Spring 入门技术栈

Spring 是一个开源的轻量级框架，由 Rod Johnson 创建，从 2003 年年初正式启动 IoC（控制反转）和 AOP（面向切面编程）是其最具代表性的技术。Spring 框架主要由七大模块组成

参考资料：Spring 官网中的 reference：
https://docs.spring.io/spring/docs/5.0.3.BUILD-SNAPSHOT/spring-framework-reference/core.html#spring-core

1. Spring 入门

1.1 概念

1) 开源的轻量级框架
2) AOP 面向切面编程
   简单来讲就是拓展功能不是修改源代码实现
3) IoC 控制反转
   对于类中非静态的方法需要通过关键字 new 来创建对象然后才能调用，如果某个实体类的类名或者方法名字修改了，牵一发而动全身，业务层里面所有的类名和方法都需要修改，这种情况下实体类和业务层的耦合度太高了。后来人们为了解决这个问题，创造了工厂模式（静态方法 return 实体类实例）进行解耦。在工厂模式下，服务类和逻辑业务层虽然解耦了，但是又产生了新的耦合——逻辑业务层和工厂类的耦合，而在 IoC 技术下，可以直接交给 spring 进行配置；
4) 一站式框架
   Spring 在 javaee 的三层架构中，每一层都提供给不同额解决技术，如 web 层是 springMVC，service 层是 IoC，dao 层是 spring 的 jdbcTemplate；
5) 版本

1.2 IoC 底层原理

1) 如果原来不用 IoC 时用 new 方法创建新类，改变实体类名的时候，需要修改很多的东西，缺陷是耦合度太高了，后来人们采用工厂方法来解耦合，但是又会与工厂类耦合；
2) xml 配置 文件；
   <bean id=”user” class=”com.lvhongbin.bean”/>
3) dom4j 解决 xml；

4) 工厂设计模式；
5) 反射

```
//创建工厂，使用 dom4j 解析配置文件+反射
    //返回 UserService 对象的方法
    public static User getUser() {
        //使用 dom4j 解析配置文件
        //跟据 ID 值得到所对应的 class 属性值
        String classValue ="class 属性值";
        //使用反射创建类对象
        Class clazz=Class.forName(classValue);
        User user=clazz.newInstance();
        return user
}
```

1.3 IoC 入门案例

1) 导入 jar 包
   每个 jar 由三个 jar 组成，核心 jar,说明文档 jar 和源代码 jar
   核心 jar 包：Beans，Core，Context 和 expression
   日志：commons-logging-1.2，log4j
2) 创建类，在类中创建方法；
3) 创建 spring 配置文件，配置创建类；
   Spring 核心配置名称和位置不是固定的，建议放到 src 下面，官方建议放 applicationContext.xml
   然后引入 schema 约束，在 core.html 文件里面有例子，路径：spring\spring-framework-5.0.2.RELEASE-docs\spring-framework-reference\core.html：

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

    配置对象的创建，id 值可以随便取，但是最好取首字母小写的类名
    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>
```

```
    <!-- more bean definitions go here -->

</beans>
```

4) 写代码测试对象创建；

```
ApplicationContext context =new ClassPathXmlApplicationContext("applicationContext.xml");
Book book=(Book)context.getBean("book");
```

或者

```
Resource resource =new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
//得到配置文件中的对象
Book book=(Book)factory.getBean("book");
```

1.4 ApplicationContext 接口的应用

BeanFactory 实现了 IoC 控制，所以可以称为 IoC 容器，而 ApplicationContext 拓展了 BeanFactory 容器并添加了 I18N（国际化），生命周期时间的发布监听等更加强大的功能，使之成为 Spring 中强大的企业级 IoC 容器。ApplicationContext 接口的实现类主要有三个：

1) ClassPathXmlApplicationContext 类
   它从当前类路径中检索配置文件并装载它来创建容器的实例，参数 configLocation 指明了配置文件的路径和名称：
   ApplicationContext context =new ClassPathXmlApplicationContext(String configLocation);
2) FileSystemXmlApplicationContext 类
   跟 ClassPathXmlApplicationContext 类似，只是 FileSystemXmlApplicationContext 类可以获取当前类路径以外的资源：
   ApplicationContext context =new FileSystemXmlApplicationContext(String configLocation);
3) WebApplicationContext 类
   Spring 的应用容器，可以在 servlet 中使用

2 Spring 的 bean 管理（配置文件）

2.1 创建对象

1) Bean 实例化的三种方式实现
   BeanFactorygetBean()方法在调用前不会实例化任何对象，只有在创建 javaBean 实例对象的时候，才会分配资源空间，
   a) 使用类的无参构造创建，如果没有无参构造则会报错；

