

**Javac编译过程**

**Javac编译器的作用**

将符合Java语言规范的**源代码**转化成符合Java虚拟机规范的Java**字节码**。

Javac编译过程大致分为4个过程，分别是：

1、词法分析

词法分析是将源代码的字符流转变为标记（Token）集合。

2、语法分析

词法分析器的作用是将Java源文件的字符流转变成对应的Token流。而语法分析器是将词法分析器分的Token流组件成更加结构化的语法树。也就是将一个个单词组装成一句话，一个完整的语句。

3、语义分析

语义分析是要在语法树的基础上再做一些处理，如给类添加默认的构造函数，检查变量在使用前是否已经初始化，将一些常量进行合并处理，检查操作变量类型是否匹配，检查所有的操作语句是否可达，检查checked exception是否正确处理。

语义分析阶段分为：填充符号表、标注检查、数据及控制流分析。

4、代码生成

字节码阶段不仅仅把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。

实例构造器<init>方法和类构造器<clinit>方法就是在这个阶段添加到语法树中的。

生成java字节码需要经过以下两个步骤：

·将java方法中的代码块转化成符合JVM语法的命令形式，JVM的操作都是基于栈的，所有的操作都必须经过出栈和进栈来完成。

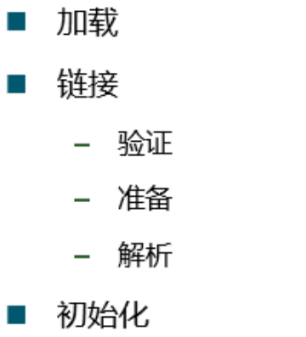
·按照JVM的文件组织格式将字节码输出到以class为扩展名的文件中。

**JVM类加载机制**

**类加载过程**

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示。





加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都已HotSpot为基准。

**·加载**

在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下3件事情：

·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**（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；

·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；

加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

**·验证**

验证是连接阶段的第一步，这一阶段的**目的是为了确保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的安全。

验证阶段大致会完成4个阶段的检验动作：

·1.**文件格式验证**：验证字节流是否符合Class文件格式的规范；例如：是否以魔术0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。

·2.**元数据验证**：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。

·3.**字节码验证**：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。

·4.**符号引用验证**：确保解析动作能正确执行。

验证阶段是非常重要的，但**不是必须的**，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

**·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，这些变量所使用的内存都将在**方法区中**进行分配。这时候进行内存分配的**仅包括类变量（被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堆中。其次，这里所说的初始值“通常情况”下是数据类型的**零值**，假设一个类变量的定义为：

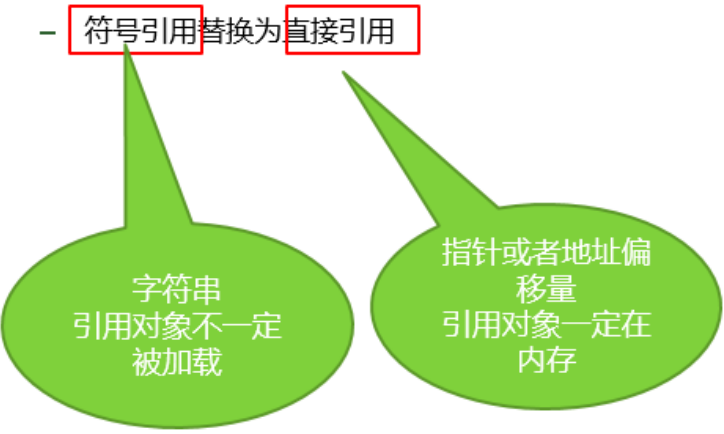
public static int value=123;

那变量value在准备阶段过后的**初始值为0**而不是123.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<clinit>()方法之中，所以把value赋值为123的动作将在**初始化阶段**才会执行。

至于“特殊情况”是指：public static final int value=123，即当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以标注为**final**之后，**value的值在准备阶段初始化为123而非0**.

**·解析**

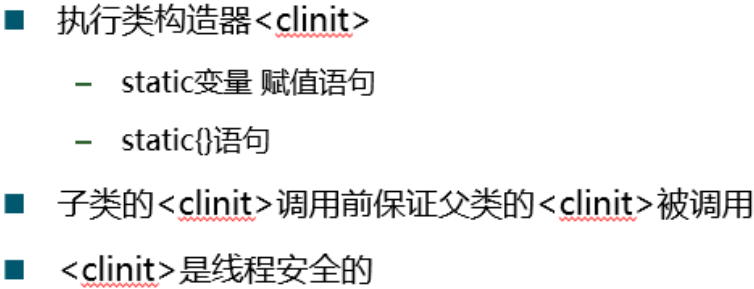
解析阶段是虚拟机**将常量池内的符号引用替换为直接引用的过程**。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。



**·初始化**

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备阶段，变量已经赋过一次系统要求的初始值，而在**初始化阶段，则根据程序猿通过程序制定的主管计划去初始化类变量和其他资源**，或者说：初始化阶段是**执行类构造器<clinit>()方法的过程**.

