第07课：Enable\* 框架实现原理与实战

通过之前的文章《@EnableAutoConfiguration 原理实战》已经学习到了 @EnableAutoConfiguration 是如何获取 jar 包中的自动配置文件，并将它们装配到 Spring 上下文中的。本文将从注解使用的角度出发，剖析下 Spring Boot 中常用的 EnableXX 注解的套路原理与实现。并在掌握了原理的基础上，自定义实现个我们自己的自动配置注解。

当需要使用 Spring-Data-Jpa 时，只需要引入对应的 starter：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

添加上注解 @EnableJpaRepositories。

当需要整合 Web 开发环境时，只需要添加对应的 starter：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

就连 EnableXXX 都省了。

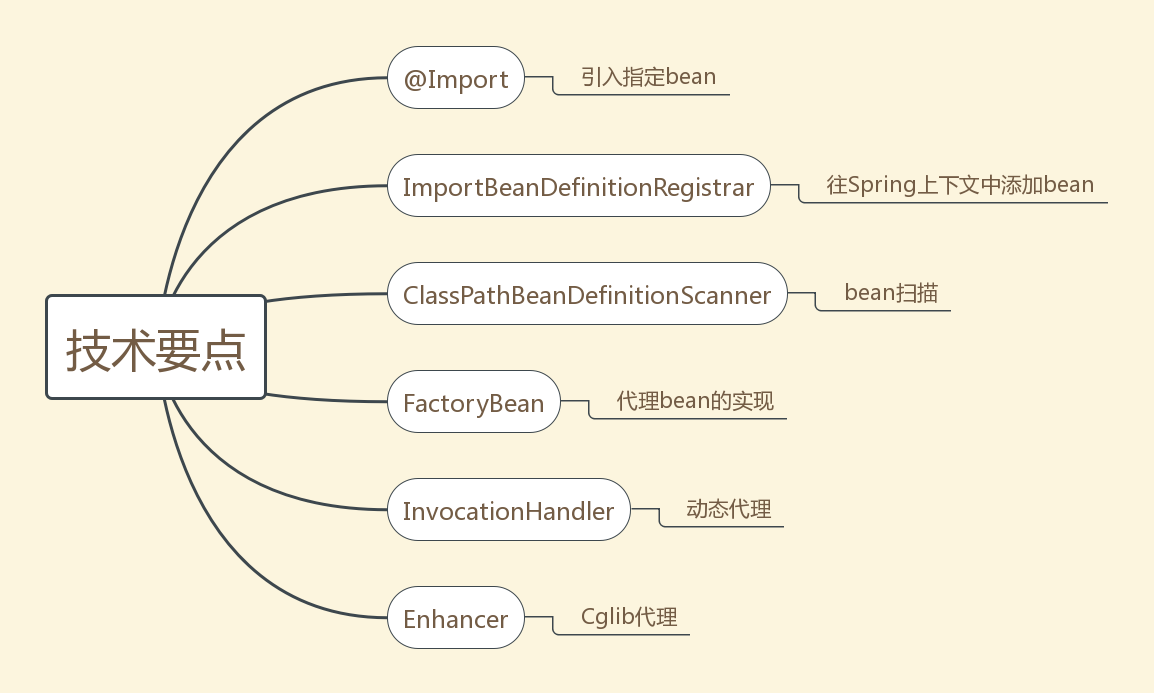
当你想要使用 MyBatis 时，只需要添加 @MapperScan，配置好数据源和对应的 SqlSessionFactory 就可以使用了，会自动将 mapper.xml 文件对应的 mapper 接口注入到 Spring 上下文中，就可以直接使用了。

这么轻易的就整合上了其他框架，是不是用得很爽？来剖析一下其中的奥秘吧！由于是原理分析和自定义实现，所以本文中的代码会比较多，阅读时建议打开 IDE 边看代码边阅读。

通过本文将学习到：

* 如何实现自己的 EnableXXX(AutoConfiguration) 框架；
* 如何使用 ImportBeanDefinitionRegistrar 动态注入 bean；
* 如何使用 FactoryBean 实现任意接口功能；
* 如何使用 JDK 动态代理代理接口；
* 如何使用 Cglib 实现代理接口。

对应的思维导图：



阅读下文前需要先了解下几个类的作用：ImportBeanDefinitionRegistrar，FactoryBean，ClassPathBeanDefinitionScanner，@Import。

