1、类加载的方式
（https://blog.csdn.net/qq_29064815/article/details/104161807）

2、HotSpot VM是目前使用最广泛的java虚拟机，使用了热点代码探测技术。

2、内存溢出【 out of memory】：是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。内存泄漏堆积最终会造成内存溢出。

2、内存泄露 【memory leak】：是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

3、目前的Java虚拟机支持Client和Server两种运行模式。使用参数-client可以指定使用Client模式，使用参数-server可以指定使用Server模式。默认情况下，虚拟机会根据当前计算机系统环境自动选择运行模式。使用java -version参数可以查看当前的模式，64位机器一般都是server模式：

```
C:\Users\Lujiaquan>java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

3、gc参数前面的加减号 + -：加好代表开启，减号代表关闭的意思。

3、VM Server模式与client模式启动，最主要的差别在于：-Server模式启动时，速度较慢，但是一旦运行起来后，性能将会有很大的提升。

3、直接内存：一般在进行网络编程的时候使用。因为网卡接收到数据是在系统态（内核态），Jvm是在用户态，内核空间到用户空间需要进行一次拷贝，用户态和内核态切换比较耗时、费资源（内存、cpu）。使用直接内存以后网卡接收到的数据放在直接内存，避免了内核空间到用户空间的拷贝。

4、jdk8中方法区被元空间（Metaspace）替代了。

4、堆内存和方法区（永久代）的数据是被所有线程共享的。所以会有并发安全问题，为什么会出现并发安全需要了解JMM内存模型就明白了。程序计数器和虚拟机栈是线程私有的，即每个线程都有自己的程序计数器和栈。

5、栈和栈帧的区别：

- 每个线程都有一个栈，栈里面存放着各种基本数据类型和对象的引用。
- 每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程。

6、栈上分配：虚拟机提供的一种优化技术。基本思想是，对于线程私有的对象，将它打散分配在栈上，而不分配在堆上。过程是JVM通过逃逸分析确定该对象是否会被外部访问，不会则通过标量替换将该对象分解在栈上分配内存，这样该对象所占用的内存空间就可以随栈帧出栈而销毁，好处是对象跟着方法调用自行销毁，不需要进行垃圾回收，可以提高性能（https://www.jianshu.com/p/580f17760f6e）：

- 逃逸分析：逃逸分析是编译语言中的一种优化分析，而不是一种优化的手段。通过对象的作用范围的分析，为其他优化手段提供分析数据从而进行优化。包括全局变量赋值逃逸、方法返回值逃逸、实例引用发生逃逸、线程逃逸。JDK8默认开启。
  Tips：任何在线程之间共享的对象一定属于逃逸对象。
- 标量替换：通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间。

7、TLAB（Thread Local Allocation Buffer）：TLAB的目的是在为新对象分配内存空间时，让每个Java应用线程能在使用自己专属的分配指针来分配空间，减少同步开销。如果设置了虚拟机参数 -XX:+UseTLAB，在线程初始化时，同时也会申请一块指定大小的内存，只给当前线程使用，这样每个线程都单独拥有一个Buffer，如果需要分配内存，就在自己的Buffer上分配，这样就不存在竞争的情况，可以大大提升分配效率，当Buffer容量不够的时候，再重新从Eden区域申请一块继续使用。TLAB只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已。当一个TLAB用满（分配指针top撞上分配极限end了），就新申请一个TLAB。（https://www.jianshu.com/p/8be816cbb5ed）。

8、虚拟机提供了两种给新生对象分配内存的方式，选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定，因此在使用

Serial、ParNew等待Compact过程的收集器时，系统采用的分配算法是指针碰撞，而使用CMS这种基于Mark-Sweep算法的收集器时，通常采用的空闲列表：

- **指针碰撞：如果Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离。**
- **空闲列表：如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录。**

9、并发情况下为新对象分配内存不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况，解决方案有两种：

- **虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。**
