IoC控制反转和DI依赖注入

控制反转(Inversion of Control ,IoC)的概念

- ✓ 所谓控制反转就是应用本身不负责依赖对象的创建及维护，依赖对象的创建及维护是由外部容器负责的。这样控制权就由应用转移到了外部容器，控制权的转移就是所谓反转。
- ✓ 例如在A类中需要使用一个B类的对象实例，这个B类的对象不是由A类创建的，而是由外部容器（例如Spring容器）创建的。

依赖注入（Dependency Injection，DI）的概念

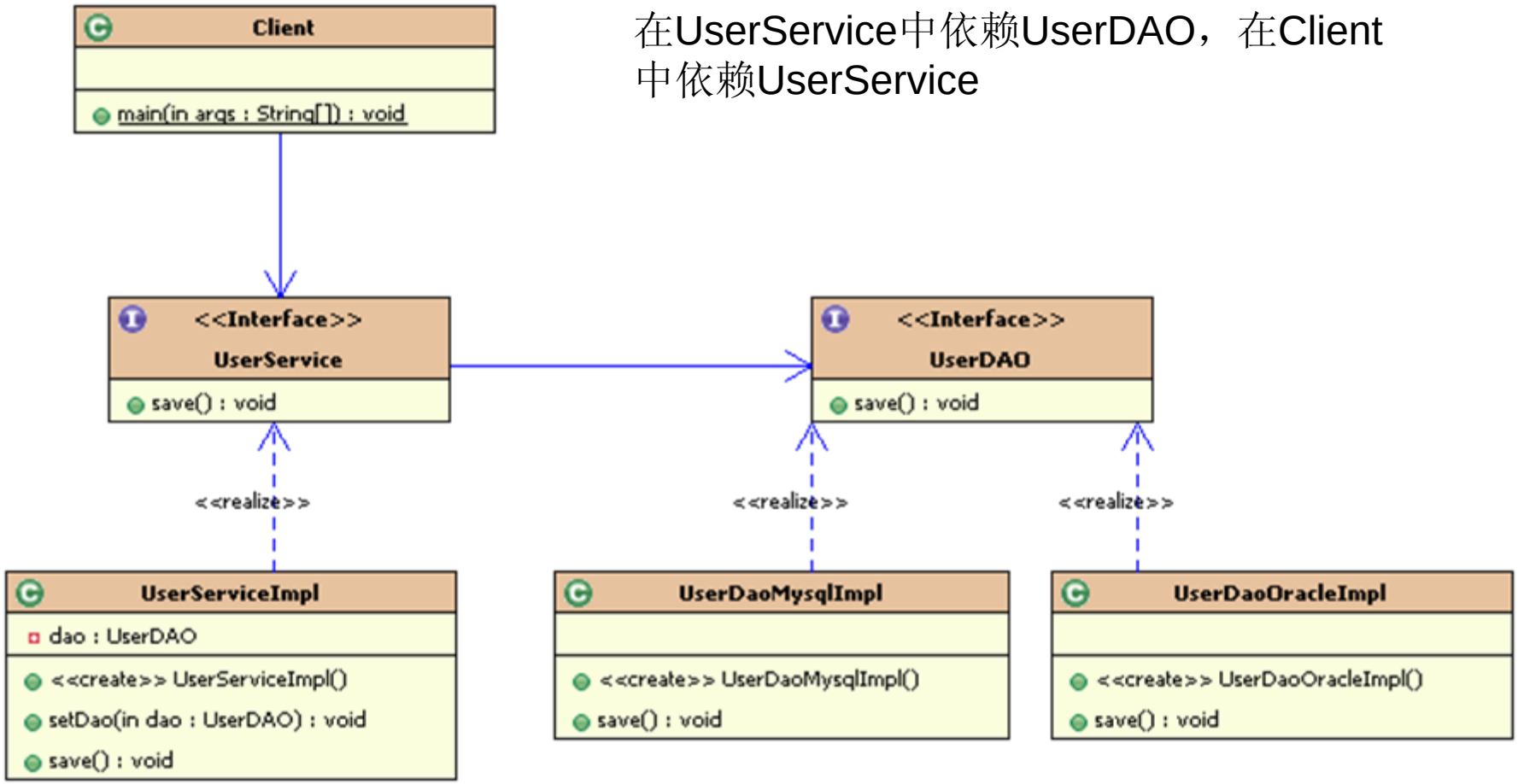
- ✓ 所谓依赖注入就是指在运行期，由外部容器动态地将依赖对象注入到组件中。
- ✓ 例如在A类中声明一个B类类型的成员变量，在运行期间容器会给这个成员变量赋值，也可以说容器创建了一个B类的对象注入到A类中。
- ✓ 在Spring中控制反转和依赖注入指的都是同一个东西

