Spring 培训教材

腾科 Java 教学部

目录

1.Spring 入门

Spring是一个开源框架，它由Rod Johnson创建。它是为了解决企业应用开发的复杂性而创建的。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

目的：解决企业应用开发的复杂性；

功能：负责管理JavaBean对象的生命周期，主要是用来降低模块与模块之间的耦合度；

范围：任何Java应用；

简单来说，**Spring**是一个轻量级的控制反转**(IoC)**和面向切面**(AOP)**的容器框架。

1.1. Spring 的特性

- **轻量**——从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的处理开销也是微不足道的。此外，Spring是非侵入式的，典型地，Spring应用中的对象不依赖于Spring的特定类。
- **控制反转(IOC)**——Spring通过一种称作控制反转（IoC-Inverse of control）的技术促进了松耦合。
- **面向切面(AOP)**——Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务(例如审计(auditing)和事务(Transaction)管理)进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。同时，实现在不修改组件的前提下，为组件提供扩展的服务。
- **容器**——Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置原型（prototype），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。
- **框架**——Spring可以将简单的组件配置、组合成为复杂的应用。Spring也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

1.2. 术语解释

A. 轻量级：相对于重量级的容器(如EJB容器)来说的，需要的资源很少，一个轻量级容器有如下特点：
   1)非侵入性：可以管理业务代码，但是不应该给代码强加入对容器的依赖。
   2)可以快速启动。
   3)部署简单。
   4)轻量级容器是用纯JAVA开发的，不依赖JavaEE容器。

B. 非侵入性：应该程序对容器的依赖很少，几乎可以感受不到框架的存在，依赖注入就体现了这一点；

C. 容器：指应用代码的运行框架。容器可以管理对象的生成、资源取得、销毁等生命周期，甚至可以建立对象与对象之间的依赖关系。

D. IOC(Inversion of Control)：中文译为控制反转，目前Java社群中流行的各种轻量级容器的实现都是以IoC模式作为基础的。控制反转意味着在系统开发过程中，设计的类将交由容器去控制，而不是在类的内部去控制，类与类之间的关系将交由容器处理,一个类

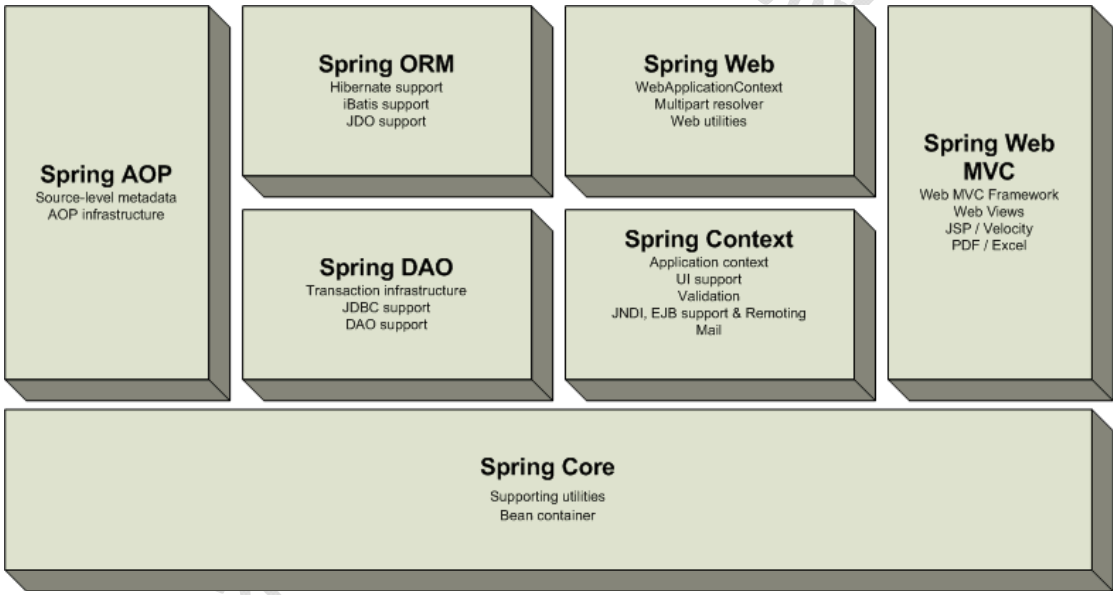
在需要调用另一个类时,只要调用另一个类在容器中注册的名字就可以得到这个类的实例,与传统的编程方式有了很大的不同,”Don’t call me,I’ll call you”,这就是控制反转的含义。

E. DI(Dependency Inection)：中文为依赖注入，控制反转是同一个概念。具体含义是:当某个角色(可能是一个Java实例，调用者)需要另一个角色(另一个Java实例，被调用者)的协助时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在Spring里，创建被调用者的工作不再由调用者来完成，因此称为控制反转;创建被调用者实例的工作通常由Spring容器来完成，然后注入调用者，因此也称为依赖注入。

F. AOP(Aspect-oriented programming)：面向切面编程

G. 持久层：直接操作持久化数据的层面。Spring提供对持久层的整合，如对JDBC的使用加以封装与简化，提供程式事务(Programmatic Transaction)与声明式事务(Declarative Trasaction)管理功能。对于O/R Mapping工具(Hibernate、iBATIS)的整合及使用上的简化，Spring也提供了对应的解决方案。

1.3. Spring 框架的七个模块



(1) 核心容器：

核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

(2) Spring 上下文：

Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

在开发中，需要配置【ApplicationContext.xml】配置文件，实现那对核心容器中的【Bean Factory】的扩展与支持。

(3) Spring AOP

通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架

中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

(4) Spring DAO

JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

从功能上简单讲，该模块简化 JDBC，和事务的声明。

(5) Spring ORM

Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

该模块简化使用 Hibernate ,toplink 等持久化框架，方便的事务管理

(6) Spring Web 模块

Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。

(7) Spring MVC 框架

MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

1.4. Spring 框架的优势

1. 使 J2EE 易用和促进好编程习惯

Spring 不重新开发已有的东西。因此，在 Spring 中你将发现没有日志记录的包,没有连接池,没有分布事务调度。这些均有开源项目提供(例如 Commons Logging 用来做所有的日志输出，或 Commons DBCP 用来作数据连接池),或由你的应用程序服务器提供。因为同样的的原因，我们没有提供 O/R mapping 层，对此，已有好的解决办法如 Hibernate 和 JPA。

2. 使已存在的技术更加易用

例如,尽管我们没有底层事务协调处理，但我们提供了一个抽象层覆盖了 JTA 或任何其他的事务策略。

3. 没有直接和其他的开源项目竞争。

例如，许多开发人员，我们从来没有为 Struts 高兴过，并且感到在 MVC 框架中还有改进的余地。在某些领域，例如轻量级的 IoC 容器和 AOP 框架，Spring 有直接的竞争，但是

在这些领域还没有已经较为流行的解决方案。(Spring 在这些区域是开路先锋。)

4. Spring 在应用服务器之间是可移植的。

保证可移植性总是一项挑战，但是 Spring 避免任何特定平台或非标准化,并且支持在 WebLogic，Tomcat，Resin，JBoss，WebSphere 和其他的应用服务器上的用户。

5. 方便解耦，简化开发。

通过 Spring 提供的 IoC 容器，我们可以将对象之间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。有了 Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

6. AOP 编程的支持。

通过 Spring 提供的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付。

7. 声明式事务的支持

在 Spring 中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

8. 方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，在 Spring 里，测试不再是昂贵的操作，而是随手可做的事情。

9. 方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，相反，Spring 可以降低各种框架的使用难度，Spring 提供了对各种优秀框架（如 Struts,Hibernate）等的直接支持。

10. 降低 Java EE API 的使用难度

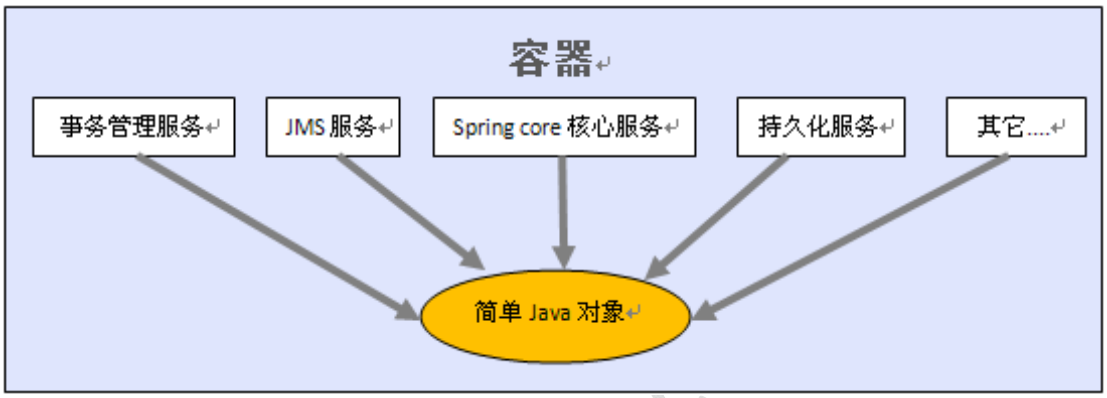
Spring 对很多难用的 Java EE API（如 JDBC，JavaMail，远程调用等）提供了一个薄薄的封装层，通过 Spring 的简易封装，这些 Java EE API 的使用难度大为降低。

1.5. 为何使用 spring

至少在我看来，在项目中引入 spring 立即可以带来下面的好处

- 降低组件之间的耦合度,实现软件各层之间的解耦。
- 可以使用容器提供的众多服务，如：事务管理服务、消息服务等等。当我们使用容器管理事务时，开发人员就不再需要手工控制事务.也不需处理复杂的事务传播。
- 容器提供单例模式支持，开发人员不再需要自己编写实现代码。

- 容器提供了 AOP 技术，利用它很容易实现如权限拦截、运行期监控等功能。
- 容器提供的众多辅作类，使用这些类能够加快应用的开发，如： JdbcTemplate、HibernateTemplate。
- Spring 对于主流的应用框架提供了集成支持，如：集成 Hibernate、JPA、Struts 等，这样更便于应用的开发。



1.6. 第一个 spring 例子

1.6.1. 搭建 Spring 开发框架

A. 直接建立一个 JAVA 项目

B. 导入 spring 的 jar 包（这里以 Spring3.2 为例）

- spring-core-3.2.0.M1.jar -- 核心依赖 jar 包
- spring-context-3.2.0.M1.jar -- Spring 容器包
- spring-beans-3.2.0.M1.jar Spring -- beans 的管理包
- spring-expression-3.2.0.M1.jar
- spring-asm-3.2.0.M1.jar

注：和hibernate一起用时这个JAR会冲突，解决方法删掉它就是了
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'sessionFactory' defined in ServletContext resource [/WEB-INF/classes/applicationContext.xml]: Invocation of init method failed; nested exception is java.lang.NoSuchMethodError: org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/String;Ljava/lang/String;)
Caused by: java.lang.NoSuchMethodError:org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/String;Ljava/lang/String;)

除此之外，还有需要一个 Apache common 的 JAR 包 ：

- commons-logging-1.1.1.jar-- 日志记录

注：如果忘记添加会报错
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory

C. 编写 XML 配置文件

Spring 的最大的作用就是提供 bean 的管理功能，在 spring 中 bean 的管理是通过 XML 实现的，要用此功能，需要把 bean 配置到 spring 的 xml

1. 新建立一个*.xml，*名字任意，如 applicationContext.xml,或者 text.xml 都可以
2. 添加 xml 头定义

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
>
<bean></bean>
</beans>
```

说明：

① **Xmlns(XML NameSpace):**声明命名空间，建议是用网址做命名空间，但并不会去访问改网址，仅仅是 namespace 和 xsd（xsd 是 spring 这个 xml 的 schema 文件，里面定义配置内容）里声明的 targetNamespace 保持一致 .

② **xsi:schemaLoacation**:用于绑定命名空间的 schema 文件，通常是用 URL 值对，中间用空格隔开,前面 URL 是命名空间，后面 URL 为 schema 的文件地址

1.6.2. 测试案例

(1) Bean 类

Spring 是基于对 JavaBean 类的管理，因而在使用 Spring 框架时，编写 Bean 类时必不可少的，下面是一个典型的 Bean 类，也即对某一类型的抽象，并使用 Java 封装特性完成。