```
<bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

b) 使用静态工厂创建；

```
<bean id="testSpringBookFactory"
class="com.lvhongbin.factory.TestSpringBookFactory"
factory-method="bookFactory">
</bean>
```

c) 使用实例工厂创建；

```
<bean id="testSpringBookFactory2"
class="com.lvhongbin.factory.TestSpringBookFactory">
</bean>
<bean id="testSpringBookFactory3" factory-
bean="testSpringBookFactory2"
factory-method="bookFactory2"></bean>
```

2) Bean 标签的常用属性

a) id 属性

不能包含特殊符号

b) class 属性

创建类的全路径，用”.”表示

c) name 属性

旧版使用的，跟 id 功能相同，可以包含特殊符号。

d) scope 属性

singleton 默认值，单例；

prototype 多例的

request，web 项目中，spring 创建一个 Bean 对象，并将该对象存入 request 域中；

session，web 项目中，spring 创建一个 Bean 对象，并将该对象存入 session 域中；

globalSession，web 项目中，spring 创建一个 Bean 对象，应用于 Prolet 环境，如果没有 Prolet 环境那么 globalSession 并将该对象存入相当于 session 域；

2.2 属性注入

1) 有参构造，spring 框架支持；

```
<bean id="..." class="...">
    <!—使用有参构造注入 -->
    <constructor-arg name="price" value="100RMB "></constructor-arg>
</bean>
```

Or

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

2) Set 方法，spring 框架支持；

```
<bean id="book" class="com.lvhongbin.bean.Book">
        <property name="author" value="吕鸿斌"></property>
</bean>
```

3) 接口注入，spring 框架不支持；

4) 对象类型属性注入，使用 ref 属性

```
<bean id="user" class="com.lvhongbin.bean.User">
    <property name="name" value="lvhongbin"></property>
</bean>
<bean id="book" class="com.lvhongbin.bean.Book">
    <property name="user" ref="user" ></property>
</bean>
```

5) P 名称空间注入

配置文件约束中添加：xmlns:p=http://www.springframework.org/schema/p

```
<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.xsd">
```

然后直接在 bean 标签下使用 p:

```
<bean id="book" class="com.lvhongbin.bean.Book" p:pbookCount="80%">
```

可惜的是并没有成功，不知道什么原因，也没有报错—< —

6) 匿名内部类的注入

直接在 bean 标签下再建一个 bean 标签即可，后者可不必写 name 属性或者 id 属性；

7) 复杂类型的注入

数组

```
<!-- results in a setSomeArrays(java.util.Arrays) call -->
```

```
            <property name="arr">
                <list>
                    <value>吕</value>
                    <value>鸿</value>
                    <value>斌</value>
                </list>
        </property>
```
List 集合
```
        <!-- results in a setSomeList(java.util.List) call -->
            <property name="list">
                <list>
                    <value>a list element followed by a reference</value>
                    <ref bean="myDataSource" />
                </list>
        </property>
```
Map 集合
```
        <!-- results in a setSomeMap(java.util.Map) call -->
            <property name="map">
                <map>
                    <entry key="an entry" value="just some string"/>
                    <entry key ="a ref" value-ref="myDataSource"/>
                </map>
        </property>
```
Set 集合
```
        <!-- results in a setSomeSet(java.util.Set) call -->
            <property name="set">
                <set>
                    <value>just some string</value>
                    <ref bean="myDataSource" />
                </set>
        </property>
```
Properties 类型
```
        <!-- results in a setAdminEmails(java.util.Properties) call -->
            <property name="property">
                <props>
                    <prop
key="administrator">administrator@example.org</prop>
                    <prop key="support">support@example.org</prop>
                    <prop
key="development">development@example.org</prop>
                </props>
        </property>
```

2.3 IoC 和 DI 的区别

IoC：控制反转，inversion of Control，把对象的创建交给 spring 进行配置；
DI：依赖注入，Dependency Injection，向类中的属性设置值
关系，DI 不能单独存在，需要在 IoC 中设置

