目录

1. java 虚拟机的特性

(1)移植性

无论是 GC 还是 Hotspot 都可以用在任何 Java 可用的地方。比方说，JRuby 可以运行在其他平台上，Rails 应用就可以运行在 IBM 主机上的 JRuby 上，而且这台 IBM 主机运行的是 CP/CMS。实际上，由于 Java 和 OpenJDK 项目的开源，我们正在看到越来越多的平台的衍生，因此 JVM 的移植性也将越来越棒。

(2)成熟

JVM 已有超过 15 年的历史，在过去的这些年里，许多开发者为它做出了许多贡献，使得它的性能一次又一次地提升，让 JVM 变得更加稳定、快速和广泛。

(3)覆盖面

JRuby 和 JVM 上的其他语言项目已经被开发者所承认。JSR 越来越配合新的语言，JVM 已不再是 Java 一个人定制规则。JVM 正在构建成为类如 JRuby 等项目的优良平台。

2. 谈谈对 jvm 的理解

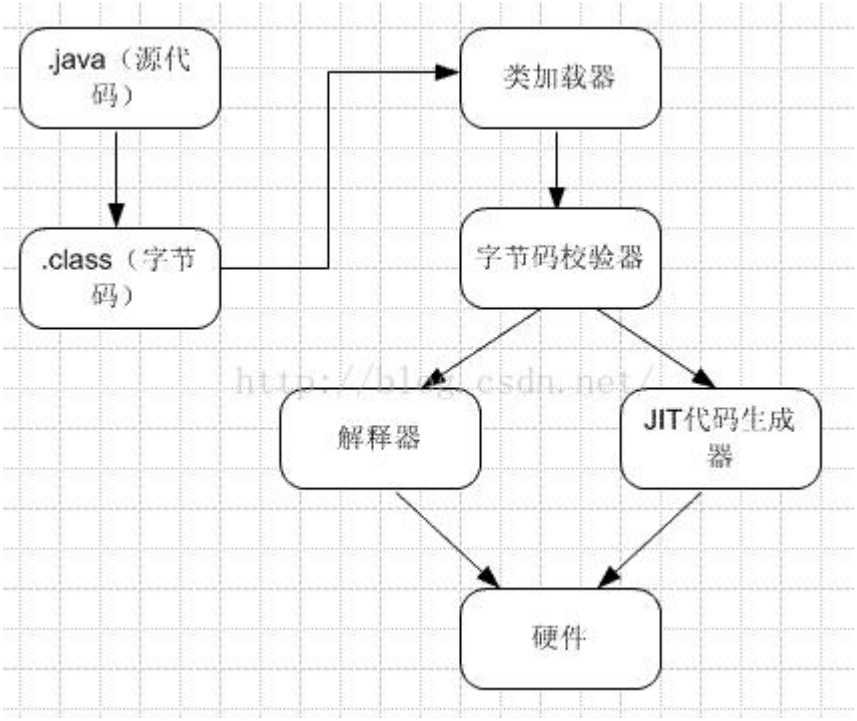
一、JVM 介绍：

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，JVM 是一

种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

Java 虚拟机包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。 JVM 屏蔽了与具体操作系统平台相关的信息，使 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。JVM 在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

JVM 是 JRE 的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。JVM 有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java 语言最重要的特点就是跨平台运行。使用 JVM 就是为了支持与操作系统无关，实现跨平台。



二、JVM 作用

与平台无关性：JVM 屏蔽了与具体操作系统平台相关的信息，使得 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

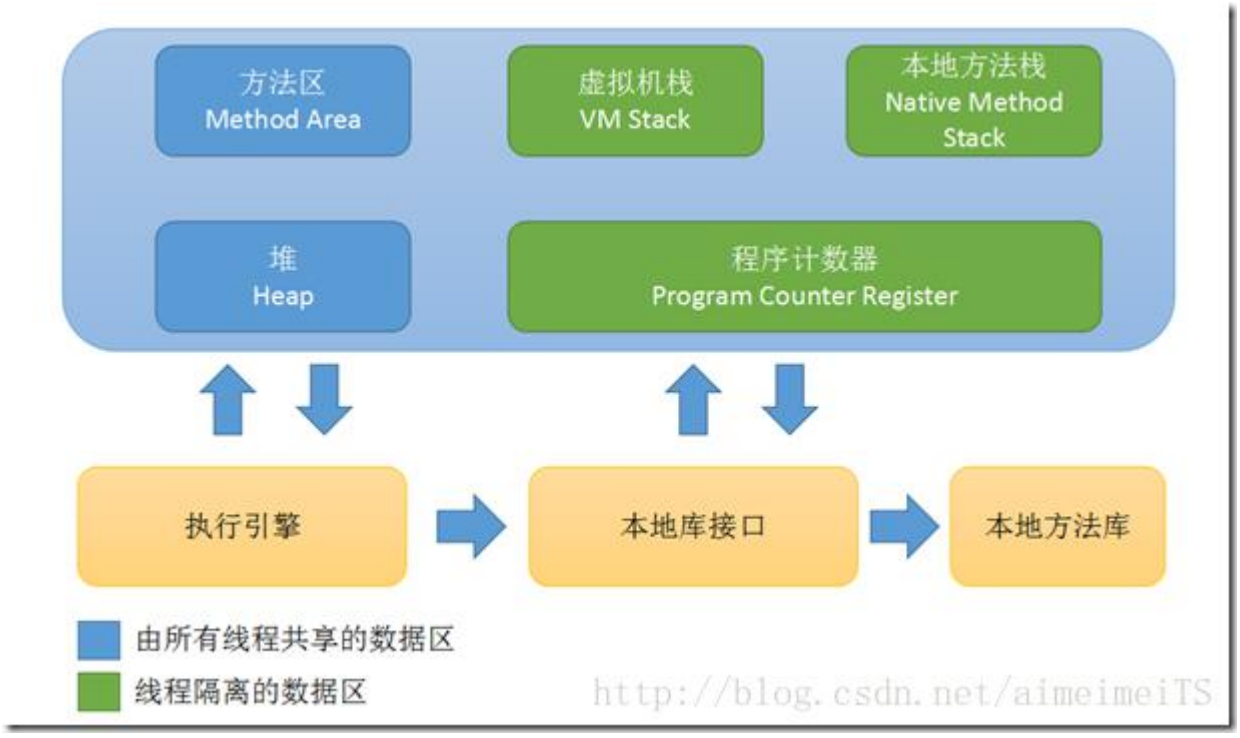
Java 中的所有类，必须被装载到 jvm 中才能运行，这个装载工作是由 jvm 中的类装载器完成的，类装载器所做的工作实质是把类文件从硬盘读取到内存中 JVM 对中央处理器（CPU）所执行的一种软件操作，用于执行编译过的 Java 程序码（Applet 与应用程序）。

