# 一、Spring IOC

## 1、IOC和DI理解

（1）IOC反转是将Bean的控制权（创建，销毁以及查找依赖）交给容器控制；

（2）IOC实现的方式：依赖注入以及依赖查找

（3）DI注入方式：setter注入（通过setter方法注入属性），构造注入（通过构造方法注入属性）、工厂注入

（4）spring管理Bean的方式 3种：

①xml

②注解

@Bean，@Component，@Repository，@Service，@Controller

@AutoWired (required =f alse) //先Type后Name，required默认为true，false时表示允许为null

@Resource //先Name后Type

@Condition(MyCondition.class) //条件装配，MyCondition.class需实现Condition接口

@Primary //首选项

@Qualifier(“dog”) //限定符，与@AutoWired搭配使用，以类型和名称查找对应的bean进行注入

@Value// 3种使用方式：

@Value(“1”) ：基本数值

@Value(“#{beanName.str}”) ：SPEL表达式#{} ，引用bean属性

@Value(“${database.driverName}”) ：属性占位符${} ，读取配置在属性文件中的内容

@ConfigurationProperties(“database”) //database与POJO属性名组成全限定名，在配置文件中查找

@PropertySource(value={“classpath:jdbc.properties”,ignoreResourceNotFound=true})

//使用jdbc.properties属性文件，ignoreResourceNotFound表示忽略配置文件找不到的问题

③配置类

      @Configuration                           //声名为配置类

@ComponentScan("com")                    //指定扫面的包

@ImportResource("classpath:spring.xml")  //指定配置文件

public class AppConfig {

参数：basePackages/value、

includeFilters/excludeFilters={@Filter(type=...,value=....class),...}

lazyInit

           }

## 2、BeanFactory和ApplicationContext有什么区别

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

1. Bean实例化时间不同（懒加载原因）

①BeanFactory采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFactory加载后，直至第一次使用调用getBean方法才会抛出异常。

②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

## 3、Spring Bean

（1）作用域

|  |  |  |
| --- | --- | --- |
| 作用域类型 | 使用范围 | 作用域描述 |
| Single | 所有Spring应用 | 默认值，IOC容器只存在单例 |
| Prototype | 所有Spring应用 | 每当从IOC容器中获取Bean，都会创建一个新的Bean |
| session | Spring Web应用 | 在HTTP会话中，一个Bean定义对应一个实例 |
| application | Spring Web应用 | Web工程生命周期 |
| request | Spring Web应用 | Web工程单次请求 |
| globalSession | Spring Web应用 | 在一个全局的HTTPSession中，一个Bean定义对应一个实例 |

ConfigurableBeanFactory ：只提供了SCOPE\_SINGLETON和SCOPE\_PROTOTYPE作用域

WebApplicationContext：提供了SCOPE\_SINGLETON、SCOPE\_PROTOTYPE、SCOPE\_SESSION、SCOPE\_APPLICATION和 SCOPE\_REQUEST作用域

例：@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE,ProxyMode=ScopeProxyMode.INTERFACES)

//ProxyMode表示代理模式：接口（或cglib -> TARGET\_CLASS）

（2）Bean初始化流程、生命周期

Bean初始化流程

非lazyInit

解析，并将Bean定义保存到BeanDefinition

将Bean定义发布到IOC容器中

资源定位 (如@ComponentScan和xml等)

实例化

依赖注入(DI)

检查Aware相关接口，并设置相关依赖

实现BeanNameAware接口调用setBeanName方法

实现BeanClassLoaderAware接口调用setBeanClassLoader方法

实现BeanFactoryAware接口调用setBeanFactory方法

实现ApplicationContextAware接口 调用setBeanFactory方法 (容器需实现ApplicationContext接口)

调用BeanPostProcessor后置处理PostProcessorAfterInitialization方法

调用BeanPostProcessor前置处理PostProcessorBeforeInitialization方法

注解@PostConstruct标注的方法(自定义初始化方法)

实现InitializingBean接口 调用AfterPropertiesSet方法

自定义的initMethod方法(@Bean(initMethod=”init”))

使用中

注解@PostDestroy标注的方法(自定义初始化方法)

自定义的destroyMethod方法(@Bean(destroyMethod=”mDestroy”))

实现DisposableBean接口 调用destroy方法

结束

1. Spring EL

${...}：代表占位符； #{...}：代表启用Spring表达式；

判断是否为null

①读取属性文件 ：$(dataSource.driverName)

