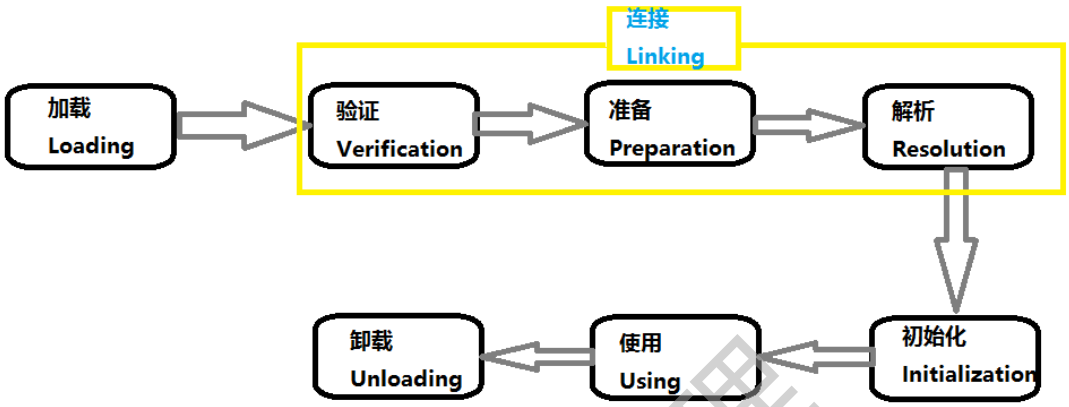
JVM面试题

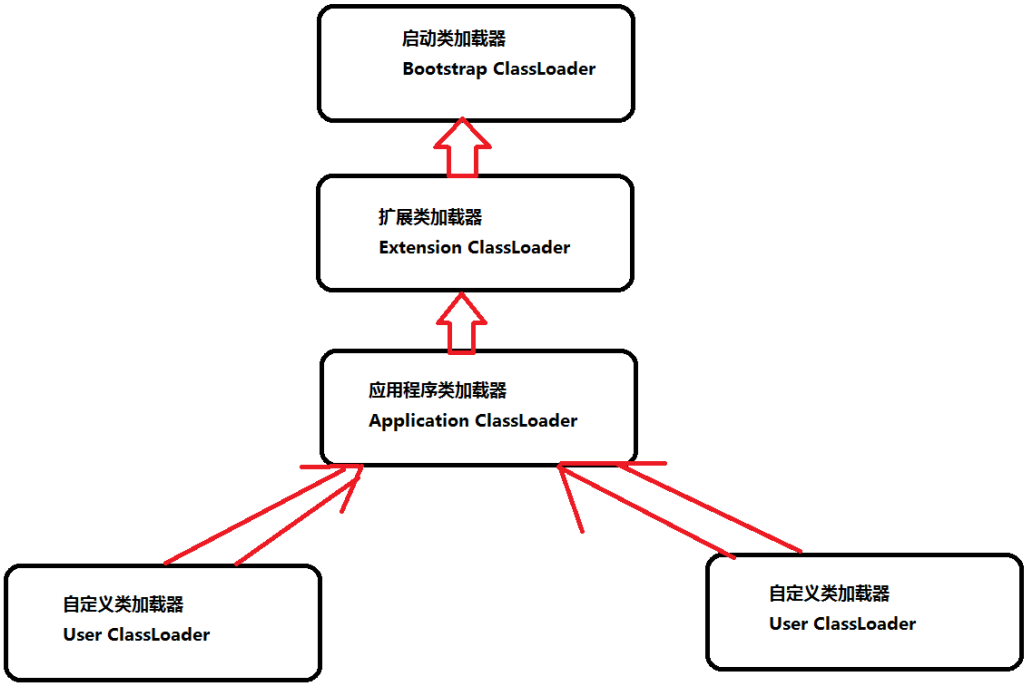
ClassLoader类加载器

类加载过程



- **加载**
- 将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在内存上创建一个 java.lang.Class 对象用来封装类在方法区内的数据结构作为这个类的各种数据的访问入口。
- **验证**
- 主要是为了确保class文件中的字节流包含的信息是否符合当前JVM的要求，且不会危害JVM自身安全，比如校验文件格式、是否是cafe baby魔数、字节码验证等等。
- **准备**
- 为类变量分配内存并设置类变量（是被static修饰的变量，变量不是常量，所以不是final的，就是static的）初始值的阶段。这些变量所使用的内存在方法区中进行分配。比如 private static int age = 26; 类变量age会在准备阶段过后为 其分配四个（int四个字节）字节的空间，并且设置初始值为0，而不是26。
- 若是final的，则在编译期就会设置上最终值。
- **解析**
- JVM会在此阶段把类的二进制数据中的符号引用替换为直接引用。
- **初始化**
- 初始化阶段是执行类构造器 () 方法的过程，到了初始化阶段，才真正开始执行类定义的Java程序代码（或者说字节码 ）。比如准备阶段的那个age初始值是0，到这一步就设置为26。
- **使用**
- 对象都出来了，业务系统直接调用阶段。
- **卸载**
- 用完了，可以被GC回收了。

