JAVA 达摩班

Spring Core

Spring 核心概念 1

Spring发展史

Spring框架由Rod Johnson设计，是目前最流行的Java EE开发框架之一，基本一统江湖。

- **2002**年10月 - Spring最早出现在Rod Johson的书《Expert One-on-One J2EE Design and Development》，还不是框架
- **2003**年6月 – 第一次作为框架发布，基于Apache许可
- 2006年1月 - 同时获得两个奖项JoIt productivity award 和 JAX Innovation Award
- 2006年10月 – Spring 2.0发布
- 2007年11月 – Spring 2.5发布
- **2009**年12月 – Spring 3.0发布
- 2011年12月 – Spring 3.1发布
- 2012年6月 – Spring 3.2发布
- **2013**年12月 – Spring 4.0发布
- 2015年7月 – Spring 4.2发布
- 2016年6月10日 – Spring 4.3发布
- **2017**年12月 – Spring 5发布 基于Reactive Streams，最低需要Java EE 7，兼容Tomcat 8/9, JBOss EAP 7和WebSphere 9

Spring核心

控制反转（Inversion of Control，IOC）- 面向对象一个原则，对象的控制（创建和管理）由容器或者框架实现，而不是开发者直接调用。开发者只需要实现接口或者放入自己的类。

依赖注入（Dependency Injection，DI）- 是设计模式中结构型模式的组合形式。每个方法都有一个独立对象（服务），通过接口去调用其他对象（依赖）。这些依赖对象在服务对象创建时候才实现，这是一种逆向的对象创建方式。Spring中DI通过constructor或setter实现。实现了DI的库叫做IoC容器。

面向切面编程（Aspect oriented programming，AOP）- 一种编程范式，允许定义横跨多个应用功能的切面（cross-cutting concern，横切关注点），用于表示同时用在多个函数上的功能，例如log。相比OOP中的class，AOP的关键元素是aspect。DI分离应用类成多个独立模块，而AOP用于从对象中分离切面。

