目录

1、Spring 框架介绍：

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

Spring 本质：工厂，工厂设计模式的体现，工厂模式主要作用解耦合。

Spring 特点：轻量级，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。

2、Spring 框架结构：

可以在 www.springframework.org 下载 download 到 Spring 的源码。

spring 框架的目录结构：

aspectj 对AOP支持
* dist spring.jar
    |- resources
        |- spring配置文件信息
* doc 文档参考手册
    jarcontent spring schema 支持
* lib 第三方jar
    mock 模拟测试目录
    samples 例子
    src 源代码
    test 测试目录
    tiger spring 对 5.0 新特性的支持

3、Spring 框架开发环境的搭建：

1 jar 导入项目：

1 spring.jar，可以在 dist 文件夹下找到。

2 第三方 jar，可以在 lib 文件夹下找到。

2 框架配置文件导入项目：

Spring 配置名字：可以随便命名配置文件；

建议配置文件命名：beans.xml、applicationContext.xml

Spring 配置文件的放置位置：可以随便放置，建议放置到 src 下。

3 Spring Core API：

核心 ApplicationContext 工厂创建对象，ApplicationContext 实例的创建：

非 web 环境，可以通过 ClassPathXmlApplicationContext 创建

ApplicationContext 实例。

在 web 环境，可以通过 WebApplicationContext 创建 ApplicationContext 实例。

4 Spring 框架的开发步骤

1 beans.xml 配置文件中，配置需要创建的对象；

2 创建 Spring 工厂，并且创建需要的对象。注意：在 spring 配置文件中，

<bean id="u" class="xxx.User"/>

Spring 会自动进行对象的创建(User u = new User();)

4、Spring 框架注入的方式：

Spring 框架在创建对象的同时可以在配置文件中，对于成员变量进行赋值（注入）。

注入:通过配置文件对于成员变量的赋值。

A. **set** 方式的赋值**(set** 注入，推荐使用**)**：

前提：必须为成员变量提供 set、get 方法。

```
<bean id="u" class="User">
    <property name="name"> --- setXXX方法
        1 jdk 数据类型的赋值
        2 自定义类型
    </property>
</bean>
```

1 jdk 数据类型的赋值

赋值内容是String或者8种基本类型数据，应用<value></value>把赋值内容包含起来。

赋值内容为set 、list集合：

```
<list></list>
<set></set>
```

赋值内容为数组：

```
<list>
    <value></value>
</list>
```

赋值内容为 map 集合：

Entry 是 Map 中的 key 和 value 为属性，封装在一起的对象。

```
<map>
    <entry>
        <key></key>
        xxx
    </entry>
</map>
```

赋值内容 Properties 格式文件：Properties 是特殊的 Map，key value 都是字符串，Properties 既是 Map<String,String>

```
<props>
```

```
            <prop key="">xxxx</prop>
        </props>
```

2 自定义类型：

```
        <bean id="uD" class="userDAO"/>
        <bean id="" class="userService">
            <propety name="userDAO">
                <ref local="uD"/>
            </property>
        </bean>
```

B. 通过构造方法对于成员变量赋值(构造注入)：

前提：必须提供构造方法（构造器）。

1 配置文件：

```
    <bean>
            <constructor-arg>
                xxx
            </constructor-arg>
        </bean>
```

2 构造方法可以重载：

a 构造方法的构造参数个数不同，Spring 根据<constructor-arg>，标签数量进行构造方法的选取。

b 构造方法的构造参数个数相同，通过<constructor-arg type=""/>参数类型的不同，进行选取。

C. 自动为成员变量赋值（自动注入，不推荐使用)：

1 配置文件：

```
        <bean id="userDAO" class="day1.ioc.set.JdbcUserDAOImpl"/>
            <bean autowire="byType" id="userService"
                                class="day1.ioc.set.UserServiceImpl">
        </bean>
```

