**JAVA虚拟机**

**001： JDK-Java Development ToolKit（JAVA开发工具包），可以理解为JAVA API，它实际上就是前辈们写好的一堆工具**

**002： JRE-Java Runtime Enviromental (JAVA运行时环境)**

**003： JVM-Java Virtual Machinal （JAVA虚拟机）它是JRE的一部分，它是一台虚拟的计算机，它的功能是解释自己的指令集并映射到本地操作系统的指令集，例如IO功能。在咱们的业务代码中要访问计算机的IO。那么实际的操作流程是：在我们的业务代码里调用JDK里的IO相关的API，然后这个API方法会去调用虚拟机里面IO指令并触发实际操作系统CPU的IO指令从而实现IO操作。咱们的J2EE项目都是在TOMCAT启动的时候启动的JVM**

**004： JVM的内存有栈区、堆区和方法区组成（方法区）。栈区是线程私有的，堆区和方法区是所有线程共享的。JVM为每一个线程动态分配一个栈区，这个栈区在线程启动时产生，在线程结束时被回收。OutOfMemoryError一般是指堆内存报的错误，StackOverFlow一般是指栈内存报的错误。**

**栈区：栈区只存放方法中（不包括对象的成员变量）的基础数据类型和自定义对象的引用（注意是基本数据类型和对象的引用，不是基本数据类型和对象本身），基本数据类型和对象本身是存在堆区的。栈又分为三个区域：程序计数器、Java虚拟机栈、本地方法栈。JAVA代码最终在计算机中执行，是由编译器将JAVA代码编译成很多计算机指令。而在JVM中任务是以线程为单位的，即要完成某一件事情，就需要这样一个线程。一个线程包含了很多指令，在这个线程的这一组事先被设定好执行顺序的情况下，计算机怎么知道执行到哪一条指令了呢？这就是程序计数器的作用，用于记录当前线程中，计算机执行到哪一条指令了。程序计数器是线程私有的。Java虚拟机栈实际上可以理解成一个“栈（数据结构中的栈）”，每一个线程都要执行很多个方法，每一个方法在执行时都会创建一个栈帧，用来描述方法的参数，局部变量，返回值等等，JVM每创建一个线程就创建一个Java虚拟机栈用来存放这些栈帧，供这些栈帧入栈和出栈。当一个方法开始执行时，那么这个方法对应的栈帧就入栈，当方法执行完毕，这个方法对应的栈帧就出栈。本地方法栈，实际上和Java虚拟机栈是一样的，只不过Java虚拟机栈存放的是Java方法的栈帧，而本地方法栈存放的是本地方法的栈帧。本地方法就是使用非Java语言实现的方法。定义一个非JAVA方法的写法如下：native public void fun(int x);**

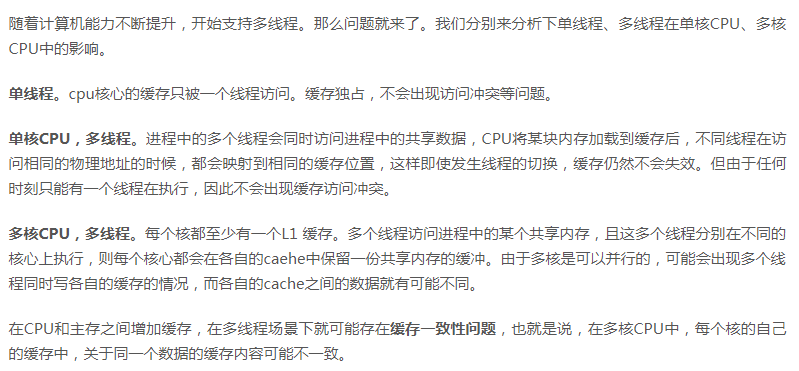
**堆区：存储的全是基本数据类型和对象实例（本身），每一个实例都包含一个与之对应的class信息（class信息存放在方法区）。Java的垃圾回收GC多半都是指的回收这一部分的内存。**

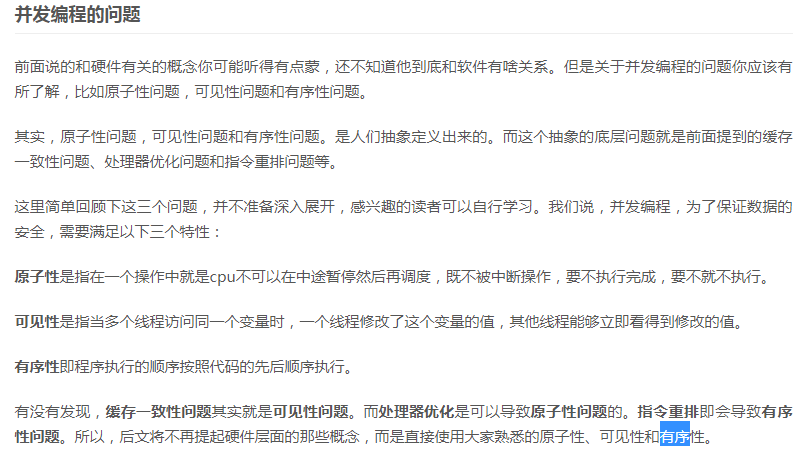
**方法区：它的作用是存储Java类的结构信息，当我们创建对象实例后，对象的类型信息存储在方法区，实例数据存放在堆区；实例数据指的是对象本身和它的值，结构信息是指类的修饰符、类中定义的常量、静态变量、方法和方法字段等。在JAVA7以前方法区和堆内存是共用一块内存区域的，它只是一个概念上的区域。而JAVA8以后它实际上是JVM所在物理计算机的一块物理内存，也不再叫方法区，而是叫做元空间，理论上来说这个物理内存有多大，那么元空间就有多大，但是可以通过参数来设置它的大小，如果实际使用到的元空间大于了设定的大小就会触发GC来回收元空间里那些已经没有被任何引用的内存。运行时常量池：是.class文件中每一个类或者接口的常量池表的运行时表示形式，它的作用是：存放编译器生成的各种字面量和符号引用。当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类创建或运行时解析、翻译到具体的内存地址中去。字面量：通俗地说就是用final修饰的常量，符号引用是编译原理里面的概念，包括了下面三类常量：类和接口的全限定名、字段的名称和描述符、方法的名称和描述符（描述符就是public private等等）。由于堆区是所有线程共享的，那么多个线程可能会同时访问堆区存放的同一个数据，这就是常说的Java线程间的共享信息通信（另一种是发送消息传递通信）。**