Spring架构

Core container

Bean：管理bean生命周期

Core：核心实现，如DI和IoC

Context：访问设置对象，如ApplicationContext

SpEL: 执行时操作对象的表达式语言

Web

Web：创建rest服务，web应用或下载文件

Servlet：Spring-MVC的实现

WebSocket：实现CS之间基于socket的web通信

Portlet：门户环境下的MVC实现

Data access

JDBC：数据库抽象层，用于实现数据库连接

ORM：集成Hiberntate，JDO，JPA的ORM

OXM：连接Object / XML – XMLBeans, JAXB

Transaction：类事务管理

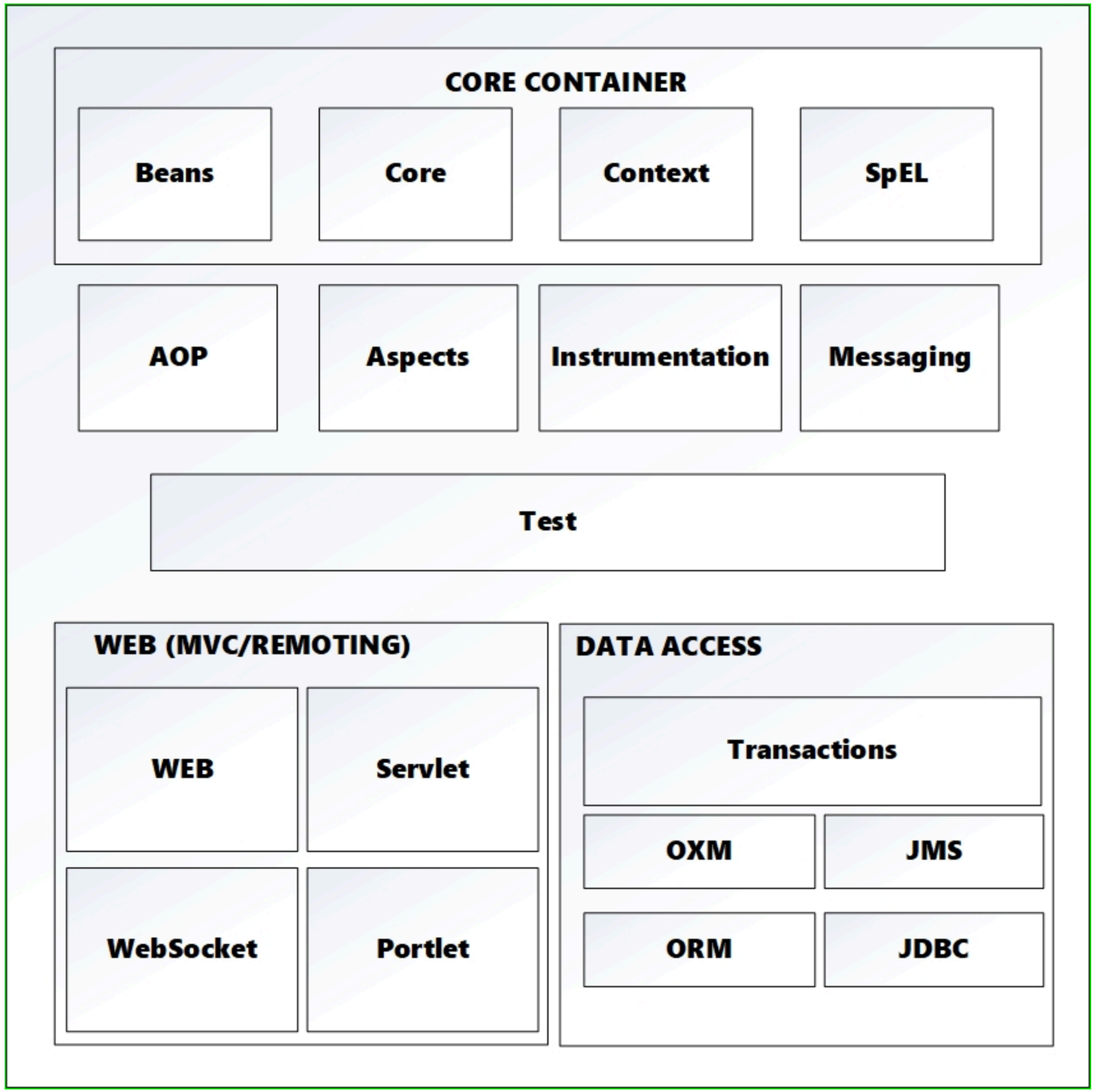
Miscellaneous

AOP：面向切面的实现

Aspects：集成AspectJ

Messaging：STOMP支持

Test：集成JUnit等框架实现spring测试



扩展阅读

深入理解IoC和DI

控制反转是OOP设计原则，它不是直接“new object”，而是由其他人（IoC 容器）来创建对象，管理他们生命周期。
控制反转有多种实现方式：策略模式，服务定位模式，工厂模式和依赖注入。

相对而言，DI属于实现IoC的设计模式，用于解决“硬编码”依赖问题。
三种方式实现动态插入依赖：constructor，setter和interface。

应用：使用*@Component*标示属于*Spring*容器管理的实例，而是使用*@Autowired*标示由*Spring*容器在需要时插入依赖，而不用开发者手动创建和管理。

```
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}
```

DI的朴素实现

```
public class Store {
    private Item item;

    public Store(Item item) {
        this.item = item;
    }
}
```

