1. JVM知识点（类加载、内存、GC）

## 类的加载机制

主要关注点：

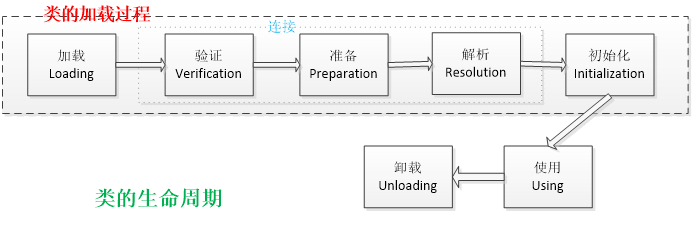
* 什么是类的加载
* 类的生命周期
* 类加载器
* 双亲委派模型

什么是类的加载？

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

类的生命周期？

类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程,如下图；



* 加载：查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
* 连接：连接又包含三块内容：验证、准备、初始化。1）验证，文件格式、元数据、字节码、符号引用验证；2）准备，为类的静态变量分配内存，并将其初始化为默认值；3）解析，把类中的符号引用转换为直接引用
* 初始化：为类的静态变量赋予正确的初始值
* 使用：new出对象程序中使用
* 卸载：执行垃圾回收

几个小问题？1、JVM初始化步骤 ？

1. 类初始化时机 ？

3、哪几种情况下，Java虚拟机将结束生命周期？在如下几种情况下，Java虚拟机将结束生命周期

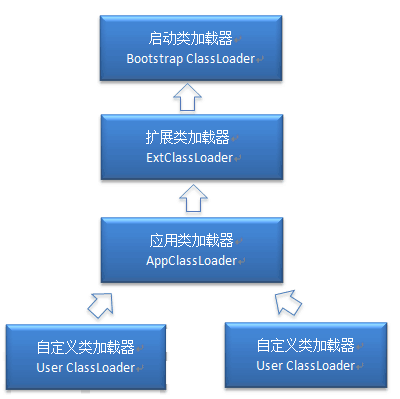
– 执行了System.exit()方法

– 程序正常执行结束

– 程序在执行过程中遇到了异常或错误而异常终止

– 由于操作系统出现错误而导致Java虚拟机进程终止

类加载器？



* 启动类加载器：Bootstrap ClassLoader，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库。
* 扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.\*开头的类），开发者可以直接使用扩展类加载器。
* 应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器

类加载机制？

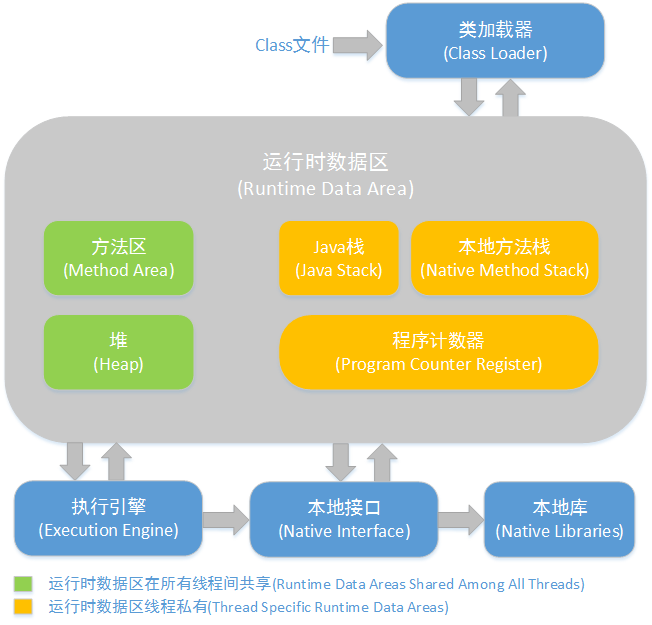
* 全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入
* 父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类
* 缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

## jvm内存结构

主要关注点：

* jvm内存结构都是什么
* 对象分配规则

jvm内存结构？



*方法区和堆是所有线程共享的内存区域；而java栈、本地方法栈和程序员计数器是运行是线程私有的内存区域。*

* Java堆（Heap）,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
* 方法区（Method Area）,方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
* 程序计数器（Program Counter Register）,程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。
* JVM栈（JVM Stacks）,与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
* 本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

对象分配规则？

* 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
* 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
* 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
* 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。
* 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

## GC算法（垃圾回收）

主要关注点：

* 对象存活判断
* GC算法
* 垃圾回收器

对象存活判断？

判断对象是否存活一般有两种方式：

* 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
* 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

GC算法？

GC最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

* 标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。
* 复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。
* 标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存
* 分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

垃圾回收器？

* Serial收集器，串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。
* ParNew收集器，ParNew收集器其实就是Serial收集器的多线程版本。
* Parallel收集器，Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。
* Parallel Old 收集器，Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法
* CMS收集器，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。
* G1收集器，G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

## GC分析（命令调优）

主要关注点：

* GC日志分析
* 调优命令
* 调优工具

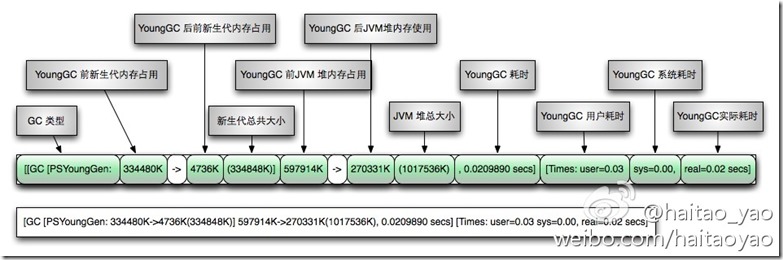
GC日志分析

摘录GC日志一部分（前部分为年轻代gc回收；后部分为full gc回收）：

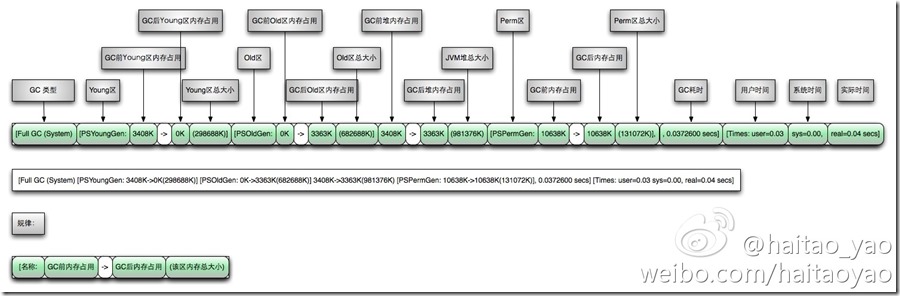
|  |
| --- |
| 2016-07-05T10:43:18.093+0800:25.395:[GC [PSYoungGen: 274931K->10738K(274944K)] 371093K->147186K(450048K), 0.0668480 secs] [Times: user=0.17 sys=0.08, real=0.07 secs]  2016-07-05T10:43:18.160+0800: 25.462: [Full GC [PSYoungGen: 10738K->0K(274944K)] [ParOldGen: 136447K->140379K(302592K)] 147186K->140379K(577536K) [PSPermGen: 85411K->85376K(171008K)], 0.6763541 secs] [Times: user=1.75 sys=0.02, real=0.68 secs] |

