

# 轻量级 J2EE 框架应用

## E 7 A Simple Controller with DI

学号：SA18225433

姓名：杨帆

报告撰写时间：2018/01/06

# 1.主题概述

## 1、IOC 和 DI。

IOC 字面意思为控制反转，IOC 容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖，完成 Bean 的装配，将对象的控制权由人为 new 一个对象，交由容器来管理，当需要某个对象时，容器直接将对应的对象实例化并注入到需要对象的代码中，DI 的字面意思为依赖注入，把有依赖关系的类放到 IOC 容器中，在目标代码中存在一个依赖类的注入点，在运行过程中不需要目标代码实例化此依赖对象，而是由 IOC 容器解析出这些类的实例，就是依赖注入，目的是实现类的解耦。

## 2、内省机制。

内省(Introspector)是 Java 语言对 Bean 类属性、事件的一种缺省处理方法。JavaBean 是一种特殊的类，主要用于传递数据信息，这种类中的方法主要用于访问私有的字段，且方法名符合某种命名规则。如果在两个模块之间传递信息，可以将信息封装进 JavaBean 中，这种对象称为“值对象”(Value Object)，或“VO”。方法比较少。这些信息储存在类的私有变量中，通过 set()、get() 获得。内省机制是通过反射来实现的，BeanInfo 用来暴露一个 bean 的属性、方法和事件，以后我们就可以操纵该 JavaBean 的属性。

## 2.假设

本次作业能够使用 IDEA 编写 Controller 具有 IOC 容器以及依赖注入的思想，包括能够解析依赖关系的 xml 文件,将各个 bean 类放到 ioc 容器中,将具有依赖关系的 bean 类通过 java 内省机制注入到目标 bean 中，并实例化存放到 IOC 容器中。

### 3. 实现或证明

#### 1. 实现成果

e7: <https://github.com/saaaaaail/J2eee7>

#### 2. 基于 E6。在 UseSC 工程中新建一个 XML 文件名为 di.xml，格式可参考如下：；

```
<?xml version="1.0" encoding="UTF-8" ?>
<sc-di>
  <bean id="userBean" class="sail.ustc.model.UserBean"></bean>
  <bean id="login" class="sail.ustc.action.LoginAction">
    <field name="userBean" bean-ref="userBean"></field>
  </bean>
  <!--some beans-->
</sc-di>
```

3. 在 di.xml 中定义<bean>节点。<bean>节点中指明当前 bean 的名字、class 类型及 bean 属性是否引用另一个 bean。如示例中 loginAction 的属性 userBean 引用名为 user 的 bean。

定义的 di.xml 文件如上图所示。

4. 修改 SimpleController 的控制器代码。当有 http 登录请求被拦截后，在 UseSC 工程配置文件 controller.xml 中查找是否有对应的 Action 请求。如果无，直接响应客户端“无法识别该请求”。

在 ActionPackage 类中调用工具类 XmlUtil 去解析 controller.xml 文件，

```
String actionMess = xmlUtil.parseXml(fileString, field: "action", actionName);
```

如果无对应的 Action 请求，则返回"action:failure"字符串，如果存在对应的 Action 请求，则将类名和方法名拼接为字符串并返回，

```

public String parseXml(String fileString, String field, String name) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse(new File(fileString));
        NodeList fields = doc.getElementsByTagName(field);

        for(int i=0; i<fields.getLength(); i++) {
            NamedNodeMap map = fields.item(i).getAttributes();
            if(map.getNamedItem("name").getNodeValue().equals(name)) {
                return map.getNamedItem("class").getNodeValue()
                    + ", " + map.getNamedItem("method").getNodeValue();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "action:failure";
}

```

然后在 `doAction()` 方法中获得返回的字符串，若字符串为 "action:failure" 字符串，则返回该字符串到 `SimpleController` 中，并使用 `dispatch()` 方法打印输出“无法识别该请求”。

```

private void dispatch(HttpServletRequest req, HttpServletResponse resp, String result, PrintWriter out) {
    try {
        String[] tmp = result.split(regex);
        //System.out.println("dispatch: "+tmp[1]);
        switch (tmp[0]) {
            case "forward": out.println(tmp[1]); req.getRequestDispatcher(tmp[1]).forward(req, resp); break;
            case "redirect": out.println(tmp[1]); resp.sendRedirect(tmp[1]); break;
            case "action:failure": out.println("无法识别该请求"); break;
            case "result:failure": out.println("没有请求的资源"); break;
        }
    } catch (ServletException e) {
    }
}

```

5. 如果找到该请求对应的 Action，则在 `di.xml` 查找与请求 Action 同名的 bean 节点。如果 `di.xml` 中无指定 `<bean>` 节点，Controller 直接通过反射构造 Action 实例，并将请求分发给 Action 实例进行处理。