```
public class User {
    private String userName;
    private String userPassword;
    public User(){
    }
    public User( String userName, String userPassword ){
        this.userName = userName;
        this.userPassword = userPassword;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getUserPassword() {
        return userPassword;
    }
    public void setUserPassword(String userPassword) {
```

```
        this.userPassword = userPassword;
    }
}
```

(2) Bean 的配置：

实现 Bean 类开发后，需要在 Spring 配置文件中对这个 Bean 类进行配置。

A. 该配置文件可以在任意路径下，习惯是在 class 路径下（就是类的根目录下，如 src 下），习惯名称为【spring-config.xml/ApplicationContext.xml】。

B. 配置文件对 Bean 类的管理使用 IOC，通过依赖注入实现 Bean 的实例化。下面使用依赖注入中的构造器注入。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <constructor-arg index="0">
            <value>lomen</value>
        </constructor-arg>
        <constructor-arg index="1">
            <value>123456</value>
        </constructor-arg>
    </bean>
</beans>
```

(3) 业务逻辑编写

```
public class Main {
    public static void main( String [] args ){
       FileSystemResource fs =
            new FileSystemResource("src/spring-config.xml");
        BeanFactory bf = new XmlBeanFactory(fs);
            //获取一个实例
        User user = ( User )bf.getBean( "userBean" );
        System.out.println( user.getUserName() + " - " +
user.getUserPassword() );
    }
}
```

2. Spring IOC

2.1. 关于 IoC

(1) IoC 定义

- IoC：全称是 Inversion of Control，反转控制。
- IoC 模式：又称为 DI,即 Dependency Injection,叫做依赖注入或依赖注入

(2) IoC 的意义

- IoC 设计模式,重点关注组件的依赖性,配置以及生命周期.
- 它主要是为了解决程序设计中松散耦合的问题,依赖关系不出现在任何代码中,而只出现在 XML 配置文件里.

2.2. IoC – 构造器注入

- 意义：
  - 通过构造函数，来初始化 JavaBean 中各属性的值。

(1) Bean 的开发

```
public class User {
    private String userName;
    private String userPassword;
    public User(){}
    public User( String userName, String userPassword ){
        this.userName = userName;
        this.userPassword = userPassword;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getUserPassword() {
        return userPassword;
    }
    public void setUserPassword(String userPassword) {
        this.userPassword = userPassword;
    }
}
```

(2) Bean 的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <constructor-arg index="0">
            <value>lomen</value>
        </constructor-arg>
        <constructor-arg index="1">
            <value>123456</value>
        </constructor-arg>
    </bean>
</beans>
```

(3) 业务逻辑编写

```
public class Main {
    public static void main( String [] args ){
        ApplicationContext ac = new
FileSystemXmlApplicationContext( "src/spring-config.xml" );
            //获取一个实例
        User user = ( User )ac.getBean( "userBean" );
        System.out.println( user.getUserName() + " - " +
user.getUserPassword() );
    }
}
```

如果没有这个构造器将报错：

```
Exception in thread "main"
org.springframework.beans.factory.BeanCrea
tionException: Error creating bean with name 'userBean' defined in
file [D:\note\spring\src\spring-config.xml]: 2 constructor
arguments specified but no matching constructor found in bean
'userBean' (hint: specify index arguments for simple parameters to
avoid type ambiguities)
```

如果构造器中没有写任何代码将会：

得到结果为：null - null

2.3. IoC - Setter 方法注入

【Setter注入】方式是先通过无参数构造器实例化Bean类对象，再通过Bean类里的Set()方法来初始化属性值，从而实现依赖注入。Bean类开发和业务逻辑的编写无区别，区别在于Bean的配置。

(1) Bean 的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.
Springfram
ework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <property name="userName">
            <value>Eric</value>
        </property>
        <property name="userPassword">
            <value>12345678</value>
        </property>
    </bean>
</beans>
```

2.4. IoC – 混合注入

【混合注入】是允许同时混合使用构造器注入和 Setter 注入两者，即先使用有参数构造

器实例化部分属性，再通过 Setter 方法为剩余属性实现注入。适用于 Bean 类中部分属性固定值或者自动生成，而另一部分属性需要依赖外部输入的情况。

(1) Bean 的开发

```
public class User {
    private String userName;
    private String userPassword;
    public User(){}
    public User( String userName ){
        this.userName = userName;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getUserPassword() {
        return userPassword;
    }
    public void setUserPassword(String userPassword) {
        this.userPassword = userPassword;
    }
}
```

(2) Bean 的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <constructor-arg index="0">
            <value>lomen</value>
        </constructor-arg>
        <property name="userPassword">
            <value>12345678</value>
        </property>
    </bean>
</beans>
```

执行逻辑与之前一样

2.5. 指定注入类型

指定注入类型，指的是在实现对属性的注入时，指定注入值的类型，以便能符合 Bean 类属性的类型。

(1) Setter 注入

在使用 Setter 注入时，8 种基本数据类型，能够自动转换

(2) 构造器注入

默认为 String 类型或者指定注入类型

<value **type="int"**>100</value>

```
Bean
    public Student( int studentAge ) {
            this.studentAge = studentAge;
    }
Bean-配置
<constructor-arg index="0">
    <value type="int">20</value>
</constructor-arg>
```

3. Bean 的应用与配置

3.1. Bean 的配置

spring 可调用 bean 的构造方法，或通过工厂方法生产 bean 对象。

(1) 利用 bean 的构造方法创建 bean

- 无参的构造方法
- 有参的构造方法，需指明构造方法的参数列表；

通过以上方法获得的 bean 实例默认都是单例的

- sigleton="true|false"
- scope 属性 scope="singleton|prototype|request|session"；表示 javabean 的取值范围，默认值是 singleton，其它值如下：
  - prototype：每一次请求 spring 容器会创建一个新的 javabean 对象
  - singleton：单例，一个类只创建一个对象
  - request|session：后两者用于 web 应用中

注意：如果 javabean 主要的作用是用来保存数据，则用多例；如果 javabean 主要是提供一些功能让调用使用，则用单例

3.2. 读取 Bean 配置资源文件

- ClassPathResource：在类路径下查找资源
- FileSystemResource：在系统路径下查找资源
- XmlBeanFactory：
  - 读取 xml 文件中的配置信息
  - 生产 bean
  - 解决 bean 依赖

```
FileSystemResource fs = new
                        FileSystemResource("F:/spring-config.xml");
       BeanFactory bf = new XmlBeanFactory(fs);
       User user = (User)bf.getBean("sAndCBean");
//要求classpath配置【.】,【./】取当前项目路径
ClassPathResource cr = new ClassPathResource("./spring-config.xml");
```

```
        bf = new XmlBeanFactory(cr);
        User user2 = (User)bf.getBean("sAndCBean");
```

3.3. 集合的注入

在 Bean 类中，某些属性类型为集合，也可实现依赖注入。

依赖的目标类型分成三种形式：

1) 基本类型+String：<value>data</value> --类型自动转化
2) 对其他 bean 的引用：<ref bean="target"/>
3) 集合类型

(1) Bean 的配置

- Setter 方法的 Bean 的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframe
work.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <property name="list">
            <list>
                <value>篮球</value>
                <value>足球</value>
                <value>乒乓球</value>
            </list>
        </property>
        <property name="set">
            <set>
                <value>C</value>
                <value>Java</value>
                <value>VB</value>
            </set>
        </property>
        <property name="map">
            <map>
                <entry key="a">
                    <value>A</value>
                </entry>
                <entry key="b">
                    <value>B</value>
                </entry>
                <entry key="c">
                    <value>C</value>
                </entry>
            </map>
        </property>
```

```
    </bean>
</beans>
```

- 构造器注入的 Bean 的配置（本例的构造器为 public User(List < String > list, Map < String, String > map, Set < String > set)）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <constructor-arg index="0">
            <list>
                <value>篮球</value>
                <value>足球</value>
                <value>乒乓球</value>
            </list>
        </constructor-arg>
        <constructor-arg index="2">
            <set>
                <value>C</value>
                <value>Java</value>
                <value>VB</value>
            </set>
        </constructor-arg>
        <constructor-arg index="1">
            <map>
                <entry key="a">
                    <value>A</value>
                </entry>
                <entry key="b">
                    <value>B</value>
                </entry>
                <entry key="c">
                    <value>C</value>
                </entry>
            </map>
        </constructor-arg>
    </bean>
</beans>
```

(2) Bean 的开发

- Setter 方法注入的 Bean 开发：

```
public class User {
    private List < String > list = new Vector();
    private Map < String, String > map = new HashMap < String, String >
```

```
();
    private Set < String > set = new HashSet < String > ();
    public List<String> getList() {
        return list;
    }
    public void setList(List<String> list) {
        this.list = list;
    }
    public Map<String, String> getMap() {
        return map;
    }
    public void setMap(Map<String, String> map) {
        this.map = map;
    }
    public Set<String> getSet() {
        return set;
    }
    public void setSet(Set<String> set) {
        this.set = set;
    }
```

(3) 构造器注入的 Bean 开发：

```
public class User {
    private List < String > list = new Vector();
    private Map < String, String > map = new HashMap < String, String >
();
    private Set < String > set = new HashSet < String > ();
    public User(){}
    public User( List < String > list, Map < String, String > map,
Set < String > set ){
        this.list = list;
        this.set = set;
        this.map = map;
    }
    public List<String> getList() {
        return list;
    }
    public void setList(List<String> list) {
        this.list = list;
    }
    public Map<String, String> getMap() {
        return map;
    }
    public void setMap(Map<String, String> map) {
        this.map = map;
```

```
    }
    public Set<String> getSet() {
        return set;
    }
    public void setSet(Set<String> set) {
        this.set = set;
    }
}
```

(4) 业务逻辑

```
public class Main {
    public static void main( String [] args ){
        ApplicationContext ac = new
FileSystemXmlApplicationContext( "src/spring-config.xml" );
        //获取一个实例
        User user = ( User )ac.getBean( "userBean" );
        List < String > list = user.getList();
        Map < String, String > map = user.getMap();
        Set < String > set = user.getSet();
        // 打印
        for( String value: list ) {
            System.out.println( value );
        }
        for( String value: set ){
            System.out.println( value );
        }
        Set < String > keySet = map.keySet();
        for( String key: keySet ){
            String value = map.get( key );
            System.out.println( key + " - " + value );
        }
    }
}
```

3.4. 管理 Bean 生命周期

(1) 生命周期方法执行顺序：

无参构造方法→setXxx()→回调初始化方法(init-method,destroy)→getBean 返回

(2) 回调方法：

A、两个属性指明

init-method

destroy-method

B、实现两个接口：

InitializingBean

DisposableBean

(3) 测试实例：

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
public class LifeBean implements InitializingBean,DisposableBean {
    //对象被消毁时会调用到的方法 ， 方法名随意写 ApplicationContext
    public void destroy(){
        System.out.println("LifeBean's destroy()===>"+this);
    }
    public void afterPropertiesSet() throws Exception {
        // 获取当前Bean里的对象
        System.out.println("LifeBean's
afterPropertiesSet()===>"+this);
    }
}
<!-- 测试生命周期方法  -->
<bean id="lifeBean" class="com.ioc.beans.LifeBean"></bean>
```

3.5. 其它 Bean 设置

(1) null 值设置

```
<property name="list">
        <null/>
</property>
```

(2) 其他类型数据的设置

```
<property name="number">
        <value type="int">500</value>
</property>
```

(3) 重用 bean 定义

- parent：继承 bean 定义，bean 之间不一定有继承关系
- abstract：只作为模板，不可以被实例化
- depends-on：要创建某个 javabean 对象，必须先创建 depend-on 指定的 javabean 对象

