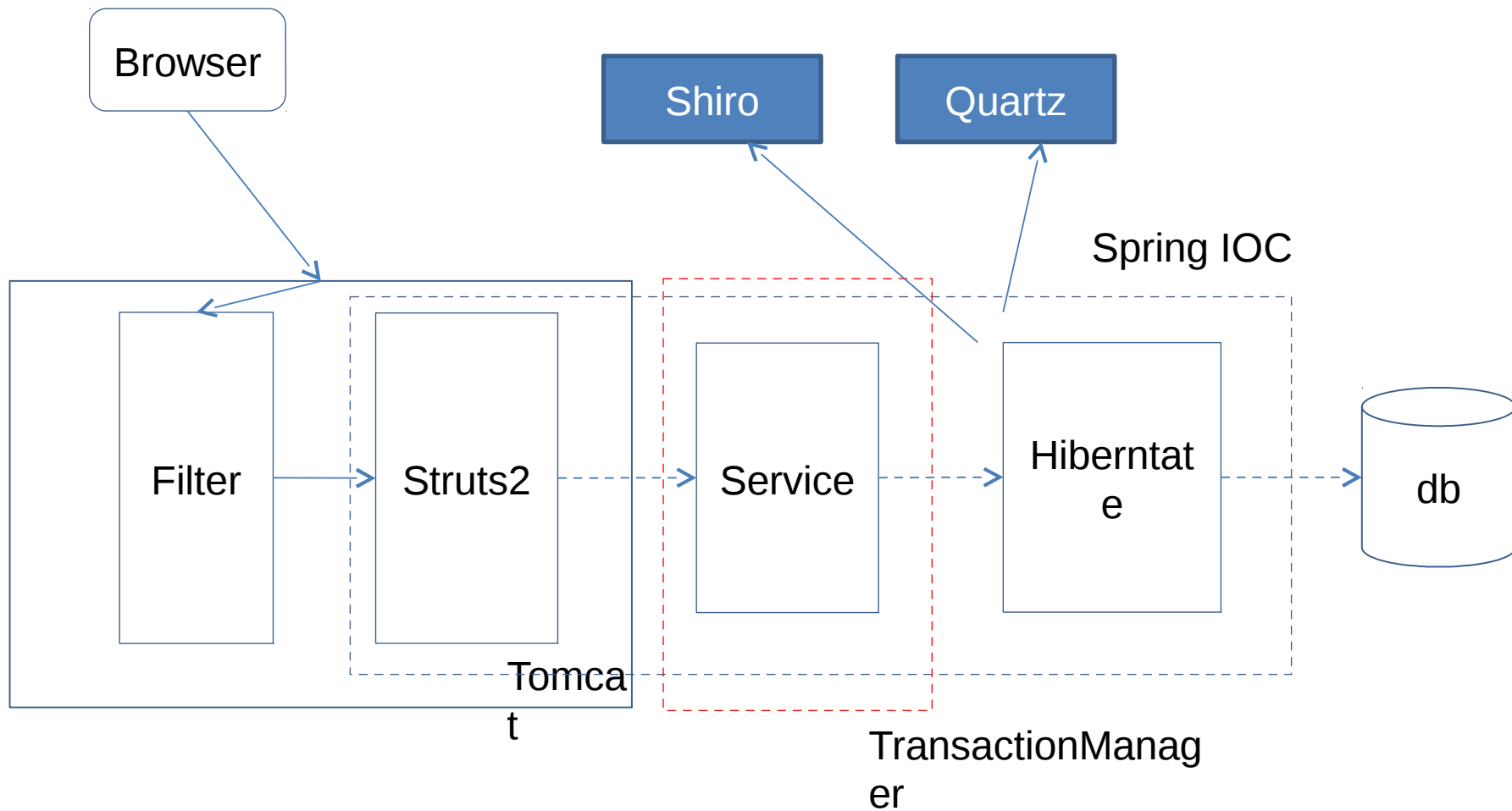




讲师：佟刚
新浪微博：尚硅谷 - 佟刚

B/S



Hello World

□□ : □□

□□□□ : □□□ - □□

Spring 入门 (1)

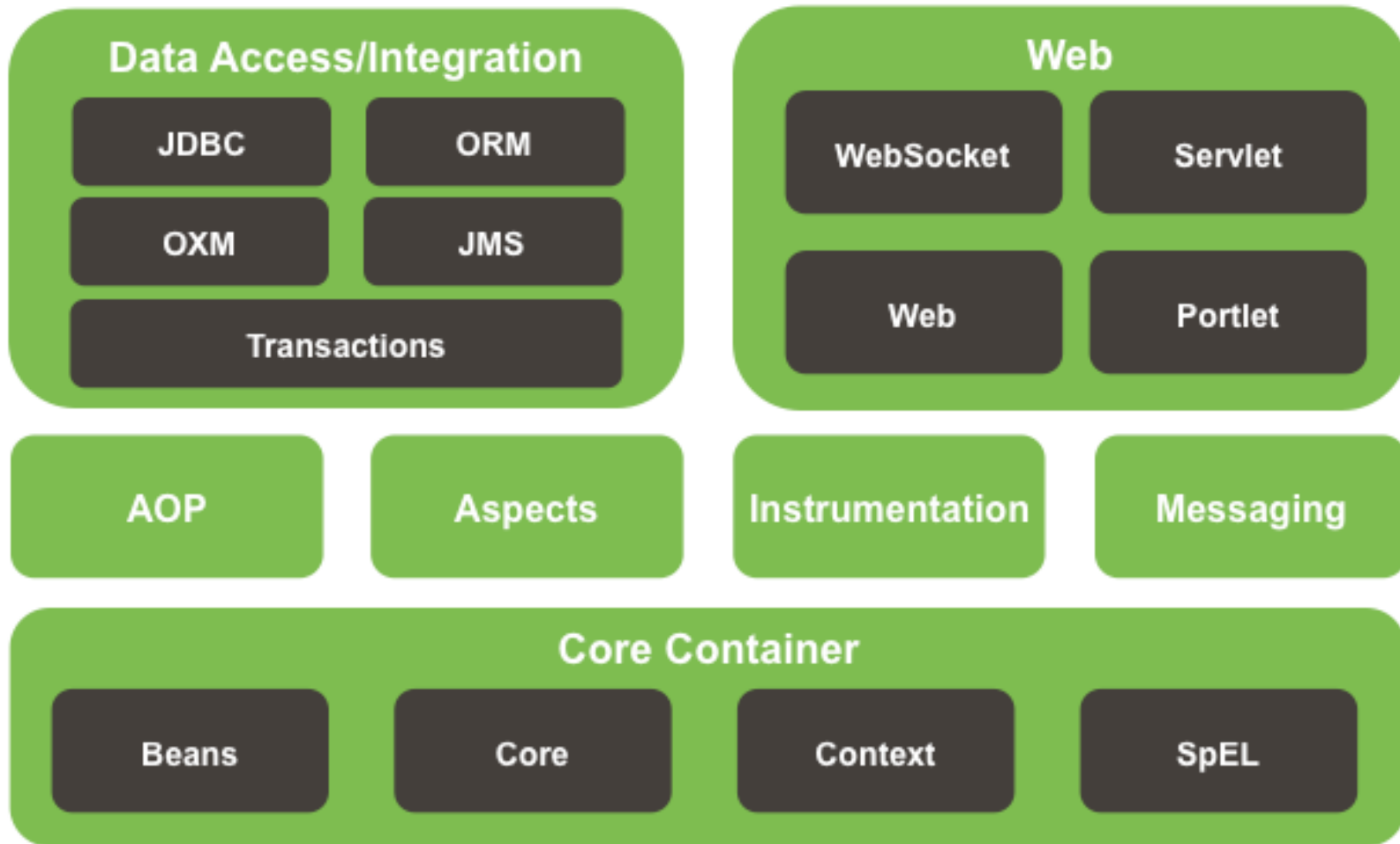
- Spring 入门 .
- Spring 入门 . 是 Spring 入门 JavaBean 入门 EJB 入门 .
- Spring 入门 IOC(DI) 与 AOP 入门 .

J2ee without ejb

Spring 入门 (2)

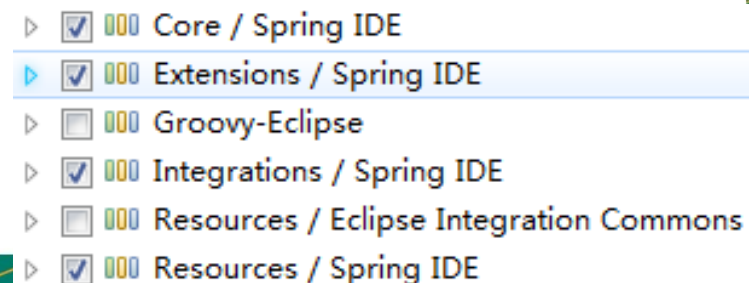
- 了解 Spring:
 - 了解 **Spring** 框架 - 是 Spring 框架提供 Spring 的 API
 - 了解 (DI --- dependency injection 即 IOC)
 - 了解 (AOP --- aspect oriented programming)
 - 了解 : Spring 框架 , 框架提供框架
 - 了解 : Spring 框架提供框架 . 是 Spring 框架 XML 与 Java 框架
 - 了解 IOC 与 AOP 框架提供框架 框架 Spring 框架 SpringMVC 与 框架 Spring JDBC 与

Spring



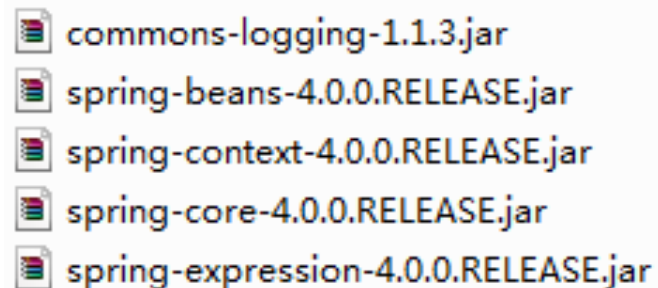
☐☐ SPRING TOOL SUITE

- SPRING TOOL SUITE ☐☐ Eclipse ☐☐☐☐☐☐☐☐☐☐ Eclipse ☐☐☐☐☐ Spring ☐☐☐
- ☐☐☐☐☐☐ springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip ☐☐
 - **Help** --> **Install New Software...**
 - Click Add...
 - In dialog Add Site dialog, click **Archive...**
 - Navigate to **springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip** and click **Open**
 - Clicking **OK** in the Add Site dialog will bring you back to the dialog 'Install'
 - Select the **xxx/Spring IDE** that has appeared
 - Click **Next** and then **Finish**
 - **Approve the license**
 - Restart eclipse when that is asked



Spring 环境搭建

- 需要 jar 包的路径 classpath 为：



```
commons-logging-1.1.3.jar  
spring-beans-4.0.0.RELEASE.jar  
spring-context-4.0.0.RELEASE.jar  
spring-core-4.0.0.RELEASE.jar  
spring-expression-4.0.0.RELEASE.jar
```

- Spring 环境搭建：需要 Spring 环境搭建的 Bean 环境，需要搭建 Spring IOC 环境 Bean. Bean 环境搭建 **classpath** 为，需要搭建环境。

Spring

```
package com.atguigu.spring.helloworld;

public class HelloWorld {

    private String userName;

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public void hello(){
        System.out.println("Hello: " + userName);
    }

}
```

HelloWorld.java

```
<bean id="helloWorld"
      class="com.atguigu.spring.helloworld.HelloWorld">
    <property name="userName" value="Spring"></property>
</bean>
```

applicationContext.xml

Spring

```
public static void main(String[] args) {  
  
    //1. 创建 Spring 的 IOC 容器  
    ApplicationContext ctx =  
        new ClassPathXmlApplicationContext("applicationContext.xml");  
  
    //2. 从容器中获取 Bean  
    HelloWorld helloWorld = (HelloWorld) ctx.getBean("helloWorld");  
    System.out.println(helloWorld);  
  
    //3. 调用方法  
    helloWorld.hello();  
}
```

Spring 框架 Bean 管理

□□ : □□

□□□□ : □□□ - □□

目录

- **IOC & DI** 简介
- 什么是 bean
 - 通过 XML 配置文件来配置
 - Bean 的创建由 BeanFactory & ApplicationContext 负责
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的 XML 文件
 - 配置 Bean 的注解
 - 配置 Bean 的 YAML 文件
 - bean 的 scope 属性
 - bean 的 singleton 与 prototype 与 WEB 容器
 - 配置 Bean 的别名
 - spEL
 - IOC 容器 Bean 的初始化
 - Spring 4.x 的 Bean 生命周期

IOC 与 DI

- IOC(Inversion of Control) 反转控制 . 将原本由程序自己创建的对象 , 交给第三方容器来创建 . 容器 , 管理对象 . 容器 IOC 容器 , 管理对象 , 管理对象 , 管理对象 . 容器 IOC 容器 . 容器 IOC 容器 .
- DI(Dependency Injection) — IOC 容器管理对象 (容器 : setter 方法) 管理对象 . 容器 IOC 容器


```
class A{
```

```
class B{
```

```
    private A a;
```

```
    public void setA(A a){
```

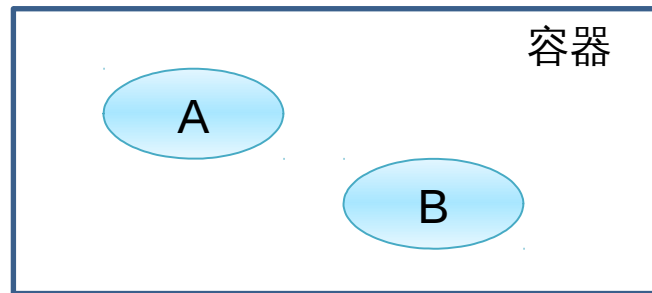
```
        this.a = a;
```

```
    }
```

```
}
```



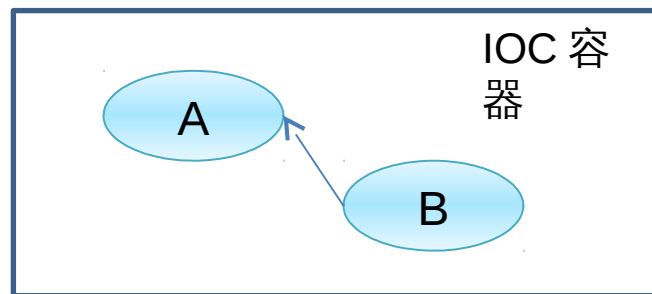
需求：从容器中获取 B 对象，并使 B 对象的 a 属性被赋值为容器中 A 对象的引用



```
A a = getA();
```

```
B b = getB();
```

