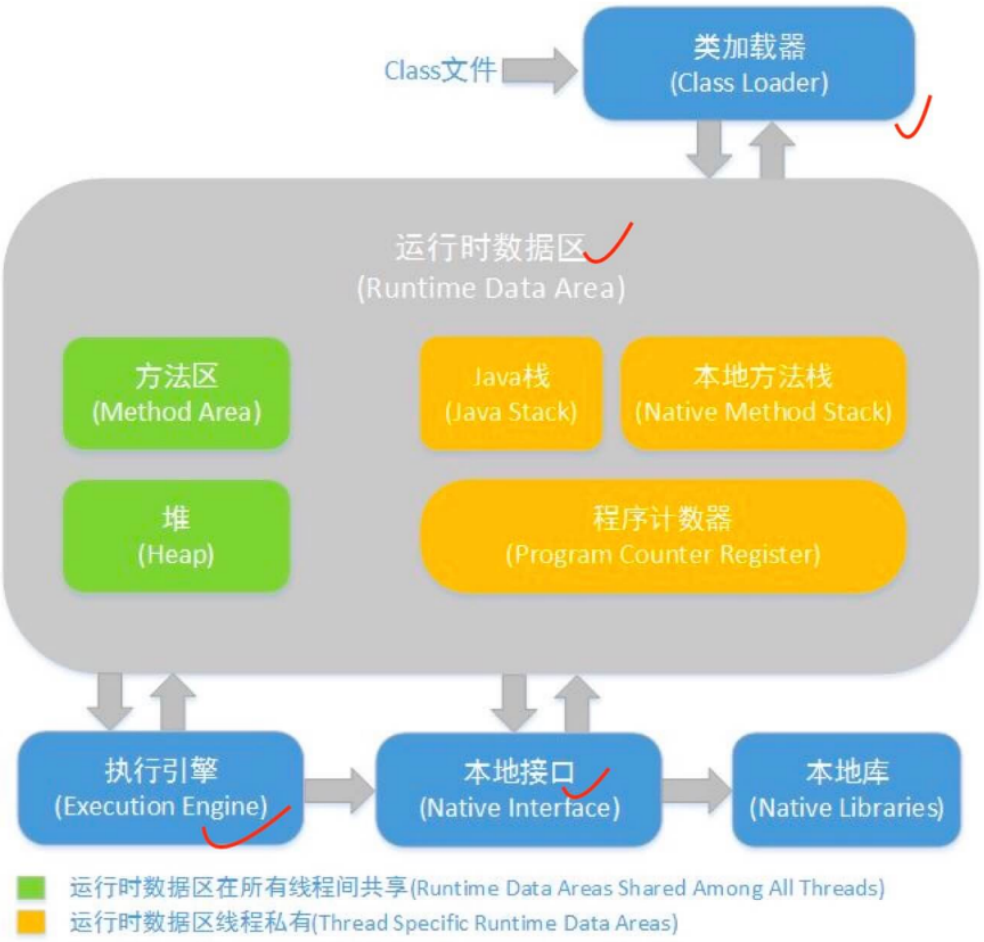
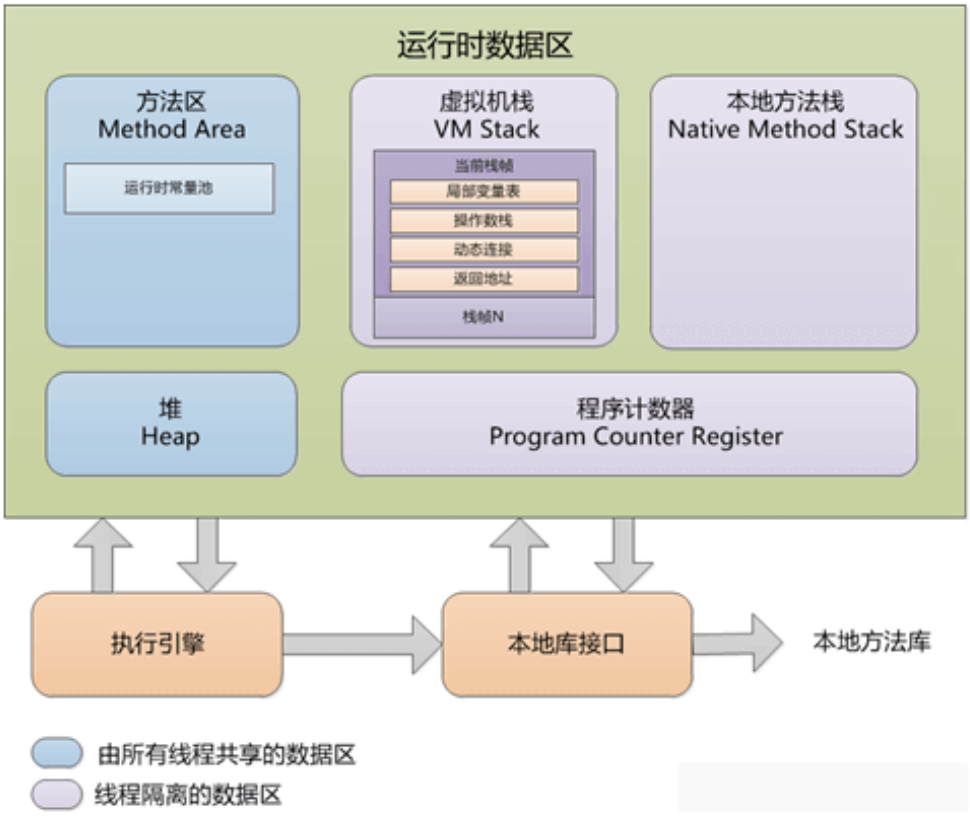
虚拟机相关

