# Spring

# Day1

## spring runtime

1. Container(容器)：Spring当作一个大容器，能取到Spring中所有内容
2. Test：单元测试
3. Core container：核心容器，Spring启动的最基本条件

3.1、 Beans：Spring 负责创建类对象并管理对象

3.2、Core：核心类

3.3、Context:上下文参数。获取外部资源或管理注解

3.4、SpEl：expression.jar

4、AOP：实现aop功能需要的依赖

5、Aspects：切面AOP依赖的包

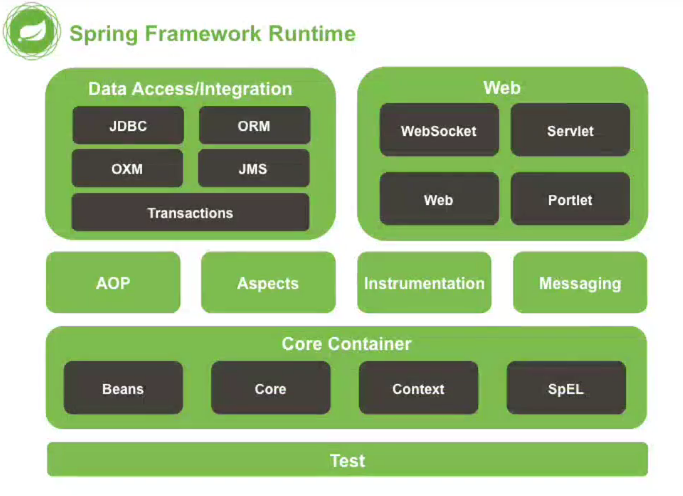
6、Date Access/Integration:Spring封装数据访问层相关内容

6.1、JDBC：Spring对JDBC封装后的代码

6.2、ORM：封装持久层的代码 如：Hibernate

6.3、Transactions:对应spring-tx.jr，声明式事务使用

7、WEB：需要Spring完成web相关功能时需要

7.1：例如：由tomcat加载spring配置文件时新需要 spring-web.jar

## IoC: 控制反转

1. 是什么？

IOC完成的事情原先由程序员主动通过new实例化对象 的事情，转交给Spring负责。

**控制**指的是**控制类的对象**，**反转**指的是**转交给Spring负责**

1. 作用：**解耦：**解除了对象管理和程序员之间的耦合

## Spring环境搭建

1. 导入jar：4个核心包，一个日志包
2. 在src下新建 applicationContext.xml

2.1、文件名称和路径自定义

2.2、Spring容器ApplicationContext，applicationContext.xml配置的信息最终储存到了ApplicationContext容器中

1. Spring配置文件是基于schema

3.1、schema文件的扩展名是： .xsd，需要把schema理解成DTD的升级版，他比DTD具备更好的扩展性

|  |
| --- |
| <?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"*>  <!-- **id表示取到对象标识**  **class 创建哪个类的实例**  -->  <bean id=*"peo"* class=*"com.jun.pojo.People"*></bean>  </beans> |

测试方法：getBean(“<bean>标签id值“，返回值类型),如果没有第二个参数默认是Object

getBeanDefinitionNames()：spring容器中目前所管理的全部对象

|  |
| --- |
| // People people = new People();  //通过IOC创建  ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");  People people = ac.getBean("peo", People.**class**);  System.***out***.println(people); |

## 四、spring创建对象的三种方式

1. 通过构造方法创建
   * 1. 无参构造创建：默认
     2. 有参构造创建：需要明确配置
        + 1. 需要在类中提供有参构造
          2. 在applicationContext.xml中设置调用那个构造方法创建对象：若设定的条件匹配多个构造方法，则执行最后一个

**index:参数的索引，从0开始**

**name：参数名**

**type：参数类型，和类中一致**

<bean id=*"peo"* class=*"com.jun.pojo.People"*>

<constructor-arg index=*"0"* name=*"name"* type=*"String"* value=*"zs"*></constructor-arg>

