# 类加载

在Java代码中，类型的加载、连接与初始化过程都是在程序运行期间完成的。这提供了更大的灵活性，增加了更多的可能性。

## 类加载器

每个Java类都是由类加载器加载到内存中的。

在如下情况下，Java虚拟机将结束程序的生命周期：

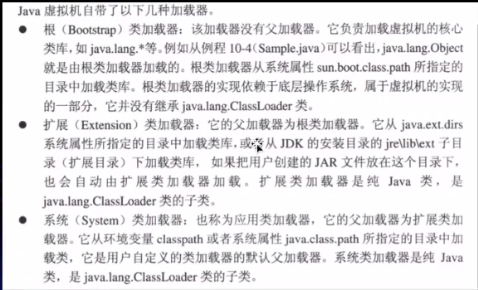
1. 程序执行了System.exit() 方法

2. 程序正常执行结束

3. 程序在执行过程中遇到了异常或错误而异常终止

4. 由于操作系统出现错误而导致Java虚拟机进程终止

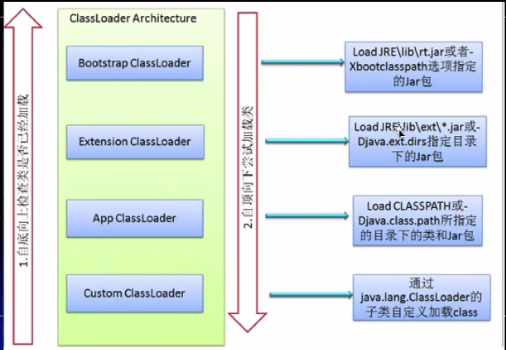
从JDK1.2开始，类的加载过程采用父亲委托机制，这种机制能更好地保证Java平台的安全。在此委托机制中，除了Java虚拟机自带的根加载器以外，其余的类加载器都有且只有一个父加载器。例如，当Java程序请求加载器loader1加载Sample类时，loader1首先委托自己的父加载器去加载Sample类，若父加载器能加载，则由父加载器完成加载任务，否则才由加载器loader1本身加载Sample类。



### 类加载器的父亲委托机制

在父亲委托机制中，各个加载器按照父子关系形成了**树形结构**，除了根类加载器之外，其余的类加载器都有且只有一个父加载器。

各级类加载器在没有特殊配置的情况下，只负责加载指定路径下的类和jar包。



在委托加载的过程中，只要有一个加载器成功把类加载，就会返回加载成功。

Oracle JDK的类加载器采用了双亲委托机制，但并不是所有规范的JVM都会按照这一标准进行类的加载操作，例如OSGI的JVM就不遵循双亲委托机制。

**Bootstrap ClassLoader**——启动类加载器

负责加载JavaHome中jre/lib/rt.jar或环境变量**sun.boot.class.path**里所有的class，由C++实现，不是ClassLoader的子类。

在Oracle的Hotspot实现中，系统属性sun.boot.class.path如果修改错了，则运行会出错，提示如下错误信息：

Error occurred during initialization of VM

java/lang/NoClassDefFoundError: java/lang/Object

**Extension ClassLoader**——扩展类加载器，是**系统类加载器的parent**

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

**扩展类加载器只加载jar包里的class文件**，而不能加载文件系统中的class文件，实例参见《23\_类加载器命名空间总结与扩展类加载器要点分析》14分钟以后的内容。

**App ClassLoader**——系统类加载器

负责加载**classpath**或环境变量**java.class.path**中指定的jar包及目录中的class。

通过ClassLoader.getSystemClassLoader()方法可以返回用于委托的系统类加载器，详见这个方法的JavaDoc以及第25课视频4分钟左右的位置。

在运行java命令时，指定java.system.class.loader参数，可以指定自定义类加载器作为系统类加载器，但前提是自定义的类加载器必须实现一个ClassLoader类型参数作为单一参数的构造方法（参考MyTest18和MyTest16的代码）。这样定义的原因参考第26课的视频17分钟以前的内容。

#### 源码实现

扩展类加载器和系统类加载器的类ExtClassLoader和AppClassLoader属于sun.misc.Launcher的静态内部类。

Launcher类的构造方法解析（详见第25课视频）：

1. 创建扩展类加载器，使用ExtClassLoader.getExtClassLoader()创建；

2. 创建系统类加载器，使用AppClassLoader.getAppClassLoader(扩展类加载器)创建；

3. 把创建的系统类加载器设置为当前线程的上下文类加载器；

4. 设置系统安全管理器（System security manager）

ClassLoader类的getSystemClassLoader方法的源码实现，详见第26课视频。

若有一个类加载器能够成功加载Test类，那么这个类加载器被称为**定义类加载器**，所有能成功返回Class对象引用的类加载器（包括定义类加载器）都被称为**初始类加载器**。

**表面上看，用户自定义类加载器🡪系统类加载器🡪扩展类加载器🡪启动类加载器，好像是一种树形的层次关系，但实际上是子加载器包含了父加载器的引用。**

需要指出的是

加载器之间的父子关系实际上指的是加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。在子加载器对象中包装了一个父加载器对象。例如以下loader1和loader2都是MyClassLoader类的实例，并且loader2包装了loader1，loader1是loader2的父加载器。

|  |
| --- |
| ClassLoader loader1 = new MyClassLoader();  // 参数loader1将作为loader2的父加载器  ClassLoader loader2 = new MyClassLoader(**loader1**); |

内建于JVM中的启动类加载器会加载java.lang.ClassLoader以及其他的Java平台类（如java.lang.Object、java.lang.String等等），当JVM启动时，一块特殊的机器码会运行，它会加载扩展类加载器与系统类加载器。这块特殊的机器码叫做启动类加载器（Bootstrap）。

启动类加载器并不是Java类，而其他的加载器则都是Java类。

启动类加载器是特定于平台的机器指令，它负责开启整个加载过程。

除了启动类加载器以外的所有类加载器都被实现为Java类。不过，总归要有一个组件来加载第一个Java类加载器，从而让整个加载过程能够顺利进行下去，加载第一个纯Java类加载器就是启动类加载器的职责。

启动类加载器还会负责加载供JRE正常运行所需要的基本组件，这包括java.util与java.lang包中的类等等。

扩展类加载器与系统类加载器也是由启动类加载器所加载的。

#### 命名空间

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所加载的类组成**。

在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类。

在不同的命名空间中，有可能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，相同的类可以被加载多次。

##### **重要说明**

（详细参考MyTest16、MyTest17类代码以及第19、20两课视频，通过对代码修改和class文件的删除更能加深理解）

1. 子加载器所加载的类能够访问父加载器所加载的类

2. 父加载器所加载的类无法访问子加载器所加载的类

在运行期，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）所共同决定的。

如果同样名字（即相同的完全限定名）的类是由两个不同的类加载器所加载，那么这些类就是不同的，即便.class文件的字节码完全一样，并且从相同的位置加载亦如此。

##### 不同类加载器的命名空间的关系

|  |
| --- |
| 同一个命名空间内的类是相互可见的。  **子加载器的命名空间包含所有父加载器的命名空间**。因此由子加载器加载的类能看见父加载器加载的类。例如，系统类加载器加载的类能看见根类加载器加载的类。  由父加载器加载的类不能看见子加载器加载的类。  如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见。 |

