JAVA安全基础（三）-- java动态代理机制

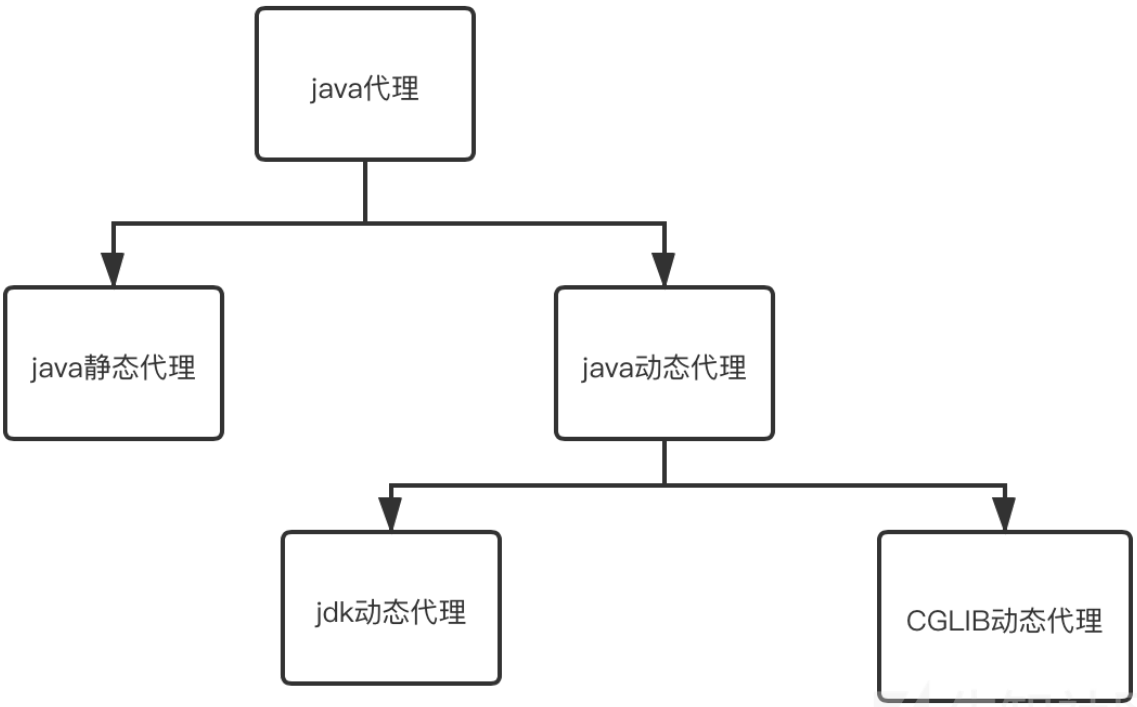
小阳 / 2021-03-04 14:48:00 / 浏览数 13451

0x01 前言

在上篇文章当中，我们了解了反射的机制，可以绕过java私有访问权限检查，反射获取并调用Runtime类执行命令执行。而Java的自带jdk动态代理机制，位于java.lang.reflect.proxy包中，其本质实现是通过反射执行invoke方法来动态获取执行方法。大家可以总结进行对比学习下。

0x02 概念

代理模式Java当中最常用的设计模式之一。其特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。而Java的代理机制分为静态代理和动态代理，而这里我们主要重点学习java自带的jdk动态代理机制。



0x03 静态代理示例讲解

在讲动态代理之前，我们先了解下什么是静态代理。静态代理在编译使用时,定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类。我们用一个出租房子作为实例讲解。

定义一个接口

```
public interface Rental {
    public void sale();
}
```

委托类

// 委托类，实现接口的方法

```
public class Entrust implements Rental{
    @Override
    public void sale() {
        System.out.println("出租房子");
    }
}
```

代理类

```
public class AgentRental implements Rental{
    private Rental target; // 被代理对象
    public AgentRental(Rental target) {
        this.target = target;
    }
    @Override
    public void sale() {
        System.out.println("房子出租价位有1k-3k"); // 增加新的操作
        target.sale(); // 调用Entrust委托类的sale方法
    }
}
```

测试类

// 测试类，生成委托类实例对象，并将该对象传入代理类构造函数中。

```
public class Test {
    // 静态代理使用示例
    public static void consumer(Rental subject) {
        subject.sale();
    }
    public static void main(String[] args) {
        Rental test = new Entrust();
        System.out.println("---使用代理之前---");
        consumer(test);
        System.out.println("---使用代理之后---");
        consumer(new AgentRental(test));
    }
}
```