向 IOC 容器传入 `actionName`，若返回 null 对象，则说明在 `di.xml` 文件中无 `<bean>` 节点，则直接通过反射，根据 `actionName` 解析出来的类的全限定名，获得 Action 的类变量，进而得到对象实例，

```
//从容器中获得bean实例
Object object = di.getObject(actionName);
Class actionClass ;
if (object==null){
    actionClass = Class.forName(mess[0]);
    object = actionClass.newInstance();
}else {
    actionClass = object.getClass();
}
Method method = actionClass.getMethod(mess[1], String.class, String.class);
obj = method.invoke(object,userid,userpass);
```

6.如果在 di.xml 中找到指定的<bean>节点，则查看该节点是否有属性依赖其他<bean>节点。如果无依赖，则直接通过反射构造该<bean>实例，并将请求分发至其处理。

对于 di.xml 文件中定义的 bean 类与有依赖的 bean 类都预先存放到 IOC 容器中，详细见第 7 步骤。

7.如果有依赖，则需先通过反射构造被依赖的<bean>实例，之后再构造依赖<bean>实例；并通过属性的 setter 方法（Java 内省机制，Introspector）将被依赖的<bean>实例注入依赖<bean>实例。

首先解析 di.xml 文件地址，并开始解析，需要先构建三个实体类用来存放解析数据，Bean 类存放文件中每一条 Bean 类信息，

```
public class Bean {
    private String id;
    private String clazz;
    private List<Field> fields;
```

Field 类存放每条 Bean 类所依赖的 Bean 类，

```
public class Field {
    private String name;
    private String beanRef;
```

DiEntity 类用来解析 di.xml 文件，并使用 map 存储，map 为类静态变量，保证文件仅解析一次，

```

public Map parseDi() {
    if(map==null) {
        map = new HashMap<>();
        Document document = getDocument();
        NodeList beans = document.getElementsByTagName("bean");
        for(int i=0;i<beans.getLength();i++) {
            Node node = beans.item(i);
            NamedNodeMap namedNodeMap = node.getAttributes();
            String beanId = namedNodeMap.getNamedItem("id").getNodeValue();
            String beanClass = namedNodeMap.getNamedItem("class").getNodeValue();
            List<Field> fields = new ArrayList<>();
            NodeList clist = node.getChildNodes();
            for(int j=0;j<clist.getLength();j++) {
                if (clist.item(j).getNodeName().equals("field")) {
                    NamedNodeMap cNodeMap = clist.item(j).getAttributes();
                    String fieldName = cNodeMap.getNamedItem("name").getNodeValue();
                    String beanRef = cNodeMap.getNamedItem("bean-ref").getNodeValue();
                    Field field = new Field(fieldName, beanRef);
                    fields.add(field);
                }
            }
            Bean bean = new Bean(beanId, beanClass, fields);
            map.put(beanId, bean);
        }
    }
    return map;
}

```

然后创建 IOC 容器类 Container,

```
public class Container {
```

构建两个 map，一个用来做底层的 IOC 容器，一个用来获得 di.xml 的解析数据，

```

//ioc容器
private static Map<String, Object> iocMap;

//保存解析xml的bean类信息
private static Map<String, Bean> beanMap;

```

创建 loadDiXml()方法从 DiEntity 中获得 di.xml 的解析数据，

```

//加载xml解析对象
public void loadDiXml() {
    DiEntity diEntity = DiEntity.getInstance();
    beanMap = diEntity.parseDi();
}

```

创建 store()方法，用来从 beanMap 中读取 Bean 类的 id 与全限定名，使用反射创建实例，并以 id 为 key，以对应实例为 value，存入到 iocMap 中，即存到 IOC 容器中，

```

//存储到IOC容器
public void store() {
    if(iocMap==null) {
        iocMap = new HashMap<>();
        Set keys = beanMap.keySet();
        Iterator it = keys.iterator();
        while (it.hasNext()) {
            String key = (String) it.next();
            Bean bean = beanMap.get(key);
            Class cls=null;//类变量
            Object newObj=null;//实例
            try {
                cls = Class.forName(bean.getClassName());
                newObj = cls.newInstance();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            }
            iocMap.put(beanMap.get(key).getId(), newObj);
        }
    }
}

```

保存所有 Bean 类到 IOC 容器中，然后需要判断 Bean 类是否有依赖对象，若有依赖对象，将依赖的对象实例化，并使用 setter 方法注入到目标对象，因此创建 Di 类，

```

public class Di {

    private Di() {container.process();}
}