JVM 就是我们常说的 java 虚拟机，它是整个 java 实现跨平台的最核心的部分，所有的 java 程序会首先被编译为.class 的类文件，这种类文件可以在虚拟机上执行。也就是说 class 并不直接与机器的操作系统相对应，而是经过虚拟机间接与操作系统交互，由虚拟机将程序解释给本地系统执行。当然只有 JVM 还不能成 class 的执行，因为在解释 class 的时候 JVM 需要调用解释所需要的类库 lib，而 jre 包含 lib 类库。

三、JVM 重要特征

1.内存管理机制

Java 虚拟机内存模型包括程序计数器、虚拟机栈、本地方法栈、方法区、堆，如图所示



Java 虚拟机运行时内存模型

（1）程序计数器

程序计数器是一块较小的内存空间，可以看作当前线程所执行的字节码行号指示器。需要注意以下几点内容：

1）程序计数器是线程私有，各线程之间互不影响

2）如果正在执行 java 方法，计数器记录的是正在执行的虚拟机字节码指令地址

3）如果执行 native 方法，这个计数器为 null

4）程序计数器也是在 Java 虚拟机规范中唯一没有规定任何 OutOfMemoryError 异常情况的区域

（2）虚拟机栈

虚拟机栈即我们平时经常说的栈内存，也是线程私有，是 Java 方法执行时的内存模型，每个方法在执行时都会创建一个栈帧用于储

存以下内容：

1）局部变量表：32 位变量槽，存放了编译期可知的各种基本数据类型、对象引用、returnAddress 类型。

2）操作数栈：基于栈的执行引擎，虚拟机把操作数栈作为它的工作区，大多数指令都要从这里弹出数据、执行运算，然后把结果压回操作数栈。

3）动态连接：每个栈帧都包含一个指向运行时常量池（方法区的一部分）中该栈帧所属方法的引用。持有这个引用是为了支持方法调用过程中的动态连接。Class 文件的常量池中有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另一部分将在每一次的运行期间转化为直接应用，这部分称为动态连接。

4）方法出口：返回方法被调用的位置，恢复上层方法的局部变量和操作数栈，如果无返回值，则把它压入调用者的操作数栈。

（3）本地方法栈

本地方法栈是线程私有，与虚拟机栈类似，为 native 方法服务。

（4）方法区

线程共享，用于储存已被虚拟机加载的类信息、常量、静态变量，即编译器编译后的代码，方法区也称**持久代**（Permanent Generation），主要存放 java 类定义信息，与垃圾回收关系不大，但不是没有垃圾回收，这个区域的内存回收目标主要是针对常量池的

回收和对类型的卸载。运行时常量池，方法区的一部分，虚拟机加载 Class 后把常量池中的数据放入运行时常量池。

（5）堆

堆是 JVM 中最大的一块区域，线程共享，此区唯一的目的就是存放对象实例，几乎所有对象实例都在这里分配

1）新生代：包括 Eden 区、From Survivor 区、To Survivor 区，系统默认大小 Eden:Survivor=8:1

2）老年代：在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。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。

2、垃圾回收机制

说起 GC，大部分人都会把这项技术当作 Java 语言的产物，其实 GC 的历史比 Java 久远。GC 中不外乎两个步骤：1.确定哪些是垃圾，2.进行垃圾回收

（1）对象已死的判定

如何确定一个对象是否“死亡”？目前有两种方式：

1）引用计数算法：给对象添加一个计数器，每当有一个地方引用它时，计数器加 1；当引用失效，计数器减 1；计数器为 0 的对象就是不可能再被使用的。目前在微软的 COM 技术、Python 语言都广泛使用该算法进行内存管理，但是至少主流的 Java 虚拟机没有选择

该算法来管理内存对象，其中最主要原因是它无法解决对象之间的相互循环引用问题。

2)可达性分析算法：基本思想就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链时，则证明此对象是不可用的。

（2）垃圾回收算法

1)标记-清除算法：首先标记出所有需要回收的对象，然后进行统一的回收，不足之处有两个：效率低、碎片多。

2)复制算法：将可用内存划分成大小相等的两块，每次只使用一块，当一块用完了，就将还存活的对象复制到另外一块上，然后把已使用的内存空间清理掉。不足之处是将内存缩小到一半，利用率不高。

3)标记-整理算法：与标记-清除类似，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉端边界以外的区域

4)分代收集算法：分代收集是目前 jvm 普遍采用的算法，即新生代采用复制算法，因为有大量新生对象死去，只有少量存活；老年代采用标记-整理，因为老年代中对象存活率高，没有额外的空间对它进行担保。

（3）垃圾回收器

如果说垃圾回收算法是内存回收的方法论，那么垃圾收集器就是内存回收的实现。Java 虚拟机规范中并没有对垃圾收集器应该如何

实现作相应规定，因此不同厂商、不同版本差异很大。在 JDK1.7 以后开始采用 G1。

垃圾回收器

G1 收集器将整个 Java 堆划分为多个大小相等的独立域（Region），虽然还保留新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region 的集合。

G1 跟踪各个 Region 中垃圾堆积的价值大小（回收所获得的空间大小及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。Region 之间的对象引用以及其他垃圾回收器中的新生代与老年代之间的对象引用，虚拟机都是使用 Remembered Set 来避免全堆扫描的，G1 中每个 Region 中都有一个与之对应的 Remembered Set，虚拟机发现程序在对 Reference 类型的数据进行读写操作时，会产生一个 Write Barrier 暂时中断写操作，检查 Reference 引用的对象是否处于不同 Region 中，如果是，便通过 CardTable 把相关引用信息记录到被引用对象所属的 Region 的 Remembered Set 中。