<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{}**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，在前面的静态语句块可以赋值，但是不能访问。



<clinit>()方法与实例构造器<init>()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<init>()方法执行之前，父类的<clinit>()方法方法已经执行完毕。

<clinit>()方法对于类或者接口来说并**不是必需的**，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生产<clinit>()方法。

**接口中不能使用静态语句块**，但**仍然有变量初始化的赋值操作**，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，**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使用时，父接口才会初始化**。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的**加锁、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去执行这个类的<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。如果在一个类的<clinit>()方法中有好事很长的操作，就可能造成多个线程阻塞，在实际应用中这种阻塞往往是隐藏的。

需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其**他线程唤醒之后不会再次进入<clinit>()方法**。同一个类加载器下，一个类型只会**初始化一次**。

**初始化额外：**

虚拟机规范严格规定了有且只有5中情况（jdk1.7）必须对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1.遇到new,getstatic,putstatic,invokestatic这失调字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、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，以及调用一个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，如果类没有进行过初始化，则需要先触发其初始化。

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，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

5.当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果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初始化，则需要先出触发其初始化。

**注意：通过子类引用付了的静态字段，不会导致子类初始化。**

站在Java开发人员的角度来看，**类加载器可以大致划分为以下三类**：

**启动类加载器： BootstrapClassLoader**，负责加载存放在**JDK\jre\lib**(JDK代表JDK的安装目录，下同)下，或被 -Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的**java.开头的类**均被 BootstrapClassLoader加载）。启动类加载器是无法被Java程序直接引用的。

**扩展类加载器： ExtensionClassLoader**，该加载器由 sun.misc.Launcher$ExtClassLoader实现，它负责加载 **JDK\jre\lib\ext**目录中，或者由 java.ext.dirs系统变量指定的路径中的所有类库（如javax.开头的类），开发者可以直接使用扩展类加载器。

**应用程序类加载器：** ApplicationClassLoader，该类加载器由 sun.misc.Launcher$AppClassLoader来实现，它负责加载**用户类路径（ClassPath）所指定的类**，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。

**JVM类加载机制**

**·全盘负责**，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

**·父类委托**，**先让父类加载器试图加载该类**，**只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类**

**·缓存机制**，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

**双亲委派模型**

双亲委派模型的工作流程是：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，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

**双亲委派机制:**

·1、当 AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

·2、当 ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader```去完成。

·3、如果 BootStrapClassLoader加载失败（例如在 $JAVA\_HOME/jre/lib里未查找到该class），会使用 ExtClassLoader来尝试加载；

·4、若ExtClassLoader也加载失败，则会使用 AppClassLoader来加载，如果 AppClassLoader也加载失败，则会报出异常 ClassNotFoundException。

**JVM内存空间解析**

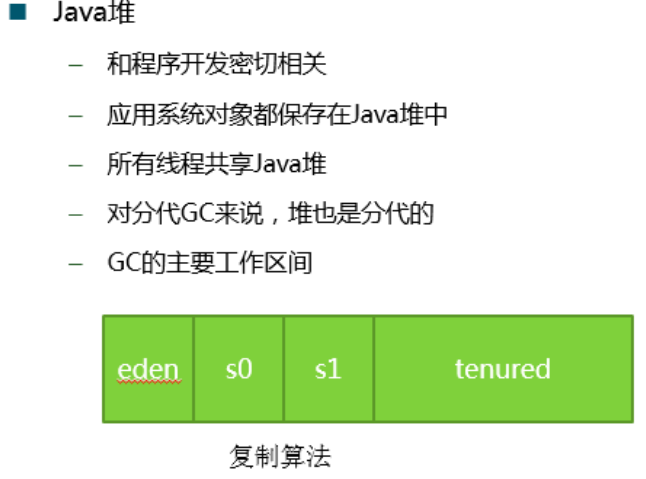
**（一）**



方法区和持久代的关系：

方法区**物理上存在于堆里**，而且是在堆的持久代里面；但在**逻辑上，方法区和堆是独立的**。一般说**堆的持久代**就是说方法区，因为一旦JVM把方法区（类信息，常量池，静态字段，方法）加载进内存以后，这些内存**一般是不会被回收**的了。

**（二）**



Java堆虚拟机启动时创建，存放**对象实例和数组**，所占内存最大。

Java堆

Eden区

新生代

老年代

servior区

From space

To space

8

1

**·年轻代**

所有新生成的对象首先都是放在年轻代。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代一般分3个区，1个Eden区，2个Survivor区（from 和 to）。

大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当另一个Survivor区也满了的时候，从前一个Survivor区复制过来的并且此时还存活的对象，将可能被复制到年老代。

2个Survivor区是对称的，没有先后关系，所以同一个Survivor区中可能同时存在从Eden区复制过来对象，和从另一个Survivor区复制过来的对象；而复制到年老区的只有从另一个Survivor区过来的对象。而且，**因为需要交换的原因，Survivor区至少有一个是空的**。特殊的情况下，根据程序需要，Survivor区是可以配置为多个的（多于2个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

针对年轻代的垃圾回收即 **Young GC**。

**·年老代**

在年轻代中经历了N次（可配置）垃圾回收后仍然存活的对象，就会被复制到年老代中。因此，可以认为年老代中存放的都是一些**生命周期较长的对象**。

针对年老代的垃圾回收即 **Full GC**。

**·持久代**

用于**存放静态类型数据**，如 Java Class, Method 等。**持久代对垃圾回收没有显著影响**。但是有些应用可能动态生成或调用一些Class，例如 hibernate CGLib 等，在这种时候往往需要设置一个比较大的持久代空间来存放这些运行过程中动态增加的类型。