#### 类加载器的双亲委托模型的缺陷：

父加载器加载的类不能看见子加载器加载的类，那么在以下这种情景下，双亲委托机制就会出现缺陷。

情景是：接口Interface由启动类加载器加载，而这个接口的实现类由第三方实现后再加入ClassPath中，就应该被系统类加载器加载，所以按照父加载器加载的类不能看见子加载器加载的类的限制，接口就看不见实现类，即Connection conn = DriverManager.getConnection()这样的代码是不能正常执行的。类加载器的双亲委托模型无法实现SPI（Service Provider Interface）机制。

线程上下文类加载器可以解决双亲委托机制带来的这种缺陷。

#### 类加载器的双亲委托模型的好处：

1. 可以确保Java核心库的类型安全：所有的Java应用都至少会引用java.lang.Object类，也就是说在运行期，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应用自己的类加载器所完成的，那么很可能就会在JVM中存在多个版本的java.lang.Object类，而且这些类之间还是不兼容的，相互不可见的（正是命名空间在发挥着作用）。

2. 可以确保Java核心类库所提供的类不会被自定义的类所替代。

3. 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Java虚拟机中，只需要用不同的类加载器来加载他们即可。不同类加载器所加载的类之间是不兼容的，这就相当于在Java虚拟机内部创建了一个又一个相互隔离的Java类空间，这类技术在很多框架中都得到了实际应用。

### 获得ClassLoader的途径

获取当前类的ClassLoader

clazz.getClassLoader();

获取当前线程上下文的ClassLoader

Thread.currentThread().getContextClassLoader()

获得系统的ClassLoader

ClassLoader.getSystemClassLoader()

获取调用者的ClassLoader

DriverManager.getCallerClassLoader()

### Jar hell问题以及解决办法

当一个类或者一个资源文件存在于多个jar中，就会存在jar hell问题。

可以通过以下代码来诊断问题：

|  |
| --- |
| ClassLoader classLoader = Thread.currentThread().getContextClassLoader();  String resourceName = "java/lang/String.class";  Enumeration<URL> urls = classLoader.getResources(resourceName);  while (urls.hasMoreElements()) {  URL url = urls.nextElement();  System.out.println(url);  } |

如果一个项目中依赖于某个不同版本的jar包，就可能会出现这个问题。

### Java Doc中对 ClassLoader类的部分描述

支持并发类加载的加载器叫做 parallel capable 加载器，它们被要求在它们的类在初始化时调用ClassLoader.registerAsParallelCapable方法来注册自己，以表示它们可以做并发加载。ClassLoader类默认被注册为Parallel Capable，即可以做并发加载。但是，ClassLoader的子类如果要成为Parallel Capable，依然需要注册自己。

在委托模型不是严格的层次化的环境下，类加载器需要成为Parallel Capable的，否则会出现死锁，因为加载器的锁在类加载的过程中会被一直持有而不会释放。

正常来说, JVM以一种系统独立的方式从本地文件系统加载class文件。例如 ,在unix系统上, JVM从“CLASSPATH”这个环境变量定义的路径中加载class文件。

然而, 一些class可能不是来源于文件, 它们可能出自于其他的来源, 比如网络或者被一个应用构建出来。ClassLoader的实例方法defineClass可以将一个字节数组转换为一个Class类的实例。这个新定义的Class类的实例可以通过方法Class.newInstance创建。

由类加载器创建的对象中，它的方法和构造方法可能会引用其他的类。通过类加载器创建的方法和构造器可能引用其他的类，为了判断引用的类型, JVM调用最初创建这个类的类加载器的loadClass方法。

示例：

ClassLoader loader = new NetworkClassLoader(host, port);

Object main = loader.loadClass("Main", true).newInstance();

网络类加载器的子类必须定义findClass和loadClassData方法以从网络上加载一个类。一旦这个网络类加载器下载完用来构建类的字节码之后，它应该使用defineClass方法来创建这个从网络上下载的类的实例。

示例：

class NetworkClassLoader extends ClassLoader {

String host;

int port;

public Class findClass(String name) {

byte[] b = loadClassData(name);

return defineClass(name, b, 0, b.length);

}

private byte[] loadClassData(String name) {

// load the class data from the connection

 . . .

}

}

ClassLoader是一个没有抽象方法的抽象类。

## 线程上下文类加载器（Context ClassLoader）

### 当前类加载器（Current ClassLoader）

每个类都会使用自己的类加载器（即加载自身的类加载器）来加载其他类（指的是其所依赖的类），如果ClassX引用了ClassY，那么ClassX的类加载器就会去加载ClassY（前提是ClassY尚未被加载）。

线程上下文类加载器是从JDK1.2开始引入的，类Thread中的getContextClassLoader()与setContextClassLoader(ClassLoader c1)分别用来获取和设置上下文类加载器。

如果没有通过setContextClassLoader(ClassLoader c1)进行设置的话，线程将继承其父线程的上下文类加载器。**Java应用运行时的初始线程的上下文类加载器是系统类加载器（通过Launcher类的代码中的构造方法Thread.currentThread().setContextClassLoader(this.loader)将系统类加载器设置为当前线程的上下文类加载器）**。在线程中运行的代码可以通过上下文类加载器来加载类与资源。

示例代码可参考MyTest24。

### 线程类上下文类加载器的重要性：

父ClassLoader可以使用当前线程Thread.currentThread().getContextClassLoader()所指定的ClassLoader加载的类。这就改变了父类加载器不能使用子类加载器加载的类的情况，即改变了双亲委托模型。

线程上下文类加载器就是当前线程的Current ClassLoader（当前类加载器）。

在双亲委托模型下，类加载器是由下至上的，即下层的类加载器会委托上层进行加载。但是对于SPI来说，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来加载的，而这些接口的实现却来自于不同的jar包（厂商提供），Java的启动类加载器是不会加载其他来源的jar包，这样，传统的双亲委托模型就无法满足SPI的要求。而通过给当前线程设置上下文类加载器，就可以由设置的上下文类加载器来实现对于接口实现类的加载。

### 线程上下文类加载器的一般使用模式

1. 获取

2. 使用

3. 还原

代码示例：

|  |
| --- |
| ClassLoader classloader = Thread.currentThread.getContextClassLoader(); // 获取  try {  Thread.currentThread.setContextClassLoader(targetTccl); // 使用  myMethod(); // myMethod方法里调用了获取到的当前线程的上下文类加载器做某些事情  } finally {  Thread.currentThread.setContextClassLoader(classloader); // 还原  } |

如果一个类由类加载器A加载，那么当这个类的依赖类之前没有被加载过的话，依赖类也是由相同的类加载器加载的。

ContextClassLoader的作用就是为了破坏Java的类加载委托机制。

当高层提供了同一的接口让低层去实现，同时又要在高层加载（或实例化）低层的类时（如SPI机制），就必须要通过线程上下文类加载器来帮助高层的ClassLoader找到并加载该类