3 类加载机制

（1）Class 文件结构

商业和开源机构已经在 Java 语言之外发展出一大批在 Java 虚拟机上运行的语言，如 Groovy、JRuby、Scala 等。

实现语言无关性的基础仍然是虚拟机和字节码存储格式，Java 虚拟机不和任何语言绑定，它只与 Class 文件的二进制文件格式相关联，理论上讲，任一门功能性语言都可以表示为一个能被 Java 虚拟机所接受的有效的 Class 文件。

Class 文件结构包括以下内容：

1)魔数：确定这个文件能否被 Java 虚拟机接受，值为 0xCAFFBABE

（咖啡宝贝？）

2）版本号：Class 文件的版本号

3）常量池：Class 文件的资源仓库

4）访问标志：用于识别一些类或者接口层次的访问信息，是类还是接口？是否为 public？

5）索引集合：包括类索引、父类索引、接口索引

6）字段表集合：描述接口或类中声明的变量，但不包括方法内部的局部变量

7）方法表集合：代码在方法表中的属性集合“Code”属性

8）属性表集合：字段表、方法表都可以携带自己的属性表，以用于描述某些场景专用信息

（2）类加载过程

加载

加载阶段虚拟机需要完成以下 3 件事：

(1) 通过一个类的全限定名来获取此类的二进制字节流

(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

验证

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。

准备

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进行分配，注意是类变量（static 修饰），不是实例变量。

解析

虚拟机将常量池内的符号引用替换为直接引用的过程，包括类或接口的解析、字段解析、类方法解析、接口方法解析。

初始化

初始化类和其他资源

（3）类加载器

类加载器用于实现类的加载动作，对于任意一个类，都需要由加载它的类加载器和这个类本省一同确立其在 Java 虚拟机中的唯一性，每个类加载器都有一个独立的类名称空间，比较两个类是否相等，只有在这两个类是同一个类加载器加载的前提下才有意义。例如 Class 对象的 equals()、isInstance()。

从 Java 开发人员的角度看，类加载器可划分为 3 种：

1)启动类加载器：负责加载存放在<JAVA_HOME>\lib 下面的类库

2)扩展类加载器：负责加载存放在<JAVA_HOME>\lib\ext 下面的类库

3)应用程序类加载器：负责加载用户路径上的类库

双亲委派模型：

双亲委派模型的工作过程是：如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去加载，每一层次的类加载器都是如此，因此所有的加载请求最终都应传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去加载。

4 性能监控调优

jvm 启动参数

参数名称	说明
-Xms	初始堆大小，物理内存的 1/64(<1GB)
-Xmx	最大堆大小，物理内存的 1/4(<1GB)
-Xmn	年轻代大小，此处的大小是（eden+ 2 survivor space)
-XX:PermSize	设置持久代初始值，物理内存的 1/64
-XX:MaxPermSize	设置持久代最大值 物理内存的 1/4
-Xss	每个线程的堆栈大小，JDK1.5 以后为 1M
-XX:NewRatio	年轻代(包括 Eden 和两个 Survivor 区)与年老代的比值

性能优化四个命令：

1）jps：查看 java 进程

2）jstat：显示本地或者远程虚拟机垃圾回收，例如：jstat

-gcutil $pid 1000 5

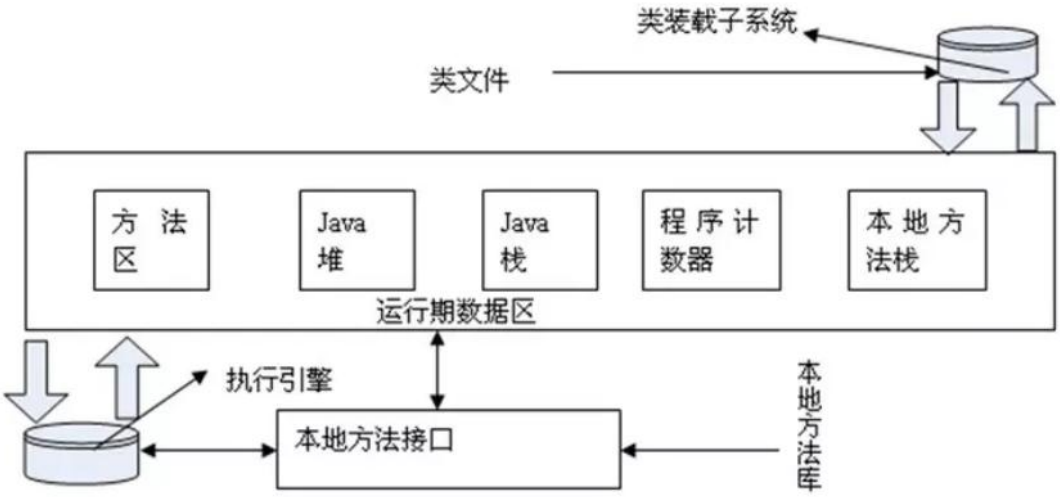
3）**jmap**：查看 JVM 堆中对象详细占用情况，例如：jmap -histo [pid]

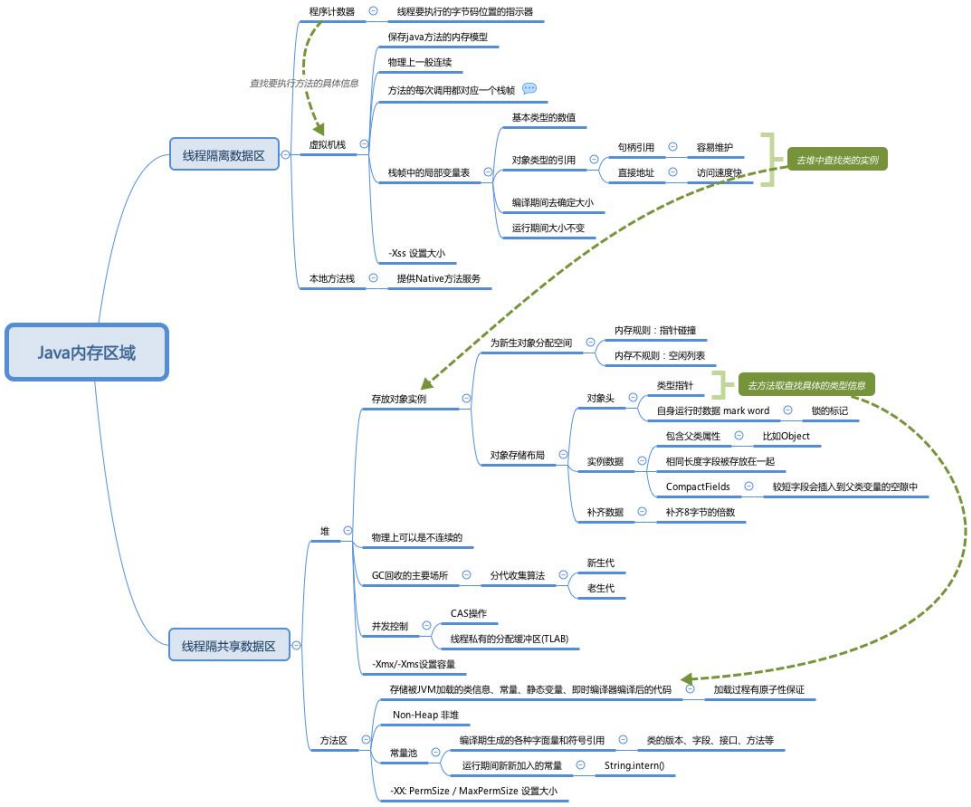
4）**jstack**：用于生成虚拟机当前线程快照，jstack -l [pid]

