**《Java编程思想》笔记:**

**一.单根继承结构优势:**

1.在单根继承中所有对象共用一个接口,所以归根结底都是相同的基本类型.

2.单根继承结构保证了所有对象都具备某些功能.

3.单根继承结构使垃圾回收器的实现变的容易的多(因为能够明确对象类型).

**二.终结清理和垃圾回收:**

1.如果在类中定义了finalize()方法(要尽量避免使用该方法),当垃圾器准备释放对象所用内存时,会首先调用该方法,在下一次垃圾回收动作发生时真正的回收对象占用的内存.

2.对象可能不被垃圾回收; 垃圾回收并不等于”析构”.(“析构”: c++中的概念,用于销毁对象.); 垃圾回收只于内存有关.

3.无论是”垃圾回收”还是”终结”,都不保证一定发生.如果Java虚拟机(JVM)并未面临内存耗尽的的情形,它是不会浪费时间去执行垃圾回收以恢复内存的.

4.System.gc():用于强制执行终结动作,在类中定义finalize()方法验证终结条件.

**三.初始化:**

1.成员初始化:

对于局部变量如果不初始化,当使用时会报错,编译不通过.但是对于类的成员(字段),基本类型都有默认值0,对象引用则为null.

2.构造器初始化:

及时变量定义散布于方法定义之间,他们仍然会在任何方法(包括构造器)之前得到初始化.

3.静态初始化:

静态初始化只有在必要时刻才会进行,也就是创建对象或者第一次访问数据时,他们才会被初始化,此后不会再次初始化.

初始化的顺序是先静态对象,然后非静态对象,先父类,后子类。

**父类静态字段初始化->父类静态块->子类静态字段初始化->子类静态块->父类非静态字段初始化->父类构造器->子类非静态字段初始化->子类构造器**

**四.对象的创建过程总结(以Dog类为例):**

**1.即使没有显示的使用static关键字,构造器也是静态方法.当首次创建Dog对象时,获取Dog类的静态方法或静态域被访问时,Java解释器必须查找类路径,以定位Dog.class文件.**

**2.然后载入Dog.class文件,有关静态初始化的所有动作都会执行.因此,静态初始化只在Class对象首次加载的时候进行一次.**

**3.当用new Dog()创建对象的时候,首次将在堆上为Dog对象分配足够的存储空间.**

**4.这块存储空间会被清零,这就自动的将Dog对象中的所有基本类型数据都设置成了默认值(对数字来说是0,对布尔型和字符型也相同),而引用则被设置成null.**

**5.执行所有出现于字段定义处的初始化动作.**

**6.执行构造器.**

**五.数组初始化:**

数组是相同数据类型的、用一个标识符名称封装到一起的一个对象序列活基本类型数据序列。数组是通过方括号下标操作符[]来定义和使用的.

自动的编译器错误和运行时优化都可以提高数组访问的速度.

六.可变参数列表:

要放在参数的最后;一个方法只能有一个;可以传入一个值或多个值,会自动合并成数组;可变参数列表不依赖于自动包装机制,实际上使用的是基本类型,当然可变参数列表可以与自动包装机制和谐相处.

**七.final关键字:**

使用final的三种情况：数据、方法、类。

1.final数据:

永不改变的编译期常量;在运行时被初始化的值;可以是基本类型和String类型（基本类型值不可以改变，引用类型地址值不可以改变，对象中的属性可以改变）;既是static又是final的域只占据一段不能改变的存储空间.

空白final:

Java允许生成"空白final",所谓空白final是指被声明为final但又未给定初始值的域.无论什么情况,编译器都确保空白final在使用前必须被初始化.必须在域的定义处或者每个构造器中用表达式对final进行赋值,这正是final域在使用前总是被初始化的原因所在.



final参数:

Java允许在参数列表中以声明的方式将参数指明为final,这意味着无法在方法中更改参数引用所指向的对象.

2.final方法:

被final修饰的方法不可以被覆盖.这样做还可以有效的"关闭"动态绑定.

3.final类:

被final修饰的类不可以被继承.

**八.内部类:**

1.如果想从外部类的非静态方法之外的任意位置创建某个内部类的对象,那么必须像在main()方法中那样,具体的指明这个对象的类型:OutClassName.InnerClassName.



2.当生成一个内部类的对象时,此对象与制造它的外围对象之间就有了一种联系,所以它能访问其外围对象的所有成员,而不需要任何特殊条件,此外,内部类还拥有其外部类的所有元素的访问权。这是因为内部类对象获取到了一个指向外部类对象的引用,当访问外部类的成员时，就是用这个引用来选择外部类的成员。外部类不可以直接获取内部类的成员。





3.如果需要生成对外部类对象的引用，可以使用外部类的名字后面紧跟远点和this。这样产生的引用自动的具有正确的类型，这点在编译期就被知晓并受到检查，没有任何运行时开销。



嵌套类（静态内部类）：



4.在方法和作用域内的内部类:

4.1.一个定义在方法中的类

4.2.一个定义在作用域内的类,此作用域在方法的内部

4.3.一个实现了接口的匿名类

4.4.一个匿名类,它扩展了有非默认构造器的类

4.5.一个匿名类,它执行字段初始化

4.6.一个匿名类,它通过实例初始化实现构造(匿名类不可能有构造器)

匿名内部类与正规的继承相比有些受限，因为匿名内部类既可以扩展；类，也可以实现接口，但是不能两者兼备。而且如果是实现接口，只能实现一个接口。匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。匿名内部类中不能存在任何的静态成员变量和静态方法。





匿名内部类和方法和作用域内的内部类实现字段初始化得到的值必须是final的.内部类并不是直接调用方法传递的参数，而是利用自身的构造器对传入的参数进行备份(还有外部类对象引用)，自己内部方法调用的实际上是自己的属性而不是外部方法传递进来的参数。简单理解就是，拷贝引用，为了避免引用值发生改变，例如被外部类的方法修改等，而导致内部类得到的值不一致，于是用final来让该引用不可改变。故如果定义了一个匿名内部类，并且希望它使用一个其外部定义的参数，那么编译器会要求该参数引用是final的。主要就是为了保持该值在内外类中的一致性。(但是如果是全局变量的话可以不用是final,该变量是通过外部类对象引用(this)得到的,非静态内部类的对象都保存了外部类的对象的引用，因而内部类中对任何外部类成员变量的修改都会真实地反应到外部类实例本身，所以不需要用final来修饰它。)





接口内部的类：



多层嵌套类：



5.闭包与回调:

闭包：是一个可调用的对象,它记录了一些信息,这些信息来自于创建它的作用域。通过定义可以看出，内部类是面向对象的闭包，因为它不仅仅包含外部类对象的信息，还自动拥有一个指向外部类对象的引用，在此作用域内，内部类有权操作所有的成员，包括private成员。

回调：通过回调，对象能够携带一些信息，这些信息允许它稍后的某个时刻调用初始对象。

**九.Collection与Map（基础）:**

1、List:

1.1.ArrayList长于随机访问元素,但是在List的中间插入和移除元素时较慢。

1.2.LinkedList（双向链表），它通过代价较低的在List中间进行插入和删除操作，提供了优化的顺序访问。LinkedList在随机访问方面相对较慢，但它的特性集较ArrayList更大。

2、迭代器：

Java的Iterator只能单向移动，这个Iterator能用来：

1）、使用方法iterator（）方法要求容器返回一个Iterator；

2）、使用next（）获得序列中的下一个元素；

3）、使用hasNext（）检查序列中是否还有元素；

4）、使用remove（）将迭代器新近返回的元素删除。

ListIterator是一个更加强大的Iterator的子类型，他能用于各种List类的访问，可以双向移动。

3、Stack（栈）：

先进后出的容器，最后压入的元素第一个弹出栈。

4、Set：

元素不允许重复；无序。

HashSet提供最快的查询速度;TreeSet保持元素处于排序状态;LinkedHashSet以插入顺序保持元素。

5、Map：

键值对Entry数组。

6、Queue：

队列是先进先出容器。

7、foreach与迭代器：

foreach可以用于数组、任何Collection对象、任何实现Iterable的类、可以产生Iterable对象的方法。

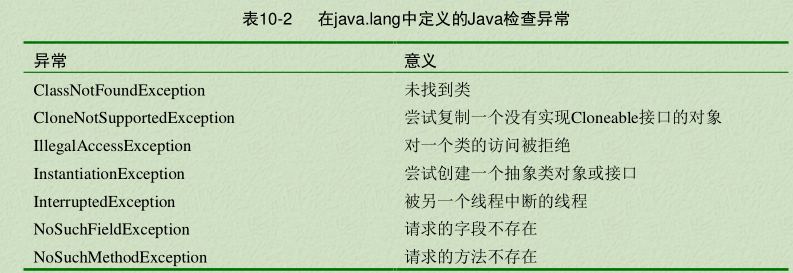


**十.Java异常机制:**

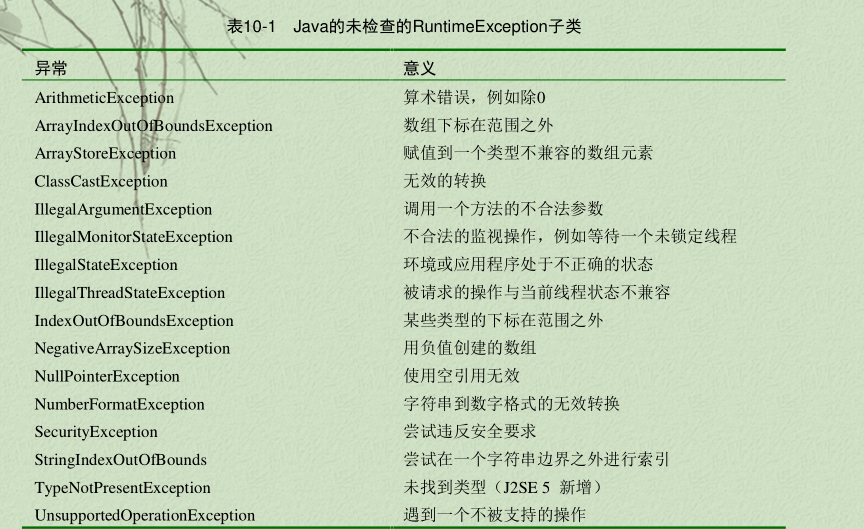
**Throwable**这个Java类被用来表示任何可以作为异常被抛出的类。分为两种类型：Error用来表示编译时和系统错误；Exception是可以被抛出的基本类型。

