Spring 总结-基于 Maven 工具

【<<<<控制反转（IoC--Inversion Of Control）>>>>】

一、配置：

核心思想：（1）通过反射创建对象

（2）把创建的对象都存起来，下次要用的时候直接在存的地方取就行

问题：对象存在哪里？

我们存的对象很多，并且有查询需求，所以使用 Map 最佳，在应用加载时候，创建一个 Map，存放 3 层对象，并称为容器

这种被动接收的方式获取对象的方式就是控制反转，是 Spring 的核心之一；

它包括依赖注入和依赖查找

明确目的：削减计算机程序的耦合

1、引入 Spring 核心依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

2、创建 beans.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
http://www.springframework.org/schema/beans 引入名称空间
约束：
dtd：MyBatis
schema：Spring

http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd 引入约束
-->
<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
     https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="唯一标识" class="生成对象的全限定名"></bean>

</beans>
```

3、创建容器对象，根据配置文件 id 获取对象

```
ApplicationContext applicationContext=new ClassPathXmlApplicationContext("beans.xml");
 Object mavenServiceImp = applicationContext.getBean("MavenServiceImp");
 System.out.println(mavenServiceImp);
```

二、IoC 细节:

1、容器对象的类结构图

a、beanFactory 是 Spring 的顶层接口

b、接口 ApplicationContext 是 beanFactory 的子接口

实现类：ClassPathXmlApplicationContext --从类路径下读取 xml 文件

AnnotationConfigApplicationContext --从绝对路径下读取 xml 文件

FileSystemXmlApplicationContext --以纯注解配置使用

c、BeanFactory 和 ApplicationContext 使用的区别：

--ApplicationContext 在创建容器的时候创建单例模式对象 一次就把对像创建出来 直接用就行

```
ApplicationContext applicationContext=new ClassPathXmlApplicationContext("beans.xml");
    Object mavenServiceImp = applicationContext.getBean("MavenServiceImp");
    System.out.println(mavenServiceImp);
```

--BeanFactory 在创建容器的时候没有创建对象 而是什么时候用什么时候才创建对象

```
    Resource resource=new ClassPathResource("beans.xml");
    BeanFactory f =new XmlBeanFactory(resource);
    Object mavenServiceImp = f.getBean("MavenServiceImp");
```

2、getBean()

--获取对象时候 可以直接通过 id

```
ApplicationContext a=new ClassPathXmlApplicationContext("beans.xml");
Object mavenServiceImp1 = a.getBean("MavenServiceImp1");
```

--也可以通过接口/实现类.class 还可以在一个接口有多个实现类的时候 id 和接口.class 都指定

```
ApplicationContext a=new ClassPathXmlApplicationContext("beans.xml");
MavenService mavenServiceImp1 = a.getBean("MavenServiceImp1", MavenService.class);
```

3、bean 标签

作用：

用于配置对象让 Spring 来创建：

默认调用类中的【无参构造】没有无参构造则不能创建成功

属性：

id：唯一标识，用于获取对象

class：类的全限定名，用于反射创建对象

scope:"singleton"---单例模式 单例模式对象在容器创建时候创建在容器销毁时候销毁：

"prototype"---多例模式 获取时候创建 当对象长时间没有被引用的时候被垃圾回收机制回收

"request" 对象存入 request 域中

"session" 对象存入 session 域中

4、创建 bean 对象的三种方式

a、直接创建（使用）

```
<bean id="MavenServiceImp2" class="com.zyq.serviceimp.MavenServiceImp"></bean>
```

b、静态工厂创建（麻烦）

```
<bean id="MavenServiceImp3" class="com.zyq.factory.BeanFactory" factory-method="getMavenService"></bean>
```

c、使用非静态工厂创建对象 第一步创建工厂对象 第二步创建对象（麻烦）

```
<bean id="factory" class="com.zyq.factory.BeanFactory_Instance"></bean>
<bean id="MavenServiceImp4" factory-bean="factory" factory-method="getMavenService"></bean>
```

三、依赖注入：

1、什么是依赖注入？

业务层需要持久层的对象，在配置文件中给业务层传入持久层的对象

2、构造方法注入：

value:只能注入简单数据类型和 String

ref:可以注入 pojo 类型 相当于自己 new 个对象

```
<bean id="User" class="com.zyq.pojo.User">
```

方法一（索引）：

```
<constructor-arg index="0" value="1"></constructor-arg>
<constructor-arg index="1" value="麒麒哥"></constructor-arg>
```

方法一（type）：

