1. 简述ThreadLocal的作用，用在什么地方？

ThreadLocal为每个线程维护一个本地变量，采用空间换时间，用于线程间的数据隔离，为每一个使用该变量的线程提供一个副本，每个线程都可以独立改变自己的副本，而不会和其他线程的副本冲突。

ThreadLocal类中维护一个map，key是每个线程对象，值是对应线程的变量副本。

注意Synchronized和ThreadLocal根本不是一回事！Synchronized用于多线程访问共享变量的安全性，而ThradLocal仅仅是多线程之间数据隔离！

1. 既然了解JVM，那么说说你说了解的吧？
   1. 什么事JVM？

Java Virtual Machine，包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。JVM屏蔽了与具体操作系统的相关信息，是java程序只需生成在JVM上运行字节码，就可以在多平台上运行，jvm在执行字节码的时候会把指令解释成具体平台系统的机器执行码。一次编译到处执行。

2.2 JRE/JDK/JVM它们是什么关系？

JRE：java运行环境，所有java程序都要在jre下运行，jvm是jre的一部分

JDK：是开发者用来编译、调试java程序的开发工具包，jdk本身也是java程序，同样需要jre环境才能运行。

JVM：是JRE的一部分，虚构出来的计算机，在真实物理机上模拟各种计算机功能来实现的。它有自己的硬件架构。

2.3 JVM工作流程？



.java源码——.class字节码——类加载器——字节码校验器——解释器——硬件

关于解释器与JIT代码生成器的区别？

我们已经知道了JVM的整个工作流程是java到class到加载器到可执行代码的过程，那么解析器就是将二进制的一行一行的字节码解释成本系统可以执行的语言。而JIT的作用就是将经常被调用执行（根据调用次数决定）代码段或者方法翻译成机器码，缓存起来，这样效率会高。

2.4 JVM内存结构或者说是体系结构？

注意要和JMM内存模型区分！



从上图看出，线程私有的是栈、程序计数寄存器、本地方法栈；共享的有方法区、堆

①java内存栈：栈内存地址是不连续的，每个线程都拥有自己的栈。栈里面存着StacFrame（栈针）。StackFrame包含三类信息：局部变量，执行环境，操作数栈。

局部变量用来存储一个类的方法中所用到的局部变量。所以迭代、递归都可能导致栈溢出。

执行环境用于保存解释器对于java字节码进行解释过程中需要的信息，包括上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。

操作数栈用于存储运算所需要的操作数和结果。

②Java堆是用来存放对象信息的。和stack不同，stack代表着一种运行状态是运行时单位，解决程序该如何运行，而堆是存储的单位，解决数据存储的问题。heap是伴随着JVM的启动而创建，负责存储所有对象实例和数组的。堆的存储空间和栈一样不需要连续。

③程序计数寄存器是一块较小的内存空间，作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型中，字节码解释器工作就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器。

④方法区，又称持久代（jdk7之后从持久代剔除了）。方法区中存放了所有加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义的final类型的常量、类中的fields信息、类中方法信息，方法区是全局共享的，一定条件下会被gc

⑤运行时常量池，存放的为类中的固定的常量信息、方法和firld的引用信息等，其空间从方法区中分配。

⑥本地方法栈，JVM采用本地方法堆栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

需要注意：：：这也是一直困扰我的问题就是方法区中到底存放的是什么，一定要搞明白，不然后面的多态性将变得模糊！

首先我们常说的常量池——指的是运行时常量池，在方法区中分配，而一开始运行时常量池是不存在的，也就是说类的基本信息、方法、字段及其符号引用信息、静态变量、常量都在方法区中，当类加载完毕时，就会出现运行时常量池。所以总结如下：

①当刚刚开始类加载的时候

方法区：存放着.class类文件的基本信息，如类版本、字段、方法等

静态常量池（类的常量池）：存放着一些字面量和符号引用，字面量包括字符串，基本数据类型的常量（基本数据类型的范围是-128-127），符号引用其实引用的就是常量池里面的字符串，但符号引用不是直接引用，而是一种索引，静态常量池属于方法区

②当类加载完毕的时候

方法区：仍然存放着.class类文件的基本信息

动态常量池：静态常量池中的全部移到这里，其中一部分的符号引用变为直接引用，比如静态方法或私有方法，构造方法，父类方法，因为这些方法不能被重写其他版本，所以能在类加载的时候直接变为直接引用，而其他的一些方法在这个方法第一次被调用的时候才会将符号引用转变为直接引用。

另外动态常量池可以动态手动将变量放入常量池，但是jvm不会自己这样做。

jdk1.8之后移除了方法区，改用metaspace元空间替代，同时将String常量放到了堆中。

既然谈到了这个问题就说一下吧：

其实在jdk1.7的时候就已经开始转移永久代了，比如string字面量和静态变量都放在heap中而不是PermGen（永久代，方法区中），但是永久代还存在，而jdk1.8时就移除了方法区，而持久代变成的元空间，关键是元空间并不占用jvm内存，而是本地内存！！！

所以在使用jdk1.8时不需要指定永久代大小了而是需要指定元空间大小

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间适当提高该值。

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

-XX:MinMetaspaceFreeRatio，在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集

-XX:MaxMetaspaceFreeRatio，在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

至于为什么要这么做，主要有两个原因，第一字符串字面量在永久代所以极易导致oom；第二类和方法的信息等比较难以确定其大小，如果永久代设置过大，那么年老代可能会溢出，相反永久代会溢出。

2.5 JMM内存模型？

聊java memory model之前需要先知道计算机系统的内存模型的规范，在学习linux的cache、buffer缓存的时候已经知道buffer的目的是降低对磁盘的危害，通过缓冲多次写操作达到提高性能、保护磁盘的目的。而cache主要针对文件数据的读操作，主要分为两方面，第一种通过将文件数据缓存到内存中减少磁盘寻道次数和时间提高性能；另一种是通过现代cpu自带的多级缓存进一步提高程序访问内存的速度。此时出现一个问题就是多核cpu情况下怎么保证缓存的一致性？？？——————内存屏障（各系统实现不同）。

