这个范式调用hashCode，如果这个域的值为null，则返回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值，然后根据步骤2.b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中增加的一个Arrays.hashCode方法。**

**b、按照下面的公式，把2.a中计算得到的散列码c合并到result中：**

**result = 31 \* result + c。**

**3）、返回result。**

**注意：在散列码的计算中，可以把冗余域排除在外，换句话说，如果一个域的值可以根据参与计算的其他域值计算出来，则可以把这样的域排除在外。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。**

下面是String类的hashCode方法：



**10.始终要覆盖toString（）方法**

toString的约定进一步指出：建议所有的子类都覆盖这个方法。

当对象被传递给println、printlf、字符串联操作符（+）以及assert或者被调试器打印出来时，toString方法会被自动调用。

**11.谨慎地覆盖clone（）方法**

**12.考虑实现Comparable接口**

**compareTo方法的通用约定与equals方法相似：**

**将这个对象与指定的对象进行比较。当该对象小于、等于或大于指定对象的时候，分别返回同一个负整数、零或者正整数。如果由于指定对象的类型无法与该对象进行比较，则抛出ClassCastException异常。**

**在下面的说明中，符号sgn表示数学中的signum函数，它根据表达式的值为负值、零和正值，分别返回-1、0和1：**

**1）、实现者必须确保所有的x、y都满足sgn（x.compareTo（y）== -sgn（y.compareTo（x）））。**

**2）、实现者还必须确保这个比较关系是可传递的：（x.compareTo（y）>0&&y.compareTo（z））暗示着x.compareTo（z）>0。**

**3）、最后，实现者必须确保x.compareTo（y）==0暗示着所有的z都满足sgn（x.compareTo（z）== sgn（y.compareTo（z）））**

**4）、强烈建议（x.compareTo（y）==0）==（x.equals（y）），但这并非绝对。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。推荐使用这个说法：“注意：该类具有内在的排序功能，但是与equals不一致”。**

**在我的理解中，comparaTo方法更倾向于对象“数值”的比较，equals方法则是对象“本身”的比较。**



**13.使类和成员的可访问性最小化**

要区别设计良好的模块与设计不好的模块，最重要的因素在于，这个模块对于外部其他的模块而言，是否隐藏其内部数据和其他实现细节。设计良好的模块会隐藏所有的实现细节，把它的API与它的实现清晰地隔离开来。然后模块之间只通过它们的API进行通信，一个模块不需要知道其他模块的内部工作情况，这个概念称为信息隐藏或封装，是软件设计的基本原则之一。

信息隐藏的优势：

1）、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。这样可以加快系统开发的速度。

2）、减轻了维护的负担。

3）、提高了系统的可重用性。

4）、降低了构建大型系统的风险。

Java程序设计语言提供了许多机制来协助信息隐藏。访问控制机制决定了类、接口和成员的可访问性。实体的可访问性是由该实体声明所在的位置，以及该实体声明中出现的访问修饰符共同决定的。

**四种访问级别：**

**1）、私有的（private）：只有在声明该成员的顶层类内部才可以访问这个成员。**

**2）、包级私有：声明该成员的包内部的任何类都可以访问这个成员。它也被称为“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。**

**3）、受保护的（protected）：声明该成员的类的子类可以访问这个成员，并且声明该成员的包内部的任何类都可以访问这个成员。**

**4）、公有的（public）：在任何地方都可以访问这个成员。**

**信息隐藏的规则：**

1）、尽可能的使每个类或者成员不被外界访问。

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：包级私有的和公有的。

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。

4）、受保护的成员应该尽量少用或者不用。

5）、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，这样可以确保任何使用超类实例的地方也都可以使用子类的实例。如果一个类实现了一个接口，那么接口中所有的类方法在这个类中也都必须被声明为公有的，之所以如此，是因为接口中的所有方法都隐含着公有访问级别。

6）、为了便于测试，可以试着使类、接口或者成员变得更容易访问。

7）、实例域决不能是公有的，包含公有可变域的类并不是线程安全的。该规则同样适用于静态域。

8）、静态final域可以公有，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，这些域的名称由大写字母组成，单词之间用下划线隔开，这些域要吗包含基本类型的值，要吗包含指向不可变的对象的引用。如果final域包含可变对象的引用，它便具有非final域的所有缺点，虽然引用本身不能被修改，但是它所引用的对象却可以被修改。

9）、长度非零的数组总是可变的，所有，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。可以使公有的数组变成私有的，并增加一个公有的不可变列表。或者使公有的数组变成私有的，并添加一个公有方法，它返回私有数组的一个备份。

总而言之，应该始终尽可能降低可访问性，在设计了一个最小的公有的API之后，应该防止把任何散乱的类、接口或者成员变成API的一部分，除了公有静态final域的特殊情形之外，公有类都不应该包含公有域。并且确保公有静态final域所引用的对象都是不可变的。

**14.在公有类中使用访问方法而非公有域**

对于可变的类来说，应该用包含私有域和公有设值方法（setter）的类代替。

如果类在它所在的包的外部进行访问，就提供访问方法，以保留将来改变该类内部表示法的灵活性。

如果类是包级私有的，或者使私有的嵌套类，直接暴露它的数据域并没有本质的错误。

公有类永远都不应该暴露可变的域。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。

**15.使可变性最小化**

不可变类其实只是实例不能被修改的类，每个实例中包含的所有信息都必须在创建该实例时候就提供，并在对象的整个生命周期内固定不变。

**为了使类成为不可变，要遵循以下五条规则：**

**1）、不要提供任何会修改对象状态的方法。**

**2）、保证类不会被扩展。**

**3）、使所有的域都是final的。**

**4）、使所有的域都成为私有的。**

**5）、确保对于任何可变组件的互斥访问。**





**不可变对象的特点：**

**1）、不可变对象可以只有一种状态，即被创建时的状态。可变的对象可以有任意复杂的状态空间。**

**2）、不可变对象本质上是线程安全的，它们不要求同步。**

**3）、不可变对象可以提供一些静态工厂，它们把频繁被请求的实例缓存起来，从而当现有实例可以符合请求的时候，就不必创建新的实例。（变异的“单例模式”）。**

**4）、不可变对象可以被自由的共享，甚至也可以共享它们的内部信息。**

**5）、不可变对象为其他对象提供了大量的构件。**

**6）、不可变对象的唯一缺点就是对于不同的值都需要一个单独的对象。**

为了确保不变性，类绝对不允许自身被子类化。除了“使类成为final”的这种方法之外，还可以让类所有的构造器都变成私有的或者包级私有的，并添加公有的静态工厂来代替公有的构造器。如下：



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成为可变的类，否则就应该是不可变的。如果类不能被做成不可变的，仍然应该尽可能地限制它的可变性。除非有令人信服的理由要使域变成非final的，否则则要是每个域都是final的。

构造器应该创建完全初始化的对象，并建立起所有的约束关系，不要在构造器或静态工厂之外再提供公有的初始化方法，除非有令人信服的理由必须这么做，同样的，也不应该提供“重新初始化”方法。

**16.复合优于继承**

与方法调用不同的是，继承打破了封装性。换句话说，子类依赖于其超类中特定功能的实现细节。超类的实现有可能会随着发行版本的不同而有所变化，如果真的发生了变化，子类可能会被破坏，及时它的代码完全没有改变。

导致子类脆弱的另一个相关的原因是，超类有可能在后续的发行版本中获得新的方法。

如果在扩展一个类的时候，仅仅是增加新的方法，而不覆盖现有的方法，可能会认为这是安全的。虽然这种扩展方法比较安全，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，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。

**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，这种设计被称为复合，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，新类中的方法被称为转发方法。**