2 自动注入方式：

autowire="byType"：spring 自动在配置文件中找到一个和 userSevice 成员变量 userDAO 类型相同的对象赋值进去。

autowire="byName"：spring 自动在配置文件中找到一个和 userSevice 成员变量 userDAO 同名对象赋值进去。

5、复杂对象的创建（FactoryBean）：

复杂对象的创建布置：

1 xxx(复杂对象) implements FactoryBean

实现 FactoryBean 的方法：

`public Object getObject();`

作用：用于书写创建复杂对象的代码；

返回值：创建的复杂对象；

`public Class getObjectType();`

作用：返回所创建的赋值对象的类型；

`public boolean isSingleton();`

作用:用于控制复杂对象生产次数；

返回:`true` 通过 `spring` 工厂操作几次都获得是同一个复杂对象（单例模式）；

返回:`false` 通过 `spring` 工厂每一次获得的都是新的复杂对象 。

实现接口规律：

1 了解所需要实现方法的作用；

2 绝大多数不是程序员调用；

3 方法参数不用关心那里来的，直接使用；

4 按照要求提供方法返回值。

2 配置文件： <bean id="" class=""/>

注意:如果配置的是 FactoryBean 接口的实现类，那么通过 id 的值，获得的是 FactoryBean 接口实现类，所创建的复杂对象。

3 通过 ApplicationContext.getBean("xxx")获得复杂对象的实例。

6、Spring 核心 IOC 介绍：

IoC（inverse of control）即反转的控制；

概念: 对成员变量赋值的控制权从代码中反转(转移)到配置文件中进行赋值；

好处: 降低代码的耦合性，利于维护。

DI（dependency injection）即依赖注入；

概念:需要那个对象等效于依赖这个对象，当依赖某个对象时，就可以把它作为成员变量， 附加：`<bean id="" class="" scope=" "/>`

`scope` 作用:控制简单对象生成次数；

singleton:每一次从工厂中获得的都是同一个对象 （默认）

prototype:每一次从工厂中获得的都是不同的对象

Spring 为什么控制对象的生产次数：减少内存侵占通过 spring 配置文件进行赋值。

7、Spring 工厂高级特性：

1 Spring 工厂创建对象的方式：

结论: Spring 会在工厂创建的同时，创建工厂生产的对象。

验证：<bean id="u" class="XXXXX.User"/>

ApplicationContext ctx = new ClassPathXmlApplication();

ctx.getBean("u");

2 Spring 工厂生产对象的生命周期：

什么时候创建:工厂创建，同时对象创建，提供初始化方法:

<bean id="" class="" init-method=""/>

提供销毁方法:<bean id="" class="" destroy-method=""/>

什么时候销毁:工厂关闭，同时对象销毁

3 Spring 配置文件的参数化**(PropertyHolder)**：

把 spring 配置文件中经常需要修改的字符串配置信息，从 spring 大的配置文件中，转移到一个小的配置文件。

a. 准备小的配置文件：xxx.properties 文件名 ：随便命名；放置位置 ：随便放置。

b. spring 提供了一个类，作用:把小的配置文件中的信息和大配置文件进行整合 spring 配置文件中配置使用：org.springframework.beans.factory.config. PropertyPlacehoderConfigurer

c. 把原有大配置文件中，经常变化字符串的位置替换成${} ${username}

作用：到小配置文件中找到以 usernam，为 key 的内容来充当现在的值。

4 自定义类型转换器 **(CustomEditor)**：

Spring自动把配置文件中书写的字符串，变量对应的类型，进行赋值：

类型转换器： int = Integer.parseInt("23");

自定义开发：

a. 引入jdk中的java.beans.xxx

开发类 implements PropertyEdito 接口，主要实现 setAsText();方法。

开发类 extends PropertyEditorSupport 类，重写 setAsText();方法。

b. 配置文件

1 声明类型转换器

2 注册类型转换器

具体配置文件，例子：

```
<bean id="dateConverter" class="day2.custom.DateConverter">
</bean>
<bean id="customEditor"
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
    <map>
        <entry>
            <key>
                <value>java.util.Date</value>
            </key>
                <ref local="dateConverter"/>
        </entry>
    </map>
    </property>
</bean>
```

