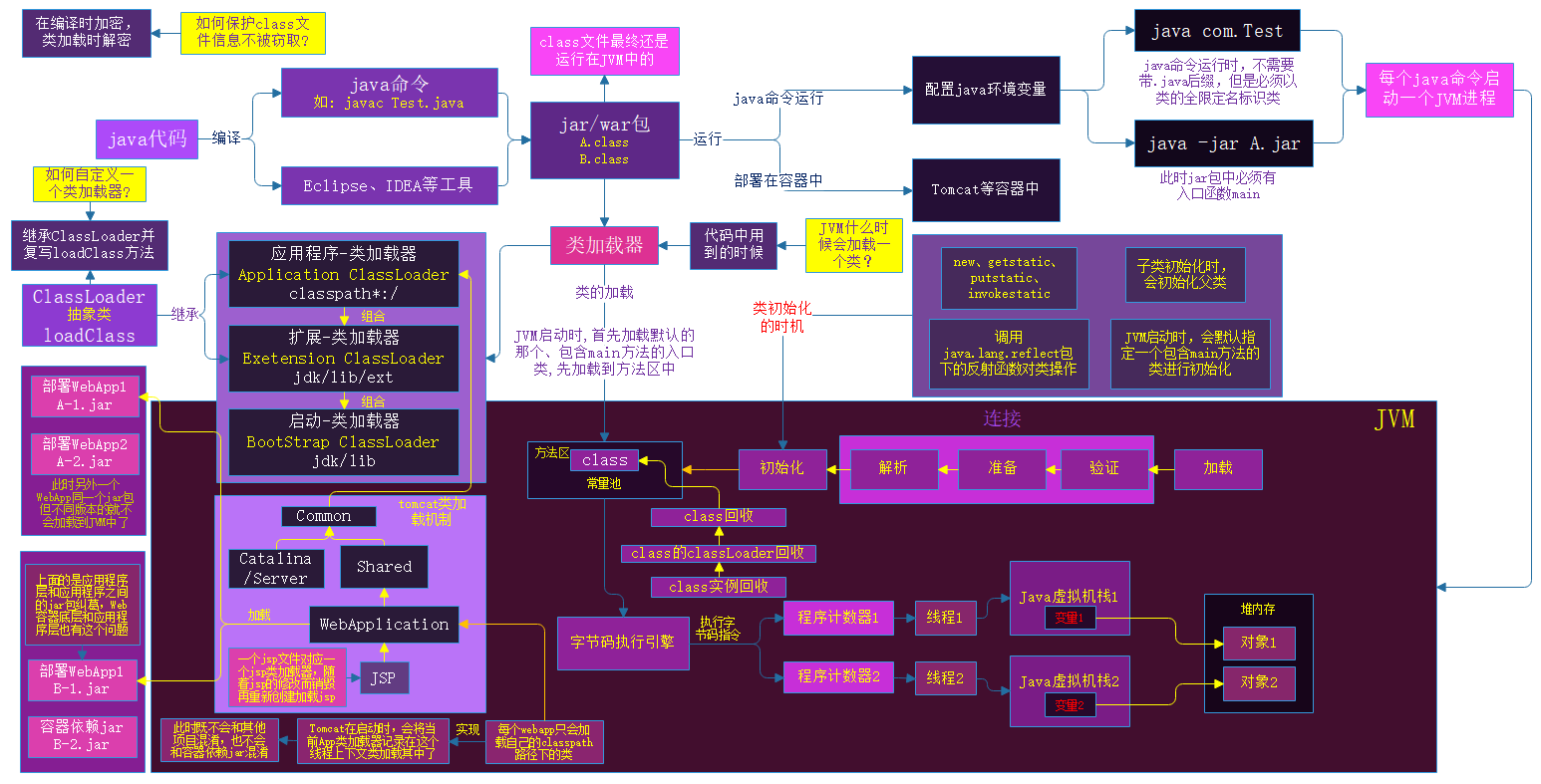
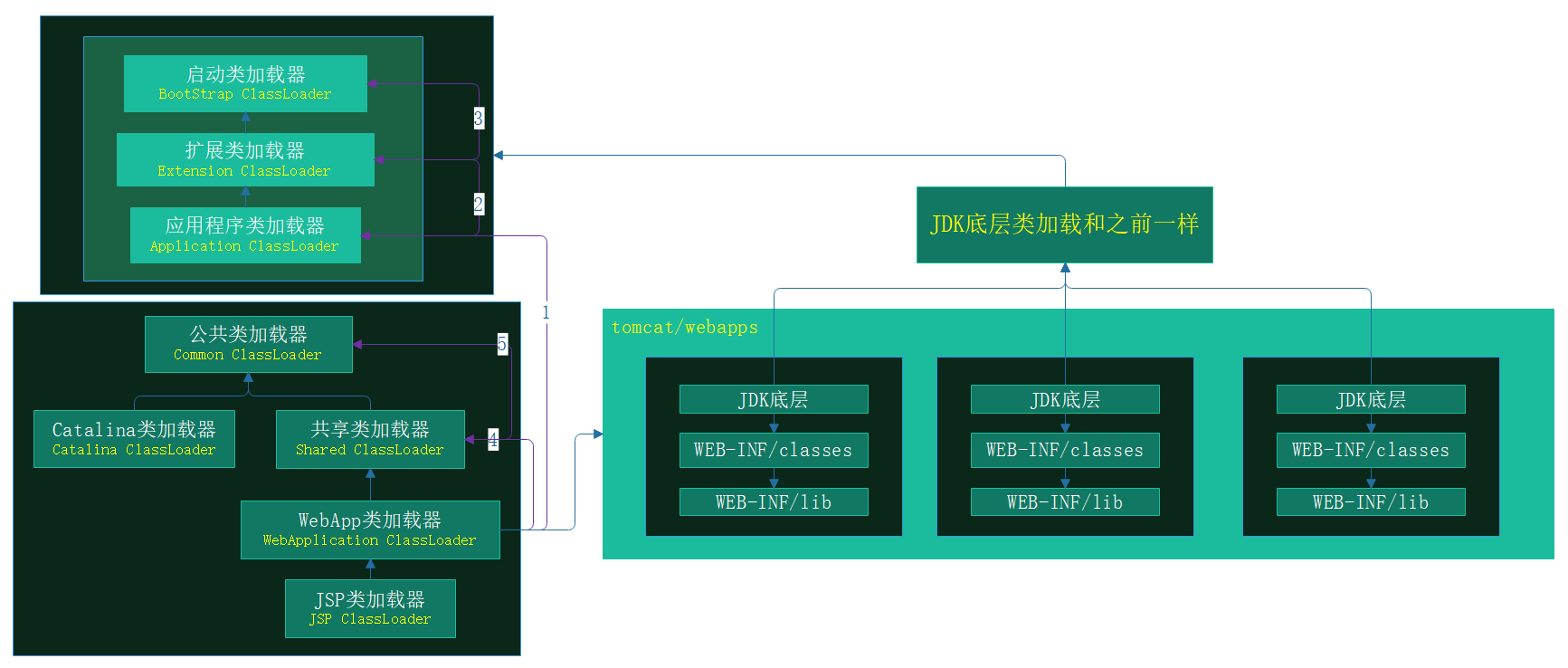
# 上帝视角下的JVM

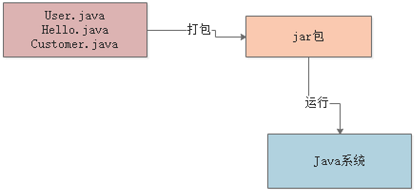




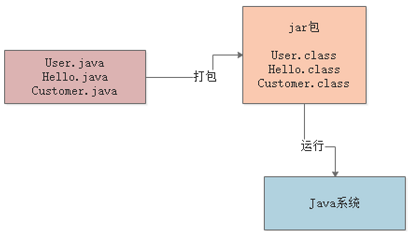
## Java代码到底是如何运行的

### 常规代码部署流程

将eclipse、IDEA等开发工具中的java代码打包成jar或war文件，最简单的部署方式是部署在tomcat这类容器当中，也可以通过java命令来运行一个jar包中的代码，如下图所示：



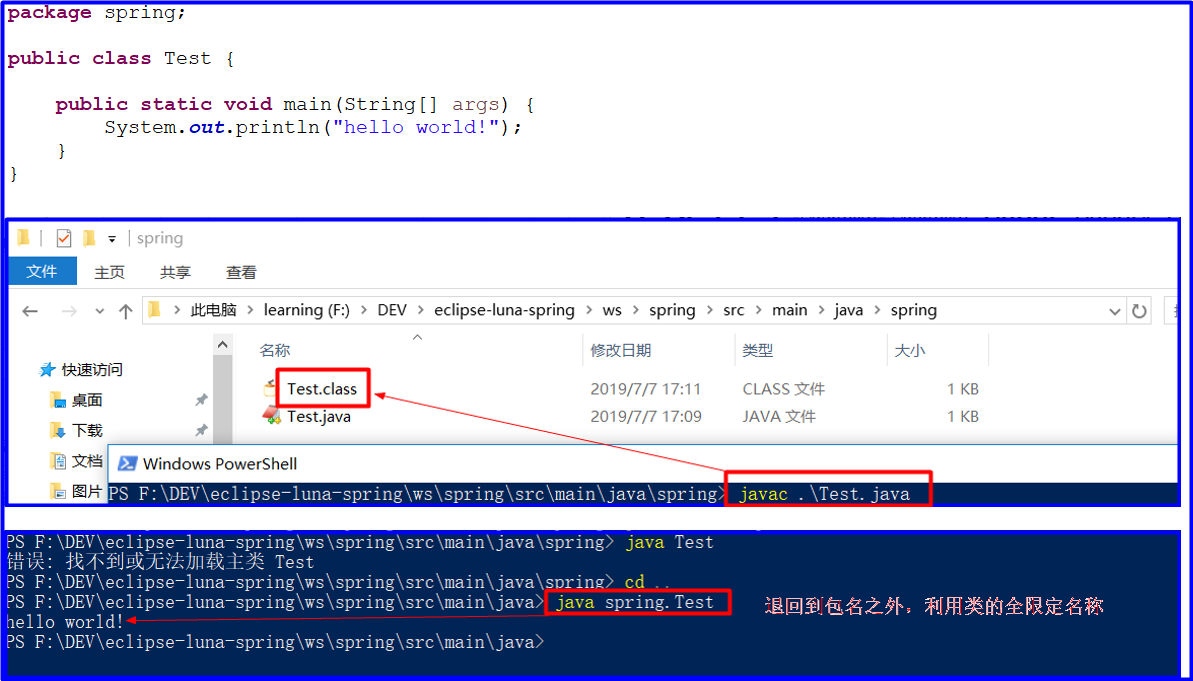
上图过程当中有一个非常重要的过程，就是编译，发生在打包过程当中，会将java文件编译成.class字节码文件，而”.class”文件才是真正可以被运行起来的，如下图所示：



### class文件是如何运行的

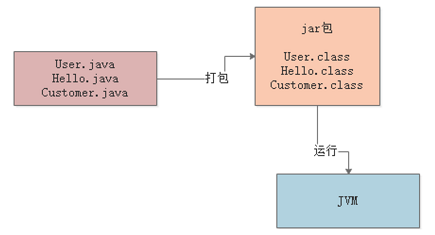
（1）利用java命令运行

首先需要配置好Java环境变量，Java执行class文件，对包路径是强依赖的，就是会在用户的当前路径之下，以类的全限定名解析路径，然后找到class文件进行运行；另外，如果类无包名，需要保证在环境变量classpath变量配置,;，表示可以在当前路径下寻找class文件，利用命令执行java如下图所示：



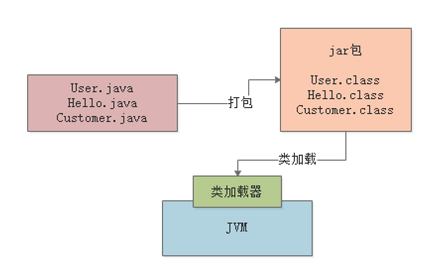
（2）JVM进程的来由

由上可知，能够运行的是.class文件，而运行它需要执行java –jar等命令，一旦使用java命令，此时就会启用一个JVM进程，演变为如下图所示：



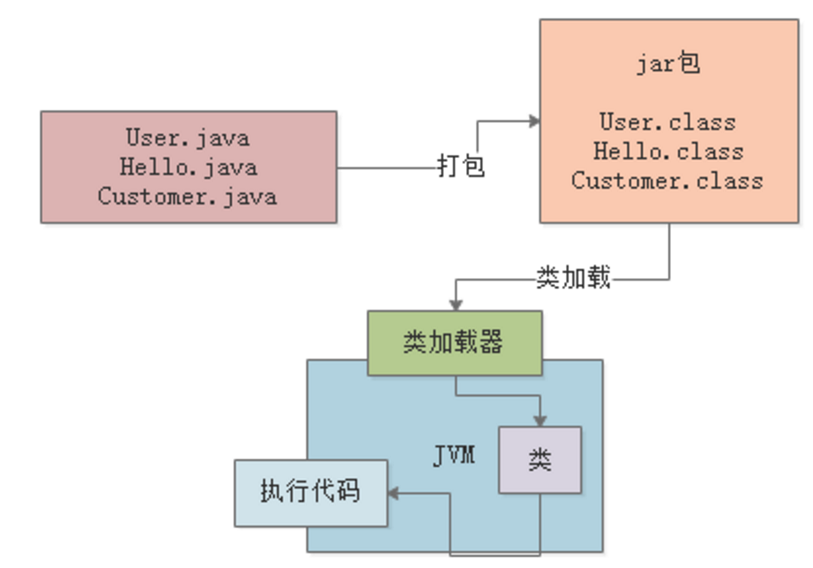
（3）类加载器的引入

既然已经知道一个java类，只有经过编译后得到的class文件才能被运行，执行编译操作往往在打包的过程当中，当然一些开发工具如eclipse会即时编译；运行class的是JVM，每当执行一个java xxx命令时，都会启动一个JVM进程，而class文件的运行都是在JVM中的，但是此时的问题是，class文件是如何被弄到JVM中的，完成这一个操作的执行者就是类加载器，如下图所示：



（4）代码的执行

此时，JVM会根据自己的字节码执行引擎，来执行那些我们已经加载到内存中的那些类了，类加载器会根据需要将类加载到内存中，整体流程如下图所示：



## JVM类加载机制的轮番轰炸

### JVM在什么情况下会加载一个类

很简单，就是代码中用到某个类的时候就会去加载该类到JVM中，其中JVM进程启动时，首先会将包含main方法的类加载到JVM中，然后遇到需要什么类就去加载什么类到JVM中，如下图所示：



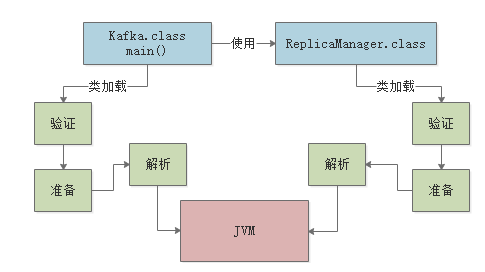
### 类的验证、准备、解析和初始化

类的连接，包括三个阶段，分别是验证、准备和解析。类的加载总共分为五个阶段，分别是加载、验证、准备、解析、初始化，其中验证、准备、解析为类的连接，加载过程是类加载过程最开始的一个子过程，主要作用就是通过类的全限定名获取到该类的二进制字节流，然后开始进行类的连接处理。

**验证**，虽然JVM可以运行class文件，但是没有严格按照JVM规范的class文件、被窜改过的class文件显然在JVM中不能被运行，所以加载到JVM之前必须对class文件进行验证是一件合情合理的事情。

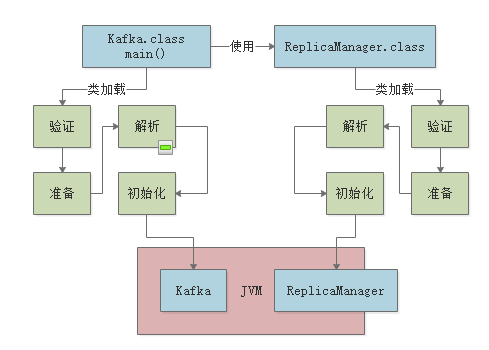
**准备**，该过程是类的连接过程中，最重要的一个过程，在准备阶段会给类分配一定的内存空间，并且会给类成员变量赋零值，此处能被赋予零值的成员变量要求比较特殊，必须得要是八大基本数据类型或者是String，并且它们都没有被final修饰。

**解析**，该过程主要作用是将符号引用转换为直接引用。



**初始化**，首先如果是八大基本数据类型或String，此时如果被final修饰的话，会在准备阶段就为它们赋初始值，因为这些基本数据类型在方法区中的常量池中比较容易事先获取到，如果没有被final修饰，则这些变量都在准备阶段赋零值，只有在初始化时才被赋予真实值。如果不是八大基本数据类型或者是String，那这些变量不管是否被final修饰，都只有在初始化时，才被赋予真实值。

该结论可以方便理解单例设计模式当中，饿汉模式的两种写法：直接赋值和在静态代码块中赋值，得到的效果是一样的，因为引用被赋值都是在初始化时期，并不会因为赋值语句写在静态代码块中，而对类的加载顺序有所改变。



### 触发类初始化的四种场景

第一，执行了四种字节码指令，new，表示new一个对象时;getstatic、putstatic和invokestatic，分别表示获取一个静态变量的值、设置一个静态变量的值以及执行一个静态方法。

第二，当要初始化一个类时，如果该类是继承了父类，那么该类的父类会优先被初始化，也就是说当一个即将要被初始化的类的父类会被初始化。

第三，当一个JVM进程启动时，会将包含了main方法的类初始化，也就是说包含了main方法的类会被初始化；JVM启动时，常见的方式是执行main函数，就有限加载那一个类就是。