- **开启TLAB，即设置启动参数-XX:+UseTLAB。**

10、新对象内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值(如int值为0，boolean值为false等等)。

11、什么是对象实例数据和对象类型数据：

- **对象实例数据（堆）:对象中各个实例字段的数据。**
- **对象类型数据（方法区）：对象的类型、父类、实现的接口、方法等。**
- **静态区（也在方法区中）用来存放静态变量，静态块。**

12、主流的对象访问定位有句柄定位和直接指针两种
(https://zhuanlan.zhihu.com/p/88411800)

13、java中的方法分java方法和本地方法两种：

- **java方法：是由JAVA编写的，编译成字节码，存储在class文件中。**
- **本地方法：被native关键字修饰的方法，由其他语言（如C、C++ 或其他汇编语言）编写。**

13、java虚拟机栈和本地方法栈的区别
（https://www.cnblogs.com/manayi/p/9293302.html）

14、栈溢出不止和方法调用深度有关系，同时和栈帧的大小也有关系。栈帧的大小取决于方法参数的多少以及局部变量定义的多少。

15、jdk8中GC就是将堆和永久代进行垃圾回收。

16、判断对象是否存活两种算法：1、引用计数法；2、可达性分析。java中采用了可达性分析算法。

17、引用计数法：优点是快，方便，实现简单。缺点是当对象相互引用形成闭环时，很难判断对象是否应该回收。

18、可达性分析算法（重点）：算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），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达的。作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象，即目前正在被调用的方法中引用的对象。
- 方法区中类静态属性引用的对象（例如：public static User user = new User(); ）。
- 方法区中常量引用的对象。（例如：public static final User user = new User(); ）
- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

18、可达性分析算法是从GC Roots出发，然后去判断堆中的那些对象被GC Roots中的对象引用了，引用了就说明可达，没引用就说明不可达，不可达的对象都可以回收。即从GC Roots出发到达不了的对象都是可以回收的。

19、java中的引用分强引用【StrongReference】、软引用【SoftReference】、弱引用【WeakReference】、虚引用【PhantomReference】，垃圾回收时GC Roots不可达

的对象无论是那种引用都会被回收，而GC Roots可达的对象会根据它的引用类型进行不同的回收策略。

19、java中的各种引用（https://blog.csdn.net/baidu_22254181/article/details/82555485）。

20、强引用：在方法内部定义的强引用对象随着方法的出栈就会被回收，但如果强引用对象为全局变量时就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

20、软引用：如果一个对象只具有软引用，则内存空间充足时，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。

20、弱引用：在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

20、软引用和弱引用用在内存资源比较紧俏的场景下。WeakHashMap和ThreadLocal中使用了弱引用。

21、三种GC回收算法（https://www.jianshu.com/p/5d612f36eb0b）：

- 标记-清除算法（Mark-Sweep）：分为标记和清除两阶段：首先标记出所有需要回收的对象，然后统一回收所有被标记的对象。缺点：1、标记阶段和清除阶段的效率都不高。2、清除后产生了大量不连续的内存碎片，导致在程序运行过程中需要分配较大对象的时候，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。
- 复制算法（Copying）：将可用内存按容量划分为大小相等的两块，每次只用其中一块。当这块内存用完了，就将还存活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，同时GC线程将更新存活对象的内存引用地址指向新的内存地址。优点：每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要按顺序分配内存即可。2、实现简单，运行高效。缺点：1、浪费了一半的内存空间。
- 标记-整理算法（Mark-Compact）：标记出所有需要回收的对象，在标记完成后，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。优点：1、弥补标记/清除算法当中，内存区

