

Spring系列-9 Async注解使用与原理

原创

Ewen Seong

已于 2024-06-02 14:41:57 修改

阅读量4.9k

收藏 9

点赞数 5

分类专栏:

Spring系列

文章标签:

spring

java

mybatis



Spring系列 专栏收录该内容

19 订阅

13 篇文章

背景:

本文作为Spring系列的第九篇，介绍@Async注解的使用、注意事项和实现原理，原理部分会结合 **Spring框架** 代码进行。

本文可以和Spring系列-8 AOP原理进行比较阅读

1.使用方式

@Async一般注解在方法上，用于实现方法的异步：方法调用者立即返回，待调用的方法提交给Spring的线程池执行。@Async也可以注解在类上，等价有方法上添加该注解。需要注意@Async只对Spring管理的对象生效。

案例介绍:

1.1 基本用例

在配置类或者启动类上注解@EnableAsync，用于开启异步功能：

```
1 | @Configuration
2 | @ComponentScan(basePackages = "com.seong.async")
3 | @EnableAsync
4 | public class AsyncConfiguration {
5 | }
```

在方法上添加@Async注解：

```
1 | package com.seong.async;
2 |
3 | @Component
4 | public class ComponentA {
5 |     @Async
6 |     public void print() {
7 |         System.out.println(Thread.currentThread().getName()+":"+ " test Async call.");
8 |     }
9 | }
```

用例如下：

```
1 | @Slf4j
2 | public class AsyncApplication {
3 |     public static void main(String[] args) {
4 |         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(AsyncConfiguration.class);
5 |         //用id取出业务逻辑类的bean
6 |         ComponentA componentA = (ComponentA) applicationContext.getBean("componentA");
7 |         LOGGER.info("test begin...");
8 |         componentA.print();
9 |         LOGGER.info("test end!");
10 |     }
11 | }
```



Ewen Seong

已关注

得到如下结果：

```
16:03:44.489 [main] INFO com.seong.async.AsyncApplication - test begin...
16:03:44.498 [main] DEBUG org.springframework.scheduling.annotation.AnnotationAsyncExecutionInterceptor - Could not find
org.springframework.beans.factory.NoSuchBeanDefinitionException Create breakpoint : No qualifying bean of type 'org.springfr
at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
at org.springframework.aop.interceptor.AsyncExecutionAspectSupport.getDefaultExecutor(AsyncExecutionAspectSupport.java:107)
at org.springframework.aop.interceptor.AsyncExecutionInterceptor.getDefaultExecutor(AsyncExecutionInterceptor.java:107)
at org.springframework.aop.interceptor.AsyncExecutionAspectSupport.lambda$configure$2(AsyncExecutionAspectSupport.java:107)
at org.springframework.util.function.SingletonSupplier.get(SingletonSupplier.java:100)
at org.springframework.aop.interceptor.AsyncExecutionAspectSupport.determineAsyncExecutor(AsyncExecutionAspectSupport.java:107)
at org.springframework.aop.interceptor.AsyncExecutionInterceptor.invoke(AsyncExecutionInterceptor.java:107)
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:753)
at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:698)
at com.seong.async.ComponentA$$EnhancerBySpringCGLIB$$739f4f6b.print(<generated>)
at com.seong.async.AsyncApplication.main(AsyncApplication.java:13)
16:03:44.498 [main] INFO org.springframework.scheduling.annotation.AnnotationAsyncExecutionInterceptor - No task executor
16:03:44.500 [main] INFO com.seong.async.AsyncApplication - test end!
16:03:44.524 [SimpleAsyncTaskExecutor-1] INFO com.seong.async.ComponentA - SimpleAsyncTaskExecutor-1: test Async call.
```

结果显示：ComponentA对象的print()方法被异步执行了，且DEBUG日志抛出了 `No qualifying bean of type 'org.springframework.core.task.TaskExecutor' available` 异常；这是因为业务代码没有配置TaskExecutor类型的Bean对象导致。

1.2 配置线程池

被@Async注解的方法会提交给线程池执行，这里可以手动指定线程池或者使用默认的线程池：

(1) 手动指定线程池：

通过Async的value属性指定线程池Bean对象(通过beanName指定)，如：

```
1 | @Component
2 | public class ComponentA {
3 |     @Async("myTaskExecutor")
4 |     public void print() {
5 |         System.out.println(Thread.currentThread().getName()+" test Async call.");
6 |     }
7 | }
8 |
9 | @Configuration
10 | @EnableAsync
11 | public class AsyncConfiguration {
```

