JVM学习

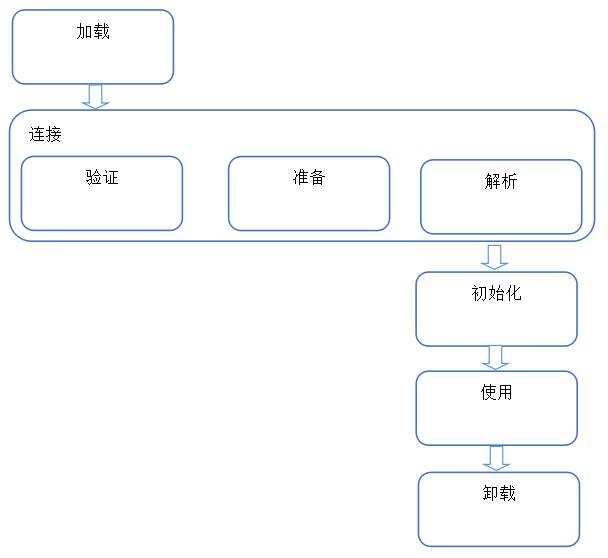
# JVM运行基本过程

Java源码==》javac编译==》java运行

## 类加载机制

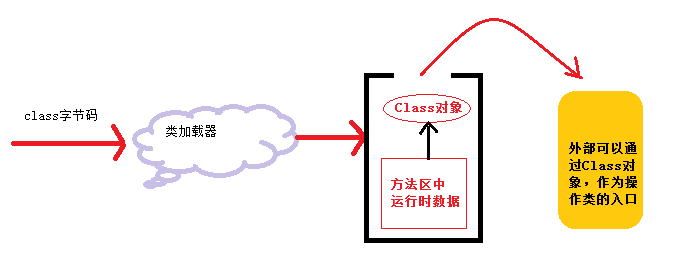
JVM将class文件加载到内存中，并对数据及逆行校验、解析、初始化，最终形成JVM可以直接使用的java类型的过程。

加载（loading）==》（验证（verification）==》准备（pregaration）==》解析（resolution））（链接）==》初始化（initializtion）==》使用（use）==》unloading（卸载）



**加载**

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，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。



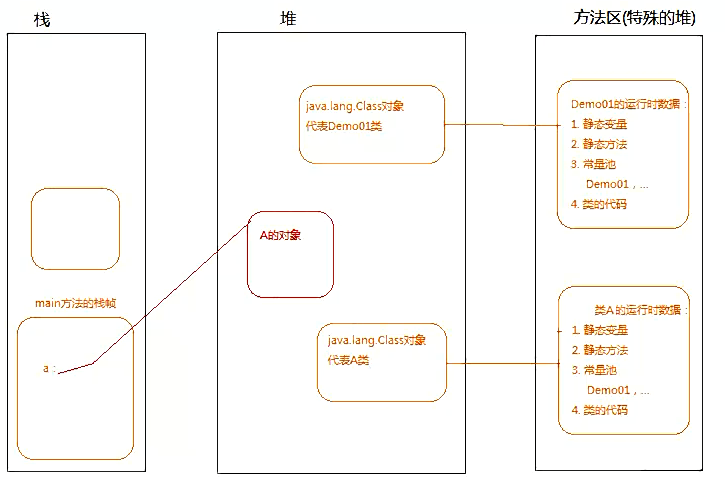
**链接**

将java类的二进制代码合并到JVM的运行状态之中的过程,包括三个阶段：

1. 验证：确保加载的类信息符合JVM规范，没有安全方面的问题
2. 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去中进行分配
3. 解析：虚拟机常量池的符号引用替换为字节引用过程

**初始化**

1. 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
2. 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
3. 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
4. 当范围一个Java类的静态域时，只有真正声名这个域的类才会被初始化



**内存中存在栈、堆（放创建好的对象）、方法区（实际也是一种特殊堆）**

* 1. JVM加载Demo01时候，首先在方法区中形成Demo01类对应静态数据（类变量、类方法、代码…），同时在堆里面也会形成java.lang.Class对象（反射对象），代表Demo01类，通过对象可以访问到类二进制结构。然后加载变量A类信息，同时也会在堆里面形成a对象，代表A类。
  2. main方法执行时会在栈里面形成main方法栈帧，一个方法对应一个栈帧。如果main方法调用了别的方法，会在栈里面挨个往里压，main方法里面有个局部变量A类型的a，一开始a值为null，通过new调用类A的构造器，栈里面生成A（）方法同时堆里面生成A对象，然后把A对象地址付给栈中的a,此时a拥有A对象地址。
  3. 当调用A.width时，调用方法区数据。

**类的主动引用（一定会发生类的初始化）**

1. new一个类的对象
2. 调用类的静态成员（除了final常量）和静态方法
3. 使用java.lang.reflect包的方法对类进行反射调用
4. 当虚拟机启动，java Demo01,则一定会初始化Demo01类，说白了就是先启动main方法所在的类
5. 当初始化一个类，如果其父类没有被初始化，则先初始化它父类