而jvm是的目的是跨平台运行，所以jvm就要有自己的内存模型，并且这个模型要符合计算机内存规范。

注意JMM和jvm结构的不同，jvm结构代表jvm内存的划分情况而JMM指的是共享内存区域中的变量、对象到底是如何共享的，是怎样的机制可以保证数据的一致性。说白了JMM如何实现数据在线程中与共享内存中的一致性。因为线程相互隔离的，所以多线程操作共享变量的时候怎么个机制。



java内存模型对主存与工作内存之间具体交互协议定义了八种操作：lock、unlock、read、load、use、assign、store、write

jmm还会对执行进行重排序操作，编译器和处理器经常对执行进行重排序：编译器优化重排序、指令级并行重排序、内存系统重排序。

总结：JMM内存模型符合并发编程的原子性、可见性、有序性。

原子性：操作要么执行完要么不执行不可打断，比如说long和double是64位的，在32位操作系统中需要两次操作，所以存在非原子操作。可以通过synchronized保证原子性。

可见性：当一个线程改变了一个共享变量，其他线程立即感知到变化，一般来说是做不到的，但是可以通过volitile修饰符保证变量的可见性，每次使用volitile修饰的变量都需要从主存中刷新，也可以用synchronized、lock、cinal保证变量可见性。

有序性：java内存模型中的指令重排序不会影响单线程的执行顺序，但是会影响并发执行的正确定。为保证并发情况下的有序性，可以通过volitile、synchronized、lock保证有序性，syncronized和lock保证了某一个时刻只有一个线程执行同步代码。

2.6 GC垃圾回收机制？

关注三个问题：哪些内存要回收、什么时候回收、怎么回收

从JVM内存结构可知，内存划分为5大区域（上面写了6个），其中线程私有的栈、程序计数寄存器、本地方法栈是随着栈自生自灭的，栈针的空间在编译的时候就确定了的，所以不用过度关心。而堆和方法区则不然，一个接口的实现类都不一样、一个方法中多个分支需要的内存也不一样，只有运行期间知道，所以这部分的内存的分配和回收都是动态的，gc就是关注这部分内存。其中堆是重点关注。

堆区域划分：新生代、老年代、永久代（1.8之后采用元空间，就不在堆中了）

第一个关注点：判断对象是否存活算法：

①引用计数法，采用引用计数器，当为0时对象不再被引用。优点是效率高，弊端是循环相互引用问题无法避免。

②可达性分析算法，通过多个可以称为GC Roots的对象为起点，所有所经过的路径为引用链，当一个对象到GC Roots没有任何引用则证明对象不可用了。优点是避免了相互利用，弊端就是效率低。

那么随之而来的问题就是GC Roots对象怎么选择？



注意：并不是不可达对象就一定会被回收。因为finalize函数如果被重写的话，可能存在自救的过程，但是这个finalize只能执行一次，下次再发现不可达就直接gc掉。注意：gc不保证对象的finalize一定被调用（出于安全的考虑），所以如果JVM认为这个对象不可达，同时没有重写finalize（或者已经被调用过），那么可以直接回收，如果不可达，但是重写了finalize（还没调用过）此时会看JVM的心情，JVM认为没必要调用finalize，那就可以直接回收，如果认为有必要调用finalize，就会加入执行finalize的对象队列，注意：：jvm可不确定一定让你执行完！！！因为谁也不知道finalize方法中是什么，可能是死循环，此时当对这些可能自救的对象进行小规模的第二次可达性分析时，把这个标记上了，那么就可以直接回收，如果发现没有再次标记，那么说明finalize自救成功。所以重写了finalize方法的类对象可能会经过两次标记！

当然了既然是对象的引用与否，肯定涉及引用的分类：强、若、弱、虚，在后面介绍。

引申问题：GC Roots怎么确定根节点呢？

已经知道可以作为GC Roots节点主要是一些全局引用（静态属性、常量）、执行上下文如栈针中本地变量表中的变量。问题是难道当进行gc时所有执行线程停掉之后对方法区和栈进行遍历？效率太低！

解决办法：实际上当系统停下来之后JVM不需要一个个检查引用，而是通过OopMap数据结构（Hotspot的叫法）来标记对象引用：

①在类加载的时候，计算对象内什么偏移量上是什么类型的数据，这样对象内的引用就可以一网打尽。

②在JIT编译时，会记录栈、寄出去你中的哪些位置时引用的对象、这样就不用遍历的。

注意：:引用是可以发生变化的，所以OopMap也是变化的，不可能每次变化都修改（或新增）OopMap吧！！所以出现了safepoint安全点——特定位置极力Oopmap引用关系，要知道safepoint太多可能有gc压力，太少的话可能gc等待太久，会导致stop the world时间过长，一般来说safepoint选取在运行时间长的地方比如循环调用或结束、方法调用、异常调转等。

当运行到这些safepoint的时候GC会让所有线程（执行线程）停顿：中断线程的方式：

①抢断式中断————基本不用了

gc发生时直接停掉所有执行线程，然后判断是都在sagepoint上，如果不在则让其运行到safepoint上

②主动式中断

gc发生时不会停掉线程，而是在safepoint处添加一个标志，所有执行线程会不断判断自己是否在这个标志处（safepoint处），如果是就停掉自己。

注意：：：还有一个概念是saferegion安全区，为什么需要saferegion，不是有了safepoint了吗？因为如果一个线程当前并未运行而是blocking状态，那么无论如何都不会运行到safepoint，所以提出了saferegion安全区，只要出于安全区的线程就可以参与GC，那么什么是saferegion——一段并不会发生引用改变的代码区域。

第二个关注点：怎么回收那些可以回收的内存？

三大垃圾回收算法：标记/清除算法、复制算法、标记/整理算法

JVM采用终极回收算法————分代收集算法（其实就是组合）

①标记/清除算法：分为标记和清除两个阶段，标记是通过gc roots可达性分析，将可达的对象的head标记可达，第二步遍历堆内存将head中没有标记可达的对象清除；弊端：两次遍历的效率问题、清除后空间碎片过多，很难申请庞大的对象空间

②复制算法：将堆划分两部分，每次只使用其中一块，回收时将存活对象复制到另一块空间中，然后将原来的空间直接全部清除。有点：效率高了一些，空间碎片问题也避免了。弊端：存在极端情况大部分对象存活时间很长不需要清理，那么每次都复制很多对象，并且可用空间减少一半，这样会存在性能问题。

