

Spring 学习目的

简化开发

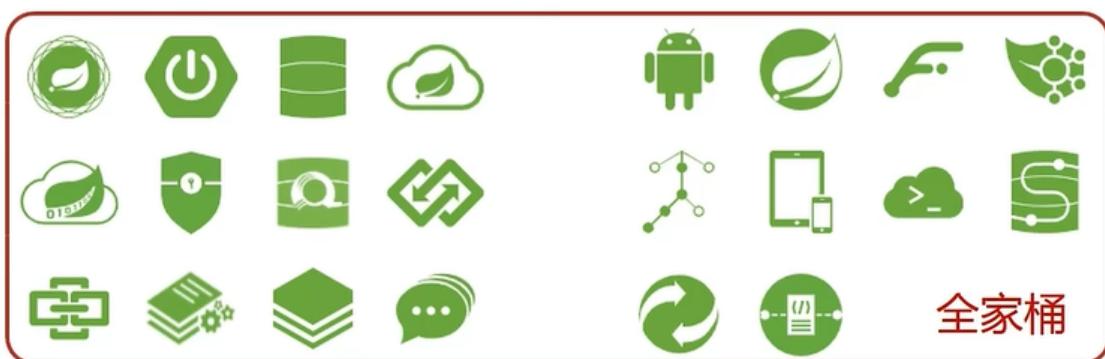
- IoC
- AOP
 - 事务处理

框架整合

- MyBatis
- MyBatis-plus
- Struts
- Struts2
- Hibernate

初识Spring

- 官网: spring.io
- Spring发展到今天已经形成了一种开发的生态圈, Spring提供了若干个项目, 每个项目用于完成特定的功能



Spring Framework



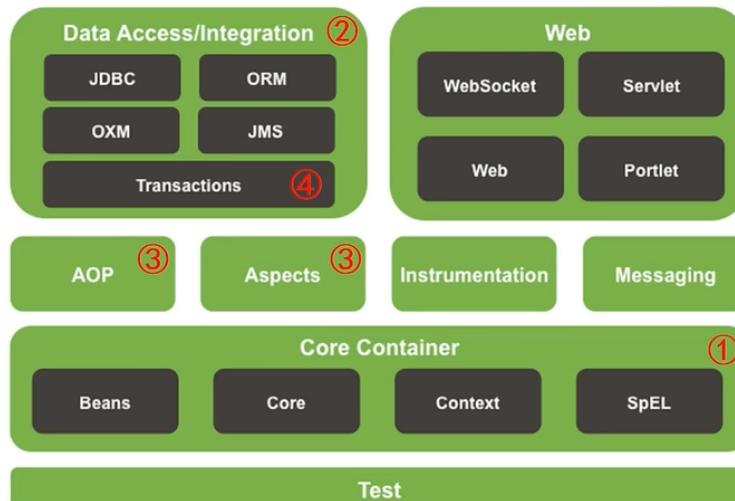
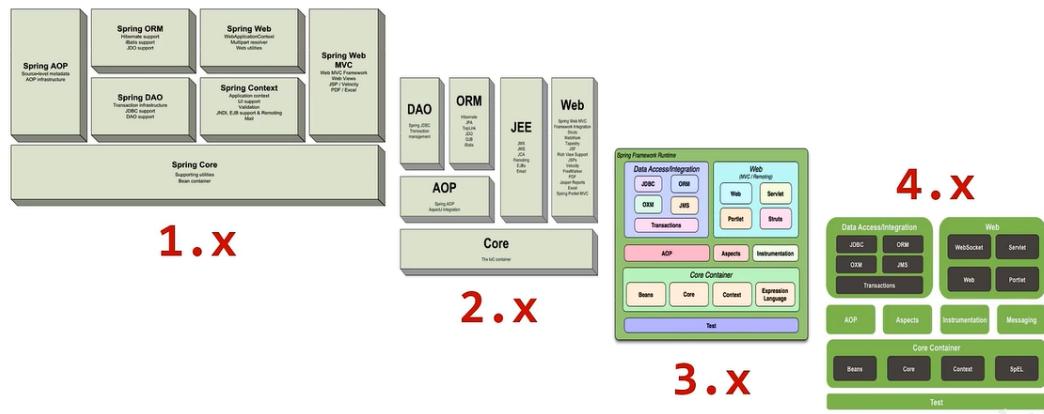
Spring Boot



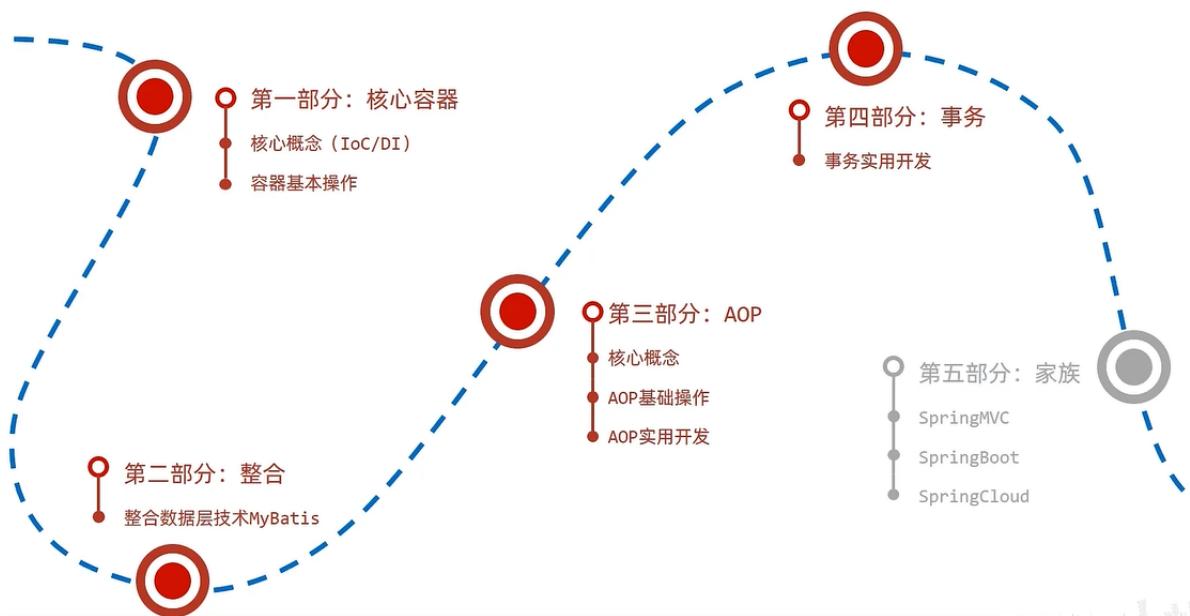
Spring Cloud

Spring Framework

- Spring Framework是Spring生态圈中最基础的项目, 是其他项目的根基



- Data Access: 数据访问
- Data Integration: 数据集成
- Web: Web 开发
- AOP: 面向切面编程
- Aspects: AOP 思想实现
- Core Container: 核心容器
- Test: 单元测试与集成测试

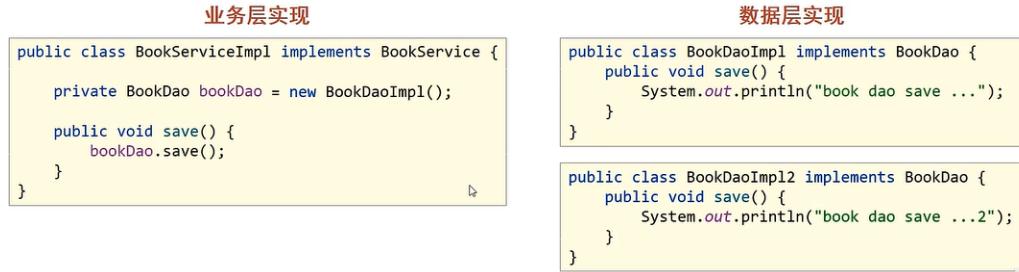


- 第一部分：核心容器
- 第二部分：整合
- 第三部分：AOP
- 第四部分：事务
- 第五部分：家族

IOC

核心概念

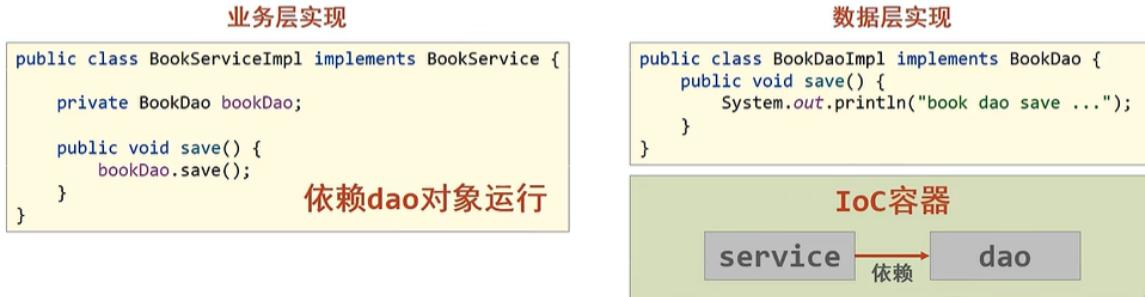
- 代码书写状况



如果要把Imp1改为Imp2,那么需要重新编译, 导致耦合度偏高

- 解决方案

- 使用对象时, 在程序中不要主动new产生对象, 转换为由尾部提供对象
- IoC (Inversion of Control) 控制反转
 - 对象的创建控制权由程序转移到外部, 这种思想称为控制反转
使用对象时, 由主动new产生对象转换为由外部提供对象, 此过程中对象创建控制权由程序转移到外部, 此思想称为控制反转
- Spring技术对IoC思想进行了实现
 - Spring提供了一个容器, 成为IoC容器, 用来充当IoC思想中的外部
 - IoC容器负责对象的创建、初始化等一系列工作, 被创建或被管理的对象在IoC容器中统称为Bean
- DI (Dependency Injection) 依赖注入
 - 在容器中建立bean与bean之间的依赖关系的整个过程, 称为依赖注入