Spring IoC和DI

Spring IoC容器

Spring IoC容器在org.springframework.beans和org.springframework.context包中。
BeanFactory是Spring IoC容器的根接口，ApplicationContext是BeanFactory的子接口。
在Spring框架，**ApplicationContext**接口代表IoC容器，它负责实例化，配置，组装对象（bean）以及管理他们的生命周期。
ApplicationContext有两个子接口ConfigurableApplicationContext和WebApplicationContext。
提供多个实现：AnnotationConfigApplicationContext，ClassPathXmlApplicationContext和FileSystemXmlApplicationContext用于Java应用，
AnnotationConfigWebApplicationContext and XmlWebApplicationContext用于Web应用。IoC容器使用配置元数据（metadata）组装bean，可以通过XML配置文件和注释两种方式实现。

Spring DI

Spring提供构造函数（constructor），设值函数（setter）和字段（field）三种方式实现DI，官方推荐为强制依赖使用构造函数注入，可选注入使用设值函数。前两种属于手动定义依赖，第三种方式属于自动装配依赖。

Spring IoC和DI

1. 构造函数

```
@Configuration
public class AppConfig {
    @Bean
    public Item item1() {
        return new ItemImpl1();
    }
    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

或

```
<bean id="item1" class="org.dharma.spring.ItemImpl1" />
<bean id="store" class="org.dharma.spring.Store">
    <constructor-arg type="ItemImpl1" index="0" name="item" ref="item1" />
</bean>
```

Spring IoC和DI

2. 设值函数

```
@Bean
public Store store() {
    Store store = new Store();
    store.setItem(item1());
    return store;
}
```

或

```
<bean id="store" class="org.dharma.spring.Store">
    <property name="item" ref="item1" />
</bean>
```

3. 字段

当没有构造函数和设值函数，容器使用反射动态注入Item对象

简单，但官方不推荐，因为反射实现代价大，XML实现复杂，破坏SRP

```
public class Store {
    @Autowired
    private Item item;
}
```

Spring 自动装配

自动装配（Autowring）

通过识别已定义的bean，Spring的装配机制允许自动解析bean之间的依赖。它提供四种方式：

1. **默认值**：不会自动装配，必须自定义依赖；
2. ***byName***：使用*property name*寻找*bean*；
3. ***byType***：使用*property type*寻找*bean*，多个同类型*bean*会抛出异常；
4. ***constructor***：使用构造函数参数类型寻找*bean*

以byType为例

```
@Bean(autowire = Autowire.BY_TYPE)
public class Store {

    private Item item;

    public setItem(Item item){
        this.item = item;
    }
}
```

```
public class Store {

    @Autowired
    @Qualifier("item1")
    private Item item;
}
```

或

```
<bean id="store" class="org.dharma.spring.Store" autowire="byType"> </bean>
```

Spring Bean

由Spring容器创建的对象叫Spring Bean。如果POJO（Plain Old Java Object）经过配置后，并且由容器实例化的对象就是Bean。除了以下作用域，Bean支持自定义作用域。

Spring Bean = 标准Java类 + 生命周期 + IoC容器管理

Bean提供五种类型的作用域：

1. **singleton**：每个容器只创建该bean的一个实例，防止数据不一致性
2. **prototype**：每次请求都会创建新的实例
3. **request**：和prototype一样，用于web应用。每个HTTP请求都会创建一个新实例
4. **session**：每个HTTP会话会创建一个新实例
5. **global-session**：为Portlet应用创建全局会话bean

Spring Bean配置

1. 基于**注释**配置：Scope可以通过@Scope结合@Service和@Component实现
2. 基于**XML**配置：使用xml配置，且Spring MVC下的web.xml配置可以自动加载
3. 基于**Java**配置：从3.0开始，使用@Configuration, @ComponentScan和@Bean注释实现

Spring Bean作用域

1. Singleton

```
@Bean
//@Scope(“singleton")
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public Person personSingleton() {
    return new Person();
}

<bean id="personSingleton" class="org.dharma.scopes.Person" scope="singleton"/>
```

2. Prototype

```
@Bean
//@Scope("prototype")
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public Person personPrototype() {
    return new Person();
}