**每一个InstrumentedSet实例都把另一个Set实例包装起来了，所以InstrumentedSet类被称为包装类，这也正是Decorator模式。**

**17.要么为继承而设计，并提供文档说明，要么就禁止继承**

对于专门为了继承而设计并且具有良好文档说明的类，必须有文档说明它可覆盖的方法的自用性。对于每个公有的或受保护的方法或者构造器，它的文档必须指明该方法或者构造器调用了哪些可覆盖的方法，是以什么顺序调用的，每个调用的结果又是如何影响后续的处理过程的（所谓可覆盖的方法是指非final的，公有的或受保护的）。更一般地，类必须在文档中说明，在哪些情况下它会调用可覆盖的方法。

好的API文档应该描述一个给定的方法做了什么工作，而不是描述它是如何做到的。

为了允许继承，构造器决不能调用可被覆盖的方法，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。如果该覆盖版本的方法依赖于子类构造器所执行的任何初始化工作，该方法将不会如预期般执行。







**对于那些并非为了安全地进行子类化而设计和编写文档的类，要禁止子类化。有两种方法可以进行子类化。一种是是将类声明为final的，另一种是把所有的构造器变成私有的，或者包级私有的，并增加一些工厂方法来替代构造器。**

**18.接口优于抽象类**

Java程序设计语言提供了两种机制，可以用来定义允许多个实现的类型：接口和抽象类。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（Java8中接口可以有默认方法了）。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的哪个位置。因为Java只允许单继承，所以，抽象类作为类型定义受到了极大的限制。

接口的优势：

1）、现有的类可以很容易被更新，以实现新的接口。

2）、接口是定义mixin（混合类型）的理想选择。

3）、接口允许我们构造非层次结构的类型框架。

**通过对导出的每个重要接口都提供一个抽象的骨架实现类，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，但是骨架实现类接口了所有与接口实现相关的工作。**

骨架实现的美妙之处在于，它们为抽象类提供了实现上的帮助，但又不强加“抽象类被用作类型定义时”所特有的严格限制。对于接口的大多数实现来讲，扩展骨架实现类是个很显然的选择，但并不是必须的。如果预置的类无法扩展骨架实现类，这个类始终可以手工实现这个接口。此外，骨架实现类仍然能够有助于接口的实现。实现了这个接口的类可以把对于接口方法的调用，转发到一个内部私有类的实例上，这个内部私有类扩展了骨架实现类。这种方法被称为模拟多重继承。

**19.接口只用于定义类型**

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（动态方法调度）。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。为了其他任何目的定义接口都是不恰当的。

有一种接口叫做常量接口，这种接口不包含任何方法，只包含静态的final域，每个域都导出一个常量，常量接口模式是对接口的不良使用。

接口应该只被用来定义类型，它们不应该被用来导出常量。（多态）

**20.类层次优于标签类**

标签类：带有两种或多种风格的实例的类，类中充斥着样板代码，包括枚举声明、标签域、条件语句等。多个乱七八糟的实现挤在了单个类中，破坏了可读性。

标签类过于冗长，容易出错，并且效率低下。

类层次的优点：

1）、类层次纠正了标签类的所有缺点，每个类型的实现都配有自己的类，这些类都没有受到不相关的数据域的拖累，所有的域都是final的。

2）、类层次的另一个好处是可以用来反映类型之间本质上的层次关系，有助于增强灵活性，并进行更好的编译时类型检查。

**21.用函数对象表示策略**

有些语言支持函数指针、代理、lambda表达式（Java8），或者支持类似的机制，允许程序“调用特殊函数的能力”存储起来并传递这种能力。这种机制通常用于允许函数的调用者通过传入第二个函数。

函数指针的主要用途就是实现策略模式。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，它的类通常就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。

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







**22.优先考虑静态成员类**

嵌套类是指被定义在另一个类的内部的类。嵌套类存在的目的只是为它的外围类提供服务。如果嵌套类将来可能会用于其他的某个环境中，它就应该是顶层类。嵌套类有四种：静态成员类、非静态成员类、匿名类、局部类。

静态成员类是最简单的一种嵌套类。它可以访问外围类的所有成员，包括那些声明为私有的成员。静态成员类是外围类的一个静态成员，与其他的静态成员一样，遵守通用的可访问性规则。

静态成员类与非静态成员类之间唯一的区别是，静态成员类的声明包含修饰符static。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相关联。在非静态成员类的实例方法内部，可以调用外围实例上的方法，或者利用修饰过的this构造获得外围实例的引用。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：在没有外围类实例的情况下，要想创建非静态成员类的实例是不可能的。

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，如果省略了static修饰符，则每个实例都将包含一个额外的指向外围对象的引用。保存这份引用要消耗时间和空间，并且会导致外围实例在符合垃圾回收时却依然得以保留。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。

匿名类的一种常见用法是动态地创建函数对象；另一种常见用法是创建过程对象；第三种常见用法是在静态工厂方法的内部。

如果一个嵌套类需要在单个方法之外仍然是可见的，或者它太长了，不适合于放在方法内部，就应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做成非静态的；否则就做成静态的。假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类，否则就做成局部类。

**23.请不要在新代码中使用原生态类型**

声明中具有一个或多个类型参数的类或者接口，就是泛型类或接口。

每种泛型定义一组参数化的类型，构成格式为：先是类或者接口的名称，接着用尖括号（<>）把对应于泛型形式类型参数的实际类型参数列表括起来。

每个泛型都定义一个原生态类型，即不带任何实际类型参数的泛型名称。如果使用原生态类型，就失掉了泛型在安全性和表述性方面的所有优势。为了提供兼容性，所以Java才允许使用原生态类型。

泛型有子类型化的规则。

Set<Object>是个参数化类型，表示可以包含任何对象类型的一个集合；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；Set则是个原生态类型，它脱离了泛型系统。前两种是安全的，最后一种不安全。

**24.消除非受检警告**

用泛型编程时，会遇到许多编译器警告：非受检强制转化警告、非受检方法调用警告、非受检普通数组警告、非受检转换警告。

要尽可能地消除每一个非受检警告，如果消除了所有警告，就可以确保代码是类型安全的，这意味着不会在运行时出现ClassCastException。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，可以用一个@SuppressWarnings("unchecked")注解来禁止这条警告。

应该始终在尽可能小的范围中使用@SuppressWarnings("unchecked")注解，永远不要在整个类上使用，这么做有可能会掩盖了重要的警告。如果发现在长度不止一行的方法或者构造器中使用了@SuppressWarnings("unchecked")注解，可以将它移到一个局部变量的声明中。将@SuppressWarnings("unchecked")注解放在return语句中是非法的，因为它不是一个声明。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。

总而言之，非受检警告很重要，不要忽略它们。每一条警告都表示可能在运行时抛出ClassCastException异常。要尽最大的努力消除这些警告。如果无法消除非受检警告，同时可以证明引起警告的代码是类型安全的，就可以在尽可能小的范围内用@SuppressWarnings("unchecked")注解禁止该警告。要用注释把禁止该警告的原因记录下来。

**25.列表优先于数组**

数组与泛型的区别：

1）、数组是协变的，假如Sub是Super的子类，那么数组Sub[]就是Super[]的子类型；泛型是不可变的，对于任意两个不同类型Type1和Type2，List<Type1>既不是List<Type2>的子类型，也不是List<Type2>的超类型。

2）、数组是具体化的，因此数组会在运行时才知道检查它们的元素类型约束；泛型是通过擦除来实现的，因此泛型只在编译时强化它们的类型信息，并在运行时丢弃它们的元素类型信息，擦除就是使泛型可以与没有使用泛型的代码随意进行互用。

由于以上的区别，因此数组和泛型不能很好的混合使用，创建泛型、参数化类型或者类型参数的数组是非法的，因为它不是类型安全的。

不可具体化类型是指运行时表示法包含的信息比它在编译时表示法包含的信息更少的类型。唯一具体化的参数化类型是无限制的通配符类型，如List<?>。

总而言之，数组是协变且可以具体化的；泛型是不可变的且可以被擦除的。因此，数组提供了运行时的类型安全，但是没有编译时的类型安全，反之，对于泛型也一样。一般来说，数组和泛型不能很好的混合使用，应该用列表代替数组。