2.4 Spring 整合 web 项目原理

加载 spring 核心配置文件

1) New 对象，功能可以实现，效率很低；
2) 实现思想：把加载配置文件和创建对象的过程在服务器启动的时候完成；
3) 实现原理
   ServletContext 对象；
   监听器；
   具体使用；
   在服务器启动后，当监听到 ServletContext 对象建立时，加载 spring 配置文件，把配置文件配置对象创建，把创建出来的对象放到 ServletContext 域中（setAttribute 方法），获取对象的时候，到 ServletContext 域得到（getAttribute 方法）

3 Spring 的 bean 管理（注解）

3.1 相关说明

1) 代码里面特殊标记，使用注解可以完成功能；
2) 注解写法@注解名称（属性名称=属性值）；
3) 注解使用在类上面，方法上面和属性上面；
4) 只有在创建对象的时候进行对象创建和属性注入

3.2 准备工作

1) 导入 jar 包：核心 jar 包+AOP jar 包；
2) 创建配置文件，引入约束；

```
xmlns: context="http://www.springframework.org/schema/context"
    xsi: schemaLocation ="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
```

3) 开启注解扫描

```
<!--        扫描类、方法和属性上的注释        -->
<context:component-scan base-package="com.lvhongbin"></context:component-scan>

<!--        扫描属性上的注释        -->
    <context:annotation-config></context:annotation-config>
```

3.3 创建对象的四个重要注解

“Spring provides further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated above, @Repository is already supported as a marker for automatic exception translation in your persistence layer.@Component”

1) @Controller Web 层
2) @Service 业务层
3) @Repository 持久层

其实就相当于省去了在配置文件中写<bean>标签

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. However, sometimes you need a different scope which can be specified via the @Scope annotation. Simply provide the name of the scope within the annotation:

@Scope("prototype")

3.4 属性注入的注解

@Autowired 自动装注

其实就相当于省去在配置文件中写<property>标签 和在实体类中省去写 setter 方法。

首先在 Book 中添加@Component(value="book1")

然后在需要自动注入属性的类 Book2 中，并省去 setter 的方法

```
@Autowired
private Book book1;
```

最后在需要用到类 Book2 的场合可以直接得到 Book

@Resource(name="book1")自动装注

跟@Autowired 一样，但是需要用 name 属性，指明用的是哪一个对象。

```
@Resource(name="book1")
private Book book2;
```

3.5 配置文件和注解的混合使用

使用配置文件创建对象

使用注解进行属性注入

3.6 Action 中@Resource 为 null 的情况

struts2 中 Action 如果需要使用 spring 的注解注入，则需要将 Action 交给 spring 管理。

1) 需要引入 struts2-spring-plugin-2.3.15.1.jar（不同 struts 版本对应不同的包），jar 包中有 xml 配置文件。其中设置了 ObjectFactory 为 spring 这个 struts 会自动加载不用设置
2) 需要在 Action 类中增加注释

```
@Component("TestAction")
@Scope("prototype")
public class TestActionextends ActionSupport {}
```

告诉 spring 这个 action 需要管理为 bean，然后这个类就可以使用 spring 的@Resource 来注入 serviceBean 实例

3) 修改 struts.xml

<action name="test" class="com.action.TestAction">

改为

<action name="test" class="TestAction"> class 和上面@Component("TestAction") 定义的 bean id 对应

此时应该不会出现在 Action 中@Resource service 为 null 的情况了。

4 AOP 概述

4.1 AOP 概念

AOP: Aspect Oriented Programing 面向切面编程，拓展功能不通过修改代码实现;

AOP 采用横向抽取机制，取代了传统的纵向继承体系的重复性代码；

4.2 AOP 原理

使用动态代理的方法

1) 第一种，有接口的情况
   创建跟 DaoIml 平级的代理对象，实现和 DaoIml 相同的功能：

```
public interface Dao{
    Public void add();
}
public class DaoIml implements Dao{
        Public void add(){
            //添加逻辑
        }
}
```

2) 第二种，没有接口的情况，也叫 cglib 动态代理
   创建 User 类的子类的代理对象；
   在子类中调用父类的方法完成增强；

4.3 AOP 操作术语

1) Joinpoint 连接点
   类里面哪些方法可以被增强的方法；
2) Pointcut 切入点
   在类里面有很多方法可以被增强，但只有在实际操作中被增强的方法才叫切入点；
3) Advice 通知增强
   实际需要增强的逻辑，如前置通知，后置通知，异常通知，最终通知，环绕通知（在方法之前和之后执行）；
4) Introduction 引介
   动态地向类中增加属性和方法；
5) Target 目标对象
   代理的目标对象（要增强的类）；
6) Aspect 切面
   把增强应用到具体方法上的过程称为切面；
7) Weaving 织入
   把增强应用到目标的过程，即把 advice 应用到 target 的过程；
8) Proxy 代理
   一个类被 AOP 织入增强后产生的一个结果代理类；