JVM 由哪些部分组成



- 类加载器，在 JVM 启动时或者类运行时将需要的 class 加载到 JVM 中。
- 内存区，将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块，如实际机器上的各种功能的寄存器或者PC指针的记录器等。
- 执行引擎，执行引擎的任务是负责执行class文件中包含的字节码指令，相当于实际机器上的 CPU 。
- 本地方法调用，调用 C 或 C++ 实现的本地方法的代码返回结果。

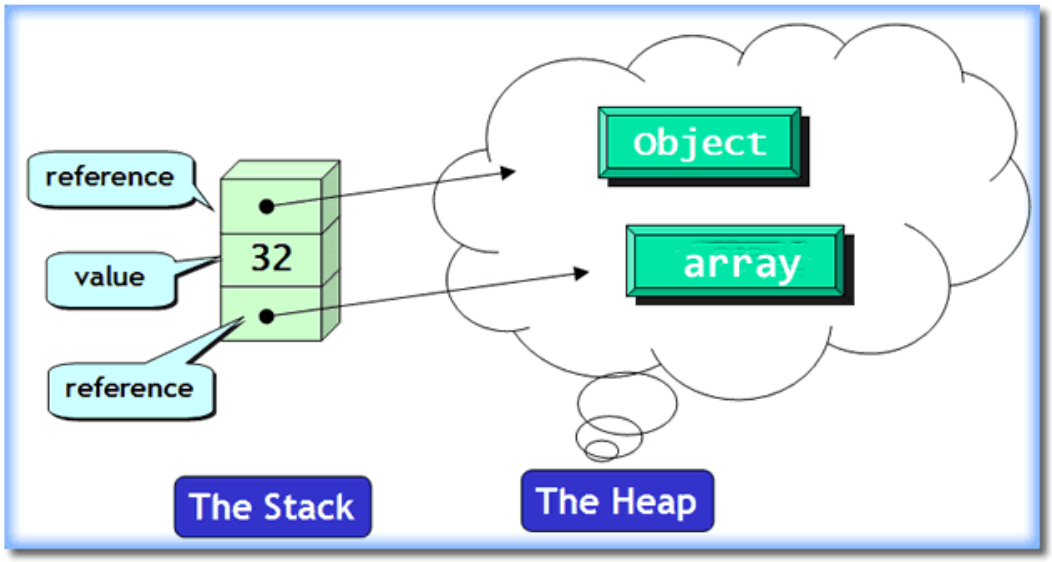
JVM 运行内存的分类？


运行时数据区
方法区
Method Area
运行时常量池
虚拟机栈
VM Stack
当前栈帧
局部变量表
操作数栈
动态连接
返回地址
栈帧N
本地方法栈
Native Method Stack
堆
Heap
程序计数器
Program Counter Register
执行引擎
本地库接口
本地方法库
由所有线程共享的数据区
线程隔离的数据区

- 程序计数器： Java线程私有，类似于操作系统里的PC计数器，它可以看做是当前线程所执行的字节码的行号指示器。
    - 如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器值则为空（Undefined）。
    - 此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError 情况的区域。
- 虚拟机栈（栈内存）：Java线程私有，虚拟机栈描述的是 Java 方法执行的内存模型：
    - 每个方法在执行的时候，都会创建一个栈帧用于存储局部变量、操作数、动态链接、方法出口等信息。
    - 每个方法调用都意味着一个栈帧在虚拟机栈中入栈到出栈的过程。
- 本地方法栈 ：和 Java 虚拟机栈的作用类似，区别是该区域为 JVM 提供使用 Native 方法的服务。
- 堆内存（线程共享）：所有线程共享的一块区域，垃圾收集器管理的主要区域。
    - 目前主要的垃圾回收算法都是分代收集算法，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等，默认情况下新生代按照 8:1:1 的比例来分配。
    - 根据 Java 虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘一样。
- 方法区（线程共享）：各个线程共享的一个区域，用于存储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。
    - 虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。
    - 运行时常量池：是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。

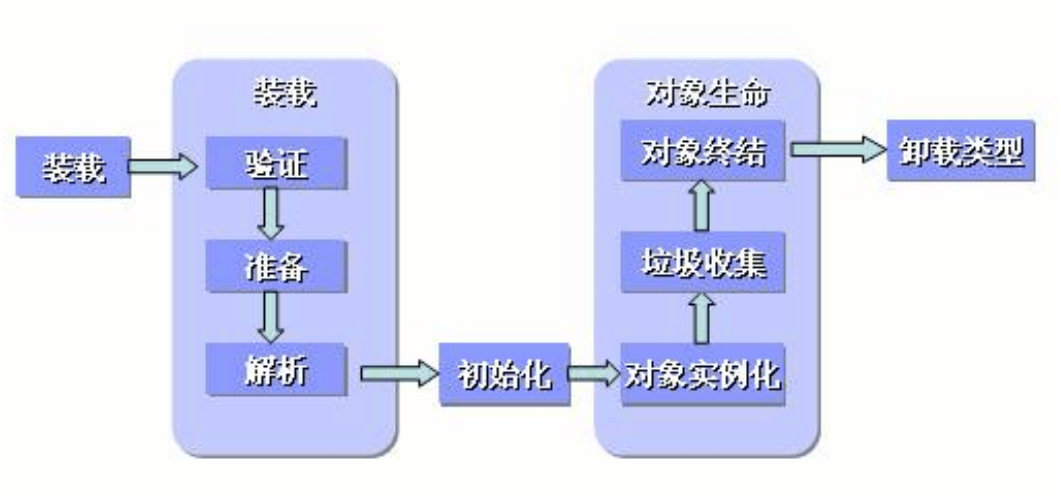
Java 内存堆和栈区别？

- 栈内存用来存储基本类型的变量和对象的引用变量；堆内存用来存储Java中的对象，无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。
- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存；堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。
- 如果栈内存没有可用的空间存储方法调用和局部变量，JVM 会抛出 java.lang.StackOverFlowError 错误；如果是堆内存没有可用的空间存储生成的对象，JVM 会抛出 java.lang.OutOfMemoryError 错误。
- 栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。-Xss 选项设置栈内存的大小，-Xms 选项可以设置堆的开始时的大小。



