链滴

Spring 基础篇 - Spring In Action 读书笔记

作者：ZephyrJung
原文链接：https://ld246.com/article/1478847661846
来源网站：链滴

一、 Spring之旅

1. 简化Java开发

为了降低Java开发的复杂性，Spring采取了以下4种关键策略：

- 基于POJO的轻量级和最小入侵性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

2. 依赖注入

通常，每个对象负责管理与自己相互协作的对象（即它所依赖的对象）的引用，这将会导致高度耦合难以测试的代码。如下面的Knight类：

```
package com.springinaction.knights;
public class DaselRescuingKnight implements Knight{
 private RescueDamselQuest quest;
 public DamselRescuingKnight(){//自己管理依赖对象
  quest=new RescueDamselQuest();
 }
 public void embarkOnQuest() throws QuestException{
  quest.embark();
 }
}
```

耦合具有两面性。一方面，紧密耦合的代码难以测试，难以复用，难以理解，并且典型的表现出“打鼠”式的bug特性（修复一个bug，导致出现新的或者更多的bug）。另一方面，一定程度的耦合又必须的——完全没有耦合的代码什么也做不了。为了完成有实际意义的功能，不同的类必须以适当的式进行交互。

另一种方式，通过依赖注入，对象的依赖关系将由负责协调系统中各个对象的第三方组件在创建对象设定。对象无需自行创建或管理它们的依赖关系——依赖关系将被自动注入到需要它们的对象中去。下面的Knight类：

```
package.com.springinaction.knights;
public class BraveKnight implements Knight{
 private Quest quest;
 public BraveKnight(Quest quest){//构造器注入
  this.quest=quest;
 }
 public void embarkOnQuest() throws QuestException{
  quest.embark();
 }
}
```

如此，可以向BraveKinght注入任意一种Quest实现。创建应用组件之间协作的行为通常称为装配。

面是一种XML装配：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
 <bean id="knight" class="com.springinaction.knights.BraveKnight">
  <constructor-arg ref="quest"/>
 </bean>
 <bean id="quest" class="com.springinaction.knights.SlayDragonQuest"/>
</beans>
```

Spring通过应用上下文（Application Context）装在Bean的定义并把它们组装起来。下面创建了一Spring应用上下文：

```
package com.springinaction.knights;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class KnightMain{
 public static void main(String[] args){
  ApplicationContext context=new ClassPathXmlApplicationContext("knights.xml");
  Knight knight=(Knight)context.getBean("knight");
  knight.embarkOnQuest();
 }
}
```

3. 应用切面

系统由许多不同组件组成，每一个组件各负责一块特定的功能。除了实现自身核心的功能之外，这些件还经常承担着额外的职责。诸如之日、事务管理和安全此类的系统服务经常融入到自身核心业务逻的组件中去，这些系统服务通常被称为横切关注点，因为它们总是跨越系统的多个组件。

AOP使这些服务模块化，并以声明的方式将它们应用到它们需要影响的组件中去。结果是这些组件具更高内聚性以及更加关注自身业务，完全不需要了解可能涉及的系统服务的复杂性。如下图所示：

image

下面有个吟游诗人类以记载骑士的所有事迹：

```
package com.springinaction.knights;
public class Minstrel{
 public void singBeforeQuest(){
  System.out.println("Fa la la; The knight is so brave!");
 }
 public void singAfterQuest(){
  System.out.println("Tee hee he; The brave knight did embark on a quest!");
 }
}
```

为了使吟游诗人工作，传统代码如下：

```
package com.springinaction.knights;
public class BraveKnight implements Knight{
```

```
 private Quest quest;
 private Minstrel minstrel;
 public BraveKnight(Quest quest,Minstrel minstrel){
  this.quest=quest;
  this.minstrel=minstrel;
 }
 public void embarkOnQuest() throws QuestException{
  minstrel.singBeforeQuest();
  quest.embark();
  minstrel.singAfterQuest();
 }
}
```

然而，吟游诗人这种通用功能分散到多个组件中，将给代码引入双重复杂性：

- 遍布系统的关注点实现代码将会重复出现在多个组件中。这意味如果要改变这些关注点的逻辑，你须修改各个模块的相关实现。即使你把这些关注点抽象为一个独立的模块，其他模块只是调用它的方，但方法的调用还是重复出现在各个模块中；
- 你的组件会因为那些与自身核心业务无关的代码儿变得混乱。

把Minstrel抽象为一个切面，只需要在一个Spring配置文件中声明它：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xmlns:aop="..." xsi:schemaLocation="...">
 <bean id="knight" class="com.springinaction.knights.BraveKnight">
  <constructor-arg ref="quest"/>
 </bean>
 <bean id="quest" class="com.springinaction.knights.SlayDragonQuest"/>
 <bean id="minstrel" class="com.springinaction.knights.Minstrel"/>
 <aop:config>
  <aop:aspect ref="minstreal">
   <aop:pointcut id="embark" expression="execution(* *.embarkOnQuest(..))"/>
   <aop:before pointcut-ref="embark" method="singBeforeQuest"/>
   <aop:after pointcut-ref="embark" method="singAfterQuest"/>
  </aop:aspect>
 </aop:config>
</beans>
```

4. Spring容器

在基于Spring的应用中，应用对象生存于Spring容器中，Spring容器创建对象，装配它们，配置它们管理它们的整个生命周期，从生存到死亡（或者从创建到销毁）。Spring自带两类容器实现：Bean厂及应用上下文，后者更为常用，且分为以下三种：

- ClassPathXmlApplicationContext：从类路径下的XML配置文件中加载上下文定义，把应用上下定义文件当作类资源。
- FileSystemXmlApplicationContext：读取文件系统下的XML配置文件并加载上下文定义。
- XmlWebApplicationContext：读取Web应用下的XML配置文件并装载上下文定义。

通过现有的应用上下文引用，可以调用上下文的getBean()方法从Spring容器中获取Bean。

二、 装配Bean

1. 声明Bean

假设有一个选秀比赛，有各种各样的人上台表演。定义一个表演者接口如下：

```
package com.springinaction.springidol;
public interface Performer{
 void perform() throws PerformanceException;
}
```

创建如下一个表演者Bean，杂技演员：

```
package com.springinaction.springidol;
public class Juggler implements Performer{
 private int beanBags=3;
 public Juggler(){}
 public Juggler(int beanBags){
  this.beanBags=beanBags;
 }
 public void perform() throws PerformanceException{
  System.out.println("JUGGLING "+beanBags+" BEANBAGS");
 }
}
```

Spring配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
 <bean id="duke" class="com.springinaction.springidol.Juggler">
  <!--构造器注入属性值-->
  <constructor-arg value="15"/>
 <bean>
</beans>
```

如此，可以通过上下文获取到这个Bean并调用其perform方法。

假设一个Bean中引用了另一个bean，比如一个会朗诵诗歌的杂技演员，定义如下：

```
package com.springinaction.springidol;
public class PoeticJuggler extends Juggler{
 private Poem poem;
 public PoeticJuggler(Poem poem){
  super();
  this.poem=poem;
 }
 public PoeticJuggler(int beanBags,Poem poem){
  super(beanBags);
  this.poem=poem;
 }
 public void perform() throws PerformanceException{
  super.perform();
```

```
  System.out.println("While reciting...");
  poem.recite();
 }
}
```

Poem是一个接口：

