阅读笔记1

第一章 Spring之旅

这里我就节选一点东西了

依赖注入

任何一个实际意义的应用都是由两个或者更多的类组成 这些类相互之间进行协作来完成特定的业务逻辑 通常 每个对象负责管理与自己相互协作的对象（即它所引用的对象）的引用 这将会导致高度耦合的难以测试的代码

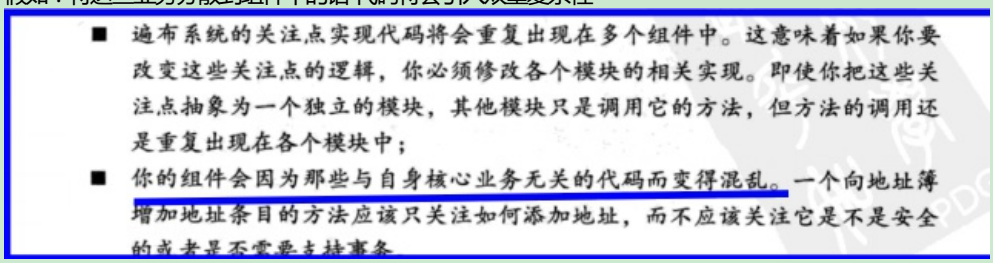
但是**耦合又有俩面性** 一方面 紧密耦合的代码难以测试 难以复用 难以理解 并且典型的表示为“打地鼠”的bug特性(修复一个bug 导致出现一个新的或者更多的bug) 另一方面 一定的程度的耦合又是必须的 完全没有耦合的代码什么也做不了 为了完成有实际意义的功能 不同的类必须以适当的方式进行交互 **总而言之 耦合是必须的 但是必须小心管理它**

应用切面

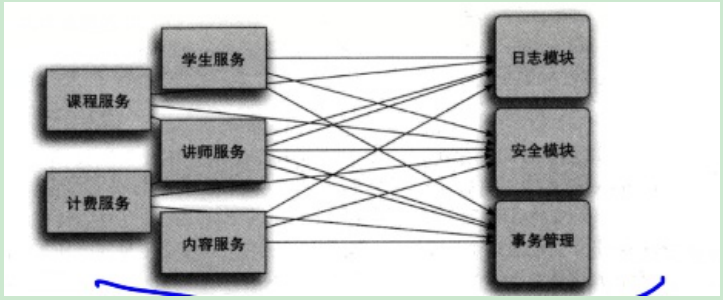
依赖注入让相互协作的软件组件保持松散耦合 而AOP编程允许你把遍布应用各处的功能分离出来形成可重复的组件

面向切面编程往往被定义为促使应用程序分离关注点的一项技术 系统由许多不同的组件组成 每一个组件个负责一块特定功能 除了实现自身核心的功能之外 这些组件还经常承担额外的职责 诸如日志 事务管理，和安全此类的系统服务经常融入到核心业务逻辑的组件中去 这些系统服务通常被称为 **横向关注点**

假如：将这些业务分散到组件中的话 代码将会引入双重复杂性



这个图就展现了这种复杂性



Spring自带了几种容器的实现 可以归为俩种不同的类型 一种是： **Bean工厂**(org.Springfamework,beans.factory.BeanFactory接口定义)是最简单的容器 提供基本的DI支持 另一种是：**应用上下文**(org.Springfamework.context.Applicationcontext接口定义)基于BeanFactory之上构建 并提供面向应用的服务 例如从属性文件解析文本信息的能力 以及发布应用事件给感兴趣的事件监听器的能力

主要是使用的应用上下文 Bean工厂比较低级

自带了几种类型的应用上下文 主要使用的如下

ClassPathXmlApplicationContext -从类路径下的XML配置文件中加载上下文定义 把应用上下文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