此时，运行任务会提交给"myTaskExecutor"线程池对象执行。

(2) 配置默认的线程池：

```
1 | @Bean
2 | public TaskExecutor myTaskExecutor() {
3 |     return new SimpleAsyncTaskExecutor();
4 | }
```

Spring为@Async提供了默认 **线程池配置**，可通过向IOC中注册TaskExecutor类型的Bean对象实现。也可使用如下配置注册默认线程池：

```
1 | @Bean
2 | public Executor taskExecutor() {
3 |     return Executors.newFixedThreadPool(1);
4 | }
```

Spring框架获取TaskExecutor类型的Bean对象失败时，会尝试获取BeanName为"taskExecutor"的线程池对象；但执行时日志中会给出异常信息。



Ewen Seong

已关注

(3) 使用Spring框架默认的SimpleAsyncTaskExecutor线程池。

若业务未配置默认线程池，默认使用Spring生成的SimpleAsyncTaskExecutor对象；但执行时日志中会给出异常信息。

1.3 获取返回值

由于@Async注解的方法为异步执行，因此可以通过Future来获取返回值，案例如下：

修改ComponentA的方法：

```
1 @Component
2 @Slf4j
3 public class ComponentA {
4     @Async
5     public Future<String> print() {
6         LOGGER.info(Thread.currentThread().getName() + ":" + " test Async call.");
7         return new AsyncResult<>("ComponentA finished.");
8     }
9 }
```

适配修改案例：

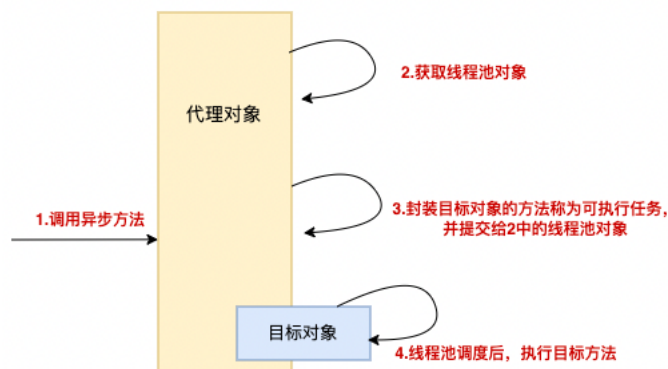
```
1 @Slf4j
2 public class AsyncApplication {
3     public static void main(String[] args) throws ExecutionException, InterruptedException {
4         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(AsyncConfiguration.class);
5         ComponentA componentA = (ComponentA) applicationContext.getBean("componentA");
6
7         LOGGER.info("test begin...");
8         final Future<String> resultFuture = componentA.print();
9         doOtherBusiness();
10        LOGGER.info("test end!");
11    }
```

运行得到如下结果：

```
[main] INFO com.seong.async.AsyncApplication - test begin...
[main] INFO com.seong.async.AsyncApplication - do other business...
[main] INFO com.seong.async.AsyncApplication - test end!
[SimpleAsyncTaskExecutor-1] INFO com.seong.async.ComponentA - SimpleAsyncTaskExecutor-1: test Async call...
[main] INFO com.seong.async.AsyncApplication - result is ComponentA finished.
```

2.原理：

本质上Async注解的实现依赖于动态代理，代理过程中将任务提交给线程池，存在以下流程：



CSDN @Ewen Seong

如上图所示，Spring为使用Async注解Bean对象生产一个动态代理对象，当Async注解的方法被调用时，Spring会生成一个代理对象，该代理对象会拦截所有的方法调用，并将任务提交给线程池执行，返回执行结果。

Ewen Seong 已关注

注意：未被@Async注解的方法则不会执行上述流程(static方法也不会)

由于异步的本质是基于代理实现，所以同一个类中的方法调用会导致被调用方的异步作用失效，该场景与Spring的事务失效原因相同，可参考：[事务-1](#)
[Spring事务传播机制](#)。