ImportBeanDefinitionRegistrar 实现动态的往 Spring 容器中注入 bean。通过方法 registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) 往 Spring 上下文中注入 bean。

FactoryBean Spring 的工厂 bean，通过此类可以实现对任意 bean 的代理。

ClassPathBeanDefinitionScanner classpath 中 bean 的扫描器，具体原理可以看《第06课：Spring 中 Bean 扫描实战》内容。

@Import 导入 bean 到上下文中。

MyBatis 实例解析

只要查看各种 EnableXX 注解源码，可以知道注解上都添加了 @Import 注解。通过 @Import 可以导入框架需要的 bean。通常引入的 bean 分两种：

* 实现了 ImportSelector 接口的 XXXSelector；
* 实现了 ImportBeanDefinitionRegistrar，接口的 XXXRegistrar。

ImportSelector 与 ImportBeanDefinitionRegistrar 区别在于，ImportSelector 可以根据当前环境动态地选择需要引入的 ImportBeanDefinitionRegistrar 来实现需要的功能。

比如 @EnableTransactionManagement 引入的 TransactionManagementConfigurationSelector，如果是使用代理模式的话，引入 AutoProxyRegistrar，如果是使用 Aspectj 来实现的话引入的是TransactionManagementConfigUtils.TRANSACTION\_ASPECT\_CONFIGURATION\_CLASS\_NAME从而达到不同环境或配置动态选择实现类的功能。源码和注解如下：

**public** **class** **TransactionManagementConfigurationSelector** **extends** **AdviceModeImportSelector**<**EnableTransactionManagement**> {

*/\*\**

*\* {@inheritDoc}*

*\* @return {@link ProxyTransactionManagementConfiguration} or*

*\* {@code AspectJTransactionManagementConfiguration} for {@code PROXY} and*

*\* {@code ASPECTJ} values of {@link EnableTransactionManagement#mode()}, respectively*

*\*/*

@Override

**protected** String[] selectImports(AdviceMode adviceMode) {

**switch** (adviceMode) {

**case** PROXY:

**return** **new** String[] {AutoProxyRegistrar.class.getName(), ProxyTransactionManagementConfiguration.class.getName()};

**case** ASPECTJ:

**return** **new** String[] {TransactionManagementConfigUtils.TRANSACTION\_ASPECT\_CONFIGURATION\_CLASS\_NAME};

**default**:

**return** **null**;

}

}

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Import(TransactionManagementConfigurationSelector.class)

**public** @interface EnableTransactionManagement {

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@Documented

@Import(MapperScannerRegistrar.class)

**public** @interface MapperScan {

了解了 ImportBeanDefinitionRegistrar 与 ImportSelector 区别后，下面选用 MyBatis 自动注入接口功能来剖析其运行原理。

首先说明下 MyBatis 机制简要流程。如下：

* 扫描 @MapperScan 设置的 basePackages 包下的接口；
* 根据上面的扫描结果，在上下文中注入 BeanDefinition；
* 通过 JDK 动态代理方式实现接口方法。

从 @MapperScan 注解源码中，可以看到使用 @Import 导入了 MapperScannerRegistrar 类（其他 EnableXXX 也是类似的）。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@Documented

@Import(MapperScannerRegistrar.class)

**public** @interface MapperScan {

查看 @Import 注释，只需要理解前两句注释：

Indicates one or more @Configuration classes to import. Provides functionality equivalent to the <import/> element in Spring XML .

意思就是该注解的作用和 xml 配置中 import 标签作用是相同的，都是引入需要的 bean。

然后继续查看类 MapperScannerRegistrar，可以发现实现了 ImportBeanDefinitionRegistrar 和 ResourceLoaderAware 接口：

**public** **class** **MapperScannerRegistrar** **implements** **ImportBeanDefinitionRegistrar**, **ResourceLoaderAware**

ImportBeanDefinitionRegistrar：

可以在 Spring 上下文中添加 BeanDefinition，也就是可以通过程序动态地注入 bean 到 Spring 上下文中。

ImportBeanDefinitionRegistrar 源码如下，已经去除了不需要的部分：

**public** **class** **MapperScannerRegistrar** **implements** **ImportBeanDefinitionRegistrar**, **ResourceLoaderAware** {

