

spring的有20多篇博客需要看

1、@value不能给static修饰的属性赋值，图一中的做法会注入失败。但是可以为静态属性提供一个set方法，在set方法上边使用@Value注解注入（这样做感觉没啥意义），如图二：

从源码上发现，理论上spring是可以对静态域注入的，只是spring没有这样做，它认为依赖注入发生的时段是在实例的生命周期，而不是类的生命周期：

```
@Value("${zk.server.addr}")  
private static String zkServerAddr ;
```

```
public static String gender;  
@Value("${bird.color}")  
public void setGender(String gender){  
    Bird.gender = gender;  
}
```

2、@Order注解用来定义Spring容器加载Bean的顺序
(<https://www.jianshu.com/p/37edf9389814>)

2、spring的监听机制。

2、Assert断言工具类可以在项目中使用。

3、AnnotationConfigApplicationContext有多种方式从容器中获取bean，可以获取某个类型的所有类。

4、spring提供了很多拓展接口，我们可以通过实现接口获取到相应的实例对象（**获取实例对象句柄**）从而对实例对象进行特殊操作（**只要获取到了实例对象句柄则可以对相应实例内存空间进行操作**），例如实现ApplicationContextAware、BeanFactoryAware等接口：

- 实现ApplicationContextAware获取bean容器:

```
public interface ApplicationContextAware extends Aware {  
  
    /**  
     * Set the ApplicationContext that this object runs in.  
     * Normally this call will be used to initialize the object.  
     * <p>Invoked after population of normal bean properties but before an init callback such  
     * as {@link org.springframework.beans.factory.InitializingBean#afterPropertiesSet()}  
     * or a custom init-method. Invoked after {@link ResourceLoaderAware#setResourceLoader},  
     * {@link ApplicationEventPublisherAware#setApplicationEventPublisher} and  
     * {@link MessageSourceAware}, if applicable.  
     * @param applicationContext the ApplicationContext object to be used by this object  
     * @throws ApplicationContextException in case of context initialization errors  
     * @throws BeansException if thrown by application context methods  
     * @see org.springframework.beans.factory.BeanInitializationException  
     */  
    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;  
}
```

- 实现BeanFactoryAware获取beanFactory实例:

```
public interface BeanFactoryAware extends Aware {  
  
    /**  
     * Callback that supplies the owning factory to a bean instance.  
     * <p>Invoked after the population of normal bean properties  
     * but before an initialization callback such as  
     * {@link InitializingBean#afterPropertiesSet()} or a custom init-method.  
     * @param beanFactory owning BeanFactory (never {@code null}).  
     * The bean can immediately call methods on the factory.  
     * @throws BeansException in case of initialization errors  
     * @see BeanInitializationException  
     */  
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;  
}
```

5、Spring提供的BeanFactoryUtils, 可以通过它获取容器中的bean:

```
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProviderConfig.class, includeNonSingletons: false, allow  
null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProtocolConfig.class, includeNonSingletons: false,
```

首先需要获取容器, 将容器传进去

```

* @throws BeansException if a bean could not be created
*/
public static <T> Map<String, T> beansOfTypeIncludingAncestors(
    ListableBeanFactory lbf, Class<T> type, boolean includeNonSingletons, boolean allowEagerInit)
    throws BeansException {

    Assert.notNull(lbf, message: "ListableBeanFactory must not be null");
    Map<String, T> result = new LinkedHashMap<>(initialCapacity: 4);
    result.putAll(lbf.getBeansOfType(type, includeNonSingletons, allowEagerInit));
    if (lbf instanceof HierarchicalBeanFactory) {
        HierarchicalBeanFactory hbf = (HierarchicalBeanFactory) lbf;
        if (hbf.getParentBeanFactory() instanceof ListableBeanFactory) {
            Map<String, T> parentResult = beansOfTypeIncludingAncestors(
                (ListableBeanFactory) hbf.getParentBeanFactory(), type, includeNonSingletons, allowEagerInit);
            for (Map.Entry<String, T> entry : parentResult.entrySet()) {
                String beanName = entry.getKey();
                if (!result.containsKey(beanName) && !hbf.containsLocalBean(beanName)) {
                    result.put(beanName, entry.getValue());
                }
            }
        }
    }
    return result;
}