```
package com.springinaction.springidol;
public interface Poem{
 void recite();
}
```

Poem的一个实现如下：

```
package com.springinaction.springidol;
public class tangshi implements Poem{
 private static String[] LINES={
  "千里莺啼绿映红，",
  "水村山郭酒旗风。",
  "南朝四百八十寺，",
  "多少楼台烟雨中。"
 };
 public tangshi(){}
 public void recite(){
  for(int i=0;i<LINES.length;i++){
   System.out.println(LINES[i]);
  }
 }
}
```

如此，可以通过在配置文件中将Poem接口的实现注入到Perform实现Bean中：

```
<bean id="poeticDuke" class="com.springinaction.springidol.PoeticJuggler">
 <constructor-arg value="15"/>
 <constructor-arg ref="tangshi"/>
</bean>
```

以上Bean（Juggler、PoeticJuggler）都具备构造方法，上下文可以通过构造方法来实例化一个Bea
，然而若没有公开的构造方法时，如下面的Stage单例类：

```
package com.springinaction.springidol;
public class Stage{
 private Stage(){}
 private static class StageSingletonHolder{
  static Stage instance=new Stage();
 }
 public static Stage getInstance(){
  return StageSingletonHolder.instance;
 }
}
```

可以在配置文件中，通过factory-method属性，调用一个指定的静态方法，从而代替构造方法来创

一个类的实例：

```
<bean id="theStage" class="com.springinaction.springidol.Stage" factory-method="getInsta
ce"/>
```

2. Bean作用域

所有的Spring Bean默认都是单例。当容器分配一个Bean时（不论通过装配还是调用getBean()方法，它总是返回Bean的同一个实例。

在配置文件中，可以通过指定scope属性的值来设置Bean的作用域：

作用域　定义

singleton　在每一个Spring容器中，一个Bean定义只有一个对象实例(默认)

prototype　允许Bean的定义可以被实例化任意次（每次调用都创建一个实例）

request　在一次HTTP请求中，每个Bean定义对应一个实例

session　在一个HTTP Session中，每个Bean定义对应一个实例

global-session　在一个全局HTTP Session中，每个Bean定义对应一个实例。

3. 初始化和销毁Bean

为Bean定义初始化和销毁操作，只需要使用init-method和destroy-method参数来配置<bean>元。假设有一个Auditoium类，有turnOnLights()和turnOffLights()两种方法，可以进行如下配置：

```
<bean id="auditorium" class="com.springinaction.springidol.Auditorium"
 init-method="turnOnLights" destory-method="turnOffLights"/>
```

如果很多Bean都拥有相同名字的初始化和销毁方法，可以定义一个默认公共属性：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..."
default-init-method="commonInit"
default-destory-method="commonDestory">
...
</beans>
```

4. 注入Bean属性

之前是构造器注入，接下来演示另一种注入方法，setter方法注入：

```
//定义表演者
package com.springinaction.springidol;
public class Instrumentalist implements Performer{
 public Instrumentalist(){}
 public void perform() throws PerformanceException{
  System.out.print("Playing "+song+" : ");
  instrument.play();
 }
 private String song;
 public void setSong(String song){
```

```
  this.song=song;
 }
 public String getSong(){
  return song;
 }
 public String screamSong(){
  return song;
 }
 private Instrument instrument;
 public void setInstrument(Instrument instrument){
  this.instrument=instrument;
 }
}
//声明乐器接口
package com.springinaction.springidol;
public interface Instrument{
 public void play();
}
//实现一个乐器
package com.springinaction.springidol;
public class Saxophone implements Instrument{
 public Saxophone(){}
 public void play(){
  System.out.println("TOOT TOOT TOOT");
 }
}
```

可以通过如下配置进行setter注入：

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist">
 <property name="song" value="Jingle Bells"/>
 <property name="instrument" value="saxophone"/>
</bean>
<bean id="saxophone" class="com.springinaction.springidol.Saxophone"/>
```

可以将saxophone声明为内部bean，使得其被kenny独有：

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist">
 <property name="song" value="Jingle Bells"/>
 <proeprty name="instrument">
  <bean class="com.springinaction.springidol.Saxophone"/>
 </property>
</bean>
```

构造器注入也有类似上述写法。

5. 使用Spring的命名空间p装配属性

只是写法上的不同，在Spring的XML配置前加上特定的声明即可：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:p="http://www.springframework.org/schema/p"
 xmlns:xsi="..." xsi:schemaLocation="...">
```

```
 <bean id="kenny" class="com.springinaction.springidol.Instrumentalist"
  p:song="Jingle Bells"
  p:instrument-ref="saxophone"/>
<beans>
```

6. 装配集合

Spring提供了4种类型的集合配置元素

<list>：装配list类型的值，允许重复

<set>：装配set类型的值，不允许重复

<map>：装配map类型的值，名称和值可以是任意类型

<props>：装配properties类型的值，名称和值必须都是String型

假设有如下表演者定义：

```
package com.springinaction.springidol;
import java.util.Collection;
public class OneManBand implements Performer{
 public OneManBand(){}
 public void perform() throws PerformanceException{
  for(Instrument instrument:instruments){
   instrument.play();
  }
 }
 private Collection<Instrument> instruments;
 public void setInstruments(Collection<Instrument> instruments){
  this.instruments=instruments;
 }
}
```

装配List、Set和Array：

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
 <property name="instruments">
  <list>
   <ref bean="guitar"/>
   <ref bean="cymbal"/>
   <ref bean="harmonica"/>
  </list>
 </proeprty>
</bean>
```

<list>可以包含另一个<list>作为其成员形成多维列表。如果Bean的属性为数组类型或java.util.Collection接口的任意实现，都可以使用<list>元素。<set>装配如下：

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
 <proeprty name="instruments">
  <set>
   <ref bean="guitar"/>
```

```
  <ref bean="cymbal"/>
  <ref bean="harmonica"/>
 </set>
</property>
</bean>
```

装配Map集合：

修改OneManBand类如下：

```
package com.springinaction.springidol;
import java.util.Map;
import com.springinaction.springidol.Instrument;
import com.springinaction.springidol.PerformanceException;
import com.springinaction.springidol.Performer;
public class OneManBand implements Performer{
 public OneManBand(){}
 public void perform() throws PerformanceException{
  for(String key:instruments.keySet()){
   System.out.print(key+" : ");
   Instrument instrument=instruments.get(key);
   instrument.play();
  }
 }
 private Map<String,Instrument> instruments;
 public void setInstruments(Map<String,Instrument> instruments){
  this.instruments=instruments;
 }
}
```

配置如下:

```
<bean id="hank" class="com.springinaction.springido.OneManBand">
 <property name="instruments">
  <map>
   <entry key="GUITAR" value-ref="guitar"/>
   <entry key="CYMBAL" value-ref="cymbal"/>
   <entry key="HARMONICA" value-ref="harmonica"/>
  </map>
 </property>
</bean>
```

键可能是其他bean的引用，此时可用属性key-ref指定，值可能为简单值，可用value指定。

装配Properties集合：

如果所配置的Map的每一个entry键和值都为String类型时，可以考虑使用java.util.Properties代替。功能和Map大致相同，但是限定键值必须为String类型。修改instruments属性如下：

