# JVM 虚拟机类加载

1. 类型的加载，连接和初始化都是在程序运行期间完成的
2. 加载：查找并加载类的二进制数据
3. 连接：

* --验证：确保被加载的类的正确性
* --准备：为类的静态变量分配内存，并将其初始化为默认值
* --解析：把类中的符号引用转化为直接引用

1. 初始化：为类的静态变量赋予正确的初始值
2. Java程序对类的使用方式分为两种
3. 主动使用
4. 被动使用
5. 所有的java虚拟机实现必须在每个类或接口被java程序“首次主动使用”的时候才进行初始化
6. 主动使用（7种方式）

* -创建类的实例 new
* -访问某个类或接口的静态变量，或对改静态变量进行赋值
* -调用类的静态方法
* -反射 例如 Class.forName(“java.util.Date”)
* -初始化一个类的子类
* -java虚拟机启动时被标注为启动类的类 Test main方法
* -动态语言调用 java.lang.invoke.MethodHanlde中实例解析结果REF\_getStatic ,REF\_putStatic ,REF\_invokeStatic 句柄中没有实例化则会实例化

案列

class parent {

public static String str = “helloworld”;

static {

System.out.println(“parent static block”);

}

}

class child extends parent{

public static String str1 = “welcome”;

static {

System.out.println(“child static block”);

}

}

void main(){

System.out.println(child.str); //对于静态字段来讲，只有定义了该字段的类才会被初始化

//因此这里的结果是，子类不会被初始化，静态代码块不会被执行；而父类会被初始化，静//态代码块被执行；static{} 静态代码块只会在初始化的时候执行一次

}

void main(){

System.out.println(child.str1); //一个类初始化时，要求其父类全部初始化完成

}

1. 接口中默认的变量是 public static final ,因此子接口倘若在编译期间不能确定常量值，父类子类必定会初始化；若是常量，必将不会初始化
2. 准备过程是初始化默认值，主动调用后会进行静态变量的初始化，有且只会初始化一次
3. 类初始化结束后，类实例化；实例化一个类生成一个对象

* 为新的对象分配内存
* 为实例变量赋予默认值
* 为实例变量赋正确的初始值
* Java编译器为它编译的每一个类至少生成一个实例初始化方法，在java 的class文件中这个实例化初始方法被称为“<init>”,针对源代码中每一个类的构造方法，java编译器都会产生一个<init >方法

1. 类加载的最终产物是存放在方法区的Class对象，它能发射出类的在方法区的数据结构
2. 类加载分为自带的类加载器与自定义类加载器

* 系统自带的类加载器
* 根加载器 bootstrap
* 扩展类加载器 Extension
* 系统应用加载器 AppSystem

1. 自定义加载器 java.lang.ClassLoader的子类，用户阔以自定义类加载器、
2. 类加载器不需要首次使用某一个类的时候才进行这个类的加载
3. jvm规范允许类加载器在预料某个类将要被使用时提前加载它，如果在预先加载过程中遇到.class 文件缺失或者存在错误的时候，类加载必须在程序首次主动使用该类的时候报告错误，如果一直未被使用则不应该报该错误（LinkageError错误）
4. 当java初始化一个类似，要求它的所有父类都已经初始化完毕，但是有些列子除外

* 当初始化一个接口时，它所继承的接口并不会被初始化
* 当一个类被初始化时，并不会初始化它所实现的接口

因此，一个父接口并不会因为因为它的子接口或者子类的初始化而初始化，只有当程序首次使用父接口的静态变量时，父接口才会被初始化（例如父接口存在静态变量在编译期间无法确定其值）

1. 双亲委托机制，父亲委托机制进行类的加载

Sample类

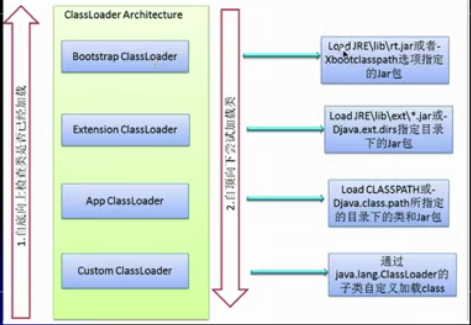
自定义Loader2加载器

自定义Loader1加载器

系统应用类加载器

扩展类加载器

根类加载器



1. 获得ClassLoader 对象的途径

* --根据类的字节码获取 clazz.getClassLoader();
* --通过线程上下文方式获取 Thread.currentThread.getContextClassLoader();
* --通过Classloader类获取应用类加载器 ClassLoader.getSystemClassLaoder();
* --通过获取调用者的classLoader DriverManager.getCallerClassLoader();