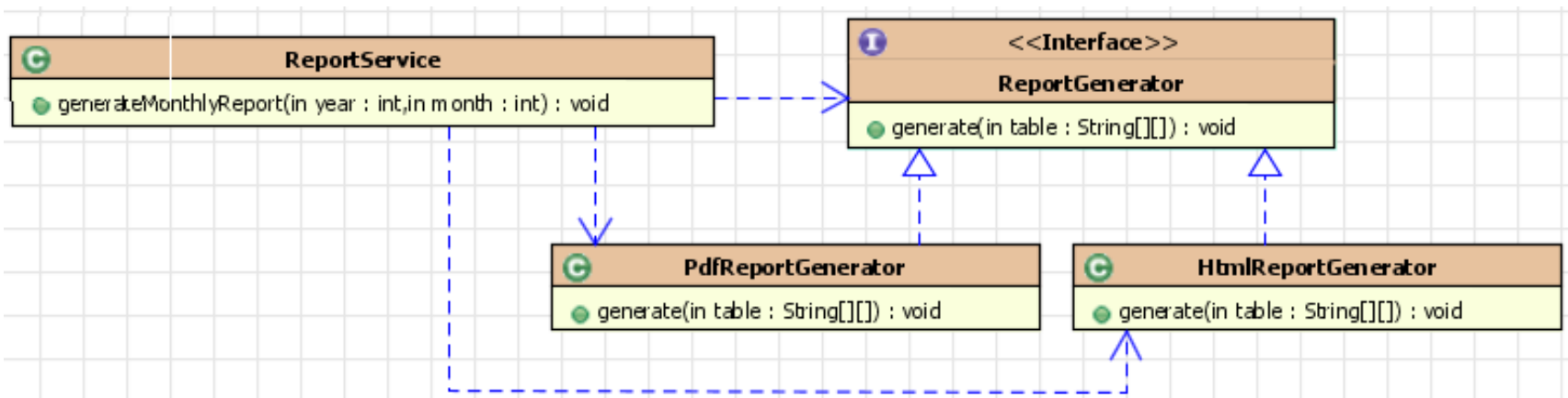
```
b.setA(a);
```



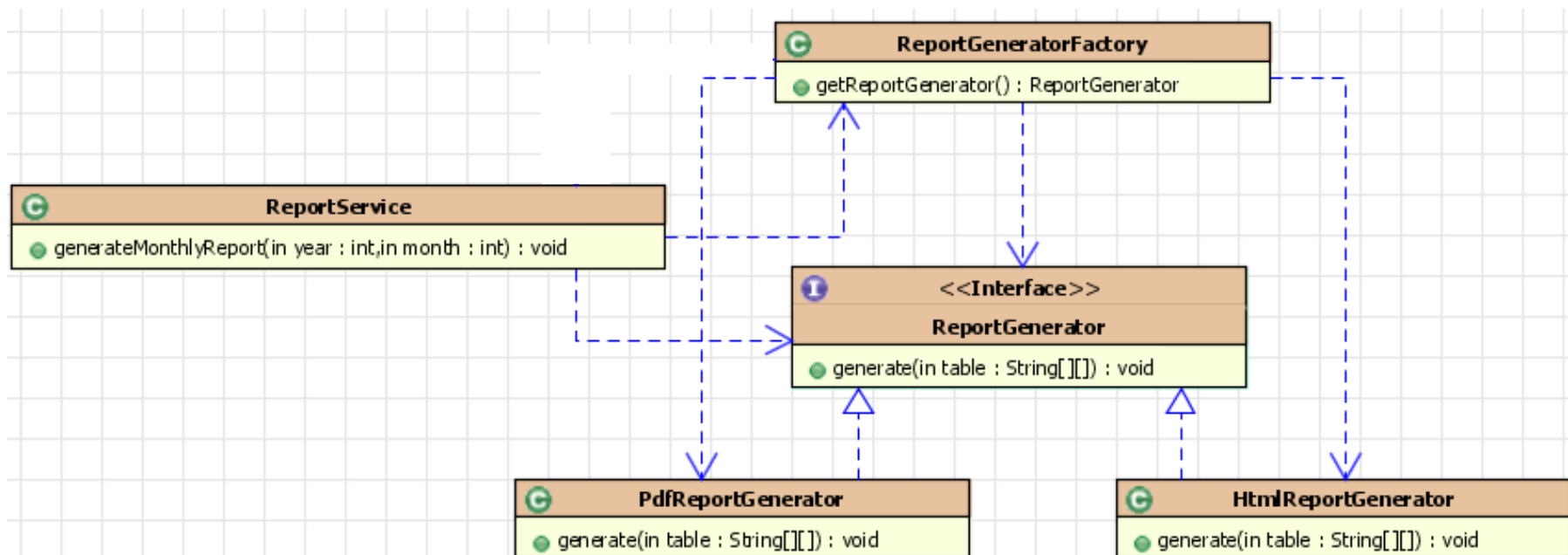
```
B b = getB();
```

IOC 的 好处 有哪些

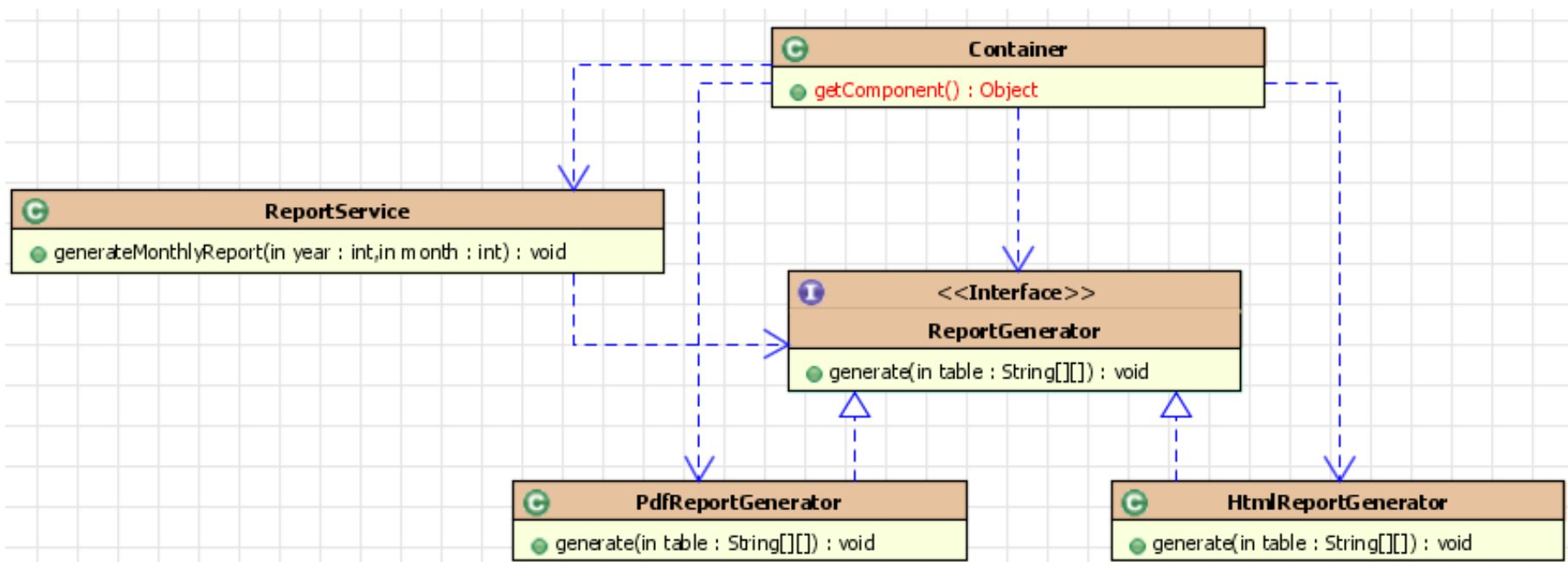
- 优点：使用 HTML 和 PDF 格式生成报告。



IOC 的 10 大好处



IOC --- 依赖注入



目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 的创建方式 & FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext
 - 配置 Bean 的 scope
 - 配置 Bean 的 lifecycle
 - bean 的命名
 - bean 的 singleton & prototype & WEB 容器
 - 配置 Bean 的 init & destroy
 - spEL
 - IOC 容器 Bean 的查找
 - Spring 4.x 的改进

Spring IOC 配置 Bean

- xml 配置 bean 配置 bean

```
<!-- 通过全类名的方式来配置 bean -->
<bean id="helloWorld"
      class="com.atguigu.spring.helloworld.HelloWorld">
</bean>
```

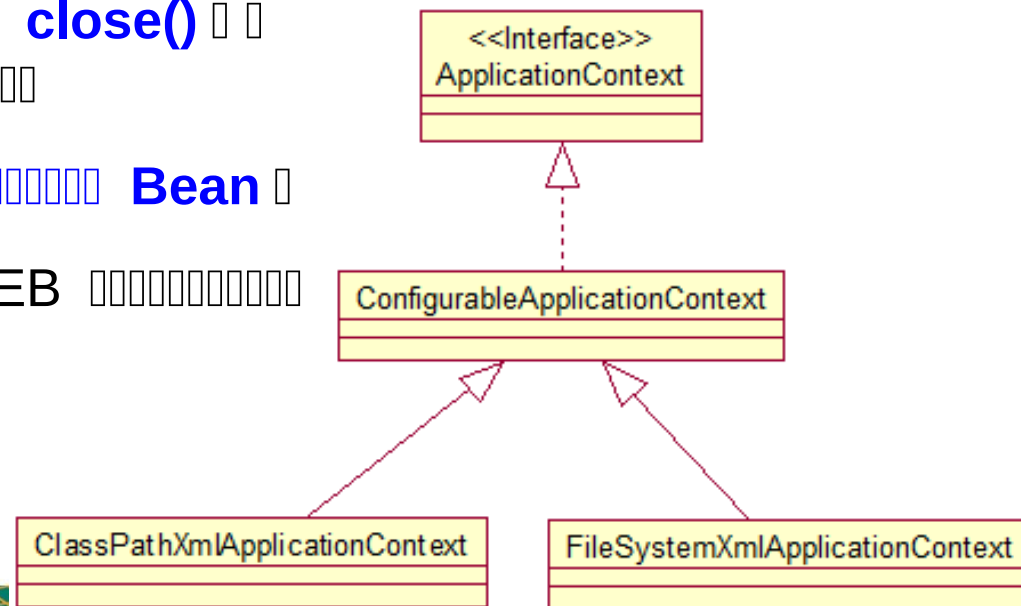
- id Bean 配置
 - IOC 配置
 - id Spring 配置 Bean
 - id 配置

Spring 简介

- Spring IOC 容器 Bean 容器 Bean 容器 , 容器容器容器 . 容器容器容器 , 容器 IOC 容器 Bean 容器 .
- Spring 容器容器 IOC 容器 .
 - **BeanFactory**: IOC 容器容器 .
 - **ApplicationContext**: 容器容器容器 . 容器 BeanFactory 容器 .
 - BeanFactory 容器 Spring 容器容器容器 Spring 容器 ApplicationContext 容器 容器 Spring 容器容器容器容器容器容器 **ApplicationContext** 容器 **BeanFactory**
 - 容器容器容器 , 容器容器容器 .

ApplicationContext

- ApplicationContext 接口
 - **ClassPathXmlApplicationContext** 通过类路径加载配置文件
 - **FileSystemXmlApplicationContext**: 通过文件系统加载配置文件
- ConfigurableApplicationContext 接口 ApplicationContext 的子接口，增加了 refresh() 和 **close()** 方法
- **ApplicationContext** 接口用于管理 Bean
- **WebApplicationContext** 接口用于管理 WEB 环境下的 ApplicationContext



□ IOC □□□□□ Bean

- □□ ApplicationContext □ getBean() □□



BeanFactory

- ^{SF} FACTORY_BEAN_PREFIX : String
- getBean(String) : Object
- getBean(String, Class<T>) <T> : T
- getBean(Class<T>) <T> : T
- getBean(String, Object...) : Object
- containsBean(String) : boolean
- isSingleton(String) : boolean
- isPrototype(String) : boolean
- isTypeMatch(String, Class<?>) : boolean
- getType(String) : Class<?>
- getAliases(String) : String[]

□□□□□□□□

- Spring □□ 3 □□□□□□□□

- □□□□

- □□□□□

- □□□□□□□□□□□□□□□□

配置

- 通过 setter 方法 Bean 配置
- 通过 `<property>` 标签, 通过 `name` 属性 Bean 配置 `value` 属性
- 通过 `<value>` 属性

```
<!-- 通过全类名的方式来配置 bean -->
<bean id="helloWorld"
      class="com.atguigu.spring.helloworld.HelloWorld">
    <property name="userName" value="atguigu"></property>
</bean>
```

工厂模式

- 工厂模式 Bean 工厂模式 Bean 工厂模式
- 工厂模式 <constructor-arg> 工厂模式 , **<constructor-arg>** 工厂
name 工厂

□□□□□□

-

```
<bean id="car" class="com.atguigu.spring.helloworld.Car">
    <constructor-arg value="奥迪" index="0"></constructor-arg>
    <constructor-arg value="长春一汽" index="1"></constructor-arg>
    <constructor-arg value="500000" index="2"></constructor-arg>
</bean>
```

- <bean id="car" class="com.atguigu.spring.helloworld.Car">
 <constructor-arg value="奥迪" type="java.lang.String"/>
 <constructor-arg value="长春一汽" type="java.lang.String"/>
 <constructor-arg value="500000" type="double"/>
</bean>

目录

- IOC & DI 简介
- 什么是 bean
 - 通过 XML 配置文件来配置
 - Bean 的创建方式 & 工厂类 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的几种方式
 - 配置 Bean 的几种方式
 - 配置 Bean 的几种方式
 - bean 的几种作用域
 - bean 的几种作用域 singleton & prototype & WEB 的几种作用域
 - 配置 Bean 的几种方式
 - spEL
 - IOC 容器 Bean 的几种作用域
 - Spring 4.x 的 Bean 的几种作用域

CDATA

- 在XML中，CDATA 是 Character Data 的缩写，表示一段未解析的字符数据。在XML中，CDATA 的语法格式为：
`<value>` 表示 value 的值。
- 在XML中，String 类型的值必须用双引号包裹。
- 在XML中，CDATA 的语法格式为：
`<![CDATA[]]>` 表示未解析的字符数据。

Bean

- Bean 是 Spring 容器管理的一个对象。一个 Bean 实例，一个 Bean 实例 Bean 实例
- Bean 实例，通过 `<ref>` 或 `ref` 属性 Bean 实例 Bean 实例。
- Bean 实例，一个 Bean 实例 Bean

```
<bean id="service" class="com.atguigu.spring.ioc.ref.Service"></bean>

<bean id="action" class="com.atguigu.spring.ioc.ref.Action">
    <!--
        为 service 属性赋值
        因为 service 属性是一个 bean 类型，可以使用 ref 指向 ioc 容器中的其他的 bean
    -->
    <property name="service" ref="service"></property>
</bean>
```

Bean

- Bean 是 Spring 容器管理的一个对象，它是由 Spring 容器创建并管理的。Bean 的定义使用 `<property>` 和 `<constructor-arg>` 标签，其中 `id` 和 `name` 属性
- Bean 的创建和销毁是由 Spring 容器管理的

如何判断一个 null 值

- 如何判断一个 **<null/>** 值 Bean 如何判断一个 null 值
- 在 Struts 和 Hiberante 中如何判断 **Spring** 如何判断一个

集合

- Spring 集合配置 xml 中 (集合 : `<list>`, `<set>` 或 `<map>`) 配置集合。
- 集合 `java.util.List` 配置, 配置 `<list>` 集合, 配置集合配置。配置集合配置 `<value>` 配置集合, 配置 `<ref>` 配置 Bean 配置。配置 `<bean>` 配置 Bean 配置。配置 `<null/>` 配置集合。配置集合配置。
- 配置集合 `List` 配置, 配置 `<list>`
- 配置 `java.util.Set` 配置 `<set>` 配置, 配置集合配置 `List` 配置。

Map

- Java.util.Map 接口 **<map>** 接口，<map> 接口实现类 **<entry>** 接口。
Map 接口。 Map 接口实现类。
- 接口 **<key>** 接口。
- 接口实现类， 接口实现类 **<value>**, **<ref>**, **<bean>** 和 **<null>** 接口。
- 接口 Map 接口实现类 **<entry>** 接口实现类： 接口实现类 key 和 value 接口； Be an 接口实现类 key-ref 和 value-ref 接口实现类。
- 接口 **<props>** 接口实现类 java.util.Properties, 接口实现类 **<prop>** 接口实现类。
接口 **<prop>** 接口实现类 **key** 接口实现类。

utility scheme

- 在 util schema 中，**Bean** 是，**Bean** 是，**Bean** 是。
- 在 util schema 中，**Bean** 是。在 util schema 中，**Bean** 是。

XML p 配置

- XML 配置文件 XML 配置文件
- Spring 2.5 配置文件 p <bean> Bean 配置
- p XML 配置文件

目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 的创建方式 & 工厂方法 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的多种方式
 - 配置 Bean 的多种方式
 - 配置 Bean 的多种方式
 - bean 的 scope 属性
 - bean 的 singleton 与 prototype 与 WEB 应用
 - 配置 Bean 的多种方式
 - spEL
 - IOC 容器 Bean 的创建
 - Spring 4.x 的改进

XML 配置 Bean 配置

- Spring IOC 容器配置 Bean. 配置 Bean 的 XML 标签是 **<bean>** 属性 **autowire** 可以配置为 **autowire** 属性
- **byType**(配置 Bean 的类名): 在 IOC 容器中配置 Bean 的类名 Bean. 配置 Bean 的类名, Spring 容器会自动配置 Bean 的类名, 配置 Bean 的类名.
- **byName**(配置 Bean 的名称): 配置 Bean 的名称 Bean 的名称.
- **constructor**(配置 Bean 的构造方法): 配置 Bean 的构造方法, 配置 Bean 的构造方法. 配置 Bean 的构造方法.

