SSM 前言

主讲：Reyco·郭

北京动力节点教育科技有限公司
动力节点课程讲义
DONGLIJIEDIANKECHENGJIANGYI
www.bjpowernode.com

SSM 前言讲义

第1章 系统架构

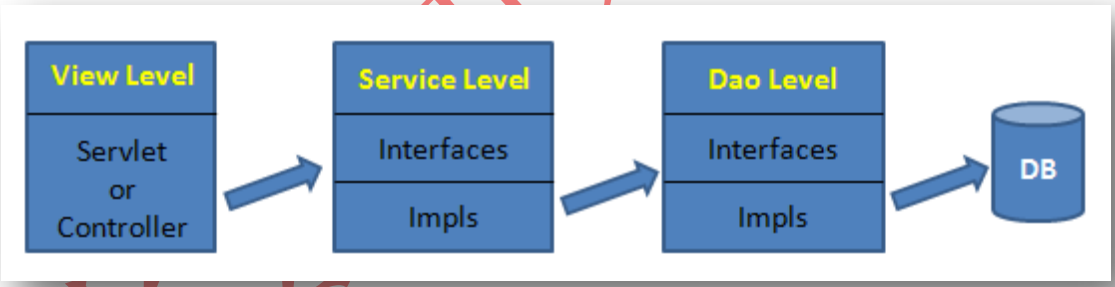
所谓系统架构是指，整合应用系统程序大的结构。经常提到的系统结构有两种：三层架构与 MVC。这两种结构既有区别，又有联系。但这两种结构的使用，均是为了降低系统模块间的耦合度。

1.1 三层架构

三层架构是指：视图层 View、服务层 Service，与持久层 Dao。它们分别完成不同的功能。

- View 层：用于接收用户提交请求的代码在这里编写。
- Service 层：系统的业务逻辑主要在这里完成。
- Dao 层：直接操作数据库的代码在这里编写。

为了更好的降低各层间的耦合度，在三层架构程序设计中，采用面向抽象编程。即上层对下层的调用，是通过接口实现的。而下层对上层的真正服务提供者，是下层接口的实现类。服务标准（接口）是相同的，服务提供者（实现类）可以更换。这就实现了层间解耦合。



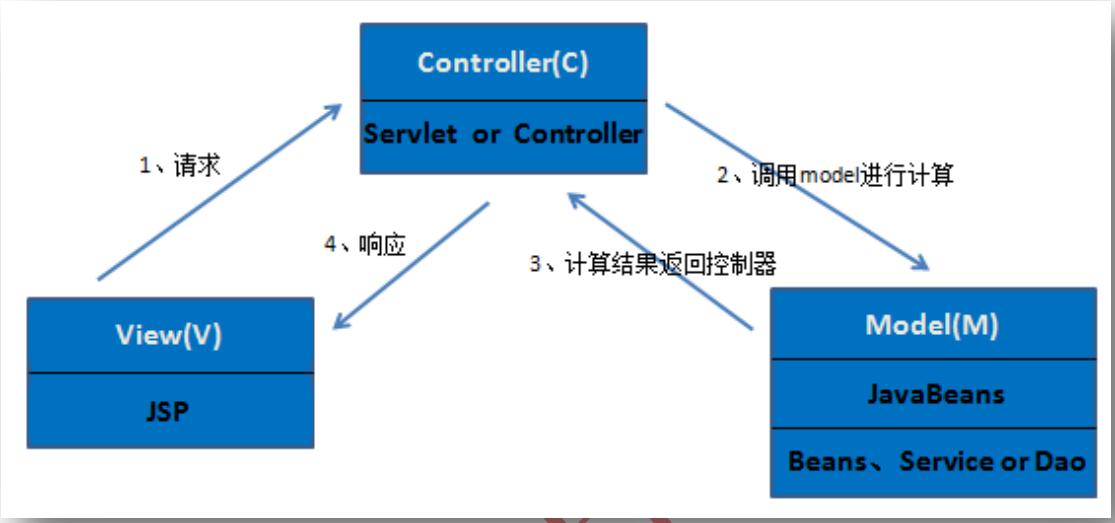
1.2 MVC

MVC，即 Model 模型、View 视图，及 Controller 控制器。

- View：视图，为用户提供使用界面，与用户直接进行交互。
- Model：模型，承载数据，并对用户提交请求进行计算的模块。其分为两类，一类称为数据承载 Bean，一类称为业务处理 Bean。所谓数据承载 Bean 是指实体类，专门用户承载业务数据的，如 Student、User 等。而业务处理 Bean 则是指 Service 或 Dao 对象，专门用于处理用户提交请求的。
- Controller：控制器，用于将用户请求转发给相应的 Model 进行处理，并根据 Model 的计算结果向用户提供相应响应。

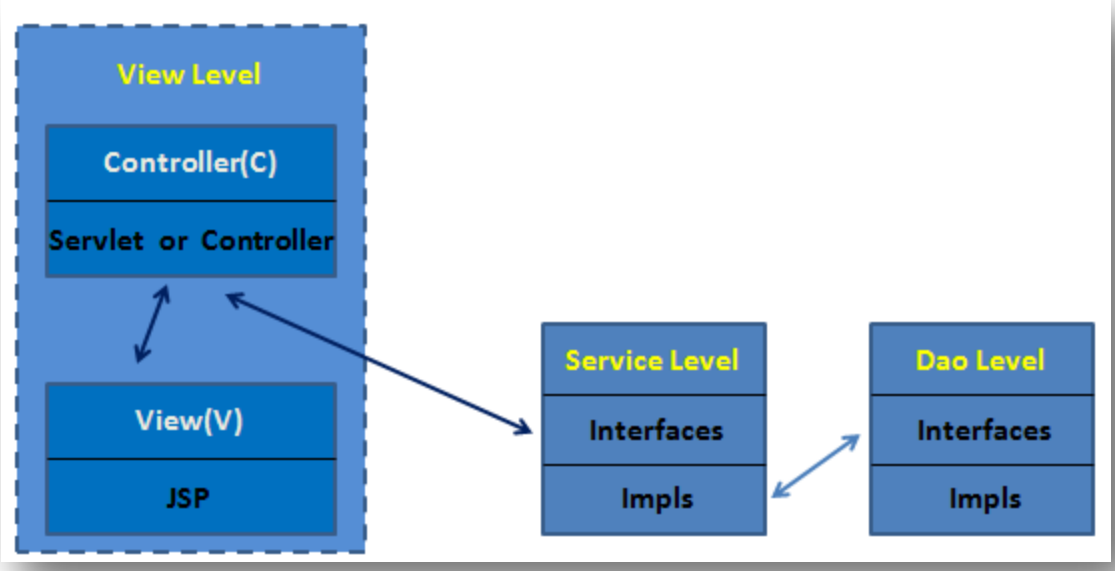
MVC 架构程序的工作流程是这样的：

（1）用户通过 View 页面向服务端提出请求，可以是表单请求、超链接请求、AJAX 请求等
（2）服务端 Controller 控制器接收到请求后对请求进行解析，找到相应的 Model 对用户请求进行处理
（3）Model 处理后，将处理结果再交给 Controller
（4）Controller 在接到处理结果后，根据处理结果找到要作为向客户端发回的响应 View 页面。页面经渲染（数据填充）后，再发送给客户端。



1.3 MVC 与三层架构的关系

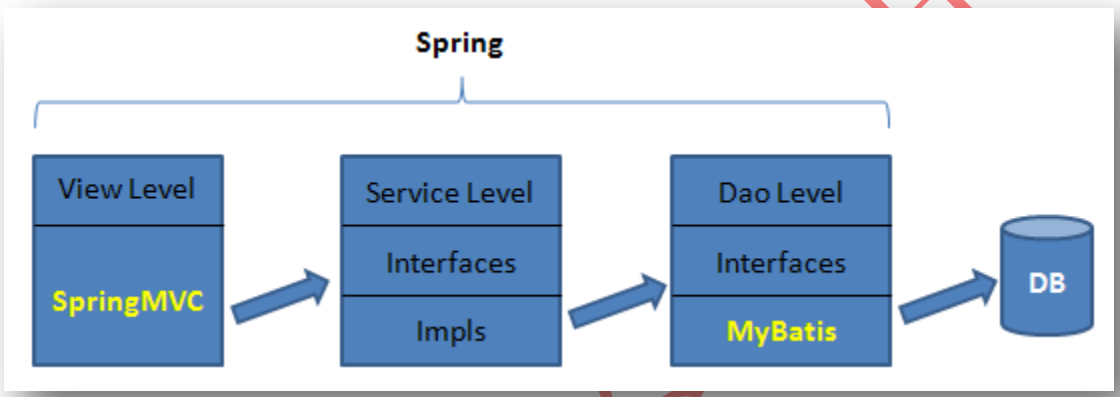
MVC 与三层架构很相似，但它们并不一样。



1.4 SSM 与三层架构的关系

SSM，即 SpringMVC、Spring 与 MyBatis 三个框架。它们在三层架构中所处的位置是不同的，即它们在三层架构中的功能各不相同，各司其职。

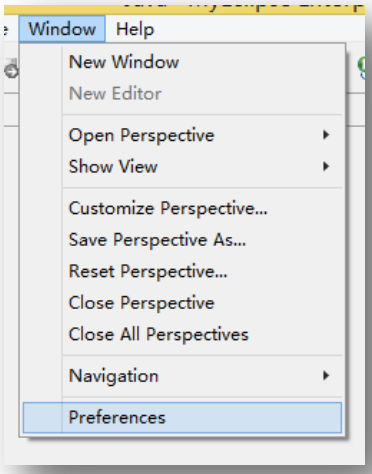
- SpringMVC：作为 View 层的实现者，完成用户的请求接收功能。SpringMVC 的 Controller 作为整个应用的控制器，完成用户请求的转发及对用户的响应。
- MyBatis：作为 Dao 层的实现者，完成对数据库的增、删、改、查功能。
- Spring：以整个应用大管家的身份出现。整个应用中所有 Bean 的生命周期行为，均由 Spring 来管理。即整个应用中所有对象的创建、初始化、销毁，及对象间关联关系的维护，均由 Spring 进行管理。



