JVM核心之

JVM运行和类加载全过程

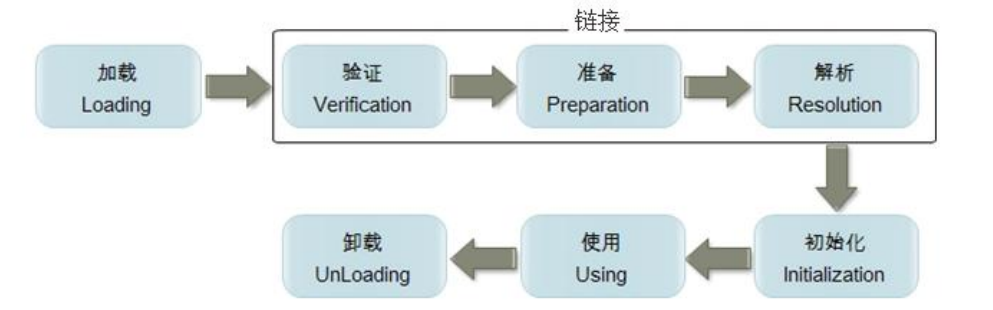
讲师：高淇　邮箱：gaoqi110@163.com

类加载全过程

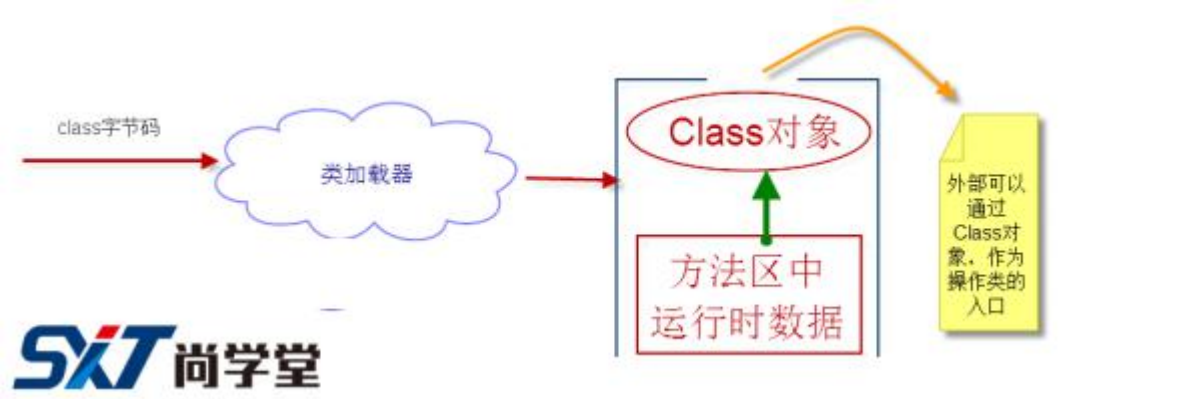
- 为什么研究类加载全过程?
  - 有助于了解JVM运行过程
  - 更深入了解java动态性，(解热部署、动态加载)，提高程序的灵活性。

类加载全过程

- 类加载机制
  - JVM把class文件加载到内存，并对数据进行校验、解析和初始化，最终形成JVM可以直接使用的Java类型的过程。



  - 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。



类加载全过程

- 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
  - 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。
  - 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、则需要先出发其父类的初始化
  - 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。

类加载全过程

- 类的主动引用（一定会发生类的初始化）
  - new一个类的对象
  - 调用类的静态成员(除了final常量)和静态方法
  - 使用java.lang.reflect包的方法对类进行反射调用
  - 当虚拟机启动，java Hello，则一定会初始化Hello类。说白了就是先启动main方法所在的类
  - 当初始化一个类，如果其父类没有被初始化，则先会初始化他的父类

- 类的被动引用(不会发生类的初始化)
  - 当访问一个静态域时，只有真正声明这个域的类才会被初始化
    - 通过子类引用父类的静态变量，不会导致子类初始化
  - 通过数组定义类引用，不会触发此类的初始化
  - 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）

