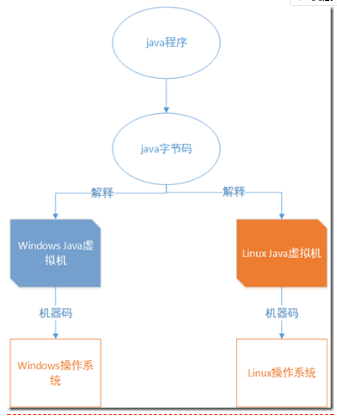
## 虚拟机理解

### 虚拟机简介

所谓虚拟机，就是一台虚拟的机器。他是一款软件，用来执行一系列虚拟计算指令，大体上虚拟机可以分为系统虚拟机和程序虚拟机， 大名鼎鼎的Visual Box、VMware就属于系统虚拟机，他们完全是对物理计算的仿真，提供了一个可以运行完整操作系统的软件平台。程序虚拟机典型代码就是Java虚拟机，它专门为执行单个计算程序而计算，在Java虚拟机中执行的指令我们成为Java自己码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。Java发展至今，出现过很多虚拟机，做初Sun使用的一款叫ClassIc的Java虚拟机，到现在引用最广泛的是HotSpot虚拟机，除了Sum意外，还有BEA的Jrockit，目前Jrockit和HostSopt都被oralce收入旗下，大有整合的趋势。

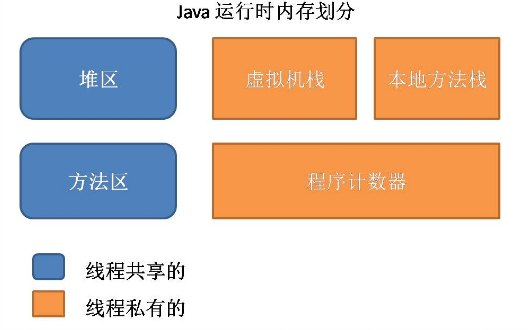
### JVM简介

Java虚拟机(JVM)实现了程序与操作系统的分离，从而实现了Java 的跨平台



## 内存管理机制

### Java内存结构



**类加载子系统**

负责从文件系统或者网络加载Class信息，加载的信息存放在一块称之方法区的内存空间。

**方法区**

为永久区，static不要定义太多，垃圾回收机制不会回收、class类的信息、常量信息、常量池信息、包括字符串字面量和数字常量等，所有线程共享，存在线程安全问题。

**栈 stack**

Java栈是一块线程私有的空间，一个栈，一般由三部分组成:局部变量表、操作数据栈和帧数据区

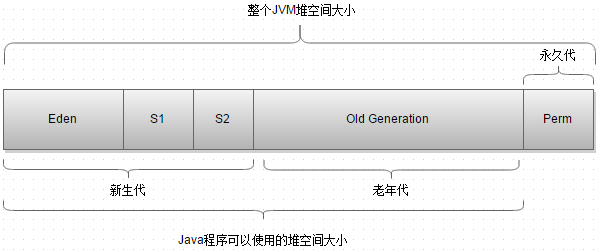
局部变量表：用于函数的参数及局部变量

操作数栈：主要保存计算过程的中间结果，同时作为计算过程中的变量临时的存储空间。

帧数据区:除了局部变量表和操作数据栈以外，栈还需要一些数据来支持常量池的解析，这里帧数据区保存着访问常量池的指针，方便计程序访问常量池，另外当函数返回或出现异常时卖虚拟机子必须有一个异常处理表，方便发送异常的时候找到异常的代码，因此异常处理表也是帧数据区的一部分。

**堆 java heap**

在Java虚拟机启动的时候建立Java堆，它是Java程序最主要的内存工作区域，几乎所有的对象实例都存放到，堆是Java 虚拟机所管理的内存中最大的一块,堆空间是所有线程共享，存在线程安全问题。堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为新生代和老年代。其中新生代存放新生的对象或者年龄不大的对象，老年代则存放老年对象。新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色的空间。绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，即如果对象在频繁的使用，则进入老年代。垃圾回收机制主要回收新生代，回收次数比老年代多。



在 JDK 1.8中移除整个永久代，取而代之的是一个叫元空间（Metaspace）的区域（永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制）。

**直接内存  Direct Memor**

直接内存并不是JVM管理的内存，可以这样理解，直接内存，就是JVM以外的机器内存，比如，你有4G的内存，JVM占用了1G，则其余的3G就是直接内存，JavaNio库允许Java程序直接内存，从而提高性能，通常直接内存速度会优于Java堆。读写频繁的场合可能会考虑使用。

**本地方法栈**

Native Method Stack中登记native方法，在Execution Engine执行时加载native libraies，本地方法栈与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法服务

**PC（Program Couneter）寄存器**

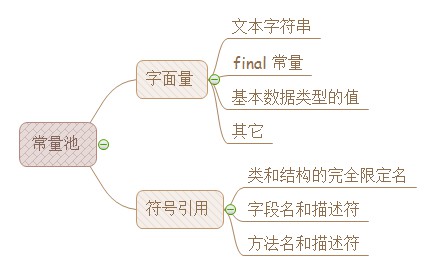
每个线程私有的空间， Java虚拟机会为每个线程创建PC寄存器，在任意时刻，一个Java线程总是在执行一个方法，这个方法称为当前方法，如果当前方法不是本地方法，PC寄存器总会执行当前正在被执行的指令，如果是本地方法，则PC寄存器值为Underfined，寄存器存放如果当前执行环境指针、程序技术器、操作栈指针、计算的变量指针等信息。

**执行引擎**

负责执行虚拟机的字节码，一般会先进行编译成机器码后执行。

**运行时常量池（Runtime Constant Pool）**

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在加载后进入方法区的运行时常量池中存放。



## 垃圾回收机制

### 自动垃圾回收

**概念**

不定时去堆内存中清理不可达对象。不可达的对象并不会马上就会直接回收， 垃圾收集器在一个Java程序中的执行是自动的，不能强制执行，即使程序员能明确地判断出有一块内存已经无用了，是应该回收的，程序员也不能强制垃圾收集器回收该内存块。程序员唯一能做的就是通过调用System.gc 方法来"建议"执行垃圾收集器，但其是否可以执行，什么时候执行却都是不可知的。这也是垃圾收集器的最主要的缺点。当然相对于它给程序员带来的巨大方便性而言，这个缺点是瑕不掩瑜的。

ps:内存泄露是指该内存空间使用完毕之后未回收，在不涉及复杂数据结构的一般情况下，Java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有时也将其称为“对象游离”。内存溢出：内存不够申请的大小。

