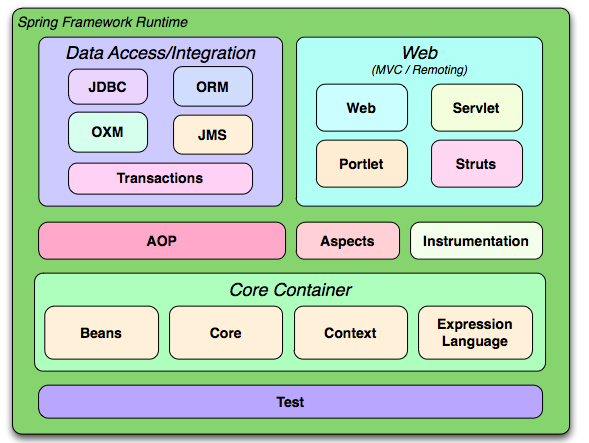
**Spring结构图**



**Spring三种配置**

* xml显示配置bean：
* java代码配置bean：
* 自动化装配bean：适用于自己编写的JavaBean，当需要注入第三方类库时，需要使用上面两种方式

**1、自动化装配**

首先配置扫描包：xml配置和java代码配置

1）xml配置：

<?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"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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-4.1.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd">

<!-- 配置扫描 -->

<context:component-scan base-package="com.wangdh.spring.demo"/>

</beans>

Java代码配置：

@Configuration

@ComponentScan(basePackages = "com.wangdh.spring.demo")

public class SpringJavaConfig {

}

2）为需要装载的bean加@Component注解

3）在需要实例化的地方加@Autowired注解，可以加载构造函数、setter、方法和字段上

**2、java代码配置**

1）兴建一个JavaConfig类,该类需要添加注解：@Configuration

2）为每个需要注入的对象创建一个方法，该方法需要添加注解：@Bean并返回注入对象的实例

@Bean

**public** Cat cat(){

**return** **new** Cat();

}

3、xml配置：对旧的项目的xml进行维护，新的项目建议适用java代码配置和自动装置

**Spring 常用配置**

**1.Bean的作用域：@Scope**

Singleton：单例模式，Spring的默认配置

Prototype：每次调用新建一个Bean实例

Request：web项目中，每个http请求新建一个Bean实例

Session：web项目中，每个http会话新建一个Bean实例

GlobalSession：portal应用中有用

**2.Bean的初始化和销毁**

Java配置方式：initMethod和destroyMethod

xml配置方式：init-method和destroy-method

注解方式：@PostConstruct和@PreDestroy

**Spring 注解注入**

在Java代码中使用@Autowired或@Resource注解方式进行装配，@Autowired默认按类型装配, @Resource默认按名称装配, 当找不到与名称匹配的Bean才会按类型匹配. @Autowired注解是按类型装配依赖对象, 默认情况下它要求依赖对象必须存在, 如果允许null值, 可以设置required=false。如果想使用按名称装配, 可以结合@Qualifier注解一起使用

spring2.5为我们引入了组件自动扫描机制, 它可以在类路径下寻找标注了@Component、@Controller、@Service、@Reponsitory注解的类， 并把这些类纳入进spring容器中管理.

@Controller通常用于标注控制层组件(如struts中的action);

@Service通常用于标注业务层组件;

@Repository通常用于标注数据访问组件, 即DAO组件;

@Component泛指组件, 当组件不好归类的时候, 我们可以使用这个注解进行标注;

**注入bean的注解**

@Autowired:Spring提供的注解

@Inject:JSR-330提供的注解

@Resource:JSR-250提供的注解

**Spring 容器**

* ClassPathXmlApplicationContext:从类路径中加载。
* FileSystemXmlApplicationContext:从文件系统加载。
* XmlWebApplicationContext:从web系统中加载。

bean工厂:最简单的容器，提供了基础的依赖注入支持。创建各种类型的Bean.