2.1 EnableAsync注解：

2.1.1 EnableAsync注解定义

@EnableAsync注解核心作用是向容器中注册AsyncAnnotationBeanPostProcessor对象。

```
1 | @Target(ElementType.TYPE)
2 | @Retention(RetentionPolicy.RUNTIME)
3 | @Documented
4 | @Import(AsyncConfigurationSelector.class)
5 | public @interface EnableAsync {
6 |     Class<? extends Annotation> annotation() default Annotation.class;
7 |
8 |     boolean proxyTargetClass() default false;
9 |
10 |     AdviceMode mode() default AdviceMode.PROXY;
11 |
12 |     int order() default Ordered.LOWEST_PRECEDENCE;
13 | }
```

@EnableAsync注解提过了以下属性用于自定义配置：

[1] annotation属性：

用于指定生效的注解，默认为@Async和EJB的@Asynchronous注解。当指定注解时，仅指定的注解生效：

```
1 | @Target({ElementType.TYPE, ElementType.METHOD})
2 | @Retention(RetentionPolicy.RUNTIME)
3 | @Documented
4 | public @interface TestAnnotation {
5 | }
6 |
7 |
8 | @EnableAsync(annotation = TestAnnotation.class)
9 | public class AsyncConfiguration {
10 | }
11 |
12 | @TestAnnotation
13 | public void print() {
14 |     System.out.println(Thread.currentThread().getName() + ":" + " test Async call");
15 | }
```

但是，仅@Async中可以通过value属性指定需要的线程池Bean对象；因为硬编码获取beanName的逻辑：来源必须为@Async注解的value属性：

```
1 | protected String getExecutorQualifier(Method method) {
2 |     Async async = AnnotatedElementUtils.findMergedAnnotation(method, Async.class);
3 |     if (async == null) {
4 |         async = AnnotatedElementUtils.findMergedAnnotation(method.getDeclaringClass(), Async.class);
5 |     }
6 |     return (async != null ? async.value() : null);
7 | }
```

[2] proxyTargetClass属性：

可用于配置代理类型：true时表示强制使用CGLIB动态代理，false时表示根据被代理类情况进行确定，默认为false。仅当模式设置为AdviceMode#PROXY

[3] mode属性：

指出应该如何应用异步通知，默认值是AdviceMode.PROXY。

[4] order属性：

指示应该Bean对象在BPP阶段应用AsyncAnnotationBeanPostProcessor的顺序，默认值是Ordered.LOWEST_PRECEDENCE，以便在所有其他后处理
这样它就可以向现有代理添加一个advisor而不是双代理。

2.1.2 EnableAsync注解作用

通过@Import注解向IOC中注入了AsyncConfigurationSelector对象，进入AsyncConfigurationSelector



Ewen Seong

已关注

```

1 public final String[] selectImports(AnnotationMetadata importingClassMetadata) {
2     //...
3     AdviceMode adviceMode = attributes.getEnum(getAdviceModeAttributeName());
4     String[] imports = selectImports(adviceMode);
5     //...
6     return imports;
7 }
8
9 public String[] selectImports(AdviceMode adviceMode) {
10     switch (adviceMode) {
11         case PROXY...

```



由于加载的AsyncConfigurationSelector对象为ImportSelector类型，Spring继续完成ProxyAsyncConfiguration类型的注入。进入ProxyAsyncConfiguration代码逻辑，发现是一个配置类，用于向容器中注入一个AsyncAnnotationBeanPostProcessor对象：

```

1 @Configuration
2 @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
3 public class ProxyAsyncConfiguration extends AbstractAsyncConfiguration {
4     @Bean(name="org.springframework.context.annotation.internalAsyncAnnotationProcessor")
5     @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
6     public AsyncAnnotationBeanPostProcessor asyncAdvisor() {
7         AsyncAnnotationBeanPostProcessor bpp = new AsyncAnnotationBeanPostProcessor();
8         bpp.configure(this.executor, this.exceptionHandler);
9         Class<? extends Annotation> customAsyncAnnotation = this.enableAsync.getClass("annotation");
10         if (customAsyncAnnotation != AnnotationUtils.getDefaultValue(EnableAsync.class, "annotation")) {
11             bpp.setAsyncAnnotationType(customAsyncAnnotation);

```



总之，@EnableAsync后IOC容器中会增加一个AsyncAnnotationBeanPostProcessor类型的对象；beanName为"org.springframework.context.annotation.internalAsyncAnnotationProcessor"。