<bean id="personPrototype" class="org.dharma.scopes.Person" scope="prototype"/>
```

Spring Bean作用域

3. Request

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator requestMessage() {
    return new HelloMessageGenerator();
}

@Controller
public class ScopesController {
    @Resource(name = "requestMessage")
    HelloMessageGenerator requestMessage;

    @RequestMapping("/scopes")
    public String getScopes(Model model) {
        requestMessage.setMessage("Good morning!");
        model.addAttribute("requestMessage", requestMessage.getMessage());
        return "scopesExample";
    }
}
```

Spring Bean作用域

4. Session

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator sessionMessage() {
    return new HelloMessageGenerator();
}

@Controller
public class ScopesController {
    @Resource(name = "sessionMessage")
    HelloMessageGenerator sessionMessage;

    @RequestMapping("/scopes")
    public String getScopes(Model model) {
        sessionMessage.setMessage("Good afternoon!");
        model.addAttribute("sessionMessage", sessionMessage.getMessage());
        return "scopesExample";
    }
}
```

Spring Bean作用域

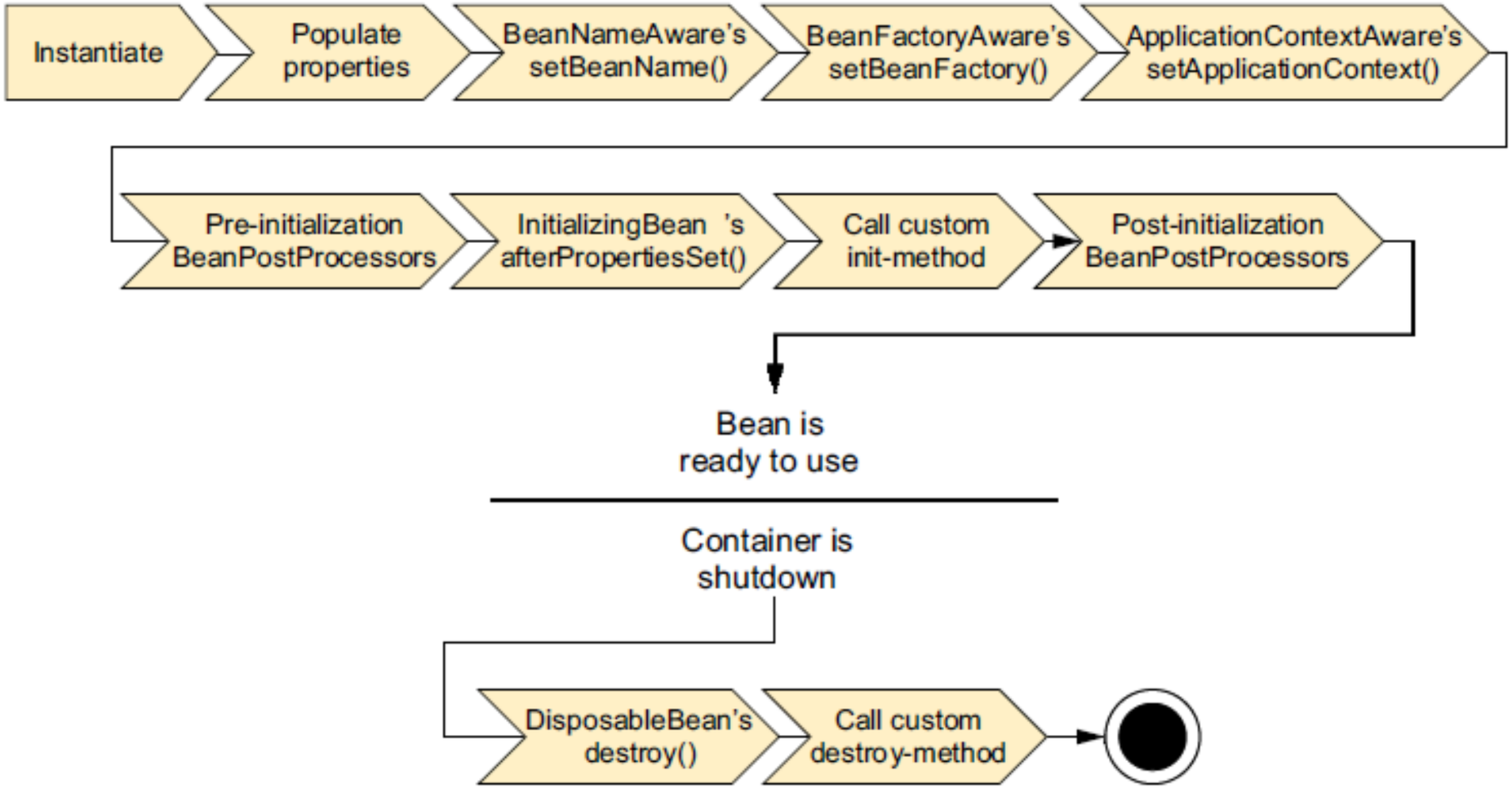
5. GlobalSession

用于Portlet容器的应用，每个portlet拥有自己的session，这种类型bean应用于所有session

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator globalSessionMessage() {
    return new HelloMessageGenerator();
}
```