第四，调用了jdk反射类相关的方法来操作字节码对象。

### 类加载器和双亲委派模式

（1）类加载器

**启动类加载器**-BootStrap ClassLoader，该类加载器最特殊，是属于JVM的一部分，采用c++编写，其他类加载都是位于JVM之外且都是用java语言编写；主要作用就是将java目录下的lib包中的jar包加载到JVM中。

**扩展类加载器**-Extension ClassLoader，该类加载器主要用来加载lib/ext目录下的jar包到JVM中。

**应用程序类加载器**-Application ClassLoader,该类加载器主要用来加载classpath路径下的java类，简单理解为加载用户编写的java类。

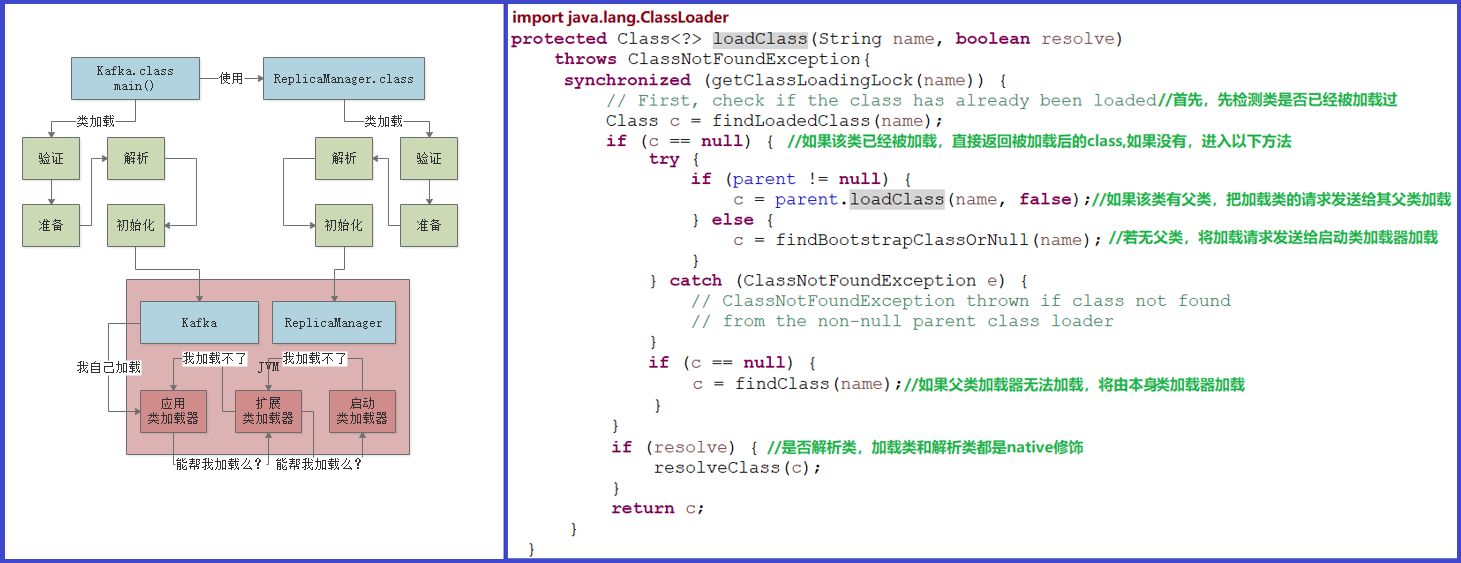
当然类加载器可以自定义，但是都需要继承抽象类ClassLoader。

（2）双亲委派模式

双亲委派模式是层级化的，因为类加载器除了最顶层的启动类加载器，其他类加载器如扩展类加载器、应用程序类加载器都是有父类的，并且子类都是复用父类功能的，这里的父类和子类严格说是组合关系，而不是真正继承的父子类。

双亲委派模式就是，当一个类加载请求过来之后，接收到类加载请求的类加载器，第一反应就是二话不说就发给自己的父类去加载，当父类发现加载不了，才依次打回给子类去加载。

这种双亲委派模式好处当然很多，比如你想自己写一个和java底层一样的类，发现加载不了，这是因为类加载器在加载你的类的时候，已经加载了java底层的类了；另外也可以得到另外一个结论，就是Object类一定是最早被加载的类，如下为类加载器的源码示例：



### 类加载器、类加载机制、加载区别

类加载器是一个代码模块，实现通过一个类的权限定名称获取到描述该类的二进制字节流的一个功能点，是一个小范围的点；而类加载机制是将描述类的数据，从class文件加载到JVM中，通过验证、转化解析、初始化等过程，转换为JVM可以直接使用的Java对象，是一个整体大流程。

### 如何保护class文件信息不被窃取

在编译时，对字节码文件进行加密或者混淆处理，在类加载的过程当中，采用自定义的类加载器对字节码文件进行解密。

### 如何自定义一个类加载器

自己写一个类，继承抽象类ClassLoader，复写父类的方法loadClass，在方法中可以定义加载哪个路径下的类到JVM中。

### 如何判断动态部署和静态部署

动态部署就是常说的热部署，比如Tomcat容器，不需要停止再重启就能将代码自动运行起来的，就是动态部署，反之则是静态部署。

### Tomcat如何打破双亲委派模式

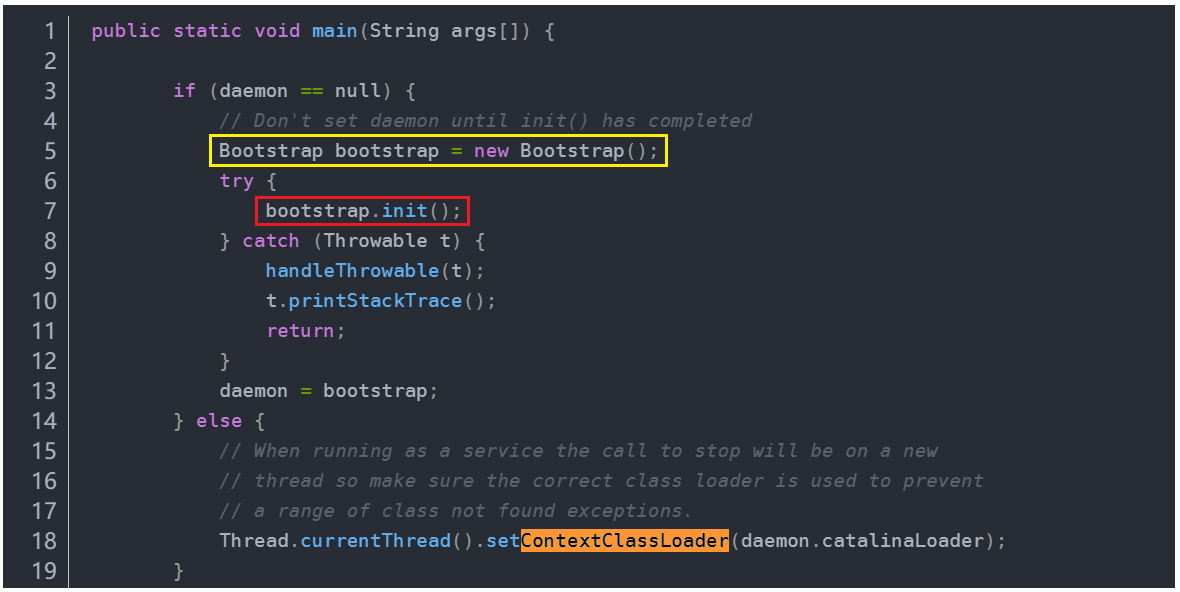
首先回答这个会提前需要先看下Tomcat为什么要打破双亲委派模式，首先Tomcat是个容器，容器当中当然不仅仅只是一个WebApp部署在上面，可能有好几个应用程序，围绕以下三个问题分析tomcat为什么、以及怎么打破双亲委派模式的：

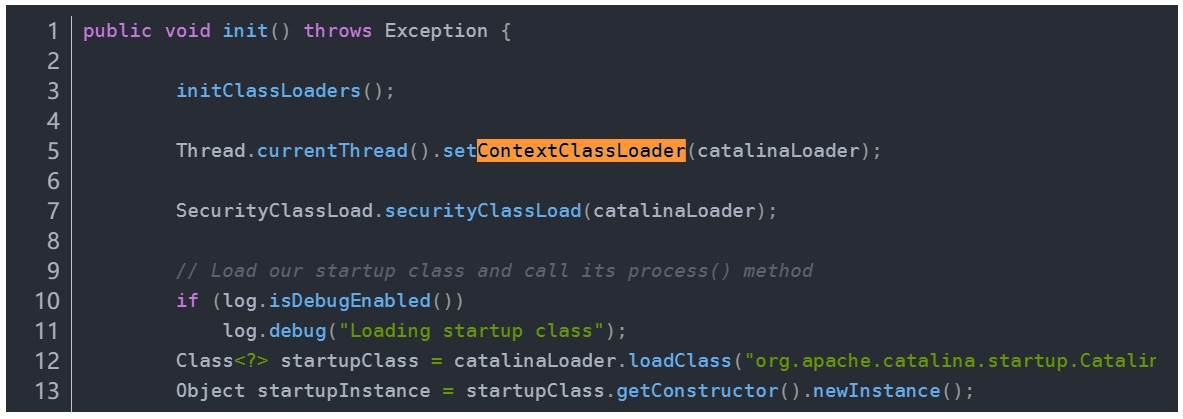
（1）对于一个jar包，有不同的版本，部署在Tomcat的webapps目录下的不同的应用程序，可能使用的是不同版本的jar包，此时如果使用默认的双亲委派模式，只会加载一种版本的jar包，因为一旦第一个版本的class加载到JVM中，JVM只认类的全限定名称，不同版本的jar包类的全限定名称当然一样，所以就导致其他版本的jar包就不能被加载到JVM中了，所以此时必须打破双亲委派模式。

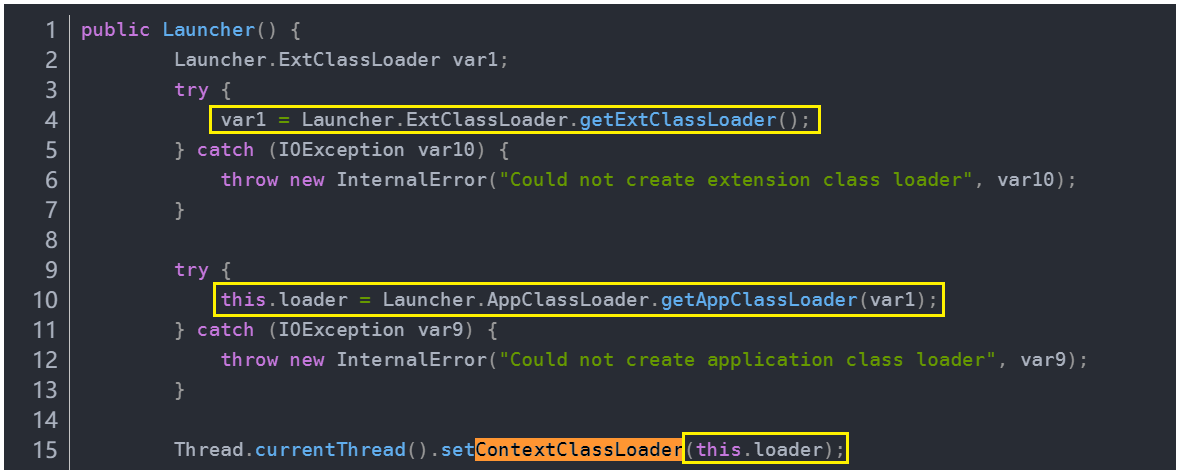
（2）应用程序依赖的类库，如JDK、mysql等，是应用程序运行时依赖的，但是Web容器它也有自己依赖的类库，出于安全考虑需要进行隔离，此时如果采用默认的双亲委派模式，同样会导致这两者中的一方的、一些类全限定名相同但版本不同的jar、无法加载或者漏加载。

（3）JSP文件一般部署之后，如果修改可以不需要重启就可以让修改生效，这个功能利用默认的双亲委派模式是无法做到的，因为JSP文件也好、Servlet也好，编译后都是class文件，加载到JVM中就是字节码文件，如果JSP文件被修改了，但是之前加载到JVM中的class对象还在，此时就不会再去加载了，但是如果你不重新加载怎们能让修改后的JSP文件生效呢？所以这里，tomcat也打破了双亲委派模式，它会给每一个JSP文件都分配一个类加载器，一旦某个JSP文件修改了，此时就会卸载掉加载该JSP编译后的class的类加载器，此时就会重新加载修改后的JSP文件编译后的class文件到JVM中，这整个过程就完成了所谓的热部署。

现在虽然已经知道了Tomcat为什么要打破双亲委派模式了，那具体Tomcat是采取什么方案解决上面这几个问题的呢，这里就要使用到线程上下文类加载器了，如下为Tomcat启动时的部分代码：







可以看到上面先创建启动类加载器，然后在初始化操作中将线程上下文类加载器赋值为catalina类加载器，这是最初初始化设置的线程上下文中的类加载器，而在启动时即Launcher函数中，此时当然就是启动部署在Tomcat容器中的某个应用程序了，它会将当前应用程序类加载器设置到线程上下文类加载器中，如此只要我们每次使用线程上下文类加载器，就可以打破双亲委派模式，避免向上级请求加载而直接使用当前的ApplicationContext类加载器加载，这样同jar包不同版本的问题也就解决了。

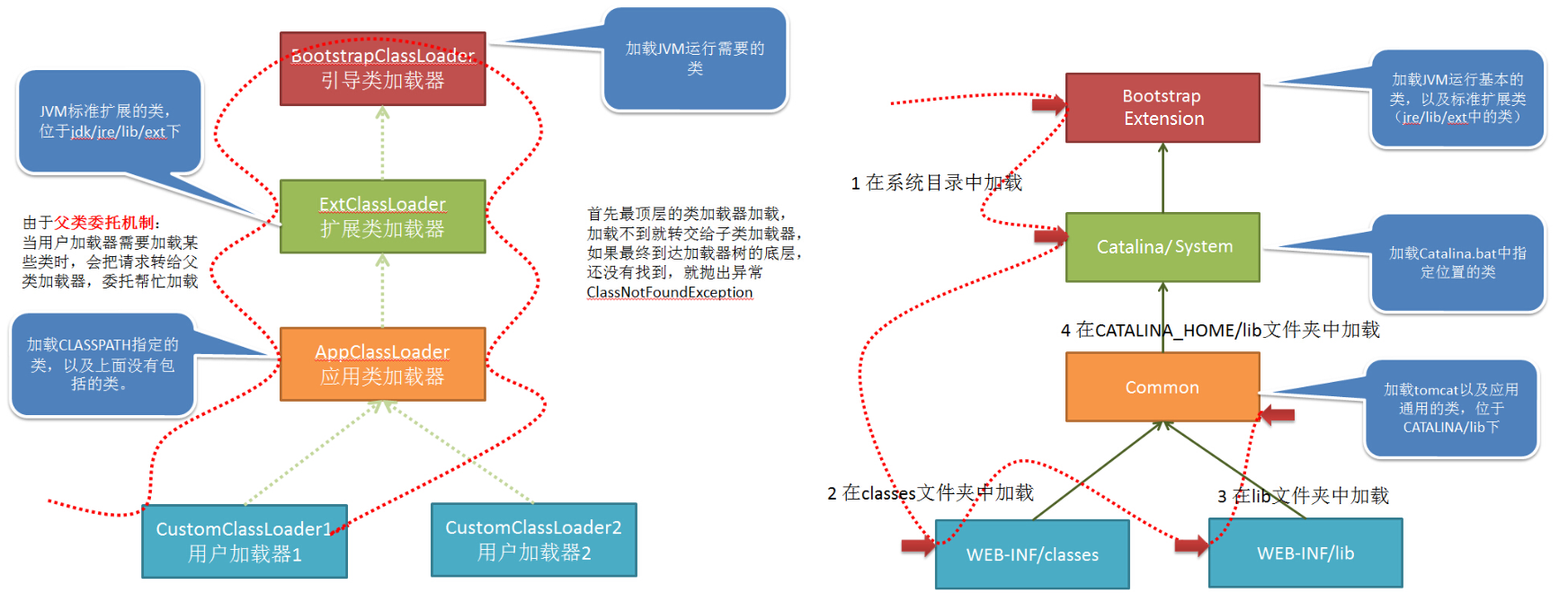
当然此时是在tomcat中，打破了双亲委派模式是因为在线程上下文中将父加载器给替换了，在没有打破双亲委派的场景中，线程上下文中存放的可能就是父类加载器了，这也是在子类加载器中使用父类记载器加载class的一种方法。

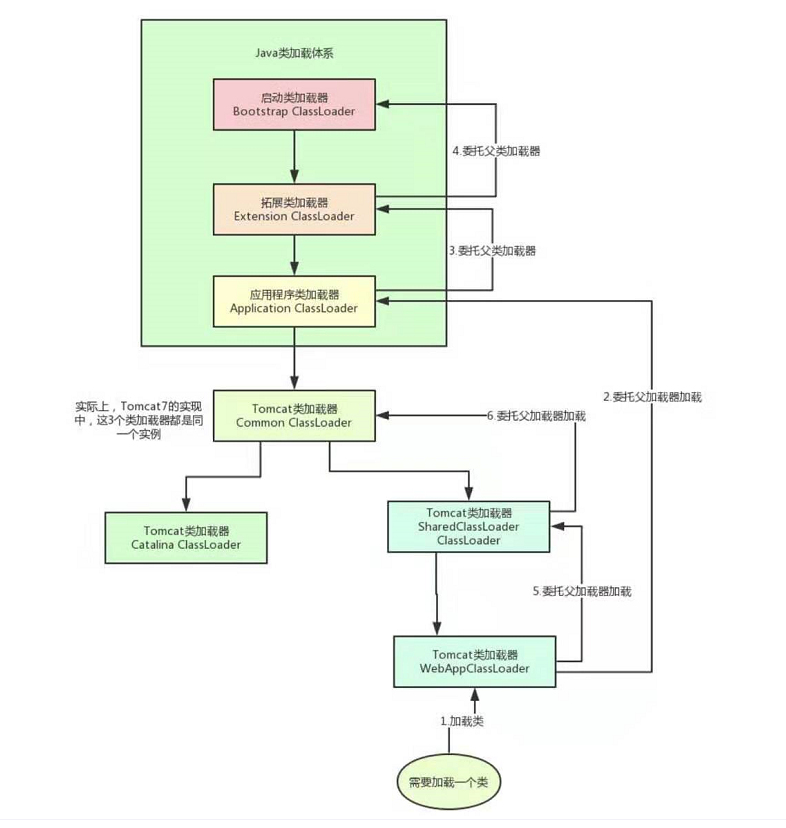
### Tomcat类加载的顺序

如下图为默认的双亲委派模式和Tomcat类加载机制加载顺序的对比图，可以看到tomcat类加载器加载的顺序为：Bootstrap和Extension负责的jar文件、catalina.sh中指定的jar文件、应用程序中WEB-INF/classes下的文件、/WEB-INF/lib中的jar、最后是Common类加载器。