**005： JAVA内存模型（JMM Java Memory Model）,它是一个抽象的概念，JMM就是一组规范线程之间共享信息通信的规定，它的作用就是有序的组织多个线程并发操作，从而避免多线程出现的很多问题。它定义了一组语法集，这些语法集映射的关键字就是：volatile和synchronized等。在JMM里将内存分为两种：一种是主内存，一种是工作内存，主内存就是所有线程公用的一块内存区域，工作内存是指线程私自占用的那一块内存。这里的主内存和工作内存与之前说的堆内存和栈内存不能直接进行类比，如果非要将两者联系起来那么主内存是指的堆内存中存放对象实例数据部分，而工作内存是指栈内存。**

**006： Java对象模型，Java是一种面向对象的语言，而Java对象在JVM的存储也是有一定结构的，这个结构就成为Java对象模型。HotSpot的JVM中的Java对象模型是：OOP-Klass Model()，OOP-Ordinary Object Pointer（普通对象指针），而Klass用来描述实例对象的具体类型。每一个Java类，在被JVM加载时，JVM会给这个类创建一个instanceKlass，保存在方法区，用来在JVM层中表示该JAVA类，当我们new创建一个对象的时候，JVM会创建一个instanceOopDesc，这个对象包含了对象头信息和实例数据（值）。**

**007： 缓存，早期的计算机的CPU执行速度很慢，和计算机的物理内存是很匹配的，后来CPU得以发展执行速度很快，但是物理内存的读写速度并没有变化，这就导致每一次执行指令的时候从内存读写数据比执行指令的时间长得多，这会让读写物理内存消耗了大量时间，从而导致CPU效率低下。为了解读CPU从物理内存读写数据慢的问题，高速缓存就出现了。它的作用过程是：在程序运行过程中，先将运算需要的数据从物理内存复制一份到缓存中，CPU在开始执行运算之前从缓存读数据，执行完毕之后将运算结果写入缓存，再把缓存中的数据整个刷新到物理内存中。后面随着CPU的速度越来越快，一级缓存已经不能解决实际问题，后来就出现了多级缓存。L3里的数据是L2数据的一部分，L2数据是L1数据的一部分，执行顺序是：CPU先从L1里找数据，找不到再从L2里找，L2找不到再从L3里找。**





**JAVA堆内存 = 新生代 + 老年代**

**新生代 = eden区 + 2个survivor区（一个from区，一个to区）组成**

**-Xms JVM启动时的内存，-Xmx JVM启动后能获得的最大堆内存**

**-Xmn 代表新生代的大小所在-Xmx确定的情况下，如果新生代越大，就表示老年代越小**

**在新生代中eden区和survivor区的大小比例是可以指定的，默认是8:1即假设JVM新生代的内存总容量是10M，那么eden区的大小就是8M，而from区和to区的大小是1M，而整个新生代的实际可用大小只有9M，因为只有eden区和from区是用的，而to区的内存对于用户线程而言是不可用的，至于to区为什么不可用，后面在进行说明。**

**008： GC ROOT GC的收集的对象是：那些不是GC ROOT 又没有被GC ROOT引用的对象。一个对象可能有多个GC ROOT，GC ROOT有以下几种：**

**Class 由系统类加载器（system class loader）加载的对象，这些类是不能够被回收的，他们可以以静态字段的方式保存持有其他对象。用户通过自定义类加载器加载的类，除非相应的java.lang.Class实例以其他的某种方式成为GC ROOT，否则它们并不是GC ROOT。**

**Thread 活着的线程**

**Stack Local Java方法的local变量或参数**

**JNI Local JNI方法的local变量或参数**

**JNI Global 全局的JNI引用**

**Monitor Used 用于同步的监控对象**

**Held by JVM JVMGC自身保持的一些对象。**

**009： 从静态的层面来分析GC过程。在JAVA中方法是存在栈上的，JVM为每一个方法创建一个栈帧，这个栈帧对应一块内存区域，这个方法的参数、局部变量就存放在这块内存区域中。所以在一个方法中定义一个int a = 1,或者String str = “abc”;那么GC是不会回收a和str这两个对象所在的内存区域的，因为这个内存区域在栈上，而GC针对的是堆内存。但是在这个方法里如果有一个String str = new String(“abc”)；那么GC回收时就会来回收str对象所在的内存区域。因为new出来的对象（这里称为B）是存放在堆区的，但是这个对象的引用（这里称为A）却放在这个方法所在的栈帧这块区域中（栈里），所以就算一个方法执行完了，但是在方法里new出来的对象仍然存在堆区，只是存在堆区里面这个对象已经没有了引用（因为它的引用A所在方法已经执行完，它所在的栈帧所占的内存被系统回收），而这个引用A就称为GC ROOT，在GC的时候首先枚举所有的GC ROOT（在所有的GC ROOT里面没有A ）在枚举完GCROOT 之后就开始扫描青年代或者老年代，扫描到了对象B，发现对象B没有被任何GC ROOT引用（因为它的引用A所在的栈帧内存已经被JVM回收了，GC线程是怎么发现B没有任何GC ROOT引用它的呢，这就是OopMap的作用了，据我猜想应该GC去检查某个OopMap里面有没有描述引用A和对象B之间的映射关系，如果有则判定对象B是有引用的，否则判定B是无引用的，这个过程成为可达性分析算法，所谓可达性就是指对象本身到所有的GC ROOT中的任何一个是否可达，即是否有引用与被引用关系）。于是将它标记为要“处死”的对象。在下一次GC扫描新生代和老年代的时候，发现B已经被标记为要“处死”的对象，于是GC就将其处死，收回它所占用的内存。OopMap是在程序运行过程中产生的，它的作用是描述方法里GC ROOT引用 与堆上实际对象之间的映射关系。**