自动垃圾回收是一种在堆内存中找出哪些对象在被使用，还有哪些对象没被使用，并且将后者删掉的机制。所谓使用中的对象（已引用对象），指的是程序中有指针指向的对象；而未使用中的对象（未引用对象），则没有被任何指针给指向，因此占用的内存也可以被回收掉。在JDK1.2之前，创建的对象只有在处于可触及（reachable）的状态下，才能被程序使用。也就是说，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。垃圾回收器一旦发现这些无用对象，就会对其进行回收。但是，在某些情况下，我们会希望有些对象不需要被立即回收，或者说从全局的角度来说没有立即回收的必要性。比如缓存系统的设计，在内存不吃紧或者说为了提高运行效率的情况下，一些暂时不用的对象仍然可放置在内存中，而不是立即进行回收。为了满足这种要求，从JDK1.2版本开始，Java的设计人员把对象的引用细分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）和虚引用（Phantom Reference）四种级别，细分的准则是体现在被GC回收的优先级上：强引用>软引用>弱引用>虚引用。这样，从JDK1.2开始，GC垃圾回收器回收对象时，对象的有效性分析不仅仅是需要考虑对象可达性，还需要考虑对象的引用强度，从而使程序可以更加灵活地控制对象的生命周期。可以用一个公式概括：对象的有效性=可达性+引用类型。java.lang.ref包就是创建对象的各种引用及使用细节的api。

**引用介绍**

1.强引用（FinalReference）

Student student = new Student(); // 这就是强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。FinalReference是不公开的类，不让程序员实例包装，强引用jvm会自动包装。

2.软引用（SoftReference）

Student student = new Student();

SoftReference softReference = new SoftReference(student);

如果一个对象只具有软引用，那就类似于可有可物的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存，一般缓存框架都会用到，既避免了对象不回收造成的内存浪费，也可以有效避免创建新对象的资源消耗。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA虚拟机就会把这个软引用加入到与之关联的引用队列中。

3.弱引用（WeakReference）

Student student = new Student();

WeakReference weakReference = new WeakReference(student);  
如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4.虚引用（PhantomReference）

ReferenceQueue referenceQueue = new ReferenceQueue();

PhantomReference phantomReference = new PhantomReference(object， queue);  
"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

**特别注意**

在程序设计中一般很少使用弱引用与虚引用，使用软用的情况较多，这是因为软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。

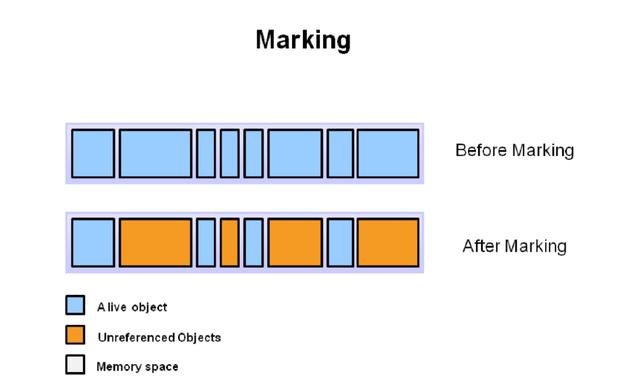
一般项目中用的都是强引用类似Student student = new Student();但在一些缓存框架中会用的java.lang.ref包下的类如软引用Student student = new Student();

SoftReference softReference = new SoftReference(student);

### 垃圾回收过程

**第一步：标记**

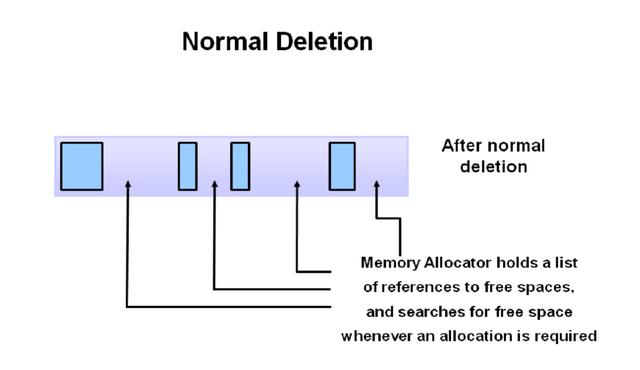
垃圾回收的第一步是标记。垃圾回收器此时会找出哪些内存在使用中，还有哪些不是。



上图中，蓝色表示已引用对象，橙色表示未引用对象。垃圾回收器要检查完所有的对象，才能知道哪些有被引用，哪些没。如果系统里所有的对象都要检查，那这一步可能会相当耗时间

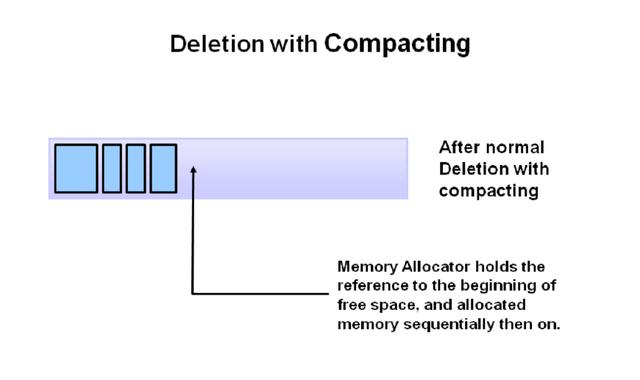
**第二步：清除**

这一步会删掉标记出的未引用对象。内存分配器会保留指向可用内存的引用，以供分配新对象。



**第三步：压缩**

为了提升性能，删除了未引用对象后，还可以将剩下的已引用对象放在一起（压缩），这样就能更简单快捷地分配新对象了。



这里必须点出一个很重要的误区：不可达的对象并不会马上就会被直接回收，而是至少要经过两次标记的过程。 第一次被标记过的对象，会检查该对象是否重写了finalize()方法。如果重写了该方法，则将其放入一个F-Query队列中，否则，直接将对象加入“即将回收”集合。在第二次标记之前，F-Query队列中的所有对象会逐个执行finalize()方法，但是不保证该队列中所有对象的finalize()方法都能被执行，这是因为JVM创建一个低优先级的线程去运行此队列中的方法，很可能在没有遍历完之前，就已经被剥夺了运行的权利。那么运行finalize()方法的意义何在呢？这是对象避免自己被清理的最后手段：如果在执行finalize()方法的过程中，使得此对象重新与GC Roots引用链相连，则会在第二次标记过程中将此对象从F-Query队列中清除，避免在这次回收中被清除，恢复成了一个“正常”的对象。但显然这种好事不能无限的发生，对于曾经执行过一次finalize()的对象来说，之后如果再被标记，则不会再执行finalize()方法，只能等待被清除的命运。之后，GC将对F-Queue中的对象进行第二次小规模的标记，将队列中重新与GC Roots引用链恢复连接的对象清除出“即将回收”集合。所有此集合中的内容将被回收。