JAVA 对象创建的过程？

Java对象的创建过程：



Java 中对象的创建就是在堆上分配内存空间的过程：

1）检测类是否被加载 当虚拟机遇到 new指令时，首先先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就执行类加载过程。
2）为对象分配内存
类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。

具体的分配内存有两种情况：第一种情况是内存空间绝对规整，第二种情况是内存空间是不连续的。

- 对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为“指针碰撞”。
- 对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为“空闲列表”。

多线程并发时会出现正在给对象 A 分配内存，还没来得及修改指针，对象 B 又用这个指针分配内存，这样就出现问题了。解决这种问题有两种方案：

- 第一种，是采用同步的办法，使用 CAS 来保证操作的原子性。
- 另一种，是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（ Thread Local Allocation Buffer,TLAB ），分配内存的时候再TLAB上分配，互不干扰。可以通过 -XX:+/-UseTLAB 参数决定。

3）为分配的内存空间初始化零值

对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用。

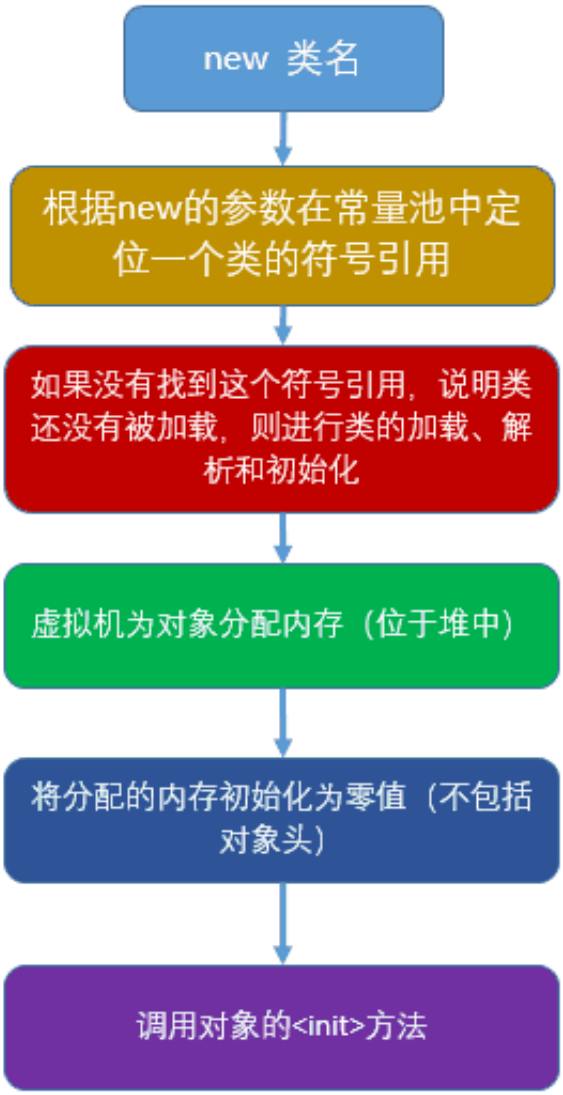
4）对对象进行其他设置

分配完内存空间，初始化零值之后，虚拟机还需要对对象进行其他必要的设置，设置的地方都在对象头中，包括这个对象所属的类，类的元数据信息，对象的 hashcode，GC 分代年龄等信息。

5）执行 init 方法

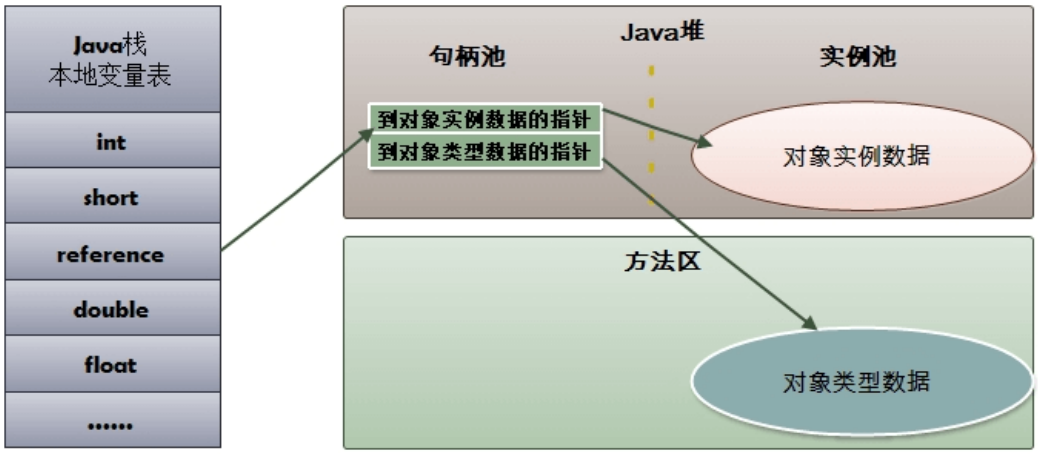
执行完上面的步骤之后，在虚拟机里这个对象就算创建成功了，但是对于 Java 程序来说还需要执行 init 方法才算真正的创建完成，因为这个时候对象只是被初始化零值了，还没有真正的去根据程序中的代码分配初始值，调用了 init 方法之后，这个对象才真正能使用。

