【jvm jdk】类加载器1 sun.misc.Launcher程序入口类 & 构建ExtClassLoader，ExtClassLoader

原创

云川之下 于 2020-10-10 14:35:16 发布 892 收藏 3

分类专栏： jvm jdk 文章标签： launcher 类加载器路径

本文链接：https://blog.csdn.net/m0_45406092/article/details/108984101

通过这篇文章，我们能获得如下知识：

1. `sun.misc.Launcher` 类初始化过程

2.Launcher类中定义了ExtClassLoader，ExtClassLoader这两个类加载器，并创建他们的实例

3.解释了ExtClassLoader加载 `<JAVA_HOME>/lib/ext` 下的类，ExtClassLoader加载classpath下的类的原因

文章目录

1. 简介

`sun.misc.Launcher` 类是java的入口，在启动java应用的时候会首先创建 `Launcher` 类，创建 `Launcher` 类的时候会准备应用程序运行中需要的类加载器，位于 `jre/lib/rt.jar` 中。

2. Launcher是由BootStrapClassLoader加载的

`Launcher` 作为JAVA应用的入口，根据双亲委派模型， `Laucher` 是由JVM创建的，它的类加载器是 `BootStrapClassLoader` ， 这是一个C++编写的类加载器，是java应用体系中最顶层的类加载器，负责加载JVM需要的一些类库(`<JAVA_HOME>/lib`)，而Launcher类在rt.jar中，正好处于该加载器的加载路径下。可以通过一个简单的代码验证一下我们的想法。

```
package my.test;

import sun.misc.Launcher;

public class Test {

    public static void main(String[] args) {
        ClassLoader launcherClassLoader = Launcher.class.getClassLoader();
        System.out.println(launcherClassLoader);   //输出null

        ClassLoader testClassLoader = Test.class.getClassLoader();
        System.out.println(testClassLoader); // 输出 sun.misc.Launcher$AppClassLoader@19821f
    }
}
```

这里的`classLoader`是`null`，说明`Launcher`确实是`BootstrapClassLoader`加载的。

> 为何`classLoader`是`null`，说明`Launcher`确实是`BootstrapClassLoader`加载的？因为`java.lang.Class`类的`getClassLoader()`方法用于获取此实体的`classLoader`，该实体可以是类，数组，接口等。我们知道类加载器类型包括四种，分别是`启动类加载器（Bootstrap ClassLoader）`、`扩展类加载器（Extension ClassLoader）`、`应用程序类加载器（Application ClassLoader）`、`用户自定义加载器`，如果是后3种类型，一般会打印出具体的信息，而前面代码中的打印2条结果，第二条`sun.misc.Launcher$AppClassLoader@19821f`说明应用了`程序类加载器`，含有关键词'AppClassLoader'，与此类似，如果是Ext或自定义类型，也有相关的关键词，而第一条为null，不是后3种的类型之一，据此反推第一条是Bootstrap ClassLoader。

其实也不需要反推，根本原因在于`java.lang.Class`类的`getClassLoader()`方法实现机制，我们来看下该方法的源码和注释：

```
//JDK 1.8
public final class Class<T>  {
       /**
     * Returns the class loader for the class.  Some implementations may use
     * null to represent the bootstrap class loader. This method will return
     * null in such implementations if this class was loaded by the bootstrap
     * class loader.
     ...

     */
    @CallerSensitive
    public ClassLoader getClassLoader() {
        ClassLoader cl = getClassLoader0();
        if (cl == null)
            return null;
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            ClassLoader.checkClassLoaderPermission(cl, Reflection.getCallerClass());
        }
        return cl;
    }
```

`Some implementations may use null to represent the bootstrap class loader`

该方法的注释明确说明当返回值为null时，表示使用的是bootstrap class loader。

3. Launcher的初始化过程

Launcher的源码：

```
//JDK 1.8

public class Launcher {
    private static Launcher launcher = new Launcher(); //类被加载时，触发创建静态实例对象，进而触发构造函数

 public Launcher() {    //构造函数
   ExtClassLoader extClassLoader;
   try {
     extClassLoader = ExtClassLoader.getExtClassLoader();   // 获取ExtClassLoader
   } catch (IOException iOException) {
     throw new InternalError("Could not create extension class loader");
   }
   try {
     this.loader = AppClassLoader.getAppClassLoader(extClassLoader);  //以ExtClassLoader 作为父类加载器创建一个AppC
lassLoader
   } catch (IOException iOException) {
     throw new InternalError("Could not create application class loader");
   }
   Thread.currentThread().setContextClassLoader(this.loader);   //设置默认加载器
   ....
   }
}
```

