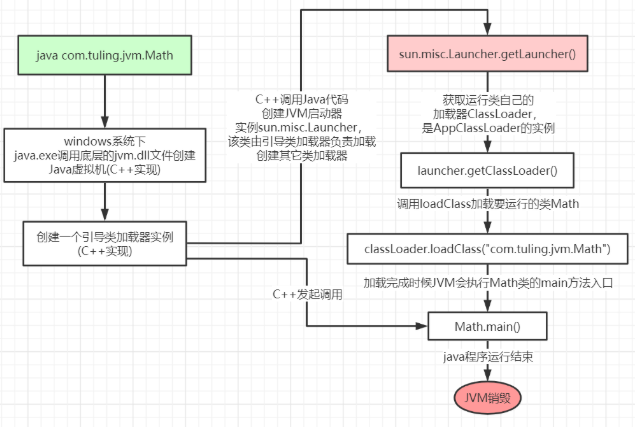
**JVM**

**类加载机制**

**全文核心：**

**一个类加载器+一个全限定类名 标记为JVM中唯一的一个类**

Java命令执行代码的大体流程



**loadClass的类加载过程**

**加载 >> 验证 >> 准备 >> 解析 >> 初始化 >>** 使用 >> 卸载

* 加载：在硬盘上查找并通过IO读入字节码文件，使用到类时才会加载，例如调用类的main()方法，new对象等等，在加载阶段会在内存中生成一个**代表这个类的java.lang.Class对象（是在堆里面）**，作为方法区这个类的各种数据的访问入口（这个class对象是给我们开发人员用的，可以理解为方法区这个类元信息的镜像）。

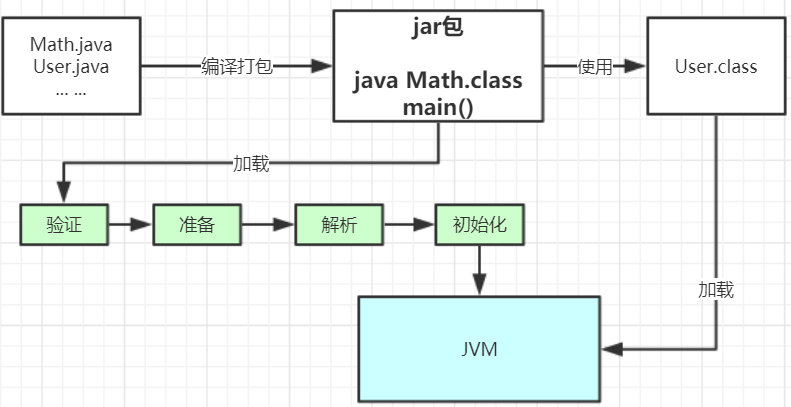
**使用到类的时候把class字节码文件从磁盘加载到JVM的方法区**（可以理解为这个类的class字节码加载进去，只是加载）class常量池-》运行时常量池。把类名方法名，变成方法区中的一个个符号

* 验证：校验字节码文件的正确性
* 准备：给类的静态变量分配内存，并赋予默认值（不是初始值）。（boolean就是false，int就是0，对象就是null）
* 解析：将**符号引用**替换为直接引用，该阶段会把一些静态方法(符号引用，比如main()方法)替换为指向数据所存内存的指针或句柄等(直接引用)，这是所谓的**静态链接**过程(类加载期间完成)，**动态链接**是在程序运行期间完成的将符号引用替换为直接引用，下节课会讲到动态链接

**会把类中的静态方法，由方法区里的一 个个符号转化为链接地址，就是所谓静态链接**

* **初始化**：对类的静态变量初始化为指定的值，执行静态代码块

可以看到，上面的每个过程，不管是对变量，方法，还是代码块，都是 **静态**。



类被加载到方法区中后主要包含 **运行时常量池、类型信息、字段信息、方法信息、类加载器的引用、对应class实例的引用**等信息。

**类加载器的引用**：这个类到类加载器实例的引用

**对应class实例的引用**：类加载器在加载类信息放到方法区中后，会创建一个对应的Class 类型的对象实例放到堆(Heap)中, 作为开发人员访问方法区中类定义的入口和切入点。

**注意，**主类在运行过程中如果使用到其它类，会逐步加载这些类。jar包或war包里的类不是一次性全部加载的，是使用到时才加载。

**类加载器**

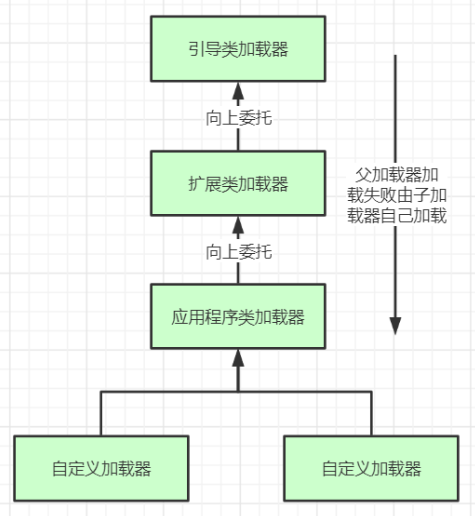
上面的类加载过程主要是通过类加载器来实现的，Java里有如下几种类加载器

* 引导类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
* 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
* 应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
* 自定义加载器：负责加载用户自定义路径下的类包

类加载器初始化过程**：**

类运行加载时会创建JVM启动器实例sun.misc.Launcher。在Launcher构造方法内部，其创建了两个类加载器，分别是sun.misc.Launcher.ExtClassLoader(扩展类加载器)和sun.misc.Launcher.AppClassLoader(应用类加载器)。JVM默认使用Launcher的getClassLoader()方法返回的类加载器AppClassLoader的实例加载我们的应用程序。

**双亲委派机制**



这里类加载其实就有一个**双亲委派机制**，加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。

比如我们的Math类，最先会找应用程序类加载器加载，应用程序类加载器会先委托扩展类加载器加载，扩展类加载器再委托引导类加载器，顶层引导类加载器在自己的类加载路径里找了半天没找到Math类，则向下退回加载Math类的请求，扩展类加载器收到回复就自己加载，在自己的类加载路径里找了半天也没找到Math类，又向下退回Math类的加载请求给应用程序类加载器，应用程序类加载器于是在自己的类加载路径里找Math类，结果找到了就自己加载了。。

**双亲委派机制说简单点就是，先找父亲加载，不行再由儿子自己加载**

**为什么要设计双亲委派机制？**

* **避免类的重复加载**：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，保证**被加载类的唯一性**
* **沙箱安全机制**：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改

全盘负责委托机制

“全盘负责”是指当一个ClassLoder装载一个类时，除非显示的使用另外一个ClassLoder，该类所依赖及引用的类也由这个ClassLoder载入**。**

**机制实例**

自己定义一个java.lang.string，看看是否能加载并替换jre核心类的String

|  |
| --- |
| Java package java.lang; public class String {  public static void main(String[] args) {  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*My String Class\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  } }  运行结果： 错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:  public static void main(String[] args) 否则 JavaFX 应用程序类必须扩展javafx.application.Application |

**原因 分析**：根据双亲委派，自定义的类加载器会一直向上委派，直到引导类加载器。引导类加载器，检查自己扫描的核心类库目录下就有这个类，就会加载JRE核心类库下的java.lang.String.class进JVM。由于已经被引导类加载器加载，自定义的类加载器自然不会再去加载用户自己实现的java.lang.String.class。核心类库中的String自然没有main方法，运行会报错。 这就是双亲委派的沙箱机制。防止有人篡改核心类库下的代码。

**普通自定义类加载器**

自定义的类加载器可以加载指定路径下的class文件，默认情况下，会遵循双亲委派逐层的向上，

自定义类加载器-》appClassLoader-》扩展类加载器-》引导类加载器。发现找不到该类后，又会逐层向下查找，直到自定义类加载器在指定路径下找到该class文件，完成加载进JVM。

自定义类加载器只需要继承 java.lang.ClassLoader 类，该类有两个核心方法，一个是loadClass(String, boolean)，实现了**双亲委派机制**，还有一个方法是findClass，默认实现是空方法，所以我们自定义类加载器主要是**重写findClass方法**。

|  |
| --- |
| Java public class MyClassLoaderTest {  static class MyClassLoader extends ClassLoader {  private String classPath;   public MyClassLoader(String classPath) {  this.classPath = classPath;  }   private byte[] loadByte(String name) throws Exception {  name = name.replaceAll("\\.", "/");  FileInputStream fis = new FileInputStream(classPath + "/" + name  + ".class");  int len = fis.available();  byte[] data = new byte[len];  fis.read(data);  fis.close();  return data;  }    protected Class<?> findClass(String name) throws ClassNotFoundException {  try {  byte[] data = loadByte(name);  //defineClass将一个字节数组转为Class对象，这个字节数组是class文件读取后最终的字节数组。  return defineClass(name, data, 0, data.length);  } catch (Exception e) {  e.printStackTrace();  throw new ClassNotFoundException();  }  }   }   public static void main(String args[]) throws Exception {   **初始化自定义类加载器，会先初始化父类ClassLoader，其中会把自定义类加载器的父加载器设置为应用程序类加载器AppClassLoader**  MyClassLoader classLoader = new MyClassLoader("D:/test");   **D盘创建 test/com/tuling/jvm 几级目录，将User类的复制类User1.class丢入该目录**  Class clazz = classLoader.loadClass("com.tuling.jvm.User1");  Object obj = clazz.newInstance();  Method method = clazz.getDeclaredMethod("sout", null);  method.invoke(obj, null);  System.out.println(clazz.getClassLoader().getClass().getName());  } }  运行结果： =======自己的加载器加载类调用方法======= com.tuling.jvm.MyClassLoaderTest$MyClassLoader   **一句话，自定义类加载器可以加载指定路径下的class文件** |