```
private Properties instruments;
public void setInstruments(Properties instruments){
```

this.instruments=instruments;

}

配置如下：

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
 <property name="instruments">
  <props>
   <prop key="GUITAR">STRUM STRUM STRUM</prop>
   <prop key="CYMBAL">CRASH CRASH CRASH</prop>
   <prop key="HARMONICA">HUM HUM HUM</prop>
  </props>
 </property>
</bean>
```

装配空值：

使用<null/>元素

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

7. SpEL表达式

SpEL是一种强大、简洁的装配Bean的方式，它通过运行期执行的表达式将值装配到Bean的属性或构器参数中。

字面值：

```
<property name="count" value="#{5}"/>
<!--可以与非SpEL表达式混用-->
<property name="message" value="The value is #{5}"/>
<!--使用浮点数-->
<proeprty name="frequency" value="#{89.7"/>
<!--使用科学计数法-->
<property name="capacity" value="#{1e4}"/>
<!--使用字符串-->
<proeprty name="name" value="#{'Chuck'}"/>
<!--使用布尔值-->
<property name="enabled" value="#{false}"/>
```

引用Bean、Properties和方法：

```
<!--等价于ref="saxophone"-->
<proeprty name="instrument" value="#{saxophone}"/>
<!--获取Bean属性-->
<property name="song" value="#{kenny.song}"/>
<!--调用Bean的某个方法-->
<property name="song" value="#{songSelector.selectSong()}"/>
<!--可以对方法的返回值再次调用一个方法（应该是类型相关）-->
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
<!--使用null-safe存取器保证SpEL表达式值不为NULL-->
```

```
<property name="song" value="#{songSelector.selectSong()?.toUpperCase()}"/>
```

装配指定类的静态方法和常量：

```
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>
<property name="randomNumber" value="#{T{java.lang.Math).random()}"/>
```

8. 在SpEL值上执行操作

- 算术运算：+、-、*、/、%、^
- 关系运算：<、>、==、<=、>=、lt、gt、eq、le、ge
- 逻辑运算：and、or、not、|
- 条件运算：?:
- 正则表达式：matches

数值运算：

```
<!--使用+加运算符-->
<property name="adjustedAmount" value="#{counter.total+42}"/>
<!--使用-减运算符-->
<property name="adjustedAmount" vlaue="#{counter.total-20}"/>
<!--使用*乘运算符-->
<property name="circumference" value="#{2*T(java.lang.Math).PI*circle.radius}"/>
<!--使用%求余运算符-->
<property name="remainder" value="#{counter.total%counter.count}"/>
<!--使用^乘方运算符-->
<property name="area" value="#{T(java.lang.Math).PI*circle.radius^2}"/>
<!--使用+字符串连接-->
<property name="fullName" value="#{performer.firstName+' '+performer.lastName}"/>
```

比较值：

可以使用符号，也可以使用文本替代，因为XML中小于等于和大于等于符号有特殊含义，故而推荐后：

```
<property name="equal" value="#{counter.total==100}"/>
<property name="hasCapacity" value="#{counter.total le 100000}"/>
```

逻辑表达式：

```
<proeprty name="largeCircle" value="#{shape.kind=='circle' and shape.perimeter gt 10000}"
>
<property name="outOfStock" value="#{!product.available}"/>
<property name="outOfStock" vlaue="#{not product.available}"/>
```

条件表达式：

```
<property name="song" value="#{kenny.song!=null?kenny.song:'Greensleeves'}"/>
<!--判断是否为NULL时，可以省略中间的选项-->
<property name="song" value="#{kenny.song ?: 'Greensleeves'}"/>
```

正则表达式：

```
<property name="validEmail" value="#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-z0-9-]+\\.com'}"/>
```

9. 在SpEL中筛选集合

假设有如下City类（省略setter和getter方法）

```
package com.habuma.spel.cities;
public class City{
 private String name;
 private String state;
 private int population;
}
```

在Spring配置文件中通过util:list元素定义一个City的List集合如下：

```
<util:list id="cities">
 <bean class="com.habuma.spel.cities.City"
  p:name="Chicago" p:state="IL" p:population="2853114"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Atlanta" p:state="GA" p:population="537958"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Dallas" p:state="IL" p:population="1279910"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Houston" p:state="IL" p:population="2242193"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Odessa" p:state="IL" p:population="90943"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="El Paso" p:state="IL" p:population="613190"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Jal" p:state="IL" p:population="1996"/>
 <bean class="com.habuma.spel.cities.City"
  p:name="Las Cruces" p:state="IL" p:population="91865"/>
</util:list>
```

访问集合成员：

[]运算符也可以用来获取java.util.Map集合中的成员，假设City对象以其名字作为键放入Map集合中：

```
<property name="chosencity" value="#{cities['Dallas']}"/>
```

[]运算符的另一种用法是从java.util.Properties集合中获取值，如通过util:properties元素在Spring中在一个properties配置文件如下：

```
<util:properties id="setttings" location="classpath:settings.properties"/>
```

访问属性如下：

```
<property name="accessToken" value="#{settings['twitter.accessToken']}"/>
```

Spring提供了两种特殊的选择属性方式：SystemEnvironment和SystemProperties，这两个都是一

properties。

[]运算符同样可以通过索引来得到字符串的某个字符，如'This is a test'[3]将返回”s“。

使用.?[]查询运算符：

```
<property name="bigCities" value="#{cities.?[population gt 100000]}"/>
```

.^[]查询第一个匹配项：

```
<property name="aBigCity" value="#{cities.^[population gt 100000]}"/>
```

.$[]查询最后一个匹配项：

```
<property name="aBigCity" value="#{cities.$[population gt 100000]}"/>
```

.![]投影运算符：

```
<property name="cityNames" value="#{cities}.![name]}"/>
<property name="cityNames" value="#{cities.![name+', '+state]}"/>
<property name="cityNames" value="#{cities.?[population gt 100000].![name+', '+state]}"/>
```

三、 最小化Spring XML配置

1. 自动装配Bean属性

Spring提供了4种自动装配策略：

● byName：把与Bean的属性具有相同名字（或者ID）的其他Bean自动装配到Bean的对应属性中。果没有，则该属性不装配。

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist" autowire="byName
>
 <property name="song" value="Jingle Bells"/>
</bean>
<bean id="instrument" class="com.springinaction.springidol.Saxophone"/>
```

如此，kenny中的instrument属性可以自动将id为instrument的bean装配进来。

● byType：把与Bean的属性具有相同类型的其他Bean自动装配到Bean的对应属性中。如果没有，该属性不装配。

与byName类似，只不过autowire属性值设置为"byType"。如果Spring寻找到多个Bean，它们的类与需要自动装配的属性的类型都相匹配，则Spring将抛出异常。如此，可以设置其中一个为首选Bea，默认都是首选，故反向用之，设置为非首选。也可以将其他Bean设置为非候选：

```
<bean id="saxophone" class="com.springinaction.springidol.Saxophone" primary="false"/>
<bean id="saxophone" class="com.springinaction.springidol.Saxophone" autowire-candidat
="false"/>
```

● constructor：把与Bean的构造器入参具有相同类型的其他Bean自动装配到Bean构造器的对应参。

类似于byType，只不过是根据构造函数的入参类型进行匹配：