类加载器种类以及加载范围

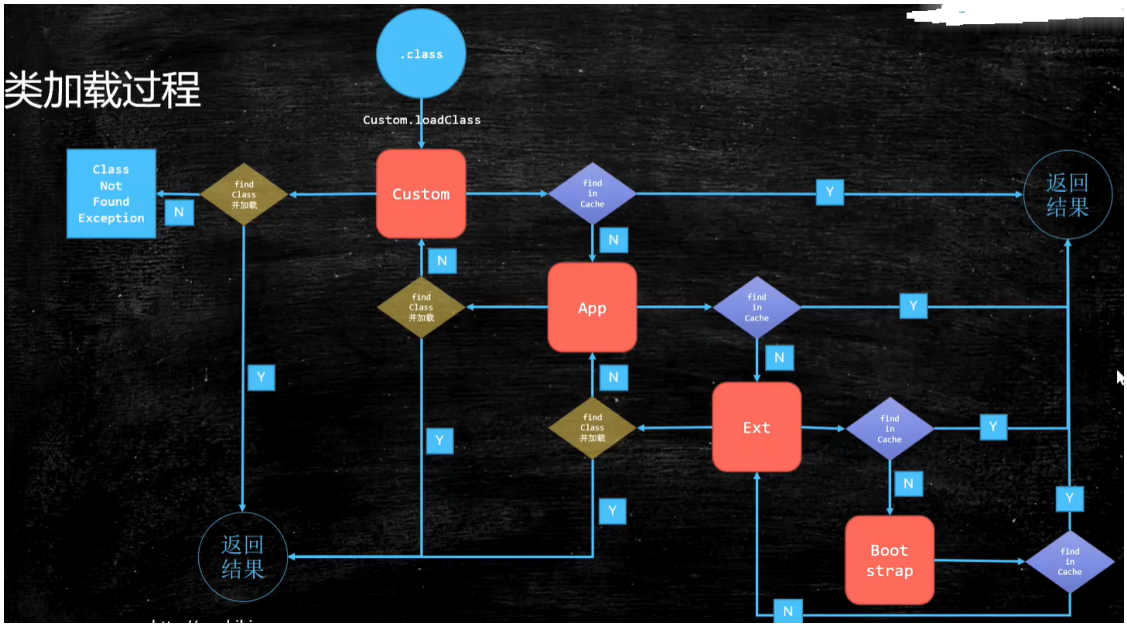


- **启动类加载器（Bootstrap ClassLoader）**
- 最顶层类加载器，他的父类加载器是个null，也就是没有父类加载器。负责加载jvm的核心类库，比如 java.lang.* 等，从系统属性中的sun.boot.class.path 所指定的目录中加载类库。他的具体实现由Java虚拟机底层C++代码实现。
- **扩展类加载器（Extension ClassLoader）**
- 父类加载器是Bootstrap ClassLoader。从 java.ext.dirs 系统属性所指定的目录中加载类库，或者从JDK的安装目录的 jre/lib/ext 子目录（扩展目录）下加载类库，如果把用户的jar文件放在这个目录下，也会自动由扩展类加载器加载。继承自 java.lang.ClassLoader 。
- **应用程序类加载器（Application ClassLoader）**
- 父类加载器是Extension ClassLoader。从环境变量classpath或者系统属性 java.class.path 所指定的目录中加载类。继承自 java.lang.ClassLoader 。
- **自定义类加载器（User ClassLoader）**

  除了上面三个自带的以外，用户还能制定自己的类加载器，但是所有自定义的类加载器都应该继承自 java.lang.ClassLoader 。比如热部署、tomcat都会用到自定义类加载器。

双亲委派是什么

- 如果一个类加载器收到了类加载的请求，他首先会从自己缓存里查找是否之前加载过这个class，加载过 直接返回，没加载过的话他不会自己亲自去加载，他会把这个请求委派给父类加载器去完成，每一层都是如此，类似递归，一直递归到顶层父类，也就是 Bootstrap ClassLoader ，只要加载完成就会返回结果，如果顶层父类加载器无法加载此class，则会返回去交给子类加载器去尝试加载，若最底层的子类加载器也没找到，则会抛出 ClassNotFoundException 。

- 

为啥要有双亲委派

- 防止内存中出现多份同样的字节码，安全。
- 比如自己重写个 java.lang.Object 并放到Classpath中，没有双亲委派的话直接自己执行了，那不安 全。双亲委派可以保证这个类只能被顶层 Bootstrap Classloader 类加载器加载，从而确保只有JVM中有且仅有一份正常的java核心类。如果有多个的话，那么就乱套了。比如相同的类 instance of 可能返回false，因为可能父类不是同一个类加载器加载的Object。