应用上下文:建立在bean工厂基础之上，提供系统架构服务。ApplicationCotext,spring更加高级的容器。功能强大：

1.提供文本信息解析工具，包括对国际化支持。

2.提供载入文件资源的通用方法，如图片。

3.可以向注册为监听器的bean发送事件

在web应用程序中，需要在web.xml文件中添加下面配置：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:spring-mybatis.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

**Spring Aware接口**

Spring的依赖注入的最大亮点是所有的Bean对Spring容器没有意识，即可以将Spring容器替换成其他容器。

当需要用到Spring容器本身的功能资源，这是Bean就需要知道Spring容器的存在，才能使用Spring容器提供的资源，这就是所谓的Spring Aware

Spring提供的Aware接口：

BeanNameAware：获得容器中Bean的名称

BeanFactoryAware：获得当前Bean factory

ApplicationContextAware：获得当前的Application context，集成了下面三个服务

MessageSourceAware：获得message source

ApplicationEventPublisherAware：应用事件发布器，可以发布事件

ResourceLoaderAware：获得资源加载器，可以获得外部资源文件

**Spring 多线程**

**@Enable\*注解**

@EnableAspectJAutoProxy：开启对AspectJ自动代理的支持

@EnableAsync：开启异步方法的支持

@EnableScheduling：开启计划任务的支持

@EnableWebMvc：开启web MVC的配置支持

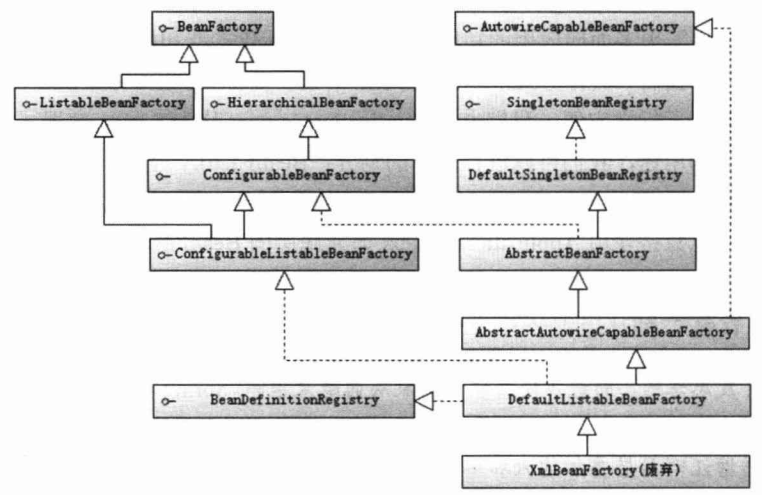
@EnableConfigurationProperties：开启对@ConfigurationProperties注解配置bean的支持

@EnableJpaRepositories：

@EnableTransactionManagement：开启注解式的事务支持

@EnableCaching：开启注解式的缓存支持

**BeanFactory类继承体系**



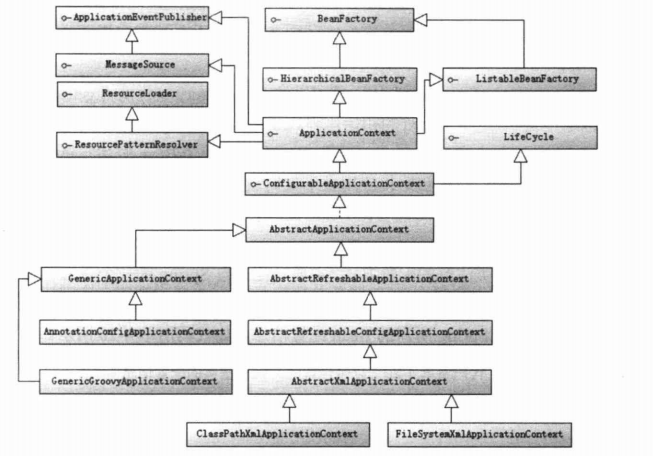
BeanFactory接口位于类结构树的顶端，主要的方法是getBean(String beanName),返回特定名称的Bean。BeanFactory的功能通过其他接口得到不断扩展：