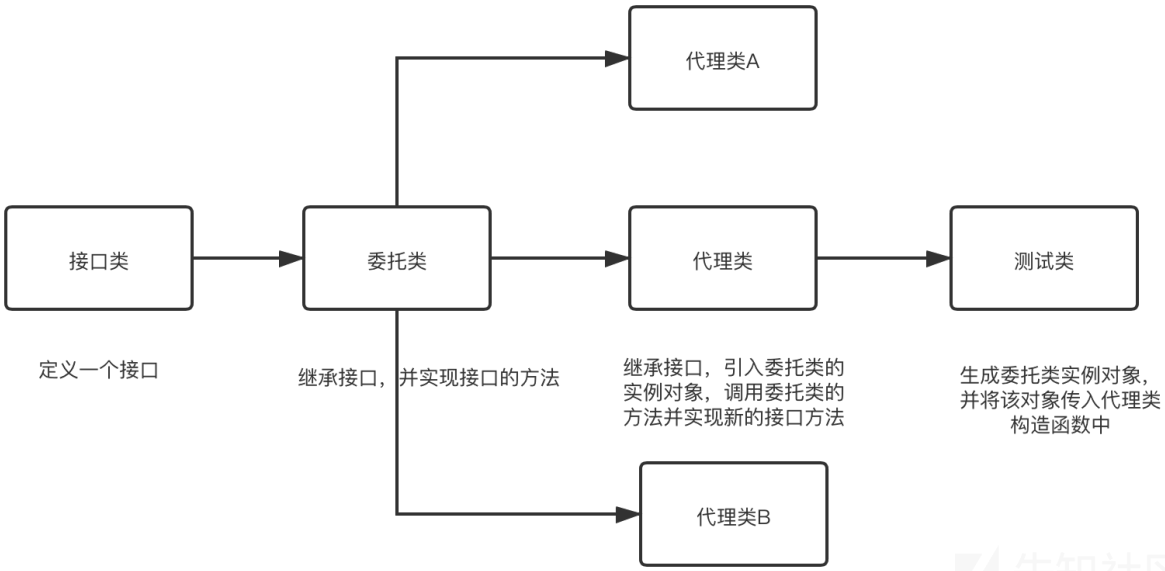
```java
package com.StaticProxy;

public class Test {

    public static void consumer(Rental subject) { subject.sale(); }

    public static void main(String[] args) {
        Rental test = new Entrust();
        System.out.println("---使用代理之前---");
        consumer(test);
        System.out.println("---使用代理之后---");
        consumer(new AgentRental(test));
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin/java ...
---使用代理之前---
出租房子
---使用代理之后---
房子出租价位有1k-3k
出租房子

Process finished with exit code 0
```


通过上面的例子，我们可以看见静态代理的优点：

我们可以在不改变Entrust委托类源代码的情况下 ,通过AgentRental代理类来修改Entrust委托类的功能，从而实现“代理”操作。在进行代理后，自定义说明房子出租价位有1k-3k的操作方法。

但这个是我们通过代理类进行实现更改的方法，如果当我们需要过多的代理类对委托类进行修改的情况下，则可能出现下图情况:



由此可以我们得知此静态代理的缺点：

当我们的接口类需要增加和删除方式的时候，委托类和代理类都需要更改，不容易维护。

同时如果需要代理多个类的时候，每个委托类都要编写一个代理类，会导致代理类繁多，不好管理。

因为java静态代理是对类进行操作的，我们需要一个个代理类去实现对委托类的更改操作，针对这个情况，我们可以利用动态代理来解决，通过程序运行时自动生成代理类。

0x04 java动态代理简介

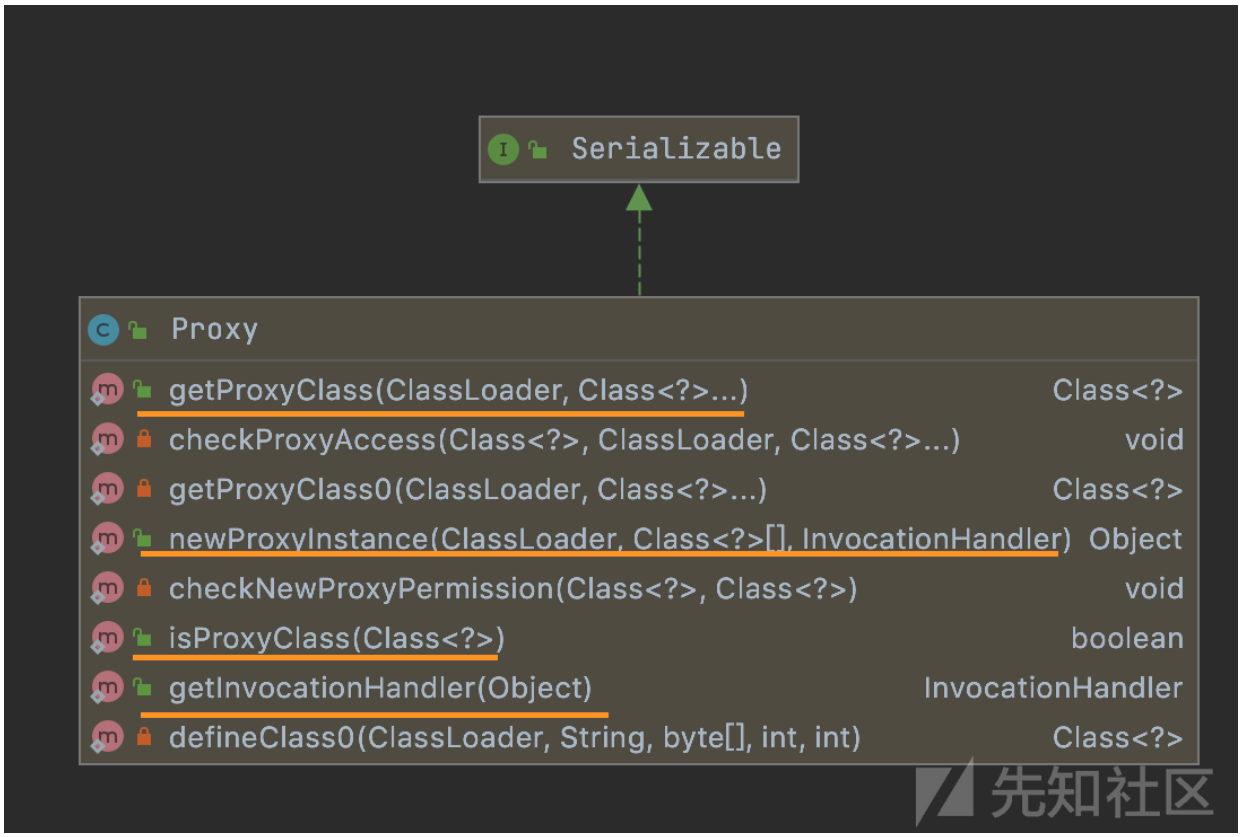
Java动态代理位于Java.lang.reflect包下，我们一般就仅涉及Java.lang.reflect.Proxy类与InvocationHandler接口,使用其配合反射，完成实现动态代理的操作。

InvocationHandler接口：负责提供调用代理操作。

是由代理对象调用处理器实现的接口，定义了一个invoke()方法，每个代理对象都有一个关联的接口。当代理对象上调用方法时，该方法会被自动转发到InvocationHandler.invoke()方法来进行调用。

Proxy类：负责动态构建代理类

提供四个静态方法来为一组接口动态生成的代理类并返回代理类的实例对象。



getProxyClass(ClassLoader,Class<?>...)：获取指定类加载器和动态代理类对象。

newProxyInstance(ClassLoader,Class<?>[],InvocationHandler)：指定类加载器，一组接口，调用处理器；

isProxyClass(Class<?>)：判断获取的类是否为一个动态代理类;

getInvocationHandler(Object)：获取指定代理类实例查找与它相关联的调用处理器实例;

0x05 实现过程

1、使用java.lang.InvocationHandler接口创建自定义调用处理器，由它来实现invoke方法，执行代理函数；

2、使用java.lang.reflect.Proxy类指定一个ClassLoader，一组interface接口和一个InvocationHandler；

3、通过反射机制获得动态代理类的构造方法，其唯一参数类型是调用处理器接口类型；

4、调用java.lang.reflect.Proxy.newProxyInstance()方法，分别传入类加载器，被代理接口，调用处理器；创建动态代理实例对象。