为什么需要破坏双亲委派模型

- **Jdbc**
- Jdbc为什么要破坏双亲委派模型?
- 以前的用法是未破坏双亲委派模型的，比如 Class.forName("com.mysql.cj.jdbc.Driver");
- 而在JDBC4.0以后，开始支持使用spi的方式来注册这个Driver，具体做法就是在mysql的jar包中的 META-INF/services/java.sql.Driver 文件中指明当前使用的Driver是哪个，然后使用的时候就不需要我们手动的去加载驱动了，我们只需要直接获取连接就可以了。 Connection con = DriverManager.getConnection(url, username, password);
- 首先，理解一下为什么JDBC需要破坏双亲委派模式，原因是原生的JDBC中Driver驱动本身只是一个接口，并没有具体的实现，具体的实现是由不同数据库类型去实现的。例如，MySQL的 mysql-connector-*.jar 中的Driver类具体实现的。 原生的JDBC中的类是放在 rt.jar 包的，是由 Bootstrap 加载器进行类加载的，在JDBC中的Driver类中需要动态去加载不同数据库类型的Driver类，而 mysql-connector-*.jar 中的Driver类是用户自己写的代码，那Bootstrap类加载器肯定是不能进行加载的，既然是自己编写的代码，那就需要由Application类加载器去进行类加载。这个时候就引入线程上下文件类加载器(Thread Context ClassLoader)，通过这个东西程序就可以把原本需要由Bootstrap类加载器进行加载的类由Application类加载器去进行加载了
- Tomcat
- Tomcat为什么要破坏双亲委派模型?
- 因为一个Tomcat可以部署N个web应用，但是每个web应用都有自己的classloader，互不干扰。比如web1里面有 com.test.A.class ， web2里面也有 com.test.A.class ，如果没打破双亲委派模型的话，那么web1加载完后，web2在加载的话会冲突。因为只有一套classloader，却出现了两个重复的类路径，所以tomcat打破了，他是线程级别的，不同web应用是不同的classloader。
- Java spi 方式，比如jdbc4.0开始就是其中之一。
- 热部署的场景会破坏，否则实现不了热部署。

如何破坏双亲委派模型

- 重写 loadClass 方法，别重写 findClass 方法，因为 loadClass 是核心入口，将其重写成自定义逻辑即可破坏双亲委派模型。

如何自定义一个类加载器

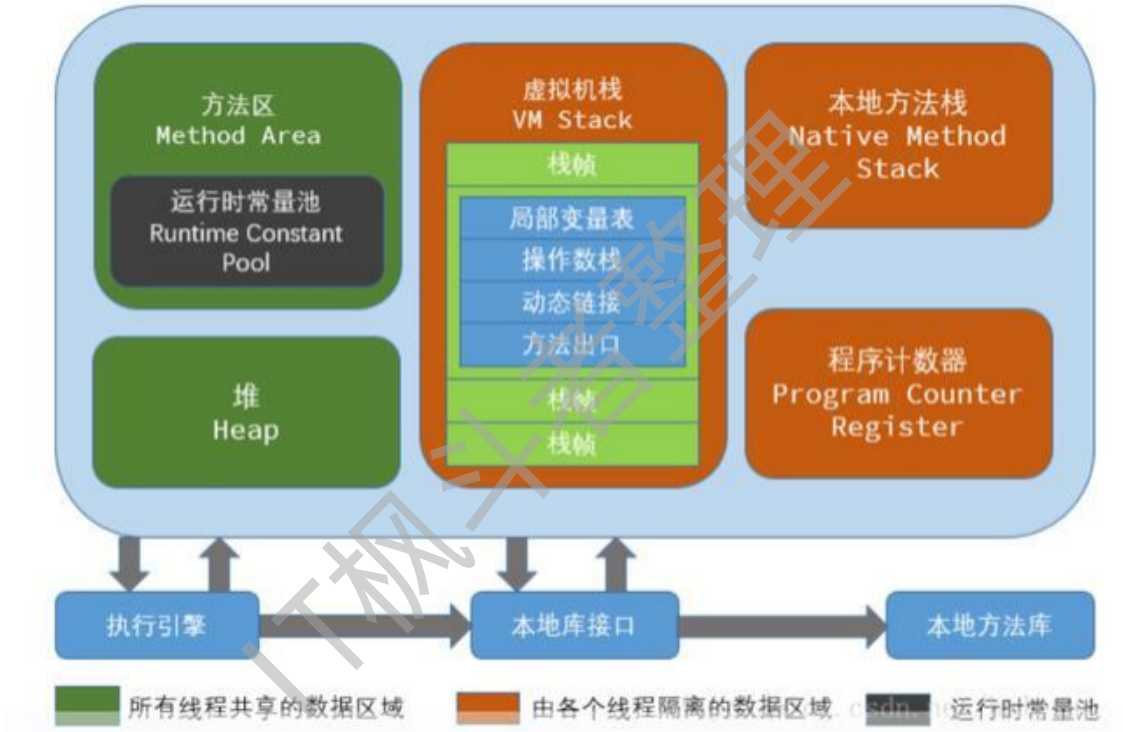
- 只需要继承 java.lang.Classloader 类，然后覆盖他的 findClass(String name) 方法即可，该方法根据参数指定的类名称，返回对应 的Class对象的引用。

热部署原理

- 采取破坏双亲委派模型的手段来实现热部署，默认的 loadClass() 方法先找缓存，你改了class字节码也不会热加载，所以自定义ClassLoader，去掉找缓存那部分，直接就去加载，也就是每次都重新加载。