③标记/整理算法：与标记/清除算法的第一步标记是一样的，区别在于第二步不是直接清除不可用对象，而是将可用对象移动到一起，将不可用对象清除。这样就避免了内存碎片问题、同时降低了复制算法浪费空间问题。弊端：效率其实并不高，不仅要标记对象而且还要改变对象引用地址，在效率上不如复制算法

总结：新生代采用复制算法，年老代采用标记/清除或者标记/整理算法

关于新生代的回收，划分区域不是对半分，而是按照8:1:1的比例划分为eden、from suvivor、to survivor，每次垃圾回收将eden存活的和from suvivor中未达到存活年龄的复制到to survivor中，然后交换from survivor和to survivor的名字（想象的）。from suvivor中达到默认15次垃圾回收就移动到年老代。

什么是垃圾回收器？

说白了就是垃圾回收算法的各种实现。可以针对不同场景及用户设置来使用不同的收集器

年轻代收集器：Serial、ParNew、Parallel Scavenge

老年代收集器：Serial Old、Parallel Old、CMS

特殊收集器：G1收集器新型

①Serial：对于Client模式下的jvm是个好选择，适合单核cpu，缺点是暂停及其线程，多核浪费

②ParNew：支持多线程，server模式下的jvm在新生代区是首选重点是可以配合CMS

③Parallel Scavenge：采用复制算法的收集器，支持多线程，特点是吞吐量大，减少用户界面由于gc原因的卡顿

④Serial Old：单线程，是Serial的老年代版本，不过采用标记清除算法

⑤Parallel Old：支持多线程，Parallel Scavenge老年代版本，采用标记整理算法

⑥CMS：以获取最短回收停顿时间为目的。之所以CMS在用户卡顿方面做得好————根本原因是可以和用户线程并发执行，这是其他收集器不能做到的。缺点是标记清除的碎片问题

⑦G1：（Garbage first）尽可能多回收垃圾，避免fullgc，同样关注卡顿时间，它解决了cms产生的空间碎片问题————空间压缩功能。特点是不分代，而是分区域，新生代老年代都可以清除。最大特点是只有初始标记的那个短暂时间stop the world，其他时候并不停掉其他线程

最后一个问题：垃圾回收模型有哪些？

Minor GC、Major GC、FULL GC、mixed GC

①Minor GC：只会清理年轻代

②Major GC：只清理年老代，但是一般来说不特别说明的话可以认为和Full GC等价，但是收集年老代同时也会伴随收集年轻代，收集整个JAVA堆，所以跟full gc还是有点区别，通常来说只说Minor GC和Full GC

③FULL GC：对年轻代、老年代、永久代（jdk8之后没有了）统一回收

④mixed GC（G1收集器特有）：这就解释了为什么G1不分代而是分区域，用于收集年轻代和老年代的不分区域

什么是空间分配担保？

这发生在两个地方，第一是当Minorgc时如果tosuvivor区域无法承受那么将大对象直接移到老年代；第二是当Minor gc要把存留时间长的对象移动到老年代的之前会做一个判断，比如之前都是移动9M，那么现在老年代时候有9M的剩余空间，如果有就进行移动，但是如果这次需要移动的空间大于9M，也就是说空间担保不行了，那么会触及Full GC。

1. String、StringBuffer、StringBuilder的区别？

String是字符串常量，顾名思义是不可变的，每次操作都会产生新对象，StringVBuilder和StringBuffer则是字符串变量，每次操作是一个对象，而区别在于StringBuffer支持同步锁机制是线程安全的。而效率上是String>StringBuilder>StringBuffer，所以少量字符串用String，大量字符串单线程用Builder，多线程用Buffer。

1. Synchronized与Lock的区别？

ReentrantLock拥有与Synchronized相同的并发和内存语义，此外还多了锁投票、定时等候锁和中断锁等候

比如：线程A和B都要获得O的锁，假如A先获得了锁，那么如果使用synchronized的话B必须等待A释放才行不能被中断，但是如果是ReentrantLock，如果A一直不释放，B等待一定时间可以终端等待干别的事

ReentrantLock获取锁的三种方法：

①lock()，如果获得锁立即返回，如果别的线程占用锁，那么一直等待，跟synchronized一样，当然还是有些区别的，下面介绍

②tryLock()，如果获得锁立即返回true，被占用的话立即返回false

③tryLock(long timeout, Timeout unit)如果获得锁立即返回true，被占用的话会等待一定时间，在等待过程中获得锁就返回true，否则返回false

③lockInterruptily()，如果获得锁立即返回，被占用的话当前线程休眠直到获得锁或者被人工中断。

注意：synchronized是JVM层面上实现的，所以如果代码中出现异常，JVM会释放锁，但是lock是代码中实现的所以一旦出现问题不能解决，并且释放锁的时候需要unlock函数的调用，并且最好把unlock方法放finally代码块中，这样出现异常时一定会释放。

总结：在资源竞争不激烈时，优先选synchronized，编译程序通常会尽可能优化synchrinize

在资源竞争激烈的时候synchronized性能下降几十倍而ReentrantLock还能维持常态。

注意：如果答得好，那么面试官还会问一下reentrantlock与AQS的关系？

乍一看AQS是个毛啊，全称是AbstreactQueuedSynchronize抽象队列同步器，是reentrantlock加锁和释放锁的实现！！！原来是这个意思啊。

具体实现：AQS对象有两个属性，一个是state默认是0，另一个是加锁线程默认是null，所以当一个线程调用lock的时候会先看state，CAS机制改变state的值，并把自己的thread值赋给加锁线程。所以AQS可以实现加锁、可重入锁。

1. 请讲一下什么是fail-fast机制，有什么手段可以避免吗？

首先说一下为什么需要用迭代器，两个原因：

①当需要遍历一个集合并向删除一些元素的时候，极有可能出现问题，比如[a,a,b,c,d]，想删除a，for(int i=0; i <size;i++){if.....list.remove(i)}当i=0时删除第一个a，但是删除之后list的索引变化了，但是i++了=1，所以就会把第二个a漏掉，解决办法三种，第一：remove之后i--;第二：从后往前遍历；第三使用迭代器

②迭代器我们知道是类似懒加载的原理，省内存啊。

但是使用迭代器的时候如果调用集合的remove就会跑出异常——fail-fast机制。

Fail-fast是java集合中的一个错误机制，当多个线程（单个线程也可能）对集合的内容操作时，就可能会产生failfast。比如当一个线程通过iterator去遍历集合时，该集合内容被其他线程修改后，那么原线程访问集合时就会抛出ConcurrentModifycationException，这就是failfast机制。

