作者博客：http://jinnianshilongnian.iteye.com/ 欢迎访问

第十二章　零配置

12.1　概述

12.1.1　什么是零配置

在 SSH 集成一章中大家注意到项目结构和包结构是不是很有规律，类库放到 WEB-INF/lib 文件夹下，jsp 文件放到 WEB-INF/jsp 文件夹下，web.xml 需要放到 WEB-INF 文件夹下等等，为什么要这么放呢？不这样放可以吗？

所谓零配置，并不是说一点配置都没有了，而是配置很少而已。通过约定来减少需要配置的数量，提高开发效率。

因此 SSH 集成时的项目结构和包结构完全是任意的，可以通过配置方式来指定位置，因此如 web.xml 完全可以不放在 WEB-INF 下边而通过如 tomcat 配置文件中新指定 web.xml 位置。

还有在 SSH 集成中还记得使用在 Struts2 配置文件中使用模式匹配通配符来定义 action，只要我们的 URL 模式将类似于/{module}/{action}/{method}.action 即可自动映射到相应的 Action 类的方法上，但如果你的 URL 不对肯定是映射不到的，这就是规约。

零配置并不是没有配置，而是通过约定来减少配置。那如何实现零配置呢？

12.1.2 零配置的实现方式

零配置实现主要有以下两种方式：

- **惯例优先原则：**也称为约定大于配置或规约大于配置（convention over configuration），即通过约定代码结构或命名规范来减少配置数量，同样不会减少配置文件；即通过约定好默认规范来提高开发效率；如 Struts2 配置文件使用模式匹配通配符来定义 action 就是惯例优先原则。
- **基于注解的规约配置：**通过在指定类上指定注解，通过注解约定其含义来减少配置数量，从而提高开发效率；如事务注解@Transaction 是不是基于注解的规约，只有在指定的类或方法上使用该注解就表示其需要事务。

对惯例优先原则支持的有项目管理工具 Maven，它约定了一套非常好的项目结构和一套合理的默认值来简化日常开发，作者比较喜欢使用 Maven 构建和管理项目；另

外还有 Strtus2 的 convention-plugin 也提供了零配置支持等等。

大家还记得【7.5 集成 Spring JDBC 及最佳实践】时的 80/20 法则吗？零配置是不是同样很好的体现了这个法则，在日常开发中同样 80%时间使用默认配置，而 20%时间可能需要特定配置。

12.1.3 Spring3 的零配置

Spring3 中零配置的支持主要体现在 Spring Web MVC 框架的惯例优先原则和基于注解配置。

Spring Web MVC 框架的惯例优先原则采用默认的命名规范来减少配置，具体详见【】。

Spring 基于注解的配置采用约定注解含义来减少配置，包括注解实现 Bean 配置、注解实现 Bean 定义和 Java 类替换配置文件三部分：

- **注解实现 Bean 依赖注入：**通过注解方式替代基于 XML 配置中的依赖注入，如使用@Autowired 注解来完成依赖注入。
- **注解实现 Bean 定义：**通过注解方式进行 Bean 配置元数据定义，从而完全将 Bean 配置元数据从配置文件中移除。
- Java 类替换配置文件：使用 Java 类来定义所有的 Spring 配置，完全消除 XML 配置文件。

12.2 注解实现 Bean 依赖注入

12.2.1 概述

注解实现 Bean 配置主要用来进行如依赖注入、生命周期回调方法定义等，不能消除 XML 文件中的 Bean 元数据定义，**且基于 XML 配置中的依赖注入的数据将覆盖基于注解配置中的依赖注入的数据**。

Spring3 的基于注解实现 Bean 依赖注入支持如下三种注解：

- **Spring 自带依赖注入注解：** Spring 自带的一套依赖注入注解；
- **JSR-250 注解：**Java 平台的公共注解，是 Java EE 5 规范之一，在 JDK6 中默认包含这些注解，从 Spring2.5 开始支持。
- **JSR-330 注解：**Java 依赖注入标准，Java EE 6 规范之一，可能在加入到未来 JDK 版本，从 Spring3 开始支持；
- **JPA 注解：**用于注入持久化上下文和尸体管理器。

这三种类型的注解在 Spring3 中都支持，类似于注解事务支持，想要使用这些注解需要在 Spring 容器中开启注解驱动支持，即使用如下配置方式开启：

```
<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"
    xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
</beans>
```

这样就能使用注解驱动依赖注入了，该配置文件位于“resources/chapter12/dependecyInjectWithAnnotation.xml”。

12.2.2 Spring 自带依赖注入注解

一、@Required：依赖检查；

对应于基于 XML 配置中的依赖检查，但 XML 配置的依赖检查将检查所有 setter 方法，详见【3.3.4 依赖检查】；

基于@Required 的依赖检查表示注解的 setter 方法必须，即必须通过在 XML 配置中配置 setter 注入，如果没有配置在容器启动时会抛出异常从而保证在运行时不会遇到空指针异常，@Required 只能放置在 setter 方法上，且通过 XML 配置的 setter 注入，可以使用如下方式来指定：

```
@Requried
setter 方法
```

1、准备测试 Bean

```
package cn.javass.spring.chapter12;
public class TestBean {
    private String message;
    @Required
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean" class="cn.javass.spring.chapter12.TestBean">
    <property name="message" ref="message"/>
</bean>
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

3、测试类和测试方法如下：

```
package cn.javass.spring.chapter12;
//省略import
public class DependencyInjectWithAnnotationTest {
    private static String configLocation =
        "classpath:chapter12/dependecyInjectWithAnnotation.xml";
    private static ApplicationContext ctx =
        new ClassPathXmlApplicationContext("configLocation");
    //1、Spring自带依赖注入注解
     @Test
    public void testRequiredForXmlSetterInject() {
        TestBean testBean = ctx.getBean("testBean", TestBean.class);
        Assert.assertEquals("hello", testBean.getMessage());
    }
}
```

在 XML 配置文件中必须指定 setter 注入，否则在 Spring 容器启动时将抛出如下异常：

```
org.springframework.beans.factory.BeanCreationException:
    Error creating bean with name 'testBean' defined in class path resource
[chapter12/dependecyInjectWithAnnotation.xml]: Initialization of bean failed;
nested exception is org.springframework.beans.factory.BeanInitializationException:
Property 'message' is required for bean 'testBean'
```

二、@Autowired：自动装配

自动装配，用于替代基于 XML 配置的自动装配，详见【3.3.3　自动装配】。

基于@Autowired 的自动装配，默认是根据类型注入，可以用于构造器、字段、方法注入，使用方式如下：

```
@Autowired(required=true)
构造器、字段、方法
```

@Autowired 默认是根据参数类型进行自动装配，且必须有一个 Bean 候选者注入，如果允许出现 0 个 Bean 候选者需要设置属性“required=false”，“required”属性含义和@Required 一样，只是@Required 只适用于基于 XML 配置的 setter 注入方式。

（1）、构造器注入：通过将@Autowired 注解放在构造器上来完成构造器注入，默认构造器参数通过类型自动装配，如下所示：

1、准备测试 Bean，在构造器上添加@AutoWired 注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean11 {
    private String message;
    @Autowired //构造器注入
    private TestBean11(String message) {
        this.message = message;
    }
    //省略message的getter和setter
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean11" class="cn.javass.spring.chapter12.TestBean11"/>
```

2 测试类如下：

```
@Test
public void testAutowiredForConstructor() {
    TestBean11 testBean11 = ctx.getBean("testBean11", TestBean11.class);
    Assert.assertEquals("hello", testBean11.getMessage());
}
```

3、在 Spring 配置文件中没有对“testBean11”进行构造器注入和 setter 注入配置，而是通过在构造器上添加@ Autowired 来完成根据参数类型完成构造器注入。

（2）、字段注入：通过将@Autowired 注解放在构造器上来完成字段注入。

1、准备测试 Bean，在字段上添加@AutoWired 注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean12 {
    @Autowired //字段注入
    private String message;
    //省略getter和setter
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean12" class="cn.javass.spring.chapter12.TestBean12"/>
```

3、测试方法如下：

```
@Test
public void testAutowiredForField() {
    TestBean12 testBean12 = ctx.getBean("testBean12", TestBean12.class);
    Assert.assertEquals("hello", testBean12.getMessage());
}
```

字段注入在基于 XML 配置中无相应概念，字段注入不支持静态类型字段的注入。

（3）、方法参数注入：通过将@Autowired 注解放在方法上来完成方法参数注入。

1、准备测试 Bean，在方法上添加@AutoWired 注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean13 {
    private String message;
    @Autowired //setter方法注入
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean14 {
    private String message;
    private List<String> list;
    @Autowired(required = true) //任意一个或多个参数方法注入
    private void initMessage(String message, ArrayList<String> list) {
        this.message = message;
        this.list = list;
    }
    //省略getter和setter
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean13" class="cn.javass.spring.chapter12.TestBean13"/>
<bean id="testBean14" class="cn.javass.spring.chapter12.TestBean14"/>
<bean id="list" class="java.util.ArrayList">
    <constructor-arg index="0">
        <list>
            <ref bean="message"/>
            <ref bean="message"/>
        </list>
    </constructor-arg>
</bean>
```

3、测试方法如下：

```
@Test
public void testAutowiredForMethod() {
    TestBean13 testBean13 = ctx.getBean("testBean13", TestBean13.class);
    Assert.assertEquals("hello", testBean13.getMessage());

    TestBean14 testBean14 = ctx.getBean("testBean14", TestBean14.class);
    Assert.assertEquals("hello", testBean14.getMessage());
    Assert.assertEquals(ctx.getBean("list", List.class), testBean14.getList());
}
```

方法参数注入除了支持 setter 方法注入，还支持 1 个或多个参数的普通方法注入，在基于 XML 配置中不支持 1 个或多个参数的普通方法注入，方法注入不支持静态类型方法的注入。

注意“**initMessage(String message, ArrayList<String> list)**”方法签名中为什么使用 ArrayList 而不是 List 呢？具体参考【3.3.3　自动装配】一节中的集合类型注入区别。

三、@Value：注入 SpEL 表达式；

用于注入 SpEL 表达式，可以放置在字段方法或参数上，使用方式如下：

```
@Value(value = "SpEL 表达式")
字段、方法、参数
```

1、可以在类字段上使用该注解：

```
@Value(value = "#{message}")
private String message;
```

2、可以放置在带@Autowired 注解的方法的参数上：

```
@Autowired
public void initMessage(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

3、还可以放置在带@Autowired 注解的构造器的参数上：

```
@Autowired
private TestBean43(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

具体测试详见 DependencyInjectWithAnnotationTest 类的 testValueInject 测试方法。

四、@Qualifier：限定描述符，用于细粒度选择候选者；

@Autowired 默认是根据类型进行注入的，因此如果有多个类型一样的 Bean 候选者，则需要限定其中一个候选者，否则将抛出异常，详见【3.3.3 自动装配】中的根据类型进行注入；

@Qualifier 限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

```
@Qualifier(value = "限定标识符")
字段、方法、参数
```

（1）、根据基于 XML 配置中的<qualifier>标签指定的名字进行注入，使用如下方式指定名称：

```
<qualifier
    type="org.springframework.beans.factory.annotation.Qualifier"
    value="限定标识符"/>
```

其中 type 属性可选，指定类型，默认就是 Qualifier 注解类，name 就是给 Bean 候选者指定限定标识符，一个 Bean 定义中只允许指定类型不同的<qualifier>，如果有多个相同 type 后面指定的将覆盖前面的。

