老大难的 Java ClassLoader 再不理解就老了

老钱
Good news everyone!

已关注

383 人赞了该文章

ClassLoader 是 Java 届最为神秘的技术之一，无数人被它伤透了脑筋，摸不清门道究竟在哪里。网上的文章也是一篇又一篇，经过本人的亲自鉴定，绝大部分内容都是在误导别人。本文我带读者彻底吃透 ClassLoader，以后其它的相关文章你们可以不必再细看了。

ClassLoader 做什么的？

顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码可以来自于磁盘文件 *.class，也可以是 jar 包里的 *.class，也可以来自远程服务器提供的字节流，字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

有很多字节码加密技术就是依靠定制 ClassLoader 来实现的。先使用工具对字节码文件进行加密，运行时使用定制的 ClassLoader 先解密文件内容再加载这些解密后的字节码。

每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。ClassLoader 就像一个容器，里面装了很多已经加载的 Class 对象。

```
class Class<T> {
  ...
  private final ClassLoader classLoader;
  ...
}
```

延迟加载

JVM 运行并不是一次性加载所需要的全部类的，它是按需加载，也就是延迟加载。程序在运行的过程中会逐渐遇到很多不认识的新类，这时候就会调用 ClassLoader 来加载这些类。加载完成后就会将 Class 对象存在 ClassLoader 里面，下次就不需要重新加载了。

比如你在调用某个类的静态方法时，首先这个类肯定是需要被加载的，但是并不会触及这个类的实例字段，那么实例字段的类别 Class 就可以暂时不必去加载，但是它可能会加载静态字段相关的类别，因为静态方法会访问静态字段。而实例字段的类别需要等到你实例化对象的时候才可能会加载。

各司其职

JVM 运行实例中会存在多个 ClassLoader，不同的 ClassLoader 会从不同的地方加载字节码文件。它可以从不同的文件目录加载，也可以从不同的 jar 文件中加载，也可以从网络上不同的服务地址来加载。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA_HOME/lib/rt.jar 文件中，我们常用内置库 java.xxx.* 都在里面，比如 java.util.*、java.io.*、java.nio.*、java.lang.* 等等。这个 ClassLoader 比较特殊，它是由 C 代码实现的，我们将它称之为「根加载器」。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA_HOME/lib/ext/*.jar 中，有很多 jar 包。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。我们自己编写的代码以及使用的第三方 jar 包通常都是由它来加载的。

那些位于网络上静态文件服务器提供的 jar 包和 class文件，jdk 内置了一个 URLClassLoader，用户只需要传递规范的网络路径给构造器，就可以使用 URLClassLoader 来加载远程类库了。URLClassLoader 不但可以加载远程类库，还可以加载本地路径的类库，取决于构造器中不同的地址形式。ExtensionClassLoader 和 AppClassLoader 都是 URLClassLoader 的子类，它们都是从本地文件系统里加载类库。

AppClassLoader 可以由 ClassLoader 类提供的静态方法 getSystemClassLoader() 得到，它就是我们所说的「系统类加载器」，我们用户平时编写的类代码通常都是由它加载的。当我们的 main 方法执行的时候，这第一个用户类的加载器就是 AppClassLoader。

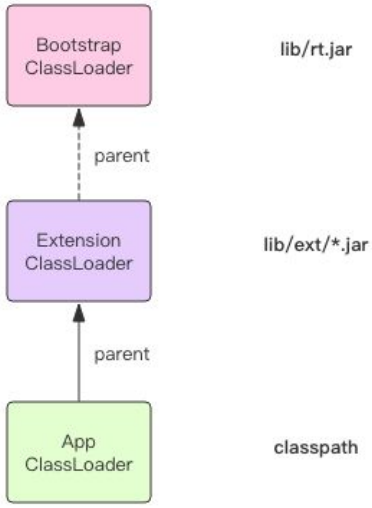
ClassLoader 传递性

程序在运行过程中，遇到了一个未知的类，它会选择哪个 ClassLoader 来加载它呢？虚拟机的策略是使用调用者 Class 对象的 ClassLoader 来加载当前未知的类。何为调用者 Class 对象？就是在遇到这个未知的类时，虚拟机肯定正在运行一个方法调用（静态方法或者实例方法），这个方法挂在哪个类上面，那这个类就是调用者 Class 对象。前面我们提到每个 Class 对象里面都有一个 classLoader 属性记录了当前的类是由谁来加载的。