1. 自底向上检查类是否已经加载，自顶向下尝试加载类
2. 命令空间

* 每个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器的类组成
* 在同一个命名空间，不会出现类的完整名字（包括类的包名）相同的两个类
* 在不同的命名空间。有可能出现类的完整名字（包含类的包名）的相同的两个类

1. 子加载器所加载的类能访问到父加载器所加载的类，但是父加载器的类不能访问子加载器所加载的类（跟命名空间有关系），往下的命名空间能访问上的命名空间，但是上面的命名空间不能访问往下的命名空间
2. 类加载器的双亲委托模型的好处（必考）

* 可以确保java核心类库的安全，所有的java应用都至少会引用java.lang.Object类，也就是在运行期java.lang.Object这个类会被加载到java虚拟机中，如果这个加载过程是由java应用自己的类加载所完成，那么jvm中会存在多个版本的java.lang.Object类，而这些类是不兼容，相互不可见（命名空间的作用）
* 可以确保java核心库所提供的类不会被自定义的类所替代。（自定义一个java.lang.String这个类去覆盖系统核心类库，会在defineClass中报securityException）
* 不同的类加载器（尽管类名一样有两个实例化也为不同的类加载器）可以为相同名称（binary name）的类创建额外的命名空间。相同名称的类可以并存在jvm中，只需要用不同的类加载器加载即可。但是这样所加载的类尽管包名类名都相同，但是他们却相互隔离，实例化出的对象是不可相互转换（不兼容）；jvm虚拟机内部创建了一个又一个相互隔离的Java类空间。