1、准备测试 Bean：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
```

```
public class TestBean31 {
    private DataSource dataSource;
    @Autowired
    //根据<qualifier>标签指定Bean限定标识符
    public void initDataSource(
        @Qualifier("mysqlDataSource") DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean31" class="cn.javass.spring.chapter12.TestBean31"/>
```

我们使用@Qualifier("mysqlDataSource")来指定候选 Bean 的限定标识符，我们需要在配置文件中使用<qualifier>标签来指定候选 Bean 的限定标识符“mysqlDataSource”：

```
<bean id="mysqlDataSourceBean"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <qualifier value="mysqlDataSource"/>
</bean>
```

3、测试方法如下：

```
@Test
public void testQualifierInject1() {
    TestBean31 testBean31 = ctx.getBean("testBean31", TestBean31.class);
    try {
        //使用<qualifier>指定的标识符只能被@Qualifier使用
        ctx.getBean("mysqlDataSource");
        Assert.fail();
    } catch (Exception e) {
        //找不到该Bean
        Assert.assertTrue(e instanceof NoSuchBeanDefinitionException);
    }
```

```
        Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"),
                            testBean31.getDataSource());
}
```

从测试可以看出使用<qualifier>标签指定的限定标识符只能被@Qualifier 使用，不能作为 Bean 的标识符，如“ctx.getBean("mysqlDataSource")”是获取不到 Bean 的。

（2）、缺省的根据 Bean 名字注入：最基本方式，是在 Bean 上没有指定<qualifier>标签时一种容错机制，即缺省情况下使用 Bean 标识符注入，但如果你指定了<qualifier>标签将不会发生容错。

1、准备测试 Bean：

```
package cn.javass.spring.chapter12;
//省略 import
public class TestBean32 {
    private DataSource dataSource;
    @Autowired
    @Qualifier(value = "mysqlDataSource2") //指定 Bean 限定标识符
    //@Qualifier(value = "mysqlDataSourceBean")
    //是错误的注入，不会发生回退容错，因为你指定了<qualifier>
    public void initDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean32" class="cn.javass.spring.chapter12.TestBean32"/>
<bean id="oracleDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"/>
```

3、测试方法如下：

```
@Test
public void testQualifierInject2() {
    TestBean32 testBean32 = ctx.getBean("testBean32", TestBean32.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean32.getDataSource());
}
```

默认情况下（没指定<qualifier>标签）@Qualifier 的 value 属性将匹配 Bean 标识符。

（3）、扩展@Qualifier 限定描述符注解：对@Qualifier 的扩展来提供细粒度选择候选者；

具体使用方式就是自定义一个注解并使用@Qualifier 注解其即可使用。

首先让我们考虑这样一个问题，如果我们有两个数据源，分别为 Mysql 和 Oracle，因此注入两者相关资源时就牵扯到数据库相关，如在 DAO 层注入 SessionFactory 时，当然可以采用前边介绍的方式，但为了简单和直观我们希望采用自定义注解方式。

1、扩展@Qualifier 限定描述符注解来分别表示 Mysql 和 Oracle 数据源

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/**  表示注入Mysql相关  */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Mysql {
}
```

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/**  表示注入Oracle相关  */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Oracle {
}
```

2、准备测试 Bean：

```
package cn.javass.spring.chapter12;
//省略 import
public class TestBean33 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Autowired
    public void initDataSource(
```

```
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
    public DataSource getMysqlDataSource() {
        return mysqlDataSource;
    }
    public DataSource getOracleDataSource() {
        return oracleDataSource;
    }
}
```

3、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean33" class="cn.javass.spring.chapter12.TestBean33"/>
```

4、在 Spring 修改定义的两个数据源：

```
<bean id="mysqlDataSourceBean"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <qualifier value="mysqlDataSource"/>
      <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
</bean>
<bean id="oracleDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
</bean>
```

5、测试方法如下：

```
@Test
public void testQualifierInject3() {
    TestBean33 testBean33 = ctx.getBean("testBean33", TestBean33.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"),
                        testBean33.getMysqlDataSoruce());
    Assert.assertEquals(ctx.getBean("oracleDataSource"),
                        testBean33.getOracleDataSoruce());
}
```

测试也通过了，说明我们扩展的@Qualifier 限定描述符注解也能很好工作。

前边演示了不带属性的注解，接下来演示一下带参数的注解：

1、首先定义数据库类型：

```
package cn.javass.spring.chapter12.qualifier;
public enum DataBase {
    ORACLE, MYSQL;
}
```

2、其次扩展@Qualifier 限定描述符注解

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DataSourceType {
    String ip();          //指定ip,用于多数据源情况
    DataBase database();//指定数据库类型
}
```

3、准备测试 Bean：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import cn.javass.spring.chapter12.qualifier.DataBase;
import cn.javass.spring.chapter12.qualifier.DataSourceType;
public class TestBean34 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
     @Autowired
    public void initDataSource(
            @DataSourceType(ip="localhost", database=DataBase.MYSQL)
            DataSource mysqlDataSource,
            @DataSourceType(ip="localhost", database=DataBase.ORACLE)
            DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
    //省略getter方法
}
```

4、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean34" class="cn.javass.spring.chapter12.TestBean34"/>
```

5、在 Spring 修改定义的两个数据源：

```
<bean id="mysqlDataSourceBean"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <qualifier value="mysqlDataSource"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="MYSQL"/>
    </qualifier>
</bean>
<bean id="oracleDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="ORACLE"/>
    </qualifier>
</bean>
```

6、测试方法如下：

```
@Test
public void testQualifierInject3() {
    TestBean34 testBean34 = ctx.getBean("testBean34", TestBean34.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"),
                        testBean34.getMysqlDataSource());
    Assert.assertEquals(ctx.getBean("oracleDataSource"),
                        testBean34.getOracleDataSoruce());
}
```

测试也通过了，说明我们扩展的@Qualifier 限定描述符注解也能很好工作。

四、自定义注解限定描述符：完全不使用@Qualifier，而是自己定义一个独立的限定注解；

1、首先使用如下方式定义一个自定义注解限定描述符：

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface CustomQualifier {
    String value();
}
```

2、准备测试 Bean：

```
package cn.javass.spring.chapter12;
//省略 import
public class TestBean35 {
    private DataSource dataSoruce;
    @Autowired
    public TestBean35(
        @CustomQualifier("oracleDataSource") DataSource dataSource) {
        this.dataSoruce = dataSource;
    }
    public DataSource getDataSoruce() {
        return dataSoruce;
    }
}
```

3、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean35" class="cn.javass.spring.chapter12.TestBean35"/>
```

4、然后在 Spring 配置文件中注册 CustomQualifier 自定义注解限定描述符，只有注册了 Spring 才能识别：

```
<bean id="customAutowireConfigurer"
    class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>cn.javass.spring.chapter12.qualifier.CustomQualifier</value>
        </set>
    </property>
</bean>
```

5、测试方法如下：

```
@Test
public void testQualifierInject5() {
    TestBean35 testBean35 = ctx.getBean("testBean35", TestBean35.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean35.getDataSource());
}
```

从测试中可看出，自定义的和 Spring 自带的没什么区别，因此如果没有足够的理由请使用 Spring 自带的 Qualifier 注解。

到此限定描述符介绍完毕，在此一定要注意以下几点：

- 限定标识符和 Bean 的描述符是不一样的；
- 多个 Bean 定义中可以使用相同的限定标识符；
- 对于集合、数组、字典类型的限定描述符注入，将注入多个具有相同限定标识符的 Bean。

12.2.3 JSR-250 注解

一、@Resource：自动装配，默认根据类型装配，如果指定 name 属性将根据名字装配，可以使用如下方式来指定：

```
@Resource(name = "标识符")
字段或 setter 方法
```

1、准备测试 Bean：

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
public class TestBean41 {
    @Resource(name = "message")
    private String message;
    //省略getter和setter
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean41" class="cn.javass.spring.chapter12.TestBean41"/>
```

3、测试方法如下：

```
@Test
public void testResourceInject1() {
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

使用非常简单，和@Autowired 不同的是可以指定 name 来根据名字注入。

使用@Resource 需要注意以下几点：

- @Resource 注解应该只用于 setter 方法注入，不能提供如@Autowired 多参数方法注入；
- @Resource 在没有指定 name 属性的情况下首先将根据 setter 方法对于的字段名查找资源，如果找不到再根据类型查找；
- @Resource 首先将从 JNDI 环境中查找资源，如果没找到默认再到 Spring 容器中查找，因此如果 JNDI 环境中有和 Spring 容器同名的资源时需要注意。

二、@PostConstruct 和 PreDestroy：通过注解指定初始化和销毁方法定义；

1、在测试类 TestBean41 中添加如下代码：

```
@PostConstruct
public void init() {
    System.out.println("==========init");
}
@PreDestroy
public void destroy() {
    System.out.println("==========destroy");
}
```

2、修改测试方法如下：

```
@Test
public void resourceInjectTest1() {
    ((ClassPathXmlApplicationContext) ctx).registerShutdownHook();
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

类似于通过<bean>标签的 init-method 和 destroy-method 属性指定的初始化和销毁

方法，但具有更高优先级，即注解方式的初始化和销毁方法将先执行。

12.2.4 JSR-330 注解

在测试之前需要准备 JSR-330 注解所需要的 jar 包，到 spring-framework-3.0.5.RELEASE-dependencies.zip 中拷贝如下 jar 包到类路径：

```
com.springsource.javax.inject-1.0.0.jar
```

一、**@Inject**：等价于默认的@Autowired，只是没有 required 属性；

二、**@Named**：指定 Bean 名字，对应于 Spring 自带@Qualifier 中的缺省的根据 Bean 名字注入情况；

三、**@Qualifier**：只对应于 Spring 自带@Qualifier 中的扩展@Qualifier 限定描述符注解，即只能扩展使用，没有 value 属性。

1、首先扩展@Qualifier 限定描述符注解来表示 Mysql 数据源

```
package cn.javass.spring.chapter12.qualifier;
//省略部分import
import javax.inject.Qualifier;
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface JSR330Mysql {
}
```

2、准备测试 Bean：

```
package cn.javass.spring.chapter12;
import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;
import cn.javass.spring.chapter12.qualifier.JSR330Mysql;
public class TestBean51 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Inject
    public void initDataSoruce(
            @JSR330Mysql    DataSource mysqlDataSource,
            @Named("oracleDataSource") DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
```

```
    }
    //省略getter
}
```

3、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<bean id="testBean51" class="cn.javass.spring.chapter12.TestBean51"/>
```

4、在 Spring 修改定义的 mysqlDataSourceBean 数据源：

```
<bean id="mysqlDataSourceBean"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <qualifier value="mysqlDataSource"/>
        <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
        <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
            <attribute key="ip" value="localhost"/>
            <attribute key="database" value="MYSQL"/>
        </qualifier>
        <qualifier type="cn.javass.spring.chapter12.qualifier.JSR330Mysql"/>
    </bean>
```

5、测试方法如下：

```
@Test
public void testInject() {
    TestBean51 testBean51 = ctx.getBean("testBean51", TestBean51.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"),
                        testBean51.getMysqlDataSource());
    Assert.assertEquals(ctx.getBean("oracleDataSource"),
                        testBean51.getOracleDataSource());
}
```

测试也通过了，说明 JSR-330 注解也能很好工作。

从测试中可以看出 JSR-330 注解和 Spring 自带注解依赖注入时主要有以下特点：

- Spring 自带的@Autowired 的缺省情况等价于 JSR-330 的@Inject 注解；
- Spring 自带的@Qualifier 的缺省的根据 Bean 名字注入情况等价于 JSR-330 的 @Named 注解；
- Spring 自带的@Qualifier 的扩展@Qualifier 限定描述符注解情况等价于

JSR-330 的@Qualifier 注解。

12.2.5 JPA 注解

用于注入 EntityManagerFactory 和 EntityManager。

1、准备测试 Bean：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean61 {
    @PersistenceContext(unitName = "entityManagerFactory")
    private EntityManager entityManager;

    @PersistenceUnit(unitName = "entityManagerFactory")
    private EntityManagerFactory entityManagerFactory;

    public EntityManager getEntityManager() {
        return entityManager;
    }
    public EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }
}
```

