链滴

JVM 底层之类加载

作者：Giles

原文链接：https://ld246.com/article/1596649358683

来源网站：链滴

<h2 id="JVM底层之类加载">JVM 底层之类加载</h2>
<h2 id="klass模型">klass 模型</h2>
Java 的每个类，在 JVM 中，都有一个对应的 Klass 类实例与之对应，存储类的元信息如：常量、属性信息、方法信息……
看下 klass 模型类的继承结构

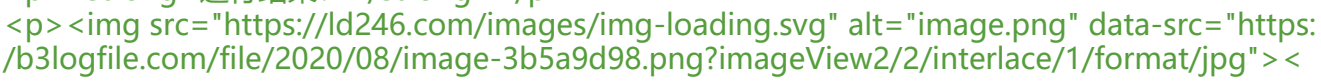
从继承关系上也能看出来，类的元信息是存储在原空间的
类加载器将.class 文件加载进系统
将.class 文件解析，生成的就是 InstanceKlass
<h3 id="MetaspaceObj">MetaspaceObj</h3>
<pre><code class="highlight-chroma">   JDK8以后类的元信息都是存储在类的元空间里的就是**MetaspaceObj** 是所有类的顶层父。
</code></pre>
<h4 id="InstanceKlass">InstanceKlass</h4>
<pre><code class="highlight-chroma">   InstanceKlass就是我们写的Java类(非数组)，InstanceKlass就是类加载器把Java文件存储到存中经过解析后生成的。

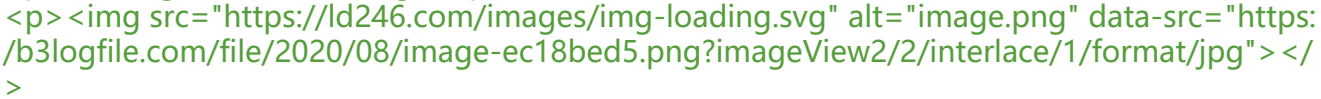
    InstanceKlass 含的一些属性：注解 _annotations、__method..
</code></pre>
普通的 Java 类在 JVM 中对应的是 instanceKlass 类的实例，再来说下它的三个子类

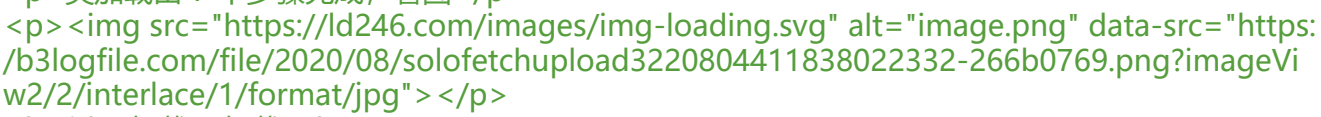
InstanceMirrorKlass：用于表示 java.lang.Class，class 对象（就是我们所说的堆区就是存储在里）Java 代码中获取到的 Class 对象，实际上就是这个 C++ 类的实例，存储在堆区，学名镜像类
InstanceRefKlass：用于表示 java/lang/ref/Reference 类的子类，（引用就是存放在这里）
InstanceClassLoaderKlass：用于遍历某个加载器加载的类

总结： 类加载器将.class 文件加载进系统，将.class 文件解析，生成的类元信息以 InstanceKlass 存储在 JVM 中
<h4 id="ArrayKlass">ArrayKlass</h4>
ArrayKlass 就是用来存储数组的元信息。
Java 的数组
<pre><code class="highlight-chroma">静态数据类型   JVM中内置的 八种数据类型

动态数组类型   运时动态生成
</code></pre>
证明： 为什么 Java 中的数组是动态生成的？
<pre><code class="highlight-chroma">public class Test_1 {
    public static void main(String[] args) {
//   System.out.printf(Test_1_B.str);
        int[] arr = new int[1];
        while (true);</code></pre>