异常还可分为检查异常和未检查异常。

检查异常：在编译时被强制检查的异常。



未检出异常：在运行时抛出的异常。都继承自RuntimeException类。



如果有finally子句，无论是否抛出异常，finally子句中的代码都会被执行。当要把除内存之外的资源恢复到它们的初始状态时，就要用到finally子句。这种需要清理的资源包括：已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界的某个开关。

**十一.String:**

1、String对象不可变。String类中的每一个看起来会修改String值的方法，实际上都是创建了一个全新的String对象以及包含修改后的字符串内容，最初的String对象丝毫未动。如果原先的内容没有发生改变，则会返回指向原对象的引用。

String对象具有只读特性，指向它的任何引用都不可能改变它的值。

String对象使用"+"操作符时,编译器会自动进行优化，创建StringBuilder对象，然后调用append（）方法，最后调用toString（）方法返回结果。

2、格式化：





**十二.类型信息:**

1、运行时类型信息使得可以在程序运行时发现和使用类型信息。

如果某个对象出现在字符串表达式中（涉及“+”和字符串对象的表达式），toString（）方法就会被自动调用，以生成表示该对象的String。

在Java中，所有的类型转换都是在运行时进行正确性检查的。

2、Class对象：

2.1、类是程序的一部分，每个类都有一个Class对象，每当编写或者编译了一个新类，就会产出一个Class对象，为了产生这个类的对象，运行这个程序的Java虚拟机将使用被称为“类加载器”的子系统。类加载器子系统实际上可以包含一条类加载器链，但是只有一个原生类加载器，它是JVM实现的一部分。

所有的类都是在对其第一次使用时动态加载到JVM中的。当程序创建第一个对类的静态成员的引用时，就会加载这个类。这个证明构造器也是累的静态方法，即使在构造器之前并没有使用static关键字，因此，使用new操作符创建类的新对象也会被当做对类的静态成员的引用。

Java程序在它开始运行之前并非被完全加载，其各个部分是在必需时才加载的（static初始化伴随着类加载进行）。类加载器首先检查这个类的Class对象是否已经加载，如果尚未加载，默认的类加载器就会根据类名查找.class文件，在这个类的字节码被加载时，他们会接受验证，以确保没有被破坏，并且不包含不良Java代码。一旦某个类的Class对象被载入内存，它就被用来创建这个类的所有对象。

2.2、Class.forName（）方法用于获取Class对象的引用；无论何时，如果想在运行时使用类型信息，就必须首先获得对恰当的Class对象的引用。

2.3、使用“.class”来创建对Class对象的引用时，不会自动的初始化该Class对象。也就说只有等到使用该引用时才会进行初始化动作。

2.4、Class引用总是指向某个Class对象，他可以制造类的实例，并包含可作用于这些实例的所有方法代码。它还包含该类的静态成员，因此，Class引用表示的就是它所指向的对象的确切类型，而该对象便是Class类的一个对象。



2.5、instanceof:

返回一个布尔值,用于判断对象是不是某个特定类型的实例。

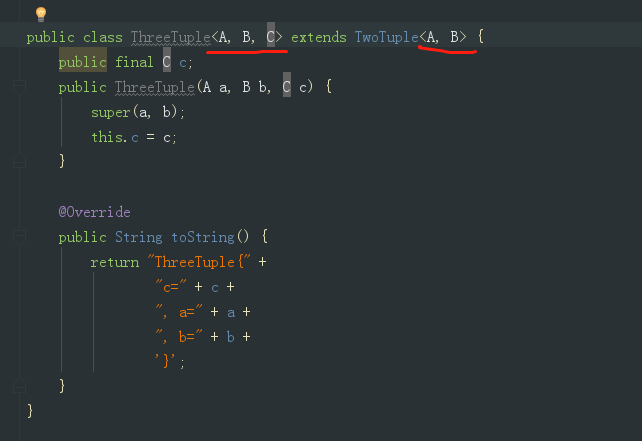
instanceof与==的区别:

instanceof保持了类型的概念,它指的是"你是这个类吗,或者你是这个类的派生类吗",而如果用==则比较的是实际的Class对象,就没有考虑继承。

**十三.泛型:**

1、泛型：参数化类型。

2、元组：将一组对象直接打包存储于其中的一个单一对象。这个容器对象允许读取其中对象，但是不允许向其中存放新的对象。元组可以具有任意长度；元组中的对象可以是任意不同的类型。



3、泛型方法：

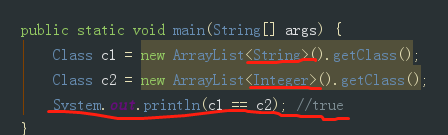
泛型方法使得方法能够独立于类而产生变化，对于一个static方法而言，无法访问泛型类的类型参数，如果一个static方法需要使用泛型能力，就必须使其成为泛型方法。

当使用泛型类时，必须在创建对象的时候指定类型参数的值，而使用泛型方法的时候，通常不必指明参数类型，因为编译器会为我们找出具体的类型，这称为类型参数推断。

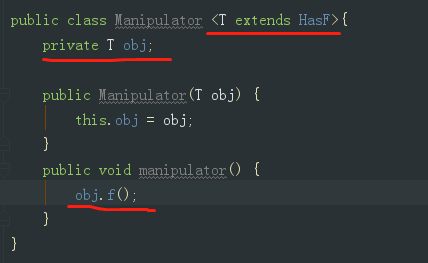
4、泛型擦除：

在泛型代码内部，无法获得任何有关泛型参数类型的信息。

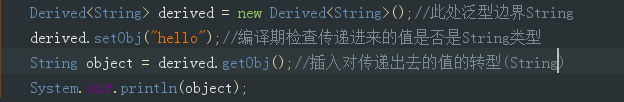
在使用泛型时，任何具体的类型信息都在编译期被擦除了，因此List<String>和List<Integer>在运行时事实上是相同的类型,这两种形式被擦除成它们的原生类型,即List。

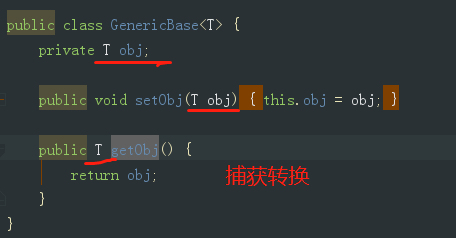


泛型类型参数将擦除到它的第一个边界，例如<T extends HasF>，编译器会把类型参数替换为它的擦除，T擦除到了HasF，就好像在类的声明中用HasF替换了T一样。



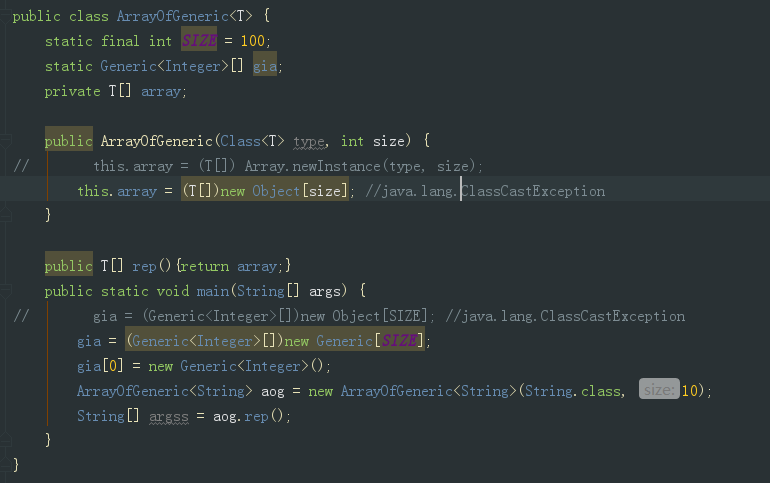
在泛型中的所有动作都发生在边界处--对传递进来的值进行额外的编译器检查,并插入对传递出去的值的转型。“边界就是发生动作的地方”。