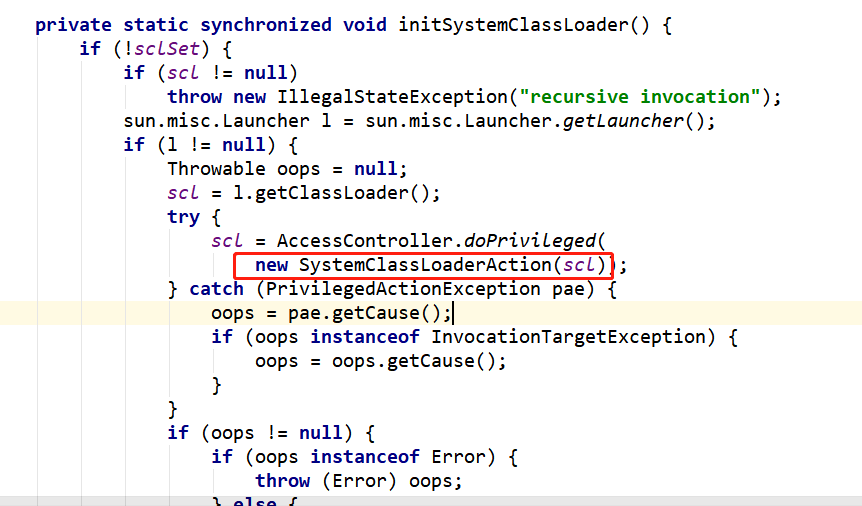
1. extClassLoader扩展类加载器的使用

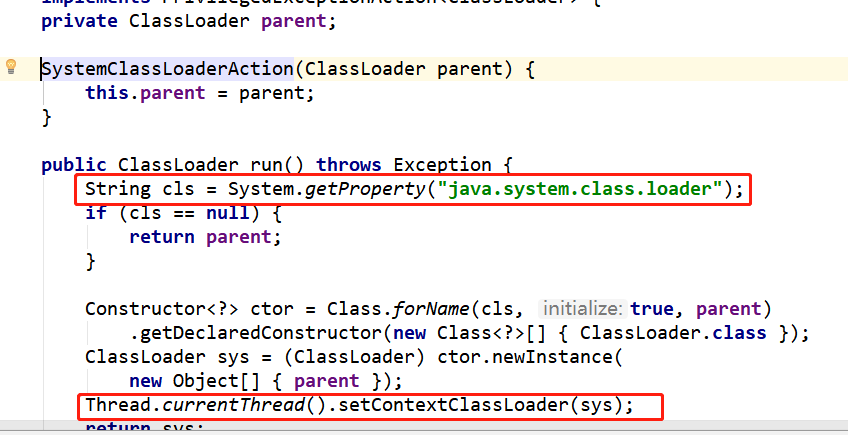
* 需要将class文件打包成jar 才会被该类加载器所加载，根加载器与系统类加载器是阔以直接加载.class文件
* 打包jar命令为：jar cvf com/jvm/classloader/Mytest.class
* 启动命令，这里更改扩展类加载的目录： java -Djava.ext.dirs=./ com/jvm/classloader/MyTest16
* 查看不同类加载器所加载的目录：System.getProperty(“sun.boot.class.path”);System.getProperty(“java.ext.dirs”);System.getProperty(“java.class.path”);

1. 在运行期间，一个java类是由该类的完全限定名（binary name 二进制名）和用于加载该类的定义类加载器（defining loader）所共同决定。 如果同样的名字（既相同的完全限定名）的类是由俩个不同的类加载所加载，那么这两个类也是不同的，即便.class 文件的字节码完全一样，并且从同样的位置加载亦如此。
2. 一对父子加载器可能是一个类的两个实例，也可能不是，子加载器对象中包装了一个父加载器对象。
3. 自定义类加载器不可能加载应该由父类加载的可靠类。双亲委托机制的优点保证核心类的安全。
4. 在oracle 的Hotspot 实现中，系统属性sun.boot.class.path如果修改错误，则会运行出错，提示信息： Error occurred during initialization of VM java/lang/noClassDefFoundError:java/lang/object
5. Jar hell 问题：当工程依赖多个jar并且有相同的，阔以通过以下手段去诊断：

ClassLoader systemClassLoader = ClassLoader.*getSystemClassLoader*();  
String resourceName = **"java"**+ File.***separator***+**"lang"**+File.***separator***+**"String"**;  
Enumeration<URL> ite = systemClassLoader.getResources(resourceName);  
**while** (ite.hasMoreElements()){  
 System.***out***.println(ite.nextElement());  
}

1. 成功加载类的类加载器就称为定义类加载器，能成功返回的类加载器为初始类加载器。
2. 一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期（Class对象指向NULL 该类就会被卸载）因此自定义的类加载只要失去引用则，对应加载的类就会结束它的生命周期。系统自带的类加载器会一直被引用，该类加载器也会一直引用它所加载的类。所以系统类加载器加载的类不会被卸载。
3. 初始化有两种方式：（1）在静态变量的声明处初始化 （2）在静态代码块中进行初始化
4. 一个类被加载，会把这个类所生成的Class对象存入类加载器中以map 的形式保存；因此该对象可以通过getClassLoader()得到这个它所被加载的类加载；然而类加载器也会存有唯一一份该类的Class对象，因此类所生成的Class对象与类加载器是一一对应
5. 一个类的实例化会引用该类的类对象（Class对象），每个类的实例化由java.lang.Object 提供getClass()获取到该类的类对象Class的引用；同理每个类都会有一个静态变量可以直接获取该类的Class对象的引用（在命名空间可见的情况下才能访问）；**不可见的情况下阔以使用Object所提供的getClass()方法去获取。、**
6. **自定义设置系统类加载器：**





当前类加载器（current classloader）

1. 每个类都会使用自己的类加载器去加载其他类（所依赖的类），如果class A 引用了class Y 那么 class A 的类加载器就会去加载Y （Y 未被加载）

线程类加载器：1.2开始引入，类中的getContextClassLoader() 与setContextClassLoader(ClassLoader)来获取，设置上下文线程类加载器，若果没有设置默认会集成父线程的上下文类加载器

