首先JVM面试一般逃不过：GC算法，类加载机制，调优（堆内存设置，算法GC算法配置）

我们同时站在面试者与被面试者去阅读深入java虚拟机，去思考自己会去考察什么，通过搜罗面试题以及有经验的大神们比较关注什么

第一章：

相当于引言，如果是我我是不会浪费时间在这个上面

第二章：Java内存区域与内存溢出异常

1作者以运行时数据区域来对标题呼应，java内存区域和运行时数据区域是一个意思么？（方法区、本地方法栈，程序计数器，堆，栈+直接内存）

2.程序计数器为什么没有规范OOM？？OOM是在什么情况下发生的？或者问程序计数器在什么情况下会抛出OOM异常？（好像有点误导的感觉）

在程序运行时内存空间达不到需求时就会抛出OOM，程序方法所需要的栈深度超过虚拟机分配的深度后会抛出Stack Overflow，一般虚拟机都会支持栈的动态扩展，一旦扩展不了的时候就会抛出OOM

3.java规范对虚拟机栈规定了两种内存异常？他们在什么情况下发生?

两种基本在第2问讲过

4.堆主要存放的是什么数据？（很多人可以答出所有的对象的实例，但是会漏掉数组不是原始类型的数组）

**5.JIT编译器有没有？逃逸分析技术呢？栈上分配，标量替换？所有的对象一定都在堆上么？**

6为什么要优化呢？在堆上不好么？堆的空间那么大，栈那么小？

堆上内存大事固然不错的，但是无法自动回收，需要依赖GC，GC在大多数时候很容易引起性能问题，但放在栈里就不用考虑垃圾回收的问题，但是您也说了，栈空间很小，在需要栈上分配的对象必须谨慎，否则也逃不过stack或者OOM

7**栈上分配技术有没有了解过**

8**标量替换有没有了解过**

9你用过哪些虚拟机（HotSpot VM、J9 VM、Zing VM，BEA Jrockit）

10方法区主要放置什么（那些不怎么变的，比如类信息，常量，jit编译的代码（马长累）），常被人成为permanent，注意姿势大家这么说，并非官方说法。因为GC有的时候还是会拜访这个区域的，只是一般都很难访问，甚至有的虚拟机都没有！

11除了刚刚说的这些内存区域，还有没有其他的？（直接内存，）

12直接内存是一个容易被忽略的boy，一般虚拟机配置内存的时候都不会将其包含进去，然而其确实又占内存，很容易出现个个内存区域总和大于物理内存限制，导致OOM

13.直接内存一般用来（NIO）

14句柄和指针（句柄的本质仍然是指针，但是指针的权限因为太大，所以就进行了封装，就像身份证证和学生证的区别吧）

15引用是用什么实现的（或者说有那两种实现方式）：使用句柄的好处在于，当对象被移动的时候，引用本身不会修改，缺点就是慢，使用指针的方式就反着看。

**16有没有遇到OOM或Stack Overflow的的情况，考察排查这两种问题的方法**

## 垃圾回收专题

**判活算法：引用计数算法和跟搜索算法**

引用计数算法是当每当对象被引用是就会将其引用计数加1，判断对象dead的方法自然也就是看引用计数是否为零，跟搜索算法可以理解为一个应用链专业一点就是图，对象到GC root总是可以找到路可走，这种判活的方法不仅可以判断某一个，还可以判断多个相互引用但无法追踪到gc root的多个对象。关于这两种算法，java选择的后者。至于为什么，其实我认为不需要过于纠结，两种算法各有利弊，而且有些编程语言也会选择引用计数法。不同取舍罢了。而且真正落实到垃圾回收层面，参考不仅仅是生死问题。

Q：GC root是什么

A:总结为4种，虚拟机栈中引用的对象（本地变量），方法区中的类静态属性引用，方法区的常量引用的对象，本地方法栈中引用的对象（为毛是这些？？？ gc root使用原则就是活跃的引用，对于java程序来说能够活跃一般都是静态变量，常量，方法使用的变量（java方法，本地方法））

Q:两次标记

A:jvm其实还是比较仁慈的，通过跟搜索算法找到的不可达对象并不是立即处以死刑，而是先标记一下，在标记的时候，会筛选一下，赛选条件就是对象没有覆盖finalize方法或者finalize方法已经被执行过，而且就算判断为有必要执行该方法，也是放到F-queue队列中，由一条低优先级的线程去执行销毁行为，也就是说这个线程也有可能不执行，这个时候如果想要重新激活对象还是有可能的，再往后就会进行第二次标记，在第二次标记前如果没有重新激活，可能就在劫难逃。