(4) 其他 Bean 的引用设置

以下为被引用类

```
public class Dd {
    private String test;
    public String getTest() {
        return test;
    }
    public void setTest(String test) {
        this.test = test;
    }
}
```

以下为引用类

```
public class User {
    private Dd dd;
```

```
    public Dd getDd() {
        return dd;
    }
    public void setDd(Dd dd) {
        this.dd = dd;
    }
}
```

✧ Bean 的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframe
work.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userBean" class="ioc.User">
        <property name="dd">
            <ref bean="Dd"/>
        </property>
    </bean>
    <bean id="Dd" class="ioc.Dd">
        <property name="test">
            <value>test</value>
        </property>
    </bean>
</beans>
```

(5) 让容器检测 bean 所有的依赖是否都已经满足

某个 bean 需要设置好所有属性，防止遗漏装配，在运行后出现莫名奇妙的情况

- dependency-check 属性指明应检查的目标类型
  - simple: 基本类型+字符串
  - objects: 集合+对其他bean的依赖
  - all：simple+objects
  - none:默认值

```
<bean id="iocBean4" class="com.ioc.beans.IocBean"
dependency-check="all">
        <property name="strVar" value="str var..."></property>
        <property name="intVar" value="30"></property>
        <property name="booleanVar" value="true"></property>
        <!--测试集合注入，list集合注入 -->
        <property name="list">
            <list>
                <value>张三</value>
                <value>李四</value>
                <value>王五</value>
            </list>
```

```
        </property>
        <!-- set集合注入 -->
        <property name="set">
            <set>
                <value>one</value>
                <value>two</value>
                <value>three</value>
            </set>
        </property>
        <!-- map集合注入 -->
        <property name="map">
            <map>
                <entry key="u1" value-ref="user"></entry>
                <entry key="u2">
                    <ref bean="u2"/>
                </entry>
            </map>
        </property>
        <!-- property注入 Properties extends Hashtable -->
        <property name="pro">
            <props>
                <prop key="url">
                    jdbc:mysql://127.0.0.1:3306/togogo
                </prop>
                <prop key="user">root</prop>
            </props>
        </property>
        <property name="user" ref="user"></property>
        <property name="userDao" ref="userDao"></property>
    </bean>
```

3.6. ApplicationContext

ApplicationContext：扩展了BeanFactory的功能；扩展功能有：

- 1)预先实例化在描述文件所描述所有的javabean实例；
- 2)国际化；
- 3)事件框架；

3.6.1. ApplicationContext 实例化 Bean

利用 ApplicationContext 预先对 Bean 的实例化，可以世界获取 Bean 对象

```
//实例化Bean
ApplicationContext ac= new FileSystemXmlApplicationContext(
            "src/spring-config.xml");//从项目 src 开始
ApplicationContext ac= new ClassPathXmlApplicationContext(
            "spring-config.xml");//从项目的 src 下开始
```

```
// 2)获取实例
        User user = (User) ac.getBean("userBean");
```

3.6.2. ApplicationContext 实现国际化

1) 注册消息源

```
<!-- 注册消息源 -->
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessa
geSource">
        <property name="basename">
            <value>MessageResource</value>
        </property>
</bean>
```

注意：bean的id必须是"messageSource"

如果想要使用多个资源文件，可以用如下描述：

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessa
geSource">
    <property name="basenames">
        <list>
            <value>MessageResource</value>
            <value>abc</value>
        </list>
    </property>
</bean>
```

注意（1）：以上的配置，国际化文件需要放在src目录下，如果放下其它目录下：

```
<property name="basename">
        <value>com/Ioc/global/MessageResource</value>
</property>
```

注意（2）：配置多个国际化文件时，name="basenames",值为复数形式的。

另外还可以这么用：

```
<property name="basenames">
            <value>MessageResource,abc</value>
</property>
```

在同时又多个国际化文件时，如果不同的文件存在相同的key，找到的是前面一个的值，也即按照顺序查找，找到后就不再继续找了

2) 使用资源文件的步骤：

建立properties文件：

A、MessageResource_en.properties：

wellcomeword=hello,{0}

B、MessageResource_zh_CN.properties:

wellcomeword=您好,{0}

3) 读取消息

- 方法一：String getMessage(String key, Object[] args, Locale loc)

第一个参数表示资源文件中的key

第二个参数是一个Object类型的数组，用来替换资源文件中的{0},{1}...

第三个参数：表示地区，如果是null，使用操作系统默认地区

* Locale 语言和区域代码的封装对象：

```
public static void getMessage1(){
        String     message     =     ac.getMessage("hello",     new
String[]{"World"},Locale.ENGLISH);
        System.out.println(message);
    }
```

- 方法二：String getMessage(String key, Object[] args, String default, Locale loc)

第一个参数:表示资源文件中的key

第二个参数:是一个Object类型的数组，用来替换资源文件中的{0},{1}...

第三个参数：表示key如果在资源里找不到，使用的缺省值

第四个参数:表示地区，如果是null，使用操作系统默认地区

注：方法一没有指定默认值，如果没找到消息，会抛出一个NoSuchMessageException异常。

原理：当一个ApplicationContext被加载时，它会自动在context中查找已定义为MessageSource类型的bean。此bean的名称须为messageSource。如果找到，那么所有对上述方法的调用将被委托给该bean。

3.7. 高级配置：

3.7.1. CustomEditorConfigurer

CustomEditorConfigurer：一个拦截器，在注入属性值的时候，可以做类型转换

步骤：

A、开发一个类型转化类：

```
public class MyEditor extends PropertyEditorSupport {
    private        SimpleDateFormat        sdf        =        new
SimpleDateFormat("yyyy-mm-dd");
    private Date birth;
    //spring会将该方法的返回值注入到相应的属性
    public Object getValue() {
        System.out.println("MyEditor's getValue()...");
        return birth;
    }
    //注入属性前被调用，text是注入的属性值
```

```
    public void setAsText(String text) throws
IllegalArgumentException {
        System.out.println("MyEditor's setAsText(String text)...");
        try {
            birth = sdf.parse(text);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

}
```

B、在配置文件注册CustomEditorConfigurer

```
<bean
class="org.springframework.beans.factory.config.CustomEditorConfi
gurer">
        <property name="customEditors">
            <map>
                <entry key="java.util.Date">
                    <bean class="com.senior.MyEditor" />
                </entry>
            </map>
        </property>
</bean>
```

3.7.2. PropertyPlaceHolderConfigurer

1. 加载PropertyPlaceholderConfigurer,将一些配置信息搬到属性文件里

```
<bean
class="org.springframework.beans.factory.config.PropertyPlacehold
erConfigurer">
        <!-- 注册一个属性文件 -->
        <!--
        <property name="location">
            <value>day2/db.properties</value>
        </property>
        -->
        <!-- 注册多个属性文件 -->
        <property name="locations">
            <list>
                <value>day2/db2.properties</value>
                <value>day2/db.properties</value>
            </list>
        </property>
    </bean>
```

2. 属性类的Bean配置

```
<bean id="dbDao" class="day2.com.senior.DBDaoImpl">
    <property name="className">
        <value>${jdbc.className}</value>
    </property>
    <property name="url">
        <value>${jdbc.url}</value>
    </property>
    <property name="username">
        <value>${jdbc.username}</value>
    </property>
    <property name="password">
        <value>${jdbc.password}</value>
    </property>
    <property name="loginTime">
        <value>1986-6-5</value>
    </property>
    <property name="user">
        <value>stone:123</value>
    </property>
</bean>
```

3. 配置文件db.properties:

```
jdbc.className=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/togogo
```

4. AOP 面向切面编程

4.1. AOP-面向切面编程

AOP ： 全称是 Aspect-Oriented Programming，中文翻译是面向方面的编程或者面向切面的编程。

功能：Spring 基于对 JavaBean 的管理，在系统中还有日志、事件、权限管理等，采用 AOP 是在调用 Bean 之前或者之后实现这些功能，将系统切分为多个层面来进行开发。

常常通过 AOP 来处理一些具有横切性质的系统性服务，如事物管理、安全检查、缓存、对象池管理等，AOP 已经成为一种非常常用的解决方案

4.2. AOP 术语

(1) 切面（Aspect）

一个关注的模块化。这个关注点可能会横切多个对象。事务管理是 J2EE 应用中一个关于横切关注点的很好的例子。在 Spring AOP 中，切面可以使用通用类（基于模式的风格）或者在普通类中以@Aspect 注解（@Aspect 风格）来实现。

即：当前关注的一个代码的流程，其中可能调用了多个类的多个方法。

(2) 连接点（Joinpoint）

程序执行的某个特定位置：如类开始初始化前、类初始化后、类某个方法调用前、调用后、方法抛出异常后。这些代码中的特定点，称为“连接点”。Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前后这些程序执行点织入

增强。

(3) 切点（Pointcut）

每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点。但在这为数众多的连接点中，如何定位到某个感兴趣的连接点上呢？AOP 通过“切点”定位特定的连接点。通过数据库查询的概念来理解切点和连接点的关系：连接点相当于数据库中的记录，而切点相当于查询条件。一个切点可以匹配多个连接点。

Spring 中，切点通过 Pointcut 接口进行描述。

(4) 增强（Advice）

增强是织入到目标类连接点上的一段程序代码。增强既包含了用于添加到目标连接点上的一段执行逻辑，又包含了用于定位连接点的方为信息，所以 Spring 所提供的增强接口都是带方位名的：BeforeAdvice（方法调用前的位置）、AfterReturningAdvice（访问返回后的位置）、ThrowsAdvice 等。

(5) AOP 代理（AOP Proxy）

AOP 框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。

Spring 缺省使用 J2SE 动态代理（dynamic proxies）来作为 AOP 的代理。这样任何接口都可以被代理。

Spring 也支持使用 CGLIB 代理. 对于需要代理类而不是代理接口的时候 CGLIB 代理是很有必要的。 如果一个业务对象并没有实现一个接口，默认就会使用 CGLIB。 作为面向接口编程的最佳实践，业务对象通常都会实现一个或多个接口。

(6) Advisor

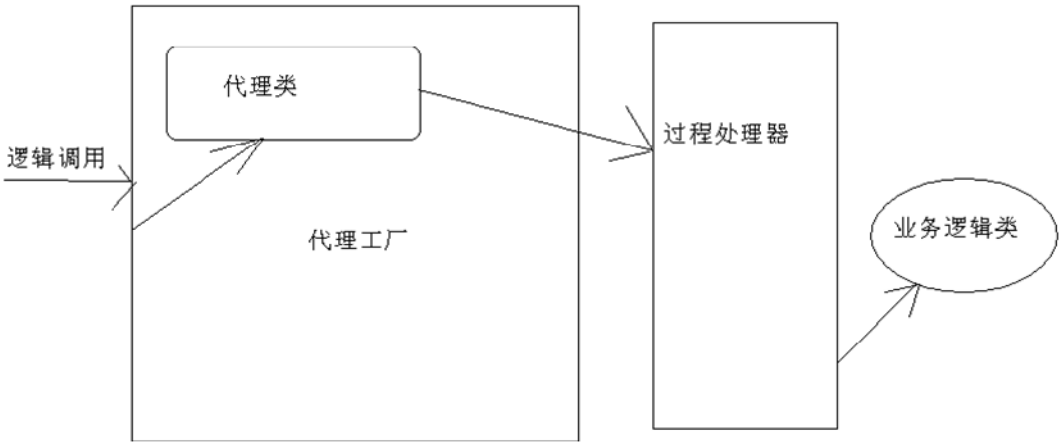
切入点配置器，由 Pointcut 和 Advice 共同构成

(7) AOP 的意义

- 现在的系统往往强调减小模块之间的耦合度，AOP 技术就是用来帮助实现这一目标的。
- 从某种角度上来讲“切面”是一个非常形象的描述，它好像在系统的功能之上横切一刀，要想让系统的功能继续，就必须先过了这个切面。
- 这些切面监视并拦截系统的行为，在某些（被指定的）行为执行之前或之后执行一些附加的任务（比如记录日志）。
- 而系统的功能流程（比如 Greeting）并不知道这些切面的存在，更不依赖于这些切面，这样就降低了系统模块之间的耦合度。
- Spring 默认使用 J2SE 动态代理（dynamic proxies）来作为 AOP 的代理。这样任何接口都可以被代理。