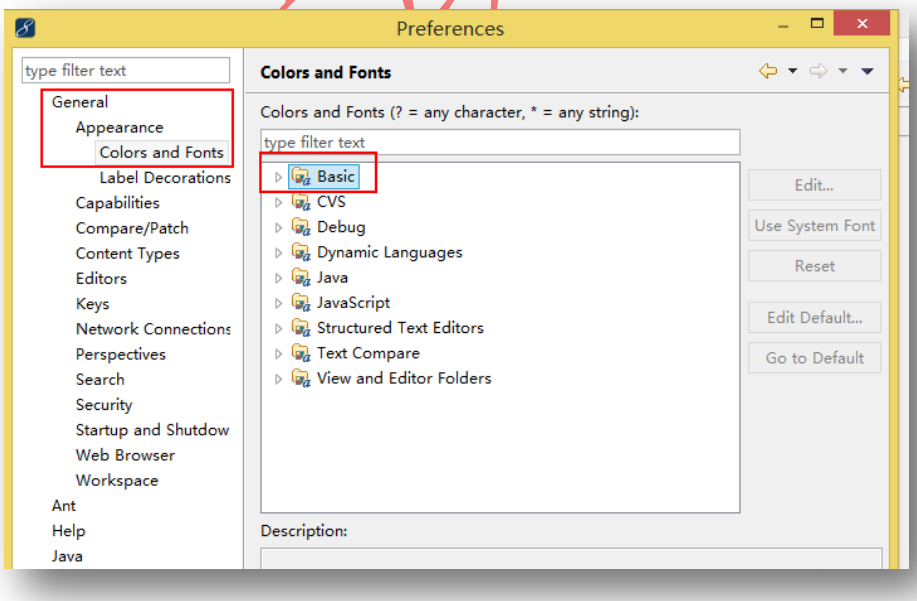
第2章 环境设置

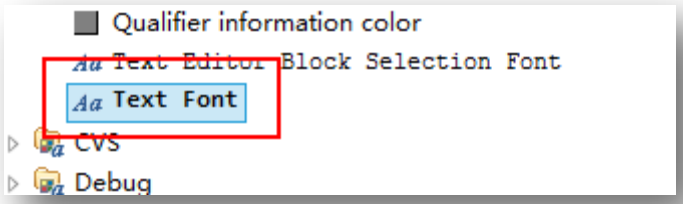
2.1 MyEclipse 环境设置

MyEclipse 的相关属性设置，一般在 Window/Preferences 下。

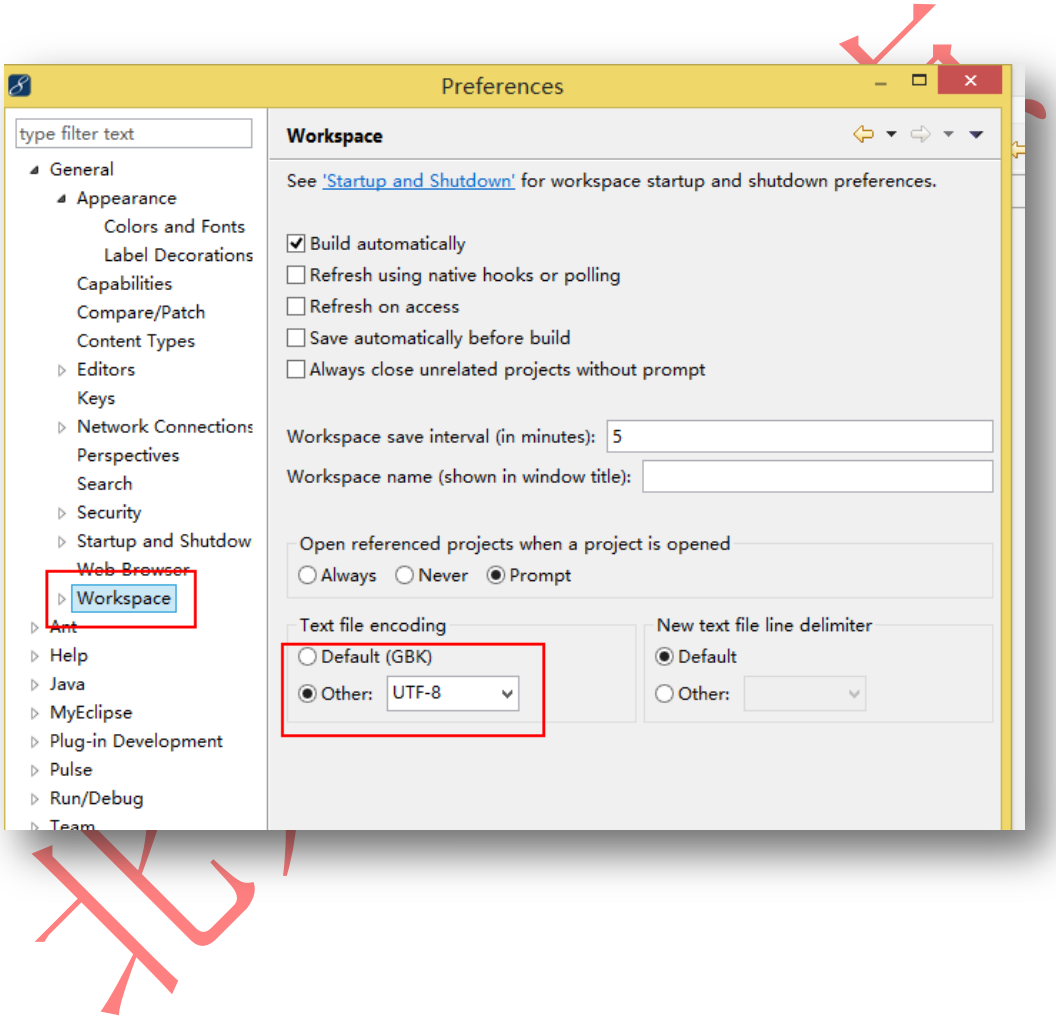


2.1.1 字体设置

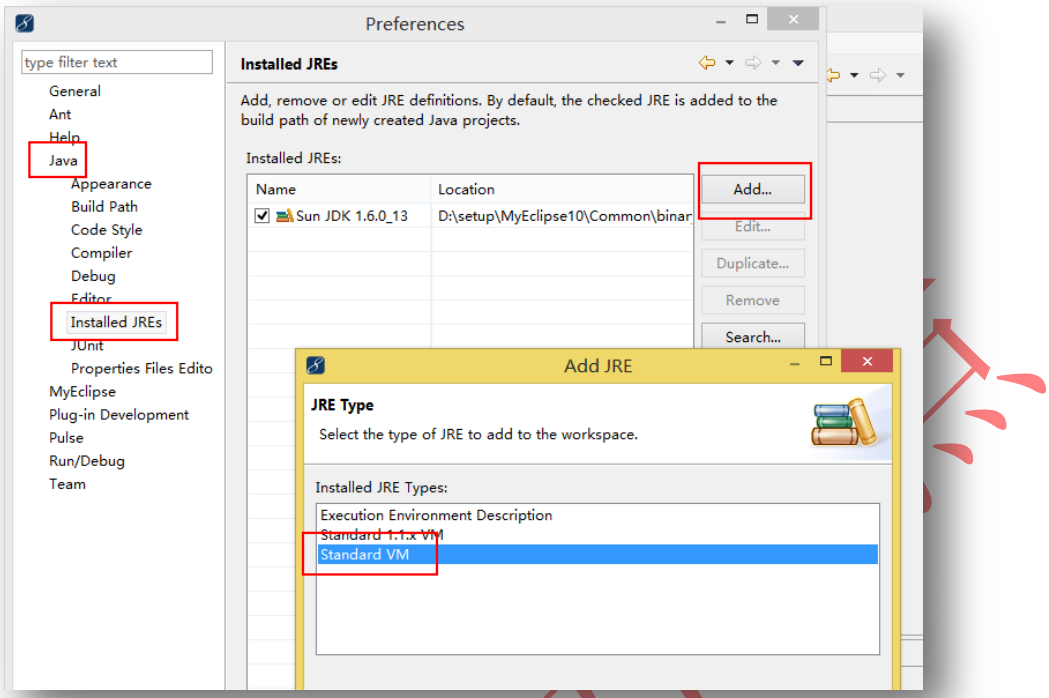




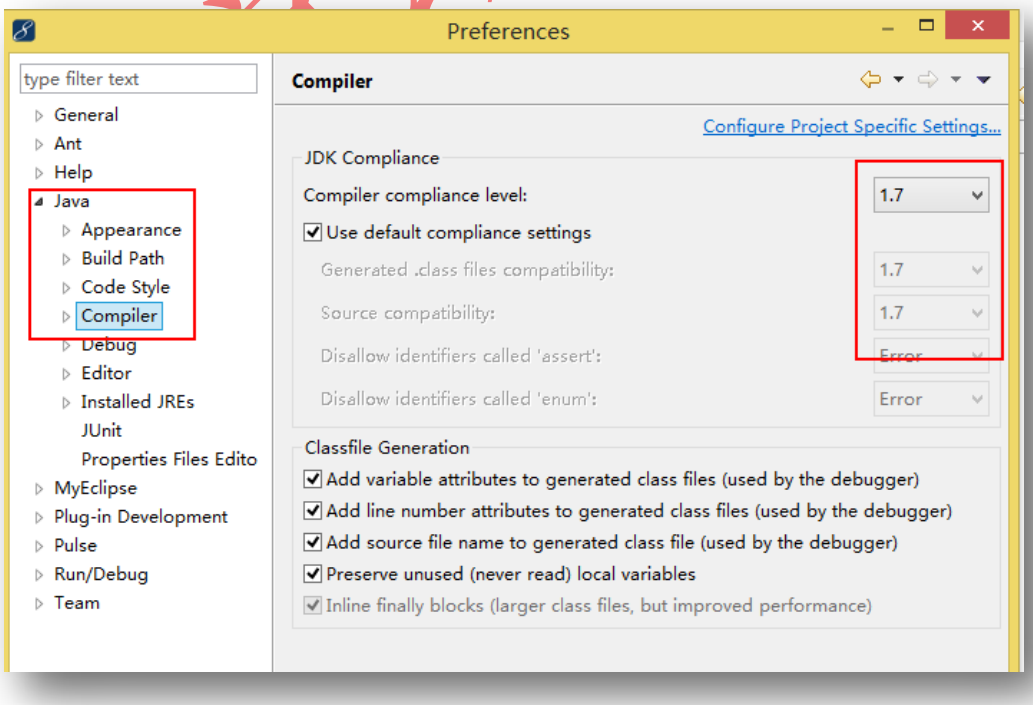
2.1.2 workspace 字符集设置



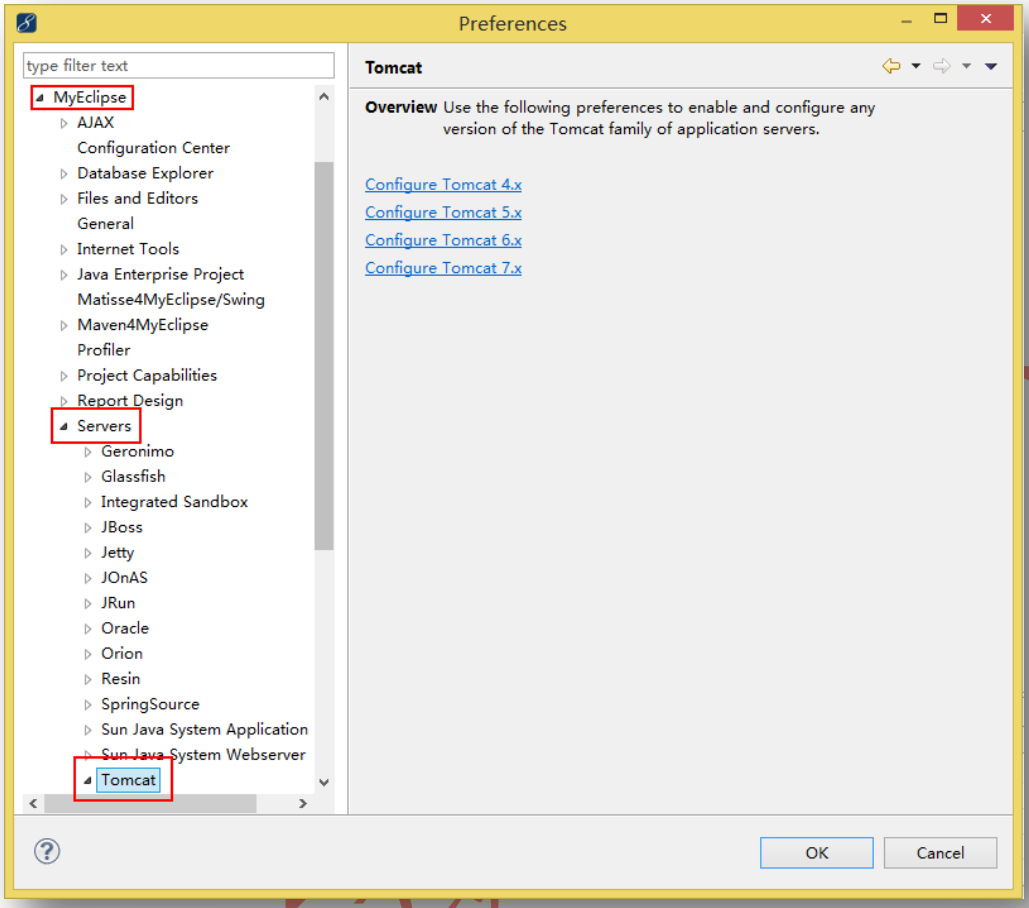
2.1.3 JDK 更换设置



2.1.4 默认编译器设置



2.1.5 Tomcat 服务器设置

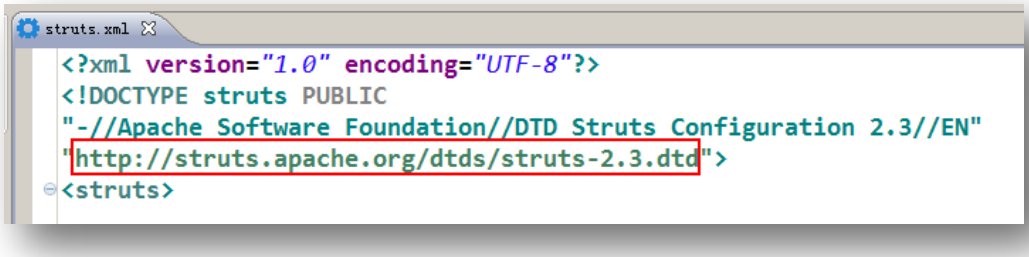


2.1.6 XML 中没有自动提示功能

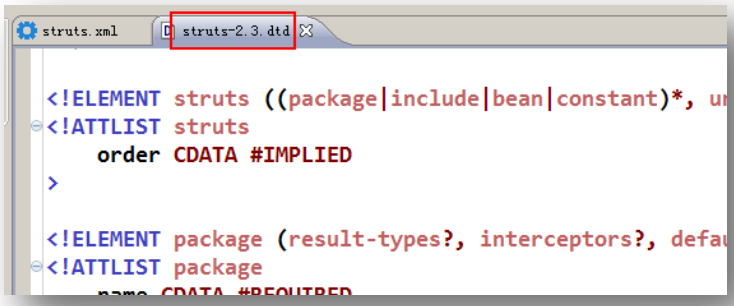
对于 xml 配置文件，若当前主机连着互联网，那么，原则上在编写 xml 配置文件时，使用 ALT + ?，可出现自动提示。但，有时由于网速等原因，或根本就没有联网，没有自动提示。此时，可通过在 MyEclipse 中进行相关设置，使自动提示出现。