Spring Bean生命周期

BeanFactory用于管理bean的生命周期，它包括两类callback函数：初始化后和销毁前。



Spring Bean生命周期

1. Spring实例化bean
2. Spring将值和bean引用注入到properties
3. 如果实现BeanNameAware接口，Spring调用setBeanName方法，参数为bean id
4. 如果实现BeanFactoryAware接口，Spring调用setBeanFactory方法，参数为bean factory
5. 如果实现ApplicationContextAware接口，Spring调用postProcessBeforeInitialization方法
6. 如果实现BeanPostProcessor接口，Spring调用postProcessBeforeInitialization方法
7. **如果实现InitializingBean接口，Spring调用afterPropertiesSet方法。如果bean有init方法声明，那么它会调用特定的初始化方法**
8. 如果实现BeanPostProcessor接口，Spring调用postProcessAfterInitialization方法
9. 如果实现BeanNameAware接口，Spring调用setBeanName方法，参数为bean id

10. Bean完成初始化，知道application context被销毁

11. **如果实现DisposableBean接口，Spring调用destroy方法。同样如果bean有destroy方法声明，那么特定方法会被调用**

Spring Bean生命周期

Spring提供四种方法控制生命周期事件：

1. InitializingBean and DisposableBean 回调接口

2. 用于特殊行为的Aware接口

3. 配置文件定制init和destroy方法

4. @PostConstruct和@PreDestroy注释

初始化执行顺序： 4 -> 1 -> 3

Spring Bean生命周期

1. InitializingBean and DisposableBean 回调接口

```
public class DemoBeanTypeOne implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        //Bean initialization code
    }

    @Override
    public void destroy() throws Exception {
        //Bean destruction code
    }
}
```

Spring Bean生命周期

2. 用于特殊行为的Aware接口

```
public class BemoBeanTypeTwo implements ApplicationContextAware,
    ApplicationEventPublisherAware, BeanClassLoaderAware,
    BeanFactoryAware,BeanNameAware, LoadTimeWeaverAware,
    MessageSourceAware, NotificationPublisherAware, ResourceLoaderAware {

   @Override
    public void setResourceLoader(ResourceLoader arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setNotificationPublisher(NotificationPublisher arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void setMessageSource(MessageSource arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setLoadTimeWeaver(LoadTimeWeaver arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setBeanName(String arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setBeanFactory(BeanFactory arg0) throws BeansException {
        // TODO Auto-generated method stub
    }

    @Override
    public void setBeanClassLoader(ClassLoader arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void setApplicationContext(ApplicationContext arg0)
            throws BeansException {
        // TODO Auto-generated method stub
    }
}
```