4.3. AOP 的机制

4.3.1. Java 动态代理模型



1. 原始业务逻辑类：

(1) 以下是业务逻辑的接口

```
public interface IUserService {
    public void register( String userName, String userPassword );
    public void delete( String userName );
}
```

(2) 以下是具体的业务逻辑实现

```
public class UserService implements IUserService {
    public void register( String userName, String userPassword ) {
        System.out.println( userName + " - " + userPassword );
        System.out.println( "欢迎你，" + userName );
    }
    public void delete(String userName) {
        System.out.println( "删除" + userName );
    }
}
```

2. 过程处理&处理类：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
/**
 * InvocationHandler是一个Listener,
 * 我们将会给我们的代理对象加入这个监听器,以便可以对代理对象进行监听,借此可以
实现在方法执行前后加入一些操作.
*/
```

```
public class MyProxy implements InvocationHandler {
    private Object source;
    /**
     * 创建代理对象
     * @param c 指被代理对象的类元对象
     * @param source 指被代理的对象
     */
    public Object createProxy(Class c, Object source) {
        this.source = source;
        // 得到要代理的业务逻辑类的ClassLoader
        ClassLoader loader = c.getClassLoader();
        // 得到要代理的业务逻辑类的接口
        Class[] interfaces = c.getInterfaces();
        /**
         * Proxy的newProxyInstance负责给被代理对象创建代理对象
         * 参数1:ClassLoader(告诉JVM到什么地方找寻该类)
         * 参数2:Class[](一个类数组,实现该接口的所有类)
         * 参数3:监听器(实现了InvocationHandler接口的对象都是监听器)
         */
        Object objProxy = Proxy.newProxyInstance(loader, interfaces,
this);
        //由Proxy生产出来的代理对象objProxy，继承了被代理类的方法，实现了被代
理类的接口，并由监听器监听其执行。
        return objProxy;
    }
    /**
     * invoke方法是InvocationHandler接口的方法,在监听器被触发的时候被执行
     * @param proxy  :要调用的方法的对象
     * @param method : 要调用的方法的方法元对象,
     * @param args   : 要调用的方法参数，如sayHello("Tom")中的Tom
     */
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println("开始写日志...");
        Object obj = null;
        try {
            // 反射的method.invoke(...)方法,表示真正执行
            obj = method.invoke(source, args);//
        } catch (Exception e) {e.printStackTrace();}
        System.out.println("方法执行完毕,你可以在这作一些释放操作!");
        // 返回执行结果
        return obj;
    }
}
```

执行业务

```
// 首先声明代理类
MyProxy myProxy = new MyProxy();
//声明被代理类对象
IUserSerivce us = new UserService();
//为该对象创建代理，并通过转接口类型，伪装成业务逻辑类
IUserService userService =
            (IUserService)myProxy.createProxy(us.getClass(), us);
// 调用业务方法
userService.register( "lomen", "123" );
userService.delete( "test" );
```

4.3.2.Spring AOP

1. 编写拦截器

Spring 以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。即：对于某个方法（连接点），进行拦截的拦截规则。

A. Before advice（前置通知）：在目标对象的方法执行之前被调用,可以实现MethodBeforeAdvice。但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
B. After returning advice（后通知）：在目标方法正常完成执行之后被调用，可以实现AfterReturningAdvice
C. Around advice（环绕通知）：在目标方法执行前后介入 Advices 的服务逻辑，可以实现 MethodInterceptor
D. After throwing advice(抛出异常后通知)：方法抛出异常时被调用，可以实现ThrowsAdvice

(1) Before Advice

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class Before implements MethodBeforeAdvice {
    /**
     * @param method - 将被调用的方法元对象
     * @param args  - 将被调用的方法对应的实参值
     * @param target  - 将被调用的方法对应的对象
     */
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        System.out.print(target.getClass().getSimpleName() + "的"
                + method.getName() + "方法将被调用，参数列表：(");
        for (Object o : args) {
            System.out.print(o.toString() + ",");
        }
        System.out.println(")");
    }
}
```

(2) After Advice

```
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
public class AfterReturn implements AfterReturningAdvice {
    /**
     * @param returnValue  :  目标方法 返回值
     * @param method   : 将被调用的方法元对象
     * @param args : 将被调用的方法对应的实参值
     * @param target  : 将被调用的方法对应的对象
     */
    public void afterReturning(Object returnValue, Method method,
Object[] args,Object target) throws Throwable {
        System.out.print(target.getClass().getSimpleName() + "的"
                + method.getName() + "方法调用完毕，参数列表：(");
        for (Object o : args) {
            System.out.print(o.toString()+",");
        }
        System.out.println("),返回值："+returnValue);
    }
}
```

(3) Around Advice

```
import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class Around implements MethodInterceptor {
    /**
     * @param invocation：目标方法对象
     * @return 可以返回目标方法返回值
     */
    public Object invoke(MethodInvocation invocation) throws
Throwable {
        // 获得目标方法对应对象的类元对象
        Class c = invocation.getMethod().getDeclaringClass();
        // 获得目标方法的元对象,来自java.lang.reflect
        Method m = invocation.getMethod();
        //获得目标方法的参数列表
        Object[] objects = invocation.getArguments();
        //获得目标方法的实现类
        Object o = invocation.getThis();
        System.out.println(o.getClass().getSimpleName());
        System.out.println("Around before:"+c.getSimpleName()+"的
"+m.getName()+"方法将被调用");
        //调用目标的方法
        Object obj = invocation.proceed();
        System.out.println("Around after:"+c.getSimpleName()+"的
```

```
"+m.getName()+"方法调用完毕");
        return obj;
    }
}
```

(4) Throw Advice

```
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
public class Throw implements ThrowsAdvice {
    /**
     * 方法出异常时调用该方法，可以在这里记录异常信息，但这里不会处理异常，异常仍然需要调用者来处理
     * @param method：目标方法元对象
     * @param args：目标方法参数列表
     * @param target：目标方法对应的对象
     * @param ex：目标方法抛出的异常
     */
    public void afterThrowing(Method method, Object[] args, Object target,Exception ex) {
        System.out.println(target.getClass().getSimpleName() + "类的"+ method.getName() + "出异常了，异常信息：" + ex.getMessage());
    }
}
```

2. 原始业务逻辑类：

(1) 以下是业务逻辑的接口

```
public interface IUserService {
    public void register( String userName, String userPassword );
    public void delete( String userName );
}
```

(2) 以下是具体的业务逻辑实现

```
public class UserService implements IUserService {
    private User user;
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
    public void register(String userName, String userPassword) {
        System.out.println(userName + " - " + userPassword);
        this.user.setUserName(userName);
        this.user.setUserPassword(userPassword);
        System.out.println("欢迎你，" + user.toString());
```

```
    }
    public void delete(String userName) {
        System.out.println("删除" + userName);
        this.user = null;
    }
}
```

3. Bean 的配置

```
<!-- 描述advice -->
    <bean id="before" class="com.AOP.advice.Before"></bean>
    <bean id="after" class="com.AOP.advice.AfterReturn"></bean>
    <bean id="around" class="com.AOP.advice.Around"></bean>
    <bean id="throw" class="com.AOP.advice.Throw"></bean>
    <!-- 声明bean对象 -->
    <bean id="user" class="com.AOP.aoptest.User"></bean>
    <bean id="userService" class="com.AOP.aoptest.UserServiceImpl">
        <property name="user" ref="user"></property>
    </bean>
    <!-- 自动为指定javabean对象产生代理对象 -->
    <bean
    class="org.springframework.aop.framework.autoproxy.BeanNameAut
oProxyCreator">
        <property name="beanNames">
            <list>
                <value>user</value>
                <value>userService</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>before</value>
                <value>after</value>
                <value>around</value>
                <value>throw</value>
            </list>
        </property>
    </bean>
```

测试

```
ApplicationContext ac = new
        FileSystemXmlApplicationContext("src/spring-config.xml");
IUserService userService = (IUserService) ac.getBean("userService");
userService.register("cjz2", "123");
```

注意：要使上面的配置生效，需要借助ApplicationContext

4.3.3.Spring AOP 标注方式(重点掌握)

1. 方法切点函数：

1）execution

【execution([可见性] 方法返回值 完整类名+方法名(参数类型))[异常]】

- ◆ 可见性（可选）- 使用public、protected、private指定可见性。也可以为*，表示任意可见性
- ◆ 异常模式（可选）- 指定方法签名中是否存在异常类型

在pointcut表达式中可以使用 * 、.. 、 +等通配符。

- ◆ * ：表示若干字符（排除 . 在外）
- ◆ .. ：表示若干字符（包括 . 在内）
- ◆ + ：表示子类 ，比如Info+ 表示Info类及其子类

例如：

A、execution(public void *(..))：所有返回值是public void的方法都会被拦截到

B、execution(public void day6.com.beans.PersonService.*(..))：表示day6.com.beans.PersonService下所有返回值是public void的方法都会被拦截到

C、execution(public void day6.com.beans.PersonService.save(java.lang.String,..))：表示day6.com.beans.PersonService类中的第一个形参类型是String的save方法会被拦截到

D、execution(public void save(..))：表示所有类中的save方法会被拦截到

E、execution(public void day6.com.service..*(..))：表示day6.com.service包下的类以及子包的类所有public void方法都会被拦截到

F、execution(public !void day6.com.service..*(..))：表示day6.com.service包下的类以及子包的类所有public 不是void返回类型的方法都会被拦截到

2）@annotation

@annotation(java.lang.Override)：加上Override注释的方法都会被拦截到

2. 方法参数切点函数：

1）args:

例如：args(java.lang.String,..)，指包含该类型参数的方法

2）@args()

登录：@args(java.lang.Override)表示任何这样的一个目标方法：它有一个形参且形参对象的类标注@Override注解。

3. 目标类切点函数：

1）within()

如within(com.baobaotao.service.*)表示com.baobaotao.service包中的所有 连接点，也即包中所有类的所有方法，而within(com.baobaotao.service.*Service)表示在 com.baobaotao.service包中，所有以Service结尾的类的所有连接点。

2）target()

如target(com.baobaotao.Waiter)定义的切点，Waiter、以及Waiter实现类NaiveWaiter中所有连接点都匹配该切点。

3）@target()

如@target(java.lang.Override)，假如NaiveWaiter标注了@ Override，则

NaiveWaiter所有连接点匹配切点。

4) @within()

4. 几种通知标注方式

- 前置通知（Before advice）：在切入点匹配的方法执行之前运行使用@Before注解来声明
- 返回后通知（After returning advice）：在切入点匹配的方法返回的时候执行。使用 @AfterReturning 注解来声明
- 抛出后通知（After throwing advice）：在切入点匹配的方法执行时抛出异常的时候运行。使用 @AfterThrowing 注解来声明
- 后通知（After (finally) advice）：不论切入点匹配的方法是正常结束的，还是抛出异常结束的，在它结束后（finally）后通知（After (finally) advice）都会运行。使用 @After 注解来声明。这个通知必须做好处理正常返回和异常返回两种情况。通常用来释放资源。
- 环绕通知（Around Advice）：环绕通知既在切入点匹配的方法执行之前又在执行之后运行。并且，它可以决定这个方法在什么时候执行，如何执行，甚至是否执行。在环绕通知中，除了可以自由添加需要的横切功能以外，还需要负责主动调用连接点(通过 proceed)来执行激活连接点的程序。 **请尽量使用最简单的满足你需求的通知。（比如如果前置通知也可以适用的情况下，就不要使用环绕通知）**