下面以 Struts2 中核心配置文件 struts.xml 的自动提示为例进行描述。

xml 配置文件中的文件头中指定的是该 xml 文件的约束。该约束默认情况下，会从互联网上查找相应的 dtd 或 xsd 约束文件。



约束文件中规定了该类型的 xml 文件中可以包含的标签，及标签的写书顺序等。Struts2 的约束文件 struts-2.3.dtd 的部分内容如下：

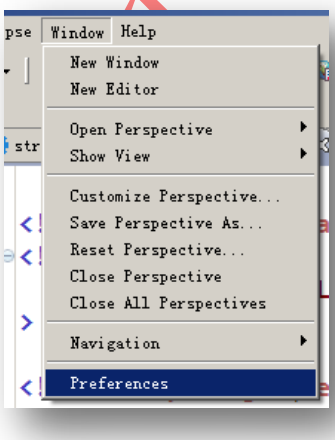


只有当前 MyEclipse 能够找到相应的约束文件，才可以根据约束文件给出自动提示。若当前系统没有连网，或网速有问题，则自动提示将会出问题。此时可通过指定让该 URI 不到互联网上查询约束文件，而从本地查找。所以，解决自动提示问题，首先要有约束文件。

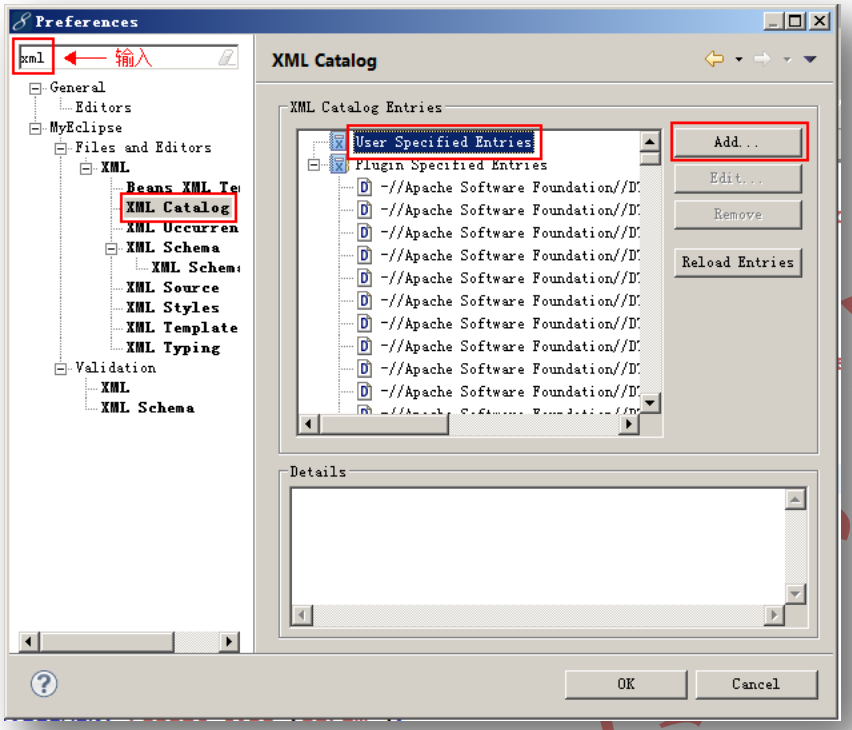
Struts2 的约束文件 struts-2.3.dtd 在 Struts2 的核心 Jar 包中。将该文件从 Jar 包中拷贝出来，放到任意位置。然后，在 MyEclipse 中进行设置：

（1） 拷贝 xml 文件头的约束 URL

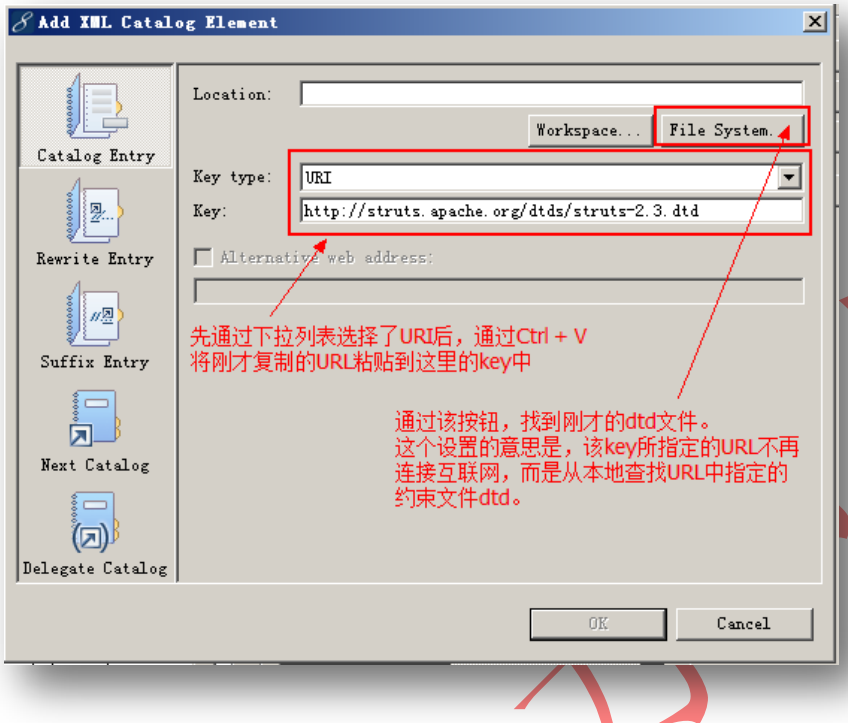
（2） Window → Preferences



（3） 输入 xml 搜索



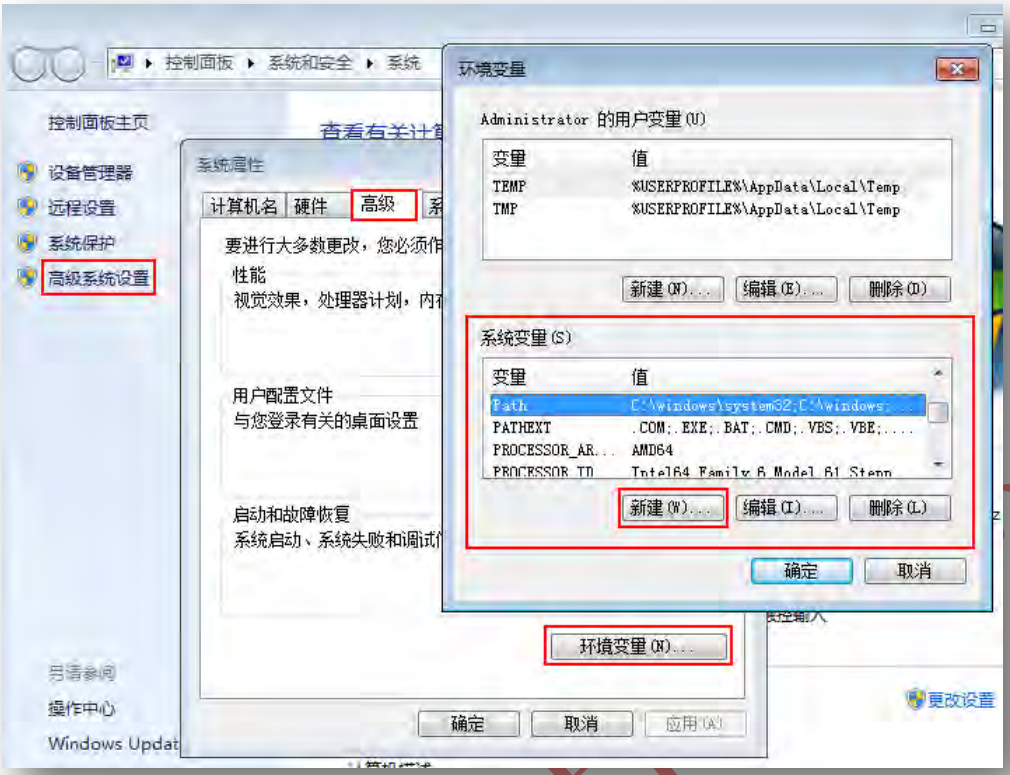
（4） 指定 URL 对应的本地文件



2.2 Tomcat 设置

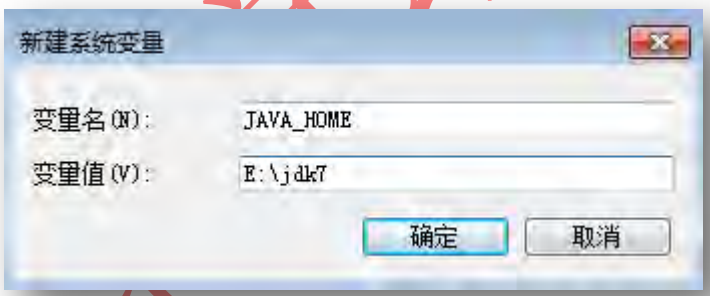
2.2.1 Tomcat 启动环境设置

为了使 Tomcat 能够在命令行启动，即不是在 MyEclipse 下启动。则必须要在环境变量中设置 JAVA_HOME 与 CATALINA_HOME。而这些变量的设置是在“我的电脑”上右击，选择“属性”，再按如下操作。



（1） 设置 JAVA_HOME

设置 JAVA_HOME，其值指定 JDK 的安装主目录。



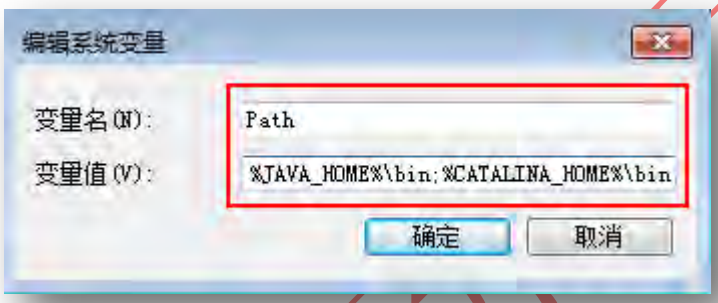
（2） 设置 CATALINA_HOME

以相同的方式设置 CATALINA_HOME，其值设置为 Tomcat 的主目录。

（3） 在 path 变量中添加 bin 目录

在“系统变量”的 Path 变量中添加 Tomcat 的主目录下的 bin 目录，为了能在命令行的任意位置可以直接运行 Tomcat 的启动命令 startup.bat 与关闭命令 shutdown.bat。

直接双击 Path 变量，在“变量值”的最后，添加如下内容即可。



2.2.2 设置 Tomcat 默认字符集

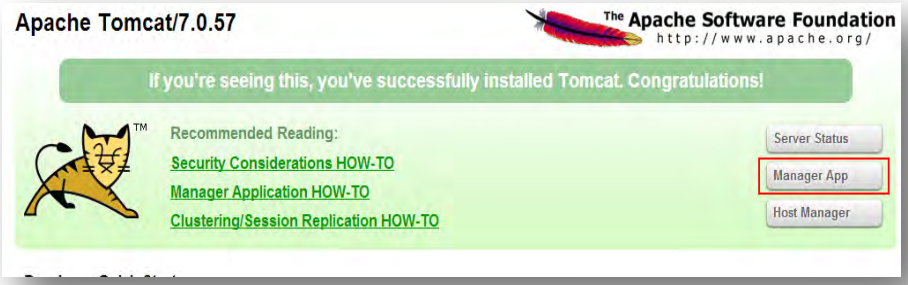
为了解决 GET 提交时的中文乱码问题，可在 Tomcat 中作如下设置：打开 Tomcat 安装目录下的 conf 中的 server.xml 文件，在如下位置添加 URIEncoding=“UTF-8”。

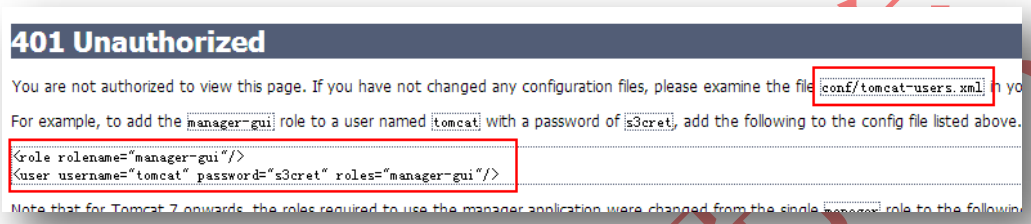
```
    Define a non-SSL HTTP/1.1 Connector on port 8080
-->
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" URIEncoding="UTF-8"/>
<!-- A "Connector" using the shared thread pool-->
```

2.2.3 设置管理应用的登录用户名与密码

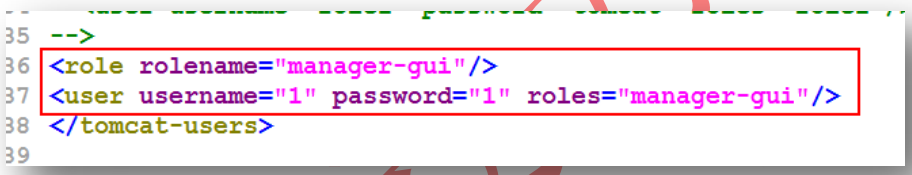
为了方便测试时对项目的访问，可通过 Tomcat 的应用管理窗口“Manager App”进行访问管理。