因为 ClassLoader 的传递性，所有延迟加载的类都会由初始调用 main 方法的这个 ClassLoader 全全负责，它就是 AppClassLoader。

双亲委派

前面我们提到 AppClassLoader 只负责加载 Classpath 下面的类库，如果遇到没有加载的系统类库怎么办，AppClassLoader 必须将系统类库的加载工作交给 BootstrapClassLoader 和 ExtensionClassLoader 来做，这就是我们常说的「双亲委派」。



AppClassLoader 在加载一个未知的类名时，它并不是立即去搜寻 Classpath，它会首先将这个类名称交给 ExtensionClassLoader 来加载，如果 ExtensionClassLoader 可以加载，那么 AppClassLoader 就不用麻烦了。否则它就会搜索 Classpath 。

而 ExtensionClassLoader 在加载一个未知的类名时，它也并不是立即搜寻 ext 路径，它会首先将类名称交给 BootstrapClassLoader 来加载，如果 BootstrapClassLoader 可以加载，那么 ExtensionClassLoader 也就不用麻烦了。否则它就会搜索 ext 路径下的 jar 包。

这三个 ClassLoader 之间形成了级联的父子关系，每个 ClassLoader 都很懒，尽量把工作交给父亲做，父亲干不了了自己才会干。每个 ClassLoader 对象内部都会有一个 parent 属性指向它的父加载器。

```
class ClassLoader {
  ...
  private final ClassLoader parent;
  ...
}
```

值得注意的是图中的 ExtensionClassLoader 的 parent 指针画了虚线，这是因为它的 parent 的值是 null，当 parent 字段是 null 时就表示它的父加载器是「根加载器」。如果某个 Class 对象的 classLoader 属性值是 null，那么就表示这个类也是「根加载器」加载的。

Class.forName

当我们在使用 jdbc 驱动时，经常会使用 Class.forName 方法来动态加载驱动类。

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

其原理是 mysql 驱动的 Driver 类里有一个静态代码块，它会在 Driver 类被加载的时候执行。这个静态代码块会将 mysql 驱动实例注册到全局的 jdbc 驱动管理器里。

```
class Driver {
  static {
    try {
       java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException E) {
       throw new RuntimeException("Can't register driver!");
    }
  }
  ...
}
```

forName 方法同样也是使用调用者 Class 对象的 ClassLoader 来加载目标类。不过 forName 还提供了多参数版本，可以指定使用哪个 ClassLoader 来加载

```
Class<?> forName(String name, boolean initialize, ClassLoader cl)
```

通过这种形式的 forName 方法可以突破内置加载器的限制，通过使用自定类加载器允许我们自由加载其它任意来源的类库。根据 ClassLoader 的传递性，目标类库传递引用到的其它类库也将会使用自定义加载器加载。

自定义加载器

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。下面我使用伪代码表示一下基本过程

```
class ClassLoader {

  // 加载入口，定义了双亲委派规则
  Class loadClass(String name) {
    // 是否已经加载了
    Class t = this.findFromLoaded(name);
    if(t == null) {
      // 交给双亲
      t = this.parent.loadClass(name)
    }
    if(t == null) {
      // 双亲都不行，只能靠自己了
      t = this.findClass(name);
    }
    return t;
  }

  // 交给子类自己去实现
  Class findClass(String name) {
    throw ClassNotFoundException();
  }

  // 组装Class对象
  Class defineClass(byte[] code, String name) {
    return buildClassFromCode(code, name);
  }
}

class CustomClassLoader extends ClassLoader {

  Class findClass(String name) {
    // 寻找字节码
    byte[] code = findCodeFromSomewhere(name);
    // 组装Class对象
    return this.defineClass(code, name);
  }
}
```