**手动垃圾回收**

程序员手动垃圾回收，可以调用System.gc()就会 Runs the garbage collector 运行垃圾回收器

|  |
| --- |
| public class JVMDemo05 {  public static void main(String[] args) {  JVMDemo05 jvmDemo05 = new JVMDemo05();  jvmDemo05 = null;  System.gc();  }  protected void finalize() throws Throwable {  System.out.println("gc在回收对象...");  }  } |

**finalize方法作用**

Java技术使用finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object类中定义的，因此所有的类都继承了它。子类覆盖finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

### 对象存活判断

**引用计数法**

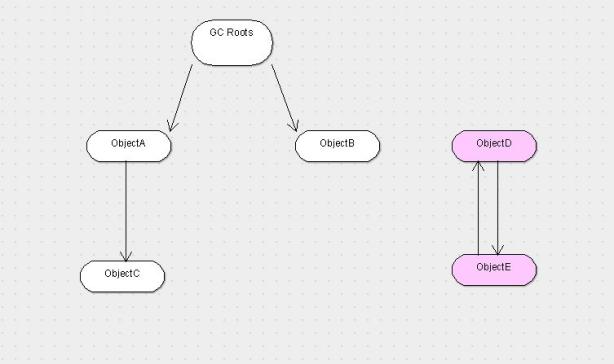
给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不再被使用的，垃圾收集器将回收该对象使用的内存。

优点：引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.而且每次加减非常浪费内存。

**根搜索算法（可达性分析算法）**

Java 和 C# 中都是采用根搜索算法来判定对象是否存活的，根搜索算法的基本思路就是通过一系列名为”GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。



ObjectD和ObjectE是互相关联的，但是由于GC roots到这两个对象不可达，所以最终D和E还是会被当做GC的对象，上图若是采用引用计数法，则A-E五个对象都不会被回收。在JAVA语言中，可以当做GC roots的对象有以下几种：

1、虚拟机栈中的引用的对象。

2、方法区中的类静态属性引用的对象。

3、方法区中的常量引用的对象。

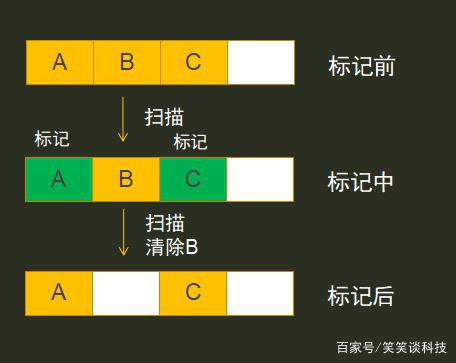
 4、本地方法栈中JNI的引用的对象。

第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

### 垃圾回收算法

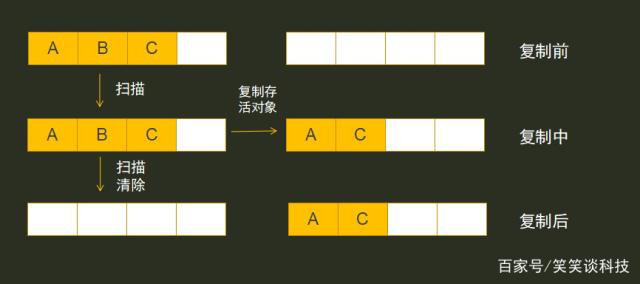
判定除了垃圾对象之后，便可以进行垃圾回收了。下面介绍一些垃圾收集算法，由于垃圾收集算法的实现涉及大量的程序细节，因此这里主要是阐明各算法的实现思想，而不去细论算法的具体实现。

**1. 标记-清除（Mark-Sweep）**



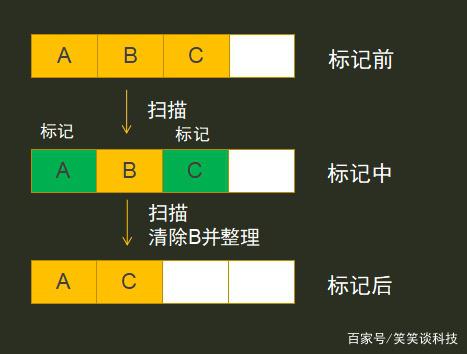
jvm会扫描所有的对象实例，通过根搜索算法，将活跃对象进行标记，jvm再一次扫描所有对象，将未标记的对象进行清除，只有清除动作，不作任何的处理，这样导致的结果会存在很多的内存碎片。

**2. 复制（copying）**



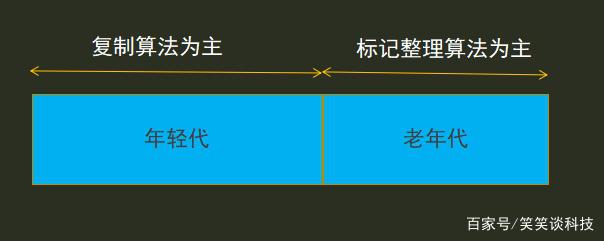
jvm扫描所有对象，通过根搜索算法标记被引用的对象，之后会申请新的内存空间，将标记的对象复制到新的内存空间里，存活的对象复制完，会清空原来的内存空间，将新的内存最为jvm的对象存储空间。这样虽然解决了内存内存碎片问题，但是如果对象很多，重新申请新的内存空间会很大，在内存不足的场景下，会对jvm运行造成很大的影响

**3. 标记-整理（Mark-compact）**



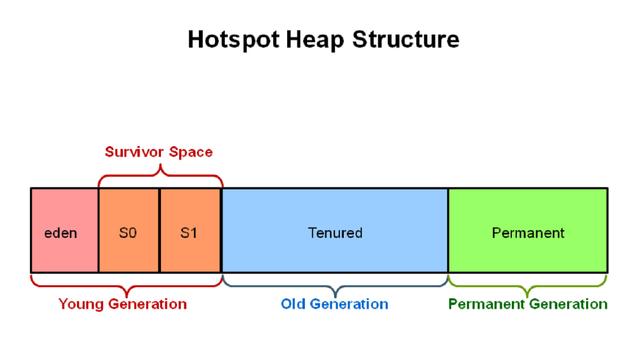
标记整理实际上是在标记清除算法上的优化，执行完标记清除全过程后，再一次对内存进行整理，将所有存活对象统一向一端移动，这样解决了内存碎片问题。