通过上面日志分析得出，PSYoungGen、ParOldGen、PSPermGen属于Parallel收集器。其中PSYoungGen表示gc回收前后年轻代的内存变化；ParOldGen表示gc回收前后老年代的内存变化；PSPermGen表示gc回收前后永久区的内存变化。young gc 主要是针对年轻代进行内存回收比较频繁，耗时短；full gc 会对整个堆内存进行回城，耗时长，因此一般尽量减少full gc的次数

young gc 日志:



Full GC日志:



调优命令

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

* jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
* jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
* jmap，JVM Memory Map命令用于生成heap dump文件
* jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看
* jstack，用于生成java虚拟机当前时刻的线程快照。
* jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

详细的命令使用参考这里[*JVM（4）：Jvm调优-命令篇*](http://www.importnew.com/?p=23761)

调优工具

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

* jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控
* jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。
* MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
* GChisto，一款专业分析gc日志的工具

1. JVM常见启动参数

-Xms / -Xmx — 堆的初始大小 / 堆的最大大小  
-Xmn — 堆中年轻代的大小  
-XX:-DisableExplicitGC — 让System.gc()不产生任何作用  
-XX:+PrintGCDetails — 打印GC的细节  
-XX:+PrintGCDateStamps — 打印GC操作的时间戳  
-XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小  
-XX:NewRatio — 可以设置老生代和新生代的比例  
-XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布  
-XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值  
-XX:TargetSurvivorRatio：设置幸存区的目标使用率

1. JAVA类生命周期

Java类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用、卸载七个阶段。

问题：

**1.类的实例化顺序，比如父类静态数据，构造函数，字段，子类静态数据，构造函数，字段，他们的执行顺序**

答：先静态、先父后子。   
先静态：父静态 > 子静态   
优先级：父类 > 子类静态代码块 > 非静态代码块 > 构造函数   
一个类的实例化过程：   
1，父类中的static代码块，当前类的static   
2，顺序执行父类的普通代码块   
3，父类的构造函数   
4，子类普通代码块   
5，子类（当前类）的构造函数，按顺序执行  
6，子类方法的执行

**2. Java 8的内存分代改进**

从永久代到元空间，在小范围自动扩展永生代避免溢出

**3. JVM垃圾回收机制，何时触发MinorGC等操作**

分代垃圾回收机制：不同的对象生命周期不同。把不同生命周期的对象放在不同代上，不同代上采用最合适它的垃圾回收方式进行回收。   
JVM中共划分为三个代：年轻代、年老代和持久代，   
年轻代：存放所有新生成的对象；   
年老代：在年轻代中经历了N次垃圾回收仍然存活的对象，将被放到年老代中，故都是一些生命周期较长的对象；   
持久代：用于存放静态文件，如Java类、方法等。   
新生代的垃圾收集器命名为“minor gc”，老生代的GC命名为”Full Gc 或者Major GC”.其中用System.gc()强制执行的是Full Gc.   
判断对象是否需要回收的方法有两种：   
1.引用计数   
当某对象的引用数为0时，便可以进行垃圾收集。   
2.对象引用遍历   
果某对象不能从这些根对象的一个（至少一个）到达，则将它作为垃圾收集。在对象遍历阶段，gc必须记住哪些对象可以到达，以便删除不可到达的对象，这称为标记（marking）对象。

触发GC（Garbage Collector）的条件：   
1)GC在优先级最低的线程中运行，一般在应用程序空闲即没有应用线程在运行时被调用。   
2)Java堆内存不足时，GC会被调用。

**4. jvm中一次完整的GC流程（从ygc到fgc）是怎样的，重点讲讲对象如何晋升到老年代等**

答：对象优先在新生代区中分配，若没有足够空间，Minor GC；   
大对象（需要大量连续内存空间）直接进入老年态；长期存活的对象进入老年态。如果对象在新生代出生并经过第一次MGC后仍然存活，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。

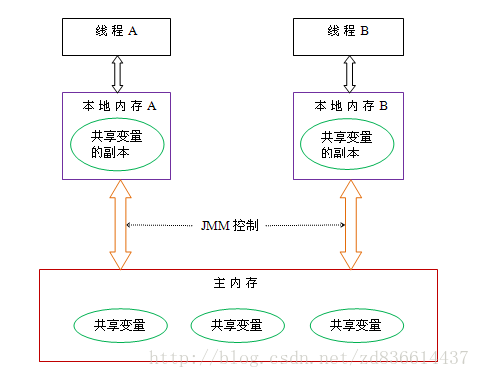
**5. Eden和Survivor的比例分配等**

默认比例8:1。   
大部分对象都是朝生夕死。   
复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

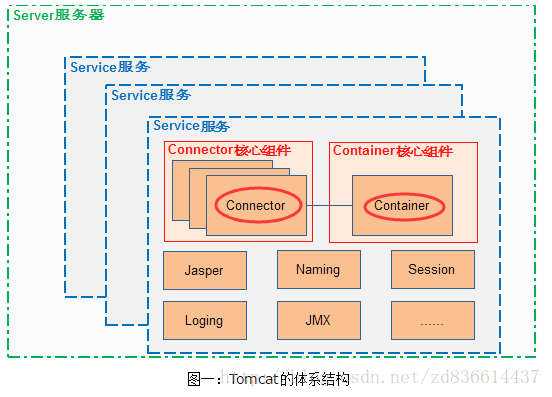
**7. 深入分析了Classloader，双亲委派机制**

ClassLoader：类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。   
双亲委派机制：某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

**9. 对Java内存模型的理解，以及其在并发中的应用**

Java内存模型的主要目标: 定义程序中各个变量的访问规则。   
Java线程之间的通信由Java内存模型（本文简称为JMM）控制。   
所有变量的存储都在主内存，每条线程还都有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作必须在工作内存完成，而不能直接读取主内存中的变量。不同的线程直接无法访问对方工作内存中的变量，线程间变量的传递均需要通过主内存来完成。   


线程间通信：   
1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。   
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