```

最终它还是从传进来的容器中获取Bean

6、BeanFactory和FactoryBean

- BeanFactory，以Factory结尾，表示它是一个工厂类(接口)，它是负责生产和管理bean的一个工厂。BeanFactory只是个接口，并不是IOC容器的具体实现，它为IOC容器提供了基本的规范。BeanFactory和ApplicationContext是Spring框架的两个IOC容器，现在一般使用ApplicationnnContext，其不但包含了BeanFactory的作用，同时还进行更多的扩展。
-

7、标签解析：

解析过程中首先是寻找元素对应的解析器，进而调用解析器中的 parse 方法，那么结合示例来讲，其实就是首先获取在 MyNameSpaceHandler 类中的 init 方法中注册的对应的 UserBean-DefinitionParser 实例，并调用其 parse 方法进行进一步解析。

```

@export
public class DubboNamespaceHandler extends NamespaceHandlerSupport {

    static {
        Version.checkDuplicate(DubboNamespaceHandler.class);
    }

    @Override
    public void init() {
        registerBeanDefinitionParser(elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "config-center", new DubboBeanDefinitionParser(ConfigCenterBean.class, required: true));
        registerBeanDefinitionParser(elementName: "metadata-report", new DubboBeanDefinitionParser(MetadataReportConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "metrics", new DubboBeanDefinitionParser(MetricsConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, required: true));
        registerBeanDefinitionParser(elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class, required: true));
        registerBeanDefinitionParser(elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class, required: false));
        registerBeanDefinitionParser(elementName: "annotation", new AnnotationBeanDefinitionParser());
    }
}

```

8、spring中很多接口实现以后我们根据方法的执行顺序可以对容器中的bean做修改，这可能就是为啥需要对bean生命周期理解的原因：



9、ref标签：用来引用另一个bean。无论bean是否在同一个xml里边配置的，只要是注入到容器中的都可以引用。项目中mybatis.xml中引用datasource的例子。

10、Spring之Bean标签中的abstract和parent属性：

1.在bean中配置abstract属性为“ true” ,则spring容器不会为该创建对象。

```
1 <bean id="person" class="spring.extend.Person" abstract="true">
2     <property name="name" value="张三"></property>
3 </bean>
```

2.在bean中配置parent属性，则可以让该bean继承父类属性的值。

```
1 <bean id="student" class="spring.extend.Student" parent="person"></bean>
```

11、@Bean修饰带参数方法

(<https://www.cnblogs.com/maohuidong/p/11764544.html>) :

@Bean注解修饰带参数方法时，参数取值

原创 虎贲啊 最后发布于2018-06-25 18:38:54 阅读数 12650 ☆ 收藏

展开

拖拽

```
/**
 * 声明队列交换机等
 * @param connectionFactory
 * @return
 */
@Bean
public RabbitAdmin rabbitAdmin(ConnectionFactory connectionFactory) {
    System.out.println(String.format("-----getRabbitAdmin:%s", connectionFactory.hashCode()));
    return new RabbitAdmin(connectionFactory);
}
```

如上，有参数connectionFactory，若spring容器中只有一个ConnectionFactory 类型的bean，则不论参数取名为何都是按类型取bean ConnectionFactory 为参数，若有多个则参数取名必须为多个bean中的一个，否则报错。