5 Spring 的 AOP 操作

5.1 AspectJ 简介

1) AspectJ 是一个面向切面的框架；
2) AspectJ 不是 Spring 的一部分，只是 Spring2.0 后新增了对 AspectJ 切点表达式的支持；
3) 实现 AOP 的两种方式：
   a) 基于 AspectJ 的 xml 配置
   b) 基于 AspectJ 的注解配置

5.2 AOP 的操作准备

1) 除了导入核心的 jar 包，还需要导入 AOP 和 aspectJ 相关的 jar 包——aspectjweaver.jar
   The @AspectJ support can be enabled with XML or Java style configuration. In either case you will also need to ensure that AspectJ's aspectjweaver.jar library is on the classpath of your application (version 1.6.8 or later). This library is available in the 'lib' directory of an AspectJ distribution or via the Maven Central repository.
2) 引入约束

```
<?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:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
           <!-- bean definitions here -->
    </beans>
```

5.3 使用表达式配置切入点

1) 切入点即实际增强的方法；
2) 常用的表达式
   execution(<访问修饰符>？<返回类型><方法名>（<参数>）<异常>)
   execution(*com.lvhongbin.bean.Book. testSpring5())
   execution(*com.lvhongbin.bean.Book. *)

execution(**. *())

execution(**. save*()) 匹配所有 save 开头的方法

3) 方法

配置 bean 对象

配置 aop 操作

```
<!-- AOP 切面 -->
<bean id="book3" class="com.lvhongbin.bean.Book"></bean>
<bean id="book4" class="com.lvhongbin.bean.Book2"></bean>
<aop:config>
    <!-- 2.1 配置 切入点 -->
    <aop:pointcut expression="execution(* cn.lvhongbin.bean.Book.testSpring5())" id="pointcut1"/>
    <!-- 2.1 配置 切入面 -->
    <aop:aspect ref="book4">
        <aop:before method="before" pointcut-ref="pointcut1"/>
    </aop:aspect>
</aop:config>
```

5.4 异常处理

异常如下：

Exception in thread "main" org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named 'hello' must be of type [com.hyq.chapter08_04_3.HelloImpl], but was actually of type [com.sun.proxy.$Proxy13]

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:376)

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:200)

at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:979)

at com.hyq.chapter08_04_3.AspectTest.main(AspectTest.java:10)

解决办法：https://www.cnblogs.com/zest/p/5878020.html

我在网上搜了一大堆解决方法都不管用，后来发现只要在 spring 的配置文件 bean.xml 中的 <aop:aspectj-autoproxy/> 加上一个属性 proxy-target-class="true"，即为：

<aop:aspectj-autoproxy proxy-target-class="true"/>

网上查了一下 proxy-target-class="true"的相关作用如下：

proxy-target-class 属性值决定是基于接口的还是基于类的代理被创建。

如果 proxy-target-class 属性值被设置为 true，那么基于类的代理将起作用（这时需要 cglib 库）。

如果 proxy-target-class 属值被设置为 false 或者这个属性被省略，那么标准的 JDK 基于接口的代理将起作用。

反思：

我怀疑是开始的时候没有创建接口，所以被迫使用 cglib 代理

6 Log4j

6.1 Log4j 介绍

1) 学习地址：http://bijian1013.iteye.com/blog/2307334；
2) 通过 Log4j 查看运行过程中的日志信息
3) 需要先导入 Log4j 的 jar 包，包括 log4j-1.2-api-2.10.0，log4j-api-2.10.0 和 log4j-core-2.10.0，下载地址：https://logging.apache.org/log4j/2.x/download.html；
4) 注意到，输出的 log 都是在 ERROR level 上的，log4j 定义了 8 个级别的 log（除去 OFF 和 ALL，可以说分为 6 个级别），优先级从高到低依次为：OFF、FATAL、ERROR、WARN、INFO、DEBUG、TRACE、 ALL。如果将 log level 设置在某一个级别上，那么比此级别优先级高的 log 都能打印出来。例如，如果设置优先级为 WARN，那么 OFF、FATAL、ERROR、WARN 4 个级别的 log 能正常输出，而 INFO、DEBUG、TRACE、 ALL 级别的 log 则会被忽略。从我们实验的结果可以看出，log4j 默认的优先级为 ERROR 或者 WARN（实际上是 ERROR）。