- XML Bean

目录

- IOC & DI 简介
- 什么是 bean
 - 通过 XML 配置文件来配置
 - Bean 的创建方式 & 工厂类 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的 scope
 - 配置 Bean 的 lifecycle
 - 配置 Bean 的 init-method & destroy-method
 - bean 的 scope: singleton & prototype & WEB 容器
 - 配置 Bean 的 lazy-init
 - spEL
 - IOC 容器 Bean 的初始化
 - Spring 4.x 的 Bean 配置

Bean

- **Spring** 管理 **bean** , 管理 bean 的 bean. 管理 Bean 的 Bean 管理 Bean
- 管理 Bean 的 Bean 管理 , 管理 Bean 管理
- 管理 Bean 管理 Bean 管理
- 管理 Bean 管理 , 管理 Bean 管理 . 管理 **Bean** 管理 , 管理 **<bean>** 管理 **abstract** 管理 **true** , 管理 Spring 管理 Bean
- 管理 **<bean>** 管理 . 管理 : autowire, abstract 管理 .
- 管理 **Bean** 管理 **class** 管理 , 管理 Bean 管理 , 管理 . 管理 **abstract** 管理 **true**

Bean 的创建

- Spring 通过 **depends-on** 属性来指定 Bean 的依赖关系。例如：
`<bean id="bean1" class="com.atguigu.spring05.Bean1" depends-on="bean2"/>`
- 通过 **Bean** 的 `set` 方法来设置 Bean 的属性。

目录

- IOC & DI 简介
- 什么是 bean
 - 通过 XML 配置文件来配置
 - Bean 的创建方式 & 工厂 Bean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的 scope
 - 配置 Bean 的 lifecycle
 - 配置 Bean 的 init & destroy 方法
 - bean 的命名规则
 - bean 的 scope: singleton & prototype & WEB 容器
 - 配置 Bean 的 lazy-init
 - spEL
 - IOC 容器 Bean 的初始化
 - Spring 4.x 的 Bean 配置

Bean 生命周期

- 在 Spring 中，通过 `<bean>` 标签指定 **scope** 属性来指定 Bean 的生命周期。
- 在 Spring 中，通过 **IOC** 容器来管理 Bean，通过 **IOC** 容器来管理 Bean，通过 `getBean()` 方法来获取 Bean 实例。在 Spring 中，默认的生命周期是 **singleton**，即单例 Bean。

类别	说明
singleton	在 SpringIOC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在
prototype	每次调用 <code>getBean()</code> 时都会返回一个新的实例
request	每次 HTTP 请求都会创建一个新的 Bean，该作用域仅适用于 <code>WebApplicationContext</code> 环境
session	同一个 HTTP Session 共享一个 Bean，不同的 HTTP Session 使用不同的 Bean。该作用域仅适用于 <code>WebApplicationContext</code> 环境

目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 的创建方式 & 工厂方法 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的多种方式
 - 配置 Bean 的多种方式
 - 配置 Bean 的多种方式
 - bean 的创建方式
 - bean 的创建方式 singleton 与 prototype 与 WEB 应用
 - 配置 Bean 的多种方式
 - spEL
 - IOC 容器 Bean 的创建
 - Spring 4.x 的更新

依赖注入

- 容器管理 Bean 的创建，容器管理 Bean 的依赖注入（即：装配，注入）。容器管理 Bean 的装配。
- Spring 容器管理 PropertyPlaceholderConfigurer 的 BeanFactory 属性，容器管理 Bean 的装配。即 Bean 的装配。\${var} 属性，PropertyPlaceholderConfigurer 容器管理，容器管理。
- Spring 容器管理 \${propName} 容器管理。

PropertyPlaceholderConfigurer

- Spring 2.0:

```
<bean  
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
    <property name="location" value="classpath:jdbc.properties"></property>  
</bean>
```

- Spring 2.5** : **<context:property-placeholder>** :

– <beans> context Schema

```
<context:property-placeholder  
    location="classpath:db.properties"/>
```

目录

- IOC & DI 简介
- 什么是 bean
 - 通过 XML 配置文件来配置
 - Bean 的创建方式 & 工厂类 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置 Bean 的几种方式
 - 配置 Bean 的几种方式
 - 配置 Bean 的几种方式
 - bean 的几种作用域
 - bean 的几种作用域 singleton & prototype & WEB 作用域
 - 配置 Bean 的几种方式
 - SpEL
 - IOC 容器 Bean 的初始化
 - Spring 4.x 的改进

Spring 配置 SpEL

- Spring 配置 SpEL 配置
- EL 与 SpEL 的 `#{...}` 表达式 SpEL
- SpEL 在 bean 配置
- SpEL 配置
 - bean 的 id 为 bean 名称
 - 配置
 - 配置
 - 配置

SpEL 表达式

- 表达式
 - 整数 `<property name="count" value="#{5}"/>`
 - 浮点 `<property name="frequency" value="#{89.7}"/>`
 - 科学计数 `<property name="capacity" value="#{1e4}"/>`
 - **String** 字符串 `<property name="name" value="#{'Chuck'}"/>` 或 `<property name='name' value='#{"Chuck"}"/>`
 - Boolean 布尔 `<property name="enabled" value="#{false}"/>`

SpEL 如何 Bean 如何如何如何 1

- 如何如何如何

<!-- 通过 value 属性和 SpEL 配置 Bean 之间的应用关系 -->

<property name="prefix" value="#{prefixGenerator}"></property>

- 如何如何如何

<!-- 通过 value 属性和 SpEL 配置 suffix 属性值为另一个 Bean 的 suffix 属性值 -->

<property name="suffix" value="#{sequenceGenerator2.suffix}"/>

<!-- 通过 value 属性和 SpEL 配置 suffix 属性值为另一个 Bean 的方法的返回值 -->

- <property name="suffix" value="#{sequenceGenerator2.toString}"/>

<!-- 方法的连缀 -->

<property name="suffix"
value="#{sequenceGenerator2.toString().toUpperCase}"/>

SpEL 表达式 1

- 运算符 +, -, *, /, %, ^

```
<property name="adjustedAmount" value="#{counter.total + 42}"/>
<property name="adjustedAmount" value="#{counter.total - 20}"/>
<property name="circumference" value="#{2 * T(java.lang.Math).PI * circle.radius}"/>
<property name="average" value="#{counter.total / counter.count}"/>
<property name="remainder" value="#{counter.total % counter.count}"/>
<property name="area" value="#{T(java.lang.Math).PI * circle.radius ^ 2}"/>
```

- 字符串拼接

```
<constructor-arg
    value="#{performer.firstName + ' ' + performer.lastName}"/>
```

- 比较运算符 <, >, ==, <=, >=, lt, gt, eq, le, ge

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


SpEL 表达式 2

- 表达式 **and, or, not, |**

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

- **if-else** 表达式 **?· (ternary) ?· (Elvis)**

```
<constructor-arg  
value="#{songSelector.selectSong()=='Jingle Bells'?piano:' Jingle Bells '}'"/>
```

- **<constructor-arg**

```
value="#{kenny.song ?: 'Greensleeves'}"/>
```

```
<constructor-arg
```

```
value="#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}'}"/>
```

- 表达式 **matches**

SpEL 表达式 Bean 表达式 2

- 表达式 `T()` 表示 Class Object 表达式

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

目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 的创建方式 & 工厂 Bean
 - IOC 的 BeanFactory & ApplicationContext
 - 配置 Bean 的 scope
 - 配置 Bean 的 lifecycle
 - 配置 Bean 的 name
 - bean 的 scope 与 singleton 与 prototype 与 WEB 的 scope
 - 配置 Bean 的 init-method
 - spEL
 - IOC 的 Bean 的注入
 - Spring 4.x 的 Bean 的注入

IOC 管理 Bean 生命周期

- **Spring IOC** 管理 **Bean** 生命周期, Spring 管理 Bean 生命周期。
- Spring IOC 管理 Bean 生命周期:
 - 创建 Bean 实例
 - 对 Bean 进行初始化 Bean 实例
 - 对 **Bean** 进行销毁
 - Bean 实例
 - 销毁, 对 **Bean** 进行销毁
- 对 Bean 进行 init-method 和 destroy-method 操作, 对 Bean 进行销毁。

Bean 生命周期

- Bean 生命周期包括：
1. Bean 的创建
2. Bean 的初始化
3. Bean 的使用
4. Bean 的销毁
- Bean 的创建：IOC 容器根据 Bean 的定义，创建 Bean 实例。
- Bean 的初始化：在 Bean 创建完成后，容器会调用 Bean 的初始化方法，如 `org.springframework.beans.factory.config.BeanPostProcessor` 接口。

Object	postProcessAfterInitialization (Object bean, String beanName) Apply this BeanPostProcessor to the given new bean instance <i>after</i> any afterPropertiesSet or a custom init-method).
Object	postProcessBeforeInitialization (Object bean, String beanName) Apply this BeanPostProcessor to the given new bean instance <i>before</i> any afterPropertiesSet or a custom init-method).

Bean 的初始化 Bean 的销毁

- Spring IOC 管理 Bean 的生命周期 :
- 创建 Bean 对象
- 对 Bean 进行属性赋值 Bean 的初始化
- 对 Bean 进行 `postProcessBeforeInitialization` 操作
- 对 Bean 进行 `postProcessAfterInitialization` 操作
- Bean 的销毁
- 销毁 Bean 时，对 Bean 进行 `destroy` 操作

目录

- IOC & DI 简介
- 创建 bean
 - 使用 XML 配置文件创建
 - **Bean** 创建方式 & **FactoryBean**
 - IOC 容器 **BeanFactory** & **ApplicationContext** 简介
 - 使用 XML 创建
 - 使用 **bean** 创建
 - 使用 **bean** 创建
 - **bean** 创建方式
 - **bean** 创建 **singleton** & **prototype** & **WEB** 创建
 - 使用 **bean** 创建
 - **spEL**
 - IOC 容器 **Bean** 简介
 - Spring 4.x 简介

创建 Bean Bean

- 创建 Bean 的方式有三种：
1. 通过类名创建，
2. 通过工厂方法创建，
3. 通过构造方法创建。
- 创建 Bean 的方式有三种：
1. 通过类名创建，
2. 通过工厂方法创建，
3. 通过构造方法创建。

目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 与 FactoryBean
 - IOC 与 BeanFactory & ApplicationContext
 - 什么是 BeanFactory
 - 什么是 ApplicationContext
 - bean 的作用
 - bean 的 scope: singleton, prototype, WEB 等
 - bean 的 lifecycle
 - spEL
 - IOC 与 Bean 的关系
 - Spring 4.x 的变化

工厂 Bean 与 Spring IOC 容器

- Spring 容器管理 Bean, 管理 Bean, 管理 Bean, 工厂 Bean.
- 工厂 Bean 管理 Bean 类, 管理 Bean 类, 管理 Bean 类 get Object 方法

```
public interface FactoryBean {  
    //FactoryBean 返回的实例  
    Object getObject() throws Exception;  
  
    //FactoryBean 返回的类型  
    Class getObjectType();  
  
    //FactoryBean 返回的实例是否为单例  
    boolean isSingleton();  
}
```


目录

- IOC & DI 简介
- 创建 bean
 - 通过XML 配置文件创建 Bean 通过 Bean 工厂
 - Bean 工厂的接口 & 实现类 FactoryBean
 - IOC 容器 BeanFactory & ApplicationContext 简介
 - 配置Bean的工厂方法
 - 配置Bean的工厂方法
 - 配置Bean的工厂方法
 - bean 工厂的接口
 - bean 工厂 singleton & prototype & WEB 工厂
 - 配置Bean的工厂方法
 - spEL
 - IOC 容器 Bean 工厂

□ classpath □□□□□

- □□□□ (component scanning): Spring □□□ classpath □□□ □□ , □□□□□□□□□□□□□□□□ .
- □□□□□□ :
 - @Component: □□□□ , □□□□□□ Spring □□□□□
 - @Repository: □□□□□□□□
 - @Service: □□□□□ (□□□) □□
 - @Controller: □□□□□□□□
- □□□□□□□□□ , **Spring** □□□□□□□□□ : □□□□□□□□ , □□□□□□□□ . □□□□□□□□□ **value** **e** □□□□□□□□□□□□□□□□

classpath 扫描

- 扫描类路径，扫描 Spring 容器 **<context:component-scan>**

- **base-package** 扫描 Spring 容器。
- 扫描，。
- resource-pattern 扫描

```
<context:component-scan  
    base-package="com.atguigu.spring.beans"  
    resource-pattern="autowire/*.class"/>
```

- **<context:include-filter>** 扫描
- **<context:exclude-filter>** 扫描
- **<context:component-scan>** 扫描 **<context:include-filter>** 或 **<context:exclude-filter>** 扫描

□ classpath □□□□□

- <context:include-filter> □ <context:exclude-filter> □□□□□□□□
□□□□□□□□

类别	示例	说明
annotation	com.atguigu.XxxAnnotation	所有标注了 XxxAnnotation 的类。该类型采用目标类是否标注了某个注解进行过滤
assinable	com.atguigu.XxxService	所有继承或扩展 XxxService 的类。该类型采用目标类是否继承或扩展某个特定类进行过滤
aspectj	com.atguigu..*Service+	所有类名以 Service 结束的类及继承或扩展它们的类。该类型采用 AspectJ 表达式进行过滤
regex	com.\atguigu\.\anno\..*	所有 com.atguigu.anno 包下的类。该类型采用正则表达式根据类的类名进行过滤
custom	com.atguigu.XxxTypeFilter	采用 XxxTypeFilter 通过代码的方式定义过滤规则。该类必须实现 org.springframework.core.type.TypeFilter 接口

