# 第一部分 知识准备篇

## 第一章 开发环境准备

1、准备源代码阅读环境

2、获取spring以及源码

## 第二章 spring初体验

### 2.1、利用官方示例

进行一场quick start。

### 2.2、如何学习开源框架

### 2.3、示例

#### 2.3.1 基于配置的依赖注入

|  |
| --- |
| **public class BraveKnight implements Knight {**  **private Quest quest;**  **public BraveKnight(Quest quest) {**  **this.quest = quest;**  **}**  **public void embarkOnQuest() {**  **quest.embark();**  **}**  **}** |

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**  **<bean id=*"knight"* class=*"com.glxt.study.spring.springinaction.chapter1.knight.BraveKnight"*>**  **<constructor-arg ref=*"quest"* />**  **</bean>**  **<bean id=*"quest"* class=*"com.glxt.study.spring.springinaction.chapter1.knight.SlayDragonQuest"*>**  **<constructor-arg value=*"#{T(System).out}"* />**  **</bean>**  **</beans>** |

#### 2.3.2 基于配置的AOP编程

|  |
| --- |
| **public class Minstrel {**  **private PrintStream stream;**    **public Minstrel(PrintStream stream) {**  **this.stream = stream;**  **}**  **public void singBeforeQuest() {**  **stream.println("Fa la la, the knight is so brave!");**  **}**  **public void singAfterQuest() {**  **stream.println("Tee hee hee, the brave knight " +**  **"did embark on a quest!");**  **}**  **}** |

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xmlns:aop=*"http://www.springframework.org/schema/aop"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop-3.2.xsd***  ***http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**  **<bean id=*"knight"* class=*"com.glxt.study.spring.springinaction.chapter1.knight.BraveKnight"*>**  **<constructor-arg ref=*"quest"* />**  **</bean>**  **<bean id=*"quest"* class=*"com.glxt.study.spring.springinaction.chapter1.knight.SlayDragonQuest"*>**  **<constructor-arg value=*"#{T(System).out}"* />**  **</bean>**  **<bean id=*"minstrel"* class=*"com.glxt.study.spring.springinaction.chapter1.knight.Minstrel"*>**  **<constructor-arg value=*"#{T(System).out}"* />**  **</bean>**    **<aop:config>**  **<aop:aspect ref=*"minstrel"*>**  **<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>** |

### 2.4 容器

spring容器分为两种类型：

1、BeanFactory

2、ApplicationContext

ApplicationContext是BeanFactory的子接口，在实际开发中，经常优先选用ApplicationContext。

#### 2.4.1 application contexts概述

方式一：**ClassPathXmlApplicationContext**

|  |
| --- |
| **public class Test**  **{**  **public void say()**  **{**  **System.*out*.println("hello world!");**  **}**  **public static void main(String[] args)**  **{**  **// 方式一 ：ClassPathXmlApplicationContext**  **ApplicationContext context = new ClassPathXmlApplicationContext("com/glxt/study/spring/springinaction/chapter1/applicationcontext/application.xml");**  **Test test = (Test) context.getBean("test");**  **test.say();**  **}**  **}** |

方式二：**FileSystemXmlApplicationContext**

|  |
| --- |
| **public static void main(String[] args)**  **{**  ***method2*();**  **}**  **public static void method2()**  **{**  **// 方式二：****FileSystemXmlApplicationContext**  **ApplicationContext context = new FileSystemXmlApplicationContext(**  **"E:/study/spring-study/spring/src/main/java/com/glxt/study/spring/springinaction/chapter1/applicationcontext/application.xml");**  **Test test = (Test) context.getBean("test");**  **test.say();**  **}** |

方式三：**AnnotationConfigApplicationContext**

|  |
| --- |
| **public static void main(String[] args)**  **{**  ***method3*();**  **}**    **public static void method3()**  **{**  **// 方式三：AnnotationConfigApplicationContext**  **ApplicationContext context = new AnnotationConfigApplicationContext(**  **com.glxt.study.spring.springinaction.chapter1.applicationcontext.Test.class);**  **Test test = (Test) context.getBean(com.glxt.study.spring.springinaction.chapter1.applicationcontext.Test.class);**  **test.say();**  **}** |

方式四： **XmlWebApplicationContext**

方式五： **AnnotationConfigWebApplicationContext**

#### 2.4.2 bean生命周期

### 2.5 实例化bean

实例化bean，分为两种方式：

1、反射模式

2、工厂方法模式

工厂静态方法模式：配置依赖factory-method

工厂方法模式：配置依赖factory-bean、factory-method

待实例化的bean，如下：

|  |
| --- |
| **public class Car**  **{**  **private int id;**  **private String name;**  **private int price;**  **public int getId()**  **{**  **return id;**  **}**  **public void setId(int id)**  **{**  **this.id = id;**  **}**  **public String getName()**  **{**  **return name;**  **}**  **public void setName(String name)**  **{**  **this.name = name;**  **}**  **public int getPrice()**  **{**  **return price;**  **}**  **public void setPrice(int price)**  **{**  **this.price = price;**  **}**  **@Override**  **public String toString()**  **{**  **return "Car [id=" + id + ", name=" + name + ", price=" + price + "]";**  **}**  **public Car()**  **{**  **}**  **public Car(int id, String name, int price)**  **{**  **super();**  **this.id = id;**  **this.name = name;**  **this.price = price;**  **}**  **}** |

#### 2.5.1 反射模式

反射模式实例化bean，可以通过调用无参或者有参构造方法实例化，一般是通过反射调用无参构造函数实例化bean。

1、无参构造函数，配置spring xml

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**  **<bean id=*"car"* class=*"com.glxt.study.spring.instantiatingbeans.reflection.nonargument.Car"*>**  **</bean>**  **</beans>** |

这种调用无参构造的方式也是经常使用。

2、有参构造函数，配置spring xml

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**  **<bean id=*"car"* class=*"com.glxt.study.spring.instantiatingbeans.reflection.argument.Car"*>**  **<constructor-arg type=*"int"* value=*"1"*></constructor-arg>**  **<constructor-arg type=*"String"* value=*"BMW"*></constructor-arg>**  **<constructor-arg type=*"int"* value=*"540000"*></constructor-arg>**  **</bean>**  **</beans>** |

#### 2.5.2 工厂方法模式

工厂方法实例化bean，分为静态工厂方法和非静态工厂方法。

1、静态工厂方法

|  |
| --- |
| **public class CarStaticFactory**  **{**  **private static Map<Integer, Car> *map* = new HashMap<Integer, Car>();**  **static**  **{**  ***map*.put(1, new Car(1, "Honda", 300000));**  ***map*.put(2, new Car(2, "Audi", 440000));**  ***map*.put(3, new Car(3, "BMW", 540000));**  **}**  **public static Car getCar(int id)**  **{**  **return *map*.get(id);**  **}**  **}** |

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**  **<bean id=*"bmwCar"* class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.staticmethod.CarStaticFactory"* factory-method=*"getCar"*>**  **<constructor-arg value=*"3"*></constructor-arg>**  **</bean>**    **<bean id=*"audiCar"* class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.staticmethod.CarStaticFactory"* factory-method=*"getCar"*>**  **<constructor-arg value=*"2"*></constructor-arg>**  **</bean>**    **</beans>** |

|  |
| --- |
| **public class Test**  **{**  **public static void main(String[] args)**  **{**  **ApplicationContext context = new ClassPathXmlApplicationContext("com/glxt/study/spring/instantiatingbeans/factorymethod/staticmethod/car.xml");**  **Car car = context.getBean("bmwCar", Car.class);**  **System.*out*.println(car);**    **Car car2 = context.getBean("audiCar", Car.class);**  **System.*out*.println(car2);**  **}**  **}** |