3. JVM 内存区域，开线程影响哪块内存

https://www.jianshu.com/p/ece1bb5fa88b

JVM 运行时数据区





概括地说来:JVM 初始运行的时候都会分配好 Method Area（**方法区**）和 Heap（**堆**），而 JVM 每遇到一个线程，就为其分配一个 Program Counter Register（**程序计数器**），VM Stack（**虚拟机栈**）和 Native Method Stack （**本地方法栈**），当线程终止时，三者（**虚拟机栈，本地方法栈和程序计数器**）所占用的内存空间也会被释放掉。

每当有线程被创建的时候，JVM 就需要为其在内存中分配虚拟机栈和本地方法栈来记录调用方法的内容，分配程序计数器记录指令执行的位置，这样的内存消耗就是创建线程的内存代价。

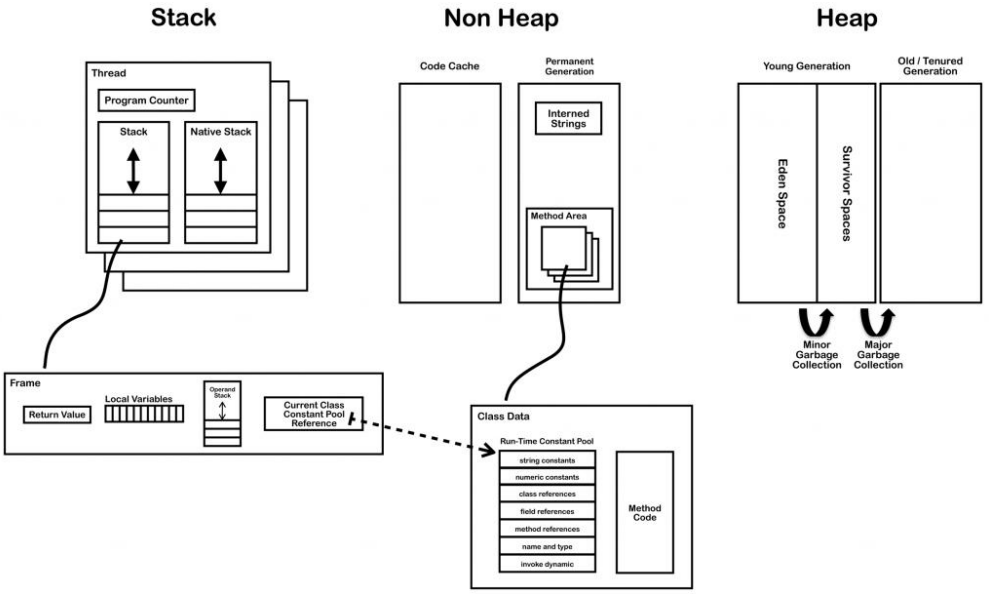
内存作为有限的资源，如果 JVM 创建了过多的线程，必然会导致资源的耗尽。因此，使用 Executor 架构复用线程可以节省内存资源，是十分必要的。

JVM 的并发是通过线程切换并分配时间片执行来实现的。 在任何一个时刻，一个处理器内核只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，JVM 需要先保存被挂起线程的上下文环境：将线程执行位置保存到程序计数器中，将调用方法的信息保存在栈中；同时将待执行线程的程序计数器和栈中的信息写入到处理器中，完成线程的上下文切换。维护线程隔离数据区中的内容在处理器中的导入导出，就是线程切换的性能代价。

对象的创建——静态区域加载的多线程安全性

为了保证多线程安全性，一些必要的操作都需要加锁来保证其原子性和可见性，但是类中静态区的代码是不需要加锁就能保证多线程安全性。这是因为什么呢？

答案在于 JVM 类加载过程的保护机制。和普通类的实例被分配在 Java 堆上不同，类的静态属性都保存在方法区，其创建收到类加载过程的影响。



类的加载过程大题分为：**加载（Loading），连接（Linking），初始化（Initialization），使用（Using）和卸载（UnLoading）**五个步骤。

这里和静态属性有关的主要是连接和初始化：

在连接步骤的准备阶段，静态属性会分配内存；在初始化步骤，JVM 会生成一个特别的方法——<clinit>方法来专门执行静态代码块和静态变量初始化。

<clinit>方法执行的过程中，JVM 会对类加锁，保证在多线程环境下，只有一个线程能成功执行<clinit>方法，其他线程都将被拥塞，并且<clinit>方法只能被执行一次，被拥塞的线程被唤醒之后也不会再去执行<clinit>方法。如果类有继承关系，JVM 还会保证父类的<clinit>方法将先于子类的<clinit>方法执行。