12、如何给通过@Bean注入的实例属性赋值：@Autowired和@Value都可以标注在方法形参上边传值，这样我们通过@Bean往容器中注入的时候就能得到和在xml中配置属性值同样的效果：

```
/**
 * 方法参数默认注入方式为Autowired: <br>
 * 1: 复杂类型可以通过@Qualifier(value="dataSource")限定; <br>
 * 2: 对于普通类型使用@Value指定; <br>
 */
@Bean(name = "dataSource")
public DataSource dataSource(@Value("${jdbc.driverClass}") String driverClassName,
    @Value("${jdbc.jdbcUrl}") String url, @Value("${jdbc.user}") String username,
    @Value("${jdbc.password}") String password) {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    return dataSource;
}

@Bean(name = "jdbcTemplate")
public JdbcTemplate jdbcTemplate(@Qualifier(value = "dataSource") DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

12、当我们需要将jar包中的bean通过@Bean注解注入到容器中时，给它的属性赋值的方式是通过在方法参数中加@Value和@Autowired注解。对于我们自己项目中的类我们可以看spring第四次课中的例子就会一目了然，因为通过@Bean最终也是将对象交给容器管理。

12、springAop是由Advice拦截链（倒序）实现的，需要研究一下。

13、spring事务模板(TransactionSynchronizationManager), mq中使用过, 记录一下。

14、BeanPostProcessor影响的是所有注入容器中的bean，即在所有bean初始化前后都会调用BeanPostProcessor中实现的两个方法。

14、实现BeanDefinitionRegistryPostProcessor、ImportBeanDefinitionRegistrar

可以动态往容器中添加bean。这两个接口动态注册bean实际是基于BeanDefinitionRegistry实现的（即这两个接口的回调方法中会将BeanDefinitionRegistry传回来，然后调用registerBeanDefinition()方法进行注册），同时通过容器注册bean底层也是通过BeanDefinitionRegistry进行注册的。

BeanDefinitionRegistry 是pring框架用于动态注册BeanDefinition信息的接口，调用registerBeanDefinition方法可以将BeanDefinition注册到Spring容器中，其中name属性就是注册的BeanDefinition的名称：

- **实现BeanDefinitionRegistryPostProcessor:**

```
import ...
@Component("ljqDefinition")
public class LjqBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor{

    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
        System.out.println("LjqBeanDefinitionRegistryPostProcessor...postProcessBeanDefinitionRegistry()...,bean的数量");
        RootBeanDefinition rbd = new RootBeanDefinition(Ljq.class);
        BeanDefinitionBuilder rbd = BeanDefinitionBuilder.rootBeanDefinition(Ljq.class);
        rbd.addPropertyValue( name: "name", value: "cdr");
        registry.registerBeanDefinition( beanName: "cdr",rbd.getBeanDefinition());
    }
}
```

- **实现ImportBeanDefinitionRegistrar:**