**10. tomcat结构，类加载器流程**   
   
目录结构：   
• /bin：存放windows或Linux平台上启动和关闭Tomcat的脚本文件   
• /conf：存放Tomcat服务器的各种全局配置文件，其中最重要的是server.xml和web.xml   
• /doc：存放Tomcat文档   
• /server：包含三个子目录：classes、lib和webapps   
• /server/lib：存放Tomcat服务器所需的各种JAR文件   
• /server/webapps：存放Tomcat自带的两个WEB应用admin应用和 manager应用   
• /common/lib：存放Tomcat服务器以及所有web应用都可以访问的jar文件   
• /shared/lib：存放所有web应用都可以访问的jar文件（但是不能被Tomcat服务器访问）   
• /logs：存放Tomcat执行时的日志文件   
• /src：存放Tomcat的源代码   
• /webapps：Tomcat的主要Web发布目录，默认情况下把Web应用文件放于此目录   
• /work：存放JSP编译后产生的class文件

**11. volatile的语义，它修饰的变量一定线程安全吗**

一个变量被定义为volatile之后，具备两重语义：①保证此变量对所有线程的可见性，即当一条线程修改了这个值，新值对于其他所有线程来说是立即得知的，普通变量需要通过主内存传递。②禁止指令重排序优化。   
Volatile修饰的变量不一定是线程安全的，eg非原子操作a++等

**12. 说一说你对环境变量classpath的理解？如果一个类不在classpath下，为什么会抛出ClassNotFoundException异常，如果在不改变这个类路径的前期下，怎样才能正确加载这个类？**

classpath是javac编译器的一个环境变量。它的作用与import、package关键字有关。package的所在位置，就是设置CLASSPATH当编译器面对import packag这个语句时，它先会查找CLASSPATH所指定的目录，并检视子目录java/util是否存在，然后找出名称吻合的已编译文件（.class文件）。如果没有找到就会报错！   
动态加载包

**13. 说一下强引用、软引用、弱引用、虚引用以及他们之间和gc的关系**

强引用：new出的对象之类的引用，   
只要强引用还在，永远不会回收   
软引用：引用但非必须的对象，内存溢出异常之前，回收   
弱引用：非必须的对象，对象能生存到下一次垃圾收集发生之前。   
虚引用：对生存时间无影响，在垃圾回收时得到通知。

**14. JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存等。**

重排序：jvm虚拟机允许在不影响代码最终结果的情况下，可以乱序执行。

内存屏障：可以阻挡编译器的优化，也可以阻挡处理器的优化

happens-before原则：

1：一个线程的A操作总是在B之前，那多线程的A操作肯定实在B之前。  
2：monitor 再加锁的情况下，持有锁的肯定先执行。  
3：volatile修饰的情况下，写先于读发生  
4：线程启动在一起之前 strat  
5：线程死亡在一切之后 end  
6：线程操作在一切线程中断之前  
7：一个对象构造函数的结束都该对象的finalizer的开始之前  
8：传递性，如果A肯定在B之前，B肯定在C之前，那A肯定是在C之前。

主内存：所有线程共享的内存空间

工作内存：每个线程特有的内存空间

**15. 讲讲JAVA的反射机制。**

在运行是，对于任意一个类，都能知道当前类的方法和属性，对于任意一个类，都能调用类的方法和属性，着用动态加载机制就是Java的反射机制。

1. 集合

0. HashMap、LinkedHashMap、ConcurrentHashMap、ArrayList、LinkedList的底层实现?

ArrayList实现原理要点概括

1. ArrayList是List接口的可变数组非同步实现，并允许包括null在内的所有元素。
2. 底层使用数组实现
3. 该集合是可变长度数组，数组扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量增长大约是其容量的1.5倍，这种操作的代价很高。
4. 采用了Fail-Fast机制，面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险
5. remove方法会让下标到数组末尾的元素向前移动一个单位，并把最后一位的值置空，方便GC

LinkedList实现原理要点概括

1. LinkedList是List接口的双向链表非同步实现，并允许包括null在内的所有元素。
2. 底层的数据结构是基于双向链表的，该数据结构我们称为节点
3. 双向链表节点对应的类Node的实例，Node中包含成员变量：prev，next，item。其中，prev是该节点的上一个节点，next是该节点的下一个节点，item是该节点所包含的值。
4. 它的查找是分两半查找，先判断index是在链表的哪一半，然后再去对应区域查找，这样最多只要遍历链表的一半节点即可找到

HashMap实现原理要点概括

1. HashMap是基于哈希表的Map接口的非同步实现，允许使用null值和null键，但不保证映射的顺序。
2. 底层使用数组实现，数组中每一项是个单向链表，即数组和链表的结合体；当链表长度大于一定阈值时，链表转换为红黑树，这样减少链表查询时间。
3. HashMap在底层将key-value当成一个整体进行处理，这个整体就是一个Node对象。HashMap底层采用一个Node[]数组来保存所有的key-value对，当需要存储一个Node对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Node时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Node。
4. HashMap进行数组扩容需要重新计算扩容后每个元素在数组中的位置，很耗性能
5. 采用了Fail-Fast机制，通过一个modCount值记录修改次数，对HashMap内容的修改都将增加这个值。迭代器初始化过程中会将这个值赋给迭代器的expectedModCount，在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map，马上抛出异常

Hashtable实现原理要点概括

1. Hashtable是基于哈希表的Map接口的同步实现，不允许使用null值和null键
2. 底层使用数组实现，数组中每一项是个单链表，即数组和链表的结合体
3. Hashtable在底层将key-value当成一个整体进行处理，这个整体就是一个Entry对象。Hashtable底层采用一个Entry[]数组来保存所有的key-value对，当需要存储一个Entry对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。
4. synchronized是针对整张Hash表的，即每次锁住整张表让线程独占

ConcurrentHashMap实现原理要点概括

1. ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。
2. 它使用了多个锁来控制对hash表的不同段进行的修改，每个段其实就是一个小的hashtable，它们有自己的锁。只要多个并发发生在不同的段上，它们就可以并发进行。
3. ConcurrentHashMap在底层将key-value当成一个整体进行处理，这个整体就是一个Entry对象。Hashtable底层采用一个Entry[]数组来保存所有的key-value对，当需要存储一个Entry对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。
4. 与HashMap不同的是，ConcurrentHashMap使用多个子Hash表，也就是段(Segment)
5. ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。

HashSet实现原理要点概括

1. HashSet由哈希表(实际上是一个HashMap实例)支持，不保证set的迭代顺序，并允许使用null元素。
2. 基于HashMap实现，API也是对HashMap的行为进行了封装，可参考HashMap

LinkedHashMap实现原理要点概括

1. LinkedHashMap继承于HashMap，底层使用哈希表和双向链表来保存所有元素，并且它是非同步，允许使用null值和null键。
2. 基本操作与父类HashMap相似，通过重写HashMap相关方法，重新定义了数组中保存的元素Entry，来实现自己的链接列表特性。该Entry除了保存当前对象的引用外，还保存了其上一个元素before和下一个元素after的引用，从而构成了双向链接列表。

LinkedHashSet实现原理要点概括

1. 对于LinkedHashSet而言，它继承与HashSet、又基于LinkedHashMap来实现的。LinkedHashSet底层使用LinkedHashMap来保存所有元素，它继承与HashSet，其所有的方法操作上又与HashSet相同。