**打破双亲委派的自定义类加载器**

我们可以自定义类加载器，打破双亲委派加载核心类库同名的类吗

**自定义类加载器示例：**

重写loadClass方法，将双亲委派的逻辑删除，从而做到打破双亲委派。

|  |
| --- |
| Java public class MyClassLoaderTest {  static class MyClassLoader extends ClassLoader {  private String classPath;   public MyClassLoader(String classPath) {  this.classPath = classPath;  }   /\*\*  \* 重写类加载方法，实现自己的加载逻辑，不委派给双亲加载  \* @param name  \* @param resolve  \* @return  \* @throws ClassNotFoundException  \*/  protected Class<?> loadClass(String name, boolean resolve)  throws ClassNotFoundException {  synchronized (getClassLoadingLock(name)) {  // First, check if the class has already been loaded  Class<?> c = findLoadedClass(name);   if (c == null) {     **标红的内容为双亲委派的逻辑，予以删除，从而打破双亲委派**  **if (parent != null) {如果当前加载器父加载器不为空则委托父加载器加载该**  **c = parent.loadClass(name, false);**  **} else { 如果当前加载器父加载器为空则委托引导类加载器加载该类**  **c = findBootstrapClassOrNull(name);**  **}**    // If still not found, then invoke findClass in order  // to find the class.  long t1 = System.nanoTime();  c = findClass(name);   // this is the defining class loader; record the stats  sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);  sun.misc.PerfCounter.getFindClasses().increment();  }  if (resolve) {  resolveClass(c);  }  return c;  }  }  }   public static void main(String args[]) throws Exception {  MyClassLoader classLoader = new MyClassLoader("D:/test");  //尝试用自己改写类加载机制去加载自己写的java.lang.String.class  Class clazz = classLoader.loadClass("java.lang.String");  Object obj = clazz.newInstance();  Method method= clazz.getDeclaredMethod("sout", null);  method.invoke(obj, null);  System.out.println(clazz.getClassLoader().getClass().getName());  } }  **打破双亲委派的自定义类加载器也不能加载核心类同名的类，这就是沙箱机制的保护** **沙箱机制保护了jre核心类库的代码不会篡改** 运行结果： java.lang.SecurityException: Prohibited package name: java.lang  at java.lang.ClassLoader.preDefineClass(ClassLoader.java:659)  at java.lang.ClassLoader.defineClass(ClassLoader.java:758) |

**Tomcat加载原理及打破双亲委派**

**场景**

一个tomcat容器下部署多个应用，他们需要共享tomcat /lib包下的类。但是每个war包下的应用又需要相互隔离，因为，多个war包下依赖第三方类库可能会不同（比如一个是依赖spring3，另外一个依赖spring4），不隔离开来被同一个类加载器 加载到就会相互影响（同一个类，spring3的加载进JVM spring4就会加载不进去）。

**实现原理**

tomcat的加载是在JVM的类加载基础上自定义了好几层的类加载器：commonClassLoader->webappclassLoader->jspClassLoader,每层classLoader圈定的范围是不一样的。

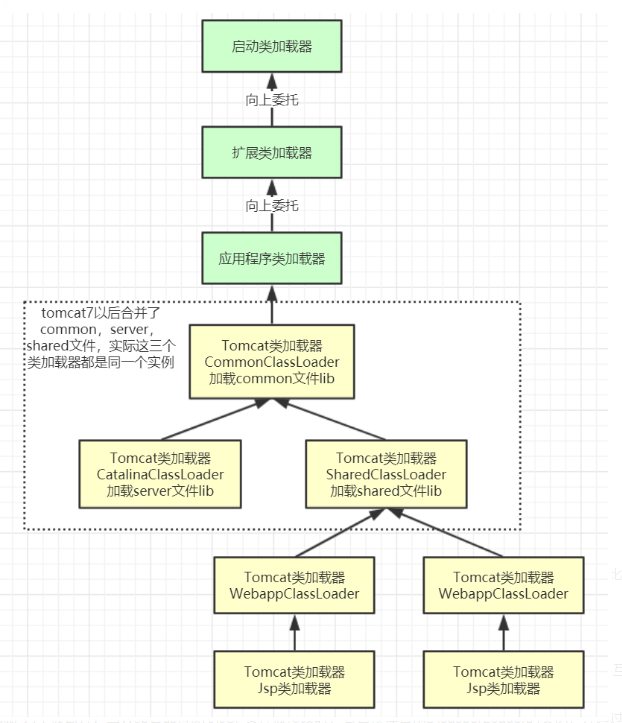
**我们知道在jvm中，一个classLoader+全限定类名标记为一个唯一的类。所以tomcate用一个自定义的类加载器commonClassLoad加载他lib包下的类，所有应用共享。每个应用war包下都有一个自己私有的自定义类加载器webappClassLoader加载自己目录下的class文件。这些webappClassLoader是打破了双亲委派的，不会传递给父记载器commonClassLoad加载（因为父加载器一旦加载就无法做到硬件之间隔离）**

同理，JSP的热加载原理和开发时候的热加载原理也是一样，JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的热加载功能。

**热加载原理**

后台启动线程监听class文件变化，如果变化了，找到该类对应的加载器引用(gcroot)，重新生成一个新的类加载器赋值给引用，加载对应的class文件，之前的那个加载器因为没有gcroot引用了，下一次gc的时候会被销毁。

一句话：监听到文件变化就干掉原来的类加载器，重新生成一个类加载器来加载class文件，从而做到热加载。



代码模拟;

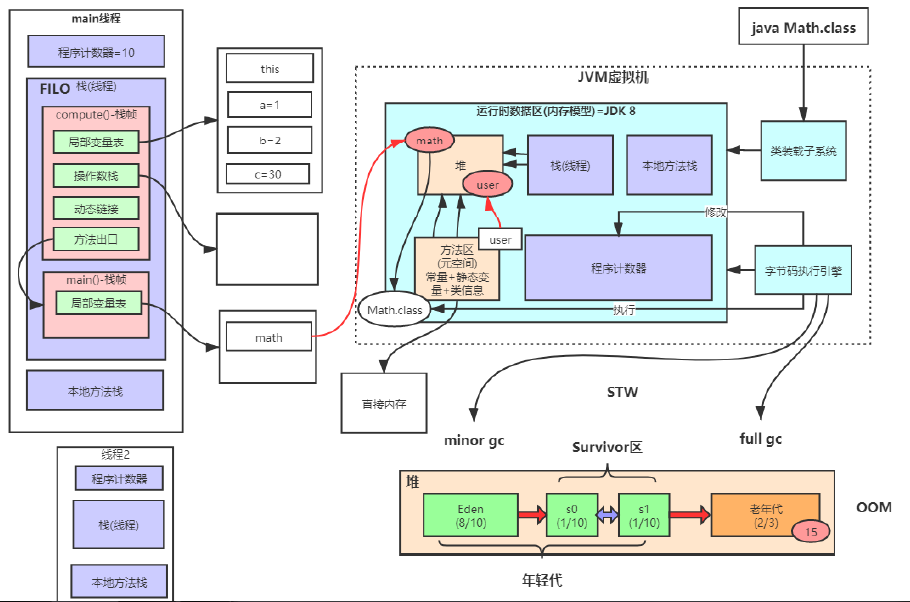
|  |
| --- |
| Java public class MyClassLoaderTest {  static class MyClassLoader extends ClassLoader {  private String classPath;   public MyClassLoader(String classPath) {  this.classPath = classPath;  }   /\*\*  \* 重写类加载方法，实现自己的加载逻辑，不委派给双亲加载  \* @param name  \* @param resolve  \* @return  \* @throws ClassNotFoundException  \*/  protected Class<?> loadClass(String name, boolean resolve)  throws ClassNotFoundException {  synchronized (getClassLoadingLock(name)) {  // First, check if the class has already been loaded  Class<?> c = findLoadedClass(name);   if (c == null) {  // If still not found, then invoke findClass in order  // to find the class.  long t1 = System.nanoTime();   **非自定义的类还是走双亲委派加载**   **if (!name.startsWith("com.tuling.jvm")){**  **c = this.getParent().loadClass(name);**  **}else{**  **c = findClass(name);**  **}**   // this is the defining class loader; record the stats  sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);  sun.misc.PerfCounter.getFindClasses().increment();  }  if (resolve) {  resolveClass(c);  }  return c;  }  }  }   public static void main(String args[]) throws Exception {  MyClassLoader classLoader = new MyClassLoader("D:/test");  Class clazz = classLoader.loadClass("com.tuling.jvm.User1");  Object obj = clazz.newInstance();  Method method= clazz.getDeclaredMethod("sout", null);  method.invoke(obj, null);  System.out.println(clazz.getClassLoader());    System.out.println();  MyClassLoader classLoader1 = new MyClassLoader("D:/test1");  Class clazz1 = classLoader1.loadClass("com.tuling.jvm.User1");  Object obj1 = clazz1.newInstance();  Method method1= clazz1.getDeclaredMethod("sout", null);  method1.invoke(obj1, null);  System.out.println(clazz1.getClassLoader());  } }  运行结果： =======自己的加载器加载类调用方法======= com.tuling.jvm.MyClassLoaderTest$MyClassLoader@266474c2  =======另外一个User1版本：自己的加载器加载类调用方法======= com.tuling.jvm.MyClassLoaderTest$MyClassLoader@66d3c617 |

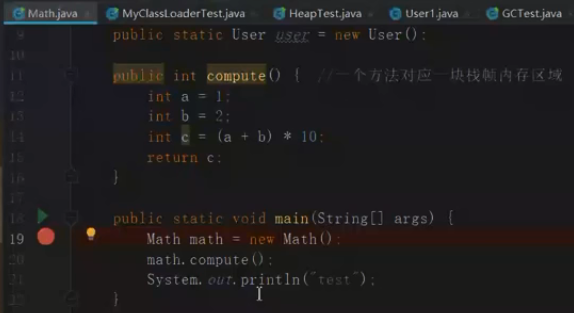
**总结**

同一个JVM内，两个相同包名和类名的类对象可以共存，因为他们的类加载器可以不一样，所以看两个类对象是否是同一个，除了看类的包名和类名是否都相同之外，还需要他们的类加载器也是同一个才能认为他们是同一个。

