# 反射机制

## 1. Java反射机制

Java反射(`Reflection`)是Java非常重要的动态特性，通过使用反射我们不仅可以获取到任何类的成员方法(`Methods`)、成员变量(`Fields`)、构造方法(`Constructors`)等信息，还可以动态创建Java类实例、调用任意的类方法、修改任意的类成员变量值等。Java反射机制是Java语言的动态性的重要体现，也是Java的各种框架底层实现的灵魂。

## 2. 获取Class对象

Java反射操作的是`java.lang.Class`对象，所以我们需要先想办法获取到Class对象，通常我们有如下几种方式获取一个类的Class对象：

1. `类名.class`，如:`com.anbai.sec.classloader.TestHelloWorld.class`。
2. `Class.forName("com.anbai.sec.classloader.TestHelloWorld")`。
3. `classLoader.loadClass("com.anbai.sec.classloader.TestHelloWorld");`

获取数组类型的Class对象需要特殊注意,需要使用Java类型的描述符方式，如下：

```
Class<?> doubleArray = Class.forName("[D");//相当于double[].class
Class<?> cStringArray = Class.forName("[[Ljava.lang.String;");// 相当于
String[][].class
```

**获取Runtime类Class对象代码片段：**

```
String className     = "java.lang.Runtime";
Class  runtimeClass1 = Class.forName(className);
Class  runtimeClass2 = java.lang.Runtime.class;
Class  runtimeClass3 =
ClassLoader.getSystemClassLoader().loadClass(className);
```

通过以上任意一种方式就可以获取`java.lang.Runtime`类的Class对象了，反射调用内部类的时候需要使用`$`来代替`.`,如`com.anbai.Test`类有一个叫做`Hello`的内部类，那么调用的时候就应该将类名写成：`com.anbai.Test$Hello`。

## 3. 反射java.lang.Runtime

`java.lang.Runtime`因为有一个`exec`方法可以执行本地命令，所以在很多的`payload`中我们都能看到反射调用`Runtime`类来执行本地系统命令，通过学习如何反射`Runtime`类也能让我们理解反射的一些基础用法。

**不使用反射执行本地命令代码片段：**

```
import org.apache.commons.io.IOUtils;
import java.io.IOException;

public class fs {
    public static void main(String[] args) {
        try {// 输出命令执行结果
```

```
 System.out.println(IOUtils.toString(Runtime.getRuntime().exec("whoami").get
InputStream(), "UTF-8"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
//因为默认没有这个包需要下载
//http://commons.apache.org/proper/commons-io/download_io.cgi
```

如上可以看到，我们可以使用一行代码完成本地命令执行操作，但是如果使用反射就会比较麻烦了，我们不得不需要间接性的调用`Runtime`的`exec`方法

**反射Runtime执行本地命令代码片段：**

```
// 获取Runtime类对象
Class runtimeClass1 = Class.forName("java.lang.Runtime");

// 获取构造方法
Constructor constructor = runtimeClass1.getDeclaredConstructor();
constructor.setAccessible(true);

// 创建Runtime类示例，等价于 Runtime rt = new Runtime();
Object runtimeInstance = constructor.newInstance();

// 获取Runtime的exec(String cmd)方法
Method runtimeMethod = runtimeClass1.getMethod("exec", String.class);

// 调用exec方法，等价于 rt.exec(cmd);
Process process = (Process) runtimeMethod.invoke(runtimeInstance, cmd);

// 获取命令执行结果
InputStream in = process.getInputStream();

// 输出命令执行结果
System.out.println(IOUtils.toString(in, "UTF-8"));
```

反射调用`Runtime`实现本地命令执行的流程如下：

1. 反射获取`Runtime`类对象(`Class.forName("java.lang.Runtime")`)。
2. 使用`Runtime`类的Class对象获取`Runtime`类的无参数构造方法(`getDeclaredConstructor()`)，因为`Runtime`的构造方法是`private`的我们无法直接调用，所以我们需要通过反射去修改方法的访问权限(`constructor.setAccessible(true)`)。
3. 获取`Runtime`类的`exec(String)`方法(`runtimeClass1.getMethod("exec", String.class);`)。
4. 调用`exec(String)`方法(`runtimeMethod.invoke(runtimeInstance, cmd)`)。

上面的代码每一步都写了非常清晰的注释，接下来我们将进一步深入的了解下每一步具体含义。

# 反射创建类实例

在Java的`任何一个类都必须有一个或多个构造方法`，如果代码中没有创建构造方法那么在类编译的时候会自动创建一个无参数的构造方法。

**Runtime类构造方法示例代码片段：**

```
public class Runtime {
   /** Don't let anyone else instantiate this class */
  private Runtime() {}

}
```