·当一组对象生成时，内存申请过程如下：

1.JVM会试图为相关Java对象在年轻代的Eden区中初始化一块内存区域。

2.当Eden区空间足够时，内存申请结束。否则执行下一步。

3.JVM试图释放在Eden区中所有不活跃的对象（Young GC）。释放后若Eden空间仍然不足以放入新对象，JVM则试图将部分Eden区中活跃对象放入Survivor区。

4.Survivor区被用来作为Eden区及年老代的中间交换区域。当年老代空间足够时，Survivor区中存活了一定次数的对象会被移到年老代。

5.当年老代空间不够时，JVM会在年老代进行完全的垃圾回收（Full GC）。

6.Full GC后，若Survivor区及年老代仍然无法存放从Eden区复制过来的对象，则会导致JVM无法在Eden区为新生成的对象申请内存，即出现“Out of Memory”。

**·OOM（“Out of Memory”）异常一般主要有如下2种原因：**

1. **年老代溢出**，表现为：java.lang.OutOfMemoryError:Javaheapspace

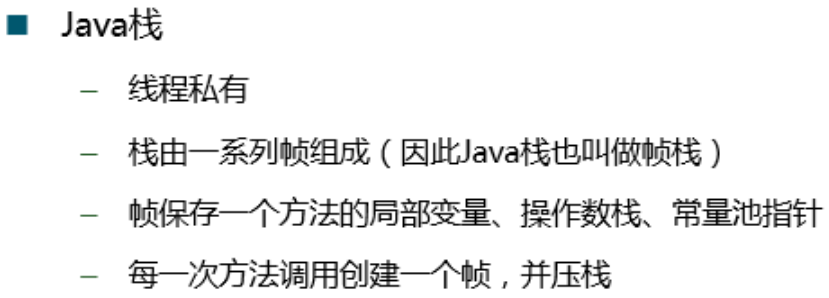
这是最常见的情况，产生的原因可能是：**设置的内存参数Xmx过小**或**程序的内存泄露及使用不当**问题。（-Xmx 设置最大堆内存，-Xms 设置初始堆内存）

例如循环上万次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存。还有的时候虽然不会报内存溢出，却会使系统不间断的垃圾回收，也无法处理其它请求。这种情况下除了**检查程序**、**打印堆内存**等方法排查，还可以借助一些内存分析工具，比如MAT就很不错。

2. **持久代溢出**，表现为：java.lang.OutOfMemoryError:PermGenspace

通常由于**持久代设置过小**，**动态加载了大量Java类而导致溢出**，解决办法唯有**将参数 -XX:MaxPermSize 调大**（一般256m能满足绝大多数应用程序需求）。将部分Java类放到容器共享区（例如Tomcat share lib）去加载的办法也是一个思路，但前提是容器里部署了多个应用，且这些应用有大量的共享类库。