**4. 分代回收**

分代回收算法

为什么用分代机制

之前说过，逐一标记和压缩 Java 虚拟机里的所有对象非常低效：分配的对象越多，垃圾回收需时就越久。不过，根据统计，大部分的对象，其实用没多久就不用了，根据这个规律，就可以用来提升JVM的效率了。把堆分成几个部分（就是所谓的分代），分别是新生代、老年代，以及永生代



**新生代**

在新生代，每次垃圾收集器都发现有大批对象死去，只有少量存活，采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集；可以参看我之前写的java垃圾回收算法之-coping复制

**老年代**

而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须“标记－清除-压缩”算法进行回收。参看java垃圾回收算法之-标记\_清除压缩新创建的对象被分配在新生代，如果对象经过几次回收后仍然存活，那么就把这个对象划分到老年代。

老年代区存放Young区Survivor满后触发minor GC后仍然存活的对象，当Eden区满后会将存活的对象放入Survivor区域，如果Survivor区存不下这些对象，GC收集器就会将这些对象直接存放到Old区中，如果Survivor区中的对象足够老，也直接存放到Old区中。如果Old区满了，将会触发Full GC回收整个堆内存。

**永久代**

永久代包含JVM用于描述应用程序中类和方法的元数据。永久代是由JVM在运行时根据应用程序使用的类来填充的。此外，Java SE类库和方法也存储在这里。

(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)垃圾回收不会发生在永久代

目前jvm常用回收算法就是分代回收，年轻代以复制算法为主，老年代以标记整理算法为主。原因是年轻代对象比较多，每次垃圾回收都有很多的垃圾对象回收，而且要尽可能快的减少生命周期短的对象，存活的对象较少，这时候复制算法比较适合，只要将有标记的对象复制到另一个内存区域，其余全部清除，并且复制的数量较少，效率较高；而老年代是年轻代筛选出来的对象，被标记比较高，需要删除的对象比较少，显然采用标记整理效率较高。

**Minor GC和Full GC**  
新生代 GC（Minor GC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。  
老年代 GC（Major GC  / Full GC）：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程） 。MajorGC 的速度一般会比 Minor GC 慢 10倍以上。

**垃圾回收时的停顿现象**

垃圾回收的任务是识别和回收垃圾对象进行内存清理，为了让垃圾回收器可以更高效的执行，大部分情况下，会要求系统进如一个停顿的状态。停顿的目的是为了终止所有的应用线程，只有这样的系统才不会有新垃圾的产生。同时停顿保证了系统状态在某一个瞬间的一致性，也有利于更好的标记垃圾对象。因此在垃圾回收时，都会产生应用程序的停顿。

### 垃圾收集器

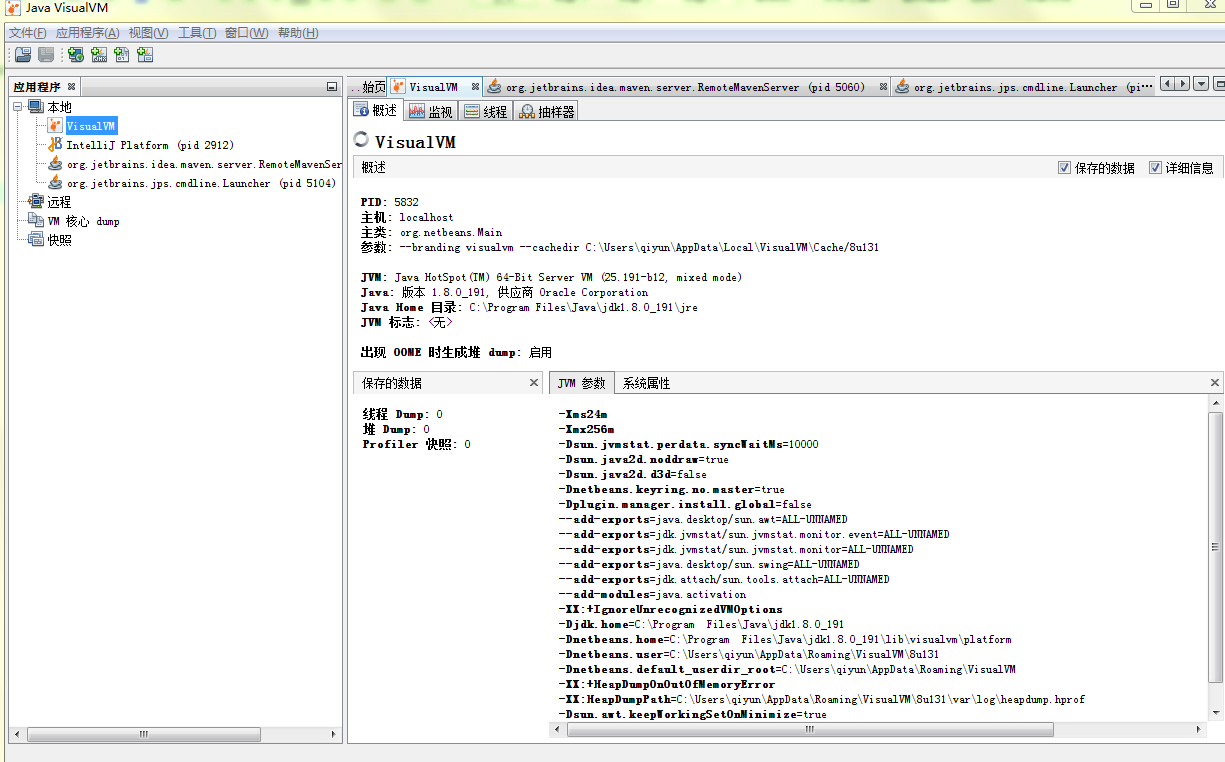
垃圾收集器是垃圾回收算法的具体实现，Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别。Sun HotSpot 虚拟机 1.6 版包含了如下收集器：Serial、ParNew、Parallel Scavenge、CMS、Serial Old、Parallel Old。这些收集器以不同的组合形式配合工作来完成不同分代区的垃圾收集工作。不同的垃圾收集器特点不同主要体现在串行（单线程）、并行（多线程）、系统停顿时间、回收效率、吞吐量等方面，没有最好只有更适合，垃圾回收调优就是在这几方面设置符合具体体统的值。