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

```
// ClassLoader 构造器
protected ClassLoader(String name, ClassLoader parent);
```

双亲委派规则可能会变成三亲委派，四亲委派，取决于你使用的父加载器是谁，它会一直递归委派到根加载器。

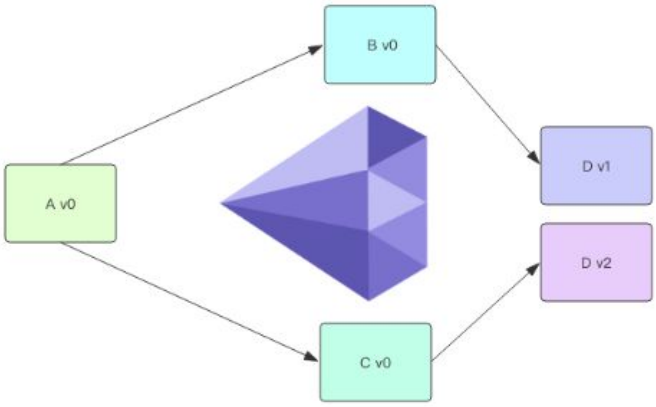
Class.forName vs ClassLoader.loadClass

这两个方法都可以用来加载目标类，它们之间有一个小小的区别，那就是 Class.forName() 方法可以获取原生类型的 Class，而 ClassLoader.loadClass() 则会报错。

```
Class<?> x = Class.forName("[I");
System.out.println(x);

x = ClassLoader.getSystemClassLoader().loadClass("[I");
System.out.println(x);

----------------------
class [I

Exception in thread "main" java.lang.ClassNotFoundException: [I
...
```

钻石依赖

项目管理上有一个著名的概念叫着「钻石依赖」，是指软件依赖导致同一个软件包的两个版本需要共存而不能冲突。



我们平时使用的 maven 是这样解决钻石依赖的，它会从多个冲突的版本中选择一个来使用，如果不同的版本之间兼容性很糟糕，那么程序将无法正常编译运行。Maven 这种形式叫「扁平化」依赖管理。

使用 ClassLoader 可以解决钻石依赖问题。不同版本的软件包使用不同的 ClassLoader 来加载，位于不同 ClassLoader 中名称一样的类实际上是不同的类。下面让我们使用 URLClassLoader 来尝试一个简单的例子，它默认的父加载器是 AppClassLoader

```
$ cat ~/source/jcl/v1/Dep.java
public class Dep {
    public void print() {
        System.out.println("v1");
    }
}

$ cat ~/source/jcl/v2/Dep.java
public class Dep {
 public void print() {
  System.out.println("v1");
```

```
 }
}

$ cat ~/source/jcl/Test.java
public class Test {
    public static void main(String[] args) throws Exception {
        String v1dir = "file:///Users/qianwp/source/jcl/v1/";
        String v2dir = "file:///Users/qianwp/source/jcl/v2/";
        URLClassLoader v1 = new URLClassLoader(new URL[]{new URL(v1dir)});
        URLClassLoader v2 = new URLClassLoader(new URL[]{new URL(v2dir)});

        Class<?> depv1Class = v1.loadClass("Dep");
        Object depv1 = depv1Class.getConstructor().newInstance();
        depv1Class.getMethod("print").invoke(depv1);

        Class<?> depv2Class = v2.loadClass("Dep");
        Object depv2 = depv2Class.getConstructor().newInstance();
        depv2Class.getMethod("print").invoke(depv2);

        System.out.println(depv1Class.equals(depv2Class));
   }
}
```