由此可见，静态代码块的多线程安全性是由 JVM 为其加锁实现的，这也是**延迟初始化占位类模式**的安全性基础。

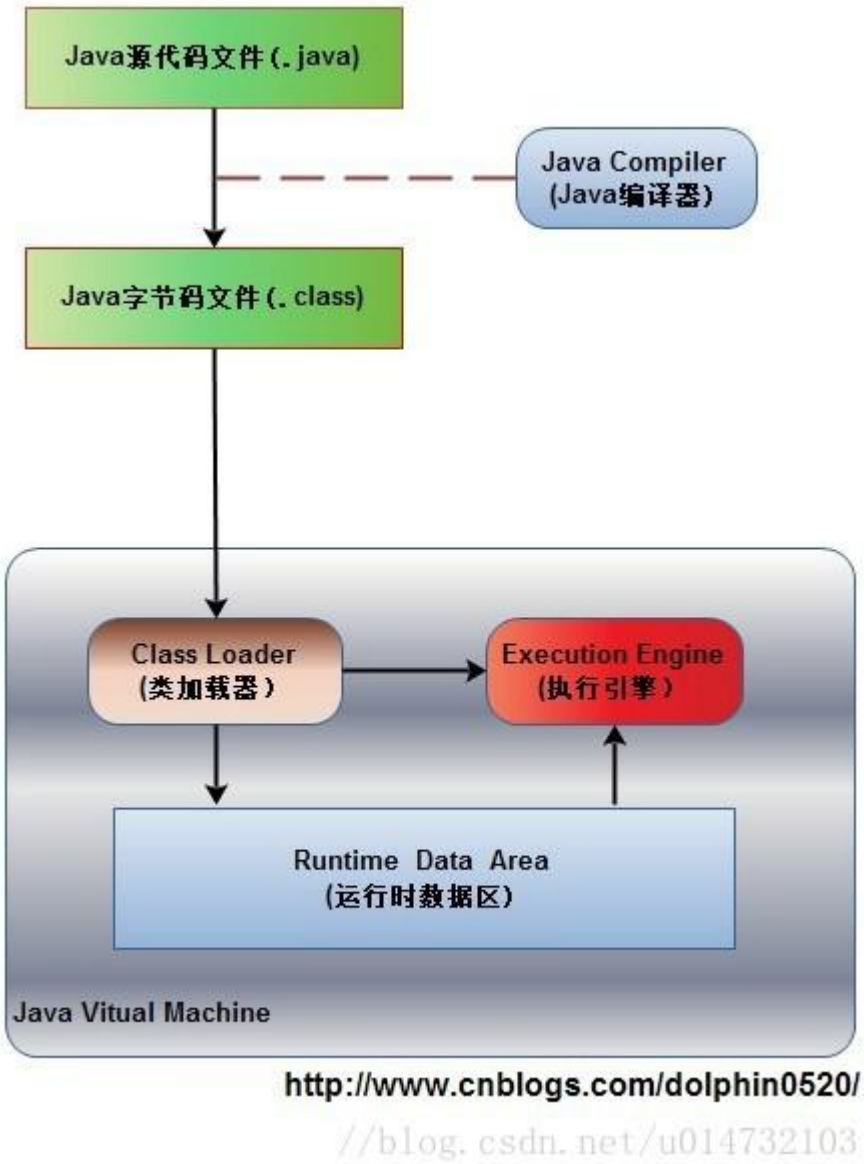
4. JVM 内存模型，内存区域

自己看总结

https://blog.csdn.net/u014732103/article/details/80241132

JVM 的内存区域划分

由于 Java 程序是交由 JVM 执行的，所以我们在谈 Java 内存区域划分的时候事实上是指 JVM 内存区域划分。在讨论 JVM 内存区域划分之前，先来看一下 Java 程序具体执行的过程：



如上图所示，

首先 Java 源代码文件(.java 后缀)会被 Java 编译器编译为字节码文件(.class 后缀)，

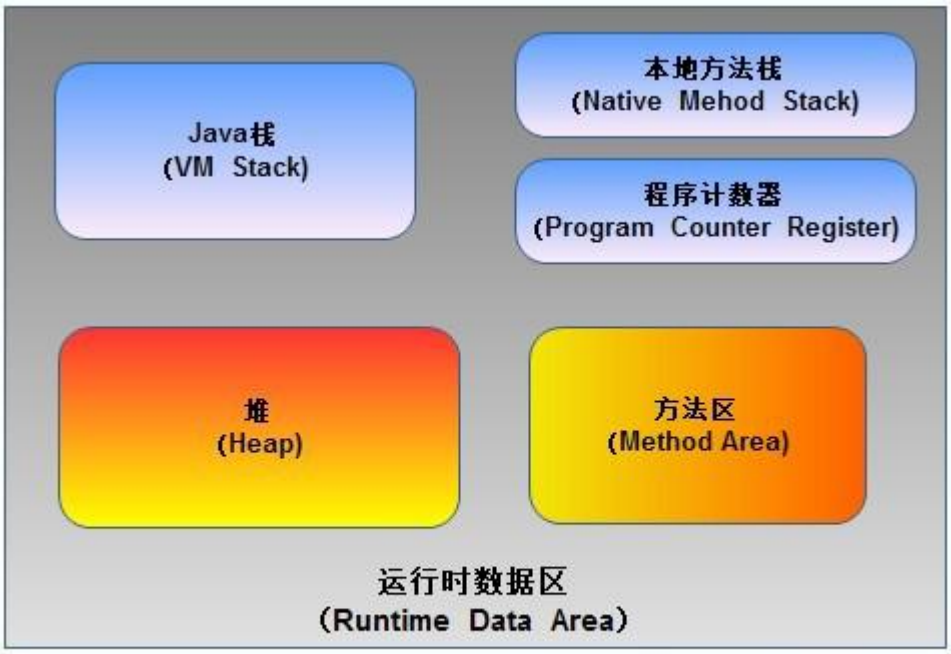
然后由 JVM 中的类加载器加载各个类的字节码文件，加载完毕之后，交由 JVM 执行引擎执行。

在整个程序执行过程中，JVM 会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一般被称作为 Runtime Data Area（运行时数据区），也就是我们常说的 JVM 内存。因此，在 Java