8、Spring 核心 AOP 介绍：

AOP（Aspect Oriented Programming）切面_面向_编程：

面向切面编程，该思想主要是考虑改善程序的结构，是对OOP(面向对象编程)的一个补充，AOP关注的重点是切面，OOP关注的重点是类。

代理设计模式 **(Proxy)**：

为其他对象提供一种代理以控制对这个对象的访问（增加额外功能）。

创建代理类：

代理3要素：

1 核心业务类(核心的功能,dao)、目标类(原始类) 、目标方法、原始方法；

2 额外功能；

3 代理类和核心业务类实现相同的接口。

静态代理设计模式：

1 原始核心业务类(原始类,目标)；

2 额外功能；

3 原始类和代理类实现相同的接口，每一个核心业务类(目标类) 都创建一代理类。

动态代理设计模式：

使用Spring框架创建代理类：

1 创建原始类，并且在spring配置文件中进行配置；

2 创建额外的功能类(生成代理类，所需要的工具类)，实现Adivce接口（子接口），并在配置文件中配置：

A MethodBeforeAdvice额外功能运行在原始方法运行之前执行：
before(Method, Object[], Object)
Method 额外的功能所增加个的那个方法
Object[] 额外的功能所增加个的那个方法的参数
Object 额外的功能所增加个的那个原始对象

B AfterReturnningAdvice 额外功能运行在原始方法运行之后执行：
afterReturning(Object, Method, Object[], Object)
Object 额外功能所增加给的那个方法的返回值
Method 额外的功能所增加个的那个方法
Object[] 额外的功能所增加个的那个方法的参数
Object 额外的功能所增加个的那个原始对象

C MethodInterceptor额外功能运行在原始方法运行之前后执行：
Object invoke(MethodInvocation)
MethodInvocation 额外功能所增加给的那个方法
return Object 额外功能所增加给的那个方法的返回值

D ThrowsAdvice 额外功能运行在原始方法抛出异常的时候执行：
afterThrowing(Method, Object[], Object, Throwable)
Method 额外功能所增加给的那个方法
Object[] 额外的功能所增加个的那个方法的参数
Object 额外的功能所增加个的那个原始对象
Throwable 额外的功能所增加个的那个方法抛出的异常

3 利用 org.springframework.aop.framework.ProxyFactoryBean 自动创建代理对象： 主要配置文件：

```
<bean id="orderService" class="day3.dynamic.OrderServiceImpl"></bean>
<bean id="arroud" class="day3.dynamic.ArroundAdvice"/>
<bean id="orderServiceProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref local="orderService"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>arroud</value>
        </list>
    </property>
</bean>
```

附加： ProxyFactoryBean 建议者(配置对于那些方法添加额外功能，进行自动创

建代理对象)：

配置文件，例子：

```
<bean id="userService" class="day3.aspect.UserServiceImpl"/>
<bean id="before" class="day3.aspect.Before"/>
<bean id="nameMatch"
    class="org.springframework.aop.support.NameMatchMethodPointcutAdvis
or">
    <property name="mappedNames">
        <list>
            <value>login</value>
            <value>register</value>
        </list>
    </property>
    <property name="advice">
        <ref local="before"/>
    </property>
</bean>

<bean id="userServiceProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref local="userService"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>nameMatch</value>
        </list>
    </property>
</bean>
```

标签继承：

基本书写方式：

```
<bean id="p" class="" abstract="true">
    <xxxx>
</bean>
<bean id="userProxy" class="ProxyFactoryBean" parent="p">
</bean>
```

例子（书写一个父标签建议者，可以通过继承，实现动态生成代理对象）：

```
<bean id="parentConfig"
    class="org.springframework.aop.framework.ProxyFactoryBean"
abstract="true">
    <property name="interceptorNames">
        <list>
            <value>nameMatch</value>
```

```
            </list>
        </property>
    </bean>
    <bean id="userServiceProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean"
    parent="parentConfig">
        <property name="target">
            <ref local="userService"/>
        </property>
     </bean>
```