```
<constructor-arg type="int" value="1"></constructor-arg>
<constructor-arg type="java.lang.String" value="type 注入"></constructor-arg>
```

方法一（name）最好：

```
<constructor-arg name="id" value="1"></constructor-arg>
<constructor-arg name="name" value="根据 name 注入"></constructor-arg>
<constructor-arg name="time" ref="time"></constructor-arg>【date 类型需要使用 ref】
</bean>
```

因为 date 可以用 value 注入 需要使用 ref 所有创建时间类型对象

注意* `<bean id="time" class="java.util.Date"></bean>`

3、Set 方法注入(需要 POJO 提供 get/set 方法)：

```
<bean id="User2" class="com.zyq.pojo.User">
    <property name="id" value="1"></property>
    <property name="name" value="张翼麒"></property>
    <property name="time" ref="time"></property>【依然需要使用 ref 注入】
    <property name="list" >
```

```
                <list>
                    <value>1</value>
                    <value>2</value>
                </list>
        </property>
        <property name="set" >
                <set>
                    <value>1</value>
                    <value>2</value>
                </set>
        </property>
        <property name="map" >
                <map>
                  <entry key="1" value="1"></entry>
                  <entry key="2" value="2"></entry>
                  <entry key="2" value-ref="time"></entry>【依然需要使用 ref 注入】
                </map>
        </property>
         <property name="array" >
                <array>
                  <value>1</value>
                  <value>2</value>
                  <value>3</value>
                </array>
              </property>
    </bean>
```

P 名称空间注入---了解

四、注解开发：

1、在 xml 配置文件中开启注解

a:需要在 xml 配置文件中开启注解

b:引入 context 名称空间-引入约束

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        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
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
```

c:需要扫描的包

```
<context:component-scan base-package="com.zyq">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller">
    </context:include-filter>
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller">
    </context:exclude-filter>
</context:component-scan>
</beans>
```

【base-package 写入包名】

context:include-filter 【指定包含过滤】

context:exclude-filter 【指定排除过滤】

type="annotation" 【按照类型过滤】

expression 【过滤表达式 只过滤标记了 Controller 注解的类】

2、@Component 注解：

a:注意：只能用在类上面，不能用在方法上

b:作用：只要类被标记，通过 xml 扫描包，就会创建对象；

c:衍生出三个子注解：

@Service-------Service 层

@Controller----------Web 层(控制层)

@Repository----------dao 层

【这三个是为了好的习惯 看着好看分为三层 不分/乱分也可以 尽量按照上面的开发】

d:相当于：<bean id="" class="" "></bean>

e:@Component("UserDao") 相当于 bean 标签 id 属性

-如果没有指定 id 默认【小写首字母后的类名】

eg:UserDaoImp---userDaoImp

3、@Autowired--自动注入

标记在属性或者是 set 方法上，如果是标记在属性上，可以没有 set 方法

特点：自动按照【类型注入】

流程：当属性/set（）标记了@Autowired 会自动在容器中寻找该类型的对象，如果只有一个，则注入

@Qualifier---【必须和@Autowired 一起使用】

使用情况：eg：一个接口有多个实现类，就不能只用@Autowired 类型注入需要搭配 @Qualifier

作用：如果按照类型注入失败，则会按照指定名称注入。eg： @Qualifier("mavenServiceImp") 默认名称

4、 @Resource--自动注入（好用）

eg：@Resource(name="mavenServiceImp")

流程：当属性/set（）标记了@Resource，会自动按照名称注入，如果名称没找到，则会按照类型注入

【尽量把属性变量名写成对象类名（默认名称）】

【区别】

@Autowired：默认按照类型注入，类型没找到则需结合@Qualifier 按照名称注入--Spring 提供

@Resource：默认按照名称注入，名称没找到，则按照类型注入（最好指定名称）--JDK 提供

5、@Configuration--标记该类为配置文件可以替换 ApplicationContext.xml

6、@ComponentSacn({"com.zyq"})--扫描包 相当于<context:component-scan base-package="com.zyq">

