配置文件解析和Environment、PropertySource对象

1、自定义启动器

1.1、简述

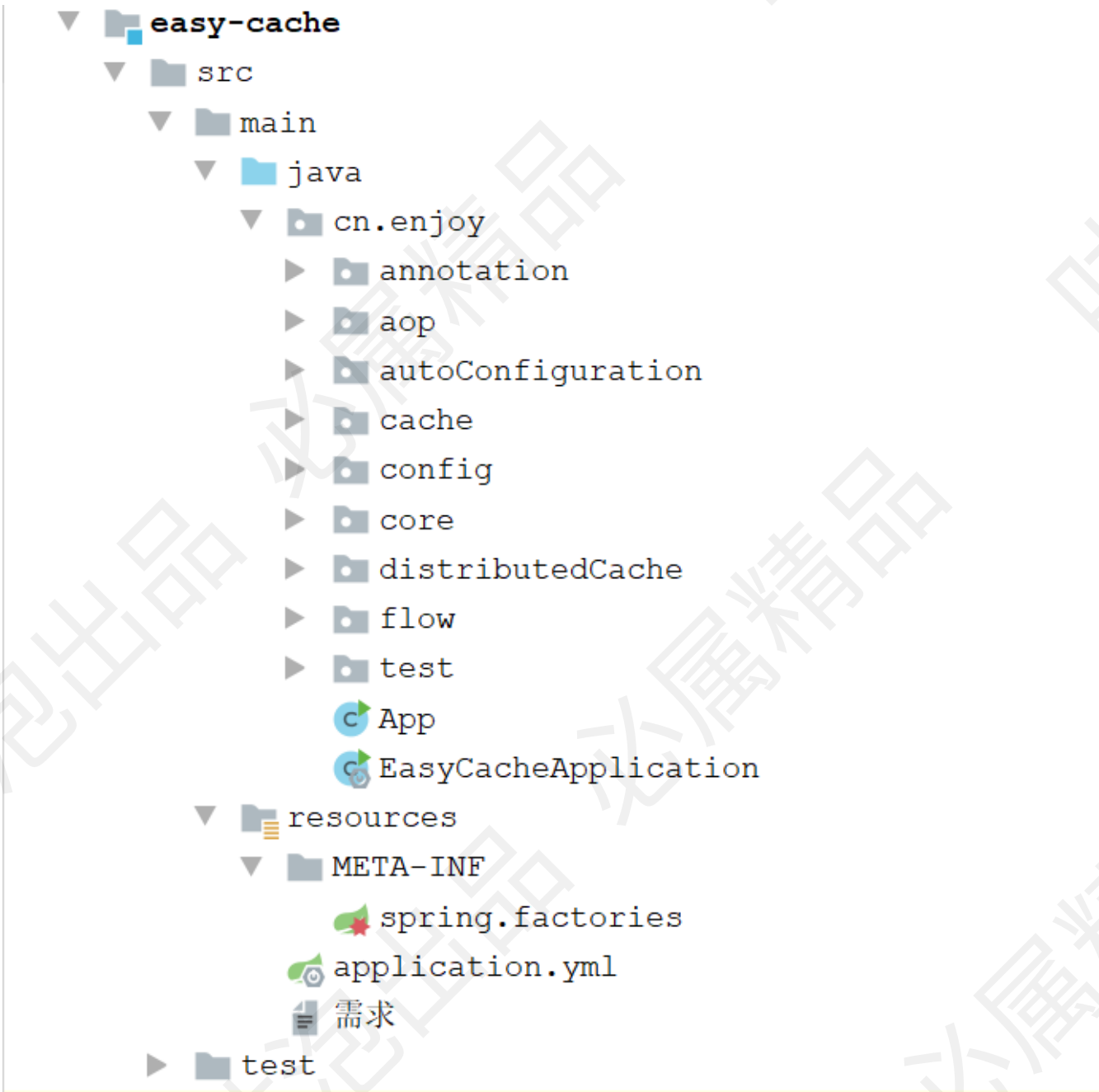
前面我们已经属性了springboot中自动配置的原理，那么接下来我们来通过自动配置的原理来自定义一个启动器

1.2、为什么要定义启动器

比如你有一个springboot工程，但是有一天你需要把一个其他部分的功能导入到你的springboot工程中，但是他们的功能很庞大，如果要引入进来需要引入很多依赖然后也需要在springboot工程中实例化很多他们的类，引入过来很麻烦，这时候就需要有一个自定义启动器的功能了，在自定义启动器工程中把他们功能中需要实例化的类在启动器工程中实例化，然后把启动器工程打成jar包，然后在你的springboot工程中就只需要maven依赖这个启动器工程就可以了，通过一个简单的maven依赖就可以把功能导入到你的springboot工程，这个就是自定义启动点的作用