2.2 AsyncAnnotationBeanPostProcessor:

AsyncAnnotationBeanPostProcessor是一个后置处理器且实现了BeanFactoryAware接口，核心逻辑在两处：Aware阶段和BPP-After阶段。

[1] Aware阶段：准备好Advisor对象(内部包含一个拦截器)

```

1 public void setBeanFactory(BeaFactory beanFactory) {
2     super.setBeanFactory(beanFactory);
3
4     // 构造AsyncAnnotationAdvisor对象，注入到this.advisor属性中
5     AsyncAnnotationAdvisor advisor
6         = new AsyncAnnotationAdvisor(this.executor, this.exceptionHandler);
7     advisor.setBeanFactory(beanFactory);
8     this.advisor = advisor;
9 }
10

```

上述方法完成了BeanFactory属性的注入以及this.advisor属性的设置，属性类型为AsyncAnnotationAdvisor，进入AsyncAnnotationAdvisor的构造函数：

```

1 public AsyncAnnotationAdvisor( executor, exceptionHandler) {
2     this.advice = buildAdvice(executor, exceptionHandler);
3     this.pointcut = buildPointcut(asyncAnnotationTypes);
4 }
5
6 protected Advice buildAdvice( executor, exceptionHandler) {
7     AnnotationAsyncExecutionInterceptor interceptor =
8         new AnnotationAsyncExecutionInterceptor(null);
9     // 设置线程池和异常处理器
10    interceptor.configure(executor, exceptionHandler);
11

```



```

12     return interceptor;
13 }
14

```

注意AnnotationAsyncExecutionInterceptor是一个MethodInterceptor接口的实现类：

```

1 // MethodInterceptor拦截器接口的invoke方法
2 public Object invoke(final MethodInvocation invocation) throws Throwable {
3     //...
4 }

```

总之，在aware阶段为AsyncAnnotationBeanPostProcessor对象注入this.advisor一个AsyncAnnotationAdvisor类型的对象，该对象包含BeanFactory和其中advice属性包含一个MethodInterceptor拦截器。

[2] BPP-After阶段：为Bean对象生成代理

在AsyncAnnotationBeanPostProcessor的初始化过程完成后，当IOC初始化单例Bean对象时，会在初始化的后置处理器阶段调用AsyncAnnotationBean的postProcessAfterInitialization方法：

```

1 @Override
2 public Object postProcessAfterInitialization(Object bean, String beanName) {
3     // this.advisor在AsyncAnnotationBeanPostProcessor构建阶段已设值
4     if (this.advisor == null || bean instanceof AopInfrastructureBean) {
5         return bean;
6     }
7     // 是否已进行了AOP代理
8     if (bean instanceof Advised) {
9         Advised advised = (Advised) bean;
10        if (!advised.isFrozen() && isEligible(AopUtils.getTargetClass(bean))) {
11            // Add our local Advisor to the existing proxy's Advisor chain

```

▽

如上述代码所示：

已完成AOP代理的不需要再次代理，将AsyncAnnotationBeanPostProcessor的this.advisor属性添加到代理对象的List<Advisor>列表中；未经过代理的@Async的对象，Spring为其创建一个代理对象，并将this.advisor属性信息保存到代理对象中。【代理过程参考：Spring系列-8 AOP原理】

[3] 接口调用

Spring提供了两种代理方式：JDK动态代理和CGLIB代理，由于底层代理细节对实现原理并无影响，这里以JDK动态代理为例进行介绍。被代理的对象在方法被调用时，进入JdkDynamicAopProxy的invoke拦截方法，注意每个代理对象对应一个JdkDynamicAopProxy对象。

[插图：解释ProxyFactory、ProxyConfig、Advised的关系，以及JdkDynamicAopProxy的传参路径]

涉及代码如下所示(保留主线逻辑)：

```

1 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
2     // hash/equals/toString check...
3
4     TargetSource targetSource = this.advised.targetSource;
5     Object target = targetSource.getTarget();
6     Class<?> targetClass = (target != null ? target.getClass() : null);
7
8     // 根据方法和字节码对象获取调用链，如果该方法未被@Async注解，则返回的集合为空；否则返回一个包含AnnotationAsyncExecutionInterceptor对象的
9     List<Object> chain =
10    this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
11