使用步骤：

1）在工程里加入：

【..\spring-framework-2.5.6\lib\aspectj\】的【aspectjrt.jar】和【aspectweaver.jar】

2）编写拦截器：

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class AspectBeanAnnotation {
    //统一定义一个拦截规则
    @Pointcut("execution(public void com.aop.IHello.sayHello(..))")
    public void myMethod(){
    }
    @Before("execution(public void com.aop.IHello.*(..))")
    //调用业务方法之前由该方法先统一处理
    public void before(){
        System.out.println("方法调用之前开始写日志...");
    }
    @After("myMethod()")
    //调用业务方法之后由该方法先统一处理
```

```
    public void myAfter(){
        System.out.println("方法调用完毕，开始写日志...");
    }
    @Around("myMethod()")
    //对应around方法
    public Object process(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("进入目标方法...");
        //调用目标方法,obj是目标方法的返回值
        Object obj = pjp.proceed();
        System.out.println("退出目标方法...");
        return obj;
    }
    @AfterThrowing("myMethod()")
    //业务方法出异常时调用该方法
    public void error(){
        System.out.println("你调用的方法出异常了...");
    }}
```

3) 编写描述文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/bean
s http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.
xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    <!-- 为每个javabean自动产生代理对象，类似于BeanNameAutoProxyCreator
该功能 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
    <bean id="aspectbean"
class="com.aop.aspect.AspectBeanAnnotation"></bean>
    <!-- 描述javabean -->
    <bean id="hello" class="com.aop. HelloImpl"></bean>
    <bean id="word" class="com.aop.WordImpl"></bean>
</beans>
```

4.3.4.Spring AOP 注解方式

```
<!-- 测试@annotation -->
<aop:pointcut id="mycut2" expression="@annotation(day6.com.annotation.Test)"/>
<!-- 测试args -->
<aop:pointcut id="mycut3" expression="args(java.lang.String,..)"/>
<!-- 测试@args -->
<aop:pointcut id="mycut4" expression="@args(day6.com.annotation.Test)"/>
<!-- 测试within -->
<aop:pointcut id="mycut5" expression="within(day6.com.service.*)"/>
<!-- 测试target -->
<aop:pointcut id="mycut6" expression="target(day6.com.service.IPersonService)"/>
```

advice：

```
<aop:before pointcut-ref="mycut" method="doAccessCheck"/>
<aop:after pointcut-ref="mycut" method="doAfter"/>
<aop:after-returning pointcut-ref="mycut" method="doAfterReturning"/>
<aop:after-throwing pointcut-ref="mycut" method="doAfterThrowing"/>
<aop:around pointcut-ref="mycut" method="doBasicProfiling"/>
```

(1) 在 spring 配置文件里加上 aop 的名称空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/bean
s http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.
xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    <!-- 为每个javabean自动产生代理对象，类似于BeanNameAutoProxyCreator
该功能 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
    <bean id="aspectbean" class="com.aop.aspect.AspectBean"></bean>
    <!-- 描述javabean -->
    <bean id="hello" class="com.aop.dynaproxy.HelloImpl"></bean>
    <bean id="word" class="com.aop.dynaproxy.WordImpl"></bean>
    <!-- 拦截规则配置 -->
    <aop:config>
```

```
        <aop:aspect id="asp" ref="aspectbean">
            <!-- 只有IHello所有方法经过切面，其它类的方法不经过切面
execution语法：execution(方法返回值 完整类名+方法名(参数类型)) -->
            <aop:pointcut expression="execution(*
                com.aop.dynaproxy.IHello.*(..))"
                id="mypointcut" />
            <!-- 只有IHello中的sayHello()方法经过切面，其它类的方法不经过切
面 -->
            <!-- <aop:pointcut expression="execution(*
com.aop.dynaproxy.IHello.sayHello(..))"
                id="mypointcut" /> -->
            <!-- 只有IHello中的sayHello(String)方法经过切面，其它类的方法
和其它sayHello()方法不经过切面 -->
            <!-- <aop:pointcut expression="execution(public void
com.aop.dynaproxy.IHello.sayHello(java.lang.String))"
                id="mypointcut" /> -->
            <aop:pointcut expression="execution(public void
com.aop.dynaproxy.IHello.*(..))"
                id="mypointcut2" />
            <!-- 测试args -->
            <aop:pointcut id="mycut3"
expression="args(java.lang.String,..)" />
            <!-- 测试within -->
            <aop:pointcut id="mycut5"
expression="within(com.aop.dynaproxy.*)" />
            <!-- 测试target -->
            <aop:pointcut id="mycut6"
expression="target(com.aop.dynaproxy.IHello)" />

            <aop:before method="before" pointcut-ref="mypointcut" />
            <aop:after method="myAfter" pointcut-ref="mypointcut" />
            <aop:around method="process" pointcut-ref="mycut6" />
            <aop:after-throwing method="error"
pointcut-ref="mypointcut" />
        </aop:aspect>
    </aop:config>
</beans>
```

(2) 编写 Interceptor

```
import org.aspectj.lang.ProceedingJoinPoint;
public class AspectBean {
    //调用业务方法之前由该方法先统一处理
    public void before(){
        System.out.println("方法调用之前开始写日志...");
```

```
    }
    //调用业务方法之后由该方法先统一处理
    public void myAfter(){
        System.out.println("方法调用完毕，开始写日志...");
    }
    //对应around方法
    public Object process(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("进入目标方法...");
        //调用目标方法,obj是目标方法的返回值
        Object obj = pjp.proceed();
        System.out.println("退出目标方法...");
        return obj;
    }
    //业务方法出异常时调用该方法
    public void error(){
        System.out.println("你调用的方法出异常了...");
    }
}
```

5. Spring DAO

5.1. 传统的 JDBC

传统型 JDBC 有许多积极的方面使之在许多 J2SE 和 J2EE 应用程序开发中占有重要地位。然而，也有一些特征使其难于使用：

- 开发者需要处理大量复杂的任务和基础结构，例如大量的 try-catch-finally-try-catch 块
- 应用程序需要复杂的错误处理以确定连接在使用后被正确关闭，这样以来使得代码变得冗长，膨胀，并且重复。
- JDBC 中使用了极不明确性的 SQLException 异常。
- JDBC 没有引入具体的异常子类层次机制。

5.2. Spring JDBC 模板

(1) Spring 所提供的 JDBC 模板由四个不同的包组成：

- 核心包包含 JdbcTemplate。这个类是一个基础类之一-由 Spring 框架的 JDBC 支持提供并使用。
- 数据源包是实现单元测试数据库存取代码的重要的一部分。它的 DriverManagerDataSource 能够以一种类似于你已经习惯于 JDBC 中的用法：只要创建一个新的 DriverManagerDataSource 并且调用 setter 方法来设置 DriverClassName，Url，Username 和 Password。
- 对象包中包含类，用于描述 RDBMS 查询、更改和存储过程为线程安全的、可重用的对象。

◆ 支持包-你可以从这里找到 SQLException 翻译功能和一些工具类。

(2) 模板设计模式

Spring JDBC 实现模板设计模式，这意味着，代码中的重复的复杂的任务部分是在模板类中实现的。这种方式简化了 JDBC 的使用，因为由它来处理资源的创建和释放。这有助于避免普通错误，例如忘记关闭连接等。它执行核心 JDBC 工作流任务，如语句创建和执行，而让应用程序代码来提供 SQL 并且提取结果。

(3) Spring JDBC 异常处理

Spring 框架特别强调在传统型 JDBC 编程中所面临的与下列方案有关的问题：

◆ Spring 提供一个抽象异常层，把冗长并且易出错误的异常处理从应用程序代码移到由框架来实现。框架负责所有的异常处理;应用程序代码则能够专注于使用适当的 SQL 提取结果。

◆ Spring 提供了一个重要的异常类层次，以便于你的应用程序代码中可以使用恰当的 SQLException 子类。

5.3. JdbcTemplate

JdbcTemplate 是 Spring JDBC 框架中最重要的类。引用文献中的话：“它简化了 JDBC 的使用，有助于避免常见的错误。它执行核心 JDBC 工作流，保留应用代码以提供 SQL 和提取结果。”这个类通过执行下面的样板任务来帮助分离 JDBC DAO 代码的静态部分：

◆ 从数据源检索连接。
◆ 准备合适的声明对象。
◆ 执行 SQL CRUD 操作。
◆ 遍历结果集，然后将结果填入标准的 collection 对象。
◆ 处理 SQLException 异常并将其转换成更加特定于错误的异常层次结构。

(1) 导 Jar 包

项目当中需要有导入：【commons-dbcp.jar】

(2) Bean 配置

借助于 Spring 的 IOC，配置连接数据库的基本信息的 JavaBean：

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost:3306/togogo</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value>admin</value>
```

```
        </property>
</bean>
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg index="0">
        <ref bean="dataSource" />
    </constructor-arg>
</bean>
```

(3) 业务逻辑

```
ApplicationContext ac = new FileSystemXmlApplicationContext(
            "src/com/dao/spring-config-jdbc.xml");
//借助IOC注入获得JdbcTemplate对象
JdbcTemplate jt = (JdbcTemplate) ac.getBean("jdbcTemplate");
//执行SQL语句
String create_sql = "create table table03(id int primary key,name
varchar(20),password varchar(20));";
String drop_spl="drop table table03";
    jt.execute(drop_spl);
    jt.execute(create_sql);
//插入信息
String insert_sql = "insert into table03 values(?,?,?)";
    jt.update(insert_sql,new Object[]{1,"cjz","123"});
        //查询单个结果
String query_sql = "select * from table03;";
    Map results1 = jt.queryForMap(query_sql);
    System.out.println("results1:"+results1.get("id")+","+results1
.get("name")+","+results1.get("password"));
        //更新信息
String update_sql = "update table03 set password = ? where id=?";
    jt.update(update_sql,new Object[]{"1234",1});
//查询多个结果
String query_sql2 = "select * from table03;";
    List results2 = jt.queryForList(query_sql2);
    for(int i=0;i<results2.size();i++){
        Map map = (Map)results2.get(i);
    System.out.println("results2:"+map.get("id")+","+map.get("name
")+","+map.get("password"));

    }
```

练习：

编写一个 UserDao,实现增删改查等功能，以下是 IUserDao 接口：

```
public interface IUserDao {
    //插入指定用户的信息
```

```
    public boolean insert(User user);
    //更新指定用户的信息
    public boolean update(User user);
    //根据指定信息查询用户信息
    public User query(User user);
    //查询用户信息列表
    public List<User> querys();
}
```

编写一个 UserService，实现登陆，注册，查看个人信息，修改个人信息，查看用户列表功能：

```
public interface IUserService {
    public boolean login(User user);
    public boolean register(User user);
    public boolean update(User user);
    public User find(User user);
    public List<User> findAll();
}
```

5.4. JdbcDaoSupport

JdbcDaoSupport 是 JDBC 数据访问对象的超类。它与特定的数据源相关联。Spring Inversion of Control(IOC)容器或 BeanFactory 负责获得相应数据源的配置详细信息，并将其与 JdbcDaoSupport 相关联。这个类最重要的功能就是使子类可以使用 JdbcTemplate 对象。

导入包:

【 org.springframewrok.jdbc-3.0.4.RELEASE.jar 】， JdbcDaoSupport 位于该包的【org.springframework.jdbc.core.support.JdbcDaoSupport】路径

Bean 配置

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/togogo</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value>togogo</value>
    </property>
</bean>
<bean id="userDao" class="com.orm.jdbc.dao.JdbcUserDaoImpl">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

Dao 类

对项目中的 Dao 类继承 JdbcDaoSupport，便可直接获取 JdbcTemplate

```
public class JdbcUserDaoImpl extends JdbcDaoSupport implements
IUserDao {
    @Override
    public void delUserById(int id) {
        String sql = "delete from orm_user where id=?";
        this.getJdbcTemplate().update(sql,new Object[]{id});
    }
}
```