```
public class LjqImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {  
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
        boolean bool1 = registry.containsBeanDefinition(beanName: "com.ljq.study6.bean.Dog");  
        boolean bool2 = registry.containsBeanDefinition(beanName: "com.ljq.study6.bean.Dog");  
        System.out.println("----->>>>>>>bean的个数"+registry.getBeanDefinitionCount());  
        if(bool1 && bool2){  
            RootBeanDefinition beanDefinition = new RootBeanDefinition(Pig.class);  
            registry.registerBeanDefinition(beanName: "pig", beanDefinition);  
        }  
    }  
}
```

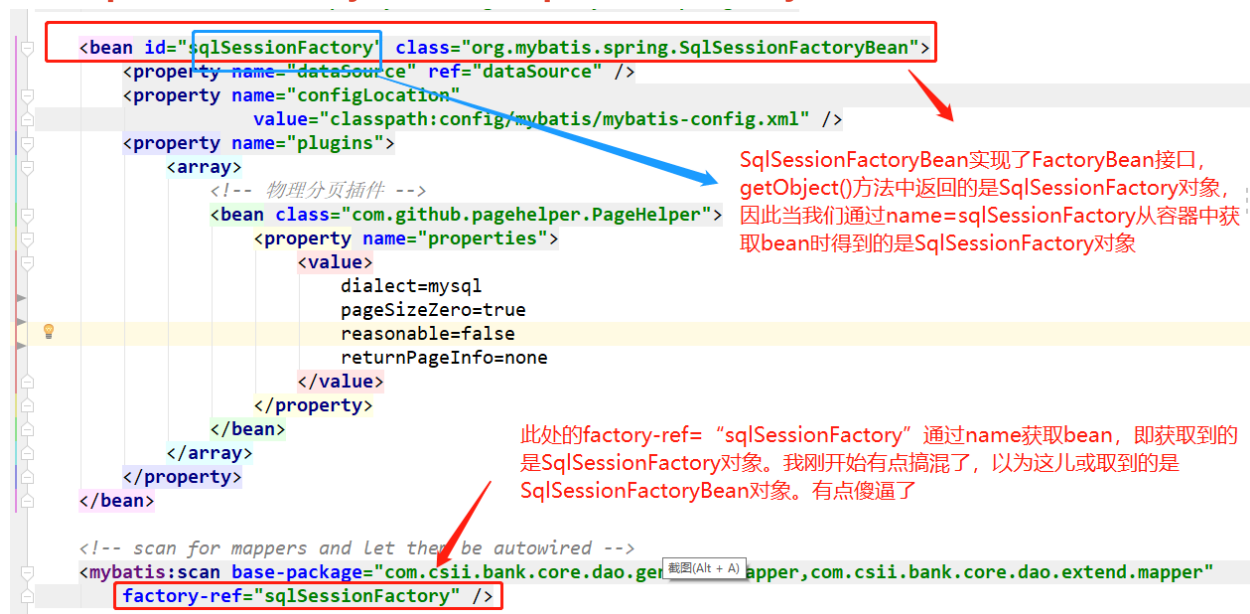
- 通过容器注入（不仅限于图片中的两种方法，其他方法可以看源码）：

```

public void test01(){
    AnnotationConfigApplicationContext app = new AnnotationConfigApplicationContext(Study2MainConfig.class);
    // 第一种方式
    app.registerBean( beanName: "pigOne", Pig.class, new BeanDefinitionCustomizer() {
        public void customize(BeaDefinition bd) {
            bd.setScope("singleton");
            bd.getPropertyValues().add( propertyName: "name", propertyValue: "piglllllllllllllllll");
        }
    });
    // 第二种方式
    RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(Pig.class);
    rootBeanDefinition.getPropertyValues().add( propertyName: "name", propertyValue: "piggggggggggggggggggg");
    app.registerBeanDefinition( beanName: "pigTwo", rootBeanDefinition);
}

```


15、通过FactoryBean注入bean时，从容器中getBean()获取到的是FactoryBean的getObject()方法中返回的对象，通过名字获取到的是getObject()方法中返回的对象，当通过@Resource或者@Autowired、@Qualifier两个注解配合使用时取到的也是getObject()方法返回的对象，而通过@Autowired注入的时候是根据类型来获取的，所以我们获取那个类型就会取到那个bean。FactoryBean与@Autowired结合使用的一些疑问：<https://blog.csdn.net/u014473112/article/details/84309965>。hivegl中通过SqlSessionFactoryBean注入SqlSessionFactory的例子：