```
<bean id="duke" class="com.springinaction.springidol.PoteicJugger" autowire="constructor"
>
```

- autodetect：首先尝试使用constructor进行自动装配，如果失败，再尝试使用byType进行自动装配

2. 默认自动装配及混合装配

如果要为Spring应用上下文中的每一个Bean或大多数配置相同的autowire属性，那么可以设置默认动装配策略：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..." default-autowire="byType">
</beans>
```

可以混合使用自动装配和手工装配，手动装配将覆盖自动装配，如此可以解决byType可能产生的装不确定性问题。

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist" autowire="byType">
 <property name="song" value="Jingle Bells"/>
 <property name="instrument" ref="saxophone"/>
</bean>
```

不能混合使用constructor自动装配策略和<constructor-arg>元素。

3. 使用注解装配

Spring容器默认禁用注解装配。启用需要在Spring配置中，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xmlns:context="..." xsi:schemaLocation="...">
 <context:annotation-config/>
</beans>
```

有如下三种注解：

- Spring自带的@Autowired注解；
- JSR-330的@Inject注解；
- JSR-250的@Resource注解。

@Autowired

可以对setter方法进行注解：

也可以标注需要自动装配Bean引用的任意方法：

```
@Autowired
public void heresYourInstrument(Instrument instrument){
 this.instrument=instrument;
}
```

甚至可以标注构造器：

```
@Autowired
```

```
public Instrumentalist(Instrument instrument){
 this.instrument=instrument;
}
```

此时，即便配置文件中没有使用<constructor-arg>配置Bean，该构造器也需要进行自动装配。

还可以直接标注属性，并删除setter方法：

```
@Autowired
private Instrument instrument;
```

应用中必须只能有一个Bean适合装配到@Autowired注解所标注的属性或参数中。否则就会遇到麻烦。

如果属性不一定非要装配，null值也是可以接受的，则可通过设置@Autowired的required属性为fals
来配置自动装配是可选的，如：

```
@Autowired(required=false)
private Instrument instrument;
```

required属性可以用于@Autowired注解所使用的任意地方。但是当使用构造器装配时，只有一个构
器可以将@Autowired的required属性设置为true，其他只能为false。此外，当使用@Autowired标
多个构造器时，Spring就会从所有满足装配条件的构造器中选择入参最多的那个构造器。

也可以对自动装配的Bean添加限定来保证@Autowired能够自动装配，如制定ID为guitar的Bean：

```
@Autowired
@Qualifier("guitar")
private Instrument instrument;
```

除了通过Bean的ID来缩小选择范围，也可通过在Bean上直接使用qualifier来缩小范围：

```
<bean class="com.springinaction.springidol.Guitar">
 <qualifier value="stringed"/>
</bean>
```

又或者在类中添加限定注解如下：

```
@Qualifier("stringed")
public class Guitar implements Instrument{
 ...
}
```

可以创建一个自定义的限定器，如此只有添加了该自定义限定的Bean类才能被装配：

```
package com.springinaction.sprngidol.qualifiers;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.beans.factory.annotation.Qualifier;
@Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument{}
```

变名即可定义其他注解，如此可以多重限定：

```
@Autowired
@StringedInstrument //限定为弦乐器
@Strummed //并且限定为击打乐器
private Instrument instrument;
```

@Inject

几乎可以完全替代@Autowired注解，但没有required属性。

可以要求@Inject注入一个Provider，Provider接口可以实现Bean引用的延迟注入及注入Bean的多实例等。

```
private Set<Knife> knives;
@Inject
public KnifeJuggler(Provider<Knife> knifeProvider){
 knives=new HashSet<Knife>();
 for(int i=0;i<5;i++){
  knives.add(knifeProvider.get());
 }
}
```

类似于@Qualifier，@Inject通过@Named注解限定，使用方法也类似。不同在于，@Qualifier注帮助我们缩小匹配Bean的选择范围（默认使用Bean的ID），而@Named通过Bean的ID来表示可选的Bean。

JSR-330也有@Qualifer注解，但更提倡用该注解创建自定义的限定器注解。方法与上面的完全一样只将Qualifer的声明变为:import javax.inject.Qualifier;

@Value

通过@Value可以直接标注某个属性、方法或者方法参数，并传入一个String类型的表达式来装配属，例如：

```
@Value("Eruption")
private String song;
```

其真正威力在于可以使用SpEL表达式：

```
@Value("#{systemProperties.myFavoriteSong}")
private String song;
```

4. 自动检测Bean

使用context:component-scan除了完成与context:annotation-config一样的工作，还允许Spring动检测Bean和定义Bean：

```
<beans xmlns="..." xmlns:xsi="..." xmlns:context="..." xsi:schemaLocation="...">
<context:component-scan base-package="com.springinaction.springidol">
</beans>
```

使用如下注解为自动检测标注Bean：

- @Component：通用的构造型注解，标识该类为Spring组件。
- @Controller：标识将该类定义为Spring MVC controller。
- @Repository：标识将该类定义为数据仓库
- @Service：标识将该类定义为服务

使用@Component标注的Bean，默认ID为小写类名，可以显示指定ID如：@Component("eddie")

过滤组件扫描：

```
<context:component-scan base-package="com.springinaction.springidol">
 <context:include-filter type="assignable" expression="com.springinaction.springidol.Instrum
nt"/>
</context:component-scan>
```

上述配置实现了自动注册所有的Instrument实现类。过滤器类型如下：

- annotation：扫描使用指定注解所标注的类，通过expression属性指定要扫描的注解
- assignable：扫描派生于expression属性所指定类型的类
- aspectj：扫描与expression属性所指定的AspectJ表达式所匹配的类
- custom：使用自定义的org.springframework.core.type.TypeFilter实现类，该类由expression属指定
- regex：扫描类的名称与expression属性所指定的正则表达式所匹配的类

除了使用context:include-filter告知context:component-scan哪些类需要注册为Spring Bean以外还可以使用context:exclude-filter来告知context:component-scan哪些类不需要注册为Spring Bea
。

5. 使用Spring基于Java的配置

首先需要少量的XML启用Java配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xmlns:context="..." xsi:schemaLocation="...">
<context:component-scan base-package="com.springination.springidol"/>
</beans>
```

也即自动注册相关Bean，作为配置的Java类使用@Configuration标注，其通过该配置被扫描到。

被@Configuration注解的Java类就等价于XML配置中的<beans>：

```
package com.springinaction.springidol;
import org.springframework.context.annotation.Configuration;
@Configuration
public class SpringIdolConfig{
 @Bean
 public Performer duke(){ //方法名即Bean的ID
  return new Juggler();
 }
 @Bean
 public Performer duke15(){
```