> 首先我们知道Launcher类是应用的入口，位于rt.jar包中，默认会被BootstrapClassLoader加载，被加载时，触发创建静态实例对象，进而触发构造函数，在构造函数内，完成ExtClassLoader及AppClassLoader的创建。

我们来分析上面的代码：

- `Launcher`在创建的时候，第一件事情就是获取`ExtClassLoader`，`ExtClassLoader`在JVM中是一个单例， 创建过程也是通过获取环境变量来获取`ext`加载的目录，生成一个ExtClassLoader，ExtClassLoader是`URLClassLoader`的子类。

```
public static Launcher.ExtClassLoader getExtClassLoader() throws IOException {
      if (instance == null) {
          Class clazz = Launcher.ExtClassLoader.class;
          synchronized(Launcher.ExtClassLoader.class) {
              if (instance == null) {
                  instance = createExtClassLoader();   //单例模式
              }
          }
      }
      return instance;
  }
```

- 在获取`ExtClassLoader`之后，以此作为`父类加载器`创建一个`sun.misc.Launcher.AppClassLoader`，`AppClassLoader`的加载路径是`java.class.path`标记的路径，相同的，`AppClassLoader`也是`URLClassLoader`的子类。
- 最终会将当前线程的`上下文类加载器`设置为`AppClassLoader`。

通过上述分析，当我们需要获取当前应用程序的`AppClassLoader`或者`ExtClassLoader`的时候，可以直接使用`Launcher`来访问。

```
  public static void main(String[] args) {
     ClassLoader appClassLoader = Launcher.getLauncher().getClassLoader();
     System.out.println(appClassLoader);  //打印AppClassLoader

     ClassLoader extClassLoader = appClassLoader.getParent();  //通过父属性得到ext loader
     System.out.println(extClassLoader); //打印ExtClassLoader
```

执行结果：

```
  sun.misc.Launcher$AppClassLoader@19821f
  sun.misc.Launcher$ExtClassLoader@addbf1
```

4. 类加载器路径

jvm提供了三种系统加载器：

- 启动类加载器（Bootstrap ClassLoader）：C++实现，在java里无法获取，负责加载`<JAVA_HOME>/lib`下的类。
- 扩展类加载器（Extension ClassLoader）： Java实现，可以在java里获取，负责加载`<JAVA_HOME>/lib/ext`下的类。
- 系统类加载器/应用程序类加载器（Application ClassLoader）：是与我们接触对多的类加载器，我们写的代码默认就是由它来加载，ClassLoader.getSystemClassLoader返回的就是它。

类加载器路径在Launcher 源码体现：

```
public class Launcher {
  //系统类加载器加载的jar路径
  private static String bootClassPath = System.getProperty("sun.boot.class.path");

  //应用程序类加载器
   static class AppClassLoader extends URLClassLoader {
     public static ClassLoader getAppClassLoader(final ClassLoader extcl) throws IOException {
      ...
      //应用程序类加载器加载的jar路径
       final String s = System.getProperty("java.class.path");
    }
}
//扩展类加载器
  static class ExtClassLoader extends URLClassLoader {
      private static File[] getExtDirs() {
      ...
      String str = System.getProperty("java.ext.dirs");  //扩展类加载加载的jar路径器
      ...
     }

}
```

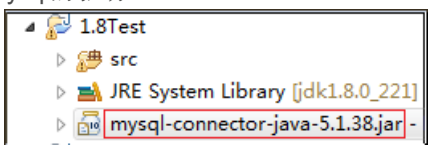
我们来验证下：

```java
package com.test;

public class Test {

    public static void main(String[] args) throws ClassNotFoundException {
        System.out.println(System.getProperty("sun.boot.class.path"));

        System.out.println("------------");

        System.out.println(System.getProperty("java.ext.dirs"));
        System.out.println("------------");
        System.out.println(System.getProperty("java.class.path"));

    }
}
```

执行结果：

```
C:\Program Files\Java\jdk1.8.0_221\jre\lib\resources.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\rt.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\sunrsasign.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\jce.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.8.0_221\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.8.0_221\jre\classes
------------
C:\Program Files\Java\jdk1.8.0_221\jre\lib\ext;C:\Windows\Sun\Java\lib\ext
------------
D:\workspace\vcn\1.8Test\bin;D:\maven\repo\mysql\mysql-connector-java\5.1.38\mysql-connector-java-5.1.38.jar
```

其中应用程序中，我引入了一个第三方包，mysql的驱动



参考：JAVA Launcher简析

classload之java程序入口sun.misc.Launcher源码分析

从 sun.misc.Launcher 类源码深入探索 ClassLoader