在运行之前，我们需要对依赖的类库进行编译

```
$ cd ~/source/jcl/v1
$ javac Dep.java
$ cd ~/source/jcl/v2
$ javac Dep.java
$ cd ~/source/jcl
$ javac Test.java
$ java Test
v1
v2
false
```

在这个例子中如果两个 URLClassLoader 指向的路径是一样的，下面这个表达式还是 false，因为即使是同样的字节码用不同的 ClassLoader 加载出来的类都不能算同一个类

```
depv1Class.equals(depv2Class)
```

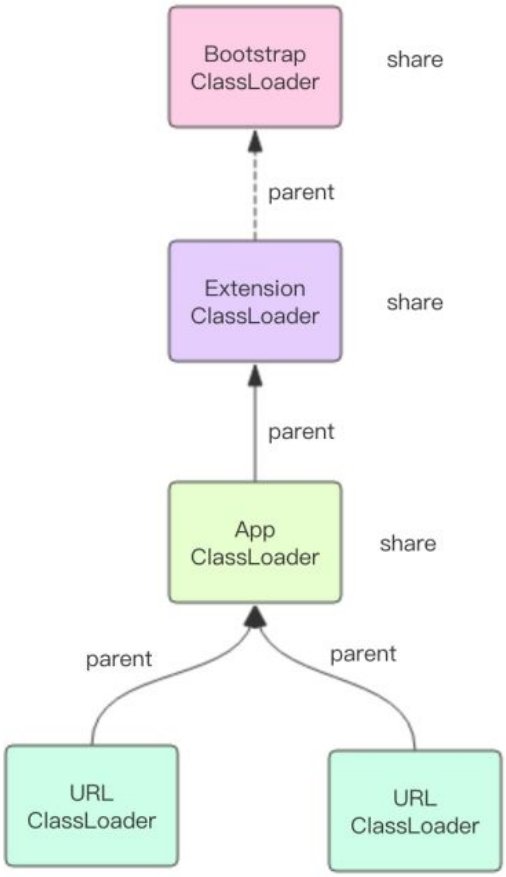
我们还可以让两个不同版本的 Dep 类实现同一个接口，这样可以避免使用反射的方式来调用 Dep 类里面的方法。

```
Class<?> depv1Class = v1.loadClass("Dep");
IPrint depv1 = (IPrint)depv1Class.getConstructor().newInstance();
depv1.print()
```

ClassLoader 固然可以解决依赖冲突问题，不过它也限制了不同软件包的操作界面必须使用反射或接口的方式进行动态调用。Maven 没有这种限制，它依赖于虚拟机的默认懒惰加载策略，运行过程中如果没有显示使用定制的 ClassLoader，那么从头到尾都是在使用 AppClassLoader，而不同版本的同名类必须使用不同的 ClassLoader 加载，所以 Maven 不能完美解决钻石依赖。
如果你想知道有没有开源的包管理工具可以解决钻石依赖的，我推荐你了解一下 sofa-ark，它是蚂蚁金服开源的轻量级类隔离框架。

分工与合作

这里我们重新理解一下 ClassLoader 的意义，它相当于类的命名空间，起到了类隔离的作用。位于同一个 ClassLoader 里面的类名是唯一的，不同的 ClassLoader 可以持有同名的类。
ClassLoader 是类名称的容器，是类的沙箱。



知乎 首发于 码洞

ClassLoader 共享同一个 parent 时，那么这个 parent 里面包含的类可以认为是所有子 ClassLoader 共享的。这也是为什么 BootstrapClassLoader 被所有的类加载器视为祖先加载器，JVM 核心类库自然应该被共享。