   另外，有 ERROR StatusLogger No log4j2 configuration file found 错误，是因为还没有配置配置文件，下面就来看看配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="error">
  <appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <ThresholdFilter level="trace" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"/>
    </Console>
    <File name="log" fileName="target/test.log" append="false">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"/>
```

```
            </File>
            <RollingFile name="RollingFile" fileName="logs/app.log"
                         filePattern="logs/$${date:yyyy-
MM}/app-%d{MM-dd-yyyy}-%i.log.gz">
              <PatternLayout pattern="%d{yyyy.MM.dd 'at' HH:mm:ss z} %-
5level %class{36} %L %M - %msg%xEx%n"/>
              <SizeBasedTriggeringPolicy size="500 MB" />
            </RollingFile>
          </appenders>
          <loggers>
            <root level="trace">
              <appender-ref ref="RollingFile"/>
              <appender-ref ref="Console"/>
            </root>
          </loggers>
</configuration>
```

7 Spring 整合 web 项目演示

7.1 演示问题

1) 先配置 struts2（在 web.xml 中添加过滤器）；
2) 创建 action；
3) 在 action 中调用 service；
4) service 调用 dao；

问题：每次访问都加载 spring 配置文件！

解决方案：

1) 在服务器启动的时候，创建对象加载配置文件
2) 底层使用监听器，servletContext 对象；

在 spring 中不需要我们自己写代码实现，框架帮我们实现；

做法：

1) 导入 spring 整合 web 项目的 jar 包，名称是 spring-web-5.0.2.RELEASE 的 jar 包；
2) 然后在 web.xml 文件中，配置接听器：

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

3) 然后配置文件的位置，如果没有配置文件的话，会出现异常：Caused by:

java.io.FileNotFoundException: Could not open ServletContext resource [/WEB-INF/applicationContext.xml]，说明默认的文件位置和名称为[/WEB-INF/applicationContext.xml]，但是也可以指定文件的路径：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

7.2 基于 aspectj 的注解 AOP 操作

1) 添加 jar 包，；
2) 添加约束；
3) 开启自动扫描；
4) 创建对象
5) 在 spring 核心配置文件中，开启 aop 操作：

```
<!—开启 aop 操作 –>
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

6) 在增强类上使用注解完成 aop 操作；
   在增强类名前添加@Aspect
   然后在类中具体的增强方法前添加不同的增强注释

```
@Before(value="execution(* com.lvhongbin.bean.Book.testSpring6())")
public void before1() {
    System.out.println("注解方式使用 AspectJ 前置增强的方法");
}
@AfterReturning(value="execution(*
com.lvhongbin.bean.Book.testSpring6())")
public void after1() {
    System.out.println("注解方式使用 AspectJ 后置增强的方法");
}
@Around(value="execution(* com.lvhongbin.bean.Book.testSpring6())")
public void around1(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
    System.out.println("注解方式方法之前.....");
    proceedingJoinPoint.proceed();
    System.out.println("注解方式方法之后.....");
}
```

7.3 Spring 的 jdbcTemperate 操作

用在 DAO 层
Spring 对不同的持久化层都进行了封装。

1) 增加
   a) 添加 spring-jdbc-5.0.2.RELEASE.jar 包和 spring-tx-5.0.2.RELEASE.jar 包和数据库 mysql-connector-java-5.1.44-bin.jar 包；
   b) 创建对象，设置数据库信息（即设置连接信息）；
   c) 创建 jdbcTemplate 对象，设置数据源；
   d) 调用 jdbcTemplate 对象里面的方法实现操作；
   其中 jdbcTemplate.update(String sql, String args, …)，包含有两种参数，前者是 sql 语句，后者是 sql 语句中问号的值；

```
public String testJDBCTemperate1() {
        //设置数据库信息
        DriverManagerDataSource dataSource =new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");

    dataSource.setUrl("jdbc:mysql://localhost:3306/jsp_db?useSSL=true");
        dataSource.setUsername("root");
        dataSource.setPassword("12345687");
        //创建 jdbcTemplate 对象，设置数据源
        JdbcTemplate jdbcTemplate =new JdbcTemplate(dataSource);
        //添加 sql 语句，调用 jdbcTemplate 里面的方法
        String sql="insert tbl_book(bookname,author,price,publicyear) values (?,?,?,?);";
        int i=jdbcTemplate.update(sql, this.name9,this.name10,this.name11,this.name12);
        System.out.println(i);
        return SUCCESS;
    }
```