```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation"
    value="classpath:config/mybatis/mybatis-config.xml" />
  <property name="plugins">
    <array>
      <!-- 物理分页插件 -->
      <bean class="com.github.pagehelper.PageHelper">
        <property name="properties">
          <value>
            dialect=mysql
            pageSizeZero=true
            reasonable=false
            returnPageInfo=none
          </value>
        </property>
      </bean>
    </array>
  </property>
</bean>
```

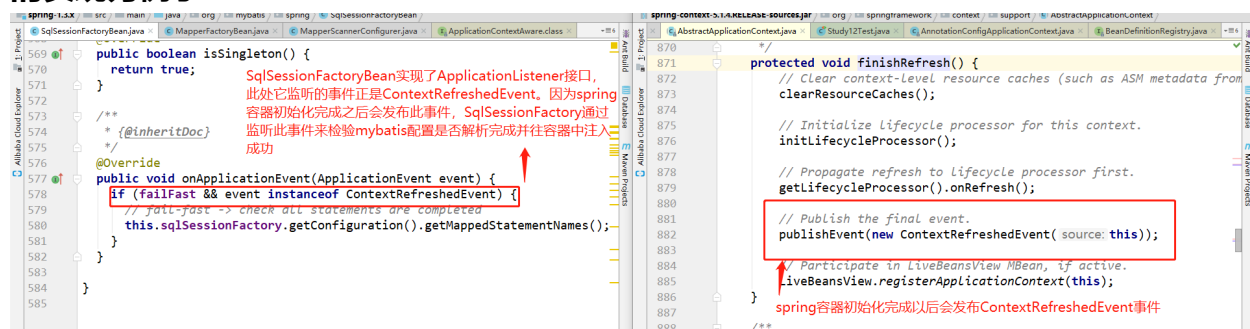
SqlSessionFactoryBean实现了FactoryBean接口，getObject()方法中返回的是SqlSessionFactory对象，因此当我们通过name=sqlSessionFactory从容器中获取bean时得到的是SqlSessionFactory对象

此处的factory-ref= "sqlSessionFactory" 通过name获取bean，即获取到的是SqlSessionFactory对象。我刚开始有点搞混了，以为这儿或取到的是SqlSessionFactoryBean对象。有点傻逼了

```
<!-- scan for mappers and let them be autowired -->
<mybatis:scan base-package="com.csii.bank.core.dao.mapper,com.csii.bank.core.dao.extend.mapper"
  factory-ref="sqlSessionFactory" />
```

16、理解Spring

ApplicationListener (<https://blog.csdn.net/liyantianmin/article/details/81017960>)，需要手动发布事件来触发监听。spring内置了一些事件 (Event) 和监听 (Listener)。我们也可以自定义事件和监听，不必拘束，完全按自己业务来就可以了。以mybatis-spring中的实现为例子：



```
public boolean isSingleton() {
  return true;
}

/**
 * {@inheritDoc}
 */
@Override
public void onApplicationEvent(ApplicationEvent event) {
  if (failFast && event instanceof ContextRefreshedEvent) {
    // fail-fast -> check all statements are completed
    this.sqlSessionFactory.getConfiguration().getMappedStatementNames();
  }
}

protected void finishRefresh() {
  // Clear context-level resource caches (such as ASM metadata from
  clearResourceCaches();

  // Initialize lifecycle processor for this context.
  initLifecycleProcessor();

  // Propagate refresh to lifecycle processor first.
  getLifecycleProcessor().onRefresh();

  // Publish the final event.
  publishEvent(new ContextRefreshedEvent( source: this));

  // Participate in LiveBeansView MBean, if active.
  liveBeansView.registerApplicationContext(this);
}
```

17、spring中bean的加载顺序根据bean之间的相互依赖而不同。当beanA中引用了beanB时，会优先加载beanB。晋城接入RocketMq时DefaultMqClient中引入了TxMQProducer，且DefaultMqClient实现了InitializingBean接口，并在afterPropertiesSet()方法中调用了TxMQProducer的start()方法，这说明TxMqProducer会在DefaultMqClient之前被加载。