<constructor-arg index=*"1"* name=*"id"* type=*"int"* value=*"111"*></constructor-arg>

</bean>

2）实例工厂

2.1、工厂设计模式：帮助穿件类对象，一个工厂可以生产多个对象

|  |
| --- |
| **//实例工厂类**  **public** **class** PeopleFactory {  **public** People newInstance() {  **return** **new** People(1, "1");  }  }  //xml bean  <bean id=*"factory"* class=*"com.jun.pojo.PeopleFactory"*></bean>  <bean id=*"peo2"* factory-bean=*"factory"* factory-method=*"newInstance"*></bean>  //测试类  People people = ac.getBean("peo2", People.**class**); |

3）静态工厂

|  |
| --- |
| //静态工厂类方法  **public** **static** People newInstance2() {  **return** **new** People(1, "1");}  //xml bean  <!-- 静态工厂 -->  <bean id=*"peo3"* class=*"com.jun.pojo.PeopleFactory"* factory-method=*"newInstance2"*></bean>  //测试类  People people = ac.getBean("peo3", People.**class**); |

## 五、如何给Bean的属性赋值（注入）

1. 通过构造方法赋值
2. 设置注入（set方法）

2．1 ：属性为基本数据类型，或String简单类型

|  |
| --- |
| <bean id=*"peo"* class=*"com.jun.pojo.People"*>  <property name=*"id"* value=*"111"*></property>  <property name=*"name"* value=*"张三"*></property>  </bean> |

2.2：为map赋值

|  |
| --- |
| <property name=*"map"*>  <map>  <entry key=*"a"* value=*"b"*></entry>  <entry key=*"c"* value=*"d"*></entry>  </map>  </property> |

## DI（Dependency Injection）依赖注入

1. DI和IoC是一样的，当一个类（A）中需要依赖另一个类（B）作为对象时，把B赋值给A的过程就叫做**依赖注入**

|  |
| --- |
| <bean id=*"peo"* class=*"com.jun.pojo.People"*>  <property name=*"apple"* ref=*"apple"* ></property>  </bean>  <bean id=*"apple"* class=*"com.jun.pojo.Apple"*>  <property name=*"id"* value=*"111"*></property>  <property name=*"price"* value=*"100.0"*></property>  </bean> |

## Spring简化Mybatis

1. 导入jar包：mybatis所有jar包和spring基本包，spring-jdbc，spring-tx，

spring-aop，mybatis-spring

1. 简化mybatis-config.xml配置

|  |
| --- |
| <!-- 数据源封装类。 数据源：获取数据库连接，spring-jdbc.jar中 -->  <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/ssm"*></property>  <property name=*"username"* value=*"root"*></property>  <property name=*"password"* value=*"admin"*></property>  </bean>  <!--创建SqlSessionFactory对象 -->  <bean id=*"factory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 数据库连接信息来源于dataSource -->  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 扫描器：相当与mybatis中扫描package。扫描com.jun.mapper包后会给对应接口创建对象 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <!-- 要扫描那个包 -->  <property name=*"basePackage"* value=*"com.jun.mapper"*></property>  <!-- 和factory产生关系 -->  <property name=*"sqlSessionFactory"*ref=*"factory"*></property>  </bean> |

3.将实现类交由spring管理

|  |
| --- |
| **//实现类中注入,并加入getter/setter方法**  **private** LogMapper logMapper;  //xml bean  <!--由spring管理实现类 -->  <bean id=*"logService"*class=*"com.jun.service.impl.LogServiceImpl"*>  <property name=*"logMapper"* ref=*"logMapper"*></property>  </bean> |

# Day2

## AOP：面向切面编程

概念：

1. **切点**：pointcut
2. **前置通知**：before advice
3. **后置通知**：after advice
4. **异常通知**：throws advice（切点执行过程若出现异常时触发）
5. **切面**：所有功能的总称。
6. **织入**：把切面嵌入到原有功能的过程。