Bean的生命周期

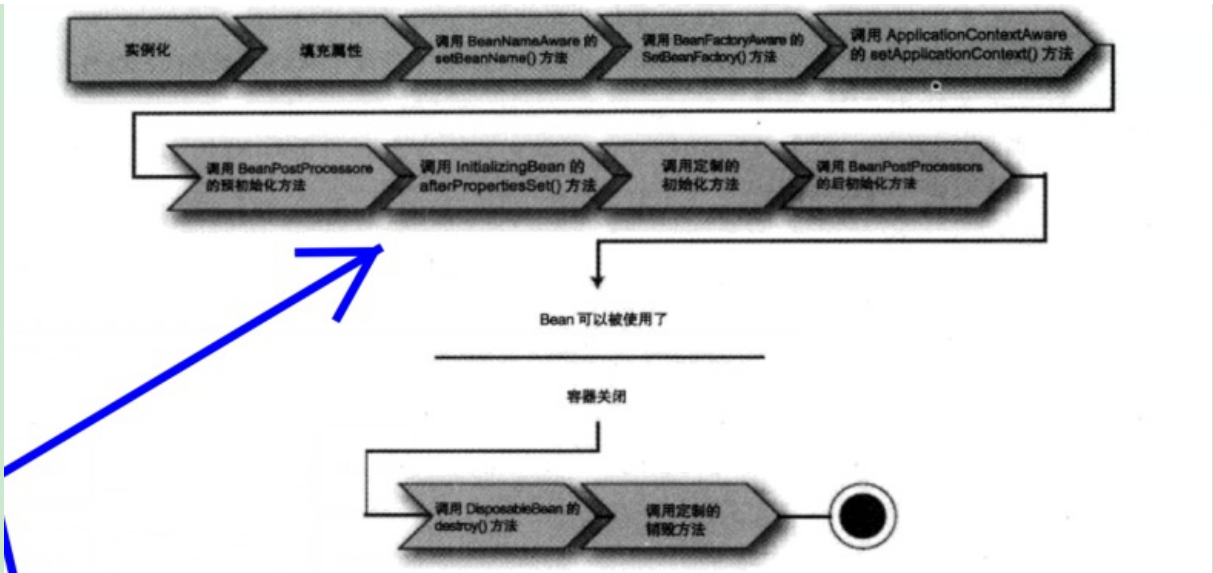


图 1.5　Bean 在 Spring 容器中从创建到销毁经历了若干阶段，每一阶段都可以针对 Spring 如何管理 Bean 进行个性化定制

正如你所见，在 Bean 准备就绪之前，Bean 工厂执行了若干启动步骤。我们对图 1.5 进行详细描述。

进行详细描述。

1. Spring 对 Bean 进行实例化。
2. Spring 将值和 Bean 的引用注入进 Bean 对应的属性中。
3. 如果 Bean 实现了 `BeanNameAware` 接口，Spring 将 Bean 的 ID 传递给 `setBeanName()` 接口方法。
4. 如果 Bean 实现了 `BeanFactoryAware` 接口，Spring 将调用 `setBeanFactory()` 接口方法，将 BeanFactory 容器实例传入。
5. 如果 Bean 实现了 `ApplicationContextAware` 接口，Spring 将调用 `setApplicationContext()` 接口方法，将应用上下文的引用传入。
6. 如果 Bean 实现了 `BeanPostProcessor` 接口，Spring 将调用它们的 `postProcessBeforeInitialization()` 接口方法。
7. 如果 Bean 实现了 `InitializingBean` 接口，Spring 将调用它们的 `afterPropertiesSet()` 接口方法。类似地，如果 Bean 使用 `init-method` 声明了初始化方法，该方法也会被调用。
8. 如果 Bean 实现了 `BeanPostProcessor` 接口，Spring 将调用它们的 `postPoressAfterInitialization()` 方法。
9. 此时此刻，Bean 已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁。