设置内容在点击如下按钮后，点取消，可看到提示设置登录用户名与密码的方法。





所以，打 Tomcat 安装目录下的 conf 中的 tomcat-users.xml 文件，在最后添加如下内容，将用户名与密码均设置为 1。



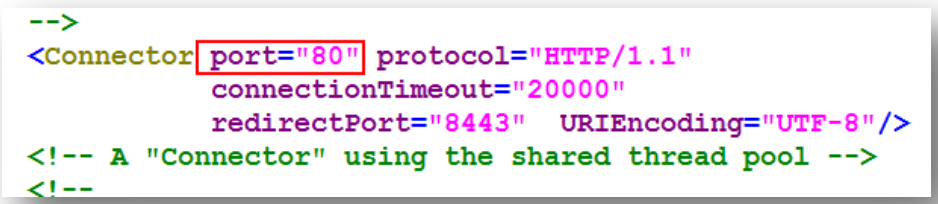
```
35 -->
36 <role rolename="manager-gui"/>
37 <user username="1" password="1" roles="manager-gui"/>
38 </tomcat-users>
39
```

2.2.4 Web 项目的部署

（1） 端口号的问题

Tomcat 默认访问的端口号为 8080，而真正上线后，是无需输入端口号的。因为浏览器默认访问的服务端端口号是 80。所以，将 Tomcat 默认的端口号修改为 80 即可使用户在访问时，不用再提交端口号了。

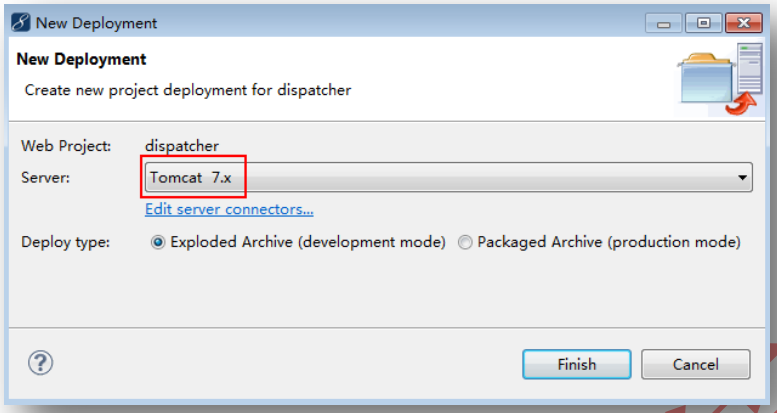
在 Tomcat 主目录下的 conf/server.xml 文件中修改。



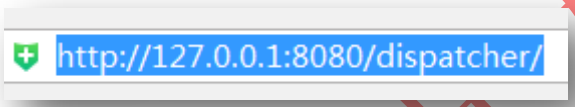
```
-->
<Connector port="80" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443"  URIEncoding="UTF-8"/>
<!-- A "Connector" using the shared thread pool -->
<!--
```

（2） 项目名称的问题

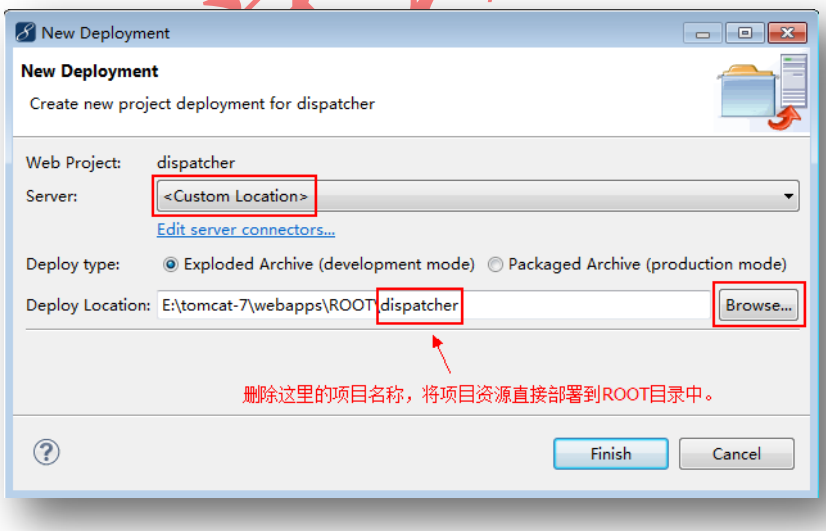
在程序的开发阶段，对代码进行调试时，一般是通过如下方式将项目部署到 Tomcat 主目录下的 webapps 目录下。



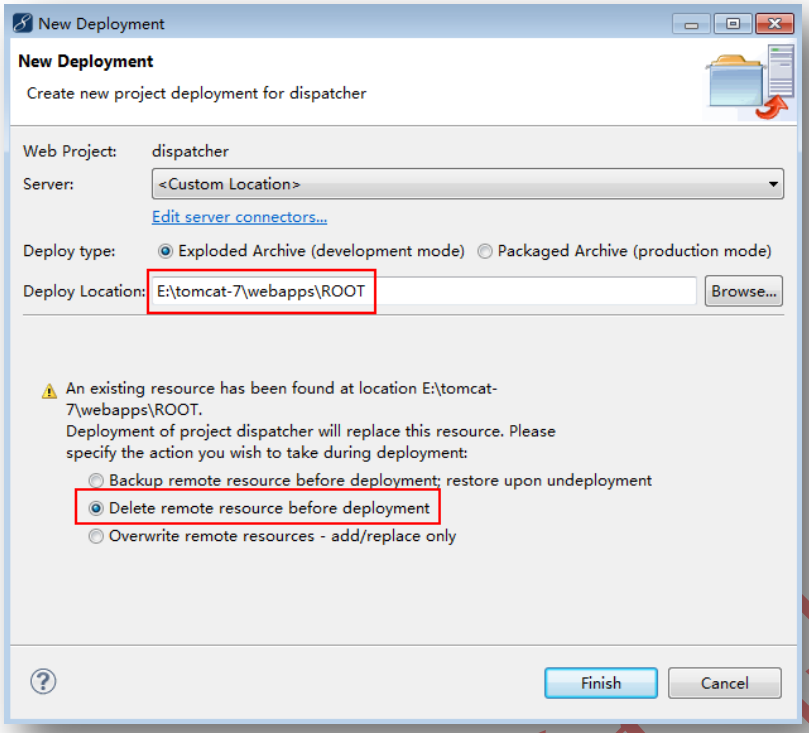
访问时在浏览器地址栏输入如下形式 URL 即可访问。即端号号后需要指定要访问哪个项目下的资源，即项目名称。



但真正在项目上线后，是不用让用户输入项目名称的。此时，需要将项目按照如下方式部署到 Tomcat 的 webapps 下的 ROOT 目录中。



选择“删除之前部署的资源”选项即可。再访问时，无需再输入项目名称了。



第3章 代理模式

> 代理模式是指，为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户类和目标对象之间起到中介的作用。
>
> 百度百科《代理模式》

换句话说，使用代理对象，是为了在不修改目标对象的基础上，增强主业务逻辑。

客户类真正的想要访问的对象是目标对象，但客户类真正可以访问的对象是代理对象。客户类对目标对象的访问是通过访问代理对象来实现的。当然，代理类与目标类要实现同一个接口。

以生活中的“代理律师”为例来理解“代理模式”。

打官司是件非常麻烦的过程：案件调查取证、查找法律条文、起草法律文书、法庭辩论、签署法律文件、申请法院执行.....但打官司的人只关心法宣判结果。此时，打官司的人就可聘请“代理律师”来完成整个打官司的所有事务。当事人只需与代理律师签订了“全权

委托协议”，那么整个打官司的过程，当事人均可不出现。

聘请代理律师时以下几点需要特别说明：

（1）当事人与代理律师是在为同一件事情努力。

（2）法院所有的具体事务都是通过代理律师完成。

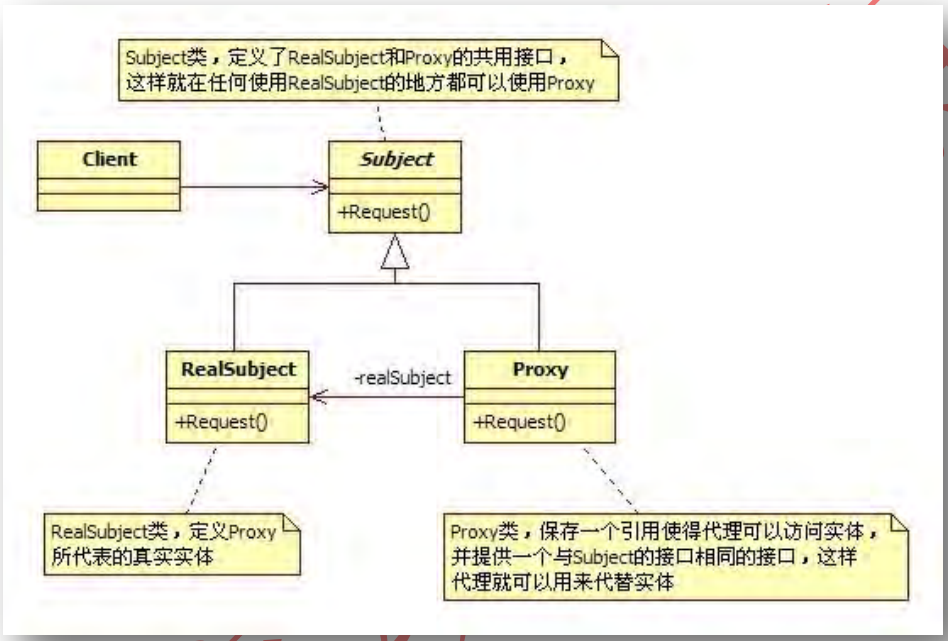
（3）在法院需要当事人完成某些工作时，代理律师会通知当事人，并为其出谋划策，即增强当事人。

对于代理模式，需要注意以下几点：

（1）代理类和目标类要实现同一个接口，即业务接口。

（2）客户类对目标类的调用均是通过代理类完成的。

（3）代理类的执行既执行了对目标类的增强业务逻辑，又调用了目标类的主业务逻辑。



根据代理关系建立的时间不同，可以将代理分为两类：静态代理与动态代理。就好比当事人与律师之间的关系一样，也分为法律顾问与代理律师两类。

3.1 静态代理

3.1.1 概念

静态代理是指，代理类在程序运行前就已经定义好，其与目标类的关系在程序运行前就已经确立。

静态代理类似于企业与企业的法律顾问间的关系。法律顾问与企业的代理关系，并不是在“官司”发生后才建立的，而是之前就确立好的一种关系。

3.1.2 代理实现与解析

静态代理实现转账： ---- proxy_static

（1）定义业务接口 IAccountService，其中含有抽象方法 transfer()。

```
public interface IAccountService {
    // 主业务逻辑：转账
    void transfer();
}
```

（2）定义目标类 AccountServiceImpl，该类实现了业务接口。在对接口方法的实现上，只实现主业务逻辑转账即可。这个方法称为目标方法。

```
public class AccountServiceImpl implements IAccountService {
    /*
     * 目标方法
     */
    @Override
    public void transfer() {
        System.out.println("调用Dao层，完成转账主业务。");
    }
}
```

（3）定义代理类 AccountProxy。这个类要实现 IAccount 接口。并且该代理类要将接口对象作为一个成员变量，还要定义一个带参的构造器，这个参数为接口对象。目的是，将目标对象引入代理类，以便代理类调用目标类的目标方法。

```java
/*
 * 定义代理类，与目标类实现相同的业务接口
 */
public class AccountServiceImplProxy implements IAccountService {
    // 声明业务接口对象
    private IAccountService target;
    public AccountServiceImplProxy() {
    }
    // 业务接口对象作为构造器参数，用于接收目标对象
    public AccountServiceImplProxy(IAccountService target) {
        this.target = target;
    }
    /*
     * 代理方法，实现对目标方法的功能增强
     */
    @Override
    public void transfer() {
        // 此处为对目标类的增强
        System.out.println("对转账人身份进行验证！");
        target.transfer();
    }
}
```

（4）定义客户类 Client。在客户类中首先要创建目标对象，再创建代理对象，并使用目标对象对其进行初始化。然后由代理对象来调用执行业务方法。

```java
public class MyTest {

    public static void main(String[] args) {
        // 创建目标对象
        IAccountService target = new AccountServiceImpl();
        // 创建代理对象，并使用目标对象初始化它
        IAccountService service = new AccountServiceImplProxy(target);
        // 此时执行的内容，就是对目标对象增加过的内容
        service.transfer();
    }

}
```

3.2 JDK 动态代理

动态代理是指，程序在整个运行过程中根本就不存在目标类的代理类，目标对象的代理对象只是由代理生成工具（如代理工厂类）在程序运行时由 JVM 根据反射等机制动态生成的。代理对象与目标对象的代理关系在程序运行时才确立。