对多个原始对象的多个方法，实现动态自动的生成代理对象：

通过 org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator实现；

注意：通过原始对象的id值获取，动态自动的生成代理对象，可以通过*匹配多个原始对象（名字要有规律）。

例子：

```
    <bean id="auto"
        class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames">
            <list>
                <value>*Service</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>nameMatch</value>
            </list>
        </property>
    </bean>
```

9、Spring 2.0 提供的新 AOP 编程方式：

1 原始对象；

2 额外的功能切面(Aspect，生成代理类，所需要的工具类)：

需要在切面类前加@Aspect 标注；

同时要在切面类里的方法前加@Before、@AfterReturning、@Around、@AfterThrowing 标注，同时添加 execution 表达式。

3 创建代理<aop:asepctj-autoproxy/>

配置文件例子：

```
<bean id="userService" class="day4.aspect2.UserServiceImpl"></bean>
<bean id="myAspect" class="day4.aspect2.MyAspect"></bean>
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

execution(表达式)：

public void register (day2.sh.User)

修饰符　　返回值　　方法名字 (参数表)

方法级别　　可以省略前面的修饰词 public）：

* register(..)　　不关心返回值和参数列表；

void　　register(String,..)　　参数列表第一个是 String 类型，其他参数不关心；

public * register*(String,..) 不关心返回值，方法名后半部分不关心。

类级别：

public * day4.aspect2.UserServiceImpl.register(String,String)

不关心返回值，必须是 day4.aspect2.UserServiceImpl 类的 register 方法，同时参数是两个 String 类型。

public * day4.aspect2.UserServiceImpl.*(..)

必须是 day4.aspect2.UserServiceImpl 类的方法。

包级别：

public * day4.*.*(..) 必须是 day4 包下的类、方法（直接包含类）

public * day4..*.*(..) 必须是 day4 下及其子包的类与方法

全级别：

* *(..)　　匹配任意的类、方法

附加：

args()：通过方法的参数进行额外功能的加入 。

args(String,String) 等同于 execution(* *(String,String))

参数必须是两个 String 类型。

args(day2.sh.User)

参数必须是一个 day2.sh 包下的 User 类型。

within()：对类或者对包选取进行额外功能的加入 。

within(day4.aspect2.UserServiceImpl)

需是 day4.aspect2 包下的 UserServiceImpl 类。

within(day4.*);

需是 day4 的直接子类。

within(day4..*) ;

需是 day4 包下的所有包及其子包的所有类。

注意：

可以在@Before、@AfterRetuning、@AfterThrowing 标注的方法加 JoinPoint 参数；

JoinPoint接口表示目标类连接点对象。JoinPoint的主要方法如下：

java.lang.Object[] getArgs() ：获取连接点方法运行时的入参列表；

Signature getSignature() ：获取连接点的方法签名对象；

java.lang.Object getTarget() ：获取连接点所在的目标对象；

java.lang.Object getThis() ：获取代理对象本身；

10、Spring 对于 DAO 层的支持：

1 **Spring** 连接池（**Connection** 连接池）：
org.apache.dbcp.commons.BasicDataSource;

2 **Spring** 对于 **JDBC** 的 **DAO** 层支持，提供 **JdbcTemplate** 工具类：
org.springframework.jdbc.core.JdbcTemplate;
A JdbcTemplate 重要部分方法（详细方法，见 API）：

int update(String sql,Object[]args)
sql：sql语句；args：占位符所指的参数。
List query(String sql, Object[] args, RowMapper rowMapper)
RowMapper：对于查询语句封装成对象的工具类。
Object queryForObject(String sql,Object[] args,RowMapper rowMapper)
查询语句只返回一个对象的情况。

B 对于查询语句封装成对象，工具类xxx implements RowMapper：
Object mapRow(ResultSet, int)方法：
Object ：返回封装的对象；
ResultSet ：查询结果集。
Int ：查询结果集当前是第几条（从0开始）。
配置文件，例子：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value>1234</value>
    </property>
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/test</value>
    </property>
</bean>
<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
</bean>
<bean id="userDAO" class="day4.dataSource.UserDAOImpl">
```

```
            <property name="jdbcTemplate">
                <ref local="jdbcTemplate"/>
            </property>
        </bean>
```