## 类的加载、连接与初始化

1. 加载：查找并加载类的二进制数据

2. 连接：

验证：确保被加载的类的正确性

准备：为类的**静态变量（类的static变量）**分配内存，并将其初始化为**默认值（不是初始值）**

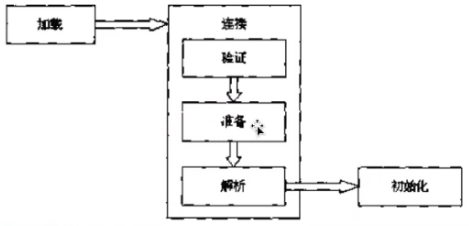
解析：**把类常量池当中的符号引用（即间接引用）转换为直接引用**

**符号引用**以一组符号来描述所引用的目标。符号引用可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可，符号引用和虚拟机的布局无关。有人理解为：在编译的时候一个每个java类都会被编译成一个class文件，但在编译的时候虚拟机并不知道所引用类的地址，多以就用符号引用来代替，而在这个解析阶段就是为了把这个符号引用转化成为真正的地址的阶段。**直接引用 ：**直接引用和虚拟机的布局是相关的，不同的虚拟机对于相同的符号引用所翻译出来的直接引用一般是不同的。如果有了直接引用，那么直接引用的目标一定被加载到了内存中。

3. **初始化：为类的静态变量赋予正确的初始值**

类的使用：调用类的静态方法、实例方法等

类的卸载：类结构从内存中卸载后，这个类就不能使用了



Java 程序对类的使用方式分为两种：主动使用、被动使用

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**”时才初始化它们（即**Java类或接口只会被初始化一次**）

主动使用（七种）：

1. 创建类的实例

2. 访问某个类或接口的静态**变量**（getstatic），或者对该静态**变量**赋值（putstatic），也就是读、写类的静态**变量**。如果通过子类去访问父类的静态变量，子类不会被主动使用。

*\* 常量在编译阶段会存入到调用这个常量的方法所在的类的常量池中*

*\* 本质上，调用类并没有直接引用到定义常量的类，因此并不会触发定义常量的类的初始化*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest2.java

3. 调用类的静态方法（invokestatic）。如果通过子类去访问父类的静态方法，子类不会被主动使用。

4. 反射（如通过Class.forName("com.test.Test") 获取一个类的Class对象）

5. 初始化一个类的子类。初始化子类时，也表示对这个子类的父类的主动使用

6. Java虚拟机启动时被标明为启动类的类（Java Test）

7. JDK1.7开始提供的动态语言支持：（纯Java程序用得很少）

Java.lang.invoke.MethodHandle实例的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic句柄对应的类没有初始化，则初始化

被动使用：

除了以上主动使用的七种情况以外，其他使用Java类的方式都被看作是对类的**被动使用，都不会导致类的初始化。**

至于被动使用是否会导致类的**加载**和**连接**过程，没有明确规定。

*对于静态字段、静态方法来说，只有直接定义了该字段、方法的类才会被初始化*

*当一个类在初始化时，要求其父类全部都已经初始化完毕了*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest1

可以使用 -XX:+TraceClassLoading，用于追踪类的加载信息并打印出来。结果如

[Opened C:\Program Files\Java\jdk1.8.0\_152\jre\lib\rt.jar]

[Loaded java.lang.Object from C:\Program Files\Java\jdk1.8.0\_152\jre\lib\rt.jar]

...

[Loaded zy.jvm.classloader.MyTest1 from file:/D:/gitRepo/jvm\_lecture/build/classes/java/main/]

...

*\* 类的静态常量在编译阶段会存入到调用这个常量的方法所在的类的常量池中  
\* 本质上，调用类并没有直接引用到定义常量的类，因此并不会触发定义常量的类的初始化*

即使在执行调用类的这个方法之前删掉所使用静态常量的类的.class文件，调用类依然可以正确执行。

*\* 当一个常量的值并非编译期间可以确定的（即运行期间才能确定），那么其值就不会被放到调用类的常量池中，  
\* 这时在程序运行时，会导致主动使用这个常量所在的类，显然会导致这个类被初始化*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest3

*\* 对于数组实例来说，其类型是由JVM在运行期动态生成的，表示为 [Lzy.jvm.classloader.MyParent4 这种形式。  
\* 动态生成的类型，其父类型就是Object。  
\* 对于数组来说，JavaDoc经常将构成数组的元素称为Component，实际上就是数组降低一个维度后的类型。  
\* 数组的声明不会导致主动使用元素类的初始化。*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest4

*\* 当一个接口在初始化时，并不要求其父接口都完成了初始化。  
\*  
\* 在使用 接口的、在编译期内确定值的 静态常量时，不要求初始化这个接口及其父接口。  
\* 在使用 接口的、在运行期内确定值的 静态常量时，要求初始化这个接口及其父接口。  
\* 对于接口来说，它只能定义静态常量。*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest5

*在类的 初始化阶段，类的静态变量赋值语句和构造方法 按代码编写顺序进行执行*

示例详见 jvm\_lecture 工程的 zy.jvm.classloader.MyTest6

### 类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在内存中创建一个java.lang.Class对象（规范并未说明Class对象位于哪里，HotSpot虚拟机将其放在了方法区中）用来封装类在方法区内的数据结构。

一个类不管生成了多少实例，它们对应的java.lang.Class对象只有一份。

加载 .class 文件的方式：

1. 从本地系统中加载；

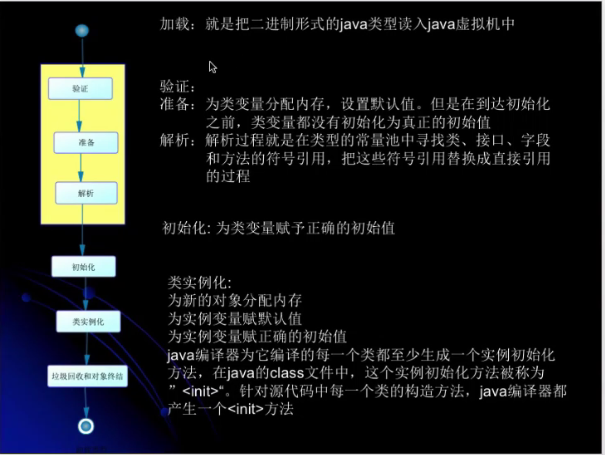
2. 通过网络下载 .class 文件；

3. 从zip、jar等归档文件中加载 .class 文件；

4. 从专有数据库中提取 .class 文件；（用得很少）

5. **将Java源文件动态编译为 .class 文件**

动态代理时、JSP页面被执行（JSP页面转换为Servlet Java代码）时，会出现第5种情况。



**类的加载的最终产品是位于内存中的Class对象。**

Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

**有两种类型的类加载器**

1. Java虚拟机自带的加载器

根类加载器（Bootstrap）

扩展类加载器（Extension）

系统（应用）类加载器（System） AppClassLoader

2. 用户自定义的类加载器

java.lang.ClassLoader的子类

用户可以定制类的加载方式

从表象上看这些类加载器是继承关系，但实际上是包含关系，即后者包含前者

根类加载器 🡨 扩展类加载器 🡨 系统类加载器 🡨 用户自定义加载器

**类加载器并不需要等到某个类被“首次主动使用”时再加载它。**

详见jvm\_lecture/zy.jvm.classloader.MyTest1，zy.jvm.classloader.MyChild1虽然没有被首次主动使用，但是也被加载到JVM中了。