中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

运行时数据区包括哪几部分？

运行时数据区通常包括这几个部分：**程序计数器**(Program Counter Register)、Java **栈**(VM Stack)、**本地方法栈**(Native Method Stack)、**方法区**(Method Area)、**堆**(Heap)。



http://www.cnblogs.com/dolphin0520/

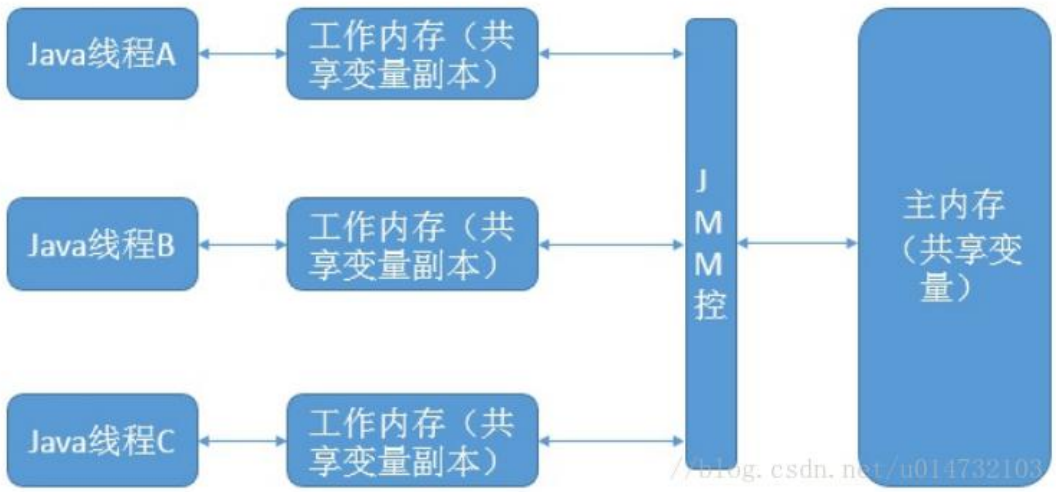
如上图所示，JVM 中的运行时数据区应该包括这些部分。在 JVM 规范中虽然规定了程序在执行期间运行时数据区应该包括这几部分，但是至于具体如何实现并没有做出规定，不同的虚拟机厂商可以有不同的实现方式。

Java 内存模型

Java 内存模型的目的：屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 java 程序在各种平台下都能达到一致的内存访问效果。

主要目标：定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量与 Java 变成中所说的变量是有所区别，它包括了实例字段，静态字段和构成数组对象的元素，但不包括局部变量和方法参数。

Java 内存模型规定了所有的变量都存储在主内存中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

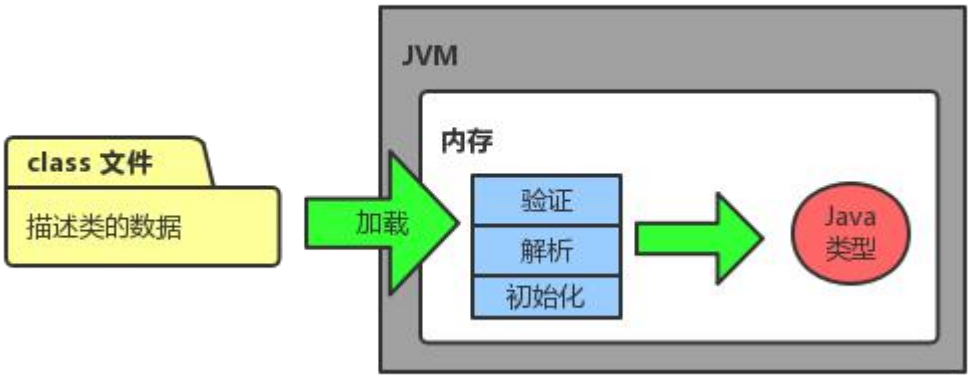


关于主内存与工作内存之间具体的交互协议，即一个变量如何从

主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了 8 种操作来完成，并且每种操作都是原子的、不可再分的。

5. 类加载机制

https://juejin.im/post/58e5e9360ce4630058492fd5



Class 文件被 JVM 加载至 JVM 内存，在内存中验证、解析、初始化之后，形成可以被 JVM 直接使用的 Java 类型。这就是类加载的简要过程。类的加载过程是在 Java 程序运行期间完成，虽然会损耗一部分性能，但是提高了 Java 语言的灵活性，体现在动态扩展方面，例如：多态（晚期绑定）。

一、类加载的时机

类的生命周期

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：

1）加载

2）验证

3）准备

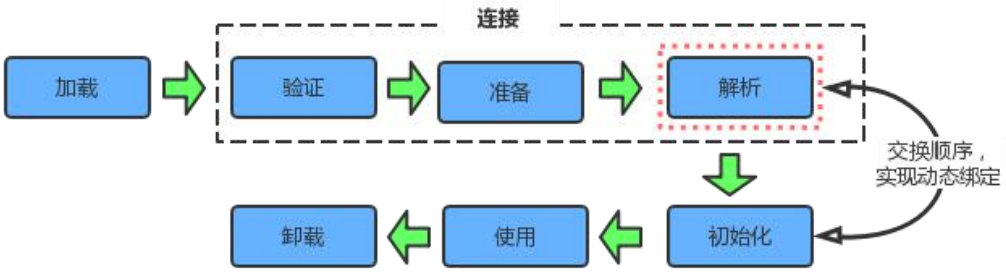
4）解析

5）初始化

6）使用

7）卸载

其中，验证、准备和解析三个部分称为连接。解析和初始化的相对顺序不是固定的，当解析在初始化之后执行时，称为动态绑定或者晚期绑定，例如：晚期绑定的多态特性。



初始化

在 JVM 规范中没有强制约束加载的时机，不过对于初始化，JVM 有严格的规范，根据《深刻理解 JVM 虚拟机》所述，有且只有 5 种情况必须对类进行初始化，但是我只能理解其中 3 种：

1)遇到 new、getstatic、putstatic 或 invokestatic 这 4 条指令

时

如果类没有进行过初始化，则需要先触发其初始化。getstatic 指读取一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）；invokestatic 指调用一个类的静态方法。

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)派生类

初始化一个派生类的时候，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

上述三种情况，被称为对一个类进行 主动引用。除此之外的引用类被称为 被动引用，不会触发初始化。

二、类加载的过程

加载

在加载阶段，虚拟机需要完成以下三件事情：

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。可以从一个 java 文件、jsp 文件获取 class 文件，即生成一个对应的 class 类。

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，作为这个方法区这个类的各种数据的访问入口

加载阶段与连接阶段的部分内容是交叉进行的，加载阶段尚未完

成，连接阶段可能已经开始。

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 文件中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。