Q:四种引用类型

A:强，软，弱，虚 ==（亲情，爱情，友情，一面之缘）SSWP

只要强在，就不会回收，软引用会在内存不够用时第二次回收进行回收，弱引用和虚引用基本上只要进行gc就会被kill掉，虚引用连对象都获取不到。似乎可以通过改变引用来认为控制对象的生存能力，也算为通过编码的方式控制对象是否可以被回收提供的方法。

Q:方法区的回收

A：方法区虽说是持久代，但不代表不回收，只是说相对长寿一些。一般回收的是不用的常量和无用的类。

Q:什么是废弃常量还是？什么又是无用的类？

A:废弃常量是判断还有没有其它地方引用这个常量。而无用的类需要满足三个条件：

1. 该类的所有的实例都已经被回收，也就是java堆里不存在该类的任何实例
2. 加载该类的classloader已经被回收
3. 该类对应的java.lang.class对象没有任何地方被引用，无法在任何地方通过反射访问该类的

Q：为什么要回收无用的类

A：主要是解决那些使用了大量反射，动态代理等框架，还有向jsp这样自定义classloader的场景

Q：垃圾回收算法

A:标记清楚，标记整理，复制。总结一下垃圾回收算法，标记清楚的算法在之前确实提到过，即两次标记，排除效率的考虑，因为算法不是万能的，在有些情况下也许很好，在某些情况下不一定了，就像排序算法。标记清除算法比较明显的就是内存碎片，带来的直接问题就是，如果继续创建对象，但是因为碎片太多导致无法创建，意外的触发GC,因为碎片的问题，继而我们提出标记整理算法，这个算法的改进就是标记之后，将标记和为标价的整理到一起，这里可以画图说明，收集的时候收集边界。复制算法的思想是将内存分块，比如把内存分为一半，一半放新创建的对象，另一半放回收后仍然存活的对象，这种算法不好就是如果效率很好，那一半的空间似乎有点显得浪费了。所以真正实现的时候不是一半，存在一定比例，就像现在新生代和老年代的意思。我在深入java虚拟机那本书上看到作者写了第四种算法，其实与其说第四种算法不如说是一种策略算法，倒不如说根据情况对以上三种算法进行策略性运用。比如老年代存活时间长，针对老年代内存区不采用复制算法而是采用清楚或整理。因为其不需要担保。

Q:垃圾回收器？

Q:判活算法？垃圾回收算法？垃圾回收器？傻傻分不清楚

A:算法是算法，收集器是收集器，收集器是算法的实现，而且具体的实现并不是只用一种算法。而且要注意哦，java自带的很有几种收集器，只是目前应用场景没用到。

<https://blog.csdn.net/tjiyu/article/details/53983650> 这篇文章总结的还可以。对于一次垃圾回收方案的确立一般不会仅仅依赖一种算法，比较年轻代和老年代内存分配也不一样，一劳永逸不存在的。除此之外，还要考虑并行和串行方案。而且随着JVM版本的不断迭代，也会有新的垃圾回收器出现。因为自己没有进行真正的调优，因为之前公司的业务并不复杂：

说一下调优的思路好了：

<http://www.importnew.com/22336.html>

第一步：性能监控，Jconsole，日志等等，看看有没有问题发生

第二步：性能分析，通过日志等信息，凭借经验缩小问题范围，用工具去验证

第三步：调参数，配置，优化代码

Q:我们一直在说的都是内存的回收，不如也来谈谈内存的分配

A:对于新创建的对象我们都知道是放在堆上？一定么？不一定，就像之前讲过，有栈上分配的能力，标量替换。OK，知道就好，那么我们就讨论一下堆上分配情况好了。堆上是怎么划分内存空间的，有年轻代和老年代，年轻代又分为eden和survivor。新创建的对象分配在哪里？一般情况下会自然分配在新生代，但是有一种情况，就是创建了需要连续内存的大对象，而新生代放不下，是有可能直接放进老年代的？为什么不进行gc再放呢？这么说吧，正常情况下确实应该先GC，但是新生代频繁GC并不好，所以有配置参数配置，多大的对象直接放入老年代，减少年轻代中GC导致的频繁内存复制。还有什么情况，对象会进入老年代，哈哈，当然是老了么，他里面有一个年龄计数，达到一定年龄就会进入老年代，一般每熬过移除minor gc就会长大一岁，长到15岁就会进入老年代。除了年龄判断，虚拟机也并不总是要求对象长大，比如如果在survivor空间中相同年龄所有对象大小的总和大于survivor空间的一半也会进入老年代。

讲了这么多年轻代，我们来谈谈空间分配担保，如果进入老年代，发现老年代不够，会进行full gc，小于的话，要看是否允许担保失败，允许失败的话，会进行minor gc，在尝试放在新生代，不允许失败还是会进行full gc.