面试题

- `<context:component-scan>` 扫描的包下 `AutowiredAnnotation` `BeanPostProcessor` 类，扫描到的类上 `@Autowired` 和 `@Resource` 和 `@Inject` 注解。

① @Autowired ② Bean

- @Autowired ③ Bean ④
 - ⑤ , ⑥ (⑦ public), ⑧ @Autowired ⑨
 - ⑩ , ⑪ @Autowired ⑫ . ⑬ Spring ⑭ Bean ⑮ , ⑯ , ⑰ ⑱ , ⑲ @Autowired ⑳ required ㉑ false
 - ㉒ , ⑳ IOC ㉓ Bean ㉔ , ㉕ . ㉖ @Qualifier ㉗ Bean ㉘ . Spring ㉙ @Qualifiter ㉚ Bean ㉛
 - @Authwired ㉜ , ㉝ Spring ㉞ Bean ㉟ .
 - @Authwired ㊱ , ㊲ Spring ㊳ , ㊴ Bean.
 - @Authwired ㊵ java.util.Map ㊶ , ㊷ Map ㊸ String, ㊹ Spring ㊺ M ap ㊻ Bean, ㊼ Bean ㊽

① @Resource ② @Inject ③ Bean

- Spring ③ @Resource ② @Inject ④ @Autowired ⑤
⑥
- @Resource ⑦ Bean ⑧ Bean
⑨
- @Inject ② @Autowired ⑩ Bean ⑪ required ⑫
- ⑬ @Autowired ⑭

目录

- IOC & DI 简介
- 什么是 bean
 - 什么是 XML 配置文件
 - Bean 的创建方式 & 工厂 Bean
 - IOC 的 BeanFactory & ApplicationContext
 - 配置 Bean 的 scope
 - 配置 Bean 的 name
 - bean 的 scope
 - bean 的 singleton & prototype & WEB 的 scope
 - 配置 Bean 的 class
 - spEL
 - IOC 的 Bean 的 scope
 - Spring 4.x 的 Bean 的 scope

- ```

classDiagram
 class BaseService["BaseService<T>"]
 class BaseRepository["BaseRepository<T>"]
 class UserService
 class UserRepository
 class RoleService
 class RoleRepository

 BaseService <|-- UserService
 BaseService <|-- RoleService
 BaseRepository <|-- UserRepository
 BaseRepository <|-- RoleRepository
 BaseService --> BaseRepository
 UserService --> UserRepository
 RoleService --> RoleRepository

```

## 资源定位

- Spring 通过 `<import>` 指定资源文件的位置，Spring 通过资源文件的位置来加载资源
- `import` 指定 resource 文件 Spring 加载资源

| 地址前缀       | 示例                                        | 对应资源类型                                  |
|------------|-------------------------------------------|-----------------------------------------|
| classpath: | classpath:spring-mvc.xml                  | 从类路径下加载资源，classpath: 和 classpath:/ 是等价的 |
| file:      | file:/conf/security/spring-shiro.xml      | 从文件系统目录中装载资源，可采用绝对或相对路径                 |
| http://    | http://www.atguigu.com/resource/beans.xml | 从 WEB 服务器中加载资源                          |
| ftp://     | ftp://www.atguigu.com/resource/beans.xml  | 从 FTP 服务器中加载资源                          |



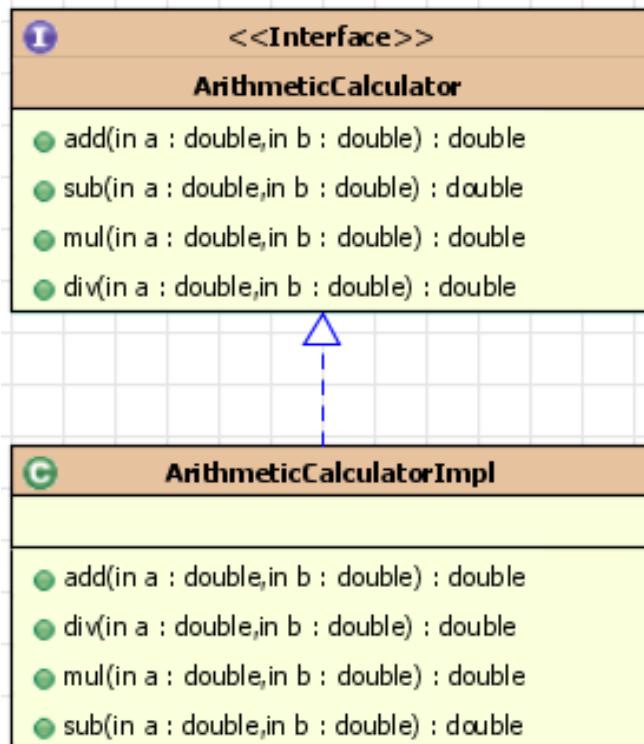
# Spring AOP

□□ : □□

□□□□ : □□□ - □□

# AOP 二

- WHY AOP



1- 接口定义

2- 实现类实现

□□□□□□

```
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
```

```
@Override
```

```
public void add(int i, int j) {
```

```
 System.out.println("日志:The method add begins with [" +
 i + ", " + j + "]");
```

```
 int result = i + j;
```

```
 System.out.println("result: " + result);
```

```
 System.out.println("日志:The method add ends with " + result);
```

```
}
```

```
@Override
```

```
public void sub(int i, int j) {
```

```
 System.out.println("日志:The method sub begins with [" +
 i + ", " + j + "]");
```

```
 int result = i - j;
```

```
 System.out.println("result: " + result);
```

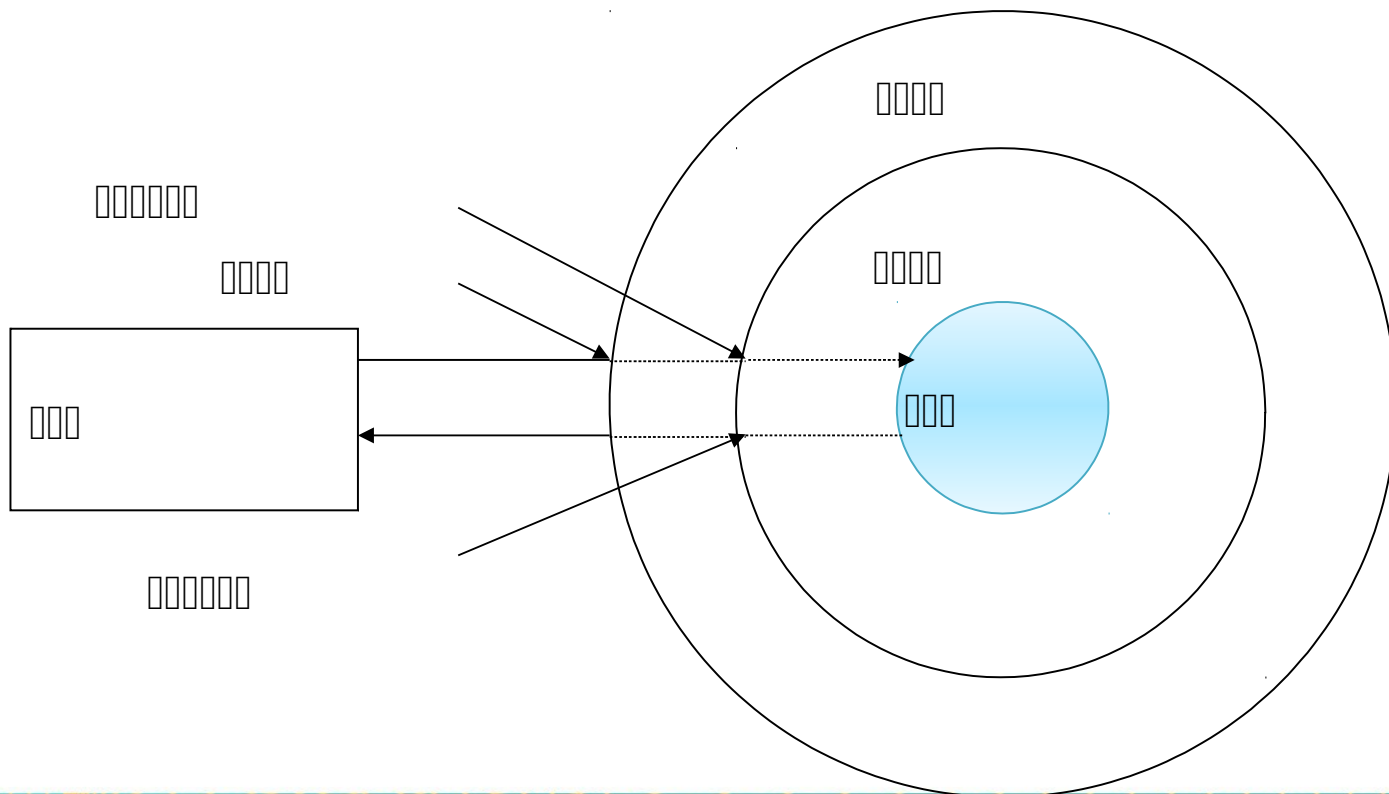
```
 System.out.println("日志:The method sub ends with " + result);
```

```
}
```

- 時間空間計算量が  $O(n^2)$  だが、メモリ量は  $O(1)$  .  
動的計画法
- 状態： 現在位置， 現在時刻， 現在持っているアイテムの種類 .  
最適解を求めたいとき， 状態の遷移を考える .

# 数据库系统组成

- 数据库系统组成：数据库、数据库管理系统、数据库管理员、数据库应用系统。





# CalculatorLoggingHandler

```
public class CalculatorLoggingHandler implements InvocationHandler {

 private Log log = LogFactory.getLog(this.getClass());

 private Object target;

 public CalculatorLoggingHandler(Object target) {
 super();
 this.target = target;
 }

 public Object invoke(Object proxy, Method method, Object[] args)
 throws Throwable {
 log.info("The method " + method.getName() + "() begins with " + Arrays.toString(args));
 Object result = method.invoke(target, args);
 log.info("The method " + method.getName() + "() ends with " + result);
 return result;
 }

 public static Object createProxy(Object target){
 return Proxy.newProxyInstance(target.getClass().getClassLoader(),
 target.getClass().getInterfaces(),
 new CalculatorLoggingHandler(target));
 }
}
```

# CalculatorValidationHandler

```
public class CalculatorValidationHandler implements InvocationHandler {
 private Object target;

 public CalculatorValidationHandler(Object target) {
 this.target = target;
 }

 public Object invoke(Object proxy, Method method, Object[] args)
 throws Throwable {
 for(Object arg : args){
 validate((Double) arg);
 }
 Object result = method.invoke(target, args);
 return result;
 }

 public static Object createProxy(Object target){
 return Proxy.newProxyInstance(target.getClass().getClassLoader(),
 target.getClass().getInterfaces(),
 new CalculatorValidationHandler(target));
 }

 private void validate(double a){
 if(a < 0)
 throw new IllegalArgumentException("Positive numbers only");
 }
}
```

□□□□

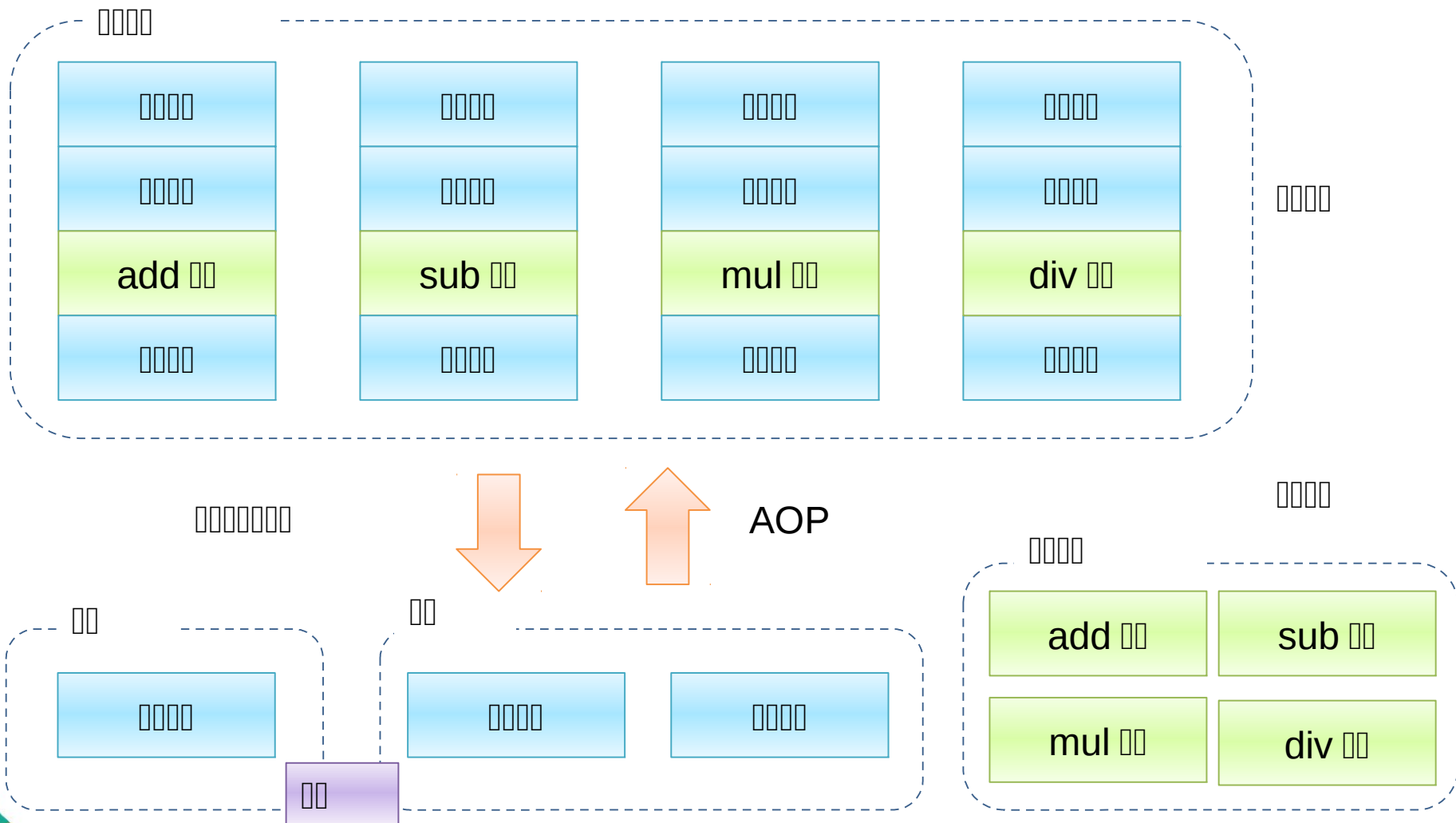
```
public class Main {
 public static void main(String[] args) {
 ArithmeticCalculator arithmeticCalculatorImpl =
 new ArithmeticCalculatorImpl();

 ArithmeticCalculator arithmeticCalculator
 = (ArithmeticCalculator) CalculatorValidationHandler
 .createProxy(CalculatorLoggingHandler
 .createProxy(arithmeticCalculatorImpl));
 System.out.println(arithmeticCalculator.add(-12, 13));
 }
}
```