```
  return new Juggler(15); //类似于使用<constructor-arg>装配参数
 }
 @Bean
 public Performer kenny(){
  Instrumentalist kenny=new Instrumentalist();
  kenny.setSong("Jingle Bells"); //setter注入
  return kenny;
 }
 @Bean
 private Poem sonnet29(){
  return new Sonnet29();
 }
 @Bean
 public Performer poeticDuke(){
  return new PoeticJuggler(sonnect29()); //装配另一个Bean的引用
 }
}
```

四、 面向切面的Spring

1. 定义AOP术语

通知（Advice）：

通知定义了切面是什么以及何时使用。Spring切面可以应用5种类型的通知

- Before：在方法被调用之前调用通知
- After：在方法完成之后调用通知，无论方法执行是否成功
- After-returning：在方法成功执行之后调用通知
- After-throwwing：在方法抛出异常后调用通知
- Around：在方法调用之前和之后执行通知自定义的行为。

连接点（Joinpoint）：

连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

切点（Poincut）：

切点的定义会匹配通知所要织入的一个或多个连接点。通常使用明确的类和方法名称来指定这些切点或是利用正则表达式定义匹配的类和方法名称模式来指定这些切点。

切面（Aspect）：

切面是通知和切点的结合。通知和切点共同定义了关于切面的全部内容。

引入（Introduction）：

引入允许我们向现有的类添加新方法或属性。

织入（Weaving）：

织入是将切面应用到目标对象来创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中在目标对象的生命周期里有多个点可以进行织入。

Spring提供了4种各具特色的AOP支持：

- 基于代理的经典AOP；
- @AspectJ注解驱动的切面；
- 纯POJO切面；
- 注入式AspectJ切面（适合Spring各版本）

Spring所创建的通知都是用标准的Java类编写的，定义通知所应用的切点通常在Spring配置文件里采XML来编写的。通过在代理类中包裹切面，Spring在运行期将切面织入到Spring管理的Bean中。Sprig只支持方法连接点。

2. 使用切点选择连接点

在Spring AOP中，需要使用AspectJ的切点表达式语言来定义切点：

- arg()：限制连接点匹配参数为指定类型的执行方法
- @arg()：限制连接点匹配参数由指定注解标注的执行方法
- execution()：用于匹配是连接点的执行方法
- this()：限制连接点匹配AOP代理的Bean引用为指定类型的类
- target()：限制连接点匹配目标对象为指定类型的类
- @target()：限制连接点匹配特定的执行对象，这些对象对应的类要具备指定类型的注解
- within()：限制连接点匹配指定的类型
- @within()：限制连接点匹配指定注解所标注的类型
- @annotation：限制匹配带有指定注解连接点

编写切点：

下面的切点表达式表示当Instrument的play()方法执行时会触发通知：

execution(* com.springinaction.springidol.Instrument.play(..))
&&within(com.springinaction.springidol.*)

&&操作符后面的within限定了匹配范围，添加and条件。还可以使用||操作符标识or关系，！操作符识非操作。在基于XML配置来描述切点时，可以使用and、or和not代替。

使用Spring的bean()指示器：

在执行Instrument的play()方法应用通知，但限定Bean的ID为eddie，可以定义切点如下：

execution(* com.springinaction.springidol.Instrument.play()) and bean(eddie)

还可以定义为除了eddie的Bean：

execution(* com.springinaction.springidol.Instrument.play()) and !bean(eddie)

3. 在XML中声明切面

- aop:advisor：定义AOP通知器
- aop:after：定义AOP后置通知（不管被通知的方法是否执行成功）
- aop:after-returning：定义成功执行后通知
- aop:after-throwing：定义执行失败后通知
- aop:around：定义AOP环绕通知
- aop:aspect：定义切面
- aop:aspectj-autoproxy：启用@AspectJ注解驱动的切面
- aop:before：定义AOP前置通知
- aop:config：顶层AOP配置元素
- aop:declare-parents：为被通知的对象引入额外的接口，并透明地实现
- aop:pointcut：定义切点

定义如下观众类：

```
package com.springinaction.springidol;
public class Audience{
 public void takeSeats(){
  System.out.println("The audience is taking their seats.");
 }
 public void turnOffCellPhones(){
  System.out.println("The audience is turning off their cellphones");
 }
 public void applaud(){
  System.out.println("CLAP CLAP CLAP CLAP CLAP");
 }
 public void demandRefund(){
  System.out.println("Boo! We want our money back!");
 }
}
```

在配置文件中将该类配置为一个Bean：

```
<bean id="audience" class="com.springinaction.springidol.Audience"/>
```

声明前置和后置通知：

```
<aop:config>
 <aop:aspect ref="audience">
  <aop:before pointcut="execution(* com.springinaction.springidol.Performer.perform(..)) me
hod="takeSeats"/>
  <aop:before pointcut="execution(* com.springinaction.springidol.Performer.perform(..))" me
hod="turnOffCellPhones"/>
  <aop:after-returning pointcut="execution(* com.springinaction.springidol.Performer.perfor
(..))" method="applaud"/>
```

```
  <aop:after-throwing pointcut="execution(* com.springinaction.springidol.Performer.perform
..))" method="demandRefund"/>
 </aop:aspect>
</aop:config>
```

可以定义命名切点，消除冗余的切点定义：

```
<aop:config>
 <aop:aspect ref="audience">
  <aop:pointcut id="performance" expression="execution(* com.springinaction.springidol.Per
ormer.perform(..))"/>
  <aop:before pointcut-ref="performance" method="takeSeats"/>
  <aop:before pointcut-ref="performance" method="turnOffCellPhones"/>
  <aop:after-returning pointcut-ref="performance" method="applaud"/>
  <aop:after-throwing pointcut-ref="performance" method="demandRefund"/>
 </aop:aspect>
</aop:config>
```

声明环绕通知：

使用环绕通知，可以不借助成员变量在前置通知和后置通知之间共享信息。环绕通知的方法如下：

```
public void watchPerformance(ProceedingPoint joinpoint){
 try{
  System.out.println("The audience is taking their seats.");
  System.out.println("The audience is turning off their cellphones");
  long start=System.currentTimeMillis();
  joinpoint.proceed();
  long end=System.currentTimeMillis();
  System.out.println("CLAP CLAP CLAP CLAP CLAP");
  System.out.println("The performance took "+(end-start)+" milliseconds.");
 }catch(Throwable t){
  System.out.println("Boo! We want our money back!");
 }
}
```

环绕通知声明如下：

```
<aop:config>
 <aop:aspect ref="audience">
  <aop:pointcut id="performance2" expression="execution(* com.springinaction.springidol.Pe
former.perform(..))"/>
  <aop:around pointcut-ref="performance2" method="watchPerformance()"/>
 </aop:aspect>
</aop:config>
```

为通知传递参数：

假设有个读心者，由MindReader接口定义：

```
package com.springinaction.springidol;
```

```
public interface MindReader{
 //截听志愿者的内心感应
 void interceptThoughts(String thoughts);
 //显示内心思想
 String getThoughts();
}
```

有如下实现类：

```
package com.springinaction.springidol;
public class Magician implements MindReader{
 private String thoughts;
 public void interceptThoughts(String thoughts){
  System.out.println("Intercepting volunteer's thoughts");
  this.thoughts=thoughts;
 }
 public String getThoughts(){
  return thoughts;
 }
}
```

定义思想者接口如下：

```
package com.springinaction.springidol;
public interface Thinker{
 void thinkOfSomething(String thoughts);
}
```

思想者实现类如下：

```
package com.springinaction.springidol;
public class Volunteer implements Thinker{
 private String thoughts;
 public void thinkOfSomething(String thoughts){
  this.thoughts=thoughts;
 }
 public String getThoughts(){
  return thoughts;
 }
}
```

配置如下：

```
<aop:config>
 <aop:aspect ref="magician">
  <aop:pointcut id="thinking" expression="execution(* com.springinaction.springidol.Thinker.
hinkOfSomething(String)) and args(thoughts)"/>
  <aop:before pointcut-ref="thinking" method="interceptThoughts" arg-names="thoughs"/>
 </aop:aspect>
</aop:config>
```

通过切面引入新功能：