**26.优先考虑泛型**

不可以创建不可具体化的数组，解决这个问题的方法：**直接绕过创建泛型数组的禁令，创建一个Object数组，并将它转换成泛型数组类型。**



使用泛型比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。在设计新类型的时候，要确保它们不需要这种转换就可以使用。这通常意味着要把类做成是泛型的。只要时间允许，就把现有的类型都泛型化（**还是要视情况而定，不可一概而论**）。这对于这些类型的新用户来说会变得更加轻松，又不会破坏现有的客户端。

**27.优先考虑泛型方法**

泛型方法的一个显著特征是，无需明确指定类型参数的值，不像调用泛型构造器的时候必须指定。编译器通过检查方法参数的类型来计算类型参数的值，这叫做类型推倒。例如，编译器发现show方法的参数时Set<String>类型，因此知道类型参数T必须为String。



通过某个包含类型参数本身的表达式来限制类型参数是允许的，这就是递归类型限制。



**28.利用有限制通配符来提升API的灵活性**

为了获得最大限度的灵活性，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，那么通配符类型就没有什么好处了。

**PESC表示producer-extends，consumer-super。**

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>。

**所有的Comparable和Comparator都是消费者。**



**29.优先考虑类型安全的异构容器**

集合API说明了泛型的一般用法，限制每个容器只能有固定数目的类型参数。可以通过将类型参数放在键上而不是容器上来避开这一限制。对于这种类型安全的异构容器，可以用Class对象作为键。以这种方式使用的Class对象称作类型令牌。也可以使用定制的键类型。

**30.用enum代替int常量**

枚举类型是指由一组固定的常量组成合法值的类型。

Java枚举类型背后的基本想法：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。

枚举类型是实例受控的，它们是单例的泛型化，本质上是单元素的枚举。枚举提供了编译时的类型安全。包含同名常量的多个枚举类型可以在一个系统中和平共处，因为每个类型都有自己的命名空间。枚举类型还允许添加任意的方法和域，并实现任意的接口。它们提供了所有的Object方法的高级实现，实现了Conparable和Serializable接口，并针对枚举类型的可任意改变性设计了序列化方式。

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是final的，它们可以是公有的，但最好将它们做成私有的，并提供公有的访问方法。

与枚举常量关联的有些行为，可能只需要用在定义了枚举的类或包中，这种行为最好实现成私有的或者包级私有的方法。如果一个枚举具有普遍适用性，它就应该成为一个顶层类；如果它只是被用在一个特定的顶层类中，它就应该成为该顶层类的一个成员类。

有一种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方法，并在特定于常量的类主体中，用具体的方法覆盖每个常量的抽象方法，这种方法被称作特定于常量的方法实现。

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。



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

**策略枚举：**





**31.用实例域代替序数**

所有的枚举类型都有ordinal方法，它返回每个枚举常量在类型中的数字位置，但是最好完全避免使用ordinal方法。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。



**32.用EnumSet代替位域**

**33.用EnumMap代替序数索引**

**最好不要用序数来索引数组，而要使用EnumMap。**

**34.用接口模拟可伸缩的枚举**

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。

**35.注解优先于命名模式**

命名模式：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测试框架原本要求用户一定要用test作为测试方法名称的开头。

命名模式缺点：

1）、文字拼写错误会导致失败，且没有任何提示。

2）、无法确保它们只用于相应的程序元素上。

3）、没有提供将参数值与程序元素关联起来的好方法。

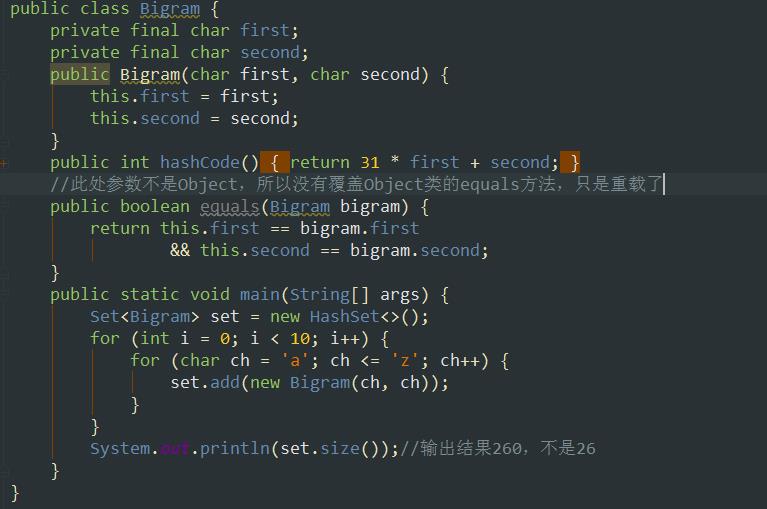
Test注解类型的声明就是他自身通过Retention和Target注解进行了注解，注解类型声明中的这种注解称为元注解。

标记注解：没有参数，只是“标注”被注解的元素。

所有的程序猿都应该使用Java平台所提供的预定义的注解类型，还要考虑使用IDE或者静态分析工具所提供的任何注解。这种注解可以提升由这些工具所提供的诊断信息的质量。但是要注意这些注解还没有标准化。

**36.坚持使用Override注解**

Override注解只能用在方法声明中，它表示被注解的方法声明覆盖了超类型中的一个声明。



总而言之，在每个方法声明中使用Override注解来覆盖超类声明，编译器就可以防止大量的错误，但有一个例外，在具体的类中，不必标注确信覆盖了抽象方法声明的方法（虽然这么做也没什么坏处）。

**37.用标记接口定义类型**

标记接口：是没有包含方法声明的接口，而只是声明一个类实现了具有某种属性的接口。例如Serializable接口。

标记接口强于标记注解的原因：

1）、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。就Serializable接口而言，如果它的参数没有实现该接口，ObjectOutPutStream.write方法将会失败。

2）、标记接口可以被更加精确的进行锁定，如果注解类型利用@Target（ElementType.TYPE）声明，它就可以被应用到任何类或者接口。

标记注解强于标记接口的原因：

1）、标记注解可以通过默认的方式添加一个或多个注解类型元素，给已被使用的注解类型添加更多的信息。

2）、标记注解是更大的注解机制的一部分。

标记接口和标记注解的使用时机：

如果标记是应用到任何程序元素而不是类或者接口，就必须使用注解，因为只有类或者接口可以用来实现或扩展接口。如果标记只应用给类或者接口，就应该优先使用标记接口而非注解。

**38.检查参数的有效性**

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。

在设计方法时，应该使它们尽可能的通用，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。

简而言之，每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。应该把这些限制写到文档中，并且在这个方法体的开头处，通过显示的检查来实施这些限制。

**39.必要时进行保护性拷贝**

对于构造器的每个可变参数进行保护性拷贝是必要的。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。对于参数类型可以被不可信任方子类化的参数，请不要使用clone方法进行保护性拷贝。

参数的保护性拷贝并不仅仅针对不可变类。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。

只要有可能，都应该使用不可变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。保护性拷贝可能会带来性能损失。

总而言之，如果类具有从客户端得到或者返回到客户端的可变组件，类就必须保护性地拷贝这些组件。如果拷贝的成本受到限制，并且类信任它的客户端不会不恰当地修改组件，就可以在文档中指明客户端的职责是不得修改受到影响的组件，以此来代替保护性拷贝。

**40.谨慎设计方法签名**

1、谨慎地选择方法的名称，方法的名称应该始终遵循标准的命名习惯。

2、不要过于追求提供便利的方法，每个方法都应该尽其所能。

3、避免过长的参数列表，目标是四个参数或者更少。

缩短过长的参数列表的三种方法：

1）、把方法分解成多个方法，每个方法只需要这些参数的一个子集。

2）、创建辅助类，用来保存参数的分组。

3）、从对象构建到方法调用都采用Builder模式。

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以用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。