深入类加载器

讲师：高淇　邮箱：gaoqi110@163.com

深入类加载器

内容大纲

01 类加载器原理

02 类加载器树状结构、双亲委托(代理)机制

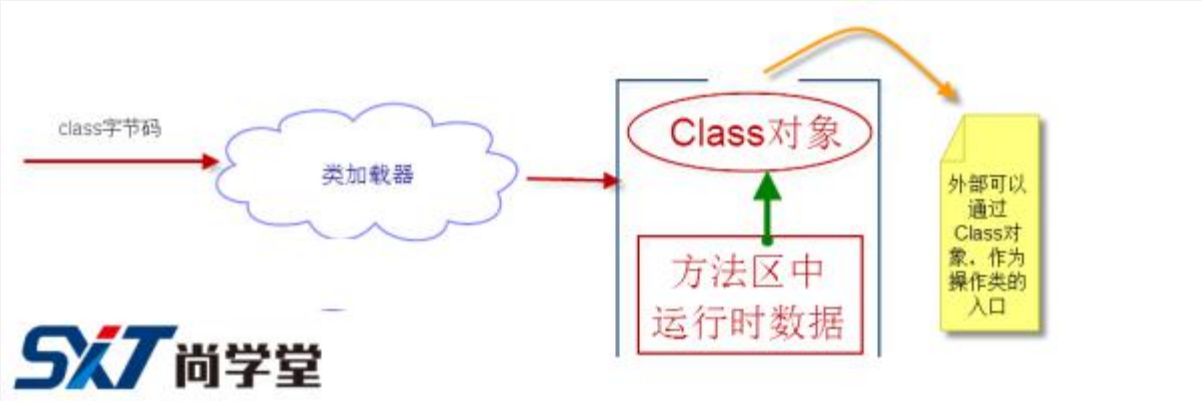
03 自定义类加载器(文件、网络、加密)

04 线程上下文类加载器

05 服务器类加载原理和OSGI介绍

类加载器的作用

- 类加载器的作用
  - 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对象，作为方法区类数据的访问入口。



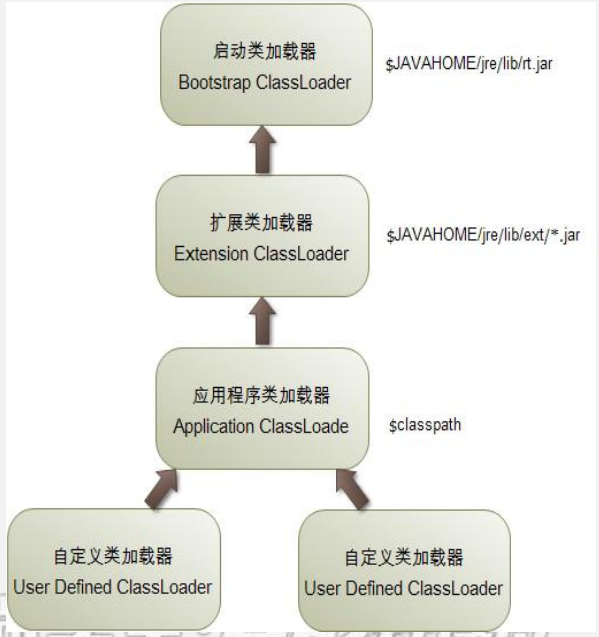
- 类缓存
  - 标准的Java SE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过，JVM垃圾收集器可以回收这些Class对象。
  -