对比静态代理，静态代理是指在程序运行前就已经定义好了目标类的代理类。代理类与目标类的代理关系在程序运行之前就确立了。

3.2.1 概念

动态代理类似于普通当事人与聘请的律师间的关系。律师是在“官司”发生后，才由当

事人聘请的。即代理关系是在“官司”发生后才确立的。

动态代理的实现方式常用的有两种：使用 JDK 的 Proxy，与通过 CGLIB 生成代理。

通过 JDK 的 java.lang.reflect.Proxy 类实现动态代理，会使用其静态方法 newProxyInstance()，依据目标对象、业务接口及业务增强逻辑三者，自动生成一个动态代理对象。

```
public static newProxyInstance ( ClassLoader loader, Class<?>[] interfaces,
                                  InvocationHandler handler)
```

loader：目标类的类加载器，通过目标对象的反射可获取

interfaces：目标类实现的接口数组，通过目标对象的反射可获取

handler：业务增强逻辑，需要再定义。

InvocationHandler 是个接口，其具体介绍如下：

实现了 InvocationHandler 接口的类用于加强目标类的主业务逻辑。这个接口中有一个方法 invoke()，具体加强的代码逻辑就是定义在该方法中的。程序调用主业务逻辑时，会自动调用 invoke()方法。

invoke()方法的介绍如下：

```
public Object invoke ( Object proxy, Method method, Object[] args)
```

proxy：代表生成的代理对象

method：代表目标方法

args：代表目标方法的参数

由于该方法是由代理对象自动调用的，所以这三个参数的值不用程序员给出。

第二个参数为 Method 类对象，该类有一个方法也叫 invoke()，可以调用目标类的目标方法。这两个 invoke()方法，虽然同名，但无关。

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

obj：表示目标对象

args：表示目标方法参数，就是其上一层 invoke 方法的第三个参数

该方法的作用是：调用执行 obj 对象所属类的方法，这个方法由其调用者 Method 对象确定。

在代码中，一般的写法为

method.invoke(target, args);

其中，method 为上一层 invoke 方法的第二个参数。这样，即可调用了目标类的目标方法。

3.2.2 代理实现与解析

动态代理实现转账： ---- proxy_dynamic

（1）定义业务接口 IAccounSevice，其中含有抽象方法 transfer()。

（2）定义目标类 AccountServiceImpl，该类实现了业务接口。在对接口方法的实现上，只实

现主业务逻辑。这个方法称为目标方法。

以上两步与静态代理类 staticproxy 中代码相同。

（3）定义主业务增强逻辑类 MyExtension，该类需实现接口 InvocationHandler。在该类中定义一个 Object 类型的成员变量，还要定义一个带参的构造器，这个参数为 Object 对象。目的是，将目标对象引入该类，以便通过反射调用目标方法。

当然，也可以将这个属性定义为 IAccount 接口类型，但，最好不要这样做，最好将其定义为 Object。因为，这样这个主业务增强逻辑可以适用于本项目中的任何类型的目标类，而不仅仅拘泥于某一个类。

```
/*
 * 定义主业务增强逻辑
 */
public class MyExtension implements InvocationHandler {
    private Object target;

    public MyExtension() {
    }
    public MyExtension(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        // 增加主业务逻辑代码
        System.out.println("对转账用户进行身份验证");
        // 无论主业务方法有无参数，有无返回值，下面的写法均可兼顾到
        return method.invoke(target, args);
    }
}
```

（4）定义客户类 Client。客户类中主要语句有三句：

A、定义目标对象。在生成代理对象时会需要目标对象对其初始化。

B、定义代理对象。需要注意的是，代理类 Proxy 会通过反射机制，自动实现 IAccount 接口。代理对象需要使用目标对象对其进行初始化。

C、代理对象调用主业务方法。

```
public class MyTest {

    public static void main(String[] args) {
        // 定义目标对象
        IAccountService target = new AccountServiceImpl();
        // 创建代理对象，并使用目标对象初始化它
        IAccountService service = (IAccountService) Proxy.newProxyInstance(
                        target.getClass().getClassLoader(), // 获取目标对象的类加载器
                        target.getClass().getInterfaces(),  // 获取目标类实现的所有接口
                        new MyExtension(target));           // 增强业务逻辑
        // 此时执行的内容，就是对目标对象增加过的内容
        service.transfer();
    }

}
```

3.3 CGLIB 动态代理

3.3.1 概念

使用 JDK 的 Proxy 实现代理，要求目标类与代理类实现相同的接口。若目标类不存在接口，则无法使用该方式实现。

但对于无接口的类，要为其创建动态代理，就要使用 CGLIB 来实现。CGLIB 代理的生成原理是生成目标类的子类，而子类是增强过的，这个子类对象就是代理对象。所以，使用 CGLIB 生成动态代理，要求目标类必须能够被继承，即不能是 final 的类。

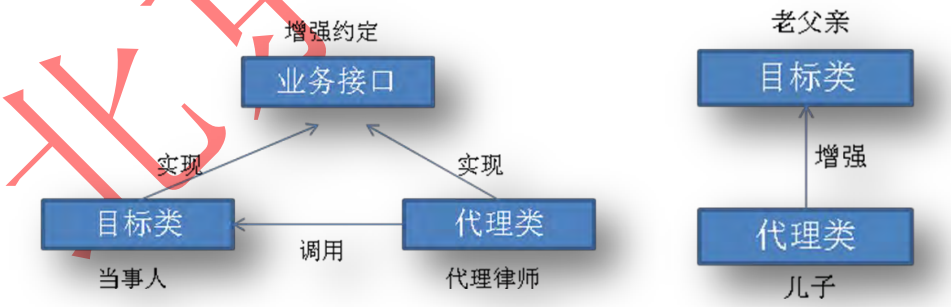
> CGLIB(Code Generation Library)是一个开源项目，是一个强大的、高性能的、高质量的代码生成类库。它可以在运行期扩展和增强 Java 类。Hibernate 用它来实现持久对象的字节码的动态生成，Spring 用它来实现 AOP 编程。
>
> ---- 百度百科《CGLIB》

CGLIB 包的底层是通过使用一个小而快的字节码处理框架 ASM(Java 字节码操控框架)，来转换字节码并生成新的类。CGLIB 是通过对字节码进行增强来生成代理的。

如果说静态代理是“法律顾问（律师）”，动态代理是“代理律师”，那么 CGLIB 代理则是“老父亲的儿子”。

当事人是打官司的人，而律师也是打相同官司的人。但他们事先有个约定，就是律师只对官司相关的事情进行“增强”。这个约定是通过“接口”来实现的。这种是 JDK 的代理原理。

老父亲是需要被增强的目标类，而儿子则是用于增强老父亲的代理类。不同于当事人与律师关系的是，老父亲与儿子间对于要增强的事情，事先是没有约定的。老父亲需要儿子帮助他做什么事情，即增强哪些方法，事先是没有约定的。老父亲让其增强什么，儿子就需要增强什么。即，他们是不需要“接口”的。



3.3.2 代理实现与解析

使用 CGLIB 创建代理步骤： ---- cglibproxy

Setp1：导入 CGLIB 的 Jar 包： cglib-full.jar。

Step2：定义目标类。注意不用实现任何接口。

```java
// 目标类
public class AccountService {
    //目标方法
    public void transfer() {
        System.out.println("调用Dao层，完成转账主业务。");
    }
    //目标方法
    public void getBalance() {
        System.out.println("调用Dao层，完成查询余额主业务。");
    }
}
```

Step3：创建代理类的工厂。该类要实现 MethodInterceptor 接口。该类中完成三样工作：

```java
// CGLIB代理类的生成工厂。CGLIB代理生成原理是生成业务类的子类。子类是增强过的。
public class AccountServiceCglibProxyFactory implements MethodInterceptor {
    //  （1）声明目标类的成员变量，并创建以目标类对象为参数的构造器。用于接收目标对象。
    //  （2）定义代理的生成方法，用于创建代理对象。代理对象即目标类的子类。
    //  （3）定义回调接口方法。对目标类的增强这在这里完成。
}
```

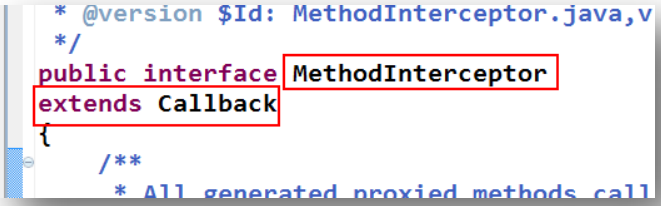
（1）声明目标类的成员变量，并创建以目标类对象为参数的构造器。用于接收目标对象。

```java
public class AccountServiceCglibProxyFactory implements MethodInterceptor {
    //  （1）声明目标类的成员变量，并创建以目标类对象为参数的构造器。用于接收目标对象。
    private AccountService target;

    public AccountServiceCglibProxyFactory(AccountService target) {
        this.target = target;
    }
```

（2）定义代理的生成方法，用于创建代理对象。方法名是任意的。代理对象即目标类的子类。

```java
// 定义代理的创建方法，方法名随意
AccountService createProxy(){
    // 创建增强器
    Enhancer enhancer = new Enhancer();
    // 初始化增强器：将目标类指定为父类
    enhancer.setSuperclass(AccountService.class);
    // 初始化增强器：设置回调
    enhancer.setCallback(this);
    // 使用增强器创建代理对象
    return (AccountService) enhancer.create();
}
```

注意，只所以在 setCallback()方法中可以写上 this，是因为 MethodIntecepter 接口继承自 Callback，是其子接口。查看源码：



```
 * @version $Id: MethodInterceptor.java,v
 */
public interface MethodInterceptor
extends Callback
{
    /**
     * All generated proxied methods call
```

（3）定义回调接口方法。对目标类的增强这在这里完成。

```
@Override
public Object intercept(Object proxy, Method method, Object[] args,
        MethodProxy methodProxy) throws Throwable {
    // 若为transfer方法，则进行增强
    if("transfer".equals(method.getName())){
        System.out.println("开始时间：" + System.currentTimeMillis());
        // 调用目标类业务方法方式一：通过调用代理类proxy，即目标类的子类的父类方法执行
        // Object result = methodProxy.invokeSuper(proxy, args);
        // 调用目标类业务方法方式二：直接调用目标对象的业务方法执行。
        // 以上两种方式等效。
        Object result = method.invoke(target, args);
        System.out.println("结束时间：" + System.currentTimeMillis());
        return result;
    }
    // 其它方法不增强，则执行目标类的业务方式，即代理类的父类业务方法
    //return methodProxy.invokeSuper(proxy, args);
    return method.invoke(target, args);
}
```

intercept()方法中各参数的意义：

* proxy：代理对象
* metho：代理对象的方法，即增强过的业务方法
* args[]：方法参数
* methodProxy：代理对象方法的代理对象

Step4：创建测试类。

```
public static void main(String[] args) {
    // 创建目标对象
    AccountService target = new AccountService();
    // 创建代理对象，并使用目标对象初始化它
    AccountService service = new AccountServiceCglibProxyFactory(target).createProxy();
    // 此时执行的内容，就是对目标对象增加过的内容
    service.transfer();
    service.getBalance();
}
```

3.3.3 方法回调设计模式

在 Java 中，就是类 A 调用类 B 中的某个方法 b，然后类 B 又在某个时候反过来调用类 A 中的某个方法 a，对于 A 来说，这个 a 方法便叫做回调方法。Java 的接口提供了一种很好的方式来实现方法回调。这个方式就是定义一个简单的接口，在接口之中定义一个我们希望回调的方法。这个接口称为回调接口。

在前面的例子中，我们定义的 AccountServiceCglibProxyFactory 类就相当于前面所说的 A 类，而 Enhancer 类则是 B 类。A 类中调用了 Enhancer 类的 setCallback(this)方法，并将回调对象 this 作为实参传递给了 Enhancer 类。Enhancer 类在后续执行过程中，会调用 A 类中的 intercept()方法，这个 intercept()方法就是回调方法。

第4章 适配器模式

适配器模式的定义是，Convert the interface of a class into another interface clients expect，将某个类的接口转换为接口客户所需的类型。换句话说，适配器模式解决的问题是，使得原本由于接口不兼容而不能一起工作、不能统一管理的那些类可以在一起工作、可以进行统一管理。

这样解释适配器的概念还是很抽象。下面以不同工种的工作内容不同，实现统一管理为例来解释适配器设计模式。

需求：厨师的工作是 cook()，程序员的工作是 program()，司机的工作是 drive()，教师的工作是 teach()，不同的工种，其具体工作内容不同。现在程序要将这些（例如有 30 个不同工种）不同工种的工作内容全部输出。

解决方案一：逐个访问每个工种对象的相应工作方法。无法循环遍历，无法统一管理。

解决方案二：使用模板设计模式，将这些不兼容的具体工作转换为一个统一的工作，实现循环遍历。

4.1 基本元素定义

这里举个适配器模式的例子：这里有厨师接口 ICooker、程序员接口 IProgrammer，分别用于定义他们各自工种的具体工作。然后又定义了全聚德的厨师 QjdCooker、京东的程序员 JdProgrammer。这些不同的工种所做的工作都各自是不同的，无法进行统一管理，协同工作。

所以，此时就需要定义一个员工适配器接口 IWorkerAdapter，用于将这些不同的工种进行统一管理。

项目：adapter-0

4.1.1 定义 ICooker

```
public interface ICooker {
    String cook();
}
```

4.1.2 定义 IProgrammer

```
public interface IProgrammer {
    String program();
}
```

4.1.3 定义 QjdCooker

```
public class QjdCooker implements ICooker {

    @Override
    public String cook() {
        return "美味烤鸭";
    }

}
```