2) 修改

```
public String testJDBCTemperate2() {
        try {
            JdbcTemplate jdbcTemplate =new JdbcTemplate(DriverManagerDataSourceFactory.getInstance());
           String sql="update tbl_book set price=? where bookname=?";
            int i=jdbcTemplate.update(sql, this.name11,this.name9);
            System.out.println("修改成功 "+i);
            this.feedbackMessage="";
            this.feedbackMessage="修改成功 ";
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("修改失败 ");
```

```
                this.feedbackMessage="";
                this.feedbackMessage="修改失败 ";
            }finally {

            }
            return SUCCESS;
        }
```

3) 删除

```
        public String testJDBCTemperate3() {
            try {
                JdbcTemplate jdbcTemplate =new
    JdbcTemplate(DriverManagerDataSourceFactory.getInstance());
                String sql="delete from  tbl_book where author=?;";
                int i=jdbcTemplate.update(sql, this.name9);
                System.out.println("删除成功 "+i);
                this.feedbackMessage="";
                this.feedbackMessage="删除成功 ";
            } catch (Exception e) {
                e.printStackTrace();
                System.out.println("删除失败 ");
                this.feedbackMessage="";
                this.feedbackMessage="删除失败 ";
            }finally {

            }
            return SUCCESS;
        }
```

4) 查询

JdbcTemplate 实现查询，有接口 RowMapper，但是没有提供实现类，得到不同的类型数据需要自己进行数据封装；

a) 查询返回某一个值；

jdbcTemplate.queryForObject(String sql, Class<T> requiredType, Object obj…);//后者表示返回值基本数据类型的包装类.class。

b) 查询返回某一个对象，但是需要注意的是如果检索的记录数量大于 1 的话使用 queryForObject 方法会报错；

jdbcTemplate.queryForObject(sql, RowMapper <T>rowMapper, Object obj…);

rowMapper 是一个接口；

obj…是一个可变参数；

```
        sql="SELECT * FROM tbl_book WHERE bookname = ?;";
            Book book=jdbcTemplate.queryForObject(sql, new
RowMapper<Book>() {
```

```
                @Override
                public Book mapRow(ResultSet arg0, int arg1) throws
    SQLException {
                    Book book=new Book();
                    String bookname=arg0.getString("bookname");
                    String price=arg0.getString("price");
                    String author=arg0.getString("author");
                    String publicyear=arg0.getNString("publicyear");
                    book.setBookname(bookname);
                    book.setPrice(Double.parseDouble(price));
                    book.setAuthor(author);
                    book.setPublicyear(publicyear);
                    return book;
                }
            },this.name9);
```

c) 查询返回某一个集合；

```
        sql="SELECT * FROM tbl_book WHERE bookname = ?;";
        List<Book> list=jdbcTemplate.query(sql, new RowMapper<Book>() {
                @Override
                public Book mapRow(ResultSet arg0, int arg1) throws
        SQLException {
                    Book book=new Book();
                    String bookname=arg0.getString("bookname");
                    String price=arg0.getString("price");
                    String author=arg0.getString("author");
                    String publicyear=arg0.getNString("publicyear");
                    book.setBookname(bookname);
                    book.setPrice(Double.parseDouble(price));
                    book.setAuthor(author);
                    book.setPublicyear(publicyear);
                    return book;
                }
            },this.name9);
```

7.4 Spring 配置连接池

1) 介绍（来自：http://blog.csdn.net/qq441568267/article/details/52951767）

1.JDBC 数据库连接池的必要性

在使用开发基于数据库的 web 程序时，传统的模式基本是按以下步骤：

在主程序（如 servlet、beans）中建立数据库连接。

进行 sql 操作

断开数据库连接。

这种模式开发，存在的问题:

普通的 JDBC 数据库连接使用 DriverManager 来获取，每次向数据库建立连接的时候都要将 Connection 加载到内存中，再验证用户名和密码(得花费 0.05s～1s 的时间)。需要数据库连接的时候，就向数据库要求一个，执行完成后再断开连接。这样的方式将会消耗大量的资源和时间。数据库的连接资源并没有得到很好的重复利用.若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。

对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将导致重启数据库。

这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃.