5.5. 声明式事务处理

1. 声明事务管理器：class -【DataSourceTransactionManager】

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransacti
onManager">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>
</bean>
```

2. 使用spring aop配置事务属性

1) 配置文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    <!-- 产生代理对象来完成事务的管理 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
    <bean id="ds" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName">
```

```
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost:3306/hibernate_db</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value>togogo</value>
        </property>
    </bean>
    <bean id="userDao" class="com.springdao.beans.UserDaoImpl"
init-method="init">
        <property name="dataSource" ref="ds"></property>
    </bean>
    <bean id="transaction" class="com.springdao.beans.TransactionImpl">
        <property name="dataSource" ref="ds"></property>
        <property name="userDao">
            <ref bean="userDao"/>
        </property>
    </bean>
     <!-- 声明事务管理器 -->
    <bean id="transactionManager"

    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
>
        <property name="dataSource" ref="ds" />
    </bean>
    <!--
    配置事务属性，常见事务属性：
    1、REQUIRED :支持当前的事务，如果不存在就创建一个新的。这是最常用的选择。
    2、SUPPORTS :支持当前的事务，如果不存在就不使用事务。
    3、MANDATORY :支持当前的事务，如果不存在就抛出异常。
    4、REQUIRES_NEW :创建一个新的事务，并暂停当前的事务（如果存在）。
    5、NOT_SUPPORTED :不使用事务，并暂停当前的事务（如果存在）。
    6、NEVER :不使用事务，如果当前存在事务就抛出异常。
    7、NESTED :如果当前存在事务就作为嵌入事务执行，否则与
```

```
PROPAGATION_REQUIRED类似。
    8、readOnly：表示该方法是只读的，而不能更新数据库里的数据
    -->
    <tx:advice id="tx" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="test*" propagation="REQUIRED"/>
            <tx:method name="delete*" propagation="REQUIRED"/>
            <tx:method name="find*" read-only="true"/>
        </tx:attributes>
    </tx:advice>
    <aop:config>
        <aop:pointcut expression="execution(* com.springdao.beans..*(..))"
id="txCut"/>
        <aop:advisor advice-ref="tx" pointcut-ref="txCut"/>
    </aop:config>
</beans>
```

2) 事务属性含义

- PROPAGATION_MANDATORY:
  规定了方法必须在事务中运行，否则会抛出异常
- PROPAGATION_NESTED:
  使方法运行在嵌套事务中，否则这个属性和PROPAGATION_REQUIRED属性的义相同
- PROPAGATION_NEVER
  使当前方法永远不在事务中运行，否则抛出异常
- PROPAGATION_NOT_SUPPORTED
  定义为当前事务不支持的方法，在该方法运行期间正在运行的事务会被暂停
- PROPAGATION_REQUIRED
  规定当前的方法必须在事务中，如果没有事务就创建一个新事务，一个新事务和方法一同开始，随着方法的返回或抛出异常而终止
- PROPAGATION_REQUIRED_NEW
  当前方法必须创建新的事务来运行，如果现存的事务正在运行就暂停它
- PROPAGATION_SUPPORTS
  规定当前方法支持当前事务处理，但如果没有事务在运行就使用非事务方法执行
- readOnly：表示该方法是只读的，而不能更新数据库里的数据

6.Spring ORM

Spring 在资源管理，Dao 实现支持以及事务策略等方面提供了与 Hibernate，JDO，Oralce TopLink,Apache OJB,Ibatis SQL Mapping 以及 JPA 的集成。以 Hibernate 为例，Spring 通过使用许多 IOC 的便捷特性对它提供了一流的支持，帮助你处理很多典型的 Hibernate 整合的问题。所有的这些支持都遵循 Spring 通用的事务和 Dao 异常体系

6.1. HibernateDaoSupport

Spring 为 Hibernate 的 DAO 提供工具类：HibernateDaoSupport。该类主要提供了两个方法：

- public final HibernateTemplate getHibernateTemplate()
- public final void setSessionFactory(SessionFactory sessionFactory)

其中，setSessionFactory 方法接收来自 Spring 的 applicationContext 的依赖注入，接收了配置在 Spring 中的 SessionFactory 实例，getHibernateTemplate 方法用来利用刚才的 SessionFactory 生成 Session，再生成 HibernateTemplate 来完成数据库的访问。

(1) 导包

实现 Hibernate 的支持，需要：【org.springframework.orm-3.0.5.RELEASE.jar】

(2) Bean配置

这里直接读取 Hibernate 的配置文件，可以实现与 Hibernate 的整合。

类 LocalSessionFactoryBean 来自上面的包，是一个 SessionFactory.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryB
ean">
        <!-- 要求在hibernate.cfg.xml要设置自动提交 -->
        <property name="configLocation"
value="classpath:hibernate.cfg.xml"></property>
</bean>
    <bean id="userDao2" class="com.dao.Impl.UserDao2">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>
```

(3) Dao类的实现

Dao 实现类，需要继承 HibernateDaoSupport，就可以直接获取 HibernateTemplate。

```
import org.springframework.orm.hibernate3.HibernateTemplate;
import
org.springframework.orm.hibernate3.support.HibernateDaoSupport;
public class UserDaoImpl extends HibernateDaoSupport implements
IUserDao {
    public void save(User user) {
        HibernateTemplate ht = this.getHibernateTemplate();
        ht.save(user);
    }
    public User findUserById(int id) {
        User user = this.getHibernateTemplate().get(User.class, id);
        return user;
    }
    public List<User> findUsers() {
        String hql = "from User";
```

```
        List<User> list = this.getHibernateTemplate().find(hql);
        return list;
    }
    public void updateUser(User user) {
        this.getHibernateTemplate().update(user);
        User u = this.getHibernateTemplate().get(User.class, 2);
        u.setName("张三");
        this.getHibernateTemplate().update(u);
    }
    public void delete(User user) {
        this.getHibernateTemplate().delete(user);
    }
}
```

实 际上，DAO 的实现依然借助了 HibernateTemplate 的模板访问方式，只是，HibernateDaoSupport 将依赖注入 SessionFactory 的工作已经完成，获取 HibernateTemplate 的工作也已经完成。注意，这种方法须在 Spring 的配置文件中配置 SessionFactory。

在继承 HibrnateDaoSupport 的 DAO 实现里，Hibernate Session 的管理完全不需要 Hibernate 代码打开，而由 Spring 来管理。Spring 会根据实际的操作，采用“每次事务打开一次”session”的策略，自动提高数据库访问的性能。

6.2. Spring 对 Hibernate 事务支持与 Jdbc 模板类似

7. Spring Web(重点掌握)

Spring 中的 Web 包提供了基础的针对 Web 开发的集成特性，例如多方文件上传，利用 Servlet listeners 进行 IoC 容器初始化和针对 Web 的 application context。当与 WebWork 或 Struts 一起使用 Spring 时，这个包使 Spring 可与其他框架结合

7.1. 关键类

- **ContextLoaderListener：**它的作用就是启动 Web 容器时，自动装配 ApplicationContext 的配置信息。因为它实现了 ServletContextListener 这个接口，在 web.xml 配置这个监听器，启动容器时，就会默认执行它实现的方法
- **WebApplicationContext：**继承 ApplicationContext，它仅仅是一个拥有 web 应用必要功能的普通 ApplicationContext。它和一个标准的 ApplicationContext 的不同之处在于它能够解析主题，并且它知道和那个 servlet 关联（通过到 ServletContext 的连接）。WebApplicationContext 被绑定在 ServletContext 上，当你需要的时候，可以使用 RequestContextUtils 找到 WebApplicationContext。
- **WebApplicationContextUtils：**从 web 应用的根目录读取配置文件，需要先在 web.xml 中配置，可以配置监听器或者 servlet 来实现

```
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>
```

```
<servlet-name>context</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
```

```
<load-on-startup>1</load-on-startup>
</servlet>
```

这两种方式都默认配置文件为 web-inf/applicationContext.xml，也可使用 context-param 指定配置文件

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/myApplicationContext.xml</param-value>
</context-param>
```

7.2. 如何在 web 环境中配置 applicationContext.xml 文件

1. 在 web.xml 描述监听器

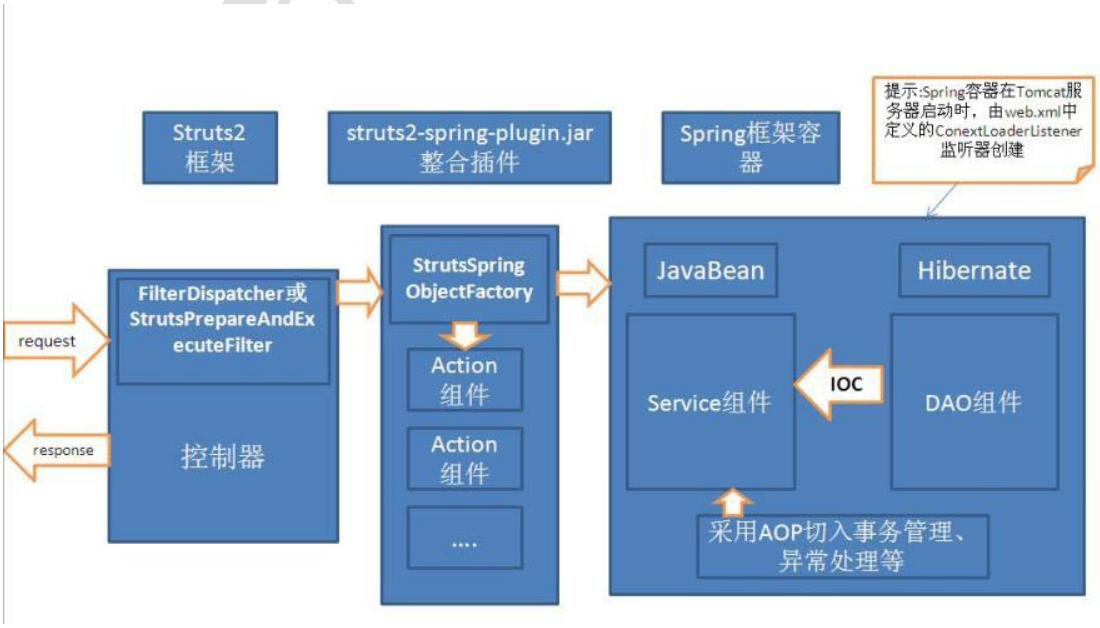
```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
</context-param>
    <!--web 监听器,默认会解析 classpath:applicationContext.xml -->
<listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

2. 获取 ApplicationContext 对象

```
ApplicationContext ac = WebApplicationContextUtils.getWebApplicationContext(servletContext)
```

7.3. ssh 整合开发

7.3.1. SSH 的整合方案之一



1. 处理流程

1) 请求（request）发出后，该请求要调用某个Action进行处理

2) 拦截器（FilterDistatcher）照惯例拦截请求（request）， 此时，如果拦截器（FilterDispatcher）发现项目中已经引入了 struts2-spring-plugin.jar整合插件 ,那么接下来 ,拦截器就将请求(request）交给Struts2-spring-plugin.jar整合插件来创建Action组件对象

在插件struts2-spring-plugin.jar中有个非常重要的类：

StrutsSpringObjectFactory：创建Action组件并且到Spring框架中将 Service组件和DAO组件取出，注入到Action中去 当然，在Spring框架内部，就各种使用IoC戒者AOP，就和我们ㄗ前讲的一样。

2. 整合步骤

1) 新建 web 工程，将 ssh 相关 jar 加到工程中

2) 在 web.xml 描述监听器和 struts2 控制器

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
</context-param>
    <!--web 监听器,默认会解析 classpath:applicationContext.xml -->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<!-- struts 控制器  -->
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

3) 在 src/struts.properties 描述

