



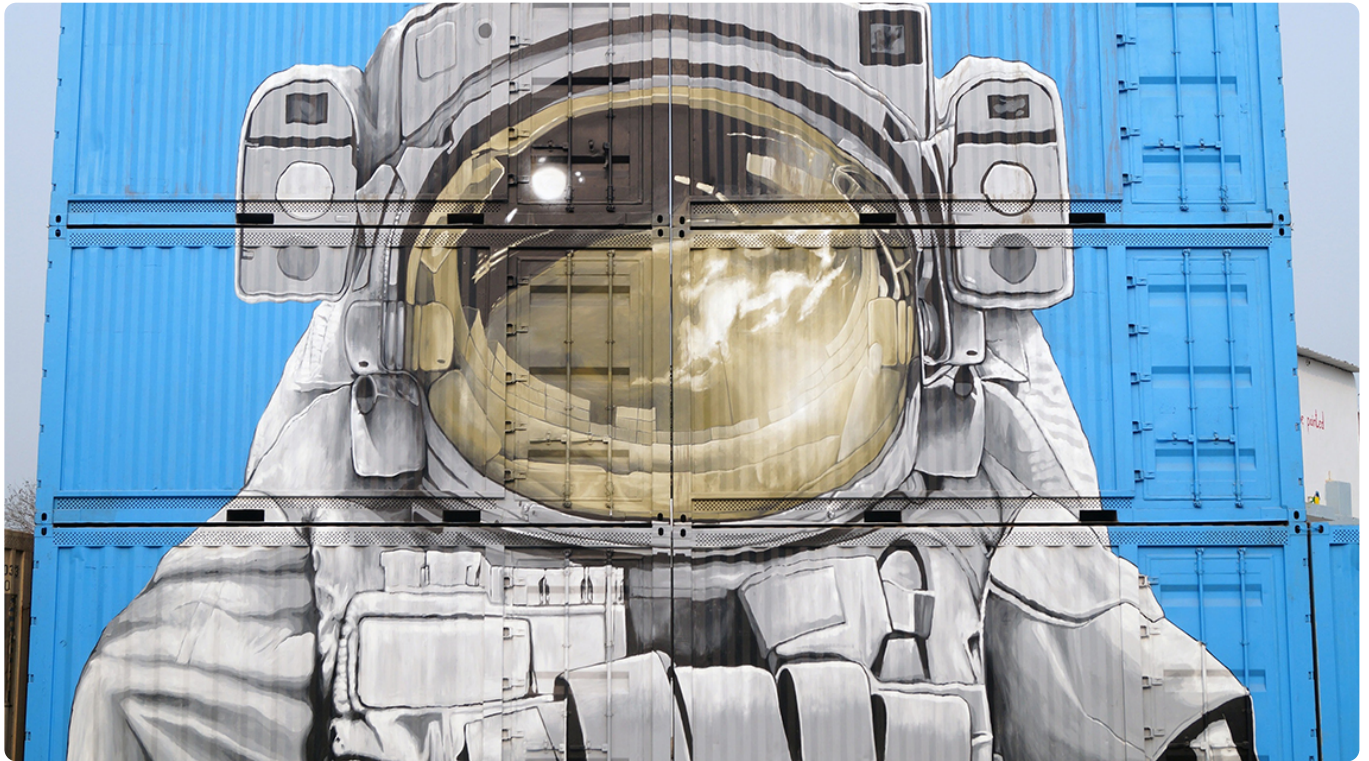
下载APP



08 | 答疑现场：Spring Core 篇思考题合集

2021-05-07 傅健

Spring编程常见错误50例

[进入课程 >](#)**讲述：傅健**

时长 01:40 大小 1.54M



你好，我是傅健。

如果你看到这篇文章，那么我真的非常开心，这说明第一章的内容你都跟下来了，并且对于课后的思考题也有研究，在这我要手动给你点个赞。繁忙的工作中，还能为自己持续充电，保持终身学习的心态，我想我们一定是同路人。