```

▽

invoke方法的主线逻辑：

- [1] 从targetSource属性中取出被代理的对象；
- [2] 从advised属性中获取拦截器链；
- [3] 判断拦截器链是否为空，为空则直接通过反射调用该方法；否则进入拦截器对象；
- [4] 进行结果类型的校验，并返回结果对象。



Ewen Seong

已关注

ReflectiveMethodInvocation介绍:

ReflectiveMethodInvocation是对调用链进行的一层封装:

```
1 public Object proceed() throws Throwable {
2     // We start with an index of -1 and increment early.
3     if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
4         return invokeJoinpoint();
5     }
6
7     Object interceptorOrInterceptionAdvice =
8         this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
9
10    return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
11 }
12
```

interceptorsAndDynamicMethodMatchers为构建ReflectiveMethodInvocation对象时传入的chain参数; 即依次调用拦截器的拦截invoke方法, 最后执行invokeJoinpoint()方法。

注意: 调用拦截器时将this作为入参(该ReflectiveMethodInvocation对象), 拦截器中可通过调用该对象的proceed()实现回调。

2.3 AnnotationAsyncExecutionInterceptor介绍:

当异步方法被调用时, 堆栈逻辑进入到AnnotationAsyncExecutionInterceptor的invoke方法:

```
1 public Object invoke(final MethodInvocation invocation) throws Throwable {
2     Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);
3     Method specificMethod = ClassUtils.getMostSpecificMethod(invocation.getMethod(), targetClass);
4     final Method userDeclaredMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);
5     // 1. 确定待执行的任务的线程池
6     AsyncTaskExecutor executor = determineAsyncExecutor(userDeclaredMethod);
7
8     // 2. 封装执行逻辑成一个任务
9     Callable<Object> task = () -> {
10         try {
11             // 回调ReflectiveMethodInvocation的proceed方法
12         }
13     }
14 }
```

AnnotationAsyncExecutionInterceptor的invoke方法主线逻辑如下:

- [1] 获取待执行的任务的线程池;
- [2] 回调ReflectiveMethodInvocation的逻辑(将 invocation.proceed()) 封装为一个任务;
- [3] 将任务提交给线程池执行。

当提交给线程池的任务被执行时, 即 invocation.proceed() 执行时进入到 ReflectiveMethodInvocation的invokeJoinpoint()方法, 反射调用目标方法。

获取执行@Async方法的线程池:

```
1 protected AsyncTaskExecutor determineAsyncExecutor(Method method) {
2     AsyncTaskExecutor executor = this.executors.get(method);
3     if (executor == null) {
4         Executor targetExecutor;
5         String qualifier = getExecutorQualifier(method);
6         if (StringUtils.hasLength(qualifier)) {
7             targetExecutor = findQualifiedExecutor(this.beanFactory, qualifier);
8         }
9         else {
10             targetExecutor = this.defaultExecutor.get();
11         }
12     }
13 }
```

其中, this.executors对象用于缓存方法与线程池的关系, 减少查询次数。



Ewen Seong 已关注

主线逻辑为：

如果@Async注解的value有值，则根据该值从beanFactory中获取对应的线程池Bean对象；否则获取默认的线程池，获取步骤如下：

```

1 | protected Executor getDefaultExecutor(@Nullable BeanFactory beanFactory) {
2 |     Executor defaultExecutor = super.getDefaultExecutor(beanFactory);
3 |     return (defaultExecutor != null ? defaultExecutor : new SimpleAsyncTaskExecutor());
4 | }

```

逻辑较为简单：通过父类的getDefaultExecutor方法获取默认的线程池，获取失败则使用SimpleAsyncTaskExecutor线程池对象。

父类getDefaultExecutor方法逻辑如下：

```

1 | protected Executor getDefaultExecutor(@Nullable BeanFactory beanFactory) {
2 |     if (beanFactory != null) {
3 |         try {
4 |             return beanFactory.getBean(TaskExecutor.class);
5 |         } catch (NoUniqueBeanDefinitionException ex) {
6 |             return beanFactory.getBean("taskExecutor", Executor.class);
7 |         } catch (NoSuchBeanDefinitionException ex) {
8 |             return beanFactory.getBean("taskExecutor", Executor.class);
9 |         }
10 |    }
11 |    return null;
12 | }

