Java EE 企业应用系统开发
类加载和反射

王晓东
wangxiaodong@ouc.edu.cn

中国海洋大学

December 22, 2017

References

1. 疯狂 Java 讲义
2. 回顾：什么是反射？
   http://www.sczyh30.com/posts/Java/java-reflection-1/

大纲

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

接下来…

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

反射机制

- 程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言。从这个观点看，Perl、Python、Ruby 是动态语言，C++、Java、C# 不是动态语言。
- 但是 Java 有着一个非常突出的动态相关机制：Reflection，用在 Java 身上指的是我们可以于运行时加载、探知、使用编译期间完全未知的类。换句话说，Java 程序可以加载一个运行时才得知名称的类，获悉其完整构造（但不包括 methods 定义），并生成其对象实体、或对其 fields 设值、或唤起调用其 methods。

反射机制

- 反射机制就是 Java 语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。
- 反射机制是在运行状态（注意是运行而不是编译）中：
  - 在运行时判断任意一个对象所属的类；
  - 在运行时构造任意一个类的对象；
  - 在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用 private 方法）；
  - 在运行时调用任意一个对象的方法。

反射的主要用途

❖ 一个常见的场景

当我们在使用 IDE（如 Eclipse）时，当我们输入一个对象或类并想调用它的属性或方法，一按点号，编译器就会自动列出它的属性或方法，这里就会用到反射。

反射的主要用途 2

❖ 反射最重要的用途就是开发各种通用框架

很多框架（比如 Spring、Strut）都是基于配置化的，比如通过 XML 文件配置 JavaBean 和 Action。为了保证框架的通用性，需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

一个例子 Struts2 框架开发中会在 struts.xml 里配置 Action

```
1    <action name="login" class="ouc.j2ee.action.LoginAction" method="execute">
2      <result name="success">index.jsp</result>
3      <result name="error">login.jsp</result>
4    </action>
```

XML 配置文件与 Action 实现建立了映射关系。用户请求 login.action 会被 StrutsPrepareAndExecuteFilter 拦截并解析 struts.xml 文件，检索其中 name 为 login 的 Action，并根据 class 属性创建 LoginAction 的实例，并用 invoke 方法来调用 execute 方法。

这个过程是基于 Java 反射框架完成的。

反射的主要用途 3

❖ 依赖注入

有两个组件 A 和 B，A 依赖于 B。

```
public class A {
  public void importantMethod() {
    B b = ...; // get an instance of B
    b.usefulMethod();
    ...
  }
}
```

我们需要获得 B 的实例的引用。如果 B 是接口且有多个实现该怎么做？

反射的主要用途 3

❖ 依赖注入

接管对象的创建工作，并将该对象的引用注入需要该对象的组件。例如，我们使用 Spring 框架将对象 B 注入到对象 A 中，A 需要进行如下修改，加入 setB() 方法：

```
public class A {
  private B b;
  public void importantMethod() {
    B b = ...; // get an instance of B
    b.usefulMethod();
    ...
  }

  public void setB(B b) {
    this.b = b;
  }
}
```

```
1  <bean id="b" class="ouc.j2ee.sample.B" />
2  <bean id="a" class="ouc.j2ee.sample.A">
3    <property name="b" ref="b"></property>
4  </bean>
```

接下来…

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

JVM 和类

同一个 JVM 的所有线程、所有变量都处在同一个进程里，它们都使用该 JVM 进程的内存区。当系统出现以下几种情况时，JVM 进程将被终止：

- 程序运行至正常结束。
- 程序运行到使用 System.exit() 或 Runtime.getRuntime().exit() 代码结束程序。
- 程序执行过程中遇到未捕获的异常或者错误而结束。
- 程序所在的平台强制结束了 JVM 进程。

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤对该类进行初始化。

类的加载

类的加载是指将类的 class 文件读入内存，并为之创建一个 java.lang.Class 对象。
（类是某一类对象的抽象，都是 java.lang.Class 的实例）

- JVM 提供类加载器（系统类加载器）来完成对类的加载。
- 此外，开发者可以通过继承 ClassLoader 基类来创建自己的类加载器。
- 可以从本地文件系统、jar 包和网络方式加载类的 class 文件。
- 类加载器通常无须等到“首次使用”该类时才加载此类，Java 虚拟机允许系统预先加载某些类。