所以从这里可以知道，虽然说每个应用程序都会有专属自己的类加载器，但是在加载自己之前，JDK的基本jar总需要先加载完成，然后此时因为是启动Tomcat，类似JVM，Tomcat也有Bootstrap等基础jar需要加载，接下来才每个应用程序互不干扰地加载。

从加载的顺序，可以知道为什么从WEB-INF/lib中的class文件中取出一个class反编译后放到工程中，就可以达到复写jar中类的效果了。

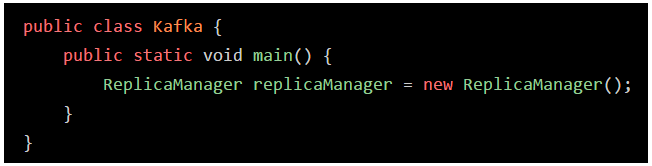


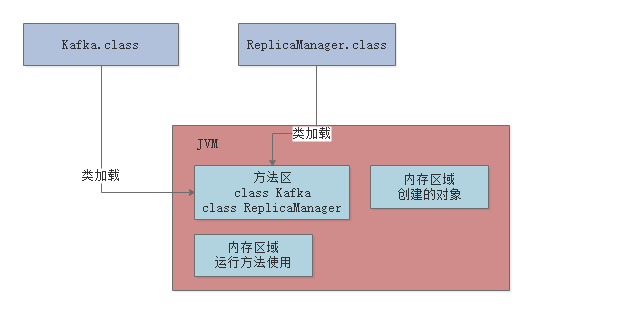


## 大面：JVM有哪些内存区域及作用

### 方法区

该方法区在jdk1.8时，名称变更为了“Metaspace”，可以为元数据空间，存放的是类相关的信息，如类变量等，包含常量池，是线程共享的区域之一。



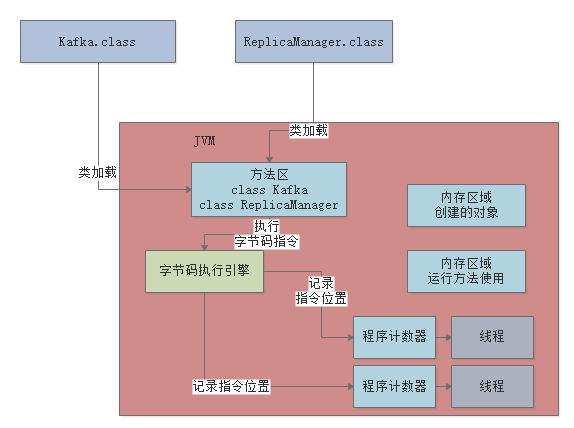


### 程序计数器

程序计数器并不是孤立存在的，因为Java代码是给人看的，只有编译成了class字节码文件才能被计算机识别。而class文件中都是由字节码指令组成，每一条字节码指令都会对应让计算机执行的一个动作，在JVM中，通过字节码执行引擎来执行字节码指令。

JVM是支持多线程的，多个线程之间是通过线程轮流切换来分配处理器的执行时间的，任意一个时刻，只能执行一个线程中的一条字节码指令，所以为了使线程切换回来后能够知道之前字节码指令执行的位置，所以每个线程都需要维护一个程序计数器。

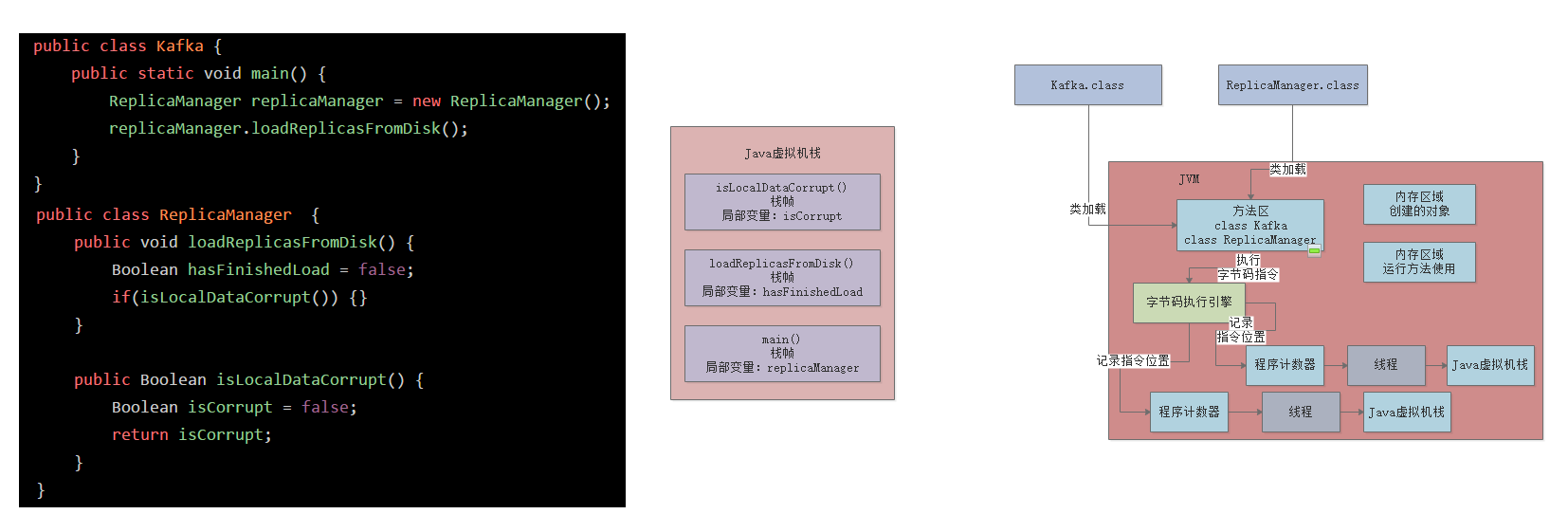
程序计数器记录的是字节码指令的地址，当程序计数器为空，表示当前指向了native方法了，该位置除了程序计数器为空，也是JVM中唯一一处没有规定OutOfMemoryError的区域。



### Java虚拟机栈

根据以上的分析可知，每个java文件需要编译成class字节码文件才能有被加载到JVM的可能，加载到JVM之后，字节码中的字节码指令由字节码执行引擎执行，每个线程负责维护一个程序计数器。而当一个方法中含有成员变量时，这些成员变量又该放到哪里呢？

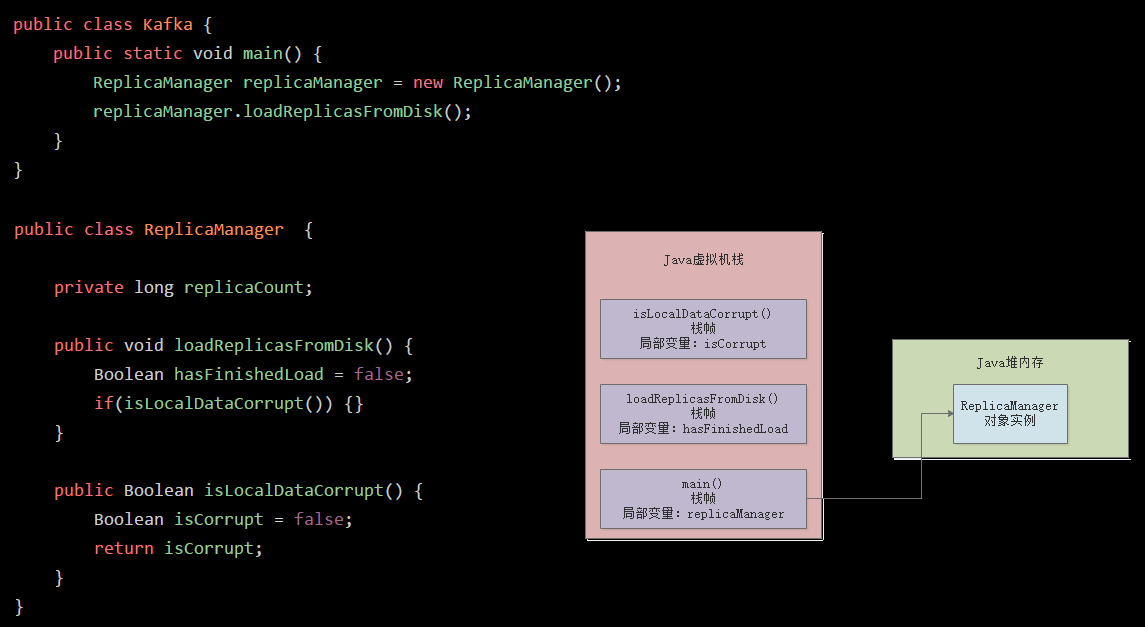
每个线程不光有自己的程序计数器，也有自己的Java虚拟机栈，因为每个线程都要执行方法啊。根据方法执行的顺序，依次会将方法中的信息如局部变量压入栈中，当最底层的方法最后压入栈中，每当一个方法执行完毕，该方法的信息将会弹出栈，如下图所示：



该区域规定了两种异常，就是StackOverflowError和OutOfMemoryError。

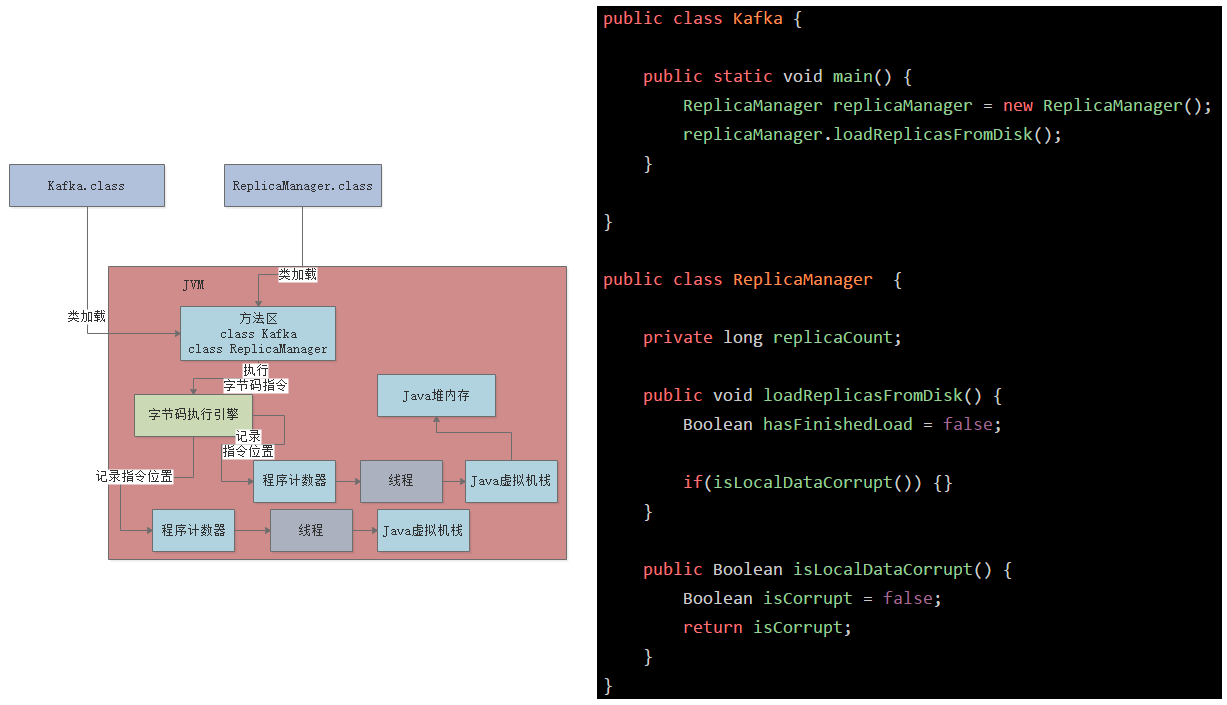
### Java堆内存

通过以上分析，连贯性越来越强。加载字节码中的数据到方法区中后，再通过字节码执行引擎依次执行字节码，通过一个程序计数器绑定一个线程执行方法，执行方法时为每一个方法创建栈帧并依次压入到Java虚拟机栈中，栈帧中涉及到局部变量存储，而局部变量如果接收到一个对象时，如下图中的replicaManager接受ReplicaManager对象时，此时new出来的对象存储在Java的堆内存中，而局部变量replicaManager则记录了堆内存中对象的地址。



### 流程小结

首先JVM进程启动**--->**将java类加载到内存(方法区)中**--->**有一个线程(这里是主线程main)执行方法**--->**线程会关联一个程序计数器，记录字节码执行引擎执行字节码指令的位置**--->**执行方法时，从最表层开始，只要遇到一个方法，就为它创建一个栈帧，用来存放局部变量和其他信息，并将栈帧压入到JVM栈中**--->**在执行方法的过程当中，如果碰到需要创建对象赋值语句，就会在JVM中的堆内存中创建一个对象，并利用局部变量来记录堆内存中该对象的地址**--->**然后从方法的最底层开始，每执行完一个方法，就会将该方法的栈帧弹出JVM栈，知道方法全部执行完毕。



## 垃圾回收目标

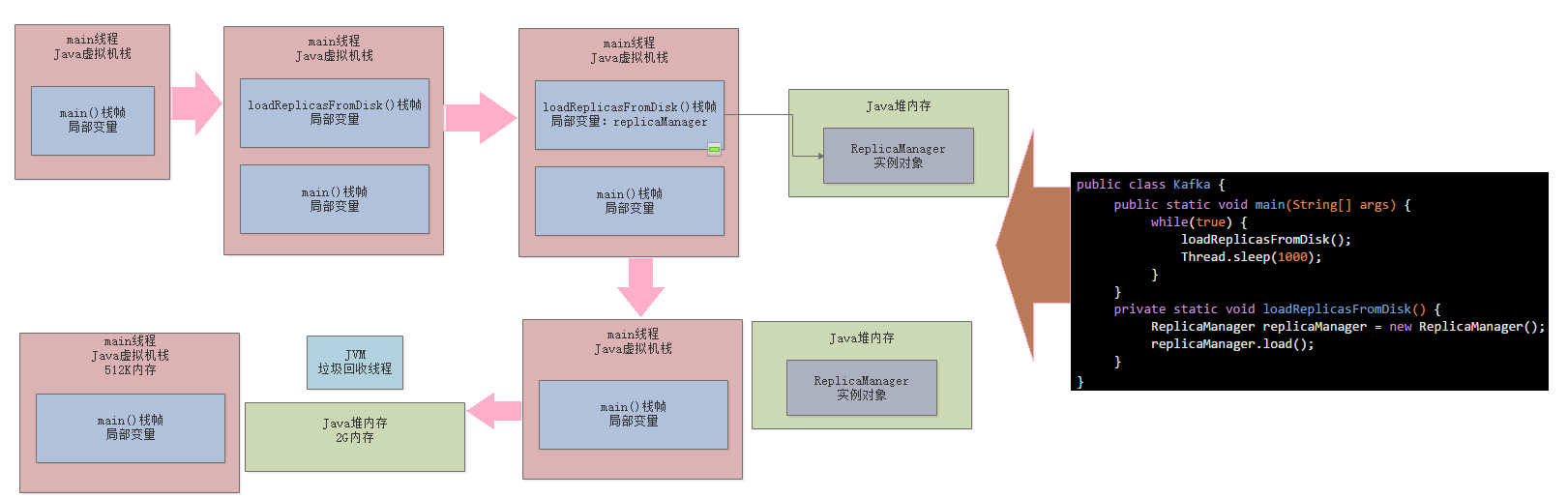
垃圾回收主要的JVM区域是堆内存，当然还有方法区，对方法区中的class对象的回收必须满足：该class对象的实例都被回收、该class的classLoader都被回收、该class对象没有引用链，此时才可以会回收该class对象，这是JVM规定的。

# 生产环境中堆栈内存及永久代如何设置

## JVM分代模型：年轻代、老年代、永久代

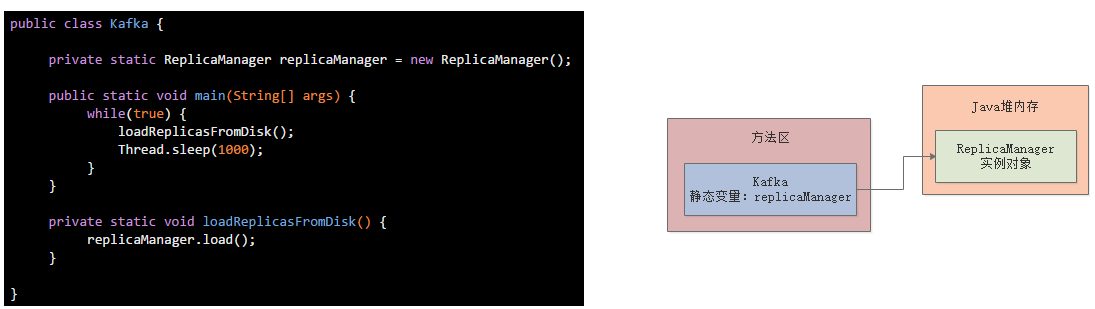
### 大部分对象存活周期极短

局部变量往往随着方法的结束、栈帧的消失而消失，但是局部变量指向堆中的对象却没有随着栈帧的消失而消失，而是被随之而来的垃圾回收线程给回收了，这类被局部变量引用的堆对象，生命周期很短。