4.1.4 定义 JdProgrammer

```
public class JdProgrammer implements IProgrammer{

    @Override
    public String program() {
        return "完美程序";
    }

}
```

4.2 不使用适配器模式

若不使用适配器模式，则调用者需要定义出所有的工种对象，然后逐个工种对象的工作方法进行调用。有 30 个工种，就应调用 30 个工作方法。很麻烦。

不使用适配器程序结构示意图

不使用适配器模式的测试类定义为：

```
public class MyTest {

    public static void main(String[] args) {
        ICooker cooker = new QjdCooker();
        IProgrammer programmer = new JdProgrammer();

        // 必须逐个访问
        System.out.println(cooker.cook());
        System.out.println(programmer.program());
    }

}
```

4.3 只定义一个适配器实现类

这种方式类似于多功能充电器，一个电源插头上接着多种类型的充电接口。用户在使用时需要使用电器接口与多功能充电器上的充电接口逐个进行对比，接口匹配，则可以充电。

其程序结构图如下。

项目：adapter-1。在项目 apapter-0 基础上修改。

4.3.1 定义 IWorkerAdapter

```
public interface IWorkAdapter {
    // 为了兼容所有工种员工，这里的参数必须为Object类型
    void work(Object worker);
}
```

4.3.2 定义 WorkerAdapter

```
public class WorkAdapter implements IWorkAdapter {

    @Override
    public void work(Object worker) {
        if(worker instanceof IProgrammer) {
            System.out.println(((IProgrammer) worker).program());
        }

        if(worker instanceof ICooker) {
            System.out.println(((ICooker)worker).cook());
        }
    }

}
```

4.3.3 定义测试类

```
public class MyTest {

    public static void main(String[] args) {
        ICooker cooker = new QjdCooker();
        IProgrammer programmer = new JdProgrammer();
        Object[] workers = {cooker, programmer};

        IWorkAdapter adapter = new WorkAdapter();
        for (Object worker : workers) {
            adapter.work(worker);
        }
    }

}
```

4.4 为每一个工种都定义一个适配器

为每一个工种定义一个适配器，其程序结构如下：

使用多适配器实现类程序结构示意图

项目：apater-2。在项目 apater-1 项目上修改。

4.4.1 修改 IWorkerAdapter

```
public interface IWorkAdapter {

    void work(Object worker);

    // 判断当前适配器是否支持指定的工种对象
    boolean supports(Object worker);
}
```

4.4.2 定义 CookerAdatper

```
public class CookerAdapter implements IWorkAdapter {

    @Override
    public void work(Object worker) {
        System.out.println(((ICooker)worker).cook());
    }

    @Override
    public boolean supports(Object worker) {
        return (worker instanceof ICooker);
    }

}
```

4.4.3 定义 ProgrammerAdapter

```
public class ProgrammerAdapter implements IWorkAdapter {

    @Override
    public void work(Object worker) {
        System.out.println(((IProgrammer) worker).program());
    }

    @Override
    public boolean supports(Object worker) {
        return (worker instanceof IProgrammer);
    }

}
```

4.4.4 定义测试类

```
11 public class MyTest {
12
13     public static void main(String[] args) {
14         ICooker cooker = new QjdCooker();
15         IProgrammer programmer = new JdProgrammer();
16         Object[] workers = {cooker, programmer};
17
18         for (Object worker : workers) {
19             IWorkAdapter adapter = getAdapter(worker);
20             adapter.work(worker);
21         }
22     }
23
24
25     // 根据工种对象获取支持该工种的适配器对象
26     private static IWorkAdapter getAdapter(Object worker) {
27         IWorkAdapter cookerAdapter = new CookerAdapter();
28         IWorkAdapter programmerAdapter = new ProgrammerAdapter();
29         // 获取到所有适配器
30         IWorkAdapter[] allAdapters = {cookerAdapter, programmerAdapter};
31
32         // 遍历每一个适配器，尝试哪一个适配器支持当前参数工种对象worker
33         for (IWorkAdapter adapter : allAdapters) {
34             if(adapter.supports(worker)){
35                 return adapter;
36             }
37         }
38
39         return null;
40     }
41
42 }
```

4.5 缺省适配器模式

缺省适配器模式是由适配器模式简化而来，省略了适配器模式中目标接口，也就是源接口和目标接口相同，源接口为接口，目标接口为类。

典型的缺省适配器模式是 JavaEE 规范中的 Servlet 接口与 GenericServlet 抽象类。

Servlet 接口中包含五个抽象方法，而其中的 service()方法才是用于实现业务逻辑的、必须要实现的方法，另外四个方法一般都是空实现，或简单实现。

GenericServlet 抽象类实现了 Servlet 接口的 service()方法以外的另外四个方法，所以自定义的 Servlet 只需要继承 GenericServlet 抽象类，实现 service()方法即可。无需再实现 Servlet 接口了。

第5章 模板方法设计模式

在现实生活中，完成某件事情是需要 n 多个固定步骤的。如“在淘宝网进行购物”这件事情的完成一般需要三个步骤：登录网站、挑选商品、付款。但对于登录网站与付款这两步，每个人几乎都是相同的操作。但不同的地方是，每个人所挑选的商品是不同的。

在软件开发过程中同样存在这样的情况。某类的某个方法的实现，需要几个固定步骤。在这些固定步骤中，对于该类的不同对象，有些步骤的实现是固定不变的，有些步骤的实现是大相径庭的，有些步骤的实现是可变可不变的。对于这种情况，就适合使用模板方法设计模式编程。

模板方法设计模式的定义是：定义一个操作中某种算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类在不改变一个算法结构的前提下，对某些步骤实现个性化定义。

5.1 模板方法程序构成

在模板方法设计模式中，存在一个父类。其中包含两类方法：模板方法与步骤方法。

模板方法，即实现某种算法的方法步骤。而这些步骤都是调用的步骤方法完成的。

步骤方法，即完成模板方法的每个阶段性方法。每个步骤方法完成某一特定的、完成总算法的一部分功能。步骤方法有三种类型：抽象方法、最终方法与钩子方法。

抽象方法，是要求子类必须实现的方法，是完成模板方法的算法步骤中必须由子类完成的个性化定义。

最终方法，是子类不能重写的方法，是若要完成模板方法的算法步骤，对于所有子类执行都一样的步骤。

钩子方法，是父类给出了默认实现，但子类也可以重写的方法。

5.2 程序举例

本例用于模拟“在淘宝网购物”这件事情的完成过程。

5.2.1 定义父类

这个父类指的是包含模板方法的父类。本例是个抽象类。

```
public abstract class ShoppingService {
    // 模板方法
    public void shopping(){
        login();    // 登录网站
        choose();   // 挑选商品
        pay();      // 付款
    }

    // 子类不能重写
    public final void login() {
        System.out.println("用户成功登录");
    }

    // 子类必须实现
    public abstract void choose();

    // 钩子方法：子类可以重写
    public void pay() {
        System.out.println("用户使用支付宝拍支付");
    }

}
```

5.2.2 定义子类

这里定义了两个子类，其中一个重写了父类的钩子方法。

```
public class ShoseShoppingService extends ShoppingService {

    @Override
    public void choose() {
        System.out.println("用户购买了老人头皮鞋、NewBalance运动鞋");
    }

}
```

```java
public class DressShoppingService extends ShoppingService {

    @Override
    public void choose() {
        System.out.println("用户购买了利郎男装、柒牌男装");
    }

    @Override
    public void pay() {
        System.out.println("用户使用银联支付");
    }

}
```

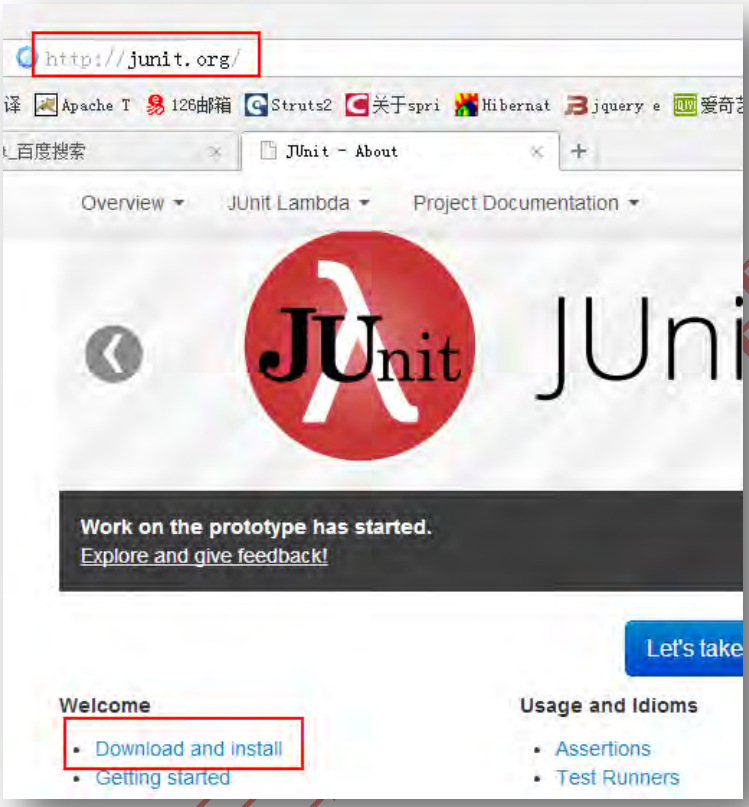
5.2.3 定义测试类

```java
public class MyTest {

    public static void main(String[] args) {
        DressShoppingService dss = new DressShoppingService();
        dss.shopping();

        ShoseShoppingService sss = new ShoseShoppingService();
        sss.shopping();
    }

}
```

第6章 JUnit 测试

6.1 Jar 包的下载

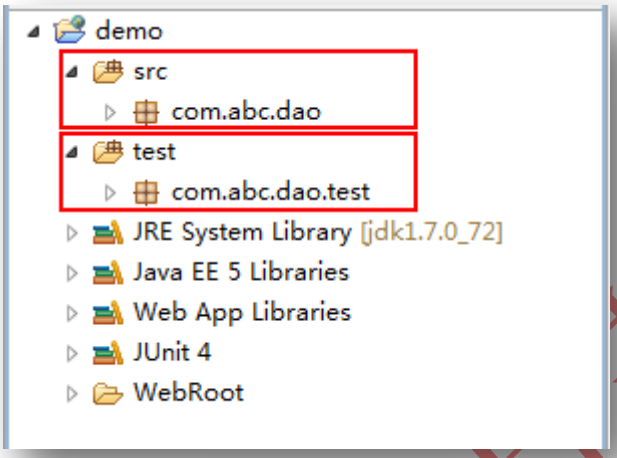
使用 JUnit 进行测试，需要导入 Jar 包。JUnit 的官网为：http://junit.org



6.2 测试类的创建

对于测试类的创建有这样一些习惯：

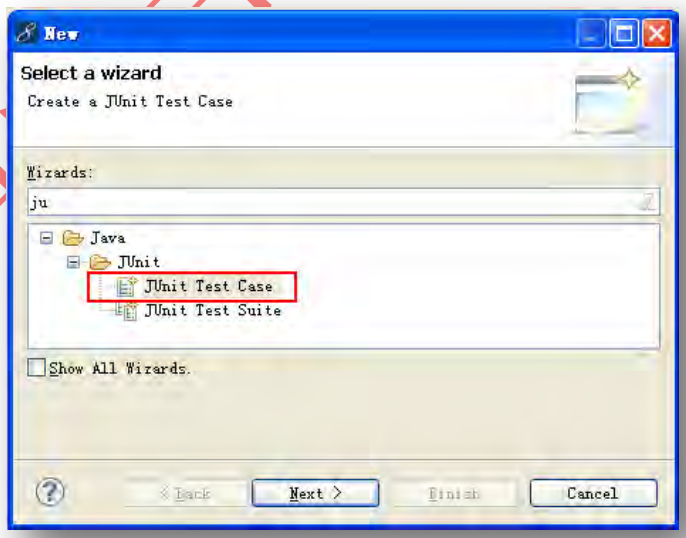
- 在项目中新建一个 source folder，并命名为 test，将来所有测试类均定义在这个目录中。
- 对于测试类所在包，一般是被测类所在包的包名后加再加一个 test 子包。
- 对于测试类类名，一般是被测类类名后加上 Test。