## AOP的两种实现

1. **Schema-based**:每个通知都**需要实现接口或类**，配置时在**<aop:config>**中
2. 导入jar包：aopalliance.jar,aspectjweaver.jar…
3. 新建前置，后置通知，并在xml中配置<bean>

**前置**通知中**before**中的参数：

**Method arg0：切点方法对象**

**Object[] arg1：方法参数**

**Object arg2：对象（方法在哪个类中）**

**后置**通知中**afterReturning**中的参数：

**Object arg0：方法返回值**

**Method arg0：切点方法对象**

**Object[] arg1：方法参数**

**Object arg2：对象（方法在哪个类中）**

|  |
| --- |
| //前置通知  **public** **class** MyBeforeAdvice **implements** **MethodBeforeAdvice** {  @Override  **public** **void** before**(Method arg0, Object[] arg1, Object arg2**) **throws** Throwable {  System.***out***.println("执行前置通知");  }  }  //后置通知  **public** **class** MyAfterAdvice **implements** **AfterReturningAdvice** {  @Override  **public** **void** afterReturning(**Object arg0, Method arg1, Object[] arg2, Object arg3**) **throws** Throwable {  System.***out***.println("执行后置通知");  }  } |
| <bean id=*"mybefore"* class=*"com.jun.advice.MyBeforeAdvice"*></bean>  <bean id=*"myafter"* class=*"com.jun.advice.MyAfterAdvice"*></bean> |

1. 引入命名空间

|  |
| --- |
| xmlns:aop=[*http://www.springframework.org/schema/aop*](http://www.springframework.org/schema/aop)  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd* |

1. 切面配置

**\*表示通配符，匹配任意包名，任意类名，任意报名**

如果希望匹配任意方法参数：**（..）**

当方法中有参数时，并且需要前置或后置调用时，需要

**!!!*execution(\* com.jun.test.Demo.demo2(String，int)) and args(name,age)***

|  |
| --- |
| <!-- 配置切面 -->  <aop:config>  <!--配置切点 -->  <aop:pointcut expression=*"execution(****\**** *com.jun.test.Demo.demo2****()****)"* id=*"****mypoint****"*/>  <!-- 引入通知 -->  **<aop:advisor advice-ref=*"mybefore"* pointcut-ref=*"mypoint"*/>**  **<aop:advisor advice-ref=*"myafter"* pointcut-ref=*"mypoint"*/>**  </aop:config> |

1. **AspectJ:**不需要实现接口或类，配置时在**<aop:config>的子标签<aop:aspect>**中配置

aop:after : **无论程序有没有异常都执行下去**

after-returning： **程序有异常 则不执行**

这两个执行顺序与配置顺序有关

|  |
| --- |
| <!-- 配置切面 ：2、 AspectJ方式-->  <aop:config>  <aop:aspect>  <!--切点配置 -->  <aop:pointcut expression=*"execution(\* com.jun.test.Demo.demo2())"* id=*"mypoint"*/>  <aop:before method=*"myBefore"*  pointcut-ref=*"mypoint"*/>  <**aop:after** method=*"myAfter"*  pointcut-ref=*"mypoint"*/>  <**aop:after-returning** method=*"myAfterReturning"* pointcut-ref=*"mypoint"*/>  </aop:aspect>  </aop:config> |

## 异常通知的配置

1、AspectJ方式实现

异常通知：只有**切点报异常**才触发异常通知，只有在AspectJ配置

|  |
| --- |
| <!-- 异常通知类配置 -->  <bean id=*"****mythrow****"*class=*"com.jun.advice.MyThrowAdvice"*></bean>  <!-- 配置切面 ：2、 AspectJ方式-->  <aop:config>  <aop:aspect>  <!--切点配置 -->  <aop:pointcut expression=*"execution(\* com.jun.test.Demo.demo2())"* id=*"mypoint"*/>  <!-- 异常通知配置 method：触发异常通知时调用的方法 -->  <aop:after-throwing method=*"* **myexception** *"*  pointcut-ref=*"****mypoint****"* throwing=*"e"*/>  </aop:aspect>  </aop:config>  //异常通知类  **public** **class** MyThrowAdvice {  **public** **void** myexception(**Exception e**) {  System.***out***.println("执行异常通知 " + e.getMessage());  }  } |