到此为止一个对象就产生了，这就是 new 关键字创建对象的过程。过程如下：



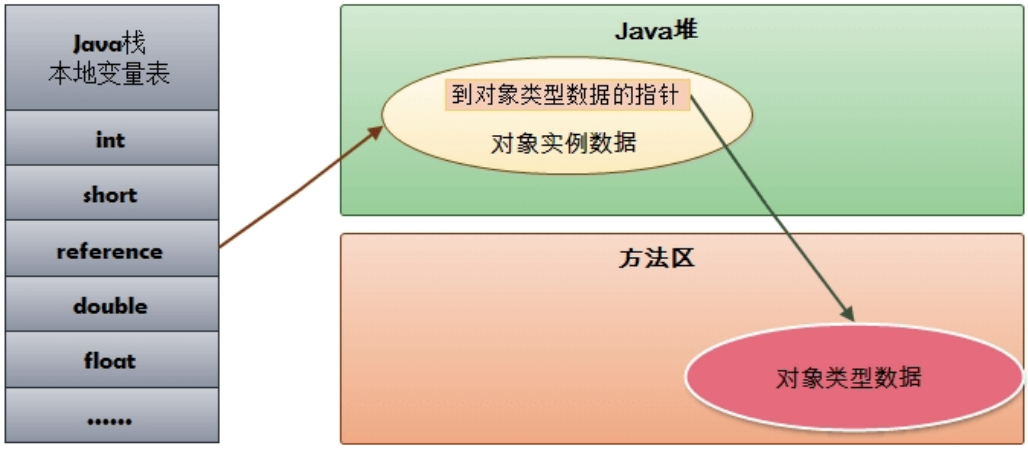
对象是如何定位访问的？

- 句柄定位：Java 堆会画出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。



句柄方式访问对象

- 直接指针访问：Java堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址。



直接指针方式访问对象

这两种对象访问方式各有优势。

- 使用句柄来访问的最大好处，就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。
- 使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

我们目前主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

有哪些 OutOfMemoryError 异常？

在 Java 虚拟机中规范的描述中，除了程序计数器外，虚拟机内存的其它几个运行时区域都有发生的 OutOfMemoryError(简称为“OOM”）异常的可能。

- Java 堆溢出
- 虚拟机栈和本地方法栈溢出
- 方法区和运行时常量池溢出（从JDK8开始，就变成元数据区的内存溢出。）
- 本机直接内存溢出

1）Java 堆溢出

Java堆用于存储对象实例，只要不断地创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆的容量限制后就会产生内存溢出异常。

```
/**
 * VM Args：-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * 限制Java堆的大小为20MB，不可扩展（将堆的最小值-Xms参数与最大值-Xmx参数设置为一样即可避免堆自动扩展），
 * 通过参数-XX:+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后进行分析。
 */
public class HeapOOM {

    static class OOMObject {

    }

    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();

        while (true) {
            list.add(new OOMObject());
        }
    }
}
```

Java堆内存的OOM异常是实际应用中常见的内存溢出异常情况。当出现Java堆内存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap space”。

要解决这个区域的异常，一般的手段是先通过内存映像分析工具（如Eclipse Memory Analyzer）对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。图2-5显示了使用Eclipse Memory Analyzer打开的堆转储快照文件。

如果是内存泄露，可进一步通过工具查看泄露对象到GC Roots的引用链。于是就能找到泄露对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄露对象的类型信息及GC Roots引用链的信息，就可以比较准确地定位出泄露代码的位置。

如果不存在泄露，换句话说，就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

2）虚拟机栈和本地方法栈溢出

关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。
- 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。

重现方式，参见 《OutOfMemoryError异常——虚拟机栈和本地方法栈溢出》 文章。

3）运行时常量池溢出

因为 JDK7 将常量池和静态变量放到 Java 堆里，所以无法触发运行时常量池溢出。如果想要触发，可以使用 JDK6 的版本。

重现方式，参见 《JVM 内存溢出 - 方法区及运行时常量池溢出》 文章。

4）方法区的内存溢出

因为 JDK8 将方法区溢出，所以无法触发方法区的内存溢出溢出。如果想要触发，可以使用 JDK7 的版本。

重现方式，参见 《Java方法区溢出》 文章。

5）元数据区的内存溢出

实际上，方法区的内存溢出在JDK8中，变成了元数据区的内存溢出。所以，重现方式，还是参见《Java方法区溢出》文章，只是说，需要增加 -XX:MaxMetaspaceSize=10m VM 配置项。

6）本机直接内存溢出

重现方式，参见 《JVM内存溢出——直接内存溢出》 文章。

另外，非常推荐一篇文章，胖友耐心阅读，提供了更多有趣的案例，《Java 内存溢出(OOM)异常完全指南》 。

当出现了内存溢出，你怎么排错？

- 1、首先，控制台查看错误日志。
- 2、然后，使用 JDK 自带的 jvisualvm 工具查看系统的堆栈日志。
- 3、定位出内存溢出的空间：堆，栈还是永久代（JDK8以后不会出现永久代的内存溢出）。
  - 如果是堆内存溢出，看是否创建了超大的对象。
  - 如果是栈内存溢出，看是否创建了超大的对象，或者产生了死循环。

Java 对象有哪些引用类型？

《Java 中的四种引用类型》提供的代码示例。

- 强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

- 软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可物的生活用品。如果内存空间足够，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。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

- 弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。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。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

- 虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

为什么要有不同的引用类型？

不像 C 语言，我们可以控制内存的申请和释放，在Java中有时候我们需要适当的控制对象被回收的时机，因此就诞生了不同引用类型，可以说不同的引用类型实则是对GC回收时机不可控的妥协。有以下几个使用场景可以充分的说明：

- 利用软引用和弱引用解决 OOM 问题。用一个 HashMap 来保存图片的路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM 会自动回收这些缓存图片对象所占用的空间，从而有效地避免了 OOM 的问题.
- 通过软引用实现 Java 对象的高速缓存。比如我们创建了一 Person 的类，如果每次需要查询一个人的信息，哪怕是几秒中之前刚刚查询过的，都要重新构建一个实例，这将引起大量 Person 对象的消耗，并且由于这些对象的生命周期相对较短，会引起多次 GC 影响性能。此时，通过软引用和 HashMap 的结合可以构建高速缓存，提供性能。