准备

准备阶段是正式为类变量（被 static 修饰的变量）分配内存并设置类变量初始值（0 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。

方法区中分配的是静态变量的内存，并为其设置 0 值，具体的值将在初始化阶段后再赋值。成员变量（实例变量）随对象一起在 java 堆中分配内存，具体的值也是在初始化阶段后再赋值。

但是如果静态变量被 final 修饰，那么该静态变量在准备阶段就会被赋实际的值。

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。

初始化

初始化阶段是类加载过程的最后一步，在此之前的阶段，基本都是由迅疾主导和控制，在此阶段，才真正开始执行类中定义的 Java 程序代码。

在准备阶段，类变量已经赋过一次初始值（0 值），在此阶段，则根据程序员的主观计划去初始化类变量和其他资源。

有一种说法：

初始化阶段是执行类构造器 < clinit>()方法的过程

1)类构造器< clinit>() 方法 与 实例构造器< init>()方法(构造方法） 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的< clinit>() 方法执行前，父类的< clinit>()方法已经执行完毕。因此在虚拟机中，第一个被执行的< clinit>()方法的类肯定是 java.lang.Object。

2)< 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{}块）中的语句合并产生的。如果没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不生成< clinit>()方法。

3)由于父类的< clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量复制操作。

4)接口中不能使用静态语句块，但是仍然有变量初始化的赋值操作，因为接口与类一样，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。实现类与接口不是继承关系，所以不存在< clinit>()方法执行的先后顺序。

5)在多线程中，虚拟机会保证一个类的< clinit>()方法在被正确地加锁、同步，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，知道活动线程执行<clinit>()方法完毕。并且< clinit>()方法只会被执行一次，当其他线程被唤醒后，是不会再进入< clinit>()方法。

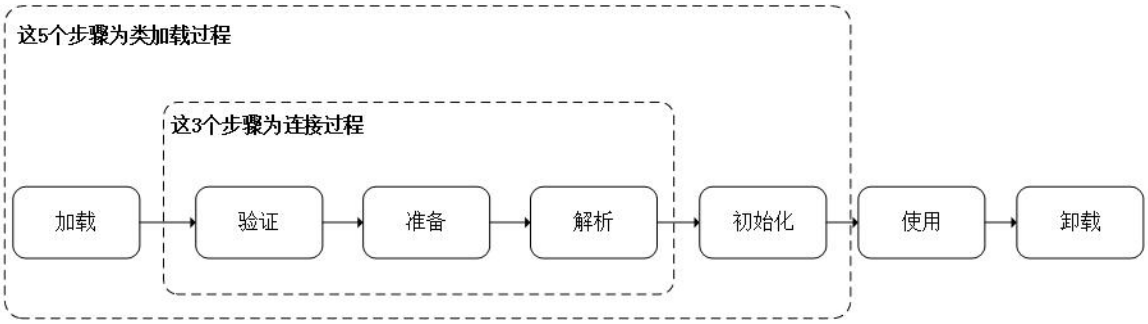
6. 谈谈对 ClassLoader(类加载器)的理解

自己看总结

https://www.cnblogs.com/leefreeman/p/7429112.html

https://blog.csdn.net/LZH984294471/article/details/51721159

一、类装载流程



1、加载

加载是类装载的第一步，首先通过 class 文件的路径读取到二进制流，并解析二进制流将里面的元数据（类型、常量等）载入到方法区，在 java 堆中生成对应的 java.lang.Class 对象。

2、连接

连接过程又分为 3 步，验证、准备、解析

2.1、验证

验证的主要目的就是判断 class 文件的合法性，比如 class 文件一定是以 0xCAFEBABE 开头的，另外对版本号也会做验证，例如如果使用 java1.8 编译后的 class 文件要再 java1.6 虚拟机上运行，因为版本问题就会验证不通过。除此之外还会对元数据、字节码进行验证，具体的验证过程就复杂的多了，可以专门查看相关资料去了解。

2.2、准备

准备过程就是分配内存，给类的一些字段设置初始值，例如：

```
public static int v=1;
```

这段代码在准备阶段 v 的值就会被初始化为 0，只有到后面类初始化阶段时才会被设置为 1。

但是对于 static final（常量），在准备阶段就会被设置成指定的值，例如：

```
public static final   int v=1;
```

这段代码在准备阶段 v 的值就是 1。

2.3、解析

解析过程就是将符号引用替换为直接引用，例如某个类继承 java.lang.object，原来的符号引用记录的是“java.lang.object”这个符号，凭借这个符号并不能找到 java.lang.object 这个对象在哪里？而直接引用就是要找到 java.lang.object 所在的内存地址，建立直接引用关系，这样就方便查询到具体对象。

3、初始化

初始化过程，主要包括执行类构造方法、static 变量赋值语句，staic{}语句块，需要注意的是如果一个子类进行初始化，那么它会事先初始化其父类，保证父类在子类之前被初始化。所以其实在 java 中初始化一个类，那么必然是先初始化 java.lang.Object，因为所有的 java 类都继承自 java.lang.Object。

说完了类加载过程，我们来介绍一下这个过程当中的主角：类加载器。

二、类加载器

类加载器 ClassLoader，它是一个抽象类，ClassLoader 的具体实例负责把 java 字节码读取到 JVM 当中，ClassLoader 还可以定制以满足不同字节码流的加载方式，比如从网络加载、从文件加载。ClassLoader 的负责整个类装载流程中的“加载”阶段。