[Loaded zy.jvm.classloader.MyParent1 from file:/D:/gitRepo/jvm\_lecture/build/classes/java/main/]

[Loaded zy.jvm.classloader.MyChild1 from file:/D:/gitRepo/jvm\_lecture/build/classes/java/main/]

JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在**程序首次主动使用**该类时才报告错误（**LinkageError错误**）。

如果这个类一直没有被程序主动使用，那么**类加载器就不会报告错误**

**注意：**静态内部类在被调用之前，不会被加载到内存中。详见zy.jvm.classloader.MyTest22

### 类的验证

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

类的验证的内容主要包括：

类文件的结构检查；语义检查；字节码验证；二进制兼容性的验证

### 类的准备

在准备阶段，Java虚拟机为类的静态变量分配内存，并设置默认的初始值。如：为int型的静态变量分配4个字节的内存空间，且赋予默认值0；为long型的静态变量分配8个字节的内存空间，且赋予默认值0。

### 类的初始化

在初始化阶段，Java虚拟机执行类的初始化语句，为类的静态变量赋予初始值。在程序中，静态变量的初始化有两种途径：

1. 在静态变量的声明处进行初始化。 static int a=1;

2. 在静态代码块中进行初始化。 public static long b; static{ b=2; }

**静态变量的声明语句，以及静态代码块都被看做类的初始化语句**，Java虚拟机会按照初始化语句在类文件中的先后顺序来一次执行它们。如：static int a=1; static{ a=4; }，类被初始化后，a的取值为4。

**类的初始化步骤：**

假如这个类还没有被加载和连接，那就先进行加载和连接。

假如类存在直接父类，并且这个父类还没有被初始化，那就先初始化直接父类。

假如类中存在初始化语句，那就依次执行这些初始化语句。

#### 类的初始化时机

主动使用的七种情况，详见[主动使用](#主动使用)

当Java虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则并不适用于接口：

在初始化一个类时，并不会先初始化它所实现的接口。

在初始化一个接口时，并不会先初始化它的父接口。

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。

只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用。

调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。

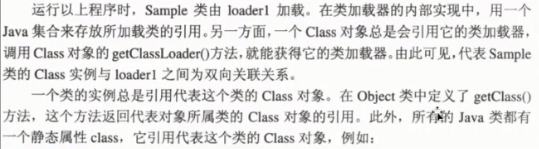
### 类的卸载

当MySample类被加载、连接和初始化后，它的生命周期就开始了。当代表MySample类的Class对象不再被引用，即不可触及时，Class对象就会结束生命周期，MySample类在方法区内的数据也会被卸载，从而结束MySample类的生命周期。

**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。**

由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。Java虚拟机自带的类加载器包括根、扩展、系统三种类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。

用户自定义的类加载器所加载的类是可以被卸载的。



### JVM选项设置

-XX:+<option> 表示开启默认为关闭的option

-XX:-<option> 表示关闭默认为开启的option

-XX:<option>=<value> 表示将option的值设置为value

### Java虚拟机监控工具

jvisualvm

### 助记符

反编译class文件命令：javap -c <class根路径>/<class名>

ldc——表示将int、float或是String类型的常量值从常量池中推送至栈顶

bipush——表示将单字节（-128 ~ 127）的常量值推送至栈顶

sipush——表示将一个短整型常量值（-32768 ~ 32767）推送至栈顶

iconst\_1——表示将int类型1推送至栈顶（iconst\_m1 (-1)，iconst\_0，iconst\_1 ~ iconst\_5）

助记符 本身 是由java的一些内部类实现的。

anewarray——表示创建一个**引用**类型的（如类、接口、数组）数组，并将其引用值压入栈顶

newarray——表示创建一个指定的**原始**类型的（如int、float、char等）数组，并将其引用值压入栈顶

# Java字节码

1. 使用 javap -verbose 命令分析一个字节码文件时，将会分析该字节码的魔数、版本号、常量池、类信息、类的构造方法、类中的方法信息、类变量与成员变量等信息。

javap -p <编译后的java类> 可以显示私有方法和私有成员变量。

IDE工具：jclasslib用来查看java字节码文件，Hex Fiend用来查看class文件的16进制内容

2. 魔数：所有的 .class 字节码文件的**前4个字节**都是魔数，魔数值为固定值：0xCAFEBABE

3. 魔数之后的**4个字节**为版本信息，前两个字节表示minor version（次版本号），后两个字节表示major version（主版本号）。例如，版本号为00 00 00 34，换算成十进制表示此版本号为0，主版本号为52，即 1.8.0。可以通过 java -version 命令来验证这一点。

4. 常量池（constant pool）：紧接着主版本号之后的就是常量池入口。一个Java类中定义的很多信息都是由常量池来维护和描述的，可以将常量池看作是Class文件的资源仓库，比如说Java类中定义的方法与变量信息，都是存储在常量池中。

常量池中主要存储两类常量：字面量与符号引用。

字面量如文本字符串，Java中声明为final的常量值等；符号引用如类和接口的全局限定名，字段的名称和描述符，方法的名称和描述符等。

5. 常量池的总体结构：

Java类所对应的常量池主要由常量池数量与常量池数组这两部分共同构成。常量池数量紧跟在主版本号后面，占据**2个字节**；常量池数组则紧跟在常量池数量之后。常量池数组与一般的数组不同的是，常量池数组中不同的元素的类型、结构都是不同的，长度也就不同；但是，每一种元素的第一个数据都是一个u1类型，该字节是个**标志位（tag）**，**占据1个字节**。JVM在解析常量池时，会根据这个u1类型来获取元素的具体类型。

值得注意的是，常量池数组中元素的个数 = 常量池数 - 1，其中 0 暂时不使用，目的是满足某些常量池索引值的数据在特定情况下需要表达“不引用任何一个常量池”的含义；根本原因在于索引为0也是一个常量（保留常量），只不过它不位于常量表中，这个常量就对应null值；所以，常量池的索引从1而非0开始。

6. 在JVM规范中，每个变量/字段都有描述信息，描述信息主要的作用是描述字段的数据类型、方法的参数列表（包括数量、类型与顺序）与返回值。

根据描述符规则，基本数据类型和代表无返回值的void类型都用一个大写字符来表示，对象类型则使用字符L加对象的全限定名称来表示。为了压缩字节码文件的体积，对于基本数据类型，JVM都只使用一个大写字母来表示，如下所示：

B - byte，C - char，D - double，F - float，I - int，J - long，Z - boolean，V - void，

L - 对象类型（例如Ljava/lang/String;）

7. 对于数组类型来说，每一个维度使用一个前置的方括号“[”来表示，如int[]被记录为“[I”、String[][]被记录为[[Ljava/lang/String;

8. 用描述符描述方法时，按照先参数列表，后返回值的顺序来描述，参数列表按照参数的严格顺序放在一组()之内，如方法：String getRealnamebyIdAndNickname(int id, String name)的描述符为 (I, Ljava/lang/String;)Ljava/lang/String;