2、在 Spring 配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下 Bean 配置：

```
<import resource="classpath:chapter7/applicationContext-resources.xml"/>
<import resource="classpath:chapter8/applicationContext-jpa.xml"/>
<bean id="testBean61" class="cn.javass.spring.chapter12.TestBean61"/>
```

此处需要引用第七章和八章的配置文件，细节内容请参考七八两章。

3、测试方法如下：

```
@Test
public void testJpaInject() {
    TestBean61 testBean61 = ctx.getBean("testBean61", TestBean61.class);
    Assert.assertNotNull(testBean61.getEntityManager());
    Assert.assertNotNull(testBean61.getEntityManagerFactory());
}
```

测试也通过了，说明 JPA 注解也能很好工作。

JPA 注解类似于@Resource 注解同样是先根据 unitName 属性去 JNDI 环境中查找，如果没找到在到 Spring 容器中查找。

12.3 注解实现 Bean 定义

12.3.1 概述

前边介绍的 Bean 定义全是基于 XML 方式定义配置元数据，且在【12.2 注解实现 Bean 依赖注入】一节中介绍了通过注解来减少配置数量，但并没有完全消除在 XML 配置文件中的 Bean 定义，因此有没有方式完全消除 XML 配置 Bean 定义呢？

Spring 提供通过扫描类路径中的特殊注解类来自动注册 Bean 定义。同注解驱动事务一样需要开启自动扫描并注册 Bean 定义支持，使用方式如下（resources/chapter12/componentDefinitionWithAnnotation.xml）：

```
<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"
    xmlns:context="http://www.springframework.org/schema/context"
    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/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <aop:aspectj-autoproxy />
    <context:component-scan base-package="cn.javass.spring.chapter12"/>
</beans>
```

使用<context:component-scan>标签来表示需要要自动注册 Bean 定义，而通过 base-package 属性指定扫描的类路径位置。

<context:component-scan>标签将自动开启“**注解实现 Bean 依赖注入**”支持。

此处我们还通过<aop:aspectj-autoproxy/>用于开启 Spring 对@AspectJ 风格切面的支持。

Spring 基于注解实现 Bean 定义支持如下三种注解：

- **Spring 自带的@Component 注解及扩展@Repository、@Service、@Controller**，如图 12-1 所示；

- **JSR-250 1.1 版本中中定义的@ManagedBean 注解**，是 Java EE 6 标准规范之一，不包括在 JDK 中，需要在应用服务器环境使用（如 Jboss），如图 12-2 所示；
- **JSR-330 的@Named 注解**，如图 12-3 所示。

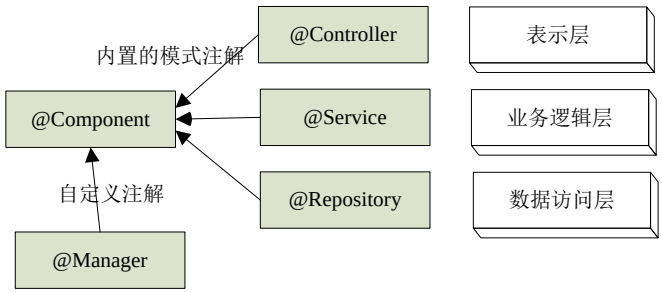

图 12-1 Spring 自带的@Component 注解及扩展

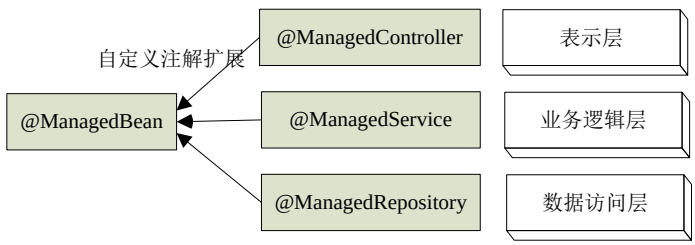

图 12-2 JSR-250 中定义的@ManagedBean 注解及自定义扩展

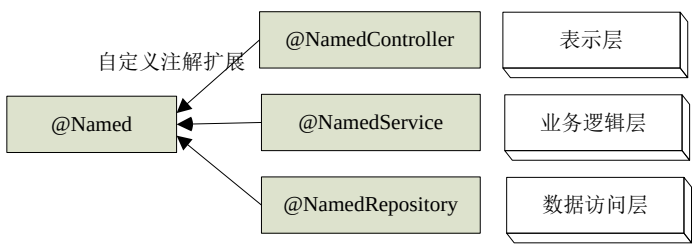

图 12-3 JSR-330 的@Named 注解及自定义扩展

图 12-2 和图 12-3 中的自定义扩展部分是为了配合 Spring 自带的模式注解扩展自定义的，并不包含在 Java EE 6 规范中，在 Java EE 6 中相应的服务层、DAO 层功能由 EJB 来完成。

在 Java EE 中有些注解运行放置在多个地方，如@Named 允许放置在类型、字段、方法参数上等，因此一般情况下放置在类型上表示定义，放置在参数、方法等上边一般代表使用（如依赖注入等等）。

12.3.2 Spring 自带的@Component 注解及扩展

一、@Component：定义 Spring 管理 Bean，使用方式如下：

```
@Component("标识符")
POJO 类
```

在类上使用@Component 注解，表示该类定义为 Spring 管理 Bean，使用默认 value（可选）属性表示 Bean 标识符。

1、定义测试 Bean 类:

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;
@Component("component")
public class TestCompoment {
    @Autowired
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试类和测试方法：

```
package cn.javass.spring.chapter12;
//省略 import
public class ComponentDefinitionWithAnnotationTest {
    private static String configLocation =
        "classpath:chapter12/componentDefinitionWithAnnotation.xml";
    private static ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocation);
    @Test
    public void testComponent() {
        TestCompoment component =
            ctx.getBean("component", TestCompoment.class);
        Assert.assertNotNull(component.getCtx());
    }
}
```

测试成功说明被@Component 注解的 POJO 类将自动被 Spring 识别并注册到 Spring 容器中，且自动支持自动装配。

@AspectJ 风格的切面可以通过@Compenent 注解标识其为 Spring 管理 Bean，而@Aspect 注解不能被 Spring 自动识别并注册为 Bean，必须通过@Component 注解来完成，示例如下：

```
package cn.javass.spring.chapter12.aop;
//省略 import
@Component
@Aspect
public class TestAspect {
    @Pointcut(value="execution(* *(..))")
    private void pointcut() {}
    @Before(value="pointcut()")
    public void before() {
        System.out.println("=======before");
    }
}
```

通过@Component 将切面定义为 Spring 管理 Bean。

二、@Repository：@Component 扩展，被@Repository 注解的 POJO 类表示 DAO 层实现，从而见到该注解就想到 DAO 层实现，使用方式和@Component 相同；

1、定义测试 Bean 类:

```
package cn.javass.spring.chapter12.dao.hibernate;
import org.springframework.stereotype.Repository;
@Repository("testHibernateDao")
public class TestHibernateDaoImpl {

}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testDao() {
    TestHibernateDaoImpl dao =
        ctx.getBean("testHibernateDao", TestHibernateDaoImpl.class);
    Assert.assertNotNull(dao);
}
```

测试成功说明被@Repository 注解的 POJO 类将自动被 Spring 识别并注册到 Spring 容器中，且自动支持自动装配，并且被@Repository 注解的类表示 DAO 层实现。

三、@Service：@Component 扩展，被@Service 注解的 POJO 类表示 Service 层实现，从而见到该注解就想到 Service 层实现，使用方式和@Component 相同；

1、定义测试 Bean 类:

```
package cn.javass.spring.chapter12.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import cn.javass.spring.chapter12.dao.hibernate.TestHibernateDaoImpl;
@Service("testService")
public class TestServiceImpl {
    @Autowired
    @Qualifier("testHibernateDao")
    private TestHibernateDaoImpl dao;
    public TestHibernateDaoImpl getDao() {
        return dao;
    }
}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testService() {
    TestServiceImpl service = ctx.getBean("testService", TestServiceImpl.class);
    Assert.assertNotNull(service.getDao());
}
```

测试成功说明被@Service 注解的 POJO 类将自动被 Spring 识别并注册到 Spring 容器中，且自动支持自动装配，并且被@Service 注解的类表示 Service 层实现。

四、@Controller：@Component 扩展，被@Controller 注解的类表示 Web 层实现，从而见到该注解就想到 Web 层实现，使用方式和@Component 相同；

1、定义测试 Bean 类:

```
package cn.javass.spring.chapter12.action;
//省略import
@Controller
public class TestAction {
```

```
    @Autowired
     private TestServiceImpl testService;

     public void list() {
          //调用业务逻辑层方法
     }
}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testWeb() {
     TestAction action = ctx.getBean("testAction", TestAction.class);
     Assert.assertNotNull(action);
}
```

测试成功说明被@Controller 注解的类将自动被 Spring 识别并注册到 Spring 容器中，且自动支持自动装配，并且被@Controller 注解的类表示 Web 层实现。

大家是否注意到@Controller 中并没有定义 Bean 的标识符，那么默认 Bean 的名字将是以小写开头的类名（不包括包名），即如“TestAction”类的 Bean 标识符为“testAction”。

六、自定义扩展：Spring 内置了三种通用的扩展注解@Repository、@Service、@Controller ，大多数情况下没必要定义自己的扩展，在此我们演示下如何扩展@Component 注解来满足某些特殊规约的需要；

在此我们可能需要一个缓存层用于定义缓存 Bean，因此我们需要自定义一个@Cache 的注解来表示缓存类。

1、扩展@Component：

```
package cn.javass.spring.chapter12.stereotype;
//省略import
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Cache{
     String value() default "";
}
```

扩展十分简单，只需要在扩展的注解上注解@Component 即可，**@Repository**、@Service、@Controller 也是通过该方式实现的，没什么特别之处。

2、定义测试 Bean 类:

```
package cn.javass.spring.chapter12.cache;
@Cache("cache")
public class TestCache {

}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testCache() {
    TestCache cache = ctx.getBean("cache", TestCache.class);
    Assert.assertNotNull(cache);
}
```

测试成功说明自定义的@Cache 注解也能很好的工作，而且实现了我们的目的，使用@Cache 来表示被注解的类是 Cache 层 Bean。

12.3.3 JSR-250 中定义的@ManagedBean 注解

@javax.annotation.ManagedBean 需要在实现 Java EE 6 规范的应用服务器上使用，虽然 Spring3 实现了，但@javax.annotation.ManagedBean 只有在 Java EE 6 环境中才有定义，因此测试前需要我们定义 ManagedBean 类。

1、定义 javax.annotation.ManagedBean 注解类：

```
package javax.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ManagedBean {
    String value() default "";
}
```

其和@Component 完全相同，唯一不同的就是名字和创建者（一个是 Spring，一个是 Java EE 规范）。

2、定义测试 Bean 类:

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
import org.springframework.context.ApplicationContext;
@javax.annotation.ManagedBean("managedBean")
public class TestManagedBean {
    @Resource
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testManagedBean() {
    TestManagedBean testManagedBean =
        ctx.getBean("managedBean", TestManagedBean.class);
    Assert.assertNotNull(testManagedBean.getCtx());
}
```

测试成功说明被@ManagedBean 注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component 来完成如图 12-2 所示的自定义扩展部分。

12.3.4 JSR-330 的@Named 注解

@Named 不仅可以用于依赖注入来指定注入的 Bean 的标识符，还可以用于定义 Bean。即注解在类型上表示定义 Bean，注解在非类型上（如字段）表示指定依赖注入的 Bean 标识符。

1、定义测试 Bean 类:

```
package cn.javass.spring.chapter12;
//省略 import
@Named("namedBean")
public class TestNamedBean {
```

```
    @Inject
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring 配置文件使用 chapter12/ componentDefinitionWithAnnotation.xml 即可且无需修改；

3、定义测试方法：

```
@Test
public void testNamedBean() {
    TestNamedBean testNamedBean =
        ctx.getBean("namedBean", TestNamedBean.class);
    Assert.assertNotNull(testNamedBean.getCtx());
}
```

测试成功说明被@Named 注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component 来完成如图 12-3 所示的自定义扩展部分。

12.3.5 细粒度控制 Bean 定义扫描

在 XML 配置中完全消除了 Bean 定义，而是只有一个<context:component-scan>标签来支持注解 Bean 定义扫描。

前边的示例完全采用默认扫描设置，如果我们有几个组件不想被扫描并自动注册、我们想更改默认的 Bean 标识符生成策略该如何做呢？接下来让我们看一下如何细粒度的控制 Bean 定义扫描，具体定义如下：

```
<context:component-scan
        base-package=""
        resource-pattern="**/*.class"
        name-generator="
            org.springframework.context.annotation.AnnotationBeanNameGenerator"
        use-default-filters="true"
        annotation-config="true">
    <context:include-filter type="aspectj" expression=""/>
    <context:exclude-filter type="regex" expression=""/>
</context:component-scan>
```

- **base-package**：表示扫描注解类的开始位置，即将在指定的包中扫描，其他包中的注解类将不被扫描，默认将扫描所有类路径；
- **resource-pattern**：表示扫描注解类的后缀匹配模式，即“base-package+resource-pattern”将组成匹配模式用于匹配类路径中的组件，默认后缀为“**/*.class”，即指定包下的所有以.class 结尾的类文件；
- **name-generator**：默认情况下的 Bean 标识符生成策略，默认是 AnnotationBeanNameGenerator，其将生成以小写开头的类名（不包括包名）；可以自定义自己的标识符生成策略；
- **use-default-filters**：默认为 true 表示过滤@Component、@ManagedBean、@Named 注解的类，如果改为 false 默认将不过滤这些默认的注解来定义 Bean，即这些注解类不能被过滤到，即不能通过这些注解进行 Bean 定义；
- **annotation-config**：表示是否自动支持注解实现 Bean 依赖注入，默认支持，如果设置为 false，将关闭支持注解的依赖注入，需要通过<context:annotation-config/>开启。

默认情况下将自动过滤@Component、@ManagedBean、@Named 注解的类并将其注册为 Spring 管理 Bean，可以通过在<context:component-scan>标签中指定自定义过滤器将过滤到匹配条件的类注册为 Spring 管理 Bean，具体定义方式如下：

```
<context:include-filter type="aspectj" expression=""/>
<context:exclude-filter type="regex" expression=""/>
```

- **<context:include-filter>**：表示过滤到的类将被注册为 Spring 管理 Bean；
- **<context:exclude-filter>**：表示过滤到的类将不被注册为 Spring 管理 Bean，它比<context:include-filter>具有更高优先级；
- **type**：表示过滤器类型，目前支持注解类型、类类型、正则表达式、aspectj 表达式过滤器，当然也可以自定义自己的过滤器，实现 org.springframework.core.type.filter.TypeFilter 即可；
- **expression**：表示过滤器表达式。

一般情况下没必要进行自定义过滤，如果需要请参考如下示例：

1、cn.javass.spring.chapter12.TestBean14 自动注册为 Spring 管理 Bean：

```
<context:include-filter type="assignable"
                        expression="cn.javass.spring.chapter12.TestBean14"/>
```

2、把所有注解为 org.aspectj.lang.annotation.Aspect 自动注册为 Spring 管理 Bean：

```
<context:include-filter type="annotation"
                        expression="org.aspectj.lang.annotation.Aspect"/>
```

3、将把匹配到正则表达式“cn\.javass\.spring\.chapter12\.TestBean2*”排除，不注册为 Spring 管理 Bean：

```
<context:exclude-filter type="regex"
                        expression="cn\.javass\.spring\.chapter12\.TestBean2*"/>
```

4、将把匹配到 aspectj 表达式“cn.javass.spring.chapter12.TestBean3*”排除，不注册为 Spring 管理 Bean：

```
<context:exclude-filter type="aspectj"
                        expression="cn.javass.spring.chapter12.TestBean3*"/>
```

具体使用就要看项目需要了，如果以上都不满足需要请考虑使用自定义过滤器。

12.3.6 提供更多的配置元数据

1、@Lazy：定义 Bean 将延迟初始化，使用方式如下：

```
@Component("component")
@Lazy(true)
public class TestCompoment {
……
}
```

使用@Lazy 注解指定 Bean 需要延迟初始化。

2、**@DependsOn：**定义 Bean 初始化及销毁时的顺序，使用方式如下：

```
@Component("component")
@DependsOn({"managedBean"})
public class TestCompoment {
……
}
```

3、@Scope：定义 Bean 作用域，默认单例，使用方式如下：

```
@Component("component")
@Scope("singleton")
public class TestCompoment {
……
}
```

4、@Qualifier：指定限定描述符，对应于基于 XML 配置中的<qualifier>标签，使用方式如下：

```
@Component("component")
@Qualifier("component")
public class TestCompoment {
……
}
```

可以使用复杂的扩展，如@Mysql 等等。

5、@Primary：自动装配时当出现多个 Bean 候选者时，被注解为@Primary 的 Bean 将作为首选者，否则将抛出异常，使用方式如下：

```
@Component("component")
@Primary
public class TestCompoment {
……
}
```

1.4 基于 Java 类定义 Bean 配置元数据

12.4.1 概述

基于 Java 类定义 Bean 配置元数据，其实就是通过 Java 类定义 Spring 配置元数据，且直接消除 XML 配置文件。

基于 Java 类定义 Bean 配置元数据中的@Configuration 注解的类等价于 XML 配置文件，@Bean 注解的方法等价于 XML 配置文件中的 Bean 定义。

基于 Java 类定义 Bean 配置元数据需要通过 AnnotationConfigApplicationContext 加载配置类及初始化容器，类似于 XML 配置文件需要使用 ClassPathXmlApplicationContext 加载配置文件及初始化容器。

基于 Java 类定义 Bean 配置元数据需要 CGLIB 的支持，因此要保证类路径中包括 CGLIB 的 jar 包。

12.4.2 Hello World

首先让我们看一下基于 Java 类如何定义 Bean 配置元数据，具体步骤如下：

1、 通过@Configuration 注解需要作为配置的类，表示该类将定义 Bean 配置元数据；
2、 通过@Bean 注解相应的方法，该方法名默认就是 Bean 名，该方法返回值就是 Bean 对象；
3、 通过 AnnotationConfigApplicationContext 或子类加载基于 Java 类的配置。

接下来让我们先来学习一下如何通过 Java 类定义 Bean 配置元数据吧：

1、定义配置元数据的 Java 类如下所示：

```
package cn.javass.spring.chapter12.configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ApplicationContextConfig {
    @Bean
    public String message() {
        return "hello";
    }
}
```

2、定义测试类，测试一下 Java 配置类是否工作：

```
package cn.javass.spring.chapter12.configuration;
//省略 import
public class ConfigurationTest {
    @Test
    public void testHelloworld () {
        AnnotationConfigApplicationContext ctx =
          new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
        Assert.assertEquals("hello", ctx.getBean("message"));
    }
}
```

测试没有报错说明测试通过了，那 AnnotationConfigApplicationContext 是如何工作的呢，接下来让我们分析一下：

- 使用@Configuration 注解配置类，该配置类定义了 Bean 配置元数据；
- 使用@Bean 注解配置类中的方法，该方法名就是 Bean 的名字，该方法返回值就是 Bean 对象。
- 使用 new AnnotationConfigApplicationContext(ApplicationContextConfig.class)

创建应用上下文，构造器参数为使用@Configuration 注解的配置类，读取配置类进行实例化相应的 Bean。

知道如何使用了，接下来就详细介绍每个部分吧。

12.4.3 @Configuration

通过@Configuration 注解的类将被作为配置类使用，表示在该类中将定义 Bean 配置元数据，且使用@Configuration 注解的类本身也是一个 Bean，使用方式如下所示：

```
import org.springframework.context.annotation.Configuration;
@Configuration("ctxConfig")
public class ApplicationContextConfig {
    //定义 Bean 配置元数据
}
```

因为使用@Configuration 注解的类本身也是一个 Bean，因为@Configuration 被@Component 注解了，因此@Configuration 注解可以指定 value 属性值，如“ctxConfig”就是该 Bean 的名字，如使用“ctx.getBean("ctxConfig")”将返回该 Bean。

使用@Configuration 注解的类不能是 final 的，且应该有一个默认无参构造器。

12.4.4 @Bean

通过@Bean 注解配置类中的相应方法，则该方法名默认就是 Bean 名，该方法返回值就是 Bean 对象，并定义了 Spring IoC 容器如何实例化、自动装配、初始化 Bean 逻辑，具体使用方法如下：

```
@Bean(name={},
        autowire=Autowire.NO,
        initMethod="",
        destroyMethod="")
```

- **name：**指定 Bean 的名字，可有多个，第一个作为 Id，其他作为别名；
- **autowire：**自动装配，默认 no 表示不自动装配该 Bean，另外还有 Autowire.BY_NAME 表示根据名字自动装配，Autowire.BY_TYPE 表示根据类型自动装配；
- **initMethod 和 destroyMethod：**指定 Bean 的初始化和销毁方法。

示例如下所示（ApplicationContextConfig.java）

```
@Bean
public String message() {
    return new String("hello");
}
```

如上使用方式等价于如下基于 XML 配置方式

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

使用@Bean 注解的方法不能是 private、final 或 static 的。

12.4.5 提供更多的配置元数据

详见【12.3.6 提供更多的配置元数据】中介绍的各种注解，这些注解同样适用于@Bean 注解的方法。

12.4.6 依赖注入

基于 Java 类配置方式的 Bean 依赖注入有如下两种方式：

- 直接依赖注入，类似于基于 XML 配置方式中的显示依赖注入；
- 使用注解实现 Bean 依赖注入：如@Autowired 等等。

在本示例中我们将使用【第三章 DI】中的测试 Bean。

1、 直接依赖注入：包括构造器注入和 setter 注入。

- **构造器注入：**通过在@Bean 注解的实例化方法中使用有参构造器实例化相应的 Bean 即可，如下所示(ApplicationContextConfig.java)：

```
@Bean
public HelloApi helloImpl3() {
    //通过构造器注入,分别是引用注入（message()）和常量注入（1）
    return new HelloImpl3(message(), 1); //测试Bean详见【3.1.2 构造器注入】
}
```

- **setter 注入：**通过在@Bean 注解的实例化方法中使用无参构造器实例化后，通过相应的 setter 方法注入即可，如下所示(ApplicationContextConfig.java)：

```
@Bean
public HelloApi helloImpl4() {
    HelloImpl4 helloImpl4 = new HelloImpl4();//测试Bean详见【3.1.3 setter注入】
    //通过setter注入注入引用
    helloImpl4.setMessage(message());
    //通过setter注入注入常量
    helloImpl4.setIndex(1);
```