避免手段其一：使用iterator.remove方法替代collction.remove方法（适用于单线程），iterator的remove方法内部实现时帮我们做了类似i--的操作。每次调用next()方法的时候都会检查exceptionmodcount与modcount是否相等，不相等就代表list被人修改了就跑concurrentmodfication异常，但是调用iterator的remove方法时内部将next复制为lastref，lastref=-1，然后将modcount赋值给modcount，这样就达到了帮我管理索引不抛异常的目的。

避免手段其二：使用java并发包java.util.concurrent中的类替代ArrayList和HashMap，比如CopyOnWriteArrayList，原理是并不是在原集合中修改，而是新建一个集合，最后把指针指向新集合。使用替代HashMap，原理其实上面差不多，并不是当时改变数据而是new新的对象，最后改变指向即可。

1. 阐述volitile和synchronized或者Lock的异同点？

从语义上来讲synchronized和lock是一样的！具体区别看上面已经记录的问题。

在并发编程中一定要考虑缓存一致性问题。在介绍JMM的时候已经说过了为什么会出现JMM，是为了模仿各个操作系统的内存模型而实现的。在cpu的高速缓存和主存之间存在缓存屏障——缓存一致性协议。一致性需要三个特点：

①原子性，操作不可中断，不然别的线程读取的是错误的值，syn和lock可以解决

②可见性，修改后立即将变量刷新回主存，并且通知其他线程这个值变了，volatile和syn和lock都可以解决

③有序性，编译期间、cpu执行期间会进行代码重排序和指令重排序，java虽然已经保证重排序不会带来结果不一致的情况，但是多线程情况下还会出问题，syn和lock和volatile都可以解决，其实java自身有一定的先天的有序性称为——hapends-before，后面介绍。

总结：Volatile只能保证可见性和有序性，不保证原子性，下面一一介绍

Volatile保证可见性：简单说就是当A线程从主存read&load过来变量a进行assign修改后会立即store&write到主存中并告知其他线程的高速缓存该变量失效，需要重新从主存中读，这就保证了立即可见性。

Volatile保证有序性：指令重排涉及到被volatile修饰的变量的读写操作时是不能重排序的，也就是说在读写之前的执行必须执行完！

Volatile不保证原子性：这就是和syn和lock的区别所在，说白了就是在多线程同时操作变量时，由于不是原子性，那么会导致修改过程中当前进程中断其他进程读到的变量时假的。但是非常适用于只有一个线程修改变量的情况！！

1. 什么是hadoops-before原则？

这是jvm自带的先天性的有序性。之前说过jvm虽然保证单线程下的有序性，但是无法保证多线程下的有序性，hadoops-before在一定程度上减弱了一下并发问题。但是仍然存在不安全性。下面介绍一下它的8条（实际上就是4条）原则：

①程序顺序规则：一个线程中的保证代码的顺序执行，但是不是表面的意思而是保证单线程的有序性，jvm还是会重排序。

②监视器锁规则：unlock要发生在lock之前！

③volatile可见性规则：写操作的结果要发生在读操作之前。

④传递性，Ahadoops-beforeB,Bhapends-beforeC，那么A也要hadoops-beforeC.

1. CAS(Compare And Swap)无锁算法是什么，有什么作用？

非阻塞同步算法！！为什么叫做这个名字很重要！，下面解释一下：

非阻塞的意思我们知道就是当前操作不阻塞其他线程对当前变量的操作！

同步的意思我们也知道是数据一致性。

既然这样他俩按理说不能同时保证啊，很显然的例子就是上面提到的原子性操作，Volatile不保证原子性操作即非阻塞，会导致不同步，那么CAS到底是如何实现非阻塞情形下的数据同步的呢？？？

Jdk5出现的java.util.concurrent.atomic包下提供了原子操作类：对基本数据类型的加减操作进行了封装保证其操作原子性！！所以如果我们使用AtomicInteger inc = new AtomicInteger();而不是直接用int的话，就可以在非阻塞条件下保证数据的原子操作即保证数据的同步。

原理：CAS是乐观锁技术，当多个线程尝试修改atomic变量时只有一个线程能改变它的值，而其他线程都会失败，失败的线程并不是挂起！！如果挂起那就是阻塞了（类似syn、lock）。而是告知其他线程竞争失败请再次尝试。CAS有三个操作数：内存值A，旧的值B，新的值C，只有当内存值A和新值C相同时才算成功。

注意：jdk8有优化：像刚才所说的，当大量线程并发的时候很容易出现大量线程的空循环，也就是说大量时间都询问，并没有实际的操作。LongAdder这个类是jdk8新推出的一个类，它尝试使用分段CAS以及自动分段迁移的方式来大幅度提升多线程高并发执行CAS操作的性能。

具体实现：当出现上述大量空循环的时候，底层就会自动分出多个base值，也就是分布式的思想！！找第一个base发现不行那就找第二个....当需要获取值的大小时，就会把所有的base分段的值加起来。————并行的思想。

1. 类加载工作机制你了解如何？

简单说就是将类的.class文件中的二进制数据读入到内存中，方法运行时数据区的方法区中，然后在堆中创建一个对应的java.lang.Class对象，用来封装在方法去中的数据结构，这个对象向开发者提供访问方法区内数据结构的接口。

加载.class文件的方式有①从本地系统中直接加载②通过网络下载.class文件③从zipjar等归档文件中记载.class文件④从专有数据库中加载⑤将java源文件或者代码动态编译为.class加载



①加载：查找并加载类的二进制数据，虚拟机需要完成三件事：第一、通过一个类的全限定名获取其定义的二进制字节流，第二、将这个字节流所代表的的静态存储结构转化为方法区的运行时数据结构，第三、在java堆中声称一个代表这个类的Class对象

这一步是jvm提供的类加载器在jvm外部实现的，采用双亲委派模型，后面会介绍

②验证（连接的第一步）：确保被加载类的正确性，确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并不会危害虚拟机的安全，验证大致会完成四个阶段的验证：其一：文件格式验证，验证字节流是否符合Class文件格式规范；其二：元数据验证：堆字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范；其三：字节码验证，通过数据流和控制流分析，确保程序语义是合法的、符合逻辑的；其四：符号引用验证，确保解析动作能正确执行。