域分散的缺点。2、消除了复制算法当中，内存减半的高额代价。缺点：1、效率不高，不仅要标记所有存活对象，还要整理所有存活对象的引用地址。

22、垃圾回收器中的并行和并发概念和java中的优点区别：

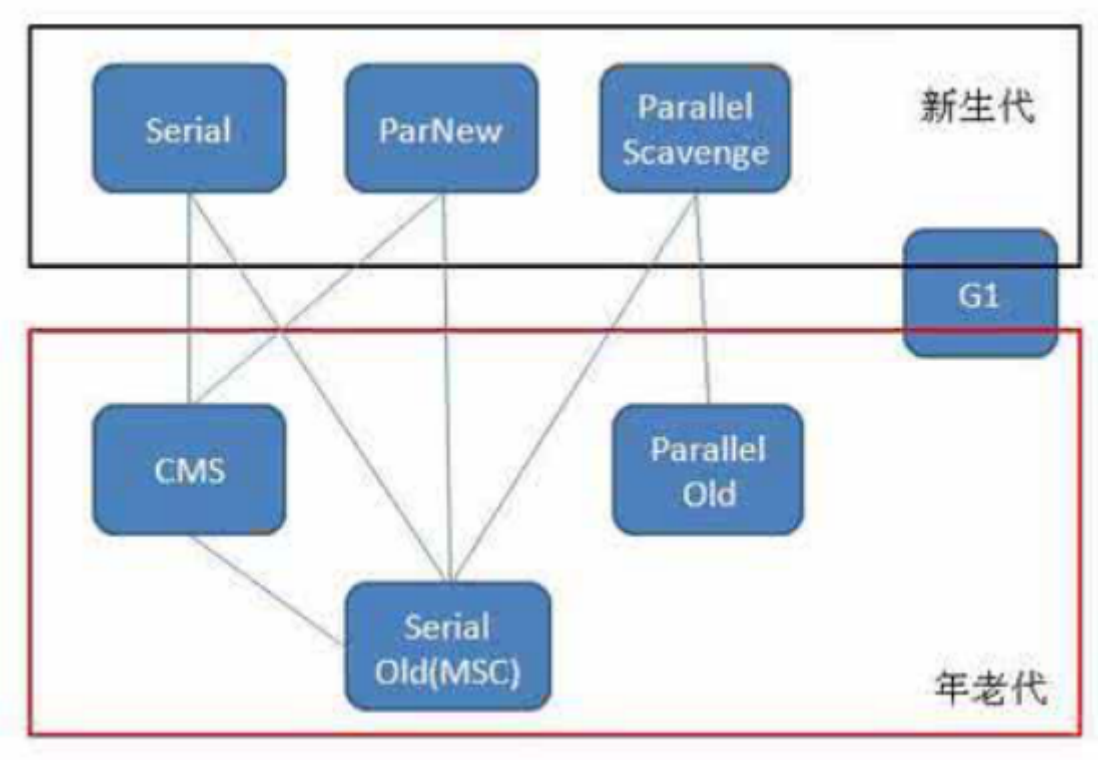
- 并行：垃圾收集多线程同时进行。
- 并发：垃圾收集的多线程和应用的多线程同时进行。

23、新生代所有垃圾回收器选用的都是复制算法。并且都会STW。因为新生代内存被划分为Eden和Survivor区，同时每次minor gc后存活的对象不是很多，所以新生代垃圾回收器都采用了复制算法。

24、G1垃圾回收器会将内存分成2014个region，每次垃圾回收之后它的内存都很规整（此处的规整是对于每一个region来说的（每个region块中不会存在内存碎片），即垃圾回收后每个region中数据存放要么是连续要么整个region就是空的）。

25、GC垃圾回收器发展衍进的方向就是缩短Stop The World的时间。GC调优也是为了尽量缩短STW的时间。

26、需要记忆下图，老年代与新生代垃圾回收器之间不是随意搭配的，搭配关系如下图：



27、新生代垃圾回收器有三种：

1. Serial：最古老、最成熟、单线程、独占式、适合单cpu服务器的垃圾回收器，采用了复制算法，整个回收过程都会STW。
2. ParNew：ParNew是Serial的多线程版本，除了使用多线程进行垃圾收集之外，其它行为和Serial和完全一样，即意味着它也采用了复制算法，整个回收过程也会STW，只不过停顿时间比Serial少。
3. Parallel Scavenge（详细看书籍）：使用了复制算法，多线程并行进行垃圾回收，整个回收过程会STW。Parallel Scavenge收集器的目的是达到一个可控制的吞吐量，所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%，如果对于Parallel Scavenge收集器运作原理不太了解，手工优化存在困难，可以将参数-XX:+UseAdaptiveSizePolicy 打开，这个参数打开之后，就不需要手工指定新生代大小、Eden和Survivor区的比例、晋升老年代对象大小等细节参数了，虚拟机会自适应调节。自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

28、老年代垃圾回收器：

1. Serial Old：Serial Old是Serial的老年代版本，是一个单线程收集器，使用“标记-整理”算法，整个过程会STW。Serial Old的用途之一是作为CMS垃圾回收器的后备预案。
2. Parallel Old：Parallel Old是Parallel Scavenge的老年代版本，采用“标记-整理”算法，使用多线程进行回收。在注重吞吐量以及CPU资源敏感的场合，和Paralle Scavenge搭配使用。
3. Concurrent Mark Sweep（CMS）：CMS收集器是一种以获取最短回收停顿时间为目标的收集器，采用了“标记-清除”算法，整个过程分为四步，初始标记和重新标记时需要STW：
    a. 初始标记：仅仅标记一下GC Roots能直接关联到的对象，速度很快。
    b. 并发标记：和用户的应用程序同时进行，进行GC RootsTracing的过程，即追踪初始标记关联的对象。

c. 重新标记： 为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

d. 并发清除：将最终标记的对象回收。

28、CMS回收过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

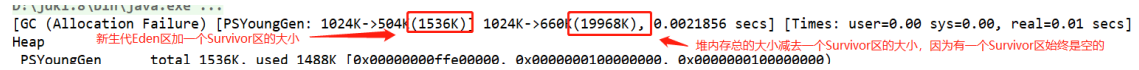
28、由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。同时用户的线程还在运行，需要给用户线程留下运行的内存空间。-XX:CMSInitialOccupyFraction ，因为以上两点，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK 早期版本的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX：CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX：CMSInitiatingOccupancyFraction设置得太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。-XX:+UseCMSCompactAtFullCollection为了解决这个问题，CMS收集器提供了一个这个开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

-XX：CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值为0，表示每次进入FullGC时都进行碎片整理）。