类的连接

类被加载生成对应的 Class 对象后，进入连接阶段，负责把类的二进制数据合并到 JRE 中。

1. 验证：用于检验被加载的类是否有正确的内部结构，并和其它类协调一致。
2. 准备：负责为类的静态属性分配内存，并设置默认初始值。
3. 解析：将类的二进制数据中的符号引用替换成直接引用。

类的初始化

主要负责对静态属性进行初始化：

- 声明静态属性时指定初始值。
- 使用静态初始化块为静态属性指定初始值。

```
public class Test {
  static int a = 5;
  static int b;
  static int c;
  static {
    b = 6;
  }
  ... ...
}
```

初始化上述代码后：a=5 b=6 c=0。

注意：程序主动使用一个类时，系统会保证该类以及所有父类（包括直接父类和间接父类）都会被初始化，所有 JVM 最先初始化的总是 java.lang.Object 类。

类的初始化时机

当 Java 程序首次通过下面 6 种方式使用某个类或接口时，系统会初始化该类或接口：

- 创建类的实例，包括通过 new 操作符、通过反射、通过反序列化的方式。
- 调用某个类的静态方法。
- 访问某个类或接口的静态属性，或为该静态属性赋值。
- 使用反射方式来强制创建某个类或接口对应的 java.lang.Class 对象。例如，Class.forName(”Person”)。
- 初始化某个类的子类，其父类都会被初始化。
- 直接使用 java.exe 运行某个主类时。

类的初始化时机

当使用 ClassLoader 类的 loadClass() 方法来加载某个类时，该方法只是加载该类，并不会执行该类的初始化。

```
ClassLoader cl = ClassLoader.getSystemClassLoader();
cl.loadClass("Tester");
```

当使用 Class 的 forName() 静态方法时导致强制初始化该类。

```
Class.forName("Tester");
```

接下来…

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

类加载器简介

类加载器负责加载所有的类，在内存中生成 java.lang.Class 的实例。一个载入 JVM 的类也有一个唯一的标识。
注意：在 Java 中，一个类用其全限定名作为标识。在 JVM 中，一个类用其全限定名和其类加载器作为唯一标识。例如：包 pg 中的 Person 类，被类加载器 ClassLoader 的实例 k1 负责加载，则该 Person 类对应的 Class 对象在 JVM 中表示为 (Person, pg, k1)。类加载器不同，即使加载同一个类，所加载的类的实例也是完全不同、互不兼容的。
JVM 启动时，会形成由三个类加载器组成的初始类加载器层次结构：

- Bootstrap ClassLoader ➪ 根类加载器
- Extension ClassLoader ➪ 扩展类加载器
- System ClassLoader ➪ 系统类加载器

ClassLoader

- Bootstrap ClassLoader 根类加载器负责加载 Java 的核心类。它非常特殊，并不是 java.lang.ClassLoader 的子类，而是由 JVM 自身实现的。
- Extension ClassLoader 扩展类加载器，负责加载 JRE 的扩展目录（JAVA_HOME/jre/lib/ext 或者 java.ext.dirs 系统属性指定的目录中的 JAR 的类包。
- System ClassLoader 系统（应用）类加载器，负责在 JVM 启动时，加载来自 java 中的 -classpath 选项或 java.class.path 系统属性，或 CLASSPATH 环境变量所指定 JAR 包和类路径。程序可以通过 ClassLoader 的静态方法 getSystemClassLoader() 获得该类加载器。没有特别指定，用户自定义的类加载器都以该类加载器作为父加载器。

类加载机制

❖ 关于类加载机制的几点说明

1. 全盘负责：当一个类加载器复负责加载某个 Class 时，该 Class 所依赖和引用的其它 Class 也将由该类加载器负责载入，除非显式使用另一个类加载器载入。
2. 父类委托：先让父类加载器试图加载该 Class，只有父类加载器无法加载该类时才尝试从自己的类路径中加载该类。
3. 缓存机制：类加载器先从缓存中搜索 Class，只有当缓存中不存在该 Class 对象时，系统才会重新读取该类对应的二进制数据。