3 Spring 对于 **Hibernate** 的 **DAO** 层支持，提供 **HibernateTemplate** 工具类：
org.springframework.orm.hibernate3.HibernateTemplate

A HibernateTemplate 封装了 Session（更强大的 Session）：

B 配置文件：
可以通过 org.apache.commons.dbcp.BasicDataSource、
org.springframework.orm.hibernate3.LocalSessionFactoryBean
不用编写 hibernate.cfg.xml 的配置文件。
配置文件，例子：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value>1234</value>
    </property>
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/jd1105db</value>
    </property>
</bean>

<bean id="sf" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- 连接相关参数 -->
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <!-- 自身属性相关参数 dialect show_sql format_sql-->
    <property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect"> org.hibernate.dialect.MySQL5Dialect</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.format_sql">true</prop>
    </props>
    </property>
```

```
        <!-- 映射文件注册 -->
        <property name="mappingResources">
        <list>
            <value>day2/sh/User.hbm.xml</value>
        </list>
        </property>
    </bean>

    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate">

        <property name="sessionFactory">
            <ref local="sf"/>
        </property>
    </bean>

    <bean id="userDAO" class="day5.hibernate.UserDAOImpl">
        <property name="template">
            <ref local="hibernateTemplate">
            </ref>
        </property>
    </bean>
```

附加：HibernateTemplate 封装Session ，提供更强大的功能：

a hibernateTemplate 获得Session时， 自动对Session进行线程绑定；

b hibernateTemplate 会为 dao 自动的加入事务；好处:测试方便；

c 简化hql 操作：

```
String hql=”from User as u where u.name = ?”;
Object args = {"suns"};
List l=hibernateTemplate.find(String hql,Object[] args);
```

d 通过HibernateCallback，使用原始的Hinernate的session执行HQL：（建议使用内部类的方式）

```
HibernateTemplate .execute(my);
public my implements HibernateCallback {
    public Object doInHibernate(Session session){
    session 为所欲为
    }
}
```

4 Spring对于**Hibernate**的事务支持：

a 什么是事务：事务是保证业务操作完整性的机制。

b 事务在多数那里控制： Service层。

c 事务怎么控制：

Jdbc通过Connection实现：

Connection.setAutoCommit(false);

Connection.commit();

Connection.rollback();

Hibernate通过session实现：

Transaction = session.beginTranasction();

tx.commit();

tx.rollback();

d Spring框架中怎么控制事务： AOP (代理)

在AOP 1.2中的实现：

1 创建原始对象；

2 添加额外功能:MethodInterceptor(4中建议者)

org.springframework.orm.hibernate3.HibernateTransactionManager

DataSourceTranasctionManager

3 加入切面：

org.springframework.tranaction.interceptor.TranasctionInterceptor

4 自动创建代理对象： BeanNameAutoProxyCreator

例子：

```
<!-- transaction begin -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value>1234</value>
    </property>
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/jd1105db</value>
    </property>
</bean>
<bean id="sf" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect"> org.hibernate.dialect.MySQL5Dialect</prop>
        <prop key="hibernate.show_sql">true</prop>
```

```
            <prop key="hibernate.format_sql">true</prop>
        </props>
        </property>
            <property name="mappingResources">
            <list><value>day2/sh/User.hbm.xml</value></list>
        </property>
</bean>
<bean id="userService" class="day5.hibernate.UserServiceImpl">
        <property name="userDAO">
            <ref local="userDAO"/>
        </property>
 </bean>
<bean id="hibernateTransactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref local="sf"/>
        </property>
</bean>
<bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager">
            <ref local="hibernateTransactionManager"/>
        </property>
        <property name="transactionAttributes">
        <props>
            <prop key="register"></prop>
        </props>
        </property>
</bean>
<bean id="auto"
        class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames">
            <list>
                <value>*Service</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
        </property>
</bean>
```