JVM：



方法区：类的信息 静态变量 常量池

堆：对象实例

Jvm栈：方法出参入参引用变量 引用对象地址

本地方法栈：JNI OOM

程序计数器：代码执行指令地址

GC

堆分为新生代(1)、老年代(2)、元空间

GCRoots:

1. 被synchronized持有的对象
2. 方法区中的常量池 静态变量对象
3. Jvm栈中本地变量引用的对象
4. 本地方法栈中的JNI引用的对象 OOM异常对象等

可达性分析：

三色分析法 黑色：已遍历，灰色：遍历中，白色：未遍历

通过写屏障对节点的保护 避免了 黑色对象引用白色对象时 遗漏的问题，

写屏障：就是一个循坏切面 在黑色对象引用白色对象时 将白色对象置为灰色

如何保证分代GC的边界问题：

在YGC中 jvm调用了do\_oop\_work()方法，使用boundary对对象的地址进行边界判定，从而使得YGC时只操作新生代对象

新生代垃圾回收过程：

新生代：Eden from to 8:1:1

由于新生代对象存活率较低，所以选择复制的方法进行垃圾回收。

当Eden区满的时候，触发mionrGC

1. 遍历GCroots+cardtable，将得到的对象放入to区
2. 根据to区中的对象，利用可达性分析法迭代对象图，得到的存活对象放入TO区，在to区中有两个指针用来标记迭代进度，在迭代过程中如果TO区未满，那么两个指针相遇既代表遍历完成，如果TO区已满，那么将剩余的对象直接放入老年代。(直接放入老年代，会导致新的跨带引用的出现，同时会触发cardtable的更新，带来新的开销。解决方法：在程序设计的时候：在GC之前根据Eden的存活率计算所需空间的大小，合理设置from与to区的大小，规避直接晋升情况的发生)
3. 跨带引用(老年代中引用新生代对象)：jvm为了提升扫描效率引入了generation remember set的概念，将老年代分为多个小的空间，用dirty与clean标记是否被引用。具体通过cardtable实现，通过写屏障在cardtable中存放被引用对象的地址
4. 清空Eden 与from 区，交换from与to空间

老年代垃圾回收过程：

触发：当老年代满的时候或者放入大对象之前触发majorGC

由于老年代存活率较高 所以选用标记清除的方法CMS

1. 初始标记--STW
   1. 对GCroots +老年代可达对象
   2. 新生代的对象存活率低 变化快，所以老年代对新生代的遍历没有采取分块标记的策略，
2. 并发标记
   1. 根据GCROOT得到的节点，对对象图遍历，在这个过程中由于没有STW 所以应用服务的进程依然会对新生代和老年代的对象引用进行操作，老年代对象的变更通过预处理放到moduniontable中(功能与cardtable类似)
3. 重新标记(扫描对象新增新生代对象)—STW（由于重新标记是STW的 所以为了减少STW的时间 可以在重新标记之前再次触发一次YGC 可以设置参数CMSscavergBeforeRemark）
   1. 遍历新生代 重新标记
   2. 遍历GCROOTS 重新标记
   3. 遍历老年代 moduniontable 重新标记
4. 并发清理
   1. 清空老年代工作区
5. 待续

G1垃圾回收机制：与CMS类似

1. 初始标记—STW
   1. 扫描根集合 使用外部bitmap记录标记信息
2. 并发标记
   1. 从扫描栈取出引用 递归扫描整个堆对象，每扫描一个就对其标记 并将字段压入扫描栈 重复此过程 直到栈为空,扫描STAB write barrier记录下的引用
3. 最终标记--STW
   1. 处理剩下的SATB write barrier(最开始的快照)记录下的引用(控制并发标记：GC开始的时候 活的对象就被认为是活的，在GC过程中新分配的对象也是活的(放入pre-write-barrier中)，避免了像CMS那样 在重新标记阶段进行全部扫描；同样的这样也产生了更多的浮动垃圾)
4. 清理—STW
   1. 清点和重置标记状态。不是在堆上sweep实际对象，而是在markingbitmap中统计每个region被标记为活的对象有多少 如果发现某个region没有活对象 就直接回收
   2. 拷贝活对象：把活对象拷贝到空的region中 然后回收原本的region （CSset：部分收集：任意多个region来独立收集 构成收集集合），有了CSset 使用复制算法进行即可

清理模式：YGC 与mixedGC

YGC：选定所有young gen里的region 通过控制region个数来控制YGC的开销

Mixed GC:选定所有young gen里的region 外加global concurrent marking统计得到的收益高的若干 old gen region，在用户指定的开销范围内尽可能的选择收益高的OGR