7、@Import--引入其他配置文件

相当于 xml <import resource=""></import>

8、@Bean--通过方法创建对像

9、@Scope("singleton/prototype")--配置单例或者多例

相当于<bean>中的属性 Scope

10、@PostConstruct--相当于 bean 标签的 init-method,指定初始化方法（了解）

11、@ProDestroy--相当于 bean 标签属性 destory-method，指定销毁方法（了解）

12、@Value--直接给【简单类型】赋值 相当于 bean 标签的子标签 property 进行赋值，遇到 pojo 类型还要用@Autowired

【<<<<面向切面编程(AOP--Aspect Oriented Programming)>>>>】

一、概念：

AOP 是 OOP 的延续，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，

从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

简单说就是---【把程序重复的代码抽取出来，在需要执行的时候，使用动态代理技术，在不修改源码的情况下，

对已有的方法经行增强】

作用：在不修改源码的情况下，对已有的方法经行增强

优势：减少重复代码；

提高开发效率；

维护方便；

实现方式：动态代理---JDK

---CGLIB

(1)Aspect(切面):切入点+通知==>织入

(2)JointPoint(连接点):程序执行过程中明确的点，是指那些被拦截的点，在 Spring 中指的就是方法

(3)Advice(通知):AOP 在特定的切入点上执行的增强处理，有 before,after,afterReturning,afterThrowing,around

(4)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式

(5)Weaving（织入）：将 Aspect 和其他对象连接起来, 并创建 Adviced object 的过程

二、XML 配置

1、引入依赖：1.8.7 以上（支持 Spring5）

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.9</version>
</dependency>
```

2、配置 Spring Xml 文件：

(1)、 创建要通知的对象

```
<bean id="Example1" class="类全限定名"></bean>
```

(2)、通知类型：

1、前置通知 :方法执行前执行

2、后置通知：方法执行后、返回值之前执行 若有异常停止执行

3、最终通知：方法执行后总会执行：finally

4、异常通知：方法出现异常执行

5、环绕通知：前置+后置+最终+异常

(3)、配置 AOP：

```
<aop:config>
【配置切面=切点+通知 指定通知对象是谁-ref】
    <aop:aspect ref="Example1">

        【配置切点】
        【id：唯一标志】
        【expression:表达式】
            【 * com.zyq.serviceimp.*.*(..) 】
            第一个*：任意返回值
            com.zyq.serviceimp:包名
            第二个*：包内任意类
            第三个*：任意方法
            (..):任意参数、任意参数个数、任意参数顺序

        <aop:pointcut id="A" expression="execution(* com.zyq.serviceimp.*.*(..))"></aop:pointcut>

        【织入】
        <aop:before method="before" pointcut-ref="A"></aop:before>
        <aop:after-returning method="afterReturn" pointcut-ref="A"></aop:after-returning>
```

```
        <aop:after method="after" pointcut-ref="A"></aop:after>
        <aop:after-throwing method="exaction" pointcut-ref="A"></aop:after-throwing>

          【around 可代替以上】
          <aop:around method="around" pointcut-ref="A"></aop:around>
      </aop:aspect>
    </aop:config>
```

【 <aop:before/after-returning/after/after-throwing method="通知对象内的方法" 】

【pointcut-ref="切面"】

3、连接点---JoinPoint 拦截到的方法

(1)获取拦截方法信息 eg：

```
public void before(JoinPoint joinPoint){
    //被代理对象
    Object target = joinPoint.getTarget();
    //获取拦截的类名
    String name = target.getClass().getName();
    System.out.println("拦截到的类名:"+name);
    //获取方法对象
    Signature signature = joinPoint.getSignature();
    String name1 = signature.getName();
    System.out.println("拦截的方法名："+name1);
    System.out.println("前置通知~~~");
}
```

(2)若出现异常 获取异常信息 eg：throwing="e"

需要在 after-throwing 标签对应的方法传入参数【Exception e】,并且在标签里面写入属性【throwing="e"】

```
public void exaction(Exception e){
        System.out.println("出现异常:"+e);
        System.out.println("异常通知~~~");
    }
```

(3)环绕通知(around)

而其他通知只能在拦截的方法前后增加操作不能接收返回值 不能自己执行

eg:

```
 public void around( ProceedingJoinPoint point){
        try {
            获取连接的方法信息
            String classname = point.getClass().getName();
            System.out.println("拦截的类:"+classname);
            String methonname = point.getSignature().getName();
            System.out.println("拦截的方法："+methonname);
```

```
            System.out.println("前置通知~~~");

            //执行被拦截的原方法 可以接收返回值
            point.proceed();