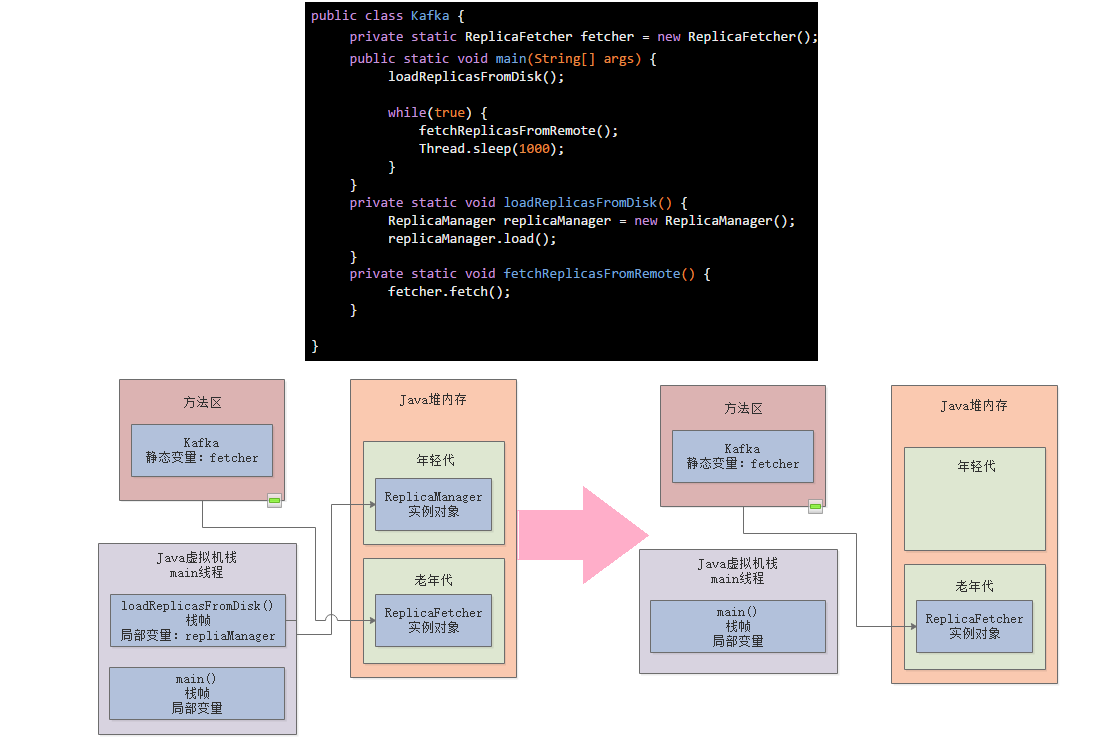


### 少数对象时长期存活的

但是，如果稍加改变代码，将局部变量变成类成员变量，它的生命周期就和类的生命周期一样长了，并且replicaManager又是一个强引用，就只会随着JVM进程的结束而结束，能长期存活。另外GC一般只会回收引用中的软引用、弱引用和虚引用。



### JVM分代模型：年轻代和老年代



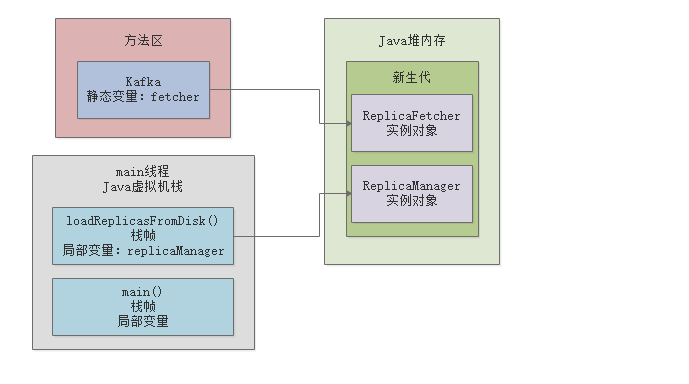
栈帧中的变量，指向的堆内存的对象，往往生命周期比较短，往往只能待在新生代，而类成员变量因为生命周期和类一样长，所以生命周期会比较长，在新生代待了一段时间时候会流转到老年代。

## 大面：对象在JVM如何分配和流转的

对象优先分配在新生代、新生代对象满了会触发Minor GC、如果有新生代的对象躲过了15次GC，那么它将被流转到老年代、老年代的对象过多时也会被回收。

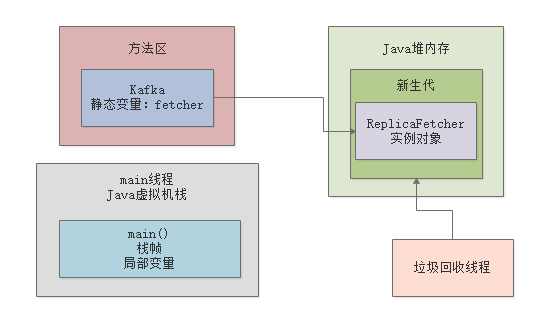
### 大部分对象都会先在新生代中分配内存

虽然说以上两种生命周期的对象最终都会被分配到堆内存中的不同区域，但是在最开始的时候，两者都在新生代中分配内存，而具体在什么时候新生代中的对象流转到老年代时接下来需要探讨的。



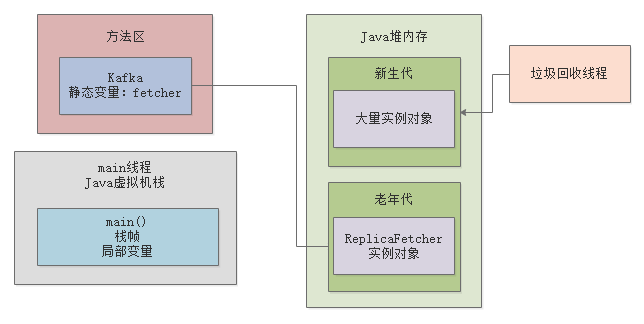
### 什么情况下触发新生代垃圾回收

当新生代中的内存已经占满了，此时会触发一次新生代的垃圾回收，称为Minor GC，也称为Young GC。



### 长期存活的对象可以躲过几次垃圾回收

通过上文可知，刚开始创建出的对象并不分新生代老年代，全部都放到新生代中，新生代中内存不够了，才会触发新生代的Minor GC，而在新生代中的被类变量引用的对象是不会被回收的，每当一个对象躲过一次回收，年龄就加1，默认躲过15次，表示15岁了，此时就会将该对象，从新生代移动到老年代中。



### 老年代会垃圾回收吗

和新生代一样，当老年代中的内存满了也会回收，第三章详解。

### 项目中的对象内存分布

利用Spring的项目中，需要依赖注入的，如service，一般都是老年代，在方法中创建的对象并赋给局部变量的，一般都为新生代；扩展一下，需要依赖注入的，被注解注入的都为老年代分配内存。

Scope为singleton的为老年代，而为prototype的为新生代。

## 线上系统部署时如何JVM内存大小

### 跟JVM相关的几个核心参数



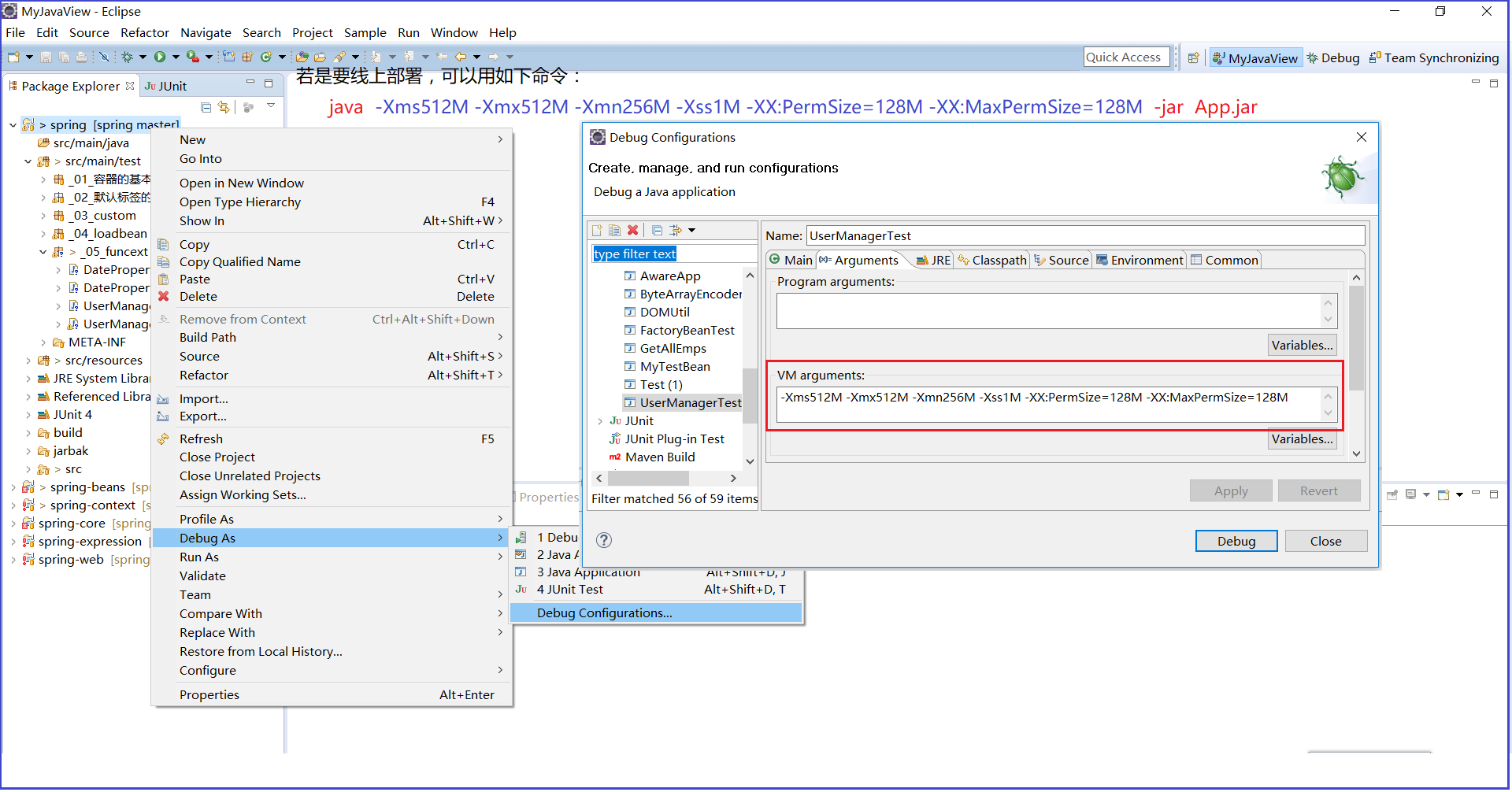
**-Xms和-Xmx**分别为堆内存大小已经允许堆内存扩张到的最大内存大小，一般这两个参数会设置成一样大，总之它们限定了JVM对内存的大小。

**-Xmn**为堆内存中新生代的大小，堆内存中扣除了新生代大小就剩老年代了。

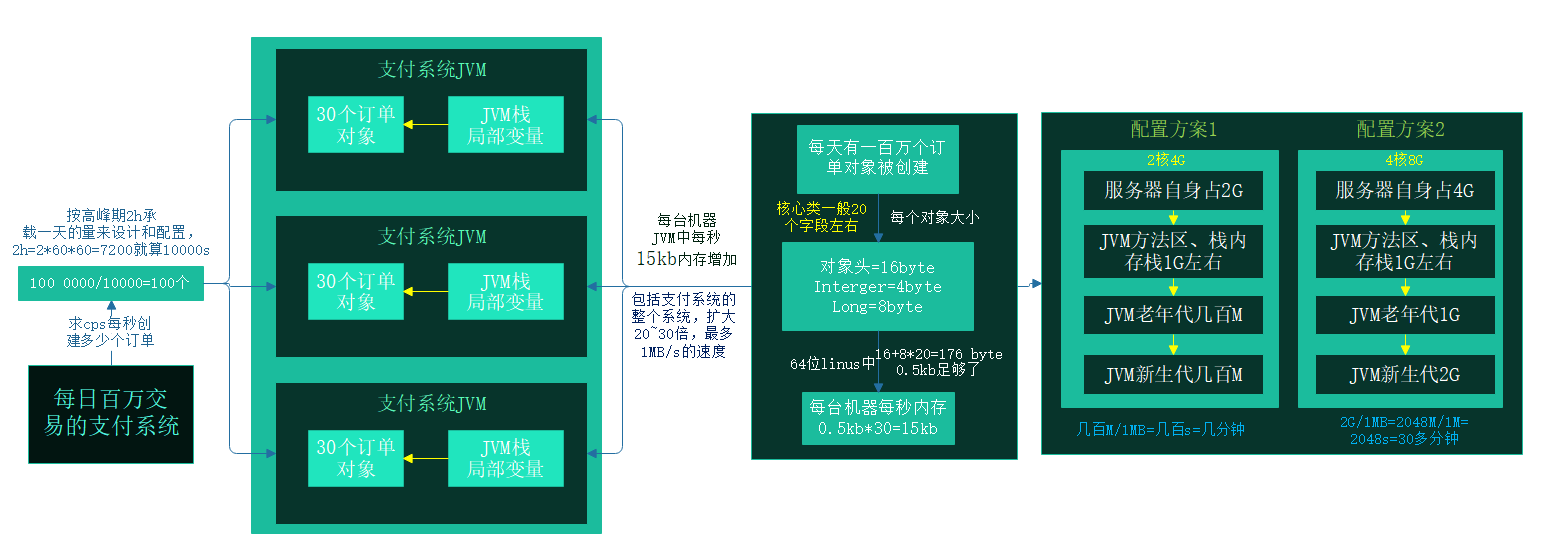
**-XX:PermSize和-XX:MaxPermSize**分别表示永久代的内存大小以及永久代允许扩张的最大内存，一般这两个参数也会设置成一样的大小。在JDK1.8时，不光方法区的名称从方法区变成了Metaspace元数据空间，这两个参数也相应的变成了**-XX:Metaspace和-XX:MaxMetaspace**。

-Xss表示每个线程的栈空间大小。

### 如何在系统启动时设置JVM参数



## 每日百万交易的系统，如何设置JVM堆内存大小



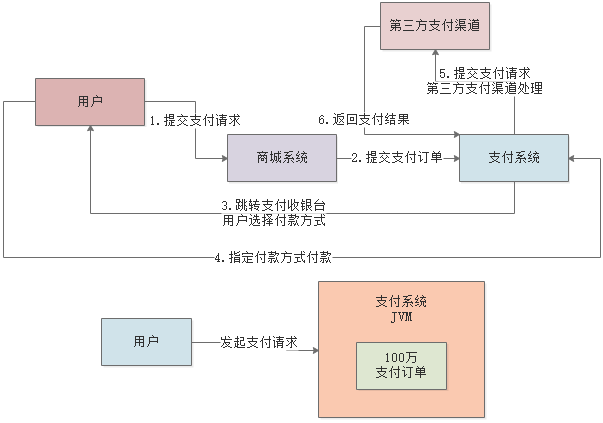
核心问题就是，该系统核心压力点在哪？每秒处理多少订单？每个订单耗时多久？每个订单占多少内存？系统每秒承受的压力？

### 支付系统核心业务流程

支付系统业务场景非常复杂，就最核心的支付流程如下图，每当用户发起一个支付请求时，都会发送到商城中，商城此时会将支付请求转交给支付系统，此时支付系统内会生成一个支付订单对象，此时会流转回用户方选择支付方式，当支付系统接收到用户选择的支付方式后转交给第三方支付渠道支付，并将支付结果返回给支付系统。

其中压力最大的就是，一天之内要处理百万级别的订单量，就JVM角度说就是每天会有百万个对象在堆内存中创建和销毁。

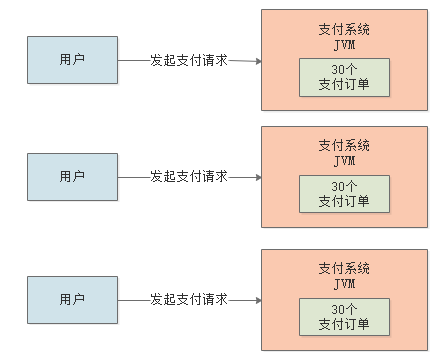
那么问题来了，需要几台机器、每台机器内存多大、每台机器JVM内存多大才能保证这样打的内存开销。

****

### 支付系统每秒需要处理多少笔支付订单

因为本次讨论的是百万级支付系统，而它业务量的最高峰一般都会在中午或者晚上就几个小时的时间内，假设就两个小时左右，将一天的全部订单量全都假设发生在这两个小时内，来评估系统、机器分配和JVM调参的依据。

两个小时2\*60\*60=7200s就算1000s计算，一百万订单每秒就是一百个订单，这一百个订单假设此时放在三台机器上，大概每台机器每秒支撑30个订单的处理，如下图所示：