类加载机制

File: ClassLoaderPropTest.java

```
import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;

public class BootstrapTest {
  public static void main(String[] args) throws IOException {
    ClassLoader systemLoader = ClassLoader.getSystemClassLoader();
    System.out.println("系统类加载器: " + systemLoader);
    // 系统类加载器的加载路径通常由 CLASSPATH 环境变量指定
    // 如果操作系统没有指定 CLASSPATH 环境变量，默认以当前路径
    // 作为系统类加载器的加载路径

    Enumeration<URL> eml = systemLoader.getResources("");
    while(eml.hasMoreElements()) {
      System.out.println(eml.nextElement());
    }
    ClassLoader extensionLoader = systemLoader.getParent();
    System.out.println("扩展类加载器: " + extensionLoader);
    System.out.println("扩展类加载器的加载路径: " + System.getProperty("java.ext.dirs"));
    System.out.println("扩展类加载器的parent: " + extensionLoader.getParent());
  }
}
```

类加载机制

上述代码的执行结果：

```
系统类加载器：sun.misc.Launcher$AppClassLoader@dda25b
file:/C:/Java_Workspace/Test/bin/
扩展类加载器：sun.misc.Launcher$ExtClassLoader@ce2187
扩展类加载器的加载路径：C:\Program Files\Java\jre7\lib\ext;C:\WINDOWS
\Sun\Java\lib\ext
扩展类加载器的parent：null
```

- 可以看出，扩展类加载器的 getParent() 方法返回 null，并不是根类加载器。这是因为根类加载器没有继承自 ClassLoader 抽象类，所以返回空。但实际上，根类加载器确实是扩展类加载器的父类加载器。
- 可以看出，系统类加载器是 AppClassLoader 的实例，扩展类加载器是 ExtClassLoader 的实例；实际上，这两个类都是 URLClassLoader 的实例。

类加载器加载 Class 的大致步骤

1. 检查此 Class 是否载入过（即缓存中是否存在），如果有则直接进入第 8 步，否则进入第 2 步。
2. 如果父加载器不存在（如果父加载器不存在，要么 parent 一定是根加载器，要么本身就是根加载器），则跳到第 4 步。如果父加载器存在，则接着执行第 3 步。
3. 请求父加载器加载目标类，如果成果则进入第 8 步，否则执行第 5 步。
4. 请求使用根加载器载入目标类，如果成功则进入第 8 步，否则跳到第 7 步。
5. 从与此 ClassLoader 相关的类路径中寻找 Class 文件，如果找到则执行第 6 步，否则跳到第 7 步。
6. 从文件中载入 Class，成功载入后则跳到第 8 步。
7. 抛出 ClassNotFoundException 异常。
8. 返回 Class。

创建并使用自定义的类加载器

略。

URLClassLoader 类

Java 为 ClassLoader 提供了一个 URLClassLoader 实现类，该类是系统类加载器和扩展类加载器的父类。该类既可以从本地文件系统获取二进制文件来加载类，也可以从远程主机获取二进制文件来加载类。

URLClassLoader 提供了如下两个构造器：

- URLClassLoader(URL[] urls) 使用默认的父类加载器创建一个 ClassLoader 对象，该对象将从 urls 所指定的系列路径中查询并加载类。
- URLClassLoader(URL[] urls, ClassLoader parent) 使用指定的父类加载器创建一个 ClassLoader 对象。

一旦得到了 URLClassLoader 对象后，就可以调用该对象的 loadClass 方法加载指定类。

从文件系统中加载 MySQL 驱动的示例

```
URL[] urls = {new URL("file:mysql-connector-java-***-bin.jar")};
URLClassLoader myClassLoader = new URLClassLoader(urls);
Driver driver = (Driver) myClassLoader.loadClass("com.mysql.jdbc.Driver").newInstance();
... ...
Connection conn = driver.connect("jdbc:mysql://localhost:3306/mysql", props);
```

- 该类加载器的加载路径为当前路径下的 mysql-connector-java-***-bin.jar 文件。这里 file: 前缀表明从本地文件系统加载，也可以以 http: 或 ftp: 为前缀，表示通过网络加载。
- 使用 ClassLoader 的 loadClass 加载指定类，并调用 Class 对象的 newInstance() 方法创建一个该类的实例。