java.class.ClassLoader类

- 作用：
  - java.lang.ClassLoader类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class类的一个实例。
  - 除此之外，ClassLoader还负责加载 Java 应用所需的资源，如图像文件和配置文件等。
- 相关方法
  - getParent()　　返回该类加载器的父类加载器。bootstrap因为由c编写，所以返回值为null
    loadClass(String name)　　加载名称为 name的类，返回的结果是 java.lang.Class类的实例。
    findClass(String name)　　查找名称为 name的类，返回的结果是 java.lang.Class类的实例。
    findLoadedClass(String name)　查找名称为 name的已经被加载过的类，返回的结果是 java.lang.Class类的实例。
    defineClass(String name, byte[] b, int off, int len)　把字节数组 b中的内容转换成 Java 类，返回的结果是 java.lang.Class类的实例。这个方法被声明为 final的。
    resolveClass(Class<?> c)　　链接指定的 Java 类。
  - 对于以上给出的方法，表示类名称的 name参数的值是类的二进制名称。需要注意的是内部类的表示，如 com.example.Sample$1和com.example.Sample$Inner等表示方式。

类加载器的层次结构(树状结构)

- 引导类加载器（bootstrap class loader）
  - 它用来加载 Java 的核心库(JAVA_HOME/jre/lib/rt.jar,或sun.boot.class.path路径下的内容)，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。
  - 加载扩展类和应用程序类加载器。并指定他们的父类加载器。
- 扩展类加载器（extensions class loader）
  - 用来加载 Java 的扩展库(JAVA_HOME/jre/ext/*.jar，或java.ext.dirs路径下的内容)。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
  - 由sun.misc.Launcher$ExtClassLoader实现
- 应用程序类加载器（application class loader）
  - 它根据 Java 应用的类路径（classpath，java.class.path 路径下的内容）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。
  - 由sun.misc.Launcher$AppClassLoader实现
- 自定义类加载器
  - 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。



类加载器的代理模式

- 代理模式
  - 交给其他加载器来加载指定的类
- 双亲委托机制
  - 就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次追溯，直到最高的爷爷辈的，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。
  - 双亲委托机制是为了保证 Java 核心库的类型安全。
    - 这种机制就保证不会重写string类的情况,由顶级父加载器确认
  - 类加载器除了用于加载类，也是安全的最基本的屏障。
- 双亲委托机制是代理模式的一种
  - 并不是所有的类加载器都采用双亲委托机制。
  - tomcat服务器类加载器也使用代理模式，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的

java.class.ClassLoader类API

- 相关方法
  - getParent()　　返回该类加载器的父类加载器。
  - loadClass(String name)　　加载名称为 name的类，返回的结果是 java.lang.Class类的实例。
    - 此方法负责加载指定名字的类，首先会从已加载的类中去寻找，如果没有找到；从parent ClassLoader[ExtClassLoader]中加载；如果没有加载到，则从Bootstrap ClassLoader中尝试加载(findBootstrapClassOrNull方法), 如果还是加载失败，则自己加载。如果还不能加载，则抛出异常ClassNotFoundException。
    - 如果要改变类的加载顺序可以覆盖此方法；
  - findClass(String name)　　查找名称为 name的类，返回的结果是 java.lang.Class类的实例。
  - findLoadedClass(String name)　查找名称为 name的已经被加载过的类，返回的结果是 java.lang.Class类的实例。
  - defineClass(String name, byte[] b, int off, int len)　把字节数组 b中的内容转换成 Java 类，返回的结果是 java.lang.Class类的实例。这个方法被声明为 final的。
  - resolveClass(Class<?> c)　链接指定的 Java 类。
  -
  -

    表示类名称的 name参数的值是类的名称。需要注意的是内部类的表示，如 com.example.Sample$1和 com.example.Sample$Inner等表示方式。

自定义类加载器

- 文件系统类加载器

- 自定义类加载器的流程：
  - 1、首先检查请求的类型是否已经被这个类装载器装载到命名空间中了，如果已经装载，直接返回；否则转入步骤2
  - 2、委派类加载请求给父类加载器（更准确的说应该是双亲类加载器，真个虚拟机中各种类加载器最终会呈现树状结构），如果父类加载器能够完成，则返回父类加载器加载的Class实例；否则转入步骤3
  - 3、调用本类加载器的findClass（…）方法，试图获取对应的字节码，如果获取的到，则调用defineClass（…）导入类型到方法区；如果获取不到对应的字节码或者其他原因失败，返回异常给loadClass（…）， loadClass（…）转抛异常，终止加载过程（注意：这里的异常种类不止一种）。
  - **注意：被两个类加载器加载的同一个类，JVM不认为是相同的类。**