Java内存区域

-



- **程序计数器**
- 当前线程所执行字节码的行号指示器。若当前方法是native的，那么程序计数器的值就是undefined。
- 线程私有，Java内存区域中唯一一块不会发生OOM或StackOverflow的区域。
- **虚拟机栈**
- 就是常说的Java栈，存放栈帧，栈帧里存放局部变量表等信息，方法执行到结束对应着一个栈帧的入栈到出栈。 线程私有，会发StackOverflow。
- **本地方法栈**
- 与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的。线程私有，会发生StackOverflow。
- **堆**
- Java 虚拟机中内存最大的一块，几乎所有的对象实例都在这里分配内存。 是被所有线程共享的，会发生OOM。
- **方法区**
- 也称非堆，用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。是被所有线程共享的，会发生OOM。
- **运行时常量**

- 是方法区的一部分，存常量（比如static final修饰的，比如String 一个字符串）和符号引用。是被所有线程共享的，会发生OOM。

对象怎么定位

- 如下两种，具体用哪种有JVM来选择，hotspot虚拟机采取的直接指针方式来定位对象。
- **直接指针**
- 栈上的引用直接指向堆中的对象。好处就是速度快。没额外开销。
- **句柄**
- Java堆中会单独划分出一块内存空间作为句柄池，这么一来栈上的引用存储的就是句柄地址，而不是真实对象地址，而句柄中包含了对象的实例数据等信息。好处就是即使对象在堆中的位置发生移动，栈上的引用也无需变化。因为中间有个句柄。

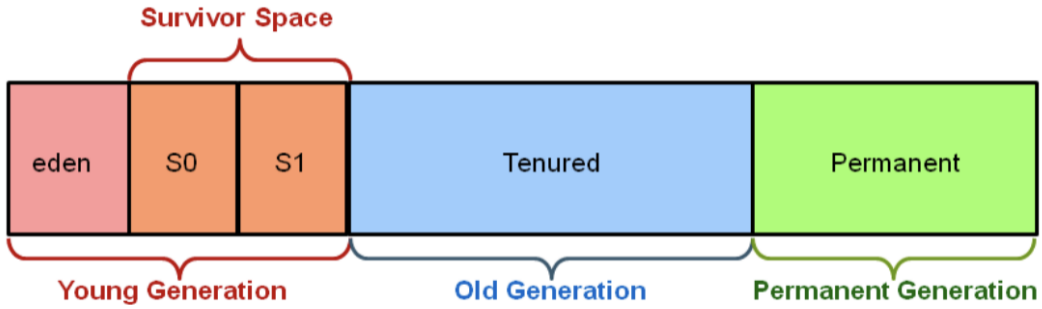
判断对象是否能被回收的算法

- **引用计数法**
- 给对象添加一个引用计数器，每当有一个地方引用他的时候该计数器的值就+1，当引用失效的时候该计数器的值就-1；当计数器的值为0的时候，jvm判定此对象为垃圾对象。存在内存泄漏的bug，比如循环引用的时候，所以jvm虚拟机采取的是可达性分析法。
- **可达性分析法**
- 有一些根节点GC Roots作为对象起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连的时候，则证明此对象为垃圾对象。
  - 补充：哪些可作为GC Roots?
  - 虚拟机栈中的引用的对象
  - 方法区中的类静态属性引用的对象
  - 方法区中常量引用的对象
  - 本地方法栈中JNI（native方法）引用的对象

如何判断对象是否能被回收

- 该对象没有与GC Roots相连
- 该对象没有重写finalize()方法或finalize()已经被执行过则直接回收（第一次标记）、否则将对象加入到F-Queue队列中（优先级很低的队列）在这里finalize()方法被执行，之后进行第二次记，如果对象仍然应该被GC则GC，否则移除队列。 （在finalize方法中，对象很可能和其他 GC Roots中的某一个对象建立了关联，那就自救了，就不会被GC掉了，finalize方法只会被调用一次，且不推荐使用finalize方法）

Java堆内存组成部分

- 

- 堆大小 = 新生代 + 老年代。如果是Java8则没有Permanent Generation，Java8将此区域换成了Metaspace。

- 其中新生代(Young) 被分为 Eden和S0（from)和S1(to)。
- 默认情况下Edem : from : to = 8 : 1 : 1 ， 此比例可以通过 –XX:SurvivorRatio 来设定

Java中会存在内存泄漏吗，请简单描述。

- 虽然Java会自动GC，但是使用不当的话还是存在内存泄漏的，比如ThreadLocal忘记remove的情况。

栈帧是什么？包含哪些东西

- 栈帧中存放的是局部变量、操作数栈、动态链接、方法出口等信息，栈帧中的局部变量表存放基本类型+对象引用+returnAddress，局部变量所需的内存空间在编译期间就完成分配了，因为基本类型和对象引用等都能确定占用多少slot，在运行期间也是无法改变这个大小的。

简述一个方法的执行流程

- 方法的执行到结束其实就是栈帧的入栈到出栈的过程，方法的局部变量会存到栈帧中的局部变量表里，递归的话会一直压栈压栈，执行完后进行出栈，所以效率较低，因为一直在压栈，栈是有深度的。

方法区会被回收吗

- 方法区回收价值很低，主要回收废弃的常量和无用的类。
- 如何判断无用的类：
  - 该类所有实例都被回收（Java堆中没有该类的对象）
  - 加载该类的ClassLoader已经被回收
  - 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方利用反射访问该类