```

先尝试根据TaskExecutor类型从IOC中获取Bean对象，获取失败再次根据"taskExecutor"名称获取Executor类型的Bean对象，都获取失败时返回null；当getDefaultExecutor方法获取线程池结果为空时，会使用 SimpleAsyncTaskExecutor线程池对象。

注意：

SimpleAsyncTaskExecutor对象每次执行任务都会创建一个新的线程(且没有最大线程数设置)，当并发量较大时可能导致严重的性能问题；建议使用@Async定义beanName为 "taskExecutor"且类型为TaskExecutor的线程池。

2.4 代理类型：

Spring提供了两种动态代理类型：JDK动态代理和CGLIB动态代理(可参考：[AVASE-14 静态代理与动态代理](#))，并支持通过配置对其进行指定。

AopProxy工厂(DefaultAopProxyFactory)创建AopProxy对象的过程如下：

```

1 | @Override
2 | public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
3 |     if (!NativeDetector.inNativeImage() &&
4 |         (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config))) {
5 |         Class<?> targetClass = config.getTargetClass();
6 |
7 |         if (targetClass == null) {
8 |             throw new AopConfigException("TargetSource cannot determine target class: " +
9 |                 "Either an interface or a target is required for proxy creation.");
10 |        }
11 |        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {

```

NativeDetector.inNativeImage() 和 config.isOptimize() 在整个代理和方法调用过程(未涉及配置)取默认值，可直接忽略；且通过 Class<?> targetClass = config.getTargetClass(); 得到的对象为(已被实例化的)Bean对象的字节码类型，不可能为接口类型或空对象，可直接忽略。*[框架代码不同于业务代码，场景的适配]*

因此，上述代码可以简化为：

```

@Override
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
1 |
2 |     if (config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
3 |         Class<?> targetClass = config.getTargetClass();
4 |         if (Proxy.isProxyClass(targetClass)) {
5 |             return new JdkDynamicAopProxy(config);
6 |         }
7 |         return new ObjenesisCglibAopProxy(config);
8 |     } else {

```



Ewen Seong

已关注

9
10
11



结果比较清晰:

- 【1】如果目标对象已被JDK动态代理过，则选择JDK动态代理；
- 【2】如果ProxyTarget属性为true 或者没有实现接口，使用CGLIB代理；否则使用JDK动态代理。

顺便提一下：JDK动态代理生成代理类的速度较快(相对CGLIB快8倍)，但是运行速度较慢(比CGLIB慢10倍)。

文章知识点与官方知识档案匹配，可进一步学习相关知识

Java技能树 注解 基本语法 150170 人正在系统学习中

Spring @Async 使用

湖人

前言：任何与业务逻辑没有直接关联的逻辑（横切关注点）或在调用者上下文中不需要响应以确定下一个流或任何业务的逻辑是Asyncronization的理想候选者。在Spring中使

SpringBoot中@EnableAsync和@Async注解的使用

ACGkaka

SpringBoot中@EnableAsync和@Async注解的使用

SpringBoot中@EnableAsync和@Async注解的使用_enableasync注解-CSDN...

1.@EnableAsync 注解 @EnableAsync 是一个 Spring Boot 中用于启动异步方法调用的注解。使用 @EnableAsync 注解时,需要将其放置在一个配置类上,并且在配置类中通过

异步注解@Async避坑指南_enableasync要加吗

1、启动类要加上@EnableAsync注解,否则异步不生效 2、在任何Service或者Component类的方法上使用@Async注解 直接上代码: packagecom.example.springbootdemo;ir

@EnableAsync & @Async 实现方法异步调用 最新发布

凤

默认情况下,使用内置的线程池来异步调用方法,不过也可以自定义异步执行任务的线程池。在Spring容器中定义一个线程池类型的bean, bean名称必须是/** 为async手动

async注解的使用

weixin_5630777

启动类增加注解@EnableAsync @EnableAsync @SpringBootApplication public class DemoApplication { public static void main(String[] args) { SpringApplication.run(Demo/

Spring的@EnableAsync与@Async使用详解

文章浏览阅读7.2k次,点赞3次,收藏21次。@EnableAsync的javadoc@EnableAsync可以让Spring启用异步方法执行,就跟在xml中配置task:* 效果是一样的。它可以跟@Config

springboot和spring使用@Async注意事项_enableasync注解 不加会有什么...