目标: 充分解耦

使用IoC容器管理Bean (IoC)

在IoC容器内将有依赖关系的Bean进行关系绑定 (DI)

最终效果:

使用对象是不急你可以直接从IoC容器获取, 并且获取到的bean已经绑定了所有的依赖关系

IoC/DI

IoC容器

Bean

Bean管理XML方式

IOC入门案例思路分析

1. 管理什么? Service和Dao
 2. 如何将被管理的对象告知IoC容器? 配置
 3. 被管理的对象交给IoC容器, 如何获取到IoC容器? 接口
 4. IoC容器得到后, 如何从容器中获得bean? 接口方法
 5. 使用Spring导入哪些坐标? pom.xml
- pom.xml导坐标或者导入jar包

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.karos</groupId>
    <artifactId>Spring_Go</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Spring_Go</name>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-
context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.21</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

- 创建Spring的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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--导入Spring依赖, Maven项目导入坐标-->
    <!--配置bean-->
    <!--    id为 id-->
    <!--    class为定义bean的类型-->
    <bean id="bookdao" class="com.Karos.Dao.Demo.BookDemo"></bean>
    <bean id="bookservice" class="com.Karos.Services.Demo.BookServiceDemo">
    </bean>
</beans>
```

```

public class App2 {
    public static void main(String[] args) {
        //获取ioc容器
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("appilcationContext.xml");
        BookService A = (BookService) ctx.getBean("bookservice");
        A.save();
    }
}

```

```

public class BookServiceDemo implements BookService {
    BookDao book = new BookDemo();
    public void save() {
        System.out.println("业务层打印");
        book.save();
    }
}

```

```

public class BookDemo implements BookDao {

    public void save() {
        System.out.println("数据层打印");
    }
}

```

DI入门案例思路分析

1. 基于IoC管理bean
2. Service中使用new形式创建的Dao对象是否够保留? 否
3. Service中需要Dao对象如何进入Service中? 提供方法
4. Service与Dao间的关系描述?

DI关系, 一个set方法, 然后再bean中写

```

<bean id="bookdao" class="com.Karos.Dao.Demo.BookDemo"></bean>
<bean id="booktest" class="com.Karos.Dao.Demo.BookTest"></bean>
<bean id="bookservice" class="com.Karos.Services.Demo.BookServiceDemo">
<!--
    name为哪一个具体属性
    ref为这个属性的bean
-->
    <property name="book" ref="booktest"></property>
</bean>

```

```

public class App2 {
    public static void main(String[] args) {
        //获取ioc容器
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("appilcationContext.xml");
        BookService A = (BookService) ctx.getBean("bookservice");
        A.save();
    }
}

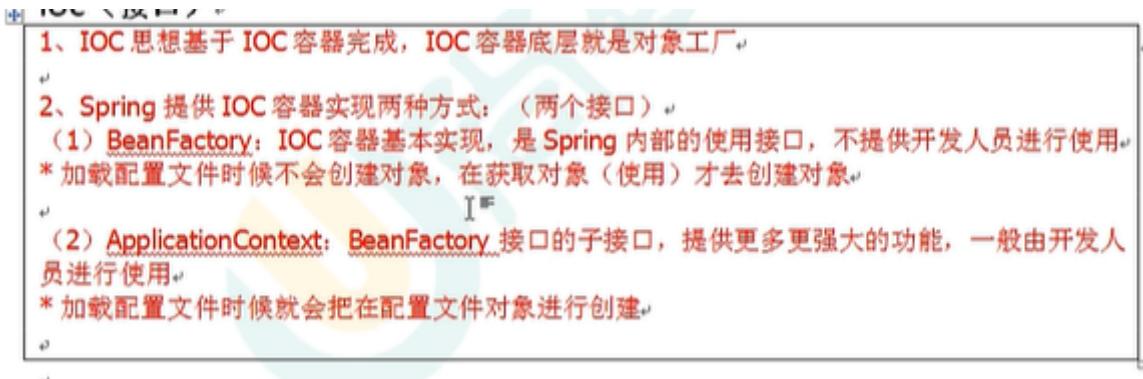
```

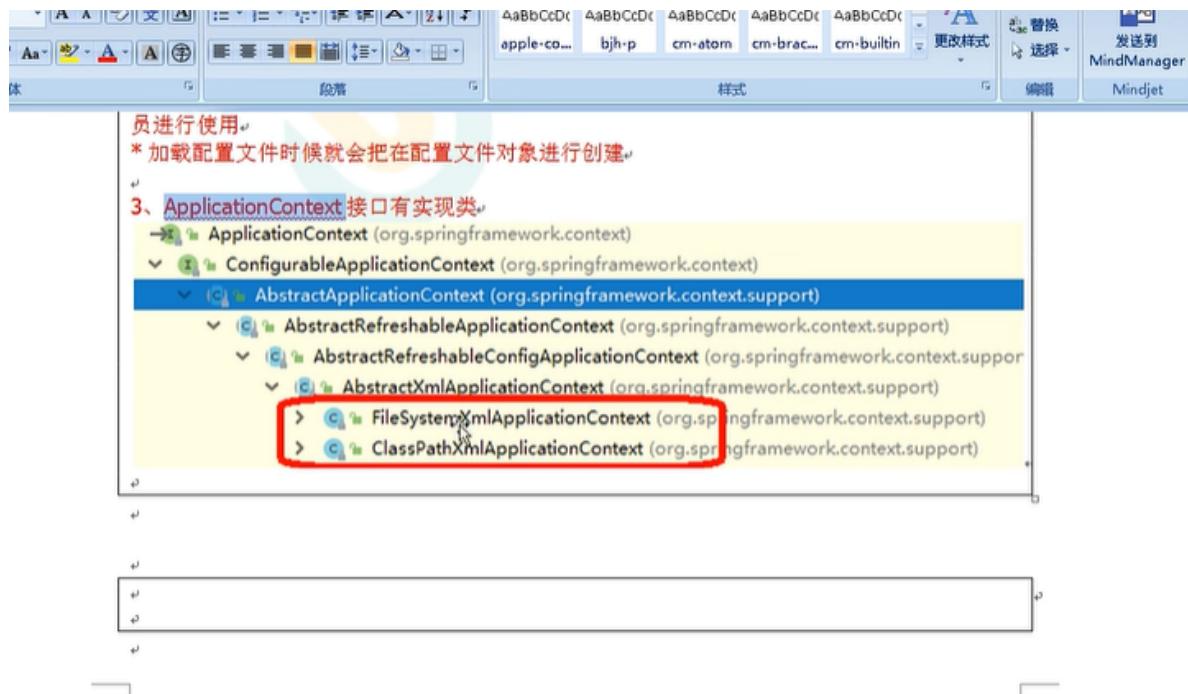
```
public class BookServiceDemo implements BookService {  
    BookDao book ;  
    public void save() {  
        System.out.println("业务层打印");  
        book.save();  
    }  
  
    public void setBook(BookDao book) {  
        this.book = book;  
    }  
}
```

```
public class BookDemo implements BookDao {  
  
    public void save() {  
        System.out.println("数据层打印");  
    }  
}  
  
class BookTest implements BookDao{  
  
    public void save() {  
        System.out.println("数据打印测试");  
    }  
}
```

BeanFactory和AppicationContext的区别

前者为需要用时创建对象，后者为配置项读取时直接创建





Bean管理【尚硅谷】

1. Spring创建对象
2. Spring注入属性

Bean管理操作的两种实现方式

1. 基于XML配置文件方式实现
2. 基于注解方式实现

基于XML方式创建对象

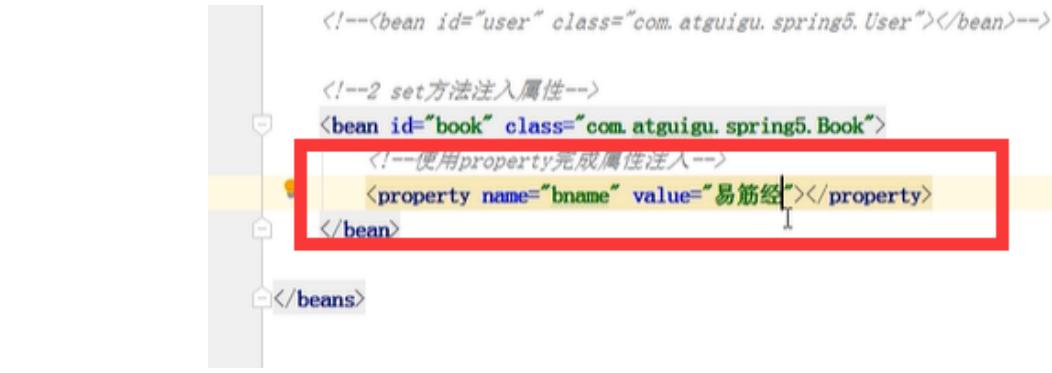
```
xsi:schemaLocation="http://www.springframework.org/schema/beans http  
<bean id="bookdao" class="com.Karos.Dao.Demo.BookDaoDemo"></bean>  
<bean id="booktest" class="com.Karos.Dao.Demo.BookTest"></bean>
```

1. 在Spring配置文件中，使用Bean标签，在标签里面添加对应属性，就可以实现对象创建
2. 在不然标签中有很多的属性，介绍常用的属性
 1. id 唯一标识
 2. class 类全路径
 3. name
3. 创建对象的时候，默认使用无参构造
4. p名称空间简化注入：简化基于xml配置方式
XML添加约束 xmlns:p="<http://www.springframework.org/p>"

基于XML方式注入属性

1. DI：依赖注入 注入属性
 1. 使用set方式进行注入（看前）

2. 通过有参构造注入



```
<bean id="dd" class="java.lang.String">
    <constructor-arg><value>123</value></constructor-arg>
</bean>
```