G1对write barrier的优化 ：通过信息标记log 使得单线程执行变成了并行执行，提高吞吐率

ZGC垃圾回收机制：

将内存分成大小不同的块，采用着色指针 读屏障 java14中还没有定型

类加载：

<https://blog.csdn.net/qq_35114086/article/details/80371859>

强引用：最普通的引用 不会被GC回收 内存不足时OOM

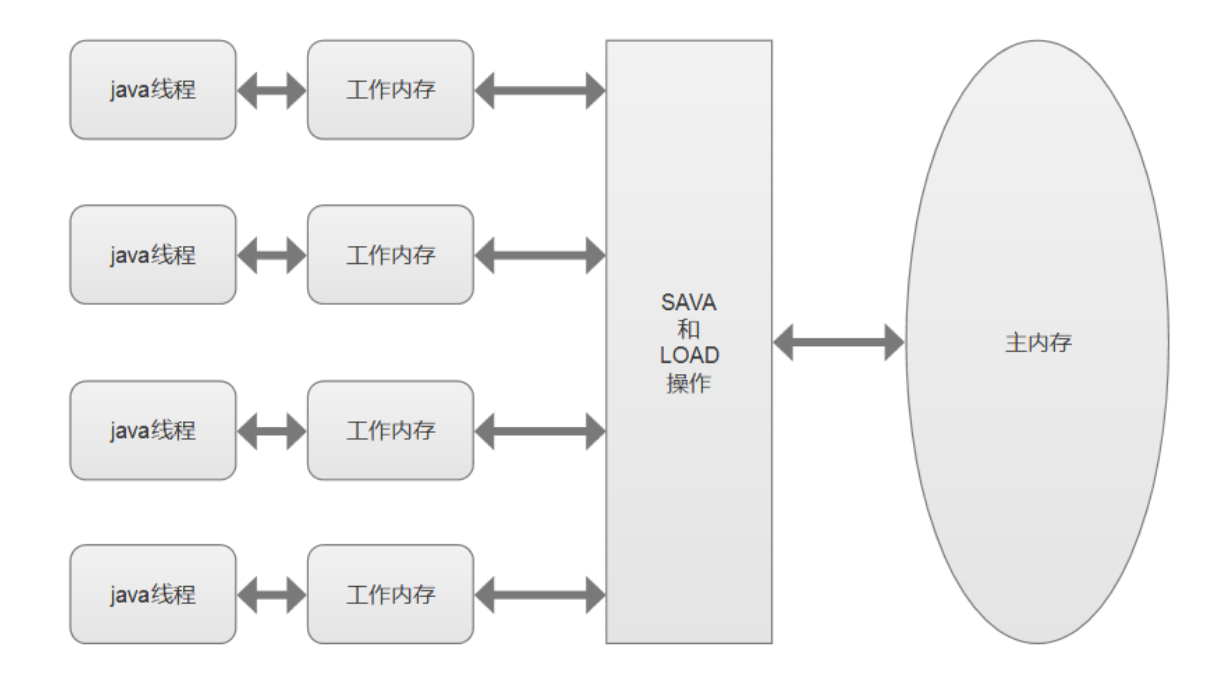
软引用：内存空间充足时，GC不会回收它；如果内存空间不足了，就会回收，在抛出OOM之前回收(ReferenceQueue：保证优先最近最少使用对象) 对象缓存

(String str = new String("abc");)