```
struts.objectFactory=spring
```

4) 在 src/struts.xml 文件描述

```
<action name="userAction" class="springUserAction">
    <result>/list.jsp</result>
</action>
```

5) 在 spring 配置文件 src/applicationContext.xml 描述

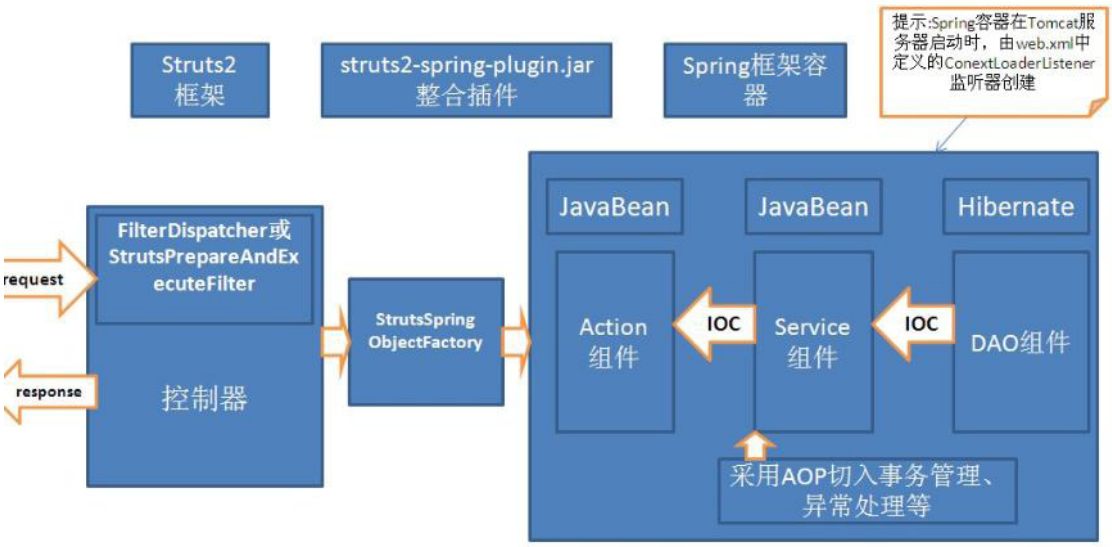
```
<bean id="springUserAction" class="com.actions.UserAction" singleton="false">
        <property name="userDao">
```

```
            <ref bean="userDao"/>
        </property>
</bean>
```

6) 编写 UserAction 类

```
public class UserAction{
    private UserDao userDao;
    ...
}
```

7.3.2. SSH 的整合方案之二



方案 2 中，Action 组件不再由整合插件创建了，Action 组件也被纳入到 Spring 容器当中； 整合插件虽然不再创建 Action 对象，但是我们仍然需要整合插件来访问 Spring 容器

1. 整合流程

与方案一处理流程类似

2. 整合步骤

1) 在 web.xml 描述监听器和 struts2 控制器

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <!--web 监听器,默认会解析/WEB-INF/applicationContext.xml-->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
```

```
<!-- struts 控制器 -->
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2) 编写 Action 类

```
public class BaseAction extends ActionSupport{
    private ApplicationContext context;
    public Object getBean(String beanId){
//      根据上下文获得 ApplicationContext 对象
        context = WebApplicationContextUtils.getWebApplicationContext(context);
        return context.getBean(beanId);
    }
}
```

```
public class StudentAction extends BaseAction {
    private User user;
    ...
    public String addStudent(){
        UserDao userDao = (UserDao)this.getBean("userDao");
        userDao.addStudent(user);
        return "addUser";
    }
}
```

3) 在 struts.xml 文件描述

```
<action name="userAction" class="com.action.UserAction">
        <result>/list.jsp</result>
</action>
```

7.4. 在 spring 配置 log4j

在我们的日常开发中，日志记录非常重要。 我们可以在测试中检测代码变量变化，跟踪代码运行轨迹。同时也可以创建一些基本的应用级别日志功能。 Log4j 是 Apache 的一个开放源代码项目，它提供了一种细腻的日志管理方式。 通过一个配置文件，我们可以多选择的控制每条日志的输出格式和目的地。通过定义信息的级别，我们也可以灵活开关代码中的反馈信息。

spring 配置 log4j 的步骤

1) 在 web.xml 中的定义

```
<!--
定义以后，在 Web Container 启动时将把 ROOT 的绝对路径写到系统变量里。
然后 log4j 的配置文件里就可以用${webApp.root}来表示 Web 目录的绝对路径，把 log
文件存放于 webapp 中
-->
<context-param>
    <param-name>webAppRootKey</param-name>
    <param-value>webApp.root</param-value>
</context-param>
<!-- 指定 log4j 配置文件的路径 -->
<context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
</context-param>
<!--Spring 默认刷新 Log4j 配置文件的间隔,单位为 millisecond-->
<context-param>
    <param-name>log4jRefreshInterval</param-name>
    <param-value>60000</param-value>
</context-param>
<!-- Log4jConfigListener 是 spring 提供的一个监听器，负责解析 log4j.properties 内容 -->
<listener>
    <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
</listener>
```

2) 编写 log4j.properties 文件

```
#rootLogger：日志记录器
#设置日志级别，级别有：DEBUG,INFO,WARN,ERROR,FATAL
#输出目的地，目的地可以有多个：ConsoleAppender，FileAppender，RollingFileAppender
log4j.rootLogger=INFO,appender1,appender2
#控制台目的地配置
#目的地的日志输出格式有：SimpleLayout，PatternLayout，HTMLLayout
log4j.appender.appender1=org.apache.log4j.ConsoleAppender
log4j.appender.appender1.layout=org.apache.log4j.PatternLayout
#日志信息格式中几个符号所代表的含义：
#    －X 号: X 信息输出时左对齐；
#     %p: 日志信息级别
#     %d{}: 日志信息产生时间
#     %c: 日志信息所在地（类名）
#     %m: 产生的日志具体信息
#     %n: 输出日志信息换行
log4j.appender.appender1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%c] [%p] %m%n
#普通文件目的地配置
log4j.appender.appender2=org.apache.log4j.FileAppender
#webApp.root 对应 web.xml 配置的<param-name>webAppRootKey</param-name>
log4j.appender.appender2.File=${webApp.root}/WEB-INF/logs/log.log
```

```
#TTCCLayout：用预定义的格式输出
log4j.appender.appender2.layout=org.apache.log4j.SimpleLayout
```

8. 邮件服务

邮件服务，在企业应用中很常见。借助于 Spring 提供的 JavaMail 抽象，开发者能够很容易同 JavaMail 进行交互，而不用理会太多的 JavaMail 细节。而且，这种抽象同 Spring 提供的其他 JavaEE 组件类似，这对于降低 Spring 的学习曲线很有帮助

8.1. Spring 对 JavaMail 的支持

目前，Spring 框架提供的 JavaMail 抽象主要由如下几个包构成。

- org.springframework.mail：Spring 框架提供的 JavaMail 抽象的主要内容都在这里。其中，包含了若干异常处理类，这同 Spring DAO 提供的异常处理类似，使得开发者有权选择是否需要捕捉未受查运行时异常。
- org.springframework.mail.cos：基于 Jason Hunter 提供的 COS 而开发的 MailSender 接口实现。这是一个基于 SMTP 协议的邮件发送实现
- org.springframework.mail.javamail：基于 JavaMail API 的 MailSender 实现。其中，还包括了 MimeMessageHelper、MimeMessagePreparator 等类

本章节主要介绍如何使用简单的 Spring 邮件抽象层来实现邮件发送功能，对于 JavaMail 中的 API 并不做介绍。通过对比 JavaMail 的 API 和 Spring 的邮件抽象层， Spring 的邮件抽象层优点就是简化了代码量，并能充分利用 IOC 功能；缺点就是要使用部分 Spring API，使程序与第三方框架耦合。关于这方面的内容，可以参考 Spring 的参考手册。

8.2. Spring 邮件抽象常用的接口和类

1. MailSender ，它是发送邮件的主要接口，代码如下：

```
public interface MailSender {
    void send(SimpleMailMessage[] simpleMessages) throws MailException;
}
```

2. 一个简单邮件信息的实现类 SimpleMailMessage 。
3. JavaMailSender 接口，提供使用 JavaMail 中 MimeMessage ，代码如下：

```
public interface JavaMailSender extends MailSender {
    MimeMessage createMimeMessage();
    MimeMessage createMimeMessage(InputStream contentStream) throws MailException;
    void send(MimeMessage mimeMessage) throws MailException;
    void send(MimeMessage[] mimeMessages) throws MailException;
    void send(MimeMessagePreparator mimeMessagePreparator) throws MailException;
    void send(MimeMessagePreparator[] mimeMessagePreparators) throws MailException;
}
```

4. MimeMessagePreparator ，支持 MimeMessage 的回调接口，代码如下：

```
public interface MimeMessagePreparator {
    void prepare(MimeMessage mimeMessage) throws Exception;
}
```

8.3. spring 邮件编程

8.3.1. 简单邮件开发

1. 导入 spring mail 所需要的两个 jar 包：mail.jar 和 activation.jar
2. 编写配置文件 mail_config.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"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="mailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <!-- 网易邮箱的smtp地址(发送邮件服务器地址) -->
        <property name="host" value="smtp.163.com"></property>
        <!-- 发送邮件服务器端口号-->
        <property name="port" value="25"></property>
        <!-- 发送邮件的用户名 -->
        <property name="username" value="stonefangsq@163.com"></property>
        <!-- 发送邮件用户的密码 -->
        <property name="password" value="..."></property>
        <!-- 设置发送邮件的一些属性 -->
        <property name="javaMailProperties">
            <props>
                <prop key="mail.smtp.auth">true</prop>
            </props>
        </property>
    </bean>
</beans>
```

3. 编写发送邮件类 MyMailSender

```
public class MyMailSender {
    public static void main(String[] args) {
            ApplicationContext ac = new
ClassPathXmlApplicationContext("com/mail/mail_config.xml");
        //获得邮件发送者对象
        JavaMailSender sender = (JavaMailSender)ac.getBean("mailSender");
        //创建邮件消息对象
        MimeMessage mime = sender.createMimeMessage();
        try {
            //设置邮件信息对象
```

```
            MimeMessageHelper helper = new
MimeMessageHelper(mime,true,"utf-8");
            //设置邮件发送者
            helper.setFrom("stonefangsq@163.com");
            //设置邮件接收者
            helper.setTo("1611834035@qq.com");
            //设置邮件主题
            helper.setSubject("大家好");
            //设置邮件内容
            helper.setText("欢迎使用spring发送邮件...");
            //发送邮件
            sender.send(mime);
            System.out.println("邮件发送成功...");
        } catch (MessagingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

8.3.2. 带有附件邮件开发

```
    public class AttachMailSender implements MimeMessagePreparator {
        public void prepare(MimeMessage mm) throws Exception {
            //设置邮件接收者
            mm.setRecipient(Message.RecipientType.TO, new InternetAddress(
                    "1611834035@qq.com"));
            //设置邮件发送者
            mm.setFrom(new InternetAddress("stonefangsq@163.com"));
            //设置邮件的主题
            mm.setSubject("测试邮件附件");
            //邮件内容对象
            Multipart mp = new MimeMultipart();
            //邮件主体对象，一个邮件内容对象可以包含多个邮件主体对象
            MimeBodyPart mbp = new MimeBodyPart();
            mbp.setText("发送邮件带附件");
            mp.addBodyPart(mbp);
            //设置附件文件
            String[] files = new String[] {

        "src/com/mail/MyMailSender.java","src/com/mail/mail_config.xml",};
            //将附件文件加到邮件内容对象中
            for (String f : files) {
```