```
<bean id="dd" class="java.lang.String">
    <constructor-arg value="123"/>
</bean>
```

多个参数记得配置name或者像下面写

```
<constructor-arg type="int" value="10"/>
<constructor-arg type="java.lang.String" value="mysql"/>
```

或者直接以位置定位

```
<constructor-arg index="0" value="mysql"/>
<constructor-arg index="1" value="10"/>
```

字面量注入

1. null值

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

不要value="null", 修改的为字符串

2. 特殊符号

1. 在行内用<等标记语言替换
2. 在内部>]]>

注入属性-外部Bean

1. 创建两个类 service 和dao
2. 在service中调用dao的方法

注入属性-内部Bean 级联赋值

1. 一对多的关系：部门和员工
一个部门有多个员工，一个员工属于一个部门
部门是一，员工是多
2. 在实体类中来表示一对多的关系
将ref链接外部bean改为在内部写一个bean

级联赋值

1. 外部Bean
2. 设置geter, 通过获取的geter设置子属性对象的某个属性 value

注入集合属性

记得生成对应的set方法

注入数组类型属性

```
<bean id="stu" class="Stu">
    <property name="temp">
        <array>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
        </array>
    </property>
</bean>
```

注入list集合类型属性

```
<bean id="stu" class="Stu">
    <property name="temp">
        <list>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
            <value>"123"</value>
        </list>
    </property>
</bean>
```

注入Map集合属性

```
<bean id="stu" class="Stu">
    <property name="temp">
        <map>
            <entry key="java" value="123"></entry>
        </map>
    </property>
</bean>
```

注入set集合属性

```

<bean id="stu" class="Stu">
    <property name="temp">
        <set>
            <value>123</value>
        </set>
    </property>
</bean>

```

注入Properties集合属性

```

<bean id="stu" class="Stu">
    <property name="temp">
        <props>
            <prop key="key">value</prop>
            <prop key="key">value</prop>
            <prop key="key">value</prop>
            <prop key="key">value</prop>
        </props>
    </property>
</bean>

```

思考：如何注入vector容器属性？

答：vector继承于list，通常情况下我们会想到使用list强转，但是我们从IOC注入的list是原生的list，并不是vector的应用，所以只有设置Setter，传入List，将list的每个元素全部取出来再全部加进vector去

在集合中设置对象类型的值

将内部的value改为ref

```

<bean id="stu" class="Stu">
    <property name="temp">
        <list>
            <ref bean=""></value>
        </list>
    </property>
</bean>

```

试一试，仍然写value，在value里面写bean，直接写bean即可

把集合注入部分提取出来在外部定义

1. 在Spring配置文件中引入名称空间 util

```

xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"|

```

2. util:list

```

<util:list id="list">
    <bean id="dd" class="java.lang.String">
        <constructor-arg value="123"/></constructor-arg>
    </bean>
    <value>"123"</value>
    <value>"123"</value>
    <value>"123"</value>
    <value>"123"</value>
</util:list>
<bean id="stu" class="Stu">
    <property name="vc" ref="list">
    </property>
</bean>

```

Bean基础配置

类别	描述
名称	bean
类型	标签
所属	beans标签
功能	定义Spring核心容器管理的对象
格式	<beans> <bean/> <bean></bean> </beans>
属性列表	id : bean的id , 使用容器可以通过id值获取对应的bean , 在一个容器中id值唯一 class : bean的类型 , 即配置的bean的全路径类名
范例	<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/> <bean id="bookService" class="com.itheima.service.impl.BookServiceImpl"></bean>

bean + name 起别名

类别	描述
名称	name
类型	属性
所属	bean标签
功能	定义bean的别名 , 可定义多个 , 使用逗号(,)分号(;)空格()分隔
范例	<bean id="bookDao" name="dao bookDaoImpl" class="com.itheima.dao.impl.BookDaoImpl"/> <bean name="service,bookServiceImpl" class="com.itheima.service.impl.BookServiceImpl"/>

注意事项

获取bean无论是通过id还是name获取 , 如果无法获取到 , 将抛出异常NoSuchBeanDefinitionException
 NoSuchBeanDefinitionException: No bean named 'bookServiceImpl' available

bean的作用对象

中的scope属性默认位Single，需要自行配置

使用工厂创建对象

静态工厂（自写）

这里以单例模式为例子：

单例类：

```
class SingleDemo{  
    private static SingleDemo me=null;  
    private SingleDemo(){}
    public static SingleDemo getInstance(){  
        Object A = 1;  
        if (me==null) {  
            synchronized (A) {  
                if (me == null) me = new SingleDemo();  
            }  
        }  
        return me;  
    }
}
```

双重检索，不做解释

主类

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx=new
ClassPathXmlApplicationContext("ApplicationContext.xml");
        SingleDemo singleTest= (SingleDemo) ctx.getBean("singleDemo");
        singleTest.test();
    }
}
```

xml

```
<bean id="SingleDemo" class="SingleDemo" factory-method="getInstance"></bean>
```

实例工厂实例化

法一

```
class Factory{
    public Factory(){}
    public FactoryDemo getInstance(){
        System.out.println("qwe");
        return new FactoryDemo();
    }
}
class FactoryDemo{
    public FactoryDemo(){}
    public void test(){
```

```

        System.out.println("SingleDemo Created,Address is: ");
    }
}

public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx=new
ClassPathXmlApplicationContext("Applicationcontext.xml");
        FactoryDemo singleTest= (FactoryDemo) ctx.getBean("FactoryTest");
        singleTest.test();
    }
}

```

```

<!--创建一个Factory对象-->
<bean id="FactoryDemo" class="Factory"></bean>
<bean id="FactoryTest" factory-method="getInstance" factory-bean="FactoryDemo">
</bean>
<!--从Factory的对象中获取getInstance方法-->

```

如果工厂是自己，那么无参构造会被factory-method替代

法二

实现FactoryBean接口

```

public class BookDemoFactory implements FactoryBean<BookDemo> {
//    通过工厂获取对象
@Override
public BookDemo getObject() throws Exception {
    return new BookDemo();
}
//    对象所属类，创建Bean
@Override
public Class<?> getObjectType() {
    return BookDemo.class;
}
}

```

```

public class BookDemo {
    public void test(){
        System.out.println("test");
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationcontext.xml");
        BookDemo me= (BookDemo) ctx.getBean("BookDemo");
        me.test();
    }
}

```

```

<bean id="BookDemo" class="BookDemoFactory"></bean>

```

Bean的作用域【尚硅谷】

1. 在Spring中，设置创建Bean实例是单实例还是多实例
2. 在Spring里面，默认下是单实例对象
3. 如何设置单实例还是多实例 - scope
singleton 单实例 prototype 多实例
设置为单实例的时候，加载spring配置文件时候就会创建你单实例对象，多实例的时候是在调用getBean方法的时候创建多实例对象，设置为request session时会放到request session中去

Bean的生命周期

- 生命周期：从创建到消亡的完整过程
- bean生命周期: bean从创建到销毁的整体过程
- bean生命周期控制：在bean创建后到销毁前做一些事情

xml: init-method destroy-method

- 关闭容器方式：

- 手工关闭容器

ConfigurableApplicationContext接口close()操作

- 注册关闭钩子，在虚拟机退出前先关闭容器再退出虚拟机

ConfigurableApplicationContext接口registerShutdownHook()操作

destroy-method要看到必须关闭容器或者注册生命周期钩子

除了上述方法外，还可以实现InitializingBean和DisposableBean接口

生命周期钩子：

关闭钩子

registerShutdownHook()

*Bean的生命周期阶段

- 初始化容器
 1. 创建对象（内存分配）
 2. 执行构造方法
 3. 执行属性注入（set操作）
 4. 执行bean初始化方法
- 使用Bean
 1. 执行业务操作
- 关闭/销毁容器
 1. 执行Bean销毁方法

依赖自动装配

- IoC容器根据Bean所依赖的资源在容器中自动查找并注入到Bean中的过程称为自动装配 autowire
- 自动装配的方式 (在当前Bean中找) :
 - 按类型 (常用)

```
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>  
<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl" autowire="byType"/>
```
 - 按名称
id值与setter名相同
 - 按构造方法
 - 不启用自动装配

自动装配特征

- 自动装配用于引用类型依赖注入，不能对简单类型进行操作
- 使用按类型装配时 (byType) 必须保障容器中相同类型的Bean唯一，推荐使用
- 使用按名称装配时 (byName) 必须保障容器中具有指定名称的Bean，因变量名与配置耦合，不推荐使用
- 自动装配优先级低于setter注入与构造器注入，同时出现自动装配配置失效

数据源对象管理

- 通过Maven或者自己导入jar包
- 写bean，自行根据源代码判断采用setter注入还是构造注入
- getbean

加载properties文件配置信息

- 开启context名字空间: xmlns:context="....\context", xsi:schemaLocation="把beans改成context" 和开启util名字空间一样
- 使用context空间加载外部properties文件

```
<context:property-placeholder location="文件目录"></context:property-placeholder>
```

- 使用属性占位符\${}替换

```
属性名=值
```

placeholder 属性占位符 \${属性名}

第三方Bean管理

```
<!--直接配置连接池-->  
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  
    <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver"/>  
    <property name="url" value="jdbc:derby://localhost:1527/test;create=true"/>  
    <property name="username" value="sa"/>  
    <property name="password" value=""/>  
</bean>
```

容器

创建容器

1. 通过类路径加载 ClassPathXml...
2. 通过盘符加载 FileSystemXml...
3. 加载多个配置文件:

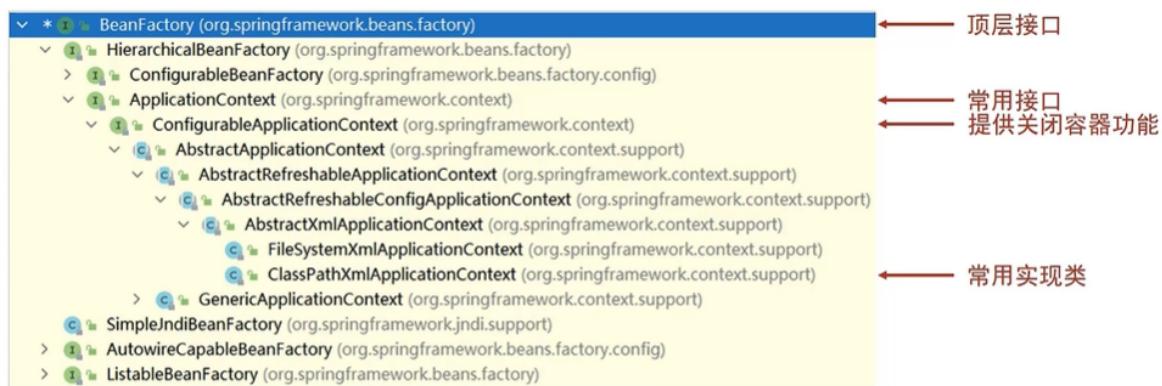
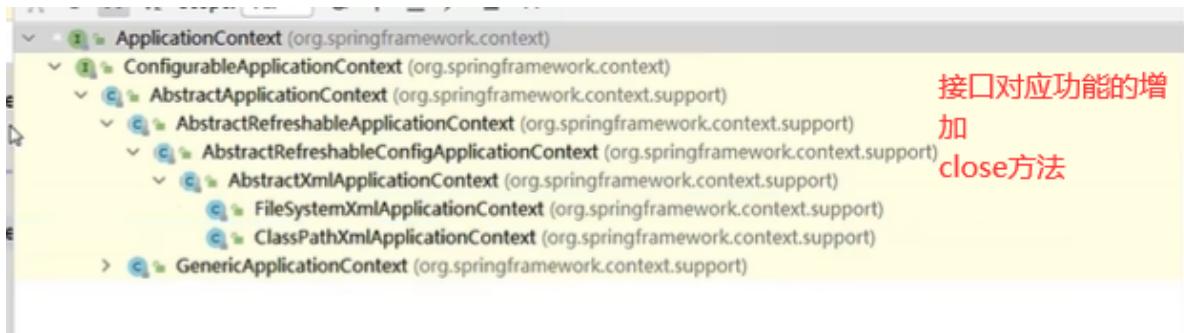
```
ApplicationContext ctx=new  
ClassPathXmlApplicationContext("bean1.xml","bean2.xml");
```

前面两种在前面讲过，最后一种了解即可

获取Bean

```
BookDao bookDao = (BookDao)ctx.getBean("bookDao");  
BookDao bookDao = ctx.getBean("bookDao",BookDao.class);  
BookDao bookDao = ctx.getBean(BookDao.class); //有点像自动类型装配，但是容器中该类型的  
bean必须唯一
```

容器类层次结构



BeanFactory

在Spring1.0中出现

BeanFactory (org.springframework.beans.factory)

- > HierarchicalBeanFactory (org.springframework.beans.factory)
- > ConfigurableBeanFactory (org.springframework.beans.factory.config)
- > ApplicationContext (org.springframework.context)
- > ConfigurableApplicationContext (org.springframework.context)
- > AbstractApplicationContext (org.springframework.context.support)
 - > AbstractRefreshableApplicationContext (org.springframework.context.support)
 - > AbstractRefreshableConfigApplicationContext (org.springframework.context.support)
 - > AbstractXmlApplicationContext (org.springframework.context.support)
 - > FileSystemXmlApplicationContext (org.springframework.context.support)
 - > ClassPathXmlApplicationContext (org.springframework.context.support)
 - > GenericApplicationContext (org.springframework.context.support)
 - > SimpleJndiBeanFactory (org.springframework.jndi.support)
 - > AutowireCapableBeanFactory (org.springframework.beans.factory.config)
 - > ListableBeanFactory (org.springframework.beans.factory)

```

spring_03_beans_instance D:\workspace\sp
spring_04_beans_lifecycle D:\workspace\sp
spring_05_di_set D:\workspace\sp\pri
spring_06_di_constructor D:\workspace\sp\pri
spring_07_di_autoware D:\workspace\sp\pri
spring_08_di_collection D:\workspace\sp\pri
spring_09_datasource D:\workspace\sp\pri
spring_10_container D:\workspace\sp\pri
  
```

src

main

java

 - com.itheima
 - dao
 - impl
 - BookDaoImpl
 - BookDao
 - App
 - AppForBeanFactory

resources

applicationContext.xml

AppForBeanFactory

```

import com.itheima.dao.BookDao;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class AppForBeanFactory {
    public static void main(String[] args) {
        Resource resources = new ClassPathResource("applicationContext.xml");
        BeanFactory bf = new XmlBeanFactory(resources);
        BookDao bookDao = bf.getBean(BookDao.class);
        bookDao.save();
    }
}
  
```

D:\soft\jdk1.8.0_172\bin\java.exe ...
book dao save ...

Process finished with exit code 0

核心区别

Bean加载的时机不一样, BeanFactory延迟加载, 要的时候才加载, 而AppXXX是直接加载

可以给AppXXX在配置中添加lazy-init="true"即可

Bean管理注解方式

注解开发定义BEAN

@Component 组件

@Component(BookDao)

添加标签, 扫描方式 递归

(67条消息) Java自定义注解及个性化扫描注解曾梦想仗剑走天涯PJ的博客-CSDN博客 java 扫描
注解

```

<context:component-scan base-package="com.***.***(类所在的位置)">
    <content:component-scan base-package="com.Karos.day3"></content:component-
    scan>
    <!--通常写厂商名-->
  
```

如果不给组件命名, 则必须在getBean中填写XXX.class, 注意: Scan是把这个类下的所有组件都给扫描, 然后会自动实例化

@Component的三个衍生注解

- @Controller: 用于表现层Bean定义
- @Service: 用于业务层Bean定义
- @Repository: 用于数据层定义

纯注解开发

- Spring3.0升级了纯注解开发模式，使用java替代配置文件，开启了Spring快速开发赛道

写配置类

写一个类（别写成其他的东西），再加上加上注解@Configuration

加上了注解，等同于加上了

```
<?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:content="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">
</beans>
```

再加上@ComponentScan,后面可以换上包名,代替

```
<content:component-scan base-package="com.Karos.day3"></content:component-scan>
```

加载配置类

```
ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
```

*注意事项

- @Configuration注解用于设置当前类为配置类
- @ComponentScan注解用于设定扫描路径,此注解只能添加一次,多个数据请用数组格式
- 读取Spring核心配置文件初始化容器对象切换为读取Java配置类初始化容器对象

```
//加载配置文件初始化容器
ApplicationContext ctx = new ClassPathXmlApplicationContext();
//加载配置类初始化容器
ApplicationContext ctx = new AnnotationConfigApplicationContext();
```

注解开发bean作用范围与生命周期管理

作用范围

- 控制变为非单例对象 @Scope

生命周期

- @PostConstruct 构造方法后调用的方法
- @PreDestroy 在彻底销毁前 在非单例无法使用

依赖注入

自动装配

注入引用类型

- @Autowired 注入，注解管理没有构造注入和Setter注入，使用暴力反射直接给，没有Setter入口也可以，默认按照类型装配，若有多个相同类型的，则使用@Qualifier
- @Qualifier()
- 使用@Autowired注解开启自动装配模式（按类型）

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }
    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
}
```

- 注意：自动装配基于反射设计创建对象并暴力反射对应属性为私有属性初始化数据，因此无需提供setter方法
- 注意：自动装配建议使用无参构造方法创建对象（默认），如果不提供对应构造方法，请提供唯一的构造方法

提供无参构造方法

- 使用@Qualifier注解开启指定名称装配bean

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    @Qualifier("bookDao")
    private BookDao bookDao;
}
```

- 注意：@Qualifier注解无法单独使用，必须配合@Autowired注解使用

注入简单类型

- @value(值) 值可来自于外部配置文件

@Resource【尚硅谷】

- @Resource 根据类型注入
- @Resource(name="name") 根据名称注入

Resource不是Spring的哦，在这里提一下

```
<!-- https://mvnrepository.com/artifact/javax.annotation/javax.annotation-api -->
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.2</version>
</dependency>
```

导入外部配置文件

- @PropertySource() 后面在引号中使用\${}
- 多个文件仍然以数组传入，且不支持使用通配符

```
String name() default "";
```

Indicate the resource location(s) of the properties file to be loaded.

Both traditional and XML-based properties file formats are supported — for example, "classpath:/com/myco/app.properties" or "file:/path/to/file.xml".

Resource location wildcards (e.g. **/*.properties) are not permitted; each location must evaluate to exactly one .properties resource.

\${...} placeholders will be resolved against any/all property sources already registered with the Environment. See [above](#) for examples.