一个对象包含多少个字节

- 会占用16个字节。

为什么把堆栈分成两个

- 栈代表了处理逻辑，堆代表了存储数据，分开后逻辑更清晰，面向对象模块化思想。栈是线程私有，堆是线程共享区，这样分开也节省了空间，比如多个栈中的地址指向同一块堆内存 中的对象。
- 栈是运行时的需要，比如方法执行到结束，栈只能向上增长，因此会限制住栈存储内容的能力，而堆中的对象是可以根据需要动态增长的。

栈的起始点是哪

- main函数，也是程序的起始点。

为什么基本类型不放在堆里

- 因为基本类型占用的空间一般都是1-8个字节（所需空间很少），而且因为是基本类型，所以不会出现动态增长的情况（长度是固定的），所以存到栈上是比较合适的。反而存到可动态增长的堆上意义不大。

Java参数传递是值传递还是引用传递

- 值传递。

- 基本类型作为参数被传递时肯定是值传递；引用类型作为参数被传递时也是值传递，只不过“值”为对应的引用。假设方法参数是个对象引用，当进入被调用方法的时候，被传递的这个引用的值会被程序解释到堆中的对象，这个时候才对应到真正的对象，若此时进行修改，修改的是引用对应的对象，而不是引用本身，也就是说修改的是堆中的数据，而不是栈中的引用。

为什么不推荐递归

- 因为递归一直在入栈入栈，短时间无法出栈，导致栈的压力会很大，栈也有深度的，容易爆掉，所以效率低下。

为什么参数大于2个要放到对象里

- 因为除了double和long类型占用局部变量表2个slot外，其他类型都占用1个slot大小，如果参数太多的话会导致这个栈帧变大，因为slot大，放个对象的引用上去的话只会占用1个slot，增加堆的压力减少栈的压力，堆自带GC，所以这点压力可以忽略。

为什么新生代要分为eden、s1、s2三块，且默认比例是8:1:1?

- 因为复制算法，如果不分这些的话，那么比如1G内存，将浪费500MB，因为一分为二，其中500MB浪费掉。分为这三块的话，默认8:1:1的比例来看的话就是eden800M，s1和s2各100M，这就相当于有900MB可用，大大提升了内存利用率。复制算法特性还不会产生碎片。

GC垃圾回收

GC是什么？为什么要GC

- GC：垃圾收集，GC能帮助我们释放jvm内存，可以一定程度避免OOM问题，但是也无法完全避免。Java的GC是自动工作的，不像C++需要主动调用。当new对象的时候，GC就开始监控这个对象的地址大小和使用情况了，通过可达性分析算法寻找不可达的对象然后进行标记看看是否需要GC回收掉释放内 存。

你能保证GC执行吗?