2、非静态工厂方法

|  |
| --- |
| **public class CarFactory**  **{**  **private Map<Integer, Car> map = new HashMap<Integer, Car>();**  **public void setMap(Map<Integer, Car> map)**  **{**  **this.map = map;**  **}**  **public CarFactory()**  **{**  **}**  **public Car getCar(int id)**  **{**  **return map.get(id);**  **}**  **}** |

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd"*>**    **<!-- Instance Factory Method:**  **1.must create a bean for the Instance Factroy First**  **-->**  **<bean id=*"carFactory"* class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.nonstaticmethod.CarFactory"*>**  **<property name=*"map"*>**  **<map>**  **<entry key=*"1"*>**  **<bean class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.nonstaticmethod.Car"*>**  **<property name=*"id"* value=*"1"*></property>**  **<property name=*"name"* value=*"Honda"*></property>**  **<property name=*"price"* value=*"300000"*></property>**  **</bean>**  **</entry>**  **<entry key=*"2"*>**  **<bean class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.nonstaticmethod.Car"*>**  **<property name=*"id"* value=*"2"*></property>**  **<property name=*"name"* value=*"Audi"*></property>**  **<property name=*"price"* value=*"440000"*></property>**  **</bean>**  **</entry>**  **<entry key=*"3"*>**  **<bean class=*"com.glxt.study.spring.instantiatingbeans.factorymethod.nonstaticmethod.Car"*>**  **<property name=*"id"* value=*"3"*></property>**  **<property name=*"name"* value=*"BMW"*></property>**  **<property name=*"price"* value=*"540000"*></property>**  **</bean>**  **</entry>**  **</map>**  **</property>**  **</bean>**    **<!-- 2.use Factory bean to get bean object**  **factory-bean : the bean define above**  **factory-method: method of get Bean Object**  **constructor-arg: parameters of factory-method**  **-->**  **<bean id=*"bmwCar"* factory-bean=*"carFactory"* factory-method=*"getCar"*>**  **<constructor-arg value=*"3"*></constructor-arg>**  **</bean>**    **<bean id=*"audiCar"* factory-bean=*"carFactory"* factory-method=*"getCar"*>**  **<constructor-arg value=*"2"*></constructor-arg>**  **</bean>**    **</beans>** |

### 2.6 基于注解的容器配置

#### @Required

[Spring依赖检查](http://www.yiibai.com/spring/spring-properties-dependency-checking.html) bean 配置文件用于确定的特定类型(基本，集合或对象)的所有属性被设置。在大多数情况下，你只需要确保特定属性已经设置但不是所有属性..

对于这种情况，你需要 @Required 注解。

#### @Autowired

@Autowired注解提供更细粒度地控制在何处以及如何使用自动装配时应完成。这个注解就是spring可以自动帮你把bean里面引用的对象的setter/getter方法省略，它会自动帮你set/get。

<http://www.yiibai.com/spring/spring_autowired_annotation.html>

#### @Qualifier

可能会有这样一种情况，当你创建多个具有相同类型的 bean 时，并且想要用一个属性只为它们其中的一个进行装配，在这种情况下，你可以使用 **@Qualifier** 注释和 **@Autowired** 注释通过指定哪一个真正的 bean 将会被装配来消除混乱。

示例：package com.glxt.study.spring.annotation.qualifier

### 2.7 Classpath scanning and managed components

#### @ComponentScan

它一般搭配@Configuration，扫描@Component组件。

<http://javarticles.com/2016/01/spring-componentscan-annotation-example.html>

**基本使用**

|  |
| --- |
| **@Configuration**  **@ComponentScan(basePackages={"com.glxt.study.spring.annotation.componentscan.packageA",**  **"package com.glxt.study.spring.annotation.componentscan.packageB"})**  **public class ComponentScanAnnotationExample**  **{**  **public ComponentScanAnnotationExample()**  **{**  **System.*out*.println("ComponentScanAnnotationExample Constructor");**  **}**  **public static void main(String[] args)**  **{**  **AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ComponentScanAnnotationExample.class);**  **System.*out*.println(ctx.getBean("beanA"));**  **System.*out*.println(ctx.getBean("componentScanAnnotationExample"));**  **System.*out*.println(ctx.getBean("beanB"));**  **}**    **@Override**  **public String toString()**  **{**  **return "ComponentScanAnnotationExample";**  **}**  **}** |

**黑名单过滤**

|  |
| --- |
| **@Configuration**  **@ComponentScan(basePackageClasses=BeanA.class,**  **excludeFilters = @ComponentScan.Filter(type = FilterType.*ASSIGNABLE\_TYPE*,value=BeanA1.class))**  **public class ComponentScanExcludeAnnotationExample**  **{**  **public static void main(String[] args)**  **{**  **AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ComponentScanExcludeAnnotationExample.class);**  **System.*out*.println("Contains BeanA? : " + ctx.containsBean("beanA"));**  **System.*out*.println("Contains BeanA1? : " + ctx.containsBean("beanA1"));**  **}**  **}** |

**白名单过滤**

|  |
| --- |
| **@Configuration**  **@ComponentScan(basePackages={"com.glxt.study.spring.annotation.componentscan.packageA",**  **"com.glxt.study.spring.annotation.componentscan.packageB"},**  **includeFilters = @ComponentScan.Filter(type = FilterType.*ASSIGNABLE\_TYPE*, value = BeanB.class),**  **useDefaultFilters = false)**  **public class ComponentScanIncludeAnnotationExample**  **{**  **public static void main(String[] args)**  **{**  **AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ComponentScanIncludeAnnotationExample.class);**  **System.*out*.println("Contains BeanA? : " + ctx.containsBean("beanA"));**  **System.*out*.println("Contains BeanA1? : " + ctx.containsBean("beanA1"));**  **System.*out*.println("Contains BeanB? : " + ctx.containsBean("beanB"));**  **}**  **}** |

注意：在使用白名单过滤时，如果想选出指定的类，需要使用useDefaultFilters = false。

**XML方式触发**

|  |
| --- |
| **@Configuration**  **@ComponentScan(basePackages={"com.glxt.study.spring.annotation.componentscan.packageA",**  **"com.glxt.study.spring.annotation.componentscan.packageB"})**  **public class ComponentScanAnnotationViaXMLExample**  **{**  **public ComponentScanAnnotationViaXMLExample()**  **{**  **System.*out*.println("Constructor ComponentScanAnnotationViaXMLExample");**  **}**  **public static void main(String[] args)**  **{**  **ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/glxt/study/spring/annotation/componentscan/beans.xml");**  **System.*out*.println(ctx.getBean("beanA"));**  **System.*out*.println(ctx.getBean("beanA1"));**  **System.*out*.println(ctx.getBean("beanB"));**  **}**  **}** |

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xmlns:context=*"http://www.springframework.org/schema/context"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd***  ***http://www.springframework.org/schema/context***  ***http://www.springframework.org/schema/context/spring-context.xsd"*>**    **<context:annotation-config/>**  **<bean class=*"com.glxt.study.spring.annotation.componentscan.ComponentScanAnnotationViaXMLExample"*></bean>**    **</beans>** |

### 2.8 基于java的容器配置

#### @Configuration

@Configuration是一个类级别的注解，它是定义bean的源头。简单地说，它相当于在xml中定义了bean。

|  |
| --- |
| **@Configuration**  **public class HelloWorldConfig**  **{**  **@Bean**  **public HelloWorld helloWorld()**  **{**  **return new HelloWorld();**  **}**  **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(**  **HelloWorldConfig.class);**  **HelloWorld helloWorld = ctx.getBean(HelloWorld.class);**  **helloWorld.setMessage("Hello World!");**  **helloWorld.getMessage();**  **}**  **}** |