注意，验证阶段不是必须的。

③准备（连接第二步）：为类的静态变量分配内存，并将其初始化为默认值

准备阶段正式为类变量分配内存并设置类变量初始值的阶段，这些内存都讲在方法区中分配。其一：这时候内存分配仅仅包括类的静态变量不包括实例变量（普通属性），实例变量会在对象创建时随着对象一起分配到堆中。其二：这里所设置的初始值通常情况下是数据类型默认的零值，比如0、0L、null、false，而不是被在java代码中被显式赋予的值。其三：如果类字段的字段属性表中存在ConstanValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstaVlaue属性所指定的值。

③解析（连接的第三步）：把类中的符号引用转换为直接引用

解析阶段是虚拟机讲常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符七类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。直接引用就是直接指向目标的指针、相对偏移量或者一个间接定位到目标的句柄。

④初始化：为类的静态变量赋予正确的初始值，jvm负责对类进行初始化，主要对类变量进行初始化，在java中对类变量进行初始化方式：其一：声明类变量是指定初始值；其二：使用静态代码块为类变量指定初始值。

JVM初始化步骤：第一：假设这个类还没有被加载和连接，则程序先加载并连接该类；第二：驾驶该类的直接父类还没有被初始化，则先初始化其直接父类；第三：假设类中有初始化语句，则系统一次执行这些初始化语句。

类初始化时机：

其一：创建类的实例，new的时候

其二：访问某个类或者接口的静态变量或者对该静态变量赋值

其三：调用类的静态方法

其四：反射（如Class.forName）

其五：初始化某个类的自雷，则其父类也会被初始化

其六：java虚拟机启动时被表明为启动类的类如main函数所在的类

⑤使用⑥卸载

* 1. 既然提到了类加载机制，能说一说类加载器和双亲委派模型吗？

注意：上面有一个小错误，站在JVM角度来看启动类加载器Bootstrap ClassLoader是JVM内部的C++实现的，属于jvm一部分，而扩展类加载器Extension ClassLoader和应用程序类加载器Application ClassLoader和自定义的都是JVM外部java代码实现的。

①启动类加载器：负责将JAVA\_HOME/lib下的或者-Xbootclasspath参数指定的路径下并且可以被JVM识别的类加载到JVM内存中

②扩展类加载器，负责将JAVA\_HOME/lib/ext下的或者被java.ext.dirs系统变量指定的路径下的所有的类加载到jvm内存

③应用程序类加载器：由ClassLoad.getSystemClassLoader()方法返回对象获取这个类加载器负责将用于路径classpath下的类库加载到jvm中



该图展示的是双亲委派模型，除了启动加载器其他都有自己的父类加载器。

工作流程：

如果一个类加载器收到了类加载请求，它不会首先自己加载而是委派给父类加载，所以最终所有的都会传递到顶层，只有当父类无法加载时子类加载器才会自己去加载。

1. 一致性哈希是什么，有什么应用？与哈希有什么区别？

一致性哈希算法常用于负载均衡中要求资源尽可能均匀分布，并且可以快速定位数据所在节点上。比如MemCache集群（redis、cassandra等）要求数据均分、快速定位、增删节点不会有大震荡或者雪崩，再比如RPC过程中服务提供做N个节点的集群部署，为了能在服务商维护一些业务状态，希望同一种请求每次都落在同一台服务器上。

所以一致性哈希算法必须解决两个问题：

①节点增加或者减少不会让很多节点数据失效

②数据分布不均问题

一致性hash算法，不管有多少个实例，都是使用2^32作为模，并且将0和2^32-1首尾相连，结成一个环，这样所有使用一致性哈希算法的数据大概率均匀落在环上。加入物理节点，同样计算物理节点标识的hash值，同样在环上，数据会顺时针落在最近的节点上，这样就建立了数据与节点的映射关系。（当然还存在hash分布的问题），当有新的节点node4进入的时候如下图



它的加入只会影响node4和node4之前的node之间的数据失效，其他数据正常。而不会像一开始提到的直接用哈希取模导致大部分节点都失效。

但是！！！！到现在为止一致性哈希还不错，但是仍然没有解决上述的两个根本问题，比如环上的三个节点中的一个宕机了，所有的请求就会落在下一个node上造成压力过大最终导致集群崩溃。同时假如我们只有3个节点，用比较低级的取模hash，得到0 1 2哈希值，如果数据很多落在了大于2的话，那么0这个节点会被压死，所以现在一致性哈希还没有解决我们的问题。

解决办法：

采用更好的hash算法让节点的hash值尽量分散，但是并没有改变根本问题而且也没有改变雪崩问题。

最好的解决办法是采用————虚拟节点！！！

所谓虚拟节点就是一台物理机作为多个虚拟节点的意思。比如A B C三台节点，分为A1A2A3B1B2B3C1C2C3虚拟节点，使它们均匀交叉分布在环上，但是A1A2A3实际上就在一台服务器上，这样做的目的就是解决上述两个问题，比如当节点A宕机的时候，那么会将节点A上的数据分成1 2 3份分别散落在环上！！而不是直接丢给B节点，这样就达到了数据的均匀性和避免雪崩的目的。



1. 介绍一下Redis支持的数据结构，最好分别说一下他们用在何处。

Redis5.0之前有五种数据结构，5.0添加了stream数据类型。

String—字符串（key-value 类型）

Hash—字典(hashmap) Redis的哈希结构可以使你像在数据库中更新一个属性一样只修改某一项属性值

List—列表 实现消息队列

Set—集合 利用唯一性

Sorted Set—有序集合 可以进行排序 可以实现数据持久化

说stream之前要说一下5.0之前的pub/sub给予channel的一种消息发布和订阅，但是有一个极大的缺陷就是不保存历史消息以及持久化不强，缓存容易丢失。

那么list呢，也可以作为mq通道，虽然有持有化机制，但是极大的缺陷是一条消息只能被消费一次，虽然可以消费后再add到另一个list中，但是需要等待....另一个思路是一条消息被add到多个list中，但是这种方案还是有缺陷就是复制过程以及扩展性不强。

Stream的消费组借鉴了kafka的消费组，当然也是有区别的，基本消费流程是一样的，一个消费组内的消费者不能重复消费，但是stream根本还只是一个队列，并没有分区，而kafka有分区。同时stream支持阻塞和非阻塞消费模式。类似于tail -f tail -n。同时stream会记录消费者的消费状态，已消费、正消费信息，保证再次消费时的连续性。