2、使用注解实现 Bean 依赖注入：详见【12.2　注解实 Bean 依赖注入】。

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void testDependencyInject() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
    ctx.getBean("helloImpl3", HelloApi.class).sayHello();
    ctx.getBean("helloImpl4", HelloApi.class).sayHello();
}
```

12.4.7　方法注入

在基于 XML 配置方式中，Spring 支持查找方法注入和替换方法注入，但在基于 Java 配置方式中只支持查找方法注入，一般用于在一个单例 Bean 中注入一个原型 Bean 的情况，具体详见【3.3.5　方法注入】，如下所示（ApplicationContextConfig.java）：

```
@Bean
@Scope("singleton")
public HelloApi helloApi2() {
    HelloImpl5 helloImpl5 = new HelloImpl5() {
        @Override
        public Printer createPrototypePrinter() {
            //方法注入，注入原型Bean
            return prototypePrinter();
        }
        @Override
        public Printer createSingletonPrinter() {
            //方法注入，注入单例Bean
            return singletonPrinter();
        }
    };
    //依赖注入,注入单例Bean
    helloImpl5.setPrinter(singletonPrinter());
    return helloImpl5;
}
```

```
@Bean
@Scope(value="prototype")
public Printer prototypePrinter() {
      return new Printer();
  }
@Bean
@Scope(value="singleton")
public Printer singletonPrinter() {
      return new Printer();
}
```

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void testLookupMethodInject() {
      AnnotationConfigApplicationContext ctx =
            new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
      System.out.println("=======prototype sayHello======");
      HelloApi helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
      helloApi2.sayHello();
      helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
      helloApi2.sayHello();
}
```

如上测试等价于【3.3.5　方法注入】中的查找方法注入。

12.4.8　@Import

类似于基于 XML 配置中的<import/>，基于 Java 的配置方式提供了@Import 来组合模块化的配置类，使用方式如下所示：

```
package cn.javass.spring.chapter12.configuration;
//省略 import
@Configuration("ctxConfig2")
@Import({ApplicationContextConfig.class})
public class ApplicationContextConfig2 {
      @Bean(name = {"message2"})
      public String message() {
            return "hello";
      }
```

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void    importTest() {
      AnnotationConfigApplicationContext ctx =
            new AnnotationConfigApplicationContext(ApplicationContextConfig2.class);
      Assert.assertEquals("hello", ctx.getBean("message"));
}
```

使用非常简单，在此就不多介绍了。

12.4.9 结合基于 Java 和基于 XML 方式的配置

基于 Java 方式的配置方式不是为了完全替代基于 XML 方式的配置，两者可以结合使用，因此可以有两种结合使用方式：

- 在基于 Java 方式的配置类中引入基于 XML 方式的配置文件；
- 在基于 XML 方式的配置文件中中引入基于 Java 方式的配置。

一、在基于 Java 方式的配置类中引入基于 XML 方式的配置文件：在@Configuration 注解的配置类上通过@ImportResource 注解引入基于 XML 方式的配置文件，示例如下所示：

1、定义基于 XML 方式的配置文件(chapter12/configuration/importResource.xml)：

```
<bean id="message3" class="java.lang.String">
      <constructor-arg index="0" value="test"></constructor-arg>
</bean>
```

2、修改基于 Java 方式的配置类 ApplicationContextConfig，添加如下注解：

```
@Configuration("ctxConfig") //1、使用@Configuration注解配置类
@ImportResource("classpath:chapter12/configuration/importResource.xml")
public class ApplicationContextConfig {
……
}
```

使用@ImportResource 引入基于 XML 方式的配置文件，如果有多个请使用@ImportResource({"config1.xml", "config2.xml"})方式指定多个配置文件。

二、在基于 XML 方式的配置文件中中引入基于 Java 方式的配置：直接在 XML 配置文件中声明使用@Configuration 注解的配置类即可，示例如下所示：

1、定义基于 Java 方式的使用@Configuration 注解的配置类在此我们使用 ApplicationContextConfig.java。

2、定义基于 XML 方式的配置文件（chapter12/configuration/xml-config.xml）：

```
<context:annotation-config/>
<bean id="ctxConfig"
        class="cn.javass.spring.chapter12.configuration.ApplicationContextConfig"/>
```

- <context:annotation-config/>：用于开启对注解驱动支持，详见【12.2 注解实现 Bean 依赖注入】；
- <bean id="ctxConfig" class="……"/>：直接将使用@Configuration 注解的配置类在配置文件中进行 Bean 定义即可。

3、测试代码如下所示(ConfigurationTest.java)：：

```
public void testXmlConfig() {
    String configLocations[] = {"chapter12/configuration/xml-config.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(configLocations);
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

测试成功，说明通过在基于 XML 方式的配置文件中能获取到基于 Java 方式的配置文件中定义的 Bean，如“message”Bean。

12.4.10 基于 Java 方式的容器实例化

基于 Java 方式的容器由 AnnotationConfigApplicationContext 表示，其实例化方式主要有以下几种：

一、对于只有一个@Configuration 注解的配置类，可以使用如下方式初始化容器：

```
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
```

二、对于有多个@Configuration 注解的配置类，可以使用如下方式初始化容器：

```
AnnotationConfigApplicationContext ctx1 =
    new AnnotationConfigApplicationContext(
        ApplicationContextConfig.class, ApplicationContextConfig2.class);
```

或者

```
AnnotationConfigApplicationContext ctx2 =
    new AnnotationConfigApplicationContext();
    ctx2.register(ApplicationContextConfig.class);
    ctx2.register(ApplicationContextConfig2.class);
```

三、对于【12.3　注解实现 Bean 定义】中通过扫描类路径中的特殊注解类来自动注册 Bean 定义，可以使用如下方式来实现：

```
public void testComponentScan() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext();
    ctx.scan("cn.javass.chapter12.confiuration");
    ctx.refresh();
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

以上配置方式等价于基于 XML 方式中的如下配置：

```
<context:component-scan base-package="cn.javass.chapter12.confiuration"/>
```

四、在 web 环境中使用基于 Java 方式的配置，通过修改通用配置实现，详见**【10.1.2 通用配置】**：

1、修改通用配置中的 Web 应用上下文实现，在此需要使用 AnnotationConfigWebApplicationContext：

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
```

2、指定加载配置类，类似于指定加载文件位置，在基于 Java 方式中需要指定需要加载的配置类：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig,
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig2
    </param-value>
```

- **contextConfigLocation**：除了可以指定配置类，还可以指定“扫描的类路径”，其加载步骤如下：

  1、首先验证指定的配置是否是类，如果是则通过注册配置类来完成 Bean 定义加载，即如通过 ctx.register(ApplicationContextConfig.class)加载定义；

  2、如果指定的配置不是类，则通过扫描类路径方式加载注解Bean定义，即将通过ctx.scan("cn.javass.chapter12.confiuration")加载Bean定义。

12.5 综合示例

12.5.1 概述

在第十一章中我们介绍了 SSH 集成，在进行 SSH 集成时都是通过基于 XML 配置文件配置每层的 Bean，从而产生许多 XML 配置文件，本节将通过注解方式消除部分 XML 配置文件，实现所谓的零配置。

12.5.2 项目拷贝

1、拷贝【第十一章 SSH 集成开发】中的“pointShop”项目将其命名为“pointShop2”；

2、修改“pointShop2”项目下的“.settings”文件夹下的“org.eclipse.wst.common.component”文件，将“**<property name="context-root" value="pointShop"/>**”修改为“**<property name="context-root" value="pointShop2"/>**”，即该 web 项目的上下文为“pointShop2”，在浏览器中可以通过 http://localhost:8080/pointShop2 来访问该 web 项目。

12.5.3 数据访问层变化

将 dao 层配置文件中的 dao 实现 Bean 定义删除，通过在 dao 实现类头上添加“@Repository”来定义 dao 实现 Bean，并通过注解@Autowired 来完成依赖注入。

1、删除 DAO 层配置文件(cn/javass/point/dao/applicationContext-hibernate.xml)中的如下

配置：

```
<bean id="abstractDao" abstract="true" init-method="init">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao" class="cn.javass.point.dao.hibernate.GoodsHibernateDao"
      parent="abstractDao"/>
<bean id="goodsCodeDao" class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao"
      parent="abstractDao"/>
```

2、修改通用 DAO 实现 cn.javass.commons.dao.hibernate.BaseHibernateDao，通过注解实现依赖注入和指定初始化方法：

```
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable>
    extends HibernateDaoSupport implements IBaseDao<M, PK>, InitializingBean {
     //省略类字段
    @Autowired @Required
    public void setSf(SessionFactory sf) {
        setSessionFactory(sf);
    }
    @PostConstruct
    @SuppressWarnings("unchecked")
     public void init() {
         //省略具体实现代码
     }
}
```

- setSf 方法：通过@Autowired 注解自动注入 SessionFactory 实现；
- init 方法：通过@PostConstruct 注解表示该方法是初始化方法；

3、修改 cn.javass.point.dao.hibernate.GoodsHibernateDao，在该类上添加@Repository 注解来进行 DAO 层 Bean 定义：

```
@Repository
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer>
implements IGoodsDao {
……
}
```

4、修改 cn.javass.point.dao.hibernate.GoodsCodeHibernateDao，在该类上添加@Repository 注解来进行 DAO 层 Bean 定义：

```
@Repository
public class GoodsCodeHibernateDao extends BaseHibernateDao<GoodsCodeModel, Integer> implements IGoodsCodeDao {
……
}
```

DAO 层到此就修改完毕，其他地方无需修改。

12.5.4 业务逻辑层变化

将 service 层配置文件中的 service 实现 Bean 定义删除，通过在 service 实现类头上添加“@Service”来定义 service 实现 Bean，并通过注解@Autowired 来完成依赖注入。

1、删除 Service 层配置文件(cn/javass/point/service/applicationContext-service.xml)中的如下配置：

```
<bean id="goodsService" class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
<bean id="goodsCodeService"
     class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

1、修改 cn.javass.point.service.impl.GoodsServiceImpl，在该类上添加@Service 注解来进行 Service 层 Bean 定义：

```
@Service
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer>
    implements IGoodsService {

    @Autowired @Required
    public void setGoodsDao(IGoodsDao dao) {
        setDao(dao);
    }
}
```

- **setGoodsDao 方法：**用于注入 IGoodsDao 实现，此处直接委托给 setDao 方法。

2、修改 cn.javass.point.service.impl.GoodsCodeServiceImpl，在该类上添加@Service 注

解来进行 Service 层 Bean 定义：

```
@Service
public class GoodsCodeServiceImpl extends BaseServiceImpl<GoodsCodeModel, Integer>
    implements IGoodsCodeService {
    @Autowired @Required
    public void setGoodsCodeDao(IGoodsCodeDao dao) {
        setDao(dao);
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeDao 方法：**用于注入 IGoodsCodeDao 实现，此处直接委托给 setDao 方法；
- **setGoodsService 方法：**用于注入 IGoodsService 实现。

Service 层到此就修改完毕，其他地方无需修改。

12.5.5 表现层变化

类似于数据访问层和业务逻辑层修改，对于表现层配置文件直接删除，通过在 action 实现类头上添加“@Controller”来定义 action 实现 Bean，并通过注解@Autowired 来完成依赖注入。

1、 删除表现层所有 Spring 配置文件(cn/javass/point/web)：

```
cn/javass/point/web/pointShop-admin-servlet.xml
cn/javass/point/web/pointShop-front-servlet.xml
```

2、修改表现层管理模块的 cn.javass.point.web.admin.action.GoodsAction，在该类上添加@Controller 注解来进行表现层 Bean 定义，且作用域为“prototype”：

```
@Controller("/admin/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
  private IGoodsService goodsService;
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsService 方法：**用于注入 IGoodsService 实现。