### 每个支付订单耗时多久

每台机器，大概是一秒钟接收到30个订单的请求，然后在JVM的新生代中创建出来，一秒钟后处理完毕，就会被GC了。

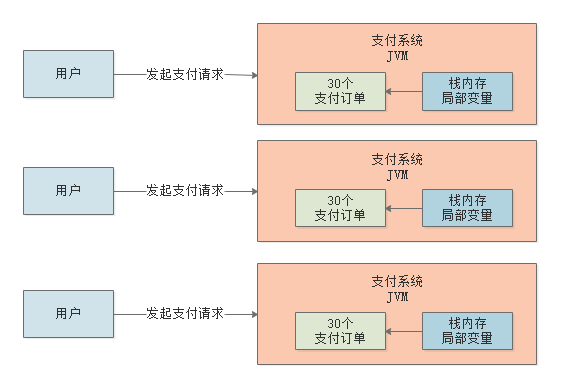
### 每个支付订单大概需要多大的内存空间

一般一个对象占内存大小分两部分，对象本身的一些信息，如64位linus上对象头占16字节，对象实力变量Integer占4字节、Long占8字节，再根据其他类型计算就差不多了，一般支付系统核心类按20个实例变量计算，一般一个对象几百字节，姑且大一点500字节内存空间，0.5kb也不算大。

### 每秒发起的支付请求对内存的占用

按如上文分析，如果按每台机器30个订单对象的分析，那也只不过是30\*0.5=15kb的内存占用，其实是很小的。随着系统的运行，新生代中的订单对象越来越多，知道某一刻达到了内存的阈值，启动Minor GC，将对象回收，依次往复。

因为此时分析的是最核心的支付流程，在订单创建的过程当中肯定包含其他数十种对象的创建，不妨将数量级扩大10~20倍，一秒的内存占用量也就几百kb~1MB之间。



### 支付系统JVM堆内存最终如何去设置

一般的服务器有两种常见规格，2核4G和4核8G的。假设此时用2核4G的，服务器本身差不多要占2G空间，剩下的2G需要分配给方法区、栈内存和堆内存，此时堆内存差不多能分到1G内存，但是堆内存中的老年代至少也需要几百M空间，所以剩下给新生代的空间也就剩下几百M了，根据完整的支付系统估算，一秒需要占用1M新生代内存空间，所以此时差不多几百秒几分钟就会GC一次，而频繁的GC对系统的稳定性总是不好的。

假如此时采用4核8G的内存，能分给JVM的内存至少有4G，这4G内存分给方法区和栈内存1G就差不多了，剩下给堆内存的空间至少有3G了，3G的堆内存分给新生代2G，触发一次Minor GC差不多需要半小时到一个小时之间，这样不至于太频繁的GC操作。如果此时系统的压力更大了，可以水平扩容，多加几台机器，这样每台机器上的压力就更小了。

### 查看linus配置的一些信息

//查看linus的cpu个数

**more /proc/cpuinfo |grep "physical id"|uniq|wc –l**

//查看linus的物理cpu个数，和cpu个数一样

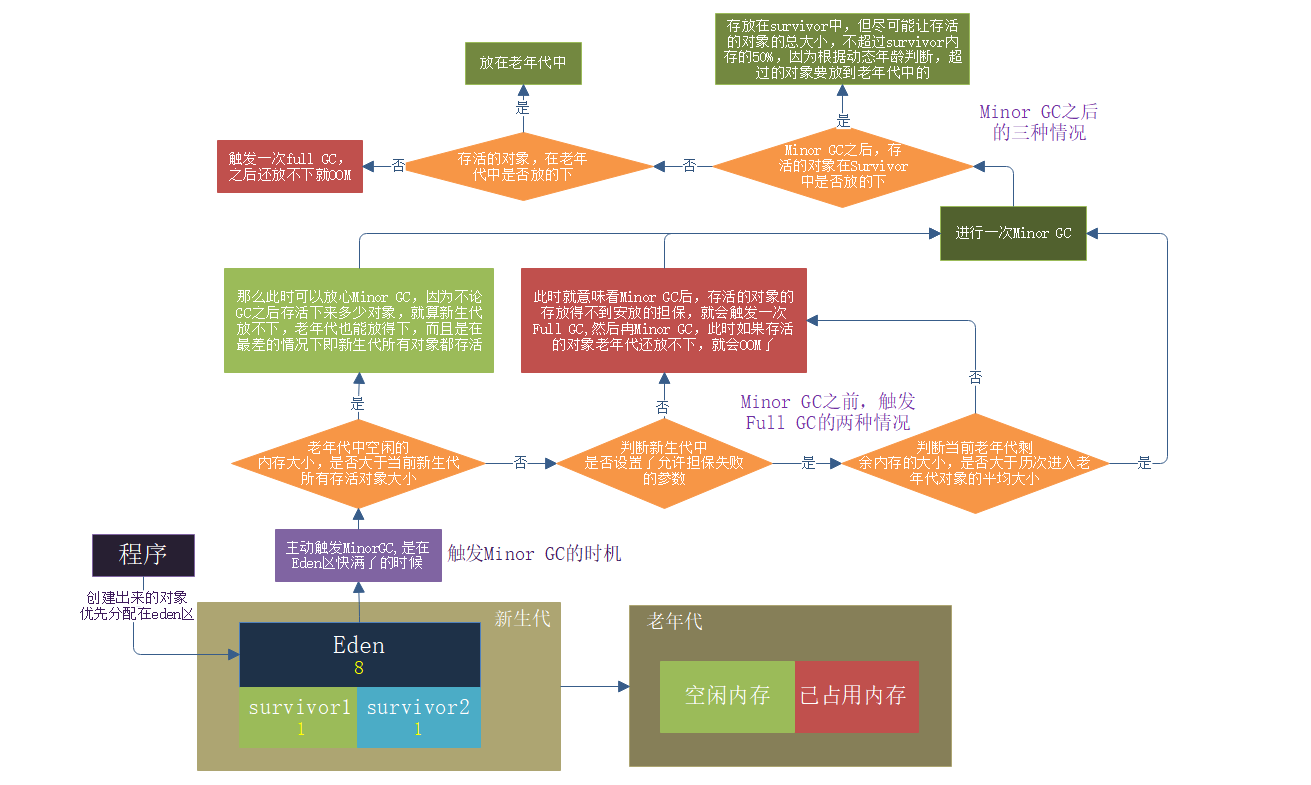
**cat /proc/cpuinfo |grep "physical id"|sort |uniq|wc –l**

//linus中查看内存

**free**



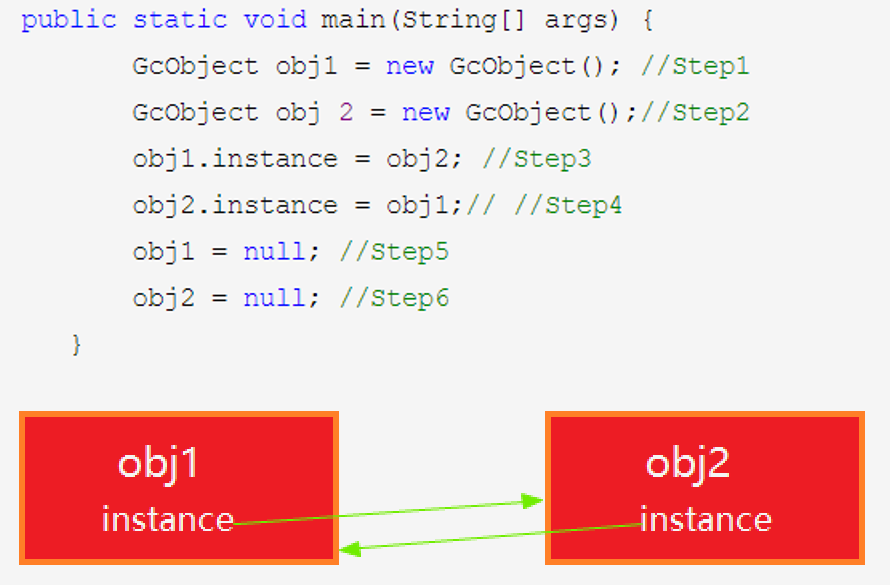
# JVM中各种垃圾回收算法



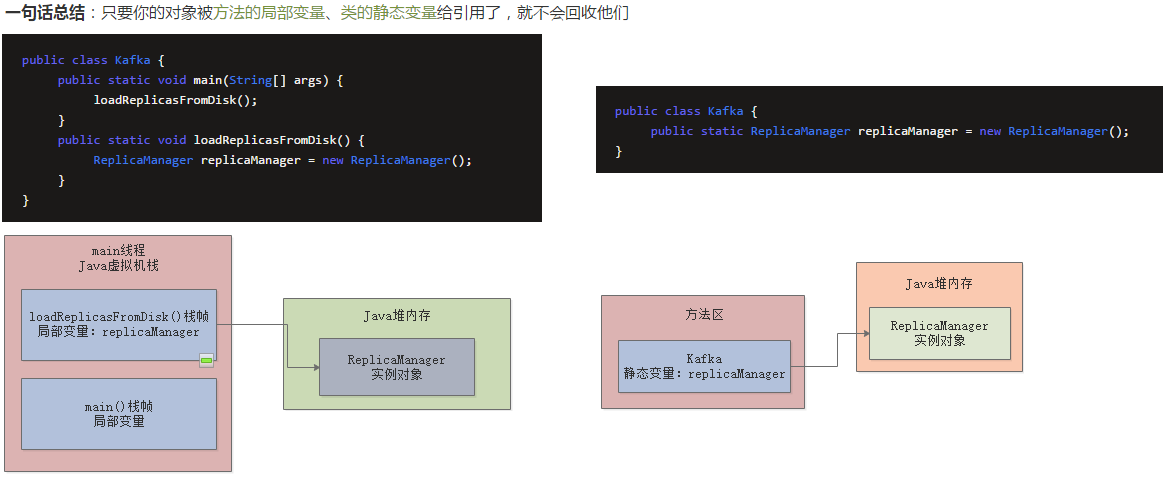
## 大面：什么情况下JVM的一个对象会被回收

### 引用计数算法-循环引用的对象

首先，两个局部变量obj1和obj2分别指向堆内存的起始地址，就是它们刚new出来的；然后它们互相指向对方，即将它们指向的对象中的成员变量分别都互相指向对方内存地址了，成员变量代替了obj1和obj2的指向作用，建立引用，这就意味着堆内存中的两个被obj1和obj2指向的对象的引用计数器都为1了，此时又把obj1和obj2都指向null，此时堆内存中的两个创建的对象就没人引用它们了，此时发生GC时一检查，又发现引用计数器不为0，这永远就回收不了了。



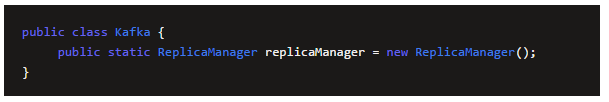
### GC Roots保持引用链的对象不会被回收



### 强、软、弱引用

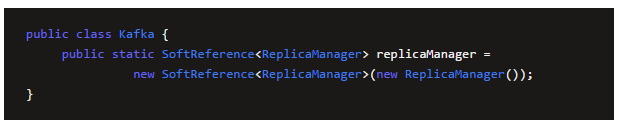
（1）强引用

强引用就是一个变量应用一个对象；该被应用的堆中的对象，宁可报OutOfMemoryError错误，也都不会被回收，



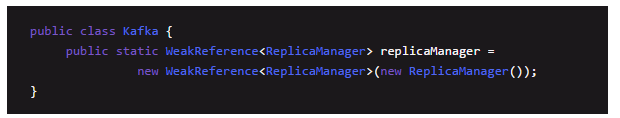
（2）软引用

软引用，在java中就是被SoftReference包裹起来的对象；只有内存不够了，才会被回收。



（3）弱引用

弱引用的的表达形式和软应用类似，就是通过WeakReference包裹起来的对象；弱引用的效果几乎和没引用差不多，所以只要垃圾回收，它就会被回收。



## 标记-清除和复制算法

### 算法的思路

标记-清除算法适用于老年代，主要过程就是将一些对象经过两次标记，即可达性算法分析和finalize方法中标记，如果这两次的标记都未能拯救到改对象，就可以回收了，在内存中标记好这些可以回收的对象，然后清除就是。

复制算法，是适用于新生代的，它会将新生代的内存分为等额的两份，正常工作时只会使用其中一块内存空间，当需要Minor GC时，将其中一块内存中的还存活的对象复制到另外一块空着的内存中，然后一次性清空之前的那块内存。

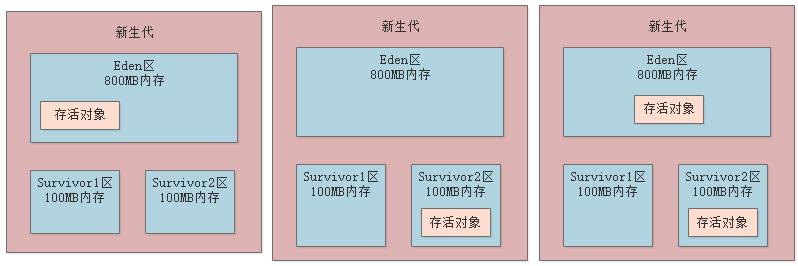
### 算法的优劣

标记-清除算法最大的缺点就是浪费内存，因为它是适用于老年代的，标记清除之后会导致老年代的空间存在大量的内存碎片，此时如果有一个大对象直接进入老年代，本来这些内存碎片的总和是可以存放该对象的，但是这些内存却是碎片的形式存在，因此会提前触发Full GC，而Full GC的速度至少要比Minor GC至少慢十倍以上，且浪费内存。

而复制算法，相比于复制-清除算法减少了内存碎片的问题，但是由于每次只使用一半内存，也就意味着每次有一半的内存空间被浪费了，即内存比例划分不合理，有待改善。

### 复制算法的改进

复制算法常常用在新生代中，而新生代中的对象一般存活时间都很短，每经历一次Minor GC都会导致绝大多数的对象被回收了；根据这个特点可以将新生代的内存比例进行调整，将新生中的内存比例从等额的两份调整为 Eden:Survivor:Survivor=8:1:1的三部分，每次只使用Eden和其中一个Survivor区，当它们都占满了时触发Minor GC，将Eden和其中一个Survivor区的存活对象都迁移到另外一个Survivor区中,此时另外一个Survivor开始被闲置直到下一次Minor GC和另外一个Survivor交换。



### 复制算法为什么不用9:1比例

复制算法改进的目的，就是为了能更高效的使用内存空间，如果使用9:1，如果9部分的空间满了，存活对象都移到1那里，此时9的空间却被闲置了，太浪费内存了。

8:1:1的目的在于，每次Minor GC之后都保证8那部分的空间一直不会被闲置，只是每一次的Minor GC，两个Survivor空间随闲置，每次都发生了变化。

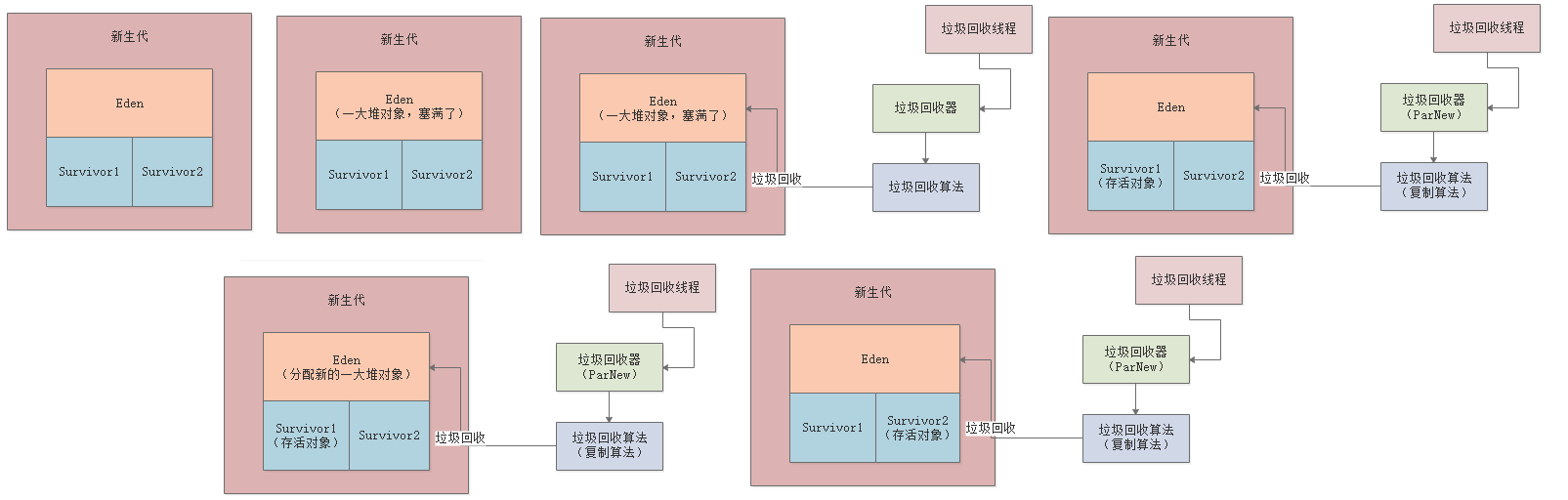
## 内存分配与回收策略

### Eden区优先分配策略

对象创建时优先分配在新生代中的Eden区，此时可能会有一个疑问，既然对象优先分配在Eden区，那和Eden区一起使用的那块Survivor区中的内存岂不是没机会放对象了，其实需要从大的角度来分析。

每一次Minor GC开始，会将Eden区和其中一块Survivor区中存活的对象移到空闲的那块Survivor中，而此时空闲的这块Survivor内存才会和Eden再次组合使用，只有第一次发生Minor GC之前，两个Survivor区都为空，其他情况下都是只有一个survivor为空。

Survivor中如果有对象，那一定是在Minor GC时，从其他两个区域被迁移过来的，而不是对象自动被分配过来的，因为对象优先在Eden区分配内存，Eden区内存不够也不会放在Survivor中，而是发起Minor GC，如下图的垃圾回收线程，会利用ParNew垃圾收集器，采用复制算法进行垃圾清除，但是此时也是会Stop The World的。