Spring模块

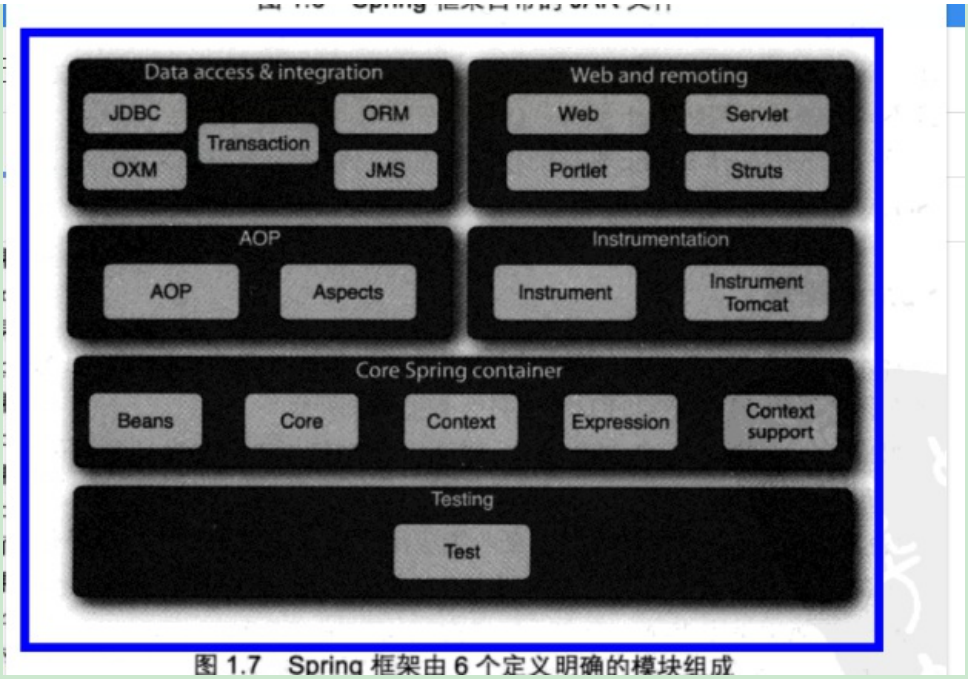


图 1.7 Spring 框架由 6 个定义明确的模块组成

第二章 装配Bean

表 2.1 Java 自带了多种 XML 命名空间，通过这些命名空间可以配置 Spring 容器

命名空间	用 途
aop	为声明切面以及将 @AspectJ 注解的类代理为 Spring 切面提供了配置元素
beans	支持声明 Bean 和装配 Bean，是 Spring 最核心也是最原始的命名空间
context	为配置 Spring 应用上下文提供了配置元素，包括自动检测和自动装配 Bean、注入非 Spring 直接管理的对象
jee	提供了与 Java EE API 的集成，例如 JNDI 和 EJB
jms	为声明消息驱动的 POJO 提供了配置元素
lang	支持配置由 Groovy、JRuby 或 BeanShell 等脚本实现的 Bean
mvc	启用 Spring MVC 的能力，例如面向注解的控制器、视图控制器和拦截器
oxm	支持 Spring 的对象到 XML 映射配置
tx	提供声明式事务配置
util	提供各种各样的工具类元素，包括把集合配置为 Bean、支持属性占位符元素

这本书还介绍了一种方法来创建一种方法来创建bean
那就是工厂创建

这是一种创建bean的方法 是如果没有公开的构造方法时 如果是单例模式，可以使用到的，
这是Demo：

```
public class Stage {

private Stage() {

    }

private static class StageSingletonHolder {
static Stage instance=new Stage();
}

public static Stage getInstance() {
return StageSingletonHolder.instance;
}
}
```

这是XML的配置方式

```
<bean id="theStage" class="com.spring.bean.Stage" factory-method="getInstance"/>
```

关于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。该作用域仅在基于 Web 的 Spring 上下文（例如 Spring MVC）中才有效
session	在一个 HTTP Session 中，每个 Bean 定义对应一个实例。该作用域仅在基于 Web 的 Spring 上下文（例如 Spring MVC）中才有效
global-session	在一个全局 HTTP Session 中，每个 Bean 定义对应一个实例。该作用域仅在 Portlet 上下文中才有效