注意stream是根据ID（时间戳+id）来标记message的，而kafka是用offset进行标记的。

1. 谈一谈java的自动装箱和拆箱（autoboxing和unboxing）

自动装箱和拆箱是针对java的八大基本数据类型而言：byte、short、int、float、long、double、boolean、charcter而言的。比如下面这句话

Integer total = 99;

Int totleprim = total;

实际上上述的代码系统替我们执行了

Integer total = Integer.valueOf(99);

Int totalprim = total.intValue();

Integer的源码如下

public static Integer valueOf(int i) {

return i >= 128 || i < -128 ? new Integer(i) : SMALL\_VALUES[i + 128];

}

如果i大于等128或者小于-128的话系统会给我们真的新创建一个Integer对象，如果在-128和127之间的话不会重新创建对象，系统早就已经创建好了一个对象数组等着被调用，而且是可以重用的！！！但是注意这个范围仅仅针对integer、short、byte、charcter、long而言的，double、fload由于-128到127是无限的所以不存在这种性质。对于boolean一共就俩值肯定早就创建好了。

针对装箱和拆箱问题需要着重注意一个问题就是==和.equal()两个东西

==是比较两个对象是不是一个对象。.equal()比较两方面一个是不是同一类型，在比较字面值是不是相同。

下面举一个不好理解的例子

1 Integer num1 = 100;

2 int num2 = 100;

3 Long num3 = 200l;

4 System.out.println(num1 + num2); //200

5 System.out.println(num3 == (num1 + num2)); //true

6 System.out.println(num3.equals(num1 + num2)); //false

按照刚才说的==是判断是不是同一个对象，最后一行调用了.equals方法，num3是Long类型，而num1+num2是Integer类型，即使字面值相同，返回的还是false，针对倒数第二行，即使我们认为是错误的，但是这触发了拆箱的问题，当一个基础数据类型与封装类进行==、+、-、\*、/运算时，会将封装类进行拆箱，对基础数据类型进行运算。 所以是true。

编程中一定要注意，设计拆箱时一定要判断是不是null，因为Integer a = null;是完全没问题的，但是一旦设计拆箱比较那么就会抛异常了！！

1. 谈谈你对java反射机制的理解，用在什么地方？

反射机制是将框架和类柔和在一起的调和剂。

在面向对象世界中所有都是对象，但是java并非纯净的面向对象，比如基本数据类型和static修饰不是对象而是类，当然了类也是对象可以认为是java.lang.Class的实例对象。

我们可以理解：任何一个类都是Class类的实例对象。所以才可以调用People.class或者p.getClass()来获取Class对象。

那么到底反射机制是什么？

说白了就是动态获取类的信息以及动态调用对象的方法的功能就称为java语言的反射机制。在运行状态中，对于任意一个类，能够获取这个类的所有属性和方法，对于任意一个对象能够调用它的任意一个方法和属性（包括私有方法和属性）。

获取字节码文件对象的三种方式。

①Class clazz1 = Class.forName("全限定类名");　　//通过Class类中的静态方法forName，直接获取到一个类的字节码文件对象，此时该类还是源文件阶段，并没有变为字节码文件。

②Class clazz2 = Person.class;　　　　//当类被加载成.class文件时，此时Person类变成了.class，在获取该字节码文件对象，也就是获取自己， 该类处于字节码阶段。

③Class clazz3 = p.getClass();　　　　//通过类的实例获取该类的字节码文件对象，该类处于创建对象阶段

反射机制作用：

①通过字节码对象（Class对象）创建实例对象

Class clz = Class。forName(....User);

User u= (User)clz.newInstance()

②获取指定构造器方法、所有构造器方法

③获取成员变量field

④获取方法并使用

⑤获取该类的所有接口

⑥做动态代理

关于动态代理，实际上就是在程序运行过程中动态通过cassloader加载真实对象的代理类，然后调用真实对象的某些方法会按照代理类那样执行。

1. 如何写一个不可变类？

不可变对象对于缓存是非常好的选择，因为你不需要担心它的值会被更改。不可变类的另外一个好处是它自身是线程安全的，你不需要考虑多线程环境下的线程安全问题。

要创建不可变类，要实现下面几个步骤：

①将类声明为final，所以它不能被继承

②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

③对变量不要提供setter方法

④将所有可变的成员声明为final，这样只能对它们赋值一次

⑤通过构造器初始化所有成员，进行深拷贝(deep copy)

⑥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

1. HashMap与HashTable与TreeMap与LinkedHashMap的区别以及HashMap的底层实现？

①HashMap是非线程安全的，HashTable是线程安全的，所以HashMap效率高些

②HashMap的键和值都允许null存在，而HashTable不行

③HashTable线程安全，效率过低，键和值都不允许为null

④LinkedHashMap不安全，继承自hashmap，底层增加一个双向链表，Entry中增加一个上一个指针和下一个指针，同时添加了一个成员变量head指针，这样就可以记录所有put的顺序（默认是put的顺序，可以是读取的顺序）

其实如果想线程安全，那么请用jdk5出现的concurrentHashMap吧，后续会介绍它是采用了分段锁技术，HashMap底层是数组+链表实现的，链表的每个节点是一个Entry键值对的数据结构，查找快，插入也快，每次插入新数据将会指向链表的首部。jdk8之后当链表长度大于8时将会用红黑树替代。

既然提到了hashmap和红黑树的问题，也要提一下treemap，TreeMap是基于红黑树实现的、是有序的KV组合，同样也是非安全的。

TreeMap适用于按自然顺序或者自定义顺序遍历key。

16.解决hash冲突的方法？

①开放定址法（线性探测再散列、二次探测再散列、伪随机探测再散列）

②再哈希法

③链地址法——HashMap

④建立一个公共溢出区

17.通过15的问题不可避免要聊一聊红黑树了？

红黑树是一种近似平衡二叉树的数据结构，在C++的标准库被中经常被用到，而平衡二叉树是为了解决二叉搜索数的极端情况，导致树的两支不平衡，这样查询效率会低很多，所以平衡二叉树严格追求左右子树高度差不能大于1，这样的话会保证查询的效率，但是每次插入和删除基本都会进行旋转平衡。此时红黑树就是一个折中，它并不是非常严格要求左右子树高度不大于1，只是要做黑节点平衡。