**010： 从动态层面来分析GC过程。所有的GC都是需要Stop The World（暂停所有用户线程）的。那么系统到底是什么时候进行GC的呢，GC的触发有几种：在某个方法中new对象的时候发现eden区的内存不足以存放需要创建的对象时，触发Minor GC,在Minor GC过程中，有些对象需要从新生代移动到老年代去（或者有些JVM实现的机制是分配大对象的时候直接将对象存储于老年区），发现老年代的空间不足会触发Full GC。在我们的JAVA代码里执行System.gc()实际上也是执行Full GC。这只是一般情况，但是还要考虑空间分配的担保问题，即在发生Minor GC之前，JVM会检查老年代的最大连续可用空间是否大于新生代所有的对象所占空间的总和，如果大于则进行Minor GC,如果小于则看HandlePromotionFailure设置是否允许担保失败（不允许的话就直接Full GC）,如果允许，那么GC线程会尝试计算当前老年代内最大连续可用空间是否大于历次进入老年代的对象的平均大小，如果大于则进行Minor GC,如果尝试计算失败或者计算结果小于则Full GC。在JVM发起GC请求之后并不是立刻就发起GC线程，而是要等待所有的用户线程都到达最近的安全点之后才发起GC线程（安全点就是那些就算当前的实际业务线程暂停也不会对线程结果产生影响的点，一般就是方法刚进入时，或者循环结束后，或者方法准备返回时）。等到所有的线程都到达最近的安全点之后,JVM就发起GC线程。**

**011： GC清理内存的算法一般有两种：标记-清除算法和复制算法**

**012： 标记清除算法 它的做法是标记那些要被清理的内存区域（即新生代或老年代被宣判死刑的那些对象所占用的那些内存空间），当标记完成之后，回收这些内存空间。这样做会导致一个问题那就是内存回收之后，剩下的内存区域是碎片化的，即不是连续性的，例如内存位置有ABCDE五个位置都有值了，而BD被清除之后，剩下的内存就是A C D。原来BD所占的内存空间被C隔开了。这样很可能会导致系统内没有一块连续的较大的内存区域来存放那些较大的对象。但是标记-清除算法有一个好处，那就是标记完之后就立马清除，所以它的效率是很高的，节约时间成本。**

**013： 复制算法 就是在内存区域中专门开辟一块内存区域用来存放那些被标记为存活的对象。例如现代的JVM的新生代采用的就是复制算法，它的具体做法是把新生代分成三块：eden区、两个survivor区（幸存则区、from区和to 区），GC轮训完eden区和from区之后把这两个区内存活的对象全部移动到to区。这样eden区和from区在GC之后就是一块连续的区域了。这种算法虽然能够创造出大片的连续内存空间，但是却需要额外的内存复制，消耗CPU和时间资源，而且它还需要一块额外的内存空间来进行转存。**

**014：分带收集算法 分带收集算法的宗旨就是将内存区域分成新生代和老年代，对新生代采用复制算法来进行GC，在老年代采取标记-清除的算法来进行GC。一般新生代的内存区域比较小，如果采用标记-清除算法的话，就会导致经常都发生大内存的对象无法直接存到新生代从而触发Full GC。老年代采用标记-清除算法主要是因为老年代的内存一般都开辟的比较大，一般不会出现没有较大的连续空闲内存空间。**

**015： 平台无关性 JAVA语言本身并不是平台无关性的，真正实现平台无关性的是虚拟机和字节码文件(ByteCode,即.class文件)。JAVA语言在经过javac编译器编译过后形成一系列的字节码文件，虚拟机就是将字节码文件里的指令翻译成本地计算机的CPU指令来让程序工作的。也就是说虚拟机执行的指令从本质上说它和JAVA语言本身是没有关系的，它只认字节码文件，任何语言只要经过它的编译器编译过后能形成有效的（即虚拟机能识别的）字节码文件，那么这种语言编写的程序就能够被虚拟机执行。例如JRuby、Groovy等等这些语言编写的程序经过它们编译编译后能都能JAVA虚拟机执行。**

**016： 任何一个Class文件都对应着唯一一个类或者接口的定义信息，但是反过来说，类或接口并不一定都得定义在Class文件或者其他文件里，因为有些类或者接口都是由类加载器直接生成。**

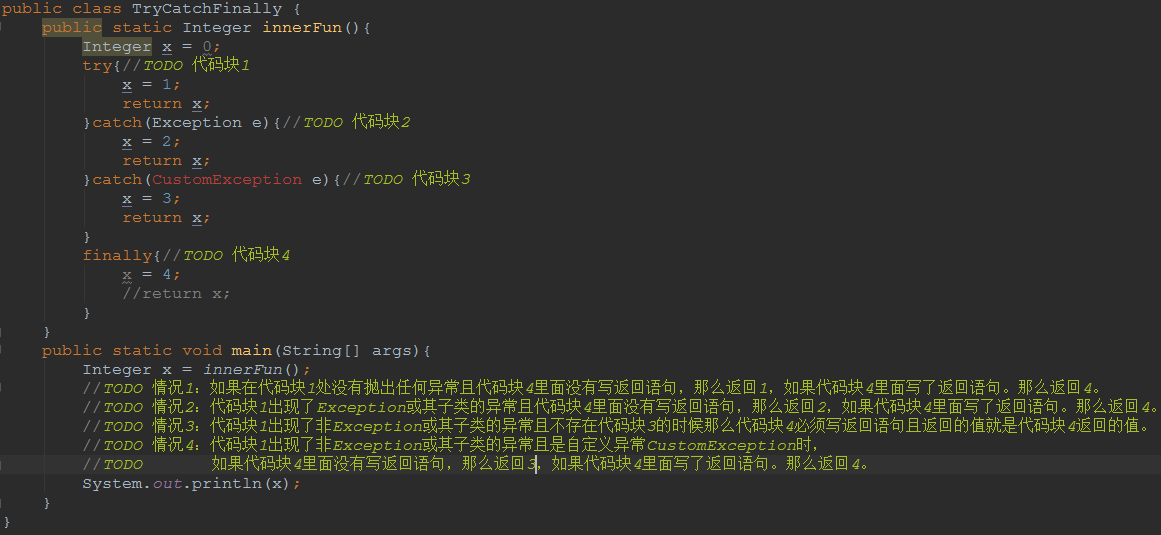
**017： class文件 class文件是一组以8位字节为单位的二进制流，各个数据项按照严格的顺序紧凑第排列在class文件中，中间没有添加任何分隔符，一个数据项可占用多个8位字节。**