（三）



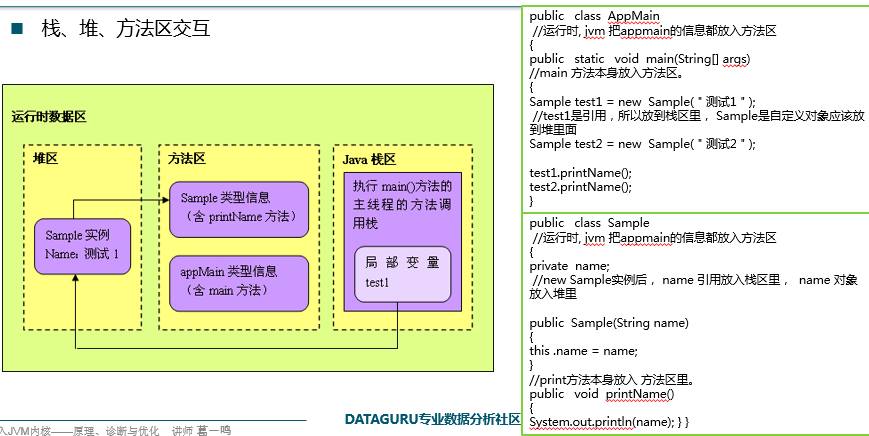
（四）本地方法栈（native 方法）

**线程私有**。与Java栈类似，但是不是为Java方法（字节码）服务，而是为本地非Java方法服务。也会抛StackOverflowError和OOM。

（五）程序计数器

**线程私有**。是一块**较小的内存**，是当前线程所执行的**字节码的行号指示器**。是Java虚拟机规范中**唯一没有规定OOM**（OutOfMemoryError）的区域。

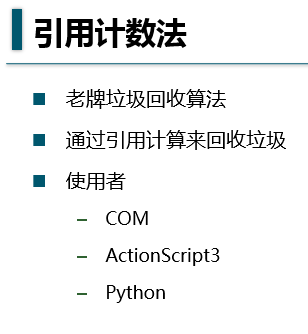
结合：

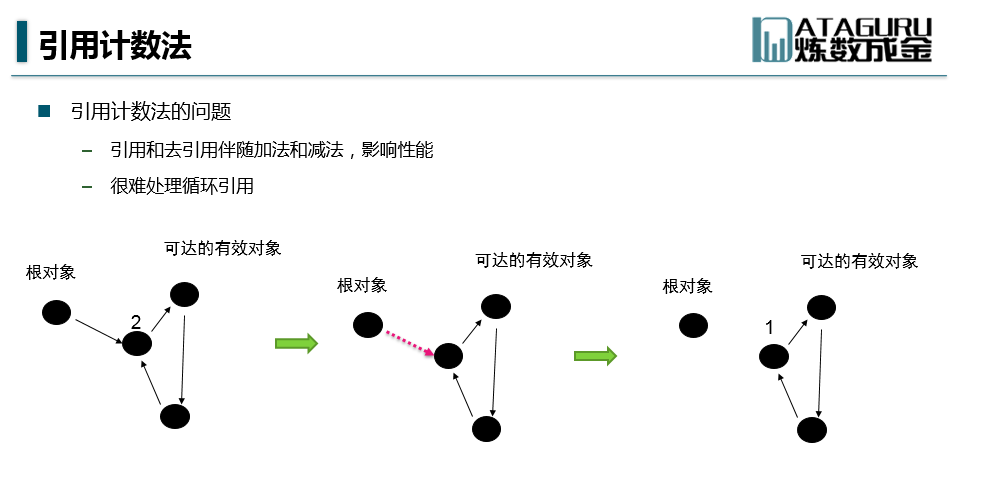
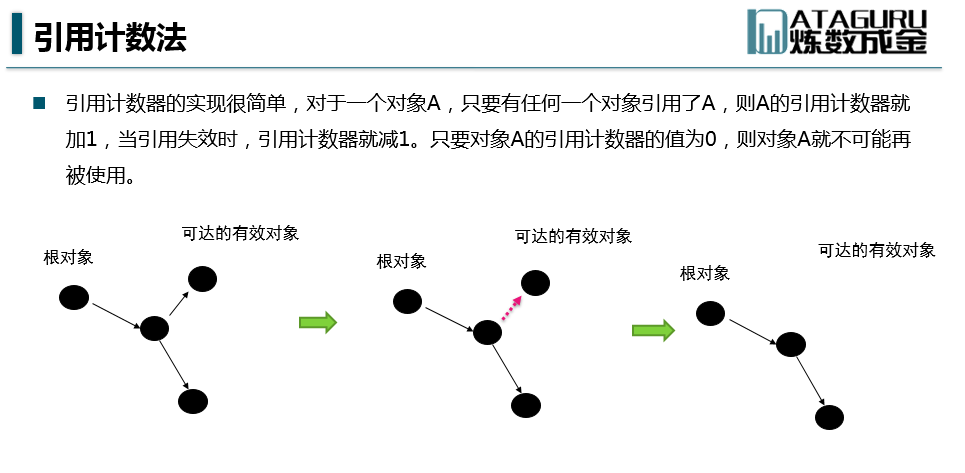