1. ListableBeanFactory：定义了访问Bean基本信息的方法，如：查看Bean的个数，获取某一类型Bean的配置名，查看容器中是否包含某一Bean等。
2. HierarchicalBeanFactory：父子级联IoC容器接口，子类可以通过接口方法访问父容器；
3. ConfigurableBeanFactory：增强IoC容器的可定制性，定义了设置类装载器，属性编辑器，容器初始化后置处理器等方法；
4. AutowireCapableBeanFactory：定义了将容器中的Bean按某种规则进行自动装配的方法；
5. SingletonBeanFactory：定义了运行在运行期向容器注册单例Bean的方法；
6. BeanDefinitionRegistry：Spring配置文件每一个<bean>节点元素在Spring容器里都通过一个BeanDefinition对象表示，描述了Bean的配置信息；而BeanDefinitionRegistry接口提供了向容器手动注册BeanDefinition对象的方向。

>> 通过BeanFactory启动IOC容器时，并不会初始化配置文件中定义的Bean，初始化动作发生在第一个调用.

>> 初始化BeanFactory时，必须为其提供一种日志框架，否则启动会报错。

**ApplicationContext类结构体系**



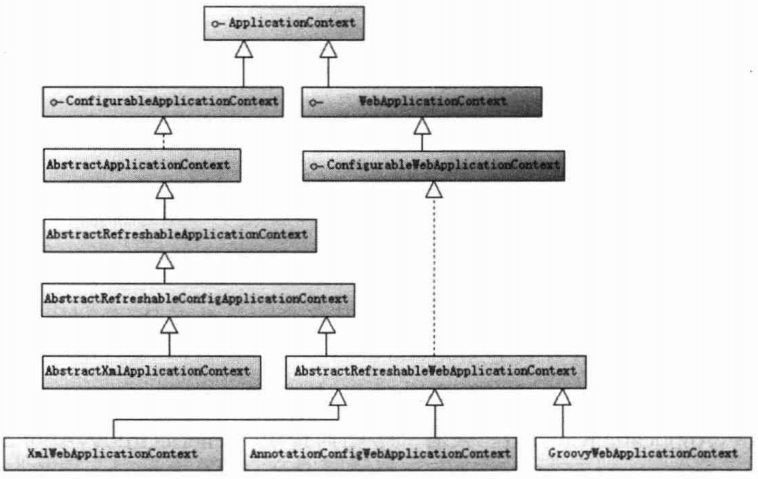
ApplicationContext的主要实现类是：ClassPathXmlApplicationContext、FileSystemXmlApplicationContext、AnnotationConfigApplicationContext。

ApplicationContext继承了HierarchicalBeanFactory和ListableBeanFactory接口，在此基础上，通过其他多个接口扩展了BeanFactory的功能：

1. ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括启动事件，关闭事件等，实现了ApplicationListener事件监听接口的Bean可以接受到容器事件，并对事件进行相应处理。
2. MessageSource：为应用提供i18n国际化消息访问的功能
3. ResourcePatternResolver：所有ApplicationContext实现类都实现了类似于PathMatchingResourcePatternResolver的功能，可以通过带前缀的Ant风格的资源文件路径装载Spring的配置文件
4. LifeCycle：提供了start()和Stop()两个方法，主要用于控制异步处理过程。
5. ConfigurableApplicationContext：扩展于ApplicationContext，新增了两个主要的方法：refresh()和close()让ApplicationContext具有启动，刷新和关闭应用上下文的能力。

>> ApplicationContext在初始化应用上下文时就实例化所有单例的Bean

**WebApplicationContext类体系结构**



WebApplicationContext的初始化方式和BeanFactory、ApplicationContext有所不同，因为WebApplicationContext需要ServletContext实例，即必须在拥有Web容器的前提下才能完成启动。