JVM 垃圾回收算法？

1）标记-清除算法

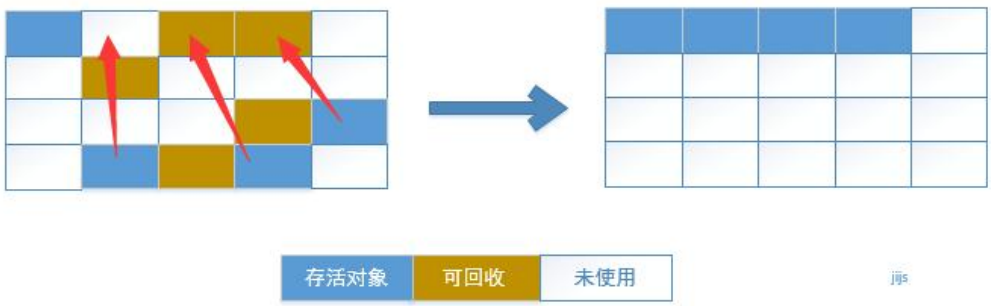
标记-清除（Mark-Sweep）算法，是现代垃圾回收算法的思想基础。分为两个阶段（标记阶段、清除阶段）

一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象（好多资料说标记出要回收的对象，其实明白大概意思就可以了）。然后，在清除阶段，清除所有未被标记的对象。

- 缺点：
  - 1、效率问题，标记和清除两个过程的效率都不高。
  - 2、空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

2）标记-整理算法

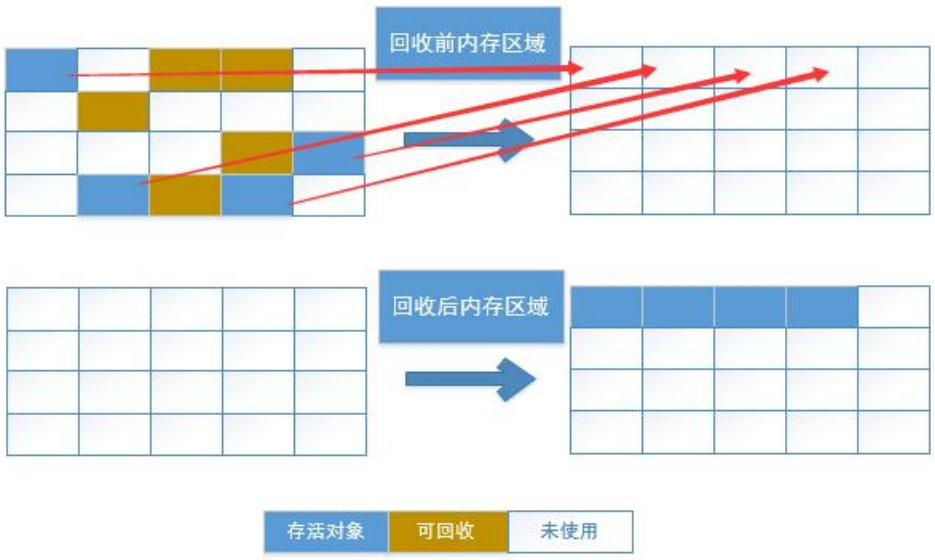
标记整理算法，类似与标记清除算法，不过它标记完对象后，不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。



- 优点：
  - 1、相对标记清除算法，解决了内存碎片问题。
  - 2、没有内存碎片后，对象创建内存分配也更快速了（可以使用TLAB进行分配）。
- 缺点：
  - 1、效率问题，（同标记清除算法）标记和整理两个过程的效率都不高。

3）复制算法

复制算法，可以解决效率问题，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，当这一块内存用完了，就将还存活着的对象复制到另一块上面，然后再把已经使用过的内存空间一次清理掉，这样使得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，只要移动堆顶指针，按顺序分配内存即可（还可使用TLAB进行高效分配内存）。



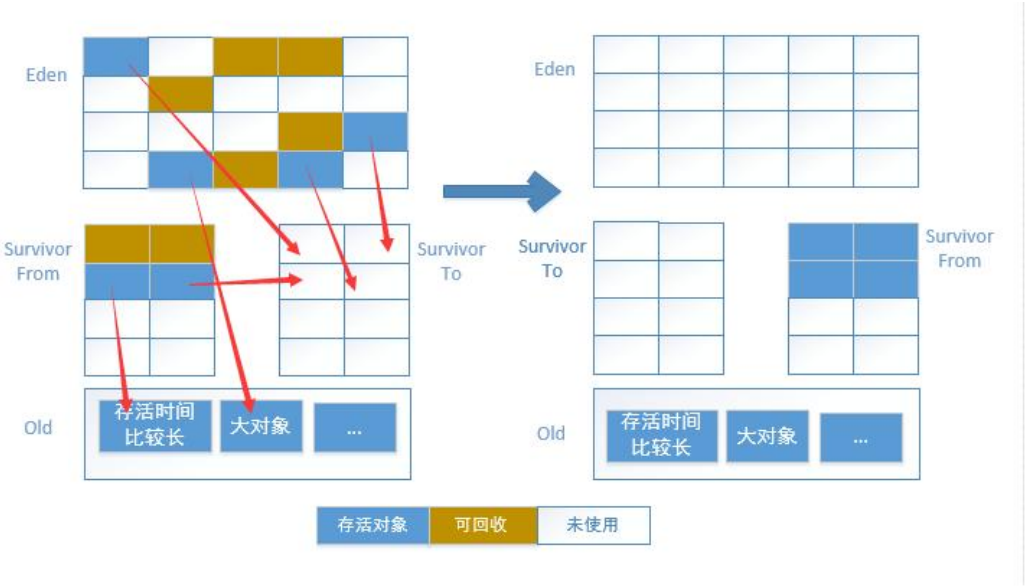
图的上半部分是未回收前的内存区域，图的下半部分是回收后的内存区域。通过图，我们发现不管回收前还是回收后都有一半的空间未被利用。