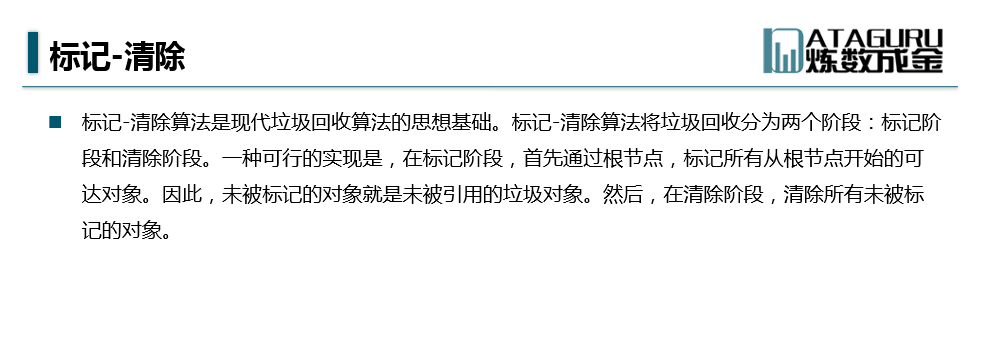
**JVM垃圾回收(GC)机制**

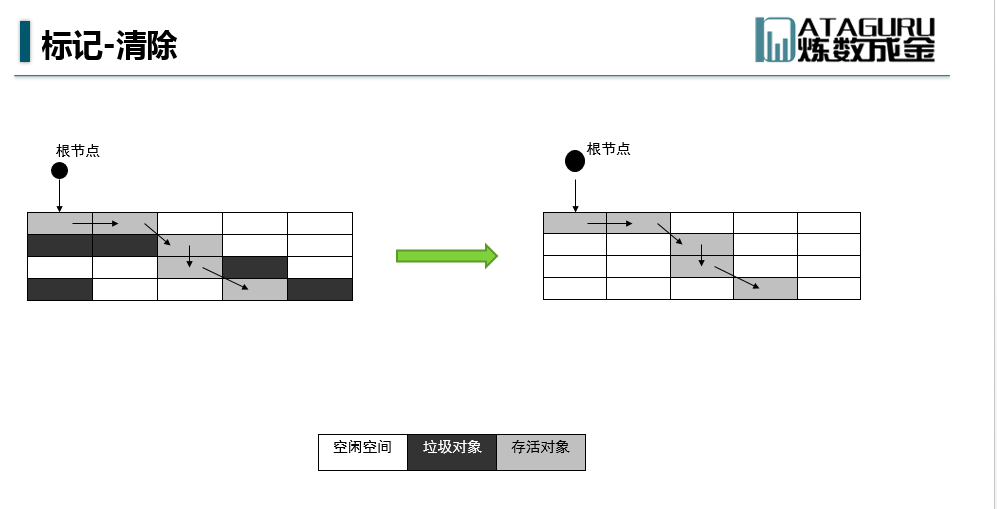
**（一）**

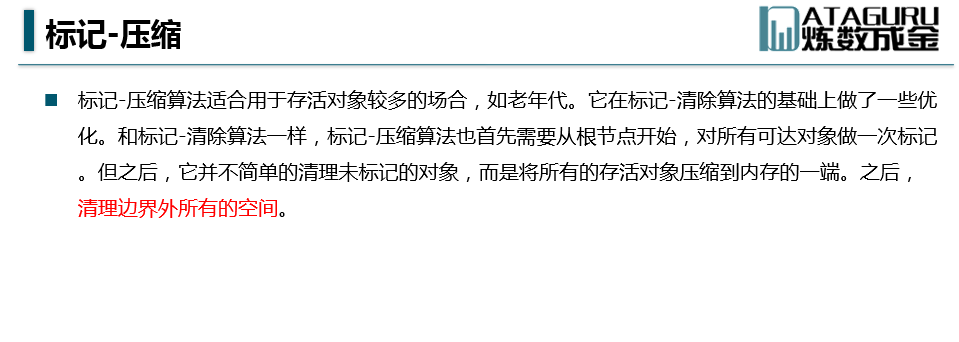


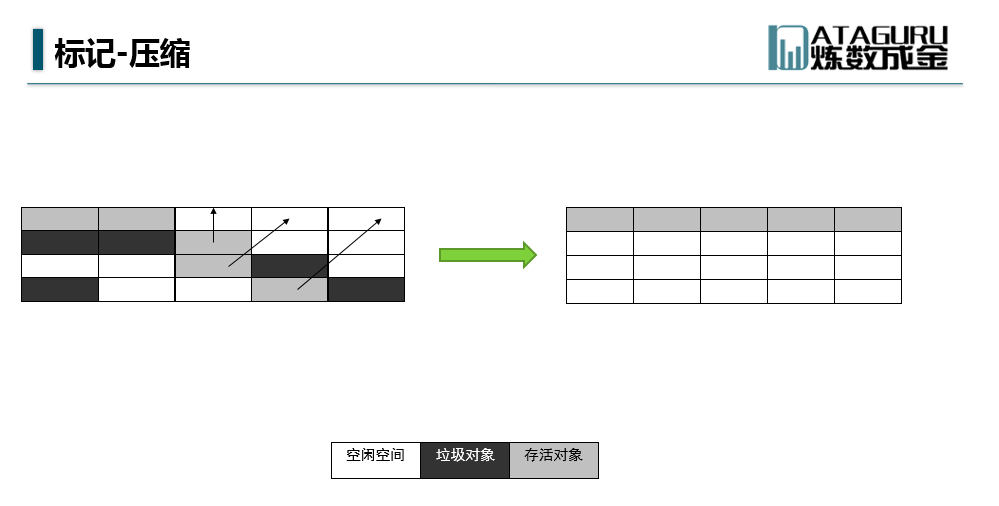