- 文件类加载器
- 网络类加载器
- 加密解密类加载器（取反操作，DES对称加密解密）

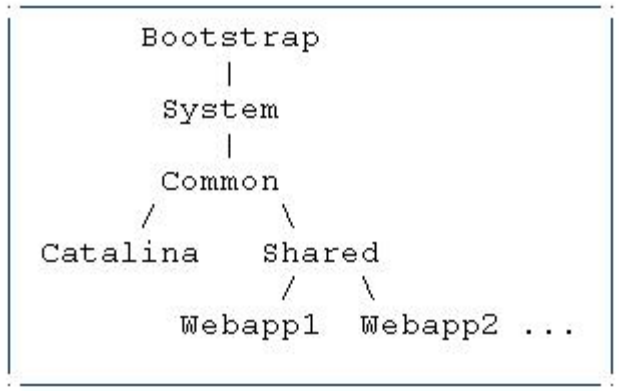
线程上下文类加载器

- 双亲委托机制以及默认类加载器的问题
  - 一般情况下, 保证同一个类中所关联的其他类都是由当前类的类加载器所加载的.。
    比如，ClassA本身在Ext下找到，那么他里面new出来的一些类也就只能用Ext去查找了（不会低一个级别），所以有些明明App可以找到的，却找不到了。
  - JDBC API，他有实现的driven部分（mysql/sql server），我们的JDBC API都是由Boot或者Ext来载入的，但是JDBC driver却是由Ext或者App来载入，那么就有可能找不到driver了。在Java领域中，其实只要分成这种Api+SPI（Service Provide Interface，特定厂商提供）的，都会遇到此问题。
  - 常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 javax.xml.parsers 包中。**SPI 的接口是 Java 核心库的一部分，是由引导类加载器来加载的;SPI 实现的 Java 类一般是由系统类加载器来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。**

- 通常当你需要动态加载资源的时候，你至少有三个 ClassLoader 可以选择：
  - 1.系统类加载器或叫作应用类加载器 (system classloader or application classloader)
  - 2.当前类加载器
  - 3.当前线程类加载器
- 当前线程类加载器是为了抛弃双亲委派加载链模式。
  - 每个线程都有一个关联的上下文类加载器。如果你使用new Thread()方式生成新的线程，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的话，程序中所有的线程将都使用系统类加载器作为上下文类加载器。
- Thread.currentThread().getContextClassLoader()

TOMCAT服务器的类加载机制

- 一切都是为了安全！
  - TOMCAT不能使用系统默认的类加载器。
    - 如果TOMCAT跑你的WEB项目使用系统的类加载器那是相当危险的，你可以直接是无忌惮是操作系统的各个目录了。
    - 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。该类加载器也使用代理模式(不同于前面说的双亲委托机制)，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的。但也是为了保证安全，这样核心库就不在查询范围之内。
- 为了安全TOMCAT需要实现自己的类加载器。
  - 我可以限制你只能把类写在指定的地方，否则我不给你加载！



OSGI原理介绍

- OSGi™是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式用来管理软件的生命周期。
- OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。**Eclipse 就是基于 OSGi 技术来构建的。**
- **原理：**
  - OSGi 中的每个模块（bundle）都包含 Java 包和类。模块可以声明它所依赖的需要导入（import）的其它模块的 Java 包和类（通过 Import-Package），也可以声明导出（export）自己的包和类，供其它模块使用（通过 Export-Package）。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。**这是通过 OSGi 特有的类加载器机制来实现的。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。**当它需要加载 Java 核心库的类时（以 java开头的包和类），它会代理给父类加载器（通常是启动类加载器）来完成。**当它需要加载所导入的 Java 类时，它会代理给导出此 Java 类的模块来完成加载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性 org.osgi.framework.bootdelegation 的值即可。**