Each location will be added to the enclosing Environment as its own property source, and in the order declared.

```
String[] value();
```

关于外部配置文件有几点说明的：

- 如果配置项名称为username，那么读取的是当前系统的用户名,是系统环境变量
- 如果读取的配置项不存在，那么属性占位符将被作为普通字符串输出

第三方Bean管理

将ComponentScan去掉

由于不能把配置写在别人的源代码中,只有我们自己写

在SpringConfig中

1. 定义一个方法获得要管理的对象，回到原始的手写代码状态
2. 添加Bean,括号里面写名称,建议方法名也是名称

```
@Bean
public DataSource dataSource(){
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName();
    ds.setUrl();
    ...
    return ds;
}
```

好了，上面又是白学，你想一想，你调用的外部Bean多了岂不是要爆炸？

把上面的函数放在一个新的类里面，然后再Springconfig里面开启Scan，在新的类里，然后

- 法一

```
@Configuration
public class JdbcConfig {
    //1. 定义一个方法获得要管理的对象
    //2. 添加@Bean，表示当前方法的返回值是一个bean
    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}
```

- 给SpringConfig添加扫描,扫描配置文件所在的包

两个主配置类，不太建议这样使用

- 法二

- 使用@import导入需要的所有配置类,多个用数组

第三方Bean依赖注入

通常各个配置来自于外部文件

简单类型

@Value

```
public class JdbcConfig {
    @Value("com.mysql.jdbc.Driver")
    private String driver;
    @Value("jdbc:mysql://localhost:3306/spring_db")
    private String url;
    @Value("root")
    private String userName;
    @Value("root")
    private String password;
    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(userName);
        ds.setPassword(userName);
        return ds;
    }
}
```

引用类型

1. 在SpringConfig中开启扫描
2. 向bean定义方法中加入形参 [自动装配给形参]

- 引用类型依赖注入

```
@Bean  
public DataSource dataSource(BookService bookService){  
    System.out.println(bookService);  
    DruidDataSource ds = new DruidDataSource();  
    //属性设置  
    return ds;  
}
```

- 引用类型注入只需要为bean定义方法设置形参即可，容器会根据类型自动装配对象

[@Bean注解全解析 - 程序员cxuan - 博客园\(cnblogs.com\)](#)

XML和注解方式对比

功能	XML配置	注解
定义bean	bean标签 ● id属性 ● class属性	@Component ● @Controller ● @Service ● @Repository @ComponentScan
设置依赖注入	setter注入(set方法) ● 引用/简单 构造器注入(构造方法) ● 引用/简单 自动装配	@Autowired ● @Qualifier @Value
配置第三方bean	bean标签 静态工厂、实例工厂、FactoryBean	@Bean
作用范围	● scope属性	@Scope
生命周期	标准接口 ● init-method ● destroy-method	@PostConstructor @PreDestroy

开发中遇见的问题

singleton和prototype

小写，如果要手动执行单例模式，建议prototype，还有就是两个分别是在什么时候被IOC实例化

@Bean和@Component

组件可以用于自动装配，而@Bean不能，但是@Bean所标记的方法的形参可以被自动装配

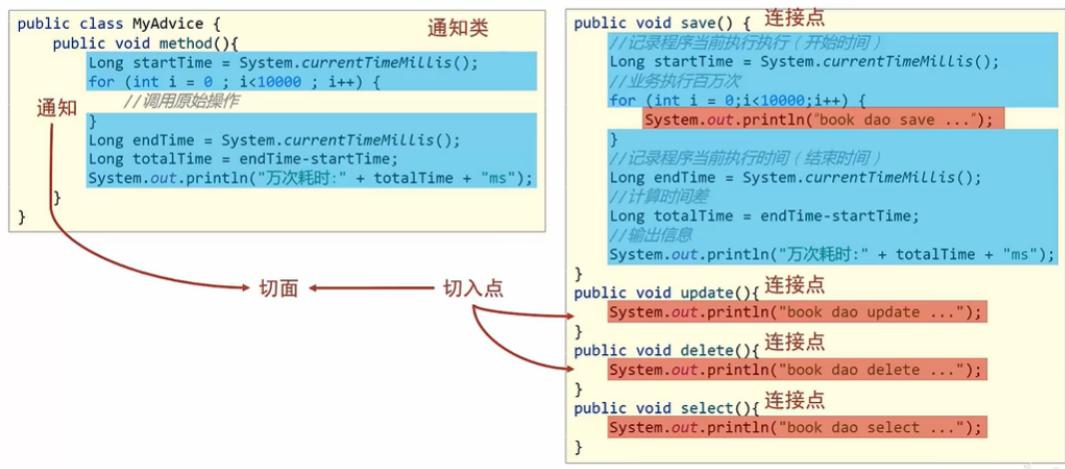
AOP

简介

AOP核心概念

- AOP(Aspect Oriented Programming)面向切面编程，一种编程范式，指导开发者如何组织程序结构
 - OOP(Object Oriented Programming)面向对象编程
- 作用：在不惊动原始设计的基础上为其进行功能增强

- Spring理念：无入侵式/无侵入式



- 连接点 (JoinPoint) : 执行程序过程中的任意位置、粒度为执行方法、抛出异常、设置变量等
 - 在SpringAOP中，可以理解为方法的执行
- 切入点 (Pointcut) : 匹配连接点的式子
 - 在SpringAOP中，一个切入点可以只描述一个具体方法，也可以匹配多个方法
 - 一个具体方法：com.Karos.dao包下的BookDao接口中的无形参返回值的 `save`方法
 - 匹配多个方法：所有的`save`方法，所有的`get`开头的方法，所有以`Dao`结尾的接口中的任意方法，所有带有一个参数的方法
- 通知 (Advice) : 在切入点处执行的操作，也就是共性功能
 - 在SpringAOP中，功能最终以方法的形式呈现
- 通知类：定义通知的类
- 切面 (Aspect) : 描述通知与切入点的对应关系

AOP主要作用

在不惊动原始设计的基础上为其进行功能增强

AOP入门案例

- 案例设定：测定接口执行效率
- 简化设定：在接口执行前输出当前系统时间
- 开发模式：XML or 注解
- 思路分析：
 - 导入坐标 `pom.xml`
 - 导入`Spring.aop`
 - 导入`org.aspectj`
 - 制作连接点方法（原始操作，`Dao`接口与实现类）
 - 制作共性功能（通知类与通知）
 - 定义切入点
 - `@Pointcut("excution(类型 方法)")`
 - 绑定切入点与通知关系（切面）
 - `@Before("切入点函数") 标记通知`

- 类上加@Aspect
- 在SpringConfig中设置注解@EnableAspectJAutoProxy

```
/* 切面类 */
@Component
@Aspect //说明这个类为切面类
public class Aop {
//    切入点定义依托一个不具有实际意义的方法进行,即无参数,无返回值,方法体无实际逻辑
    @Pointcut("execution(void com.Karos.Aop.Dao.BookDao.save())")
    public void pt(){}
    @Before("pt()")
    public void test(){
        System.out.println("测试");
    }
}
```

```
/* Spring配置类 */
@Configuration
@ComponentScan("com.Karos.Aop")
@EnableAspectJAutoProxy
public class SpringConfig {
```

```
/* Dao类 */
@Component
public class BookDao {

    public void save(){
        System.out.println("保存...");
    }
}
```

```
/* App类 */
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context=new
        AnnotationConfigApplicationContext(SpringConfig.class);
        BookDao A =context.getBean(BookDao.class);
        A.save();
    }
}
```

AOP工作流程

1. Spring容器启动

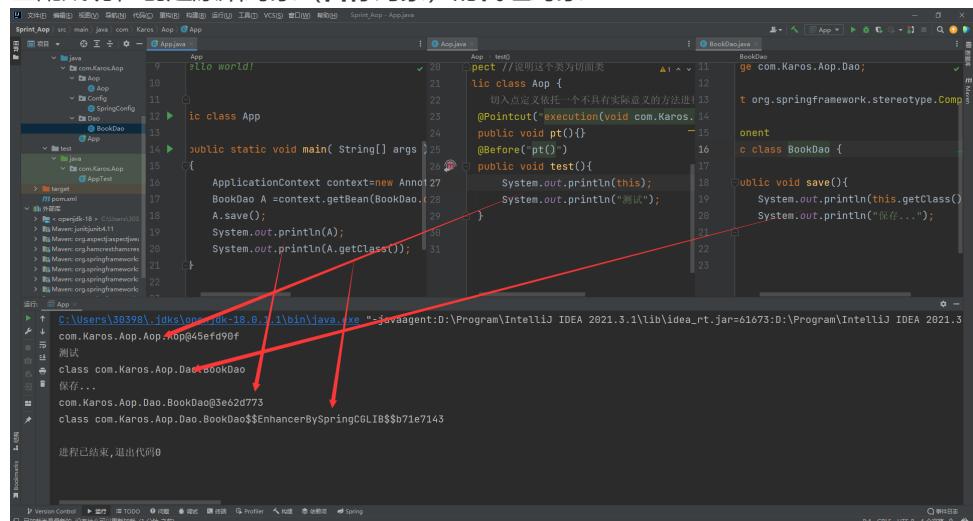
2. 读取所有切面配置中的切入点

```
@Component  
@Aspect  
  
public class MyAdvice {  
    @Pointcut("execution(void com.itheima.dao.BookDao.save())")  
    private void ptx(){  
  
        @Pointcut("execution(void com.itheima.dao.BookDao.update())")  
        private void pt(){  
  
            @Before("pt()")  
            public void method(){  
                System.out.println(System.currentTimeMillis());  
            }  
        }  
    }  
}
```

只读取配置了的切入点，即pt()

3. 初始化Bean，判断Bean对应的类中的方法是否匹配到任意切入点

- 匹配失败，创建对象
- 匹配成功，创建原始对象（目标对象）的代理对象

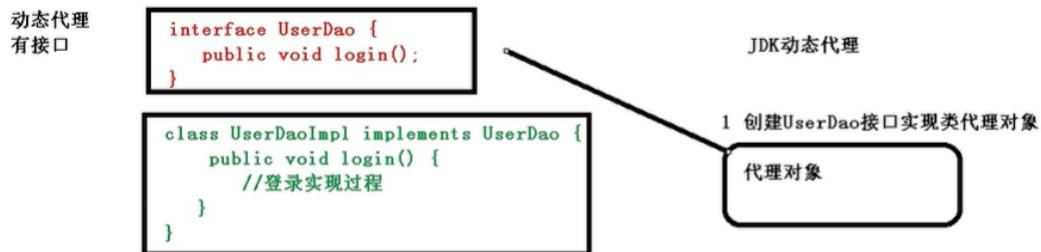


AOP底层原理（尚硅谷）

AOP底层使用动态代理

有两种情况动态代理

- 有接口情况：使用JDK动态代理（静态代理是个人都可以想到，动态代理详解[JDK动态代理-超详细源码分析 - 简书\(jianshu.com\)Java代理\(Proxy\)模式 - 简书\(jianshu.com\)](#)）