## jvm调优

### JVM调优工具

Jconsole : jdk自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪

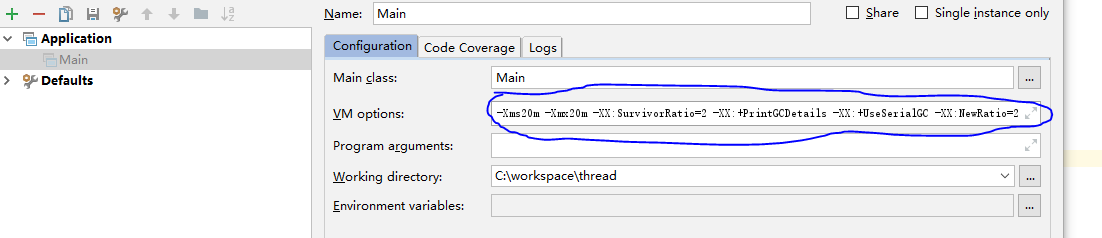
VisualVM：Netbeans的profile子项目，已在JDK中自带，能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈。在JDK\_HOME/bin(默认是C:\Program Files\Java\jdk1.6.0\_13\bin)目录下面，有一个jvisualvm.exe文件，双击打开



### 设置参数方式

根据调优工具查看系统内存情况，确定最佳参数最后设置到jvm中。

**Java方式**



**web应用方式**

修改tomcat文件catalina.sh 修改JVM堆内存大小

JAVA\_OPTS="-server -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:MaxNewSize=512m"

### JVM参数调优总结

在虚拟机运行的过程中，如果可以跟踪系统的运行状态，那么对于问题的故障排查会有一定的帮助，为此，在虚拟机提供了一些跟踪系统状态的参数，使用给定的参数执行Java虚拟机，就可以在系统运行时打印相关日志，用于分析实际问题。进行虚拟机参数配置，其实就是围绕内存管理和垃圾回收方向调优，主要对以下参数根据实际情况配置合理的值。

|  |  |
| --- | --- |
| -XX:+PrintGC | 每次触发GC的时候打印相关日志 |
| -XX:+UseSerialGC | 串行回收 |
| -XX:+PrintGCDetails | 更详细的GC日志 |
| -Xms | 堆初始值 |
| -Xmx | 堆最大可用值 |
| -Xmn | 新生代堆最大可用值 |
| -XX:SurvivorRatio | 用来设置新生代中eden空间和from/to空间的比例 |

在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到以下目标

* GC的时间足够的小
* GC的次数足够的少
* 发生Full GC的周期足够的长

前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

堆设置经验

1.针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值  
2.年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

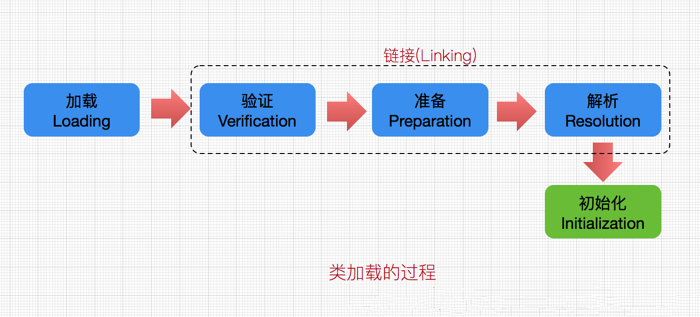
3.年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC，更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短，大的年老代会减少Full GC的频率。如何选择应该依赖应用程序对象生命周期的分布情况，如果应用存在大量的临时对象，应该选择更大的年轻代，如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间。

# 类加载机制

java.lang.ClassLoader：类加载器父类**public abstract class** ClassLoader

## 加载的机制的层次结构

每个编写的”.java”拓展名类文件都存储着需要执行的程序逻辑，这些”.java”文件经过Java编译器编译成拓展名为”.class”的文件，”.class”文件中保存着Java代码经转换后的虚拟机指令，当需要使用某个类时，虚拟机将会加载它的”.class”文件，并创建对应的class对象，将class文件加载到虚拟机的内存，这个过程称为类加载，这里我们需要了解一下类加载的过程，如下：Jvm执行class文件

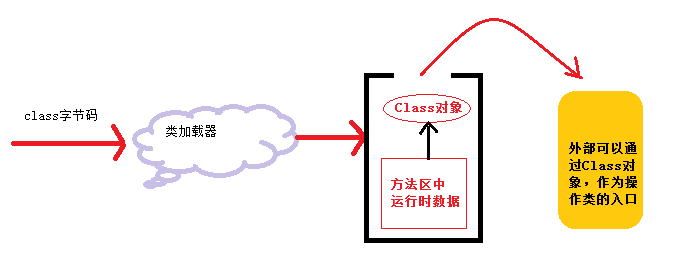


### 步骤一、类加载机制

将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，在堆中生成一个代表这个类的java.lang.Class对象（public final class Class<T> implements java.io.Serializable,GenericDeclaration,Type,AnnotatedElemen），作为方法区类数据的访问入口，这个过程需要类加载器参与。

当系统运行时，类加载器将.class文件的二进制数据从外部存储器（如光盘，硬盘）调入内存中，CPU再从内存中读取指令和数据进行运算，并将运算结果存入内存中。内存在该过程中充当着"二传手"的作用，通俗的讲，如果没有内存，类加载器从外部存储设备调入.class文件二进制数据直接给CPU处理，而由于CPU的处理速度远远大于调入数据的速度，容易造成数据的脱节，所以需要内存起缓冲作用。

类将.class文件加载至运行时的方法区后，会在堆中创建一个Java.lang.Class对象，用来封装类位于方法区内的数据结构，该Class对象是在加载类的过程中创建的，每个类都对应有一个Class类型的对象，Class类的构造方法是私有的，只有JVM能够创建。因此Class对象是反射的入口，使用该对象就可以获得目标类所关联的.class文件中具体的数据结构。



类加载的最终产物就是位于堆中的Class对象（注意不是目标类对象），该对象封装了类在方法区中的数据结构，并且向用户提供了访问方法区数据结构的接口，即Java反射的接口。

### 步骤二、连接过程

将java类的二进制代码合并到JVM的运行状态之中的过程

验证：确保加载的类信息符合JVM规范，没有安全方面的问题

准备：正式为类变量（static变量）分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配

解析：虚拟机常量池的符号引用替换为字节引用过程

### 步骤三、初始化

初始化阶段是执行类构造器<clinit>（）方法的过程。类构造器<clinit>（）方法是由编译器自动收藏类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生，代码从上往下执行。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