2、Schame-base方式实现

|  |
| --- |
| <bean id=*"****mythrow2****"*class=*"com.jun.advice.MyThrowAdvice2"*></bean>  <!-- 异常通知配置Schame-base方式 -->  <aop:config>  <aop:pointcut expression=*"execution(\* com.jun.test.Demo.demo2())"* id=*"mypoint"*/>  <aop:advisor advice-ref=*"mythrow2"* pointcut-ref=*"mypoint"*/>  </aop:config>  // 异常通知类  **public** **class** MyThrowAdvice2 **implements** ThrowsAdvice {  //手动实现，**参数必须是1个或者4个**  **public** **void** afterThrowing(Exception ex) **throws** Throwable {  System.***out***.println("执行异常通知：schame-base方式");  }  } |

## 环绕通知

1. schame-base实现

通知类需要实现：**MethodInterceptor，导入**aopalliance包

|  |
| --- |
| <!--环绕通知bean -->  <bean id=*"****myaround****"* class=*"com.jun.advice.MyAround"*/>  <aop:config>  <aop:pointcut expression=*"execution(\* com.jun.test.Demo.demo2())"* id=*"****mypoint****"*/>  <aop:advisor advice-ref=*"****myaround****"* pointcut-ref=*"****mypoint****"*/>  </aop:config>  **import** org.aopalliance.intercept.MethodInterceptor;  **import** org.aopalliance.intercept.MethodInvocation;  **public** **class** MyAround **implements MethodInterceptor** {  @Override  **public** Object invoke(MethodInvocation arg0) **throws** Throwable {  System.***out***.println("环绕---前置");  //放行  **Object result = arg0.proceed();**  System.***out***.println("环绕---后置");  return result;  }  } |

1. AspectJ方式

方法中 包含ProceedingJoinPoint

|  |
| --- |
| <!-- 环绕通知AspectJ 方式实现-->  <aop:config>  <aop:aspect ref=*"myaround"*>  <aop:pointcut expression=*"execution(\* com.jun.test.Demo.demo2())"* id=*"mypoint"*/>  <aop:around method=*"myAround"*  pointcut-ref=*"mypoint"*/>  </aop:aspect>  </aop:config>  **public** Object myAround(ProceedingJoinPoint p) **throws** Throwable {  System.***out***.println("环绕前");  Object proceed = p.proceed();  System.***out***.println("环绕后");  return proceed;  } |

## 使用注解（基于AspectJ）

1. 引入xmlns：context

|  |
| --- |
| xmlns:context=[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)  *http://www.springframework.org/schema/context* [*http://www.springframework.org/schema/context/spring-*](http://www.springframework.org/schema/context/spring-) *context.xsd* |

1. 告诉spring扫描那个包下的注解

|  |
| --- |
| <**context:component-scan**  base-package=*"com.jun.advice,com.jun.Demo"/*> |

1. 替代bean标签：

<bean id="demo" class="com.jun.test.Demo">

|  |
| --- |
| @Component//此时 demo为Demo的id  **public** **class** Demo |

1. ……..太麻烦

## 代理设计模式

优点：保护真实对象；让真实对象职责更明确；易扩展

1. **静态代理**
2. **动态代理**
3. JDK动态代理：需要实现接口，重写方法
4. cglib动态代理：基于字节码（效率更高），生成真实对象的子类，不需要实现接口
5. 使用spring aop时，只要出现Proxy和真实对象转换异常，在xml中

|  |
| --- |
| <aop:aspectj-autoproxy  proxy-target-class=*"true"*>  </aop:aspectj-autoproxy> |

# Day3

## 自动注入

1、在**spring配置文件**中，**对象名和ref=”id” id名相同**时使用自动注入，可以不配置<property/>，自动注入 autowire

autowire有5个值:

1. default:与全局 **default-autowire**相同，默认为no
2. no: 不自动注入
3. byName：通过名称自动注入,在spring容器中找其他类的id
4. byType:根据类型注入，spring容器中不可以出现两个相同的bean
5. construct：必须提供有参构造方法，底层使用的byName，参数名和某个bean的id相同

## 通过软编码实现数据库链接

1. 编写**db.properties**

|  |
| --- |
| **jdbc.driver**=com.mysql.jdbc.Driver  **jdbc.url**=jdbc:mysql://localhost:3306/ssm  **jdbc.password**=admin  **jdbc.username**=root |

1. Xml文件

|  |
| --- |
| Xmlns：  xmlns:context=[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  db.properties文件位置  <context:property-laceholder location=*"classpath:db.properties"*/>  <!-- 获取数据源 -->  <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"****${jdbc.driver}****"*></property>  <property name=*"url"* value=*"****${jdbc.url}****"*></property>  <property name=*"username"* value=*"****${jdbc.username}****"*></property>  <property name=*"password"* value=*"****${jdbc.password}****"*></property>  </bean>  <!-- 使用软编码连接数据库时 在扫描器中-->  <property name=*"****sqlSessionFactoryBeanName****"* value=*"factory"*></property> |

## scope属性

**singleton** ：默认， 单例; **prototype**:多例

**request**：每次请求重新实例话 **session**：每次对话内对象为单例

**application**：在application对象内是单例

## 单例设计模式

1. 作用：在应用程序中保证最多只有一个实例
2. 好处：提升运行效率；实现数据共享
3. 懒汉式设计模式：对象只有被调用时才去创建，由于锁的存在，效率低

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *singleton*;  /\*  \* 私有化构造方法，不能通过new创建对象  \*/  **private** Singleton() {  }  /\*  \* 实例方法（静态），要想得到这个对象，必须调用此方法  \*/  **public** **static** Singleton getInstance() {  //如果已经实例化过 直接返回  **if** (*singleton* == **null**) {  //多线程访问下，可能出现if同时成立，需要添加锁  **synchronized** (Singleton.**class**) {  *singleton* = **new** Singleton();  }  }  **return** *singleton*;  }  } |

1. 饿汉式：

|  |
| --- |
| **public** **class** Singleton {  //在类加载时进行实例化  **private** **static** Singleton *singleton* = **new** Singleton();  **private** Singleton() {  }  **public** **static** Singleton getInstance() {  **if** (*singleton* == **null**) {  *singleton* = **new** Singleton();  }  **return** *singleton*;  }} |

## 声明式事务

1. 编程式事务：由程序员编程事务控制代码；OpensessionView编程式事务
2. 声明式事务：事务控制代码已经由spring写好，只需要声明出哪些方法需要进行事务控制和如何进行事务控制

主要针对于 ServiceImpl类下的方法，基于Aop的advice

|  |
| --- |
| Xml配置：  1、引入命名空间  xmlns:tx=[*http://www.springframework.org/schema/tx*](http://www.springframework.org/schema/tx)  *http://www.springframework.org/schema/tx*  [*http://www.springframework.org/schema/tx/spring-tx.xsd*](http://www.springframework.org/schema/tx/spring-tx.xsd)  2、实例化类  <!-- spring-jdbc.jar中 datasource.DataSourceTransactionManager-->  <bean id=*"txManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  3、配置声明式事务  <tx:advice id=*"****txAdvice****"* transaction-manager=*"****txManager****"*>  <tx:attributes>  <!--那些方法需要进行事务控制 -->  <!-- name可以用通配符：ins\*(所有ins开头的方法都控制)-->  <tx:method name=*"insert"*/>  </tx:attributes>  </tx:advice>  4、配置切点  <aop:config >  <aop:pointcut expression=*"excution(\* com.jun.service.impl.\*(..))"* id=*"****mypoin****"*/>  <aop:advisor advice-ref=*"****txAdvice****"* pointcut-ref=*"****mypoin****"*/>  </aop:config> |