```
<aop:aspect>
 <aop:declare-parents types-matching="com.springinaction.springidol.Performer+"
  implement-interface="com.springinaction.springidol.Contestant"
  default-impl="com.springinaction.springidol.GraciousContestant"/>
</aop:aspect>

<aop:aspect>
 <aop:declare-parents types-matching="com.springinaction.springidol.Performer+"
  implement-interface="com.springinaction.springidol.Contestant"
  delegate-ref="contestantDelegate"/>
</aop:aspect>
```

4. 注解切面

创建新的Audience类如下，通过注解将其标注为一个切面：

```
package com.springinaction.sprngidol;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class Audience{
    //@Pointcut注解用于定义一个在@AspectJ切面内可重用的切点
    @Pointcut("execution(* com.springinaction.springidol.Performer.perform(..))")
    public void performance(){}//方法名称即切点名称，该方法无需具体功能
    //@Before标识前置通知方法
    @Before("performance()")
    public void takeSeats(){
        System.out.println("The audience is taking their seats.");
    }
    @Before("performance()")
    public void turnOffCellPhones(){
        System.out.println("The audience is turning off their cellphones;")
    }
    //@AfterReturning标识后置通知方法
    @AfterReturning("performance()")
    public void applaud(){
        System.out.println("CLAP CLAP CLAP CLAP CLAP CLAP");
    }
    //@AfterThrowing标识抛出异常时调用的方法
    @AfterThrowing("performance()")
    public void demandRefund(){
        System.out.println("Boo! We want our money back!");
    }
}
```

该类仍可以像下面一样在Spring中进行装配:

```
<bean id=” audience” class=” com.springinaction.springidol.Audience” />
```

为了让Spring将Audience应用为一个切面，可以添加一个配置元素：aop:aspectj-autoproxy/

为了使用该元素，需要在Spring的配置文件中包含aop命名空间：

```
<beans xmlns=http://www.springframework.org/schema/beans
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="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">
```

注解环绕通知

```
@Around("performance()")
public void watchPerformance(ProceedingJoinPoint joinpoint){
 try{
  System.out.println("The audience is taking their seats.");
  System.out.println("The audience is turning off their cellphones");
  long start=System.currentTimeMillis();
  joinpoint.proceed();
  long end=System.currentTimeMillis();
  System.out.println("CLAP CLAP CLAP CLAP CLAP");
  System.out.println("The performance took "+(end-start)+" milliseconds.");
 }catch(Throwable t){
  System.out.println("Boo! We want our money back!");
 }
}
```

传递参数给所标注的通知

```
package com.springinaction.springidol;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class Magician implements MindReader{
 private String thoughts;
 @Pointcut("execution(* com.springinaction.springidol."+"Thinker.thinkOfSomething(String))
&args(thoughts)")
 public void thinking(String thoughts){}
 @Before("thinking(thoughts)")
 public void interceptThoughts(String thoughts){
  System.out.println("Intercepting volunteer's thoughts : "+thoughts);
  this.thoughts=thoughts;
 }
 public String getThoughts(){
  return thoughts;
 }
}
```

标注引入

```
package com.springinaction.springidol;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Declareparents;
@Aspect
public class ContestantIntroducer{
 @DeclareParents(value="com.springinaction.springidol.Performer+",
defaultImpl=GraciousContestant.class)
 public static Contestant contestant;
}
```

五、 Spring JDBC

1. 配置数据源

使用JNDI数据源：

利用Spring，可以像使用Spring Bean那样配置JNDI中数据源的引用并将其配置到需要的类中：

```
<jee:jndi-lookup id="dataSource" jndi-name="/jdbc/SpitterDS" resource-ref="true"/>
```

其中，jndi-name属性用于指定JNDI中资源的名称。如果应用程序运行在Java应用程序服务器中，需将resource-ref设置为true，这样给定的jndi-name将自动添加java:comp/env/前缀。

使用数据源连接池：

Spring并没有提供数据源连接池实现，可以使用DBCP(http://jakarta.apache.org/commons/dbcp)目。可以如下配置：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
 <property name="url" value="jdbc:hsqldb.hsql://localhost/spitter/spitter"/>
 <property name="username" value="sa"/>
 <property name="password" value=""/>
 <property name="initialSize" value="5"/>
 <property name="maxActive" value="10"/>
</bean>
```

BasicDataSource的池配置属性

- initialSize：池启动时创建的连接数量
- maxActive：同一时间可从池中分配的最多连接数。如果设置为0，表示无限制
- maxIdle：池里不会被释放的最多空闲连接数。如果设置为0，表示无限制
- maxOpenPreparedStatements：在同一时间能够从语句池中分配的预处理语句的最大数量。如果置为0，表示无限制。
- maxWait：在抛出异常之前，池池等待连接回收的最大时间（当没有可用连接时）。如果设置为-1表示无限等待
- minEvictableIdleTimeMillis：连接在池中保持空闲而不被回收的最大时间

- minIdle：在不创建新连接的情况下，池中保持空闲的最小连接数
- poolPreparedStatements：是否对预处理语句进行池管理（布尔值）

基于JDBC驱动的数据源：

Spring提供了两种数据源对象（均位于org.springframework.jdbc.datasource包中）：

- DriverManagerDataSource：在每个连接请求时都会返回一个新建的连接。
- SingleConnectionDataSource：在每个连接请求时都会返回同一个连接。

配置如下：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSour
e">
 <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
 <property name="url" value="jdbc:hsqldb:hsql://localhost/spitter/spitter"/>
 <property name="username" value="sa"/>
 <property name="password" value=""/>
</bean>
```

2. 在Spring中使用JDBC

使用SimpleJdbcTemplate访问数据

在Spring中作如下配置：

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate
>
 <constructor-arg ref="dataSource"/>
</bean>
<bean id="spitterDao" class="com.habuma.spitter.persistence.SimpleJdbcTemplateSpitterDa
">
 <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
```

将jdbcTemplate装配到DAO中并使用它来访问数据库：

```
public class JdbcSpitterDAO implements SpitterDAO{
...
 private SimpleJdbcTemplate jdbcTemplate;
 public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate){
  this.jdbcTemplate=jdbcTemplate;
 }
}
```

添加、查询方法如下：

```
public void addSpitter(Spitter spitter){
 jdbcTemplate.update(SQL_INSERT_SPITTER,
  spitter.getUsername(),
  spitter.getPassword(),
  spitter.getFullName(),
```

```
  spitter.getEmail(),
  spitter.isUpdateByEmail());
 spitter.setId(queryForIdentity());
}
public Spitter getSpitterById(long id){
 return jdbcTemplate.queryForObject(
  SQL_SELECT_SPITTER_BY_ID,
  new ParameterizedRowMapper<Spitter>(){
   public Spitter mapRow(ResultSet rs,int rowNum) throws SQLException{
    Spitter spitter=new Spitter();
    spitter.setId(rs.getLong(1));
    spitter.setUsername(rs.getString(2));
    spitter.setPassword(rs.getString(3));
    spitter.setFullName(rs.getString(4));
    return spitter;
   }
  },
  id
 );
}
```

上例中，queryForObject()方法有3个参数：

- String，包含了要从数据库中查找数据的SQL；
- ParameterizedRowMappter对象，用来从ResultSet中提取值并构建域对象
- 可变参数列表，列出了要绑定到查询上的索引参数值