那么到今天为止，我们已经学习了 17 个案例，解决的问题也不算少了，不知道你的感受如何？收获如何呢？



我还记得 [开篇词](#) 的留言区中有一位很有趣的同学，他说：“作为一线 bug 制造者，希望能少写点 bug。” 感同身受，和 Spring 斗智斗勇的这些年，我也经常为一些问题而抓狂

过，因不能及时解决而焦虑过，但最终还是觉得蛮有趣的，这个专栏也算是沉淀之作，希望能给你带来一些实际的帮助。

最初，我其实是想每节课都和你交流下上节课的思考题，但又担心大家的学习进度不一样，所以就有了这次的集中答疑，我把我的答案给到大家，你也可以对照着去看一看，也许有更好的方法，欢迎你来贡献“选项”，我们一起交流。希望大家都能在问题的解决中获得一些正向反馈，完成学习闭环。

🔗 第 1 课

在案例 2 中，显示定义构造器，这会发生根据构造器参数寻找对应 Bean 的行为。这里请你思考一个问题，假设寻找不到对应的 Bean，一定会如案例 2 那样直接报错么？

实际上，答案是否定的。这里我们不妨修改下案例 2 的代码，修改后如下：

[📄 复制代码](#)

```
1 @Service
2 public class ServiceImpl {
3     private List<String> serviceNames;
4     public ServiceImpl(List<String> serviceNames){
5         this.serviceNames = serviceNames;
6         System.out.println(this.serviceNames);
7     }
8 }
```

参考上述代码，我们的构造器参数由普通的 String 改成了一个 List，最终运行程序会发现这并不会报错，而是输出 []。

要了解这个现象，我们可以直接定位构建构造器调用参数的代码所在地（即 ConstructorResolver#resolveAutowiredArgument）：

[📄 复制代码](#)

```
1 @Nullable
2 protected Object resolveAutowiredArgument(MethodParameter param, String beanNa
3     @Nullable Set<String> autowiredBeanNames, TypeConverter typeConverter, b
4
5     //省略非关键代码
6     try {
7         //根据构造器参数寻找 bean
```

```
8         return this.beanFactory.resolveDependency(  
9             new DependencyDescriptor(param, true), beanName, autowiredBeanName  
10        )  
11    catch (NoUniqueBeanDefinitionException ex) {  
12        throw ex;  
13    }  
14    catch (NoSuchBeanDefinitionException ex) {  
15        //找不到 “bean” 进行fallback  
16        if (fallback) {  
17            // Single constructor or factory method -> let's return an empty arra  
18            // for e.g. a vararg or a non-null List/Set/Map parameter.  
19            if (paramType.isArray()) {  
20                return Array.newInstance(paramType.getComponentType(), 0);  
21            }  
22            else if (CollectionFactory.isApproximableCollectionType(paramType)) {  
23                return CollectionFactory.createCollection(paramType, 0);  
24            }  
25            else if (CollectionFactory.isApproximableMapType(paramType)) {  
26                return CollectionFactory.createMap(paramType, 0);  
27            }  
28        }  
29        throw ex;  
30    }  
31 }
```

当构建集合类型的参数实例寻找不到合适的 Bean 时，并不是不管不顾地直接报错，而是会尝试进行 fallback。对于本案例而言，会使用下面的语句来创建一个空的集合作为构造器参数传递进去：

```
CollectionFactory.createCollection(paramType, 0);
```


上述代码最终调用代码如下：

```
return new ArrayList<>(capacity);
```

所以很明显，最终修改后的案例并不会报错，而是把 serviceNames 设置为一个空的 List。从这一点也可知，**自动装配远比想象的要复杂。**

🔗 第 2 课

我们知道了通过 @Qualifier 可以引用想匹配的 Bean，也可以直接命名属性的名称为 Bean 的名称来引用，这两种方式如下：

 复制代码

```
1 //方式1: 属性命名为要装配的bean名称
2 @Autowired
3 DataService oracleDataService;
4
5 //方式2: 使用@Qualifier直接引用
6 @Autowired
7 @Qualifier("oracleDataService")
8 DataService dataService;
```