2.数据库连接池（connection pool）

为解决传统开发中的数据库连接问题，可以采用数据库连接池技术。

数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。

数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个。

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数来设定的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。

3.数据库连接池技术的优点

(1)资源重用：由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。

(2)更快的系统反应速度:数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间

(3)新的资源分配手段对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置实现某一应用最大可用数据库连接数的限制避免某一应用独占所有的数据库资源.

(4)统一的连接管理，避免数据库连接泄露在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

4.c3p0 数据库连接池

据说 c3p0 数据库连接池是最优秀的，下面我就来介绍如何使用 c3p0 连接池。

(1)下载 jar 包

(2)配置文件

作为一个数据库连接池自然有很多参数要设置，当然就算你不设置也有默认的，不过那不一定能满足你的要求。这里的配置文件没有什么特别的要求，可以是 xml 也可以是 properties 甚至与 txt 都行，当然如果你愿意也可以写死在程序中。

下面是我的配置文件，我把它放在了 src 目录下，名字叫“c3p0.properties”

[plain] view plain copy

```
#jdbc 基本信息
driverClass=oracle.jdbc.driver.OracleDriver
jdbcUrl=jdbc:oracle:thin:@192.168.61.250:1521:SXPPMS
user=qzp
password=111111

#c3p0 连接池信息
c3p0.minPoolSize=3
c3p0.maxPoolSize=25

#当连接池中的连接耗尽的时候 c3p0 一次同时获取的连接数
c3p0.acquireIncrement=3
#定义在从数据库获取新连接失败后重复尝试的次数
c3p0.acquireRetryAttempts=60
#两次连接中间隔时间，单位毫秒
c3p0.acquireRetryDelay=1000
#连接关闭时默认将所有未提交的操作回滚
c3p0.autoCommitOnClose=false
#当连接池用完时客户端调用 getConnection()后等待获取新连接的时间，超时后将抛出 SQLException,如设为 0 则无限期等待。单位毫秒
c3p0.checkoutTimeout=3000
#每 120 秒检查所有连接池中的空闲连接。Default: 0
c3p0.idleConnectionTestPeriod=120
#最大空闲时间,60 秒内未使用则连接被丢弃。若为 0 则永不丢弃。Default: 0
c3p0.maxIdleTime=600
#如果设为 true 那么在取得连接的同时将校验连接的有效性。Default: false
c3p0.testConnectionOnCheckin=true
#c3p0 将建一张名为 c3p0TestTable 的空表，并使用其自带的查询语句进行测试。
jdbc.automaticTestTable = c3p0TestTable
```

2) Spring 配置 c3p0 连接池

导入 c3p0-0.9.5.2.jar 和 mchange-commons-java-0.2.11.jar 包

```
public class ComboPooledDataSourceFactory {
```

```
        private static final ComboPooledDataSource comboPooledDataSource=new ComboPooledDataSource();
        private ComboPooledDataSourceFactory() {
            // TODO Auto-generated constructor stub
        }
        static {
            try {

    comboPooledDataSource.setDriverClass("com.mysql.jdbc.Driver");
            } catch (PropertyVetoException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

    comboPooledDataSource.setJdbcUrl("jdbc:mysql://localhost:3306/jsp_db?useSSL=true");
            comboPooledDataSource.setUser("root");
            comboPooledDataSource.setPassword("12345687");
        };

        public static ComboPooledDataSource getInstance() {
            return comboPooledDataSource;
        }

}
```

或者直接在 spring 配置文件中配置：

```
        <!-- 配置 c3p0 连接池 -->
        <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
            <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
            <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/jsp_db?useSSL=true"></property>
            <property name="User" value="root"></property>
            <property name="password" value="12345687"></property>
        </bean>
```