**018： class文件的数据类型只有两种：无符号数和表，无符号数属于基本的数据类型，以u1、u2、u4、u8分别代表1个字节、2个字节、4个字节、8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成的字符串值。表是有多个无符号数或者其他表作为数据项的复合数据类型。所有的表都以“\_info”结尾。**

**019： 16进制数0xABC各部分的意思是：0x是表示当前这个数字是16进制数，A在16进制里代表10、B、C、D、E、F分别代表11、12、13、14、15。最末尾就是C\*16的0次方。倒数第二位就表示B\*16的1次方，倒数第三位表示A\*16的2次方。所以0xABC所代表的意思就是：10\*16\*16 + 11\*16 +12\*1 = 2748。**

**020： 操作系统的32位和64位，里面的位表示CPU一次性执行的数据宽度，32位即表示CPU一次性能执行32bit的数据，而64位操作系统一次性能执行64bit的数据。32位于64位的区别还在于寄存器的大小不一样，而寄存器有两个作用：存储数据并计算，32位CPU一次性能处理的数据大小为32位，即两个48位的数据相加，则先计算低32位，再计算高16位。而64位操作系统就是直接相加；间接寻址32位CPU最大支持4GB的内存地址，而64位则支持**16777216TB**的内存寻址。所以如果在32位CPU上安装操作4GB的内存那是没用的。**

**021： 代码中trycatchfinally的返回值 如果try代码块中写了返回值，并且没有任何异常抛出，那么分为两种情况，分别是finally语句块里有些返回语句和没有写返回语句，如果finally语句块里写了返回语句，那么返回的值就是finally语句块里返回的值，如果finally语句没有写返回语句，那么返回的值就是try语句块里面的返回的值。如果在try语句块抛出了异常，那么也分为两种情况：一种是抛出的异常是Exception类或其子类的异常，这时候又分为两种情况，如果如果finally语句块里写了返回语句，那么返回的值就是finally语句块里返回的值，如果finally语句没有写返回语句，那么返回的值就是catch语句块里面的返回的值。如果抛出的异常是非Exception或其子类的异常时，那么就不会执行catch语句块的异常而是直接执行finally语句块的代码。只要代码中写了finally语句块，那么在方法执行过程中finally语句块的代码始终都是最后执行的。另外还有一种情况就是用户自定义的异常，如果catch的时候写了抓取自定义异常的代码块,同时有写了catch\_Exception语句块和finally语句块。如果自定义异常处理语句块里有返回值，且finally语句块没有返回值时，那么返回的值就是自定义异常处理语句块的返回值。例如**



**022： 实例变量和类变量 实例变量和类变量实际上都是类的成员变量，实例变量就是非静态变量，类变量就是静态变量。**

**023： 虚拟机类加载 虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化、最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。在Java语言里面，类型的加载、连接和初始化过程都是在程序运行期间完成的。一个类从被加载到虚拟机内存开始，到被从虚拟机内存移除位置，一共要经历7个阶段：加载、验证、准备、解析、初始化、使用和卸载。其中验证、准备、解析3个部分统为连接。**

**024： 类加载的过程**

**①通过一个类的全限定名例如com/shyfay/admin/Constant来获取定义此类的二进制字节流**

**②将这个字节流代表的静态存储结构转化为方法区的运行时数据结构**

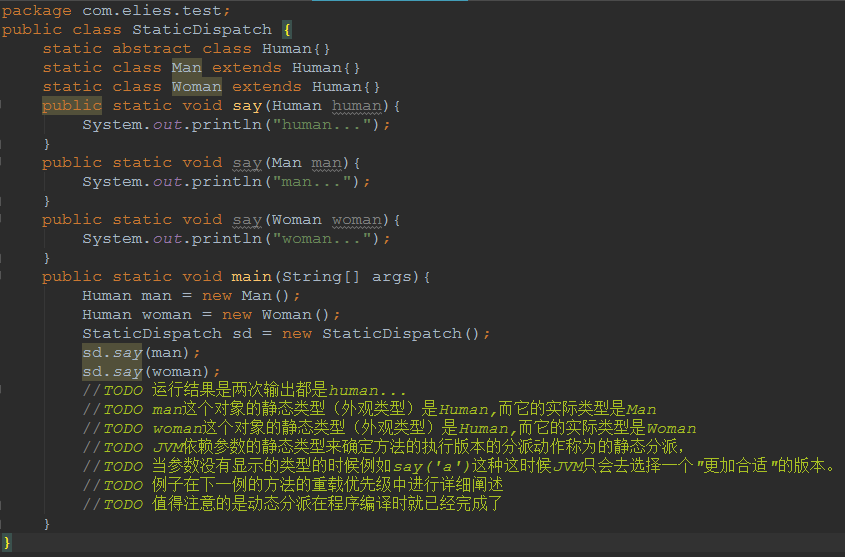
**③在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口**