## 虚拟机加载子系统

Q：类初始化的时机

A：概括来说，是在主动引用的时候，如果

Q：类加载过程

A：家宴尊姐携四畜，除了解析阶段，其余都是按照顺序开始的，但是并不是串行，比如加载过程结束后才会执行验证流程

Q：他们是串行的还是并行的

A：上面已经回答了这个问题

Q：加载阶段

A：加载阶段是将二进制字节流加载进虚拟机的过程，需要注意一下几点：

1. 字节流的来源可以是zip，jar，war，动态生成等等
2. 这里涉及到两个内存区域，一个是方法区，存放类的信息，一个是堆，存放该类的class对象

Q：主动引用和被动引用

A：只有主动引用才会引发类的初始化，当你使用静态字段的时候，只有直接定义这个字段的类才会被初始化。四种必须立即对类初始化的情况（区别final的情况，final在编译器已经把结果放入常量池）：

1. new一个对象，读写一个静态变量
2. 通过反射调用的时候
3. 当初始化一个类发现其父类还没初始化话，则要先初始化其父类
4. 当虚拟机启动时。用户需要指定一个主类，包含main方法的那个类，虚拟机会先初始化这个主类

Q:验证阶段

A：一些检查工作，保证class文件确确实实是对的，能够确保解析动作能正常执行。一般情况下都是没有问题，除非这个class文件采用了什么特殊的方式生成。而且这里面的校验工作也许可以关闭，减少校验的时间。

Q：准备阶段

A：为类变量分配内存空间，注意是类变量。一般是静态变量，比如static int a = 123，但这个时候会a = 0；在初始化阶段在为其赋值，但是常量不同常量这个时候就会进行赋值

Q：初始化

A:初始化阶段才是真正的执行java代码，一般来说是那些类变量赋值操作和静态代码块里的语句，这里要区分两个方法clinit和init，顾名思义一个是类初始化函数，一个是实例初始化函数。**静态代码块只能访问到定义在静态语句块之前的变量，定义在后面的变量只能赋值不能访问**，clinit方法不需要显示调用父类的该方法，虚拟机会给予保证，更更更更关键的一点就是clinit并不是必须的，当，当一个类里没有静态块或者也没有对类变量的赋值操作，那么编译器可以不为这个类生成clinit方法。虚拟机会保证一个类的clinit方法在多线程环境中被正确的加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的clinit方法，其它线程都需要阻塞等待，直到活动线程执行clinit方法完毕。

Q：类可以被初始化几次

Q：局部变量和字段会受final影响么

A：并不会，局部变量与字段是有区别的，他在常量池里没有constant\_field\_info的符号引用，自然就没有访问标志，甚至连名称都不会保留下来，，自然在class文件中不可能知道一个局部变量是不是被声明为final

Q：并发有哪些特性？

A：有序性，可见性，原子性，除了synchronized，很多都无法同时保证三者，比如volatile只能保证可见性和有序性不能保证原子性。原子类可以保证原子性，但是不能保证可见性和有序性

Q:线程安全有几种类型？

A:有五类，不可变，绝对的线程安全，相对的线程安全，线程兼容，线程对立。不可变对于原始值来说就是其本身的值，对于对象来说，就是其内部的属性变量，在这里可以提出有状态和无状态。无状态的变量肯定最好的啦。

Q：vector线程安全经典问题

A:vector基于for的迭代过程恰恰印证了vector是一个相对安全的类，在迭代场景下仍然需要同步。

Q：线程兼容

A：可以外部使用同步手段使得java类变成一个线程安全的类，几乎所有的类在适当的同步下都可以成为线程安全的

Q：线程对立

A：线程对立是一种极端，在于无论采取何种同步措施都无法在多线程环境中使用。当然这种操作还是很少的，举两个例子：Thread的suspend方法和resume方法（这两个已经弃用），System.setIn,System.setOut

Q:如何理解同步和互斥

A：互斥是因，同步是果，互斥是方法，同步是目的

Q：互斥的方法有哪些？

A：临界区，互斥量，信号量都是实现互斥的手段

Q：同步语法和同步类

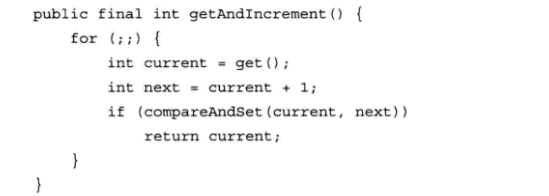
A：一个是基于语法的，一个是基于api，api提供的功能要多于语法

Q：阻塞同步与非阻塞同步

A：互斥同步主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也被称为阻塞同步。属于一种悲观的并发策略。，无论共享数据是否真的会出现竞争，都要进行加锁。随着硬件指令集的发展，我们有了另一个选择:基于冲突检查乐观并发策略