29、动态对象年龄判断（https://www.jianshu.com/p/989d3b06a49d）

29、空间分配担保：重点在于每次Minor GC之前进行检查
（https://www.jianshu.com/p/62c37dc7d638）

30、gc日志解析：

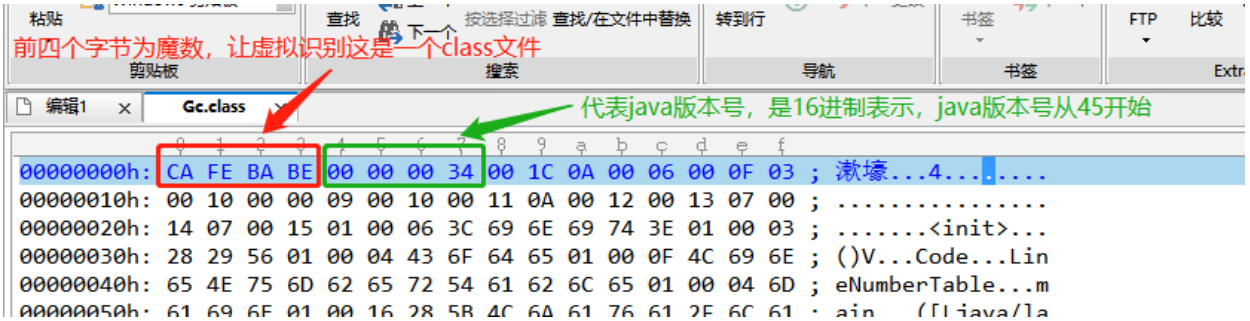


31、JDK为我们提供的工具（jps、jstat等）无关乎操作系统，只要装了jdk就能使用。使用jconsole、jvisualvm时jdk小版本号也必须一致。

30、浅堆和深堆：

- **浅堆（Shallow Heap）：是指一个对象所消耗的内存。在32位系统中，一个对象引用会占据4个字节，一个int类型会占据4个字节，long型变量会占据8个字节，每个对象头需要占用8个字节。**
- **深堆（Retained Heap）：要理解深堆，首先需要了解保留集（Retained Set）。对象A的保留集指当对象A被垃圾回收后，可以被释放的所有的对象集合（包括对象A本身，例如一个map对象被回收，则其里边没逃逸的元素都可以被回收），即对象A的保留集可以被认为是只能通过对象A被直接或间接访问到的所有对象的集合。通俗地说，就是指仅被对象A所持有的对象的集合。深堆是指对象的保留集中所有的对象的浅堆大小之和。**
- **区别：浅堆指对象本身占用的内存，不包括其内部引用对象的大小。一个对象的深堆指只能通过该对象访问到的（直接或间接）所有对象的浅堆之和，即对象被回收后，可以释放的真实空间。**
- **举例：对象A引用了C和D，对象B引用了C和E。那么对象A的浅堆大小只是A本身，不含C和D，而A的实际大小为A、C、D三者之和。而A的深堆大小为A与D之和，由于对象C还可以通过对象B访问到，因此不在对象A的深堆范围内。**

31、认识二进制字节码文件（class文件），每个Class文件的头4个字节称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件。使用魔数而不是扩展名来进行识别主要是基于安全方面的考虑，因为文件扩展名可以随意地改动。文件格式的制定者可以自由地选择魔数值，只要这个魔数值还没有被广泛采用过同时又不会引起混淆即可：



31、Class文件是一组以8位字节为基础单位的二进制流。Class文件实际上它并不一定以磁盘文件的形式存在，也可以将它存在数据库中。

31、在以war包形式往tomcat中部署以后，假如项目特别大，项目有改动时，我们不需要重新部署整个项目，只需用改过的java文件编译出来的class文件去替换线上旧的class文件。前人踩过的坑：定义的一个常量类，修改之后替换线上class文件之后不生效，这时候就需要重启项目，因为静态常量（被static final修饰的常量）在编译时期会被放入常量池，因此获取静态常量不会进行所属类初始化从而导致改的文件没生效。

31、所谓字节码文件就是.java文件编译之后生成的class文件。只有你足够强，完全可以跳过java文件去直接写class文件，也是可以运行的。cglib就是直接编辑的class文件。

32、ManagementFactory是一个JDK为我们提供获取各种JVM信息的工厂类
(https://www.jianshu.com/p/5d854245051d)

33、什么是成员变量、类变量、实例变量等
(https://blog.csdn.net/zm13007310400/article/details/77513000)

33、类变量、实例字段、局部变量会不会被赋零值的问题：

- 实例字段：当对对象分配内存完成后，虚拟机需要将分配到的内存空间都初始化为零值，这一步操作保证了对象的实例字段在java代码中可以不赋初始值就可以直接使用，因为程序能访问到这些字段的数据类型所对应的零值。
- 类变量：类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值（零值），另外一次在初始化阶段，赋予程序员定义的初始值。因此，即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值。
- 局部变量：一个局部变量定义了但没有赋初始值是不能使用的，不要认为java中任何情况下都存在诸如整型变量默认为0，布尔型默认为false等这样的默认值，下图中的代码就会编译报错：

```
public static void main(String[] args) {
    int a;
    System.out.println(a);
}
```

编译报错，因为没有赋初始值

33、类常量和实例常量在创建的时候一定要赋我们需要的值，不要误以为创建的时候不赋值，可以延时赋值（因为局部常量刚创建的时候可以不赋值，既可以延时赋值）。底层原因

是因为类加载的时候会给类常量赋零值，创建对象的时候也会给实例常量赋零值，而局部变量不会被赋零值，因为常量只能被赋值一次，所以类常量和实例常量一定要在创建的时候赋值，不然取到的就是它的零值。

34、为什么类变量（静态变量）、类方法（static方法）可以通过类名访问，而实例字段、实例方法只能通过对象调用？---------->当我们熟悉了类的加载过程以及对象的创建过程后这个问题就很明了。