②表示字面值：#{9}, #{“Hello”}

③引用bean属性/方法：#{bean.name}, #{bean.getName()}, #{bean.getName()?.toUpperCase()}

T(...)代表引入的类 java.lang.\*包为java默认加载包，非此包下类需全限定名

④调用类方法：#{T(System).currentTimeMillis()}

⑤SPEL运算符

|  |  |
| --- | --- |
| 算数运算符 | +、-、\*、/、%、^ |
| 比较运算符 | <、<=、>、>=、==、lt、le、gt、ge、eq |
| 逻辑运算符 | and、or、not、| |
| 条件运算符 | a ? b:c (三元运算符)、a?b (a为null，则a=b) |
| 正则运算符 | match |

⑥计算集合

#{arrays[3]} ：arrays数组第4个元素

#{books.^[name eq “spring”]} ：匹配第一个

#{books.$[name eq “spring”]} ：匹配最后一个

#{books.![name]} ：投影，得到name数组

#{books.?[author eq “qyx”]} ：过滤，得到符合条件的子集

# **二、Spring AOP**

## 1、术语

关注点：重复的代码

连接点(Join point)：具体被拦截的方法

切点(Poincut)：拦截哪些方法

增强/通知(Advice)：表示添加到切点的一段逻辑代码，并定位连接点的方位信息(定义了是干什么的，具体是在哪干)

织入(Weaving)：将增强/通知添加到目标类的具体连接点上的过程。

引入(Introduction)：向现有的类添加新方法或属性。是一种特殊的增强！

切面(Aspect)：由切点和增强/通知组成

## 2、原理

JDK动态代理

Cglib动态代理

注：SpringAOP中，如果使用的是单例，推荐使用CGLib。

原因：JDK在创建代理对象时的性能要高于CGLib代理，而生成代理对象的运行性能却比CGLib的低

## 3、Spring AOP5种通知类型

@Component

@Aspect//指定为切面类

方法

返回任意类型

public class AOP {

@Pointcut("execution (\* aa.\*.\*(..))")// 指定切入点表达式，拦截哪个类的哪些方法

public void pt() {}

使用任意参数

方法所属类

@Before("pt()") // 前置通知 : 在执行目标方法之前执行

public void begin(){

System.out.println("开始事务/异常");

}

@Around("pt()") // 环绕通知：环绕目标方式执行

public void around(ProceedingJoinPoint pjp) throws Throwable{

System.out.println("环绕前....");

pjp.proceed(); // 执行目标方法

System.out.println("环绕后....");

}

@After("pt()") //后置通知：在执行目标方法之后执行【无论是否出现异常最终都会执行】

public void after(){ System.out.println("提交事务/关闭"); }

@AfterReturning("pt()") //返回后通知：在调用目标方法结束后执行 【出现异常不执行】

public void afterReturning() { System.out.println("afterReturning()"); }

@AfterThrowing("pt()") //异常通知：当目标方法执行异常时候执行此关注点代码

public void afterThrowing(){ System.out.println("afterThrowing()"); }

}

## 4、引入新功能

@Aspect

Public class MyAspect{

@Declareparents (value=”com.qyx.service.UserServiceImpl+”, //value指向需要增强功能的目标对象

defaultImpl=userValidatorImpl.class) //引入增强功能的类

Public UserValidator userValidator;

}

注：userValidator对象可以在UserValidator和UserService之间转化

## 5、通知中获取参数

@Before("pt() && args(user)")

public void beginParam(JoinPoint point, User user){

Object[] args = point.getArgs(); //环绕通知中用ProceedingJoinPoint.getArgs()

System.out.println("before ...");

}

## 6、多个切面

默认无序，实现有序方式 ① @Order(1)/@Order(2)