3、修改表现层管理模块的 cn.javass.point.web.admin.action.GoodsCodeAction，在该类上添加@Controller 注解来进行表现层 Bean 定义，且作用域为“prototype”：

```
@Controller("/admin/goodsCodeAction")
@Scope("prototype")
public class GoodsCodeAction extends BaseAction {
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeService 方法：**用于注入 IGoodsCodeService 实现；
- **setGoodsService 方法：**用于注入 IGoodsService 实现。

3、修改表现层前台模块的 cn.javass.point.web.front.action.GoodsAction，在该类上添加@Controller 注解来进行表现层 Bean 定义，且作用域为“prototype”：

```
@Controller("/front/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
}
```

- **setGoodsCodeService 方法：**用于注入 IGoodsCodeService 实现；
- **setGoodsService 方法：**用于注入 IGoodsService 实现。

12.5.6 其他变化

1、定义一个基于 Java 方法的配置类，用于加载 XML 配置文件：

```
package cn.javass.point;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
@Configuration
@ImportResource(
        {"classpath:applicationContext-resources.xml",
          "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
          "classpath:cn/javass/point/service/applicationContext-service.xml"
        })
public class AppConfig {
}
```

该类用于加载零配置中一般不变的 XML 配置文件，如事务管理，数据源、SessionFactory，这些在几乎所有项目中都是类似的，因此可以作为通用配置。

2、修改集成其它 Web 框架的通用配置，将如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext-resources.xml,
        classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
        classpath:cn/javass/point/service/applicationContext-service.xml,
        classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
        classpath:cn/javass/point/web/pointShop-front-servlet.xml
    </param-value>
</context-param>
```

修改为如下配置：

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
     org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>cn.javass.point</param-value>
</context-param>
```

- **contextClass：** 使用 notationConfigWebApplicationContext 替换默认的 XmlWebApplicationContext；
- **contextConfigLocation：**指定为“cn.javass.point”，表示将通过扫描该类路径“cn.javass.point”下的注解类来进行加载 Bean 定义。

启动 pointShop2 项目，在浏览器输入 http://localhost:8080/pointShop2/admin/goods/list.action 访问积分商城后台，如果没问题说明零配置整合成功。

到此零配置方式实现 SSH 集成已经整合完毕，相对于基于 XML 方式主要减少了配置的数量和配置文件的数量。

第十三章　测试

1.1　概述

13.1.1　测试

软件测试的目的首先是为了保证软件功能的正确性，其次是为了保证软件的质量，软件测试相当复杂，已经超出本书所涉及的范围，本节将只介绍软件测试流程中前两个步骤：单元测试和集成测试。

Spring 提供了专门的测试模块用于简化单元测试和集成测试，单元测试和集成测试一般由程序员实现。

13.2　单元测试

13.2.1　概述

单元测试是最细粒度的测试，即具有原子性，通常测试的是某个功能（如测试类中的某个方法的功能）。

采用依赖注入后我们的代码对 Spring IoC 容器几乎没有任何依赖，因此在对我们代码进行测试时无需依赖 Spring IoC 容器，我们只需要通过简单的实例化对象、注入依赖然后测试相应方法来测试该方法是否完成我们预期的功能。

在本书中使用的传统开发流程，即先编写代码实现功能，然后再写测试来验证功能是否正确，而不是测试驱动开发，测试驱动开发是指在编写代码实现功能之前先写测试，然后再根据测试来写满足测试要求的功能代码，通过测试来驱动开发，如果对测试驱动开发感兴趣推荐阅读【测试驱动开发的艺术】。

在实际工作中，应该只对一些复杂的功能进行单元测试，对于一些简单的功能（如数据访问层的 CRUD）没有必要花费时间进行单元测试。

Spring 对单元测试提供如下支持：

- Mock 对象：Spring 通过 Mock 对象来简化一些场景的单元测试：
  - **JNDI 测试支持**：在 org.springframework.mock.jndi 包下通过了 SimpleNamingContextBuilder 来来创建 JNDI 上下文 Mock 对象，从而无需依赖特定 Java EE 容器即可完成 JNDI 测试。

- **web 测试支持：**在 org.springframework.mock.web 包中提供了一组 Servlet API 的 Mock 对象，从而可以无需 Web 容器即可测试 web 层的类。

➢ **工具类**：通过通用的工具类来简化编写测试代码：

- **反射工具类：**在 org.springframework.test.util 包下的 ReflectionTestUtils 能通过反射完成类的非 public 字段或 setter 方法的调用；
- **JDBC 工具类：**在 org.springframework.test.util 包下的 SimpleJdbcTestUtils 能读取一个 sql 脚本文件并执行来简化 SQL 的执行，还提供了如清空表、统计表中行数的简便方法来简化测试代码的编写。

接下来让我们学习一下开发过程中各层代码如何编写测试用例。

13.2.2 准备测试环境

1、Junit 安装：将 Junit 4 包添加到“pointShop”项目中，具体方法请参照【2.2.3 Hello World】。

2、jMock 安装：到 jMock 官网【http://www.jmock.org/】下载最新的 jMock 包，在本书中使用 jMock 2.5.1 版本，将下载的“jmock-2.5.1-jars.zip ”包中的如下 jar 包拷贝到项目的 lib 目录下并添加到类路径：

```
objenesis-1.0.jar
jmock-script-2.5.1.jar
jmock-legacy-2.5.1.jar
jmock-junit4-2.5.1.jar
jmock-junit3-2.5.1.jar
jmock-2.5.1.jar
hamcrest-library-1.1.jar
hamcrest-core-1.1.jar
bsh-core-2.0b4.jar
```

注：cglib 包无需添加到类路径，因为我们之前已经提供。

3、添加 Spring 测试支持包：将下载的 spring-framework-3.0.5.RELEASE-with-docs.zip 包中的如下 jar 包拷贝到项目的 lib 目录下并添加到类路径：

```
dist\org.springframework.test-3.0.5.RELEASE.jar
```

4、在“pointShop”项目下新建 test 文件夹并将其添加到【Java Build Path】中，该文件夹用于存放测试代码，从而分离测试代码和开发代码。

到此测试环境搭建完毕。

13.2.3 数据访问层

数据访问层单元测试，目的是测试该层定义的接口实现方法的行为是否正确，其实本质是测试是否正确与数据库交互，是否发送并执行了正确的 SQL，SQL 执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层单元测试通过 Mock 对象与数据库交互的 API 来完成测试。

接下来让我们学习一下如何进行数据访问层单元测试：

1、在 test 文件夹下创建如下测试类：

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsHibernateDaoUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};
    //2、Mock HibernateTemplate类
    private final HibernateTemplate mockHibernateTemplate =
        context.mock(HibernateTemplate.class);
    private IGoodsDao goodsDao = null;

    @Before
    public void setUp() {
        //3、创建IGoodsDao实现
        GoodsHibernateDao goodsDaoTemp = new GoodsHibernateDao();
        //4、通过ReflectionTestUtils注入需要的非public字段数据
        ReflectionTestUtils.setField(goodsDaoTemp, "entityClass", GoodsModel.class);
        //5、注入mockHibernateTemplate对象
        goodsDaoTemp.setHibernateTemplate(mockHibernateTemplate);
        //6、赋值给我们要使用的接口
        goodsDao = goodsDaoTemp;
    }
}
```

- Mockery：jMock 核心类，用于创建 Mock 对象的，通过其 mock 方法来创建相应接口或类的 Mock 对象。
- goodsDaoTemp：需要测试的 IGoodsDao 实现，通过 ReflectionTestUtils 注入需要的非 public 字段数据。

2、测试支持写完后，接下来测试一下 IGoodsDao 的 get 方法是否满足需求：

```
@Test
public void testSave () {
    //7、创建需要的Model数据
    final GoodsModel expected = new GoodsModel();
    //8、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
        {
            //9、表示需要调用且只调用一次mockHibernateTemplate的get方法，
            //且get方法参数为(GoodsModel.class, 1)，并将返回goods
            one(mockHibernateTemplate).get(GoodsModel.class, 1);
            will(returnValue(expected));
        }
    });
    //10、调用goodsDao的get方法，在内部实现中将委托给
    //getHibernateTemplate().get(this.entityClass, id);
    //因此按照第8步定义的预期行为将返回goods
    GoodsModel actual = goodsDao.get(1);
    //11、来验证第8步定义的预期行为是否调用了
    context.assertIsSatisfied();
    //12、验证goodsDao.get(1)返回结果是否正确
    Assert.assertEquals(goods, expected);
}
```

- **context.checking()：**该方法中用于定义预期行为，其中第9步定义了需要调用一次且只调用一次mockHibernateTemplate的get方法，且get方法参数为(GoodsModel.class, 1)，并将返回goods对象。
- **goodsDao.get(1)：**调用goodsDao的get方法，在内部实现中将委托给“getHibernateTemplate().get(this.entityClass, id)”。
- **context.assertIsSatisfied()：**来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(expected, actual)：**用于验证“goodsDao.get(1) ”返回的结果是否是预期结果。

以上测试方法其实是没有必要的，对于非常简单的CRUD没有必要写单元测试，只有相当复杂的方法才有必要写单元测试。

这种通过Mock对象来测试数据访问层代码其实一点意义没有，因为这里没有与数据库交互，无法验证真实环境中与数据库交互是否正确，因此这里只是告诉你如何测试数据访

问层代码，在实际工作中一般通过集成测试来完成数据访问层测试。

13.2.4 业务逻辑层

业务逻辑单元测试，目的是测试该层的业务逻辑是否正确并通过 Mock 数据访问层对象来隔离与数据库交互，从而无需连接数据库即可测试业务逻辑是否正确。

接下来让我们学习一下如何进行业务逻辑层单元测试：

1、在 test 文件夹下创建如下测试类：

```
package cn.javass.point.service.impl;
//省略 import
public class GoodsCodeServiceImplUnitTest {
    //1、Mock 对象上下文，用于创建 Mock 对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持 Mock 非接口类，默认只支持 Mock 接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeDao 接口
    private IGoodsCodeDao goodsCodeDao = context.mock(IGoodsCodeDao.class);;

    private IGoodsCodeService goodsCodeService;

    @Before
    public void setUp() {
        GoodsCodeServiceImpl goodsCodeServiceTemp =
            new GoodsCodeServiceImpl();
        //3、依赖注入
        goodsCodeServiceTemp.setDao(goodsCodeDao);
        goodsCodeService = goodsCodeServiceTemp;
    }
}
```

以上测试支持代码和数据访问层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下购买商品 Code 码是否满足需求：

测试业务逻辑时需要分别测试多种场景，即如在某种场景下成功或失败等等，即测试应该全面，每个功能点都应该测试到。

2.1、测试购买失败的场景：

```
@Test(expected = NotCodeException.class)
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(null));
      }
    });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
}
```

- **context.checking()：**该方法中用于定义预期行为，其中第5步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回null。
- **goodsCodeService.buy("test", goodsId)：**调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回null，因此buy方法将抛出“NotCodeException”异常，从而表示没有Code码。
- **context.assertIsSatisfied()：**来验证前边定义的预期行为是否执行，且是否正确。
- 由于我们在预期行为中调用getOneNotExchanged将返回null，因此测试将失败且抛出NotCodeException异常。

2.2、测试购买成功的场景：

```
@Test()
public void testBuySuccess () {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //6、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //7、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(goodsCode));
          //8、表示需要调用goodsCodeDao对象的save方法一次且仅一次
          //且参数为goodsCode
          one(goodsCodeDao).save(goodsCode);
      }
    });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
    Assert.assertEquals(goodsCode.isExchanged(), true);
}
```

- **context.checking()：**该方法中用于定义预期行为，其中第7步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回goodsCode对象；第8步定义了需要调用goodsCodeDao对象的save一次且仅一次。
- **goodsCodeService.buy("test", goodsId)：**调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回goodsCode，因此buy方法将成功执行。
- **context.assertIsSatisfied()：**来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(goodsCode.isExchanged(), true)：**表示goodsCode已经被购买过了。
- 由于我们在预期行为中调用getOneNotExchanged将返回一个goodsCode对象，因此测试将成功，如果失败说明业务逻辑写错了。