            System.out.println("后置通知~~~");
        } catch (Throwable e) {
            System.out.println("异常通知~~~");
            e.printStackTrace();
        }finally {
            System.out.println("最终通知~~~");
        }
    }
```

三、注解开发

1、 开启 AOP 自动代理 注解代理

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

2、在通知对象中配置 eg：

```
@Component
@Aspect 【设置切面=切点+通知】
public class Logger {

    【设置切入点】
    @Pointcut("execution(* com.zyq.serviceimp.*.*(..))")
    public void pointcut(){

    }
【织入】
  @Before("pointcut()")
  public void before(){
      System.out.println("前置通知~~~");
  }

  @AfterReturning("pointcut()")
  public void afterReturn(){
      System.out.println("后置通知~~~");
  }

  @After("pointcut()")
  public void after(){
      System.out.println("最终通知~~~");
```

```
}
【异常通知需要设置异常参数 e】
@AfterThrowing(value = "pointcut()",throwing = "e")
 public void exaction(Exception e){
      System.out.println("出现异常:"+e);
      System.out.println("异常通知~~~");
 }
```

3、注解

```
@Component
@Aspect
@Pointcut("execution(* com.zyq.serviceimp.*.*(..))")
@Before("pointcut()")
@AfterReturning("pointcut()")
@After("pointcut()")
@AfterThrowing(value = "pointcut()",throwing = "e")
@Around("pointcut()")
```

四、JdbcTemplate(了解--以后使用最多为 MyBatis)

1、常见的数据库访问：

JDBC--dbUtils--JdbcTemplate(Spring 提供)--Mybatis(主流)--Spring data jpa(趋势)

2、常见数据源

--C3P0 数据库连接池

--DBCP 数据库连接池

--Spring 自带数据库连接池(JdbcTemplate)

3、配置

(1) 配置数据源

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="username" value="${jdbc.user}"></property>
        <property name="password" value="${jdbc.password}"></property>
         <property name="driverClassName" value="${jdbc.driverClass}"></property>
         <property name="url" value="${jdbc.jdbcUrl}"></property>
 </bean>
```

(2)配置 JdbcTemplate

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

(3)使用：eg：

```
ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");
```

【获取 IoC 容器中 JdbcTemplate 实例】

```
JdbcTemplate jdbcTemplate=(JdbcTemplate) ctx.getBean("jdbcTemplate");
String sql="insert into user (name,deptid) values (?,?)";
int count= jdbcTemplate.update(sql, new Object[]{"caoyc",3});
System.out.println(count);
```

(4)方法：

execute：没有返回值，可以执行所有 SQL 语句，一般用于执行 DDL 语句。

update：返回的是一个 int 值，影响的行数， 用于执行 INSERT、UPDATE、DELETE 等 DML 语句。

增删改都只是使用到了一个方法： update(sql,Object…args)

queryXxx：用于 DQL 数据查询语句。

【queryXxx】

queryForObject(sql,数据类型.class) 查询单个对象

queryForMap(sql,参数) 查询单个对象，返回一个 Map 对象

queryForObject(sql,new BeanPropertyRowMapper(类型),参数) 查询单个对象,返回单个实体类对象

queryForList(sql,参数) 查询多个对象，返回一个 List 对象，List 对象存储是 Map 对象

query(sql,new BeanPropertyRowMapper(),参数) 查询多个对象，返回的是一个 List 对象，List 对象存储是实体类

五、Spring 的事务控制

1、什么是事务？

事务是逻辑上的一组操作，组成这组操作的各个逻辑单元，要么一起成功，要么一起失败。

2、事务的特性（ACID）

原子性：一个事务中所有对数据库的操作是一个不可分割的操作序列，要么全做要么全不做

一致性：数据不会因为事务的执行而遭到破坏

隔离性：一个事物的执行，不受其他事务的干扰，即并发执行的事物之间互不干扰

持久性：一个事物一旦提交，它对数据库的改变就是永久的

3、隔离级别：

(1)未提交读（read uncommited） :

脏读：读到了未提交的数据

不可重复读：一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致

幻读：一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

产生问题：脏读，不可重复读，幻读(虚读)

(2)已提交读 （read commited）:

产生问题：不可重复读，幻读

(3)可重复读 （repeatable read）:

产生问题：幻读

(4)串行化的(序列化) （serializable） :

避免以上所有读问题

4、数据库支持的隔离级别：

MySQL:read uncommited--read commited--repeatable read--serializable

默认：repeatable read

Oracle：read commited--serializable--read only(只读)