}
}
</code></pre>
运行结果：

结果生成一个 newarray 顾名思义就是生成一个数组嘛。
我们查看字节码手册也是可以看的到的如下信息：
<table>
<thead>
<tr>
<th>指令码</th>
<th>助记符</th>
<th>说明</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbc</td>
<td>newarray</td>
<td>创建一个指定原始类型（如 int, float, char...）的数组，并将其引用值压入栈顶</td>
</tr>
</tbody>
</table>
我们再测试一个引用类型的数组：
<pre><code class="highlight-chroma">public class Test_1 {
public static void main(String[] args) {
//   System.out.printf(Test_1_B.str);
int[] arr = new int[1];

Test_1[] arr2 = new Test_1[1];
while (true);
}
}
</code></pre>
输出：

可以看出输出的为 anewarray ，字节码文档解释如下：
<table>
<thead>
<tr>
<th>指令码</th>
<th>助记符</th>
<th>说明</th>
</tr>
</thead>

<tbody>
<tr>
<td>0xbd</td>
<td>anewarray</td>
<td>创建一个引用型（如类，接口，数组）的数组，并将其引用值压入栈顶</td>
</tr>
</tbody>
</table>
以上得知什么？简单来说就是，
newarray 也就是基本数据类型，在 JVM 中的存在形式是以 TypeArrayKlass 存在的。
anewarray 也就是引用类型的数组，在 JVM 中的存在形式是以 ObjArrayKass 存在的。
<h2 id="类加载的过程">类加载的过程</h2>
类加载由 7 个步骤完成，看图

<h3 id="加载">加载</h3>

通过类的全限定名获取存储该类的 class 文件（没有指明必须从哪获取）
解析成运行时数据，即 instanceKlass 实例，存放在方法区
在堆区生成该类的 Class 对象，即 instanceMirrorKlass 实例

全限定名： 包名 + 类名
程序随便你怎么写，随便你用什么语言，只要能达到这个效果即可
就是说你可以改写 openjdk 源码，你写的程序能达到这三个效果即可
何时加载
主动使用时

new、getstatic、putstatic、invokestatic
反射
初始化一个类的子类会去加载其父类
启动类（main 函数所在类）
当使用 jdk1.7 动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结 REF_getstatic,REF_putstatic,REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进初始化，则需要先出触发其初始化

预加载：包装类、String、Thread
因为没有指明必须从哪获取 class 文件，脑洞大开的工程师们开发了这些

从压缩包中读取，如 jar、war
从网络中获取，如 Web Applet
动态生成，如动态代理、CGLIB
由其他文件生成，如 JSP
从数据库读取
从加密文件中读取