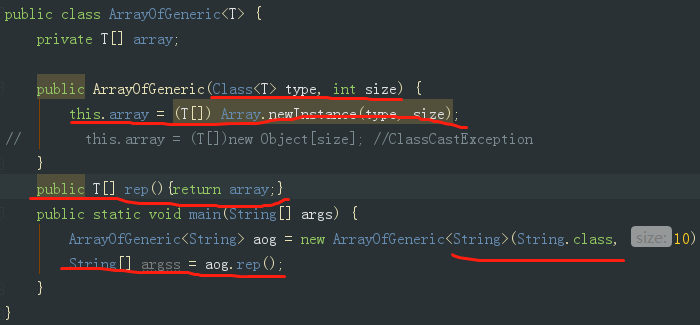


5、泛型数组：

数组将跟踪它们的实际类型，而这个类型是在数组被创建时确定的，即使gia已经被转型为Generic<Integer>[]，但是这个信息只存在于编译期，用于类型检查。在运行时，它仍旧是Object数组。成功创建泛型数组的唯一方式就是创建一个被擦除类型的新数组，然后对其转型。

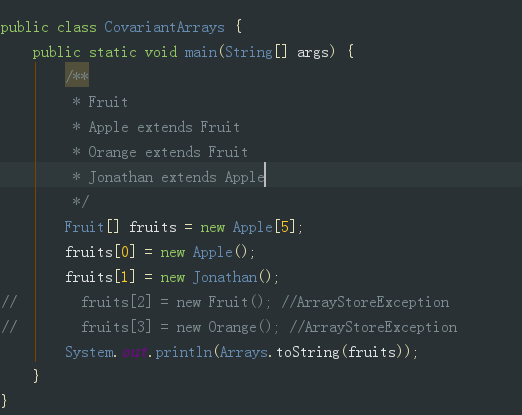


类型标记Class<T>被传递到构造器中,以便从擦除中恢复,使得我们可以创建需要的实际类型的数组，一旦我们获得了实际类型，就可以返回它，并获得想要的结果，该数组的运行时类型是确切类型T[]。



6、通配符：

6.1、数组是协变的，泛型以及泛型集合不是协变的。



6.2、? Super T和? Extend T的区别:

**PECS**

请记住PECS原则：生产者（Producer）使用extends，消费者（Consumer）使用super。

· 生产者使用extends

如果你需要一个列表提供T类型的元素（即你想从列表中读取T类型的元素），你需要把这个列表声明成<? extends T>，比如List<? extends Integer>，因此你不能往该列表中添加任何元素。

· 消费者使用super

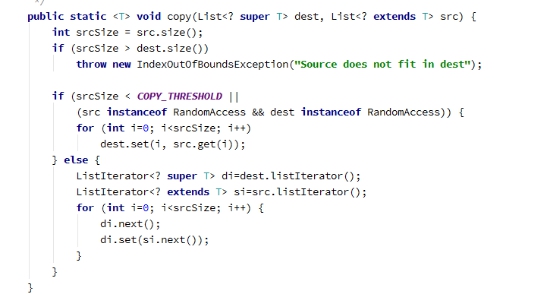
如果需要一个列表使用T类型的元素（即你想把T类型的元素加入到列表中），你需要把这个列表声明成<? super T>，比如List<? super Integer>，因此你不能保证从中读取到的元素的类型。

· 即是生产者，也是消费者

如果一个列表即要生产，又要消费，你不能使用泛型通配符声明列表，比如List<Integer>。

? Extend T描述了通配符的上界,即?为T本身或子类

? Super T描述了通配符的下界,即?为T本身或父类



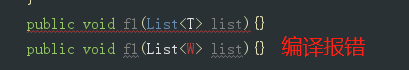
7、问题：

7.1、任何基本类型都不能作为类型参数。

7.2、一个类不能实现同一个泛型接口的两种变体，由于擦除的原因，这两个变体会成为相同的接口。

7.3、使用带有泛型类型参数的转型或instanceof不会有任何效果。

7.4、由于擦除的原因，带有泛型的方法参数如果产生唯一的参数列表时，重载方法将产生相同的类型签名，必须提供有明显区别的方法名。



**十四.数组:**

1、数组：元素的集合，使用整型索引值访问元素,尺寸不可以改变。

无论使用哪种类型的数组，数组标识符其实只是一个引用，指向在堆中创建的一个真实对象，这个（数组）对象用以保存指向其他对象的引用。可以作为数组初始化语法的一部分隐式地创建此对象，或者用new表达式显式地创建。只读成员length是数组对象的一部分，表示此数组对象可以存储多少元素。“[]”语法是访问数组对象的唯一方式。

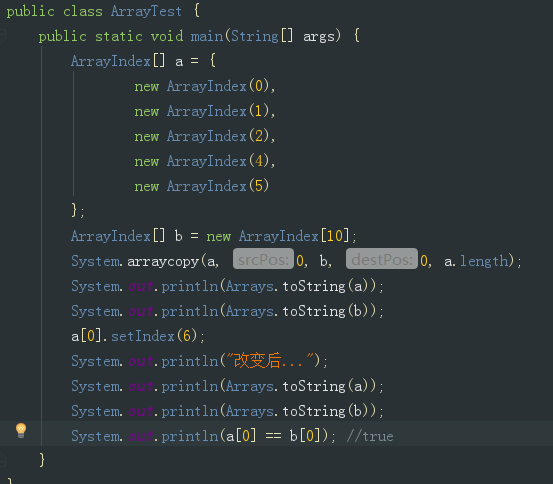
对象数组和基本类型数组在使用上几乎是相同的，唯一的区别就是对象数组保存的是引用，基本类型数组直接保存基本类型的值。

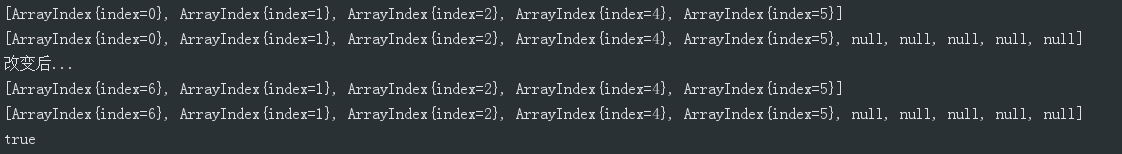
2、不可以实例化具有参数化类型的数组。擦除会移除参数类型信息，数组必须知道他们所持有的确切类型，以强制保证类型的安全。

3、Arrays使用功能：

equals（）用于比较两个数组是否相等（deepEquals（）用于多维数组）；fill（）填充数组；sort（）用于对数组排序；binarySearch（）用于在已经排序的数组中查找元素；toString（）产生数组的String表示；hashCode（）产生数组的散列码；asList（）方法接收任意的序列或数组作为参数，并将其转变为List容器。System.arraycopy（）用于复制数组，但是不会执行自动包装和自动拆包，两个数组必须具有相同的确切类型。

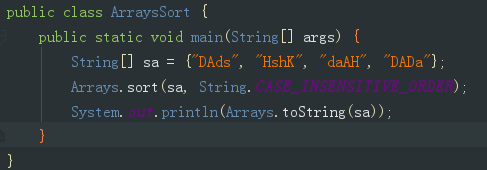
System.arraycopy（）如果复制的是对象数组，那么只是复制了对象的引用，而不是对象的拷贝，这是浅复制。



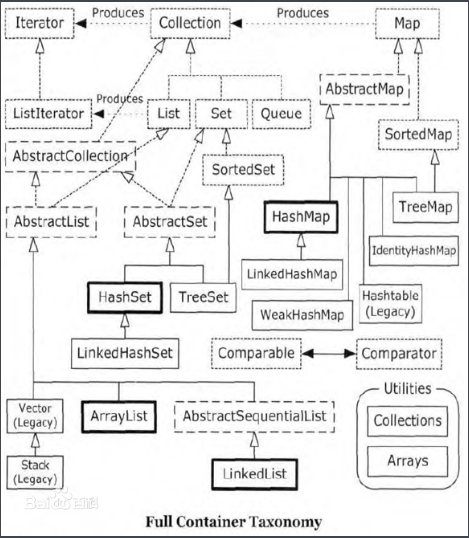


Arrays.sort（）用于对数组排序：

String排序算法依据词典编排顺序排序，所以大写字母开头的词都放在前面输出，然后才是小写字母开头的词。如果想忽略大小写，可以使用String.CASE\_INSENSITIVE\_ORDER。



**十五.Collection与Map（深入）:**



1、Set：

HashSet：为快速查找而设计的Set，底层是HashMap。默认推荐使用。

TreeSet：保持次序的Set，底层是TreeMap。

LinkedHashSet：具有HashSet的查询速度，且内部使用链表维护元素的顺序（插入的次序）。底层是LinkedHashMap。

SortedSet：保证元素可以处于排序状态。

Object first（）返回容器中的第一个元素。

Object last（）返回容器中的最末一个元素。

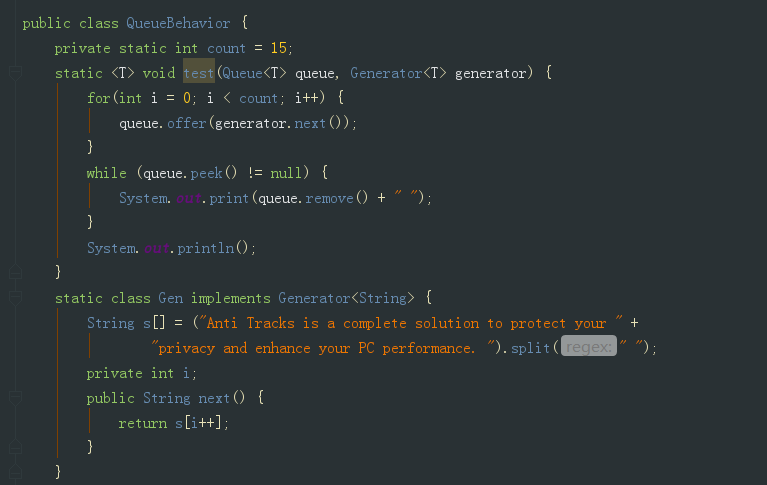
SortedSet subSet（fromElement，toElement）生成此Set的子集，范围从 fromElement（包含）到toElement（不包含）。