*//略。。。*

**public** **void** **registerBeanDefinitions**(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

*//获取MapperScan注解上属性对应的值*

AnnotationAttributes annoAttrs = AnnotationAttributes.fromMap(importingClassMetadata.getAnnotationAttributes(MapperScan.class.getName()));

ClassPathMapperScanner scanner = **new** ClassPathMapperScanner(registry);

*//其他属性设置，这里省略了。。。*

*//mapper对应的接口包*

List<String> basePackages = **new** ArrayList<String>();

**for** (String pkg : annoAttrs.getStringArray("value")) {

**if** (StringUtils.hasText(pkg)) {

basePackages.add(pkg);

}

}

**for** (String pkg : annoAttrs.getStringArray("basePackages")) {

**if** (StringUtils.hasText(pkg)) {

basePackages.add(pkg);

}

}

**for** (Class<?> clazz : annoAttrs.getClassArray("basePackageClasses")) {

basePackages.add(ClassUtils.getPackageName(clazz));

}

*//根据以上条件进行设置扫描后要筛选的类*

scanner.registerFilters();

*//开始扫描mapper文件*

scanner.doScan(StringUtils.toStringArray(basePackages));

}

*//略...*

}

registerBeanDefinitions 方法在 spring 容器启动时会回调此方法进行通过 BeanDefinitionRegistry 注入 bean。

ClassPathMapperScanner 继承了 ClassPathBeanDefinitionScanner。

Candidate classes are detected through configurable type filters. The default filters include classes that are annotated with Spring's @Component, @Repository, @Service, or @Controller stereotype.

默认的过滤器使用的是 AnnotationTypeFilter 只会扫描 @Component、@Repository、@Service or @Controller 注解的 bean。接口是不会扫描的。

ClassPathMapperScanner 源码：

**public** **class** **ClassPathMapperScanner** **extends** **ClassPathBeanDefinitionScanner** {

*//略...*

*//进行扫描的筛选，只需要筛选出符合mapper条件的接口*

**public** **void** **registerFilters**() {

**boolean** acceptAllInterfaces = **true**;

*// if specified, use the given annotation and / or marker interface*

**if** (**this**.annotationClass != **null**) {

addIncludeFilter(**new** AnnotationTypeFilter(**this**.annotationClass));

acceptAllInterfaces = **false**;

}

*// override AssignableTypeFilter to ignore matches on the actual marker interface*

**if** (**this**.markerInterface != **null**) {

addIncludeFilter(**new** AssignableTypeFilter(**this**.markerInterface) {

@Override

**protected** **boolean** **matchClassName**(String className) {

**return** **false**;

}

});

acceptAllInterfaces = **false**;

}

**if** (acceptAllInterfaces) {

*// default include filter that accepts all classes*

addIncludeFilter(**new** TypeFilter() {

**public** **boolean** **match**(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) **throws** IOException {

**return** **true**;

}

});

}

*// exclude package-info.java*

addExcludeFilter(**new** TypeFilter() {

**public** **boolean** **match**(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) **throws** IOException {

String className = metadataReader.getClassMetadata().getClassName();

**return** className.endsWith("package-info");

}

});

}

*//这里是重点，重写了父类的方法，将访问控制符变为public类型的*

@Override

**public** Set<BeanDefinitionHolder> **doScan**(String... basePackages) {

*//根据registerFilters方法中配置的筛选条件扫描指定包下的接口*

Set<BeanDefinitionHolder> beanDefinitions = **super**.doScan(basePackages);

**if** (beanDefinitions.isEmpty()) {

logger.warn("No MyBatis mapper was found in '" + Arrays.toString(basePackages) + "' package. Please check your configuration.");

} **else** {

*//筛选出对应的接口后，设置接口的实现*

**for** (BeanDefinitionHolder holder : beanDefinitions) {

GenericBeanDefinition definition = (GenericBeanDefinition) holder.getBeanDefinition();

**if** (logger.isDebugEnabled()) {

logger.debug("Creating MapperFactoryBean with name '" + holder.getBeanName()

+ "' and '" + definition.getBeanClassName() + "' mapperInterface");

}

*// the mapper interface is the original class of the bean*

*// but, the actual class of the bean is MapperFactoryBean*

definition.getPropertyValues().add("mapperInterface", definition.getBeanClassName());

*//通过FactoryBean设置对应的接口实现类*

definition.setBeanClass(MapperFactoryBean.class);

definition.getPropertyValues().add("addToConfig", **this**.addToConfig);