1.Java集合框架是什么？说出一些集合框架的长处？

每种编程语言中都有集合。最初的Java版本号包括几种集合类：Vector、Stack、HashTable和Array。

随着集合的广泛使用。Java1.2提出了囊括全部集合接口、实现和算法的集合框架。

在保证线程安全的情况下使用泛型和并发集合类，Java已经经历了非常久。它还包括在Java并发包中，堵塞接口以及它们的实现。

集合框架的部分长处例如以下：

（1）使用核心集合类减少开发成本，而非实现我们自己的集合类。

（2）随着使用经过严格測试的集合框架类。代码质量会得到提高。

（3）通过使用JDK附带的集合类，能够减少代码维护成本。

（4）复用性和可操作性。

2.集合框架中的泛型有什么长处？

Java1.5引入了泛型。全部的集合接口和实现都大量地使用它。泛型同意我们为集合提供一个能够容纳的对象类型，因此。假设你加入其他类型的不论什么元素，它会在编译时报错。这避免了在执行时出现ClassCastException。由于你将会在编译时得到报错信息。

泛型也使得代码整洁，我们不须要使用显式转换和instanceOf操作符。

它也给执行时带来长处。由于不会产生类型检查的字节码指令。

3.Java集合框架的基础接口有哪些？

Collection为集合层级的根接口。

Set是一个不能包括反复元素的集合。

List是一个有序集合。能够包括反复元素。

Map是一个将key映射到value的对象.一个Map不能包括反复的key：每一个key最多仅仅能映射一个value。

一些其他的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

4.为何Collection不从Cloneable和Serializable接口继承？

Collection接口指定一组对象，对象即为它的元素。怎样维护这些元素由Collection的详细实现决定。

比如。一些如List的Collection实现同意反复的元素。而其他的如Set就不同意。非常多Collection实现有一个公有的clone方法。

然而。把它放到集合的全部实现中也是没有意义的。这是由于Collection是一个抽象表现。重要的是实现。

当与详细实现打交道的时候，克隆或序列化的语义和含义才发挥作用。所以，详细实现应该决定怎样对它进行克隆或序列化，或它能否够被克隆或序列化。

在全部的实现中授权克隆和序列化，终于导致更少的灵活性和很多其他的限制。特定的实现应该决定它能否够被克隆和序列化。

5.为何Map接口不继承Collection接口？

虽然Map接口和它的实现也是集合框架的一部分。但Map不是集合。集合也不是Map。

因此，Map继承Collection毫无意义，反之亦然。

假设Map继承Collection接口，那么元素去哪儿？Map包括key-value对，它提供抽取key或value列表集合的方法，可是它不适合“一组对象”规范。

6.Iterator是什么？

Iterator接口提供遍历不论什么Collection的接口。

我们能够从一个Collection中使用迭代器方法来获取迭代器实例。迭代器代替了Java集合框架中的Enumeration。迭代器同意调用者在迭代过程中移除元素。

7.Enumeration和Iterator接口的差别？

Enumeration的速度是Iterator的两倍，也使用更少的内存。

Enumeration是非常基础的，也满足了基础的须要。可是，与Enumeration相比，Iterator更加安全，由于当一个集合正在被遍历的时候。它会阻止其他线程去改动集合。

迭代器代替了Java集合框架中的Enumeration。

迭代器同意调用者从集合中移除元素，而Enumeration不能做到。

为了使它的功能更加清晰，迭代器方法名已经经过改善。

8.为何没有像Iterator.add()这种方法。向集合中加入元素？

语义不明。已知的是，Iterator的协议不能确保迭代的次序。然而要注意。ListIterator没有提供一个add操作，它要确保迭代的顺序。

9.为何迭代器没有一个方法能够直接获取下一个元素。而不须要移动游标？

它能够在当前Iterator的顶层实现。可是它用得非常少，假设将它加到接口中，每一个继承都要去实现它。这没有意义。

10.Iterater和ListIterator之间有什么差别？

（1）我们能够使用Iterator来遍历Set和List集合，而ListIterator仅仅能遍历List。

（2）Iterator仅仅能够向前遍历。而LIstIterator能够双向遍历。

（3）ListIterator从Iterator接口继承，然后加入了一些额外的功能，比方加入一个元素、替换一个元素、获取前面或后面元素的索引位置。

11.遍历一个List有哪些不同的方式？

List<String> strList = new ArrayList<>();

//使用for-each循环

for(String obj : strList){

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext()){

String obj = it.next();

System.out.println(obj);

}

使用迭代器更加线程安全。由于它能够确保，在当前遍历的集合元素被更改的时候。它会抛出ConcurrentModificationException。

12.通过迭代器fail-fast属性，你明确了什么？

每次我们尝试获取下一个元素的时候，Iterator fail-fast属性检查当前集合结构里的不论什么改动。假设发现不论什么改动。它抛出ConcurrentModificationException。Collection中全部Iterator的实现都是按fail-fast来设计的（ConcurrentHashMap和CopyOnWriteArrayList这类并发集合类除外）。

13.fail-fast与fail-safe有什么差别？

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中不论什么改动的影响。

Java.util包中的全部集合类都被设计为fail-fast的。而java.util.concurrent中的集合类都为fail-safe的。

Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

14.在迭代一个集合的时候，怎样避免ConcurrentModificationException？

在遍历一个集合的时候，我们能够使用并发集合类来避免ConcurrentModificationException，比方使用CopyOnWriteArrayList。而不是ArrayList。

15.为何Iterator接口没有详细的实现？

Iterator接口定义了遍历集合的方法。但它的实现则是集合实现类的责任。每一个能够返回用于遍历的Iterator的集合类都有它自己的Iterator实现内部类。

这就同意集合类去选择迭代器是fail-fast还是fail-safe的。比方，ArrayList迭代器是fail-fast的。而CopyOnWriteArrayList迭代器是fail-safe的。

16.UnsupportedOperationException是什么？

UnsupportedOperationException是用于表明操作不支持的异常。

在JDK类中已被大量运用，在集合框架java.util.Collections.UnmodifiableCollection将会在全部add和remove操作中抛出这个异常。

17.在Java中，HashMap是怎样工作的？

HashMap在Map.Entry静态内部类实现中存储key-value对。

HashMap使用哈希算法。在put和get方法中。它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候。HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。

Entry存储在LinkedList中，所以假设存在entry。它使用equals()方法来检查传递的key是否已经存在。假设存在，它会覆盖value。假设不存在。它会创建一个新的entry然后保存。

当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。下面的图片解释了详细内容。

其他关于HashMap比較重要的问题是容量、负荷系数和阀值调整。HashMap默认的初始容量是32，负荷系数是0.75。

阀值是为负荷系数乘以容量，不管何时我们尝试加入一个entry，假设map的大小比阀值大的时候，HashMap会对map的内容进行又一次哈希。且使用更大的容量。容量总是2的幂。所以假设你知道你须要存储大量的key-value对，比方缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