- 不能，我只能通过手动执行 System.gc() 方法通知GC执行，但是他是否执行的未知的。

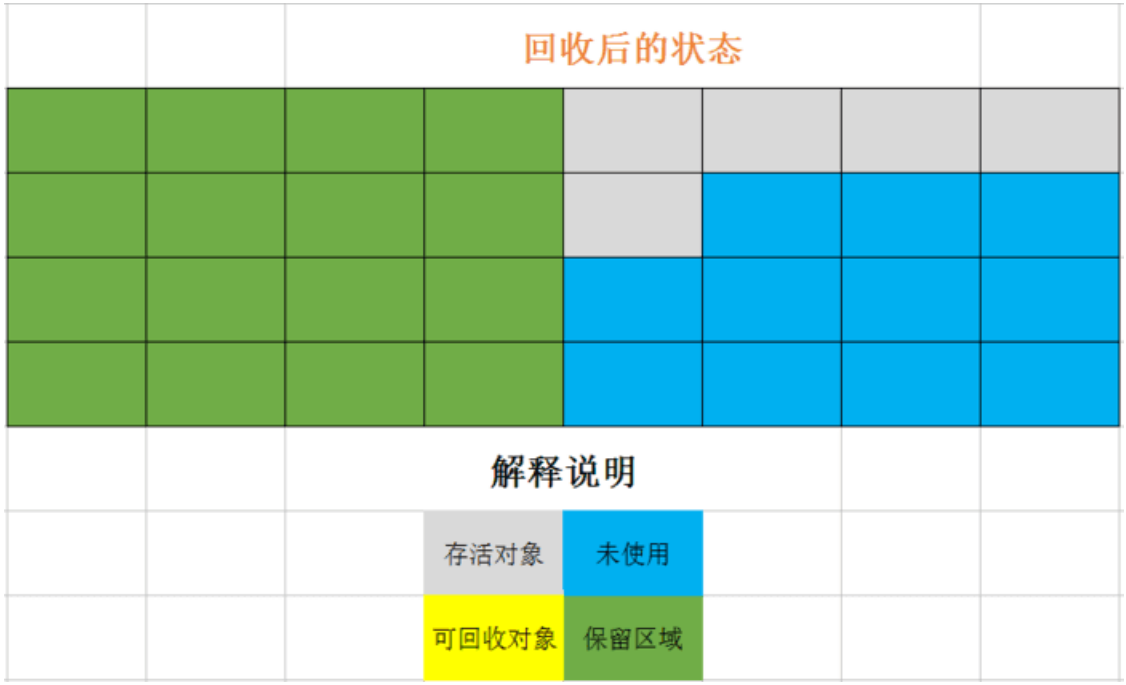
垃圾收集算法有哪些

- **标记清除**
- 分为两步：标记和清除。
- 首先需要标记出所有需要回收的对象，然后进行清除回收变为可用内存。
- 缺点：效率低，会产生垃圾碎片 。

- 

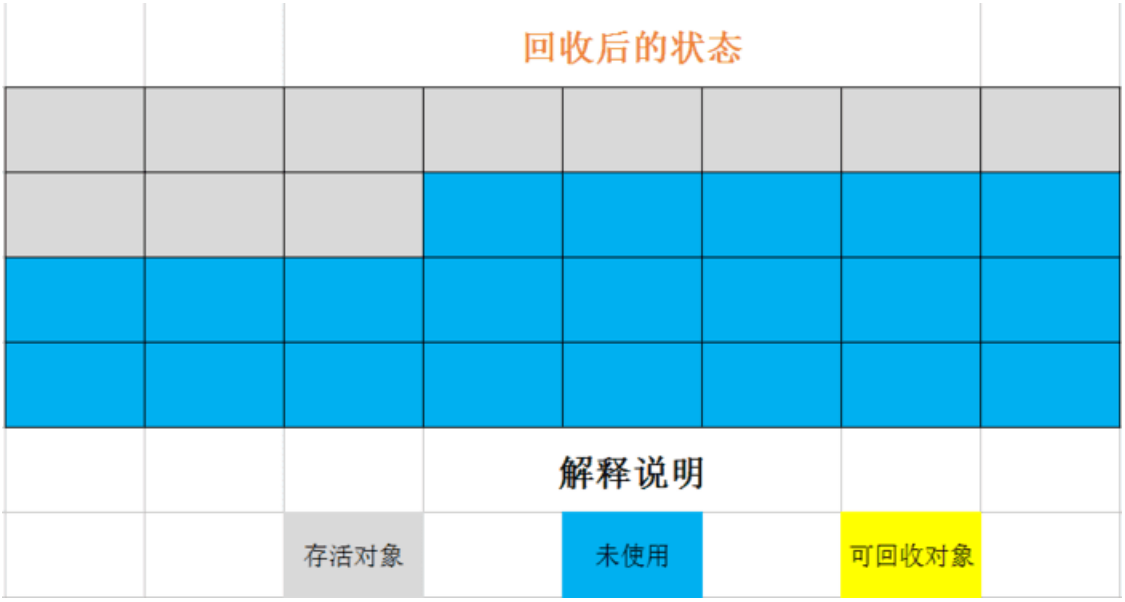
- **复制算法**
- 将可用堆内存按照容量分为大小相等的两块，每次只用一块，当这块内存快用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存一次清理掉。
- 年轻代from/to（s1/s2）采取的就是此种算法。老年代一般不会采取此种算法，因为老年代都是大对象且存活的久的，空间压缩一半代价略高。
- 优点：效率较高、不会产生碎片。
- 缺点：将内存缩小为原来的一半，代价略高。

- 

- 

- **标记整理**
- 分为两步：标记和整理。
- 整理其实也是两步：整理+清除。
- 整理让所有存活的对象都移动到一端，然后清理掉边界以外的内存。
- 优点：不会产生碎片问题，适合年老代的大对象存储，不像复制算法那样浪费空间。
- 缺点：效率赶不上复制算法。

- 

- 

- **分代算法**
- 并不是新算法，而是根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

为什么要分代

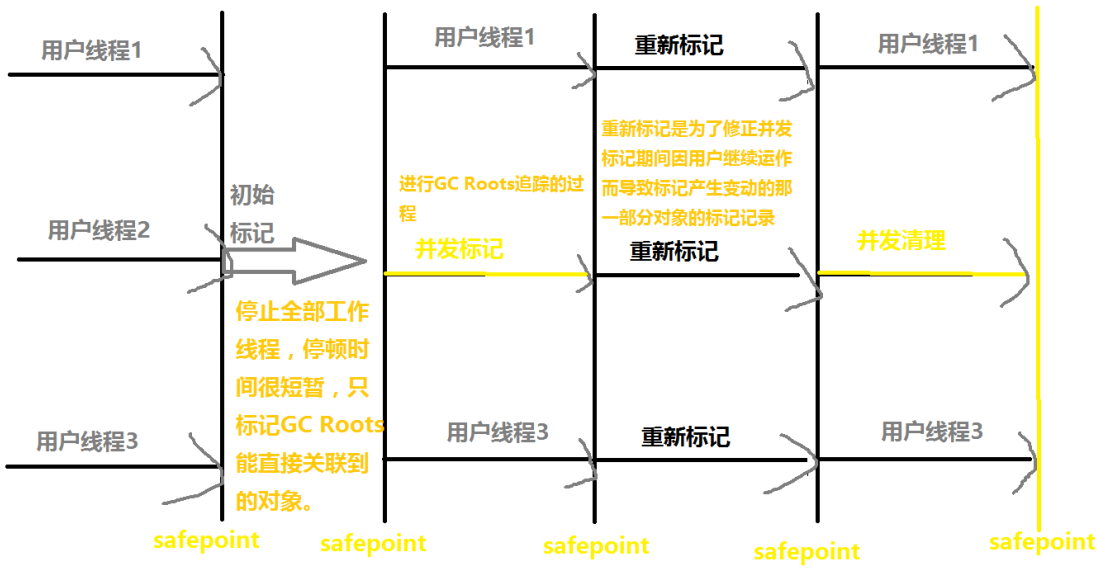
- 因为在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间会相对 较长，也有很多对象完全没必要遍历，比如大对象存活的时间更长，遍历下来发现不需要回收，这样更浪费时间。所以才有了分代，分治的思想，进行区域划分，把不同生命周期的对象放在不同的区域，不 同的区域采取最适合他的垃圾回收方式进行回收。