- 在类加载过程的准备阶段会为类变量和类方法在metaspace区分配内存，在初始化阶段会对类变量进行赋值，所以只要类被加载到虚拟机内存中了，那么类变量和类方法肯定也已经加载到内存中，因此我们可以通过类名直接访问类变量和类方法，同时也意味着类变量和类方法的生命周期与类的加载和卸载同步。
- 而通过对象的创建过程我们可以知道，实例字段和实例方法只有在我们new对象的时候才会为他们分配内存，实例字段在堆中分配，而实例方法在metaspace中分配。因此实例字段和实例方法的生命周期和实例对象同步，即只有new出对象了，实例字段和实例方法才会被加载到内存中，此时我们才可以访问。

35、class文件中的常量池和运行时常量池辨析：

- class文件中中的常量池主要存放字面常量和符号引用。
  - 字面常量：比较接近于java语言层面的常量概念，如文本字符串、声明为final的常量值（包括实例常量，虽然实例常量只能通过对象访问，但是实例常量在创建的时候就需要初始化，之后它就不会在改变，所以我觉得实例常量和类常量差不多，只不过实例常量不能通过类访问）
  - 符号引用：属于编译原理方面的概念，包括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。
- 运行时常量池：class文件中常量池的内容在类加载后进入方法区的运行时常量池中存放。运行期间也可能将新的常量放入常量池中，比如String的intern()方法。

36、疑问？运行时常量池到底在那个区域，jdk8中是真的在堆中吗？

----------类加载机制----------

1、类加载机制中的类初始化是指类的初始化，而并不是对象的初始化。类初始化是将类变量、类方法加载到内存，而对象初始化是将实例字段、实例方法加载到内存。

1、类初始化时机：

- 遇到new、getstatic、putstatic、invokestatic这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化，生成这四条指令最常见的场景：
  - new指令：使用new关键字实例化对象。
  - getstatic指令：读取一个类的静态字段（非静态常量）。（静态内部类实现单例的原理就和它挂钩）
  - putstatic指令：设置一个类的静态字段（非静态常量）。
  - invokestatic指令：调用一个类的静态方法。
- 对类进行反射调用时，如果类没初始化，则需先触发其初始化。如class.forName()
- 当初始化一个类的时候，如果发现其父类还没初始化，则需先触发其父类初始化。
- 当虚拟机启动时，用户需指定一个主类（包含main()方法的类），虚拟机会先初始化这个类

1、调用类中的static final常量不会触发该类的加载。因为静态常量在编译时会被加载到方法区的常量池中。

1、java为什么非静态内部类中不能有static修饰的属性，但却可以有静态常量？，因为静态常量是被放在常量池中的，读取静态常量不会触发类加载。

2、"加载"和"类加载"两个名字不要混淆：

- 类加载：类加载是一个过程，包括加载、验证、准备、解析、类初始化、使用、卸载。
- 加载：加载是类加载的一个阶段，相当于类加载过程的其它阶段，一个非数组类的加载阶段（准确说加载阶段中如何获取二进制字节流的动作）是开发人员可控性最

强的。

3、比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。这里所指的“相等”，包括代表类的Class对象的equals（）方法、isAssignableFrom（）方法、isInstance（）方法的返回结果，也包括使用instanceof关键字做对象所属关系判定等情况。

3、在自定义类加载器的时候该重写那个方法：一种是重写loadClass方法，另一种是重写findClass方法。其实这两种方法本质上差不多，毕竟loadClass也会调用findClass，但是从逻辑上讲我们最好不要直接修改loadClass的内部逻辑。因为loadClass这个方法是实现双亲委托模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委托模型框架内进行小范围的改动，不破坏原有的稳定结构。同时，也避免了自己重写loadClass方法的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择。

4、双亲委派模型一般不是通过继承关系实现的，而是通过组合关系来复用父类加载器的代码（所谓组合就是子类加载器中定义一个成员属性来引用父类加载器对象，然后通过引用去调用父类加载器的方法完成类加载）。

5、栈帧组成部分：

- 局部变量表：用于存放方法参数和方法内部定义的局部变量。
- 操作数栈：
- 动态连接：
- 返回地址：

6、解释执行和编译执行的区别：

- 解释执行：将编译好的字节码一行一行地翻译为机器码执行。
- 编译执行：以方法为单位，将字节码一次性翻译为机器码后执行。编译过程是多线程进行的