# AOP 简介

- AOP(Aspect-Oriented Programming, 面向切面 ): 与面向对象编程 (Object-Oriented Programming, 面向对象 ) 类似。
- AOP 面向切面 (aspect), 面向切面编程。
- 与 AOP 类似, 面向切面编程, 面向切面编程, 面向切面编程, 面向切面编程, 面向切面编程。面向切面编程 ( 切面 )。
- AOP 简介：
  - 面向切面编程, 面向切面, 面向切面
  - 面向切面, 面向切面。

# AOP





# AOP 简介

- 切面 (Aspect): 横切关注点 (横切关注点) 的模块化实现
- 切点 (Advice): 切面要执行的代码
- 目标 (Target): 被切面的类
- 代理 (Proxy): 代理类
- 连接点 Joinpoint 是指程序运行到哪个位置时，切面要执行的代码。例如：ArithmeticCalculator#add() 方法调用的时候，切面要执行的代码。
- 切入点 pointcut 是指程序运行到哪个位置时，切面要执行的代码。例如：ArithmeticCalculator 类中的 add() 方法。AOP 是 Spring 框架中的一个重要特性，org.springframework.aop.Pointcut 是 AOP 的核心接口。

# Spring AOP

- **AspectJ** 是 Java 实现 AOP 的框架。
- 在 Spring2.0 之前，使用 AspectJ 需要通过 XML 配置 AOP

## Spring 集成 AspectJ 的步骤

- 在 Spring 项目中添加 AspectJ 包，在 **classpath** 中添加 **Aspect J** 包：aopalliance.jar 和 aspectj.weaver.jar 和 spring-aspects.jar
- 在 **aop Schema** 中添加 **<beans>** 元素。
- 在 Spring IOC 容器中配置 AspectJ 代理，在 **Bean** 配置项中添加 **XML** 元素 **<aop:aspectj-autoproxy>**
- 在 Spring IOC 容器中配置 Bean 配置项 **<aop:aspectj-autoproxy>** 元素，在配置项 **AspectJ** 配置项 **Bean** 配置项。

## □ AspectJ □□□□□□

- □□ **Spring** □□□ **AspectJ** □□ , □□□□ **IOC** □□□□□□□□ **Bean** □□ . □□ Spring IOC □□□□□□ AspectJ □□□□ , Spring IOC □□□□□□□□ □□ AspectJ □□□□□□ Bean □□□□ .
- □ **AspectJ** □□□ , □□□□□□□□ **@Aspect** □□□ **Java** □ .
- □□□□□□□□□□□□□□ **Java** □□ .
- AspectJ □□ 5 □□□□□□□□ :
  - **@Before:** □□□□ , □□□□□□□□□□
  - **@After:** □□□□ , □□□□□□□□□□
  - **@AfterRunning:** □□□□ , □□□□□□□□□□□□
  - **@AfterThrowing:** □□□□ , □□□□□□□□□□
  - **@Around:** □□□□ , □□□□□□□□

## 切面

- 切面 : 横切关注点
- 切面通过 `@Before` 注解 , 对目标方法进行拦截 .

```
@Aspect // 切面
public class CalculatorLoggingAspect {
 private Log log = LoggerFactory.getLog(this.getClass());

 @Before("execution(* ArithmeticCalculator.add(..))")
 public void logBefore(){
 log.info("The method add() begins");
 }
}
```

切面类 `CalculatorLoggingAspect` , 通过 `@Before` 注解对 `ArithmeticCalculator` 类中的 `add()` 方法进行拦截 , 在 `add()` 方法执行前 , 先执行 `logBefore()` 方法。



## AspectJ 切面

- 切面定义：
  - execution \* com.atguigu.spring.ArithmeticCalculator.\*(..): 对 ArithmeticCalculator 类的所有方法，在方法执行前，先执行 \* 后面的切面逻辑。.. 表示任意参数。
  - execution public \* ArithmeticCalculator.\*(..): 对 ArithmeticCalculator 类的所有方法。
  - execution public double ArithmeticCalculator.\*(..): 对 ArithmeticCalculator 类的所有方法，返回类型为 double。
  - execution public double ArithmeticCalculator.\*(double, ..): 对 ArithmeticCalculator 类的所有方法，第一个参数为 double。
  - execution public double ArithmeticCalculator.\*(double, double): 对 ArithmeticCalculator 类的所有方法，前两个参数均为 double。

## 织入点

- 在 AspectJ 中，织入点使用 `&&`, `||`, `!` 来指定。

```
@Pointcut("execution(* *.add(int, ..)) || execution(* *.sub(int, ..))")
private void loggingOperation(){}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}
```

## Spring AOP

- 通过 AOP 代理类，通过 `JoinPoint` 对象，可以获取到被代理类的方法名、参数列表、返回值类型等。

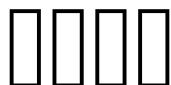
```
@Aspect
public class CalculatorLoggingAspect {
 private Log log = LoggerFactory.getLog(this.getClass());

 @Before("execution(* *.*(..))")
 public void logBefore(JoinPoint joinPoint) {
 log.info("The method " + joinPoint.getSignature().getName()
 + "() begins with " + Arrays.toString(joinPoint.getArgs()));
 }
}
```

通过 `JoinPoint` 对象，可以获取到被代理类的方法名、参数列表、返回值类型等。

通过 `JoinPoint` 对象，可以获取到被代理类的方法名、参数列表、返回值类型等。

通过 `JoinPoint` 对象，可以获取到被代理类的方法名、参数列表、返回值类型等。



- ```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LoggerFactory.getLog(this.getClass());

    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }

    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```



- ```
@Aspect
public class CalculatorLoggingAspect {
 private Log log = LoggerFactory.getLog(this.getClass());

 @Before("execution(* *.*(..))")
 public void logBefore(JoinPoint joinPoint){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() begins with " + Arrays.toString(joinPoint.getArgs()));
 }

 @AfterReturning("execution(* *.*(..))")
 public void logAfterReturning(JoinPoint joinPoint){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() ends");
 }
}
```



## 返回结果后置通知

- 返回结果后置通知，使用 **returning** 属性，@AfterReturning 注解，返回结果后置通知。返回结果后置通知。
- 返回结果后置通知，使用 **returning** 属性，Spring AOP 返回结果后置通知。
- 返回结果后置通知 **pointcut** 属性

```
@AfterReturning(pointcut="execution(* *.*(..))", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() ends with " + result);
}
```

## 异常

- 异常的分类
- **throwing** 异常 **@AfterThrowing** 异常, 异常处理. Throwable 异常. 异常处理.
- 异常处理, 异常处理. 异常处理.

```
@AfterThrowing(pointcut="execution(* *.*(..))", throwing="e")
public void logAfterThrowing(JoinPoint joinPoint, ArithmeticException e){
 log.info("An exception " + e + " has been throwing in "
 + joinPoint.getSignature().getName() + "()");
}
```

## 织梦

- 织梦框架的织梦器，织梦器。织梦器。
- 织梦器，织梦器 **ProceedingJoinPoint**。织梦器 **JoinPoint** 织梦器，织梦器，织梦器。
- 织梦器 **ProceedingJoinPoint** 的 **proceed()** 织梦器。  
织梦器，织梦器。
- 织梦器：织梦器，织梦器 **joinPoint.proceed();** 织梦器，织梦器。

# 织梦网

```
@Around("execution(* *.*(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable{
 log.info("The method " + joinPoint.getSignature().getName()
 + "() begins with " + Arrays.toString(joinPoint.getArgs()));

 try {
 joinPoint.proceed();
 log.info("The method " + joinPoint.getSignature().getName()
 + "() ends");
 } catch (Throwable e) {
 log.info("An exception " + e + " has been throwing in "
 + joinPoint.getSignature().getName() + "()");
 throw e;
 }
}
```

## Ordered

- 使用Ordered接口，@Order注解，Ordered接口实现类。
- 使用Ordered接口 Ordered 接口 @Order 注解。
- 实现 Ordered 接口，getOrder() 方法返回一个值，表示排序。
- 使用 @Order 注解，实现Ordered接口

```
@Aspect
@Order(0)
public class CalculatorValidationAspect {

 @Aspect
 @Order(1)
 public class CalculatorLoggingAspect {
```



## 织梦工厂

- 织梦 AspectJ 织梦 , 织梦织梦织梦织梦织梦 . 织梦织梦织梦织梦织梦织梦 .
- 织梦 AspectJ 织梦 , 织梦 @Pointcut 织梦织梦织梦织梦 . 织梦织梦织梦 , 织梦织梦织梦织梦织梦织梦 .
- 织梦织梦织梦织梦织梦织梦 . 织梦织梦织梦 , 织梦织梦织梦织梦 . 织梦 , 织梦织梦 public. 织梦织梦 , 织梦织梦 . 织梦织梦织梦织梦 , 织梦织梦 .
- 织梦织梦织梦织梦织梦 .

# 日志打印

```
@Pointcut("execution(* *.*(..))")
private void loggingOperation(){}

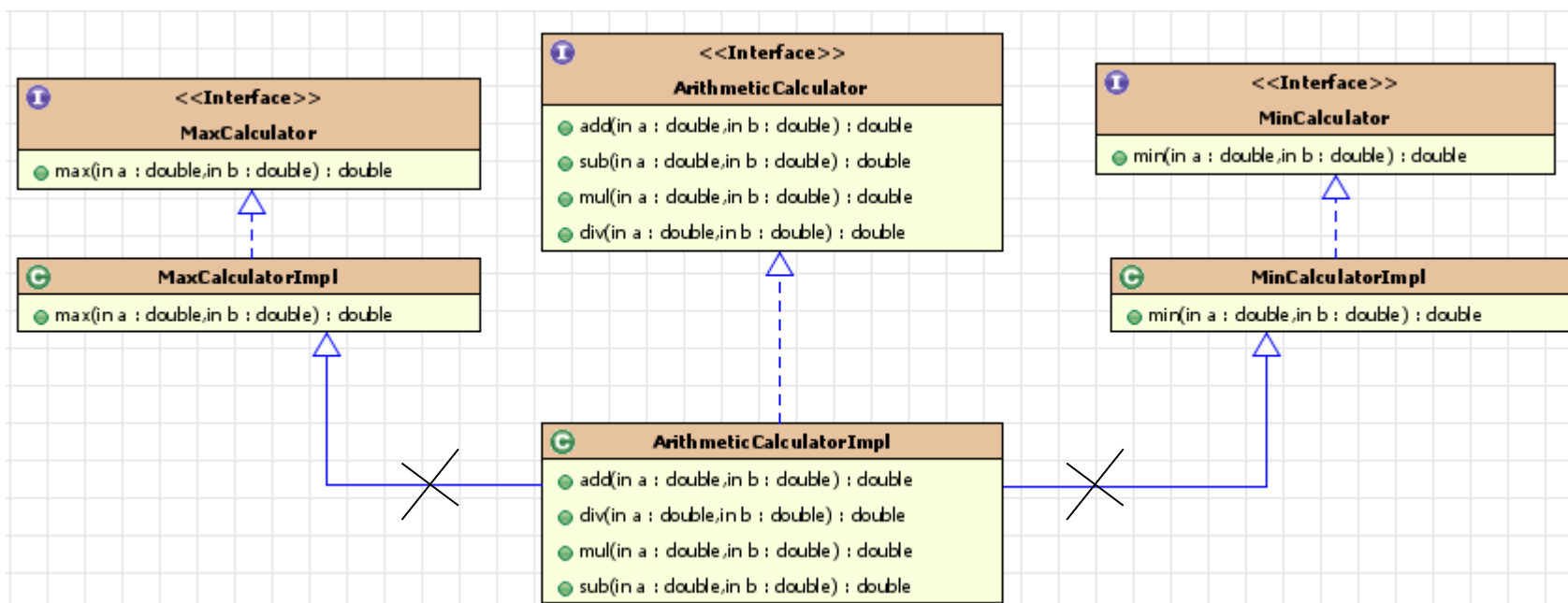
@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}

@AfterReturning(pointcut="loggingOperation()", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result){
 log.info("The method " + joinPoint.getSignature().getName()
 + "() ends with " + result);
}

@AfterThrowing(pointcut="loggingOperation()", throwing="e")
public void logAfterThrowing(JoinPoint joinPoint, ArithmeticException e){
 log.info("An exception " + e + " has been throwing in "
 + joinPoint.getSignature().getName() + "()");
}
```

# 接口

- 接口是抽象的，不能实例化，只能被实现，接口是抽象的，不能实例化，只能被实现。



## 工厂方法

- 工厂方法 `MaxCalculatorImpl` 和 `MinCalculatorImpl`, 和 `ArithmeticCalculatorImpl` 实现 `MaxCalculator` 和 `MinCalculator` 接口。工厂方法 `MaxCalculatorImpl` 和 `MinCalculatorImpl` 实现 `ArithmeticCalculatorImpl` 接口。
- 工厂方法
- 工厂方法, 工厂方法 `@DeclareParents` 工厂方法。
- 工厂方法 `value` 工厂方法 `.value` 工厂方法 `AspectJ` 工厂方法, 工厂方法 `defaultImpl` 工厂方法。

## □□□□□□□□

```
@Aspect
public class CalculatorLoggingAspect implements Ordered{
 private Log log = LogFactory.getLog(this.getClass());

 @DeclareParents(value="* *.Arithmetic*", defaultImpl=MaxCalculatorImpl.class)
 private MaxCalculator maxCalculator;

 @DeclareParents(value="* *.Arithmetic*", defaultImpl=MinCalculatorImpl.class)
 private MinCalculator minCalculator;

 MinCalculator minCalculator = (MinCalculator)
 ctx.getBean("airthmeticCalculator");
 minCalculator.min(1, 2);
}
```



## 聊聊 XML 配置

- 聊聊 AspectJ 配置, Spring 聊聊 Bean 配置. 聊聊 aop schema 的 XML 配置.
- 聊聊, 聊聊 XML 配置. 聊聊 AspectJ 的, 聊聊 Aspect J 的, 聊聊 XML 配置 Spring 的. 聊聊 AspectJ 的 AOP 配置, 聊聊.

## XML ----

- XML 配置文件, 使用 `<beans>` 定义 aop Schema
- 定义 Bean 配置, 使用 Spring AOP 配置 `<aop:config>` 配置  
点切面, 使用 `<aop:aspect>` 配置切面 Bean 配置.
- 使用 Bean 配置, 使用 `<aop:aspect>` 配置

## 配置AOP

```
<bean id="calculatorLoggingAspect"
 class="org.simpleit.CalculatorLoggingAspect"></bean>

<bean id="calculatorValidationAspect"
 class="org.simpleit.CalculatorValidationAspect"></bean>

<aop:config>
 <aop:aspect id="loggingAspect"
 ref="calculatorLoggingAspect"></aop:aspect>

 <aop:aspect id="validationAspect"
 ref="calculatorValidationAspect"></aop:aspect>
</aop:config>
```

## XML ----

- `<aop:pointcut>` 定义
- `<aop:aspect>` 定义 , `<aop:config>` 定义 .
  - `<aop:aspect>` 定义 : 定义
  - `<aop:config>` 定义 : 定义
- XML 中 AOP 配置

# 计算器