SPI(service provider interface）

父classloader可以使用当前线程所指定的上下文所指定的classloader加载类，这就改变了父classloader 不能使用子的classloader或者是没有直接父子关系的classloader加载的类的情况，既改变了双亲委托模型

线程上下文类加载器就是current class loader

在双亲委托模型下，类的加载是由下至上的，既下层的累加器会委托父加载器进行加载，但是对于spi来讲，有些接口是java核心库所提供的，而java核心库是由启动类加载器加载的，而这些接口的实现来至于不同的厂商，java的启动类加载器是无法加载来源去其它jar包，这样传统的双亲委托模型就无法满足SPI的要求，而通过恰当给当前的线程时设置上下文类加载器，就阔以有设置的上下文类加载器来实现对于接口的实现类的加载。

线程上下文类加载的一般使用：获取 使用 还原

ClassLoader cl = thread.currentThread.getContectClassLoader()

Try{

thread.currentThread.setContectClassLoader(target);

Method();

}finally{

thread.currentThread.setContectClassLoader(cl );

}

**当高层提供了统一接口让底层去实现，同时又要在高层加载或者实例化底层的类时，就必须通过线程上下文类加载器来帮助高层classloader找到并加载该类。**

1. Class字节码分析： 魔术 （4字节 CAFE BABE），minor version ，major version , 常量数量，常量池（Class-info,Field-info （tag ,index,index）, Method-info(tag,index,index),UTF-8-info(tag,length,bytes[length])，String-info(tag 1字节,index 2字节)，nameAndType-info(tag ,index 名字,index 声明类型)等）， access flag， this class ，super class， interface-count ,interface ,field count ,field ,method count,method , attribute count,attribute
2. Jvm中字节码有两种数据类型： 字节数据直接量（U1 U2 U4 U8） 表（数组）
3. 常量池中的个数为 常量数量转换为十进制减一；0索引不占用表示NULL
4. 字节码在分析方法时，会自动生成构造方法，为实例变量赋值；会为每个方法产生两个attribute表分别是linenumberTable（每行机器码对应的java代码便于调试），localvariableTable 传入 this 变量，与方法参数变量；因此可以非静态方法中使用this关键字
5. 有静态属性字段，编译器会自动生成<clinit>方法对静态属性赋值（多个也只会生成唯一一个）；没有构造方法则会自动产生，有显示声明的构造放法，同样也会首先执行相同的指令（调用父类构造方法invokeSpecial ，为非静态变量赋值），然后再执行编写的代码进行初始化。
6. Synchronize 关键字；产生的字节码关键字为monitorenter ，monitorexit ;monitorenter唯一一个；但是monitorexit 会用多个，异常时也需要释放锁，可以重入。
7. 对于java类中的每一个方法（非static方法），其在编译后所生成的字节码当中，方法参数的数量总是会比实例方法参数的数量多一个（this 参数），它位于方法的第一个参数；因此就阔以在方法中使用 this来访问对象的属性及方法。
8. 该操作**是在编译期完成**，由javac编译器在编译的时候将对this的访问转化为对一个普通实例参数的访问，在运行期间又jvm虚拟机在调用实例方法时，自动向实例方法传入this参数，在实例方法的局部变量表中，至少会有一个指向当前对象的局部变量。
9. Java字节码对异常的处理：

* 采取异常表的方式处理异常{start pc , end pc , handler pc , catch type(0 处理所有默认都包含)}
* 当异常处理存在finally ,jvm虚拟机会把finally产生的字节码拼接到各个异常块中；采用复制的方式；有多少个catch就会拼接多个。

1. 栈帧本是一种数据结构，封装了方法的局部变量，动态链接信息，方法的返回地址与操作数栈信息等
2. **符号引用转直接引用两种方式**：有些符号引用是在类的加载阶段或第一次使用机会转换为直接引用，这种转换叫做静态解析；另外一些符号引用则是在每次运行期转换为直接引用，这种转换叫做动态转换；体现为java的多态性。