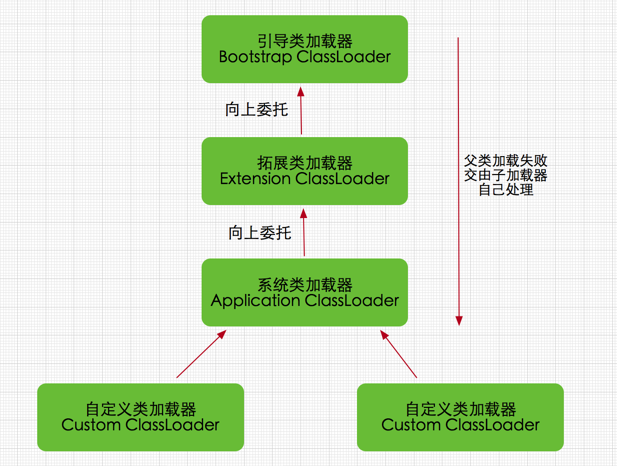
虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确加锁和同步

## 类加载器的层次结构

启动（Bootstrap）类加载器

扩展（Extension）类加载器

系统（Application）类加载器



### 启动（Bootstrap）类加载器

启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，它负责将 <JAVA\_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中，注意必由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类)。

### 扩展（Extension）类加载器

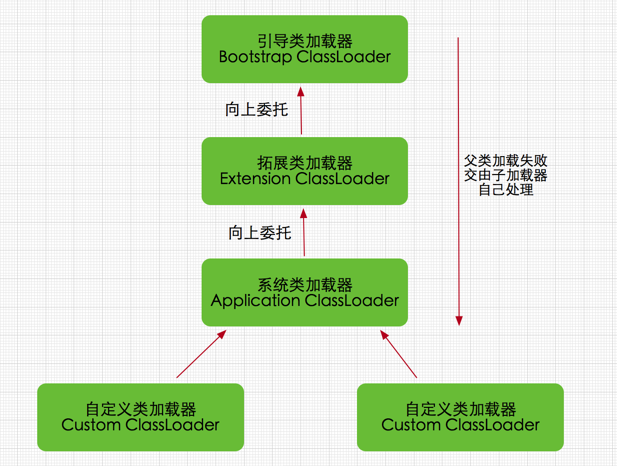
扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher$ExtClassLoader类，由Java语言实现的，是Launcher的静态内部类，它负责加载<JAVA\_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定位路径中的类库，开发者可以直接使用标准扩展类加载器。

### 系统（System）类加载器、

也称应用程序加载器是指 Sun公司实现的sun.misc.Launcher$AppClassLoader。它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认的类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。 在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它一种任务委派模式，下面我们进一步了解它。

## 理解双亲委派模式

下面我们从代码层面了解几个Java中定义的类加载器及其双亲委派模式的实现，它们类图关系如下



双亲委派模式是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完成，这不就是传说中的实力坑爹啊？那么采用这种模式有啥用呢?

### 双亲委派模式优势

采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出如下异常

java.lang.SecurityException: Prohibited package name: java.lang

### 类加载器间的关系

我们进一步了解类加载器间的关系(并非指继承关系)，主要可以分为以下4点

启动类加载器，由C++实现，没有父类。

拓展类加载器(ExtClassLoader)，由Java语言实现，父类加载器为null

系统类加载器(AppClassLoader)，由Java语言实现，父类加载器为ExtClassLoader

自定义类加载器，父类加载器肯定为AppClassLoader。

|  |
| --- |
| **public class** Main {  **public static void** main(String[] args) **throws** InterruptedException {  *//application class loader* System.***out***.println(ClassLoader.*getSystemClassLoader*());  *//extensions class loader* System.***out***.println(ClassLoader.*getSystemClassLoader*().getParent());  *//bootstrap class loader* System.***out***.println(ClassLoader.*getSystemClassLoader*().getParent().getParent());  } } |
| 输出结果  sun.misc.Launcher$AppClassLoader@18b4aac2  sun.misc.Launcher$ExtClassLoader@4554617c  null |
| 结论：  ClassLoader类是由AppClassLoader加载的。他的父亲是ExtClassLoader，ExtClassLoader的父亲无法获取是因为它是用C++实现的 |

## 类加载器常用方法

### loadClass(String)

该方法加载指定名称（包括包名）的二进制类型，该方法在JDK1.2之后不再建议用户重写但用户可以直接调用该方法，loadClass()方法是ClassLoader类自己实现的，该方法中的逻辑就是双亲委派模式的实现，其源码如下，loadClass(String name, boolean resolve)是一个重载方法，resolve参数代表是否生成class对象的同时进行解析相关操作。

正如loadClass方法所展示的，当类加载请求到来时，先从缓存中查找该类对象，如果存在直接返回，如果不存在则交给该类加载去的父加载器去加载，倘若没有父加载则交给顶级启动类加载器去加载，最后倘若仍没有找到，则使用findClass()方法去加载（关于findClass()稍后会进一步介绍）。从loadClass实现也可以知道如果不想重新定义加载类的规则，也没有复杂的逻辑，只想在运行时加载自己指定的类，那么我们可以直接使用this.getClass().getClassLoder.loadClass("className")，这样就可以直接调用ClassLoader的loadClass方法获取到class对象。

### findClass(String)

在JDK1.2之前，在自定义类加载时，总会去继承ClassLoader类并重写loadClass方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadClass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中，从前面的分析可知，findClass()方法是在loadClass()方法中被调用的，当loadClass()方法中父加载器加载失败后，则会调用自己的findClass()方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委托模式。需要注意的是ClassLoader类中并没有实现findClass()方法的具体代码逻辑，取而代之的是抛出ClassNotFoundException异常，同时应该知道的是findClass方法通常是和defineClass方法一起使用的(稍后会分析)

### defineClass(byte[] b, int off, int len)

defineClass()方法是用来将byte字节流解析成JVM能够识别的Class对象(ClassLoader中已实现该方法逻辑)，通过这个方法不仅能够通过class文件实例化class对象，也可以通过其他方式实例化class对象，如通过网络接收一个类的字节码，然后转换为byte字节流创建对应的Class对象，defineClass()方法通常与findClass()方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖ClassLoader的findClass()方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用defineClass()方法生成类的Class对象

### resolveClass(Class≺?≻ c)

使用该方法可以使用类的Class对象创建完成也同时被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用。

