# JVM

1、ClassLoader(双亲委托)

做[**Java**](http://lib.csdn.net/base/javaee)开发，对于ClassLoader的机制是必须要熟悉的基础知识，本文针对[**Java**](http://lib.csdn.net/base/java)ClassLoader的机制做一个简要的总结。因为不同的JVM的实现不同，本文所描述的内容均只限于Hotspot Jvm.

本文将会从JDK默认的提供的ClassLoader，双亲委托模型，如何自定义ClassLoader以及Java中打破双亲委托机制的场景四个方面入手去讨论和总结一下。

**JDK默认ClassLoader**

JDK 默认提供了如下几种ClassLoader

1.  Bootstrp loader  
Bootstrp加载器是用C++语言写的，它是在Java虚拟机启动后初始化的，它主要负责加载%JAVA\_HOME%/jre/lib,-Xbootclasspath参数指定的路径以及%JAVA\_HOME%/jre/classes中的类。

1.  ExtClassLoader    
Bootstrp loader加载ExtClassLoader,并且将ExtClassLoader的父加载器设置为Bootstrploader.ExtClassLoader是用Java写的，具体来说就是 sun.misc.Launcher$ExtClassLoader，ExtClassLoader主要加载%JAVA\_HOME%/jre/lib/ext，此路径下的所有classes目录以及java.ext.dirs系统变量指定的路径中类库。

2.  AppClassLoader  
Bootstrp loader加载完ExtClassLoader后，就会加载AppClassLoader,并且将AppClassLoader的父加载器指定为 ExtClassLoader。AppClassLoader也是用Java写成的，它的实现类是sun.misc.Launcher$AppClassLoader，另外我们知道ClassLoader中有个getSystemClassLoader方法,此方法返回的正是AppclassLoader.AppClassLoader主要负责加载classpath所指定的位置的类或者是jar文档，它也是Java程序默认的类加载器。

综上所述，它们之间的关系可以通过下图形象的描述：

**双亲委托模型**

Java中ClassLoader的加载采用了双亲委托机制，采用双亲委托机制加载类的时候采用如下的几个步骤：

1.  当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2.  当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.

3.  当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

说到这里大家可能会想，Java为什么要采用这样的委托机制？理解这个问题，我们引入另外一个关于Classloader的概念**“命名空间”**， 它是指要确定某一个类，需要类的全限定名以及加载此类的ClassLoader来共同确定。也就是说即使两个类的全限定名是相同的，但是因为不同的ClassLoader加载了此类，那么在JVM中它是不同的类。明白了命名空间以后，我们再来看看委托模型。采用了委托模型以后加大了不同的 ClassLoader的交互能力，比如上面说的，我们JDK本生提供的类库，比如hashmap,linkedlist等等，这些类由bootstrp 类加载器加载了以后，无论你程序中有多少个类加载器，那么这些类其实都是可以共享的，这样就避免了不同的类加载器加载了同样名字的不同类以后造成混乱。

1. 我们知道，jdk中有一个List类，是在java.util包里的。如果我们也自己定义一个包名为java.util，里面新建一个List类

Package java.util

Public class List{

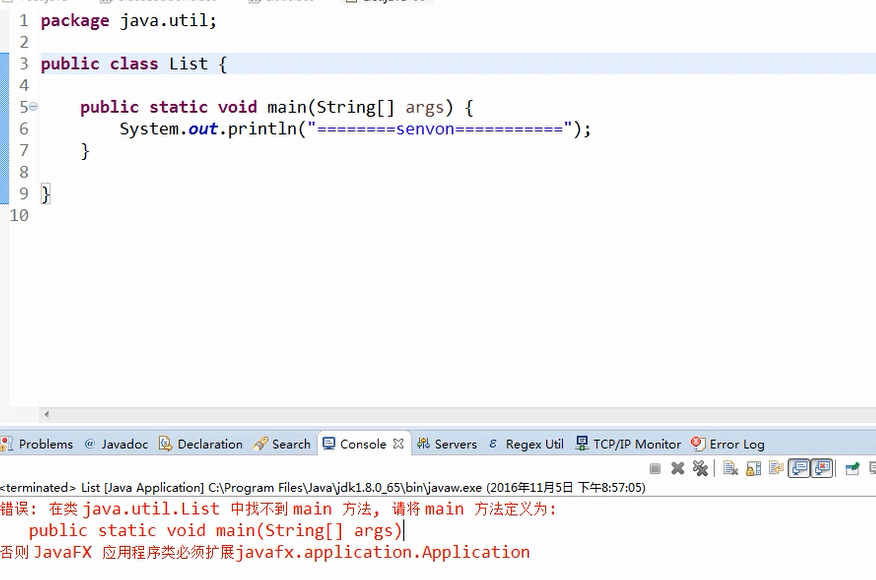
Public static void main(String[] args){

System.out.prinln(“aaa”);