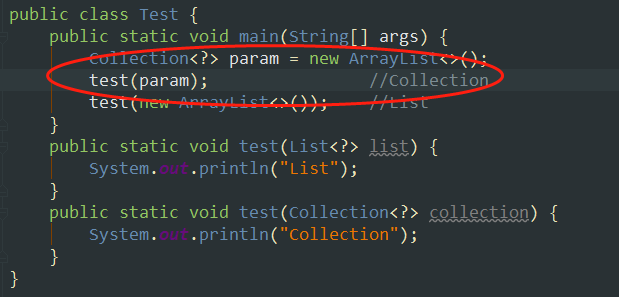
对于boolean参数，要优先使用两个元素的枚举类型。

**41.慎用重载**

要调用哪个重载方法是在编译时决定的。对于重载方法的选择是静态的，而对于被覆盖的方法的选择则是动态的。选择被覆盖的方法的正确版本是在运行时进行的，选择的依据是被调用的方法所在对象的运行时类型。当一个子类包含的方法声明与其祖先类中的方法声明具有相同的签名时，方法就被覆盖了。如果实例方法在子类中被覆盖了，并且这个方法是在该子类的实例上被调用的，那么子类中的覆盖方法将会执行，而不管该子类实例的编译时类型到底是什么。

当调用被覆盖的方法时，对象的编译时类型不会影响到哪个方法将被执行；“最为具体的”那个覆盖版本总是会得到执行。这与重载的情形相比，对象的运行时类型并不影响“哪个重载版本将被执行”；选择工作是在编译时进行的，完全基于参数的编译时类型。

应避免胡乱地使用重载机制，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，保守的策略是根本不要重载（重载目前更广泛用于参数过长的方法）。



对于构造器，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，构造器不可能被覆盖。

总而言之，“能够重载方法”并不意味着就“应该重载方法”。一般情况下，对于多个具有相同参数数目的方法来说，应该尽量避免重载方法。在某些情况下，特别是涉及构造器的时候，要遵循这个建议也许是不可能的。在这种情况下，至少应该避免这样的情形：同一组参数只需经过类型转换就可以被传递给不同的重载方法。如果不能避免这种情形，例如，因为正在改造一个现有的类以实现新的接口，就应该保证：当传递相同的参数时，所有重载方法的行为必须一致。如果不能做到这一点，程序猿就很难有效地使用被重载的方法构造器，他们就不能理解它为什么不能正常的工作。

**42.慎用可变参数**

可变参数方法接受0个或多个指定类型的参数。可变参数机制通过先创建一个数组，数组的大小为在调用位置所传递的参数数量，然后将参数值传到数组中，最后将数组传递给方法。

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。

在重视性能的情况下，使用可变参数机制要特别小心。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。假设确定对每个方法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。

简而言之，在定义参数数目不定的方法时，可变参数方法是一种很方便的方式，但是它们不应该被过度滥用，如果使用不当，会产生混乱的结果，还会有性能影响。

**43.返回零长度的数组或者集合，而不是null**

**44.为所有导出的API元素编写文档注释**

如果要想使一个API真正可用，就必须为其编写文档。Java语言环境提供了一种被称为Javadoc的实用工具，从而使这项任务变得很容易。Javadoc利用特殊格式的文档注释，根据源代码自动产生API文档。

为了正确编写API文档，必须在每个被导出的类、接口、构造器、方法和域声明之前增加一个文档注释。方法的文档注释应该简洁地描述出它和客户端之间的约定。文档注释应该列举出这个方法的所有前提条件和后置条件，所谓前提条件是指为了使客户能够调用这个方法，而必须要满足的条件；所谓后置条件是指在调用成功之后，哪些条件必须被满足。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，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。

再也没有必要再在文档注释中使用HTML<code>和<tt>标签了：Javadoc{@code}标签更好，因为它避免了转义HTML元字符。

为泛型或者方法编写文档时，确保要在文档中说明所有的类型参数。为枚举类型编写文档时，要确保在文档中说明常量，以及类型，还有任何公有的方法。为注解类型编写文档时，要确保在文档中说明所有成员，以及类型本身。

**45.将局部变量的作用域最小化**

将局部变量的作用域最小化，可以增强代码的可读性和可维护性，并降低出错的可能性。

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。

几乎每个局部变量的声明都应该包含一个初始化表达式。如果没有足够的信息对一个变量进行初始化，就应该推迟这个声明，直到可以初始化为止。这个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，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”。

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，使用for循环与使用while循环相比还有另一个优势：更简短，从而增强了可读性。

最后一种“将局部变量的作用域最小化”的方法就是使方法小而集中。

**46.for-each循环优先于传统的for循环**

Java 1.5发行版本中引入的for-each循环，通过完全隐藏迭代器或者索引变量，避免了混乱和出错的可能。

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，某些情况下，比起普通的for循环，它还稍有性能优势，因为它对数组索引的边界值只计算一次。

在对多个集合进行嵌套式迭代时，for-each循环相对于传统的for循环优势更加明显。

for-each循环不仅可以遍历集合和数组，还可以遍历任何实现了Iterable接口的对象，包括返回Iterable接口实现类的方法。

总之，for-each循环在简洁性和预防Bug方法有着传统的for循环无法比拟的优势，并且没有性能损失。应该尽可能的使用for-each循环。不过，有三种情况无法使用for-each循环：

1）、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。

2）、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。

3）、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。

**47.了解和使用类库**

使用标准类库的好处：

1）、通过使用标准类库，可以充分利用这些编写标准类库的专家的知识，以及在你之前的其他人的使用经验。

2）、不必浪费时间为那些与工作不太相关的问题提供特别的解决方案。

3）、它们的性能往往会随着时间的推移而不断提高，无需自己做任何努力。

4）、标准类库会随着时间的推移增加新的功能。

5）、可以使自己的代码融入主流，这样的代码更易读、更易维护、更易被大多数的开发人员重用。

在每个重要的发行版本中，都会有许多新的特性被加入到类库中，所以与这些新特性保持同步是值得的。每个程序猿都应该熟悉java.lang、java.util，某种程度上还有java.io的内容。在1.2发行版本中，Collection Framework被加入到了java.util包中。1.5发行版本中，在java.util.concurrent包中增加了一组并发实用工具。

**48.如果需要精确的答案，请避免使用float和double**

float和double类型主要是为了科学计算和工程计算而设计的，它们执行二进制浮点运算，这是为了在广泛的数值范围上提供较为精确的快速近似计算而精心涉及的。但是，它们并没有提供完全精确的结果，所以不应该被用于需要精确结果的场合。float和double类型尤其不适合用于货币计算。

使用BigDecimal、int和long进行货币计算。使用BigDecimal有两个缺点：与使用基本运算类型相比，这样做很不方便，而且慢。

**49.基本类型优先于装箱基本类型**

Java有一个类型系统由两部分组成，包含基本类型和引用类型。每个基本类型都有一个对应的引用类型，称为装箱基本类型。Java 1.5发行版本中增加了自动装箱和自动拆箱。

基本类型与装箱基本类型的区别：

1）、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。

2）、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。

3）、基本类型通常比装箱基本类型更节省时间和空间。

对装箱基本类型运用==操作符几乎总是错误的。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。



使用装箱基本类型的场景：

1）、作为集合中的元素、键和值。

2）、在参数化类型中必须使用装箱基本类型，Java不允许使用基本类型。

3）、在进行反射的方法调用时，必须使用装箱基本类型。

**50.如果其他类型更合适，则尽量避免使用字符串**

不应该使用字符串的情形：

1）、字符串不适合代替其他的值类型。

2）、字符串不适合代替枚举类型，枚举类型比字符串更适合用来表示枚举类型的常量。

3）、字符串不适合代替聚集类型。如果一个实体有多个组件，用一个字符串来表示这个实体通常是不恰当的。

4）、字符串也不适合代替能力表。有时候字符串被用来对某种功能进行授权。

**51.当心字符串连接的性能**

字符串连接操作符（+）是把多个字符串合并为一个字符串的便利途径。为连接n个字符串而重复使用字符串连接操作符，需要n的平方级的时间。这是由于字符串不可变而导致的不幸后果。当两个字符串被连接在一起时，它们的内容都要被拷贝。