**④验证阶段，验证文件格式（魔数是或否是0xCOFEBABY）、元数据验证等等**

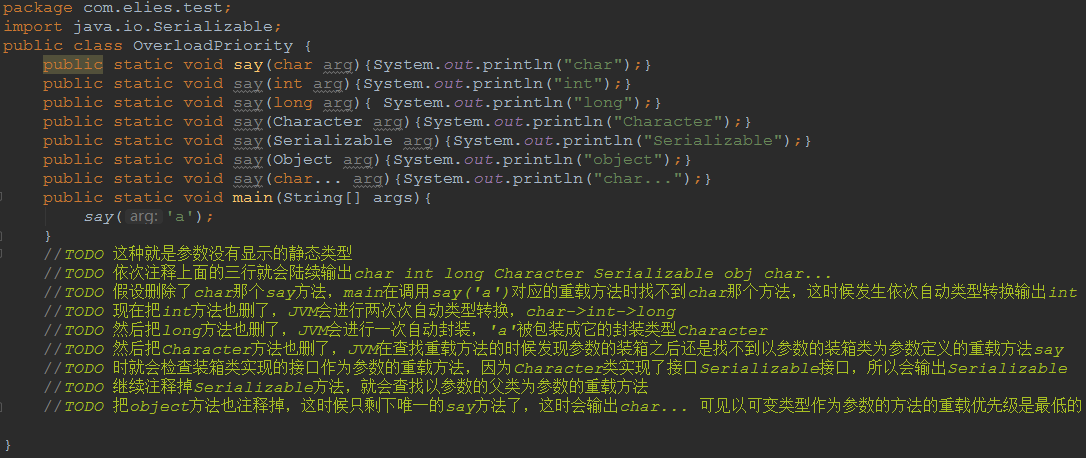
**⑤解析，解析就是虚拟机将常量池的符号引用替换为直接引用的过程。**

**⑥初始化，根据程序员指定的值去初始化类的变量和其他资源，实际上初始化就是调用类的clinit方法。**

**025： 类加载器（ClassLoader）从虚拟机角度来说只有两种加载器，启动类加载器和非启动类加载器，从程序员的角度来说分为三种类加载器：启动类加载器、扩展类加载器、应用程序类加载器。类加载的过程是自定义加载器-应用程序类加载器-启动类加载器。作用过程是先交给顶层的加载器去加载类，依次向下，如果处于上面的加载器执行类加载过程之后如果加载成功会把类的是否被加载标志设置成是，如果加载不成功就返回一个失败状态，下层的类加载器在读取到这个加载失败状态之后再来加载这个类，发现这个类还没有被加载过，于是自己开始加载。这称为双亲委派模型。**

**026： 静态分派 虚拟机依赖静态类型来确定方法的执行版本的分派动作称为静态分派。**

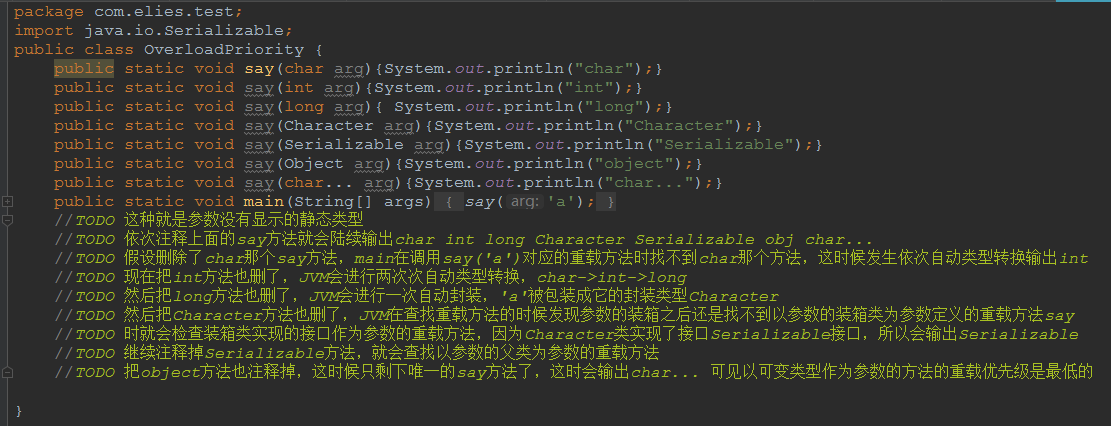
**027： 方法重载优先级 这是在参数没有显式的静态类型的前提下，虚拟机确定方法的执行版本的一种查找先后顺序的机制。**



**028： 动态分派 JAVA继承的精髓**



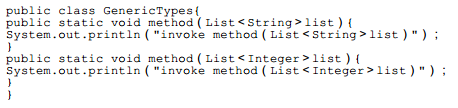
**028： JAVA方法重载优先级**



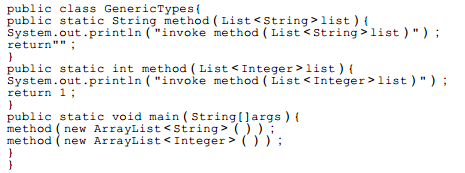
**029： 动态语言和静态语言**

**动态语言的关键特征是它的类型检查的主体过程是在运行期而不是编译期，静态语言的关键特征是它的类型检查的主题过程是在编译期而不是运行期。JAVA是静态语言、JavaScript、Groovy等等就属于动态语言。静态语言的好处是提供严谨的类型检查，这有利于程序的稳定性，更适合编写大规模的程序。动态语言的好处是可以为程序员提供更好的灵活性。**

**030： JAVA的泛型 C#里面的泛型无论是在程序源码中、编译后IL中、还是运行期的CLR中都是切实存在的，List<int>与List<String>就是两个不同的类型，它们在系统运行期生成，有自己的需方发表和类型数据，这种实现称为类型膨胀，基于类型膨胀实现的泛型称之为真实泛型。Java语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经替换成原生类型了，并且在相应的地方插入了强制类型转换代码，因此对于运行期的Java程序来说，ArrayList<int>与ArrayList<String>就是同一个类，这种实现方法称为类型擦除（即在编译期对泛型进行类型还原），基于类型擦除实现的泛型称为伪泛型。下面这段代码是无法通过编译的，因为在编译期List<String>与List<Integer>是同一个类，即第一个method与第二个method这两个方法完全一样，无法实现重载。**



**但是同样的下面这段代码就可以通过编译且成功运行（前提是JDK版本是1.6，在其他版本的JDK中是不能运行成功的）。**



**这是不是对Java语言中返回值不参与方法重载选择的基本认知相悖？**

**但是上面的代码当然不是根据返回值来确定的，之所以能编译通过且成功运行是因为两个method方法加入了不同的返回值后才能共存在同一个class文件中实现重载，在JAVA中方法重载要求方法拥有不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载，但是在Class文件格式中，只要描述符不是完全一致的两个方法就可以共存。也就是说，两个方法如果具有相同的名称和特征签名，但是返回值不同，那它们也可以合法地共存于同一个Class文件中**

**031： 指令重排序，即在一组指令流中，CUP并不一定是按照指令流中指令的先后顺序来执行指令的。假设有指令A=1;A++;B=A+1这三条指令，那么是不会发生指令重排序的，但是假设有指令A=1;A++;B=2;这一组指令那么最后一个指令可能在第一条和第二条指令之前也可能在他们之间执行。**