在web.xml中配置自启动的Servlet或定义Web容器监听器(ServletContextListener)，借助两者中的任何一个，就可以完成启动Spring Web应用上下文的工作。

Spring提供用于启动WebApplicationContext的Servlet和Web容器监听器：

org.springframework.web.context.ContextLoaderServlet(**3.0已移除**)

org.springframework.web.context.ContextLoaderListener

Web容器监听器启动：

<!-- 配置Spring配置文件路径 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:spring-root.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

使用Java代码配置WebApplicationContext而非Xml配置：

<!-- 通过指定contextClass参数，让Spring启动AnnotationConfigWebApplicationContext -->

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.xxx.AppConfig</param-value>

</context-param>

<listener>

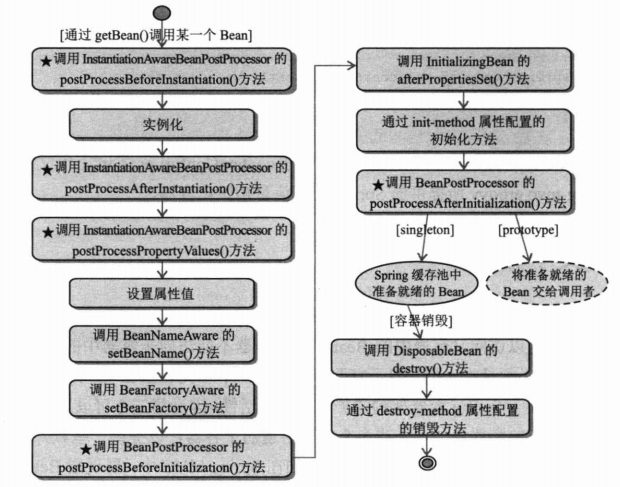
<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

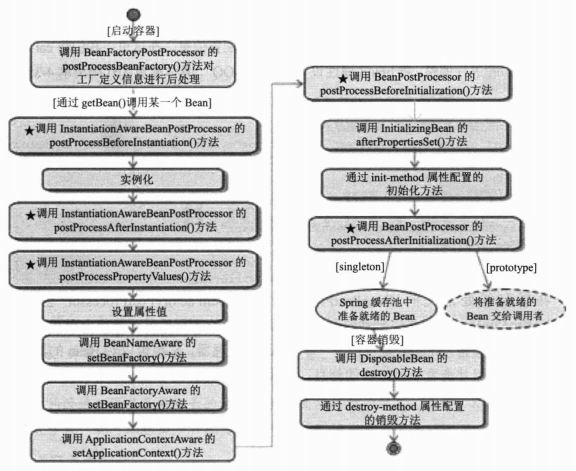
**BeanFactory中Bean的生命周期**

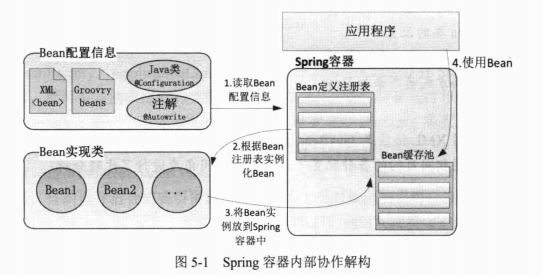


上面涉及的方法大致分为4类：

1. Bean自身的方法：构造函数、Setter设置属性及通过init-method和destroy-method指定的方法；
2. Bean级生命周期接口方法：如BeanNameAware、BeanFactoryAware、InitializingBean和DisposableBean
3. 容器级生命周期接口方法：InstantiationAwareBeanPostProcessor和BeanPostProcessor
4. 工厂后处理接口方法：AspectJWeAVingEnabler、CustomAutowireConfigurer、ConfigurationClassPostProcessor