## Java字节码结构

**Class字节码中有两种数据类型：**

字节数据直接量：

这是基本的数据类型。共细分为u1、u2、u4、u8四种，分别代表连续的1个字节、2个字节、4个字节、8个字节组成的整体数据。

表（数组）：

表是由多个基本数据或其他表，按照既定顺序组成的大的数据集合。表是有结构的，它的结构体现在：组成表的成分所在的位置和顺序都是已经严格定义好的。



上面的表中描述了11种数据类型的结构，其实在jdk1.7之后又增加了3种：

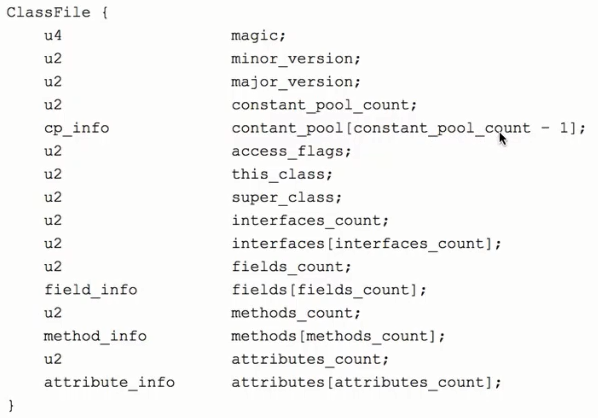
CONSTANT\_MethodHandle\_info、CONSTANT\_MethodType\_info、CONSTANT\_InvokeDynamic\_info，它们对应的是动态调用。这样一共是14种。

**字节码的整体结构**

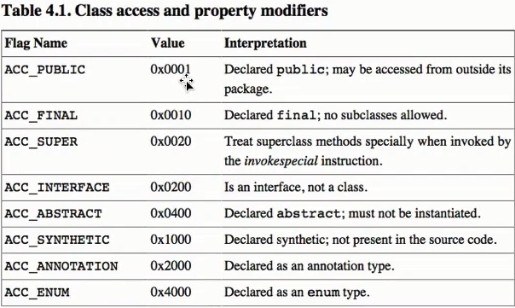


**完整的字节码结构**





## Access\_Flag访问标志



|  |
| --- |
| ACC\_PRIVATE 0x0002 Declared private |
| ACC\_STATIC 0x0008 Declared static |

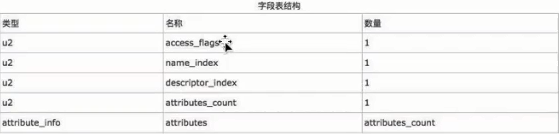
0x0021：是0x0020和0x0001的并集，表示ACC\_PUBLIC与ACC\_SUPER

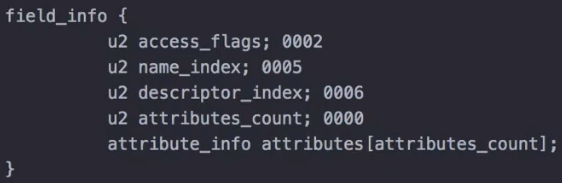
## 字段表集合

字段表用于描述类和接口中声明的变量。这里的字段包含了**类级别变量以及实例变量**，但是不包括方法内部声明的局部变量。

fileds\_count: u2

fileds[fileds\_count]：



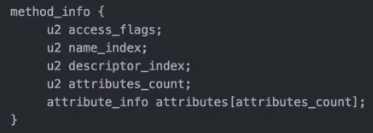


## 方法表

methords\_count: u2

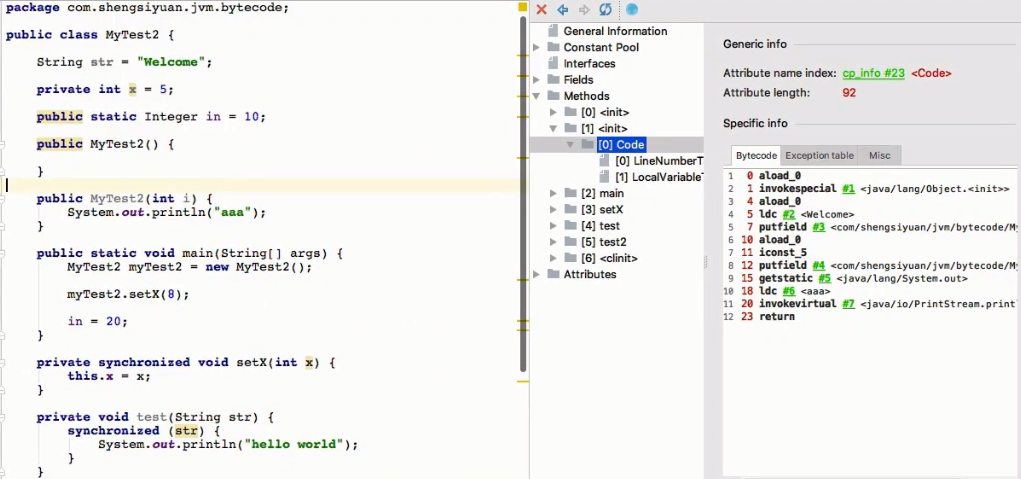
methords[methords\_count]：





编译器会自动生成<init>方法作为实例的构造方法，如果类中有静态成员变量，编译器则会自动生成<cinit>方法作为类的Class对象的构造方法。

<init>方法中，会对类的成员变量进行赋值。如果在类中自定义了无参或有参的多个构造方法，对非静态成员变量的赋值指令将会存在于为多个构造方法生成的<init>方法中。所以，无论调用哪个构造方法，为非静态成员变量赋值的语句都会被执行到。



与<init>类似，<cinit>方法中会有对静态成员变量赋值的指令，如果有多个static代码块，则对静态成员变量赋值的指令会存在于多个对应于static代码块的<cinit>方法中。

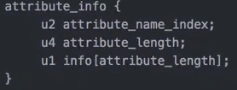
### Java的字节码文件可以不包含<init>方法

通过javac编译java类文件生成的字节码文件会包含至少一个<init>方法，但通过其他语言开发的代码经编译后生成的Java字节码文件中可能不包含<init>方法，例如使用Kotlin语言开发的包级别的属性或函数的源码文件，编译生成的Java字节码文件中不包含<init>方法。

字节码规范没有要求字节码文件必须包含<init>方法。

### 方法的属性结构：

方法中每个属性都是一个attribute\_info结构

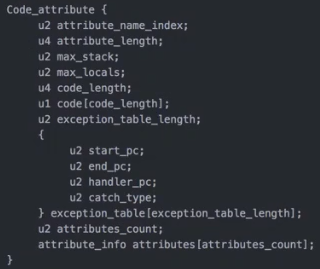


JVM预定义了部分attribute，但是编译器自己也可以实现自己的attribute写入class文件里，供运行时使用。

不同的attribute通过attribute\_name\_index来区分

#### Code结构

Code attribute的作用是保存该方法的结构，如所对应的字节码



attribute\_length表示attribute所包含的字节数，不包含attribute\_name\_index和attribute\_length字段。

max\_stack表示这个方法运行的任何时刻所能达到的操作数栈的最大深度。

max\_locals表示方法执行期间创建的局部变量的数目，包含用来表示传入的参数的局部变量。

code\_length表示该方法所包含的字节码的字节数及具体的指令码。

具体字节码即是该方法被调用时，虚拟机所执行的字节码。

exception\_table，这里存放的是处理异常的信息。

每个exception\_table表项由start\_pc，end\_pc，handler\_pc，catch\_type组成。

#### 使用javap -verbose命令查看一个类的字节码文件





stack=3：表示这个方法运行的任何时刻所能达到的操作数栈的最大深度是3。

locals=4：表示这个方法执行期间创建的局部变量数目是4。第一个参数是this，第二个是is，第三个是serverSocket，第四个是3个catch中的其中一个变量。