**032： 现行发生原则 所谓现行发生原则是指线程A的某个操作A1在线程B的B1操作发生之前发生，那么A1操作所产生的影响能够被B1操作所感知。这种影响包括内存中共享变量的值、发送了消息调用了方法等等。先行发生的实例如下**

**//A线程中的操作**

**a=1;**

**//B线程中的操作**

**b=a;**

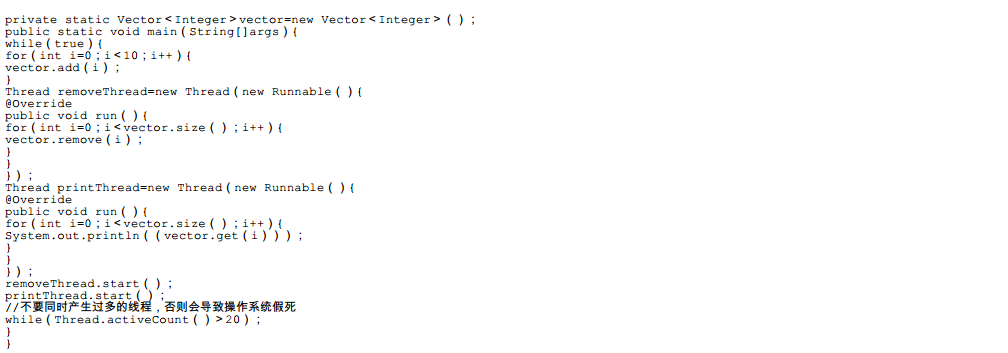
**//C线程中的操作**

**a=2;**

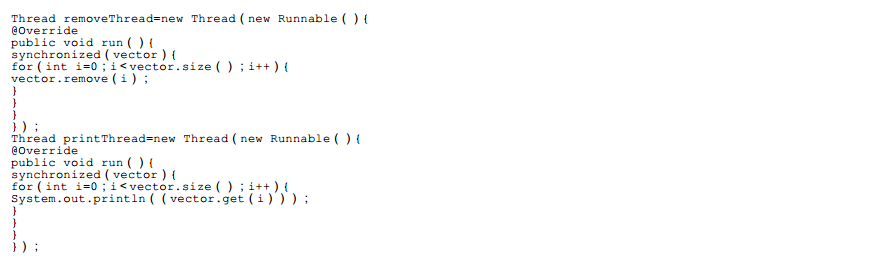
**先行发生原则是：如果a=1这个操作在CPU中先与b=a这个操作执行，那么在B线程中执行b=a后b的值一定是1，这时如果线程存在线程C而且它和线程B没有先行发生原则，那么在B线程中执行b=a后b的值就不一定是1了而是无法确定的。先行发生原则与代码位置和某一段代码的执行时间上的先后顺序无关，它所针对的只是在指令在CPU中执行的先后顺序。意思就是假设线程A先启动，线程B后启动，在线程B执行完b=a操作之后b的值不一定等于1，因为即使线程A比线程B先启动，但是由于线程复杂度和指令重排等等，会导致线程A中的操作a=1所对应的指令并不一定比线程B中的操作b=a对应的指令先在CPU中执行。**

**033： JAVA线程状态 JAVA线程只有5中状态：新建、运行、等待、阻塞、结束。阻塞和等待的区别是：“阻塞状态”是在等待其持有的排它锁被另外的线程释放，处于粗赛状态的线程可能在另外一个线程释放这个它持有的排它锁的时候获得这个排它锁之后处于运行状态，线程阻塞是程序自身决定何时阻塞和何时被激活的。而等待则是程序员让写代码让线程停住等待，程序员可以通过特定的代码唤醒线程。**

**034： 线程安全 当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要额外的同步，获取在调用方进行任何其他的协调操作，调用这个对象的行为都能得到正确的结果，那么这个对象就是线程安全的。我们将JAVA语言中的操作共享数据分为以下五类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。不可变：使用final关键字修饰的对象就是不可变对象，不可变对象是线程安全的。如果共享数据是一个不可变对象，那么就要求对象的行为不会对它自身的状态缠身影响，例如String类就是一个不可变的对象，它的substring、replace、concat等都是不会对本身的字符串产生任何影响的，它只会返回一个新的String对象。绝对线程安全：java.util.Vector是一个线程安全的容器，但是它不是绝对线程安全的，只能说它只是针对它自身定义的方法是线程安全的，因为它的add、get、和size这类方法全部都定义成synchronized的。但是我们依然可以写代码来让Vector容器对象变得线程不安全例如下面的代码：**



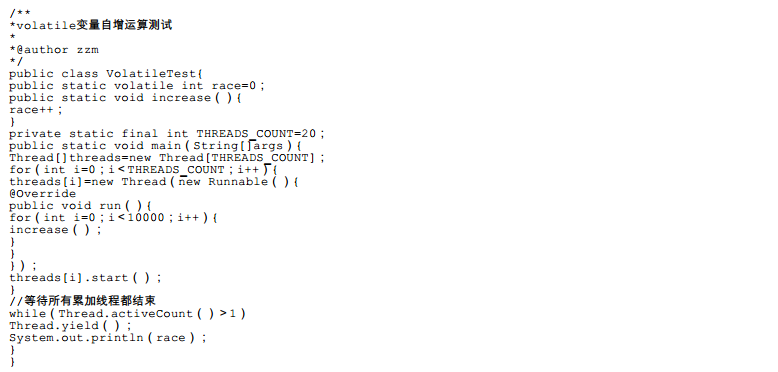
**以上这段代码会抛出java.lang.ArrayIndexOutOfBoundsException异常。因为一个线程在往容器里写的时候，另外一个线程在从里取，而Vector容器的add、remove等方法都是被synchronized修饰的方法，这只能保证在执行add或者remove那一瞬间整个容器的节点不被修改，但是在这些瞬间之间存在一个时间空隙，在这些空隙间别的线程就可以对容器的节点进行删除和增加操作了。所以说上面那一段代码不是绝对线程安全的。改成下面的代码就是绝对线程安全的了**