```

并创建注入方法 Inject()，思路大概为获得每个 Bean 的 Field 依赖对象的 list，若 list 不为空则说明该 Bean 类有依赖对象，首先判断依赖对象是否保存到 beanMap 中，若存在则返回依赖 refBean 对象，使用 java 内省机制获得当前目标 curBean 类的 beaninfo 对象，通过 beanInfo.getPropertyDescriptors()方法获得当前 curBean 类的保存的类变量的所有属性，遍历属性与依赖 refBean 的 id 进行比较找到属性对应得依赖类，通过 pd.getWriteMethod()获得当前 curBean 类的写方法，使用 invoke 执行写方法将其依赖 refBean 类对象保存的类变量实例化注入，最后更新 IOC 容器中的 Bean。



```

public boolean Inject() {
    boolean flag = false;
    try {
        System.out.println("开始注入!!");
        Set keys = beanMap.keySet();
        Iterator it = keys.iterator();
        while (it.hasNext()) {
            String id = (String) it.next();
            Bean curBean = beanMap.get(id);
            Class curClazz = Class.forName(curBean.getClassName()); //当前bean的类变量
            Object curObj = curClazz.newInstance(); //当前bean的实例
            List<Field> fields = curBean.getFields();
            if (fields != null) {
                fields = curBean.getFields();
                for (Field f : fields) {
                    Bean refBean = judgeRef(f); //获得依赖bean的xml解析类
                    //下面获得当前bean类的setter方法，注入依赖bean类
                    BeanInfo beanInfo = Introspector.getBeanInfo(curClazz, Object.class);
                    PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();
                    for (PropertyDescriptor pd : pds) {
                        System.out.println("pd.getName(): " + pd.getName());
                        //匹配当前依赖bean，找到对应的属性加载器
                        if (pd.getName().toString().equals(refBean.getId())) {
                            flag = true;
                            Method method = pd.getWriteMethod();
                            method.invoke(curObj, Class.forName(refBean.getClassName()).newInstance());
                        }
                    }
                }
            }
            container.addObject(curBean.getId(), curObj);
        }
    } catch (Exception e) {
    }
}

```

然后在依赖注入类中提供 get 接口，用来获得容器中的实例对象，

```

public Object getObject(String id) {
    try {
        return container.getObject(id);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

```

8.完成以上工作后，修改 UseSC 工程中的 LoginAction 的代码，将 UserBean 对象属性初始化代码语句移除。重新打包工程测试。

删除 UserBean 对象的属性初始化代码，

```
public class LoginAction {
    private UserBean userBean;

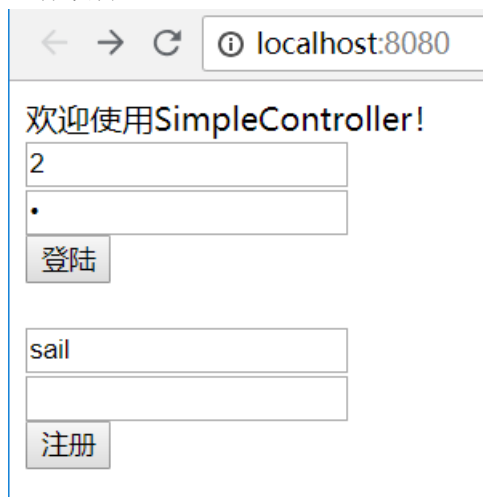
    public String handleLogin(String name, String password) {
        System.out.println("执行handleLogin...");
        if (userBean.signIn(name, password)) {
            return "success";
        } else {
            return "failure";
        }
    }

    public UserBean getUserBean() { return userBean; }

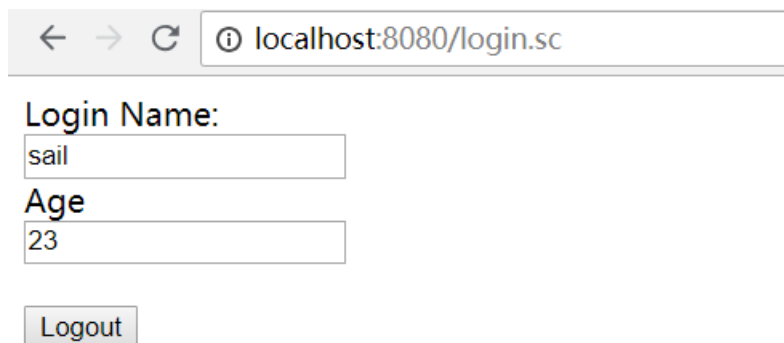
    public void setUserBean(UserBean userBean) { this.userBean = userBean; }
}
```

9.基于以上内容，修改 E6 中关于对象依赖的代码，将对象依赖关系通过 di.xml 进行管理。重新打包 SimpleController、UseSC 工程，测试是否能够正确运行。如修改 DAO 的依赖代码，修改 Model 的依赖代码等。

运行项目，



输入 id=2，密码=1，点击登录，  
页面成功跳转，



## 4. 结论

对主题的总结，结果评论，发现的问题，或你的建议和看法。

本次作业学习了 spring 的 IOC 控制反转，依赖注入的思想，并首先使用一个 map 保存 xml 文件的解析数据，然后根据第一个 map 中的类信息，创建 bean 实例并保存到第二个 map 中作为 IOC 容器的底层实现，然后判断 xml 文件每个 Bean 类是否有依赖类，若有则使用 java 内省机制获得写方法，将对应的依赖对象写入到对应的属性中。

## 5.参考文献

- 1.[参阅 github 学长代码编写风格](#)
- 2.[Java 的内省技术](#)
- 3.[Java 反射和内省实现 spring 的 IOC 和 DI](#)
- 4.[Java 代码实现依赖注入](#)
- 5.[Java 内省机制](#)
- 6.[IOC 容器基本原理](#)
- 7.[依赖注入是什么？](#)