#### @Bean

|  |
| --- |
| **@Configuration**  **public class AppConfig {**  **@Bean(initMethod = "init")**  **public Foo foo() {**  **return new Foo();**  **}**  **@Bean(destroyMethod = "cleanup")**  **public Bar bar() {**  **return new Bar();**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(**  **AppConfig.class);**  **Foo foo = ctx.getBean(Foo.class);**  **}**  **}** |

@Bean可以管理对象生命周期的调用。

#### @Scope

@Scope是用来指定Bean的作用域，因而它的先决条件就是存在bean。

|  |
| --- |
| **@Configuration**  **public class AppConfig {**  **@Bean**  **@Scope("prototype")**  **public Bar bar() {**  **return new Bar();**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(**  **AppConfig.class);**    **System.*out*.println("First Bar: ");**  **Bar bar1 = ctx.getBean(Bar.class);**    **System.*out*.println("Second Bar: ");**  **Bar bar2 = ctx.getBean(Bar.class);**  **}**  **}** |

如果不为@Bean显式地配置@Scope，默认情况会为@Bean配置@Scope(“singleton”)。

#### 2.8.1 组装基于java的配置

##### @Import

|  |
| --- |
| **@Configuration**  **@Import(ConfigA.class)**  **public class ConfigB**  **{**  **@Bean**  **public B createB()**  **{**  **return new B();**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);**  **A a = ctx.getBean(A.class);**  **B b = ctx.getBean(B.class);**  **System.*out*.println(a);**  **System.*out*.println(b);**  **}**  **}** |

##### @ImportResource

@ImportResource与@Import的区别：

@ImportResource导入xml文件中的bean资源；

@Import导入java注解类中的bean资源。

|  |
| --- |
| **@Configuration**  **@Import(ConfigA.class)**  **@ImportResource("classpath:/com/glxt/study/spring/annotation/importresource/beans.xml")**  **public class ConfigB**  **{**  **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);**  **A a = ctx.getBean(A.class);**  **B b = ctx.getBean(B.class);**  **System.*out*.println(a);**  **System.*out*.println(b);**  **}**  **}** |

### 2.9 容器配置小结

简单地理解基于XML、注解(Annotation based)、java三种方式的容器配置：

* 基于XML的配置，就是完全使用XML文件配置spring context。
* 基于注解的配置(Annotation based configuration)，一部分使用XML文件配置spring bean，一部分使用注解关联XML中的bean。
* 基于java的配置(Java based configuration)，就是代替XML文件配置，完全使用类注解配置spring context。
* <context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined.

### 2.10 @Profile

一个Profile对象就是一系列为特定领域的集合。类似抽象工厂设计模式，对应着某个产品家族簇。

### 2.11 @PropertySource

|  |
| --- |
| **@Configuration**  **@PropertySource("classpath:/com/glxt/study/spring/annotation/propertysource/app.properties")**  **public class AppConfig**  **{**  **@Autowired**  **Environment environment;**    **@Bean**  **public Car getCar()**  **{**  **Car car = new Car();**  **car.setName(environment.getProperty("car.name"));**  **car.setPrice(Double.*parseDouble*(environment.getProperty("car.price")));**  **return car;**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);**  **Car car = ctx.getBean(Car.class);**  **System.*out*.println(car);**  **}**  **}** |

### 2.12 ApplicationEvent与ApplicationListener

Event handling in the ApplicationContext is provided through the **ApplicationEvent** class and **ApplicationListener** interface。Essentially, this is the standard Observer design pattern

#### 1、通过implement ApplicationListener方式处理事件

|  |
| --- |
| **public class EmailEvent extends ApplicationEvent**  **{**  **private String address;**  **private String text;**    **public EmailEvent(Object source)**  **{**  **super(source);**  **}**    **public EmailEvent(Object source, String address, String text)**  **{**  **super(source);**  **this.address = address;**  **this.text = text;**  **}**    **public void print()**  **{**  **System.*out*.println("hello spring event!");**  **}**  **public String getAddress()**  **{**  **return address;**  **}**  **public void setAddress(String address)**  **{**  **this.address = address;**  **}**  **public String getText()**  **{**  **return text;**  **}**  **public void setText(String text)**  **{**  **this.text = text;**  **}**  **}** |

|  |
| --- |
| **@Configuration**  **public class EmailListener implements ApplicationListener<EmailEvent>**  **{**  **public void onApplicationEvent(EmailEvent emailEvent)**  **{**  **emailEvent.print();**  **System.*out*.println("the source is:"+emailEvent.getSource());**  **System.*out*.println("the address is:"+emailEvent.getAddress());**  **System.*out*.println("the email's context is:"+emailEvent.getText());**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(EmailListener.class);**  **EmailEvent event = new EmailEvent("hello","boylmx@163.com","this is a email text!");**  **EmailListener emailListener = new EmailListener();**  **// 3、发布事件**  **ctx.publishEvent(event);**  **}**  **}** |

#### 2、通过@EventListener注解方式处理事件

|  |
| --- |
| **@Configuration**  **public class EmailNotifier**  **{**  **@EventListener**  **public void processEmailEvent(EmailEvent emailEvent)**  **{**  **emailEvent.print();**  **System.*out*.println("the source is:"+emailEvent.getSource());**  **System.*out*.println("the address is:"+emailEvent.getAddress());**  **System.*out*.println("the email's context is:"+emailEvent.getText());**  **}**    **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new AnnotationConfigApplicationContext(EmailNotifier.class);**  **EmailEvent event = new EmailEvent("hello","boylmx@163.com","this is a email text!");**  **EmailNotifier emailNotifier = new EmailNotifier();**  **// 3、发布事件**  **ctx.publishEvent(event);**  **}**  **}** |

### 2.13 BeanFactory or ApplicationContext?

Use an **ApplicationContext** unless you have a good reason for not doing so.

The **BeanFactory** provides the underlying basis for Spring’s IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring.



## 第三章 spring概览

3.1 使用场景

3.2 使用技术

依赖注入和控制反转

3.3 模块

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test, as shown in the following diagram.



### 3.3.1 Core Container

The [Core Container](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction) consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules [provide the fundamental parts of the framework](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction), including the IoC and Dependency Injection features.

The [Context](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#context-introduction) (spring-context) module builds on the solid base provided by the [Core and Beans](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction) modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry.

spring-context-support provides support for integrating common third-party libraries into a Spring application context for caching (EhCache, Guava, JCache), mailing (JavaMail), scheduling (CommonJ, Quartz) and template engines (FreeMarker, JasperReports, Velocity).

The spring-expression module provides a powerful [Expression Language](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#expressions) for querying and manipulating an object graph at runtime.

### 3.3.2 AOP and Instrumentation

The spring-aop module provides an [AOP](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop-introduction) Alliance-compliant aspect-oriented programming

The separate spring-aspects module provides integration with AspectJ

The spring-instrument module provides class instrumentation support and classloader implementations

The spring-instrument-tomcat module contains Spring’s instrumentation agent for Tomcat

### 3.3.3 Messaging

Spring Framework 4 includes a spring-messaging module with key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications.

### 3.3.4 Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

The spring-jdbc module provides a [JDBC](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jdbc-introduction)-abstraction layer

The spring-tx module supports [programmatic and declarative transaction](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#transaction) management for classes

The spring-orm module provides integration layers for popular [object-relational mapping](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-introduction) APIs, including [JPA](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-jpa), [JDO](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-jdo), and [Hibernate](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-hibernate).

The spring-oxm module provides an abstraction layer that supports [Object/XML mapping](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#oxm) implementations such as JAXB, Castor, XMLBeans, JiBX and XStream

The spring-jms module ([Java Messaging Service](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms)) contains features for producing and consuming messages.