1. HashMap、ConcurrentHashMap？

HashMap不保证线程安全，在高并发修改时会出现死循环，HashTable虽保证线程安全，但是高并发情况写会阻塞读，所以效率很低。

ConcurrentHashMap采用锁分段的技术，一个容器有多个锁，这样就会环境锁竞争的问题。结构上是ConcurrentHashMap是一个Segments的数组，而每个segment就是一个小型的hashMap，HashMap则是Entry链表数组。

1. Vector和ArrayList和LinkedList的区别？

①ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

②Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。

③LinkedList是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了List接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。

1. 死锁的必要条件，有什么好的方法解决死锁吗？

①互斥，至少有一个资源处于非共享状态

②占有并等待

③非抢占

④循环等待

解决死锁：

①上述四点不能同时成立

②使用银行家算法，如果该进程请求的资源操作系统剩余量可以满足，那么久分配

关于什么是银行家算法？

其实就是一个资源安全分配的问题，在接受一个请求的时候要考虑是否安全，那么具体是如何做的？

①遍历当前请求，如果剩余资源不允许当前请求执行则跳过，否则则执行并且把当前请求所占用的资源作为可回收的资源

②继续遍历剩下的，直到产生一个安全执行序列

1. 进程间通信的方式？

①管道( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

②有名管道 (named pipe) ： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

③信号量( semophore ) ： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

④消息队列( message queue ) ： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

⑤信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

⑥共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

⑦套接字( socket ) ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

1. 请讲一讲进程与线程的区别和联系？

①进程是资源分配的最小单位，线程是程序执行、资源调度的最小单位

②没有线程的进程就是一个单线程

②进程有自己独立的地址空间，每启动一个进程系统会为它分配地址空间、建立数据表来维护代码段、堆栈段和数据段。而线程是共享进程中的数据的，使用相同的地址空间，线程切换花费开销远远小于进程间的cpu切换。

③线程间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以IPC进行。但是存在多线程数据同步的问题。

④多进程程序更加健壮，但是线程如果有一个死掉，那么该进程也死掉了。

1. 说一说java实现多线程的方式？

①继承Thead类

②实现Runable接口

③实现Callable接口（有返回值）

1. Java的四种引用：强弱软虚以及应用场景？

①强引用：如果一个对象具有强引用，它就不会被垃圾回收器回收。即使当前内存空间不足，JVM也不会回收它，而是抛出 OutOfMemoryError 错误，使程序异常终止。如果想中断强引用和某个对象之间的关联，可以显式地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象。

②软引用：在使用软引用时，如果内存的空间足够，软引用就能继续被使用，而不会被垃圾回收器回收，只有在内存不足时，软引用才会被垃圾回收器回收。所以可以做高速缓存如网页图片

③弱引用：具有弱引用的对象拥有的生命周期更短暂。因为当 JVM 进行垃圾回收，一旦发现弱引用对象，无论当前内存空间是否充足，都会将弱引用回收。不过由于垃圾回收器是一个优先级较低的线程，所以并不一定能迅速发现弱引用对象。

④虚引用：顾名思义，就是形同虚设，如果一个对象仅持有虚引用，那么它相当于没有引用，在任何时候都可能被垃圾回收器回收。

总结：可以用软引用和弱引用避免OOM的情况，如果网页中大量的图片用hashmap来保存，key是路径，value是图片的软引用，典型的是weakHashMap

1. hashCode与equals有什么区别和联系？

已经知道hashcode是计算当前对象的hash值，.equals是判断对象内容是否相同，同时也知道重写任何一个方法必须要重写另一个方法。接下来就仔细整理一下，不然总是模模糊糊。

说他俩的关系之前说一下equals和==的区别吧：

Object基类中有.equals()函数，默认是this == obj的实现，所以任何一个类如果不重写.equals方法的话equals和==是一模一样的！！！！！都是判断是不是同一个对象，但是大多情况下类都重写了这个equals方法用来判断两个对象内容是不是相同而不是是不是同一个对象地址。

现在说一说hashCode函数：

它是获取哈希码，或者叫做散列码，同样是Object基类中的方法返回一个int值，但是只有当类要存入哈希表数据结构的时候才会起作用如HashSet和HashMap。

现在我们来看看hahCode和equals的关系吧：

如果跟散列表如hashset、hashmap、hashtable没有关系，那么他俩也没有半毛钱关系

如果涉及散列表此时如果两个对象相等，那么hashcode一定相同，如果两个对象的hashcode相同，他们不一定相等。那么此时想要进入散列表，那么必须要知道key不能重复，此时通过两个阶段，先进行hashcode判断，如果相同再通过equals判断，如果返回false则证明可以插入。否则证明是相等的对象。

然而有一个问题就是equals方法我们肯定要进行重写，因为这个自定义类得有个比较的函数，如果不重写hashcode与equals对应的话，极有可能出现equals相等，但是hashcode不同，这样就造成了一个冲突问题。

这就是为什么ahshcode和equals要同时重写的原因。

1. Override和Overload的区别你说一说？

Override是重写的意思，子类继承父类的时候子类中可以定义某个父类的方法，当子类调用该函数的时候会覆盖父类的函数，该方法的函数名、参数都一样。

Overload是重新加载的意思，表现为类的多态性，函数名相同，但是参数个数、顺序、参数类型不同这就是重载。

可以从以下两点区别：

①范围区别：重载是同一个类中，重写不是同一个类中

②参数区别：重载的参数列表一定不同，重写一定相同

1. 抽象类和接口的区别？

①一个类只能继承单个类，但是可以实现多个接口

②抽象类中可以有构造方法，接口中不能有构造方法

③抽象类中的所有方法并不一定要是抽象的，你可以选择在抽象类中实现一些基本的方法。而接口要求所有的方法都必须是抽象的

④抽象类中可以包含静态方法，接口中不可以

⑤抽象类中可以有普通成员变量，接口中不可以

1. 解析XML的几种方式的原理和特点？

DOM、SAX、PULL

①DOM：消耗没存，先把xml加载到内存中，然后用DOM API访问树结构

②SAX：解析效率高，占用内存少，基于事件驱动的：更加简单的说就是对文档进行顺序扫描，当扫描到document的开始与结束、元素element的开始和结束时通知事件处理函数，由事件处理函数做相应的动作然后继续同样的扫描知道文档结束。