```
public void afterPropertiesSet() throws Exception {  
    Assert.notNull(this.producer, message: "Property 'producer' is required");  
    this.producer.start();  
}
```

18、spring监听事件的原理其实就是指定接口（ApplicationListener），用户实现改接口，当publish一个事件的时候底层实际是去找监听该event的ApplicationListener的实现类，然后在调用onApplicationEvent（）方法。我们也可以借助这个思想实现自己的监听机制并不难（**观察者模式**）。spring容器初始化完成时会发布ContextRefreshedEvent事件（在finishRefresh()方法中）。

19、@Cacheable注解的使用方式（<https://www.jianshu.com/p/d9ecd56710c4>）

20、ConfigurableEnvironment环境变量。可一从容器中获取或者通过@Value注解或者通过Environment类获取。

21、ApplicationEventMulticaster事件派发器。

22、泸州版本接入kafka时向容器中注入Properties的方式的思考（Properties是一种特殊的map，只要是java类我们都可以将其注入到容器中，map、list、数组等都不例外，只不过容器不同于普通的bean，给人感觉不一样，细思一下没啥不同）----->向spring容器中注入map、list、数组等的方式：

- 通过xml配置方式：

```
<bean id="hashMap" class="java.util.HashMap">  
    <constructor-arg>  
        <map>  
            <entry key="name" value="ljq"></entry>  
            <entry key="age" value="26"></entry>  
        </map>  
    </constructor-arg>  
</bean>
```

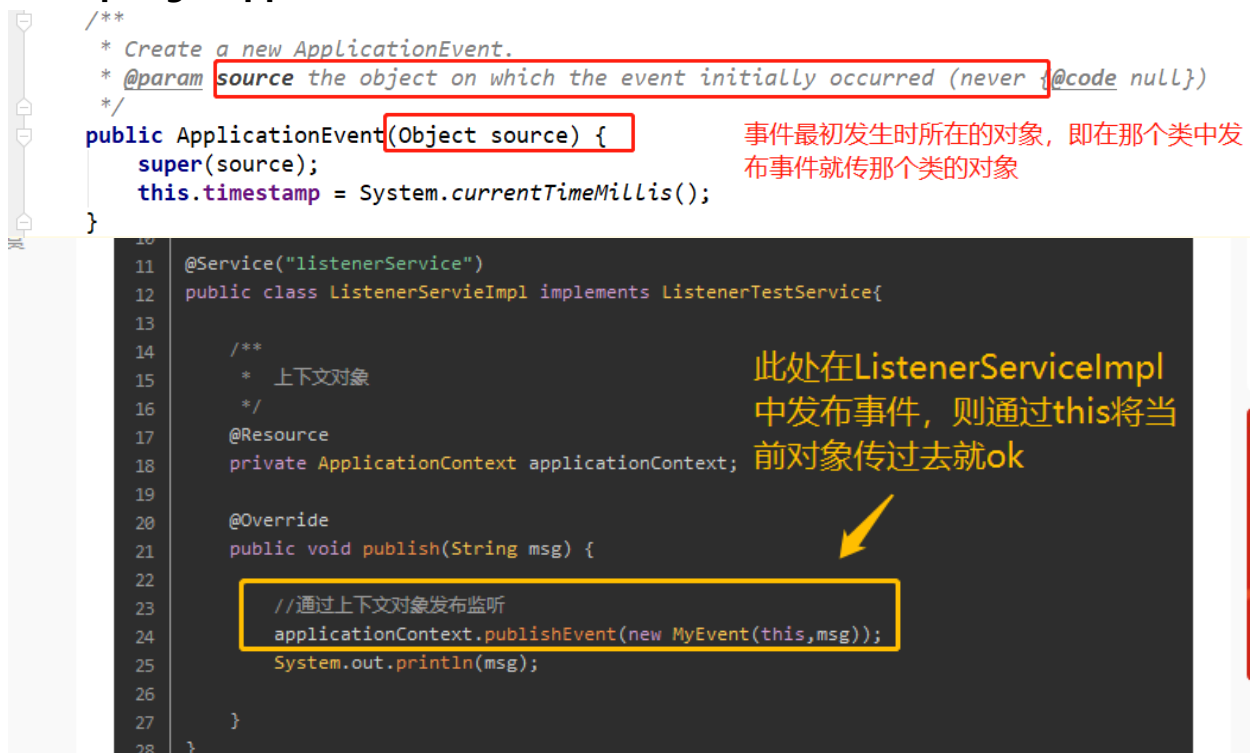

- 通过@Bean注入：