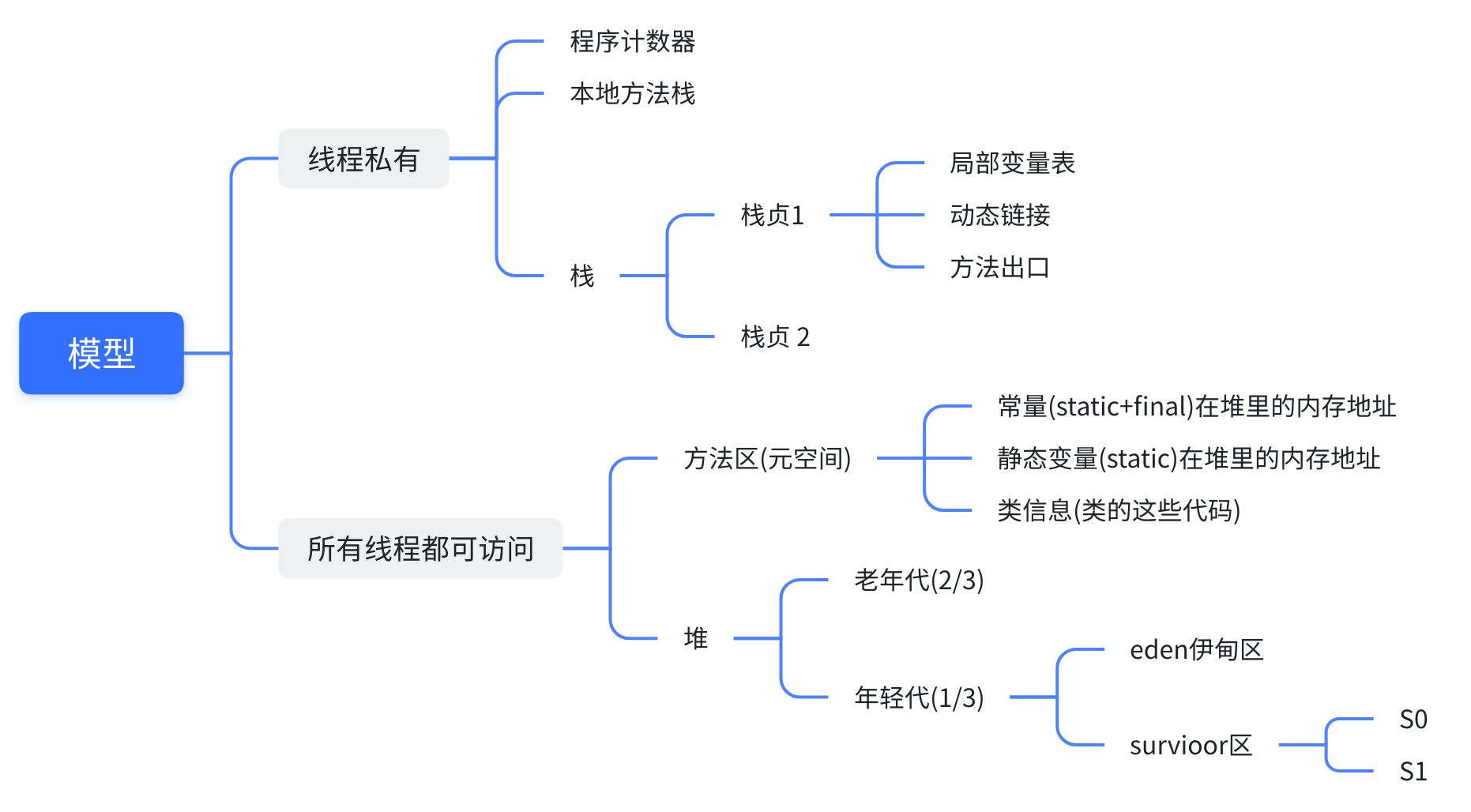
**JVM内存模型**

总体说明：JVM内存模型里面我觉得主要分成还有一种是所有线程都可访问的，里面主要是元空间和堆，元空间就是加载class类存放的地方里面有一些类的信息，堆是XXX。还有一种是线程私有的，里面有 1 2 3





JVM内存模型脑图：



**线程私有的**：

1. **程序计数器：**其实java编译后的class文件javaP-c可以看到，就是被编译成了一条条指令，这些指令都是有编号的，正常来说CPU会按照编号顺序执行，程序计数器就是记录CPU执行到哪一行指令的，因为CPU时间片有时候被其他线程抢走去执行，回来再从哪有一条执行呢，就是看程序计数器上的记录。
2. **本地方法栈**：本地方法native 方法(jvm底层C++实现的方法)运行时，读取dll文件（类似于jar包），也是需要有一块内存空间，就是存储这个本地方法栈。
3. **栈**（又叫线程栈）：是线程私有的，来一个线程就创建了线程栈，他的**生命周期和线程相同。栈**里面，主要由一个个栈贞组成。

栈贞：**每个方法**自己**独立的内存区域**，里面有这个方法的局部变量表,动态链接和方法出口等.

栈贞和栈的关系：我们知道，方法调用的时候，是一个个嵌套调用的，先调用的方法最后执行，比如：A()里面 嵌套调用B(),B方法里又嵌套调用C。那么实际执行的时候，肯定是C->B->A。也就是先进后出FILO ，**在栈里面的表现**就是先压入A方法的栈贞 ，再压入B方法栈贞，最后压入C方法栈贞。出栈的时候先出C再出B最后出A。

备注 ：栈贞中的三个概念

1) 局部变量表：栈贞中局部变量指向堆的内存地址

2) 动态链接：代码执行到非静态的方式时，把元空间里的符号（class文件加载到元空间里是一个个符号）转换成链接地址

3) 方法出口：这个方法执行完了，在父方法的哪一行继续向下执行，就标记在这里。

**所有线程可访问的**：

1. **方法区**（1.8之后叫元空间，以前叫永久代）：.class文件，常量，静态变量（也是地址，new出来的在堆里面），类信息。
2. **堆**：存放new出来的对象。里面分为年轻代和老年代，默认比例是1比2，年轻代里面又分为 伊甸园区和survioor区，默认是8比2

**年轻代为什么有两个Survivor，而不是一个或者其他数量**

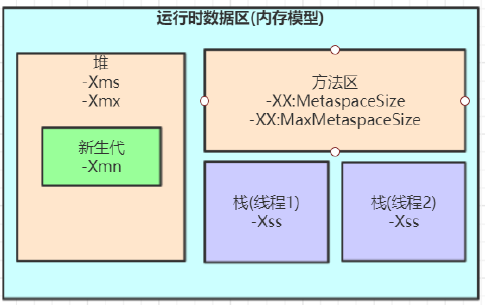
如果只有一个Survivor区，第一次YGC时Eden区的可以进入Survivor区，那下一次YGC时呢，由于Survivor已满，那么其实本次YGC进入Survivor区的对象必须在下次YGC时进入年老代或被回收，这样YGC的效率就不高了，这肯定都不是我们希望的。

而两个Survivor的话就不一样了，**第一次YGC时Eden区和Survivor 0的可以进入Survivor 1区**,

然后**清理掉Eden和刚才用过的Survivor 0**，接下来把**Survivor 0 留作下次GC时**，Eden和Survivor1存活下来的对象的存放地方，也即是说**两个Survivor区来回交替使用**。（本质原因是，**没有清理完后，从Survivor 往Eden放回的动作**）

正是因为存在两个Survivor区，使得Survivor区的对象也可以来回复制，而不是马上进入老年代，直到达到一定年龄才能进入老年代，尽可能保证进入老年代的是真的够资格的

**参数说明**



Spring Boot程序的JVM参数设置格式(Tomcat启动直接加在bin目录下catalina.sh文件里)：

**最重要的几个参数：设置每个线程的最大栈空间的大小，元空间大小，堆大小，以及堆里面年轻代和老年代的比例，年轻代里面伊甸园区和suroor区的比例。**

|  |
| --- |
| Java java ‐Xms2048M ‐Xmx2048M ‐Xmn1024M ‐Xss512K ‐XX:MetaspaceSize=256M ‐XX:MaxMetaspaceSize=256M ‐jar microservice‐eureka‐server.jar |

-Xss：**每个线程的栈大小**

-Xms：设置堆的**初始**可用大小，默认物理内存的1/64

-Xmx：设置堆的**最大**可用大小，默认物理内存的1/4

-Xmn：新生代大小

-XX:NewRatio：默认2表示新生代占年老代的1/2，占整个堆内存的1/3。

-XX:SurvivorRatio：默认8表示一个survivor区占用1/8的Eden内存，即1/10的新生代内存。

关于元空间的JVM参数有两个：-XX:MetaspaceSize=N和 -XX:MaxMetaspaceSize=N

-XX：MaxMetaspaceSize： 设置元空间最大值， 默认是-1， 即不限制， 或者说只受限于本地内存大小。

-XX：MetaspaceSize： 指定元空间触发Fullgc的初始阈值(元空间无固定初始大小)， 以字节为单位，默认是21M左右，达到该值就会触发full gc进行类型卸载， 同时收集器会对该值进行调整： 如果释放了大量的空间， 就适当降低该值； 如果释放了很少的空间， 那么在不超过-XX：MaxMetaspaceSize（如果设置了的话） 的情况下， 适当提高该值。这个跟早期jdk版本的-XX:PermSize参数意思不一样，-XX:PermSize代表永久代的初始容量。

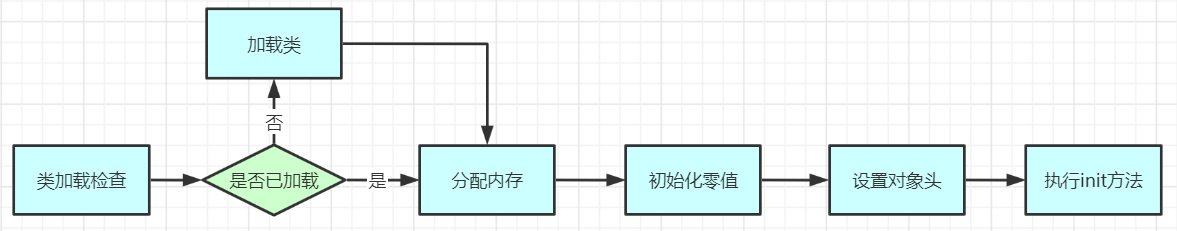
由于调整元空间的大小需要Full GC，这是非常昂贵的操作，如果应用在启动的时候发生大量Full GC，通常都是由于永久代或元空间发生了大小调整，基于这种情况，一般建议在JVM参数中将MetaspaceSize和MaxMetaspaceSize设置成一样的值，并设置得比初始值要大，对于8G物理内存的机器来说，一般我会将这两个值都设置为256M。