**ApplicationContext中Bean的生命周期**







## Spring容器高级主题

### 使用外部属性文件：PropertyPlaceholderConfigurer

引入外部属性的两种方式：

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"

p:location="classpath:jdbc.properties"

p:fileEncoding="utf-8"/>

<context:propery-placeholder location="classpath:jdbc.properties"/>

使用外部属性：

<bean class = "com.wangdh.spring.propertyPlaceHolderConfigure.JdbcConfig"

p:url="${jdbc.url}"

p:classDriverName="${jdbc.classDriverName}"

p:password="${jdbc.password}"

p:userName="${jdbc.userName}"/>

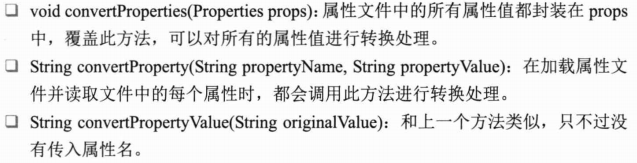
@Value("${jdbc.url}")

**private** String url;

使用加密的属性文件

PropertyPlaceholderConfigurer继承自PropertyResourceConfigurer

后者有几个protected方法，用于在属性使用之前对属性列表中的值进行转换：



### 国际化信息

Java的本地化对象：

// 获取指定地区，指定语言

Locale **locale1** = **new** Locale("zh", "CN");

// 获取指定语言，系统默认地区

Locale **locale2** = **new** Locale("zh");

Locale **locale4** = Locale.***CHINESE***;

// 获取指定地区，系统默认语言

Locale **locale3** = Locale.***CHINA***;

// 获取系统默认地区和语言

Locale **locale5** = Locale.*getDefault*();

Java与本地化相关的格式化类：NumberFormat、DateFormat、MessageFormat

Locale **en\_US** = **new** Locale("en", "US");

Locale **zh\_CN** = **new** Locale("zh", "CN");

NumberFormat **currFormat** = NumberFormat.*getCurrencyInstance*(en\_US);

**double** **amt** = 123456.78;

System.***out***.println(currFormat.format(amt));

Date **date** = **new** Date();

DateFormat **dateFormat** = DateFormat.*getDateInstance*(DateFormat.***MEDIUM***,zh\_CN);

System.***out***.println(dateFormat.format(date));

String **pattern** = "{0},您好！您于{1}在工商银行存入{2}元。";

String **pattern2** = "At {1,time,short} On {1,date,long},{0} paid {2,number,currency}.";

MessageFormat **messageFormat** = **new** MessageFormat(pattern2, en\_US);

Object[] **params** = {"John",**new** GregorianCalendar().getTime(),1.0E3};

String **msg1** = MessageFormat.*format*(pattern, params);

System.***out***.println(msg1);

String **msg2** = messageFormat.format(params);

System.***out***.println(msg2);

Java本地化资源文件：

命名格式：<资源名>\_<语言代码>\_<国家/地区代码>.properties

ResourceBundle **resourceBundle** = ResourceBundle.*getBundle*("locale/resource", en\_US);

System.***out***.println(MessageFormat.*format*(resourceBundle.getString("greeting.common"), params));

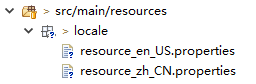
resourceBundle = ResourceBundle.*getBundle*("locale/resource");

System.***out***.println(MessageFormat.*format*(resourceBundle.getString("greeting.morning"), params));

// 没找到指定语言的资源文件，会找系统默认的本地资源文件

resourceBundle = ResourceBundle.*getBundle*("locale/resource", Locale.***CANADA***);

System.***out***.println(MessageFormat.*format*(resourceBundle.getString("greeting.afternoon"), params));





Spring封装的国际化信息：MessageSource接口

ResourceBundleMessageSource

ReloadableResourceBundleMessageSource

ApplicationContext **context** = **new** ClassPathXmlApplicationContext("i18n.xml");