依赖注入的类型

- ✓ 构造器注入
  - ➢ 通过类的构造方法注入依赖关系
- ✓ 设值方法注入
  - ➢ 通过类的setter方法注入依赖关系

依赖注入示例

在UserService中依赖UserDAO，在Client中依赖UserService



DAO接口和两个实现类

```
public interface UserDAO {
	void save();
}

public class UserDaoOracleImpl implements UserDAO {
	public void save() {
		System.out.println("Oracle数据库实现。。。");
	}
}

public class UserDaoMysqlImpl implements UserDAO {
	public void save() {
		System.out.println("Mysql数据库实现。。。");
	}
}
```

Service接口和实现类

```
public interface UserService {
	void save();
}

public class UserServiceImpl implements UserService{
	private UserDAO dao;
	//构造方法
	public UserServiceImpl(UserDAO dao){
		this.dao = dao;
	}
	public void save() {
		dao.save();
	}
}
```

配置文件applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!-- 定义DAO的两个实现类的Bean，id的名称任意 -->
    <bean id="userDaoOracleImpl" class="com.spring.dao.UserDaoOracleImpl"/>
    <bean id="userDaoMysqlImpl" class="com.spring.dao.UserDaoMysqlImpl"/>

    <!-- 定义Service实现类的Bean，id的名称任意 -->
    <bean id="userService" class="com.spring.service.UserServiceImpl">
        <!-- 以构造方法参数的形式注入属性值是id为userDaoMysqlImpl的实例 -->
        <constructor-arg ref="userDaoMysqlImpl"/>
    </bean>
</beans>
```

Client客户端类测试

```
public class Client {
	public static void main(String[] args) {
		//通过配置文件实例化Spring容器
		ApplicationContext context = new ClassPathXmlApplicationContext(
					new String[] { "applicationContext.xml" });
		//获得id为userService的Bean对象
		UserService service = (UserService) context.getBean("userService");
		//调用方法
		service.save();
	}
}
```

通过set方法注入

```
public class UserServiceImpl implements UserService {
	private UserDAO dao;

	//提供set方法以实现注入
	public void setDao(UserDAO dao) {
		this.dao = dao;
	}

	public void save() {
		dao.save();
	}
}
```

通过set方法注入

```
<!-- 定义Service实现类的Bean，id的名称任意 -->
<bean id="userService" class="com.spring.service.UserServiceImpl">
        <!-- 以set方法的形式注入引用值是id为userDaoMysqlImpl的实例 -->
        <property name="dao" ref="userDaoMysqlImpl"/>
</bean>
```

内部bean

- ✓ 外部Bean类可以使用，但是该bean不能被其他bean使用
- ✓ 示例

```
<bean id="userService" class="com.spring.service.UserServiceImpl">
        <property name="dao">
                <bean class="com.spring.dao.UserDaoMysqlImpl"/>
        </property>
</bean>
```

延迟加载

- ✓ 默认情况下，当Spring容器对象被启动，配置文件中所有声明的Bean类都会自动创建对象，在上例的每个构造方法中加入打印语句就就会发现这个问题，可以在配置文件的<bean>标签中配置一下lazy-init属性值为true，即延迟加载，只有第一次使用到这个Bean类对象的时候才会创建对象
- ✓ 例如：
  - ➢ <bean id="userDaoOracleImpl" class="com.spring.dao.UserDaoOracleImpl" lazy-init="true"/>
  - ➢ <bean id="userDaoMysqlImpl" class="com.spring.dao.UserDaoMysqlImpl" lazy-init="true"/>
- ✓ 如果希望所有的Bean都延迟加载，可以在<beans>根标签中设置default-lazy-init=“true”，这样所有的Bean都会延时加载
- ✓ 例如
  - ➢ <beans default-lazy-init="true">

容器中Bean的作用域的问题

- ✓ 当Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以通过scope属性为Bean指定特定的作用域，比较常用的是singleton和prototype两种作用域
  - ➢ singleton：单例模式，Bean实例只产生一次，默认
  - ➢ prototype：程序每次请求该id的Bean，spring都会新建一个Bean实例，spring容器实例化的时候不会实例化bean，只有在通过容器getBean的时候实例化
- ✓ 示例