演示一个栈溢出，**StackOverflowError**

|  |
| --- |
| TypeScript // JVM设置 -Xss128k(默认1M) public class StackOverflowTest {  static int count = 0;  static void redo() {  **count**++;  redo();  }  public static void main(String[] args) {  try {  redo();  } catch (Throwable t) {  t.printStackTrace();  System.out.println(count);  }  } } 运行结果： java.lang.StackOverflowError  at com.tuling.jvm.StackOverflowTest.redo(StackOverflowTest.java:12)  at com.tuling.jvm.StackOverflowTest.redo(StackOverflowTest.java:13)  at com.tuling.jvm.StackOverflowTest.redo(StackOverflowTest.java:13)  ...... **结论：** -Xss设置越小**count**值越小，说明一个线程栈里能分配的栈帧就越少，但是对JVM整体来说能开启的线程数会更多 |

**对象从创建到消亡**

**创建**



**1.类加载检查**

虚拟机遇到一条new（new关键词、对象克隆、对象序列化等）指令时，首先将去检查这个类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

**2.分配内存**

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类 加载完成后便可完全确定，为对象分配空间的任务等同于把 一块确定大小的内存从Java堆中划分出来。

划分内存的办法：指针碰撞+CAS

* “指针碰撞”（Bump the Pointer）(默认用指针碰撞)

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。

* CAS（compare and swap）

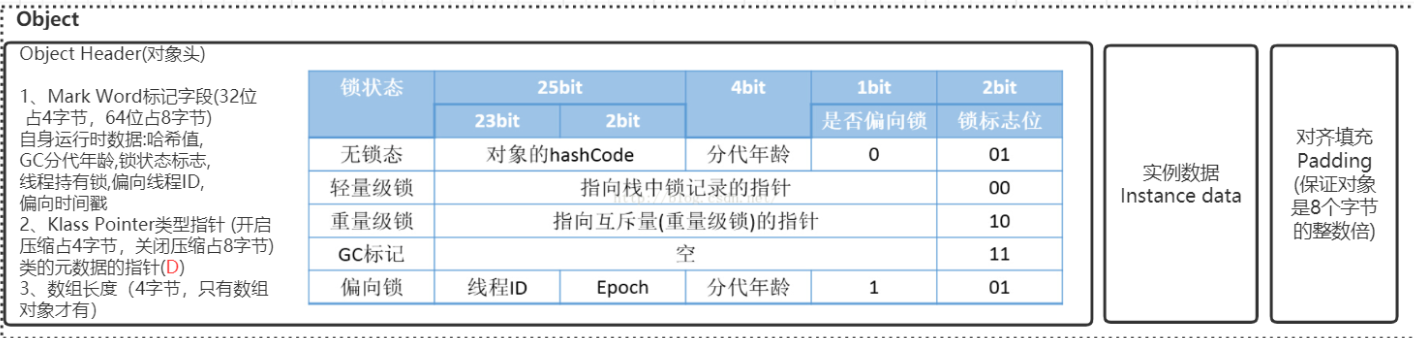
虚拟机采用CAS配上失败重试的方式保证更新操作的原子性来对分配内存空间的动作进行同步处理。

**3.初始化零值**

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（不包括对象头）。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直接使用，程序能访问到这些字段的数据类型所对应的零值。

**4.设置对象头**

初始化零值之后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头Object Header之中。



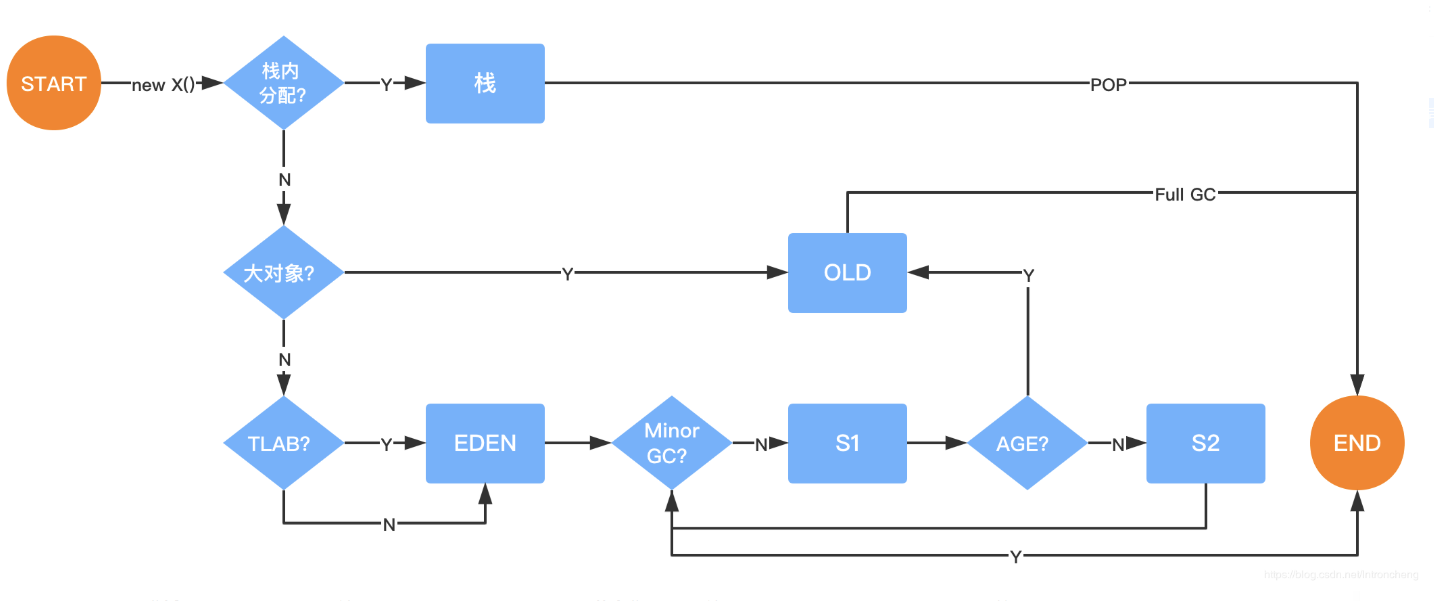
对象在内存中存储的布局主要有两块区域：对象头（Header）、 实例数据（Instance Data）。 对象头包括两部分信息，第一部分用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

**5.执行<init>方法**

执行<init>方法，即对象按照程序员的意愿进行初始化。对应到语言层面上讲，就是为属性赋值（注意，这与上面的赋零值不同，这是由程序员赋的值），和执行构造方法。

**6.** 把对象的引用也就是内存地址给到栈/方法区

**对象内存分配**



**对象栈上分配**

我们通过JVM内存分配可以知道JAVA中的对象都是在堆上进行分配，当对象没有被引用的时候，需要依靠GC进行回收内存，如果对象数量较多的时候，会给GC带来较大压力，也间接影响了应用的性能。为了减少临时对象在堆内分配的数量，JVM通过**逃逸分析**确定该对象不会被外部访问。如果不会逃逸可以将该对象在**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，就减轻了垃圾回收的压力。

|  |
| --- |
| Java **对象逃逸分析**：就是分析对象动态作用域，当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中。  public User test1() {  User user = new User();  user.setId(1);  user.setName("laozhang");  //TODO 保存到数据库  return user; }  public void test2() {  User user = new User();  user.setId(1);  user.setName("laozhang");  //TODO 保存到数据库 }   很显然test1方法中的user对象被返回了，**这个对象的作用域范围不确定**，test2方法中的user对象我们**可以确定当方法结束这个对象就可以认为是无效对象了**，对于这样的对象我们其实可以将其分配在栈内存里，让其在方法结束时跟随栈内存一起被回收掉。 |

**对象在Eden区分配**

大多数情况下，对象在新生代中 Eden 区分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次Minor GC。我们来进行实际测试一下。

在测试之前我们先来看看 **Minor GC和Full GC 有什么不同呢？**

* **Minor GC/Young GC**：指发生新生代的的垃圾收集动作，Minor GC非常频繁，回收速度一般也比较快。
* **Major GC/Full GC**：一般会回收老年代 ，年轻代，方法区的垃圾，Major GC的速度一般会比Minor GC的慢10倍以上。

**Eden与Survivor区默认8:1:1**

大量的对象被分配在eden区，eden区满了后会触发minor gc，可能会有99%以上的对象成为垃圾被回收掉，剩余存活的对象会被挪到为空的那块survivor区，下一次eden区满了后又会触发minor gc，把eden区和survivor区垃圾对象回收，把剩余存活的对象一次性挪动到另外一块为空的survivor区，因为新生代的对象都是朝生夕死的，存活时间很短，所以JVM默认的8:1:1的比例是很合适的，**让eden区尽量的大，survivor区够用即可。**

**长期存活的对象将进入老年代**

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为1。对象在 Survivor 中每熬过一次 MinorGC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁，CMS收集器默认6岁，不同的垃圾收集器会略微有点不同），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数来设置

**大对象直接进入老年代**

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。JVM参数 -XX:PretenureSizeThreshold 可以设置大对象的大小，如果对象超过设置大小会直接进入老年代，不会进入年轻代，这个参数只在 Serial 和ParNew两个收集器下有效。

比如设置JVM参数：-XX:PretenureSizeThreshold=1000000 (单位是字节) -XX:+UseSerialGC ，再执行下上面的第一个程序会发现大对象直接进了老年代

**为什么要这样呢？**

为了避免为大对象分配内存时的复制操作而降低效率。

**大对象提前进入老年代（对象动态年龄判断）**

由于JVM参数和实际生产需求有差异，造成的。生产上实际遇到并且调优过的。

当前放对象的Survivor区域里(其中一块区域，放对象的那块s区)，一批对象的总大小大于这块Survivor区域内存大小的50%(-XX:TargetSurvivorRatio可以指定)，那么此时**大于等于**这批对象年龄最大值的对象，就可以**直接进入老年代**了，例如Survivor区域里现在有一批对象，年龄1+年龄2+年龄n的多个年龄对象总和超过了**Survivor区域的50%**，此时就会把年龄n(含)以上的对象都放入老年代。这个规则其实是希望那些可能是长期存活的对象，尽早进入老年代。**对象动态年龄判断机制一般是在minor gc之后触发的。**