### 升到老年代规则-超过一定年龄

如果一个对象它在最开始的时候是在新生代中的，当时它经过了15(默认)次的Minor GC都还没有被清理掉，那么它就会被升到老年代中；一般强引用或者是到GC Roots有引用链的对象都不会被轻易清除掉，它们可能会到达一定年龄的时候被升到老年代中。

### 升到老年代规则-动态年龄判断

动态年龄的判断，主要作用于Survivor区域，在该区域中，如果有一批年龄相同的对象，它们的总内存大于该Survivor总内存的一半，那么此时如果有对象过来了，只要过来的对象年龄大于或等于这批相同年龄对象的年龄，那么这些新来的对象直接到老年代区。

实际上的情况是，如果当前survivor中某个年龄或者比该年龄小的对象占内存总和大于50%，此时如果进来某个对象的年龄大于该年龄，就会直接进入老年代中。

其实目地和到一定年龄就升到老年代类似，新生代中只会保留年龄相对小的对象，年龄大的对象尽早让它们到老年代中，免得占内存。

### 升到老年代规则-大对象分配

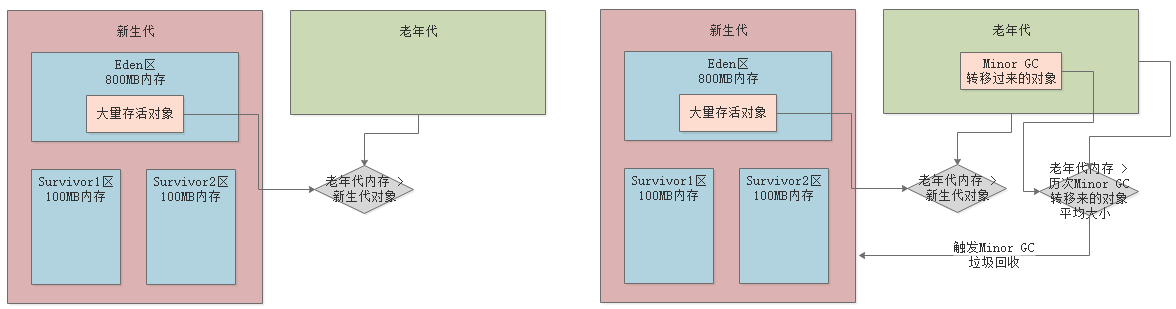
有一个参数，如果当前需要分配到新生代中的对象的大小超过了这个对象，那直接分配到老年代中，因为如果这个对象在Survivor中来回复制是非常耗性能的。

### 分配担保机制

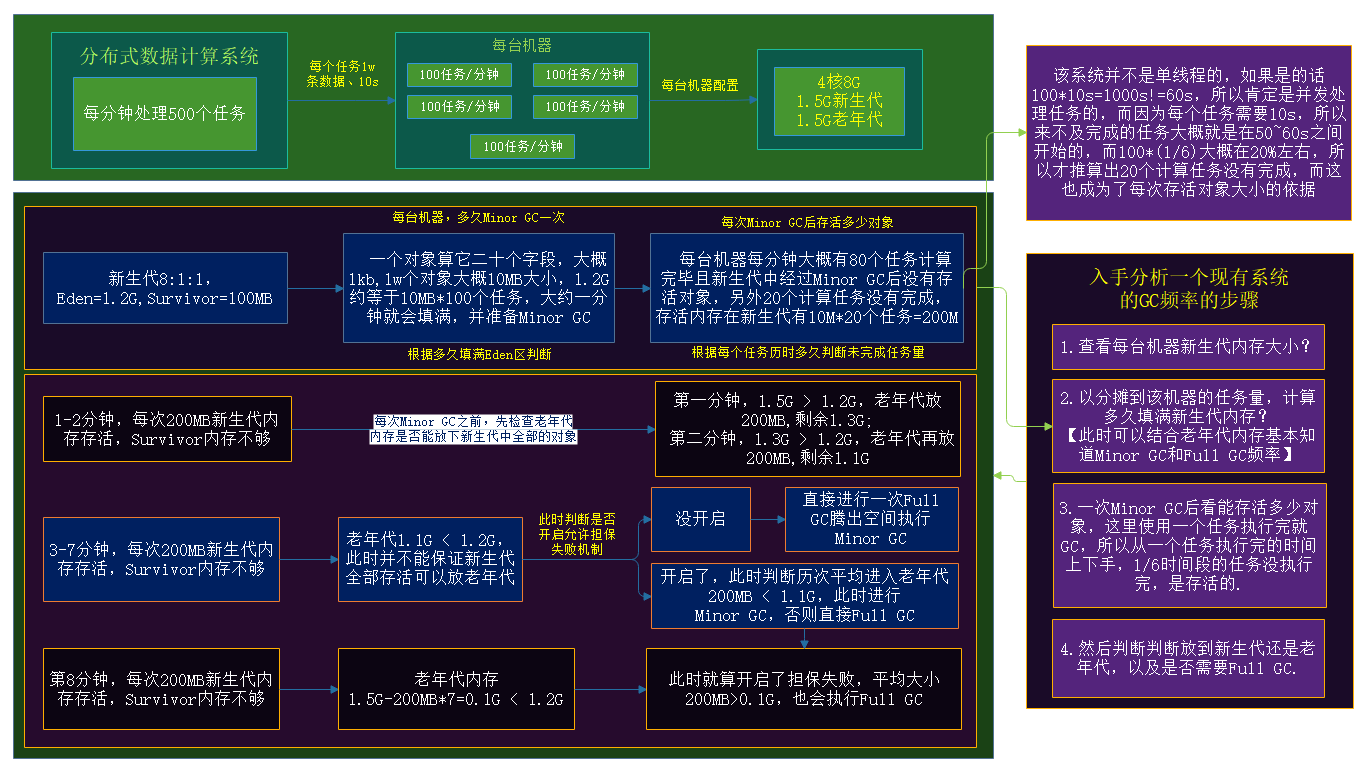
该过程涉及到类似生活中的贷款担保的思想，在每一次Minor GC之前，都会去判断老年代中的剩余内存空间大小是否大于新生代中所有存活对象（Eden和其中一块Survivor）的内存大小。

如果大于就可以放心进行Minor GC了，因为此时最坏的情况---一次Minor GC之后新生代中所有的对象都存活也被考虑到，就算此时Survivor放不下也可以升到到老年代中；如果小于，此时会再去判断是否设置了分配担保失败的参数-XX:-HandlePromotionFailure。

如果设置了表示是允许担保失败的，此时可以赌一下，以历次进入老年代中每个对象的平均大小和老年代中当前剩余空间比较，如果大于剩余空间，没的选择只能进行一次Full GC，然后再去Minor GC；如果小于，那就冒险去Minor GC一下，此时如果不幸冒险失败进入老年代中的对象大于剩余空间，那就再Full GC呗；当然大前提下，不允许担保失败即没配置参数-XX:-HandlePromotionFailure，那肯定不能冒险，直接先Full GC，再Minor GC。



### 案例-分布式数据计算系统



-------------------------------------------------------------------------------

这里最大的缺点就是每次Minor GC后存活的对象基本上都不能放到Survivor中，导致没分钟都要Minor GC一次，且几乎每次Minor GC后存活的对象都放到了老年代中，造成了七、八分钟一次Full GC，而这么频繁的Full GC和Minor GC对系统的性能影响非常大，尤其减少Full GC的频率是优化的目标。

此时可以适当增加新生代的内存比例，从新生代1.5和老年代1.5G改成新生代2G老年代1G，此时Survivor为200M，刚好可以存放的下一次Minor GC的对象，这样从Full GC七八分钟一次的频率改为了几个小时一次。

除了适当增强新生代内存的比例，还可以修改Eden：Survivor的比例，让Survivor的内存占比大一些，也还是避免过早的让对象进入到老年代，带来不必要的频繁Full GC。

-------------------------------------------------------------------------------

总结下优化的最终目的就是减少Minor GC和Full GC的频率，而增加这些频率的原因还是，Eden区内存小，导致Minor GC频繁，Survivor内存小导致Minor GC后的存活对象会晋升到老年代而导致FullGC，由于Full GC影响的结果更大，以增大Survivor内存为直接目标。

（1）内存有限情况

第一，可以将Survivor在新生代的占比增大些，虽然这样会让Minor GC频率增大，但还是有效减少Full GC的。

第二，可以将新生代的占比在整个堆内存中增加占比，这样Full GC也有可能频率下不来，随意不推荐。

（2）内存充足情况

将整个堆内存空间增大。

-------------------------------------------------------------------------------

### 内存分配经典问题

（1）什么时候会尝试触发Minor GC

Eden区的内存快满了的时候。

（2）触发Minor GC之前会如何检查老年代的大小、涉及那几个步骤和条件

触发Minor GC之前，首先会判断当前老年代剩余内存是否大于新生代中所有对象的大小，如果大于，表示可以放心Minor GC，反正最差情况下所有新生代对象存活、老年代都放得下，不会OOM。

如果小于，此时会检查是否设置了担保失败的配置，如果设置了，此时会判断历次进入老年代的平均对象大小是否大于老年代剩余内存大小，如果大于则发生Full GC，Full GC后老年代还放不下，没得救了、只好OOM了；如果小于可以冒险Minor GC。

（3）什么时候在Minor GC之前就会提前触发一次Full GC

老年代的剩余内存小于新生代中所有对象的大小，在该前提下，有两种情况会触发，一种是分配担保失败机制没有开启；一种是分配担保失败机制开启了，但是历次进入老年代的平均对象大小大于老年代剩余内存大小。

（4）Full GC的算法

老年代的标记-整理算法，CMS标记清除比较特殊。

（5）Minor GC过后对应哪几种情况

存活的对象小于Survivor区，放在Survivor区中；

存活的对象大于Survivor区，小于老年代剩余内存大小，放到老年代中；

存活的对象既大于Survivor区，也大于老年代剩余内存大小，此时Full GC，之后老年代还放不下，只好OOM了。

（6）哪些情况下Minor GC后的对象会进入老年代

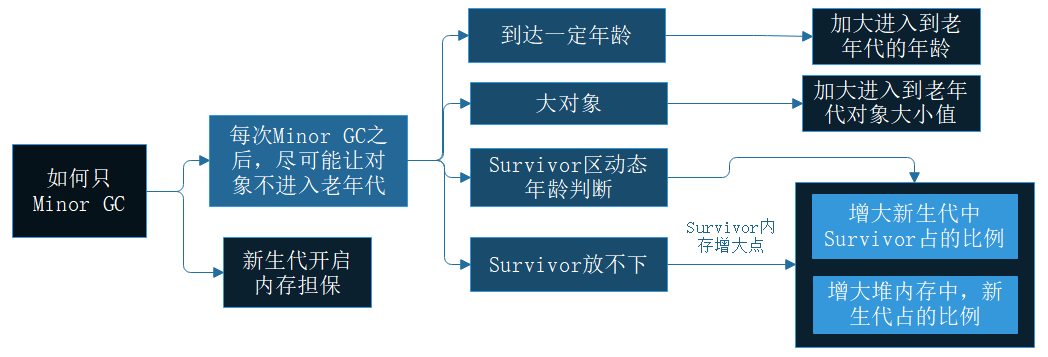
首先年龄超过15岁(默认)，晋升到老年代中;

大对象，内存大小大于某个参数值时，直接进入老年代；

Survivor区中，年龄相同的对象大小超过该区一半，再进来的对象只要大于这年龄，直接到老年代。

Survivor区放不下的对象；

（7）parnew+cms的gc，如何保证只ygc，jvm参数如何配置



如何保证只ygc而不经常的fgc，如案例所示，导致频繁的fgc最关键的原因就是survivor区域无法存放每一次ygc之后存活的对象，只要将survivor内存比例增加就可以保证，每次ygc之后存活的对象都能放的下，同时考虑到survivor动态年龄判断，还需要保证survivor中存活的对象不能超过50%的总内存，才能最大程度保证新一轮ygc之后，不会有对象到老年代。

所以优化的手段以增加survivor的内存为目标，可以增加新生代占堆内存的比例、增加survivor区域在新生代中的比例。

## 大面：GC在什么时候、对什么东西、做什么处理

### 什么时候

（1）代码层面上

调用System.gc()时;

（2）内存层面上

当堆内存中，新生代的Eden内存满了的时候，此时会发生一次Minor GC，将这两个区域中的还存活的对象放到另外一个Survivor中。

如果此时幸存的对象的大小大于另外一个Survivor的内存，此时会使用分配担保机制，将幸存对象放到老年代中，此时如果老年代的内存空间也不够或者当前幸存的对象的大小大于老年代中设置的某个临界值时，就会触发Full GC。

（3）调优建议

可以通过控制Eden和Survivor的比例，或者通过参数控制新生代中的对象升到老年代的年龄实现调优。

### 对什么东西

（1）和GC Roots对象没有任何引用链的对象；

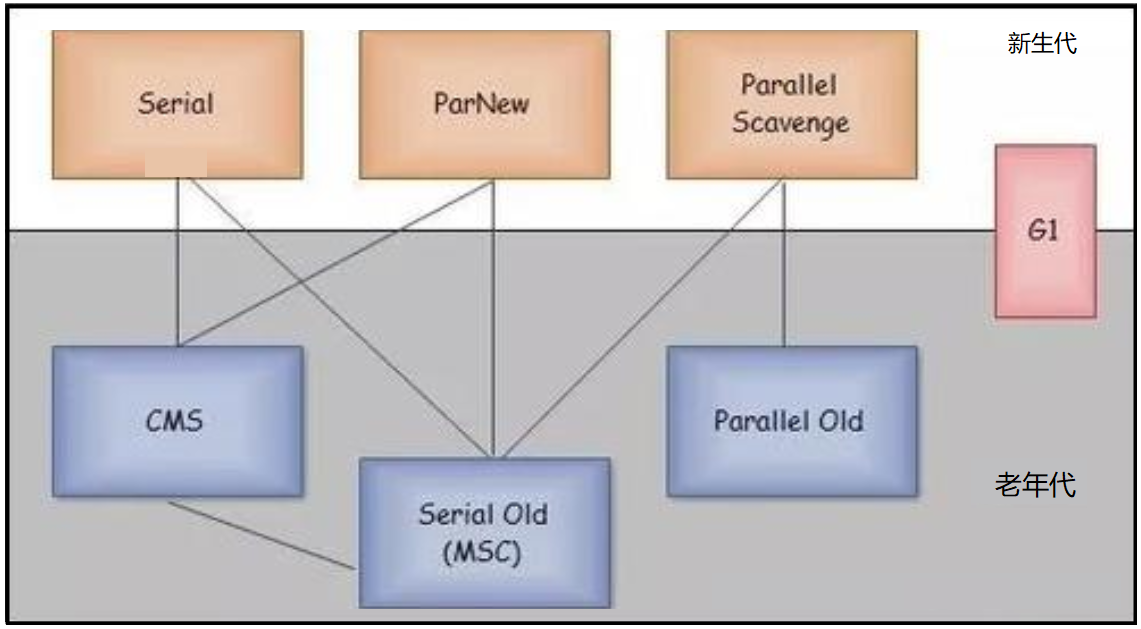
（2）在第二次标记的时候，没有成功在finalize方法中，和GC Roots关联上。

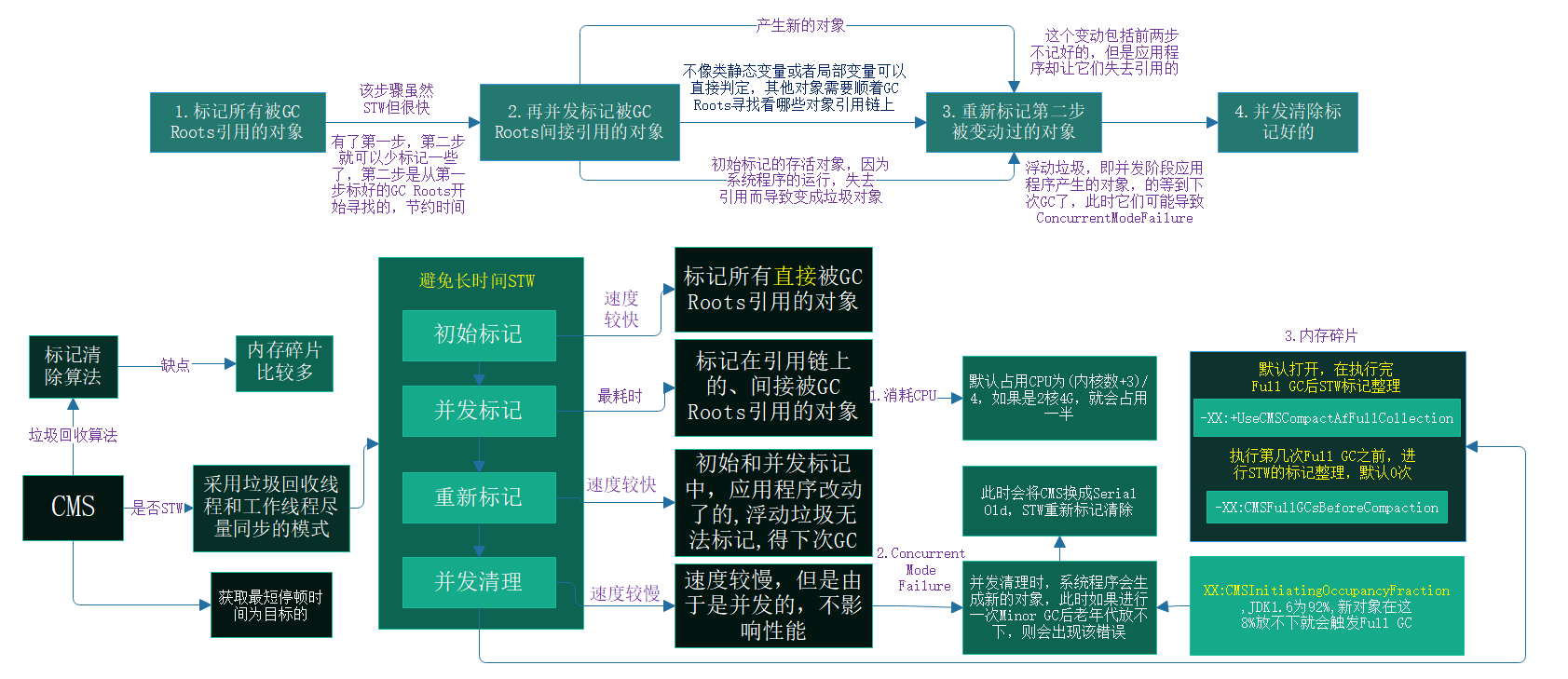
### 做什么处理

处理要分新生代和老年代分别处理，比如新生代常见用复制算法，因为该区域的对象都是朝生夕灭的，所以使用复制算法比较好；老年代中的对象，生命周期会比较长更适合使用标记清除和标记整理算法，而标记清除算法缺点就是留下很多内存碎片，如果此时老年代的内存不够并且此时又有一个比较大的对象进入到了老年代，此时找不到一块连续的内存存放，此时就会提早触发一次Full GC。