到此业务逻辑层测试完毕，在进行业务逻辑层测试时我们只关心业务逻辑是否正确，而不关心底层数据访问层如何实现，因此测试业务逻辑层时只需Mock 数据访问层对象，然后定义一些预期行为来满足业务逻辑测试需求即可。

13.2.5 表现层

表现层测试包括如 Struts2 的 Action 单元测试、拦截器单元测试、JSP 单元测试等等，在此我们只学习 Struts2 的 Action 单元测试。

Struts2 的 Action 测试相对业务逻辑层测试相对复杂一些，因为牵扯到使用如 Servlet API、ActionContext 等等，这里需要通过 stub（桩）实现或 mock 对象来模拟如 HttpServletRequest 等对象。

一、首先学习一些最简单的 Action 测试：

1、在 test 文件夹下创建如下测试类：

```
package cn.javass.point.web.front;
import cn.javass.point.service.IGoodsCodeService;
import cn.javass.point.web.front.action.GoodsAction;
//省略部分 import
public class GoodsActionUnitTest {
    //1、Mock 对象上下文，用于创建 Mock 对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持 Mock 非接口类，默认只支持 Mock 接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeService 接口
    private IGoodsCodeService goodsCodeService =
        context.mock(IGoodsCodeService.class);

    private GoodsAction goodsAction;

    @Before
    public void setUp() {
        goodsAction = new GoodsAction();
        //3、依赖注入
        goodsAction.setGoodsCodeService(goodsCodeService);
    }
}
```

以上测试支持代码和业务逻辑层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下前台购买商品 Code 码是否满足需求：

类似于测试业务逻辑时需要分别测试多种场景，测试 Action 同样需要分别测试多种场景。

2.1、测试购买失败的场景：

```
@Test
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且抛出NotCodeException异常
          one(goodsCodeService).buy("test", goodsId);
          will(throwException(new NotCodeException()));
      }
    });
    //6、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(ReflectionTestUtils.getField(
        goodsAction, "BUY_RESULT"), actualResultCode);
    Assert.assertTrue(goodsAction.getActionErrors().size() > 0);
}
```

- **context.checking()：**该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将抛出NotCodeException异常。
- **goodsAction.setGoodsId(goodsId)：**用于模拟Struts注入请求参数，即完成数据绑定。
- **goodsAction.buy()：**调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()：**来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(**
  goodsAction, "BUY_RESULT"), actualResultCode)：验证返回的Result是否是我们指定的。
- **Assert.assertTrue(goodsAction.getActionErrors().size() > 0)：**表示执行Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能是错误的，需要修改。

2.2、测试购买成功的场景：

```
@Test
public void testBuySuccess() {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //7、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //8、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且返回goodsCode对象
          one(goodsCodeService).buy("test", goodsId);
          will(returnValue(goodsCode));
      }
    });
    //9、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(
        ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode);
    Assert.assertTrue(goodsAction.getActionErrors().size() == 0);
}
```

- **context.checking()：**该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将返回goodsCode对象。
- **goodsAction.setGoodsId(goodsId)：**用于模拟Struts注入请求参数，即完成数据绑定。
- **goodsAction.buy()：**调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()：**来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode)：**验证返回的Result是否是我们指定的。
- **Assert.assertTrue(goodsAction.getActionErrors().size() == 0)：**表示执行Action时没有错误，即Action动作正确。如果条件不成立，说明我们Action功能是错误的，需要修改。

通过模拟 ActionContext 对象内容从而可以非常容易的测试 Action 中各种与 http 请求相关情况，无需依赖 web 服务器即可完成测试。但对于如果我们使用 htpp 请求相关对象的该如何测试？如果我们需要使用 ActionContext 获取值栈数据应该怎么办？这就需要 Struts 提供的 junit 插件支持了。我们会在集成测试中介绍。

对于表现层其他功能的单元测试本书不再介绍，如 JSP 单元测试、拦截器单元测试等等。

13.3 集成测试

13.3.1 概述

集成测试是在单元测试之上，通常是将一个或多个已进行过单元测试的组件组合起来完成的，即集成测试中一般不会出现 Mock 对象，都是实实在在的真实实现。

对于单元测试，如前边在进行数据访问层单元测试时，通过 Mock HibernateTemplate 对象然后将其注入到相应的 DAO 实现，此时单元测试只测试某层的某个功能是否正确，对其他层如何提供服务采用 Mock 方式提供。

对于集成测试，如要进行数据访问层集成测试时，需要实实在在的 HibernateTemplate 对象然后将其注入到相应的 DAO 实现，此时集成测试将不仅测试该层功能是否正确，还将测试服务提供者提供的服务是否正确执行。

使用 Spring 的一个好处是能非常简单的进行集成测试，无需依赖 web 服务器或应用服务器即可完成测试。Spring 通过提供一套 TestContext 框架来简化集成测试，使用 TestContext 测试框架能获得许多好处，如 Spring IoC 容器缓存、事务管理、依赖注入、Spring 测试支持类等等。

13.3.2 Spring TestContext 框架支持

Spring TestContext 框架提供了一些通用的集成测试支持，主要提供如下支持：

一、**上下文管理及缓存：**

对于每一个测试用例（测试类）应该只有一个上下文，而不是每个测试方法都创建新的上下文，这样有助于减少启动容器的开销，提供测试效率。可通过如下方式指定要加载的上下文：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
               "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
public class GoodsHibernateDaoIntegrationTest {
}
```

- **locations**：指定 Spring 配置文件位置；
- **inheritLocations**：如果设置为 false，将屏蔽掉父类中使用该注解指定的配置文件位置，默认为 true 表示继承父类中使用该注解指定的配置文件位置。

二、**Test Fixture（测试固件）的依赖注入：**

Test Fixture 可以指运行测试时需要的任何东西，一般通过@Before 定义的初始化 Fixture 方法准备这些资源，而通过@After 定义的销毁 Fixture 方法销毁或还原这些资源。

Test Fixture 的依赖注入就是使用 Spring IoC 容器的注入功能准备和销毁这些资源。可通过如下方式注入 Test Fixture：

```
@Autowired
private IGoodsDao goodsDao;
@Autowired
private ApplicationContext ctx;
```

即可以通过 Spring 提供的注解实现 Bean 的依赖注入来完成 Test Fixture 的依赖注入。

三、**事务管理：**

开启测试类的事务管理支持，即使用 Spring 容器的事务管理功能，从而可以独立于应用服务器完成事务相关功能的测试。为了使测试中的事务管理起作用需要通过如下方式开启测试类事务的支持：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
               "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(
    transactionManager = "txManager", defaultRollback=true)
public class GoodsHibernateDaoIntegrationTest {
}
```

- **transactionManager:**指定事务管理器；
- **defaultRollback**：是否回滚事务，默认为 true 表示回滚事务。

Spring 还通过提供如下注解来简化事务测试：

- **@Transactional**：使用@Transactional 注解的类或方法表示需要事务支持；
- **@NotTransactional**：只能注解方法，使用@NotTransactional 注解的方法表示不需要事务支持，即不运行在事务中，Spring 3 开始已不推荐使用；
- **@BeforeTransaction 和@AfterTransaction**：使用这两个注解注解的方法

定义了在一个事务性测试方法之前或之后执行的行为，且被注解的方法将运行在该事务性方法的事务之外。

- **@Rollback(true)**：默认为 true，用于替换@TransactionConfiguration 中定义的 defaultRollback 指定的回滚行为。

四、常用注解支持：Spring 框架提供如下注解来简化集成测试：

- **@DirtiesContext**：表示每个测试方法执行完毕需关闭当前上下文并重建一个全新的上下文，即不缓存上下文。可应用到类或方法级别，但在 JUnit 3.8 中只能应用到方法级别。
- **@ExpectedException**：表示被注解的方法预期将抛出一个异常，使用如@ExpectedException(NotCodeException.class)来指定异常，定义方式类似于 Junit 4 中的@Test(expected = NotCodeException.class)，@ExpectedException 注解和@Test(expected =……)应该两者选一。
- **@Repeat**：表示被注解的方法应被重复执行多少次，使用如@Repeat(2)方式指定。
- **@Timed**：表示被注解的方法必须在多长时间内运行完毕，超时将抛出异常，使用如@Timed(millis=10)方式指定，单位为毫秒。注意此处指定的时间是如下方法执行时间之和：测试方法执行时间（或者任何测试方法重复执行时间之和）、@Before 和@After 注解的测试方法之前和之后执行的方法执行时间。而 Junit 4 中的@Test(timeout=2)指定的超时时间只是测试方法执行时间，不包括任何重复等。
- **除了支持如上注解外，还支持【第十二章 零配置】中依赖注入等注解。**

五、TestContext 框架支持类：提供对测试框架的支持，如 Junit、TestNG 测试框架，用于集成 Spring TestContext 和测试框架来简化测试，TestContext 框架提供如下支持类：

- **JUnit 3.8 支持类**：提供对 Spring TestContext 框架与 Junit3.8 测试框架的集成：
  - **AbstractJUnit38SpringContextTests**：我们的测试类继承该类后将获取到 Test Fixture 的依赖注入好处。
  - **AbstractTransactionalJUnit38SpringContextTests**：我们的测试类继承该类后除了能得到 Test Fixture 的依赖注入好处，还额外获取到事务管理支持。
- **JUnit 4.5+支持类**：提供对 Spring TestContext 框架与 Junit4.5+测试框架的集成：
  - **AbstractJUnit4SpringContextTests**：我们的测试类继承该类后将获取到 Test Fixture 的依赖注入好处。
  - **AbstractTransactionalJUnit4SpringContextTests**：我们的测试类继承该类后除了能得到 Test Fixture 的依赖注入好处，还额外获取到事务管理支持。
- **定制 Junit4.5+运行器**：通过定制自己的 Junit4.5+运行器从而无需继承 JUnit 4.5+支持类即可完成需要的功能，如 Test Fixture 的依赖注入、事务管理支持，
  - **@RunWith(SpringJUnit4ClassRunner.class)**：使用该注解注解到测试类

上表示将集成 Spring TestContext 和 Junit 4.5+测试框架。

- **@TestExecutionListeners：**该注解用于指定 TestContext 框架的监听器用于与 TestContext 框架管理器发布的测试执行事件进行交互，TestContext 框架提供如下三个默认的监听器：DependencyInjectionTestExecutionListener、DirtiesContextTestExecutionListener、TransactionalTestExecutionListener 分别完成对 Test Fixture 的依赖注入、@DirtiesContext 支持和事务管理支持，即在默认情况下将自动注册这三个监听器，另外还可以使用如下方式指定监听器：

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsHibernateDaoIntegrationTest {
}
```

如上配置将通过定制的 Junit4.5+运行器运行，但不会完成 Test Fixture 的依赖注入、事务管理等等，如果只需要 Test Fixture 的依赖注入，可以使用@TestExecutionListeners({DependencyInjectionTestExecutionListener.class})指定。

- **TestNG 支持类：**提供对 Spring TestContext 框架与 TestNG 测试框架的集成：
  - **AbstractTestNGSpringContextTests：**我们的测试类继承该类后将获取到 Test Fixture 的依赖注入好处。
  - **AbstractTransactionalTestNGSpringContextTests：**我们的测试类继承该类后除了能得到 Test Fixture 的依赖注入好处，还额外获取到事务管理支持。

到此 Spring TestContext 测试框架减少完毕了，接下来让我们学习一下如何进行集成测试吧。

13.3.3 准备集成测试环境

对于集成测试环境各种配置应该和开发环境或实际生产环境配置相分离，即集成测试时应该使用单独搭建一套独立的测试环境，不应使用开发环境或实际生产环境的配置，从而保证测试环境、开发、生产环境相分离。

1、拷贝一份 Spring 资源配置文件 applicationContext-resources.xml，并命名为 applicationContext-resources-test.xml 表示用于集成测试使用，并修改如下内容：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:resources-test.properties</value>
        </list>
    </property>
</bean>
```