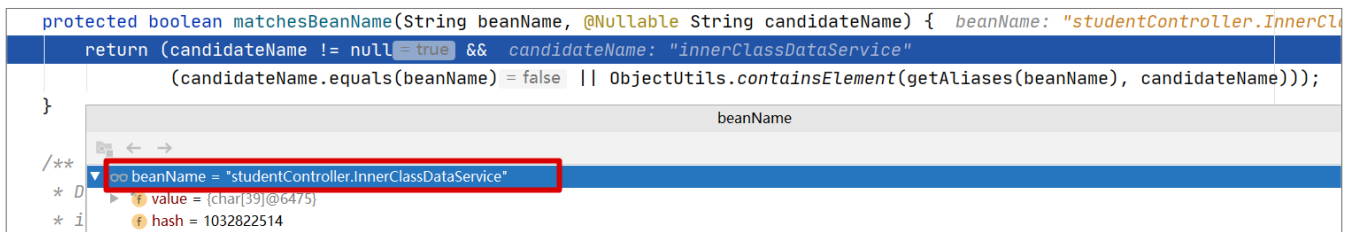
那么对于案例 3 的内部类引用, 你觉得可以使用第 1 种方式做到么? 例如使用如下代码:

@Autowired

DataService studentController.InnerClassDataService;

实际上, 如果你动手或者我们稍微敏锐点就会发现, 代码本身就不能编译, 因为中间含有 "."。那么还有办法能通过这种方式引用到内部类么?

查看决策谁优先的源码, 最终使用属性名来匹配的执行情况可参考
DefaultListableBeanFactory#matchesBeanName 方法的调试视图:



我们可以看到实现的关键其实是下面这行语句:

```
candidateName.equals(beanName) ||
ObjectUtils.containsElement(getAliases(beanName), candidateName))
```

很明显, 我们的 Bean 没有被赋予别名, 而鉴于属性名不可能含有 ".", 所以它不可能匹配上带 "." 的 Bean 名 (即 studentController.InnerClassDataService)。

综上, 如果一个内部类, 没有显式指定名称或者别名, 试图使用属性名和 Bean 名称一致来引用到对应的 Bean 是行不通的。

第 3 课

在案例 2 中，我们初次运行程序获取的结果如下：

```
[Student(id=1, name=xie), Student(id=2, name=fang)]
```

那么如何做到让学生 2 优先输出呢？

实际上，在案例 2 中，我们收集的目标类型是 List，而 List 是可排序的，那么到底是如何排序的？在案例 2 的解析中，我们给出了

DefaultListableBeanFactory#resolveMultipleBeans 方法的代码，不过省略了一些非关键的代码，这其中就包括了排序工作，代码如下：

[复制代码](#)

```
1 if (result instanceof List) {
2     Comparator<Object> comparator = adaptDependencyComparator(matchingBeans);
3     if (comparator != null) {
4         ((List<?>) result).sort(comparator);
5     }
6 }
```

而针对本案例最终排序执行的是 OrderComparator#doCompare 方法，关键代码如下：

[复制代码](#)

```
1 private int doCompare(@Nullable Object o1, @Nullable Object o2, @Nullable Order
2     boolean p1 = (o1 instanceof PriorityOrdered);
3     boolean p2 = (o2 instanceof PriorityOrdered);
4     if (p1 && !p2) {
5         return -1;
6     }
7     else if (p2 && !p1) {
8         return 1;
9     }
10
11     int i1 = getOrder(o1, sourceProvider);
12     int i2 = getOrder(o2, sourceProvider);
13     return Integer.compare(i1, i2);
14 }
```