为了获得可以接受的性能，要使用StringBuilder类代替String。字符串连接操作符的开销随项目数量而呈平方级增加，StringBuilder则是线性增加。

原则很简单：不要使用字符串连接操作符来合并多个字符串，除非性能无关紧要，相反，应该使用StringBuilder的append方法。另一种方法是，使用字符数组，或者每次只处理一个字符串，而不是将它们组合起来。

**52.通过接口引用对象**

应该使用接口而不是用类作为参数的类型。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、变量和域来讲，就都应该使用接口类型进行声明。

不存在适当接口类型的情形：

1）、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。

2）、对象属于一个框架，而框架的基本类型是类，不是接口。如果对象属于这种基于类的框架，就应该用相关的基类（往往是抽象类）来引用这个对象，而不是它的实现类。

3）、类实现了接口，但是它提供了接口中不存在的额外方法。

**53.接口优先于反射机制**

核心反射机制java.lang.reflect，提供了“通过程序来访问关于已装载的类的信息”的能力。反射机制运行一个类使用另一个类，即使当前者被编译的时候后者还根本不存在。

反射机制的缺点：

1）、丧失了编译时类型检查的好处。

2）、执行反射访问所需要的代码非常笨拙和冗长。

3）、性能损失。

反射功能只是在设计时被用到，通常，普通应用程序在运行时不应该以反射方式访问对象。如果只是以非常有限的形式使用反射机制，虽然也要付出少许代价，但是可以获得很多好处。

**54.谨慎地使用本地方法**

Java Native Interface（JNI）允许Java应用程序可以调用本地方法，所谓本地方法是指本地程序设计语言（比如C或者C++）来编写的特殊方法。本地方法可以在本地语言中执行任意的计算任务，并返回到Java程序设计语言。

本地方法的三种用途：

1）、提供了“访问特定于平台的机制”的能力，比如访问注册表和文件锁。

2）、提供了访问遗留代码库的能力。

3）、可以通过本地语言编写程序中注重性能的部分。但是使用本地方法来提供性能的做法不值得提倡。Java本身已经越来越快，性能越来越好。

本地方法的缺陷：

1）、本地语言不是安全的。

2）、本地语言与平台相关，不可以再自由移植了。

3）、使用本地方法更难调试。

4）、在进入和退出本地代码时，需要相关的固定开销，所以，如果本地代码只是做少量的工作，本地方法就可能降低性能。

5）、需要“胶合代码”的本地方法编写起来单调乏味，并且难以阅读。

**55.谨慎地进行优化**

不要费力的编写快速的程序--应该努力编写好的程序，速度自然而然随之而来。在设计系统的时候，特别是在设计API、线路层协议和永久数据格式的时候，一定要考虑性能的因素，当构建完系统之后，要测量它的性能。如果它足够快，任务就完成了。如果不够快，则可以在性能剖析器的帮助下，找到问题的根源，然后设法优化系统中相关的部分。第一个步骤是检查所选择的算法：更多的底层优化也无法弥补算法的选择不当。必要时重复这个过程，在每次改变之后都要测量性能，知道满意为止。

**56.遵守普遍接受的命名惯例**

关于这点可以阅读《阿里巴巴Java开发手册》。

**57.只针对异常的情况才使用异常**

在现代的JVM实现上，基于异常的模式比标准模式要慢得多。异常应该只用于异常的情况下，它们永远不应该用于正常的控制流。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”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”的方法，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。

**58.对可恢复的情况使用受检异常，对编程错误使用运行时异常**

Java提供了三种可抛出结构：受检的异常、运行时异常和错误。

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的恢复，对于这种情况就应该使用受检的异常。

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，并出现适当的错误信息。

用运行时异常来表明编程错误。大多数的运行时异常都表示前提违例，所谓前提违例是指API的客户没有遵守API规范建立的约定。

总而言之，对于可恢复的情况，使用受检的异常；对于程序错误，则使用运行时异常。

**59.避免不必要地使用受检的异常**

**60.优先使用标准的异常**

重用现有的异常的好处：

1）、会使API更加易于学习和使用，因为它与程序猿已经熟悉的习惯用法是一致的。

2）、对于用到这些API的程序而言，它们的可读性会更好，因为它们不会出现很多程序眼不熟悉的异常。

3）、异常类越少，意味着内存印迹越小，装载这些类的时间开销也越少。

常用的异常：

IllegalArgumentException：非null的参数值不正确。

IllegalStateException：对于方法调用而言，对象状态不合适。

NullpointException：在禁止使用null的情况下参数值为null。

IndexOutOfBoundsException：下标参数值越界。

ConcurrentModificationException：在禁止并发修改的情况下，检测到对象的并发修改。

UnsupportedOperationException：对象不支持用户请求的方法。

**61.抛出与抽象相对应的异常**

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，这种做法叫做异常转译。

一种特殊的异常转译形式称为异常链，如果低层的异常对于调试导致高层异常的问题有帮助，使用异常链就很合适。低层的异常被传到高层的异常，高层的异常提供访问方法来获得低层的异常。

高层异常的构造器将原因传到支持链的超级构造器，因此它最终将被传给Throwable的其中一个运行异常链的构造器。大多数标准的异常都有支持链的构造器。对于没有支持链的异常，可以利用Throwable的initCause方法设置原因。异常链不仅可以让你通过程序（getCause）访问原因，还可以将原因的堆栈轨迹集成到更高层的异常中。

尽管异常转译与不加选择地从低层传递异常的做法相比有所改进，但是它也不能被滥用。

总而言之，如果不能阻止或者出来来自低层的异常，一般的做法是使用异常转译，除非低层方法碰巧可以保证它抛出的所有异常对高层也合适才可以将异常从低层传播到高层，异常链对高层和低层异常都提供了最佳的功能：它允许抛出适当的高层异常，同时又能捕获低层的原因进行失败分析。

**62.每个方法抛出的异常都要有文档**

始终要单独地声明受检的异常，并且利用Javadoc和@throws标记，准备的记录下抛出的每个异常的条件。**永远不要声明一个方法“throws Exception”或者“throws Throwable”。**

使用Javadoc的@throws标签记录下一个方法可能抛出的每个未受检异常，但是不要使用throws关键字将未受检的异常包含在方法的声明中。如果一个类中的许多方法出于同样的原因而抛出同一个异常，在该类的文档注释中对这个异常建立文档，这是可以接受的，而不是为每个方法单独建立文档。

**63.在细节消息中包含能捕获失败的信息**

当程序由于未被捕获的异常而失败的时候，系统会自动地打印出改异常的堆栈轨迹。在堆栈轨迹中包含该异常的字符串表示法，即它的toString方法的调用结果。它通常包含该异常的类名，紧随其后的是细节消息。

为了捕获失败，异常的细节信息应该包含所有“对该异常有贡献”的参数和域的值。

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，一种方法是在异常的构造器而不是字符串细节消息中引入这些消息。然后，有了这些消息，只要把它们放到消息描述中，就可以自动产生细节消息。

**64.努力使失败保持原子性**

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。

实现失败原子性的做法：

1）、设计一个不可变的对象。

2）、对于在可变对象上执行操作，可以在执行操作之前检查参数的有效性，这就可以使在对象的状态被修改之前，先抛出适当的异常。

3）、一种类似的获得失败原子性的办法是，调整计算处理过程的顺序，使得任何可能会失败的计算部分都在对象状态被修改之前发生。如果对参数的检查只有在执行了部分计算之后才能进行，这种办法实际上就是上一种办法的自然扩展。

4）、编写一段恢复代码，由它来拦截操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。这种办法主要用于永久性的数据结构。

5）、在对象的一份临时拷贝上执行操作，当操作完成之后再用临时拷贝中的结果代替对象的内容。

**65.不要忽略异常**

要忽略一个异常很容易，只需要将方法调用通过try语句块包围起来，并包含一个空的catch块。空的catch块会使异常达不到应有的目的，即强迫处理异常的情况。至少，catch块应该包含一条说明，解释为什么可以忽略这个异常。