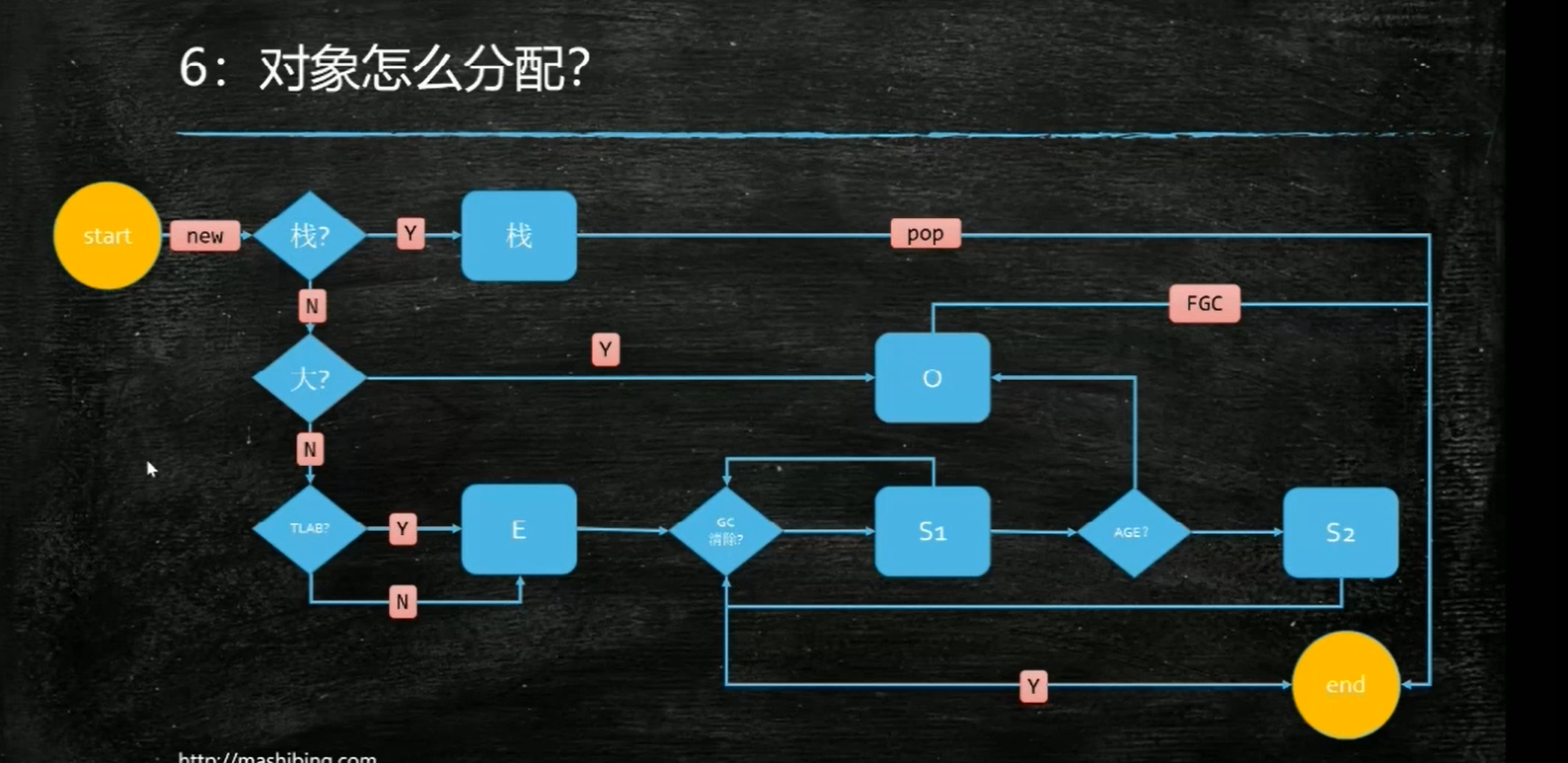
弱引用：弱引用的对象，不管当前内存空间足够与否，都会回收，但不一定被及时发现； 对象缓存

虚引用：虚引用主要用来跟踪对象被垃圾回收器回收的活动。

JMM



1. 对象的创建过程
   1. 开辟空间 默认值
   2. 赋值
   3. 指向引用地址
2. DCL(double check lock)与volatile的问题
   1. 单例中必须用vola 禁止指令重排 才能安全
3. 对象在内存中的存储布局
   1. Markworld
   2. ClassPointer
   3. 实例数据
   4. 对齐
4. 对象头具体包括什么（markworld+classpointer）
   1. Sync锁信息
5. 对象怎么定位
   1. 直接指针
   2. 句柄方式
6. 对象怎么分配



首先在栈上分配 栈帧失效得同时对象也清理掉 不需要GC 所以性能好

能在栈上分配需要具备1、逃逸分析2、标量替换

不能分配栈的 如果比较大 直接进入老年代

比较小的进入伊甸园 这时候 有一个TLAB(本地线程缓存)这里没有锁 性能好

1. Object o=new Object() 在内存中占有多少字节

CPU利用率过高 排查：

1. top –c 查看占用CPU高的进程
2. top –p pid H 查看进程的线程信息
3. 记录线程ID 同时用jstack pid 得到堆栈信息
4. 将线程ID转为16进制 然后在堆栈信息中搜索nid 如果堆栈信息里没有具体异常信息 那通常就是内存的问题了
5. 使用 jmap –histo:live pid 看一下内存存活对象统计情况 能够具体到类名

JVM类加载机制

数据从class文件加载到内存 并对数据进行校验、转换解析和初始化 最终形成可以被虚拟机直接使用的java类型

加载->验证->准备->解析->初始化->使用->卸载 (连接)

加载、验证、准备、初始化、卸载这5个步骤的顺序是确定的

解析在某些情况下可以在初始化阶段之后在开始 这是为了支持java语言的运行时绑定

加载：虚拟机完成3件事

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3）在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

验证：

验证是连接阶段的第一步，这一步主要的目的是确保class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。