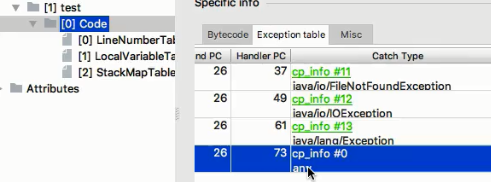
args\_size=1：表示这个方法本身所接收的参数数量是1。对于java类中的每个实例方法（非静态的），它在字节码中的参数个数总是比java代码中多一个（this），它位于方法的第一个参数位置处；这样，我们就可以在Java的实例方法中使用this来去访问当前对象的属性以及其他方法。这个操作是在编译期间完成的，即由javac编译器在编译的时候将对this的访问转化为对一个普通实例方法参数的访问，接下来在运行期间，由JVM在调用实例方法时，自动向实例方法传入该this参数。所以，在实例方法的局部变量表中，至少会有一个指向当前对象的局部变量。

##### exception\_table结构：

start\_pc和end\_pc表示在code数组中的从start\_pc到end\_pc处（包含start\_pc，不包含end\_pc）的指令抛出的异常会由这个表项来处理。

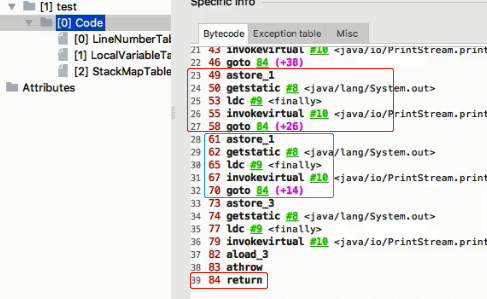
handler\_pc表示处理异常的代码的开始处。catch\_typ表示会被处理的异常类型，它指向常量池里的一个异常类。当catch\_type为0时，表示处理所有的异常。

如下图，除了代码中写入的三种可能捕获的异常，在字节码中，还会多一个处于保留位置（#0）的异常类型any，它位于在最后，表示除了代码中的异常以外，其他情况都由any处理。



异常的处理，在字节码中是使用goto指令进行指令的跳转实现的。

在字节码中，finally语句块内的代码都会转化为相应的指令存在于try语句块和每个异常处理的catch语句块所转化的指令的后面，并且在finally的最后一条代码指令结束后都会goto到return指令所在的偏移位置。如下图



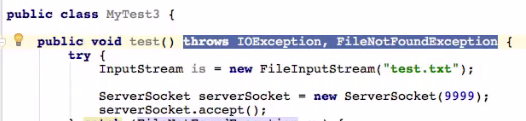
##### Java字节码对于异常的处理方式：

1. 统一采用异常表的方式来对异常进行处理；

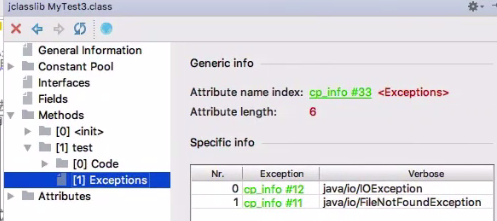
2. 在jdk1.4.2之前的版本中，并不是使用异常表的方式来对异常进行处理的，而是采用特定的指令方式；

3. 当异常处理存在finally语句块时，现代化的JVM采取的处理方式是将finally语句块的字节码拼接到每一个catch块后面，换句话说，程序中存在多少个catch块，就会在每一个catch块后面重复多少个finally语句块的字节码。所以在字节码中finally语句块的字节码会有N+2份，N是java代码中的catch块数量，1个位于try块后面，1个位于字节码的any类型异常后面。

##### 方法抛出异常时Java字节码的处理方式



如图，test方法抛出 IOException和FileNotFoundException，字节码如下图所示

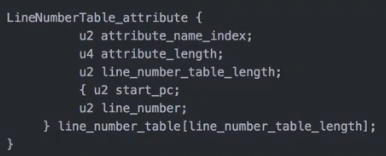


在字节码中，要抛出异常的信息位于与Code同级的Exceptions中。

##### 方法的附加属性

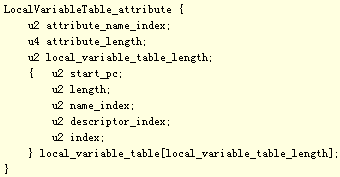
LineNumberTable：这个属性用来表示code数组中的字节码和Java代码行数之间的关系。这个属性可以用来在调试的时候定位代码执行的行数。

LineNumberTable的结构

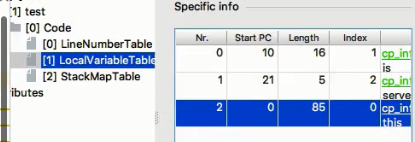


LocalVariableTable：局部变量表，其结构与LineNumberTable的结构相似。

在字节码中，this作为第一个参数被隐式地传入到实例方法中。



下图中，局部变量表中各个变量的Length，指的是从StartPC开始的有效范围（可见范围）。



例如，在字节码中，局部变量is从Code的10开始，到26结束。

## synchronized修饰符对字节码的影响：

**synchronized修饰方法时**，方法的字节码中只含有ACC\_SYNCHRONIZED修饰符，而方法体的字节码中不含有synchronized相关的操作指令。

**synchronized修饰一个对象时**，会在方法体的字节码中产生monitorenter和monitorexit指令，用来表示程序的进入和退出被同步的方法体。在异常发生的情况下，也要有monitorexit指令。所以monitorenter有一个，而monitorexit会有多个。

**synchronized修饰静态方法时**，方法的字节码中只含有ACC\_SYNCHRONIZED修饰符，而方法体的字节码中不含有synchronized相关的操作指令。这时，synchronized是对这个类的Class对象加锁。

## 栈帧

栈帧本身一种用于帮助虚拟机执行方法调用与方法执行的数据结构。

栈帧归属于一个个线程，每个线程只会独有一个栈帧，同一个栈帧不存在并发或同步的概念。

栈帧本身是一种数据结构，封装了方法的局部变量表、动态链接信息、方法的返回地址以及操作数栈帧等信息。

有些符号引用是在类加载阶段或是第一次使用时就会转换为直接引用，这种转换就叫做静态解析；另外一些符号引用则是在每次运行期转换为直接引用，这种转换叫做动态链接，这体现为Java的多态性。

## JVM中是如何执行方法调用的

### Java代码中new作用：

1. 为对象在内存特别是堆上开辟一个内存空间

2. 去执行其构造方法

3. 将构造方法执行完后所返回的这个在堆上对象的引用返回

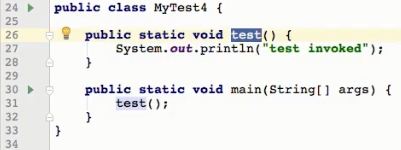
这三条作用对应于字节码中的4条指令，new、dup、invokespecial、astore\_n（n表示局部变量的序号）