5、通过代理对象调用目标方法；

我们继续使用前面那个例子进行讲解，因为接口类和委托类不用更改，这里就不重复了。

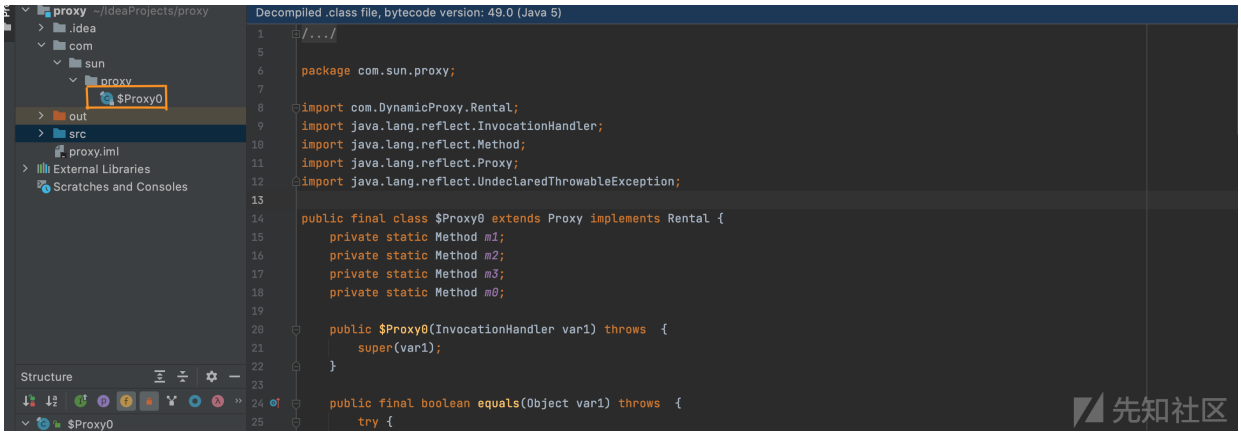
```java
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
public class TestAgent implements InvocationHandler {
    // target变量为委托类对象
    private Object target;
    public TestAgent(Object target) {
        this.target = target;
    }
    // 实现 java.lang.reflect.InvocationHandler.invoke()方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 添加自定义的委托逻辑
        System.out.println("房子出租价位有1k-3k");
        // 调用委托类的方法
        Object result = method.invoke(target,args);
        return result;
    }
}
```

```java
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
public class test {
    public static void main(String[] args) {
        // 获取委托类的实例对象
        Entrust testEntrust = new Entrust();
        // 获取CLassLoader
        ClassLoader classLoader = testEntrust .getClass().getClassLoader();
        // 获取所有接口
        Class[] interfaces = testEntrust .getClass().getInterfaces();
        // 获取一个调用处理器
        InvocationHandler invocationHandler = new TestAgent(testEntrust);
        // 查看生成的代理类
        System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");
        // 创建代理对象
        Rental proxy = (Rental) Proxy.newProxyInstance(classLoader,interfaces,invocationHandler);
        // 调用代理对象的sayHello()方法
        proxy.sale();
    }
}
```

```java
public class test {
    public static void main(String[] args) {
        // 获取委托类的实例对象
        Entrust testEntrust = new Entrust();

        // 获取CLassLoader
        ClassLoader classLoader = testEntrust .getClass().getClassLoader();

        // 获取所有接口
        Class[] interfaces = testEntrust .getClass().getInterfaces();

        // 获取一个调用处理器
        InvocationHandler invocationHandler = new TestAgent(testEntrust );

        // 查看生成的代理类
        System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

        // 创建代理对象
        Rental proxy = (Rental) Proxy.newProxyInstance(classLoader,interfaces,invocationHandler);

        // 调用代理对象的sayHello()方法
        proxy.sale();
    }
}
```

```
test (1)
/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin/java ...
房子出租价位有1k-3k
出租房子
```