- 无接口情况：使用CGLIB动态代理（[CGLIB动态代理机制 - 简书\(jianshu.com\)](#)）
可以仔细看看上面的截图



JDK动态代理实现

使用JDK动态代理，使用Proxy类创建代理对象

```
/**  
 * 演示接口  
 */  
public interface Tester {  
    public void test();  
    void test1();  
}
```

```
/**  
 * 演示类  
 */  
public class Test implements Tester{  
    @Override  
    public void test() {  
        System.out.println("测试函数1");  
    }  
  
    @Override  
    public void test1() {  
        System.out.println("测试函数2");  
    }  
}
```

```
}
```

```
/**  
 * InvocationHandler接口的实现,这个接口主要就是对其他的方法进行引用,并增加功能  
 */  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
  
public class MyInvocationHandler implements InvocationHandler {  
    private Object object;  
  
    public MyInvocationHandler(Object object) {  
        this.object = object;  
    }  
  
    /**  
     *  
     * @param proxy 代理实例上调用方法  
     * @param method 方法实例对应接口的代理实例上调用方法。  
     * @param args 一个对象数组包含在方法调用中传递的参数的值在代理实例,或者 null接口方法不  
     * 包含任何参数。原始类型的参数裹着适当的原始包装器类的实例  
     * @return  
     * @throws Throwable  
     *  
     * 第一个参数是指定代理类的类加载器（我们传入当前测试类的类加载器）  
     * 第二个参数是代理类需要实现的接口（我们传入被代理类实现的接口，这样生成的代理类和被代理  
     * 类就实现了相同的接口）  
     * 第三个参数是invocation handler, 用来处理方法的调用。这里传入我们自己实现的handler  
     */  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws  
    Throwable {  
        System.out.println("Before");  
        //第一个参数是实现的方法,第二个参数是代理类的对象  
        Object invoke = method.invoke(object, args);  
        System.out.println("After");  
        return invoke;  
    }  
}
```

```
/**  
 * APP类  
 */  
import java.lang.reflect.Proxy;  
  
public class App {  
    public static void main(String[] args) {  
        Test A=new Test();  
        //咱们把需要代理的A传进去,内部的invoke函数会获取A的所有方法,并对所有方法进行增强作用  
        MyInvocationHandler B = new MyInvocationHandler(A);  
        //利用Proxy的静态方法,创建一个同样的对Tester接口的实现  
        Tester pro = (Tester) Proxy.newProxyInstance(App.class.getClassLoader(),  
        Test.class.getInterfaces(), B);  
        pro.test();  
        pro.test1();
```

```
    }  
}
```

结果：

Before

测试类

After

Before

测试2

After

AOP操作（尚硅谷）

Spring框架一半都是基于AspectJ实现AOP操作

- AspectJ不是Spring的组成部分，独立AOP框架

XML

context开启 aop开启

(4) 在 spring 配置文件中开启生成代理对象。

```
<!-- 开启 Aspect 生成代理对象 -->  
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>  
  
<!--配置aop增强-->  
<aop:config>  
    <!--切入点-->  
    <aop:pointcut id="p" expression="execution(* com.atguigu.spring5.aopxml.Book.buy(..))"/>  
  
    <!--配置切面-->  
    <aop:aspect ref="bookProxy">  
        <!--增强作用在具体的方法上-->  
        <aop:before method="before" pointcut-ref="p"/>  
    </aop:aspect>  
  
</aop:config>
```

AOP切入点表达式

- 切入点：要进行增强的方法
- 切入点表达式：要进行增强的方法的描述方式

描述方式一：以接口方法描述

描述方式二：以类方法描述

- 切入点表达式标准格式：动作关键字 (访问修饰符 返回值 包名.类/接口名.方法名(参数) 异常名)
- 可以用通配符描述切入点，快速描述
 - *：单个独立的任意符号，可以独立出现，也可以作为前缀或后缀的匹配符出现

```
@execution(public * com.Karos.*.UserService.find*(*))
```

匹配com.Karos包下的任意包中的UserService类或接口中所有find开头的带有一个参数的方法

- ... : 多个连续的任意符号，可以独立出现，常用于简化包名与参数的书写

```
@excution(public User com..UserService.findById(..))
```

匹配com包下的任意包中的UserService类或接口中所有名称为findById的方法

- +: 专用于匹配子类类型

```
execution(* *..*Service+.*(..))
```

书写技巧

- 所有代码按照标准规范开发，否则以下技巧全部失效
- 描述切入点通常描述接口，而不描述实现类
- 访问控制修饰符针对接口开发均采用public描述（可省略访问控制修饰符描述）
- 返回值类型对于增删改类使用精准类型加速匹配，对于查询类使用*通配快速描述
- 包名书写尽量不使用..匹配，效率过低，常用*做单个包描述匹配，或精准匹配
- 接口名/类名书写名称与模块相关的采用*匹配，例如UserService书写成*Service，绑定业务层接口名
- 方法名书写以动词进行精准匹配，名词采用*匹配，例如getById书写成getBy*,selectAll书写成selectAll
- 参数规则较为复杂，根据业务方法灵活调整
- 通常不使用异常作为匹配规则

AOP通知类型

AOP通知描述了抽取的共性功能，根据共性功能抽取的位置不同，最终运行代码要将其加入到合理的位置

共分5种类型

前置通知

@Before

后置通知

@After, 类比finally

环绕通知（重点）

@Around

- 表示对原始操作的调用ProceedingJoinPoint pjp
- pjp.proceed() 注意类型

总结：

- 环绕通知必须依赖形参ProceedingJoinPoint才能实现对原始方法的调用，进而实现原始方法调用前后同时添加通知
- 通知中如果未使用ProceedingJoinPoint对原始方法进行调用将跳过原始方法的执行

- 对原始方法的调用可以不接受返回值，通知方法设置成void即可，如果接受返回值，必须设定为Object类型
- 原始方法的返回值如果是void类型，通知方法的返回值类型可以设置成void，也可以设置成Object
- 由于无法预知原始方法运行后是否为抛出异常，因而环绕通知方法必须抛出对象Throwable

返回后通知（了解）

@AfterReturning 在没有抛异常正常结束的时候运行，即遇见return后执行

抛出异常后通知（了解）

@AfterThrowing

案例：业务层接口执行效率

需求：任意业务层接口执行均可显示其执行效率（执行时长）

AOP通知获取数据

获取参数

- JoinPoint：适用于前置、后置、返回、抛出异常后通知
 - jp.getArgs
- ProceedJoinPoint：适用于环绕通知

- JoinPoint对象描述了连接点方法的运行状态，可以获取到原始方法的调用参数

```
@Before("pt()")
public void before(JoinPoint jp) {
    Object[] args = jp.getArgs();
    System.out.println(Arrays.toString(args));
}
```

- ProceedJoinPoint是JoinPoint的子类

```
@Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object[] args = pjp.getArgs();
    System.out.println(Arrays.toString(args));
    Object ret = pjp.proceed();
    return ret;
}
```

获取返回值

- 返回后通知
 - 加形参 Object ret
 - 注解设置returning值
- 环绕通知

获取异常

- 抛出异常后通知
 - 传参

```

@AfterReturning(value = "getpj()", returning = "ret")
public void Deal1(JoinPoint jp, Object ret){
    System.out.println(ret);
}

```

- 配置throwing
- 环绕通知
 - 第try...catch

注意

- JoinPoint存在的话，必须是第一个参数

案例：百度网盘密码数据兼容处理

- 简单实现密码处理即可

实现

- Spring配置

```

package com.Karos.Pandownload.Config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("com.Karos")
@EnableAspectJAutoProxy
public class SpringConfig {
}

```

- Dao层

```

package com.Karos.Pandownload.Dao;

import org.springframework.stereotype.Component;

@Component
public class PanDownload {
    private String Password;

    public PanDownload() {
    }

    public String getPassword() {
        return Password;
    }

    public void setPassword(String password) {
        Password = password;
    }

    public PanDownload(String password) {
        Password = password;
    }
}

```

```
    }  
}
```

- AOP控制

```
package com.Karos.Pandownload.AOP;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.Around;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
import org.springframework.stereotype.Component;  
  
@Component  
@Aspect  
public class AOP {  
    //设置切入点  
    @Pointcut("execution(void com...PanDownload.set*(..))")  
    public void pj(){  
  
        @Around("com.Karos.Pandownload.AOP.AOP.pj()")  
        public Object Deal(ProceedingJoinPoint pjp) throws Throwable {  
            Object[] args = pjp.getArgs();  
            if  
(args[0].getClass().equals(String.class))args[0]=args[0].toString().trim()  
);  
            Object proceed = pjp.proceed(args);  
            return proceed;  
        }  
    }  
}
```

开发中遇见的问题

1. 导包

- AspectJ 注意版本号

```
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjweaver</artifactId>  
    <version>1.9.9.1</version>  
</dependency>
```

2. ProceedingJoinPoint和JoinPoint使用点及其区别与联系

3. 如果要对切入点的参数进行修改，只能使用环绕通知

AOP——事务

事务简介

- 事务作用：在数据层保障一系列的数据库操作同成功同失败
- Spring事务作用：在**数据层或业务层**保障一系列的数据库操作同成功同失败
- 和MySQL一样的
ACID：**原子性 一致性 隔离性 持久性**

步骤：

1. 在接口上添加注解@Transactional (若写在类或者接口上，则其中的方法全部开事务)
2. 定义事务管理器 Bean

```
@Bean  
public PlatformTransactionManager transactionManager(){  
    DataSourceTransactionManager ptm =new DataSourceTranscationManager();  
    ptm.setDataSource(); //自行传参，管理啥就传啥  
    return ptm;  
}
```