### 3.3.5 Web

The Web layer consists of the spring-web, spring-webmvc, spring-websocket, and spring-webmvc-portlet modules.

### 3.3.6 Test

The spring-test module supports the [unit testing](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#unit-testing) and [integration testing](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#integration-testing) of Spring components with JUnit or TestNG

## 第四章 专题

### 4.1 resource

内置的资源实现（built-in resource implementations）

#### 4.1.1、种类

* UrlResource

The UrlResource wraps a java.net.URL, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc.

All URLs have a standardized String representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes file: for accessing filesystem paths, http: for accessing resources via the HTTP protocol, ftp: for accessing resources via FTP, etc

* ClassPathResource

This class represents a resource which should be obtained from the classpath.

* FileSystemResource

This is a Resource implementation for java.io.File handles. It obviously supports resolution as a File, and as a URL.

* ServletContextResource

This is a Resource implementation for ServletContext resources, interpreting relative paths within the relevant web application’s root directory.

* InputStreamResource

A Resource implementation for a given InputStream. This should only be used if no specific Resource implementation is applicable.

* ByteArrayResource

This is a Resource implementation for a given byte array. It creates a ByteArrayInputStream for the given byte array.

#### 4.1.2 The ResourceLoader

The ResourceLoader interface is meant to be implemented by objects that can return (i.e. load) Resource instances.

[通过Spring Resource接口获取资源](http://elim.iteye.com/blog/2016305)

**注意：**Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

### 4.2 使用spring面向切面编程（Aspect Oriented Programming with Spring）

#### 4.2.1 AOP概念

概念参考：

[面向切面编程](http://www.360doc.com/content/14/0107/10/15272201_343250604.shtml)

切面（Aspect）：其实就是共有功能的实现。如日志切面、权限切面、事务切面等。在实际应用中通常是一个存放共有功能实现的普通Java类，之所以能被AOP容器识别成切面，是在配置中指定的。

通知（Advice）：是切面的具体实现。以目标方法为参照点，根据放置的地方不同，可分为前置通知（Before）、后置通知（AfterReturning）、异常通知（AfterThrowing）、最终通知（After）与环绕通知（Around）5种。在实际应用中通常是切面类中的一个方法，具体属于哪类通知，同样是在配置中指定的。

连接点（Joinpoint）：就是程序在运行过程中能够插入切面的地点。例如，方法调用、异常抛出或字段修改等，但Spring只支持方法级的连接点。

切入点（Pointcut）：用于定义通知应该切入到哪些连接点上。不同的通知通常需要切入到不同的连接点上，这种精准的匹配是由切入点的正则表达式来定义的。

目标对象（Target）：就是那些即将切入切面的对象，也就是那些被通知的对象。这些对象中已经只剩下干干净净的核心业务逻辑代码了，所有的共有功能代码等待AOP容器的切入。

代理对象（Proxy）：将通知应用到目标对象之后被动态创建的对象。可以简单地理解为，代理对象的功能等于目标对象的核心业务逻辑功能加上共有功能。代理对象对于使用者而言是透明的，是程序运行过程中的产物。

织入（Weaving）：将切面应用到目标对象从而创建一个新的代理对象的过程。这个过程可以发生在编译期、类装载期及运行期，当然不同的发生点有着不同的前提条件。譬如发生在编译期的话，就要求有一个支持这种AOP实现的特殊编译器；发生在类装载期，就要求有一个支持AOP实现的特殊类装载器；只有发生在运行期，则可直接通过Java语言的反射机制与动态代理机制来动态实现。

引入（Introduction）：添加方法或字段到被通知的类。Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现IsModified接口，来简化缓存。

个人理解，如下：

Aspect

Advice

Join point

Join point

…

Pointcut

织入

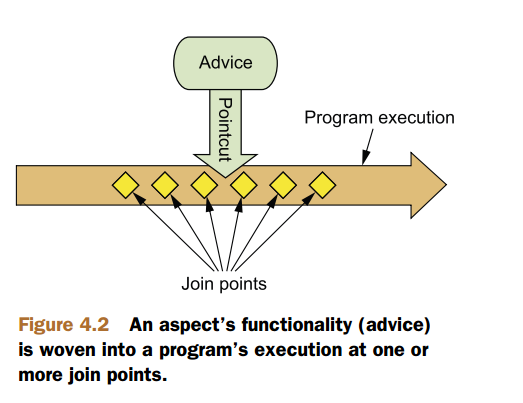
Target object

AOP proxy

1、传入

2、筛选

3、增强



关系理解：

Aspect与advice

* In AOP terms, the job of an aspect is called advice.
* Advice defines both the what and the when of an aspect. In addition to describing  
  the job that an aspect will perform, advice addresses the question of when to perform the job。

Aspect与pointcuts

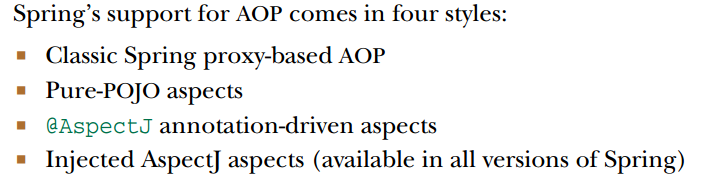
* If advice defines the what and when of aspects, then pointcuts define the where.
* A pointcut definition matches one or more join points at which advice should be woven.

##### 经典好文：

[**AOP 那点事儿**](https://my.oschina.net/huangyong/blog/161338)

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans).

##### Spring support styles



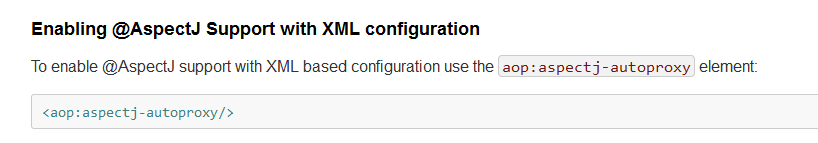
The first three styles are all variations on Spring’s own AOP implementation. Spring AOP is built around dynamic proxies. Consequently, Spring’s AOP support is limited to method interception.

(前三种方式是spring 自己实现的AOP方式，因此只能拦截方法)。

#### 4.2.2 @AspectJ support

1、Enabling @AspectJ support

@AspectJ support 能够通过两种方式配置：XML、Java style。





##### @AspectJ support with XML configuration

**Aspect定义：aspect相关的(what，when)增强advice，以及(where)切入点pointcut。**

|  |
| --- |
| **// 声明 aspect**  **@Aspect**  **public class Logging**  **{**  **// 声明pointcut**  **@Pointcut("execution(\* com.glxt.study.spring.aop.simpledemo.\*.\*(..))")**  **private void selectAll(){}**    **// 声明advice**  **// 前置增强**  **@Before("selectAll()")**  **public void beforeAdvice()**  **{**  **System.*out*.println("Going to setup student profile.");**  **}**  **}** |

**Student，其中方法作为join point。**

|  |
| --- |
| **public class Student**  **{**  **private Integer age;**  **private String name;**  **public Integer getAge()**  **{**  **System.*out*.println("Age : " + age );**  **return age;**  **}**  **public void setAge(Integer age)**  **{**  **this.age = age;**  **}**  **public String getName()**  **{**  **System.*out*.println("Name : " + name );**  **return name;**  **}**  **public void setName(String name)**  **{**  **this.name = name;**  **}**  **}** |