接下来…

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

获得 Class 对象的三种方式

1. 使用 Class 类的 forName() 静态方法。该方法需要传入字符串参数，为某个类的全限定名（包含完整的包名）。
2. 调用某个类的 class 属性获取该类的 Class 对象，例如 Person.class 将返回 Person 类对应的 Class 对象。
3. 调用某个对象的 getClass() 方法，该方法是 java.lang.Object 类的一个方法，该方法会返回该对象所属类对应的 Class 对象。

第二种方式更建议使用，具有两个优势：

- 代码更安全，程序在编译阶段就可以检查需要访问的 Class 对象是否存在。
- 程序性能更高，因为这种方式无需调用方法，所以具有更好的性能。

使用反射生成并操作对象

Class 对象可以获得该类里的成分，包括：

- 方法 由 Method 对象表示，可以通过该对象执行对应的方法。
- 构造器 由 Constructor 对象表示，可以通过该对象来调用对应的构造器创建对象。
- Field 由 Field 对象表示，可以通过该对象直接访问并修改对象属性值。

上述三个类都定义在 java.lang.reflect 包下，并实现了 java.lang.reflect.Member 接口。

创建对象

使用反射来生成对象的两种方式：

- 使用 Class 对象的 newInstance() 方法来创建该 Class 对象对应的类实例，此种方法要求该 Class 对象的对应类有默认构造器，而执行 newInstance() 方法时实际利用默认构造器来创建该类的实例。
- 先使用 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance() 方法创建该 Class 对象对应类的实例。通过这种方式可以选择使用某个类的指定构造器来创建实例。

创建对象示例

很多 Java EE 框架中都需要根据配置文件信息来创建 Java 对象。从配置文件中读取的只是某个类的字符串名，程序就需要根据该字符串创建对应的实例，就必须使用反射。

❖ 简单对象池的示例

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class ObjectPoolFactory {
  //定义对象池
  private Map<String, Object> objectPool = new HashMap<String, Object>();

  private Object createObject(String ClazzName)
  throws ClassNotFoundException, InstantiationException,
  IllegalAccessException {
    //根据字符串来获取对应的 Class 对象
    Class<?> clazz = Class.forName(ClazzName);
    return clazz.newInstance();
  }
```

创建对象示例（续）

```
//根据指定文件来初始化对象池
public void initPool(String fileName) throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
  FileInputStream fis = null;
  try {
    fis = new FileInputStream(fileName);
    Properties props = new Properties();
    props.load(fis);
    for (String name : props.stringPropertyNames()) {
      objectPool.put(name, createObject(props.getProperty(name)));
    }
  } catch (IOException ex) {
    System.out.println("读取" + fileName + "异常");
  } finally {
    try {
      if (fis != null) {
        fis.close();
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

创建对象示例（续）

```
  public Object getObject(String name) {
    return objectPool.get(name);
  }

  public static void main(String[] args) throws ClassNotFoundException,
  InstantiationException, IllegalAccessException {
    ObjectPoolFactory pf = new ObjectPoolFactory();
    pf.initPool("obj.txt");
    System.out.println(pf.getObject("a"));
  }
}
```

File: obj.txt

```
1 a=java.util.Date
2 b=javax.swing.JFrame
```

输出结果：

output

```
Mon Sep 23 11:26:40 CST 2013
```

使用指定的构造器创建对象

需要利用 Constructor 对象，每个 Constructor 对应一个构造器，步骤如下：

1. 获取该类的 Class 对象。
2. 利用 Class 对象的 getConstructor() 方法来获得指定构造器。
3. 调用 Constructor() 的 newInstance() 方法创建对象。

使用指定的构造器创建对象

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class CreateJFrame {
  public static void main(String[] args) throws ClassNotFoundException,
  IllegalArgumentException, InstantiationException,
  IllegalAccessException, InvocationTargetException,
  SecurityException, NoSuchMethodException {
    // 获取 JFrame 对应的 Class 对象
    Class<?> jframeClazz = Class.forName("javax.swing.JFrame");
    // 获取 JFrame 中带一个字符串参数的构造器
    Constructor ctor = jframeClazz.getConstructor(String.class);
    Object obj = ctor.newInstance("测试窗口");
    System.out.println(obj);
  }
}
```

注意：

- 如果要唯一的确定某类中的构造器，只要指定构造器的形参列表即可。
- 调用 Constructor 对象的 newInstance() 方法时通常需要传入参数，实际上等于调用它对应的构造器。
- 只有当程序需要动态创建某个类的对象时才会考虑使用反射；对于已知类的情形，通常没有必要通过反射创建对象（降低性能）。

调用方法

获得某个类对应的 Class 对象后，可以该对象的如下方法执行方法调用：

- getMethods() 方法：获取全部方法，返回值为 Method 对象数组；
- getMethod() 方法：获取指定方法，返回 Method 对象。

获取 Method 对象后，程序可以该对象的 invoke() 方法调用对应方法：