有一种情形可以忽略异常，即关闭FileInputStream的时候。

**66.同步访问共享的可变数据**

关键字synchronized可以保证在同一时刻，只有一个线程可以执行某一个方法，或者某一个代码块。同步不仅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，都看到同一个锁保护的之前所有的修改效果。

Java语言规范保证读或者写一个变量是原子的，除非这个变量的类型是long或者boolean的。换句话说，读取一个非long或者boolean类型的变量，可以保证返回的值是某个线程保存在该变量中的，即使多个线程在没有同步的情况下并发地修改这个变量也是如此。

volatile关键字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。

增量操作符（++）不是原子的。

让一个线程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，只同步共享对象的引用动作。然后其他线程没有进一步的同步可以读取对象，只要它没有再被修改，这种对象被称作事实上的不可变的。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。安全发布对象引用有多重方法：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域中、final域或者通过正常锁定访问的域中；或者可以将它放到并发的集合中。

简而言之，当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步。如果没有同步，就无法保证一个线程所做的修改可以被另一个线程获知。未能同步共享可变数据会造成程序的活性失败和安全性失败。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。

**67.避免过度同步**

依据情况的不同，过度同步可能会导致性能降低、死锁，甚至是不确定的行为。为了避免活性失败和安全性失败，在一个被同步的方法或者代码块中，永远不要放弃对客户端的控制。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。

在同步区域之外被调用的外来方法被称作“开放调用”。除了可以避免死锁之外，开放调用还可以极大地增加并发性。外来方法的运行时间可能会任意长。

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，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，就不要再内部同步。让客户在必要的时候从外部同步。

如果方法修改了静态域，也必须同步对这个域的访问，即使它往往只用于单个线程。

简而言之，为了避免死锁和数据破坏，千万不要从同步区域内部调用外来方法。更为一般的讲，要限制同步区域内部的工作量。当设计一个可变类的时候，要考虑一下它们是否应该自己完成同步操作。在多核的时代，这比永远不要过度同步来得更重要。只有有足够的理由一定要在内部同步类的时候，才应该这样做，同时还应该将这个决定清楚地写到文档中。

**68.executor和task优先于线程**

如果想让不止一个线程来处理癞子队列的请求，只要调用一个不同的静态工厂，这个工厂创建了一种不同的executor service，称作线程池。

不仅应该尽量不要编写工作队列，而且还应该尽量不直接使用线程。现在关键的抽象不再是Thread了，它以前可是既充当工作单元，又是执行任务。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。任务有两种：Runable及其近亲Callable。执行任务的通用机制是excutor service。

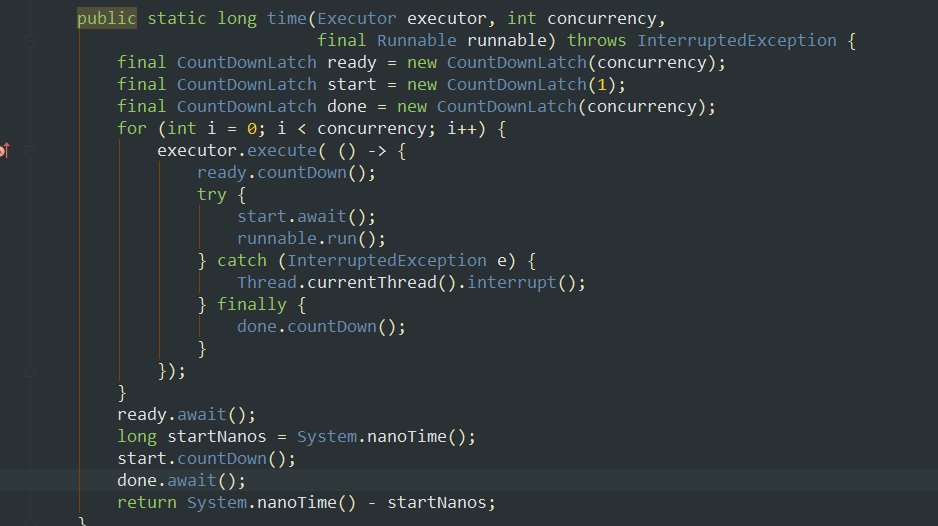
**69.并发工具优先于wait和notify**

java.util.concurrent中更高级的工具分为三类：Executor Framework，并发集合以及同步器。

优先使用ConcurrentHashMap，而不是使用Collections.synchronizedMap或者Hashtable。

同步器是一些使线程能够等待另一个线程的对象，允许它们协调动作。最常用的同步器是CountDownLatch和Semaphore。

倒计数锁存器（CountDown Latch）是一次性的障碍，允许一个或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。



对于间歇式的定时，始终应该优先使用System.nanoTime，而不是System.currentTimeMills。System.nanoTime更加准确也更加精确，它不受系统的实时时钟的调整所影响。

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。

**70.线程安全性的文档化**

在一个方法声明中出现synchronize修饰符，这是个实现细节，并不是导出的API的一部分。

**线程安全性级别：**

**1）、不可变的--这个类的实例是不变的，所以，不需要外部同步。例如String、Long和BigInteger。**

**2）、无条件的线程安全--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，无需任何的外部同步。例如Random和ConcurrentHashMap。**

**3）、有条件的线程安全--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，它们的迭代器需要外部同步。**

**4）、非线程安全--这个类的实例是可变的。为了并发的使用它们，客户必须利用自己选择的外部同步包围每个方法的调用。这样的例子包括通用的集合实现，例如ArrayList和HashMap。**

**5）、线程对立的--这个类不能安全的被多个线程并发使用，即使所有的方法调用都被外部同步使用。线程对立的根源通常在于，没有同步地修改静态数据。**

为了避免拒绝服务攻击，应该使用私有锁对象来代替同步的方法，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。私有锁对象模式只能用在无条件的线程安全的类上，有条件的线程安全类不能使用这种模式，因为它们必须在文档中说明：在执行某些方法调用序列时，它们的客户端程序必须获得哪把锁。

**71.慎用延迟初始化**

延迟初始化是延迟到需要域的值时才将它初始化的行为。如果永远不需要这个值，这个域就永远不会被初始化。这种方法既适用于静态域，也适用于实例域。

就像大多数的优化一样，对于延迟初始化，最好建议“除非非常必要，否则就不要这样做”。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，延迟初始化实际上降低了性能。

在大多数情况下，正常的初始化要优先于延迟初始化。如果利用延迟优化来破坏初始化的循环，就要使用同步访问方法。

如果处于性能的考虑：对于实例域，就使用双重检查模式；对于静态域，使用懒加载内部类模式；对于可以接受重复初始化的实例域，也可以考虑使用单重检查模式。（类似于单例模式）

**72.不要依赖线程调度器**

当有多个线程可以运行时，由线程调度器决定哪些线程将会运行，以及运行多长时间。任何依赖线程调度器来达到正确性或者性能要求的程序，很有可能都是不可移植的。不要依赖Thread.yield或者线程优先级。

**73.避免使用线程组**

**74.谨慎的实现Serializable接口**

将一个对象编码成一个字节流，称作将该对象序列化，相反的处理过程称作反序列化。一旦对象被序列化后，他的编码就可以从一台正在运行的虚拟机被传递到另一台虚拟机上，或者被存储到磁盘上，供以后反序列化时用。

要想使一个类的实例可被序列化，只要在它的声明中加入“implements Serializable”即可。

实现Serializable接口的代价：

1）、一旦一个类被发布，就大大降低了“改变这个类的实现”的灵活性。

每个可序列话的类都有一个唯一标识符与它相关联，称为序列版本UID。如果没有显式的指定该标识号，系统会自动地根据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，这个自动产生的标识号的值会受到类名称、实现的接口名称、以及所有公有的和受保护的成员的名称所影响。如果通过任何方式改变了这些信息，自动产生的序列版本UID也会发生变化。因此，如果没有显式的声明一个序列版本UID，兼容性会遭到破坏，在运行时导致InvalidClassException异常。