②切面实现Ordered接口

## 7、Spring Aop 和AspectJ关系

Spring AOP和AspectJ都是对目标类增强，

Spring AOP引入了AspectJ分格（使用了AspectJ的Annotation，可以使用@AspectJ定义切面，@Pointcut定义切点）

Spring AOP没有使用它的编译器和织入器，其实现原理是动态代理，在运行时生成代理类

它们最大的区别是Spring AOP是运行时增强，而AspectJ是编译时增强，AspectJ需要特殊的编译器（ajc）

Spring AOP支持的AspectJ切入点指示符：

|  |  |
| --- | --- |
| execution | 用于匹配方法执行的连接点 |
| this | 限制连接点匹配AOP代理Bean引用为指定类型  (注意：AOP代理对象的类型匹配，这样就可能包括引入接口和类型匹配) |
| args | 限定连接点方法参数 |
| @args | 限定连接点方法参数上的注解 |
| target | 限制目标对象类型（被代理对象）；  (注意：目标对象的类型匹配，这样就不包括引入接口也类型匹配) |
| @target | 限制目标对象配置了指定的注解 |
| within | 限制连接点匹配指定的类型 |
| @within | 限制连接点匹配指定注解类型 |
| @annotation | 限定带有指定注解的连接点 |
| bean | 用于匹配特定名称的Bean对象的执行方法 (Spring+AOP扩展的，AspectJ没有该指示符) |
| Reference pointcut | 表示引用其他命名切入点，只有AspectJ风格支持，Schema风格不支持 |

# **三、Spring 数据库**

## 1、多数据源

# 配置第一个数据源

spring.datasource.hikari.db1.jdbc-url=jdbc:mysql://127.0.0.1:3306/usms?useSSL=false&amp;rewriteBatchedStatements=true

spring.datasource.hikari.db1.username=root

spring.datasource.hikari.db1.password=153963

spring.datasource.hikari.db1.driver-class-name=com.mysql.cj.jdbc.Driver

# 配置第二个数据源

spring.datasource.hikari.db2.jdbc-url=jdbc:mysql://127.0.0.1:3306/usmshis?useSSL=false&amp;rewriteBatchedStatements=true

spring.datasource.hikari.db2.username=root

spring.datasource.hikari.db2.password=153963

spring.datasource.hikari.db2.driver-class-name=com.mysql.cj.jdbc.Driver

配置类：

@Configuration

@MapperScan(basePackages = "top.snailclimb.db1.dao", sqlSessionTemplateRef = "db1SqlSessionTemplate")

public class DataSource1Config {

    //生成数据源.  @Primary 注解声明为默认数据源

    @Bean(name = "db1DataSource")

    @ConfigurationProperties(prefix = "spring.datasource.hikari.db1")

    @Primary

    public DataSource testDataSource() {

        return DataSourceBuilder.create().build();

    }

    // 创建 SqlSessionFactory

    @Bean(name = "db1SqlSessionFactory")

    @Primary

  public SqlSessionFactory testSqlSessionFactory(@Qualifier("db1DataSource") DataSource dataSource) throws Exception {

        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

        bean.setDataSource(dataSource);

        return bean.getObject();

    }

    // 配置事务管理

    @Bean(name = "db1TransactionManager")

    @Primary

    public DataSourceTransactionManager testTransactionManager(@Qualifier("db1DataSource") DataSource dataSource) {

        return new DataSourceTransactionManager(dataSource);

    }

    @Bean(name = "db1SqlSessionTemplate")

    @Primary

public SqlSessionTemplate testSqlSessionTemplate(@Qualifier("db1SqlSessionFactory") SqlSessionFactory sqlSessionFactory)

throws Exception  {

        return new SqlSessionTemplate(sqlSessionFactory);

    }

}

## 2、事务特性

⑴ 原子性（Atomicity）：要么全部成功，要么全部失败

⑵ 一致性（Consistency）：一个事务执行之前和执行之后都必须处于一致性状态

⑶ 隔离性（Isolation）：多个并发事务之间相互隔离，互不干扰

⑷ 持久性（Durability）：一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的

## 3、事务的种类

①编程式事务

②声明式事务

## 4、隔离级别

①未提交读：允许一个事务读取另一个事务没有提交的数据

②读写提交：一个事务只能读取另一个事务已经提交的数据

③可重复读：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改

④串行化：所有的事务依次逐个执行

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复度 | 幻读 |
| 未提交读 | √ | √ | √ |
| 读写提交(Oracle默认) | × | √ | √ |
| 可重复读(mysql默认) | × | × | √ |
| 串行化 | × | × | × |

不可重复度：对应的是Update操作

幻读：对应的是insert操作

## 5、传播路径

支持当前事务的情况：

TransactionDefinition.PROPAGATION\_REQUIRED： 如果当前存在事务，则加入该事务；否则新建一个事务。

TransactionDefinition.PROPAGATION\_SUPPORTS： 如果当前存在事务，则加入该事务；否则以非事务的方式继续运行。

TransactionDefinition.PROPAGATION\_MANDATORY： 如果当前存在事务，则加入该事务；否则抛出异常。(强制性)