```
<aop:config>
 <aop:pointcut id="testOperation"
 expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>

 <aop:aspect id="loggingAspect"
 ref="calculatorLoggingAspect">
 </aop:aspect>

 <aop:aspect id="validationAspect"
 ref="calculatorValidationAspect">
 </aop:aspect>
</aop:config>
```



## XML 配置

- 使用 aop Schema 定义，使用 XML 配置。
- 使用 <pointcut-ref> 定义点切面，使用 <pointcut> 定义点切面。method 定义点切面。

□□□□□□□□

```
<aop:config>
 <aop:pointcut id="testOperation"
 expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>

 <aop:aspect id="loggingAspect"
 ref="calculatorLoggingAspect">
 <aop:after method="logBefore"
 pointcut-ref="testOperation"/>
 </aop:aspect>

 <aop:aspect id="validationAspect"
 ref="calculatorValidationAspect">
 <aop:before method="validateBefore"
 pointcut-ref="testOperation"/>
 </aop:aspect>
</aop:config>
```

## 配置

- 配置 `<aop:declare-parents>` 配置

```
<aop:aspect id="loggingAspect"
 ref="calculatorLoggingAspect">
 <aop:after method="logBefore"
 pointcut-ref="testOperation"/>
```

```
<aop:declare-parents
 types-matching="org.simpleit.bean.Arithmetic*"
 implement-interface="org.simpleit.bean.MinCalculator"
 default-impl="org.simpleit.bean.MinCalculatorImpl"/>
```

```
</aop:aspect>
```

# Spring JDBC 案例

# JdbcTemplate 简介

- 简介 JDBC 数据库连接, Spring 对 JDBC API 进行了封装, 简化了 JDBC 操作。
- 简介 Spring JDBC 数据库连接, **JDBC** 数据库连接池 JDBC 数据库连接池。数据库连接池, 数据库连接池。数据库连接池, 数据库连接池, 数据库连接池。



# ▯▯ JdbcTemplate ▯▯▯▯▯

- ▯ sql ▯▯▯▯▯▯▯▯▯▯ :

## **update**

```
public int update(String sql,
 Object... args)
 throws DataAccessException
```

- **batchUpdate**

```
public int[] batchUpdate(String sql,
 List<Object[]> batchArgs)
```

# 02 JdbcTemplate 00000

- 0000 :

## queryForObject

```
public <T> T queryForObject(String sql,
 ParameterizedRowMapper<T> rm,
 Object... args)
 throws DataAccessException
```

- [[org.springframework.jdbc.core.simple](#)  
**Class [ParameterizedBeanPropertyRowMapper](#)<T>**  
[java.lang.Object](#)  
└ [org.springframework.jdbc.core.BeanPropertyRowMapper](#)  
 └ [org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper](#)<T>

# ▯▯ JdbcTemplate ▯▯▯▯▯

- ▯▯▯▯ :

## **query**

```
public <T> List<T> query(String sql,
 ParameterizedRowMapper<T> rm,
 Object... args)
 throws DataAccessException
```

- ▯▯ **queryForObject**

```
public <T> T queryForObject(String sql,
 Class<T> requiredType,
 Object... args)
 throws DataAccessException
```

## 二 JDBC 二二二

- 二二二二二二 JdbcTemplate 二二二 , 二二二二二二 .
- **JdbcTemplate** 二二二二二二二二二 , 二二二二 IOC 二二二二二二二二二 , 二二二二二二二二二 二 DAO 二二二 .
- JdbcTemplate 二二二 Java 1.5 二二 ( 二二二 , 二二 , 二二二二 ) 二二二二
- Spring JDBC 二二二二二二 JdbcDaoSupport 二二二 DAO 二二 . 二二二 二 jdbcTemplate 二二 , 二二二 IOC 二二二二 , 二二二二二二二二二 .

## [[ JDBC ]]]

```
<bean id="jdbcTemplate"
 class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="personDAO"
 class="org.simpleit.jdbc.PersonDAO">
 <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
```



# 02 JdbcDaoSupport 0000

```
public class PersonDAO extends JdbcDaoSupport
```

```
<bean id="personDAO"
 class="org.simpleit.jdbc.PersonDAO">
 <property name="dataSource" ref="dataSource" />
</bean>
```

## □ JDBC □□□□□□□□

- □□□□ JDBC □□□ , SQL □□□□□□□ ? □□ , □□□□□□□□ . □□□□□□□□ , □□□□□□□□□□ , □□□□□□□□□□ .
- □ Spring JDBC □□□ , □□ SQL □□□□□□□□□□□□□□ (named parameter).
- □□□□ : SQL □□□ ( □□□□□□ ) □□□□□□□□□□ . □□□□□□□□□□ , □□□□□□□□ . □□□□□□□□□□□□□□□□□□□□
- □□□□□□□□ NamedParameterJdbcTemplate □□□□□□

## □ JDBC □□□□□□□□

- □ SQL □□□□□□□□ , □□□□ Map □□□□□ , □□□□
- □□□□ `SqlParameterSource` □□
- □□□□□□□□ Map □ `SqlParameterSource` □□□

### update

```
public int update(String sql,
 Map args)
 throws DataAccessException
```

### batchUpdate

```
public int[] batchUpdate(String sql,
 Map[] batchValues)
```

### update

```
public int update(String sql,
 SqlParameterSource args)
 throws DataAccessException
```

### batchUpdate

```
public int[] batchUpdate(String sql,
 SqlParameterSource[] batchArgs)
```

Spring<sup>ing</sup>

## 数据库

- 数据库系统是由数据库、数据库管理系统、数据库应用程序、数据库管理员、数据库用户等组成的。
  - 数据库 (database): 存储在计算机中、有组织的、共享的数据集合。
  - 数据库管理系统 (DBMS): 位于用户和数据库之间的一层数据管理软件。
  - 数据库应用程序 (database application): 使用数据库管理系统提供的接口，对数据库进行操作的程序。
  - 数据库管理员 (DBA): 负责数据库的日常维护和管理的人员。
  - 数据库用户 (database user): 使用数据库应用程序的人员。
- 数据库系统的特点：
  - 数据共享性: 数据库系统允许不同的用户共享数据库中的数据。
  - 数据独立性: 数据库系统具有较高的数据独立性，即数据的物理结构和逻辑结构是相互独立的。
  - 数据安全性: 数据库系统具有完善的安全机制，可以保证数据的安全。
  - 数据完整性: 数据库系统具有完善的完整性约束机制，可以保证数据的完整性。
  - 数据并发控制: 数据库系统具有完善的并发控制机制，可以保证数据在并发操作下的正确性。
  - 数据恢复: 数据库系统具有完善的数据恢复机制，可以保证数据在发生故障后的恢复。
- 数据库系统的组成 (ACID)
  - 原子性 (atomicity): 数据库事务中的所有操作，要么全部完成，要么全部不完成，不会因故障而半途而废。
  - 一致性 (consistency): 数据库事务执行前后，数据库必须处于一致状态，即数据的完整性、有效性、正确性。
  - 隔离性 (isolation): 数据库事务的执行是相互隔离的，一个事务的执行不会受到其他事务的影响。
  - 持久性 (durability): 数据库事务一旦提交，其结果就是永久性的，不会因为故障而丢失。





- 步骤 :

- 创建数据库连接池
- 使用 JDBC 连接数据库, 获取数据库连接, 释放数据库连接

```
public void purchase(String isbn, String username){
 Connection conn = null;

 try {
 conn = dataSource.getConnection();
 conn.setAutoCommit(false);

 //...

 conn.commit();
 } catch (SQLException e) {
 e.printStackTrace();
 if(conn != null){
 try {
 conn.rollback();
 } catch (SQLException e1) {
 e1.printStackTrace();
 }
 }
 throw new RuntimeException(e);
 } finally{
 if(conn != null){
 try {
 conn.close();
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
 }
}
```

# Spring 框架

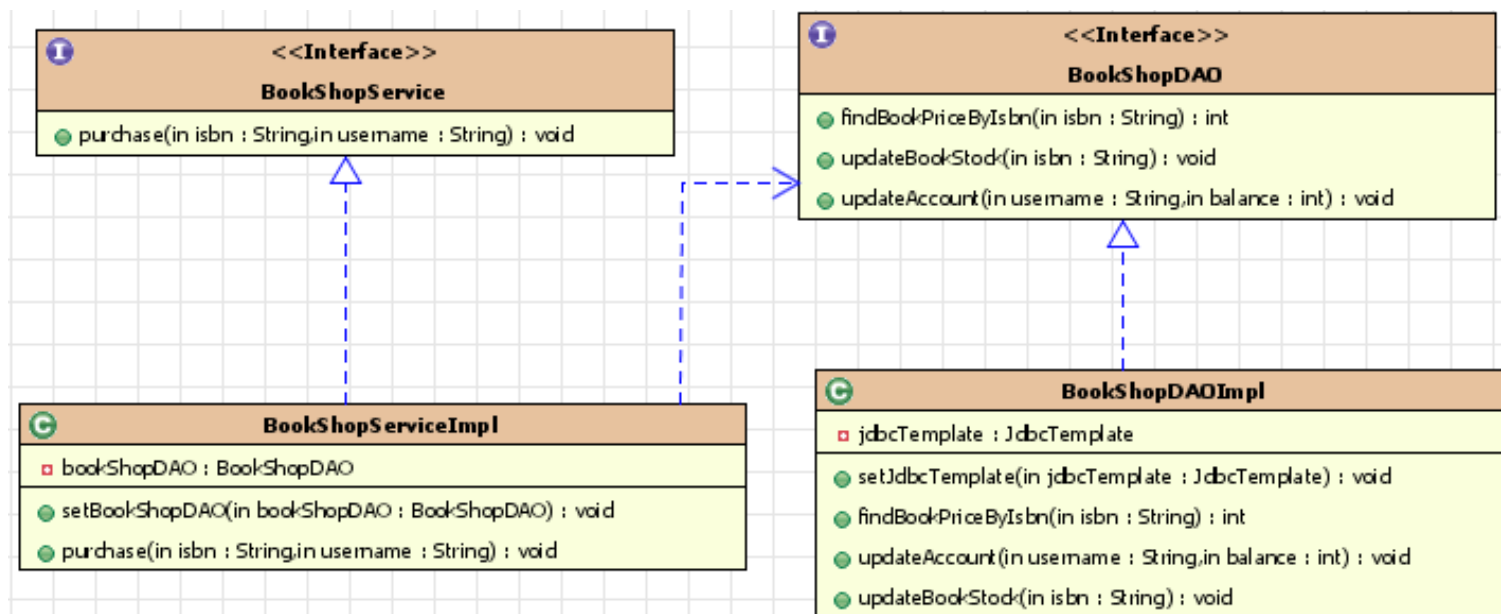
- 框架，**Spring** 框架 **API** 框架。 框架， 框架 API, 框架 Spring 框架。
- Spring 框架， 框架。
- 框架： 框架。 框架， 框架。
- 框架： 框架。 框架， 框架。 框架， 框架 AOP 框架。 **Spring** 框架 **Spring AOP** 框架。

# Spring 框架

- Spring 框架 API 文档 . 框架 API, 文档 . 框架 , 框架 .
- Spring 框架 框架 框架 . 框架  
Spring 框架 ( 框架 ), 框架 `org.springframework.transaction`  
**Interface PlatformTransactionManager**

# Spring 数据库事务管理

- org.springframework.jdbc.datasource  
**Class DataSourceTransactionManager** : 数据库事务管理 , 数据库 JDBC 事务
- org.springframework.transaction.jta  
**Class JtaTransactionManager** : 在 JavaEE 环境下 JTA(Java Transaction API) 事务
- org.springframework.orm.hibernate3  
**Class HibernateTransactionManager** 在 Hibernate 环境下
- .....
- 数据库事务 Bean 在 Spring IOC 容器





□□□□□□□□

Account □

| username ▲ | balance |
|------------|---------|
| Tom        | 30      |

Book □

| isbn ▲ | book_name | price |
|--------|-----------|-------|
| 0001   | Java      | 50    |

Book\_STOCK □

| isbn ▲ | stock |
|--------|-------|
| 0001   | 10    |

## Spring AOP 概述

- 什么是 AOP
- Spring 2.x 版本开始支持，通过 tx Schema 中的 **<tx:advice>** 来配置，通过 beans Schema 中的 **<beans>** 来配置。
- 通过 aop:config 来配置，通过 aop:config 来配置，通过 aop:config 来配置。通过 **<aop:config>** 来配置，通过 aop:config 来配置。
- Spring AOP 概述，通过 aop:config 来配置，通过 aop:config 来配置。通过 **Spring AOP** 来配置。

□□□□□□□□□□□□□□□□

```
<bean id="bookShopService"
 class="org.simpleit.transaction.BookShopServiceImpl">
 <property name="bookShopDAO" ref="bookDAO"/>
</bean>
```

□□□□□□

```
<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <property name="dataSource" ref="dataSource"></property>
</bean>
```

□□□□□□

```
<tx:advice id="bookShopTxAdvice"
 transaction-manager="transactionManager">
</tx:advice>
```

□□ □□□□□□□□□□ ( □□□□□□□□□□□□ )

```
<aop:config>
 <aop:pointcut expression="execution(* *.BookShopService.*(..))"
 id="bookShopOperation"/>
 <aop:advisor advice-ref="bookShopTxAdvice"
 pointcut-ref="bookShopOperation"/>
</aop:config>
```

## □ @Transactional 配置

- 配置事务 , 配置 Bean 事务 , Spring 配置 @Transactional 配置。
- 配置事务 , 配置 @Transactional 配置。 配置 Spring AOP 配置 , 配置。
- 配置事务 @Transactional 配置。 配置 , 配置。
- 配置 Bean 配置 <tx:annotation-driven> 配置 , 配置。
- 配置 transactionManager, 配置 <tx:annotation-driven> 配置 transaction-manager 配置。 配置。

## ① @Transactional 配置

```
<bean id="jdbcTemplate"
 class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <property name="dataSource" ref="dataSource"></property>
</bean>