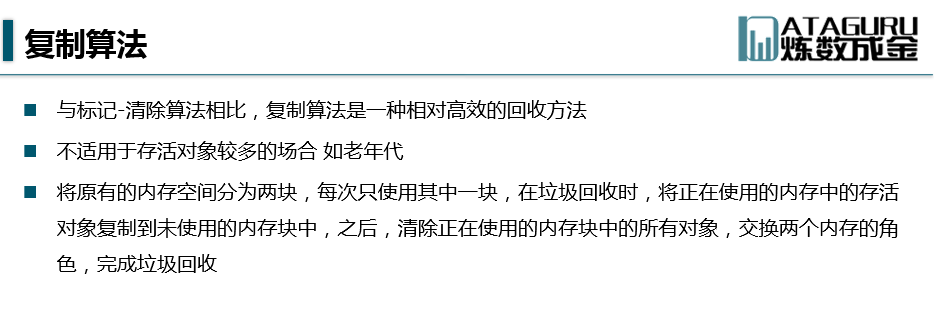


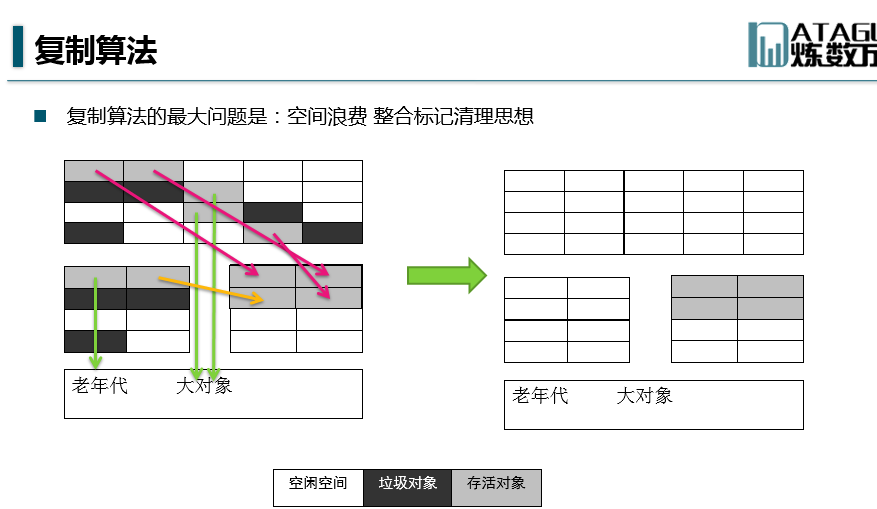
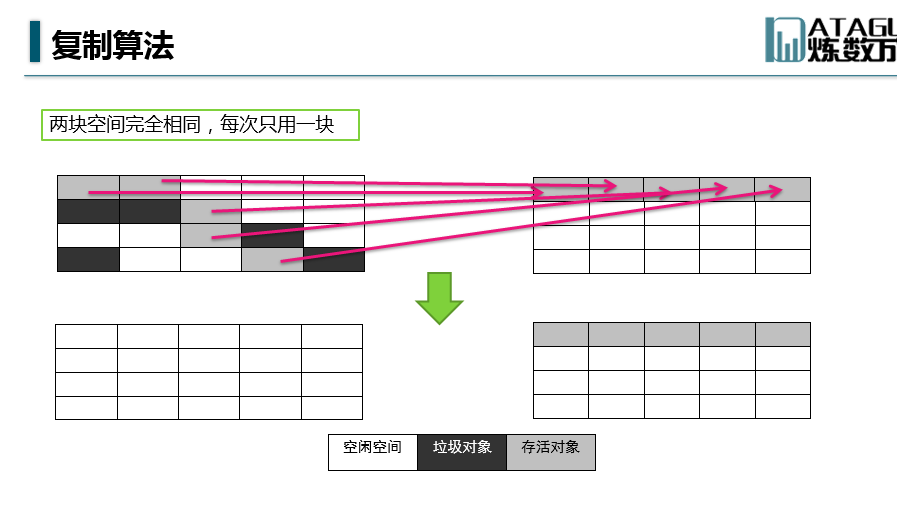




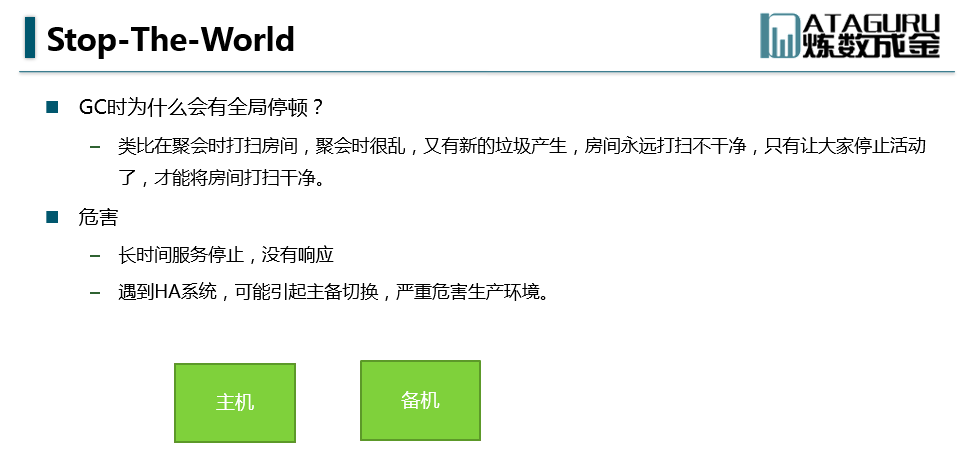
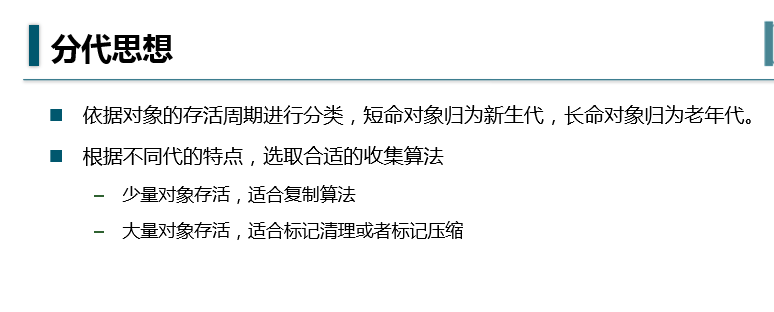