*/\*  
invokeInterface: 调用接口的方法，根据运行时期确定调用那一个实现类的方法  
invokeStatic: 调用静态方法  
invokeSpecial: 调用私有方法，<init>构造方法，父类方法  
invokeVirtual: 调用虚方法，运行期间动态查找  
invokeDynamic: 动态调用方法  
\* \*/  
  
/\*  
静态解析的4中情形  
1 静态方法  
2 父类方法  
3 构造方法  
4 私有方法  
以上4类属于非虚方法，在类加载解析时就可以由符号引用转换为直接引用  
\* \*/*

1. 方法的静态分派与方法的动态分派；主要展现在类的重载（overload）重写（override）字节码都为invokeVirtual
2. 方法的静动分派变现涉及一个重要概念：方法的接受者（方法由谁来调用）；首先从栈帧中取出栈顶的元素；然后根据该元素的实际类型作为方法的接受者；如果方法的接受者就是编译期所指定的类型；那么就根据方法重载的特征，静态分派特征去确定，给定参数静态类型是什么就指定到确切的某一个方法；而不是编译器所指定的类型；编译期制定为父类的类型，通过动态分派，从子类往父类上依次寻找；方法的重写动态特征去找到该方法从而执行。
3. 方法重载是静态分派；属于编译期行为（编译执行期）；方法重写是动态分派，属于运行期行为

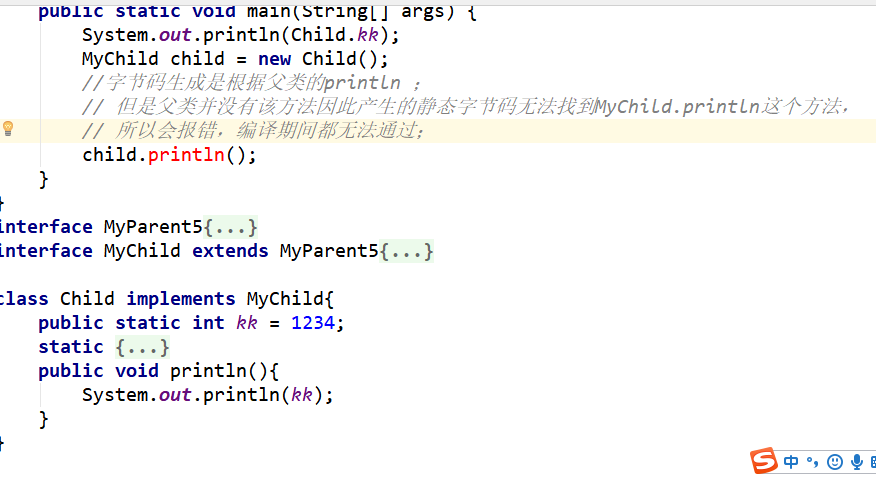
方法的静态分派：

GrandPa g = new father（）；

g的静态类型是GrandPa 而实际类型却是father

结论：变量的静态类型是不会发生变化的，而变量的实际类型是可以发生变化的（多态），实际类型是在运行期间确定的。

//方法重载是静态分派；属于编译期行为

1. 
2. 针对方法调用动态分派的过程；虚拟机会在类的方法区建立一个虚方法表的数据结构 vtable; 针对invokeInterface 虚拟机会建立一个叫做接口方法表的数据结构 itable;从而动态查找调用的方法。
3. JVM 执行的方式有两种：解释器执行（通过解释器读取字节码，遇到相应指令就解释该指令执行），编译执行（JIT编译器把热点代码所编译的机器码存下来，快速调用）混合使用两种执行器
4. 基于栈的指令集，java所采用的，可移植；基于寄存器指令集快但是与硬件相绑定
5. JVM虚拟机内存结构：

* 虚拟机栈：线程所属有；本地局部变量，操作数栈
* 程序计数器： 下一步要执行的代码（线程所开辟）
* 本地方法栈： native方法（线程所开辟）
* 堆（heap）：JVM管理最大一块内存空间；与堆相关的一个重要观念垃圾收集器，现代几乎所有的垃圾收集器都是采用的分代算法；因此对堆空间进行了划分，新生代与老年代；Eden空间，From survive 与 To survive 空间；
* 方法区：存储元信息，把它叫做元空间（meta space），运行时的常量池：方法区的一部分；操作系统本地内存（初始21M），不连续的。废除了永久代,超过初始大小会不断扩大，直到达到物理内存。存放的是一个类的Class的元信息。
* 直接内存：direct memory ，与java NIO 紧密关联，JVM通过堆上的DirectByteBuffer来操作直接内存