### 方法调用的字节码指令主要有五种情况

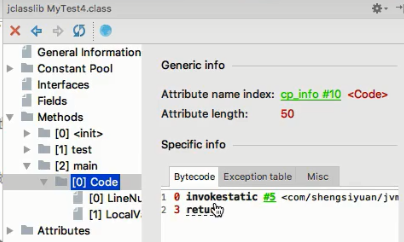
这五种情况分别对应不同的字节码指令

#### **1. invokeinterface：**

调用接口中的方法，实际上是在运行期决定的，决定到底调用实现接口的哪个对象的特定方法。



以上示例代码的字节码内容如下，



在main调用静态方法test时，使用了字节码指令invokestatic。

#### **2. invokestatic：**

调用静态方法

#### **3. invokespecial：**

调用自己的私有方法、构造方法（<init>）以及父类的方法。

#### **4. invokevirtual：**

调用虚方法，它存在运行期动态查找的过程。

#### **5. invokedynamic：**

动态调用方法（如JavaScript的方法）。

### 静态解析的4种情形：

invokestatic和invokespecial这两个字节码指令所调用的方法都是能**在类的解析阶段（属于类的连接阶段）中就能确定的**，那么符合这样类型的方法有4类（父类方法、静态方法、private方法、当前类自己的构造方法）。这4类方法在类的解析阶段就能唯一地确定自己的调用过程。所以在类加载的阶段就会把相应的符合引用转换为直接引用。

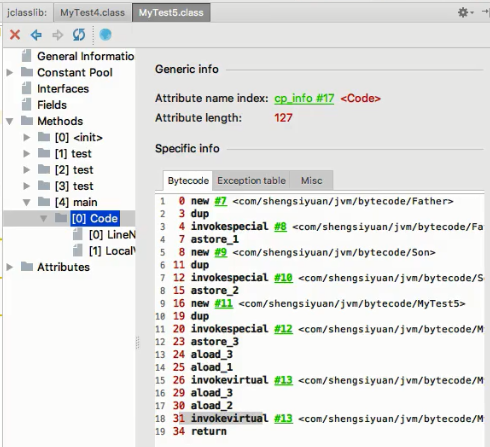
有些符号引用是在类加载阶段或是第一次使用时就会转换为直接引用，这种转换就叫做静态解析。

**4种情形是：静态方法、父类方法、构造方法、私有方法（无法被重写）。**

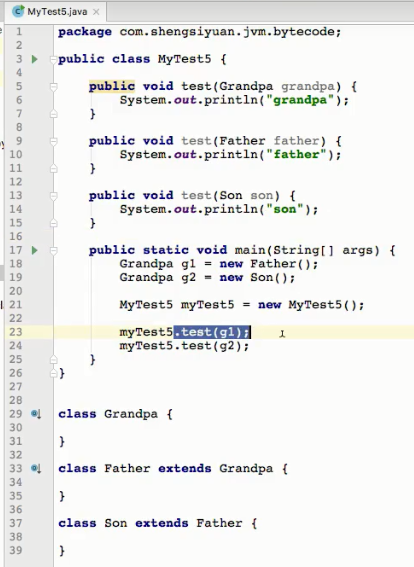
因为公有方法可以被Override，所以公有方法的调用不属于静态解析。

以上4类方法称作**非虚方法**，它们是在类加载阶段就可以将符合引用转换为直接引用的。

### 方法的静态分派



上图是下图的字节码内容



代码示例如上。

|  |
| --- |
| Grandpa g1 = new Father(); |

g1的静态类型是Grandpa，而g1的实际类型（真正指向的类型）是Father，并且随后可把Son类型的实例赋给g1，也可以把g1指向的实例强制转换为其子类型Father或Son的实例，但这些都不会改变g1本身的静态类型是不会变化的，依然是Grandpa。

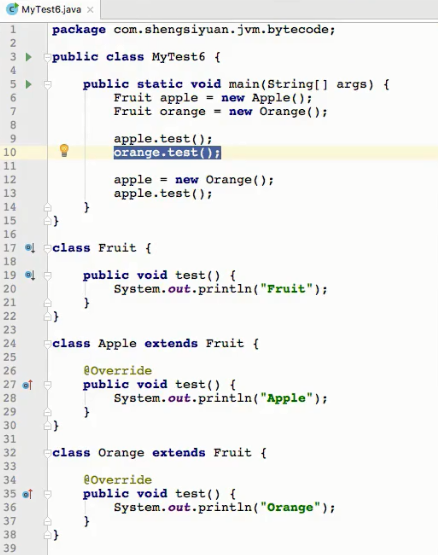
**结论：**

变量的静态类型是不会发生变化的，而变量的实际类型则是可以发生变化的（多态的一种体现），实际类型是在运行期方可确定。

对于JVM来说，方法重载是一种静态的行为，编译期可以完全确定。当传入参数到重载的方法中，JVM是根据参数的静态类型去匹配重载的方法（方法名、参数个数、参数类型这三者可以用来区分一个方法的声明）。

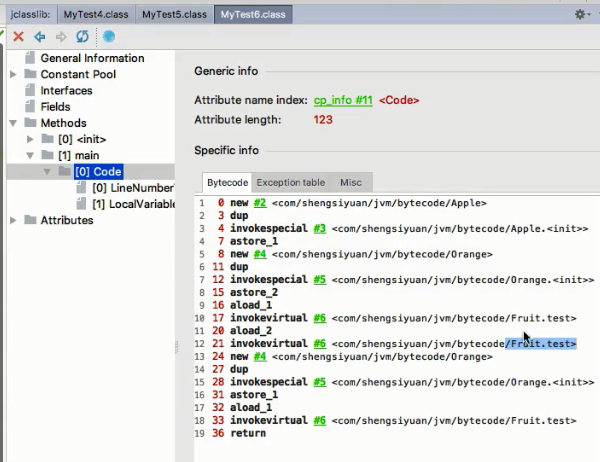
### 方法的动态分派

以如下代码为例



其执行结果为 Apple Orange Orange

其JVM字节码如下图所示



方法的分派涉及到一个重要概念：方法接收者，即方法是由哪个实例调用的。

从字节码来看，test方法都由Fruit实例调用，但这还涉及invokevirtual指令的多态查找流程。首先，到操作数的栈顶去寻找到栈顶元素所指向对象的实际类型；然后，如果在这个实际类型中，寻找到了与常量池中的描述符和名称都相同的方法，且具备相应的访问权限，它就返回目标方法的直接引用；如果没有在当前类型中找到，则会在其父类中继续寻找，一直没找到则抛出异常。

在上图示例代码字节码中，test方法的符号引用虽然指向了Fruit，但是栈顶元素的实际类型Apple和Orange中都有与要调用（也可称作要接受）的test方法的名称和描述符一致的test方法，所以，Apple和Orange类中的test方法才会被调用。

即便是对字节码中完全相同的符号引用，在运行期也会被JVM解析成不同的直接引用。这就是Java方法多态性的及其重要的一个表现。

**比较方法重载（overload）和方法重写（overwrite），可以得出这样的结论：**

方法重载是静态的，是编译期行为；方法重写是动态的，是运行期行为。

#### 虚方法表

针对于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（virtual method table, 被称作为vtable）