串行、并行（整理/不整理碎片）、CMS等搜集器可作用的年代、特点、优劣势，并且能说明控制/调整收集器选择的方式。

# JVM垃圾回收器原理、参数设置、性能调优







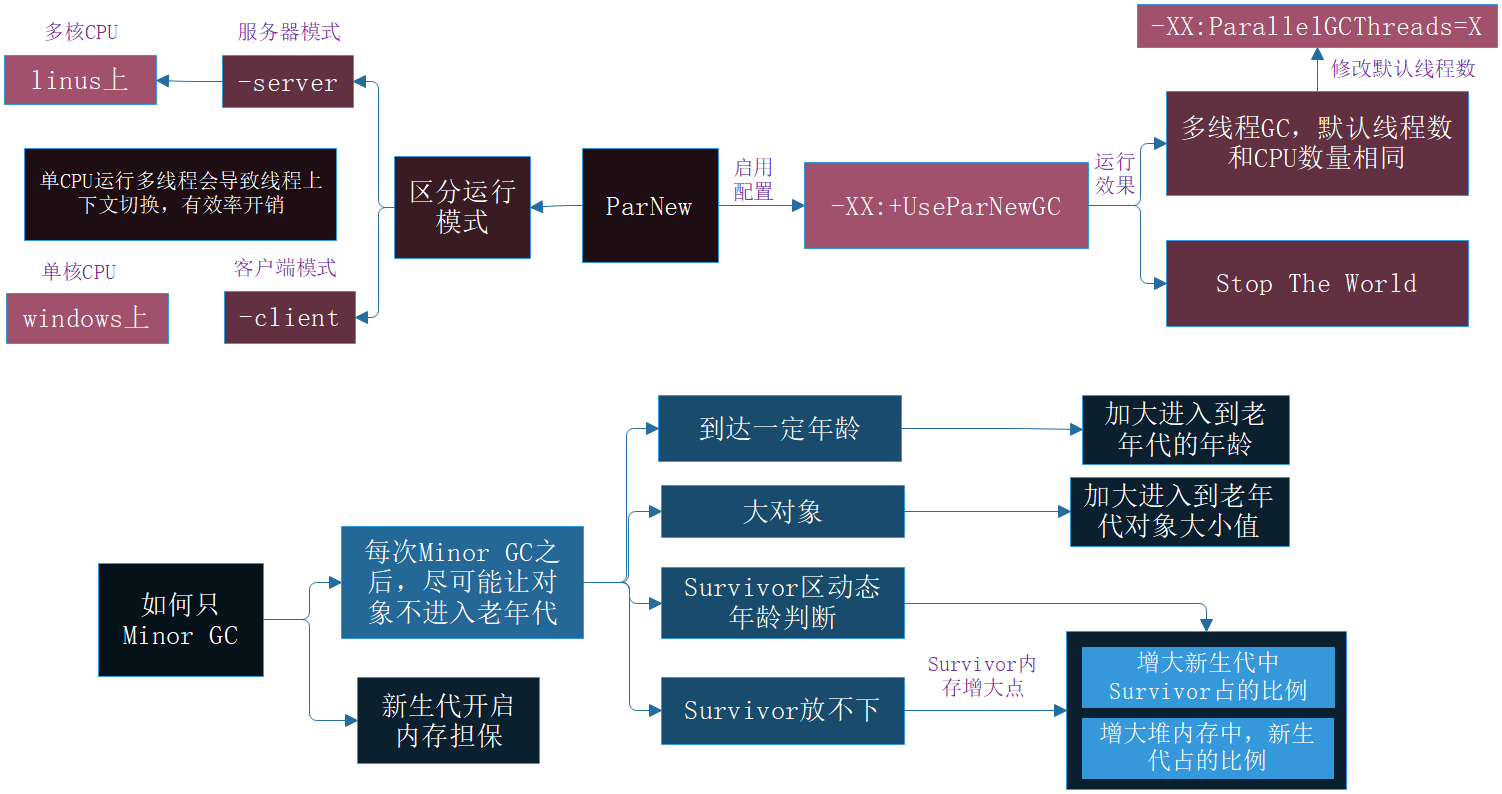
## Stop The World

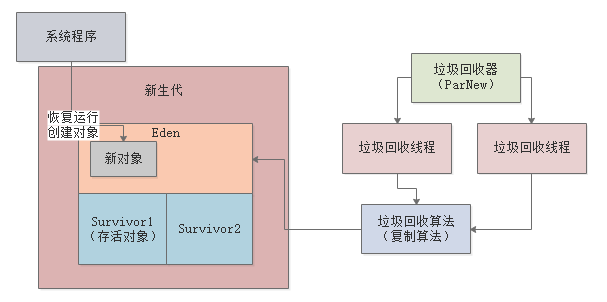
引出Stop The World之前，有一个问题就是是否允许在垃圾收集的时候再在Eden区创建对象。

此时假设新生代也是采用ParNew收集器的复制算法进行垃圾回收，并且此时已经将需要转移到Survivor的存活对象、已经在Eden区和另外一块Survivor中标记好了，正准备对Eden区的其他没标记的全部清除掉，此时如果在我们垃圾收集的时候，允许在Eden中创建对象，就有可能会把这刚刚创建好的、存活的对象都给清理掉了，因为这些对象哪些存活还没来得及标记；此时最好的思路当然是暂停其他用户线程工作，防止它们再向Eden区创建对象，以防 “伤及无辜”，这就是Stop The World。

但是Stop The World导致的后果是非常严重的，你一旦暂停了所有用户线程，用户就需要等待响应，用户体验很差，而导致Stop The World就是各种垃圾回收操作，如ParNew发生的Minor GC是罪魁祸首，之前同样也得知Full GC也是很慢很影响用户体验的，所以这里可以得出，不管是Full GC还是Minor GC，保证它们不那么频繁的发生才是JVM调优的重中之重。

## ParNew垃圾收回器的认识

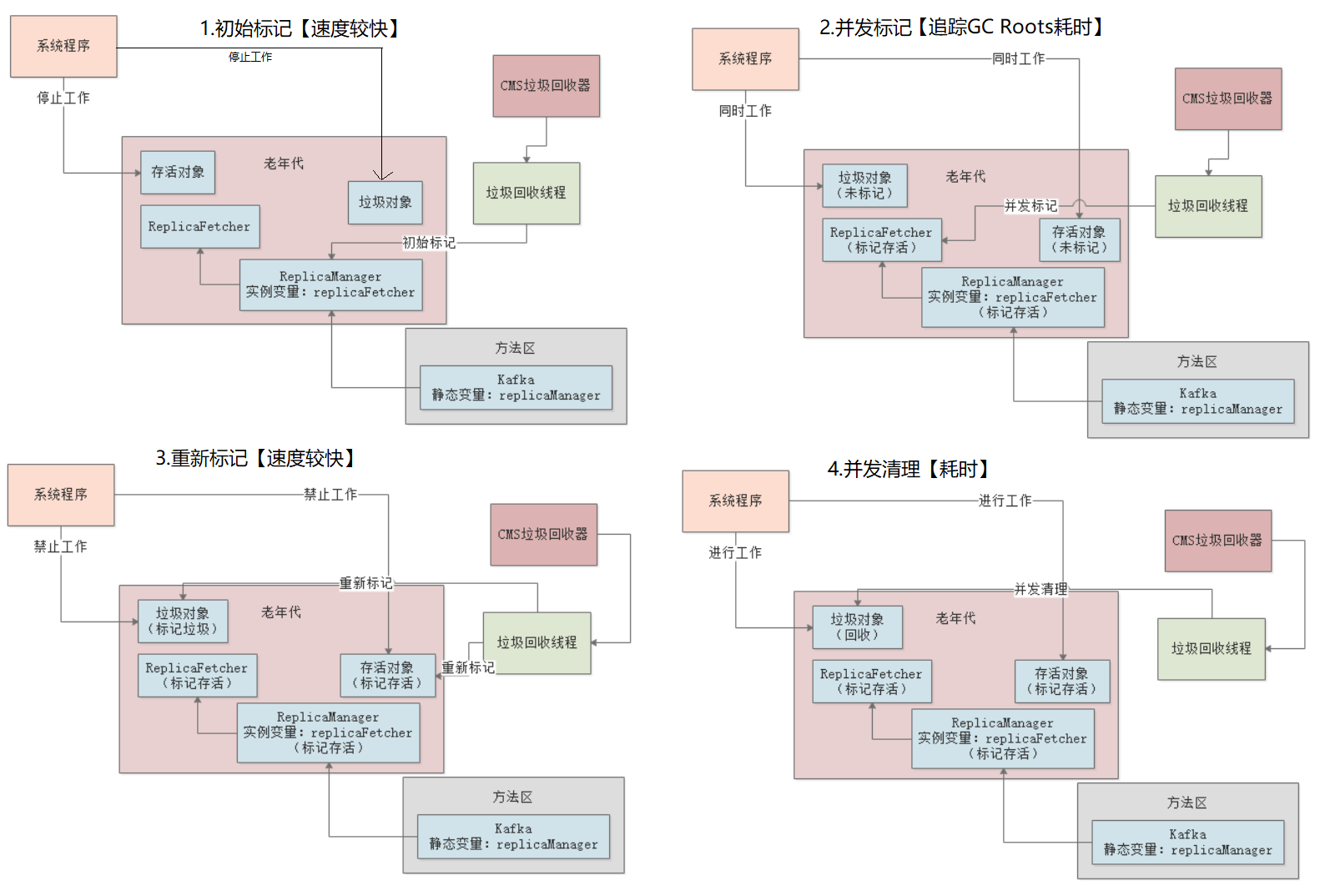




和Serial最大的不同就是ParNew是多线程的，其他特性和Serial是相同的，包括垃圾回收器工作时Stop The World、都可以和CMS与Serial Old搭配使用。

其中ParNew默认开启的线程数和CPU数量一致，当然可以通过参数-XX:ParallelGCThreads自定义开启的线程数；启用ParNew垃圾回收器，可以在JVM启动时，添加参数-XX:+UseParNewGC。

## CMS垃圾回收器的工作原理



在G1还没有商用时，ParNew+CMS是线上系统的标配，因为CMS是老年代的垃圾回收器，而老年代的对象特征是大量对象存活，少部分对象死亡，和新生代相反，所以采用标记清除或者标记整理的多，这里CMS使用的是标记清除算法。

Minor GC和Full GC都会导致Stop The World，很多算法的目标就是为了降低这种延迟，CMS在这方面做的挺好的，工作原理主要有四个步骤：

（1）初始标记-SWT

该过程主要是对老年代中、GC Roots直接引用的对象进行标记，但是因为只是简单标记被GC Roots简单引用的对象，所以速度比较快。

（2）并发标记

该过程需要这样顺着GC Roots引用链寻找，只要在引用链上的对象都标记，且老年代存活对象多，寻找起来才比较耗时；但是因为是并发标记，所以并不影响系统运行、不会造成系统卡顿。

（3）重新标记-SWT

该过程主要是对第二步操作中、系统程序运行时、对之前的标记变动过的对象进行标记，由于第二步操作并没有STW，而是和系统并发运行的，所以此时可能系统运行时新创建的对象有些没来得及标上、或者某些之前存活的对象失去引用变成垃圾对象了，对象是它们，算是为并发补坑吧；虽然当前步骤也是SWT的，但是执行速度和第一步的SWT一样，很快就能结束，所以对系统性能影响也不大。

（4）并发清除

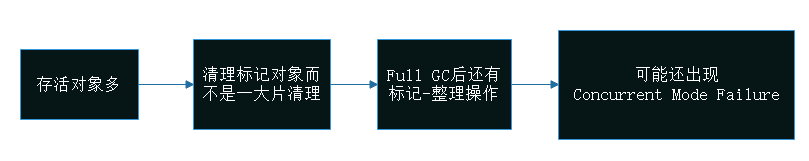
该过程主要是最前三个步骤的垃圾进行清除，虽然该过程和并发标记一样比较耗时，但也和并发标记一样都是和系统程序并发执行的，所以也不会造成系统性能的太大损耗。

注意，前三个步骤存活对象和垃圾对象都会标记，只是在并发清除时产生的垃圾对象没法处理，它们被称为浮动垃圾，得下次Full GC才能处理，CMS的妙就妙在将耗时的操作都并行了，只有后续的标记整理和发生CurrentModeFailure比较耗时。

## Full GC为什么比Minor GC慢十倍以上

为什么老年代中的Full GC会比新生代中的Minor GC慢很多倍，因为新生代中的存活对象很少，在根据GC Roots进行标记的时候很快就可以标记完了，就算后续沿着GC Roots的引用链去寻找间接被GC Roots引用的对象，速度也非常快了；而老年代中由于每次Full GC之后的存活对象非常的多，每次标记直接被GC Roots引用和追踪那些间接被GC Roots引用的对象，即标记大量对象慢、清除不是零零散散而不是一大片的内存慢、最后还要STW整理慢、倘若出现Concurrent Mode Failure换Serial Old重新SWT清除则更慢。

所以综上Full GC慢的原因和Minor GC相比有一下几点：



## CMS垃圾回收器的缺点和参数设置

上文说到CMS为了尽可能地减少垃圾回收器导致的STW带来的停顿时间，采用初始标记、并发标记、重新标记以及并发清除，初始标记和重新标记虽然需要STW，但是由于速度很快，带来的影响也不大，而并发标记和并发清除虽然比较耗时，但是由于是和系统程序并发处理的，所以也不会太大影响系统性能，但是CMS还是有很多比较明显的问题的，主要有下面三个：

（1）比较占用系统的CPU资源

CMS是一个多线程工作的垃圾回收器，默认给它分配的CPU数量为(系统CPU数量+3)/4，所以此时如果该CMS在一台2核4G的机器上，就要占用一半的CPU资源了，但是随着系统CPU数量的增加，这种影响会逐渐变小。

（2）浮动垃圾导致Concurrent Mode Failure

在并发清除的阶段，此时系统的程序也在运行，此时就会出现一边在清除垃圾对象，系统程序那一边又在不断地产生新的垃圾，这些对象中此时已经过了CMS的标记阶段了，称为“浮动垃圾”，此时一般都是大对象直接进入到老年代、成为浮动垃圾，发现此时老年代内存也不够了，此时就会发生Concurrent Mode Failure，此时就会将CMS垃圾回收器替换为 Serial Old，然后STW并重新进行标记和清除。

这里有个参数-XX:CMSInitiatingOccupancyFraction，JDK1.6时CMS默认为92%，即当老年代的内存大于该92%时，就会发生Concurrent Mode Failure错误。

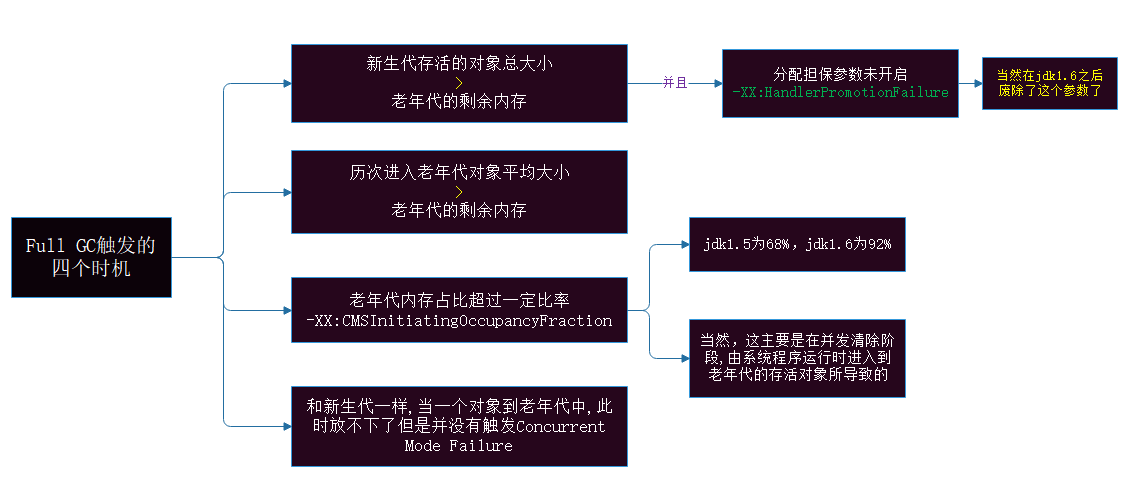
从该参数可以看到，在CMS进行Full GC时，因为还有两个步骤是系统程序也在运行的，此时CMS它会预留一部分内存空间，让正在运行的系统程序创建出的对象存放的，放的下放不下就要看该参数以及当前老年代的内存了。

（3）内存碎片问题

因为CMS是采用标记-清除算法的，所以每次清除之后就会产生很多的内存碎片，这样就会导致老年代中很难找到一块比较大的、连续的内存，此时如果有一个稍大点的对象进入老年代就会导致提前触发Full GC，所以CMS垃圾回收器也提供了两个参数，分别是-XX:+UseCMSCompactAfFullCollection和-XX:CMSFullGCsBeforeCompaction。