6、JIT（Just In Time）：将java字节码编译成机器语言，因为虚拟机能识别字节码，但是操作系统不能识别，所以需要将字节码翻译成操作系统能够识别的机器指令。当不启用JIT时，所有字节码都是被解释执行的，执行速度比较慢。jdk8中默认采用的是混合模式，即解释执行和编译执行混合使用，当使用频率比较高的热点代码达到编译阈值后jvm会将这些代码编译后并缓存下来，这个缓存区有大小限制，当缓存区满了之后jvm就会提示CodeCache is full，就表示需要增加代码缓存大小。特别注意的是这个缓存区和java内存中的堆、栈等没有关系，它是虚拟机单独申请的一块内存（https://www.cnblogs.com/lingz/archive/2018/07/31/9394238.html）：

- 调整编译阈值的参数为：-XX:CompileThreshold=N
- 调整编译代码缓存大小的参数为:–XX:ReservedCodeCacheSize=N

6、是否达到编译阈值的计数方法有两种（https://www.cnblogs.com/death00/p/11722130.html）：

- 方法调用计数器：方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器热度的衰减（Counter Decay），而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）。进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数-XX:-UseCounterDecay来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。另外，可以使用

-XX:CounterHalfLifeTime参数设置半衰周期的时间，单位是秒。

- 循环回边计数器：回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。

7、方法内联（内联默认开启，-XX:-Inline，可以关闭，但是不要关闭，一旦关闭对性能有巨大影响）：调用一个方法通常要经历压栈和出栈，调用过程会产生一定的时间和空间方面的开销（其实可以理解为一种上下文切换的精简版），那么对于那些方法体代码不是很大，又频繁调用的方法来说，这个时间和空间的消耗会很大，方法内联的优化行为就是把目标方法的代码复制到发起调用的方法之中，避免发生真实的方法调用。JVM 会自动识别热点方

法，并对它们使用方法内联进行优化。但要强调一点，热点方法不一定会被 JVM 做内联优化，如果这个方法体太大了，JVM 将不执行内联操作。而方法体的大小阈值，我们也可以通过参数设置来优化：

- 热点方法：热点方法字节码小于 325 字节的都会进行内联，我们可以通过 -XX:MaxFreqInlineSize=N来设置大小值，注意这个很热与热点编译不同，没有任何参数可以调整热度，只能通过jvm自动识别。

- 非热点方法：方法小于35个字节码，一定会内联，这个大小可以通过参数 -XX:MaxInlinesSzie=N 调整。

7、热点方法的优化可以有效提高系统性能，一般我们可以通过以下几种方式来提高方法内联：

- 通过增加方法体字节码大小阈值（热点方法的字节码小于325字节时才会内联，可以将它调大），以便更多的方法可以进行内联，但这种方式意味着需要占用更多地内存（因为需要将目标方法复制到发起调用的方法之中，多产生的这个目标方法会占额外的内存）；
- 在编程中，避免在一个方法中写大量代码，习惯使用小方法体；此处就联系到了编写优雅程序中的观点控制方法大小，一个方法中的内容越少，当该方法经常被执行时，则容易进行方法内联，从而优化性能。
- 尽量使用 final、private、static 关键字修饰方法，编码方法因为继承，会需要额外的类型检查。因为通过final、private、static修饰的方法能保证“编译器可知，运行期不可变”。

----------如何优雅的编写java程序----------

1、构造器参数特别多的时候可以使用工厂模式或者建造者模式。

2、不需要实例化的类应该构造器私有，例如一些工具类一般提供的都是静态方法，这些类不应该提供具体的实例。JDK中的Arrays、Collections、Objects等都是工具类，它们的构造器都私有化了。但是构造器私有化后虽然我们不能new它的实例，但是通过反射可以做到，对于这个问题Objects的做法很值得借鉴，他在私有构造中抛出了Error，因为反射

也是通过调用构造方法创建对象的，所以通过反射创建实例的时候就会抛异常，这样就彻底封死了所有能实例化的方法：

```
public class Collections {
    // Suppresses default constructor, ensuring non-instantiability.
    private Collections() {
    }
```

构造方法私有化

```
*/
public final class Objects {
    private Objects() {
        throw new AssertionError( detailMessage: "No java.util.Objects instances for you!");
    }
```

构造私有，通过反射调用的时候会抛出Error异常

3、不要创建不必要的对象，如自动装箱：

```