**boolean** explicitFactoryUsed = **false**;

**if** (StringUtils.hasText(**this**.sqlSessionFactoryBeanName)) {

definition.getPropertyValues().add("sqlSessionFactory", **new** RuntimeBeanReference(**this**.sqlSessionFactoryBeanName));

explicitFactoryUsed = **true**;

} **else** **if** (**this**.sqlSessionFactory != **null**) {

definition.getPropertyValues().add("sqlSessionFactory", **this**.sqlSessionFactory);

explicitFactoryUsed = **true**;

}

**if** (StringUtils.hasText(**this**.sqlSessionTemplateBeanName)) {

**if** (explicitFactoryUsed) {

logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory together. sqlSessionFactory is ignored.");

}

definition.getPropertyValues().add("sqlSessionTemplate", **new** RuntimeBeanReference(**this**.sqlSessionTemplateBeanName));

explicitFactoryUsed = **true**;

} **else** **if** (**this**.sqlSessionTemplate != **null**) {

**if** (explicitFactoryUsed) {

logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory together. sqlSessionFactory is ignored.");

}

definition.getPropertyValues().add("sqlSessionTemplate", **this**.sqlSessionTemplate);

explicitFactoryUsed = **true**;

}

**if** (!explicitFactoryUsed) {

**if** (logger.isDebugEnabled()) {

logger.debug("Enabling autowire by type for MapperFactoryBean with name '" + holder.getBeanName() + "'.");

}

definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE\_BY\_TYPE);

}

}

}

**return** beanDefinitions;

}

*//这里是重点，判断对应的bean是否满足筛选的要求，因为扫描的是接口，所以这里只筛选对应的接口数据*

@Override

**protected** **boolean** **isCandidateComponent**(AnnotatedBeanDefinition beanDefinition) {

**return** (beanDefinition.getMetadata().isInterface() && beanDefinition.getMetadata().isIndependent());

}

*//略...*

}

通过以上步骤筛选出了符合条件的接口类，下一步就是看如何实现了，核心代码在 MapperFactoryBean 中，该类实现了 FactoryBean 接口，表明此类是个工厂 bean，通过实现 getObject 来对接口进行代理。

**public** **class** **MapperFactoryBean**<**T**> **extends** **SqlSessionDaoSupport** **implements** **FactoryBean**<**T**> {

*//略...*

*//根据接口对象通过sqlSession获取到对应的实现*

**public** T **getObject**() **throws** Exception {

**return** getSqlSession().getMapper(**this**.mapperInterface);

}

*//返回代理的接口类型*

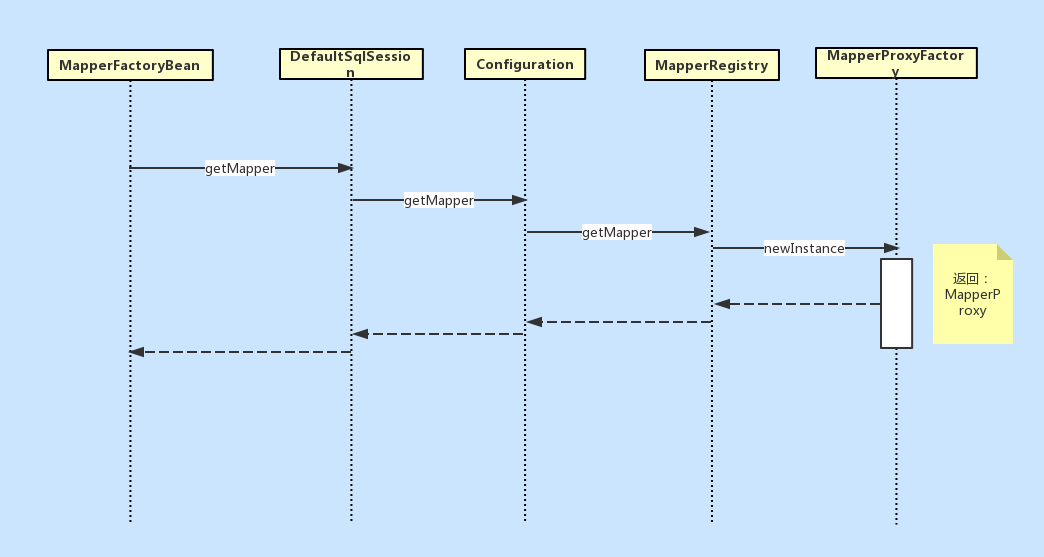
**public** Class<T> **getObjectType**() {