2）、增加了出现BUG和安全漏洞的可能性。

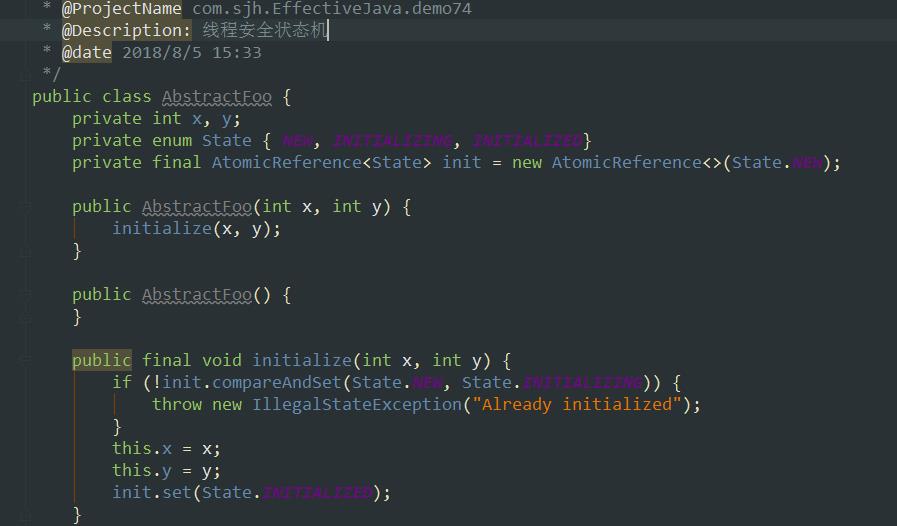
对象是利用构造器创建的；序列化机制是一种语言之外的对象创建机制。无论是接受了默认的行为，还是覆盖了默认的行为，反序列化机制都是一个“隐藏的构造器”，具备与其他构造器相同的特点。因为反序列化机制中没有显式的构造器，所以要确保：反序列化过程必须也要保证所有“由真正的构造器建立起来的约束关系”，并且不允许攻击者访问正在构造过程中的对象的内部信息。依靠默认的反序列化机制，很容易使对象的约束关系得到破坏，以及遭受到非法访问。

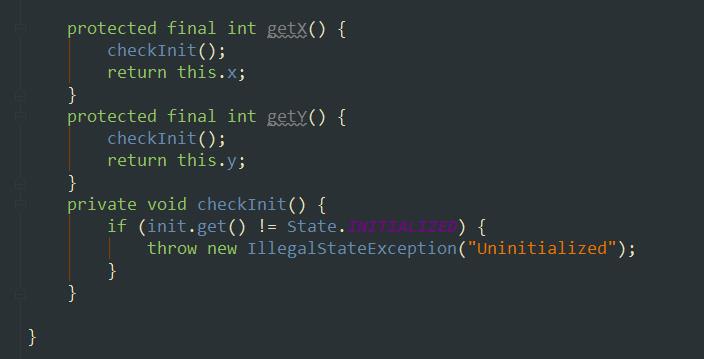
3）、随着类发行新的版本，相关的测试负担也增加了。

为了继承而设计的类应该尽可能少地实现Serializable接口，用户的接口也应该尽可能少地继承Serializable接口。然而在有些情况下，例如，如果一个类或者接口存在的目的主要是为了参与到某个框架中，该框架要求所有的参与者都必须实现Serializable接口。那么，对于这个类或者接口来说，实现或者扩展Serializable接口就是非常有意义的。

内部类不应该实现Serializable接口，它们使用编译器产生的合成域来保存指向外围实例的引用，以及保存来自外围作用域的局部变量的值。内部类的默认序列化形式是定义不清楚的，然而，静态成员类却可以实现Serializable接口。

**线程安全状态机：**





**75.考虑使用自定义的序列化形式**

当一个对象的物理表示法与它的逻辑数据内容有实质性的区别时，使用默认序列化形式会有以下4个缺点：

1）、它使这个类的导出API永远地束缚在该类的内部表示法上。

2）、会消耗过多的空间。

3）、会消耗过多的时间。

4）、会引起堆栈溢出。

transient修饰符表明实例域将从一个类的默认序列化形式省略掉。

无论是否使用默认的序列化形式，如果在读取整个对象状态的任何其他地方上强制任何同步，则也必须在对象序列化上强制这个同步。

不管选择了哪种序列化形式，都要为自己编写的每个可序列化的类声明一个显示的序列版本UID。这样可以避免潜在的不兼容根源，而且还会带来小小的性能提升。

**76.保护性的编写readObject方法**

每当编写readObject方法的时候，都要这样想：你正在编写一个公有的构造器，无论给它传递什么样的字节流，它都必须产生一个有效的实例。

有助于编写健壮的readObject方法的指导方针：

1）、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。

2）、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。

3）、如果整个对象图在被反序列化之后必须进行验证，就应该使用ObjectInputValidation接口。

4）、无论是直接方式还是间接方式，都不要调用类中任何可被覆盖的方法。

**77.对于实例控制，枚举类型优先于readResolve**

如果单例模式的类的声明中加上了“implements Serializable”，它就不再是单例的。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示的readObject方法无关。任何一个readObject方法，不管是显示的还是默认的，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。

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则都必须声明为transient的。

**78.考虑用序列化代理代替序列化实例**

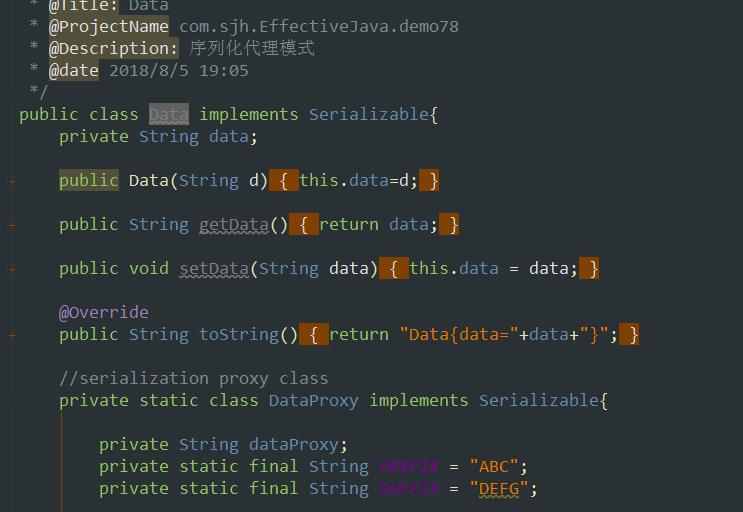
序列化代理模式相当简单：

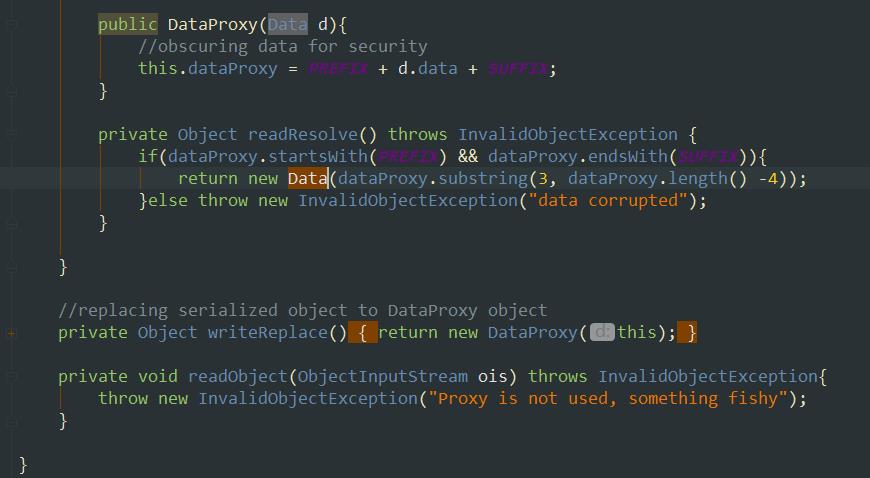
1）、首先，为可序列化的类设计一个私有的静态嵌套类，精确地表示外围类的实例的逻辑状态。这个嵌套类被称作序列化代理，它应该有一个单独的构造器，其参数类型就是那个外围类。外围类和序列化代理都必须声明实现Serializable接口。