18.hashCode()和equals()方法有何重要性？

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。

假设这些方法没有被正确地实现，在这种情况下，两个不同Key或许会产生同样的hashCode()和equals()输出，HashMap将会觉得它们是同样的，然后覆盖它们。而非把它们存储到不同的地方。

同样的，全部不同意存储反复数据的集合类都使用hashCode()和equals()去查找反复。所以正确实现它们非常重要。

equals()和hashCode()的实现应该遵循下面规则：

（1）假设o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。

（2）假设o1.hashCode() == o2.hashCode()。并不意味着o1.equals(o2)会为true。

19.我们能否使用不论什么类作为Map的key？

我们能够使用不论什么类作为Map的key，然而在使用它们之前，须要考虑下面几点：

（1）假设类重写了equals()方法。它也应该重写hashCode()方法。

（2）类的全部实例须要遵循与equals()和hashCode()相关的规则。请參考之前提到的这些规则。

（3）假设一个类没有使用equals()，你不应该在hashCode()中使用它。

（4）用户自己定义key类的最佳实践是使之为不可变的。这样，hashCode()值能够被缓存起来，拥有更好的性能。

不可变的类也能够确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比方，我有一个类MyKey，在HashMap中使用它。

//传递给MyKey的name參数被用于equals()和hashCode()中

MyKey key = new MyKey('Pankaj'); //assume hashCode=1234

myHashMap.put(key, 'Value');

// 下面的代码会改变key的hashCode()和equals()值

key.setName('Amit'); //assume new hashCode=7890

//下面会返回null，由于HashMap会尝试查找存储同样索引的key。而key已被改变了，匹配失败，返回null

myHashMap.get(new MyKey('Pankaj'));

那就是为何String和Integer被作为HashMap的key大量使用。

20.Map接口提供了哪些不同的集合视图？

Map接口提供三个集合视图：

（1）Set keyset()：返回map中包括的全部key的一个Set视图。

集合是受map支持的，map的变化会在集合中反映出来，反之亦然。当一个迭代器正在遍历一个集合时。若map被改动了（除迭代器自身的移除操作以外），迭代器的结果会变为没有定义。

集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除。从map中移除相应的映射。它不支持add和addAll操作。

（2）Collection values()：返回一个map中包括的全部value的一个Collection视图。

这个collection受map支持的。map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个collection时，若map被改动了（除迭代器自身的移除操作以外），迭代器的结果会变为没有定义。

集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除相应的映射。它不支持add和addAll操作。

（3）Set<Map.Entry<K,V>> entrySet()：返回一个map钟包括的全部映射的一个集合视图。

这个集合受map支持的，map的变化会在collection中反映出来。反之亦然。当一个迭代器正在遍历一个集合时，若map被改动了（除迭代器自身的移除操作，以及对迭代器返回的entry进行setValue外），迭代器的结果会变为没有定义。

集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除相应的映射。它不支持add和addAll操作。

21.HashMap和HashTable有何不同？

（1）HashMap同意key和value为null。而HashTable不同意。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你非常easy从HashMap转向LinkedHashMap，可是HashTable不是这种。它的顺序是不可预知的。

（4）HashMap提供对key的Set进行遍历。因此它是fail-fast的。但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被觉得是个遗留的类。假设你寻求在迭代的时候改动Map，你应该使用CocurrentHashMap。

22.怎样决定选用HashMap还是TreeMap？

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而。假如你须要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，或许向HashMap中加入元素会更快。将map换为TreeMap进行有序key的遍历。

23.ArrayList和Vector有何异同点？

ArrayList和Vector在非常多时候都非常类似。

（1）两者都是基于索引的，内部由一个数组支持。

（2）两者维护插入的顺序，我们能够依据插入顺序来获取元素。

（3）ArrayList和Vector的迭代器实现都是fail-fast的。

（4）ArrayList和Vector两者同意null值。也能够使用索引值对元素进行随机訪问。

下面是ArrayList和Vector的不同点。

（1）Vector是同步的，而ArrayList不是。

然而。假设你寻求在迭代的时候对列表进行改变。你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector快。它由于有同步。不会过载。

（3）ArrayList更加通用，由于我们能够使用Collections工具类轻易地获取同步列表和仅仅读列表。

24.Array和ArrayList有何差别？什么时候更适合用Array？

Array能够容纳基本类型和对象，而ArrayList仅仅能容纳对象。

Array是指定大小的，而ArrayList大小是固定的。

Array没有提供ArrayList那么多功能，比方addAll、removeAll和iterator等。虽然ArrayList明显是更好的选择。但也有些时候Array比較好用。

（1）假设列表的大小已经指定，大部分情况下是存储和遍历它们。

（2）对于遍历基本数据类型，虽然Collections使用自己主动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得非常慢。

（3）假设你要使用多维数组，使用[][]比List<List<>>更easy。

25.ArrayList和LinkedList有何差别？

ArrayList和LinkedList两者都实现了List接口，可是它们之间有些不同。

（1）ArrayList是由Array所支持的基于一个索引的数据结构，所以它提供对元素的随机訪问。复杂度为O(1)，但LinkedList存储一系列的节点数据。每一个节点都与前一个和下一个节点相连接。所以。虽然有使用索引获取元素的方法，内部实现是从起始点開始遍历，遍历到索引的节点然后返回元素。时间复杂度为O(n)。比ArrayList要慢。

（2）与ArrayList相比，在LinkedList中插入、加入和删除一个元素会更快。由于在一个元素被插入到中间的时候，不会涉及改变数组的大小，或更新索引。

（3）LinkedList比ArrayList消耗很多其他的内存，由于LinkedList中的每一个节点存储了前后节点的引用。

26.哪些集合类提供对元素的随机訪问？

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机訪问。

27.EnumSet是什么？

java.util.EnumSet是使用枚举类型的集合实现。当集合创建时，枚举集合中的全部元素必须来自单个指定的枚举类型，能够是显示的或隐示的。EnumSet是不同步的，不同意值为null的元素。它也提供了一些实用的方法，比方copyOf(Collection c)、of(E first,E…rest)和complementOf(EnumSet s)。

28.哪些集合类是线程安全的？

Vector、HashTable、Properties和Stack是同步类，所以它们是线程安全的，能够在多线程环境下使用。

Java1.5并发API包括一些集合类。同意迭代时改动，由于它们都工作在集合的克隆上。所以它们在多线程环境中是安全的。

29.并发集合类是什么？

Java1.5并发包（java.util.concurrent）包括线程安全集合类，同意在迭代时改动集合。

迭代器被设计为fail-fast的，会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、 ConcurrentHashMap、CopyOnWriteArraySet。