**return** **this**.mapperInterface;

}

}

getObject 方法中调用了 getSqlSession().getMapper(this.mapperInterface); 继续跟进源码 DefaultSqlSession 然后继续跟进到 MapperProxyFactory中newInstance 方法，返回 MapperProxy，MapperProxy 实现了 InvocationHandler 接口即 JDK 动态代理对象。 整理出的 getObject 调用方法时序图如下：



至此，MyBatis 接口扫描流程整理完毕了。

MyBatis 自动扫描接口的原理可总结为：

* 扫描指定 @MapperScan 中 basePackages 指定包下的接口；
* 根据上面的扫描结果，在上下文中注入 BeanDefinition；
* 通过 JDK 动态代理方式实现接口方法。

添加自己的 log 框架（EnableMyLogAutoConfiguration）

以上通过 MyBatis 示例说明了 EnableXX 框架的原理和流程，下面根据步骤来实现我们自己的框架。

这里使用简单的例子说明，扫描指定包下添加了 @MyLogScanner 注解的接口，对接口中添加了 @MyLog 注解的方法进行代理，并且将方法中的第一个参数作为 log 部分，然后输出对应的 log 到控制台。

@MyLogScanner 标注此注解表示是要使用代理：

@Documented

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

**public** @interface MyLogScanner {

}

@MyLog 定义方法中输入日志的 log 名称：

@Target(ElementType.METHOD)

@Documented

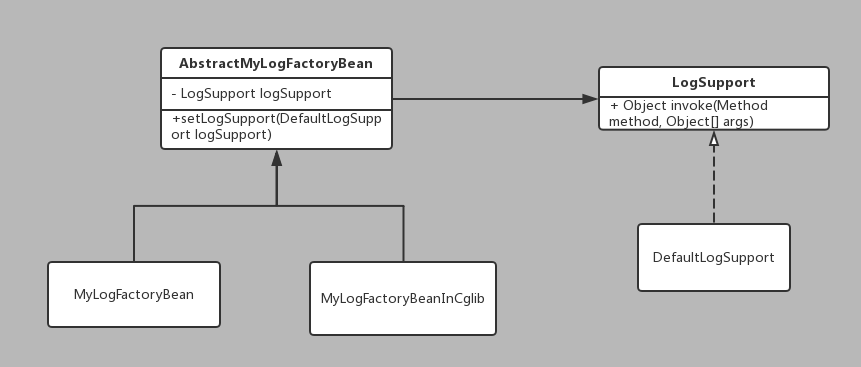
@Retention(RetentionPolicy.RUNTIME)

**public** @interface MyLog {

String **log**() **default** "";

}

log 实现类的 UML 图，其中 MyLogFactoryBean 为使用 JDK 动态代理实现类，MyLogFactoryBeanInCglib 使用的是 cglib 实现的代理类，它们都只是负责方法代理，真正实现 log 功能的是 LogSuport。



第一步，首先添加自己的入口注解 EnableMyLogAutoConfiguration，这里 basePackage 定义的扫描的包，implClass 定义了接口的代理类。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Import(MyLogAutofigurationRegistrar.class)

**public** @interface EnableMyLogAutoConfiguration {

String **basePackage**() **default** "";

*//设置代理类的实现，默认为使用JDK动态代理*

Class<? extends AbstractMyLogFactoryBean> implClass() **default** MyLogFactoryBean.class;

}

第二步，实现自己的 ImportBeanDefinitionRegistrar，对添加了 @MyLogScanner 注解的接口，使用 MyLogClasspathScanner 进行扫描。

**public** **class** MyLogAutofigurationRegistrar implements ImportBeanDefinitionRegistrar, ResourceLoaderAware, EnvironmentAware {