- 优点：
  - 1、效率高，没有内存碎片。
- 缺点：
  - 1、浪费一半的内存空间。
  - 2、复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。

4）分代收集算法

当前商业虚拟机都是采用分代收集算法，它根据对象存活周期的不同将内存划分为几块，一般是把 Java 堆分为新生代和老年代，然后根据各个年代的特点采用最适当的收集算法。

- 在新生代中，每次垃圾收集都发现有大批对象死去，只有少量存活，就选用复制算法。
- 而老年代中，因为对象存活率高，没有额外空间对它进行分配担保，就必须使用“标记清理”或者“标记整理”算法来进行回收。



- 图的左半部分是未回收前的内存区域，右半部分是回收后的内存区域。
- 对象分配策略：
  - 对象优先在 Eden 区域分配，如果对象过大直接分配到 Old 区域。
  - 长时间存活的对象进入到 Old 区域。
- 改进自复制算法
  - 现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 公司的专门研究表明，新生代中的对象 98% 是“朝生夕死”的，所以并不需要按照 1:1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor 。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。
  - HotSpot 虚拟机默认 Eden 和 2 块 Survivor 的大小比例是 8:1:1，也就是每次新生代中可用内存空间为整个新生代容量的 90%（80%+10%），只有 10% 的内存会被“浪费”。当然，98% 的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10% 的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

Java 内存分配策略？

> 对象优先Eden（新生代）分配

大多数情况下，对象新生代在Eden分配，当Eden空间不足的时候，进行Minor GC

> 大对象直接进入老年代

大对象是指需要连续内存空间的对象，典型的大对象是很长的字符串以及数组，经常出现大对象，会提前出发立即回收以获取足够的连续空间。
XX:MaxTenuringThreshold 用来定义年龄的阈值。

> 动态对象年龄判断

虚拟机并不是永远地要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor相同年龄对象大小总和大于Survivor空间的，则年龄大于或等于该年龄的对象可直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄

> 空间担保分配

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的

如果不成立的话虚拟机会查看HandlePromotionFailure设置值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那么就要进行一次 Full GC。

CMS收集器

> CMS(Concurrent Mark Sweep)操作流程

1.初始标记：仅仅只是标记一下GC Root，能直接关联到的对象，速度很快，需要停顿；
2.并发标记：进行GC Root Tracking的过程，在整个回收过程中耗时较长，但不需要停顿；
3.重新标记：为了修正并发标记期间因用户线程继续运行而导致标记产生变动的那一部分对象进行标记，需要停顿；
4.并发清楚：不需要停顿

> 特点

1.在整个过程中耗时较长的并发标记和并发清楚过程中，收集器线程都可以和用户线程一起工作，不需要停顿；
2.吞吐量低：低停顿时间是以牺牲吞吐量为代价，导致CPU利用率不够高；
3.无法处理浮动垃圾可能出现Concurrent Model Failure
4.标记清楚算法导致的空间碎片往往出现老年代空间剩余，而无法找到足够大的连续空间来分配当前对象，不得不提前出发Full GC

G1收集器

堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。

> G1(Garbage-First)操作流程

1、初始标记
2、并发标记
3、最终标记：为了修正在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标记记录，虚拟机将这段时间对象变化记录在线程的 RememberedSetLogs 里面，最终标记阶段需要把 Remembered SetLogs的数据合并到Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
4、筛选回收：首先对各个Region中的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

> 特点

1、空间整合：整体来看是基于“标记-整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。
2、可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在GC上的时间不得超过 N 毫秒。

G1 和 CMS 的区别？

- CMS ：并发标记清除。他的主要步骤有：初始收集，并发标记，重新标记，并发清除（删除）、重置。
- G1：主要步骤：初始标记，并发标记，重新标记，复制清除（整理）
- CMS 的缺点是对 CPU 的要求比较高。G1是将内存化成了多块，所有对内段的大小有很大的要求。
- CMS是清除，所以会存在很多的内存碎片。G1是整理，所以碎片空间较小。
- G1 和 CMS 都是响应优先把，他们的目的都是尽量控制 STW 时间。

什么情况下会出现 Young GC？

对象优先在新生代Eden区中分配，如果Eden区没有足够的空间时，就会触发一次 Young GC 。

Full GC 的触发条件

> 调用System.gc

> 老年代空间不足

老年代空间不足的场景为大对象分配的时候进入老年代、长期存活的对象进入老年代，为了避免以上原因引发的Full GC，应尽量不要创建过大的对象以及数组，除此之外可以通过-Xmm虚拟机参数调整新生代的大小，让对象尽量在新生代被回收，不进入老年代，可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

> 空间担保分配失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC

> JDK 1.7 及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

> Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

线上发送频繁full gc如何处理?

> 查看gc日志，发现old区fgc后大小没有变化

GC 日志:

生成下面日志使用的选项： -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:d:/GClogs/tomcat6-gc.log。

```
4.231: [GC 4.231: [DefNew: 4928K->512K(4928K), 0.0044047 secs] 6835K->3468K(15872K), 0.0045291 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

4.445: [Full GC (System) 4.445: [Tenured: 2956K->3043K(10944K), 0.1869806
```

最前面的数字 4.231 和 4.445 代表虚拟机启动以来的秒数。

[GC 和 [Full GC 是垃圾回收的停顿类型，而不是区分是新生代还是年老代，如果有 Full 说明发生了 Stop-The-World 。如果是调用 System.gc() 触发的，那么将显示的是 [Full GC (System) 。

接下来的 [DefNew, [Tenured, [Perm 表示 GC 发生的区域，区域的名称与使用的 GC 收集器相关。 Serial 收集器中新生代名为 "Default New Generation"，显示的名字为 "[DefNew"。对于ParNew收集器，显示的是 "[ParNew"，表示 “Parallel New Generation”。 对于 Parallel Scavenge 收集器，新生代名为 "PSYoungGen"。年老代和永久代也相同，名称都由收集器决定。