其中 `getOrder` 的执行, 获取到的 `order` 值 (相当于优先级) 是通过 `AnnotationAwareOrderComparator#findOrder` 来获取的:

[复制代码](#)

```
1 protected Integer findOrder(Object obj) {
2     Integer order = super.findOrder(obj);
3     if (order != null) {
4         return order;
5     }
6     return findOrderFromAnnotation(obj);
7 }
```

不难看出, 获取 `order` 值包含了 2 种方式:

1. 从 `@Order` 获取值, 参考

`AnnotationAwareOrderComparator#findOrderFromAnnotation`:

[复制代码](#)


```
1 @Nullable
2 private Integer findOrderFromAnnotation(Object obj) {
3     AnnotatedElement element = (obj instanceof AnnotatedElement ? (AnnotatedEle
4     MergedAnnotations annotations = MergedAnnotations.from(element, SearchStrat
5     Integer order = OrderUtils.getOrderFromAnnotations(element, annotations);
6     if (order == null && obj instanceof DecoratingProxy) {
7         return findOrderFromAnnotation(((DecoratingProxy) obj).getDecoratedClass
8     }
9     return order;
10 }
```

2. 从 `Ordered` 接口实现方法获取值, 参考 `OrderComparator#findOrder`:

[复制代码](#)

```
1 protected Integer findOrder(Object obj) {
2     return (obj instanceof Ordered ? ((Ordered) obj).getOrder() : null);
3 }
```

通过上面的分析, 如果我们不能改变类继承关系 (例如让 `Student` 实现 `Ordered` 接口), 则可以通过使用 `@Order` 来调整顺序, 具体修改代码如下:

 复制代码


```
1 @Bean
2 @Order(2)
3 public Student student1(){
4     return createStudent(1, "xie");
5 }
6
7 @Bean
8 @Order(1)
9 public Student student2(){
10     return createStudent(2, "fang");
11 }
```

现在，我们就可以把原先的 Bean 输出顺序颠倒过来了，示例如下：

```
Student(id=2, name=fang)][Student(id=1, name=xie)
```

🔗 第 4 课

案例 2 中的类 LightService，当我们不在 Configuration 注解类中使用 Bean 方法将其注入 Spring 容器，而是坚持使用 @Service 将其自动注入到容器，同时实现 Closeable 接口，代码如下：

 复制代码

```
1 import org.springframework.stereotype.Component;
2 import java.io.Closeable;
3 @Service
4 public class LightService implements Closeable {
5     public void close() {
6         System.out.println("turn off all lights");
7     }
8     //省略非关键代码
9 }
```

接口方法 close() 也会在 Spring 容器被销毁的时候自动执行么？

答案是肯定的，通过案例 2 的分析，你可以知道，当 LightService 是一个实现了 Closable 接口的单例 Bean 时，会有一个 DisposableBeanAdapter 被添加进去。

而具体到执行哪一种方法? shutdown()? close()? 在代码中你能够找到答案, 在 DisposableBeanAdapter 类的 inferDestroyMethodIfNecessary 中, 我们可以看到有两种情况会获取到当前 Bean 类中的 close()。

第一种情况, 就是我们这节课提到的当使用 @Bean 且使用默认的 destroyMethod 属性 (INFER_METHOD) ; 第二种情况, 是判断当前类是否实现了 AutoCloseable 接口, 如果实现了, 那么一定会获取此类的 close()。

[复制代码](#)

```
1 private String inferDestroyMethodIfNecessary(Object bean, RootBeanDefinition b
2     String destroyMethodName = beanDefinition.getDestroyMethodName();
3     if (AbstractBeanDefinition.INFER_METHOD.equals(destroyMethodName) || (destro
4         if (!(bean instanceof DisposableBean)) {
5             try {
6                 return bean.getClass().getMethod(CLOSE_METHOD_NAME).getName();
7             }
8             catch (NoSuchMethodException ex) {
9                 try {
10                     return bean.getClass().getMethod(SHUTDOWN_METHOD_NAME).getName(
11                 }
12                 catch (NoSuchMethodException ex2) {
13                     // no candidate destroy method found
14                 }
15             }
16         }
17         return null;
18     }
19     return (StringUtils.hasLength(destroyMethodName) ? destroyMethodName : null
20 }
```