从上面的`Runtime`类代码注释我们看到它本身是不希望除了其自身的任何人去创建该类实例的，因为这是一个私有的类构造方法，所以我们没办法`new`一个`Runtime`类实例即不能使用`Runtime rt = new Runtime();`的方式创建`Runtime`对象，但示例中我们借助了反射机制，修改了方法访问权限从而间接的创建出了`Runtime`对象。

`runtimeClass1.getDeclaredConstructor`和`runtimeClass1.getConstructor`都可以获取到类构造方法，区别在于后者无法获取到私有方法，所以一般在获取某个类的构造方法时候我们会使用前者去获取构造方法。如果构造方法有一个或多个参数的情况下我们应该在获取构造方法时候传入对应的参数类型数组，如：`clazz.getDeclaredConstructor(String.class, String.class)`。

如果我们想获取类的所有构造方法可以使用：`clazz.getDeclaredConstructors`来获取一个`Constructor`数组。

获取到`Constructor`以后我们可以通过`constructor.newInstance()`来创建类实例,同理如果有参数的情况下我们应该传入对应的参数值，如:`constructor.newInstance("admin", "123456")`。当我们没有访问构造方法权限时我们应该调用`constructor.setAccessible(true)`修改访问权限就可以成功的创建出类实例了。

# 4. 反射调用类方法

`Class`对象提供了一个获,取某个类的所有的成员方法的方法，也可以通过方法名和方法参数类型来获取指定成员方法。

**获取当前类所有的成员方法：**

```
Method[] methods = clazz.getDeclaredMethods()
```

**获取当前类指定的成员方法：**

```
Method method = clazz.getDeclaredMethod("方法名");
Method method = clazz.getDeclaredMethod("方法名", 参数类型如String.class，多个参数用","号隔开);
```

`getMethod`和`getDeclaredMethod`都能够获取到类成员方法，区别在于`getMethod`只能获取到`当前类和父类`的所有有权限的方法(如：`public`)，而`getDeclaredMethod`能获取到当前类的所有成员方法(不包含父类)。

**反射调用方法**

获取到`java.lang.reflect.Method`对象以后我们可以通过`Method`的`invoke`方法来调用类方法。

**调用类方法代码片段：**

```
method.invoke(方法实例对象， 方法参数值，多个参数值用","隔开);
```

`method.invoke`的第一个参数必须是类实例对象，如果调用的是`static`方法那么第一个参数值可以传`null`，因为在java中调用静态方法是不需要有类实例的，因为可以直接`类名.方法名(参数)`的方式调用。

`method.invoke`的第二个参数不是必须的，如果当前调用的方法没有参数，那么第二个参数可以不传，如果有参数那么就必须严格的`依次传入对应的参数类型`。

# 5. 反射调用成员变量

Java反射不但可以获取类所有的成员变量名称，还可以无视权限修饰符实现修改对应的值。

**获取当前类的所有成员变量：**

```
Field fields = clazz.getDeclaredFields();
```

**获取当前类指定的成员变量：**

```
Field field  = clazz.getDeclaredField("变量名");
```

`getField`和`getDeclaredField`的区别同`getMethod`和`getDeclaredMethod`。

**获取成员变量值：**

```
Object obj = field.get(类实例对象);
```

**修改成员变量值：**

```
field.set(类实例对象, 修改后的值);
```

同理，当我们没有修改的成员变量权限时可以使用:`field.setAccessible(true)`的方式修改为访问成员变量访问权限。

如果我们需要修改被`final`关键字修饰的成员变量，那么我们需要先修改方法

```
// 反射获取Field类的modifiers
Field modifiers = field.getClass().getDeclaredField("modifiers");

// 设置modifiers修改权限
modifiers.setAccessible(true);

// 修改成员变量的Field对象的modifiers值
modifiers.setInt(field, field.getModifiers() & ~Modifier.FINAL);

// 修改成员变量值
field.set(类实例对象, 修改后的值);
```

# 6. Java反射机制总结

Java反射机制是Java动态性中最为重要的体现，利用反射机制我们可以轻松的实现Java类的动态调用。Java的大部分框架都是采用了反射机制来实现的(如:`Spring MVC`、`ORM框架`等)，Java反射在编写漏洞利用代码、代码审计、绕过RASP方法限制等中起到了至关重要的作用。