第一个参数设置了之后，就会在每次的Full GC之后进行STW并标记-整理，而第二个参数的意思就是在多少次Full GC之后进行一次标记-整理，默认是0，即每次Full GC之后都要STW标记-整理，这样内存碎片就很少了。

## 触发Full GC的4个时机



### 新生代中存活对象大于老年代剩余空间且担保机制未开启

当进行Minor GC之前，检查到老年代的内存放不下新生代中的所有对象，此时分配担保参数没有打开;//但是在jdk1.6之后分配担保参数就失效了，所以直接Minor GC。

### 历次进入老年代的平均大小大于老年代剩余内存

分配担保参数打开后，但是历次进入老年代中的平均对象大小，大于老年代中剩余内存大小；

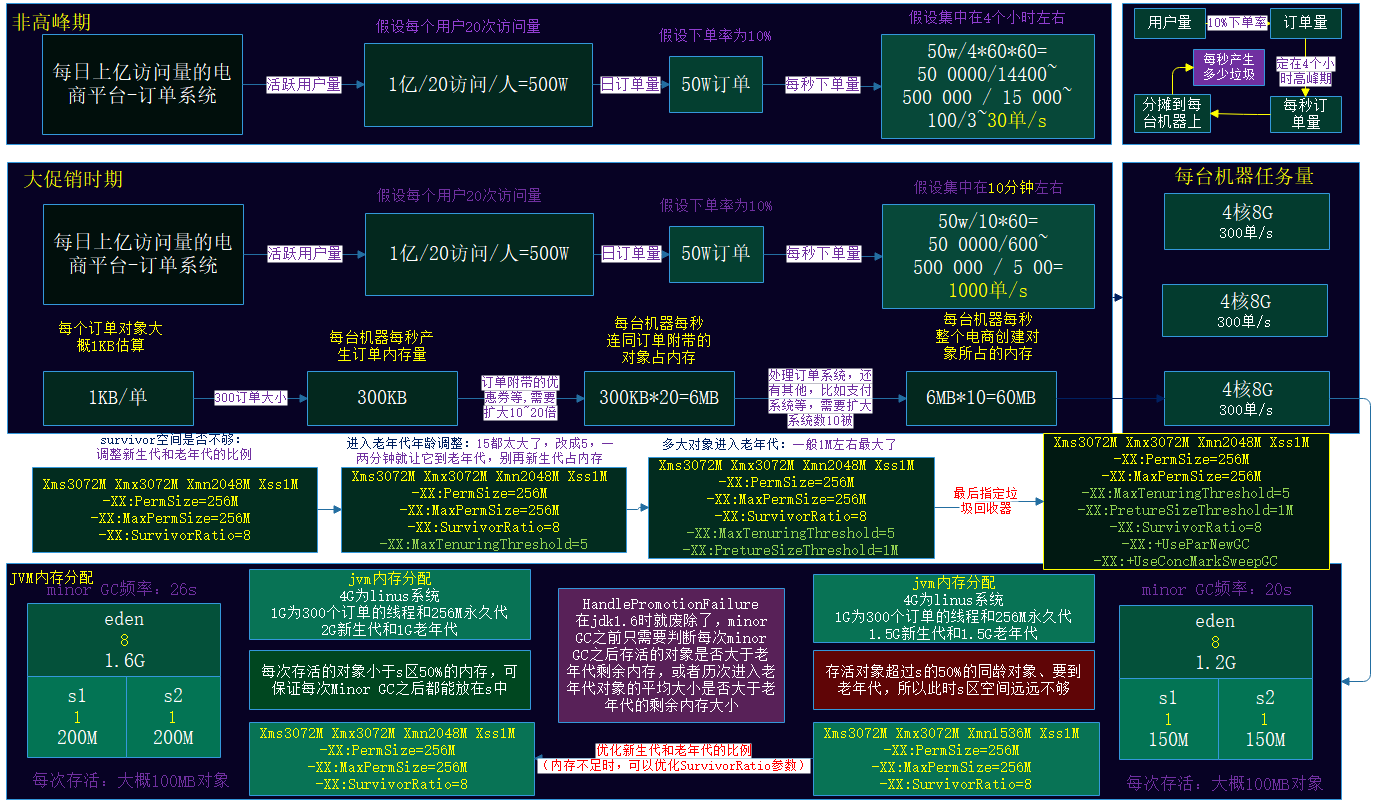
### 老年代内存占比一定时

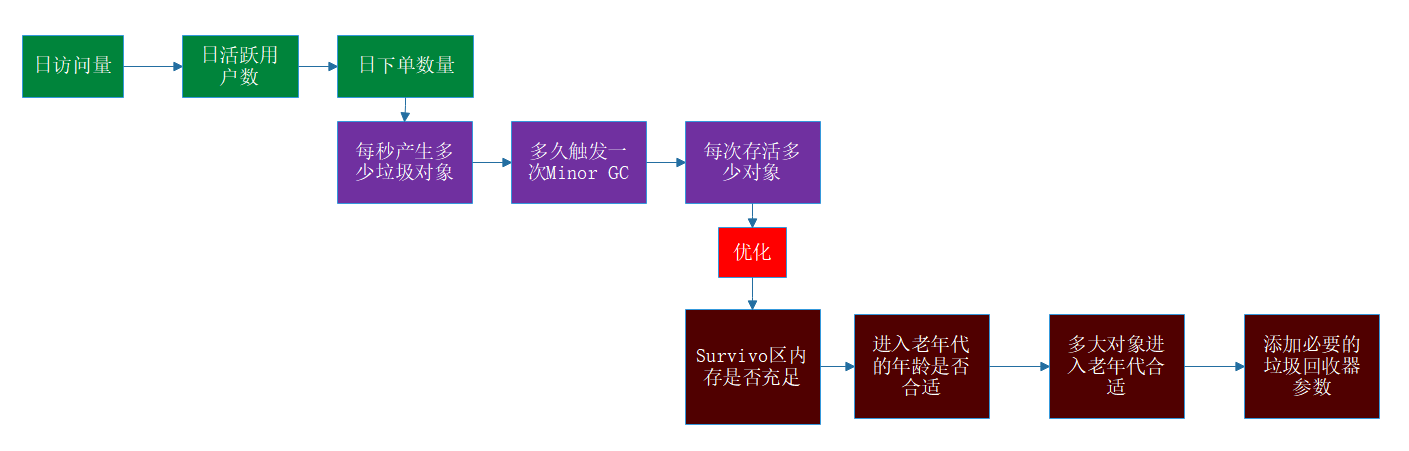
就是本节提到的，当老年代的对象，占老年代的比率大于参数-XX:CMSInitializingOccupancyFraction时，此时就会触发Concurrent Mode Failure，换成Serial Old重新STW来一次Full GC。

### 老年代放不下大对象

和新生代一样，某一次来了一个大对象，老年代放不下。//可能是和历次对象判断有偏差。

## 上亿请求量的电商系统：年轻代垃圾回收参数的优化





## 上亿请求量的电商系统：老年代垃圾回收参数的优化

对于老年代而言，因为新生代优化的工作比较好，所以可以保证半小时到一个小时才会发生一次full gc，假设过了一个小时后老年代的内存为900MB，此时可能会发生Full GC，因为Full GC时，系统程序也是也可并发运行的，此时如果在并发清除时来二十多M的垃圾对象到老年代，就会导致Concurrent Mode Failure。

其他的，比如老年代整理碎片这些配置用默认即可，所以完整JVM参数配置如下所示：

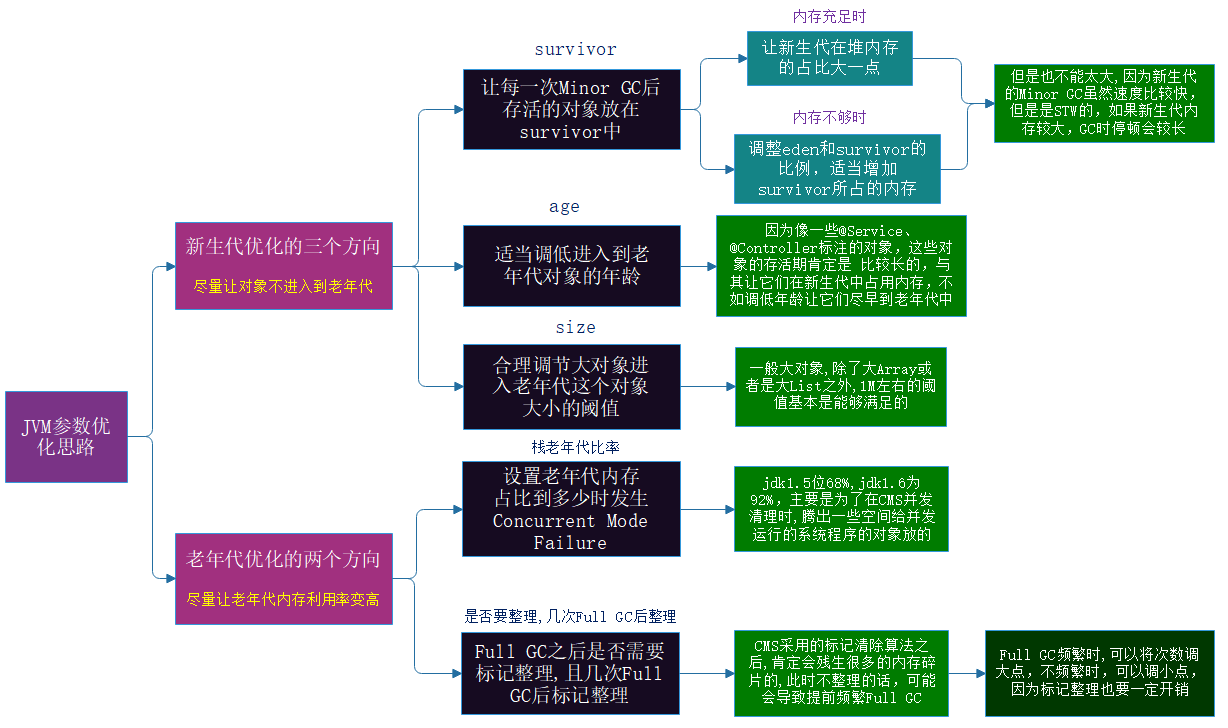
**Xms3072M Xmx3072M Xmn2048M Xss1M –XX:PermSize=256M –XX:MaxPermSize=256M**

**–XX:SurvivorRatio=8 –XX:MaxTenuringThreshold=5 –XX:PretenureSizeThreshold=1M**

**–XX:CMSInitiatingOccupancyFraction=92**

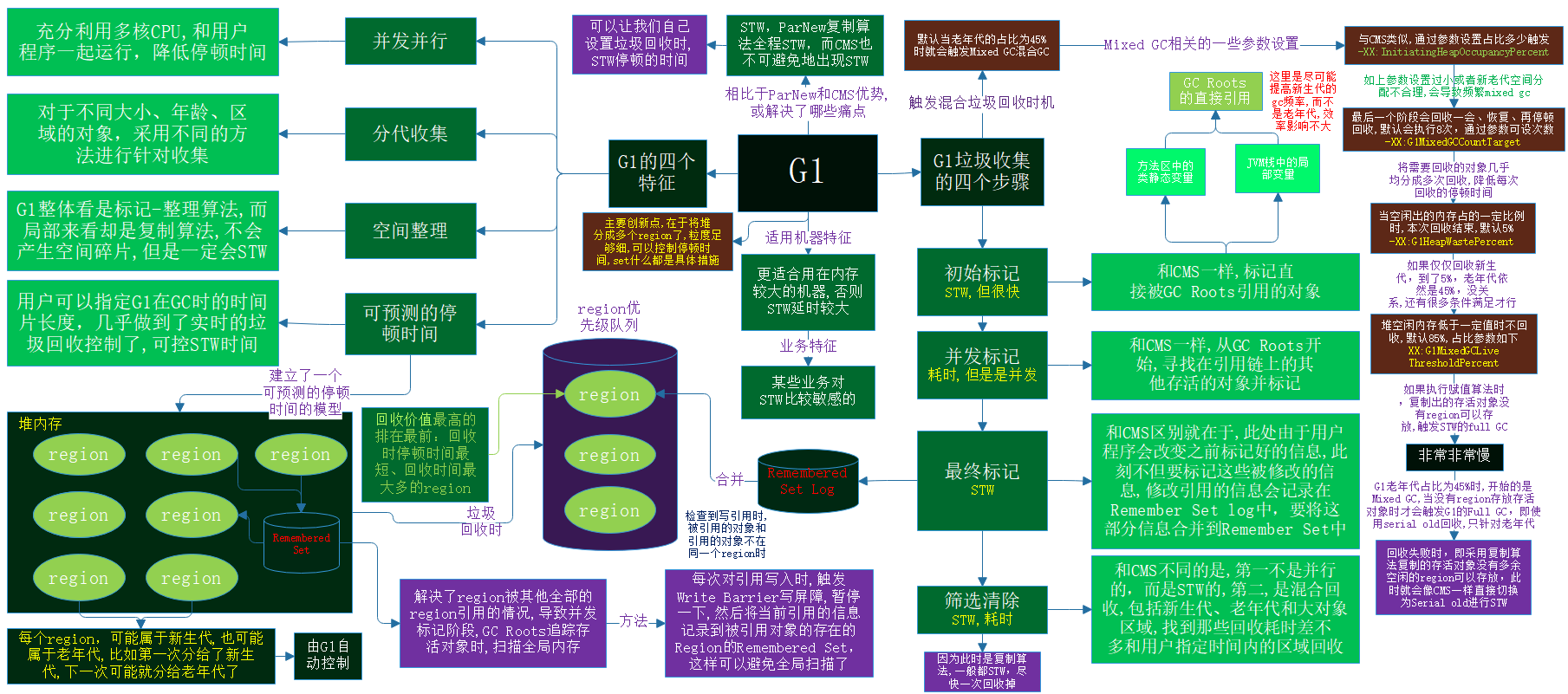
**–XX:+UseCMSCompactAtFullCollection –XX:CMSFullGCsBeforeCompaction=0**

## ParNew+CMS:JVM参数优化的5个关键思路

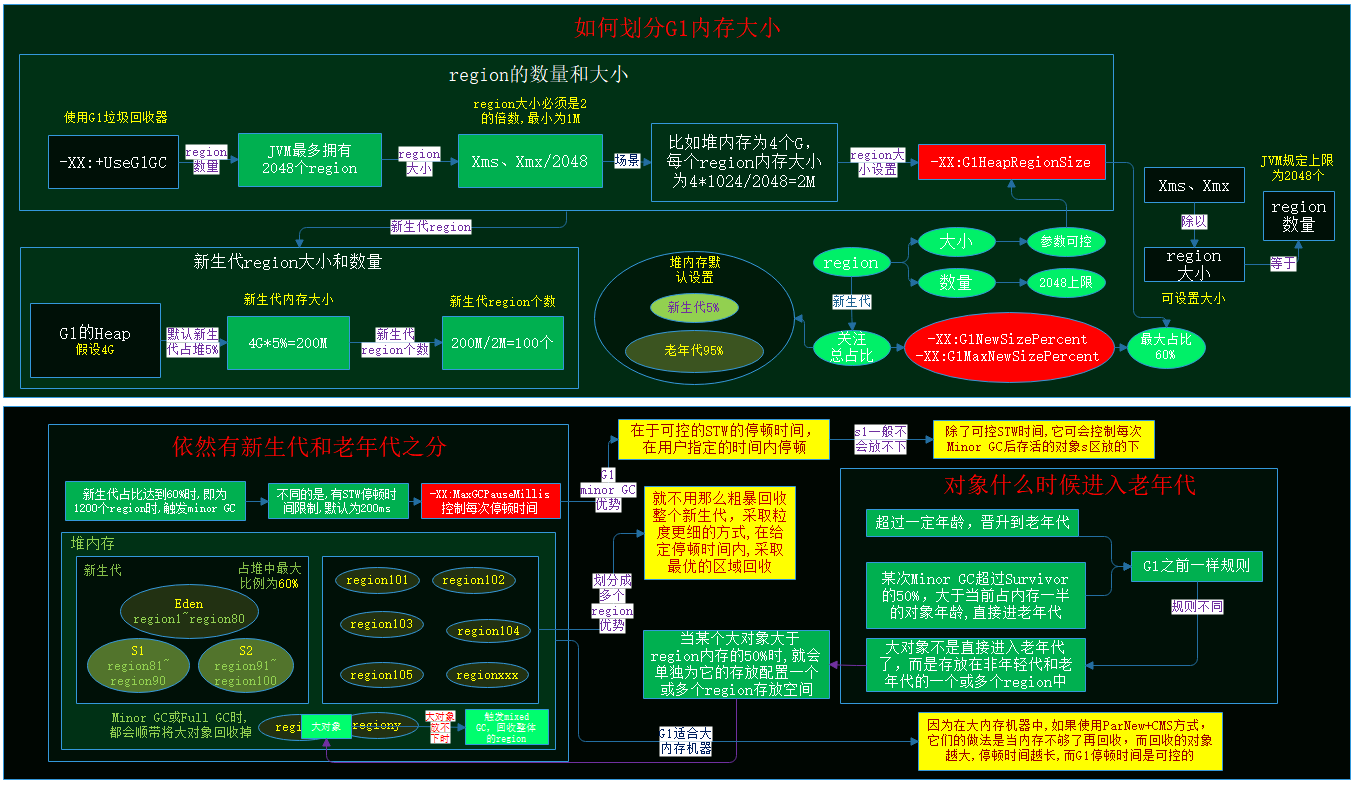


# 深入分析最新G1垃圾回收器

## G1的主要特征和核心原理



## G1的内存分配和对象流转原理



## 百万级教育平台-基于G1垃圾回收器优化