3. 开启注解式事务驱动

```
@EnableTransactionManagement
```

事务角色

事务管理员

发起事务方，在Spring中通常指代业务层开启事务的方法

事务协调员

加入事务方，在Spring中通常只带数据层方法，也可以是业务层方法

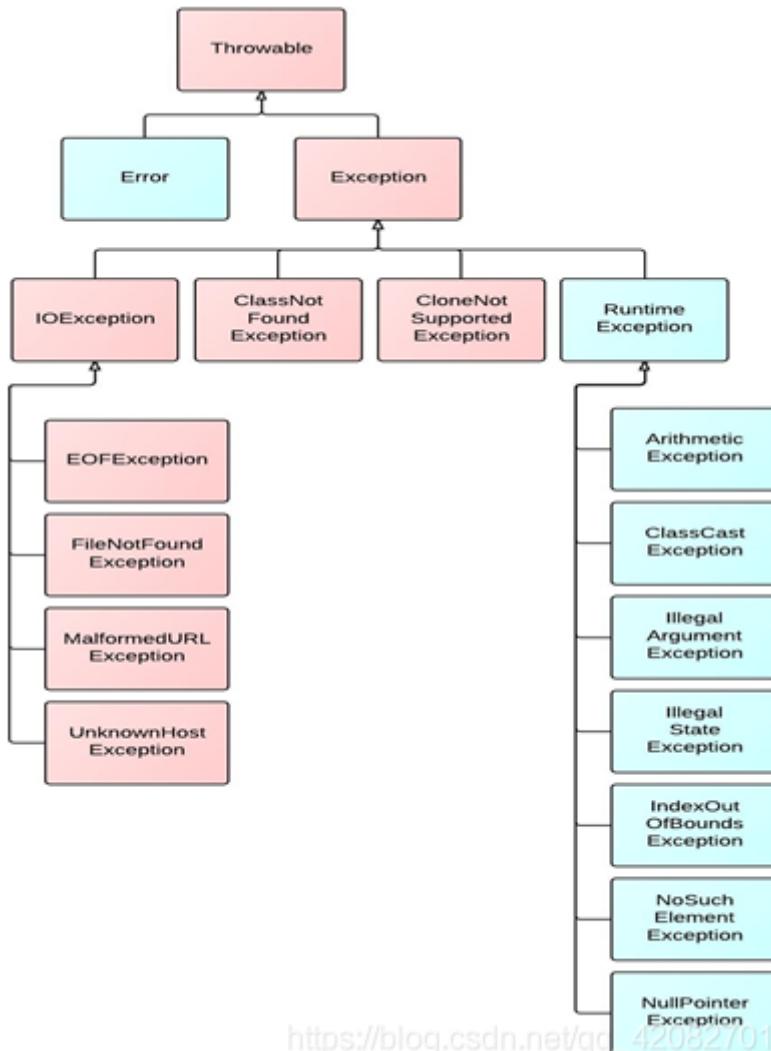
事务属性

事务配置

属性	作用	示例
readOnly	设置是否为只读事务	readOnly=true
timeout	设置事务超时时间	timeout=-1 (永不超时)
rollbackFor	设置事务回滚异常 (Class)	rollbackFor={NullPointerException.class} //遇到什么异常就回滚
rollbackForClassName	设置事务回滚异常 (String)	同上给的是字符串
noRollbackFor	设置事务不回滚异常 (Class)	ollbackFor={NullPointerException.class}
noRollbackForClassName	设置事务不回滚异常 (String)	同上给的是字符串
propagation	设置事务传播行为

异常回滚：

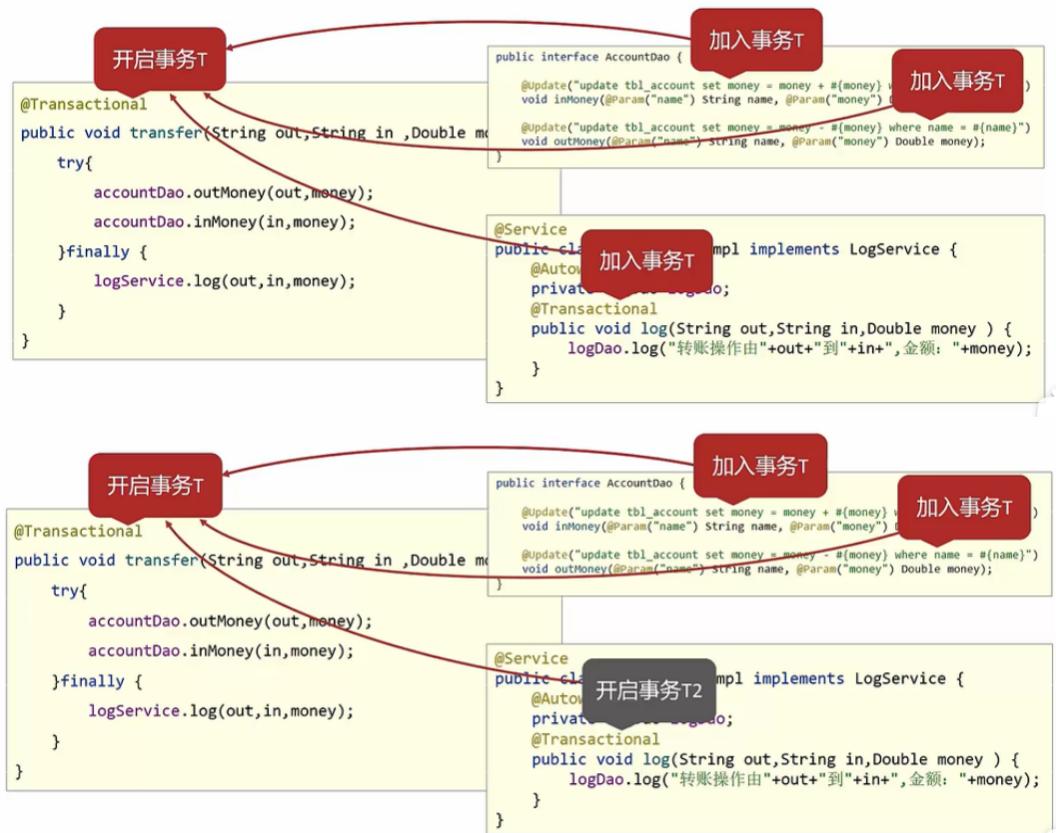
- Error异常
- 运行时异常



<https://blog.csdn.net/qd42082701>

事务传播行为

- 事务传播行为：事务协调员对事务管理员所携带事务的处理态度



开启新事务：@Transactiona(propagation = Propagation.REQUIRES_NEW)

事务传播行为设置

传播属性	事务管理员	事务协调员
REQUIRED (默认)	开启T	加入T
	无	新建T2
REQUIREDS_NEW	开启T	开启T2
	无	新建T2
SUPPORTS	开启T	加入T
	无	无
NOT_SUPPORTS	开启T	无
	无	无
MANDATORY	开启T	加入T
	无	ERROR
NEVER	开启T	ERROR
	无	无
NESTED	设置savePoint，一旦十五回滚，事务将回滚到SavePoint处，交友客户相应提交/回滚	

支不支持必就看有没有爹，命令的话就必须要有爹，没有就报错，不支持就真的不支持

XML

配置

```
1、在 spring 配置文件配置事务管理器
<!--创建事务管理器-->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源-->
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

2、在 spring 配置文件，开启事务注解

(1) 在 spring 配置文件引入名称空间

```
<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:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd"
```

(2) 开启事务注解

```
<!--开启事务注解-->
<tx:annotation-driven transaction-
manager="transactionManager"/></tx:annotation-driven>
```

XML声明式

1. 创建事务管理器

```
<!--1 创建事务管理器-->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源-->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

2. 配置通知

```
<!--2 配置通知-->
<tx:advice id="txadvice">
    <!--配置事务参数-->
    <tx:attributes>
        <!--指定哪种规则的方法上面添加事务-->
        <tx:method name="accountMoney" propagation="REQUIRED"/>
        <!--<tx:method name="account*"/>-->
    </tx:attributes>
</tx:advice>
```

3. 配置切入点和切面

```
<!--3 配置切入点和切面-->
<aop:config>
    <!--配置切入点-->
    <aop:pointcut id="pt" expression="execution(* com.atguigu.spring5.service.UserService.*(..))"/>
    <!--配置切面-->
    <aop:advisor advice-ref="txadvice" pointcut-ref="pt"/>
</aop:config>
```

Spring5 框架新功能

整个框架的代码基于java8

- 通过使用泛型等特性提高可读性
- 对java8提高直接的代码支撑
- 运行时兼容JDK9
- Java EE 7 API需要Spring相关的模块支持
- 运行时兼容Java EE8 API
- 取消的包,类和方法
- 包 beans.factory.access
- 包 dbc.support.nativejdbc
- 从spring-aspects 模块移除了包mock.staticmock,不在提AnnotationDrivenStaticEntityMockingControl支持
- 许多不建议使用的类和方法在代码库中删除

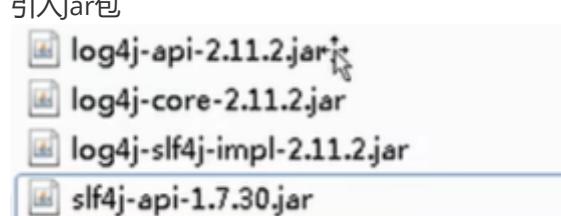
核心特性

JDK8的增强:

- 访问Resource时提供getFile或和isFile防御式抽象
- 有效的方法参数访问基于java 8反射增强
- 在Spring核心接口中增加了声明default方法的支持一贯使用JDK7 Charset和StandardCharsets的增强
- 兼容JDK9
- Spring 5.0框架自带了通用的日志封装
- 持续实例化via构造函数(修改了异常处理)
- Spring 5.0框架自带了通用的日志封装
- spring-jcl替代了通用的日志, 仍然支持可重写
- 自动检测log4j 2.x, SLF4J, JUL (java.util.Logging) 而不是其他的支撑
- 访问Resource时提供getFile或和isFile防御式抽象
- 基于NIO的ReadableChannel也提供了这个新特性