**配置文件**

|  |
| --- |
| **<?xml version=*"1.0"* encoding=*"UTF-8"*?>**  **<beans xmlns=*"http://www.springframework.org/schema/beans"***  **xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"***  **xmlns:p=*"http://www.springframework.org/schema/p"* xmlns:aop=*"http://www.springframework.org/schema/aop"***  **xsi:schemaLocation=*"http://www.springframework.org/schema/beans***  ***http://www.springframework.org/schema/beans/spring-beans.xsd***  ***http://www.springframework.org/schema/aop***  ***http://www.springframework.org/schema/aop/spring-aop.xsd"*>**    **<aop:aspectj-autoproxy />**    **<bean id=*"student"* class=*"com.glxt.study.spring.aop.simpledemo.Student"*>**  **<property name=*"name"* value=*"Zara"* />**  **<property name=*"age"* value=*"11"*/>**  **</bean>**    **<bean id=*"logging"* class=*"com.glxt.study.spring.aop.simpledemo.Logging"*>**  **</bean>**  **</beans>** |

**测试**

|  |
| --- |
| **public class Test**  **{**  **public static void main(String[] args)**  **{**  **ApplicationContext ctx = new ClassPathXmlApplicationContext("com/glxt/study/spring/aop/simpledemo/beans.xml");**  **Student student = ctx.getBean(Student.class,"student");**  **student.getName();**  **student.getAge();**  **}**  **}** |

##### @AspectJ support with java configuration

#### 4.2.3 Schema-based AOP support

# 第二部分 概述揭秘

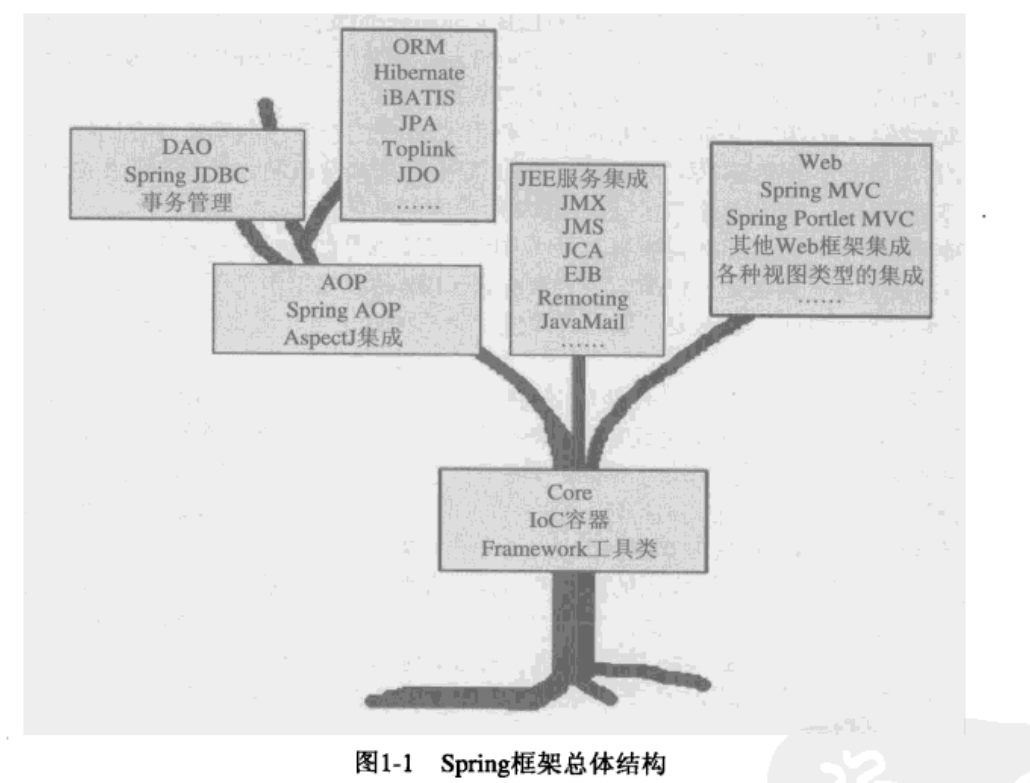
对于读者来说，须知万物溯其源，学东西不是为学而学，最初是有个场景、问题，然后是解决方案的提出、选择、权衡、实施、反思和改进。

## 2.1 spring的来世今生

Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它的最初目的主要是为了简化Java EE的企业级应用开发，相对于过去EJB时代重量级的企业应用开发而言。

## 2.2 spring框架概述





组成整个spring框架的各种服务实现被划分到了多个相互独立却又相互依赖的模块当中。正如我们在图1-1中所见到的那样，这些模块组成了spring生命之树的枝和干，说白了也就是它们组成了spring框架的核心骨架。抓住了这副骨架，也就抓住了spring框架的学习主线。