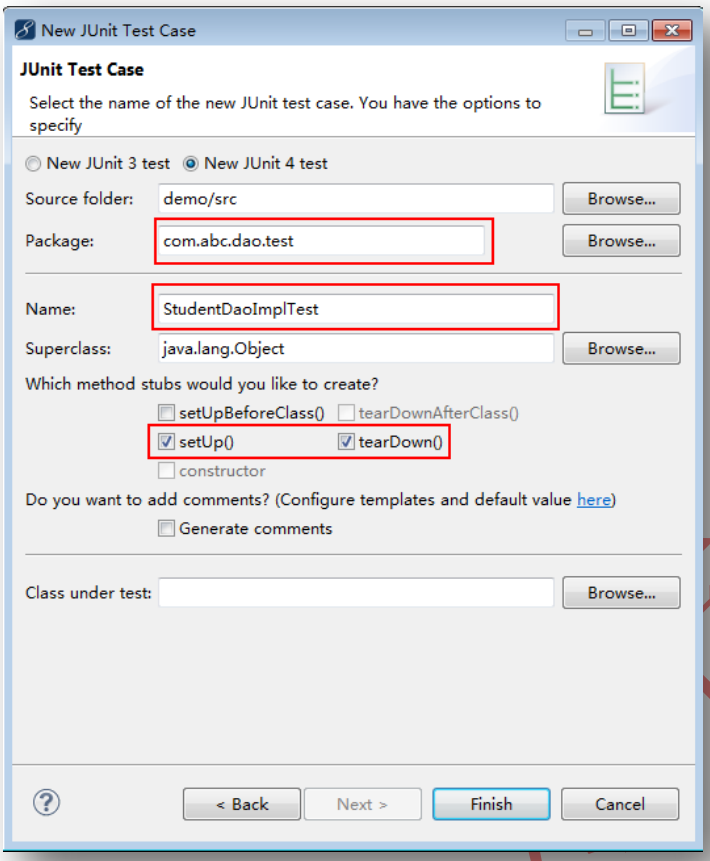


测试类的创建常用的有两种方式：

6.2.1 使用向导创建测试类

直接创建 JUnit Test Case。此时，会自动导入其需要的 Jar 包。





创建过程中，勾选上 setUp()与 tearDown()方法，这两个方法的作用是：

- setUp()：测试方法执行之前执行，主要用于测试前的初始化，如连接数据库等。
- tearDown()：测试方法执行之后执行，主要用于资源释放，如关闭数据库连接等。

创建好后，直接在 setUp()与 tearDown()中编写初始化语句与资源释放语句即可。

对于测试方法的命名，一般是以小写 test 开头，该测试方法用于测试哪个方法，就将该方法名放于 test 之后。当然，首字母要大写。如，要测试 modify()方法，则它的测试方法名一般为：testModify()。

```
public class StudentDaoImplTest {
    private IStudentDao dao;
    @Before
    public void setUp() throws Exception {
        dao = new StudentDaoImpl();
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testInsertStudent() {
        dao.insertStudent();
    }
    @Test
    public void testDeleteStudent() {
        dao.deleteStudent();
    }
}
```

6.2.2 使用注解创建测试类

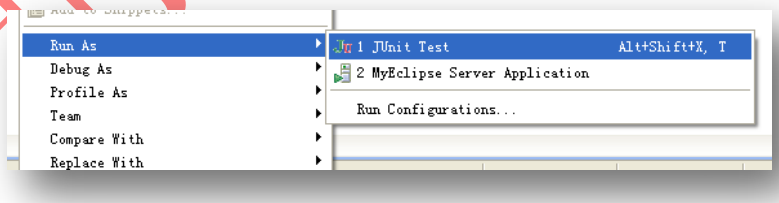
Step1：导入包需的 JUnit Jar 包
Step2：直接创建一个普通的 Java 类
Step3：在其中可以定义初始化方法与资源释放方法。方法签名要求：public void，无参。方法名随意。当然，方法名最好按照规范命名。
Step4：在方法前添加注解

添加@Before 则为初始化方法；
添加@After 则为资源释放方法；
添加@Test 则为测试方法

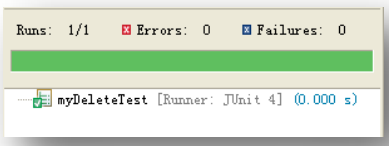
```
public class MyTest {
    private IStudentDao dao;
    @Before
    public void myBefore(){
        dao = new StudentDaoImpl();
    }
    @After
    public void myAfter(){
    }
    @Test
    public void myDeleteTest(){
        dao.deleteStudent();
    }
    @Test
    public void myUpdateTest(){
        dao.updateStudent();
    }
}
```

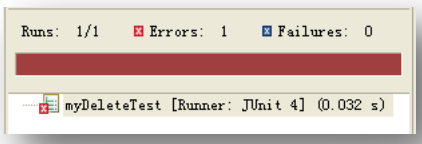
6.3 测试方法的运行

在测试方法的方法签名上右击，选择 Run As/JUnit Test



运行结果看到绿条，则说明运行成功；看到红条，说明运行出了问题。





第7章 Log4j 与 Log4j2

一个完整的软件，日志是必不可少的。程序从开发、测试、维护、运行等环节，都需要向控制台或文件等位置输出大量信息。这些信息的输出，在很多时候是使用System.out.println()无法完成的。

日志信息根据用途与记录内容的不同，分为调试日志、运行日志、异常日志等。

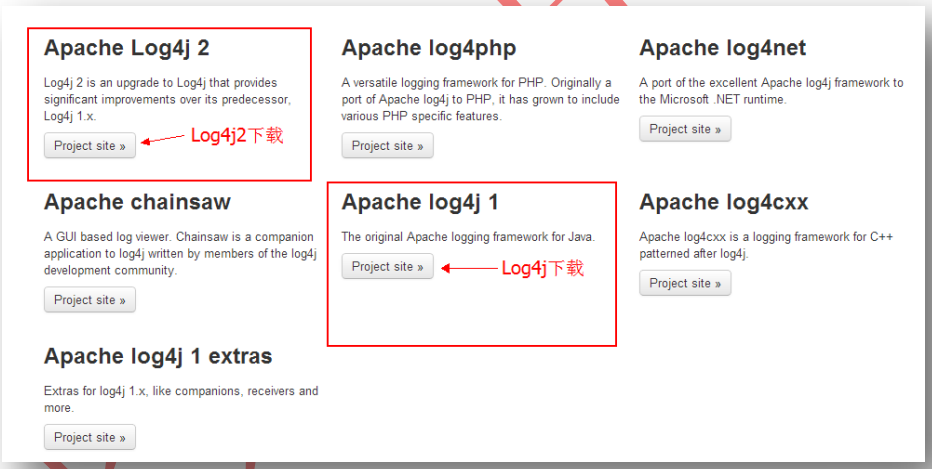
用于日志记录的技术很多，如 jdk 的 logger 技术，apache 的 log4j、log4j2 技术等。

Log4j 的全称为 Log for java，即，专门用于 java 语言的日志记录工具。其目前有两个版本：Log4j 与 Log4j2。

7.1 Log4j 基础

7.1.1 Log4j 的下载

Log4j 下载地址：http://logging.apache.org/



（1） **Log4j** 版本下载

（2） **Logfj2** 版本下载

7.1.2 日志级别

为了方便对于日志信息的输出显示，对日志内容进行了分级管理。日志级别由高到低 ，共分 6 个级别：fatal(致命的)、error、warn、info、debug、trace(堆栈)。

为什么要对日志进行分级呢？

无论是将日志输出到控制台，还是文件，其输出都会降低程序的运行效率。但由于调试、运行维护的需要，客户的要求等原因，需要进行必要的日志输出。这时就必须要在代码中加

入日志输出语句。

这些输出语句若在程序运行时全部执行，则势必会降低运行效率。例如，使用System.out.println()将信息输出到控制台，则所有的该输出语句均将执行。会大大降低程序的执行效率。而要使其不输出，唯一的办法就是将这些输出语句逐个全部删除。这是个费时费力的过程。

将日志信息进行分级管理，便可方便的控制信息输出内容及输出位置：哪些信息需要输出，哪些信息不需要输出，只需在一个日志输出控制文件中稍加修改即可。而代码中的输出语句不用做任何修改。

从这个角度来说，代码中的日志编写，其实就是写大量的输出语句。只不过，这些输出语句比较特殊，它们具有级别，在程序运行期间不一定被执行。它们的执行是由另一个控制文件控制。

7.1.3 日志输出控制文件

Log4j 的日志输出控制文件，主要由三个部分构成：

（1）日志信息的输出位置：控制日志信息将要输出的位置，是控制台还是文件等。

（2）日志信息的输出格式：控制日志信息的显示格式，即以怎样的字符串形式显示。

（3）日志信息的输出级别：控制日志信息的显示内容，即显示哪些级别的日志信息。

有了日志输出控制文件，代码中只要设置好日志信息内容及其级别即可，通过控制文件便可控制这些日志信息的输出了。

7.2 Log4j 技术

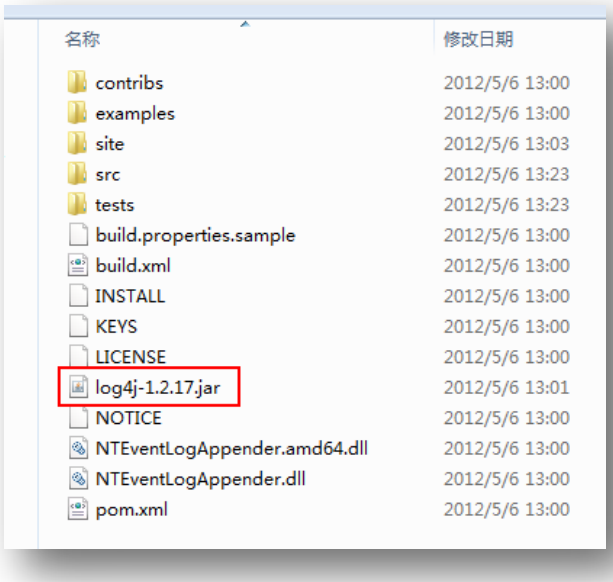
7.2.1 在程序中的日志实现步骤

项目：log4j

若要在自己的程序中写入日志语句，则可按照以下步骤进行：

（1）导入 Jar 包

在项目中导入 log4j 需要的 jar 包。将 Log4j 框架的 zip 文件进行解压，在其根目录下就有其 Jar 包。



（2） 放入日志输出控制文件

将属性文件 log4j.properties 直接放到项目的 src 下

（3） 代码中实现日志记录

在要输出日志的类中创建日志对象 Logger，并通过 Logger 的方法在代码中加入日志输出语句。在 Java 代码中进行日志输出，需要用到 Logger 类的静态方法 getLogger()。

注意，Logger 为 org.apache.log4j 包中的类。

```
import org.apache.log4j.Logger;

public class MyMain {

    public static void main(String[] args) {
        // 创建日志记录对象Logger
        Logger logger = Logger.getLogger(MyMain.class);

        logger.debug("this is debug message");
        logger.info("this is info message");
        logger.warn("this is warn message");
        logger.error("this is error message");
    }
}
```

将来这些日志输出语句，会根据 log4j.properties 文件中日志级别的设置进行输出，会输出到指定位置。其输出结果是：输出指定级别及其更高级别的信息。如指定 info 级别，则会输出 fatal、error、warn、info 级别的信息。就本例而言，会执行以下三句，而不会执行 debug() 方法。

```
<terminated> MyTest (51) [Java Application] C:\Program Files (x86)\Java\jdk1.7.0_72\bin\javaw.exe (2015-10-4 下午9:38:25)
[ERROR][2015-10-04 21:38:26] test.MyTest 11 this is error message
[WARN ][2015-10-04 21:38:26] test.MyTest 12 this is warn message
[INFO ][2015-10-04 21:38:26] test.MyTest 13 this is info message
```

7.2.2 日志输出控制文件分析

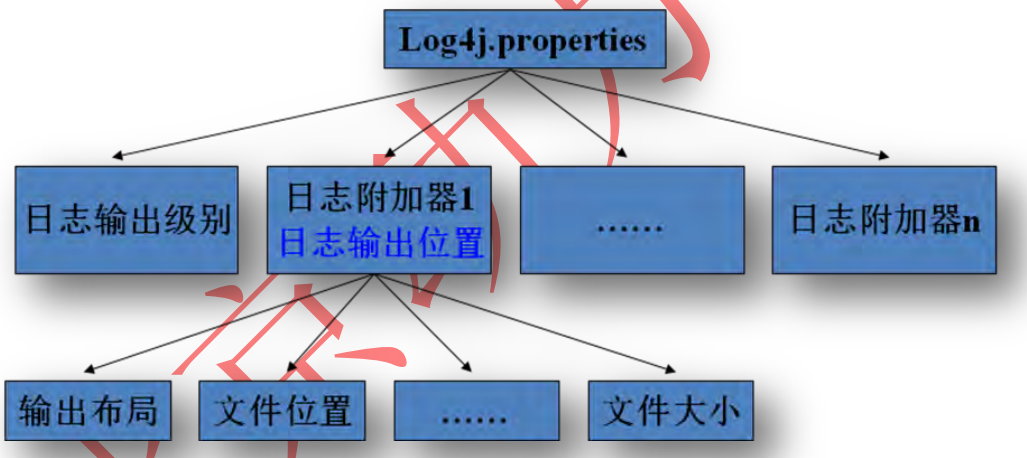
日志属性文件 log4j.properties 是专门用于控制日志输出的。其主要进行三方面控制：