验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1.文件格式验证

验证class文件格式规范，例如： class文件是否已魔术0xCAFEBABE开头 ， 主、次版本号是否在当前虚拟机处理范围之内等

2.元数据验证

这个阶段是对字节码描述的信息进行语义分析，以保证起描述的信息符合java语言规范要求。验证点可能包括：这个类是否有父类(除了java.lang.Object之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被final修饰的)、如果这个类的父类是抽象类，是否实现了起父类或接口中要求实现的所有方法。

3.字节码验证

进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

4.符号引用验证

准备：

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(static 修饰的变量),而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在java堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value = 12;

那么变量value在准备阶段过后的初始值为0而不是12，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器()方法之中，所以把value赋值为12的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，建设上面类变量value定义为：

public static final int value = 123;

编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value设置为123

解析

解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。

符号引用：符号引用是一组符号来描述所引用的目标对象，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

初始化

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器< clinit >()方法的过程。在以下四种情况下初始化过程会被触发执行：

1.遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需先触发其初始化。生成这4条指令的最常见的java代码场景是：使用new关键字实例化对象、读取或设置一个类的静态字段(被final修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用类的静态方法的时候。

2.使用java.lang.reflect包的方法对类进行反射调用的时候

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先出发其父类的初始化

4.jvm启动时，用户指定一个执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value = 12; 在准备阶段完成后 value的值为0，而在初始化阶调用了类构造器< clinit >()方法，这个阶段完成后value的值为12。

\*类构造器< clinit >()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句快可以赋值，但是不能访问。

\*类构造器< clinit >()方法与类的构造函数(实例构造函数< init >()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类< clinit >()方法执行之前，父类的< clinit >()方法已经执行完毕。因此在虚拟机中的第一个执行的< clinit >()方法的类肯定是java.lang.Object。

\*由于父类的< clinit >()方法先执行，也就意味着父类中定义的静态语句快要优先于子类的变量赋值操作。

\*< clinit >()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，那么编译器可以不为这个类生成< clinit >()方法。

\*接口中不能使用静态语句块，但接口与类不太能够的是，执行接口的< clinit >()方法不需要先执行父接口的< clinit >()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的< 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执行这个类的< clinit >()方法，其他线程都需要阻塞等待，直到活动线程执行< clinit >()方法完毕。如果一个类的< clinit >()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

类加载器

虚拟机设计团队把类加载阶段中的“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

1、类与类加载器

对于任何一个类，都需要由加载它的类加载器和这个类来确立其在JVM中的唯一性。也就是说，两个类来源于同一个Class文件，并且被同一个类加载器加载，这两个类才相等。

2、双亲委派模型

从虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），该类加载器使用C++语言实现，属于虚拟机自身的一部分。另外一种就是所有其它的类加载器，这些类加载器是由Java语言实现，独立于JVM外部，并且全部继承自抽象类java.lang.ClassLoader。

从Java开发人员的角度来看，大部分Java程序一般会使用到以下三种系统提供的类加载器：

1)启动类加载器（Bootstrap ClassLoader）：负责加载JAVA\_HOME\lib目录中并且能被虚拟机识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。该类加载器无法被Java程序直接引用。

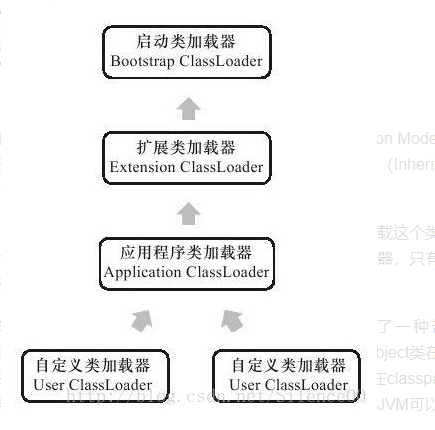
2)扩展类加载器（Extension ClassLoader）：该加载器主要是负责加载JAVA\_HOME\lib\，该加载器可以被开发者直接使用。

3)应用程序类加载器（Application ClassLoader）：该类加载器也称为系统类加载器，它负责加载用户类路径（Classpath）上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这三类加载器互相配合进行加载的。

另外还有自定义类加载器。

4)自定义类加载器(必须继承 ClassLoader)。



如上图所示的类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parent Delegation Model）。该模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。

双亲委派模型的工作过程为：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。

使用这种模型来组织类加载器之间的关系的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如java.lang.Object类，无论哪个类加载器去加载该类，最终都是由启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。否则的话，如果不使用该模型的话，如果用户自定义一个java.lang.Object类且存放在classpath中，那么系统中将会出现多个Object类，应用程序也会变得很混乱。如果我们自定义一个rt.jar中已有类的同名Java类，会发现JVM可以正常编译，但该类永远无法被加载运行。