到这, 相信你应该可以结合 Closable 接口和 @Service (或其他 @Component) 让关闭方法得到执行了。

🔗 第 5 课

案例 2 中, 我们提到了通过反射来实例化类的三种方式:

java.lang.Class.newInstance()

java.lang.reflect.Constructor.newInstance()

sun.reflect.ReflectionFactory.newConstructorForSerialization().newInstance()

其中第三种方式不会初始化类属性，你能够写一个例子来证明这一点吗？

能证明的例子，代码示例如下：

[复制代码](#)

```
1 import sun.reflect.ReflectionFactory;
2 import java.lang.reflect.Constructor;
3
4 public class TestNewInstanceStyle {
5
6     public static class TestObject{
7         public String name = "fujian";
8     }
9
10    public static void main(String[] args) throws Exception {
11        //ReflectionFactory.newConstructorForSerialization()方式
12        ReflectionFactory reflectionFactory = ReflectionFactory.getReflectionF
13        Constructor constructor = reflectionFactory.newConstructorForSerializa
14        constructor.setAccessible(true);
15        TestObject testObject1 = (TestObject) constructor.newInstance();
16        System.out.println(testObject1.name);
17        //普通方式
18        TestObject testObject2 = new TestObject();
19        System.out.println(testObject2.name);
20    }
21
22 }
```

运行结果如下：

null

fujian

🔗 第 6 课

实际上，审阅这节课两个案例的修正方案，你会发现它们虽然改动很小，但是都还不够优美。那么有没有稍微优美点的替代方案呢？如果有，你知道背后的原理及关键源码吗？顺便你也可以想想，我为什么没有用更优美的方案呢？

我们可以将“未达到执行顺序预期”的增强方法移动到一个独立的切面类，而不同的切面类可以使用 @Order 进行修饰。@Order 的 value 值越低，则执行优先级越高。以案例 2

为例, 可以修改如下:


 复制代码

```
1 @Aspect
2 @Service
3 @Order(1)
4 public class AopConfig1 {
5     @Before("execution(* com.spring.puzzle.class6.example2.ElectricService.cha
6     public void validateAuthority(JoinPoint pjp) throws Throwable {
7         throw new RuntimeException("authority check failed");
8     }
9 }
10
11
12 @Aspect
13 @Service
14 @Order(2)
15 public class AopConfig2 {
16
17     @Before("execution(* com.spring.puzzle.class6.example2.ElectricService.cha
18     public void logBeforeMethod(JoinPoint pjp) throws Throwable {
19         System.out.println("step into ->" + pjp.getSignature());
20     }
21
22 }
```

上述修改的核心就是将原来的 AOP 配置, 切成两个类进行, 并分别使用 @Order 标记下优先级。这样修改后, 当授权失败了, 则不会打印 “step into ->” 相关日志。

为什么这样是可行的呢? 这还得回溯到案例 1, 当时我们提出这样一个结论:

AbstractAdvisorAutoProxyCreator 执行 findEligibleAdvisors (代码如下) 寻找匹配的 Advisors 时, 最终返回的 Advisors 顺序是由两点来决定的: candidateAdvisors 的顺序和 sortAdvisors 执行的排序。

 复制代码

```
1 protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanNa
2     List<Advisor> candidateAdvisors = findCandidateAdvisors();
3     List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors
4     extendAdvisors(eligibleAdvisors);
5     if (!eligibleAdvisors.isEmpty()) {
6         eligibleAdvisors = sortAdvisors(eligibleAdvisors);
7     }
8     return eligibleAdvisors;
9 }
```