如果我们使用Spring作为工厂来创建领域对象的新实例时 prototype就会非常有用 如果领域对象的作用域是prototype 我们在Spring中就可以很容易的配置它们
Spring有关单例的概念限于Spring上下文的范围内 不像真正的单例 在每个类加载器中只有一个实例 Spring的单例Bean只能保证在每个应用上下文中只有一个Bean的实例 没有人可以阻止你使用传统的方式实例化同一个Bean 或者你甚至可以定义几个<bean>声明来实例化同一个Bean

关于内部Bean的设置

```
<bean id="kenny2" class="com.spring.bean.Instrumentalist">

    <propertyname="song" value="Jingle Bells"/>

    <propertyname="instrument" >

        <bean  class="com.spring.bean.Saxophone"/>

    </property>

</bean>
```

关于是Spring的命名空间P的装配属性

```
xmlns:p="http://www.springframework.org/schema/p"
```

```
<bean id="theStage" class="com.spring.bean.Stage" factory-method="getInstance"/>

<bean id="kenny" class="com.spring.bean.Instrumentalist"

p:song="Jingle Bells"

p:instrument-ref="saxopone"

/>
```

关于Bean装配集合

集合元素	用途
<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

```
public class OneManBand implements Performer {


public OneManBand(){


    }

private Collection<Instrument> instruments;
```

```
@Override
public void perform() throws PerformanceException {
for (Instrument instrument: instruments
) {
            instrument.play();
}
    }
public  void setInstruments(Collection<Instrument> instruments){
this.instruments =instruments;
}

}
```

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

下面是关于装酝的另外的几种情况，包括Map方式，properties的方式
这是Map方式的

```
<property name="storeMap" >
    <map>
        <entry key="IntergStore" value-ref="IntergStore"/>
        <entry key="StringStore" value-ref="StringStore"/>
    </map>
</property>
```

这是properties的方式的

```
<bean id="hank" class="com.spring.bean.OneManBand">
    <property name="instruments">
     <props>
         <prop key="GUITAR">stuum strum STRUM</prop>
          <prop key="cymabl">CRASH CRASH CRASH</prop>
         <prop key="harmonica">hum hum hum</prop>
     </props>
    </property>
</bean>
```

```
public class OneManBand implements Performer {

public OneManBand(){

    }
private Properties instruments;
@Override
public void perform() throws PerformanceException {
//         for (String  key: instruments.keySet()
//                 ) {
//             System.out.println(key+" : ");
//             Instrument instrument=instruments.get(key);
//             instrument.play();
//         }这是MAP装配方式时写的
}
public  void setInstruments(Properties instruments)
    {
this.instruments =instruments;
}

}
```

下面是主动给我们需要装配的值装配null值，这是显示的装配方法

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

下面是关于Spring表达式语言的部分

SpEl表达式

```
<property name="instrument" value="#{5}" >
```

这是字面值的装配

```
<bean id="carl" class="com.spring.bean.Instrumentalist">
    <property name="song" value="#{kenny.song}"/>
</bean>
```

这也是 这个是关于ref的属性的SpEl

这个的SpEl表达式的作用是还可以调用这个Bean的方法 还可以进行全部大写 还可以避免null值的错误 避免出现空指针异常
使用Null-safe存储器 使用？.的方式来使用toUpperCase方法避免，如果是空值的话，就不会调用这个toUpperCase方法

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

使用SpEl还可以使用运算符来调用类作用域的方法和常量
T()运算符 同样，这个方法是可以调用静态方法的

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

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

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

我们还可以在Spel值上面进行执行操作 比如说这样
我们可以进行加减乘除
求余，以及java不具有的乘方

```
<property name="randoNumber" value="#{T(java.lang.Math).random()+42}"/>
```