1. Java对象的创建过程

* New关键字创建的三个步骤
* 在堆内存中创建出对象的实例
* 为对象的实例成员变量赋初始值
* 讲对象的引用返回

1. 指针碰撞：前提是堆中的空间通过一个指针进行分割，一侧是已经被占用的空间，一侧是阔以使用的空间；就阔以给新对象的实例初始化空间，指针进行偏移
2. 空闲列表，堆空间已被使用与未被使用的空间是交织在一起的；虚拟机就需要用空闲列表来记录那些空间是阔以用的；接下来找出阔以容下新创建对象的空间从而存放该对象

对象在内存中的布局：

* 对象头（hash值，分代信息，运行时的信息）
* 实例数据（类中申明的成员变量）
* 对齐填充（填充）

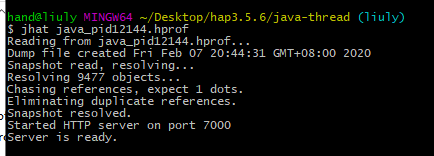
对象的引用：从本地方法栈中如何通过去访问对象

通过句柄的方式引用对象 用两部分指针：一个指针指向对象，一个指针指向类型信息（元数据信息）使用起来比较方便

直接引用该对象 （类型信息包含元数据信息的指针），对于堆空间的压缩比较方便（垃圾收集器回收）

1. 设置元空间的最大值 ： -XX:MaxMetaSpaceSize=200m -XX:+traceClassLoading
2. Jmap -heap PID 查看堆信息 jmap -clstats pid 查看类加载器的信息 **jstat -gc pid 查看元空间大小 MC 当前容量 MU 使用容量 jcmd pid GC.class\_stats JVM 详尽的类元数据树状图**
3. jmap -histo 3331 (jmap -histo:live 这个命令执行，JVM会先触发gc，然后再统计信息。)
4. Jps 与 jcmd 命令列出所有的java进程 -l -v -m 等参数
5. Jcmd pid VM.flags 查看进程的启动参数 通过**jcmd pid help** 阔以看到java进程阔以执行的操作（VM.flags GC.run等） VM.version 版本信息
6. Jcmd pid VM.flags help 查看该命令有哪些选项
7. Jcmd pid perfCounter.print 查看jvm性能相关的参数
8. Jcmd pid VM.uptime 查看该类的运行时常
9. Jcmd pid GC.class\_histogram 查看JVM 类的完整的统计信息，类的实例数，占用字节
10. Jcmd pid Thread.print 查看JVM 类的线程信息
11. Jcmd pid GC.heap\_dump <filepath+filename> 堆栈信息转储
12. Jcmd pid GC.heap\_dump /root/test.hprof 会暂停应用程序，查看堆的详细信息
13. Jcmd pid VM.system\_properties 查看jvm的属性信息
14. Jcmd pid VM.command\_line 查看jvm启动时命令行参数
15. Jstack 可以查看线程栈信息 jstack pid 就是为了获取线程的信息
16. Jmc 工具：java Mission control jfr: java flight recorder
17. 设置堆的大小： -Xms5m -Xmx5m -XX:+HeapDumpOnOutOfMemoryError 出现堆溢出存储堆信息

*// -Xss160k 指定虚拟机栈的大小 stack size  
//指定元空间大小  
//-XX:MaxMetaSpaceSize=200m -XX:+TracingClassLoading*



1. Jhat 分析堆转储文件信息
2. **类包含其对应的元数据，比如类的层级信息，方法数据和方法信息（如字节码，栈和变量大小），运行时常量池，已确定的符号引用和虚方法表（类的静态信息元数据）**
3. **栈帧本是一种数据结构，封装了方法的局部变量，动态链接信息，方法的返回地址与操作数栈信息等（存储一般都是动态的信息以消失）**