方括号内部显示的 “4928K->512K(4928K)” 表示 “GC 前该区域已使用容量 -> GC 后该区域已使用容量（该区域内存总容量）”。

再往后的 “0.0044047 secs” 表示该区域GC所用时间，单位是秒。

再往后的 “6835K->3468K(15872K)” 表示 “GC 前Java堆已使用容量 -> GC后Java堆已使用容量 （Java堆总容量）”。

再往后的 “0.0045291 secs” 是Java堆GC所用的总时间。

最后的 “[Times: user=0.00 sys=0.00, real=0.00 secs]” 分别代表 用户态消耗的CPU时间、内核态消耗的CPU时间 和 操作从开始到结束所经过的墙钟时间。墙钟时间包括各种非运算的等待耗时，如IO等待、线程阻塞。CPU时间不包括等待时间，当系统有多核时，多线程操作会叠加这些CPU时间，所以user或sys时间会超过real时间。

> 去线上dump内存看是什么对象，用memory analyzer分析,查看对象大小情况

详细说下JMM

JMM详细接收

9、Integer x =5,int y =5，比较x =y 都经过哪些步骤（装箱、拆箱）？

Integer x =5 -> 执行了Integer.valueOf(5)
int y =5 是直接将5写入y
x=y会调用if_icmp方法比较

那什么是拆箱呢？顾名思义，跟装箱对应，就是自动将包装器类型转换为基本数据类型：

```
Integer i = 10;  //装箱
int n = i;   //拆箱
```

简单一点说，装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类型转换为基本数据类型。

在装箱的时候自动调用的是Integer的valueOf(int)方法。而在拆箱的时候自动调用的是Integer的intValue方法。

Double类的valueOf方法会采用与Integer类的valueOf方法不同的实现。很简单：在某个范围内的整型数值的个数是有限的，而浮点数却不是。

```
Double i1 = 100.0;
Double i2 = 100.0;

i1==i2 为fanlse
```

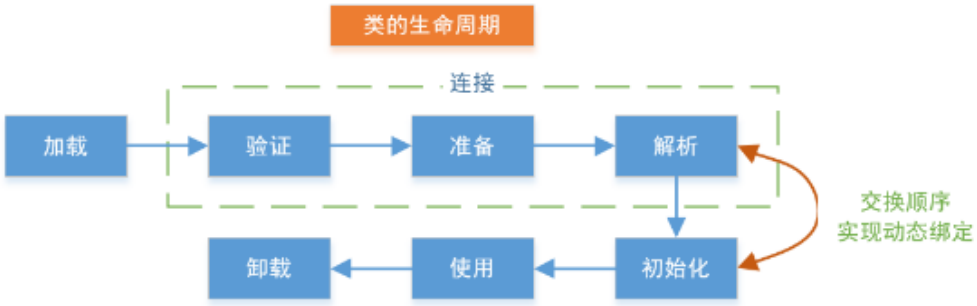
类加载发生的时机是什么时候？

- 1、遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类还没进行初始化，则需要先触发其初始化。
- 2、使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类还没进行初始化，则需要先触发其初始化。
- 3、当初始化了一个类的时候，如果发现其父类还没进行初始化，则需要先触发其父类的初始化。
- 4、当虚拟机启动时，用户需要指定一个执行的主类，即调用其 #main(String[] args) 方法，虚拟机则会先初始化该主类。
- 5、当使用 JDK7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

10、讲讲类加载机制呗都有哪些类加载器，这些类加载器都加载哪些文件?

负责读取 Java 字节代码，并转换成java.lang.Class类的一个实例；

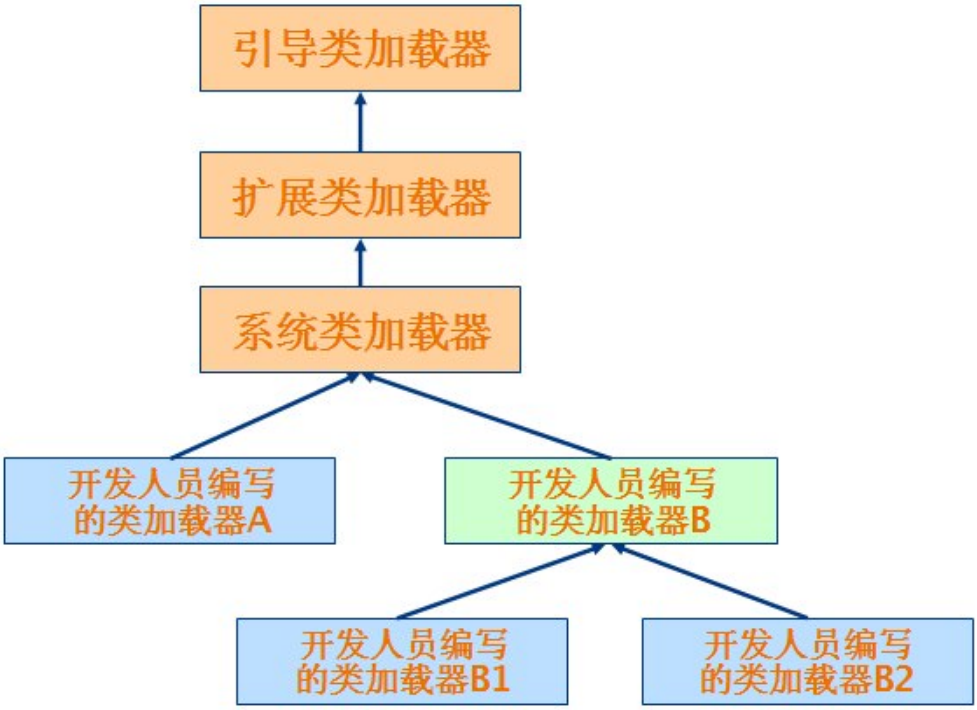
类加载步骤：



类加载顺序从上至下，如下面这题 new Singleton2();后调用构造器，value1=value2=1； 但是后面马上就初始化value2=0，所以最后的结果是value1=1，value2=0；

```
private static Singleton2 singleton2 = new Singleton2();
public static int value1;
public static int value2 = 0;


private Singleton2(){
    value1++;
    value2++;
}
```