Spring Bean生命周期

3. 配置文件定制init和destroy方法

发生在afterPropertiesSet和destroy重载方法之后

局部定义 - 单个bean

```
<beans>
    <bean id="demoBean" class="com.dharma.task.DemoBean"
          init-method="customInit" destroy-method="customDestroy"></bean>
</beans>
```

全局定义 - 当前上下文所有beans

```
<beans default-init-method="customInit" default-destroy-method="customDestroy">
    <bean id="demoBean" class="com.dharma.task.DemoBean"></bean>
</beans>
```

Spring Bean生命周期

4. @PostConstruct和@PreDestroy注释

@PostConstruct注释方法在被构造出来后，实例返回请求对象之前执行
@PreDestroy注释方法在被容器销毁之前执行
发生在afterPropertiesSet和destroy重载方法之前

```
public class BemoBean {

    @PostConstruct
    public void customInit() {
        System.out.println("Method customInit() invoked...");
    }

    @PreDestroy
    public void customDestroy() {
        System.out.println("Method customDestroy() invoked...");
    }
}
```

Spring注释

Spring注释辅助进行依赖配置，实现依赖注入（DI）
基于Java的依赖注入，**可编程**的方式实现bean组件管理

Spring注释：

1. **@Configuration**：表示该类将声明一个或多个@Bean方法。这些类由Spring容器处理，在运行时生成bean定义和bean服务请求
2. **@Bean**：注释方法表示将生成一个bean，由容器管理。最重要和常用的注释。接受参数：name，initMethod和destroyMethod

```
@Configuration
public class AppConfig {
    @Bean(name = "comp", initMethod = "turnOn", destroyMethod = "turnOff")
    Computer computer(){
        return new Computer();
    }
}

public class Computer {
    public void turnOn(){ System.out.println("Load operating system”);  }
    public void turnOff(){  System.out.println("Close all programs”);  }
}
```

Spring注释

3. **@PreDestroy和@PostConstruct**是bean的initMethod和destroyMethod的替代方法。

```
public class Computer {
  @PostConstruct
   public void turnOn(){
       System.out.println("Load operating system");
   }
   @PreDestroy
   public void turnOff(){
       System.out.println("Close all programs");
   }
}
```

4. **@ComponentScan**: 和@Configuration一起使用，指定扫描Spring组件的目录
5. **@Component**: 表示一个类是组件，它可以被配置了@configuration@ComponentScan的类监测到
6. **@PropertySource**: 提供一个声明式的机制为spring环境添加property source，和@value一起使用
7. **@Service**: 表示一个类是服务，是一种特殊的组件，允许被classpath扫描检测
8. **@Repository**: 表示一个类是仓库，是一种特殊的组件，适合和DAO类一起使用
9. **@Autowired**: 用于实现bean自动注入。经常和Spring @Qualifier配合使用，解决同类型注入导致的冲突（相当于起名字用于区分）

Spring注释

@Component就是用于定义通用的组件，刷存在感，告诉Spring：”我就是想告诉你，你可能需要一个类实例。可能因为我要请求它，或者我请求的对象需要它“
另外三个属于特殊的@Component，被赋予特定的用途：**@Repository用于持久层，@Service用于业务层，@Controller用于表示层（请求）**
共性是他们都支持自动扫描检测和bean依赖注入，理论上相互之间可以互换

Spring 2.0引入@Repository注释作为一个DAO（Data Access Object）标示（marker），包括自动异常翻译
Spring 2.5引入更多的注释，包括@Component，@Service和@Controller。 @Component被作为通用的，模版的，由Spring管理的组件，而@Repository，@Service和@Controller继承自@Component，有各自特定上下文

<context:component-scan> 只扫描@Component，但实际上他们都会被扫描，因为@Controller，@Service和@Repository的实现有@Component注释，例如：

```
@Component
public @interface Service {
    ….
}
```