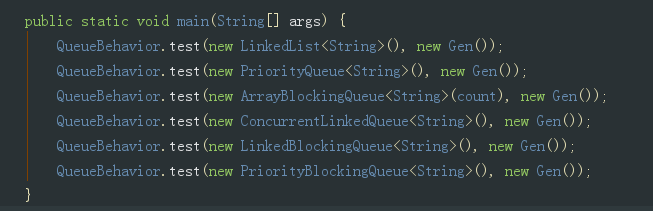
SortedSet headSet（toElement）生成此Set的子集，由小于toElement的元素组成。

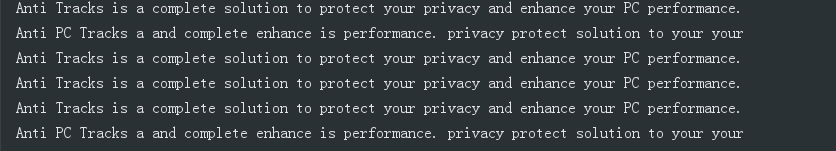
SortedSet tailSet（fromElement）生成此Set的子集，由大于或等于fromElement的元素组成。

2、Queue：

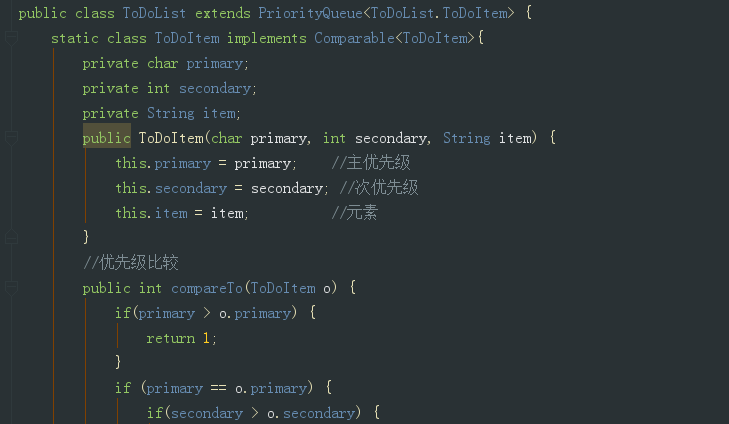
除了优先级队列，Queue将精确的按照元素被置于Queue中的顺序产生它们。

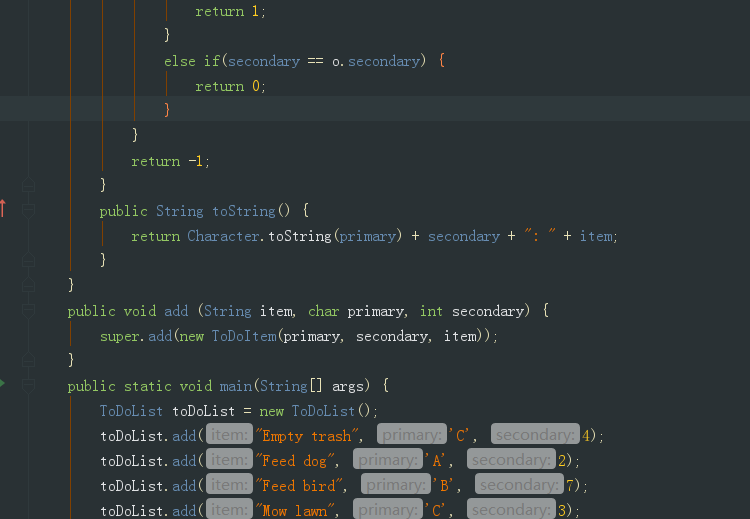


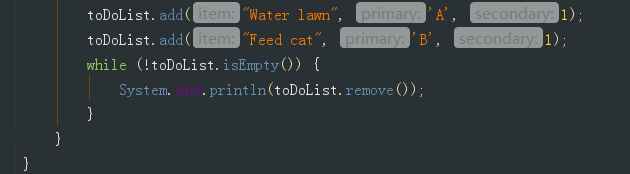




优先级队列:







3、Map：

HashMap：Map基于散列表的实现，插入与查询“键值对”的开销是固定的。可以通过构造器设置容量和负载因子，以调整容器的性能。

LinkedHashMap：类似于HashMap，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用（LRU）的次序。只比HashMap慢一点；而在迭代访问时反而最快，因为它使用链表维护内部次序。

TreeMap：基于红黑树的实现。查看“键”或“键值对”时，它们会被排序（次序由Comparable或Comparator决定）。TreeMap的特点在于，所得到的结果是经过排序的。TreeMap是唯一的带有subMap（）方法的Map，他可以返回一个子树。

WeakHashMap：弱键映射，允许释放映射所指向的对象；这是为解决某类特殊问题而设计的。如果映射之外没有引用指向某个“键”，则此“键”可以被垃圾回收器回收。

ConcurrentHashMap：一种线程安全的Map，它不涉及同步加锁。

IdentityHashMap：使用==代替equals（）对“键”进行比较的散列映射。

SortedMap：确保“键”处于排序状态，TreeMap是其现阶段的唯一实现。Comparator comparator（）：返回当前Map使用的Comparator ，或者返回null，使用“键”当前类型实现的方式。T firstKey（）返回Map中的第一个键。T lastKey（）返回Map中的最后一个键。SortedMap subMap（fromkey， toKey）：生成此Map的子集，范围由fromKey（包含）到toKey（不包含）。SortedMap headMap（toKey）：生成此Map的子集，由键小于toKey的所有键值对组成。SortedMap tailMap（fromKey）：生成此Map的子集，由键大于或等于fromKey的所有键值对组成。

4、散列与散列码：

使用散列的目的：使用一个对象来查找另一个对象。

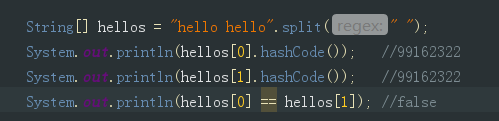
散列的价值在于速度：散列使得查询得以快速进行。

设计hashCode（）：

1）、无论何时，对同一个对象调用hashCode（）都应该生成同一的值。

2）、不应该使hashCode（）依赖于具有唯一性的对象信息。

String有个特点：如果程序中有多个String对象，都包含相同的字符串序列，那么这些String对象都映射到同一块内存区域。所有new String（“hello”）生成的两个实例，虽然都是相互独立的，但是对他们使用hashCode（）应该生成相同的结果。



设计hashCode（）步骤：

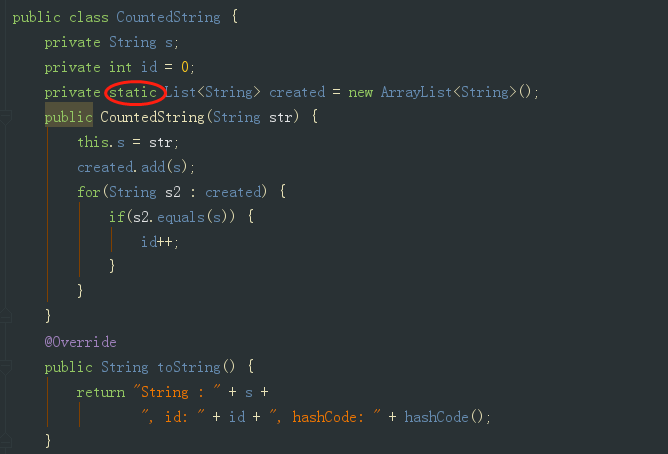
1）、给int变量result赋予某个非零值常量，例如17；

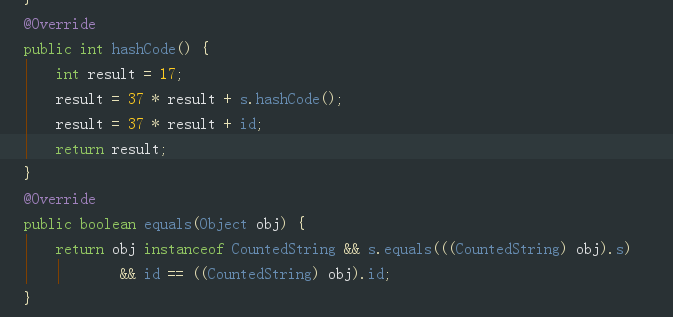
2）、为对象内每个有意义的域f（即每个可以做equals（）操作的域）计算出一个int散列码c；