- 1)引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader ,它是所有类加载器的父加载器。如果你调用String.class.getClassLoader()，会返回null，任何基于此的代码会抛出NUllPointerException异常。Bootstrap加载器被称为初始类加载器
- 2)扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类,Extension将加载类的请求先委托给它的父加载器，也就是Bootstrap，如果没有成功加载的话，再从jre/lib/ext目录下或者java.ext.dirs系统属性定义的目录下加载类。Extension加载器由sun.misc.Launcher$ExtClassLoader实现
- 3)系统类加载器（system class loader）：它根据Java应用的类路径（CLASSPATH）来加载Java类。一般来说，Java 应用的类都是由它来完成加载的。可以通过ClassLoader.getSystemClassLoader()来获取它,它负责从classpath环境变量中加载某些应用相关的类，classpath环境变量通常由-classpath或-cp命令行选项来定义，或者是JAR中的Manifest的classpath属性。Application类加载器是Extension类加载器的子加载器。通过sun.misc.Launcher$AppClassLoader实现

双亲委派模型，为什么要实现双亲委派模型

所谓双亲委派是指每次收到类加载请求时，先将请求委派给父类加载器完成（所有加载请求最终会委派到顶层的Bootstrap ClassLoader加载器中），如果父类加载器无法完成这个加载（该加载器的搜索范围中没有找到对应的类），子类尝试自己加载。

- 好处：
  - 1、共享功能：可以避免重复加载，当父亲已经加载了该类的时候，子类不需要再次加载，一些 Framework层级的类一旦被顶层的 ClassLoader 加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。
  - 2、隔离功能：主要是为了安全性，避免用户自己编写的类动态替换 Java 的一些核心类，比如String，同时也避免了重复加载，因为 JVM 中区分不同类，不仅仅是根据类名，相同的 class 文件被不同的ClassLoader加载就是不同的两个类，如果相互转型的话会抛 java.lang.ClassCaseException。（这也就是说，即使我们自己定义了一个java.util.String类，也不会被重复加载。）

双亲委派模型的工作过程？

- 1、当前 ClassLoader 首先从自己已经加载的类中，查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。
  - 每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。
- 2、当前 ClassLoader 的缓存中没有找到被加载的类的时候
  - 委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到 bootstrap ClassLoader。
  - 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

Java 虚拟机是如何判定两个 Java 类是相同的？

Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。

比如一个 Java 类 com.example.Sample ，编译之后生成了字节代码文件 Sample.class 。两个不同的类加载器 ClassLoaderA 和 ClassLoaderB 分别读取了这个 Sample.class 文件，并定义出两个 java.lang.Class 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException 。

手写一下类加载Demo

要创建用户自己的类加载器，只需要继承java.lang.ClassLoader类，然后覆盖它的findClass(String name)方法即可，即指明如何获取类的字节码流，getData()就是通过一个类名找到类读取流转换成byte返回。

```
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    byte[] classData = getData(name);
    if (classData == null) {
        throw new ClassNotFoundException();
    } else {
        return defineClass(name, classData, 0, classData.length);
    }
}
```

知道osgi是如何实现的?

OSGi™是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式用来管理软件的生命周期。OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 就是基于 OSGi 技术来构建的。

OSGi 中的每个模块（bundle）都包含 Java 包和类。模块可以声明它所依赖的需要导入（import）的其它模块的 Java 包和类（通过 Import-Package），也可以声明导出（export）自己的包和类，供其它模块使用（通过 Export-Package）。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。这是通过 OSGi 特有的类加载器机制来实现的。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。当它需要加载 Java 核心库的类时（以 java开头的包和类），它会代理给父类加载器（通常是启动类加载器）来完成。当它需要加载所导入的 Java 类时，它会代理给导出此 Java 类的模块来完成加载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性 org.osgi.framework.bootdelegation的值即可。

classforName("java.lang.String")和String classgetClassLoader() LoadClass("java.lang.String")什么区别啊?

1.Class.forName返回的Class对象可以决定是否初始化。而ClassLoader.loadClass返回的类型绝对不会初始化，最多只会做连接操作。
2.Class.forName可以决定由哪个classLoader来请求这个类型。而ClassLoader.loadClass是用当前的classLoader去请求。

cgLib知道吗?他和jdk动态代理什么区别?手写一个jdk动态代理呗?

cgLib是动态代理技术。 java的单继承，JDK代理类已经继承了Proxy类，所以必须要实现接口

能否在加载类的时候，对类的字节码进行修改

我们利用javaAgent和ASM字节码技术，在JVM加载class二进制文件的时候，利用ASM动态的修改加载的class文件，在监控的方法前后添加计时器功能，用于计算监控方法耗时，同时将方法耗时及内部调用情况放入处理器，处理器利用栈先进后出的特点对方法调用先后顺序做处理，当一个请求处理结束后，将耗时方法轨迹和入参map输出到文件中，然后根据map中相应参数或耗时方法轨迹中的关键代码区分出我们要抓取的耗时业务。最后将相应耗时轨迹文件取下来，转化为xml格式并进行解析，通过浏览器将代码分层结构展示出来，方便耗时分析

内存泄漏与内存溢出的区别

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。

19、参考资料

> 《java内存模型以及happens-before规则》
> 《内存分配与回收策略》
> 《类加载机制》
> 《垃圾回收》
> 《Java 面试知识点解析(三)——JVM篇》
> 《总结的 JVM 面试题》
> 《JAVA 对象创建的过程》
> 《Java 虚拟机详解——JVM常见问题总结》