```
@Bean
public Map<String,String> hashMap(){
    Map map = new HashMap<String,String>(initialCapacity: 8);
    map.put("name","ljq");
    return map;
}
```

23、在使用Spring-Cloud微服务框架的时候，对于@Import和@ImportResource这两个注解想必大家并不陌生。我们会经常用@Import来导入配置类或者导入一个带有@Component等注解要放入Spring容器中的类；用@ImportResource来导入一个传统的xml配置文件。

24、spring自定义监听事件的时候我们可以通过自定义被监听者（ApplicationEvent）进行业务参数透传（<https://www.jianshu.com/p/897e0a128e54>）。

25、Spring中ApplicationEvent类的构造参数source详解：



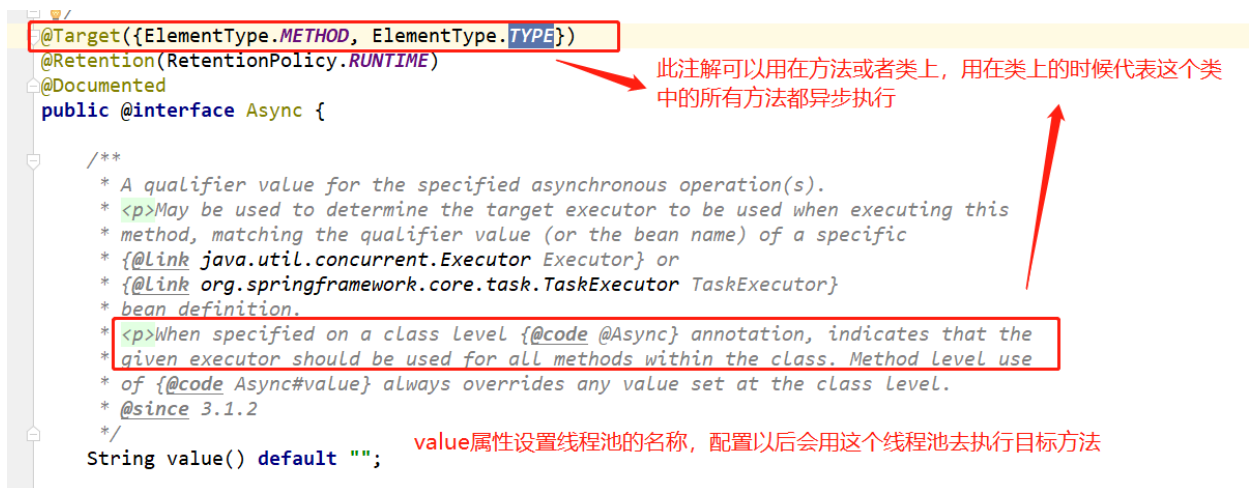
```
/**
 * Create a new ApplicationEvent.
 * @param source the object on which the event initially occurred (never null)
 */
public ApplicationEvent(Object source) {
    super(source);
    this.timestamp = System.currentTimeMillis();
}
```

事件最初发生时所在的对象，即在那个类中发布事件就传那个类的对象