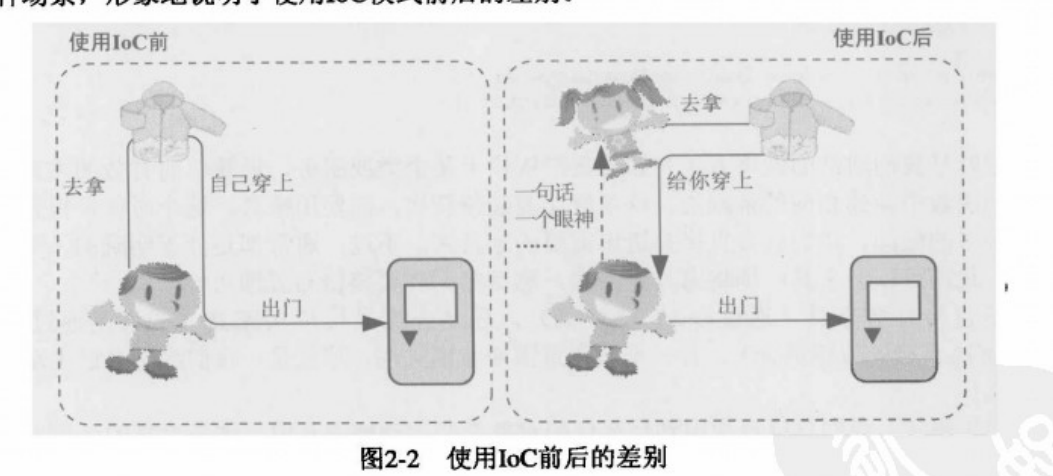
# 第三部分 核心技术

## 源头活水---spring中的设计模式

## Spring 的IoC容器

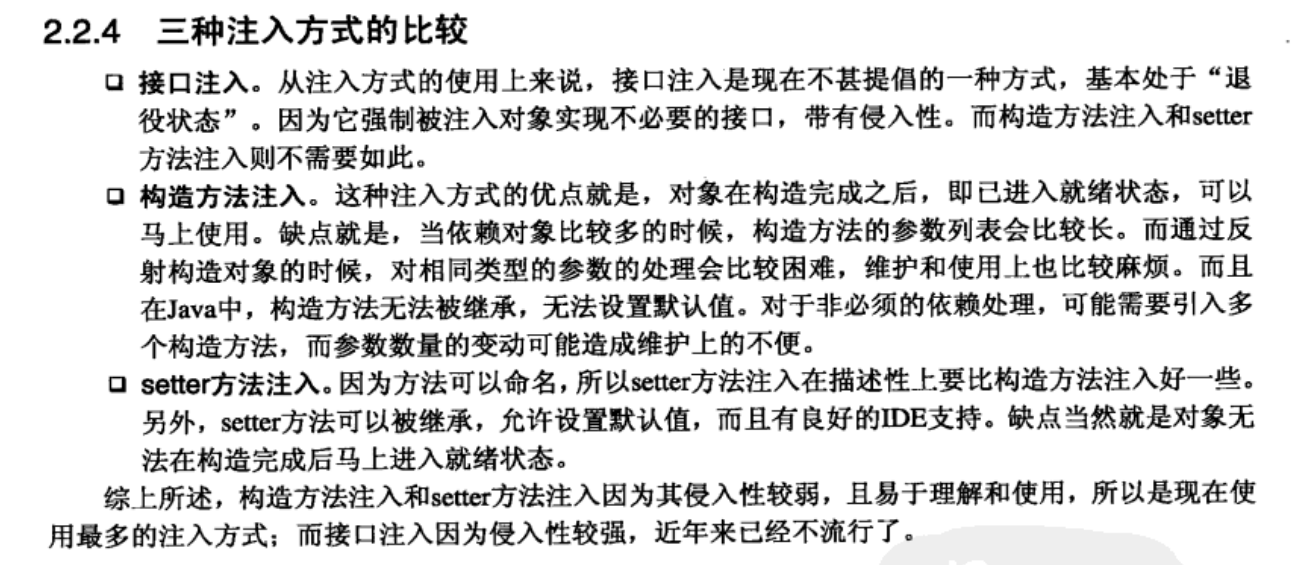
### IOC基本概念

#### 1、让别人为我所用

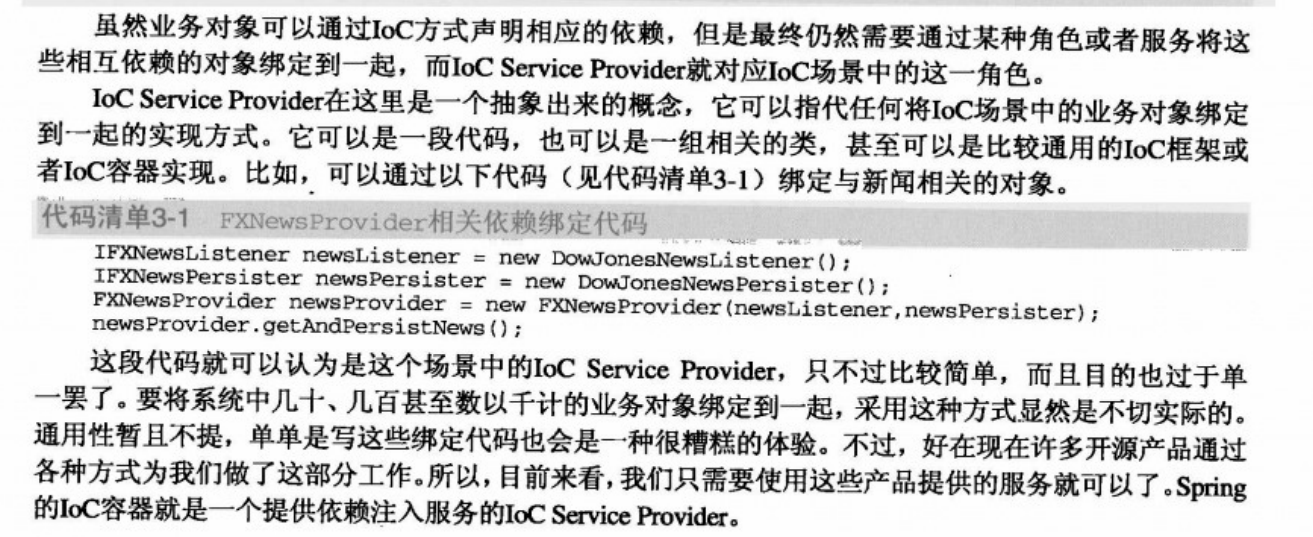


#### 2、三种依赖注入方式

* 构造方法注入
* Setter方法注入
* 接口注入



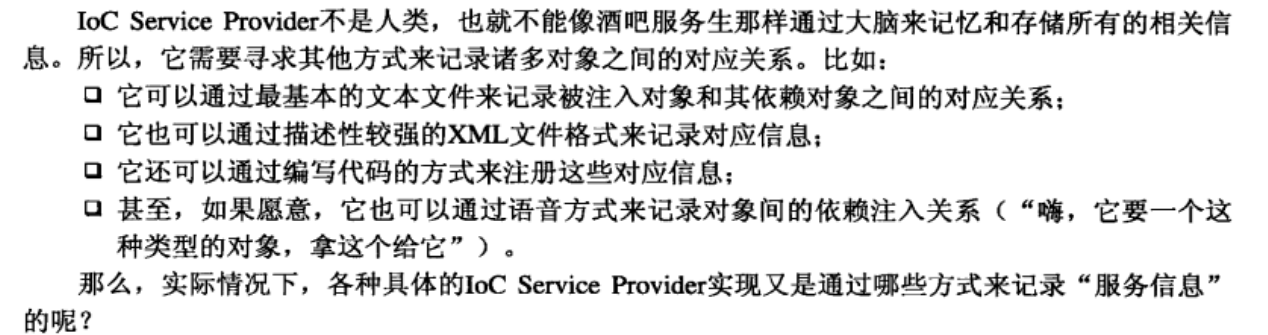
### IOC核心管理者IoC Service Provider



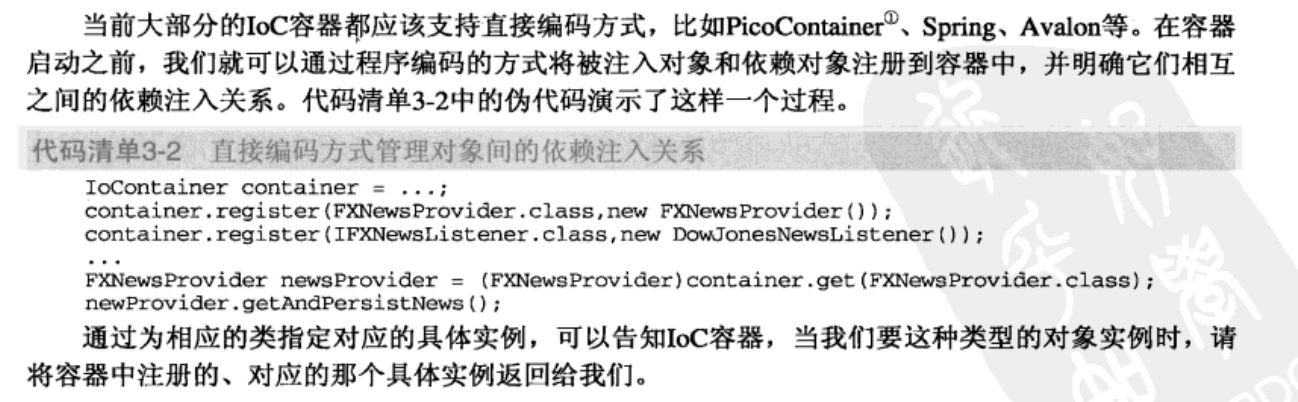
#### IoC Service Provider 的职责

主要有两个：业务对象的构建管理、业务对象间的依赖绑定。

#### IoC Service Provider如何管理对象间的依赖关系



##### 直接编码方式



所以，通过程序编码让最终的IoC Service Provider（也就是各个IoC框架或者容器实现）得以知晓服务的“奥义”，应该是管理依赖绑定关系的最基本方式。