默认：read commited

5、事务的传播：

【掌握】

REQUIRED-必要的：若没有事务，则创建一个事务，若有事务，则加入这个事务【Spring 默认】

增删改

SUPPORTS-支持的：若没有事务，则非事务执行，若有事务，则加入这个事务

查询

【了解】

MANDATORY:可以使用当前事务，如果没有事务，抛出异常

REQUERS_NEW:新建一个事务，如果存在事务，则挂起事务

NOT_SUPPORTED:必须非事务执行，如果有事务，挂起事务

NEVER:非事务执行，如果存在事务，抛出异常

NESTED：有事务，嵌套执行，没有事务 REQUIRED

6、是否为只读的事务：

若是查询，则为只读的事务 readOnly=true

若是增删改，则为非只读的事务 readOnly=false

7、API 介绍：

Spring 的事务管理类的顶层接口-->>>PlatformTransactionManager

根据不同的持久层框架提供了不同的实现类

【使用 Spring JDBC 或 iBatis 时使用】

org.springframework.jdbc.datasource.DataSourceTransactionManager

【 使用 Hibernate 时使用】

org.springframework.orm.hibernate3.HibernateTransactionManager

8、基于 XML 的声明式事务管理：

a:【编程式事务管理】

以前用的在 java 业务层中直接写事务管理的方式

b:【声明式事务管理】

在 XML 文件中声明事务对象，管理事务，业务层中没有如何事务代码

9、引入依赖：

```
<!--spring 数据源 包含事务管理类-->
    <dependency>
        <groupId>org.springframework</groupId>
```

```
        <artifactId>spring-jdbc</artifactId>
        <version>4.3.1.RELEASE</version>
    </dependency>
    <!--spring 事务管理-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>4.3.1.RELEASE</version>
    </dependency>
    <!--AOP 切面必要-->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.9</version>
    </dependency>
```

10、配置 XML：

```
<!--数据源-->
<bean id="springDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"></property>
    <property name="url"
        value="jdbc:mysql://localhost:3306/test?useUnicode=true&amp;
        characterEncoding=utf8&amp;serverTimezone=GMT%2B8&amp;useSSL=false"></property>
    <property name="username" value="root"></property>
    <property name="password" value="123456"></property>
</bean>
<!--创建事务管理器对象-->
<bean id="TxManager" class=" org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="springDataSource"></property>
</bean>

<!--事务管理器增强(过滤方法是否需要拦截-查询方法不需要开启事务)
id:唯一标识
transaction-manager：指定事务管理器
-->
<tx:advice id="TxAdvice" transaction-manager="TxManager">
    <!--方法的过滤-->
    <tx:attributes>
        <!--指定需要拦截的方法
            isolation:隔离级别
            propagation：传播的行为
            read-only：是否为只读的事务 增删改->非只读事务 false
                                查询->只读事务 true
            timeout：-1    永不超时   也可以随便设置 单位为秒
```

```
            通配符 *：只需要方法名以这些开头的就拦截
        -->
        <!--【配置方法一：增删改查询都写】-->
        <!--增删改-->
        <tx:method name="insert*" isolation="DEFAULT"
                   propagation="REQUIRED" read-only="false" timeout="-1"/>
        <tx:method name="update*" isolation="DEFAULT"
                   propagation="REQUIRED" read-only="false" timeout="-1"/>
        <tx:method name="add*" isolation="DEFAULT"
                   propagation="REQUIRED" read-only="false" timeout="-1"/>
        <tx:method name="delete*" isolation="DEFAULT"
                   propagation="REQUIRED" read-only="false" timeout="-1"/>
        <!--查询-->
        <tx:method name="find*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>
        <tx:method name="get*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>
        <tx:method name="select*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>

        <!--【配置方法二：只写查询 和其他】-->
        <tx:method name="find*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>
        <tx:method name="get*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>
        <tx:method name="select*" isolation="DEFAULT"
                   propagation="SUPPORTS" read-only="true" timeout="-1"></tx:method>
        <!--其他-->
        <tx:method name="*"></tx:method>
    </tx:attributes>
</tx:advice>

<!--配置切面=切入点+通知(增强)-->
<aop:config>
    <!--配置切面-->
    <aop:advisor advice-ref="TxAdvice" pointcut="execution(* com.zyq.serviceimp.*.*(..))">
    </aop:advisor>
</aop:config>
```