**private** **static** Logger **log** = LoggerFactory.getLogger(MyLogAutofigurationRegistrar.**class**);

**private** Environment env;

**private** ResourceLoader resourceLoader;

@Override

**public** **void** **setEnvironment**(Environment environment) {

**this**.env = environment;

**if** (**log**.isDebugEnabled()) {

**log**.debug("setEnvironment");

}

}

@Override

**public** **void** **setResourceLoader**(ResourceLoader resourceLoader) {

**this**.resourceLoader = resourceLoader;

**if** (**log**.isDebugEnabled()) {

**log**.debug("setResourceLoader");

}

}

@Override

**public** **void** **registerBeanDefinitions**(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

**if** (**log**.isDebugEnabled()) {

**log**.debug("registerBeanDefinitions");

}

*// 获取自动log配置的basePackage*

AnnotationAttributes annoAttrs = AnnotationAttributes

.fromMap(importingClassMetadata.getAnnotationAttributes(EnableMyLogAutoConfiguration.**class**.getName()));

String basePackage = annoAttrs.getString("basePackage");

Class<? extends AbstractFactoryBean> implClass=annoAttrs.getClass("implClass");

*// 扫描基础包下所有带有MyLogScanner注解的接口*

MyLogClasspathScanner scanner = **new** MyLogClasspathScanner(registry);

scanner.setResourceLoader(resourceLoader);

scanner.addIncludeFilter(**new** AnnotationTypeFilter(MyLogScanner.**class**));

*// exclude package-info.java*

scanner.addExcludeFilter(**new** TypeFilter() {

**public** boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {

String className = metadataReader.getClassMetadata().getClassName();

**return** className.endsWith("package-info");

}

});

scanner.setFactoryBeanImplClass(implClass);

scanner.doScan(basePackage);

}

}

步骤3，自定义扫描类 MyLogClasspathScanner，将扫描到的接口使用代理的方式注入到 Spring 上下文中。

**public** **class** **MyLogClasspathScanner** **extends** **ClassPathBeanDefinitionScanner** {

**private** **static** Logger log = LoggerFactory.getLogger(MyLogClasspathScanner.class);

**private** Class<? extends AbstractFactoryBean> factoryBeanImplClass;

**public** **MyLogClasspathScanner**(BeanDefinitionRegistry registry) {

**super**(registry);

}

@Override

**public** Set<BeanDefinitionHolder> **doScan**(String... basePackages) {

Set<BeanDefinitionHolder> beanDefinitions = **super**.doScan(basePackages);

log.info("find beanDefinitions :" + beanDefinitions);

**if** (!beanDefinitions.isEmpty()) {

**for** (BeanDefinitionHolder holder : beanDefinitions) {

GenericBeanDefinition defination = (GenericBeanDefinition) holder.getBeanDefinition();

defination.getPropertyValues().addPropertyValue("mapperInterface", defination.getBeanClassName());

*// 设置bean工厂和对应的属性值*

defination.setBeanClass(**this**.factoryBeanImplClass);

defination.setAutowireMode(AbstractBeanDefinition.AUTOWIRE\_BY\_TYPE);

}

}

**return** beanDefinitions;

}

*/\*\**

*\* {@inheritDoc}*

*\*/*

@Override

**protected** **boolean** **isCandidateComponent**(AnnotatedBeanDefinition beanDefinition) {

**return** (beanDefinition.getMetadata().isInterface() && beanDefinition.getMetadata().isIndependent());

}

*/\*\**

*\* {@inheritDoc}*

*\*/*

@Override

**protected** **boolean** **checkCandidate**(String beanName, BeanDefinition beanDefinition) **throws** IllegalStateException {

**if** (**super**.checkCandidate(beanName, beanDefinition)) {

**return** **true**;

} **else** {

logger.warn("Skipping MapperFactoryBean with name '" + beanName + "' and '" + beanDefinition.getBeanClassName() + "' mapperInterface"

+ ". Bean already defined with the same name!");

**return** **false**;

}

}

**public** **void** **setFactoryBeanImplClass**(Class<? extends AbstractFactoryBean> factoryBeanImplClass) {

**this**.factoryBeanImplClass = factoryBeanImplClass;

}

}

步骤4，对接口的代理，这里只是简单的代理了接口中的方法，具体实现是通过自定义类 LogSupport 来实现的，目前只是简单的返回一个字符串和在控制台输出 log。通过 JDK 动态代理实现：

*/\*\**

*\* 使用jdk自带的动态代理方式处理*

*\**

*\* @author cml*

*\**

*\* @param <T>*

*\*/*

**public** **class** **MyLogFactoryBean**<**T**> **extends** **AbstractMyLogFactoryBean**<**T**> {

@SuppressWarnings("unchecked")

@Override

**public** **void** **afterPropertiesSet**() **throws** Exception {

T target = (T) Proxy.newProxyInstance(getClass().getClassLoader(), **new** Class[] { getMapperInterface() }, **new** InvocationHandler() {