3) spring 配置文件中配置
   a) 创建 service 和 dao，配置 service 和 dao 对象，在 service 注入 dao 对象；
   b) 创建 jdbcTemplate 对象，把模板对象注入到 Dao 里面；
   c) 把 dataSource 对象注入到 jdbcTemplate 对象里面；

```
        <!-- spring 事务管理 -->
```

```
        <bean id="springUserService" class="com.lvhongbin.service.SpringUserService">
            <property name="springUserDao" ref="springUserDao"></property>
        </bean>
        <bean id="springUserDao" class="com.lvhongbin.dao.SpringUserDao">
            <property name="jdbcTemplate" ref="jdbcTemplate"></property>

        </bean>
        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
            <property name="dataSource" ref="dataSource"></property>
        </bean>
        <!-- 配置 c3p0 连接池 -->
        <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
            <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
            <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/jsp_db?useSSL=true"></property>
            <property name="User" value="root"></property>
            <property name="password" value="12345687"></property>
    </bean>
```

4) JdbcTemplate

```
        public class jdbcTemplate {

            private ComboPooledDataSource comboPooledDataSource;

            /**
             * @return the comboPooledDataSource
             */
            public ComboPooledDataSource getComboPooledDataSource() {
                return comboPooledDataSource;
            }

            /**
             * @param comboPooledDataSource the comboPooledDataSource to set
             */
            public void setComboPooledDataSource(ComboPooledDataSource comboPooledDataSource) {
                this.comboPooledDataSource = comboPooledDataSource;
            }
```

```
    }
```

5) Dao 层

```
public class SpringUserDao {

    private JdbcTemplate jdbcTemplate;

    /**
     * @return the jdbcTemplate
     */
    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }

    /**
     * @param jdbcTemplate the jdbcTemplate to set
     */
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public String add(Book book) {
        String result="";
        try {
            Double price=book.getPrice();
            String price1=price.toString();
            String sql="insert tbl_book(bookname,author,price,publicyear) values (?,?,?,?);";
            int i=jdbcTemplate.update(sql, book.getBookname(),book.getAuthor(),price1,book.getPublicyear());
            System.out.println("SpringUserDao 添加成功 "+i);
            result="SpringUserDao 添加成功 ";
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("SpringUserDao 添加失败 ");
            result="SpringUserDao 添加失败 ";
        }finally {
        }
        return result;
    }

}
```

6) Service 层

```
public class SpringUserService {

    private SpringUserDao springUserDao;

    /**
     * @return the springUserDao
     */
    public SpringUserDao getSpringUserDao() {
        return springUserDao;
    }

    /**
     * @param springUserDao the springUserDao to set
     */
    public void setSpringUserDao(SpringUserDao springUserDao) {
        this.springUserDao = springUserDao;
    }

    public String add(Book book) {
        return springUserDao.add(book);
    }
}
```

7.5 Spring 事务管理

1) 事务的概念
   a) 事务特性
   b) 不考虑隔离性产生读的问题
   c) 解决读问题
      设置隔离级别
2) Spring 事物管理 api
   编程式事务管理（一般不用）;
   声明式事务管理
   a) 基于 XML 配置文件实现
      首先配置事务管理器 bean;

```
<!-- 事务管理器 -->
<bean id="TransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager
">
    <property name="dataSource" ref="dataSource"></property>
```

```
    </bean>
```
然后利用 AspectJ 配置事务增强；
```
        <!-- 配置事务的增强 -->
        <tx:advice id="txadvice" transaction-manager="transactionManager">
            <!-- 做事务的操作 -->
            <tx:attributes>
                <!-- 设置进行事务操作的方法匹配规则 -->
                <tx:method name="add" propagation="REQUIRED"/>
            </tx:attributes>
    </tx:advice>
```
接着配置 AOP 切面产生代理；
```
        <!-- 配置切面 -->
        <aop:config>
            <aop:pointcut expression="execution(* com.lvhongbin.service.SpringUserService.add(...))" id="pointcut2"/>
            <aop:advisor advice-ref="txadvice" pointcut-ref="pointcut2"/>
    </aop:config>
```

b) 基于注解实现

配置事务管理器，同上；

配置事务注解，同上；

要在使用事务的方法所在的类上面添加注解，在业务层上添加@Transactional

3) Spring 事物管理 api 介绍
   a) PlatformTransactionManager 事务管理器

      对于不同的 dao 层框架，spring 提供接口不同的实现类
   b) TransactionDefinition 事务定义信息（隔离定义信息，传播，超时，只读）
   c) TransactionStatus 事务具体运行状态
4) 搭建转账环境，引入约束，主要是 tx 的约束

```
<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: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.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
```

http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd ">
创建数据库表，添加数据
创建 service 和 dao 类，完成注入关系。Service 层又叫做业务逻辑层，dao 层，单纯对数据库操作层，在 dao 层不添加业务；