MessageSource **messageSource** = (MessageSource)context.getBean("messageSource");

Locale **en\_US** = **new** Locale("en", "US");

// Locale zh\_CN = new Locale("zh", "CN");

Object[] **params** = {"John",**new** GregorianCalendar().getTime(),1.0E3};

System.***out***.println(messageSource.getMessage("greeting.common", params, en\_US));

System.***out***.println(messageSource.getMessage("greeting.morning", params, en\_US));

System.***out***.println(messageSource.getMessage("greeting.afternoon", params, en\_US));

**try** {

Thread.*sleep*(10000);

} **catch** (InterruptedException **e**) {

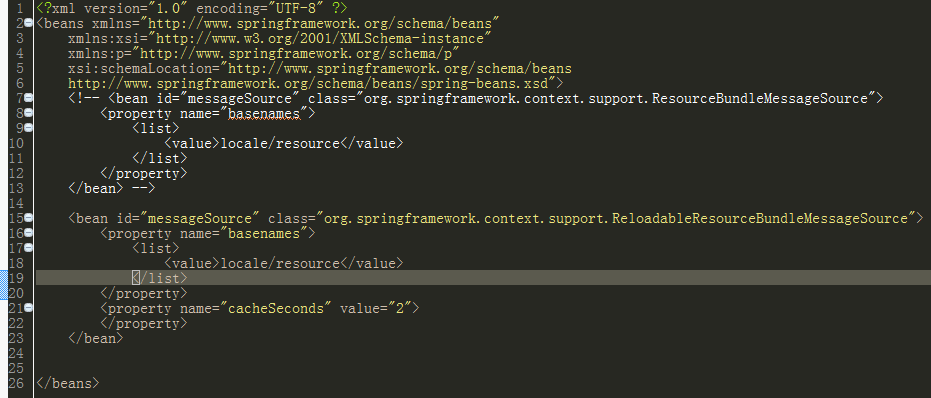
e.printStackTrace();

}

System.***out***.println(context.getMessage("greeting.common", params, en\_US));

System.***out***.println(context.getMessage("greeting.morning", params, en\_US));

System.***out***.println(context.getMessage("greeting.afternoon", params, en\_US));



Spring的容器级国际化信息资源

ApplicationContext继承MessageSource接口，因此ApplicationContext本身也是一个MessageSource对象，初始化是会查找一个叫messageSource的bean，将这个bean定义的信息资源加载为容器级的国际化信息资源。