```
                MimeBodyPart mbpFile = new MimeBodyPart();
                FileDataSource fds = new FileDataSource(f);
                mbpFile.setDataHandler(new DataHandler(fds));
                mbpFile.setFileName(fds.getName());
                mp.addBodyPart(mbpFile);
            }
            mm.setContent(mp);
            mm.setSentDate(new Date());
        }
        public static void main(String[] args) {
            BeanFactory bf = new
    ClassPathXmlApplicationContext("com/mail/mail_config.xml");
            JavaMailSender ms = (JavaMailSender)bf.getBean("mailSender");
            ms.send(new AttachMailSender());
        }
    }
```

8.4. 各大常用邮箱配置

- 163 邮箱

  POP3服务器:POP.163.COM
  SMTP服务器:SMTP.163.COM

- 126 邮箱

  POP3服务器:POP.126.COM
  SMTP服务器:SMTP.126.COM

- yahoo 邮箱

  注意：yahoo在foxmail 4.1以上的版本设置如下：(outlook 不行哟~)
  POP3服务器：pop.mail.yahoo.com
  SMTP服务器：smtp.mail.yahoo.com

- sohu 邮箱

  POP3服务器：pop3.sohu.com
  SMTP服务器：smtp.sohu.com

- sina 邮箱

  POP3服务器：pop.sina.com
  SMTP服务器：smtp.sina.com

- Gmail 邮箱

  POP3服务器是pop.gmail.com
  SMTP服务器是smtp.gmail.com

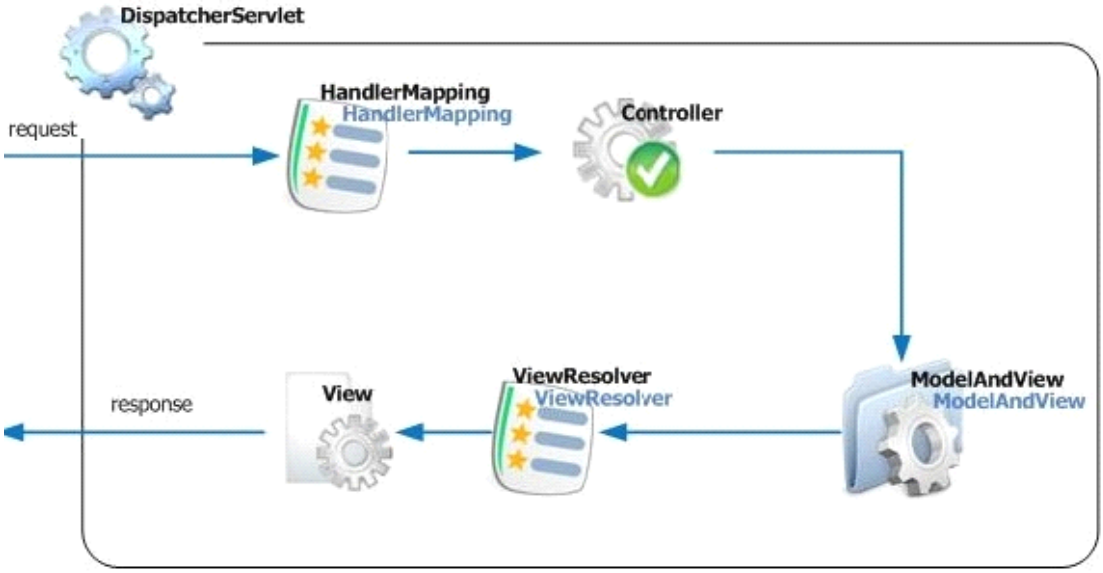
- QQ 邮箱

  POP3服务器：pop.qq.com
  SMTP服务器：smtp.qq.com
  SMTP服务器需要身份验证。

9.Spring Web MVC

Spring MVC 属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，可以选择是使用内置的 Spring Web 框架还可以是 Struts 这样的 Web 框架。通过策略接口，Spring 框架是高度可配置的，而且包含多种视图技术，例如 JavaServer Pages（JSP）技术、Velocity、Tiles、iText 和 POI。Spring MVC 框架并不知道使用的视图，所以不会强迫您只使用 JSP 技术。Spring MVC 分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制

9.1. Spring web mvc 工作流程

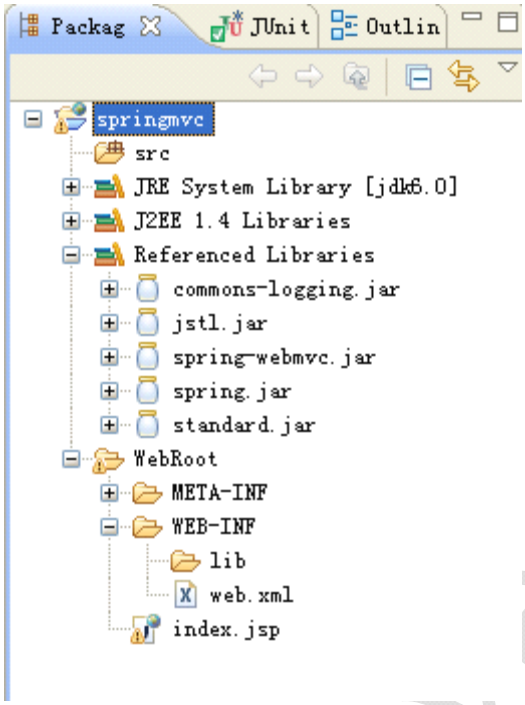


1. 客户端发送请求
2. 客户端发送的请求到达服务器控制器，服务器控制器由Servlet（DispatcherServlet）实现的，来完成请求的转发
3. 该控制器（DispatcherServlet）调用了一个用于映射的类HandlerMapping，该类用于将请求映射到对应的处理器来处理请求。
4. HandlerMapping 将请求映射到对应的处理器Controller（相当于Action），在Spring 当中如果写一些处理器组件，一般实现Controller 接口
5. 在Controller 中就可以调用一些Service 或DAO 来进行数据操作
6. ModelAndView 用于存放从DAO 中取出的数据，还可以存放响应视图的一些数据。
7. 如果想将处理结果返回给用户，那么在Spring 框架中还提供一个试图组件ViewResolver，该组件根据Controller 返回的标示，找到对应的试图，将响应response返回给用户

9.2. Spring web mvc 开发

9.2.1. 快速入门

1. 新建工程springmvc， 导入相关Jar包



2. 在web.xml配置DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <!-- 指定配置文件的位置-->
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <!-- 服务器启动即加载-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

3. 在工程src目录下新建spring-mvc.xml文件

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

4. 新建视图页面login.jsp

```
<%@ page contentType="text/html; charset=utf-8"
isELIgnored="false" pageEncoding="utf-8"%>
<html>
<head>
<title>Insert title here</title>
</head>
<body style="font-size:30px;">
<form action="login.do" method="post">
用户名: <input type="text" name="email"><br/>
密码: <input type="text" name="password"><br/>
<input type="submit" value="登录">
</form>
</body>
</html>
```

5. 新建视图页面ok.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8" %>
<html>
<head>
```

```
<title>登录成功页面</title>
</head>
<body>
<h2>登录成功!
</body>
</html>
```

6. 编写LoginControl

```
package control;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
public class LoginControl implements Controller {
@Override
    public ModelAndView handleRequest(HttpServletRequest arg0,
    HttpServletResponse arg1) throws Exception {
        String email = arg0.getParameter("email");
        String password = arg0.getParameter("password");
        if("stone@163.com".equals(email)&& "1111".equals(password)){
            return new ModelAndView("ok");
        }
        return new ModelAndView("login");
    }

}
```

7. 在spring-mvc.xml描述loginControl

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    <bean id="loginControl" class="control.LoginControl"></bean>
</beans>
```

8. **布署项目**

9. **访问http://127.0.0.1:8080/springmvc/login.jsp**

9.2.2. 相关 API 介绍

9.2.2.1. Controller

负责处理客户请求，并返回 ModelAndView 实例；Controller 必须实现接口 org.springframework.web.servlet.mvc.Controller，实现该接口中的方法 handleRequest()，在该方法中处理请求，并返回 ModelAndView 实例

9.2.2.2. HandlerMapping

DispatcherServlet根据它来决定请求由哪一个Controller处理

1) 默认情况下，DispatcherServlet 将使用 org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping，即使用和客户请求的 URL 名称一致的 Controller 的 bean 实例来处理请求，如：

```
<bean name="/loginController.do"
class="com.controller.LoginController">

</bean>
```

2) 另外一种常用的 HandlerMapping 为 org.springframework.web.servlet.handler.SimpleUrlHandlerMapping，配置如下：

```
<bean id="urlHanderMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping
">
    <property name="mappings">
        <props>
            <prop key="/login.do">loginController</prop>
        </props>
    </property>
</bean>
<!-- 按照URL作映射，这是常用的方式 -->
<bean id="loginController" class="com.controller.LoginController">

</bean>
```

在以上“mappings”属性设置中，“key”为请求的 URL，而“value”为处理请求的 Controller 的 bean 名称

9.2.2.3. ModelAndView

用来封装View与呈现在View中的Model对象；DispatcherServlet根据它来解析View名称，并处理呈现在View中的Model对象

9.2.2.4. ViewResolver

DispatcherServlet委托ViewResolver来解析View名称，常用的ViewResolver实例配置如下：

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/html/</value>
    </property>
    <property name="suffix">
        <value>.html</value>
    </property>
</bean>
```

9.2.2.5. MultiActionController

可以定义多个业务方法，例：

1) 相关 java 类：

```
public class BookController extends MultiActionController {
    private String loginPage;
    public String getLoginPage() {
        return loginPage;
    }
    public void setLoginPage(String loginPage) {
        this.loginPage = loginPage;
    }
    public ModelAndView add(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("BookController's add() is called...");
        return null;
    }
    public ModelAndView del(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("BookController's del() is called...");
        return null;
    }
    public ModelAndView list(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
```

```
        System.out.println("BookController's list() is called...");
        return null;
    }
    //拦截器方法，在调用本控制器每个业务方法之前，都会先调用该方法
    public ModelAndView handleRequestInternal(HttpServletRequest
request, HttpServletResponse response) throws Exception {
        System.out.println("BookController's
handleRequestInternal()" );
        String method = request.getParameter("method");
        if(method!=null && !method.equals("list")){
            return new ModelAndView(loginPage);
        }
        return super.handleRequestInternal(request, response);
    }
}
```

2) 相关配置信息

```
<!-- 一个控制器对应多个业务方法 -->
<bean id="paramMethodResolver"
    class="org.springframework.web.servlet.mvc.multiaction.ParameterMe
thodNameResolver">
        <property name="paramName">
            <value>method</value>
        </property>
        <property name="defaultMethodName">
            <value>list</value>
        </property>
    </bean>
    <bean id="bookController" class="com.controller.BookController">
        <property name="methodNameResolver">
            <ref bean="paramMethodResolver" />
        </property>
        <property name="loginPage">
            <value>/index.jsp</value>
        </property>
    </bean>
```

3) 页面调用：bookController.do

9.2.3. 拦截器

1. org.springframework.web.servlet.HandlerInterceptor，相关方法

1) boolean preHandle(HttpServletRequest request， HttpServletResponse response， Object handler)throws Exception

说明：会在 Controller 处理请求之前被执行，返回的 boolean 值决定是否让 Handler Interceptor 或者是 Controller 来处理请求

2) *void postHandle(HttpServletRequest request，HttpServletResponse response，Object handler，ModelAndView modelAndView) throws Exception*

说明：会在 Controller 处理完请求之后被执行

3) *void afterCompletion(HttpServletRequest request，HttpServletResponse response，Object handler，Exception ex) throws Exception*

说明：会在 View 绘制完成之后被执行

2. 拦截器例子

```
package com.interceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
public class LogInterceptor implements HandlerInterceptor {
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
        System.out.println("LogInterceptor's afterCompletion()...");
    }
    //handler:表示目标对象
    //modelAndView:表示转向的资源
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        System.out.println("LogInterceptor's postHandle()...");
    }
    //handler:表示目标对象
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception
{
        System.out.println("LogInterceptor's preHandle()...");
        return true;
    }
}
```

```
<bean id="logInterceptor" class="com.interceptor.LogInterceptor"></bean>
    <bean id="urlHanderMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
```

```
        <property name="interceptors">
            <list>
                <ref bean="logntercеptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop key="/login.do">loginController</prop>
                <prop key="/book.do">bookController</prop>
                <prop key="/hello.do">helloController</prop>
            </props>
        </property>
    </bean>
```