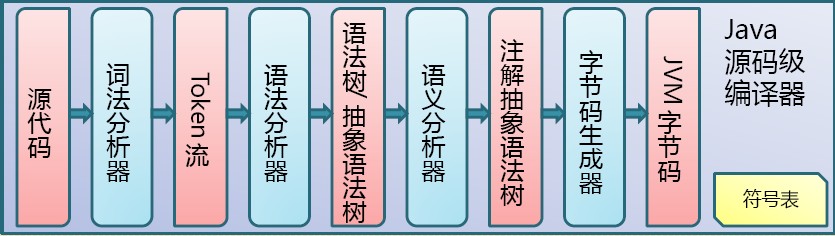
}

}

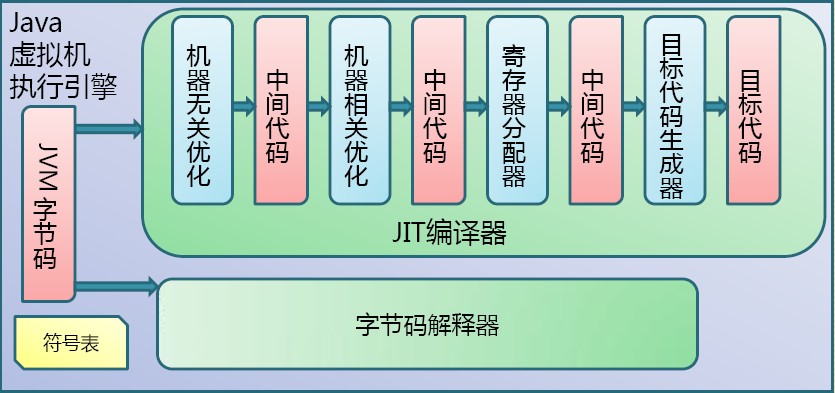
我们发现这个编译不会报错，但是具体调用的时候会报错，那是因为在虚拟机启动时，jdk下的java.util包里的List类已经被父类的加载器加载并加入到内存中去，后面不会再加载这个java.util包下的List类，因为自己定义的同包名同名字的类是不会再被加载，因为引用的时候会报错。这个模式保证了jdk提供的各种类不会因为人为的自定义而被覆盖。而我们在定义类的时候，可以存在跟jdk或者其它自定义的类的名字一样的，但是不能定义包名和类名都一样的，那么会导致运行错误。JDK认为程序安全是它的事，文件安全是系统的事



[**Java**](http://lib.csdn.net/base/javaee)代码编译是由Java源码编译器来完成，流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：



Java代码编译和执行的整个过程包含了以下三个重要的机制：

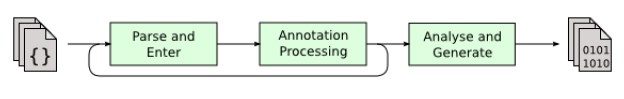
* Java源码编译机制
* 类加载机制
* 类执行机制

**Java源码编译机制**

Java 源码编译由以下三个过程组成：

* 分析和输入到符号表
* 注解处理
* 语义分析和生成class文件

流程图如下所示：

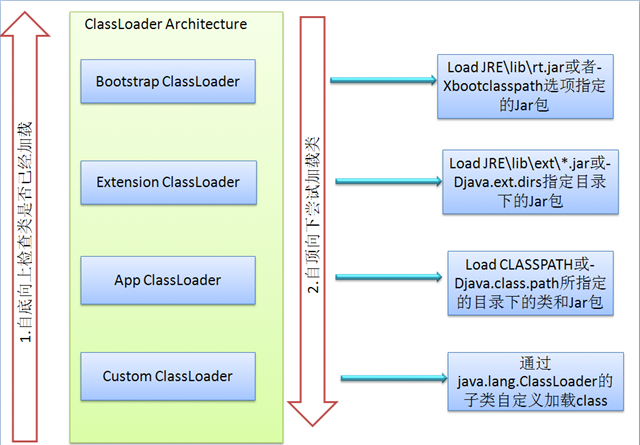


最后生成的class文件由以下部分组成：

* 结构信息。包括class文件格式版本号及各部分的数量与大小的信息
* 元数据。对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池
* 方法信息。对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息

**类加载机制**

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



1. Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

2）Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

3）App ClassLoader

负责记载classpath中指定的jar包及目录中class

4）Custom ClassLoader

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

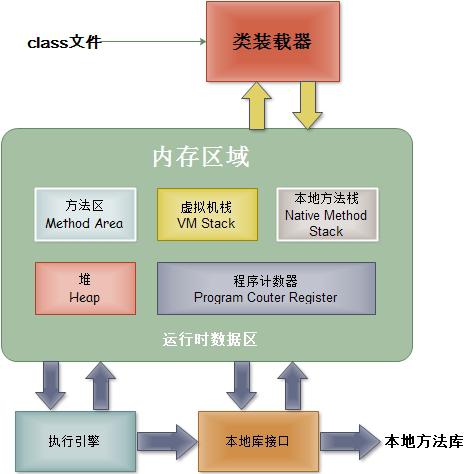
**类执行机制**

JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：



**JVM结构探究----**

**1.JVM结构示意图**



**2.JVM运行时数据区**

**1)程序计数器(Program Counter Register)**

　　程序计数器是用于存储每个线程下一步将执行的JVM指令，如该方法为native的，则程序计数器中不存储任何信息

**2)JVM栈(JVM Stack)**

　　JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈，JVM栈中存放的为当前线程中局部（局部变量）基本类型的变量（java中定义的八种基本类型：boolean、char、byte、short、int、long、float、double）、部分的返回结果以及Stack Frame，非基本类型的对象在JVM栈上仅存放一个指向堆上的地址

**3)堆(heap)**

　　它是JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收。

　　（1）堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的

　　（2）Sun Hotspot JVM为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间TLAB（Thread Local Allocation Buffer），其大小由JVM根据运行的情况计算而得，在TLAB上分配对象时不需要加锁，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配

　　（3）TLAB仅作用于新生代的Eden Space，因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效。

**4)方法区（Method Area）**

　　（1）在Sun JDK中这块区域对应的为PermanetGeneration，又称为持久代。

　　（2）方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，同时方法区域也是全局共享的，在一定的条件下它也会被GC，当方法区域需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

**5)本地方法栈（Native Method Stacks）**

　　JVM采用本地方法栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

**6)运行时常量池（Runtime Constant Pool）**

　　存放的为类中的固定的常量信息、方法和Field的引用信息等，其空间从方法区域中分配。JVM在加载类时会为每个class分配一个独立的常量池，但是运行时常量池中的字符串常量池是全局共享的。