public class Sum {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        Long sum = 0L;//对象
        for(long i=0;i<Integer.MAX_VALUE;i++) {
            sum = sum+i;
            //new 20多亿的Long的实例
        }

        System.out.println("spend time:"+(System.currentTimeMillis()-start)+"ms");
    }
}
```

应该使用long

- **自动装箱和拆箱：**
  - **定义：自动装箱就是Java自动将原始类型值转换成对应的对象，比如将int的变量转换成Integer对象，这个过程叫做装箱，反之将Integer对象转换成int类型值，这个过程叫做拆箱。装箱和拆箱过程都是由JDK编译器自动完成的。**
  - **原理：自动装箱时编译器调用valueOf将原始类型值转换成对象，同时自动拆箱时，编译器通过调用类似intValue(),doubleValue()这类的方法将对象转换成原始类型值。**
  - **缓存策略：除了double和float的自动装箱没有使用缓存（double、float是浮点型的，没有特别的热的（经常使用到的）数据，缓存效果没有其它几种类型使用效率高），每次都是new新的包装类对象，其它六种基本类型都使用了缓存策略。使用缓存策略是因为缓存的这些对象都是经常使用到的（如字符、-128至127之间的数字），防止每次自动装箱都创建一次对象的实例。以Integer为例，它维护类一个静态内部类IntegerCache，**

IntegerCache内部维护了一个Integer的静态常量数组Integer *cache*[]，在类加载的时候，执行static静态块进行初始化-128到127之间的Integer对象，存放到cache数组中。cache属于常量，存放在java的方法区中（https://www.jianshu.com/p/0ce2279c5691）：

```
//1、这个没解释的就是true
System.out.println("i=i0\t" + (i == i0));  //true
//2、int值只要在-128和127之间的自动装箱对象都从缓存中获取的，所以为true
System.out.println("i1=i2\t" + (i1 == i2));  //true
//3、涉及到数字的计算，就必须先拆箱成int再做加法运算，所以不管他们的值是否在-128和127之间，只要数字一
System.out.println("i1=i2+i3\t" + (i1 == i2 + i3));//true
//比较的是对象内存地址，所以为false
System.out.println("i4=i5\t" + (i4 == i5));  //false
//5、同第3条解释，拆箱做加法运算，对比的是数字，所以为true
System.out.println("i4=i5+i6\t" + (i4 == i5 + i6));//true
//double的装箱操作没有使用缓存，每次都是new Double，所以false
System.out.println("d1=d2\t" + (d1==d2));//false
```

4、避免使用终结方法，即finalize()方法（https://baijiahao.baidu.com/s?id=1655232869611610920&wfr=spider&for=pc）：finalize()是Object中的方法，当垃圾回收器将要回收对象所占内存之前被调用。如果自定义的类重写了finalize()方法，java虚拟机在进行垃圾回收的时候，一看到这个对象类含有finalize函数，就把这个函数交给FinalizerThread处理，而包含了这个finalize的对象就会被添加到FinalizerThread的执行队列，并使用一个链表，把这些包含了finalize的对象串起来，它的影响在于只要finalize没有执行，那么这些对象就会一直存在堆区不能被回收。

5、使类和成员的可访问性最小化，编写程序和设计架构，最重要的目标之一就是模块间的解耦，使类和成员的可访问性最小化五一是最有效的途径之一。（假如某个包下的A类不想让别的包中的类访问，那么可以将类的修饰符改成默认，需要注意类的修饰符只能是public或者default，既只能同包访问或者所有类都可以访问，具体原因：https://blog.csdn.net/yangyong0717/article/details/78379760）。

6、使类的可变性最小化，尽量使类不可变（但不要刻意），不可变的类比可变的类更加易于设计、实现和使用，而且更不容易出错，更安全。常用手段：

- 不提供任何可以修改对象状态的方法，例如不给实例字段提供set方法。
- 使所有的域都是final的。
- 是使所有的域都是私有的。

7、复合优于继承（https://www.cnblogs.com/jjfan0327/p/6961707.html）：