当时影响我们案例出错的关键点都是在 candidateAdvisors 的顺序上，所以我们重点介绍了它。而对于 sortAdvisors 执行的排序并没有多少涉及，这里我可以再重点介绍下。

在实现上，sortAdvisors 的执行最终调用的是比较器

AnnotationAwareOrderComparator 类的 compare()，它调用了 getOrder() 的返回值作为排序依据：

[复制代码](#)

```
1 public int compare(@Nullable Object o1, @Nullable Object o2) {
2     return doCompare(o1, o2, null);
3 }
4
5 private int doCompare(@Nullable Object o1, @Nullable Object o2, @Nullable Order
6     boolean p1 = (o1 instanceof PriorityOrdered);
7     boolean p2 = (o2 instanceof PriorityOrdered);
8     if (p1 && !p2) {
9         return -1;
10    }
11    else if (p2 && !p1) {
12        return 1;
13    }
14
15    int i1 = getOrder(o1, sourceProvider);
16    int i2 = getOrder(o2, sourceProvider);
17    return Integer.compare(i1, i2);
18 }
```

继续跟踪 getOrder() 的执行细节，我们会发现对于我们的案例，这个方法会找出配置切面的 Bean 的 Order 值。这里可以参考 BeanFactoryAspectInstanceFactory#getOrder 的调试视图验证这个结论：

```
@Override
public int getOrder() {
    Class<?> type = this.beanFactory.getType(this.name);
    if (type != null) {
        if (Ordered.class.isAssignableFrom(type) + "aopConfig2" beanFactory.isSingleton(this.name))
            return ((Ordered) this.beanFactory.getBean(this.name)).getOrder();
        return OrderUtils.getOrder(type, Ordered.LOWEST_PRECEDENCE);
    }
    return Ordered.LOWEST_PRECEDENCE;
}
```

上述截图中，aopConfig2 即是我们配置切面的 Bean 的名称。这里再顺带提供出调用栈的截图，以便你做进一步研究：

```
✓ "restartedMain"@1,797 in group "main": RUNNING
getOrder:136, BeanFactoryAspectInstanceFactory (org.springframework.aop.aspectj.annotation)
getOrder:95, LazySingletonAspectInstanceFactoryDecorator (org.springframework.aop.aspectj.annotation)
getOrder:181, InstantiationModelAwarePointcutAdvisorImpl (org.springframework.aop.aspectj.annotation)
findOrder:146, OrderComparator (org.springframework.core)
findOrder:64, AnnotationAwareOrderComparator (org.springframework.core.annotation)
getOrder:129, OrderComparator (org.springframework.core)
getOrder:117, OrderComparator (org.springframework.core)
doCompare:86, OrderComparator (org.springframework.core)
compare:73, OrderComparator (org.springframework.core)
compare:81, AspectJPrecedenceComparator (org.springframework.aop.aspectj.autoproxy)
compare:49, AspectJPrecedenceComparator (org.springframework.aop.aspectj.autoproxy)
compareTo:129, AspectJAwareAdvisorAutoProxyCreator$PartiallyComparableAdvisorHolder (org.springframework.aop.aspectj.autoproxy)
addDirectedLinks:71, PartialOrder$SortObject (org.aspectj.util)
addNewPartialComparable:93, PartialOrder (org.aspectj.util)
sort:129, PartialOrder (org.aspectj.util)
sortAdvisors:75, AspectJAwareAdvisorAutoProxyCreator (org.springframework.aop.aspectj.autoproxy)
findEligibleAdvisors:98, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autoproxy)
getAdvicesAndAdvisorsForBean:76, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autoproxy)
wrapIfNecessary:347, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
postProcessAfterInitialization:299, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
applyBeanPostProcessorsAfterInitialization:431, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
initializeBean:1800, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
doCreateBean:595, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
createBean:517, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
```