2、

2、拷贝一份替换配置元数据的资源文件（resources/resources.properties），并命名为 resources-test.properties 表示用于集成测试使用，并修改为以下内容：

```
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:point_shop
db.username=sa
db.password=
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.hbm2ddl.auto=create-drop
hibernate.show_sql=false
hibernate.format_sql=true
```

- **jdbc:hsqldb:mem:point_shop**：我们在集成测试时将使用 HSQLDB，并采用内存数据库模式运行；
- **hibernate.hbm2ddl.auto=create-drop**：表示在创建 SessionFactory 时根据 Hibernate 映射配置创建相应 Model 的表结构，并在 SessionFactory 关闭时删除这些表结构。

到此我们测试环境修改完毕，在进行集成测试时一定要保证测试环境、开发环境、实际生产环境相分离，即对于不同的环境使用不同的配置文件。

13.3.4 数据访问层

数据访问层集成测试，同单元测试一样目的不仅测试该层定义的接口实现方法的行为是否正确，而且还要测试是否正确与数据库交互，是否发送并执行了正确的 SQL，SQL 执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层集成测试不再通过 Mock 对象与数据库交互的 API 来完成测试，而是使用实实在在存在的与数据库交互的对象来完成测试。

接下来让我们学习一下如何进行数据访问层集成测试：

1、在 test 文件夹下创建如下测试类：

```
package cn.javass.point.dao.hibernate;
//省略 import
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
               "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(
     transactionManager = "txManager", defaultRollback=false)
public class GoodsHibernateDaoIntegrationTest {
    @Autowired
    private ApplicationContext ctx;
    @Autowired
    private IGoodsCodeDao goodsCodeDao;
}
```

- **@RunWith(SpringJUnit4ClassRunner.class)：**表示使用自己定制的 Junit4.5+运行器来运行测试，即完成 Spring TestContext 框架与 Junit 集成；
- **@ContextConfiguration：**指定要加载的 Spring 配置文件，此处注意我们的 Spring 资源配置文件为“applicationContext-resources-test.xml”；
- **@TransactionConfiguration：**开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired：**完成 Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下分页查询所有已发布的商品是否满足需求：

```
@Transactional
@Rollback
@Test
public void testListAllPublishedSuccess() {
      GoodsModel goods = new GoodsModel();
      goods.setDeleted(false);
      goods.setDescription("");
      goods.setName("测试商品");
      goods.setPublished(true);
      goodsDao.save(goods);
      Assert.assertTrue(goodsDao.listAllPublished(1).size() == 1);
      Assert.assertTrue(goodsDao.listAllPublished(2).size() == 0);
}
```

- **@Rollback**：表示替换@ContextConfiguration 指定的默认事务回滚行为，即将在测试方法执行完毕时回滚事务。

数据访问层的集成测试也是非常简单，与数据访问层的单元测试类似，也应该只对复杂的数据访问层代码进行测试。

13.3.5 业务逻辑层

业务逻辑层集成测试，目的同样是测试该层的业务逻辑是否正确，对于数据访问层实现通过 Spring IoC 容器完成装配，即使用真实的数据访问层实现来获取相应的底层数据。

接下来让我们学习一下如何进行业务逻辑层集成测试：

1、在 test 文件夹下创建如下测试类：

```
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
               "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
               "classpath:cn/javass/point/service/applicationContext-service.xml"})
@TransactionConfiguration(
transactionManager = "txManager", defaultRollback=false)
public class GoodsCodeServiceImplIntegrationTest extends
    AbstractJUnit4SpringContextTests {
    @Autowired
    private IGoodsCodeService goodsCodeService;
    @Autowired
    private IGoodsService goodsService;
}
```

- **AbstractJUnit4SpringContextTests**：表示将 Spring TestContext 框架与 Junit4.5+测试框架集成；
- **@ContextConfiguration**：指定要加载的 Spring 配置文件，此处注意我们的 Spring 资源配置文件为“applicationContext-resources-test.xml”；
- **@TransactionConfiguration**：开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired**：完成 Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下购买商品 Code 码是否满足需求：

2.1、测试购买失败的场景：

```
@Transactional
@Rollback
@ExpectedException(NotCodeException.class)
@Test
public void testBuyFail() {
    goodsCodeService.buy("test", 1);
}
```

由于我们数据库中没有相应商品的Code码，因此将抛出NotCodeException异常。

2.2、测试购买成功的场景：

```
@Transactional
@Rollback
@Test
public void testBuySuccess() {
    //1.添加商品
    GoodsModel goods = new GoodsModel();
    goods.setDeleted(false);
    goods.setDescription("");
    goods.setName("测试商品");
    goods.setPublished(true);
    goodsService.save(goods);

    //2.添加商品Code码
    GoodsCodeModel goodsCode = new GoodsCodeModel();
    goodsCode.setGoods(goods);
    goodsCode.setCode("test");
    goodsCodeService.save(goodsCode);
    //3.测试购买商品Code码
    GoodsCodeModel resultGoodsCode = goodsCodeService.buy("test", 1);
    Assert.assertEquals(goodsCode.getId(), resultGoodsCode.getId());
}
```

由于我们添加了指定商品的Code码因此购买将成功，如果失败说明业务写错了，应该重写。

业务逻辑层的集成测试也是非常简单，与业务逻辑层的单元测试类似，也应该只对复杂的业务逻辑层代码进行测试。

13.3.5 表现层

对于表现层集成测试，同样类似于单元测试，但对于业务逻辑层都将使用真实的实现，而不再是通过 Mock 对象来测试，这也是集成测试和单元测试的区别。

接下来让我们学习一下如何进行表现层 Action 集成测试：

1、准备 Struts 提供的 junit 插件，到 struts-2.2.1.1.zip 中拷贝如下 jar 包到类路径：

```
lib\struts2-junit-plugin-2.2.1.1.jar
```

2、测试支持类：Struts2 提供 StrutsSpringTestCase 测试支持类，我们所有的 Action 测试类都需要继承该类；

2、准备 Spring 配置文件：由于我们的测试类继承 StrutsSpringTestCase 且将通过覆盖该类的 getContextLocations 方法来指定 Spring 配置文件，但由于 getContextLocations 方法只能返回一个配置文件，因此我们需要新建一个用于导入其他 Spring 配置文件的配置文件 applicationContext-test.xml，具体内容如下：

```
<import resource="classpath:applicationContext-resources-test.xml"/>
<import resource="classpath:cn/javass/point/dao/applicationContext-hibernate.xml"/>
<import resource="classpath:cn/javass/point/service/applicationContext-service.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-admin-servlet.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-front-servlet.xml"/>
```

3、在 test 文件夹下创建如下测试类：

```
package cn.javass.point.web.front;
//省略 import
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsActionIntegrationTest extends StrutsSpringTestCase {
    @Override
    protected String getContextLocations() {
        return "classpath:applicationContext-test.xml";
    }
    @Before
    public void setUp() throws Exception {
        //1  指定 Struts2 配置文件
        //该方式等价于通过 web.xml 中的<init-param>方式指定参数
        Map<String, String> dispatcherInitParams = new HashMap<String, String>();
        ReflectionTestUtils.setField(this, "dispatcherInitParams", dispatcherInitParams);
        //1.1  指定 Struts 配置文件位置
        dispatcherInitParams.put("config",
                                "struts-default.xml,struts-plugin.xml,struts.xml");
        super.setUp();
    }
    @After
```

- **@RunWith(SpringJUnit4ClassRunner.class)**：表示使用自己定制的 Junit4.5+运行器来运行测试，即完成 Spring TestContext 框架与 Junit 集成；
- **@TestExecutionListeners({})**：没有指定任何监听器，即不会自动完成对 Test Fixture 的依赖注入、@DirtiesContext 支持和事务管理支持；
- **StrutsSpringTestCase**：集成测试 Struts2+Spring 时所有集成测试类必须继承该类；
- **setUp 方法**：在每个测试方法之前都执行的初始化方法，其中 dispatcherInitParams 用于指定等价于在 web.xml 中的<init-param>方式指定的参数；必须调用 super.setUp()用于初始化 Struts2 和 Spring 环境。
- **tearDown()**：在每个测试方法之前都执行的销毁方法，必须调用 super.tearDown()来销毁 Spring 容器等。

2、测试支持写完后，接下来测试一下前台购买商品 Code 码是否满足需求：

2.1、测试购买失败的场景：

```
@Test
public void testBuyFail() throws UnsupportedEncodingException, ServletException {
    //2 前台购买商品失败
    //2.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE));
    //2.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction)
        ActionContext.getContext().getActionInvocation().getAction();
    //2.3 验证前台GoodsAction的buy方法有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0);
}
```

- **initServletMockObjects()**：用于重置所有 http 相关对象，如 request 等；
- **request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE))**：用于准备请求参数；
- **executeAction("/goods/buy.action")**：通过模拟 http 请求来调用前台 GoodsAction 的 buy 方法完成商品购买
- **Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0)**：表示执行 Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能

是错误的，需要修改。

2.2、测试购买成功的场景：

```
@Test
public void testBuySuccess() throws UnsupportedEncodingException, ServletException {
    //3 后台新增商品
    //3.1 准备请求参数
    request.setParameter("goods.name", "测试商品");
    request.setParameter("goods.description", "测试商品描述");
    request.setParameter("goods.originalPoint", "1");
    request.setParameter("goods.nowPoint", "2");
    request.setParameter("goods.published", "true");
    //3.2 调用后台GoodsAction的add方法完成新增
    executeAction("/admin/goods/add.action");
    //2.3 获取GoodsAction的goods属性
    GoodsModel goods = (GoodsModel) findValueAfterExecute("goods");
    //4 后台新增商品Code码
    //4.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    request.setParameter("codes", "a\rb");
    //4.2 调用后台GoodsCodeAction的add方法完成新增商品Code码
    executeAction("/admin/goodsCode/add.action");
    //5 前台购买商品成功
    //5.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    //5.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction)
        ActionContext.getContext().getActionInvocation().getAction();
    //5.3 验证前台GoodsAction的buy方法没有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0);
}
```

- **executeAction("/admin/goods/add.action")：**调用后台 GoodsAction 的 add 方法，用于新增商品；
- **executeAction("/admin/goodsCode/add.action")：**调用后台 GoodCodeAction 的 add 方法用于新增商品 Code 码；

- **executeAction("/goods/buy.action")：**调用前台 GoodsAction 的 buy 方法，用于购买相应商品，其中 Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0)表示购买成功，即 Action 动作正确。

表现层 Action 集成测试介绍就到此为止，如何深入 StrutsSpringTestCase 来完成集成测试已超出本书范围，如果读者对这部分感兴趣可以到 Struts2 官网学习最新的测试技巧。

第十四章　IoC 扩展

第十五章　Spring 的 AOP 的接口

第十六章　深入理解 Spring 的 AOP

第十七章　校验和数据绑定

第十八章　Spring 的 MVC

第十九章　综合示例

第二十章　集成视图技术

第二十一章　集成其他 ORM

第二十一章　远程访问

第二十二章　集成 EJB

第二十三章　集成 Web 服务

第二十四章　集成 JMS

第二十五章　集成 JavaMail

第二十六章　集成 JMX

第二十七章　定时调度和线程池

第二十八章　集成动态语言