针对于invokeinterface指令来说，虚拟机会建立一个叫做接口方法表的数据结构（interface method table，即itable）。

虚方法表中的每个方法都是方法调用的一个入口地址，这个入口地址在程序执行时去调用的。如果子类继承了父类方法而没有重写，那么子类的虚方法表不会把这个父类方法的描述信息再复制一遍，而是直接指向了父类的这个方法的入口地址。如果子类重写了父类的方法，那么子类的这个方法与父类这个方法的索引是一样的，这样会提升查找效率。

虚方法表是在类加载的连接阶段创建完成的。

### JVM指令集

#### JVM执行Java代码的方式

现代JVM在执行Java代码时，通常都会将**解释执行与编译执行二者结合**起来进行。

解释执行：通过解释器来读取字节码，遇到相应的指令就去执行该指令。

编译执行：通过即时编译器（Just In Time，JIT）将字节码转换为本地机器码来执行；现代JVM会根据代码热点来生成相应的本地机器码。

#### 基于栈的指令集与基于寄存器的指令集之间的关系

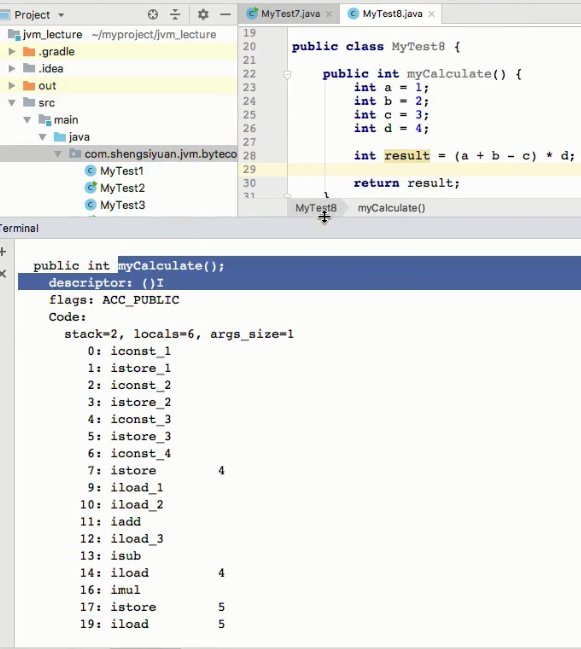
1. JVM执行指令时所采取的方式**基于栈的指令集**

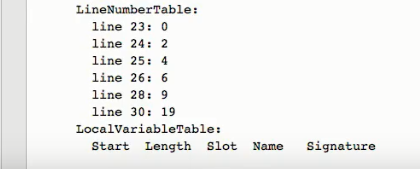
2. 基于栈的指令集主要的操作有入栈与出栈两种。

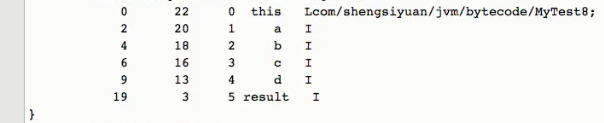
3. 基于栈的指令集的**优势**在于它可以在**不同的平台之间移植**，而基于寄存器的指令集是与硬件架构紧密关联的，无法做到可移植。

4. 基于栈的指令集的缺点在于完成相同的操作，指令数量通常要比基于寄存器的指令集数量要多；基于栈的指令集是在内存中完成操作的，中间结果也要入栈和出栈，而基于寄存器的指令集是直接由CPU来执行的，它是在高速缓冲区中进行执行的，速度要快很多。虽然虚拟机可以采用一些优化手段，但总体来说，基于栈的指令集的执行速度要慢一些。

zy.jvm.bytecode.MyTest8 用来查看基于栈的指令集







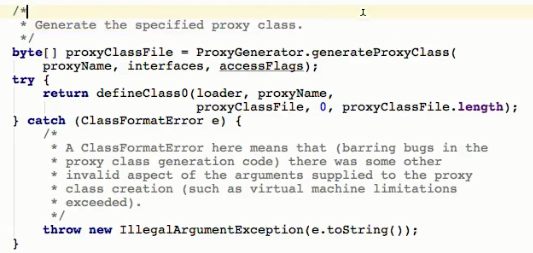
## 透过字节码生成审视Java动态代理运作机制

java.lang.reflect.Proxy.ProxyClassFactory 用来生成、定义并返回指定了ClassLoader和接口数组的代理类。

在这个类的apply方法中，会生成一个代理类的字节数组，如下图所示。

其实，动态生成字节码就是**根据给定的代理的名字**、**所实现接口的数组**、**访问标识**动态地创建一堆字节数组。

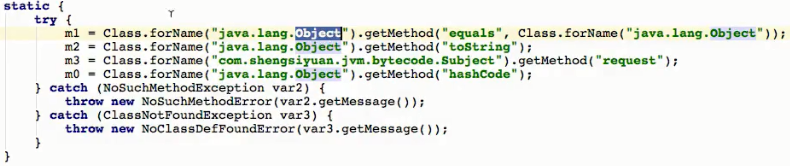
动态字节码的字节数组是由sun.misc.ProxyGenerator的generateProxyClass方法生成的，将**sun.misc.ProxyGenerator.saveGeneratedFiles属性设置为true**，就可以将动态生成的字节码写到磁盘上。



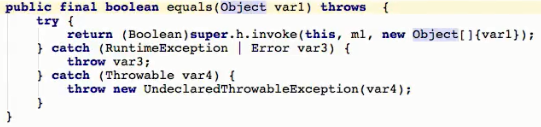
对于java.lang.Object类中的hashCode、equals和toString方法，它们会被编码并被传入Proxy.newProxyInstance方法的InvocationHandler实例分派，对这三个方法的调用会由动态代理执行。而Object类中其他方法没有被代理类重写，这些方法并没有被代理类所代理，它们的实现没有改变。

#### 查看由动态代理所生成的动态类文件$Proxy0.class

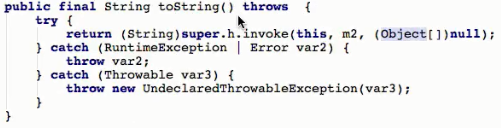
$Proxy0的静态代码块中获取到了Object类的equals、toString、hashCode方法，以及代理的Subject接口的request方法。



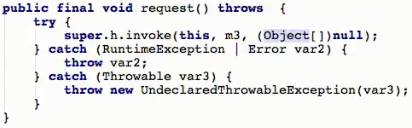
equals方法：



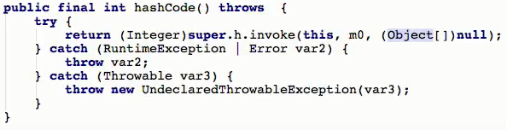
toString方法：



request方法：



hashCode方法：



#### 动态代理的优势：

可以在真实对象还不存在的情况下，就把代理的对象实例创建出来了。所以代理对象实例可以代理多种真实对象。例如SpringMVC，面向接口编程的话都会使用动态代理的方式来帮助我们生成对应代理类的实例。**动态代理非常灵活**。

其他的动态代理工具类有：cglib、javaassist