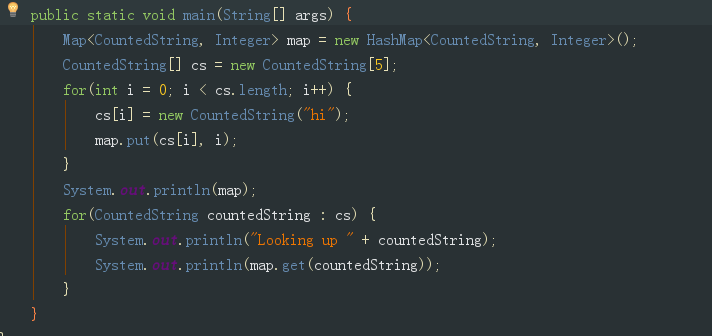
3）、合并计算得到的散列码：result=37\*result+c；

4）、返回result；

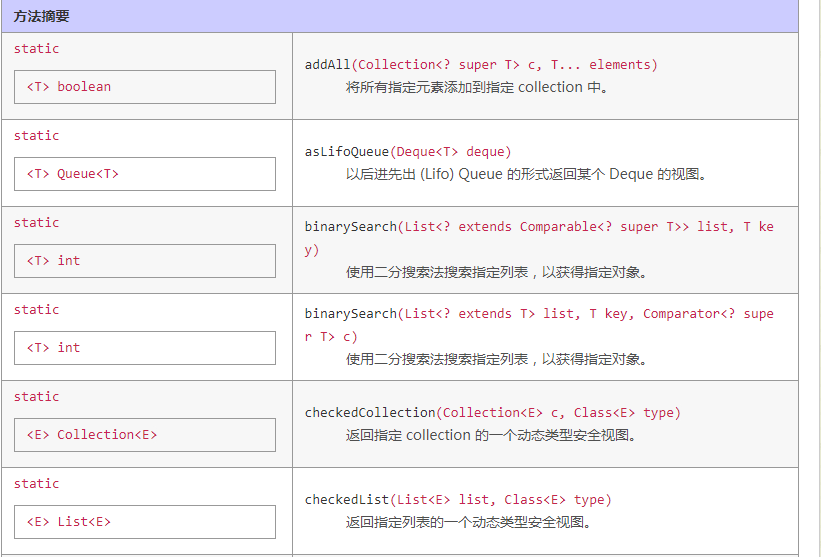
5）、检查hashCode（）最后生成的结果，确保相同的对象有相同的散列码。







5、Collections：

















**十六.IO流:**

1、InputStream：

InputStream的作用是用来表示那些从不同数据源产生输入的类,这些数据源包括:

1）、字节数组；

2）、String对象；

3）、文件；

4）、“管道”，工作方式与实际管道相似，即从一端输入从一端输出；

5）、一个由其他种类的流组成的序列，以便我们可以将它们收集合并到一个流内；

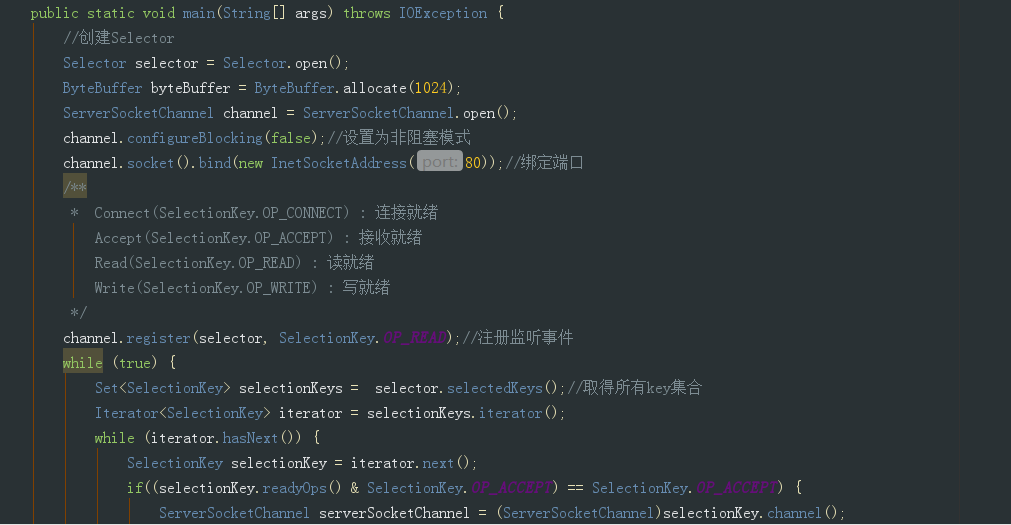
6）、其他数据源，如Internet连接。

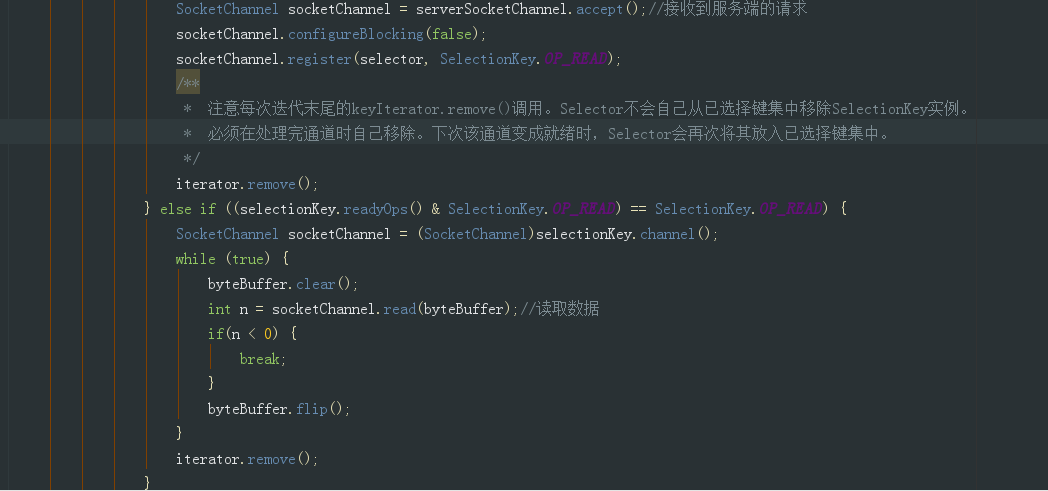
2、OutputStream：

OutputStream决定了输出所要去往的目标：字节数组（不是String，不过可以用字节数组自己创建）、文件或管道。

**3、NIO（**[**http://ifeve.com/overview/**](http://ifeve.com/overview/)**）：**

1）、NIO的执行方式：通道和缓冲器。核心类：Channel，Buffer 和 Selector。Selector（选择器）是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件做好准备的组件。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。





2）、可以想象成一个煤矿，通道是一个包含煤层（数据）的矿藏，而缓冲器则是派送到矿藏的卡车，卡车满载而归，我们再从卡车上获得煤炭。也就是说，我们并没有直接和通道交互，我们只是和缓冲器交互，并把缓冲器派送到通道，通道要吗从缓冲器获得数据，要吗向缓冲器发送数据。

缓冲器容纳的是普通的字节，为了把他们转换成字符，我们要吗在输入他们的时候对其进行编码，这样它们在输出时才具有意思，要吗在将其从缓冲器输出时对它们进行解码。

3）、向ByteBuffer插入基本类型数据的最简单的方法是：利用asCharBuffer（）、asShortBuffer（）等获得该缓冲器上的视图，然后使用视图的put（）方法。

视图缓冲器可以让我们通过某个特定的基本数据类型的视窗查看其底层的ByteBuffer。ByteBuffer依然是实际存储数据的地方，“支持”着前面的视图，因此，对视图的任何修改都会映射成为对ByteBuffer中数据的修改。

4）、字节存放次序：

不同的机器可能会使用不同的字节排序方法来存储数据。“big endian‘（高位优先）将最重要的字节存放在地址最低的存储器单元。而”little endian“（低位优先）则是将最重要的字节存放在地址最高的存储器单元。当存储量大于一个字节时，像int、float等，就要考虑字节的顺序问题了。ByteBuffer是以高位优先的形式存储数据的，并且数据在网上传送时也常常使用高位优先的形式。例：

short形式读取数据97以16位表示（前面补0）：97 -> 0000000001100001。

如果改成低位优先，则变成：97 -> 0110000100000000 -> 24832。

5）、Buffer由数据和可以高效访问及操纵这些数据的四个索引组成，这四个索引是：mark（标记）、position（位置）、limit（界限）、capacity（容量）。

6）、内存映射文件：



对内存映射文件加锁：





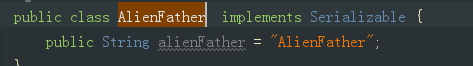
4、对象序列化：

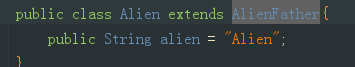
1）、Java的对象序列化将那些实现了Serializable接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。这一过程甚至可以用个网络进行。

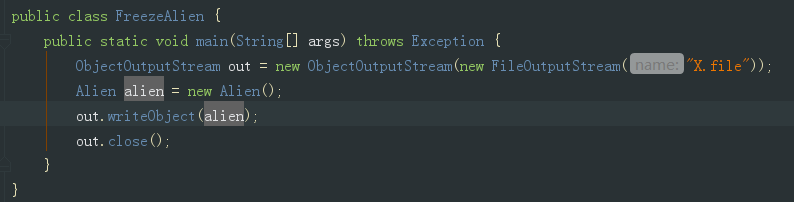
利用对象的序列化可以实现轻量级持久性，“持久性”意味着一个对象的生存周期并不取决于程序是否在运行，他可以生存于程序的调用之间。通过将一个序列化对象写进磁盘，然后在重新调用程序时恢复该对象，就能够实现持久性的效果。之所以称为“轻量级”，是因为不能用某种“persistent”（持久）关键字来简单地定义一个对象，并让系统自动维护其他细节问题。相反，对象必须在程序中显式地序列化（serialize）和反序列化还原（deserialize）。