```
10
11 @Service("listenerService")
12 public class ListenerServieImpl implements ListenerTestService{
13
14     /**
15      * 上下文对象
16      */
17     @Resource
18     private ApplicationContext applicationContext;
19
20     @Override
21     public void publish(String msg) {
22
23         //通过上下文对象发布监听
24         applicationContext.publishEvent(new MyEvent(this,msg));
25         System.out.println(msg);
26
27     }
28 }
```

此处ListenerServiceImpl中发布事件，则通过this将当前对象传过去就ok

27、spring提供了@Async注解来标识方法是否异步执行，当我们不手动配置线程池时（即向容器中手动注入一个自己配置的线程池，详情使用参考spring-boot笔记中的使用）spring默认提供的策略是每次都new一个新线程去执行目标方法：



28、**spring 事件监听同时支持同步事件及异步事件**。spring事件发布是依靠调用实现了ApplicationEventPublisher接口类的publishEvent方法进行发布事件，而publishEvent方法又是通过调用实现了ApplicationEventMulticaster（事件广播器）接口的SimpleApplicationEventMulticaster（具体的事件广播器）类的multicastEvent方法进行事件的广播的，ApplicationEventMulticaster中保存了所有的实现了ApplicationListener接口的监听器。**异步事件通知主要依靠SimpleApplicationEventMulticaster类中的Executor去实现的，如果这个变量不配置的话默认事件通知是同步的，否则就是异步通知了：**

- <https://www.cnblogs.com/angryprogrammer/p/12536752.html>，这篇博客通过自定义注解动态控制了不同监听事件可以使用同步或者异步去执行，很不错。
- <https://www.cnblogs.com/littlemonk/p/5759908.html>，通过这篇博客可以看出，对于不同的监听事件我们可以通过是否加@Async注解去控制它是否异步执行。

28、AbstractApplicationEventMulticaster是一个模板类，它实现了ApplicationEventMulticaster接口的大部分方法，但是事件广播方法multicastEvent()是留给了子类SimpleApplicationEventMulticaster来实现的（**模板方法设计模式**）：



29、spring监听事件关于异步和同步执行的两种方式：

- 利用spring的@Async注解：对于需要异步执行的监听器在onApplicationEvent()方法上边加@Async注解，对于不需要异步执行的监听器不做特殊处理，这样就可以达到部分监听器异步执行，部分同执行的效果。
- 如果我们想通过继承SimpleApplicationEventMulticaster类或者在容器中手动注入一个SimpleApplicationEventMulticaster 类对象并配置一个线程池的话一定要将bean的名称设置为applicationEventMulticaster，这样spring才会通过我们注入的这个bean去执行监听任务：

```

public class SimpleApplicationEventMulticaster extends AbstractApplicationEventMulticaster {
    @Nullable
    private Executor taskExecutor;

    @Nullable
    private ErrorHandler errorHandler;
}

else {
    getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
}

//
ApplicationEventMulticaster getApplicationEventMulticaster() throws IllegalStateException {
    if (this.applicationEventMulticaster == null) {
        throw new IllegalStateException("ApplicationEventMulticaster not initialized - " +
            "call 'refresh' before multicasting events via the context: " + this);
    }
    return this.applicationEventMulticaster;
}

```

SimpleApplicationEventMulticaster的线程池属性默认为空，此时所有监听器都是同步执行的，如果想让监听器异步执行，则我们手动注入一个SimpleApplicationEventMulticaster对象并配置线程池属性或者继承这个类实现一个子类并将线程池属性赋值再将子类对象注入到容器中，这两种方式二者选其一，并且bean的名字是固定的，看第二张图片

获取事件广播器并执行相应事件

bean名称必须是这个

30、spring实现Ordered接口可以控制bean调用的先后顺序，和@Order注解作用一样：

```

/**
 * Get the order value of this object.
 * <p>Higher values are interpreted as lower priority. As a consequence,
 * the object with the lowest value has the highest priority (somewhat
 * analogous to Servlet {@code Load-on-startup} values).
 * <p>Same order values will result in arbitrary sort positions for the
 * affected objects.
 * @return the order value
 * @see #HIGHEST_PRECEDENCE
 * @see #LOWEST_PRECEDENCE
 */
int getOrder();

```

实现getOrder()方法，返回值越大的优先级越低