启动类上使用@EnableAsync注解 @SpringBootApplication@EnableAsyncpublicclassMainApplication{publicstaticvoidmain(String[]args){SpringApplication.run(MainApplicati

Spring中@Async注解的使用

Java

通常,在Java中的方法调用都是同步调用,比如在A方法中调用了B方法,则在A调用B方法之后,必须等待B方法执行并返回后, A方法才可以继续往下执行。这样容易出现

Spring使用@Async注解

本文讲述@Async注解,在Spring体系中的应用。本文仅说明@Async注解的应用规则,对于原理,调用逻辑,源码分析,暂不介绍。对于异步方法调用,从Spring3开始提供

Spring异步注解@Async,@EnableAsync底层源码分析_spring异步执行注解...

首先从开启异步注解@EnableAsync分析,进入这个注解的内部实现,发现它通过@Import注解导入了一个类:AsyncConfigurationSelector: @Target(ElementType.TYPE) @Rete

@EnableAsync的使用、进阶、源码分析

@EnableAsync使用 基础使用 使用@EnableAsync开启异步切面,然后在异步调用的方法上加上@Async注解即可 @SpringBootApplication@EnableAsync//开启异步切面public

Async注解使用及源码分析

云原生Java Web领域技

在实际开发中,有时需要执行某个方法但不需等待该方法的执行结果或者需要执行多个方法但这些方法不需要先后执行。针对上述场景,可以通过声明并调用异步方法实现。

Spring源码之Async注解 热门推荐

何

spring @Async 注解的处理

Spring之@Async注解

@EnableAsync 注解使用 @Import 注解,注入的类型为AsyncConfigurationSelector 2.1 @Import 注解 @Import 注解注入的类一般有以下特征: 2.2 AsyncConfigurationSelecto

...springboot@Async+@EnableAsync两步开启多线程,常见的多线程...

4.1 忘记写再启动类或者配置类上增加 @EnableAsync 开启异步功能 4.2 异步方法所在的类没有被spring管理 4.3 异步方法和调用方法在同一个类中 4.4 调用的方法是异步,但

Spring中异步注解@Async的使用、原理及使用可能导致的的问题及解决方法

如果不了解 @Async 注解的使用原理和机制,可能会导致一些不可预知的问题。 @Async 注解的基本使用 -----

Spring中@Async注解执行异步任务的方法



Ewen Seong

已关注

在Spring中，@Async注解可以与TaskExecutor结合使用。TaskExecutor是Spring提供的一个接口，负责执行异步任务。使用@Async注解的方法会被TaskExecutor执行，而不会立即返回。

SpringBoot下@EnableAsync与@Async异步任务的使用_enableasync 必须引入...
转载自:SpringBoot下@EnableAsync与@Async异步任务的使用 (qq.com) 一、前言 我们在使用多线程的时候,往往需要创建Thread类,或者实现Runnable接口,如果要使用到线程池,则需要使用ThreadPoolExecutor。

关于Spring注解@Async引发其他注解失效的解决
Spring @Async 注解引发其他注解失效的解决 Spring 框架提供了多种注解来帮助开发者简化代码，例如 @Async 用于异步执行方法、@Transaction 用于事务管理等。但是，使用@Async注解可能会导致其他注解失效，这是因为@Async注解会将方法交给Spring的TaskExecutor来执行，而事务管理等注解通常依赖于方法的同步执行。

Spring中@Async注解实现异步调详解
同时，@Async注解也可以与其他Spring的机制结合使用，例如使用Spring的AOP机制来实现异步调用。@Async注解是Spring框架中一个非常重要的机制，可以极大地提高程序的并发性和响应速度。

Spring 源码学习 - @Async注解实现原理
> 本文作者：geek，一个聪明好学的朋友 ## 1. 简介 开发中我们需要异步执行某个耗时任务时候需要@Async，以下我将从源码角度解释该注解的实现原理。## 2.前提条件 @Async注解的实现原理依赖于Spring的TaskExecutor接口和AsyncTaskHandler类。

spring 注解@Async的使用 flymoringbi
开启异步 @SpringBootApplication @EnableAsync //开启异步 public class RenrenApplication { public static void main(String[] args) { SpringApplication.run(RenrenApplication.class, args); }