```
Object invoke(Object obj, Object ... args)
```

其中，obj 是主调，args 是执行该方法时传入该方法的参数。

调用方法

```
Class<?> targetClass = target.getClass();
Method mtd = targetClass.getMethod(mtdName, String.class);
mtd.invoke(target, props.getProperty(name));
```

当通过 Method 对象的 invoke 方法调用对应方法时，Java 会要求程序必须有调用该方法的权限。如果程序需要调用某个对象的 private 方法，可以先调用 Method 对象的如下方法：

- setAccessible(boolean flag) 将 Method 对象的 accessible 标志设置为指示的布尔值。值为 true 则指示该 Method 在使用时应该取消 Java 语言访问权限检查，值为 flase 则指示该 Method 在使用时应该实施 Java 语言访问权限检查。

访问属性值

通过 Class 对象的 getFields() 或 getField() 方法可以获取该类所包含的全部 Field（属性）或指定 Field。

- getXxx(Object obj) 获取 obj 对象该 Field 的属性值，此处 Xxx 对应 8 个基本类型，如果该属性的类型为引用类型则取消 get 后面的 Xxx。
- setXxx(Object obj, Xxx val) 将 obj 对象的该 Field 设置成 val 值。此处 Xxx 对应 8 个基本类型，如果该属性的类型为引用类型则取消 get 后面的 Xxx。

访问属性值

```
import java.lang.reflect.Field;
class Person{
  private String name;
  private int age;
  public String toString(){
    return "Person␣[␣name:␣" + name + ",␣age:␣" + age + "␣]";
  }
}

public class FieldTest {
  public static void main(String[] args) throws Exception {
    Person p = new Person();
    Class<Person> personClazz = Person.class;
    Field nameField = personClazz.getDeclaredField("name");
    nameField.setAccessible(true);
    nameField.set(p, "Kevin␣W");
    Field ageField = personClazz.getDeclaredField("age");
    ageField.setAccessible(true);
    ageField.setInt(p, 30);
    System.out.println(p);
  }
}
```

上述代码使用 getDeclaredField() 方法获取名为 name 的 Field，而不是使用 getField() 方法，因为 getField() 方法只能获取 public 的 Field，而 getDeclaredField() 则可以获取所有 Field。

操作数组

java.lang.reflect 包提供 Array 类，代表所有数组，程序可以通过 Array 类来动态的创建数组，操作数组元素。

- static Object newInstantce(Class<?> componentType, int... length) 创建一个具有指定元素类型、指定维度的新数组。
- static xxx getXxx(Object array, int index) 返回 array 数组中第 index 个元素，其中 xxx 是各种基本数据类型，如果数组元素为引用类型，则方法去掉 Xxx，为 get(Object array, int index)。
- static void setXxx(Object array, int index, xxx val) 将 array 数组中第 index 元素的值设为 val。其中 xxx 是各种基本数据类型，如果数组元素为引用类型，则方法去掉 Xxx，为 set(Object array, int index, Object val)。

操作数组

```
//创建一个元素类型为 String ，长度为 10 的数组
Object arr = Array.newInstance(String.class, 10);
//为数组中 index 为 5 的元素赋值
Array.set(arr, 5, "Java␣EE企业应用系统设计");
//取出 arr 数组中 index 为 5 的元素的值
Object book = Array.get(arr, 5);

//创建一个元素类型为 String 的三维数组
Object arr = Array.newInstance(String.class, 3, 4, 10);
//获取 arr 数组中 index 为 2 的元素，是二维数组
Object arrObj = Array.get(arr1, 2);
//获取 arrObj 数组中 index 为 3 的元素，应该是一维数组
Object Arr.get(arrObj, 3);
//将 arr 强制转换为三维数组
String[][][] cast = (String[][][]) arr;
```

使用反射生成 JDK 动态代理

THE END

wangxiaodong@ouc.edu.cn