这里我们可以看见我们生成的动态代理类的字节码文件，放置在程序根目录下的com.sun.proxy.$Proxy0.class文件中。




```java
import com.DynamicProxy.Rental;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;
public final class $Proxy0 extends Proxy implements Rental {
    // 私有静态构造方法
    private static Method m1;
    private static Method m2;
    private static Method m3;
    private static Method m0;
    // 获取调用处理器实例对象
    public $Proxy0(InvocationHandler var1) throws  {
        super(var1);
    }
    public final boolean equals(Object var1) throws  {
```

```
        try {
            return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }
    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }
    // 获取调用接口类的sale()方法,转发到调用处理器中的invoke()方法进行处理。
    public final void sale() throws  {
        try {
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }
    public final int hashCode() throws  {
        try {
            return (Integer)super.h.invoke(this, m0, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }
    // 利用反射进行类初始化，执行static静态代码块中的内容，主要是获取com.DynamicProxy.Rental接口类中的sale方法。
    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals", Class.forName("java.lang.Object"));
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m3 = Class.forName("com.DynamicProxy.Rental").getMethod("sale");
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

在这里生成的$Proxy0代类中，我们可以清楚知道动态代理的实现过程。实际上我们在创建代理对象时，就是通过通过反射来获取这个类的构造方法，然后来创建的代理实例。

0x06 ysoserial示例

在ysoerial工具中，其实基本很多poc或者exp都运用到了jdk代理机制。那么这个ysoerial工具是怎么实现jdk动态代理的呢？其实和ysoserial.payloads.util.Gadgets.createMemoitizedProxy()有关。

CommonsCollections1.java

```
final Map mapProxy = Gadgets.createMemoitizedProxy(lazyMap, Map.class);
```

RMIRegistryExploit.java

```
Remote remote = Gadgets.createMemoitizedProxy(Gadgets.createMap(name, payload), Remote.class);
```

Spring2.java

```
final Object typeProviderProxy = Gadgets.createMemoitizedProxy(
            Gadgets.createMap("getType", typeTemplatesProxy),
            forName("org.springframework.core.SerializableTypeWrapper$TypeProvider"));
```

比如这上面这三处payloads和exploit都使用了createMemoitizedProxy来创建代理，跟进查看分析。

```
public static <T> T createMemoitizedProxy ( final Map<String, Object> map, final Class<T> iface, final Class<?>...
ifaces ) throws Exception {
       return createProxy(createMemoizedInvocationHandler(map), iface, ifaces);
    }


    public static InvocationHandler createMemoizedInvocationHandler ( final Map<String, Object> map ) throws
Exception {
       return (InvocationHandler) Reflections.getFirstCtor(ANN_INV_HANDLER_CLASS).newInstance(Override.class, map);
    }


    public static <T> T createProxy ( final InvocationHandler ih, final Class<T> iface, final Class<?>... ifaces ) {
        final Class<?>[] allIfaces = (Class<?>[]) Array.newInstance(Class.class, ifaces.length + 1);
        allIfaces[ 0 ] = iface;
        if ( ifaces.length > 0 ) {
            System.arraycopy(ifaces, 0, allIfaces, 1, ifaces.length);
        }
        return iface.cast(Proxy.newProxyInstance(Gadgets.class.getClassLoader(), allIfaces, ih));
    }
```

我们可以对这三个函数进行简单分析下，调用createMemoitizedProxy方法传入一个Map类型对象和多个接口类，然后返回createProxy方法来创建动态代理对象。

createMemoizedInvocationHandler方法传入一个Map类型对象和多个接口类，返回生成自定义调用处理器，实现代理功能。

createProxy方法则是创建动态代理类，并且返回该代理对象实例。

我们将前面的动态代理机制学习和这三个函数进行连接在一起，可以得知就是通过createMemoitizedProxy方法传入一个Map类型对象和多个接口类，然后调用createProxy方法中第一个参数createMemoizedInvocationHandler方法来自定义调用处理器，最后返回createProxy方法创建动态代理类对象。

0x07 总结

因为java代理机制概念理解较于简单，所以网上一些java安全讲解很少关于这块内容进行单独文章讲解。但这个知识点却是java漏洞分析中不可缺少，同时对后面RMI和JDNI注入学习也有很大帮助。如果大家对其有更多兴趣的话，还可以去看看其源码深入了解下。

0x08 参考链接

https://blog.csdn.net/huashanlunjian/article/details/84384279

https://www.guildhab.top/?p=6593

https://blog.csdn.net/JacksonKing/article/details/102974895

关注 | 2　　点击收藏 | 6

上一篇： DA14531芯片固件逆向系列（3...　　下一篇： 【漏洞预警】VMware View...

0 条回复

动动手指，沙发就是你的了！

登录 后跟帖