

Spring钩子方法和钩子接口的使用详解



zhrowable 关注

1 2017.05.06 14:03:04 字数 2,122 阅读 7,036



zhrowable

关注

总资产10 (约0.97元)

博客系列-2019年时间轴

阅读 194

通过micrometer实时监控线程池的各项指标

阅读 190

Spring钩子方法和钩子接口的使用详解

[TOC]

前言

SpringFramework其实具有很高的扩展性，只是很少人喜欢挖掘那些扩展点，而且官方的Refrence也很少提到那些Hook类或Hook接口，至于是不是Spring官方有意为之就不得而知。本文浅析一下笔者目前看到的Spring的一些对外开放的扩展点、Hook接口或者Hook类，如果有什么错误，希望多多交流指正，一切以Spring的源码为准，文章编写使用的Spring版本为4.3.8.Release,对应SpringBoot的版本为1.5.3.RELEASE

1、Aware接口族

Spring中提供了各种Aware接口，方便从上下文中获取当前的运行环境，比较常见的几个子接口有：

BeanFactoryAware,BeanNameAware,ApplicationContextAware,EnvironmentAware,BeanClassLoaderAware等，这些Aware的作用都可以从命名得知，并且其使用也是十分简单。

例如我们经常看到SpringContext工具类：

```
1  @Component
2  public final class SpringContextAssisor implements ApplicationContextAware {
3
4      private static ApplicationContext applicationContext;
5
6      @Override
7      public void setApplicationContext(ApplicationContext applicationContext) {
8          SpringContextAssisor.applicationContext = applicationContext;
9      }
10
11     public static Object getBeanDefinition(String name) {
12         return applicationContext.getBean(name);
13     }
14
15     public static <T> T getBeanDefinition(String name, Class<T> clazz) {
16         return applicationContext.getBean(name, clazz);
17     }
18
19 }
```

实现ApplicationContextAware接口可以获取ApplicationContext

又例如想获取到当前的一个Spring Bean的BeanFactory:

```
1 | @Component
2 | public class OneBean implements BeanFactoryAware {
3 |     private BeanFactory beanFactory;
4 |
5 |     @Override
6 |     public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
7 |         this.beanFactory = beanFactory;
8 |     }
9 | }
```

一般来说,拿到的应该是DefaultListableBeanFactory,因为这个BeanFactory是BeanFactory一族的最底层的BeanFactory实现类,拥有所有父BeanFactory的功能。

其他的Aware可以自己尝试下功能。

PS: Aware是可以自定义扩展的,具体可以参考下面这篇的博客:

<http://www.cnblogs.com/RunForLove/p/5828916.html>

2、InitializingBean接口和DisposableBean接口

InitializingBean接口只有一个方法#afterPropertiesSet,作用是:当一个Bean实现InitializingBean, #afterPropertiesSet方法里面可以添加自定义的初始化方法或者做一些资源初始化操作(Invoked by a BeanFactory after it has set all bean properties supplied ==> "当BeanFactory 设置完所有的Bean属性之后才会调用#afterPropertiesSet方法")。

DisposableBean接口只有一个方法#destroy,作用是:当一个单例Bean实现DisposableBean, #destroy可以添加自定义的一些销毁方法或者资源释放操作(Invoked by a BeanFactory on destruction of a singleton ==> "单例销毁时由BeanFactory调用#destroy")

使用例子:

```
1 | @Component
2 | public class ConcreteBean implements InitializingBean, DisposableBean {
3 |
4 |     @Override
5 |     public void destroy() throws Exception {
6 |         System.out.println("释放资源");
7 |     }
8 |
9 |     @Override
10 |    public void afterPropertiesSet() throws Exception {
11 |        System.out.println("初始化资源");
12 |    }
13 | }
```

3、ImportBeanDefinitionRegistrar接口

功能:

先看官方的注释

```

1  /**
2   * Interface to be implemented by types that register additional bean definitions when
3   * processing {@link Configuration} classes. Useful when operating at the bean definition
4   * level (as opposed to {@code @Bean} method/instance level) is desired or necessary.
5   *
6   * <p>Along with {@code @Configuration} and {@link ImportSelector}, classes of this type
7   * may be provided to the {@link Import} annotation (or may also be returned from an
8   * {@code ImportSelector}).
9   *
10  * <p>An {@link ImportBeanDefinitionRegistrar} may implement any of the following
11  * {@link org.springframework.beans.factory.Aware Aware} interfaces, and their respective
12  * methods will be called prior to {@link #registerBeanDefinitions}:
13  * <ul>
14  * <li>{@link org.springframework.context.EnvironmentAware EnvironmentAware}</li>
15  * <li>{@link org.springframework.beans.factory.BeanFactoryAware BeanFactoryAware}</li>
16  * <li>{@link org.springframework.beans.factory.BeanClassLoaderAware BeanClassLoaderAware}</li>
17  * <li>{@link org.springframework.context.ResourceLoaderAware ResourceLoaderAware}</li>
18  * </ul>
19  *
20  * <p>See implementations and associated unit tests for usage examples.

```

翻译一下大概如下：

- 1.当处理Java编程式配置类(使用了@Configuration的类)的时候，
ImportBeanDefinitionRegistrar接口的实现类可以注册额外的bean definitions;
- 2.ImportBeanDefinitionRegistrar接口的实现类必须提供给@Import注解或者是ImportSelector接口返回值
- 3.ImportBeanDefinitionRegistrar接口的实现类可能还会实现下面
org.springframework.beans.factory.Aware接口中的一个或者多个，它们各自的方法优先于
ImportBeanDefinitionRegistrar#registerBeanDefinitions被调用
org.springframework.beans.factory.Aware的部分接口如下：

- org.springframework.context.EnvironmentAware(读取或者修改Environment的变量)
- org.springframework.beans.factory.BeanFactoryAware (获取Bean自身的Bean工厂)
- org.springframework.beans.factory.BeanClassLoaderAware(获取Bean自身的类加载器)
- org.springframework.context.ResourceLoaderAware(获取Bean自身的资源加载器)

个人理解：

- 1.首先需要自定义一个类去实现ImportBeanDefinitionRegistrar接口， #registerBeanDefinitions方法的参数有(使用了@Import的类型)元注解AnnotationMetadata以及
BeanDefinitionRegistry(Bean注册相关方法的提供接口)，通过BeanDefinitionRegistry的方法可以实现BeanDefinition注册、移除等相关操作；
- 2.为了保证1生效，必须定义一个Java配置类(带有注解@Configuration)通过@Import指定1中定义的实现类

一个例子：

目标是通过自定义注解@EnableThrowable里面的targets属性指定需要注册进去Spring容器的class，当注解使用在@Configuration的类上，实现指定class的注册，然后可以使用@Autowired实现自动注入。

定义ImportBeanDefinitionRegistrar的实现类EnableThrowableRegistrar:

```
1 public class EnableThrowableRegistrar implements ImportBeanDefinitionRegistrar, Environm
2
3 @Override
```

简书

首页

下载APP

搜索

Q

Aa

beta

登录

注册

推荐阅读

当@Transactional遇到
@CacheEvict, 你的代码还运行正
阅读 160

Lambda表达式
阅读 1,662

IDEA插件神器之Grep Console
阅读 9,112

idea2020注册码, 亲测可用!
阅读 105,633

mybatis-plus 注解实现多表关联查
询的最佳实践
阅读 4,830

```
7
8 @Override
9 public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
10                                     BeanDefinitionRegistry registry) {
11     Map<String, Object> annotationAttributes
12     = importingClassMetadata.getAnnotationAttributes(EnableThrowable.class, g
13     Class<?>[] targets = (Class<?>[]) annotationAttributes.get("targets");
14     if (null != targets && targets.length > 0) {
15         for (Class<?> target : targets) {
16             BeanDefinition beanDefinition = BeanDefinitionBuilder
17                 .genericBeanDefinition(target)
18                 .getBeanDefinition();
19             registry.registerBeanDefinition(beanDefinition.getBeanClassName(),
20                 beanDefinition);
21         }
22     }
23 }
24 }
```



33赞



赞赏

定义一个注解@EnableThrowable:

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 @Documented
4 @Import(value = {EnableThrowableRegistrar.class})
5 public @interface EnableThrowable {
6
7     Class<?>[] targets() default {};
8 }
```

定义一个Java配置类ConcreteConfiguration:

```
1 @Configuration
2 @EnableThrowable(targets = {ConcreteService.class})
3 public class ConcreteConfiguration {
4
5 }
```

定义一个非Spring管理的Service类ConcreteService:

```
1 public class ConcreteService {
2
3     public void sayHello(){
4         System.out.println("ConcreteService say hello!");
5     }
6 }
```

测试代码:

```
1 @SpringBootTest(classes = Application.class)
```

```
2 | @RunWith(SpringJUnit4ClassRunner.class)
3 | public class ConcreteServiceTest {
4 |
```

写下你的评论...

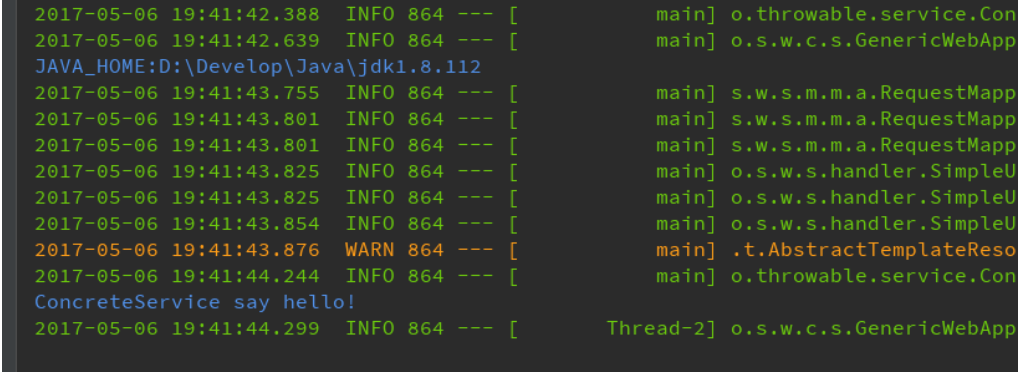
评论1

赞33

...

```
8 |     @Test
9 |     public void sayHello() throws Exception {
10 |         concreteService.sayHello();
11 |     }
12 |
13 | }
```

结果：



```
2017-05-06 19:41:42.388 INFO 864 --- [main] o.throwable.service.Con
2017-05-06 19:41:42.639 INFO 864 --- [main] o.s.w.c.s.GenericWebApp
JAVA_HOME:D:\Develop\Java\jdk1.8.112
2017-05-06 19:41:43.755 INFO 864 --- [main] s.w.s.m.m.a.RequestMapp
2017-05-06 19:41:43.801 INFO 864 --- [main] s.w.s.m.m.a.RequestMapp
2017-05-06 19:41:43.801 INFO 864 --- [main] s.w.s.m.m.a.RequestMapp
2017-05-06 19:41:43.825 INFO 864 --- [main] o.s.w.s.handler.SimpleU
2017-05-06 19:41:43.825 INFO 864 --- [main] o.s.w.s.handler.SimpleU
2017-05-06 19:41:43.854 INFO 864 --- [main] o.s.w.s.handler.SimpleU
2017-05-06 19:41:43.876 WARN 864 --- [main] .t.AbstractTemplateReso
2017-05-06 19:41:44.244 INFO 864 --- [main] o.throwable.service.Con
ConcreteService say hello!
2017-05-06 19:41:44.299 INFO 864 --- [Thread-2] o.s.w.c.s.GenericWebApp
```

01.png

可以看到读取Environment属性成功，同时普通Java类ConcreteService成功注册到Spring容器并且自动注入和调用成功。

4、BeanPostProcessor接口和BeanFactoryPostProcessor接口

一般我们叫这两个接口为Spring的Bean后置处理器接口,作用是为Bean的初始化前后提供可扩展的空间。先看接口的方法：

BeanPostProcessor

```
1 | public interface BeanPostProcessor {
2 |     Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansExce
3 |
4 |     Object postProcessAfterInitialization(Object bean, String beanName) throws BeansExcep
5 | }
```

BeanFactoryPostProcessor

```
1 | public interface BeanFactoryPostProcessor {
2 |     void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansEx
3 | }
```

BeanFactoryPostProcessor可以对bean的定义（配置元数据）进行处理。也就是说，Spring IoC容器允许BeanFactoryPostProcessor在容器实际实例化任何其它的bean之前读取配置元数

据,并有可能修改它。如果你愿意,你可以配置多个BeanFactoryPostProcessor。你还能通过设置'order'属性来控制BeanFactoryPostProcessor的执行次序。(大概可以这样理解:Spring容器加载了bean的定义文件之后,在bean实例化之前执行的)

实现BeanPostProcessor接口可以在Bean(实例化之后)初始化的前后做一些自定义的操作,但是拿到的参数只有BeanDefinition实例和BeanDefinition的名称,也就是无法修改BeanDefinition元数据,这里说的Bean的初始化是:

- 1) bean实现了InitializingBean接口,对应的方法为afterPropertiesSet
- 2) 在bean定义的时候,通过init-method设置的方法

PS:BeanFactoryPostProcessor回调会先于BeanPostProcessor

使用例子:

实现一个BeanPostProcessor==>ConcreteBeanPostProcessor