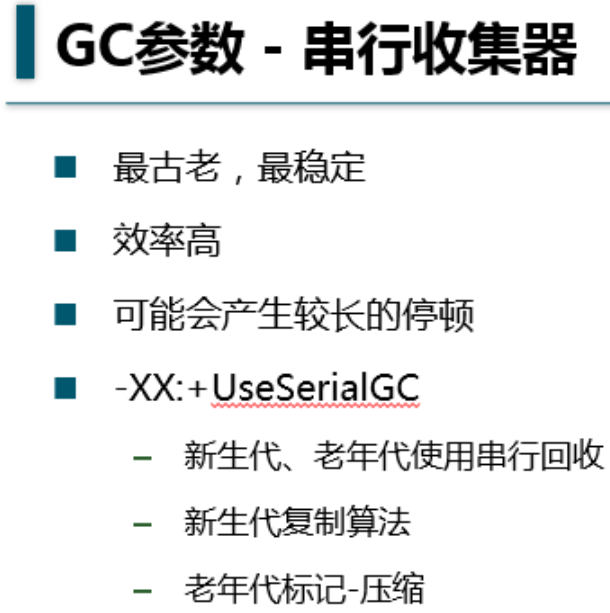
Java中采用以下思想:



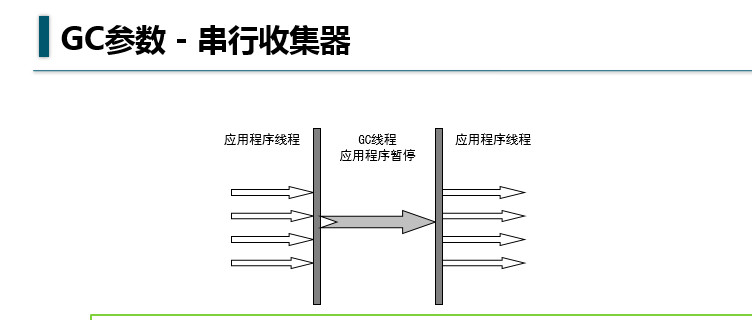
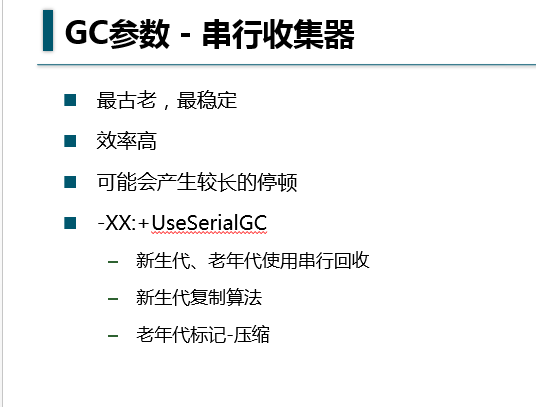
**·垃圾回收器选择**