分代垃圾回收是怎么工作的

- 分代回收基于这样一个理念：不同的对象的生命周期是不一样的，因此根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。这样来提高回收效率。
- 新生代执行流程：
  - 把 Eden + From Survivor（S1） 存活的对象放入 To Survivor（S2） 区；
  - 清空 Eden 和S1 区；
  - S1 和 S2 区交换，S1 变 S2，S2变S1。
- 每次在S1到S2移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老年代。 大对象也会直接进入老年代。
- 老年代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。

详细介绍一下 CMS 垃圾回收器?

- 采取标记清除算法，老年代并行收集器，号称以最短STW时间为目标的收集器，并发高、停顿低、STW时间短的优点。主流垃圾收集器之一。
- 主要分为四阶段：
  - 初始标记：只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。所以此阶段会STW，但时间很短。
  - 并发标记：进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。不会STW。
  - 重新标记：为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的

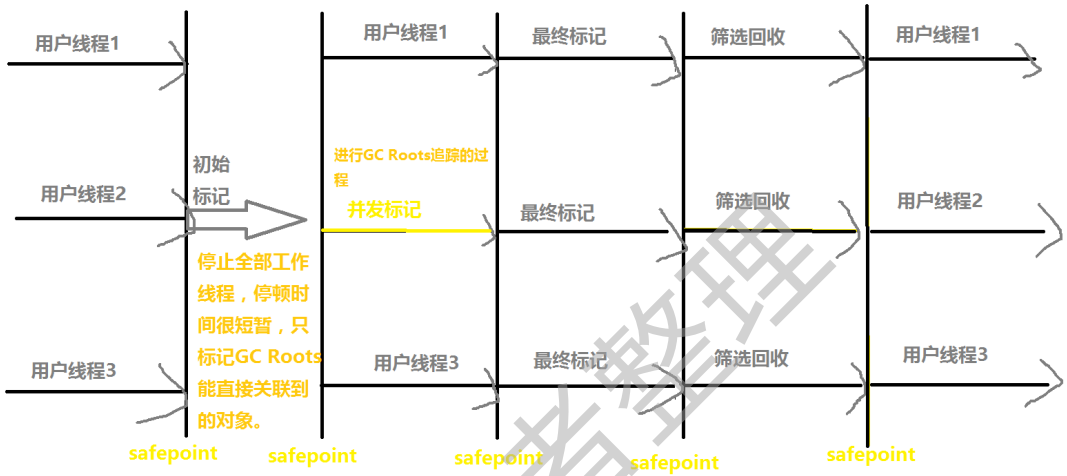
    - 标记记录，仍然需要暂停所有的工作线程。STW时间会比第一阶段稍微长点，但是远比并发标记 短，效率也很高。
    - 并发清除：清除GC Roots不可达对象，和用户线程一起工作，不需要暂停工作线程。
- 

- 所以CMS的优点是：
    - 并发高
    - 停顿低
    - STW时间短。
- 缺点：
    - 对cpu资源非常敏感（并发阶段虽然不会影响用户线程，但是会一起占用CPU资源，竞争激烈的话会导致程序变慢）。
    - 无法处理浮动垃圾，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，失败后而导致另一次Full GC的产生，由于CMS并发清除阶段用户线程还在运行，伴随程 序的运行自然会有新的垃圾产生，这一部分垃圾是出现在标记过程之后的，CMS无法在本次去处理他们，所以只好留在下一次GC时候将其清理掉。
    - 内存碎片问题（因为是标记清除算法）。当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被 降低

详细介绍一下 G1 垃圾回收器?