<h3 id="验证">验证</h3>

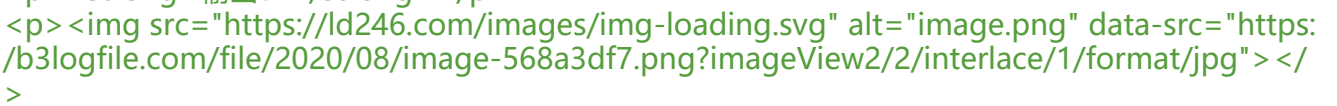
文件格式验证
元数据验证
字节码验证
符号引用验证

```
</ol>
<h3 id="准备">准备</h3>
<p>为静态变量分配内存、赋初值</p>
<p>实例变量是在创建对象的时候完成赋值的，没有赋初值一说（final 修饰）</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/solofetchupload6348652054741682805-b864f9a8.png?imageVie
2/2/interlace/1/format/jpg"></p>
<p>如果被 final 修饰，在编译的时候会给属性添加 ConstantValue 属性，准备阶段直接完成赋值，
没有赋初值这一步</p>
<p><strong>验证：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public static final int a = 10;
</span></span><span class="highlight-line"><span class="highlight-cl">public static int b
 10;
</span></span></code></pre>
<p><strong>输出：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/image-083cda90.png?imageView2/2/interlace/1/format/jpg"></
>
<h3 id="解析">解析</h3>
<p>将常量池中的符号引用转为直接引用</p>
<p>直接引用就是指向内存地址</p>
<p>间接引用指向运行时常量池</p>
<p>每个类都有一个常量池</p>
<p>class 常量池（静态的）  HSDB 常量池（动态的）</p>
<p>解析后的信息存储在 ConstantPoolCache 类实例中</p>
<ol>
<li>类或接口的解析</li>
<li>字段解析</li>
<li>方法解析</li>
<li>接口方法解析</li>
</ol>
<p>何时解析</p>
<p>思路：</p>
<ol>
<li>加载阶段解析常量池时</li>
<li>用的时候</li>
</ol>
<p>openjdk 是第二种思路，在执行特定的字节码指令之前进行解析：</p>
<p>anewarray、checkcast、getfield、getstatic、instanceof、invokedynamic、invokeinterfac
、invokespecial、invokestatic、invokevirtual、ldc、ldc_w、ldc2_w、multianewarray、new、p
tfield</p>
<h3 id="初始化">初始化</h3>
<p>执行静态代码块，完成静态变量的赋值</p>
<ol>
<li>定义一个 static 静态代码块，JVM 底层会生成一个 clinit，生成 clinit 方法</li>
<li>代码顺序和定义顺序保持一致的。</li>
</ol>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/image-4f195276.png?imageView2/2/interlace/1/format/jpg"></
>
<p>来一个小问题代码如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test_21 {
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">Test_21_A obj = Tst_21_A.getInstance();
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println Test_21_A.val1);
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println Test_21_A.val2);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_21_A {
</span></span><span class="highlight-line"><span class="highlight-cl">public static int va1;
</span></span><span class="highlight-line"><span class="highlight-cl">public static int va2 = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">public static Test_1_A instance = new Test_21_A();
</span></span><span class="highlight-line"><span class="highlight-cl">Test_21_A() {
</span></span><span class="highlight-line"><span class="highlight-cl">val1++;
</span></span><span class="highlight-line"><span class="highlight-cl">val2++;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">public static Test_1_A getInstance() {
</span></span><span class="highlight-line"><span class="highlight-cl">return instance;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>输出：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https://b3logfile.com/file/2020/08/image-12c654b9.png?imageView2/2/interlace/1/format/jpg"></p>
<p><strong>why? </strong></p>
<p>很明显：</p>
<p>val1 = 0　　val2 = 1</p>
<p>执行代码块之后各自 +1，</p>
<p>所以输出结果为 1 2</p>
<p><strong>再看一个题：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public class Test_22 {
</span></span><span class="highlight-line"><span class="highlight-cl">public static void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">Test_22_A obj = Tst_22_A.getInstance();
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println Test_22_A.val1);
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println Test_22_A.val2);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_22_A {
</span></span><span class="highlight-line"><span class="highlight-cl">public static int va1;
</span></span><span class="highlight-line"><span class="highlight-cl">public static Test_2_A instance = new Test_22_A();
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">Test_22_A() {
</span></span><span class="highlight-line"><span class="highlight-cl">val1++;
</span></span><span class="highlight-line"><span class="highlight-cl">val2++;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">public static int va
2 = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">public static Test_
2_A getInstance() {
</span></span><span class="highlight-line"><span class="highlight-cl">return instance;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
```

输出：



分析：

结合上一题这个代码会出来一个覆盖的情况，所以输出为 1 1

类加载细节

JVM 加载类是懒加载模式

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test_1 {
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.printf(
est_1_B.str);
</span></span><span class="highlight-line"><span class="highlight-cl">while (true);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_1_A {
</span></span><span class="highlight-line"><span class="highlight-cl">public static String
str = "A str";
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"A Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_1_B ext
nds Test_1_A {
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"B Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
```

输出：

证明：

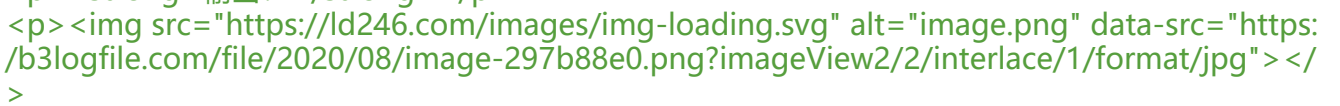
类只有在使用的时候才会加载

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test_1 {
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.printf(
ew Test_1_B().str);
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">while (true);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_1_A {
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"A Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_1_B ext
nds Test_1_A {
</span></span><span class="highlight-line"><span class="highlight-cl">public String str =
"A str";
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">    System.out.print
n("B Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">  public String str
 "A str";
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">    System.out.print
n("B Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>输出：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/file/2020/08/image-bb71e291.png?imageView2/2/interlace/1/format/jpg"><
p>
<p><strong>证明：</strong></p>
<p><strong>主动使用子类，父类也会加载。</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test_4 {
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">Test_4 arrs[] = ne
 Test_4[1];
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Test_4_A {
</span></span><span class="highlight-line"><span class="highlight-cl">static {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"Test_4_A Static Block");
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>其实没有输出，以上代码只是定义了一个数据类型。</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test_6 {
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
```

```
Test_6_A.str);
}
}
class Test_6_A {
public static final
tring str = "A Str";
static {
System.out.println
"Test_6_A Static Block");
}
}
```