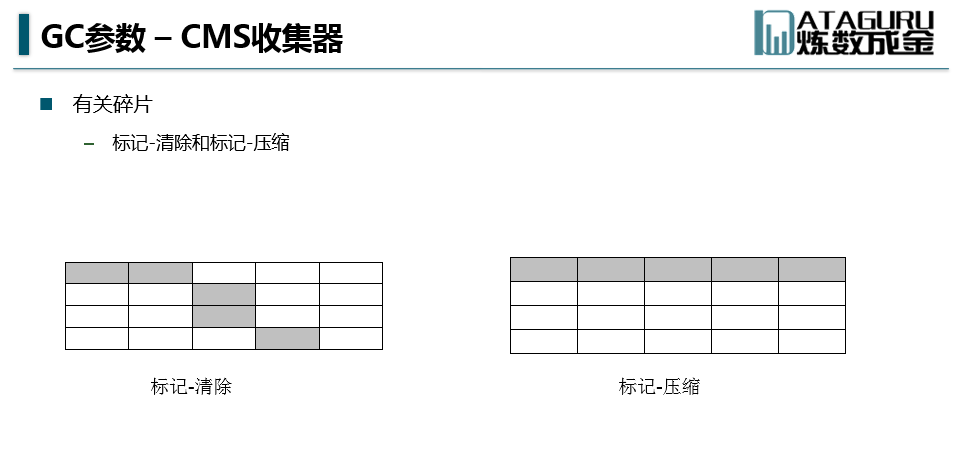
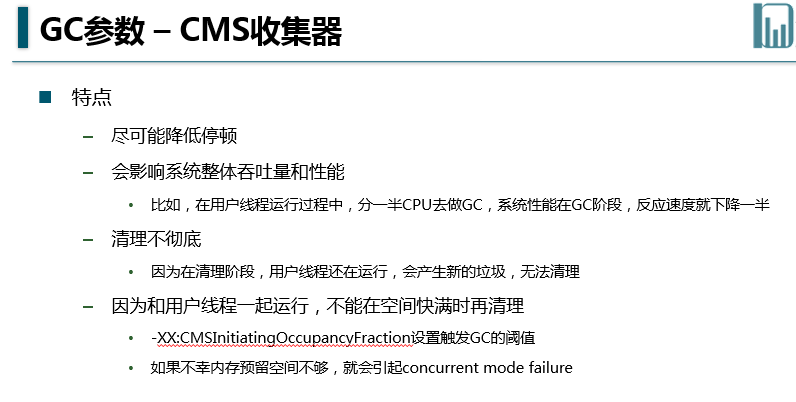
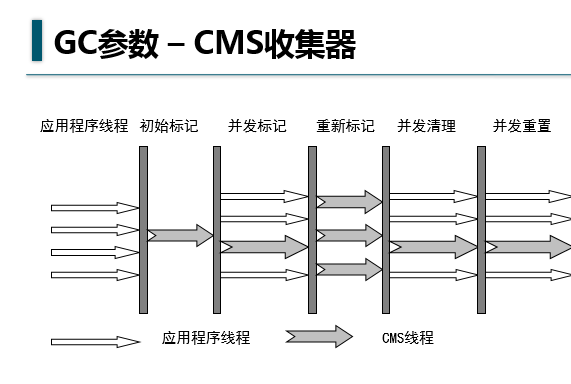
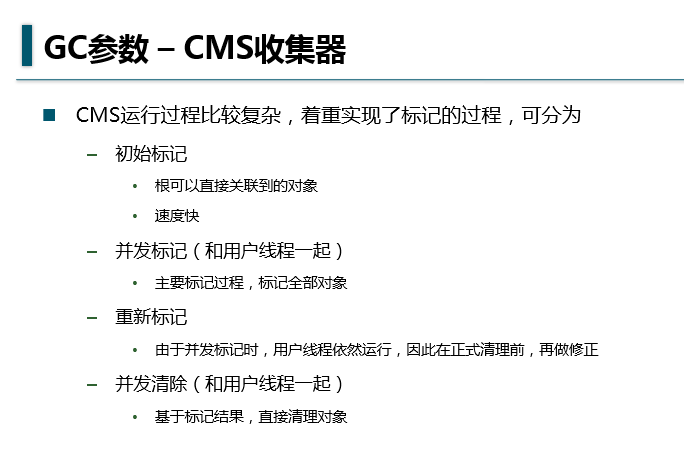
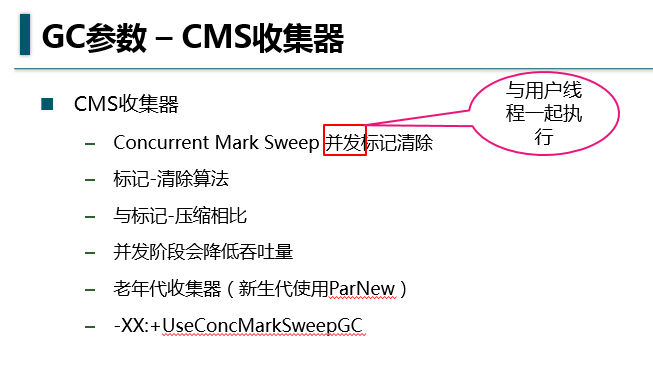
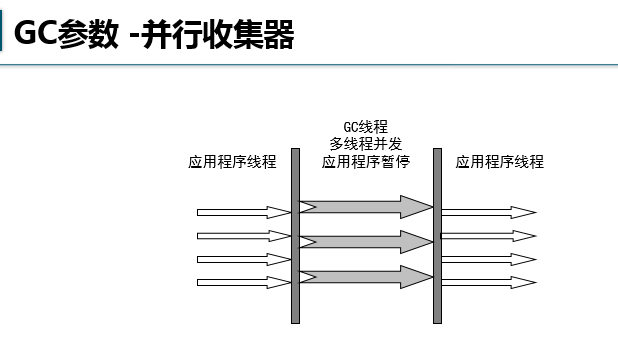
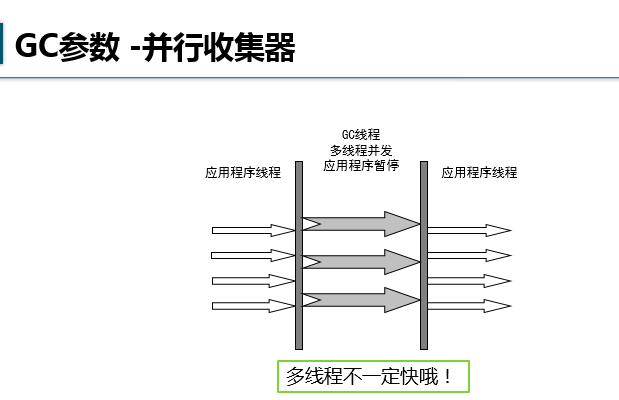
JVM给出了3种选择：串行收集器、并行收集器、并发收集器。**串行收集器只适用于小数据量**的情况，所以生产环境的选择主要是并行收集器和并发收集器。

默认情况下JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置进行智能判断。

****







**调优命令**

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

**jps**，JVM Process Status Tool,**显示指定系统内所有的HotSpot虚拟机进程**。

**jstat**，JVM statistics Monitoring是用于**监视虚拟机运行时状态信息**的命令，它可以**显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据**。

**jmap**，JVM Memory Map命令用于**生成heap dump文件**

**jhat**，JVM Heap Analysis Tool命令是**与jmap搭配使用**，用来**分析jmap生成的dump**，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，**可以在浏览器中查看**

**jstack**，用于**生成java虚拟机当前时刻的线程快照**。

**jinfo**，JVM Configuration info 这个命令作用是**实时查看和调整虚拟机运行参数**。