@Override

**public** Object **invoke**(Object proxy, Method method, Object[] args) **throws** Throwable {

**return** "JDK:"+getLogSupport().invoke(method, args);

}

});

setProxyTarget(target);

}

}

使用 cglib 实现：同 JDK 不同的是这 cglib 使用的是方法拦截的方式。

*/\*\**

*\* 使用Cglib的方式实现代理*

*\**

*\* @author cml*

*\**

*\* @param <T>*

*\*/*

**public** **class** **MyLogFactoryBeanInCglib**<**T**> **extends** **AbstractMyLogFactoryBean**<**T**> **implements** **MethodInterceptor** {

@SuppressWarnings("unchecked")

@Override

**public** **void** **afterPropertiesSet**() **throws** Exception {

Enhancer enhancer = **new** Enhancer();

enhancer.setCallback(**this**);

enhancer.setSuperclass(getMapperInterface());

setProxyTarget((T) enhancer.create());

}

@Override

**public** Object **intercept**(Object obj, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {

**return** "CGLIB:" + getLogSupport().invoke(method, args);

}

}

*//LogSupport的实现类*

**public** **class** **DefaultLogSupport** **implements** **LogSupport** {

**public** Object **invoke**(Method method, Object[] args) {

MyLog myLog = method.getAnnotation(MyLog.class);

*// 没有数据返回*

**if** (**null** == myLog) {

**return** **null**;

}

String log = myLog.log();

**if** (**null** != args && args.length == 1) {

log += ":" + args[0];

}

System.out.println(log);

**if** (method.getReturnType() == String.class || method.getReturnType() == Object.class) {

**return** log;

}

**return** **null**;

}

}

步骤5，添加测试的实现类，在类上添加 @MyLogScanner 表明这个接口是 MyLog 需要扫描到的接口，通过 @MyLog 注解设置 log 内容：

@MyLogScanner

**public** **interface** **MyLogService** {

@MyLog(log = "logStart")

**public** **void** **logStart**(String log);

@MyLog(log = "logEnd")

**public** String **logEnd**(String log);

}

步骤6，在 Application 入口类中添加注解：

@**EnableMyLogAutoConfiguration**(**basePackage** = "com.cml.learn.starter.service")

步骤7，添加 Controller：

@RequestMapping("/")

@Controller

**public** **class** **HelloWorldController** {

@Autowired

**private** MyLogService logService;

@RequestMapping

@ResponseBody

**public** String **helloWorld**() {

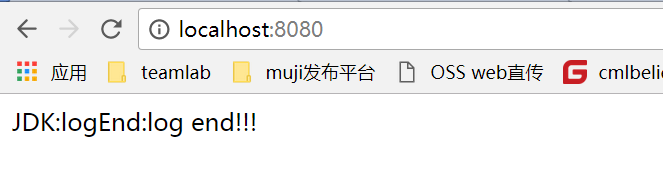
logService.logStart("hello world!!!");

**return** logService.logEnd("log end!!!");

}

}

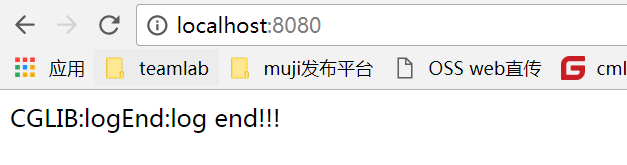
步骤8，启动服务器，访问 localhost:8080 可以看到：



修改成 cglib 的代理方式，注解上修改 implClass 类为 MyLogFactoryBeanInCglib：

@**EnableMyLogAutoConfiguration**(**basePackage** = "com.cml.learn.starter.service", implClass = MyLogFactoryBeanInCglib.class)

访问 localhost:8080：



通过以上步骤就简单地实现了我们自己的 log 自动配置功能。

嫌注解麻烦？添加自动启动功能

以上步骤需要通过添加 @Enabxxx 注解，但是 SpringBoot 整合 Web 环境的时候可是啥都没配的（在《@EnableAutoConfiguration 原理实战》中已经详细说明原理）。作为有追求的开发人员，那么必须来个不需要添加注解的实现方式：

（1）在 src/main/resource/META-INF中 添加 spring.factories 文件：

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.cml.learn.starter.framework.MyLogAutofigurationRegistrar