整合日志框架

1. Spring5.0框架自带了通用的日志封装, 但也可以整合其他日志框架, Spring5以及移除了Log4jConfigListener, 官方建议使用Log4j2
2. Spring5框架整合Log4j2
 - 引入Jar包



○ 创建Log4j2.xml配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<!—日志级别以及优先级排序：OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->
<!—Configuration后面的status用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace时，可以看到log4j2内部各种详细输出-->
<configuration status="INFO">
    <!--定义所有的appender-->
    <appenders>
        <!--输出日志信息到控制台-->
        <console name="Console" target="SYSTEM_OUT">
            <!—控制日志输出的格式-->
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-level %logger{36} - %msg%n"/>
        </console>
    </appenders>
    <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <!--root：用于指定项目的根日志，如果没有单独指定logger，则会使用root作为默认的日志输出-->
    <loggers>
        <root level="info">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>

```

○ 运行

```

D:\Program Files\Java\jdk1.8.0_181\bin\java.exe ...
2020-05-19 10:18:09.650 [main] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-1} init

```

Tests failed: 1 of 1 test – 2s 954ms

```

2020-05-19 10:18:49,383 main DEBUG createLoggers(=root)
2020-05-19 10:18:49,384 main DEBUG Configuration XmlConfiguration[location=E:\work\spring5_txdemo1\out\production\spring5_tx...
2020-05-19 10:18:49,384 main DEBUG Starting configuration XmlConfiguration[location=E:\work\spring5_txdemo1\out\production\...
2020-05-19 10:18:49,385 main DEBUG Started configuration XmlConfiguration[location=E:\work\spring5_txdemo1\out\production\...
2020-05-19 10:18:49,388 main DEBUG Shutting down OutputStreamManager SYSTEM_OUT.false.false=false
2020-05-19 10:18:49,389 main DEBUG Shut down OutputStreamManager SYSTEM_OUT.false.false=false, all resources released: true
2020-05-19 10:18:49,390 main DEBUG Appender DefaultConsole-1 stopped with status true
2020-05-19 10:18:49,390 main DEBUG Stopped org.apache.logging.log4j.core.config.DefaultConfiguration@6574a52c OK
2020-05-19 10:18:49,442 main DEBUG Registering MBean org.apache.logging.log4j:type=18b4aac2
2020-05-19 10:18:49,446 main DEBUG Registering MBean org.apache.logging.log4j:type=18b4aac2, component=StatusLogger
2020-05-19 10:18:49,448 main DEBUG Registering MBean org.apache.logging.log4j:type=18b4aac2, component=ContextSelector

```

○ 手动输出日志

```

private static final Logger log = LoggerFactory.getLogger(UserLog.class);
public static void main(String[] args) {
    log.info("hello log4j2");
    log.warn("hello log4j2");
}

```

UserLog > main()

```

2020-05-19 10:27:44,182 main DEBUG Shutdown hook enabled. Registering a new one.
2020-05-19 10:27:44,183 main DEBUG LoggerContext[name=18b4aac2, org.apache.logging.log4j.core.LoggerContext@6025elb6] started OK.
2020-05-19 10:27:44,196 [main] INFO com.atguigu.spring5.test.UserLog - hello log4j2
2020-05-19 10:27:44,200 [main] WARN com.atguigu.spring5.test.UserLog - hello log4j2
2020-05-19 10:27:44,214 pool-1-thread-1 DEBUG Stopping LoggerContext[name=18b4aac2, org.apache.logging.log4j.core.LoggerContext@6025el...
2020-05-19 10:27:44,215 pool-1-thread-1 DEBUG Shutting down OutputStreamManager SYSTEM_OUT false false

```

关于本笔记

作者博客：[Karos \(www.wzl1.top\)](http://www.wzl1.top)

笔记基于 黑马程序员视频课 和 尚硅谷视频课 而作

[黑马程序员2022最新SSM框架教程Spring+SpringMVC+Maven高级+SpringBoot+MyBatisPlus企业实用开发技术哔哩哔哩](#) [bilibili](#)

[【尚硅谷】Spring框架视频教程 \(spring5超详细源码级讲解\) 哔哩哔哩bilibili](#)

用时1周，Spring5新功能持续更新

下面是个人对于Spring的理解：

关于IOC的理解以及遇见的一些问题的解答

- IOC获取每一个Component的Class对象放入容器中，如果需要，那就造一个（单例是在IOC容器被实例化的过程中创建的）

- @Component和@Bean都是代表的Bean，但是有区别
 - @Component可以理解为是XML中的配置
 - @Bean 可以说是用来获取的，有一点像XML-bean中id的作用
 - 所以@Component可以被自动装配，而@Bean不能
- DI其实就是依赖关系嘛，对象的属性和对象属于依赖关系
- BeanFactory

这个没什么好说的，早期Spring就是靠的这个来进行IOC控制，和ApplicationContext的区别就在于实例化Bean的时期不同，然后他们是继承关系
- FactoryBean

一个接口，里面有三个方法，一个获取Bean，一个获取bean的类型，还有一个获取bean的作用范围
- 实现依赖注入的方式
 - Setter方法
 - Constructor方法
- 自动装配

注解开发中的自动装配不需要Setter和构造方法，而是直接通过底层暴力反射获取Bean
- spring 配置文件中 节点的 autowire 参数可以控制 bean 自动装配的方式
 - no - 不自动装配，需要使用 节点或参数（默认，直接找个Bean来填）
 - byName - 根据名称进行装配
 - byType - 根据类型进行装配
 - constructor - 根据构造函数进行装配
- 自动装配能用于静态成员吗？

不可以，静态成员属于类的成员，会在类加载的时候进行载入，而Bean是在IOC被实例化的时期实例化的，所以不行，但是可以通过getBean方法实现
- 既然可以通过设置Scope来设置单例模式，那线程安全吗？
 - 不安全
 - 方案一（之前自己写一个数据库管理的时候遇到过）：可以手写，但是要想一下，如果我们创建了一个双检锁的单例模式，但是Spring默认使用的也是单例模式，然后默认情况下会在IOC初始化的时候实例化Bean，而通过反射强行获得（假设你是自动装配）的Bean，和我们自行写的一个静态成员并不是同一个，在这个地方违背了单例模式的思想，有两个对象，所以如果需要自己写单例模式的话，记得设置 prototype

The screenshot shows three panels illustrating various ways to handle dependencies in Java code:

- Top Panel:** Shows a class named `Tool` annotated with `@Scope("singleton")`. It contains a static field `me` and a static method `getInstance()` that prints "工具类注入完毕".
- Middle Panel:** Shows a `Tool` class annotated with `@Scope("prototype")`. It contains a static field `me` and a static method `getInstance()`. The code uses `ApplicationContextAware` and `getBean` to manage the service.
- Bottom Panel:** Shows a `Controller` class annotated with `@Scope("prototype")`. It contains a static field `me` and a static method `getInstance()`. It uses `@Lookup` to inject the service.

Each panel includes a terminal window at the bottom showing the output of running the application, confirming successful dependency injection.

- 解决方案二：使用jdk提供的`ThreadLocal`来包装有状态的成员变量，也即是线程封闭吧，永远在单线程内使用该对象[ThreadLocal](#)，一篇文章就够了 - 知乎
(zhihu.com)

- 基于上一个问题引发一个思考：如果我Controller层 (C) 为singleton，Service层 (S) 用的是prototype，然后再C中有一个S的Bean，可能会发生什么问题？（这个问题是再往上看到的）

- 发生的问题：因为C的作用范围是singleton，所以再IOC中，C是单例的，也就是再IOC容器初始化的时候会被创建，这样的话，C中的S就会单例化
- 解决方法（来自CSDN）：
 - 两种办法

- 第一个就是使用依赖查找那种方式，controller实现`ApplicationContextAware`接口，保留一个`ApplicationContext`对象，然后每次调用service的时候，就用`getBean`方法重新请求一个新的service。[ApplicationContextAware使用理解 - 简书\(jianshu.com\)](#)

- 第二种就是controller内将service属性声明变为抽象方法，然后添加`@Lookup`注解。

由此猜测`@Lookup`有些类似自动装配，获取Bean并将Bean装配给方法。[\(68条消息\) Spring中的Lookup\(方法注入\)](#) [我知道你是高手的博客-CSDN博客](#)

- 除了@Autowired外还有没有其他注解?
还有@Resource注解和@Inject注解，均为javax包下的注解
@Resource默认按照name去匹配，匹配不到再按照类型去匹配。
@Injec主要是为了JSR规范而实现的注入规则，主要是按照类型类匹配，如果要想按照name匹配，需要
添加@Named注解来指定名称。[\(68条消息\) Java的JSR规范幽夜落雨的博客-CSDN博客jsr规范](#)

关于AOP的理解以及遇见的问题

- AOP底层使用的代理
接口 - JDK动态代理[JDK动态代理-超详细源码分析 - 简书\(jianshu.com\)Java代理 \(Proxy\) 模式 - 简书\(jianshu.com\)](#)
类 -CGLIB动态代理[CGLIB动态代理机制 - 简书\(jianshu.com\)](#)
- 切面的构成
切入点+通知
- JointPoint和PointCut的区别
JointPoint可以是任何可以被增强的方法，而PointCut是实际被增强的方法
- 引介(Introduction)
引介(Introduction) 是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过AOP的引介功能，我们可以动态地为该业务类添加接口的实例逻辑，让业务类成为这个接口的实现类。**引介让一个切面可以声明被通知的对象实现了任何他们没有真正实现的额外接口**，而且为这些对象提供接口的实现。使用 `@DeclareParents` 注解来生成一个引介。[\(68条消息\) Spring Aop \(六\) ——@DeclareParents介绍 elim168的博客-CSDN博客](#)