**类的被动引用（不会发生类的初始化）**

1. 当访问一个静态域时，只有真正声名这个域的类才会被初始化
2. 通过子类引用父类的静态变量，不会导致子类初始化
3. 通过数组定义类的引用，不会触发此类初始化
4. 引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

## 类加载器

作用：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。

### 类加载器层次结构：

1. 引导类加载器（Bootstrap类加载器 – JRE/lib/rt.jar）（用c实现的）

Bootstrap类加载器负责加载rt.jar中的JDK类文件，它是所有类加载器的父加载器。Bootstrap类加载器没有任何父类加载器，如果你调用String.class.getClassLoader()，会返回null，任何基于此的代码会抛出NUllPointerException异常。Bootstrap加载器被称为初始类加载器。

1. Extension类加载器（扩展类加载器） – JRE/lib/ext或者java.ext.dirs指向的目录（用java实现）java9中貌似移除了ExtClassloader类。

而Extension将加载类的请求先委托给它的父加载器，也就是Bootstrap，如果没有成功加载的话，再从jre/lib/ext目录下或者java.ext.dirs系统属性定义的目录下加载类。Extension加载器由sun.misc.Launcher$ExtClassLoader实现。

1. Application类加载器（应用程序类加载器） – CLASSPATH环境变量, 由-classpath或-cp选项定义,或者是JAR中的Manifest的classpath属性定义.（用java实现）

第三种默认的加载器就是System类加载器（又叫作Application类加载器）了。它负责从classpath环境变量中加载某些应用相关的类，classpath环境变量通常由-classpath或-cp命令行选项来定义，或者是JAR中的Manifest的classpath属性。Application类加载器是Extension类加载器的子加载器。通过sun.misc.Launcher$AppClassLoader实现。

1. 自定义类加载器（用java实现）

开发人员可以通过继承java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求。

除引导类加载器外，其他类加载器都必须继承自java.lang.ClassLoader类。

java.lang.ClassLoader（抽象类） API:

public final [ClassLoader](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassLoader.html" \o "class in java.lang) getParent()：得到父类加载器

public Class<?> loadClass(String name) throws ClassNotFoundException:加载名为name类

protected [Class](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Class.html)<?> findClass([String](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/String.html) name) throws [ClassNotFoundException](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassNotFoundException.html)：查找名为name的类。

protected final Class<?> findLoadedClass(String name)：查找已经加载的名为name的类

（@Deprecated）protected final [Class](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Class.html)<?> defineClass(byte[] b, int off, int len)throws [ClassFormatError](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/ClassFormatError.html)：把字节数组中的内容转化为java类

### 类加载器的双亲委托机制（代理模式）

交给其他的加载器来加载指定的类（优先由父类加载，依次追溯，直到最高的类加载器，如果最高父类加载器能完成加载，则直接加载，返回成功，只有父类加载器无法加载时才自己加载。这种方式即双亲委托机制，保证java核心库的类型安全，保证不会出现用户自定义的java.lang.Object类、java.lang.String类等）

并不是所有的类加载器都使用双亲委托机制，如tomcat 的类加载器。

### 自定义类加载器

1. extends java.lang.ClassLoader
2. 检查请求的类型是否已经被这个类加载器加载了，如果已经加载，直接返回
3. 委派类加载请求给父类加载器，父类加载器如果能完成，则返回父类加载器加载的Class实例。
4. 调用本类的类加载器findClass（）方法，视图获取字节码
5. 如果获取不到，调用definClass（…）方法返回异常给loadClass（…）, loadClass（…）抛出异常则终止加载过程。

注意：被两个类加载器同时加载的类，java认为是两个不同的类。

拓展：代码加密：使用类加载器进行类加密解密

## 线程上下文类加载器

不使用双亲委托机制实现类加载，使用线程类加载器实现。

每个线程都有一个关联的上下文类加载器，如果使用new Thread（）方式生成的新的线程，新线程将继承其父线程的上下文类加载器，如果程序对线程上下文类加载器没有任何改动，程序中所有的线程都将使用系统类加载器作为上下文类加载器。

可以使用如下方式获取或设置线程上下文类加载器（抛弃双亲委托机制）：

Thread.currentThread().getContextClassLoader()

Thread.currentThread().setContextClassLoader(classLoader)

## Tomcat类加载器机制

为了保证安全，tomcat不能使用系统默认的类加载器。

如果tomcat在运行web项目时使用系统的类加载器，用户可以直接操作系统中的各个目录，这是相当危险的。

对于运行在java EE容器中的web应用来说，类加载器的实现方式与一般的java应用有所不同。

每个web应用都有一个对应的类加载器实例。该类加载器也是用代理模式（非双亲委托机制），不同的是它首先尝试去加载某个类，如果找不到才代理给其父类加载器。为了保证安全，这种方式下核心库不在查询范围之内。

Tomcat可以限制用户只能把类卸载指定的地方，否则不加载类。

## OSGi(Open Service Gateway Initiative)

面向java的动态模块系统，Java动态化模块化系统的一系列规范。它为开发人员提供了面向服务和基于组件的运行环境。并提供标准的方式来管理软件的声明周期。Eclipse的插件机制就是基于OSGi技术来构建的。