使用命名参数

SQL语句应当如下定义：

```
private static final String SQL_INSERT_SPITTER=
 "insert into spitter (username,password,fullname) "+
 "values(:username,:password,:fullname)";
```

方法可以改写如下：

```
public void addSpitter(Spitter spitter){
 Map<String,Object> params=new HashMap<String,Object>();
 params.put("username",spitter.getUsername());
 params.put("password",spitter.getPassword());
 params.put("fullname",spitter.getFullName());
 jdbcTemplate.update(SQL_INSERT_SPITTER,params);
 spitter.setId(queryForIdentity());
}
```

六、 Spring MVC

1. 搭建SpringMVC

在web.xml中添加如下servlet声明：

```
<servlet>
 <servlet-name>spitter</servlet-name>
 <servlet-class>
  org.springframework.web.servlet.DispatcherServlet
 </servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>spitter</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping>
<listener>
 <listener-class>
  org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/spitter-security.xml
  classpath:service-context.xml
  classpath:persistence-context.xml
  classpath:dataSource-context.xml
 </param-value>
</context-param>
```

默认情况下，DispatcherServlet在加载时会从一个基于配置的servlet-name为名字的XML文件中加载pring应用上下文，如spitter-servlet.xml（位于应用程序的WEB-INF目录下）。

创建spitter-servlet.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xmlns:mvc="..." xmlns:schemaLocation="...">
 <mvc:resources mapping="/resources/**" location="/resources/"/>
 <!--注解驱动-->
 <mvc:annotation-driven/>
 <!--配置发现Controller类-->
 <context:component-scan base-package="com.habuma.spitter.mvc"/>
 <!--配置InternalResourceViewResolver-->
 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views/"/>
  <proeprty name="suffix" value=".jsp"/>
 </bean>
 <!--注册TilesViewResolver-->
 <bean class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"/>
 <!--添加TilesConfigurer-->
 <bean class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
   <lsit>
    <value>/WEB-INF/views/**/views.xml</value>
   </list>
```

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

mvc:resources建立了一个服务于静态资源的处理器。以上配置表明，所有以/resources路径开头的求都会自动由应用程序根目录下的/resources目录提供服务。因此，所有照片、样式表、JavaScript及其他静态资源都必须放在应用程序的resources目录下。

2. 编写基本的控制器

开发面向资源的控制器，即为应用程序所提供的每一种资源编写一个单独的控制器，而不是为每个用编写一个控制器。

配置注解驱动的Spring MVC

Spring提供如下处理器映射实现：

- BeanNameUrlHandlerMapping：根据控制器Bean的名字将控制器映射到URL
- ControllerBeanNameHandlerMapping：类似于上面，但Bean的名字不需要遵循URL约定
- ControllerClassNameHandlerMapping：通过使用控制器的类名作为URL基础将控制器映射到URL
- DefaultAnnotationHandlerMapping：将请求映射给使用@RequestMapping注解的控制器和控器方法
- SimpleUrlHandlerMapping：使用定义在Spring应用上下文的属性集合将控制器映射到URL

定义首页的控制器

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;
@Controller //声明为控制器
public class HomeController{
 public static final int DEFAULT_SPITTERS_PER_PAGE=25;
 private SpitterService spitterService;
 @Inject //注入SpitterService
 public HomeController(SpitterService spitterService){
  this.spitterService=spitterService;
 }
 @RequestMapping({"/","/home"}) //处理对首页的请求
 public String showHomePage(Map<String,Object> model){ //参数可以是任何事物
  model.put("spittles",spitterService.getRecentSpittles(DEFAULT_SPITTERS_PER_PAGE));
  return "home"; //返回视图名称
 }
}
```

解析视图

Spring自带多个视图解析器实现提供选择：

- BeanNameViewResolver：查找 <bean>ID与逻辑视图名称相同View的实现
- ContentNegotiatingViewResolver：委托给一个或多个视图解析器，选择哪一个取决于请求的内类型
- FreeMarkerViewResolver：查找一个基于FreeMarker的模板，它的路径根据加完前缀和后缀的逻视图名称来决定
- InternalResourceViewResolver：在Web应用程序的WAR文件中查找视图模板。视图模板的路径据加完前缀和后缀的逻辑视图名称来确定
- JasperReportViewResolver：根据加完前缀和后缀的逻辑视图名称来查找一个Jasper Report报表件
- ResourceBundleViewResolver：根据属性文件（properties file）来查找View实现
- TilesViewResolver：查找通过Tiles模板定义的视图。模板的名字与逻辑视图名称相同
- UrlBasedViewResolver：
- VelocityLayoutViewResolver：
- VelocityViewResolver：解析基于Velocity的视图，Velocity模板的路径根据加完前缀和后缀的逻视图名称来确定
- XmlViewResolver：查找在XML文件(/WEB-INF/views.xml)中声明的View实现
- XsltViewResolver：解析基于XSLT的视图，XSLT样式表的路径根据加完前缀和后缀的逻辑视图名来确定

解析Tiles视图

views.xml如下：

```
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles COnfiguration 2.1//EN"
 "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
 <definition name="template" template="/WEB-INF/views/main_template.jsp">
  <put-attribute name="top" value="/WEB-INF/views/tiles/spittleForm.jsp"/>
  <put-attribute name="side" value="/WEB-INF/views/tiles/signinsignup.jsp"/>
 </definition>
 <definition name="home" extends="template">
  <put-attribute name="content" value="/WEB-INF/views/home.jsp"/>
 </definition>
</tiles-definitions>
```

定义首页的视图

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="s" uri="http:www.springframework.org/tags"%>
<%@ tagllib prefix="t" uri="http://tiles.apache.org/tags-tiles"%>
<%@ taglib prefiex="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<div>
 <h2>A global community of friends and stragers sfksjsldfsldj</h2>
 <h3>Look at what thes people are spittling right now</h3>
 <ol class="spittle-list">
```

```
  <c:forEach var="spittle" items="${spittles}">
   <s:url value="/spitters/{spitterName}" var="spitter_url">
    <s:param name="spitterName" value="${spittle.spitter.username}"/>
   </s:url>
   <li>
    <span class="spittleListImage">
     <img src="http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
      width="48" border="0" align="middle"
      onError="this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
    </span>
   </li>
  </c:forEach>
 </ol>
</div>
```

完成Spring应用上下文

DispatcherServlet为名为spitter-servlet.xml的文件中加载Bean，其他的需要另一种方式加载，即使ContextLoaderListener，配置如上面web.xml。通过contextConfigLocation参数为ContextLoaderLstner加载哪些配置文件。如果没有指定，则默认查找/WEB-INF/applicationContext.xml。

3. 处理控制器的输入

SpitterController实现如下：

```
package com.habuma.spitter.mvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ReuquestParam;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.service.SpitterService;
import static org.springframework.web.bind.annotation.RequestMethod.*;
@Controller
@RequestMapping("/spitter") //根URL路径
public class SpitterController{
 private final SpitterService spitterService;
 @Inject
 public SpitterController(SpitterService spitterService){
  this.spitterService=spitterService;
 }
 @RequestMapping(value="/spittles", method=RequestMethod.GET){ //处理针对spitter/spittle
的Get请求
 public String listSpittlesForSpitter(@RequestParam("Spitter") String username,Model model){
  Spitter spitter=spitterService.getSpitter(username);
  model.addAttribute(spitter); //填充模型
  model.addAttribute(spitterService.getSpittlesForSpitter(username));
  return "spittles/list";
 }
}
```

渲染视图

创建Tile定义：