③PULL：与SAX类似也是基于事件驱动，可以调用next方法来获取下一个解析事件（就是文档的开始与结束、元素的开始与结束），当处于某个元素时，可以调用XmlPullParser的getAttributte方法还获取属性的值，也可以调用nextText获取本节点的值。

1. java中sleep、wait、yield、join的用法和区别？

线程的五种状态：创建、就绪、运行、阻塞、死亡

阻塞分为三种：

等待阻塞：运行的线程调用wait()方法，该线程释放占用的资源，JVM会把该线程放入等待池中，之后不能自动唤醒，必须依靠其他线程调用notifly或者notifyall方法才能回唤醒

同步阻塞：运行的线程在获取对象的锁时，若该同步锁被占用，则JVM会把该线程方所锁池中

其他阻塞：运行的线程执行sleep和join时或者发出IO请求时，JVM会把该线程设置为阻塞状态，当sleep查实、join等待线程终止后者超时或者IO完毕时线程重新进入就绪状态

①sleep：指定时间内让当前线程暂停执行，但是不释放锁

②wait：在其他线程调用notify和notifyall之前出于等待状态，会释放所有锁

③yield：暂停当前正在执行的线程。注意这个操作并不是让当前线程进入锁池或者等待池，而是直接进入可运行状态，随时可以被执行

④join：等待该线程终止后再运行当前线程

1. Java多态实现的原理？

这个问题其实说简单也简单，但是从底层看还是有点意思的。

抽象描述：多态的意思就是函数调用可以根据调用对象的不同而采用不同的行为方式！

实现原理：动态绑定

程序调用的方法在运行期才动态绑定，追溯源码可以发现JVM通过参数的自动转型来找到合适的方法。它允许基类的指针或者引用指向派生类的对象，而在具体访问时实现方法的动态绑定。

Java对于方法的动态绑定的实现主要依赖于方法表，但通过引用调用（invokevitual）和接口引用调用（invokeinterface）的实现则有所不同。

多态类型：

①编译时多态（又称静态多态）

②运行时多态（又称动态多态）

重载就是一个静态多态的例子，在编译的时候就已经确定调用哪个函数了。

我们通常所说的多态指的是运行时多态，多态通常有两种实现方法：

①子类继承父类extends

②类实现接口implements

然而不管通过继承还是接口实现动态，JVM在执行动态调用的时候都是通过动态绑定的机制来实现的。

那么JVM底层到底是如何实现动态绑定函数的调用的呢？

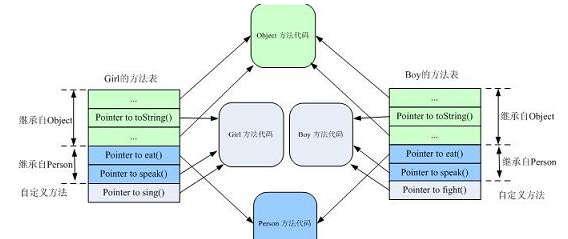
JVM执行java字节码的时候，类型信息被存放在方法区中，通常为了优化对象调用方法的速度，方法区的类型信息中增加一个指针，指向一张记录该类方法入口的表————方法表，表中的每一项都是指向相应方法的指针。

invokevirtual的实现：

由于java是单继承，而且所有的类都是Object的派生类，所以方法表中最先存放的是Object类的方法，接下来是该类的父类的方法，最后是该类本身的方法。这里的关键点在于如果子类重写了父类的方法，那么子类和父类的那些同名方法共享一个方法表项，都被认为是父类的方法。

（注意：只有非私有的方法、非静态方法才可以出现在方法表中，更加深入的说，静态方法是由JVM指令invokestatic调用的，私有方法和构造函数是由invokespecial指令调用，只有被invokevirtual和invokeinterface指令调用的方法才会出现在方法表中）

通过上述描述我们知道函数在方法表中的偏移量是固定的！！！



当实例类调用一个方法的时候，比如得到的是偏移量为3的方法，去方法表中找3，看一看如果重写了就直接调用，如果没有重写那就拿着这个3去父类去找即可。

invokeinteface的实现：

由于多实现的情况导致方法的偏移量不可用，只能遍历方法表，所以这种情况下要比继承的多态调用慢。

1. 运行java程序时的jmx参数指定你用过吗，如果用过你举个例子吧？

export OPT="-Xmx${LOG\_CONSUMER\_HEAPSIZE}m -Xms${LOG\_CONSUMER\_HEAPSIZE}m -Xmn3g -XX:PermSize=256M -XX:MaxPermSize=512m -Xss256k -XX:+UseParallelGC -XX:ParallelGCThreads=20 "

上述是师傅写的程序运行时的参数，下面会详细介绍各个参数的意义，以后还是应该养成设置执行参数的习惯！

-Xms：初始堆大小，避免垃圾回收之后JVM调整一般和-XMX一致

-Xmx：最大堆大小

-Xmn：年轻代大小，整个堆大小=年轻代+年老带+持久代。持久代一般64M，所以年轻代和年老代就是相对的，官方给出年轻代与年老代的比例是3:5。上面师傅的配置就是这个比例

-Xss：设置每个线程的栈的大小jdk5以后默认是1M，以前是256kb。如果是单线程任务，就不用减小这个值，甚至可以加一些。多线程的话可以降低一点。

-XX:PermSize=：持久代初始化大小

-XX:MaxPermSize=：持久代最大大小，设置的挺好的，适当加载持久代大小很合适，但是具体程序具体分析...怎么分析？可以打印gc信息，后面说

-XX:+UseSerialGC：设置串行年轻代收集器

-XX:+UseParallelGC：设置并行年轻代收集器

-XX:+UseParallelOldGC：设置并行年老代收集器

-XX:+UseConcMarkSweepGC：设置并发收集器，注意jdk1.8默认开启G1收集器

-XX:+PrintGC：输出GC日志

-XX:+PrintGCDETails：输出GC的详细日志

-XX:+PrintGCTimeStamps：输出GC的时间戳，标准格式

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:filename日志文件，绝对路径

-XX:ParallelGCThreads：设置并行（或并发）收集器条件下使用的线程数，最好不要超过cpu逻辑核数

-XX:MaxGCPauseMillis：这只并行收集最大暂停时间

-XX:GCTimeRatio：设置垃圾回收时间与程序运行时间的比