这个我也在线上系统有一定调优经验的。

**老年代回收fullGC触发条件（老年代空间分配担保机制）**

fullGC会STW（stop the word）

1. 年轻代每次**minor gc**之前JVM都会计算下老年代**剩余可用空间，**如果这个可用空间小于年轻代里现有的所有对象大小之和(**包括垃圾对象**)，就会触发一次Full gc，对老年代和年轻代一起回收一次垃圾。
2. minor gc之后剩余存活的需要挪动到老年代的对象大小还是大于老年代可用空间，那么也会触发full gc。

fullGC后依然没有空间存放，就会OOM。

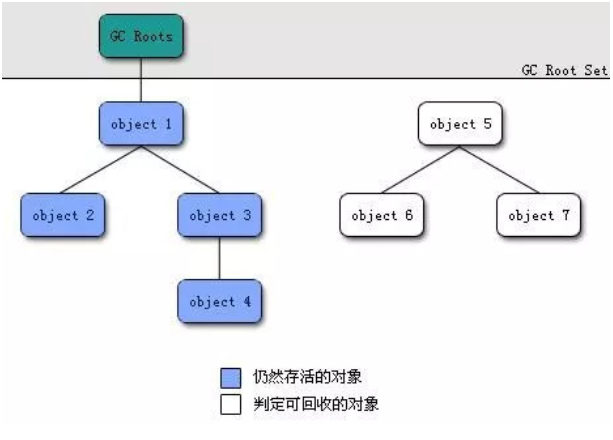
**对象内存回收**

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断哪些对象已经死亡（即不能再被任何途径使用的对象）。

可达性分析算法

将**“GC Roots”** 对象作为起点，从这些节点开始向下搜索引用的对象，找到的对象都标记为**非垃圾对象**，其余未标记的对象都是垃圾对象

**GC Roots**根节点：线程栈的本地变量、静态变量、本地方法栈的变量等等



常见引用类型

java的引用类型一般分为四种：强引用、软引用,弱引用

**强引用**：普通的变量引用

|  |
| --- |
| Java public static User user = new User(); |

**软引用**：将对象用SoftReference软引用类型的对象包裹，正常情况不会被回收，但是GC做完后发现释放不出空间存放新的对象，则会把这些软引用的对象回收掉。

|  |
| --- |
| Java public static SoftReference<User> user = new SoftReference<User>(new User()); |

**弱引用：**Weak Reference包裹的 **，**弱引用简单来说就是将对象留在内存的能力不是那么强的引用。使用WeakReference，垃圾回收器会帮你来决定引用的对象何时回收并且将对象从内存移除。创建弱引用如下



弱引用和软引用：弱引用是垃圾收集器根据内存情况来决定是不是回收。软引用会比弱引用好点，没有空间了才会回收

**虚引用 （Phantom Reference）：**被回收后有一个记录

finalize()方法最终判定对象是否存活

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历再次标记过程。

**标记的前提是对象在进行可达性分析后发现没有与GC Roots相连接的引用链。**

**如何判断一个类是无用的类**

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

类需要同时满足下面3个条件才能算是 “无用的类” ：

* 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
* 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，无法在任何地方通过反射访问该类的方法。

**垃圾回收：分代收集**

**分代收集理论**

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

当代垃圾收集都是基于分代收集理论,算法一共三种: 整理和清除比复制算法慢10倍以上

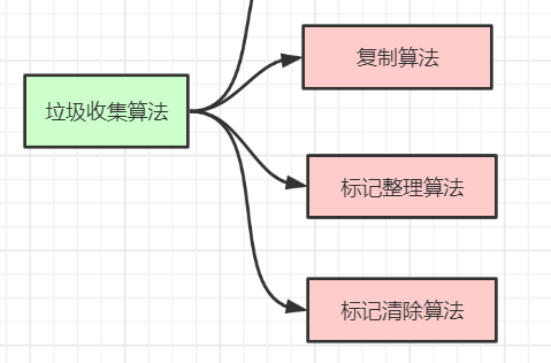
年轻代（快）：

1. 复制算法 ：内存分两块，存活的对象复制到另一半，然后把老的一半直接清除

老年代（慢）：

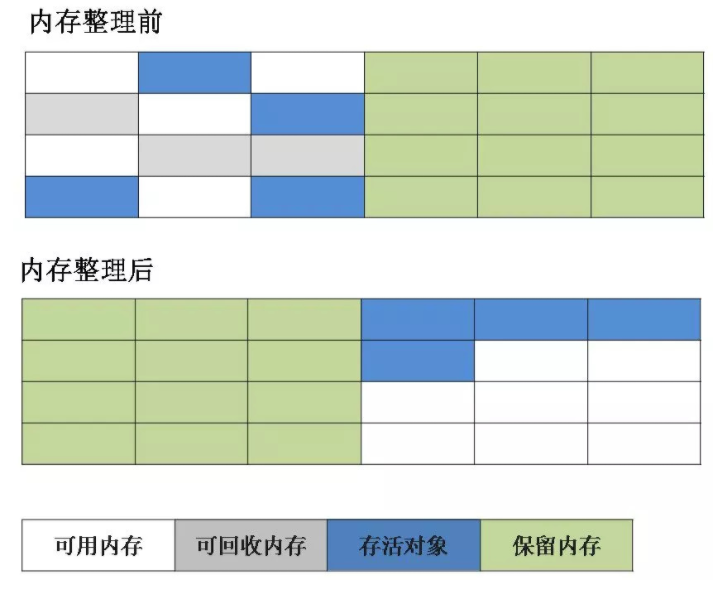
1. 标记-整理算法 ：存活的打上标记，其他的清除。问题(1.存活的对象多效率就低 2.清理后有大量不连续的空间)
2. 标记-清除算法 ：标记以后，把存活的对象移动到内存的一端，然后直接清理掉端边际以外的区域。为老年代的特点特出的一种算法

**垃圾收集算法**



标记-复制算法（年轻代）

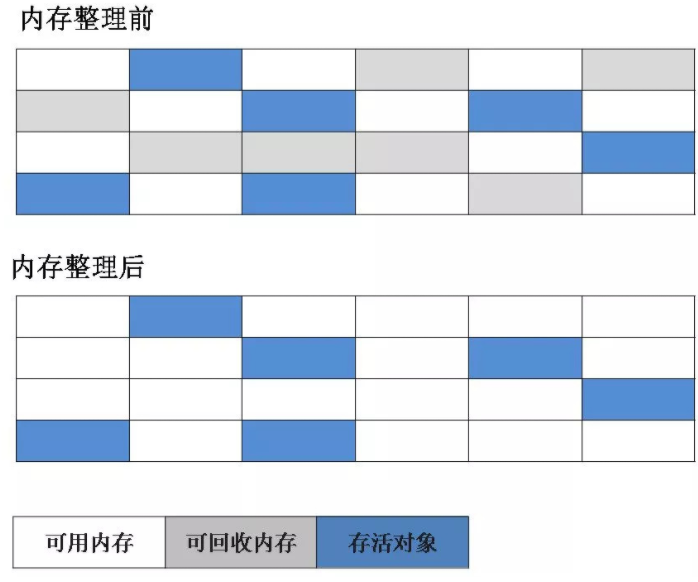
为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。



标记-清除算法（老年代）

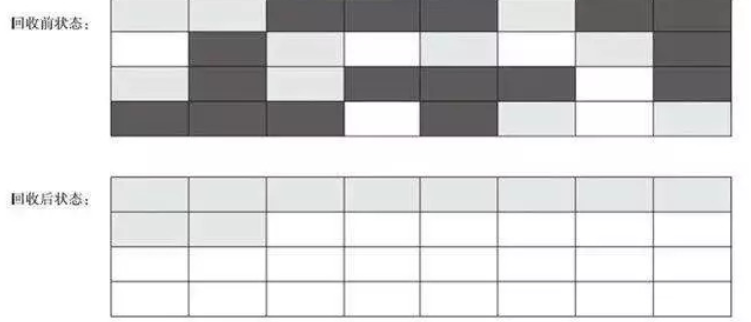
算法分为“标记”和“清除”阶段：标记存活的对象， 统一回收所有未被标记的对象(一般选择这种)；也可以反过来，标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象 。它是最基础的收集算法，比较简单，但是会带来两个明显的问题：

1. 效率问题 (如果需要标记的对象太多，效率不高)
2. 空间问题（标记清除后会产生大量不连续的碎片）

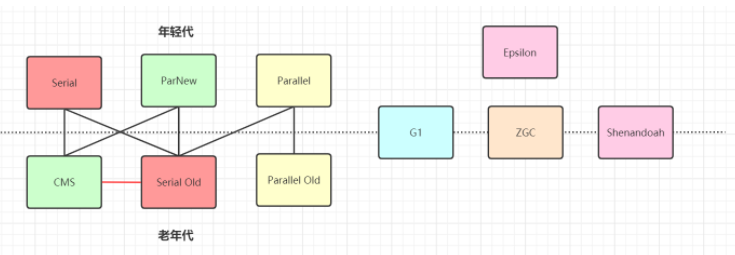


标记-整理算法（老年代）

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。



**常用的垃圾收集器**

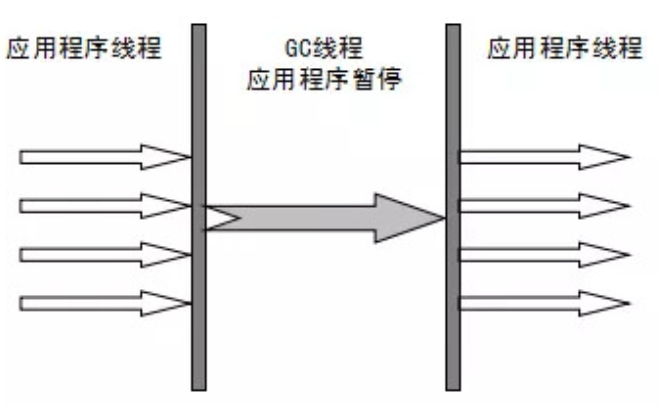


收集算法是内存回收的方法论，垃圾收集器是内存回收的具体实现

**1.1 Serial收集器(-XX:+UseSerialGC -XX:+UseSerialOldGC)**

单线程收集器，垃圾回收的时候会暂停其他所有的工作线程（ "Stop The World" ），直到其结束。

新生代和老年代能用：新生代采用复制算法，老年代采用标记-整理算法。

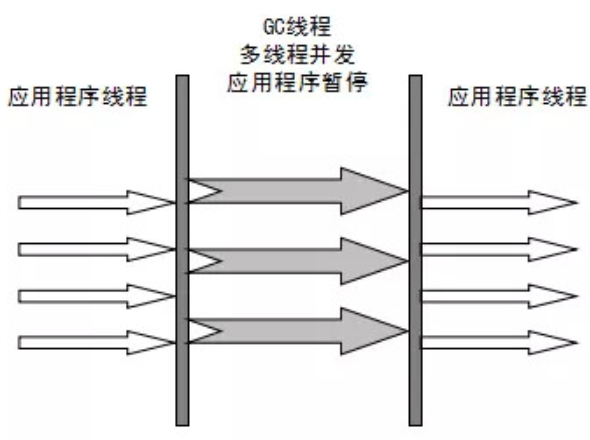


**1.2 Parallel Scavenge收集器(-XX:+UseParallelGC(年轻代),-XX:+UseParallelOldGC(老年代))**

Parallel收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和Serial收集器类似。默认的收集线程数跟cpu核数相同，当然也可以用参数(-XX:ParallelGCThreads)指定收集线程数，但是一般不推荐修改。