乐观就是：先进行操作，如果发现没有其他线程竞争就操作成功，如果有其他征用，在进行其他补偿措施。最常见的补偿措施就是不断的重试，知道成功为止。因为其不需要把线程挂起，因此也叫非阻塞同步。CPU层面支持的原子操作：

测试并设置（TAS），获取并增加（GAA），交换（SWAP），比较并交换（CAS），加载链接/条件存储（LL/SC），后面两者在最早的时候是没有滴,取钱java从api层级支持cas操作，并在原子类中得以运用，我们可以看下原子整形的加1操作：



该方法首先是一个循环体，这就印证了上面所说的非阻塞下的乐观操作，以不断重试作为补偿措施。然后就是cas操作，一旦cas操作成功就return，否则就不断重试。

Q：高效并发的理解

A：**高效并发主要把握两点：第一线程安全，第二性能稳定**（注意我说的稳定，什么意思呢，最极端的例子就是请求可以丢掉，但是服务器不能蹦，接收到的请求必须正确的执行，至少是结果正确）

Q：解决线程安全大概有哪几种方案

A：互斥同步，非阻塞同步，无同步方案

Q:什么是无同步方案

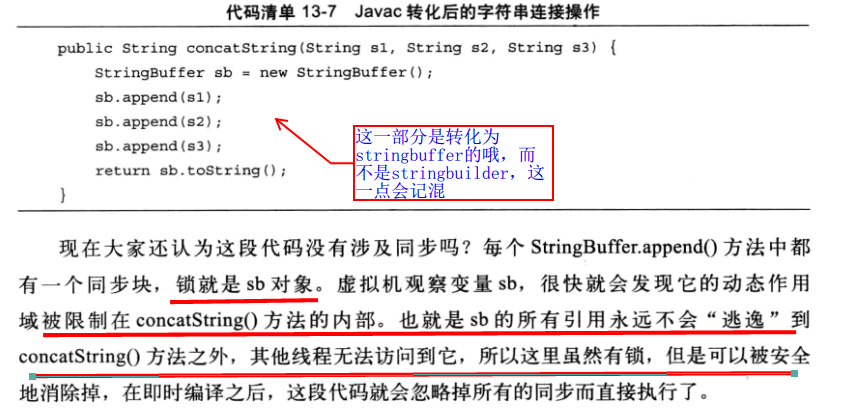
A:无同步方案主要有两种：可重入代码和本地线程存储

》》》可重入代码，这一点函数式编程：可重入性代码不依赖堆上的数据和公用的系统资源（在面向对象中可能表现为实例方法调用实例属性），用到的状态量都由参数中传入，不调用可重入的方法。我们可以通过一个简单的判断方式是：如果一个方法，他的返回结果是可以预测的，只要输入了相同的数据，就能返回相同的结果，那么他就满足可重入性要求，这一点简直就是在描述函数式编程。这也就是为什么，函数式编程具有天生的线程安全性。

》》》线程本地存储：数据竞争很多时候都是因为多个线程竞争一个变量，可是如果变量本身就是线程本身的呢？自然就不存在线程安全的问题。对于primitive value很好做到，可是复杂类型怎么办，java有ThreadLocal，而且显示生活中也有基于这种方案的衍生方案，比如生产者-消费者，宏观上看就是在一个流程里。

Q：String对象加号锁优化

A：String对象拼接会在编译后变成StringBuffer，注意不是stringbuilder，还有为什么要加锁，存在同时读写的场景。避免出现字符串重复append其实如果不是在多线程里是不需要的这样，反而还增加了锁的开销。不如手动写成StringBuilder去拼接字符串，其实jvm发现其根本不会逃逸即sb对象不会被共享出去，加锁的意义几乎没意义，即时编译后直接执行。



**这段代码到底发生什么。（确实要多看指定书籍，不然考什么你都不知道）**

**Q:轻量级锁和偏向锁**

**A：注意注意，这是在加了同步但不会放生竞争的情况下的一种处理机制，这种处理来源于经验，一旦存在锁竞争，轻量级锁和偏向锁都会失去其本身意义，而且还会降低性能。**

**Q: 轻量级锁和偏向锁是如何发挥作用的，是虚拟机自身么？**

**A：虚拟机自身实现，而且存在锁升级偏向锁>>>轻量级>>>重量级**

**Q：**synchronized和Lock

A: synchronized给出的答案是在软件层面依赖JVM，而Lock给出的方案是在硬件层面依赖特殊的CPU指令,native method

[**https://www.cnblogs.com/charlesblc/p/5994162.html**](https://www.cnblogs.com/charlesblc/p/5994162.html)