## 自定义类加载器

1、如果不想打破双亲委派模型，那么只需要重写findClass方法即可

2、如果想打破双亲委派模型，那么就重写整个loadClass方法

### 示例

(遵循双亲委派模型)

|  |
| --- |
| //准备要加载的类,编译后将Person.class文件放在D盘  **package** com.learn; **public class** Person {  **private** String **name**;  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  } } |
| **//自定义的加载器**  **class** MyClassLoader **extends** ClassLoader {  @Override  **protected** Class<?> findClass(String name) **throws** ClassNotFoundException {  File file = getClassFile(name);  **try** {  **byte**[] bytes = getClassBytes(file);  Class<?> c = **this**.defineClass(name, bytes, 0, bytes.**length**);  **return** c;  } **catch** (Exception e) {  e.printStackTrace();  }  **return super**.findClass(name);  }  **private** File getClassFile(String name) {  File file = **new** File(**"D:/Person.class"**);  **return** file;  }  **private byte**[] getClassBytes(File file) **throws** Exception {  *// 这里要读入.class的字节，因此要使用字节流* FileInputStream fis = **new** FileInputStream(file);  FileChannel fc = fis.getChannel();  ByteArrayOutputStream baos = **new** ByteArrayOutputStream();  WritableByteChannel wbc = Channels.*newChannel*(baos);  ByteBuffer by = ByteBuffer.*allocate*(1024);  **while** (**true**) {  **int** i = fc.read(by);  **if** (i == 0 || i == -1)  **break**;  by.flip();  wbc.write(by);  by.clear();  }  fis.close();  **return** baos.toByteArray();  } } |
| //测试  **public class** Main {  **public static void** main(String[] args) **throws** Exception {  MyClassLoader mcl = **new** MyClassLoader();  Class<?> c1 = Class.*forName*(**"com.learn.Person"**, **true**, mcl);  Object obj = c1.newInstance();  System.***out***.println(obj);  System.***out***.println(obj.getClass().getClassLoader());  } } |
| 结果：  com.learn.Person@677327b6  MyClassLoader@74a14482 |

注意点：

第二行的打印出来的是"sun.misc.Launcher$AppClassLoader"。造成这个问题的关键在于idea是自动编译的，Person.java这个类在ctrl+S保存之后或者在Person.java文件不编辑若干秒后，idea会帮我们用户自动编译Person.java，并生成到CLASSPATH也就是bin目录下。在CLASSPATH下有Person.class，那么自然是由Application ClassLoader来加载这个.class文件了。解决这个问题有两个办法：

1、删除CLASSPATH下的Person.class，CLASSPATH下没有Person.class，Application ClassLoader就把这个.class文件交给下一级用户自定义ClassLoader去加载了

2、MyClassLoader mcl = **new** MyClassLoader()写成"MyClassLoader mcl = new MyClassLoader(ClassLoader.getSystemClassLoader().getParent());"， 即把自定义ClassLoader的父加载器设置为Extension ClassLoader，这样父加载器加载不到Person.class，就交由子加载器MyClassLoader来加载了

### ****.class和getClass()****

它们二者都可以获取一个唯一的java.lang.Class对象。

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {  Person person=**new** Person();  **if**(person.getClass()==Person.**class**){  System.***out***.println(**"equal"**);  }**else** {  System.***out***.println(**"unequal"**);  }  }  结果：  equal。 |

1、.class用于类名，getClass()是Object类的一个final native的方法，对象实例才有这个方法，因此用于类实例

2、.class在编译期间就确定了一个类的java.lang.Class对象，但是getClass()方法在运行期间确定一个类实例的java.lang.Class对象

## 热部署

对于Java应用程序来说，热部署就是在运行时更新Java类文件。

### 热部署的原理

想要知道热部署的原理，必须要了解java类的加载过程。一个java类文件到虚拟机里的对象，要经过如下过程。

首先通过java编译器，将java文件编译成class字节码，类加载器读取class字节码，再将类转化为实例，对实例newInstance就可以生成对象。类加载器ClassLoader功能，也就是将class字节码转换到类的实例。在java应用中，所有的实例都是由类加载器，加载而来。一般在系统中，类的加载都是由系统自带的类加载器完成，而且对于同一个全限定名的java类（如com.csiar.soc.HelloWorld），只能被加载一次，而且无法被卸载。这个时候问题就来了，如果我们希望将java类卸载，并且替换更新版本的java类，该怎么做呢？既然在类加载器中，java类只能被加载一次，并且无法卸载。那是不是可以直接把类加载器给换了？答案是可以的，我们可以自定义类加载器，并重写ClassLoader的findClass方法。想要实现热部署可以分以下三个步骤：

1、销毁该自定义ClassLoader

2、更新class类文件

3、创建新的ClassLoader去加载更新后的class类文件。

### 热部署与热加载

Java热部署与Java热加载的联系和区别

联系

1.不重启服务器编译/部署项目

2.基于Java的类加载器实现

区别

部署方式

热部署在服务器运行时重新部署项目,热加载在运行时重新加载class

 实现原理

热部署直接重新加载整个应用,热加载在运行时重新加载class

 使用场景

热部署更多的是在生产环境使用,热加载则更多的实在开发环境使用

### 代码示例

User没有被修改类

|  |
| --- |
| **public** **class** User {  **public** **void** add() {  System.***out***.println("addV1,没有修改过...");  }  } |

User更新类

|  |
| --- |
| **public** **class** User {  **public** **void** add() {  System.***out***.println("我把之前的user add方法修改啦!");  }  } |

自定义类加载器

|  |
| --- |
| **public** **class** MyClassLoader **extends** ClassLoader {  @Override  **protected** Class<?> findClass(String name) **throws** ClassNotFoundException {  **try** {  // 文件名称  String fileName = name.substring(name.lastIndexOf(".") + 1) + ".class";  // 获取文件输入流  InputStream is = **this**.getClass().getResourceAsStream(fileName);  // 读取字节  **byte**[] b = **new** **byte**[is.available()];  is.read(b);  // 将byte字节流解析成jvm能够识别的Class对象  **return** defineClass(name, b, 0, b.length);  } **catch** (Exception e) {  **throw** **new** ClassNotFoundException();  }  }  } |