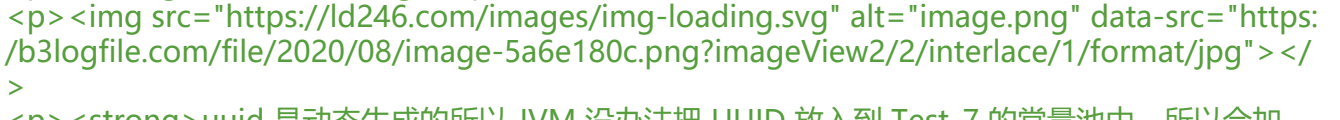
输出：



因为我们使用 final 修饰它定义的是常量，JVM 将常量 str 写入了 Test_6 的常量池中

```
public class Test_7 {
public static void
main(String[] args) {
System.out.println
Test_7_A.uuid);
}
}
class Test_7_A {
public static final
tring uuid = UUID.randomUUID().toString();
static {
System.out.println
"Test_7_A Static Block");
}
}
```

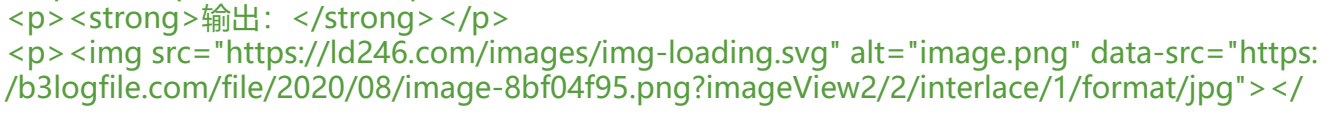
输出：



uuid 是动态生成的所以 JVM 没办法把 UUID 放入到 Test_7 的常量池中，所以会加，它生成的是动态代码段

```
public class Test_8 {
static {
System.out.println
"Test_8 Static Block");
}
public static void
main(String[] args) throws ClassNotFoundException {
Class
```

输出：



>
这就是一个反射，读取静态属性时，反射也会读取
<h2 id="读取静态字段的实现原理">读取静态字段的实现原理</h2>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Test_1 {
public static void
main(String[] args) {
System.out.printf(
ew Test_1_B().str);
while (true);
}
}
class Test_1_A {
public String str =
"A str";
static {
System.out.println
"A Static Block");
}
}
class Test_1_B ext
nds Test_1_A {
static {
System.out.println
"B Static Block");
}
}
</code></pre>
思路：
<pre><code class="highlight-chroma"><span class="highlight
cl">1. 先去Test_1_B的镜像类中去取，如果有直接返回，如果没有，会沿着继承链将请求往上抛。这
算法的性能随着继承链的death而上升，算法复杂度为0（0）；

2. 借助另外的数据
构实现，使用K-V的格式存储，查询性能为O(1)
</code></pre>
Hotspot 就是使用的第二种方式，借助另外的数据结构 ConstantPoolCache，常量池类 Consta
tPool 中有个属性_cache 指向了这个结构。每一条数据对应一个类 ConstantPoolCacheEntry。
ConstantPoolCache 主要用于存储某些字节码指令所需的解析（resolve）好的常量项，例如给[
et|put]static、[get|put]field、invoke[static|special|virtual|interface|dynamic]等指令对应的常量
项用。
！！！！撒花！！！！