（2）既然自动配置了，那么就需要能够在配置文件中指定代理类的实现和扫描的包，添加配置文件：

mylog.config.factorybean=com.cml.learn.starter.framework.factorybean.MyLogFactoryBeanInCglib

mylog.config.basepackage=com.cml.learn.starter

（3）MyLogAutofigurationRegistrar 中就需要根据注解还是自动配置来动态实现扫描包和动态代理bean的设置：

@SuppressWarnings({ "rawtypes" })

@Override

**public** **void** **registerBeanDefinitions**(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

**if** (log.isDebugEnabled()) {

log.debug("registerBeanDefinitions");

}

*// 获取自动log配置的basePackage*

AnnotationAttributes annoAttrs = AnnotationAttributes

.fromMap(importingClassMetadata.getAnnotationAttributes(EnableMyLogAutoConfiguration.class.getName()));

*// 获取扫描的basepackage和接口实现类*

String basePackage = retrieveBasePackage(annoAttrs, importingClassMetadata.getClassName());

Class<? extends AbstractFactoryBean> implClass = retrieveFactoryBean(annoAttrs);

*// 扫描基础包下所有带有MyLogScanner注解的接口*

MyLogClasspathScanner scanner = **new** MyLogClasspathScanner(registry);

scanner.setResourceLoader(resourceLoader);

scanner.addIncludeFilter(**new** AnnotationTypeFilter(MyLogScanner.class));

*// exclude package-info.java*

scanner.addExcludeFilter(**new** TypeFilter() {

**public** **boolean** **match**(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) **throws** IOException {

String className = metadataReader.getClassMetadata().getClassName();

**return** className.endsWith("package-info");

}

});

scanner.setFactoryBeanImplClass(implClass);

scanner.doScan(basePackage);

}

*/\*\**

*\* 获取对应的接口实现类*

*\**

*\* @param annoAttrs*

*\* @return*

*\*/*

@SuppressWarnings({ "rawtypes", "unchecked" })

**private** Class<? extends AbstractFactoryBean> retrieveFactoryBean(AnnotationAttributes annoAttrs) {

*// 使用Enable方式注入*

**if** (**null** != annoAttrs) {

**return** annoAttrs.getClass("implClass");

}

*// 使用jar导入的方式注入*

**try** {

*// 通过属性自定义配置*

String factoryBeanName = env.getProperty(FACTORY\_BEAN\_KEY);

**if** (**null** != factoryBeanName) {

**return** (Class<? extends AbstractFactoryBean>) Class.forName(factoryBeanName);

}

**return** (Class<? extends AbstractFactoryBean>) Class.forName(MyLogFactoryBean.class.getName());

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

**throw** **new** IllegalArgumentException("myLogFactoryBean implClass isRequired!!!");

}

*/\*\**

*\* 获取扫描基础包，没有配置的话默认是整个工程*

*\**

*\* @param annoAttrs*

*\* @return*

*\*/*

**private** String **retrieveBasePackage**(AnnotationAttributes annoAttrs, String bootClassName) {

*// 使用Enable方式注入*

**if** (**null** != annoAttrs) {

**return** annoAttrs.getString("basePackage");

}

String configBasePackage = env.getProperty(BASE\_PACKAGE\_KEY);

**if** (**null** != configBasePackage) {

**return** configBasePackage;

}

**try** {

**return** Class.forName(bootClassName).getPackage().getName();

} **catch** (ClassNotFoundException e) {

**throw** **new** IllegalArgumentException("basePackage is Required!!!");

}

}

配置完成后，只需要直接启动项目即可，不需要添加 @EnableMyLogAutoConfiguration 注解了。

总结

作为本系列课程的终极篇，本文详解了 Enable 框架的各种套路实现，读者如果可以根据本系列课程实现自己的 Enable 框架，那么就说明已经完全吸收了本课程的精华了。前面系列课程都是为本课程作为最终铺垫。本系列课程中所讲解的知识在实际项目中还是经常使用到的，不仅会使用而且还掌握了实现原理，如果以后需要类似的需求就可以非常好的自我发挥了，而且对于以后学习其他框架，会有非常大的帮助，掌握了原理就能明白为什么在 pom 中加个 starter 就可以实现功能了，可以非常快速地上手其他框架。 最后，希望读者都能在学习技术的同时，先把原理掌握，这样才能更好的吃透一个技术。