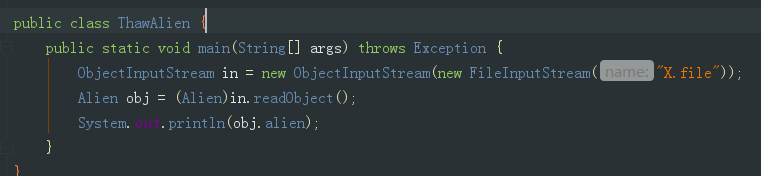
2）、实现Serializable接口进行序列化：

当某一类有父类且父类实现了Serializable接口，本类可以不用显示的实现Serializable接口也可以进行序列化。

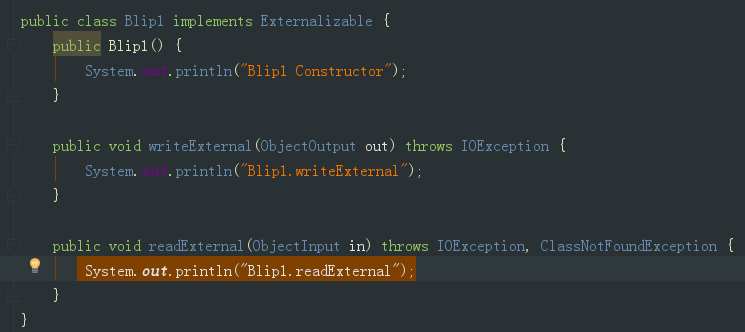


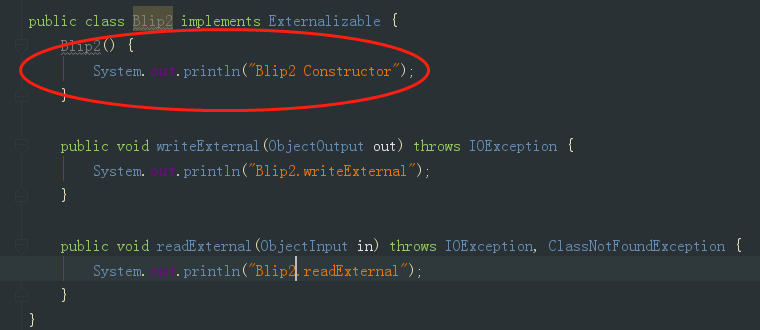


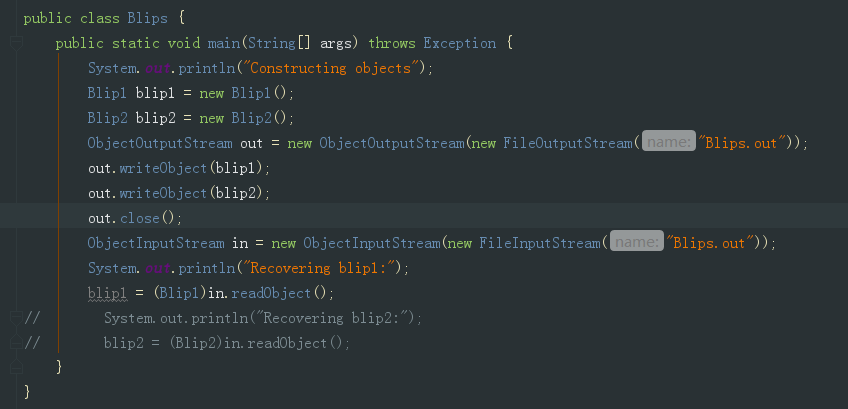




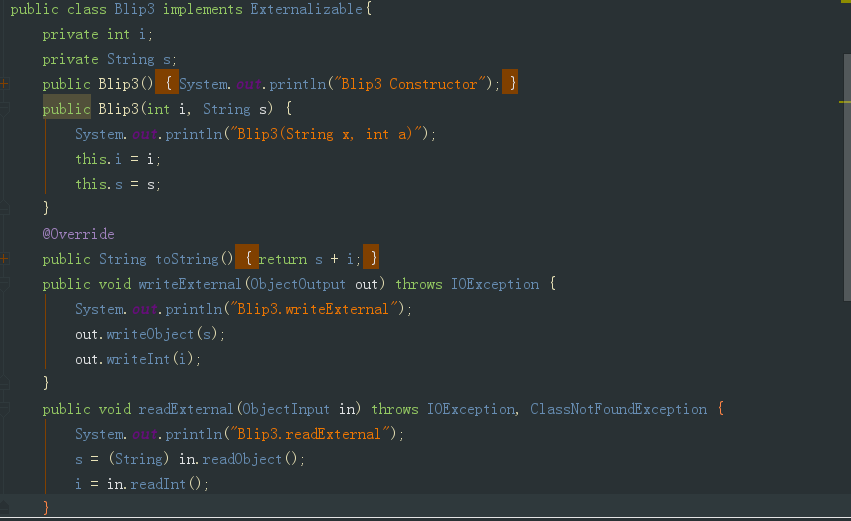
3）、实现Externalizable接口进行序列化，可以对序列化进行控制：

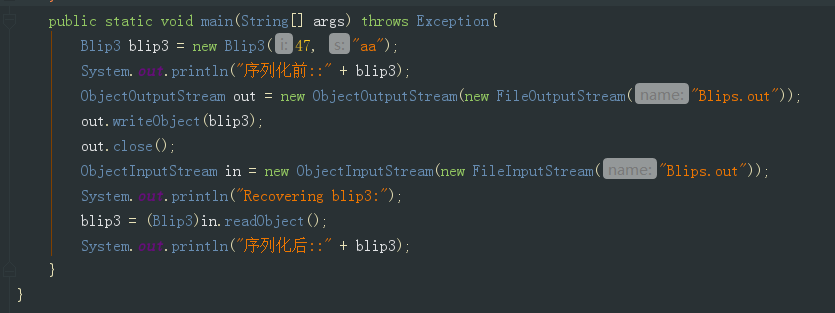






上例中没有恢复Blip2对象，因为会导致异常。Blip1的构造器是public的，而Blip2的构造器是默认的，这样就会在恢复时造成异常。恢复Blip1对象时，会调用默认构造器，这与恢复一个Serializable对象不同，对于Serializable对象，对象完全以它存储的二进制位为基础来构造，而不调用构造器。而对于一个Externalizable对象，所有普通的默认构造器都会被调用（包括在字段定义时的初始化），然后调用readExternal。

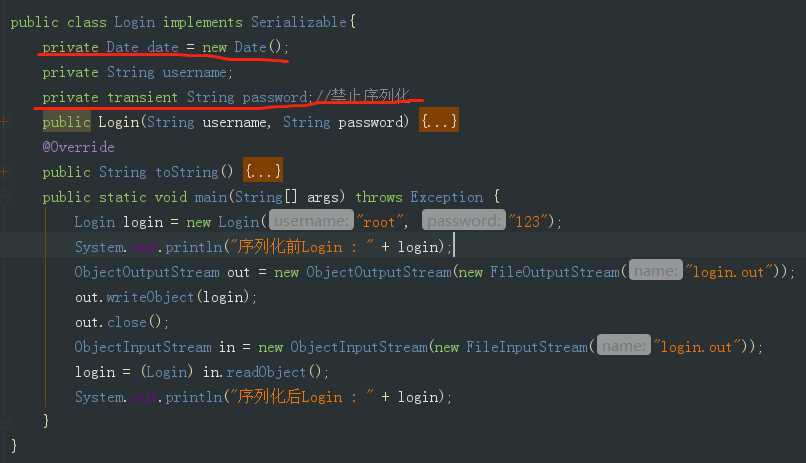




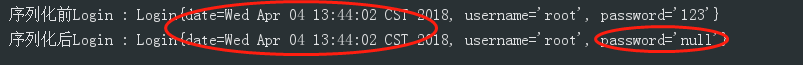
上例中，其中s和i只在第二个构造器中初始化，不是在默认的构造器初始化，这意味着如果不在readExternal（）中初始化s和i，s就会为null，i就会为0，这是因为在创建对象的第一步中将对象的存储空间清理为0（Externalizable对象，所有普通的默认构造器都会被调用，然后调用readExternal）。