不支持当前事务的情况：

TransactionDefinition.PROPAGATION\_REQUIRES\_NEW： 创建一个新的事务，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED： 以非事务方式运行，如果当前存在事务，则把当前事务挂起。

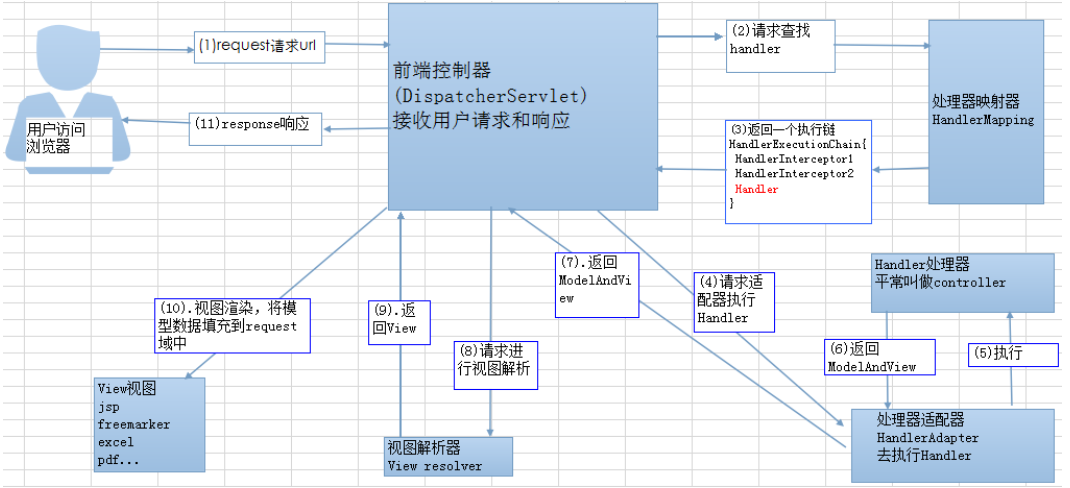
TransactionDefinition.PROPAGATION\_NEVER： 以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

TransactionDefinition.PROPAGATION\_NESTED： 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行(回滚只回滚子方法的sql)；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

# **四、Spring MVC**

## 1、MVC流程



## 2、获取控制器参数

①无注解情况：方法的参数名与HTTP请求的参数名保持一致

②使用@RequestParam 例：public String test(@RequestParam(“mete\_type”) String meteType)

③传递JSON 例：public String insert(@RequestBody User user)

注：该@RequestBody用于接收前端提交的JSON请求体（JSON请求体和User类之间属性名需保持一致）

④通过URL传递参数

例：@GetMapping(“/{id}”)

Public User get(@PathVariable Long id){...}

⑤获取格式化参数

例：@PostMapping(“/getalarm”)

Public JSONArray getAlarms(@DateTimeFormat(iso=ISO.DATE) Date date,

@NumberFormat(pattern=’#,###.##’) Double number) {...}

自定义参数转换规则

## 拦截器

（1）开发拦截器

①自定义拦截器：实现HandlerInterceptor

（常用HandlerInterceptorAdapter，需重写preHandle、postHandle和afterCompletion方法）

②注册拦截器：配置类实现WebMvcConfigurer接口

例：public class myConfigurer implement WebMvcConfigurer {

Public void addInterceptors(InterceptorRegistry registry){

//将拦截器Interceptors1注册到Spring MVC中，然后返回一个拦截器注册

InterceptorRegistryation ir = registry.addInterceptors(new Interceptors1());

//指定拦截器匹配模式

Ir.addPathPatterns(“/interceptor/\*”);

}

}

注：当注册多个拦截器时，preHandle方法是先注册先执行；

postHandle方法是先注册后执行；

afterCompletion方法是先注册后执行（只会执行返回true的拦截器的完成方法）

1. 拦截器和过滤器区别

①Filter需要在web.xml中配置，依赖于Servlet；

②Interceptor需要在SpringMVC中配置，依赖于框架；

③Filter的执行顺序在Interceptor之前

④两者的本质区别：Filter是基于函数回调，而Interceptor是基于Java的反射机制。拦截器相对更灵活些（Filter能做的事情，拦截器都能做，而且可以在请求前，请求后执行）。Filter主要是针对URL地址做一个编码的事情、过滤掉没用的参数、安全校验（比较泛的，比如登录不登录之类），太细的话，还是建议用interceptor。

## 国际化

Listen、filter

①②③④⑤