30.BlockingQueue是什么？

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在加入一个元素时，它会等待队列中的可用空间。

BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不须要操心等待生产者有可用的空间。或消费者有可用的对象。由于它都在BlockingQueue的实现类中被处理了。

Java提供了集中BlockingQueue的实现，比方ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue等。

31.队列和栈是什么，列出它们的差别？

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列同意先进先出（FIFO）检索元素，但并不是总是这样。Deque接口同意从两端检索元素。

栈与队列非常类似，但它同意对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口。

32.Collections类是什么？

Java.util.Collections是一个工具类仅包括静态方法。它们操作或返回集合。它包括操作集合的多态算法，返回一个由指定集合支持的新集合和其他一些内容。这个类包括集合框架算法的方法，比方折半搜索、排序、混编和逆序等。

33.Comparable和Comparator接口是什么？

假设我们想使用Array或Collection的排序方法时。须要在自己定义类里实现Java提供Comparable接口。

Comparable接口有compareTo(T OBJ)方法，它被排序方法所使用。我们应该重写这种方法，假设“this”对象比传递的对象參数更小、相等或更大时，它返回一个负整数、0或正整数。

可是。在大多数实际情况下，我们想依据不同參数进行排序。

比方。作为一个CEO。我想对雇员基于薪资进行排序。一个HR想基于年龄对他们进行排序。这就是我们须要使用Comparator接口的情景。由于Comparable.compareTo(Object o)方法实现仅仅能基于一个字段进行排序，我们不能依据对象排序的须要选择字段。

Comparator接口的compare(Object o1, Object o2)方法的实现须要传递两个对象參数，若第一个參数比第二个小，返回负整数；若第一个等于第二个。返回0；若第一个比第二个大。返回正整数。

34.Comparable和Comparator接口有何差别？

Comparable和Comparator接口被用来对对象集合或者数组进行排序。Comparable接口被用来提供对象的自然排序。我们能够使用它来提供基于单个逻辑的排序。

Comparator接口被用来提供不同的排序算法，我们能够选择须要使用的Comparator来对给定的对象集合进行排序。

35.我们怎样对一组对象进行排序？

假设我们须要对一个对象数组进行排序，我们能够使用Arrays.sort()方法。假设我们须要排序一个对象列表，我们能够使用Collection.sort()方法。

两个类都实用于自然排序（使用Comparable）或基于标准的排序（使用Comparator）的重载方法sort()。Collections内部使用数组排序方法，全部它们两者都有同样的性能。仅仅是Collections须要花时间将列表转换为数组。

36.当一个集合被作为參数传递给一个函数时，怎样才干够确保函数不能改动它？

在作为參数传递之前，我们能够使用Collections.unmodifiableCollection(Collection c)方法创建一个仅仅读集合，这将确保改变集合的不论什么操作都会抛出UnsupportedOperationException。

37.我们怎样从给定集合那里创建一个synchronized的集合？

我们能够使用Collections.synchronizedCollection(Collection c)依据指定集合来获取一个synchronized（线程安全的）集合。

38.集合框架里实现的通用算法有哪些？

Java集合框架提供经常使用的算法实现，比方排序和搜索。Collections类包括这些方法实现。大部分算法是操作List的，但一部分对全部类型的集合都是可用的。部分算法有排序、搜索、混编、最大最小值。

39.大写的O是什么？举几个样例？

大写的O描写叙述的是。就数据结构中的一系列元素而言，一个算法的性能。

Collection类就是实际的数据结构。我们通常基于时间、内存和性能，使用大写的O来选择集合实现。

比方：

样例1：ArrayList的get(index i)是一个常量时间操作，它不依赖list中元素的数量。所以它的性能是O(1)。

样例2：一个对于数组或列表的线性搜索的性能是O(n)，由于我们须要遍历全部的元素来查找须要的元素。

40.与Java集合框架相关的有哪些最好的实践？

（1）依据须要选择正确的集合类型。比方，假设指定了大小，我们会选用Array而非ArrayList。

假设我们想依据插入顺序遍历一个Map，我们须要使用TreeMap。假设我们不想反复。我们应该使用Set。

（2）一些集合类同意指定初始容量。所以假设我们能够预计到存储元素的数量，我们能够使用它，就避免了又一次哈希或大小调整。

（3）基于接口编程，而非基于实现编程。它同意我们后来轻易地改变实现。

（4）总是使用类型安全的泛型。避免在执行时出现ClassCastException。

（5）使用JDK提供的不可变类作为Map的key，能够避免自己实现hashCode()和equals()。

（6）尽可能使用Collections工具类，或者获取仅仅读、同步或空的集合，而非编写自己的实现。

它将会提供代码重用性，它有着更好的稳定性和可维护性。

1. 线程

**1) 什么是线程？**

　　线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒。Java在语言层面对多线程提供了卓越的支持，它也是一个很好的卖点。

**2) 线程和进程有什么区别？**

　　线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

**3) 如何在Java中实现线程？**

　　在语言层面有两种方式。java.lang.Thread 类的实例就是一个线程但是它需要调用java.lang.Runnable接口来执行，由于线程类本身就是调用的Runnable接口所以你可以继承java.lang.Thread 类或者直接调用Runnable接口来重写run()方法实现线程。

**4) 用Runnable还是Thread？**

　　这个问题是上题的后续，大家都知道我们可以通过继承Thread类或者调用Runnable接口来实现线程，问题是，那个方法更好呢？什么情况下使用它？这个问题很容易回答，如果你知道Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口好了。

**6) Thread 类中的start() 和 run() 方法有什么区别？**

　　这个问题经常被问到，但还是能从此区分出面试者对Java线程模型的理解程度。start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

**7) Java中Runnable和Callable有什么不同？**

　　Runnable和Callable都代表那些要在不同的线程中执行的任务。Runnable从JDK1.0开始就有了，Callable是在JDK1.5增加的。它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。Callable可以返回装载有计算结果的Future对象。

**8) Java中CyclicBarrier 和 CountDownLatch有什么不同？**

　　CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用。[点此查看更多信息和示例代码](http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html)。

**9) Java内存模型是什么？**

　　Java内存模型规定和指引Java程序在不同的内存架构、CPU和操作系统间有确定性地行为。它在多线程的情况下尤其重要。Java内存模型对一个线程所做的变动能被其它线程可见提供了保证，它们之间是先行发生关系。这个关系定义了一些规则让程序员在并发编程时思路更清晰。比如，先行发生关系确保了：

* 线程内的代码能够按先后顺序执行，这被称为程序次序规则。
* 对于同一个锁，一个解锁操作一定要发生在时间上后发生的另一个锁定操作之前，也叫做管程锁定规则。
* 前一个对volatile的写操作在后一个volatile的读操作之前，也叫volatile变量规则。
* 一个线程内的任何操作必需在这个线程的start()调用之后，也叫作线程启动规则。
* 一个线程的所有操作都会在线程终止之前，线程终止规则。
* 一个对象的终结操作必需在这个对象构造完成之后，也叫对象终结规则。
* 可传递性