4）、transient（瞬时）关键字

如果操作的是Serializable对象,那么所有序列化操作都会自动进行.为了能够予以控制,可以使用transient（瞬时）关键字逐个字段的关闭序列化。由于Externalizable对象在默认情况下不保存任何字段，所以transient关键字只能和Serializable对象一起使用。



date字段没有重新生成。

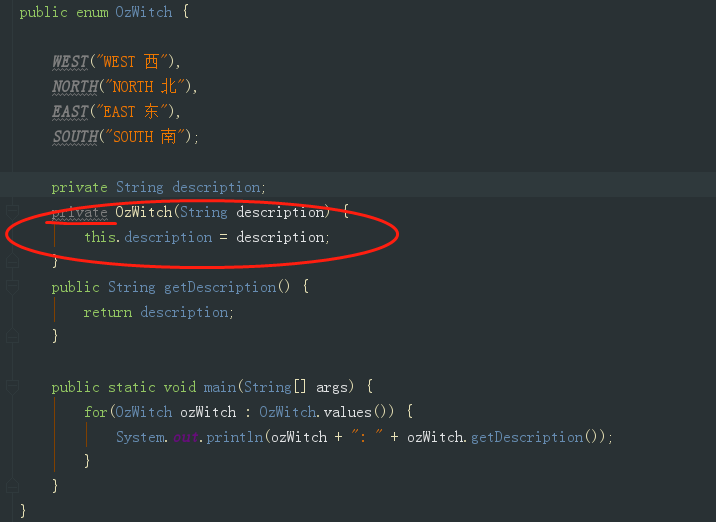


5）、static字段不可以被序列化。

**十七.枚举enum：**

1、除了不能继承自一个enum外，基本上可以把enum看做一个常规的类。

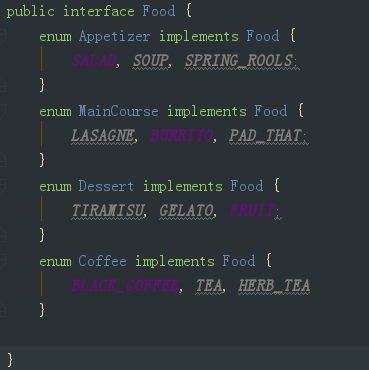
2、enum中构造器与方法和普通的类没有什么区别，不过下边的例子中我们把构造器声明为private，这对于它的可访问下没有什么影响，因为即使不声明为private，我们也只能在enum定义的内部使用其构造器创建enum实例。一旦enum定义结束，编译器就不允许在使用其构造器来创建任何实例了。

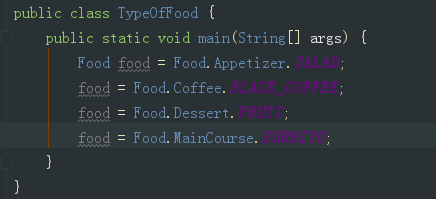


3、所有的enum都继承自Enum类，所以enum不可以再继承其他类，但是可以实现一个或多个接口。

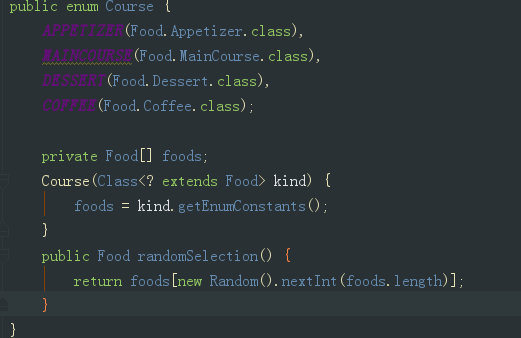
4、使用接口组织枚举：

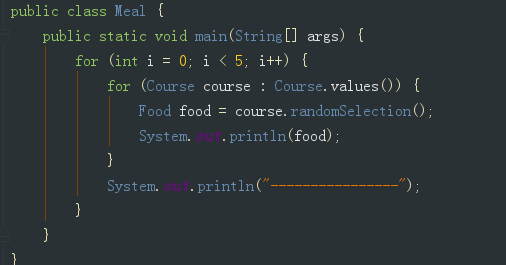
1）、





2）、

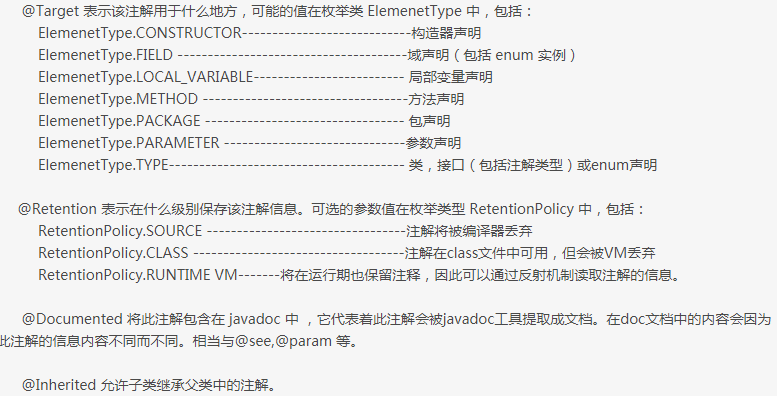




**十八.注解:**

1、注解也称元数据，为我们在代码中添加信息提供了一种形式化的方法，使我们可以在稍后的某个时刻非常方便的使用这些数据。

2、四种元注解：



3、注解元素可用的类型：

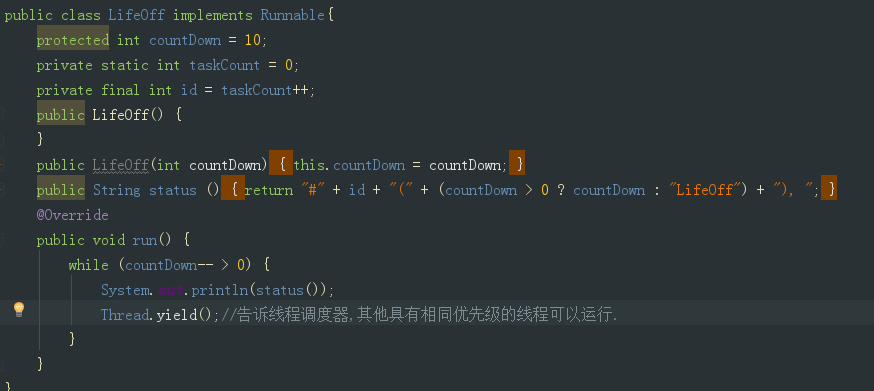
所有基本类型、String、Class、enum、Annotation、以上类型的数组。

元素不能有不确定的值，也就是说，元素要吗都有默认值，要吗在使用注解时提供元素的值。

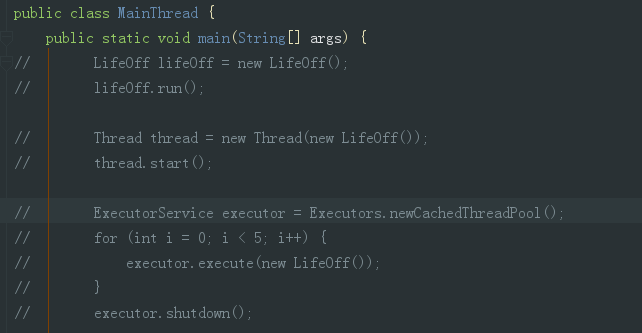
4、注解不支持继承。

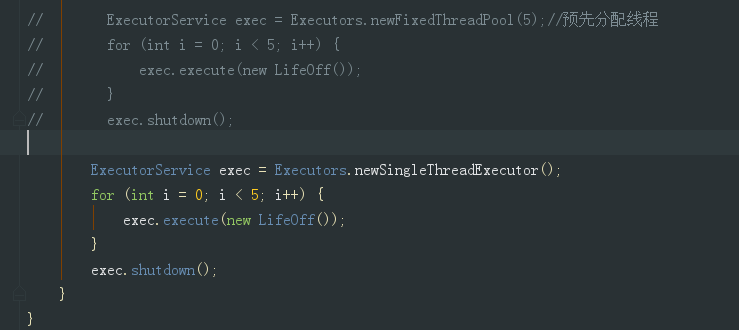
**十九.并发:**

1、并发编程是我们可以将程序划分为多个分离的、独立运行的任务。通过使用多线程机制，这些独立任务中的每一个都将由执行线程来驱动。一个线程就是在进程中的一个单一的顺序控制流，因此，单个进程可以拥有多个并发执行的任务，但是你的程序使得每个任务都好像有其自己的CPU一样。其底层机制是切分CPU时间，但通常不需要考虑它。多任务和多线程往往是使用多处理器系统的最合理方式。

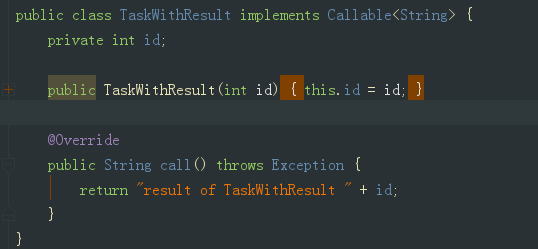


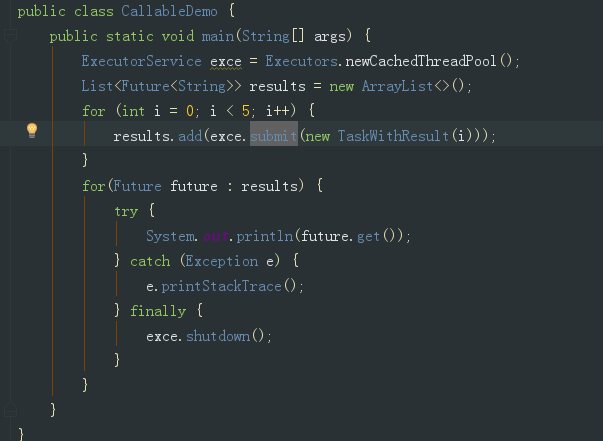
CachedThreadPool在程序执行过程中通常会创建与所需数量相同的线程,然后在它回收旧线程时停止创建新线程,因此它是合理的Executor的首选。只有当这种方式会引发问题时，才需要切换到FixedThreadPool。 SingleThreadExecutor就像是线程数为1的FixedThreadPool。如果向SingleThreadExecutor提交了多个任务，那么这些任务将排队，每个任务都会在下一个任务开始之前运行结束，所有的任务将使用相同的线程。