现在我们就知道了，将不同的增强方法放置到不同的切面配置类中，使用不同的 Order 值来修饰是可以影响顺序的。相反，如果都是在一个配置类中，自然不会影响顺序，所以这也是当初我的方案中没有重点介绍 sortAdvisors 方法的原因，毕竟当时我们给出的案例都只有一个 AOP 配置类。


🔗 第 7 课

在案例 3 中，我们提到默认的事件执行是在同一个线程中执行的，即事件发布者使用的线程。参考如下日志佐证这个结论：

```
2021-03-09 09:10:33.052 INFO 18104 --- [nio-8080-exec-1]
c.s.p.listener.HelloWorldController : start to publish event
2021-03-09 09:10:33.055 INFO 18104 --- [nio-8080-exec-1]
c.s.p.l.example3.MyFirstEventListener :
com.spring.puzzle.class7.example3.MyFirstEventListener@18faf0 received:
com.spring.puzzle.class7.example3.MyEvent[source=df42b08f-8ee2-44df-a957-
d8464ff50c88]
```


通过日志可以看出，事件的发布和执行使用的都是 nio-8080-exec-1 线程，但是在事件比较多时，我们往往希望事件执行得更快些，或者希望事件的执行可以异步化以不影响主线程。此时应该如何做呢？

针对上述问题中的需求，我们只需要对于事件的执行引入线程池即可。我们先来看下 Spring 对这点的支持。实际上，在案例 3 的解析中，我们已贴出了以下代码片段（位于 SimpleApplicationEventMulticaster#multicastEvent 方法中）：

 复制代码

```
1 //省略其他非关键代码
2 //获取 executor
3 Executor executor = getTaskExecutor();
4 for (ApplicationListener<?> listener : getApplicationListeners(event, type)
5     //如果存在 executor, 则提交到 executor 中去执行
6     if (executor != null) {
7         executor.execute(() -> invokeListener(listener, event));
8     }
9 //省略其他非关键代码
```

对于事件的处理，可以绑定一个 Executor 去执行，那么如何绑定？其实与这节课讲过的绑定 ErrorHandler 的方法是类似的。绑定代码示例如下：

 复制代码

```
1 //注意下面的语句只能执行一次，以避免重复创建线程池
2 ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
3 //省略非关键代码
4 SimpleApplicationEventMulticaster simpleApplicationEventMulticaster = applicat
5 simpleApplicationEventMulticaster.setTaskExecutor(newCachedThreadPool );
```

取出 SimpleApplicationEventMulticaster, 然后直接调用相关 set() 设置线程池就可以了。按这种方式修改后的程序, 事件处理的日志如下:

```
2021-03-09 09:25:09.917 INFO 16548 --- [nio-8080-exec-1]
c.s.p.c.HelloWorldController : start to publish event
2021-03-09 09:25:09.920 INFO 16548 --- [pool-1-thread-3]
c.s.p.l.example3.MyFirstEventListener :
com.spring.puzzle.class7.example3.MyFirstEventListener@511056 received:
com.spring.puzzle.class7.example3.MyEvent[source=cbb97bcc-b834-485c-980e-
2e20de56c7e0]
```

可以看出, 事件的发布和处理分属不同的线程了, 分别为 nio-8080-exec-1 和 pool-1-thread-3, 满足了我们的需求。

以上就是这次答疑的全部内容, 我们下一章节再见!

提建议

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 07 | Spring事件常见错误

下一篇 导读 | 5分钟轻松了解一个HTTP请求的处理过程

精选留言 (1)

写留言



哦吼掉了

2021-05-07

第七节的思考题

直接修改SimpleApplicationEventMulticaster这个bean中的属性不是太合适吧 (会不会

影响其他的逻辑,官方文档有相关的描述么)？创建一个子类，然后注入到spring容器里面是不是会好一点？

展开 ✓