Spring之@Async异步注解 sun13491
1.注解介绍 @Async注解，该注解可以被标注在方法上，以便异步地调用该方法。调用者将在调用时立即返回，方法的实际执行将提交给Spring TaskExecutor的任务中，由该任务来执行。

@Async注解的使用 weixin_43167662
异步

Spring @Async 注解的使用 基:
Spring @Async 注解的使用 Spring中用**@Async**注解标记的方法，称为异步方法，它会在调用方的当前线程之外的独立的线程中执行。调用者将在调用时立即返回，方法的实际执行由Spring的TaskExecutor来处理。

【异步任务】@Async注解使用方法及注解失效解决办法 God_WZH
SpringBoot框架下的@Async使用方及注解失效的可能问题和解决方法

@async注解的使用 persistence_PS
异步调用 1.使用： springboot中的启动类中需要添加注解@EnableAsync来开启异步调用，在需要异步执行的方法上添加@Async("taskExecutor")注解进行标注。@Target({ElementType.METHOD})

@Async注解使用步骤
@Async是Spring框架提供的一个注解，用于标注一个方法是异步执行的。使用@Async注解的方法会在调用时立即返回，而不会等待其执行完成。下面是@Async注解的使用步骤：1. 在需要异步执行的方法上添加@Async注解。2. 确保Spring容器中已经配置了TaskExecutor。

关于我们 招贤纳士 商务合作 寻求报道 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00
公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心
家长监护 网络110报警服务 中国互联网举报中心 Chrome商店下载 账号管理规范 版权与免责声明 版权申诉 出版物许可证 营业执照
©1999-2024北京创新乐知网络技术有限公司



Ewen Seong

码龄6年 暂无认证

84

原创

6986

周排名

1万+

总排名

17万+

访问



等级

1708

积分

2568

粉丝

812

获赞

31

评论

829

收藏





























私信

已关注

AI圈早知道，每日最新动态


了解全球AI新鲜事！

立即参与

大额流量券免费送

发布一篇就可获得！

去查看



Ewen Seong

已关注

搜博主文章



热门文章

- Spring系列-6 占位符使用和原理 4899
- Spring系列-9 Async注解使用与原理 4900
- Spring系列-1 启动流程 4643
- 事务-2 Spring与Mybatis事务实现原理 4281
- SpringMVC系列-1 使用方式和启动流程 4232

分类专栏

	netty	1篇
	工具类	6篇
	笔记	3篇
	前端	9篇
	Nginx系列	12篇
	三方件	7篇



最新评论

- 前端系列-7 Vue3响应式数据
- 全栈小5: 文章写的很详细，条理清晰，很容易看进去，学到了很多知识，感谢博主！ ...
- 前端系列-7 Vue3响应式数据
- ha_lydms: 非常不错的技术领域文章分享，解决了我在实践中的大问题！ 博主很有 ...
- 多线程系列-2 线程中断机制
- Ewen Seong: 可以结合"多线程系列-1 线程的状态"理解线程中断的概念
- Nginx系列-7 upstream与负载均衡
- 阿登_: 描述得很详细 很到位👍
- Lua使用方式介绍
- Ewen Seong: lua官网地址: <https://www.lua.org/>

最新文章

- Netty系列-1 NioEventLoopGroup和NioEventLoop介绍
- LocalDateTime的序列化和反序列化
- 前端系列-9 Vue3生命周期和computed和watch

2024年 36篇	2023年 18篇
2022年 17篇	2021年 13篇



Ewen Seong

已关注



免费创建你的网站

支持多语言,社媒整合,SEO友好,移动友好,自定义域名,自定义表单等功能。专业级跨境SaaS建站系统。

Strikingly.com

打开

目录

背景:

1.使用方式

案例介绍:

- 1.1 基本用例
- 1.2 配置线程池
- 1.3 获取返回值

2.原理:

2.1 EnableAsync注解:

- 2.1.1 EnableAsync注解定义
- 2.1.2 EnableAsync注解作用

2.2 AsyncAnnotationBeanPostProce...

2.3 AnnotationAsyncExecutionInterc...

2.4 代理类型:



Ewen Seong

已关注