ClassLoader 的重要方法：

1.载入并返回一个类。

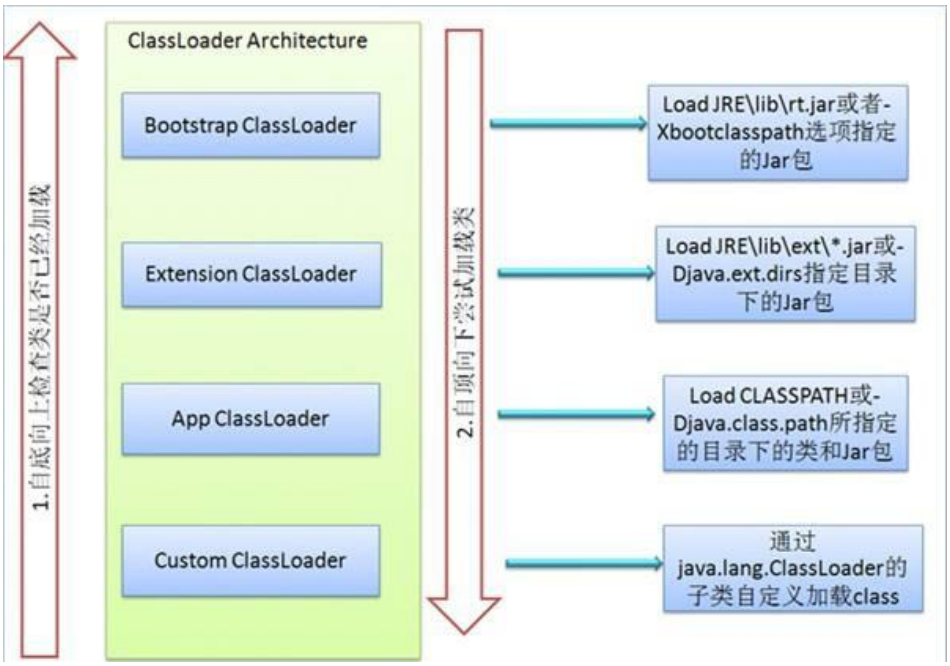
```
public Class<?> loadClass(String name) throws ClassNotFoundException
```

2.定义一个类，该方法不公开被调用。

```
protected final Class<?> defineClass(byte[] b, int off, int len)
```

3.查找类，loadClass 的回调方法

```
protected Class<?> findClass(String name) throws ClassNotFoundException
protected final Class<?> findLoadedClass(String name)
```



自下向上检查类是否被加载，一般情况下，

首先从 App ClassLoader 中调用 findLoadedClass 方法查看是否已经加载，

如果没有加载，则会交给父类，Extension ClassLoader 去查看是否加载，

还没加载，则再调用其父类，BootstrapClassLoader 查看是否已经加载，

如果仍然没有，自顶向下尝试加载类，

那么从 Bootstrap ClassLoader 到 App ClassLoader 依次尝试加载。

值得注意的是即使两个类来源于相同的 class 文件，如果使用不

同的类加载器加载，加载后的对象是完全不同的，这个不同反应在对象的 equals()、isAssignableFrom()、isInstance()等方法的返回结果，也包括了使用 instanceof 关键字对对象所属关系的判定结果。

```
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) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
```

从代码上可以看出，首先查看这个类是否被加载，如果没有则调用父类的 loadClass 方法，直到 BootstrapClassLoader（没有父类），我们把这个过程叫做双亲模式，

三、双亲模式的问题

顶层 ClassLoader，无法加载底层 ClassLoader 的类

解决

JDK 中提供了一个方法：

```
Thread. setContextClassLoader()
```

用以解决顶层 ClassLoader 无法访问底层 ClassLoader 的类的问题；

基本思想是，在顶层 ClassLoader 中，传入底层 ClassLoader 的实例。

四、双亲模式的破坏

双亲模式是默认的模式，但不是必须这么做；

Tomcat 的 WebappClassLoader 就会先加载自己的 Class，找不到再委托 parent；

OSGi 的 ClassLoader 形成网状结构，根据需要自由加载 Class。

7. 谈谈对动态加载（OSGI）的理解

自己看总结

https://www.cnblogs.com/garfieldcgf/p/6378443.html

什么是 OSGi

OSGi（Open Service Gateway Initiative，直译为“开放服务网关”）实际上是一个由 OSGi 联盟（OSGi Alliance，如图 1-1 所示）发起的以 Java 为技术平台的动态模块化规范。

OSGi 不只是一门技术，更多的是一种做系统架构的工具和方法论，如果在不适用的场景中使用 OSGi，或者在适用的场景中不恰当地使用 OSGi，都会使整个系统产生架构级的缺陷。因此，了解什么时候该用 OSGi 是与学会如何使用 OSGi 同样重要的事情。

OSGi 能让软件开发变得更容易吗

鉴于 OSGi 本身就具有较高的复杂度，而且小型系统使用 OSGi 可能导致开发成本更高。但随着系统不断发展，在代码量和开发人员都

达到一定规模之后，OSGi 带来的额外成本就不是主要的关注点了，这时候的主要矛盾是软件规模扩大与复杂度随之膨胀间的矛盾。

代码量越大、涉及人员越多的系统，软件复杂度就会越高，两者成正比关系。基于 OSGi 架构的效率优势在这时候才能体现出来：模块化推动架构师设计出能在一定范围内自治的代码，可以使开发人员只了解当前模块的知识就能高效编码，也有利于代码出现问题时隔断连锁反应。OSGi 的依赖描述和约束能力，强制开发人员必须遵循架构约束，这些让开发人员“不自由”的限制，在系统规模变大后会成为开发效率的强大推动力。

OSGi 能让系统变得更稳定吗

OSGi 会引导程序员开发出可积累可重用的软件。我们无法要求程序刚开发出来就是完全稳定的，但可以在开发过程中尽可能重用已稳定的代码来提升程序质量。

基于 OSGi 比较容易实现强鲁棒性的系统。普通汽车坏掉一个轮胎就会抛锚，但是飞机在飞行过程中即使坏了其中一个引擎，一般都还能保持正常飞行。对于软件系统来说，如果某一个模块出了问题，能够不波及其他功能的运作，这也是稳定性的一种体现。

OSGi 能让系统运行得更快吗

系统引入 OSGi 的目的可能有很多种，但一般不包括解决性能问题。如果硬要说 OSGi 对性能有什么好处，大概就是让那些有“系统洁癖”的用户可以组装出为自己定制的系统了。

首先，OSGi 是在 Java 虚拟机之上实现的，它没有要求虚拟机的

支持，完全通过 Java 代码实现模块化，在执行上不可避免地会有一些损耗。

其次，从内存用量来看，OSGi 允许不同版本的 Package 同时存在，这是个优点，但是客观上会占用更多内存。

仅从性能角度来说，OSGi 确实会让系统性能略微下降，但是这完全在可接受范围之内。使用 OSGi 开发时应该考虑到性能的影响，但不应当将其作为是否采用 OSGi 架构的主要决策依据。

8. 内存对象的循环引用及避免（暂时搜不到）