- 采取标记整理算法，并行收集器。
- 特点：
    - 并行与并发执行：利用多CPU的优势来缩短STW时间，在GC工作的时候，用户线程可以并行执行。
    - 分代收集：无需其他收集器配合，自己G1会进行分代收集。
    - 空间整合：不会像CMS那样产生内存碎片。
    - 可预测的停顿：可以手动控制一个长度为M毫秒的时间片段（可以用JVM参数 -XX:MaxGCPauseMillis指定），设置完后垃圾收集的时长不得超过这个（近实时）。
- 原理：
    - G1并不是简单的把堆内存分为新生代和老年代两部分，而是把整个堆划分为多个大小相等的独立区域（Region），新生代和老年代也是一部分不需要连续Region的集合。G1跟踪各个Region里面的垃圾堆 积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。
- 补充：
    - Region不是孤立的，也就是说一个对象分配在某个Region中，他并非只能被本Region中的其他对象引用，而是整个堆中任意的对象都可以相互引用，那么在【可达性分析法】来判断对

象是否存活的时候也无需扫描整个堆，Region之间的对象引用以及其他手机其中新生代和老年代之间的对象引用虚拟机都是使用Remembered Set来避免全堆扫描的。

- 步骤：
  - 初始标记：仅仅标记GCRoots能直接关联到的对象，且修改TAMS的值让下一阶段用户程序并发运 行时能正确可用的Region中创建的新对象。速度很快，会STW。
  - 并发标记：进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。不会STW。
  - 最终标记：为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的
  - 标记记录，仍然需要暂停所有的工作线程。STW时间会比第一阶段稍微长点，但是远比并发标记 短，效率也很高。
  - 筛选回收：首先对各个Region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。
- 

Minor GC与Full GC分别在什么时候发生

- 新生代内存（Eden区）不够用时候发生Minor GC也叫YGC。
- Full GC发生情况：
  - 老年代被写满
  - 持久代被写满
  - System.gc()被显示调用（只是会告诉需要GC，什么时候发生并不知道）

简述下对象的分配规则

- 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次YGC。并将还活着的对象放到from/to区，若本次YGC后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。
- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次YGC那么对象会进入Survivor区，之后每经过一次YGC那么对象的年龄加1，直到达到阀值对象进入老年区。默认阈值是15。可以通过 -XX:MaxTenuringThreshold 参数来设置。
- 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。无需等到 -XX:MaxTenuringThreshold 参 数要求的年龄。

实战调优

常用的调优工具有哪些?

- JDK内置的命令行：jps（查看jvm进程信息）、jstat（监视jvm运行状态的，比如gc情况、jvm内存情况、类加载情况等）、jinfo（查看jvm参数的，也可动态调整）、jmap（生成dump文件的，在dump的时候会影响线上服务）、jhat（分析dump的，但是一般都将dump导出放到mat上分析）、 jstack（查看线程的）。
- JDK内置的可视化界面：JConsole、VisualVM，这两个在QA环境压测的时候很有用。
- 阿里巴巴开源的arthas：神器，线上调优很方便，安装和显示效果都很友好。

大型项目如何进行性能瓶颈调优

- 数据库与SQL优化：一般dba负责数据库优化，比如集群主从等。研发负责SQL优化，比如索引、分库分表等。
- 集群优化：一般OP负责，让整个集群可以很容易的水平扩容，再比如tomcat/nginx的一些配置优化等。
- 硬件升级：选择最合适的硬件，充分利用资源。
- 代码优化：很多细节，可以参照阿里巴巴规范手册和安装sonar插件这种检测代码质量的工具。也可以适当的运用并行，比如CountDownLatch等工具。
- jvm优化：内存区域大小设置、对象年龄达到次数晋升老年代参数的调整、选择合适的垃圾收集器以及 合适的垃圾收集器参数、打印详细的GC日志和oom的时候自动生成dump。
- 操作系统优化

附录

GC常用参数

- -Xmn：年轻代
- -Xms：最小堆
- -Xmx ：最大堆
- -Xss：栈空间
- -XX:+UseTLAB：使用TLAB，默认打开
- -XX:+PrintTLAB：打印TLAB的使用情况
- -XX:TLABSize：设置TLAB大小
- -XX:+DisableExplictGC：禁用System.gc()不管用 ，防止FGC
- -XX:+PrintGC：打印GC日志
- -XX:+PrintGCDetails：打印GC详细日志信息
- -XX:+PrintHeapAtGC：打印GC前后的详细堆栈信息
- -XX:+PrintGCTimeStamps：打印时间戳
- -XX:+PrintGCApplicationConcurrentTime：打印应用程序时间
- -XX:+PrintGCApplicationStoppedTime：打印暂停时长
- -XX:+PrintReferenceGC：记录回收了多少种不同引用类型的引用
- -verbose:class：类加载详细过程
- -XX:+PrintVMOptions：jvm参数
- -XX:+PrintFlagsFinal：-XX:+PrintFlagsInitial 必须会用
- -Xloggc:opt/log/gc.log：gc日志的路径以及文件名称
- -XX:MaxTenuringThreshold：升代年龄，最大值15

G1常用参数

- -XX:+UseG1GC：开启G1
- -XX:MaxGCPauseMillis：建议值，G1会尝试调整Young区的块数来达到这个值
- -XX:GCPauseIntervalMillis：GC的间隔时间
- -XX:+G1HeapRegionSize：分区大小，建议逐渐增大该值，1 2 4 8 16 32。 随着size增加，垃圾的存活时间更长，GC间隔更长，但每次GC的时间也会更长 ZGC做了改进（动态区块大小）

- G1NewSizePercent：新生代最小比例，默认为5%
- G1MaxNewSizePercent：新生代最大比例，默认为60%
- GCTimeRatio：GC时间建议比例，G1会根据这个值调整堆空间
- ConcGCThreads：线程数量
- InitiatingHeapOccupancyPercent：启动G1的堆空间占用比例