1. <tx:method name=*"insert"*/>中属性解释

* **read-only**：如果为**true**，则告诉数据库此事务为**只读事务**。只要是**查询**方法，使用此事务。如果为false，则需要提交事务
* **propagation**：控制事务传播行为：当一个具有事务控制的方法被另一个有实物控制的方法调用时，需要如何控制事务（新建事务？在事务中执行？把事务挂起？报异常？）

取值：

1. **REQUIRED（默认）**：必须在**事务**中执行。如果当前有事务，就在事务中执行，如果当前没有事务，就新建事务
2. **SUPPORTS**：**事务**随便有没有。如果当前有事务，就在事务中执行，如果没有事务，就在非事务中执行。
3. **MANDATORY**：必须在**事务**内部执行。如果当前有事务就在事务中执行，没事务就报异常
4. **REQUIRED\_NEW:**必须在**事务**中执行。如果有事务，则把原来的事务挂起，新建一个事务；如果当前没有事务，则新建一个事物
5. **NOT\_SUPPORTED：**必须在**非事务**中执行。如果当前有事务，把当前事务挂起；如果没有，正常执行。
6. **NEVER：**必须在**非事务**状态下执行。如果有事务则报错；如果没有，则正常执行
7. **NESTED：**必须在**事务状态**下执行。如果当前有事务，则创建一个嵌套事务；如果没有事务，则新建事务
8. isolation：事务隔离级别

在多线程或并发访问下如何保证访问到的数据具有完整性的

1. **脏读：**一个事务（A）读取到另一个事务（B）中未提交的数据，而另一个事务中数据可能进行了改变，此时A事务读取的数据可能和数据库中数据不一致。
2. **不可重复读：**主要针对某行数据（或行中某列）的修改操作；当前事务A第一次读取事务后，事务B对事务A读取的数据进行修改，事务A中再次读取的数据和之前读取的数据不一致。
3. **幻读：**主要针对新增或删除；事务A按照特定的条件查询出结果，事务B新增了一条符合条件的数据。事务A中查询的数据和数据库中的数据不一致。事务A好像出现了幻觉

isolation的取值：

1. **DEFAULT**：默认值，由底层数据库自动判断应该使用什么隔离级别
2. **READ\_UNCOMMIT**：可以读取未提交的数据，可能出现脏读，幻读，不可重复读。效率最高
3. **READ\_COMMIT**：只能读取其他事务已提交数据**。可以防止脏读**；可能出现不可重复读和幻读
4. **REPEATABLE\_READ**：读取数据被添加锁，防止其他事务修改此数据，可以**防止不可重复读、脏读**，可能出现幻读
5. **SERIALIZABLE**：排队操作，对整个表添加锁。一个事务A操作数据时，另一个事务B等待事务A操作完成后才能操作表。**最安全、效率最低**
6. rollback-for=”异常类型全限定路径”:当出现什么异常时需要进行回滚

6、 no-rollback-for=”x”:当出现x异常时不需要进行回滚

## Spring中常用的注解

（1）、**@Component** 创建类对象，相当于配置<bean/>

（2）、**@Service**与@Component功能相同，写在ServiceImpl类上

（3）、**@Repository**与@Component功能相同，写在数据访问层类上

（4）、**@Controller**与@Component功能相同，写在控制器类上

（5）、**@Resource**：java中的注解，默认按照byName注入，如果没有，名称对象，则按照byType注入。使用此注解不需要写Getter/Setter方法，且对象名称最好写成和spring容器中的对象名一样

（6）、**@Autowired**：spring注解；默认按照byType注入，不需要写Getter/Setter方法

（7）、**@Value()**获取properties文件中的内容

（8）、**@Pointcut()** 定义切点

（9）、**@Aspect()**定义切面类

（10）、**@Before()** 前置通知

（11）、**@After()** 后置通知

（12）、**@AfterReturn()** 后置通知 必须切点正常执行

（13）、**@AfterThrowing()** 异常通知

（14）、**@Arround()**环绕通知