##### 配置文件方式

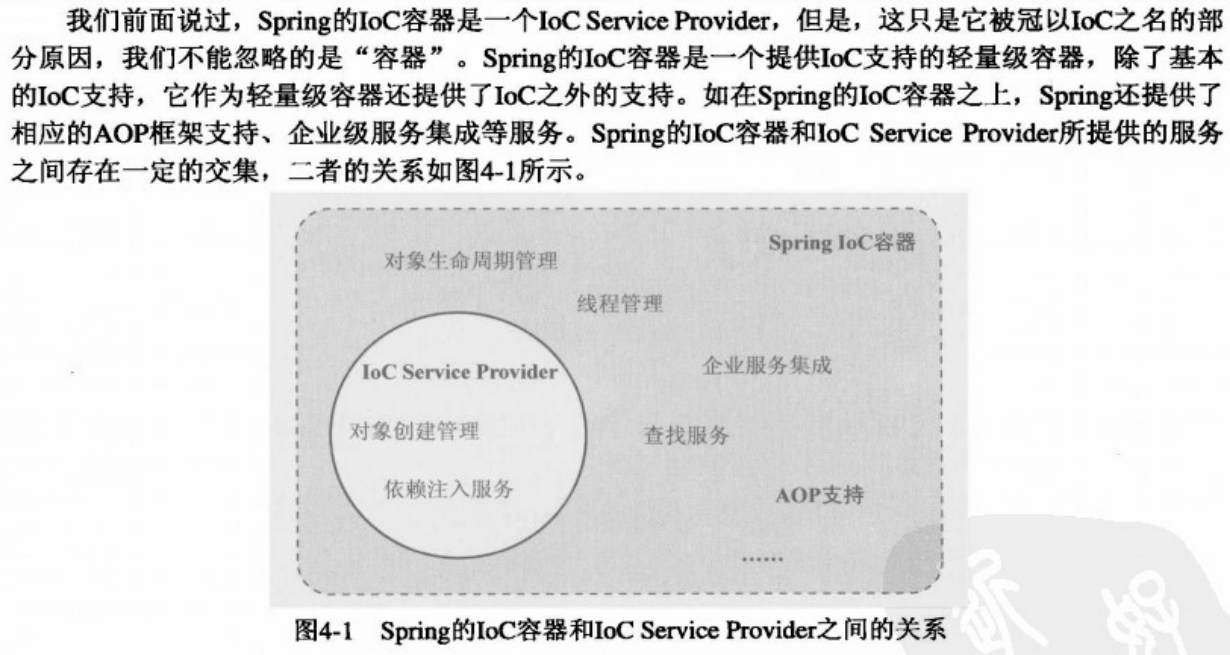


##### 元数据方式



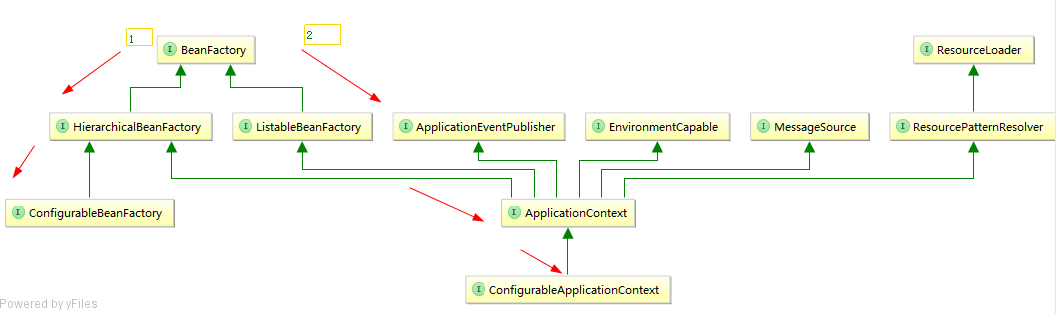
当然，注解最终也要通过代码处理来确定最终的注入关系，从这点来说，注解方式可以算作编码方式的一种特殊情况。

### Spring的IoC容器之BeanFactory



Spring提供了两种容器类型：BeanFactory和ApplicationContext.

* BeanFactory：基础类型IoC容器，提供完整的IoC服务支持。如果没有特殊指定，默认采用延迟初始化策略（lazy-load）。对于资源有限，并且功能要求不是很严格的场景，BeanFactory是比较合适的IoC容器选择。
* ApplicationContext：ApplicationContext是在BeanFactory的基础上构建，是相对比较高级的容器实现，除了拥有BeanFactory的所有支持，ApplicationContext还提供了其他高级特性，比如事件发布、国际化信息支持等。ApplicationContext所管理的对象，在该类型容器启动之后，默认全部初始化并绑定完成。所以，相对于BeanFactory来说，ApplicationContext要求更多的系统资源，同时，因为在启动时就完成所有初始化，容器启动时间较之BeanFactory也会长一些。在那些系统资源充足，并且要求更多场景中，ApplicationContext类型的容器是比较合适的选择。



BeanFactory和ApplicationContext继承关系

BeanFactory的代码定义

|  |
| --- |
| **public interface** BeanFactory {  String ***FACTORY\_BEAN\_PREFIX*** = **"&"**; Object getBean(String name) **throws** BeansException; <T> T getBean(String name, Class<T> requiredType) **throws** BeansException; <T> T getBean(Class<T> requiredType) **throws** BeansException; Object getBean(String name, Object... args) **throws** BeansException;<T> T getBean(Class<T> requiredType, Object... args) **throws** BeansException;  **boolean** containsBean(String name); **boolean** isSingleton(String name) **throws** NoSuchBeanDefinitionException;  **boolean** isPrototype(String name) **throws** NoSuchBeanDefinitionException;  **boolean** isTypeMatch(String name, ResolvableType typeToMatch) **throws** NoSuchBeanDefinitionException;  **boolean** isTypeMatch(String name, Class<?> typeToMatch) **throws** NoSuchBeanDefinitionException;  Class<?> getType(String name) **throws** NoSuchBeanDefinitionException;  String[] getAliases(String name); } |

#### 使用

#### BeanFactory的对象注册与依赖绑定方式

##### 直接编码方式

1、将bean定义注册到容器中

2、指定依赖关系

3、完成绑定

##### 外部配置文件方式

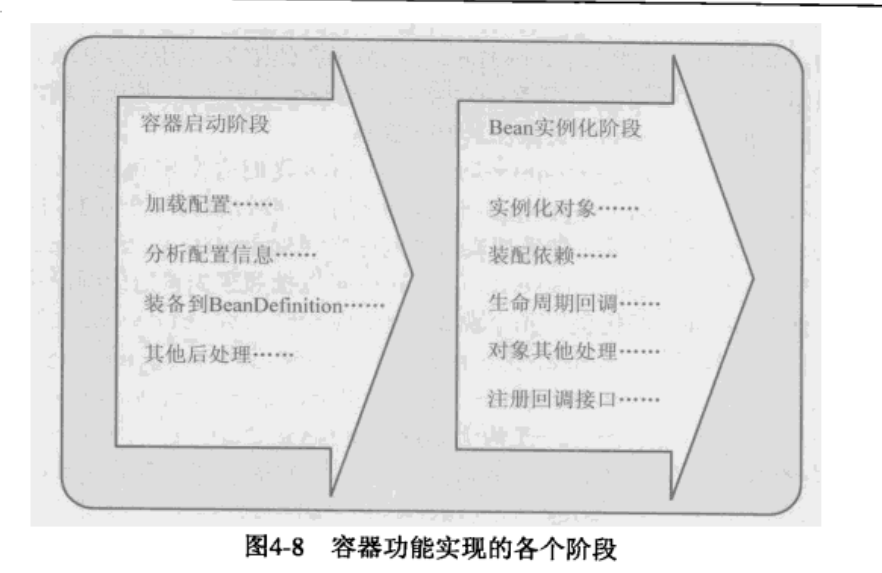
Spring的IoC容器支持两种配置文件格式：properties文件格式和XML文件格式。