更新代码

|  |
| --- |
| **public** **class** Hotswap {  **public** **static** **void** main(String[] args)  **throws** ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException,  SecurityException, IllegalArgumentException, InvocationTargetException, InterruptedException {  *loadUser*();  System.*gc*();  Thread.*sleep*(1000);// 等待资源回收  // 需要被热部署的class文件  File file1 = **new** File("F:\\test\\User.class");  // 之前编译好的class文件  File file2 = **new** File(  "F:\\itmayiedujiangke2018-02-24\\itmayiedu\_itmayiedu\_day\_17\\target\\classes\\com\\itmayiedu\\User.class");  **boolean** isDelete = file2.delete();// 删除旧版本的class文件  **if** (!isDelete) {  System.***out***.println("热部署失败.");  **return**;  }  file1.renameTo(file2);  System.***out***.println("update success!");  *loadUser*();  }  **public** **static** **void** loadUser() **throws** ClassNotFoundException, InstantiationException, IllegalAccessException,  NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException {  MyClassLoader myLoader = **new** MyClassLoader();  Class<?> class1 = myLoader.findClass("com.itmayiedu.User");  Object obj1 = class1.newInstance();  Method method = class1.getMethod("add");  method.invoke(obj1);  System.***out***.println(obj1.getClass());  System.***out***.println(obj1.getClass().getClassLoader());  }  } |

# 字节码技术

## 字节码技术应用场景

AOP技术、Lombok去除重复代码插件、动态修改class文件等

## 字节技术优势

ava字节码增强指的是在Java字节码生成之后，对其进行修改，增强其功能，这种方式相当于对应用程序的二进制文件进行修改。Java字节码增强主要是为了减少冗余代码，提高性能等。

实现字节码增强的主要步骤为：

1、修改字节码

在内存中获取到原来的字节码，然后通过一些工具（如 ASM，Javaasist）来修改它的byte[]数组，得到一个新的byte数组。

2、使修改后的字节码生效

有两种方法：

1）自定义ClassLoader来加载修改后的字节码；

2）替换掉原来的字节码：在JVM加载用户的Class时，拦截，返回修改后的字节码；或者在运行时，使用Instrumentation.redefineClasses方法来替换掉原来的字节码

## 常见的字节码操作类库

### BCEL

Byte Code Engineering Library(BCEL)，这是Apache Software Foundation的Jakarta项目的一部分。BCEL是Java classworking 广泛使用的一种框架，它可以让您深入jvm汇编语言进行类库操作的细节。BCEL与javassist有不同的处理字节码方法，BCEL在实际的jvm指令层次上进行操作(BCEL拥有丰富的jvm指令集支持) 而javassist所强调的是源代码级别的工作。

### ASM

是一个轻量级Java字节码操作框架，直接涉及到JVM底层的操作和指令

高性能，高质量

### CGLB

生成类库，基于ASM实现

### javassist

是一个开源的分析，编辑和创建Java字节码的类库。性能较ASM差，跟cglib差不多，但是使用简单。很多开源框架都在使用它。

#### Javassist优势

– 比反射开销小，性能高。

–javassist性能高于反射，低于ASM

运行时操作字节码可以让我们实现如下功能：

– 动态生成 新的类

– 动态改变某个类的结构 ( 添加 / 删除 / 修改 新的属性 / 方法 )

javassist 的最外层的 API 和 JAVA 的反射包中的 API 颇为 类似 。

它 主要 由 CtClass ， CtMethod, ，以及 CtField 几个类组成。用以执行和 JDK 反射 API 中 java.lang.Class， java.lang.reflect.Method， java.lang.reflect.Method .Field 相同的 操作 。

方法操作

– 修改已有方法的方法体（插入代码到已有方法体）

– 新增方法 删除方法

#### javassist的局限性

JDK5.0 新语法不支持 ( 包括泛型、枚举 ) ，不支持注解修改，但可以通过底层的 javassist 类来解决，具体参考： javassist.bytecode.annotation  
不支持数组的初始化，如 String[]{"1","2"} ，除非只有数组的容量为 1  
不支持内部类和匿名类  
不支持 continue 和 break表达式。  
对于继承关系，有些不支持。例如  
class A {}    
class B extends A {}   
class C extends B {}

#### 使用Javassist创建类

|  |
| --- |
| **public** **static** **void** main(String[] args)  **throws** ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException,  SecurityException, IllegalArgumentException, InvocationTargetException {  Class<?> clazz = Class.*forName*("com.itmayiedu.Test0005");  Object newInstance = clazz.newInstance();  Method method = clazz.getDeclaredMethod("sum", **int**.**class**, **int**.**class**);  Object invoke = method.invoke(newInstance, 1, 1);  }  **public** **void** sum(**int** a, **int** b) {  System.***out***.println("sum:" + a + b);  } |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** CannotCompileException, NotFoundException, IOException {  ClassPool pool = ClassPool.*getDefault*();  // 创建class文件  CtClass userClass = pool.makeClass("com.itmayiedu.entity.User");  // 创建id属性  CtField idField = CtField.*make*("private Integer id;", userClass);  // 创建name属性  CtField nameField = CtField.*make*("private Integer name;", userClass);  // 添加属性  userClass.addField(idField);  // 添加属性  userClass.addField(nameField);  // 创建方法  CtMethod getIdMethod = CtMethod.*make*("public Integer getId() {return id;}", userClass);  // 创建方法  CtMethod setIdMethod = CtMethod.*make*("public void setId(Integer id) { this.id = id; }", userClass);  // 添加方法  userClass.addMethod(getIdMethod);  // 添加方法  userClass.addMethod(setIdMethod);  // 添加构造器  CtConstructor ctConstructor = **new** CtConstructor(**new** CtClass[] { CtClass.*intType*, pool.get("java.lang.String") },  userClass);  // 创建Body  ctConstructor.setBody(" {this.id = id;this.name = name;}");  userClass.addConstructor(ctConstructor);  userClass.writeFile("F:/test");// 将构造好的类写入到F:\test 目录下  } |

#### 使用Javassist修改类文件信息

|  |
| --- |
| **public** **static** **void** main(String[] args)  **throws** NotFoundException, CannotCompileException, InstantiationException, IllegalAccessException,  NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException, IOException {  ClassPool pool = ClassPool.*getDefault*();  // 需要加载类信息  CtClass userClass = pool.get("com.itmayiedu.User");  // 需要添加的方法  CtMethod m = **new** CtMethod(CtClass.*intType*, "add", **new** CtClass[] { CtClass.*intType*, CtClass.*intType* },  userClass);  // 方法权限  m.setModifiers(Modifier.***PUBLIC***);  // 方法体内容  m.setBody("{System.out.println(\"Test003\"); return $1+$2;}");  userClass.addMethod(m);  userClass.writeFile("F:/test");// 将构造好的类写入到F:\test 目录下  // 使用反射技术执行方法  Class clazz = userClass.toClass();  Object obj = clazz.newInstance(); // 通过调用User 无参构造函数  Method method = clazz.getDeclaredMethod("add", **int**.**class**, **int**.**class**);  Object result = method.invoke(obj, 200, 300);  System.***out***.println(result);  } |