如果没有名称为messageSource的bean，使用了容器级国际化信息资源时会报错。

### 容器事件

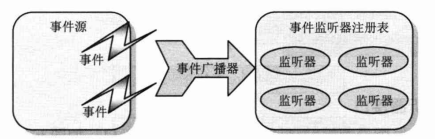
事件：java.util.EventObject

监听器：java.util.EventListener

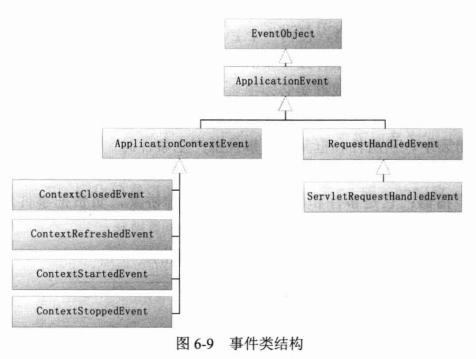
事件源：事件的产生者

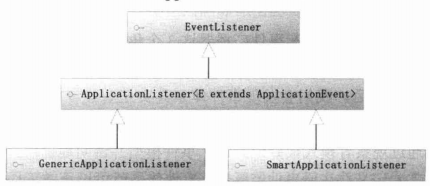
事件监听器注册表：保存事件监听器

事件广播器：负责将事件通知事件监听器



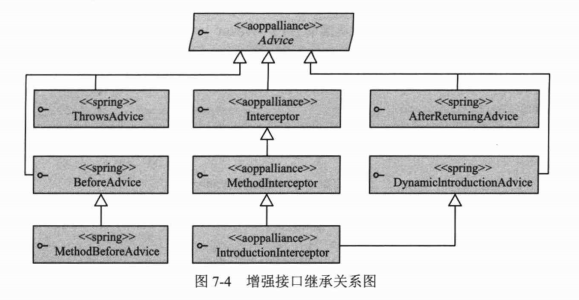
**Spring事件体系**

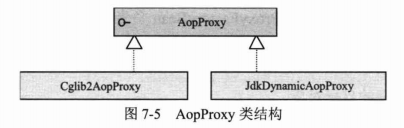






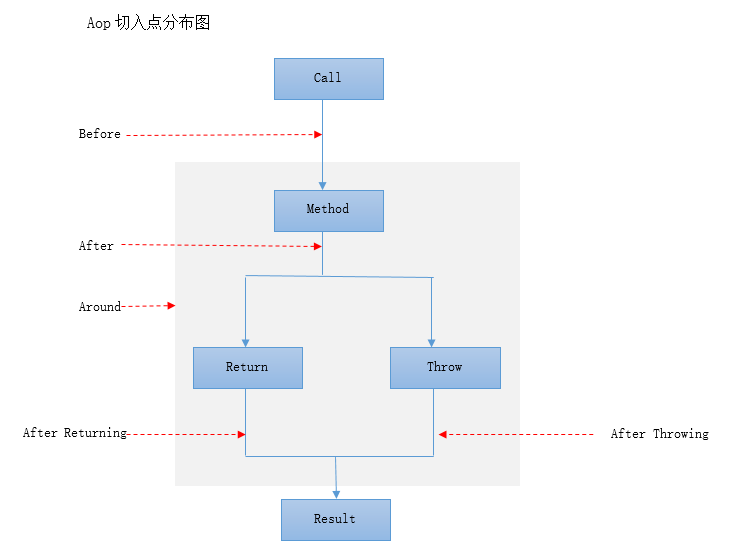
## Spring Aop









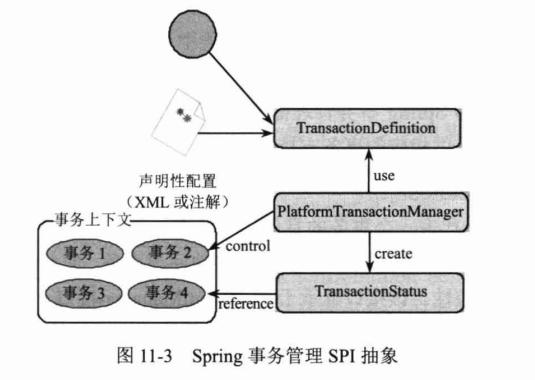


Aop切入点分布图



<https://www.cnblogs.com/zhangxufeng/p/9160869.html>

## Spring事务



Spring 事务传播行为：





## Spring 单元测试

## 7.12.5 编写基于JAVA的配置

<https://docs.spring.io/spring/docs/4.3.19.RELEASE/spring-framework-reference/htmlsingle/#beans-java-using-import>

### 使用@Import注解

就像在Spring XML文件中使用<import />元素来帮助模块化配置一样，@ Immort注释允许从另一个配置类加载@Bean定义：

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

现在，在实例化上下文时，不需要同时指定ConfigA.class和ConfigB.class，只需要显式提供ConfigB：

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

这种方法简化了容器实例化，因为只需要处理一个类，而不是要求开发人员在构造期间记住可能大量的@Configuration类。

从Spring Framework 4.2开始，@ Immort还支持对常规组件类的引用，类似于AnnotationConfigApplicationContext.register方法。 如果您想避免组件扫描，使用一些配置类作为显式定义所有组件的入口点，这将特别有用。