这是乘方
特别提一下+运算符 是可以执行字符串拼接的

```
<property name="fullName" value="#{performer.firstName+''+performer.lastName}"/>
```

还可以用比较运算符

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

我们还可以用逻辑表达式

```
<property name="larfeCircle" value="#{shape.kind =='circle  and sha[e/erimeter gt 1
0000} "/>
```

```
<property name="outOfStock" value="#{!product.available}"/>
```

奇怪的是，**SpEL没有提供文本型的and和or运算符**
条件表达式

三元表达式

```
<property name="instrument" value="#{songSelector.selectSong()=='Jingle Bells ?piano
:saxophone'}"/>
```

我们经常来检测一个值是否为null

```
<property name="song" value="#{kenny.song !=null? kenny.song :'GreenSleeves'}"/>
```

提供了变体来简化这个三元表达式

```
<property name="song" value="#{ kenny.song ?:'GreenSleeves'}"/>
```

效果是一样的 这个？：通过被称为elvis运算符

我们还可以用matches运算符来进行正则表达式判断

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

我们还可以在SpEL来筛选集合

通过<util：list>元素来配置集合

```
<util:list id="cities">

     <bean class="com.spring.bean.City"  p:name="Chicago"  p:state="IL"

p:population="2853114" > </bean>

    <bean class="com.spring.bean.City"  p:name="Atlata"  p:state="GA"

p:population="537958" > </bean>

    <bean class="com.spring.bean.City"  p:name="Dallas"  p:state="TX"

p:population="1279910" > </bean>
```

```
    <bean class="com.spring.bean.City"  p:name="Houston"  p:state="TX"
p:population="2242193" > </bean>
    <bean class="com.spring.bean.City"  p:name="pSO"  p:state="IL"
p:population="90943" > </bean>
    <bean class="com.spring.bean.City"  p:name="jAL"  p:state="IL"
p:population="1996" > </bean>
</util:list>
```

```
<property name="chosenCity" value="#{cities[T(java.lang.Math).random() *cities.size()]}"/>
```

我们将[]运算符来从通过索引来访问集合中的成员

比如如果获取Map集合内的成员的话，

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

我们还可以properties的方法

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

settings Bean是一个util.Properties类，加载了名为set-tings.propertues
的文件，通过SpEL，我们可以访问Properties的属性，就像访问Map中的成员一样

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

还有**俩种特别的属性的访问方式**
一个是systemEnvironment 一个是systemProperties
[]也可以通过索引来获得字符串的某个字符
'This is a test'[3]

systemEnvironment 包含了所在机器的上所有环境变量

```
<property name="homePath" value="#{systemEnvironment('Home')}"/>
```

systemProperties包含了java应用程序启动的时候所设置的所有属性
这是查询集合成员时的查询某一个属性多于10000的情况 并将这个属性装配到bigCities中 同时SpEl还提供其他的俩种运算符".^[]"和.$[]
这俩个第一个是查询第一个匹配项，与查询最后一个匹配项

```
<property name="bigCities" value="#{cities.?[population gt 10000]}"/>
<property name="aBigCities" value="#{cities.^[population gt 10000]}"/>
<property name="zBigCities" value="#{cities.$[population gt 10000]}"/>
```

投影集合
将集合的每一个成员选择特定的属性放入一个新的集合中，SpEL的投影运算符（.![]）

```
<property name="cityNames" value="#{cities.![name]}"/>
```

这个属性就是装配一个String类型的集合，是只装配name的属性

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

还可以变换一下 这样子就装配了俩个属性了一个是name 一个是state属性了

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

还可以混合起来用

```
<property name="cityNames" value="#{cities.?[population gt 10000].![name+' , '+state]}"/>
```

这样子，就可以看到，**我们是可以进行复杂的SpEL表达式的混用的的，可以构建很复杂的表达式，但是这个表达式是没有IDE的语法检测的，也不好做测试，所以只有在使用传统方式不容易而SpEL很方便的时候用SpEl用**