Parallel Scavenge收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。

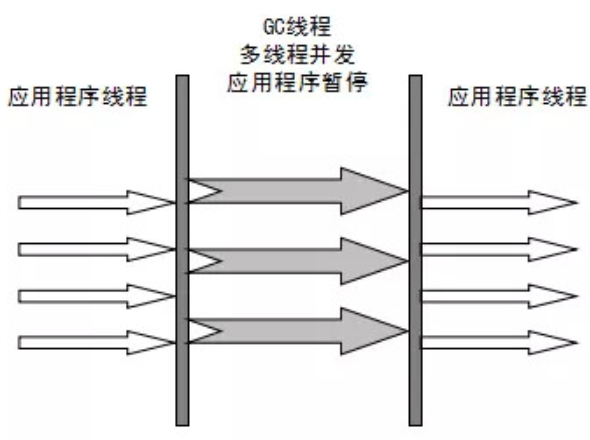
新生代和老年代能用：新生代采用复制算法，老年代采用标记-整理算法。



**1.3 ParNew收集器(-XX:+UseParNewGC)**

ParNew收集器其实**跟Parallel收集器很类似**，区别主要在于它可以和CMS收集器配合使用，他作为年轻代的垃圾收集，CMS是老年代的垃圾收集。

**新生代采用复制算法**

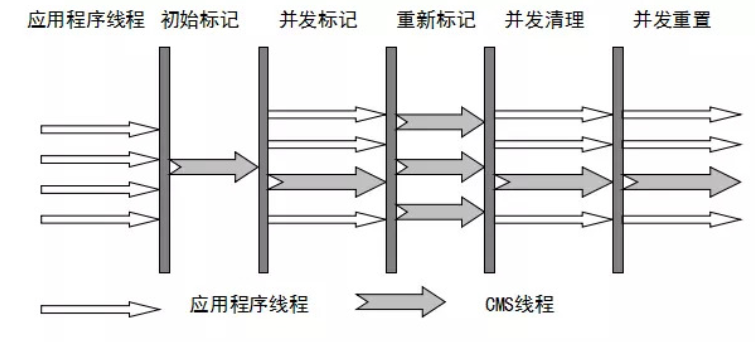


**1.4 CMS收集器(-XX:+UseConcMarkSweepGC(old))**

**CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器（只用在老年代）。它非常符合在注重用户体验的应用上使用，它是HotSpot虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。**

从名字中的**Mark Sweep**这两个词可以看出，CMS收集器是一种 **“标记-清除”算法**实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

* **初始标记：** 暂停所有的其他线程(STW)，并记录下gc roots**直接能引用的对象**，**速度很快**。
* **并发标记：** 并发标记阶段就是从GC Roots的直接关联对象开始遍历整个对象图的过程， 这个过程耗时较长但是不需要停顿用户线程， 可以与垃圾收集线程一起并发运行。因为用户程序继续运行，可能会有导致已经标记过的对象状态发生改变。
* **重新标记：**也会STW， 重新标记阶段就是为了修正并发标记期间因为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，**这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短。主要用到三色标记里的增量更新算法(见下面详解)做重新标记。**
* **并发清理：** 开启用户线程，同时GC线程开始对未标记的区域做清扫。这个阶段如果有新增对象会被标记为黑色不做任何处理(见下面三色标记算法详解)。
* **并发重置：**重置本次GC过程中的标记数据。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面几个明显的缺点：

* 对CPU资源敏感（会和服务抢资源）；
* 无法处理**浮动垃圾**(在并发标记和并发清理阶段又产生垃圾，这种浮动垃圾只能等到下一次gc再清理了)；
* 它使用的回收算法-**“标记-清除”算法**会导致收集结束时会有**大量空间碎片**产生，当然通过参数-XX:+UseCMSCompactAtFullCollection可以让jvm在执行完标记清除后再做整理
* 执行过程中的不确定性，会存在上一次垃圾回收还没执行完，然后垃圾回收又被触发的情况，**特别是在并发标记和并发清理阶段会出现**，一边回收，系统一边运行，也许没回收完就再次触发**full gc**，也就是"**concurrent mode failure**"**(并发失败)**，**此时会进入stop the world，用serial old垃圾收集器来回收**

**初始标记和重新标记的时候JVM是STW的，fullGC时候再次触发fullGC时候会STW**

**CMS的相关核心参数**

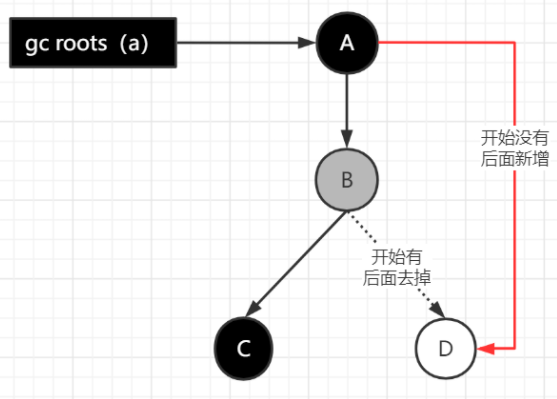
1. -XX:+UseConcMarkSweepGC：启用cms
2. -XX:ConcGCThreads：并发的GC线程数
3. -XX:+UseCMSCompactAtFullCollection：FullGC之后做压缩整理（减少碎片）
4. -XX:CMSFullGCsBeforeCompaction：多少次FullGC之后压缩一次，默认是0，代表每次FullGC后都会压缩一次
5. **-XX:CMSInitiatingOccupancyFraction: 当老年代使用达到该比例时会触发FullGC（默认是92，这是百分比）**
6. -XX:+UseCMSInitiatingOccupancyOnly：只使用设定的回收阈值(-XX:CMSInitiatingOccupancyFraction设定的值)，如果不指定，JVM仅在第一次使用设定值，后续则会自动调整
7. -XX:+CMSScavengeBeforeRemark：在CMS GC前启动一次minor gc，降低CMS GC标记阶段**(**也会对年轻代一起做标记，如果在minor gc就干掉了很多对垃圾对象，标记阶段就会减少一些标记时间**)**时的开销，一般CMS的GC耗时 80%都在标记阶段
8. -XX:+CMSParallellnitialMarkEnabled：表示在初始标记的时候多线程执行，缩短STW
9. -XX:+CMSParallelRemarkEnabled：在重新标记的时候多线程执行，缩短STW;

**CMS底层算法实现**

三色标记

在并发标记的过程中，因为标记期间应用线程还在继续跑，对象间的引用可能发生变化，多标和漏标的情况就有可能发生。Gcroots可达性分析遍历对象过程中遇到的对象， 按照“是否访问过”这个条件标记成以下三种颜色**：**

* 黑色： 表示对象已经被垃圾收集器访问过， 且这个对象的所有引用都已经扫描过。
* 灰色： 表示对象已经被垃圾收集器访问过， 但这个对象上至少存在一个引用还没有被扫描过。
* 白色： 表示对象尚未被垃圾收集器访问过。



问题描述

CMS在并发清理的时候，由于应用线程没有停，所以有浮动垃圾和漏标的情况。对此CMS用读写屏障来保护。

**多标-浮动垃圾**

在并发标记过程中，如果由于方法运行结束导致部分局部变量(gcroot)被销毁，这个gcroot引用的对象之前又被扫描过(被标记为非垃圾对象)，那么本轮GC不会回收这部分内存。这部分本应该回收但是没有回收到的内存，被称之为“**浮动垃圾**”。浮动垃圾并不会影响垃圾回收的正确性，只是需要等到下一轮垃圾回收中才被清除。

**漏标-读写屏障**

漏标会导致被引用的对象被当成垃圾误删除，这是严重bug，必须解决，有两种解决方案： **增量更新（Incremental Update） 和原始快照（Snapshot At The Beginning，SATB） 。**

**增量更新**就是当黑色对象插入新的指向白色对象的引用关系时， 就将这个新插入的引用记录下来， 等并发扫描结束之后， 再将这些记录过的引用关系中的黑色对象为根， 重新扫描一次。 这可以简化理解为， **黑色对象一旦新插入了指向白色对象的引用之后， 它就变回灰色对象了**。

**原始快照**就是当灰色对象要删除指向白色对象的引用关系时， 就将这个要删除的引用记录下来， 在并发扫描结束之后， 再将这些记录过的引用关系中的灰色对象为根， 重新扫描一次，这样就能扫描到白色的对象，将白色对象直接标记为黑色(**目的就是让这种对象在本轮gc清理中能存活下来，待下一轮gc的时候重新扫描，这个对象也有可能是浮动垃圾**)