```

1 | @Order(1)
2 | @Component
3 | public class ConcreteBeanPostProcessor implements BeanPostProcessor {
4 |
5 |     @Override
6 |     public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
7 |         if (beanName.contains("postBean"))
8 |             System.out.println(String.format("Bean初始化之前,bean:%s,beanName:%s", bean.toString(), beanName));
9 |         return bean;
10 |    }
11 |
12 |    @Override
13 |    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
14 |        if (beanName.contains("postBean"))
15 |            System.out.println(String.format("Bean初始化之后,bean:%s,beanName:%s", bean.toString(), beanName));
16 |        return bean;
17 |    }
18 | }

```

实现一个BeanFactoryPostProcessor==>ConcreteBeanFactoryPostProcessor

```

1 | @Component
2 | public class ConcreteBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
3 |
4 |     @Override
5 |     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
6 |         BeanDefinition beanDefinition = beanFactory.getBeanDefinition("postBean");
7 |         MutablePropertyValues propertyValues = beanDefinition.getPropertyValues();
8 |         propertyValues.addPropertyValue("author", "throwable");
9 |     }
10 | }

```

定义一个Spring的Bean

```

1 | @Component
2 | public class PostBean {
3 |
4 |     private String author;
5 |
6 |     public String getAuthor() {
7 |         return author;
8 |     }
9 | }

```

```

9
10     public void setAuthor(String author) {
11         this.author = author;
12     }
13
14     public void sayhello() {
15         System.out.println(String.format("author %s say hello!", author));
16     }
17 }

```

测试类:

```

1  @SpringBootTest(classes = Application.class)
2  @RunWith(SpringJUnit4ClassRunner.class)
3  public class PostBeanTest {
4      @Autowired
5      private PostBean postBean;
6
7      @Test
8      public void sayhello() throws Exception {
9          postBean.sayhello();
10     }
11 }

```

结果:



```

:: Spring Boot ::                (v1.5.3.RELEASE)

2017-05-07 00:15:23.115 INFO 14824 --- [main] org.throwable.processor.P
2017-05-07 00:15:23.116 INFO 14824 --- [main] org.throwable.processor.P
2017-05-07 00:15:23.188 INFO 14824 --- [main] o.s.w.c.s.GenericWebAppli
Bean初始化之前,bean:org.throwable.processor.PostBean@4cc6fa2a,beanName:postBean
Bean初始化之后,bean:org.throwable.processor.PostBean@4cc6fa2a,beanName:postBean
2017-05-07 00:15:24.510 INFO 14824 --- [main] s.w.s.m.m.a.RequestMappin
2017-05-07 00:15:24.570 INFO 14824 --- [main] s.w.s.m.m.a.RequestMappin
2017-05-07 00:15:24.571 INFO 14824 --- [main] s.w.s.m.m.a.RequestMappin
2017-05-07 00:15:24.592 INFO 14824 --- [main] o.s.w.s.handler.SimpleUrl
2017-05-07 00:15:24.592 INFO 14824 --- [main] o.s.w.s.handler.SimpleUrl
2017-05-07 00:15:24.626 INFO 14824 --- [main] o.s.w.s.handler.SimpleUrl
2017-05-07 00:15:24.755 INFO 14824 --- [main] org.throwable.processor.P
author throwable say hello!
2017-05-07 00:15:24.798 INFO 14824 --- [Thread-2] o.s.w.c.s.GenericWebAppli

```

02.png

PS:有兴趣可以看下Spring内置的一些实现了后置处理器接口的类, 大概有下面这些:

```

AnnotationAwareAspectJAutoProxyCreator
AspectJAwareAdvisorAutoProxyCreator
InitDestroyAnnotationBeanPostProcessor
ApplicationContextAwareProcessor
AutowiredAnnotationBeanPostProcessor
CommonAnnotationBeanPostProcessor
RequiredAnnotationBeanPostProcessor
PersistenceAnnotationBeanPostProcessor

```

...

5、BeanDefinitionRegistryPostProcessor 接口

BeanDefinitionRegistryPostProcessor 接口可以看作是BeanFactoryPostProcessor和ImportBeanDefinitionRegistrar的功能集合，既可以获取和修改BeanDefinition的元数据，也可以实现BeanDefinition的注册、移除等操作。

例子：

定义一个

BeanDefinitionRegistryPostProcessor==>ConcreteBeanDefinitionRegistryPostProcessor

```

1 | @Component
2 | public class ConcreteBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistr
3 |
4 |     private static final String beanName = "concreteRPBean";
5 |
6 |     @Override
7 |     public void postProcessBeanDefinitionRegistry(BeaDefinitionRegistry registry) throw
8 |         BeanDefinition beanDefinition = BeanDefinitionBuilder
9 |             .genericBeanDefinition(ConcreteRPBean.class)
10 |             .getBeanDefinition();
11 |     registry.registerBeanDefinition(beanName, beanDefinition);
12 | }
13 |
14 |     @Override
15 |     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throw
16 |         BeanDefinition beanDefinition = beanFactory.getBeanDefinition(beanName);
17 |         MutablePropertyValues propertyValues = beanDefinition.getPropertyValues();
18 |         propertyValues.addPropertyValue("author", "throwable");
19 |     }
20 | }
```

定义一个普通的Java类：

```

1 | public class ConcreteRPBean {
2 |
3 |     private String author;
4 |
5 |     public String getAuthor() {
6 |         return author;
7 |     }
8 |
9 |     public void setAuthor(String author) {
10 |         this.author = author;
11 |     }
12 |
13 |     public void sayHello(){
14 |         System.out.println(String.format("ConcreteRPBean call sayhello method ==> author
15 |     }
16 | }
```

测试类：

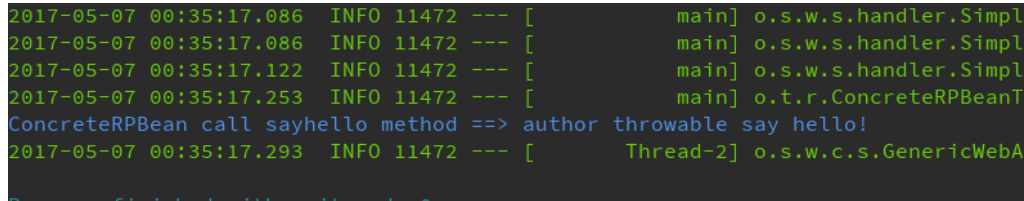
```

1 | @SpringBootTest(classes = Application.class)
2 | @RunWith(SpringJUnit4ClassRunner.class)
3 | public class ConcreteRPBeanTest {
4 |
5 |     @Autowired
```



```
6 | private ConcreteRPBean concreteRPBean;  
7 |  
8 | @Test  
9 | public void sayHello() throws Exception {  
10 |     concreteRPBean.sayHello();  
11 | }  
12 |  
13 | }
```

结果：



```
2017-05-07 00:35:17.086 INFO 11472 --- [main] o.s.w.s.handler.Simpl  
2017-05-07 00:35:17.086 INFO 11472 --- [main] o.s.w.s.handler.Simpl  
2017-05-07 00:35:17.122 INFO 11472 --- [main] o.s.w.s.handler.Simpl  
2017-05-07 00:35:17.253 INFO 11472 --- [main] o.t.r.ConcreteRPBeanT  
ConcreteRPBean call sayhello method ==> author throwable say hello!  
2017-05-07 00:35:17.293 INFO 11472 --- [Thread-2] o.s.w.c.s.GenericWebA
```

03.png

6、FactoryBean接口

首先第一眼要注意，是**FactoryBean**接口而不是**BeanFactory**接口。一般情况下，Spring通过反射机制利用bean的class属性指定实现类来实例化bean，实例化bean过程比较复杂。FactoryBean接口就是为了简化此过程，把bean的实例化定制逻辑下发给使用者。

在该接口中还定义了以下3个方法。

T getObject()：返回由FactoryBean创建的bean实例，如果isSingleton()返回true，则该实例会放到Spring容器中单实例缓存池中。

boolean isSingleton()：返回由FactoryBean创建的bean实例的作用域是singleton还是prototype。

Class<T> getObjectType()：返回FactoryBean创建的bean类型。

注意一点：通过Spring容器的getBean()方法返回的不是FactoryBean本身，而是FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()代理了getBean()方法。如果希望获取CarFactoryBean的实例，则需要在使用getBean(beanName)方法时在beanName前显示的加上 "&" 前缀。

一个例子：

实体类：