@Repository：定义数据仓库和捕获特定平台异常，重新抛出统一的非检测异常（unchecked exception）
@Controller：和@RequestMapping一起使用，用于根据URI处理用于请求
@Service：业务逻辑，调用Respository方法，用在Spring MVC

Spring注释

Spring MVC注释

1. @Controller
2. @RequestMapping
3. @PathVariable
4. @RequestParam
5. @ModelAttribute
6. @RequestBody and @ResponseBody
7. @RequestHeader and @ResponseHeader

Spring事务管理注释

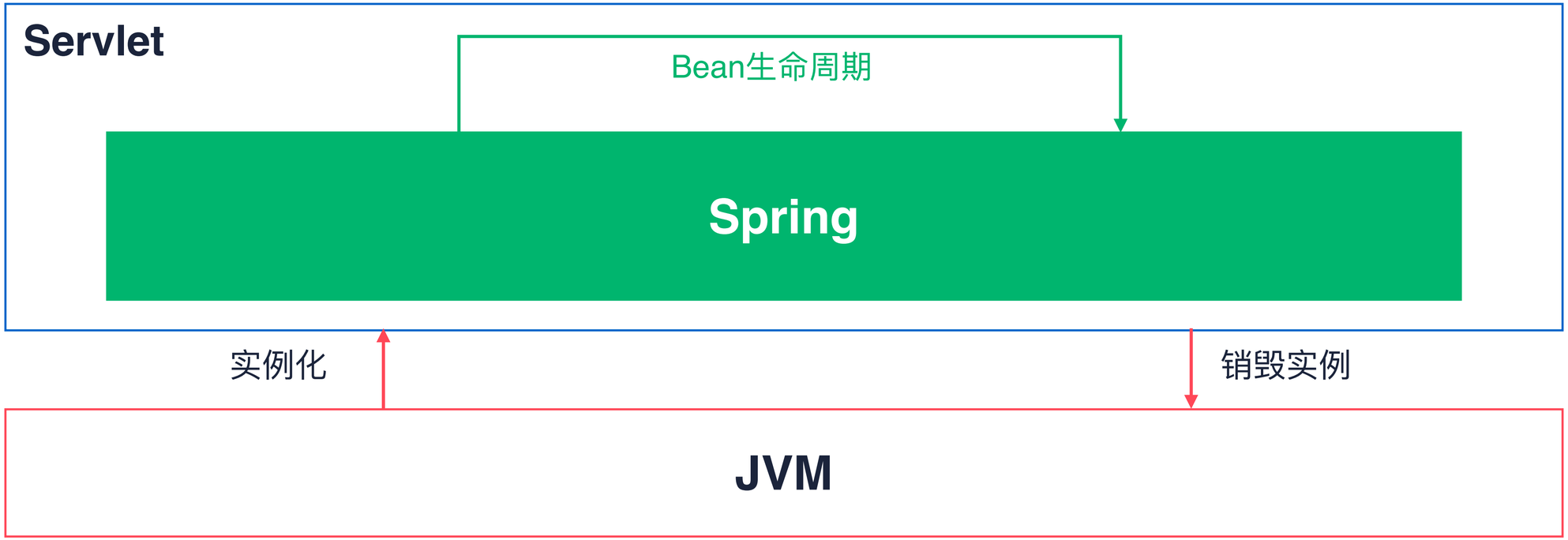
@Transactional：注释用于声明事务管理，常用在Spring MVC Hibernate

Spring Security注释

@EnableWebSecurity：和@Configuration配合使用定义安全性配置，用在Spring Security模块

Spring Boot注释

@SpringBootApplication
@EnableAutoConfiguration



Spring运行在JVM之上

本质就是IoC容器，实现依赖注入

既可以开发Java应用，也可以开发Web应用

Thanks!

Any questions?