　　我强烈建议大家阅读《Java并发编程实践》第十六章来加深对Java内存模型的理解。

**10) Java中的volatile 变量是什么？**

　　volatile是一个特殊的修饰符，只有成员变量才能使用它。在Java并发程序缺少同步类的情况下，多线程对成员变量的操作对其它线程是透明的。volatile变量可以保证下一个读取操作会在前一个写操作之后发生，就是上一题的volatile变量规则。

**11) 什么是线程安全？Vector是一个线程安全类吗？**

　　如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

**12) Java中什么是竞态条件？ 举个例子说明。**

　　竞态条件会导致程序在并发情况下出现一些bugs。多线程对一些资源的竞争的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争。

**13) Java中如何停止一个线程？**

　　Java提供了很丰富的API但没有为停止线程提供API。JDK 1.0本来有一些像stop(), suspend() 和 resume()的控制方法但是由于潜在的死锁威胁因此在后续的JDK版本中他们被弃用了，之后Java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程。

**14) 一个线程运行时发生异常会怎样？**

　　这是我在一次面试中遇到的一个[很刁钻的Java面试题](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html), 简单的说，如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理。

**15） 如何在两个线程间共享数据？**

　　你可以通过共享对象来实现这个目的，或者是使用像阻塞队列这样并发的数据结构。这篇教程[《Java线程间通信》](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html)(涉及到在两个线程间共享对象)用wait和notify方法实现了生产者消费者模型。

**16) Java中notify 和 notifyAll有什么区别？**

　　这又是一个刁钻的问题，因为多线程可以等待单监控锁，Java API 的设计人员提供了一些方法当等待条件改变的时候通知它们，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

**17) 为什么wait, notify 和 notifyAll这些方法不在thread类里面？**

　　这是个设计相关的问题，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在Object类里是有意义的，还有不把它放在Thread类里的原因。一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

**18) 什么是ThreadLocal变量？**

　　ThreadLocal是Java里一种特殊的变量。每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用ThreadLocal让SimpleDateFormat变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。线程局部变量的另一个不错的例子是ThreadLocalRandom类，它在多线程环境中减少了创建代价高昂的Random对象的个数。

**19) 什么是FutureTask？**

　　在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。

**20) Java中interrupted 和 isInterruptedd方法的区别？**

　　interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用[静态方法](http://java67.blogspot.com/2012/11/what-is-static-class-variable-method.html)Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变。

**21) 为什么wait和notify方法要在同步块中调用？**

　　主要是因为Java API强制要求这样做，如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。还有一个原因是为了避免wait和notify之间产生竞态条件。

**22) 为什么你应该在循环中检查等待条件?**

　　处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因，你可以在[Eclipse](http://res.importnew.com/eclipse)中创建模板调用wait和notify试一试。如果你想了解更多关于这个问题的内容，我推荐你阅读《[Effective Java](http://www.amazon.com/gp/product/B000WJOUPA/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000WJOUPA&linkCode=as2&tag=job0ae-20)》这本书中的线程和同步章节。

**23) Java中的同步集合与并发集合有什么区别？**

　　同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

**24） Java中堆和栈有什么不同？**

　　为什么把这个问题归类在多线程和并发面试题里？因为栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

**25） 什么是线程池？ 为什么要使用它？**

　　创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

**26） 如何写代码来解决生产者消费者问题？**

　　在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道怎么进行线程间通信来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用Semaphore 或者 BlockingQueue来实现生产者消费者模型。

**27） 如何避免死锁？**

　　Java多线程中的死锁 死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：

* 互斥条件：一个资源每次只能被一个进程使用。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
* 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**28) Java中活锁和死锁有什么区别？**

　　这是上题的扩展，活锁和死锁类似，不同之处在于处于活锁的线程或进程的状态是不断改变的，活锁可以认为是一种特殊的饥饿。一个现实的活锁例子是两个人在狭小的走廊碰到，两个人都试着避让对方好让彼此通过，但是因为避让的方向都一样导致最后谁都不能通过走廊。简单的说就是，活锁和死锁的主要区别是前者进程的状态可以改变但是却不能继续执行。

**29） 怎么检测一个线程是否拥有锁？**

　　我一直不知道我们竟然可以检测一个线程是否拥有锁，直到我参加了一次电话面试。在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

**30) 你如何在Java中获取线程堆栈？**

　　对于不同的操作系统，有多种方法来获得Java进程的线程堆栈。当你获取线程堆栈时，JVM会把所有线程的状态存到日志文件或者输出到控制台。在Windows你可以使用Ctrl + Break组合键来获取线程堆栈，Linux下用kill -3命令。你也可以用jstack这个工具来获取，它对线程id进行操作，你可以用jps这个工具找到id。

**31) JVM中哪个参数是用来控制线程的栈堆栈小的**

　　这个问题很简单， -Xss参数用来控制线程的堆栈大小。

**32） Java中synchronized 和 ReentrantLock 有什么不同？**

　　Java在过去很长一段时间只能通过synchronized关键字来实现互斥，它有一些缺点。比如你不能扩展锁之外的方法或者块边界，尝试获取锁时不能中途取消等。Java 5 通过Lock接口提供了更复杂的控制来解决这些问题。 ReentrantLock 类实现了 Lock，它拥有与 synchronized 相同的并发性和内存语义且它还具有可扩展性。

**33） 有三个线程T1，T2，T3，怎么确保它们按顺序执行？**

　　在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

**34) Thread类中的yield方法有什么作用？**

　　Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

**35） Java中ConcurrentHashMap的并发度是什么？**

　　ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

**36） Java中Semaphore是什么？**

　　Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如数据库连接池。

**37）如果你提交任务时，线程池队列已满。会时发会生什么？**

　　这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常。

**38) Java线程池中submit() 和 execute()方法有什么区别？**

　　两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

**39) 什么是阻塞式方法？**

　　阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

**40) Swing是线程安全的吗？ 为什么？**

　　你可以很肯定的给出回答，Swing不是线程安全的，但是你应该解释这么回答的原因即便面试官没有问你为什么。当我们说swing不是线程安全的常常提到它的组件，这些组件不能在多线程中进行修改，所有对GUI组件的更新都要在AWT线程中完成，而Swing提供了同步和异步两种回调方法来进行更新。

**41） Java中invokeAndWait 和 invokeLater有什么区别？**

　　这两个方法是Swing API 提供给Java开发者用来从当前线程而不是事件派发线程更新GUI组件用的。InvokeAndWait()同步更新GUI组件，比如一个进度条，一旦进度更新了，进度条也要做出相应改变。如果进度被多个线程跟踪，那么就调用invokeAndWait()方法请求事件派发线程对组件进行相应更新。而invokeLater()方法是异步调用更新组件的。