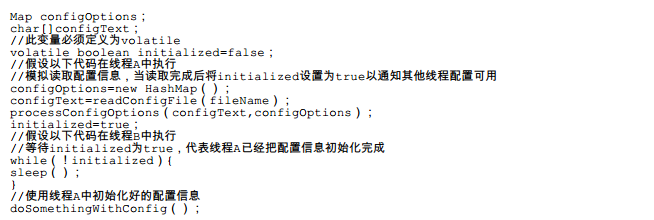
**相对线程安全：相对线程安全就是指的我们传统意义上说的线程安全，就是指针对某一个特定的原子操作是线程安全的，例如Vector的add、remove等被synchronized修饰的方法就是相对线程安全的。线程兼容：线程兼容是指对象本身不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用，例如Vector、ArrayList、HashMap等等。线程对立：线程对立是指无论在调用段是否采取了同步措施，都无法在多线程环境中并发使用的代码，一个线程对立的例子是Thread类的suspend和resume方法如果两个线程同时持有一个线程对象，一个线程尝试去中断线程，另一个尝试去恢复线程，如果并发进行的话，无论调用时是否进行了同步处理，目标线程都存在死锁的风险。如果suspend终端的线程就是即将要执行resume的那个线程，那肯定要产生死锁。常见的线程对立还有System.setIn、System.setOut和System.runFinalizersOnExit等方法。**

**035： volatile关键字 针对上面定义的JAVA内存模型，普通的共享对象在多线程并发的情况下，线程A要获取这个共享对象的值，要先将这个共享对象从主内存拷贝一份到自己的工作内存中才能获取到这个值，当线程A不暂时不需要这个共享对象的时候它的工作内存中可能没有这个共享对象的副本或者它的工作内存中这个副本的值与主内存中这个共享对象的值不等（即不是最新的）。而volatile有两个特性：第一是保证此变量对所有的线程的可见性，即volatile类型的共享变量在被一个线程修改以后，其他线程可以立即看见这个改变的，而普通变量不能做到这一点，普通变量的值在线程间传递均需要通过主内存来**

**完成，例如，线程A修改一个普通变量的值，然后向主内存进行回写，另外一条线程B在线程A回写完成了之后再从主内存进行读取操作，新变量值才会对线程B可见。但是volatile修饰的变量仍然不是线程安全的：例如下面的例子：**



**最终的输出结果始终是小于200000的。Volatile变量的另一个特性是会静止指令重拍。**

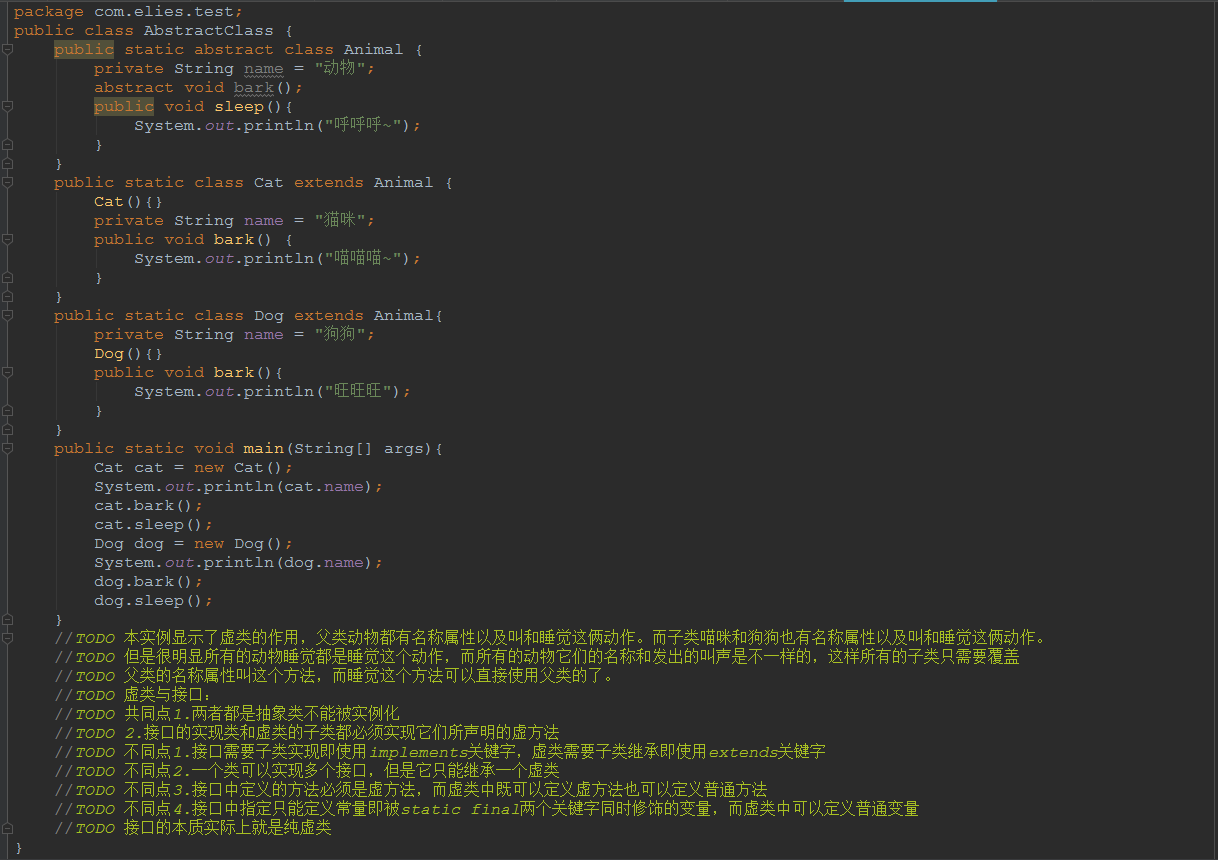


**上面的代码，如果initialized变量没有被volatile关键字修饰，那么由于指令重排有可能在A线程中的最后一句代码initialized=true比加载配置文件的指令先执行，然后B线程读取initialized的时候发现为真，而去读取配置文件，导致读取不到配置文件的内容。**

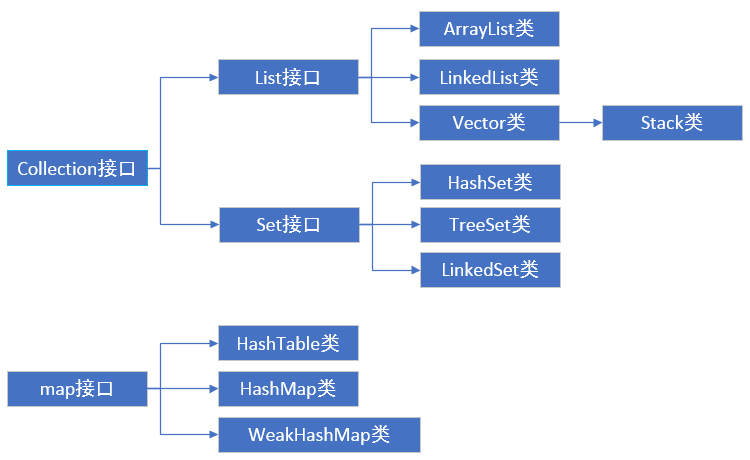
**036： Atomic 原子自增运算，针对上面35里的volatile变量线程不安全那个例子做如下改变就可以让程序正常输出200000。**