**简单的来说**

**增量更新：就是把并发清理时的变化用一个map记录下来后面维护**。

**原始快照：就是记录快照，后面来维护。**

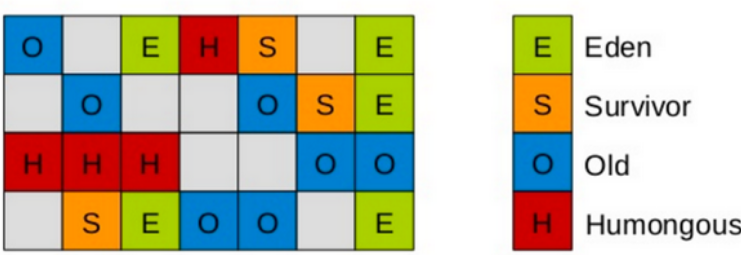
**未来的垃圾收集器**

**G1收集器(-XX:+UseG1GC)**

主要针对大容量内存的机器 ，8GB以上的**堆**内存(建议值)。是复制清除算法

要点：

1. 打破了以前年轻代老年代的物理规划。将Java堆划分为多个大小相等的独立区域（**Region**），JVM目标是不超过2048个。
2. 一个Region可能之前是年轻代，如果Region进行了垃圾回收，之后可能又会变成老年代，也就是说Region的区域功能可能会动态变化。
3. 3个区之外多了一个大对象区
4. G1收集器很强大的一个功能， 设置不同的期望停顿时间
5. region之间用复制算法



使用案例：

Kafka类似的支撑高并发消息系统大家肯定不陌生，对于kafka来说，每秒处理几万甚至几十万消息时很正常的，一般来说部署kafka需要用大内存机器(比如64G)，也就是说可以给年轻代分配个三四十G的内存用来支撑高并发处理，这里就涉及到一个问题了，我们以前常说的对于eden区的young gc是很快的，这种情况下它的执行还会很快吗？很显然，不可能，因为内存太大，处理还是要花不少时间的，假设三四十G内存回收可能最快也要几秒钟，按kafka这个并发量放满三四十G的eden区可能也就一两分钟吧，那么意味着整个系统每运行一两分钟就会因为young gc卡顿几秒钟没法处理新消息，显然是不行的。那么对于这种情况如何优化了，我们可以使用G1收集器，设置 -XX:MaxGCPauseMills 为50ms，假设50ms能够回收三到四个G内存，然后50ms的卡顿其实完全能够接受，用户几乎无感知，那么整个系统就可以在卡顿几乎无感知的情况下一边处理业务一边收集垃圾。

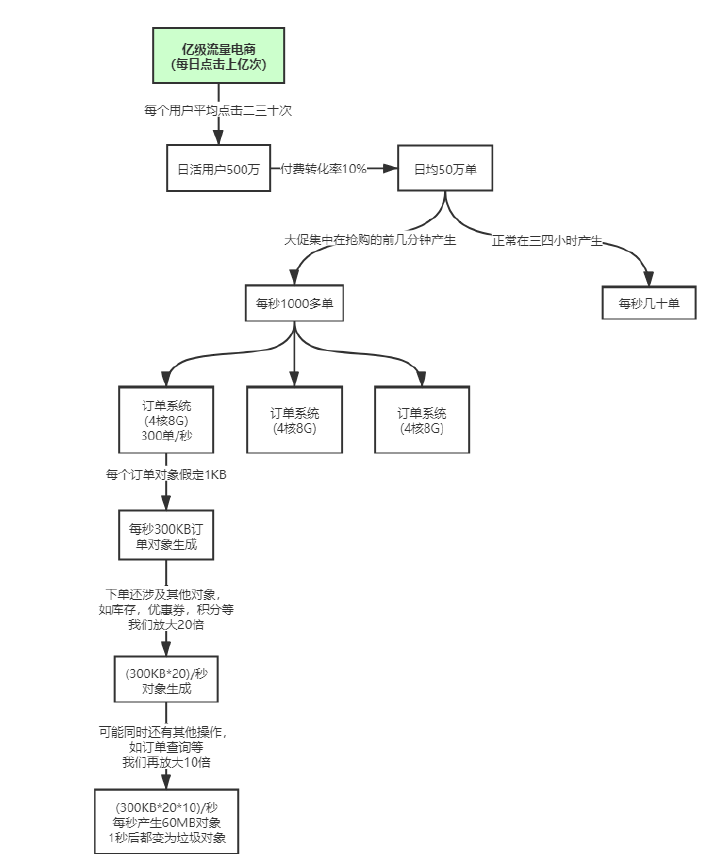
G1天生就适合这种大内存机器的JVM运行，可以比较完美的解决大内存垃圾回收时间过长的问题。

**ZGC收集器(-XX:+UseZGC)**

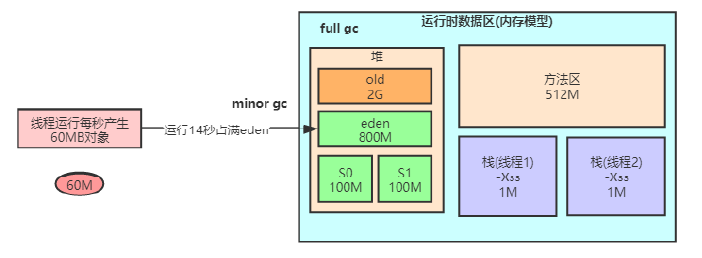
* 支持TB量级的堆。
* 最大GC停顿时间不超10ms
* 不分代，分为小中大region区

**案例分析及实例**

**案例**



对比苏宁，亿级流量电商，峰值TPS处理订单的能力是900TPS



现象：

4C8G的机器，堆分配的3G（老年代2G年轻代1G其中伊甸区800M，两个servoor区100M）。

大促抢购期，老年代不断的增长，每过几分钟就会fullGC（会STW）一次，并且每次fullGC回收率都很高。

疑问：

1. 为什么会频繁的fullGC
2. 老年代的对象应该都是稳定的，不是朝生夕失的对象，为什么没有在年轻代就干掉，会到老年代来。

原因：

线程每秒产生60M对象，14秒占满。触发yongGC，GC那一秒，还有60M存活的对象，此时按照GC的机制会把这些对象移到Seoove区，然后清空伊甸区。但是，60M的对象超过了S0区的50%。触发了**对象动态年龄判断**机制，直接放进了老年代。从而产生了频繁fullGC，并且fuGC能清理掉老年代很大一块空间。

解决方案：

扩大年轻代，把servoor区扩大，让对象达不到servoor区50%，从而不触发对象动态年龄判断机制。

**实例**

前置介绍：

我们生产上前台（开放给商家的）机器是4C4G。垃圾收集采用的是parnew+CMS。

商家报名参与活动，一开始只有页面填写报名，后来增加了批量导入功能，也就是商家参与一个秒杀或者拼购活动的商品可以通过exl一次性导进去，一个exl的上限是1W条，也就是一次能导1W个SKU，我们最初的方案是文件上传解析后起一个异步的线程去做校验和入库。

现象：

批导功能上线一小段时间以后，大促前夕，前台JBOSS（给商家用的几台JBOSS）有的机器会频繁的fullGC基本上是几分钟就一次，同时伴随着STW（系统出现各种超时异常和告警）。我们把gc日志拉下来看，发现fullGC回收的老年代里的回收率很高。

正常来说，fullGC不应该这么频繁，应该一个小时以上甚至一天。同时老年代里的东西不应该回收率这么高，因为回收率高的对象也就是朝生夕失的对象应该在年轻代就会干掉不应该进入老年代。

问题定位：

我们拉了一下heapDump文件分析了一下，发现里面大量的批导报名商品对象。报名的商品应该导完入库就销毁了，为什么会进入老年代呢。

首先我们研究了一下业务场景，大促前夕活动报名的时候，基本上是运营发布好了活动（比如双十一当天的活动），在群里发个通知，艾特下所有的商家，让所有商家来报名，在没有批导的时候，页面填报名效率很低。三千个的活跃商家，一次活动参与按照300家来算，页面报名的TPS是不会超过50的基本在二三十，系统不会有什么压力。但是上了批导以后，50个商家同时批导压力就来了，因为一次导入的是几千上W的商品（特别是每个商家一般不会改变自己大促的商品，exl都是准备好的，每次大促一到就导一次）。就算我们前台的分布式系统JBOSS是6台，由于请求有时候不是平均的，即使是平均，也基本上摊到了一次处理10个批导文件。

如果一条报名商品按照1K来估算，一个商家一次按最大导一万条，就是10M。即使不是每个商家都一次导一万条，也存在每个机器压力不一样的情况（在同一个时间点，有的可能只处理5个商家批导，有的可能要处理15个）。那压力大的机器，在一秒里面有50M的报名商品对象是很正常的。

问题就比较清楚了，为什么这些50M的商品对象没有在年轻代GC掉，不断的放进老年代，把老年代撑的最后fullGC，STW。

我们又研究了一下我们的JVM参数，用的parnew+CMS。4个G的机器，给JVM堆分配的是2个G，，采用的是默认配置：老年代三分之二 1.3，年轻代三分之一 0.7，年轻代里面伊甸园区8 survior1:1. survior查不多就是70M。

做minorGC的时候，那一刻正常应该把存活的对象，我们按照50M来算，从伊甸区挪到survior区，但是因为survior区太小了只有70M，超过了他空间的50%，触发了JVM的动态年龄判断机制，所以就直接被丢到了老年代。

fullGC触发的百分比也没有调默认是92%。当fullGC的时候再有大对象来的时候，就会发生selGC，整个应用STW（CMS在fullGC时只有两个阶段会STW，时间很短。但是如果fullGC时再来大对象触发fullGC，此时就会垃圾收集并发失败，转而用serloneGC，整个fullGC的过程都会STW，所以务必在下次大对象来之前fullGC完，比如把fullGC的触发条件调低，改为80%）。

临时解决方案：

针对这样的情况，我们把年轻代调大了，年轻代和老年代调成了1:1，fullGC频率明显降低，差不多一个小时一次。效果很明显，STW的问题基本没有了。也是证实了我们的判断是对的。

其实这种参数调优没有解决根本的问题：

1. 压力更大了怎么办（商家习惯了批导以后越来越少的人用页面报名活动了，转而大规模用批导，批导文件又是现成的），如果同时一台机器有20-30个批导，别说fullGC STW，OOM也就是眼前的事。
2. 多节点，在一某一刻压力不均衡怎么解决（分布式系统处理批导文件的时候，有时候也不是那么平均，有的请求多点，有的少点。可能有的节点压力很大，有的节点没有压力）。