- 输出位置：控制日志将要输出的位置，是控制台还是文件等。
- 输出布局：控制日志信息的显示形式
- 输出级别：控制要输出的日志级别。

日志属性文件由两个对象组成：日志附加器与根日志。

根日志，即为 Java 代码中的日志记录器，其主要由两个属性构成：日志输出级别与日志附加器。

日志附加器，则由日志输出位置定义，由其它很多属性进行修饰，如输出布局、文件位置、文件大小等。

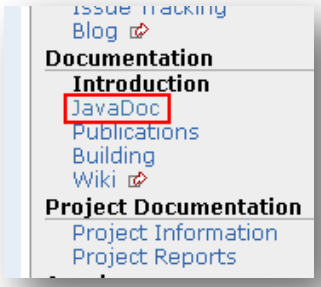


（1） 定义日志附加器 appender

所谓日志附加器，就是为日志记录器附加上很多其它设置信息。附加器的本质是一个接口，其定义语法为：log4j.appender.appenderName = 输出位置

appenderName 为自定义名称。

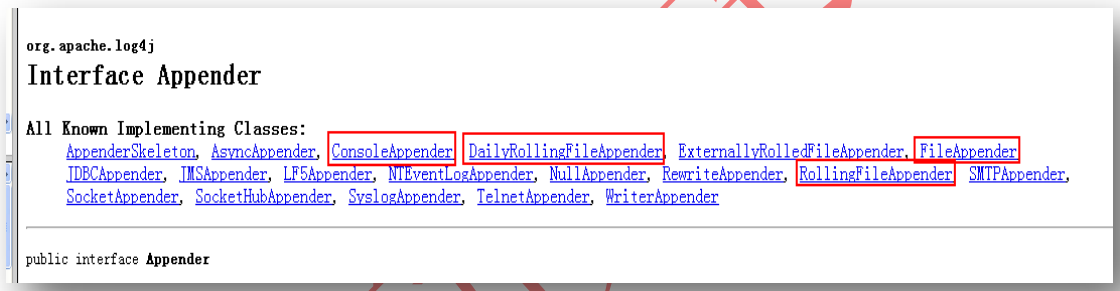
输出位置为 log4j 指定的类型，是定义好的一些 appender 接口的实现类。查看 log4j 框架解压目录下的站点目录 site 的 index.jsp 中的 JavaDoc，可看到 log4j 的 API。



例如定义了一个名称为 console 的控制台附加器：

```
##define an appender named console
log4j.appender.console=org.apache.log4j.ConsoleAppender
```

常用的附加器实现类如下：



- org.apache.log4j.ConsoleAppender：日志输出到控制台
- org.apache.log4j.FileAppender：日志输出到文件
- org.apache.log4j.RollingFileAppender：当日志文件大小到达指定尺寸的时候将产生一个新的日志文件
- org.apache.log4j.DailyRollingFileAppender：每天产生一个日志文件

（2）修饰日志附加器

所谓修饰日志附加器，就是为定义好的附加器添加一些属性，以控制到指定位置的输出。不同的附加器，其修饰属性不同。

- **控制台附加器：**

```
##define an appender named console
log4j.appender.console=org.apache.log4j.ConsoleAppender

#The Target value is System.out or System.err
log4j.appender.console.Target=System.out
#set the layout type of the apperder
log4j.appender.console.layout=org.apache.log4j.PatternLayout
#set the layout format pattern
log4j.appender.console.layout.ConversionPattern=[%-5p][%d{yyyy-MM-dd HH:mm:ss}]%m%n
```

Target：控制输出到控制台的使用目标。其值为 System.out 或 Sytem.err。它们的区别是，System.out 是以黑色字体显示到控制台，而 System.err 则是以红色字体显示。

- 文件附加器：

```
##define an appender named file
log4j.appender.file=org.apache.log4j.FileAppender

#define the file path and name
log4j.appender.file.File=d:/logfile.txt
#set the layout type of the apperder
log4j.appender.file.layout=org.apache.log4j.PatternLayout
#set the layout format pattern
log4j.appender.file.layout.ConversionPattern=[%-5p][%d{yyyy-MM-dd HH:mm:ss}]%m%n
```

File：日志要输出的文件位置及文件名称。

- 滚动文件附加器：

MaxFileSize：用于指定日志文件的最大值。若文件超过指定值，将自动产生另一个日志文件。

Log4j 常用布局类型：

- org.apache.log4j.HTMLLayout：网页布局，以 HTML 表格形式布局
- org.apache.log4j.SimpleLayout：简单布局，包含日志信息的级别和信息字符串
- org.apache.log4j.PatternLayout：匹配器布局，可以灵活地指定布局模式。其主要是通过设置 PatternLayout 的 ConversionPattern 属性值来控制具体输出格式的。ConversionPattern 的值中有很多控制字符，这些字符的意义如下表所示：

ConversionPattern 取值说明

（本表格来自互联网）

参数	说明	例子	
%c	列出 logger 名字空间的全称，如果加上{<层数>}表示列出从最内层算起的指定层数的名字空间	log4j 配置文件参数举例	输出显示媒介
		假设当前 logger 名字空间是“a.b.c”	
		%c	a.b.c
		%c{2}	b.c
		%20c	（若名字空间长度小于 20，则左边用空格填充）

		%-20c	（若名字空间长度小于 20，则右边用空格填充）
		%.30c	（若名字空间长度超过 30，截去多余字符）
		%20.30c	（若名字空间长度小于 20，则左边用空格填充；若名字空间长度超过 30，截去多余字符）
		%-20.30c	（若名字空间长度小于 20，则右边用空格填充；若名字空间长度超过 30，截去多余字符）
%C	列出调用 logger 的类的全名（包含包路径）	假设当前类是“org.apache.xyz.SomeClass”	
		%C	org.apache.xyz.SomeClass
		%C{2}	xyz.SomeClass
%d	显示日志记录时间，{<日期格式>}使用 ISO8601 定义的日期格式	%d{yyyy/MM/dd HH:mm:ss,SSS}	2005/10/12 22:23:30,117
		%d{ABSOLUTE}	22:23:30,117
		%d{DATE}	12 Oct 2005 22:23:30,117
		%d{ISO8601}	2005-10-12 22:23:30,117
%F	显示调用 logger 的源文件名	%F	MyClass.java
%l	输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数	%l	MyClass.main(MyClass.java:129)
%L	显示调用 logger 的代码行	%L	129
%m	显示输出消息	%m	This is a message for debug.
%M	显示调用 logger 的方法名	%M	main
%n	当前平台下的换行符	%n	Windows 平台下表示 rn UNIX 平台下表示 n
%p	显示该条日志的优先级	%p	INFO
%r	显示从程序启动时到记录该条日志时已经经过的毫秒数	%r	1215
%t	输出产生该日志事件的线程名	%t	MyClass
%x	按 NDC（Nested Diagnostic	假设某程序调用顺序是 MyApp 调用 com.foo.Bar	
		%c %x - %m%n	MyApp - Call com.foo.Bar.

	Context，线程堆栈）顺序输出日志		com.foo.Bar - Log in Bar MyApp - Return to MyApp.
%X	按 MDC（Mapped Diagnostic Context，线程映射表）输出日志。通常用于多个客户端连接同一台服务器，方便服务器区分是那个客户端访问留下来的日志。	%X{5}	（记录代号为 5 的客户端的日志）
%%	显示一个百分号	%%	%

（3） 配置根 Logger

配置 rootLogger，以便于代码加载来控制日志的输出。其语法为：

log4j.rootLogger = [level] , appenderName, …

其中，level 是日志记录的优先级，分为 OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL。Log4j 建议只使用四个级别，优先级从高到低分别是 ERROR、WARN、INFO、DEBUG。OFF 为关闭日志功能。

低级别的可以显示高级别的，但高级别的不能显示低级别的。所以，级别越高，将来显示的信息就越少。

7.3 Log4j2 技术

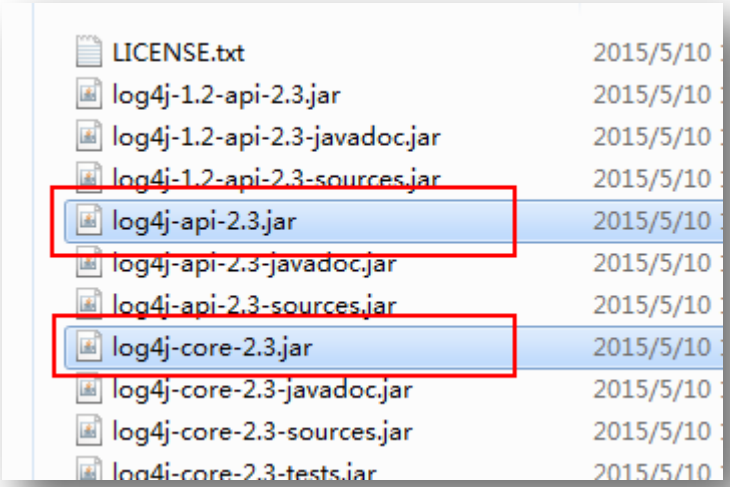
og4j2，是对 Log4j 的升级，其在配置与使用上发生了较大变化。

7.3.1 在程序中的日志实现步骤

项目：log4j2

（1） 导入 Jar 包

使用 Log4j2，需要导入其 Jar 包。Log4j 框架解压目录中找到如下两个 jar 包：



（2） 放入日志输出控制文件

将文件 log4j2.xml 直接放到项目的 src 下。log4j2 配置文件是 XML 文件，不再支持 properties 文件。默认的文件名为 log4j2.xml。其存放的位置为 classpath 中。

（3） 代码中实现日志记录

在要输出日志的类中创建日志对象 Logger，并通过 Logger 的方法在代码中加入日志输出语句。该日志对象是通过静态类 LogManager 的 getLogger()方法获取的。

注意，Logger 与 LogManager 均为 org.apache.logging.log4j 包中的类，非 org.apache.log4j 包中的。

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class MyMain {

    public static void main(String[] args) {
        // 创建日志记录对象Logger
        Logger logger = LogManager.getLogger(MyMain.class);

        logger.debug("this is debug message");
        logger.info("this is info message");
        logger.warn("this is warn message");
        logger.error("this is error message");
    }
}
```

7.3.2 日志输出控制文件分析

log4j2 配置文件是 XML 文件，不再支持 properties 文件。默认的文件名为 log4j2.xml。其存放的位置为 classpath 中。

若没有设置 log4j2.xml，则系统会使用默认的日志配置：只会输出到控制台 error 级别的信息。

```
<configuration status="OFF">
    <appenders>
        <Console name="myConsole" target="SYSTEM_ERR">
            <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n" />
        </Console>
        <File name="myLogFile" fileName="log/test.log" append="true">
            <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n"/>
        </File>
        <RollingFile name="myRollingFile" fileName="logs/app.log"
                     filePattern="logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
            <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n"/>
            <SizeBasedTriggeringPolicy size="1KB"/>
        </RollingFile>
    </appenders>
    <loggers>
        <root level="info">
            <appender-ref ref="myConsole" />
            <!-- <appender-ref ref="myLogFile" /> -->
            <!-- <appender-ref ref="myRollingFile" /> -->
        </root>
    </loggers>
</configuration>
```

配置文件说明：

（1） <configuration/>标签

<configuration/>标签的 status 属性用于设置 Log4j2 自身运行的日志显示级别，一般为 OFF，不显示。当然，也可以设置为 ERROR、DEBUG 等其它级别。

（2） <Console/>标签

<Console/>标签的 target 属性用于设置输出的目标形式，其值一般为：
SYSTEM_OUT 或 SYSTEM_ERR

```
<Console name="myConsole" target="SYSTEM_OUT">
    <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n" />
</Console>
```

（3） <File/>标签

<File/>标签的 fileName 属性用于设置文件的文件保存路径及文件名。如本例的意思是，日志文件名为 test.log，将其存放在当前项目的根目录下的 log 子目录中。

append 属性用于设置是否以追加方式将日志写入指定文件。

```
<File name="myLogFile" fileName="log/test.log" append="true">
    <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n"/>
</File>
```

<RollingFile/>标签的<SizeBasedTriggeringPolicy/>子标签用于指定每一个日志文件最大文件大小。当达到这个指定值后，会自动再新建一个日志文件。

（4） <RollingFile/>标签

fileName 指定存放目录及第一个日志文件名。filePattern 指定新创建的日志文件的文件名。本例还会对文件进行压缩。

```
<RollingFile name="myRollingFile" fileName="logs/app.log"
             filePattern="logs/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
    <PatternLayout pattern="[%-5p][%d{yyyy-MM-dd HH:mm:ss}] [%c %L] %m%n"/>
    <SizeBasedTriggeringPolicy size="1KB"/>
</RollingFile>
```

（5） <loggers/>标签

用于配置根 Logger 对象，以指定所使用的日志记录器，及显示的级别。

其子标签<root/>用于指定所使用的日志记录器。该子标签的属性 level 用于指定显示级别。而日志记录器是通过<root/>的子标签<appender-ref/>来引用<appenders/>中定义好的记录器的。

需要注意的是，只要在<appenders/>中定义了<File/>、<RollingFile/>等，且在其中指定了日志存放的目录，那么这些目录就会自动创建。无论在<loggers/>的<root/>中是否声明使用它们。

```
<loggers>
    <root level="info">
        <appender-ref ref="myConsole" />
        <!-- <appender-ref ref="myLogFile" /> -->
        <!-- <appender-ref ref="myRollingFile" /> -->
    </root>
</loggers>
```