<context:component-scan base-package="org.simpleit.transaction_1"/>

<tx:annotation-driven/>
```



## 面试题

- 面试官问：Spring 中，Bean 的初始化顺序是怎样的？  
面试官问：Spring 中，Bean 的初始化顺序是怎样的？  
面试官问：Spring 中，Bean 的初始化顺序是怎样的？
- 面试官问：Spring 中，Bean 的初始化顺序是怎样的？  
面试官问：Spring 中，Bean 的初始化顺序是怎样的？  
面试官问：Spring 中，Bean 的初始化顺序是怎样的？

# Spring 事务传播属性



| 传播属性          | 描述                                                       |
|---------------|----------------------------------------------------------|
| REQUIRED      | 如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行          |
| REQUIRED_NEW  | 当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起                |
| SUPPORTS      | 如果有事务在运行，当前的方法就在这个事务内运行。否则它可以不运行在事务中。                    |
| NOT_SUPPORTED | 当前的方法不应该运行在事务中。如果有运行的事务，将它挂起                             |
| MANDATORY     | 当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常                         |
| NEVER         | 当前的方法不应该运行在事务中。如果有运行的事务，就抛出异常                            |
| NESTED        | 如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行。否则，就启动一个新的事务，并在它自己的事务内运行。 |

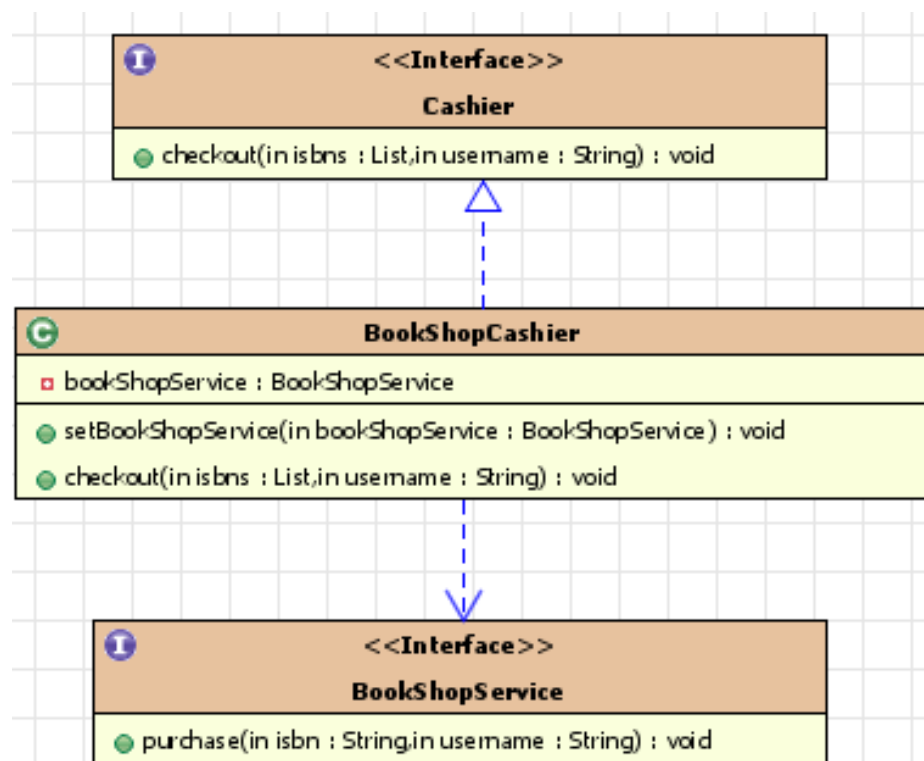
□□

- 创建 Cashier 类 : 实现接口
- 创建 Tom 类 , 实现接口 , 实现方法

| username ▲ | balance |
|------------|---------|
| Tom        | 60      |

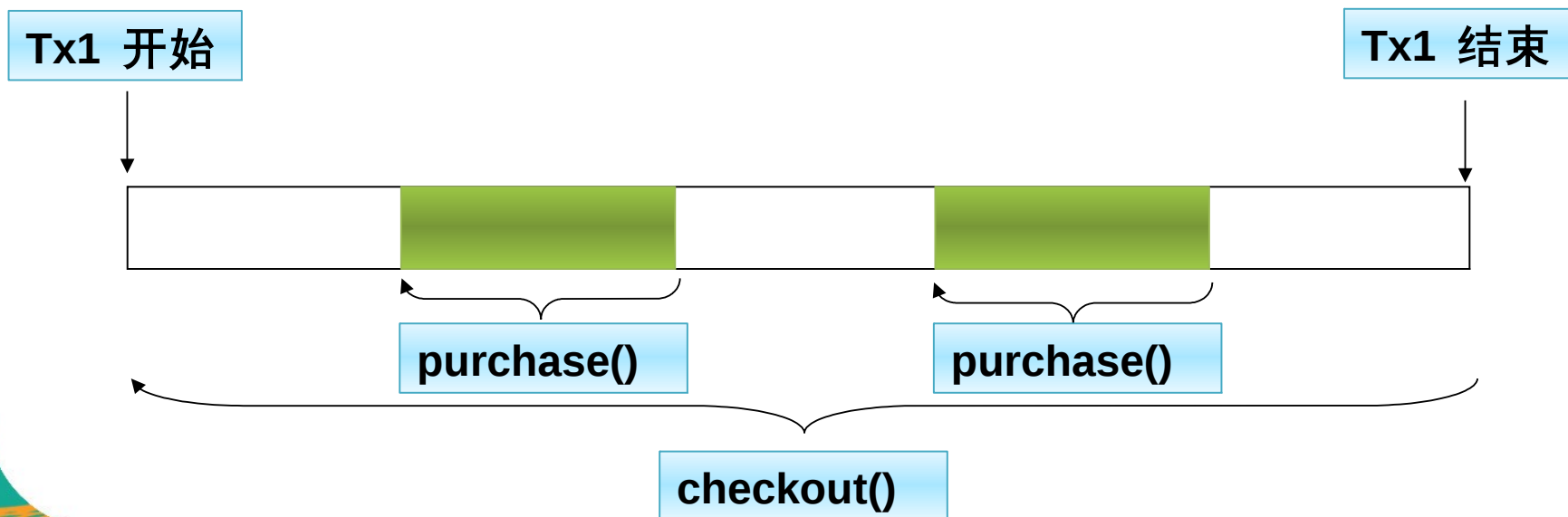
| isbn ▲ | book_name | price |
|--------|-----------|-------|
| 0002   | Oracle    | 80    |
| 0001   | Java      | 50    |

| isbn ▲ | stock |
|--------|-------|
| 0001   | 10    |
| 0002   | 10    |



# REQUIRED 案例

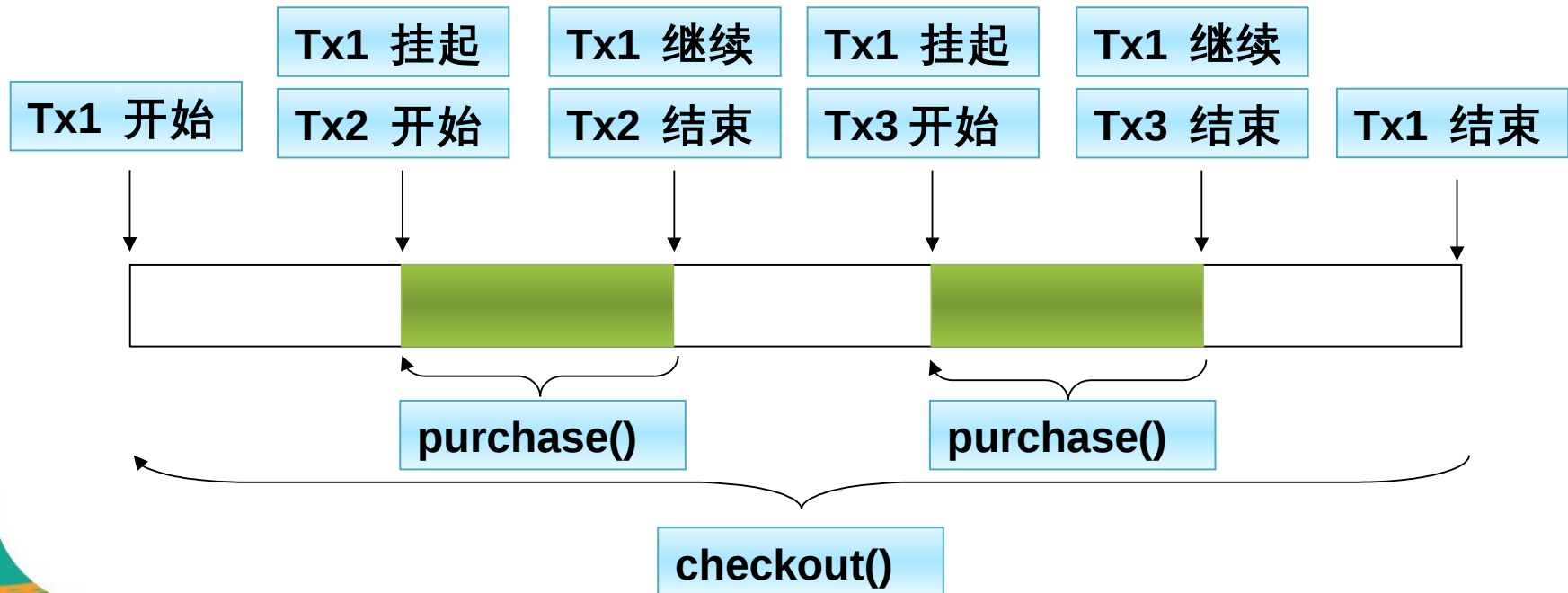
- 在 bookService 中 purchase() 方法调用了 checkout() 方法，而 checkout() 方法调用了 REQUIRED。因此 checkout() 方法必须在 purchase() 方法中调用，而不能在 purchase() 方法之外调用。
- 在 bookService 中 @Transactional 注解 propagation 属性



# REQUIRES\_NEW 解析

- REQUIRES\_NEW. 解析，解析。

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public void purchase(String isbn, String username) {
```





## □ Spring 2.x □□□□□□□□□□

- □ Spring 2.x □□□□□ , □□□□□□□□ <tx:method> □□□□□□□□□□

```
<tx:advice id="bookShopTxAdvice"
 transaction-manager="transactionManager">
 <tx:attributes>
 <tx:method name="purchase" propagation="REQUIRES_NEW"/>
 </tx:attributes>
</tx:advice>
```

- $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$
- $\text{Pr}(\text{process } i \text{ finishes}) = 1$  :
  - $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .
  - $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .
  - $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  ,  $\text{Pr}(\text{process } i \text{ finishes}) = 1$  .

[illegible]

# Spring 数据库事务隔离级别

| 隔离级别             | 描述                                                                      |
|------------------|-------------------------------------------------------------------------|
| DEFAULT          | 使用底层数据库的默认隔离级别。对于大多数数据库来说，默认隔离级别都是 READ_COMMITTED                       |
| READ_UNCOMMITTED | 允许事务读取未被其他事物提交的变更。脏读，不可重复读和幻读的问题都会出现                                    |
| READ_COMMITTED   | 只允许事务读取已经被其它事务提交的变更。可以避免脏读，但不可重复读和幻读问题仍然可能出现                            |
| REPEATABLE_READ  | 确保事务可以多次从一个字段中读取相同的值。在这个事务持续期间，禁止其他事物对这个字段进行更新。可以避免脏读和不可重复读，但幻读的问题仍然存在。 |
| SERIALIZABLE     | 确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入，更新和删除操作。所有并发问题都可以避免，但性能十分低下。  |

- 数据库事务隔离级别分为 4 种，分别是：READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE。
- Oracle 默认 2 种隔离级别：READ\_COMMITTED, SERIALIZABLE
- Mysql 默认 4 种隔离级别。

## 事务管理

- `@Transactional` 注解与 `Transaction` 注解 `isolation` 属性。

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
 isolation=Isolation.READ_COMMITTED)
public void purchase(String isbn, String username) {
```

- 在 Spring 2.x 版本中，使用 `<tx:method>` 配置。

```
<tx:advice id="bookShopTxAdvice"
 transaction-manager="transactionManager">
 <tx:attributes>
 <tx:method name="purchase"
 propagation="REQUIRES_NEW"
 isolation="READ_COMMITTED"/>>
 </tx:attributes>
 </tx:advice>
```



## 异常处理

- `RuntimeException` (Error 异常) 异常。 异常。  
异常。
- `@Transactional` 注解 `rollbackFor` 与 `noRollbackFor` 异常。 异常 `Class[]` 异常， 异常。
- `rollbackFor`: 异常
- `noRollbackFor`: 异常

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
 isolation=Isolation.READ_COMMITTED,
 rollbackFor={IOException.class, SQLException.class},
 noRollbackFor=ArithmeticException.class)
public void purchase(String isbn, String username) {
```

## 事务管理

- Spring 2.x 中，`<tx:method>` 标签用于配置事务。在 Spring 2.x 中，`<tx:method>` 标签用于配置事务。在 Spring 2.x 中，`<tx:method>` 标签用于配置事务。

```
<tx:advice id="bookShopTxAdvice"
 transaction-manager="transactionManager">
 <tx:attributes>
 <tx:method name="purchase"
 propagation="REQUIRES_NEW"
 isolation="READ_COMMITTED"
 rollback-for="java.io.IOException, java.sql.SQLException"
 no-rollback-for="java.lang.ArithmeticException"/>
 </tx:attributes>
</tx:advice>
```

[illegible]

## 事务管理

- 使用 `@Transactional` 注解 .

```
@Transactional(propagation=Propagation.REQUIRES_NEW,
 isolation=Isolation.READ_COMMITTED,
 rollbackFor={IOException.class, SQLException.class},
 noRollbackFor=ArithmeticException.class,
 readOnly=true,
 timeout=30)
public void purchase(String isbn, String username) {
```

- Spring 2.x 使用 `<tx:method>` 配置 .

```
<tx:advice id="bookShopTxAdvice"
 transaction-manager="transactionManager">
 <tx:attributes>
 <tx:method name="purchase"
 propagation="REQUIRES_NEW"
 isolation="READ_COMMITTED"
 rollback-for="java.io.IOException, java.sql.SQLException"
 no-rollback-for="java.lang.ArithmeticException"
 timeout="30"
 read-only="true"/>
 </tx:attributes>
</tx:advice>
```

# Spring 与 Hibernate



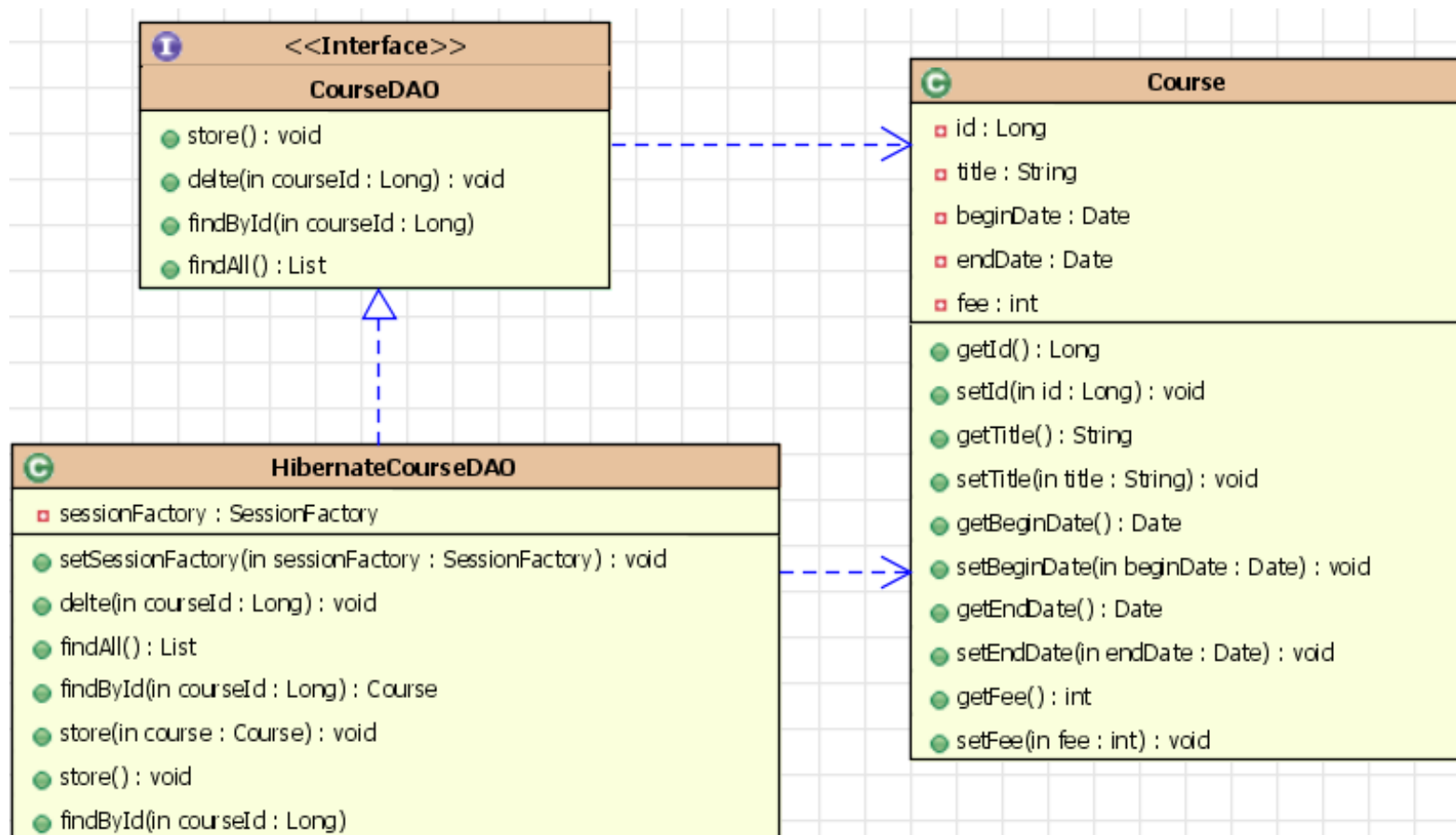
# Spring 与 Hibernate

- Spring 集成了 ORM 框架，支持 Hibernate JDO, TopLink, Ibatis 与 JPA
- **Spring** 支持 **ORM** 框架，支持 Hibernate 与 OR M 框架。
- Spring 2.0 支持 Hibernate 2.x 与 3.x。Spring 2.5 支持 Hibernate 3.1 及以上。