原理：

每个模块（bundle）都包含类，模块可以声明它所依赖的需要导入（import）的其他模块的java包和类（通过Import-Package），也可以声明导出（export）自己的包和类，供其他模块使用（通过Export-Package）。这些功能通过OSGi特有的类加载机制来实现的。OSGi的每个模块都有一个对应的类加载器。它负责加载模块自己好汉的java包和类。需要加载java核心库的类时会代理给其父类（通过启动类加载器Bootstrap）当它需要加载锁导入的java 类时，他会代理给导出此java类的模块来完成加载。模块可以显示声明某些java包和类，必须由父类加载器来加载。

# jvm结构



方法区 - 所有的类级数据将被存储在这里，包括静态变量。每个JVM只有一个方法区域，它是一个共享资源。

堆区 – 虚拟机启动时建立。所有对象及其相应的实例变量和数组将存储在此处。每个JVM还有一个堆区。由于 方法 和堆区域 共享多个线程的内存，所存储的数据不是线程安全的。

直接内存：Java堆外的直接向系统申请的内存空间。Java NIO库允许直接使用直接内存。

垃圾回收系统- 垃圾回收器可以对方法区、java堆和直接内存进行回收。Java堆是垃圾回收的重点。Java中所有的对象空间的释放时隐式的。垃圾回收式JVM优化的重点。

Java栈 – 每个java虚拟机线程都有一个java栈，一个线程的java栈在线程创建的收被创建，Java栈中保存着帧信息，java栈中保存着局部变量、方法参数，同时和java方法的调用，返回密切相关。

本地方法栈 – 本地方法栈和java栈类似，不同的是java栈用于方法的调用，本地栈用于本地方法（native method）调用，作为java虚拟机的重要扩展，Java虚拟机允许java直接调用本地方法（通常使用c编写）

PC寄存器（program counter） - 每个线程都有独立的PC寄存器，Java虚拟机为每个Java线程创建PC寄存器。在任意时刻，一个java线程总在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC寄存器就会指向当前正在被执行的指令。如果当前的方法是本地方法，那么PC寄存器的值就是undefined。一旦执行指令就保存当前执行指令的地址，PC寄存器将用下一条指令更新。

执行引擎：执行引擎是java虚拟机的最核心的组件之一，它负责执行虚拟机的字节码，现代虚拟机为了提高执行效率，会使用即时编译（just in time）技术将方法编译成机器码后再执行。

Java hotspot client

Java hotspot server

# jvm堆结构图及分代

## 内存分代策略

Java虚拟机根据对象存活周期不同，把堆内存划分为几块，一般为新生代、老年代和永久代（对hotspot虚拟机而言）。

为什么分代

堆内存是虚拟机内存管理最大的一块，也是垃圾回收最频繁的一块区域，程序所有的对象实例都放在堆内存中。给堆内存分代是为了提高对昂内存分配和垃圾回收的效率。如果堆内存没有区域划分，所有的新创建的对象和声明周期很长的对象放在一起。随着程序的执行，堆内存需要频繁进行垃圾回收，每次回收都要便利所有对象，代价巨大。

有了内存分代，新创建的对象会在新生代中分配内存，经过多次回收仍然存活下来的对象存在在老年代中，静态属性、类信息等存放在永久代中，新生代对象存活时间短，只需要在新生代区域中频繁进行GC，老年代中对象生命周期长，内存回收的频率相对较低，不需要频繁的进行回收，永久代中回收效果太差，一般不进行垃圾回收（永久代是hotspot特有的分代，hotspot使用永久代的方法来实现方法区，hotspot JDK1.7开始去永久代）。

可以根据不同年代的特点采用合适的垃圾回收算法，分代收集大大的提升了收集效率，这些都是内存分代带来的好处。

堆的结构图



## 新生代

Eden：伊甸园，对象出生区域（大对象会直接放在老年代）,垃圾回收最频繁区域。当Eden区域没有足够空间时，虚拟机会发起一次Minor GC。GC开始时对象只会存在于Eden和from区，to区时空的（作为保留区）。GC进行时，Eden区所有存活对象都会被复制到to区域，而Form区中，仍然存活的对象会根据他们的年龄值决定去向。年龄值达到（默认15，新生代中的对象每熬过一轮垃圾回收，年龄值就加1，GC分代年龄存储在对象header中。年龄值达到阈值时会移入老年代，没有达到阈值的会被放在To区。接着From和To区会交换角色，即To就是上次清空的From区，保证To区域为空。

From（s0）和To(s1)：合称survivor

Eden :From:To 空间占比8:1:1,使用这种划分方式是为了让hotspot采用复制算法回收新生代。

## 老年代

在新声带中经历多次GC后仍然存活下来的对象会进入老年代，老年代对象生命周期较长，存活率高，在老年代中进行GC的频道相对而言较低，回收速度也较慢。

## 永久代

永久代的存储类信息、常量、静态变量、即使编译器编译后的代码等数据。对这一区域java虚拟机规范指出可以不进行垃圾手机，一般而言不会进行垃圾回收。

# 垃圾回收算法

引用技术算法

# 收集器

# 性能监测工具

# 常见参数