最终解决方案：

1. 改变架构充分利用排队机制让分布式系统做消峰填谷，各个节点均摊压力

我们在架构上做了调整，前台上传只做exl的格式校验，然后上传云存储，不再直接起线程处理exl。库里记录一条待处理的记录，把云存储的objId记录下来，同时将表的主键ID丢到redis的list中。每个后台应用JBOSS（给采销报名机器也比较强，因为要跑一些任务，是4C8G的机器后台的配置比前台高）起一个线程，不断的用brpop阻塞队列的形式从redis的list中获取表的主键ID，获取到了后在去查库，获得云存储的objID将云存储上exl文件下下来处理。

这样做的好处是每个后台JBOSS在一个时间点永远只会处理一个批导文件，不用担心突然来几个大文件，在一个节点上把应用压的OOM。

如何保证队列中消息的可靠性：

1）用brpoplpush 这是个原子操作，pop的同时push进一个队列。消费者端处理完后删除，否则定时将备份队列的消息刷入原队列。

2）后台有监控，我们已经记录了文件的状态，如果长时间待处理是有问题的，设置了告警，上去手工处理。

1. 优化代码，提高单个批导文件的执行效率

我们处理批导文件时，对文件内的商品是拆到SPU级别起多线程校验和入库（提高单个批导文件的效率也就可以把这些对象尽快的gc掉）。因为要合并批导的结果（成功多少失败多少），开始使用了FutureTask，所有线程，对结果for循环，统计成功数和失败数。

这样有个不好的地方，如果一个线程没执行完，get的时候就会阻塞，其他线程都在等他执行完。

后来又改了一版，优化的方案是用countDownLanuch+automac原子类。成功失败的数量传到每个线程中去，每个线程根据自己的处理在上面加减。主线程根据线程数创建countDownLanuch，然后await。每个线程执行完后就countDown一下，记个数。当最后一个线程处理完后，主线程自动感知到，直接获取对象中的成功数量和失败数量。

总结：定位这种问题需要对业务很熟悉，JVM的调优可以临时解决一些问题。但是最终解决方案还是需要在架构和代码上下点功夫。

**总结**

**尽可能让对象都在新生代里分配和回收，尽量别让太多对象频繁进入老年代，避免频繁对老年代进行垃圾回收，同时给系统充足的内存大小，避免新生代频繁的进行垃圾回收。**

**常量池**

[完整笔记](https://note.youdao.com/ynoteshare/index.html?id=d8d6dc3589ffd9245d97bb7aa91af835&type=note&_time=1636297228530)

**Class常量池与运行时常量池**

1. 编译后的class文件，就可以理解为class常量池。
2. 这个文件被加载到内存的方法区就是运行时常量池

|  |
| --- |
| Java int a = 1; int b = 2; int c = "abcdefg"; int d = "abcdefg"; Public void computer（）{ } |

常量池中分为字面量（红色 ）和符号引用（蓝色）

**字面量**

**字面量就是指由字母、数字等构成的字符串或者数值常量**

字面量只可以右值出现，所谓右值是指等号右边的值，如：int a=1 这里的a为左值，1为右值。在这个例子中1就是字面量。

|  |
| --- |
| Java  int a = 1; int b = 2; int c = "abcdefg"; int d = "abcdefg"; |

**符号引用**

符号引用是编译原理中的概念，是相对于直接引用来说的。主要包括了以下三类常量：

* 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

上面的a，b就是字段名称，就是一种符号引用，还有Math类常量池里的 Lcom/tuling/jvm/Math 是类的全限定名，main和compute是方法名称，()是一种UTF8格式的描述符，这些都是符号引用。

这些常量池现在是静态信息，只有到运行时被加载到内存后，这些符号才有对应的内存地址信息，这些常量池一旦被装入内存就变成**运行时常量池**，对应的符号引用在程序加载或运行时会被转变为被加载到内存区域的代码的直接引用，也就是我们说的**动态链接了。例如，compute()这个符号引用在运行时就会被转变为compute()方法具体代码在内存中的地址，主要通过对象头里的类型指针去转换直接引用。**

**动态链接和静态链接**

类加载阶段（class常量池-》运行时常量池）的过程中将符号引用（一个个符号）转化为直接引用(链接地址)的叫静态链接。

比如类的static方法。

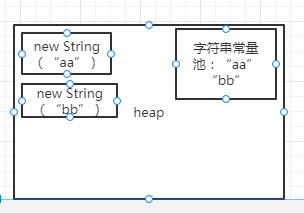
每一次运行时，将运行时常量池中的符号引用（一个个符号）转化为直接引用(链接地址)的叫动态链接。

比如实例方法的调用。

**字符串常量池**

**JDK1.8 字符串常量池，是在堆里，和元空间（以前的永久）没啥关系。**

**一个“aa”就在堆的字符串常量池里存一个，多个字符串相同，其实是对他的引用相同（内存地址）。new String(“aa”)就在堆里面字符串常量池之外再建一个，和堆里面那个常量池不是一个。**



**String常量池问题的几个例子**

示例1：

|  |
| --- |
| Java String s0="zhuge"; String s1="zhuge"; String s2="zhu" + "ge"; System.out.println( s0==s1 ); //true System.out.println( s0==s2 ); //true |

分析：因为例子中的 s0和s1中的”zhuge”都是字符串常量，它们在编译期就被确定了，所以s0==s1为true；而”zhu”和”ge”也都是字符串常量，当一个字 符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所以s2也同样在编译期就被优化为一个字符串常量"zhuge"，所以s2也是常量池中” zhuge”的一个引用。所以我们得出s0==s1==s2；

示例2：

|  |
| --- |
| Java  String s0="zhuge"; String s1=new String("zhuge"); String s2="zhu" + new String("ge"); System.out.println( s0==s1 );　　// false System.out.println( s0==s2 )；　 // false System.out.println( s1==s2 );　　// false |

分析：用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们有自己的地址空间。

s0还是常量池 中"zhuge”的引用，s1因为无法在编译期确定，所以是运行时创建的新对象”zhuge”的引用，s2因为有后半部分 new String(”ge”)所以也无法在编译期确定，所以也是一个新创建对象”zhuge”的引用;明白了这些也就知道为何得出此结果了。

示例3：

|  |
| --- |
| Java  String a = "a1";  String b = "a" + 1;  System.out.println(a == b); // true     String a = "atrue";  String b = "a" + "true";  System.out.println(a == b); // true     String a = "a3.4";  String b = "a" + 3.4;  System.out.println(a == b); // true |

分析：JVM对于字符串常量的"+"号连接，将在程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

示例4：

|  |
| --- |
| Java String a = "ab"; String bb = "b"; String b = "a" + bb; System.out.println(a == b); // false |

分析：JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给b。所以上面程序的结果也就为false。

示例5：

|  |
| --- |
| Java String a = "ab"; final String bb = "b"; String b = "a" + bb; System.out.println(a == b); // true |

分析：和示例4中唯一不同的是bb字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + bb和"a" + "b"效果是一样的。故上面程序的结果为true。

示例6：

|  |
| --- |
| Java String a = "ab"; final String bb = getBB(); String b = "a" + bb;  System.out.println(a == b); // false  private static String getBB()  {   return "b";   } |

分析：JVM对于字符串引用bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为b，故上面 程序的结果为false。

**关于String是不可变的**

通过上面例子可以得出得知：

|  |
| --- |
| Java  String s = "a" + "b" + "c"; //就等价于String s = "abc"; String a = "a"; String b = "b"; String c = "c"; String s1 = a + b + c; |

　　s1 这个就不一样了，可以通过观察其**JVM指令码**发现s1的"+"操作会变成如下操作：

|  |
| --- |
| Java StringBuilder temp = new StringBuilder(); temp.append(a).append(b).append(c); String s = temp.toString(); |

**最后再看一个例子**：

|  |
| --- |
| Java //字符串常量池："计算机"和"技术" 堆内存：str1引用的对象"计算机技术"  //堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用 String str2 = new StringBuilder("计算机").append("技术").toString(); //没有出现"计算机技术"字面量，所以不会在常量池里生成"计算机技术"对象 System.out.println(str2 == str2.intern()); //true //"计算机技术" 在池中没有，但是在heap中存在，则intern时，会直接返回该heap中的引用  //字符串常量池："ja"和"va" 堆内存：str1引用的对象"java"  //堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用 String str1 = new StringBuilder("ja").append("va").toString(); //没有出现"java"字面量，所以不会在常量池里生成"java"对象 System.out.println(str1 == str1.intern()); //false //java是关键字，在JVM初始化的相关类里肯定早就放进字符串常量池了  String s1=new String("test");  System.out.println(s1==s1.intern()); //false //"test"作为字面量，放入了池中，而new时s1指向的是heap中新生成的string对象，s1.intern()指向的是"test"字面量之前在池中生成的字符串对象  String s2=new StringBuilder("abc").toString(); System.out.println(s2==s2.intern()); //false //同上 |

**八种基本类型的包装类和对象池**

java中基本类型的包装类的大部分都实现了常量池技术(严格来说应该叫**对象池，**在堆上)，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。因为一般这种比较小的数用到的概率相对较大。

|  |
| --- |
| Java  public class Test {   public static void main(String[] args) {  //5种整形的包装类Byte,Short,Integer,Long,Character的对象，   //在值小于127时可以使用对象池   Integer i1 = 127; //这种调用底层实际是执行的Integer.valueOf(127)，里面用到了IntegerCache对象池  Integer i2 = 127;  System.out.println(i1 == i2);//输出true    //值大于127时，不会从对象池中取对象   Integer i3 = 128;  Integer i4 = 128;  System.out.println(i3 == i4);//输出false     //用new关键词新生成对象不会使用对象池  Integer i5 = new Integer(127);   Integer i6 = new Integer(127);  System.out.println(i5 == i6);//输出false    //Boolean类也实现了对象池技术   Boolean bool1 = true;  Boolean bool2 = true;  System.out.println(bool1 == bool2);//输出true    //浮点类型的包装类没有实现对象池技术   Double d1 = 1.0;  Double d2 = 1.0;  System.out.println(d1 == d2);//输出false   } } |