```
<bean id="userService" class="com.spring.service.UserServiceImpl"
        scope="prototype">
```

普通属性注入——类声明

```
public class MyBean1 {
    private String strValue;
    private int intValue;
    private String[] arrayValue;
    private List listValue;
    private Set setValue;
    private Map mapValue;

    //get和set方法
}
```

普通属性注入——配置

```
<!-- 普通属性注入 -->
<bean id="myBean1"
    class="com.spring.MyBean1">

    <property name="strValue" value="hello"/>
    <!-- 或者写成以下形式
    <property name="strValue">
        <value>world</value>
    </property>
    -->
    <property name="intValue" value="100"/>
    <property name="arrayValue">
        <list>
            <value>array1</value>
            <value>array2</value>
        </list>
    </property>
    <property name="listValue">
        <list>
            <value>list1</value>
            <value>list2</value>
        </list>
    </property>
    <property name="setValue">
        <set>
            <value>set1</value>
            <value>set2</value>
        </set>
    </property>
    <property name="mapValue">
        <map>
            <entry key="key1" value="value1"/>
            <entry key="key2" value="value2"/>
        </map>
    </property>
</bean>
```

日期时间类型注入

- ✓ 配置文件中设置的值都是字符串类型，如果是日期时间可能会出现问题
- ✓ 例如有一个日期属性需要注入：

  private Date dateValue;

- ✓ 配置：

  <property name="dateValue">
  <value>2009-10-19</value>
  </property>

- ✓ 这时候需要定义一个属性编辑器类把字符串值转换为日期时间类型注入

自动装配

- ✓ Spring的自动装配可以简化配置，无需使用ref属性显式的指定依赖关系
- ✓ 有两种自动装配的方法
  - ➢ 根据名称
  - ➢ 根据类型
- ✓ 实现自动装配式通过bean标签的autowire属性实现的，常见的有三个属性值
  - ➢ no：不使用自动装配，默认，也是推荐使用，显式的设置依赖关系可以是维护清晰
  - ➢ byName：根据属性名自动装配，查找配置的所有Bean，找出id名称和属性名相同的bean注入
  - ➢ byType：根据属性类型自动装配，查找配置的所有Bean，找出类型和属性类型相同的bean注入，如果没有找到，则什么也不发生，如果找到多个这个类型的bean，则抛出致命异常

自动装配示例

```
<!-- 正常手动装配 -->
<bean id="bean1" class="com.spring.autowire.Bean1">
    <property name="bean2" ref="bean2" />
</bean>
<!-- 通过名称自动装配
<bean id="bean1" class="com.spring.autowire.Bean1" autowire="byName"/>
-->
<!-- 或者通过类型自动装配
<bean id="bean1" class="com.spring.autowire.Bean1" autowire="byType"/>
-->
<bean id="bean2" class="com.spring.autowire.Bean2" >
    <property name="name" value="tom"/>
    <property name="age" value="25"/>
</bean>
```

公共属性注入

- ✓ 公共属性比较象java中的继承，在父类中定义的属性，子类中可以直接获得无需自己定义
- ✓ 我们可以把多个Bean中相同的那一部分属性抽取出来，放到父Bean中，然后子Bean设置parent属性继承父Bean
- ✓ 父Bean本身不需要被创建对象实例，可以加上abstract=”true”抽象属性防止容器初始化的时候实例化父Bean，加上abstract属性，就不需要加class属性

公共属性注入示例

```
package com.spring;
public class Bean3 {
    private int id;
    private String name;
    private String password;
    //getter和setter……
}

package com.spring;
public class Bean4 {
    private int id;
    private String name;
    //getter和setter……
}
```

公共属性注入示例

```
<bean id="abstractBean" abstract="true">
        <property name="id" value="1000"/>
        <property name="name" value="Jack"/>
</bean>

<bean id="bean3" class="com.spring.Bean3" parent="abstractBean">
        <property name="password" value="123456"/>
</bean>

<bean id="bean4" class="com.spring.Bean4" parent="abstractBean"/>
```