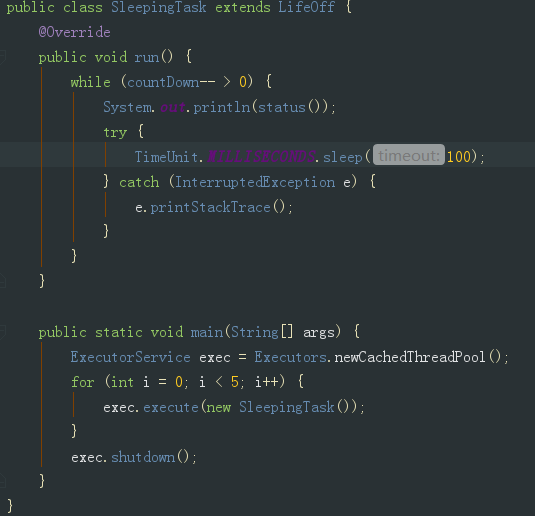


2、从任务中产生返回值：



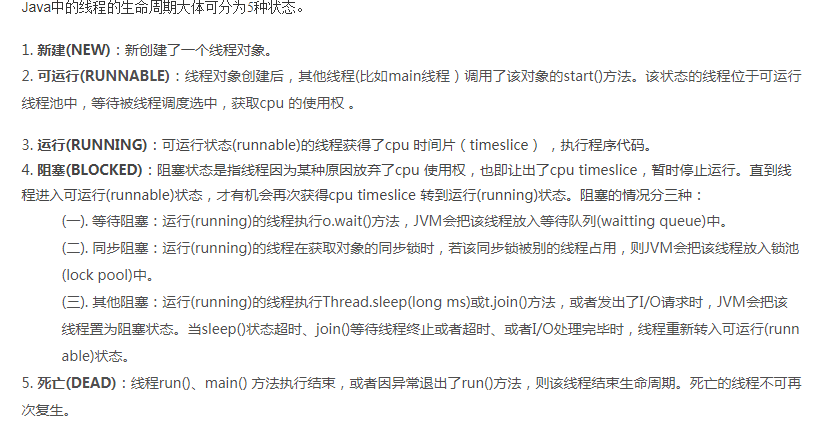


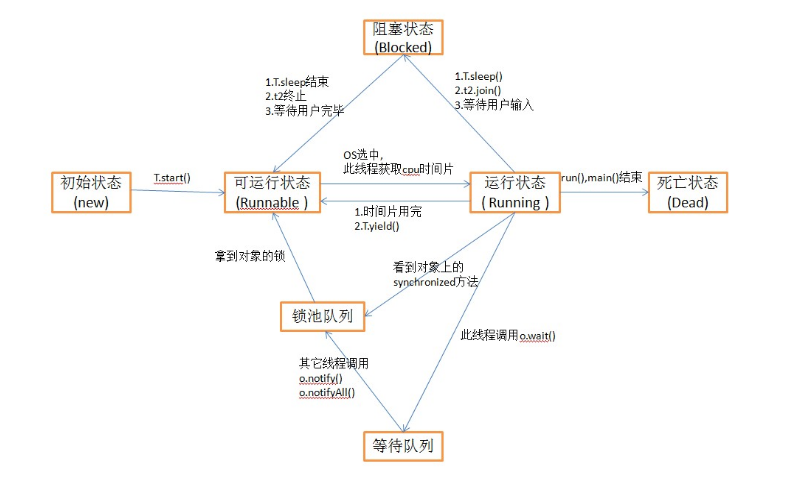
3、休眠：对sleep（）的调用会抛出异常，在run（）中捕获。因为异常不能跨线程传播回main（），所以必须在本地处理所有在任务内部产生的异常。



4、后台线程：指在程序运行的时候在后台提供一种通用服务的线程，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止。

5、线程状态：





6、共享受限资源：

1、防止线程冲突的方法就是当资源被一个任务使用时，在其上加锁。基本上所有的并发模式在解决线程冲突问题时，都是采用序列化访问共享资源的方案，这意味着在给定时刻只允许一个任务访问共享资源。Java提供了关键字synchronized，为防止资源冲突提供了内置支持。当任务要执行被synchronize关键字保护的代码片段的时候，它将检查锁是否可用，然后获取锁，执行代码，释放锁。

所有的对象都自动含有单一的锁（也称为监视器）。当在对象上调用其任意synchronized方法的时候，此对象都被加锁，这时该对象上的其他synchronized方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。注意，在使用并发时，将域设置为private是非常重要的，否则，synchronized关键字就不能防止其他任务直接访问域，这样就会产生冲突。

一个任务可以多次获得对象的锁。如果一个方法在用一个对象上调用了第二个方法，后者又调用了同一个对象上的另一个方法，就会发生这种情况。JVM负责跟踪对象被加锁的次数。如果一个对象被解锁（即锁被完全释放），其计数变为0。在任务第一次给对象加锁的时候，计数变为1，每当这个相同的任务在这个对象上获得锁时，计数都会递增。显然，只有首先获得了锁的任务才能运行继续获取多个锁。每当任务离开一个synchronized方法，计数递减，当计数为0的时候，锁被完全释放，此时别的任务就可以使用此资源。

针对每个类，也有一个锁（作为类的Class对象的一部分），所以synchronized static方法可以在类的范围内防止对static方法的并发访问。

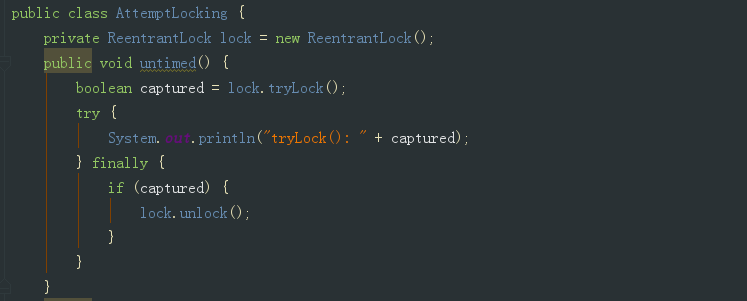
如果在类中有超过一个方法在处理临界数据，那么必须同步所有相关方法。如果只同步一个方法，那么其他方法将会随意地忽略这个对象锁，并可以在无任何惩罚的情况下被调用。这是很重要的一点：每个访问临界共享资源的方法都必须被同步，否则他们就不会正确的工作。

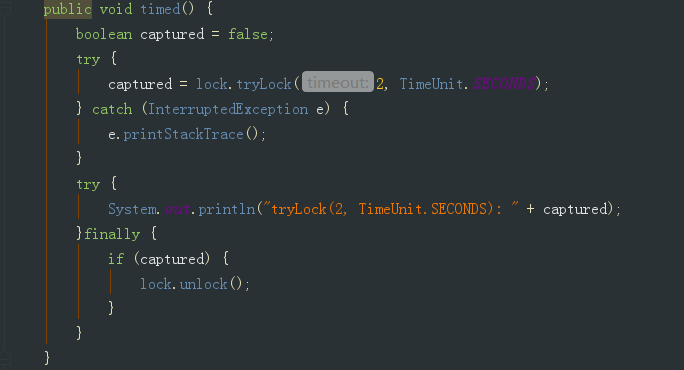
2、显示的使用Lock对象：

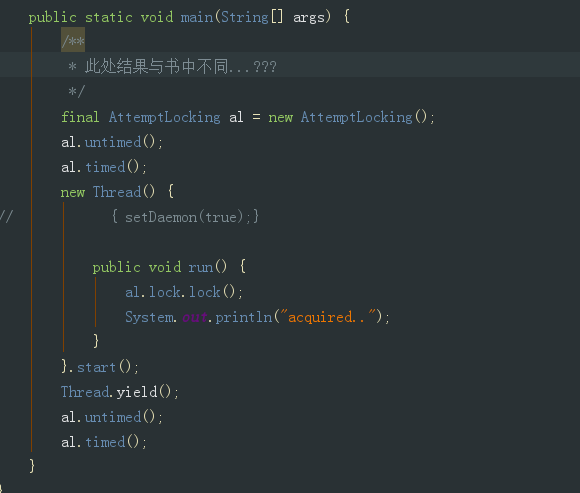
Lock对象必须被显示的创建、锁定、释放。与内建的锁形式相比，代码缺乏优雅性，但是对于解决某些类型的问题来说更加灵活。



tryLock（）：







3、原子性与易变性：

原子操作是不能被线程调度机制中断的操作；一旦操作开始，那么它一定够可以在可能发生的“上下文切换”之前（切换到其他线程执行）执行完毕。Java的递增操作不是原子性的。

定义变量时，使用volatile关键字，就会获得原子性。volatile关键字还确保了应用中的可视性。如果将一个域声明为volatile的，那么只要对这个域产生了写操作，那么所有的读操作就都可以看到这个修改。即便使用了本地缓存，情况也确实如此，volatile域会立即被写入到主存中，而读取操作就发送在主存中。

如果多个任务在同时访问某个域，那么这个域就应该是volatile的，否则，这个域就应该只能经由同步来访问。同步也会导致向主存中刷新，因此如果一个域完全由synchronized方法或语句块来防护，那就不必将其设置为是volatile的。

如果将一个域定义为volatile，那么它就会告诉编译器不要执行任何移除读取和写入操作的优化，这些操作的目的是用线程中的局部变量维护对这个域的精确同步。volatile可以禁止Java指令重排序。

4、原子类：