##### 注解方式

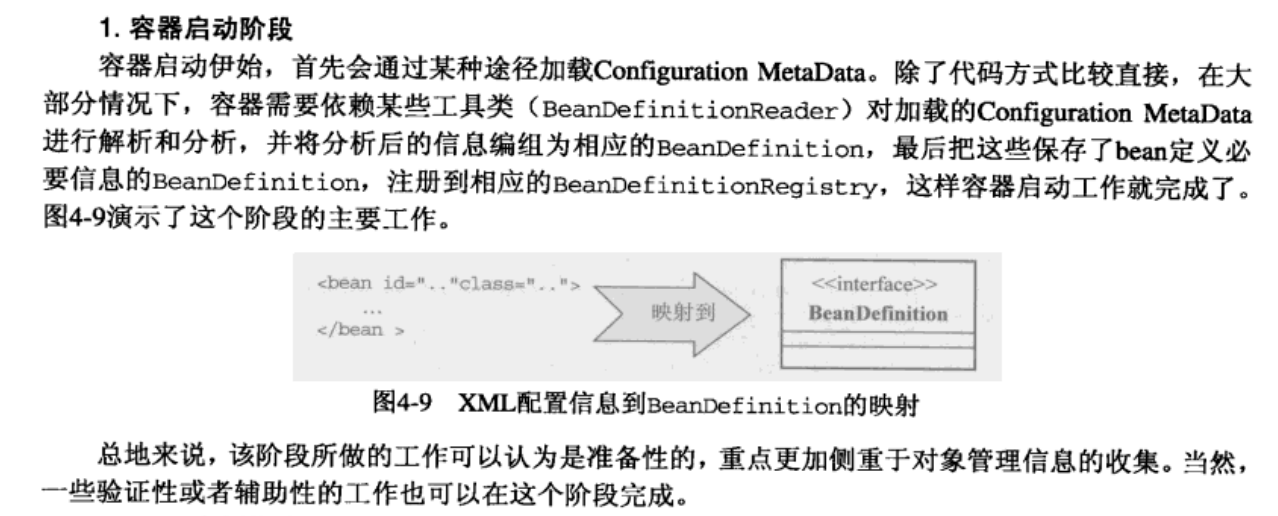
### 容器背后的秘密

Spring的IoC容器会以某种方式加载ConfigurationMetadata（通常也就是XML格式的配置信息），然后根据这些信息绑定整个系统的对象，最终组装成一个可用的基于轻量级容器的应用系统。

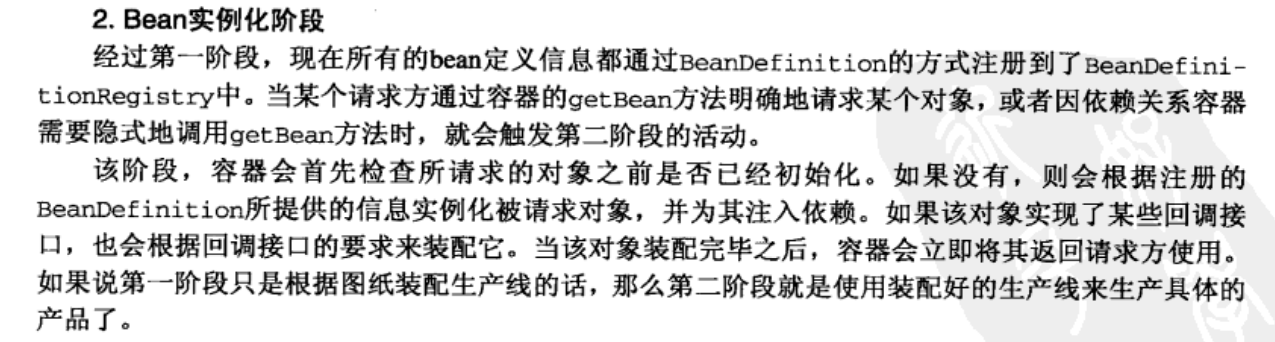
Spring的IoC容器实现以上功能的过程，基本上可以按照类似的流程划分为两个阶段，即容器启动阶段和Bean实例化阶段。



#### 容器启动阶段



#### Bean实例化阶段



#### 插手“容器的启动”

#### 了解bean的一生

#### Spring的IoC核心源码学习

参考：

[Spring IOC核心源码学习](http://yikun.github.io/2015/05/29/Spring-IOC%E6%A0%B8%E5%BF%83%E6%BA%90%E7%A0%81%E5%AD%A6%E4%B9%A0/)

1、初始化

* 准备
* 读取
* 解析
* 注册

2、注入依赖

资源的定位

AbstractBeanDefinitionRead:

|  |
| --- |
| **public int** loadBeanDefinitions(String location, Set<Resource> actualResources)  {  ResourceLoader resourceLoader = getResourceLoader();  // 资源定位  Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);  或者是下面  Resource resource = resourceLoader.getResource(location);  } |

BeanDefinition信息载入

对IoC容器来说，这个载入过程，相当于把定义的BeanDefinition在IcO容器中转化成一个spring内部表示的数据结构的过程。IoC容器对Bean的管理和依赖注入功能的实现时通过对其持有的BeanDefinition进行各种相关操作来完成。这些BeanDefinition数据在IoC容器中通过一个HashMap来保持和维护。

XmlBeanDefinitionReader

|  |
| --- |
| **public int** loadBeanDefinitions(EncodedResource encodedResource)  {  InputSource inputSource = **new** InputSource(inputStream);  **return** doLoadBeanDefinitions(inputSource, encodedResource.getResource());  }  **protected int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)  {  // 这里取得XML文件的Document对象  Document doc = doLoadDocument(inputSource, resource);  // 这里启动的是对BeanDefinition解析的详细过程  **return** registerBeanDefinitions(doc, resource);  } |

### IOC容器设计与实现

引入容器的目的是为了解决对象生命周期管理中所遇到的问题，那么这一额外的编程元素自身的设计也必须遵循一定的原则：

* **容器应该被设计成一个全局的、统一的编程元素**；

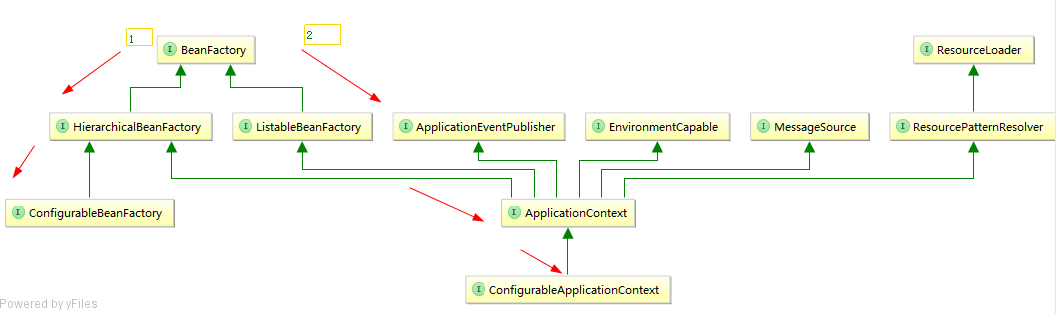
这一点几乎无须多做解释。在运行期获得对象实例可能会发生在程序的任何一个角落，因此容器作为一个辅助元素也可能随时在任何地方被调用。这就不得不要求容器对象是一个全局对象。

* **在最大程度上降低容器对业务逻辑的入侵**；

这是进行容器设计时需要考虑的一个最重要的问题。因为我们引入容器这个额外元素来管理对象的生命周期的初衷之一，就是由于我们希望能够把对象生命周期管理的部分从具体的业务逻辑中提取出来，使得业务逻辑本身能够方便地进行单元测试。

* **容器应该提供简单而全面的对象操作接口**；

引入容器的目的就是为了进行对象操作。这里的简单而全面是指，一方面针对对象的操作接口应该一目了然；另外一方面，针对对象的操作接口应该涵盖对象生命周期管理的所有内容。



分两条主线走，BeanFactory、ApplicationContext。

以ApplicationContext路线，它从两个方面着手，

* 一方面是继承BeanFactory接口体系中的HierarchicalBeanFactory、ListableBeanFactory等，具备BeanFactory IoC容器的基本功能;
* 另一方面，通过继承ApplicationEventPublisher、EnvironmentCapable、MessageSource、ResourceLoader这些接口，获取更高级的IoC容器特性。