**42) Swing API中那些方法是线程安全的？**

　　这个问题又提到了swing和线程安全，虽然组件不是线程安全的但是有一些方法是可以被多线程安全调用的，比如repaint(), revalidate()。 JTextComponent的setText()方法和JTextArea的insert() 和 append() 方法也是线程安全的。

**43) 如何在Java中创建Immutable对象？**

　　这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

**44） Java中的ReadWriteLock是什么？**

　　一般而言，读写锁是用来提升并发程序性能的锁分离技术的成果。Java中的ReadWriteLock是Java 5 中新增的一个接口，一个ReadWriteLock维护一对关联的锁，一个用于只读操作一个用于写。在没有写线程的情况下一个读锁可能会同时被多个读线程持有。写锁是独占的，你可以使用JDK中的ReentrantReadWriteLock来实现这个规则，它最多支持65535个写锁和65535个读锁。

**45) 多线程中的忙循环是什么?**

　　忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

**46）volatile 变量和 atomic 变量有什么不同？**

　　这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

**47) 如果同步块内的线程抛出异常会发生什么？**

　　这个问题坑了很多Java程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁，该功能可以在[finally block](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html)里释放锁实现。

**48） 单例模式的双检锁是什么？**

　　这个问题在Java面试中经常被问到，但是面试官对回答此问题的满意度仅为50%。一半的人写不出双检锁还有一半的人说不出它的隐患和Java1.5是如何对它修正的。它其实是一个用来创建线程安全的单例的老方法，当单例实例第一次被创建时它试图用单个锁进行性能优化，但是由于太过于复杂在JDK1.4中它是失败的，我个人也不喜欢它。无论如何，即便你也不喜欢它但是还是要了解一下，因为它经常被问到。

**49） 如何在Java中创建线程安全的Singleton？**

　　这是上面那个问题的后续，如果你不喜欢双检锁而面试官问了创建Singleton类的替代方法，你可以利用JVM的类加载和静态变量初始化特征来创建Singleton实例，或者是利用枚举类型来创建Singleton，我很喜欢用这种方法。

**50) 写出3条你遵循的多线程最佳实践**

　　这种问题我最喜欢了，我相信你在写并发代码来提升性能的时候也会遵循某些最佳实践。以下三条最佳实践我觉得大多数Java程序员都应该遵循：

* 给你的线程起个有意义的名字。 这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。
* 避免锁定和缩小同步的范围 锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。
* 多用同步类少用wait 和 notify 首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。
* 多用并发集合少用同步集合 这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。我的文章[Java并发集合](http://javarevisited.blogspot.com/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html)有更详细的说明。

**51) 如何强制启动一个线程？**

　　这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

**52) Java中的fork join框架是什么？**

　　fork join框架是JDK7中出现的一款高效的工具，Java开发人员可以通过它充分利用现代服务器上的多处理器。它是专门为了那些可以递归划分成许多子模块设计的，目的是将所有可用的处理能力用来提升程序的性能。fork join框架一个巨大的优势是它使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行。

**53） Java多线程中调用wait() 和 sleep()方法有什么不同？**

Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

初始化线程池（4种）

Java线程池的工厂类：Executors类,

初始化4种类型的线程池：

newFixedThreadPool()

说明：初始化一个指定线程数的线程池，其中corePoolSize == maxiPoolSize，使用LinkedBlockingQuene作为阻塞队列  
特点：即使当线程池没有可执行任务时，也不会释放线程。

newCachedThreadPool()

说明：初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；  
特点：在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源；当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；  
因此，使用时要注意控制并发的任务数，防止因创建大量的线程导致而降低性能。

newSingleThreadExecutor()  
说明：初始化只有一个线程的线程池，内部使用LinkedBlockingQueue作为阻塞队列。  
特点：如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行

newScheduledThreadPool()  
特定：初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据。

总结：除了newScheduledThreadPool的内部实现特殊一点之外，其它线程池内部都是基于ThreadPoolExecutor类（Executor的子类）实现的。

ThreadPoolExecutor内部具体实现：

ThreadPoolExecutor类构造器语法形式：

ThreadPoolExecutor（corePoolSize,maxPoolSize,keepAliveTime,timeUnit,workQueue,threadFactory,handle);

方法参数：  
corePoolSize：核心线程数  
maxPoolSize：最大线程数

keepAliveTime：线程存活时间（在corePore<\*<maxPoolSize情况下有用）  
timeUnit：存活时间的时间单位  
workQueue：阻塞队列（用来保存等待被执行的任务）

注：关于workQueue参数的取值,JDK提供了4种阻塞队列类型供选择：  
ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务；  
LinkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于

SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于ArrayBlockingQuene；

PriorityBlockingQuene：具有优先级的无界阻塞队列；

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，有以下四种取值

 注： 当线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

 线程池的状态（5种）

其中AtomicInteger变量ctl的功能非常强大：利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态：  
1、RUNNING：-1 << COUNT\_BITS，即高3位为111，该状态的线程池会接收新任务，并处理阻塞队列中的任务；  
2、SHUTDOWN： 0 << COUNT\_BITS，即高3位为000，该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；  
3、STOP ： 1 << COUNT\_BITS，即高3位为001，该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；  
4、TIDYING ： 2 << COUNT\_BITS，即高3位为010，该状态表示线程池对线程进行整理优化；  
5、TERMINATED： 3 << COUNT\_BITS，即高3位为011，该状态表示线程池停止工作；

向线程池提交任务（2种）

有两种方式：

Executor.execute(Runnable command);

ExecutorService.submit(Callable<T> task);

execute()内部实现

1.首次通过workCountof()获知当前线程池中的线程数，如果小于corePoolSize, 就通过addWorker()创建线程并执行该任务；否则，将该任务放入阻塞队列；

2. 如果能成功将任务放入阻塞队列中,

如果当前线程池是非RUNNING状态，则将该任务从阻塞队列中移除，然后执行reject()处理该任务；

如果当前线程池处于RUNNING状态，则需要再次检查线程池（因为可能在上次检查后，有线程资源被释放），是否有空闲的线程；如果有则执行该任务；

3、如果不能将任务放入阻塞队列中,说明阻塞队列已满；那么将通过addWoker()尝试创建一个新的线程去执行这个任务；如果addWoker()执行失败，说明线程池中线程数达到maxPoolSize,则执行reject()处理任务；

 sumbit()内部实现

会将提交的Callable任务会被封装成了一个FutureTask对象

FutureTask类实现了Runnable接口，这样就可以通过Executor.execute()提交FutureTask到线程池中等待被执行，最终执行的是FutureTask的run方法；

比较：

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

线程池的关闭（2种）

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

 线程池容量的动态调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()，

总结：

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果阻塞队列满了，那就创建新的线程执行当前任务；直到线程池中的线程数达到maxPoolSize,这时再有任务来，只能执行reject()处理该任务；