**037： 虚类与接口**



**038： java容器**



**039： JAVA容器之基本**

**JAVA的两大基础容器Collection和Map，后面的容器如List接口、ArrayList、Vector、Stack等都是Collection类的子类或其子类的子类。而HashMap和TreeMap是Map的子类。**

**040： JAVA容器之List接口 注意List是一个Interface，它不是一个Class。不能直接拿List来使用。ArrayList和LinkedList才是Class，它们都是List接口的实现类。**

**041： JAVA容器之Array、ArrayList**

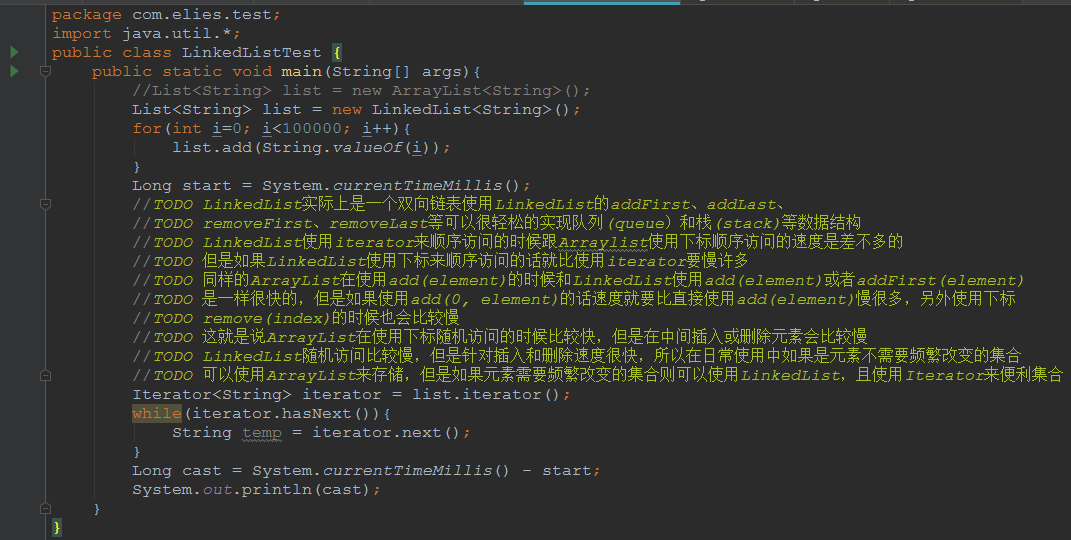
**Array可以存储基本数据类型和对象类型，ArrayList只能存储对象类型**

**Array数组长度是固定不变的不能更改，ArrayList的长度是可以改变的**

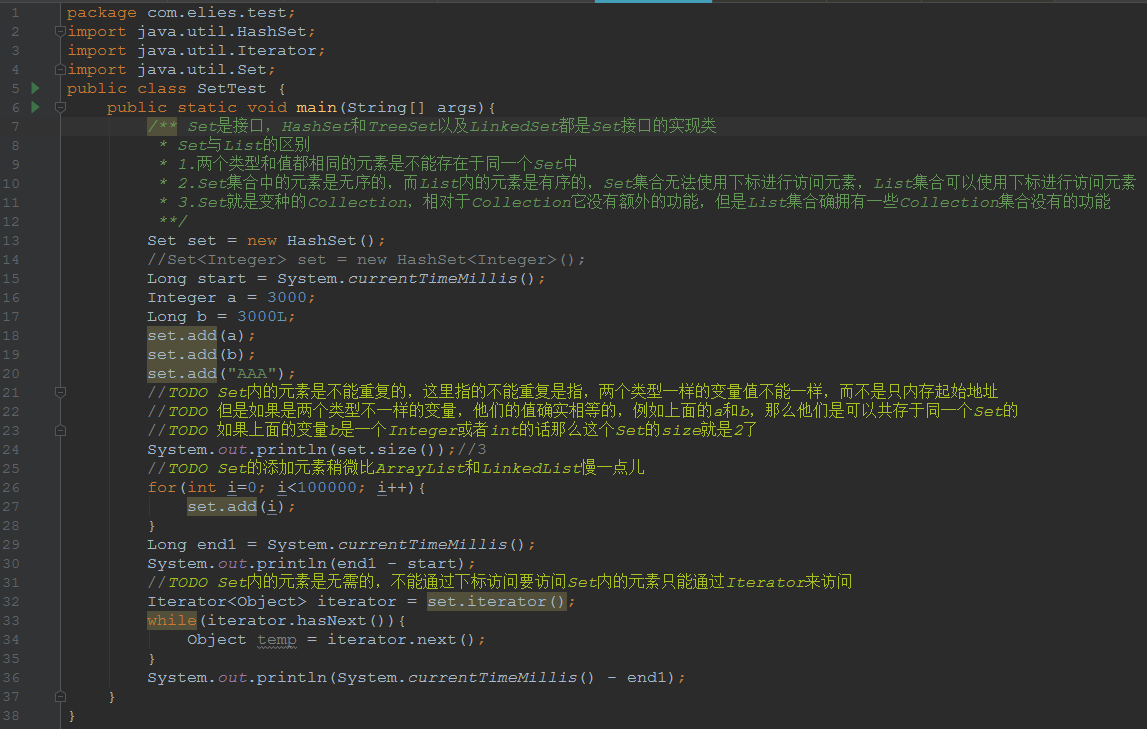
**一个Array里所有的元素类型是一样的，ArrayList里面的元素类型可以不一样**

**ArrayList有丰富的方法如addAll()、removeAll()，iterator（）等。**

**042： JAVA容器之ArrayList与LinkedList**



**043： JAVA容器之Set**



**044： Java容器之Map**