Thread.contextClassLoader

如果你稍微阅读过 Thread 的源代码，你会在它的实例字段中发现有一个字段非常特别

```
class Thread {
  ...
  private ClassLoader contextClassLoader;

  public ClassLoader getContextClassLoader() {
    return contextClassLoader;
  }

  public void setContextClassLoader(ClassLoader cl) {
    this.contextClassLoader = cl;
  }
  ...
}
```

contextClassLoader「线程上下文类加载器」，这究竟是什么东西？

首先 contextClassLoader 是那种需要显示使用的类加载器，如果你没有显示使用它，也就永远不会在任何地方用到它。你可以使用下面这种方式来显示使用它

```
Thread.currentThread().getContextClassLoader().loadClass(name);
```

这意味着如果你使用 forName(string name) 方法加载目标类，它不会自动使用 contextClassLoader。那些因为代码上的依赖关系而懒惰加载的类也不会自动使用 contextClassLoader来加载。

其次线程的 contextClassLoader 是从父线程那里继承过来的，所谓父线程就是创建了当前线程的线程。程序启动时的 main 线程的 contextClassLoader 就是 AppClassLoader。这意味着如果没有人工去设置，那么所有的线程的 contextClassLoader 都是 AppClassLoader。

那这个 contextClassLoader 究竟是做什么用的？我们要使用前面提到了类加载器分工与合作的原理来解释它的用途。

它可以做到跨线程共享类，只要它们共享同一个 contextClassLoader。父子线程之间会自动传递 contextClassLoader，所以共享起来将是自动化的。

如果不同的线程使用不同的 contextClassLoader，那么不同的线程使用的类就可以隔离开来。

如果我们对业务进行划分，不同的业务使用不同的线程池，线程池内部共享同一个 contextClassLoader，线程池之间使用不同的 contextClassLoader，就可以很好的起到隔离保护的作用，避免类版本冲突。

如果我们不去定制 contextClassLoader，那么所有的线程将会默认使用 AppClassLoader，所有的类都将会是共享的。

线程的 contextClassLoader 使用场合比较罕见，如果上面的逻辑晦涩难懂也不必过于计较。

JDK9 增加了模块功能之后对类加载器的结构设计做了一定程度的修改，不过类加载器的原理还是类似的，作为类的容器，它起到类隔离的作用，同时还需要依靠双亲委派机制来建立不同的类加载器之间的合作关系。

阅读更多精品文章，微信扫一扫上面的二维码关注公众号「码洞」

编辑于 2018-12-04

Java　Java 编程　Java类加载机制

天下无难试之HashMap面试刁难大全

老钱

java中的接口

雨那么下

Java类的继承

当你看到这篇文章
已经知道Java的
生活 Why we ne
能你有一部iphon
道在iphone 7之
3、2、1,那么在它

爪哇小姐姐

15 条评论

切换为时间排序

写下你的评论...

0xCAFFE 9 天前

好文！！

1

倪泽 9 天前

好有深度，赞美！

赞

丁一 9 天前

这个有用

赞

已赞同 383 15 条评论 分享 收藏

知乎 首发于 码洞

不错

赞

地狱少女火炮兰 8 天前

深度好文 深入浅出

赞

myan 8 天前

文章很清晰

赞

酸菜小牛肉 8 天前

好文

赞

开水Vincent 7 天前

ClassLoader.loadClass()为什么不能加载原生类型的class啊

赞

Un1ffy 回复 开水Vincent 6 天前

class.forName()除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块。

而classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容，只有在newInstance才会去执行static块。

2

老钱 (作者) 回复 Un1ffy 6 天前

已赞同 383

分享

classloader.loadclass 也有 bool 参数来控制的 ^_^

赞

展开其他 1 条回复

卧槽驿马 6 天前

[大笑][赞]

赞

Explorer 6 天前

跟马士兵讲的有些出入 但无论如何这篇文章都称得上一篇好文

赞

sigit 3 天前

叫菱形依赖不行嘛。。。非叫钻石依赖

赞

ethanhua 3 天前

写得好[赞]

赞

已赞同 383 15 条评论 分享 收藏 ···

知乎 首发于 码洞

已赞同 383

分享