- **IS-A关系和HAS-A关系有什么区别（https://cloud.tencent.com/developer/ask/195015）：IS-A关系是继承，HAS - A关系是组合。**
- **原因：继承打破了类的封装性。继承机制会把父类API中的所有缺陷传播到子类中，而复合允许新设计的API来隐藏这些细节。**
- **复合实现方式：不扩展现有的类，而是在新类中增加一个私有域，引用现有类的一个实例，新类中的每个实例方法都可以调用被包含的现有类实例中对应的方法，并返回结果。装饰器模式就是通过复合实现的。**
- **如何从继承和复合之间做出选择：比较抽象的说法是，只有子类和父类确实存在"is-a"关系的时候使用继承，否则使用复合。或者比较实际点的说法是，如果子类只需要实现超类的部分行为，则考虑使用复合。**
- **优缺点：**

继承的优缺点

优点：

- 支持扩展，通过继承父类实现，但会使系统结构较复杂
- 易于修改被复用的代码

缺点：

- 代码白盒复用，父类的实现细节暴露给子类，破坏了封装性
- 当父类的实现代码修改时，可能使得子类也不得不修改，增加维护难度。
- 子类缺乏独立性，依赖于父类，耦合度较高
- 不支持动态拓展，在编译期就决定了父类

组合的优缺点

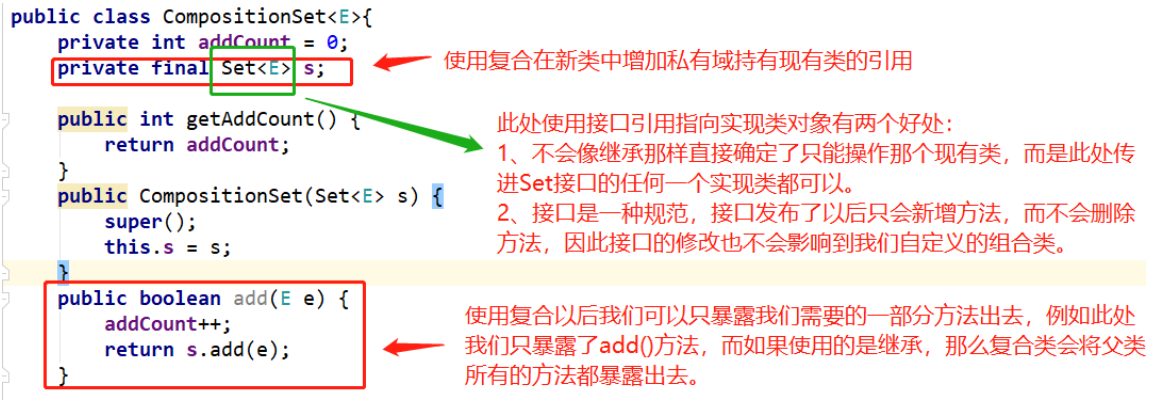
优点：

- 代码黑盒复用，被包括的对象内部实现细节对外不可见，封装性好。
- 整体类与局部类之间松耦合，相互独立。
- 支持扩展
- 每个类只专注于一项任务
- 支持动态扩展，可在运行时根据具体对象选择不同类型的组合对象(扩展性比继承好)

缺点：

- 创建整体类对象时，需要创建所有局部类对象。导致系统对象很多。

- **复合使用：**



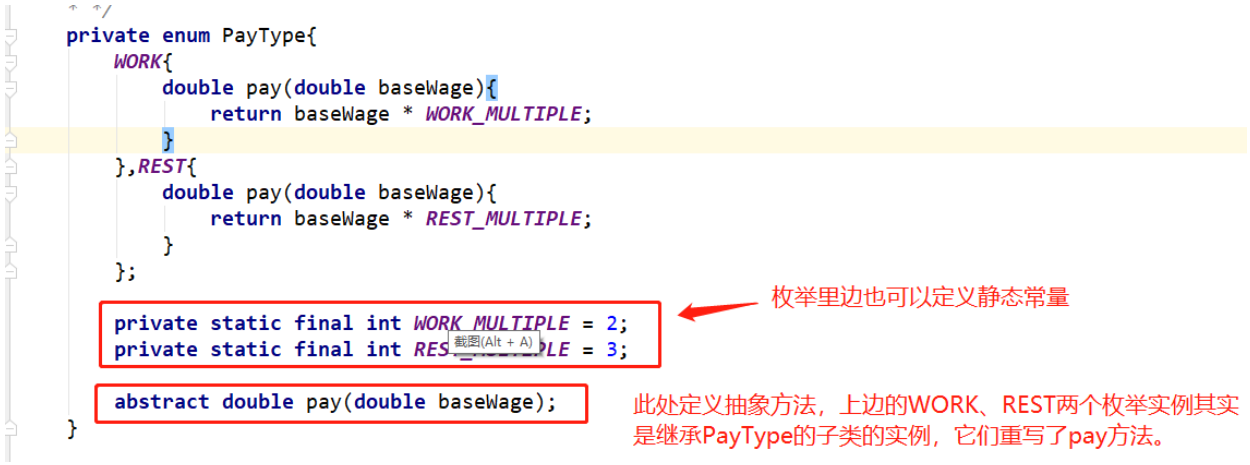
8、接口优于抽象类：java只支持单继承（不考虑间接继承），但是类允许实现多个接口，所以当发生业务变化时，新增接口，并且只需要让进行业务变化的类实现新接口即可。但是实现抽象类有可能导致不需要变化的类也不得不实现新增的业务方法（只要父类变化那么所有的子类都需要跟着改变，牵一发而动全身）。在JDK里常用的一种设计方法是：定义一个接口，声明一个抽象的骨架类实现接口，骨架类实现通用的方法，而实际的业务类可以同时实现接口又继承骨架类，也可以只实现接口。如HashSet实现了implements Set接口 但是又extends 类AbstractSet，而AbstractSet本身也实现了Set接口。其他如Map，List都是这样的设计的。

9、可变参数要慎用：可变参数是允许传0个参数的，如果是参数个数在1~多个之间的时候，要做单独的业务控制。

10、返回零长度的数组或集合，不要返回null：方法的结果返回null，会导致调用方要单独处理为null的情况。返回零长度，调用方可以统一处理，如使用foreach即可。JDK中也为我们提供了Collections.*EMPTY_LIST*这样的零长度集合。

11、优先使用标准异常（即jdk为我们提供的异常）：这样可以达到代码的复用，因为jdk已经提供了很多类型的异常，我们额外过多的自定义异常反倒让虚拟机需要加载的类变的更多，也是一种负担。

12、用枚举代替通过类常量定义状态。枚举类和正常的类使用方式一样，只不过构造私有化了，我们需要在类内部创建好对象，使用例子看vip-jvm工程（里边加减乘除使用this指向当前对象，别看糊涂了，通过this获取当前调用对象。线程池的Worker、以及自己写的动态代理代码都是使用this。这种方式没啥特殊，在源码中看到别胡思乱想就ok），策略枚举使用方式如下图：



13、将局部变量的作用域最小化，做到以下两点：

- **在第一次使用的地方进行声明（不要在方法的开头将方法中用到的所有局部变量都定义好），这样做有两个优点：**
  - **可以使栈帧越小：因为方法在被调用运行时会被打包成栈帧，而方法中定义的局部变量存放在栈帧的局部变量表中，局部变量表的容量是以变量槽（Slot）为最小单位的，同时局部变量表中的Slot是可以重用的，方法体中定义的变量，其作用域不一定会覆盖整个方法体，如果当前字节码程序计数器的值已经超出了某个变量的作用域，那么这个变量对应的Slot就可以交给其他变量使用，而如果我们在方法开头就定义所有变量，那么虚拟机就会为这些变量都分配Slot，这样就达不到Slot复用的目的。**
  - **Slot复用会影响垃圾收集行为：具体看《深入理解虚拟机》240页的内容。假如局部变量a的作用域已结束，但此之后，没用任何对局变量表的读写操作，即a变量所占用的Slot还没有被其它变量所复用，那么此时a变量就不会被回收。**
- **局部变量是需要我们手动初始化的（因为虚拟机不会给局部变量赋零值），当初始化条件不满足时，就不要声明，避免局部变量过早声明导致不正确的使用（例如在声明之后使用之前不小心将值给覆盖了）。**

14、精确计算，避免使用float和double，可以使用int、long、BigDecimal：所谓使用int、long就是加入我们需要做0.1+0.2，则我们可以转换成（1+2）/10。

15、字符串拼接比较频繁时使用StringBuilder或者StringBUffer。

16、控制方法大小，尽量控制在一屏大小。同时方法体控制的比较小的话也会提升运行执行速度，因为虚拟机会通过方法内联进行优化。