```
<definition name="spittles/lit" extends="template">
 <put-attribute name="content" value="/WEB-INF/views/spittles/list.jsp"/>
</definition>
```

其中，list.jsp如下：

```
<%@taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<div>
 <h2>Spittles for ${spitter.username}</h2>
 <table cellspacing="15">
  <c:forEach items="${spittleList}" var="spittle">
   <tr>
    <td>
     <img src="<s:url value="/resources/images/spitter_avatar.png"/>" width="48" height="4
"/>
    </td>
    <td>
     <a href="<s:url value="/spitters/${spittle.spitter.username}"/>">
      ${spittle.spitter.username}
     </a>
     <c:out value="${spittle.text}"/><br/>
     <c:out value="${spittle.when}"/>
    </td>
   </tr>
  </c:forEach>
 </table>
</div>
```

4. 处理表单

展现注册表单

```
@RequestMapping(method=RequestMethod.GET, params="new")
//该方法只处理HTTP GET请求并要求请求中必须包含名为new的查询参数
public String createSpitterProfile(Model model){
 model.addAttribute(new Spitter());
 return "spitters/edit";
}
```

访问路径如：http://localhost:8080/Spitter/spitters?new

定义表单视图：

```
<definition name="spitters/edit" extends="template">
 <put-attribute name="content" value="/WEB-INF/views/spitters/edit.jsp"/>
</definition>
```

edit.jsp如下：

```
<%@taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<div>
 <h2>Create a free Spitter account</h2>
 <sf:form method="POST" modelAttribute="spitter">
  <fieldset>
  <table cellspacing="0">
   <tr>
    <th><label for="user_full_name">Full name:</label></th>
    <td><sf:input path="fullName" size="15" id="user_full_name"/></td>
   </tr>
   <tr>
    <th><label for="user_screen_name">Username:</label></th>
    <td>
     <sf:input path="username" size="15" maxlength="15" id="user_screen_name"/>
     <small id="username_msg">No spaces,please.</small>
    </td>
   </tr>
   <tr>
    <th><label for="user_password">Password:</label></th>
    <td>
     <sf:password path="password" size="30" showPassword="true" id="user_password"/>
     <small>6 characters of more (be tricky!)</small>
    </td>
   </tr>
   <tr>
    <th><label for="user_email">Email Address:</label></th>
    <td>
     <sf:input path="email" size="30" id="user_email"/>
     <small>In case you forget  something</small>
    </td>
   </tr>
   <tr>
    <th></th>
    <td>
     <sf:checkbox path="updateByEmail" id="user_send_email_newsletter"/>
     <label for="user_send_email_newsletter">Send me email updates!</label>
    </td>
   </tr>
  </table>
 </fieldset>
 </sf:form>
</div>
```

处理表单输入

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,BindingResult bindingResult){
//@Valid需要添加hibernate-validator依赖
 if(bindingResult.hasErrors()){
  return "spitters/edit";
 }
 spitterService.saveSpitter(spitter);
 return "redirect:/spitters/"+spitter.getUsername();
```

```
}
```

处理带有路径变量的请求：

```
@RequestMapping(value="/{username}",method=RequestMethod.GET)
public String showSpitterProfile(@PathVariable String username,Model model){
 model.addAttribute(spitterService.getSpitter(username));
 return "spitters/view";
}
```

定义校验规则：

```
@Size(min=3,max=20,message="Username must be between 3 and 20 characters long.")
@Pattern(regexp="^[a-zA-z0-9]+$",message="Username must be alphanumeric with no spa
es")
private String username;
@Size(min=6,max=20,message="The password must be at least 6 characters long.")
private String password;
@Size(min=3,max=50,message="Your full name must between 3 and 50 characters long.")
private String fullName;
@Pattern(regexp="A-za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",message="Invalid email
ddress.")
private String email;
```

展现校验错误：

```
<%@taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<div>
<h2>Create a free Spitter account</h2>
<sf:form method="POST" modelAttribute="Spitter" enctype="multipart/form-data">
 <fieldset>
 <table cellspacing="0">
  <tr>
   <th><sf:label path="fullName">Full name:</sf:label></th>
   <td>
    <sf:input path="fullName" size="15"/><br/>
    <sf:errors path="fullName" cssClass="error"/>
   </td>
  </tr>
  <tr>
   <th><sf:label path="username">Username:</sf:label></th>
   <td>
    <sf:input path="username" size="15" maxlength="15"/>
    <small id="username_msg">No spaces,please.</small><br/>
    <sf:errors path="username" cssClass="error"/>
   </td>
  </tr>
  <tr>
   <th><sf:label path="password">Password:</sf:label></th>
   <td>
    <sf:password path="password" size="30" showPassword="true"/>
    <small>6 characters or more (be tricky!)</small><br/>
    <sf:errors path="password" cssClass="error"/>
   </td>
```

```
  </tr>
  <tr>
   <th><sf:label path="email">Email Address:</sf:label></th>
   <td>
    <sf:input path="email" size="30"/>
    <small>In case you forget something</small><br/>
    <sf:errors path="email" cssClass="error"/>
   </td>
  </tr>
  <tr>
   <th></th>
   <td>
    <sf:checkbox path="updateByEmail"/>
    <sf:label path="updateByEmail">Send me email updates!</sf:label>
   </td>
  </tr>
  <tr>
   <th><label for="image">Profile image:</label></th>
   <td><input name="image" type="file"/>
  </tr>
  <tr>
   <th></th>
   <td><input name="commit" type="submit" value="I accept, Create my account."></td>
  </tr>
 </table>
 </fieldset>
</sf:form>
</div>
```

5. 处理文件上传

在表单上添加文件上传域

```
<sf:form method="POST" modelAttribute="spitter" enctype="multipart/form-data">
...
 <tr>
  <th><label for="image">Profile image:</label></th>
  <td><input name="image" type="file"/></td>
 </tr>
</sf:form>
```

接收上传的文件

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
 BindingResult bindingResult,
 @RequestParam(value="image",required=false) MultipartFile iamge){
 if(bindingResult.hasErrors()){
  return "spitters/edit";
 }
 spitterService.saveSpitter(spitter);
 try{
```

```
  if(!image.isEmpty()){
   validateImage(image);
   saveImage(spitter.getId()+".jpg",image);
  }
 }catch(ImageUploadException e){
  bindingResult.reject(e.getMessage());
  return "spitters/edit";
 }
 return "redirect:/spitters/"+spitter.getUsername();
}
private void validateImage(MultipartFile image){
 if(!image.getContentType().equals("image/jpeg")){
  throw new ImageUploadException("Only JPG image accepted");
 }
}
private void saveImage(String filename,MultipartFile image)
 throws ImageUploadException{
 try{
  File file=new File(webRootPath+"/resources/"+filename);
  FileUtils.writeByteArrayToFile(file,image.getBytes());
 }catch(IOException e){
  throw new ImageUploadException("Unable to save image",e);
 }
}
```

配置Spring支持文件上传

在Spring中配置如下特定ID的Bean，使DispatcherServet能够识别上传文件：

```
<bean id="multipartResovler" class="org.springframework.web.multipart.commons.CommonMultipartResolver" p:maxUploadSize="500000"/>
```