2）、接下来，将writeReplace方法添加到外围类中，这个方法的存在导致序列化系统产生一个SerializationProxy实例，代替外围类的实例。

3）、再者，编写readObject方法，可以防止攻击者伪造该类的约束条件。

4）、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。





**EffectiveJava3新增**

**1.Lambda优先于匿名类**

函数接口：带有单个抽象方法的特殊接口。

编译器利用类型推导根据上下文去推断Lambda的类型、参数的类型及其返回值的类型。编译器是从泛型获取到得以执行类型推导的大部分类型信息的，如果没有提供这些信息，编译器就无法进行类型推导，就必须在Lambda中手工指定类型。

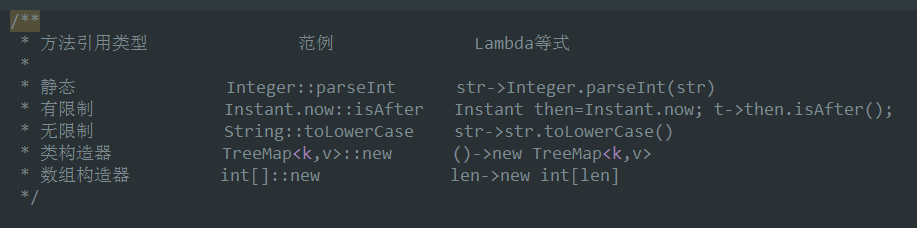
Lambda没有名称和文档；如果一个计算本身不是自描述的。或者超出了几行，就不要把他放在一个Lambda中，对于Lambda而言，三行是合理的最大极限。

Lambda限于函数接口，如果想创建抽象类的实例，可以用匿名类来完成，而不是用Lambda。Lambda无法获得对自身的引用，在Lambda中，关键字this是指外围实例，而在匿名类中，关键字this是指匿名类实例。

千万不要给函数对象使用匿名类，除非必须创建非函数接口的类型的实例。

**2.方法引用优先于Lambda**

许多方法引用都指向静态方法，但有4种没这么做，其中两个是有限制和无限制的实例方法引用。有限制引用中，接收对象是在方法引用中指定的，本质上类似于静态引用：函数对象与被引用的方法带有相同的参数；无限制引用中，接收对象是在运用函数对象时，通过在该方法的声明函数前面额外添加一个参数来指定的，无限制的引用经常用在流管道中作为映射和过滤函数；最后两种构造器引用，分别针对类和数组，构造器引用是在充当对象工厂。



只要方法引用更加简洁清晰，就用方法引用；如果方法引用并不简洁，就坚持使用Lambda。

**3.坚持使用标准的函数接口**

只要标准的函数接口能够满足需求，通常应该优先考虑，而不是专门再构建一个新的函数接口。

java.util.Function中共有43个接口，6个基础接口，基础接口作用于对象引用类型：

1）、**Operator**接口代表其结果与参数类型一致的函数：基础的为**UnaryOperator**接口，代表一个输入一个输出，类型一致；**BinaryOperator**接口代表两个输入一个输出，类型一致；

2）、**Predicate**接口代表带有一个参数并返回一个boolean的函数。

3）、**Function**接口代表一个输入一个输出，类型不一致。

4）、**Supplier**接口代表没有参数并且返回一个值的函数。

5）、**Consumer**接口代表一个参数但不返回任何值的函数。

这6个基础接口各种有3种变体，分别可以作用于int、long、double。现有的大多数标准函数接口都只支持基本类型，千万不要用带包装类型的基础函数接口来代替基本函数接口，使用装箱基本类型进行批量操作处理，最终会导致致命的性能问题。

必须使用@FunctionalInterface注解对自己编写的函数接口进行标注，它有三个目的：告诉这个类及其文档的作者，这个接口是针对Lambda设计的；这个接口不会进行编译，除非它只有一个抽象方法；避免后续维护人员不小心给该接口添加抽象方法。

**4.谨慎使用Stream**

Stream API提供了两个关键抽象：Stream（流）代表数据元素有限或无限的顺序；Stream pipeline（流管道）则代表这些元素的一个多级计算。

Stream中的元素可以来自任何位置，常见的来源包含：集合、数组、文件、正则表达式模式匹配器、伪随机数生成器以及其他Stream。Stream中的数据元素可以是对象引用，或者基本类型，支持三种基本类型：int、long、double。

一个Stream pipeline中包含一个源Stream，接着是0个或者多个中间操作和一个终止操作。每个中间操作都会通过某种方式对Stream进行转换，所有的中间操作都是将一个Stream转换成另一个Stream，其元素类型可能与输入的Stream一样，也可能不同。终止操作会在最后一个中间操作产生的Stream上执行一个最终的计算。Stream pipeline通常是lazy的：知道调用终止操作时才会开始计算，对于完成终止操作不需要的数据元素，将永远都不会被计算。

Stream API是流式的：所有包含pipeline的调用可以链接成一个表达式，在默认情况下，Stream pipeline是按顺序运行的，要使pipeline并发执行，只需在该pipeline的任何Stream上调用parallel方法即可。

有些任务最好用Stream完成，有些则需要迭代，如果不确定用Stream还是迭代比较好，那就两种都试试。

下列工作只能通过代码块而不能通过函数对象完成：

1）、从代码块中，可以读取或者修改范围内的任何局部变量；从lambda则只能读取final或者有效的final变量，并且不能修改任何local变量。

2）、从代码块中，可以从外围方法中return、break、continue外围循环，或者抛出该方法声明中要抛出的任何受检异常，从lambda中则完全无法完成这些事情。

**5.优先选择Stream中无副作用的函数**

Stream范型最重要的部分是把计算构造成一系列变型，每一级结果都尽可能靠近上一级结果的纯函数。纯函数是指其结果只取决于输入的函数：他不依赖任何可变的状态，也不更新任何状态。

静态导入Collectors的所有成员是惯例也是明智的，这样可以提升Stream pipeline的可读性。最重要的收集器工厂是toList、toSet、toMap、groupingBy、joining。没有理由使用collect(counting())，Stream接口有count()方法。forEach操作应该只用于报告Stream计算的结果，而不是执行计算。

**6.Stream要优先用Collection作为返回类型**

Collection接口是Iterable的一个子类型，它有一个stream方法，因此提供了迭代和stream访问，对于公共的、返回序列的方法，Collection或者适当的子类型通常是最佳的返回类型。千万别在内存中保存巨大的序列，将它作为集合返回即可。

**7.谨慎使用Stream并行**

如果源头是来自Stream.iterable，或者使用了中间操作的limit，那么并行pipeline也不可能提升性能。千万不要任意的使用并行Stream pipeline。

在Stream上通过并行获得的性能，最好是通过ArrayList、HashMap、HashSet和ConcurrentHashMap实例、数组、int范围和long范围等。这些数据结构的共性：都可以被精确轻松的分成任意大小的子范围；另一重要特性是它们提供了优异的引用局部性：序列化的元素引用一起保存在内存中。

Stream pipeline的终止操作本质上也会影响并行效率，如果大量工作是在终止操作中完成，而不是在中间操作中完成，并且这个操作有固有的顺序，并行效率就会受到限制。并行的最佳终止操作是做减法，用一个Stream的reduce方法，将所有从pipeline产生的元素都合并在一起，或者预先打包像min、max、count和sum这类方法。由Stream的collect方法执行的操作，都是可变的减法，不适合并行，因为并行集合的成本非常高。

并行Stream不仅可能降低性能，包括活性失败，还可能导致结果出错，以及难以预计的行为。在适当的条件下，给Stream pipeline添加parallel调用，确实可以在多处理器核的情况下实现近乎线性的倍增，如机器学习和数据处理。