# Spring 如何 SessionFactory

- 使用 Hibernate 时，需要实现 Hibernate API 中的 SessionFactory 接口。在 Spring 中，SessionFactory 是一个 Bean (即 : Spring 管理 )
- Spring 管理 Bean, 通过 IOC 管理 SessionFactory 。

二



## Spring 配置 SessionFactory(1)

- 配置 **LocalSessionFactoryBean** 的 Bean, 通过 XML 配置 SessionFactory 的。
- 配置 Bean 的 configLocation 属性 Hibernate 的。

```
<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
 <property name="configLocation" value="hibernate.cfg.xml"/>
</bean>
```

```
<bean id="courseDAO"
 class="org.simpleit.HibernateCourseDAO">
 <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

## □ Spring □□□ SessionFactory(2)

- □□□ Spring IOC □□□□□□□□ . □□□□□□□□□□ LocalSession  
FactoryBean □ dataSource □□□ . □□□□□□□□□□□□□□  
Hibernate □□□□□□□□□□ .



## Spring 配置 SessionFactory(2)

```
<context:property-placeholder location="C3P0_config.properties"/>

<bean id="dataSource"
 class="com.mchange.v2.c3p0.ComboPooledDataSource">
 <property name="user" value="${user}"/>
 <property name="password" value="${password}"/>
 <property name="jdbcUrl" value="${jdbcUrl}"/>
 <property name="driverClass" value="${driverClass}"/>

 <property name="checkoutTimeout" value="${checkoutTimeout}"/>
 <property name="idleConnectionTestPeriod" value="${idleConnectionTestPeriod}"/>
 <property name="initialPoolSize" value="${initialPoolSize}"/>
 <property name="maxIdleTime" value="${maxIdleTime}"/>

 <property name="maxPoolSize" value="${checkoutTimeout}"></property>
 <property name="minPoolSize" value="${minPoolSize}"></property>
 <property name="maxStatements" value="${maxStatements}"></property>
</bean>

<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
 <property name="configLocation" value="hibernate.cfg.xml"/>
 <property name="dataSource" ref="dataSource"></property>
</bean>
```

## Spring 配置 SessionFactory(3)

- 配置 LocalSessionFactoryBean 类, 配置 Hibernate 配置。
- 配置 LocalSessionFactoryBean 的 **mappingResources** 配置 XML 配置文件。配置 String[] 数组。配置。
- 配置 hibernateProperties 配置。

# Spring 配置 SessionFactory(3)

```
<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource"></property>

 <property name="hibernateProperties">
 <props>
 <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
 <prop key="hibernate.show_sql">true</prop>
 <prop key="hibernate.hbm2ddl.auto">update</prop>
 </props>
 </property>

 <property name="mappingResources">
 <list>
 <value>org/simpleteit/Course.hbm.xml</value>
 </list>
 </property>
</bean>
```

## □ Spring □ ORM □□□□□□

- □□□□ ORM □□ , □□□□ DAO □□□□□□□□ . □□ : □□□□ Session □□ ; □□ , □□ , □□□□ .
- □ JDBC □□ , Spring □□□□□□□□ ----- □□□□□□ **DAO** □□□□□□ OR M □□□□□□ . □□ Spring □□□□□□□□ API □□□□□□□□□□□□ . □□□□□□ ORM □□ , □□□□□□□□□□□□□□□□ .



# Spring 数据库操作

支持类	JDBC	Hibernate
模板类	JdbcTemplate	HibernateTemplate
DAO 支持类	JdbcDaoSupport	HibernateDaoSupport
事务管理类	DataSourceTransactionManager	HibernateTransactionManager

- HibernateTemplate 封装了 Hibernate 数据库操作。
- HibernateTemplate 封装了 Hibernate 数据库操作，Spring 提供了 HibernateTemplate 类。



## ▯▯ Hibernate ▯▯

- HibernateTemplate 封装了 Hibernate 的 API 。 使用 DAO 接口 封装了 Hibernate 的 API ， 使用 DAO 接口 封装了 Hibernate 的 API 。
- 使用 DAO 接口 使用 @Transactional 注解 。
- HibernateTemplate 接口 ， 使用 Bean 接口 ， 使用 Hibernate DAO 。

# hibernate 配置

```
private HibernateTemplate hibernateTemplate;

public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
 this.hibernateTemplate = hibernateTemplate;
}

@Override
@Transactional
public void delete(Long courseId) {
 Course course =
 (Course) hibernateTemplate.get(Course.class, courseId);
 hibernateTemplate.delete(course);
}

@Override
@Transactional(readOnly=true)
public List<Course> findAll() {
 return hibernateTemplate.find("FROM Count");
}

@Override
@Transactional(readOnly=true)
public Course findById(Long courseId) {
 return (Course) hibernateTemplate.get(Course.class, courseId);
}

@Override
@Transactional
public void store(Course course) {
 hibernateTemplate.saveOrUpdate(course);
}
```

# Spring 整合 Hibernate

```
<bean id="transactionManager"
 class="org.springframework.orm.hibernate3.HibernateTransactionManager">
 <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="hibernateTemplate"
 class="org.springframework.orm.hibernate3.HibernateTemplate">
 <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

<bean id="courseDAO_"
 class="org.simpleit.HibernateCourseDAO_">
 <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

# ▯ HibernateTemplate ▯▯▯ Hibernate ▯▯ Session

```
hibernateTemplate.execute(new HibernateCallback() {
 @Override
 public Object doInHibernate(Session arg0) throws HibernateException,
 SQLException {
 //...
 return null;
 }
});
```

## DAO 与 Hibernate

- Hibernate DAO 接口由 HibernateDaoSupport 实现，setSessionFactory() 与 setHibernateTemplate() 方法。其中，DAO 实现类通过 getHibernateTemplate() 方法获取 HibernateTemplate。
- 实现 HibernateDaoSupport 接口需要实现 SessionFactory 接口，实现 HibernateTemplate 接口，而 HibernateDaoSupport 接口实现 SessionFactory 接口实现 HibernateTemplate 接口，实现 hibernateTemplate 接口。



## ▣ Hibernate 管理 Session 管理

- Spring 中 HibernateTemplate 管理 Session , 通过 DAO 接口 . 通过 HibernateTemplate 管理 DAO 管理 Spring 中 API
- 通过 HibernateTemplate 管理 Session 管理 Session 管理 .
- **Hibernate 管理 Session 管理 Spring 管理 , 管理 DAO 管理**
- 通过 beans.xml 管理 , 通过 Spring 管理 , 通过 Thread Local 管理 Session 管理

```
<prop key="hibernate.current_session_context_class">thread</prop>
```

## ▣ Hibernate 异常 Session 异常

- ▣ Hibernate 异常 , 异常 HibernateException.
- 异常 , ▣ Hibernate 异常 Spring ▣ DataAccessException 异常 , 异常 DAO 异常 @Repository 异常 .
- 异常  

org.springframework.dao.annotation  
Class PersistenceExceptionTranslationPostProcessor  
异常 , 异常

  
Hibernate 异常 Spring ▣ DataAccessException 异常  
异常 . ▣ Bean 异常 @Repository 异常 Bean  
异常 .

# Hibernate 数据库 Session(1)

- 在 Hibernate 3 中, SessionFactory 通过 getCurrentSession() 方法, 返回一个“线程”级别的 Session。
- Hibernate 中 CurrentSessionContext 类负责管理 hibernate.current\_session\_context\_class 中的“线程”
  - JTASessionContext: 通过 JTA 管理 Session 线程。
  - ThreadLocalSessionContext: 通过 ThreadLocal 管理 Session 线程。
  - ManagedSessionContext: 通过 ManagedSessionContext 管理 Session 线程。ManagedSessionContext 管理 Session 线程, 通过 flush 管理 Session 线程。

# Hibernate 配置 Session(2)

- 通过 ThreadLocalSessionContext 类，Hibernate 的 Session 通过 getCurrentSession() 方法获取，并返回 Session 对象。
- 通过 JTA 配置，配置 Hibernate 的 Session 工厂。
- 配置：
  - 通过 JTA 配置 Session 工厂：

```
<property name="hibernate.current_session_context_class">thread</property>
```

- 通过 JTA 配置 Session 工厂

```
<property name="hibernate.current_session_context_class">jta</property>
```

# Struts2



# 如何 web 容器使用 Spring

- 如何在 Servlet 容器 ContextLoaderListener, Web 容器使用 Spring 的 ApplicationContext 类。 如何初始化 ApplicationContext 容器 Web 容器 ServletContext 类。 如何, Servlet 容器 ServletContext 容器使用 Spring 容器。

# 如何 web 容器使用 Spring 容器

- 在 web.xml 文件中配置 Spring 容器 Servlet 容器  
`org.springframework.web.context`  
`Class ContextLoaderListener` 容器 Spring 容器 `ApplicationContext`  
ext 容器 `ServletContext` 容器。
- 在 web 容器 `contextConf`  
`org.springframework.web.context`  
`Class ContextLoaderListener` 容器中。容器 Bean 容器，容器  
容器。 `contextConfigLocation` 容器 `/WEB-INF/applic`  
`ationContext.xml`。容器 web 容器

# web.xml

```
<context-param>
 <param-name>contextConfiguration</param-name>
 <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
 <listener-class>
 org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
```

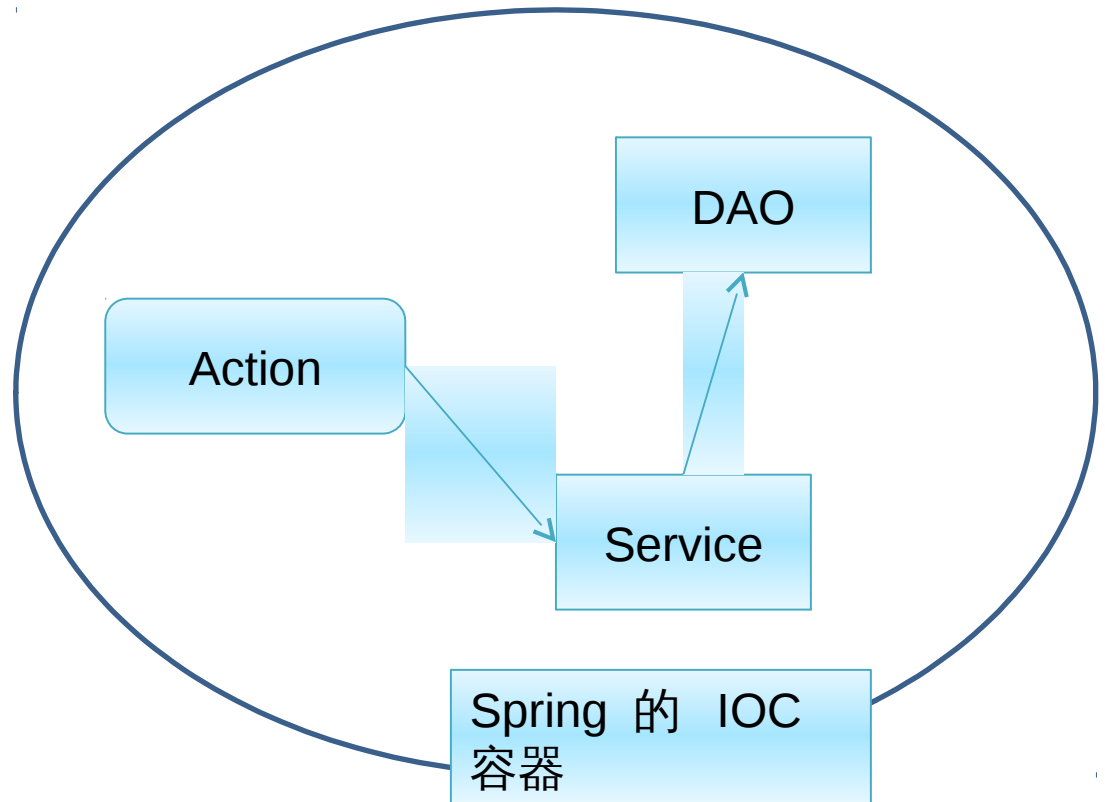
# web 容器 Spring 的 ApplicationContext 容器

- 容器  
org.springframework.web.context.support  
**Class WebApplicationContextUtils**

```
public static WebApplicationContext getRequiredWebApplicationContext(ServletContext sc)
throws IllegalStateException
```

## 容器 Spring 的 ApplicationContext 容器

```
WebApplicationContext applicationContext =
 WebApplicationContextUtils.getRequiredWebApplicationContext(this.getServletContext());
Test test = (Test) applicationContext.getBean("test");
test.hello();
```





## Struts2

- Struts2 需要 Spring 支持。
- Struts2 需要 Spring 容器：
  - 将 Action 放入 Spring 容器中，通过，配置，使用 Spring 容器 IOC 管理，使用
  - 将 Spring 容器，将 Spring 容器 Action 放入，将 Spring 容器 Action 放入。



## 配置

- 在 Spring 容器中配置，在 Spring 容器中配置 Action 类，在 Spring 容器中配置 Action 类。
- 配置项：Spring 容器中配置 struts.objectFactory.spring.autoWire 属性，配置项：
  - name: 配置项。
  - type: 配置项。配置项 type 为 Bean, 配置项；配置项 Bean, 配置项, 配置项
  - auto: Spring 配置项
  - constructor: 配置项, 配置项 constructor 配置项
- 配置项：
  - 在 Spring 中
  - 配置 struts 配置项
  - 在 spring 配置项, 配置项 Action 配置项