```
1 | public class Fruit {  
2 |  
3 |     private String name;  
4 |     private String color;  
5 |  
6 |     public String getName() {  
7 |         return name;  
8 |     }  
9 |  
10 |     public void setName(String name) {  
11 |         this.name = name;  
12 |     }  
13 | }
```

```
12     }
13
14     public String getColor() {
15         return color;
16     }
17
18     public void setColor(String color) {
19         this.color = color;
20     }
21
22     @Override
23     public String toString() {
24         return "Fruit{" +
25             "name='" + name + '\'' +
26             ", color='" + color + '\'' +
27             '}';
28     }
29 }
```

自定义FactoryBean:

```
1  @Component
2  public class FruitFactoryBean implements FactoryBean<Fruit> {
3
4      @Override
5      public Fruit getObject() throws Exception {
6          Fruit fruit = new Fruit();
7          fruit.setColor("red");
8          fruit.setName("apple");
9          return fruit;
10     }
11
12     @Override
13     public Class<?> getObjectType() {
14         return Fruit.class;
15     }
16
17     @Override
18     public boolean isSingleton() {
19         return true;
20     }
21 }
```

测试类:

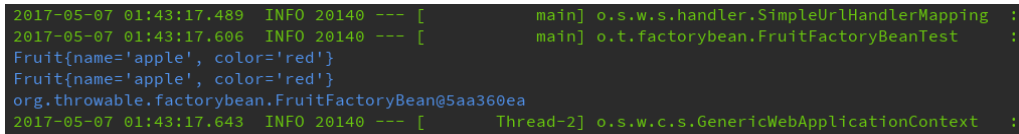
```
1  @SpringBootTest(classes = Application.class)
2  @RunWith(SpringJUnit4ClassRunner.class)
3  public class FruitFactoryBeanTest {
4
5      @Autowired
6      private FruitFactoryBean fruitFactoryBean;
7
8      @Autowired
9      private ApplicationContext applicationContext;
10
11     @Test
12     public void getObject() throws Exception {
13         //直接通过#getObject获取实例
14         Fruit apple = fruitFactoryBean.getObject();
15         System.out.println(apple.toString());
16         //通过Spring上下文获取实例
17         Fruit fruit = (Fruit) applicationContext.getBean("fruitFactoryBean");
18         System.out.println(fruit);
19         //获取FruitFactoryBean自身的实例
20     }
```

```

20         FruitFactoryBean bean = (FruitFactoryBean) applicationContext.getBean("&fruitFac
21         System.out.println(bean);
22     }
23
24 }

```

结果：



```

2017-05-07 01:43:17.489 INFO 20140 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping :
2017-05-07 01:43:17.606 INFO 20140 --- [main] o.t.factorybean.FruitFactoryBeanTest :
Fruit{name='apple', color='red'}
Fruit{name='apple', color='red'}
org.throwable.factorybean.FruitFactoryBean@5aa360ea
2017-05-07 01:43:17.643 INFO 20140 --- [Thread-2] o.s.w.c.s.GenericWebApplicationContext :

```

O4.png

结果和预期一样，通过ApplicationContext#getBean(beanName)获取到的实际上是FactoryBean#getObject的实例，ApplicationContext#getBean("&" + beanName)获取到的才是FruitFactoryBean本身的实例。

7.ApplicationListener

ApplicationListener是一个接口，里面只有一个onApplicationEvent(E event)方法，这个泛型E必须是ApplicationEvent的子类，而ApplicationEvent是Spring定义的事件，继承于EventObject，构造要求必须传入一个Object类型的source，这个source可以作为一个存储对象。将会在ApplicationListener的onApplicationEvent里面得到回调。如果在上下文中部署一个实现了ApplicationListener接口的bean，那么每当在一个ApplicationEvent发布到ApplicationContext时，这个bean得到通知。其实这就是标准的Observer设计模式。另外，ApplicationEvent的发布由ApplicationContext通过#publishEvent方法完成。其实这个实现从原理和代码上看都有点像Guava的eventbus。

贴一个例子：

EmailEvent:

```

1 public class EmailEvent extends ApplicationEvent {
2
3     private String author;
4     private String content;
5     private String date;
6
7     public EmailEvent(Object source, String author, String content, String date) {
8         super(source);
9         this.author = author;
10        this.content = content;
11        this.date = date;
12    }
13
14    public String getAuthor() {
15        return author;
16    }
17
18    public void setAuthor(String author) {
19        this.author = author;
20    }
21
22    public String getContent() {

```

```
23         return content;
24     }
25
26     public void setContent(String content) {
27         this.content = content;
28     }
29
30     public String getDate() {
31         return date;
32     }
33
34     public void setDate(String date) {
35         this.date = date;
36     }
37 }
```

EmailApplicationListener:

```
1  @Component
2  public class EmailApplicationListener implements ApplicationListener<EmailEvent> {
3
4      @Override
5      public void onApplicationEvent(EmailEvent event) {
6          System.out.println("EmailApplicationListener callback!!");
7          System.out.println("EmailEvent --> source: " + event.getSource());
8          System.out.println("EmailEvent --> author: " + event.getAuthor());
9          System.out.println("EmailEvent --> content: " + event.getContent());
10         System.out.println("EmailEvent --> date: " + event.getDate());
11     }
12 }
```

测试类:

```
1  @SpringBootTest(classes = Application.class)
2  @RunWith(SpringJUnit4ClassRunner.class)
3  public class EmailApplicationListenerTest {
4
5      @Autowired
6      private ApplicationContext applicationContext;
7
8      @Test
9      public void onApplicationEvent() throws Exception {
10         applicationContext.publishEvent(new EmailEvent("this is source",
11             "throwable", "here is emailEvent", "2017-5-16"));
12     }
13
14 }
```

控制台输出:

```
1  EmailApplicationListener callback!!
2  EmailEvent --> source: this is source
3  EmailEvent --> author: throwable
4  EmailEvent --> content: here is emailEvent
5  EmailEvent --> date: 2017-5-16
```

然后发觉简书竟然没有markdown的[toc], 有点不方便, 吐槽一下。

Updated on 2017-5-16 23:56.

Help yourselves!

我是throwable,在广州奋斗,白天上班,晚上和双休不定时加班,晚上有空坚持写下博客。

希望我的文章能够给你带来收获,共勉。

33人点赞 >

Spring拾遗

"如果觉得我的文章对您有用，请随意打赏。你的支持和鼓励是我前进动力的一部分"

赞赏支持

还没有人赞赏，支持一下

zhrowable

不再维护和更新，迁移到<http://throwable.coding.me>或者<http://t...>

总资产10 (约0.97元) 共写了5.2W字 获得196个赞 共161个粉丝

关注

海外工作招聘

gm游戏手游

开服时间表

机器人售价

调速液力耦合器

写下你的评论...

全部评论 1

只看作者

按时间倒序 按时间正序

不存在的里皮

2楼 2019.05.21 01:45

总结的不错

赞

回复

被以下专题收入，发现更多相似内容

java进阶干货

Java学习笔记

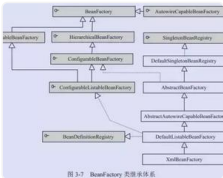
推荐阅读

更多精彩内容 >

Spring IOC原理总结

Spring容器高层视图 Spring 启动时读取应用程序提供的Bean配置信息，并在Spring容器中生成一份相...

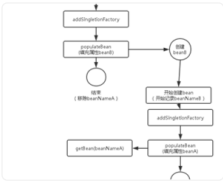
LeiLv 阅读 9,079 评论 4 赞 40



Spring

1.Spring整体架构 (1) Core container 核心容器包括了Core、Beans、Context和...

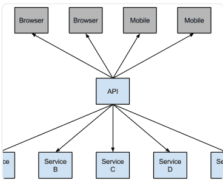
pgl2011 阅读 393 评论 0 赞 7



Spring Cloud

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智...

卡卡罗2017 阅读 75,010 评论 12 赞 116



Spring中常见面试题

什么是Spring Spring是一个开源的Java EE开发框架。Spring框架的核心功能可以应用在任何Jav...

jemmm 阅读 11,864 评论 1 赞 130

【那些年的有缘无份--生命中错过的，你和我的故事】留言精...

2017年7月28日，星期五，这里是余老诗写作研习社每日一思的活动，欢迎大家一起来留言讨论关于错过的故事~ 在对的...

浔潏 阅读 110 评论 0 赞 1