1.3、自定义启动器步骤

1、新建一个工程



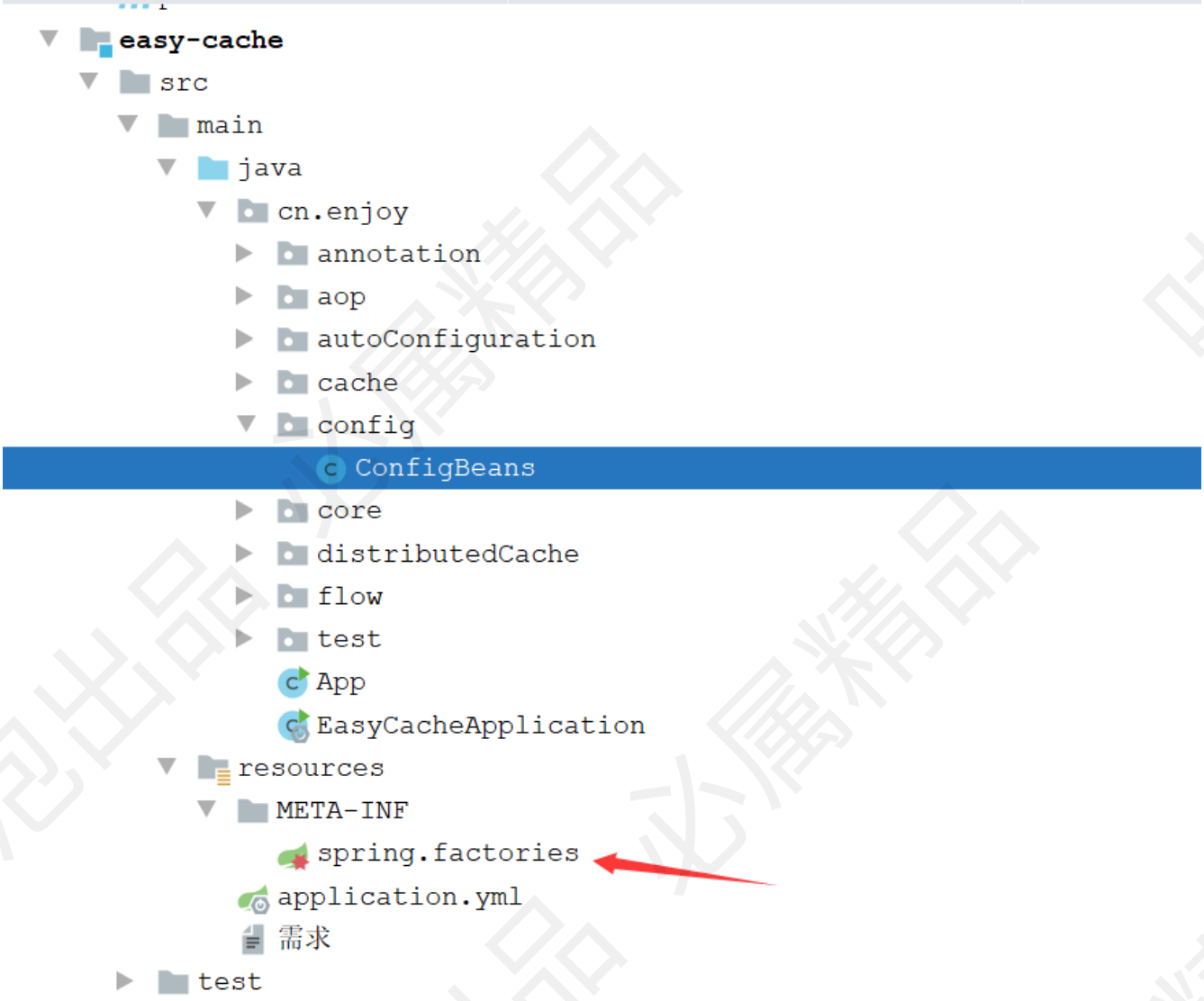
2、实例化类

把需要实例化的类集中在一个类中实例化，比如：

```
public class ConfigBeans {

    @Bean
    @ConditionalOnProperty(name = "spring.easycache.local.type", havingValue = "caffeine",
            matchIfMissing = true)
    public CaffeineCacheServiceImpl caffeineCacheService() {
        return new CaffeineCacheServiceImpl();
    }

    @Bean
    @ConditionalOnProperty(name = "spring.easycache.local.type", havingValue = "guava",
            matchIfMissing = false)
```

```
    public GuavaCacheServiceImpl guavaCacheService() {
        return new GuavaCacheServiceImpl();
    }

    @Bean
    public RedisApi redisApi() {
        return new RedisApi();
    }

    @Bean
    public CacheUtil cacheUtil() {
        return new CacheUtil();
    }

    @Bean
    @ConditionalOnProperty(name = "spring.easycache.distributed.type", havingValue =
"redis",
            matchIfMissing = true)
    public RedisCacheServiceImpl redisCacheService() {
        return new RedisCacheServiceImpl();
    }

    @Bean
    public FirstCacheHandler firstCacheHandler() {
        return new FirstCacheHandler();
    }

    @Bean
    public SecondCacheHandler secondCacheHandler() {
        return new SecondCacheHandler();
    }

    @Bean
    public ThirdCacheHandler thirdCacheHandler() {
        return new ThirdCacheHandler();
    }
}
```

3、创建spring.factories文件



4、在spring.factories中导入类

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=cn.enjoy.config.ConfigBeans
```

5、把工程打包

mvn clean install -Dmaven.test.skip=true

6、把工程依赖到其他工程

```
<dependency>
    <groupId>cn.enjoy</groupId>
    <artifactId>easy-cache</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

上述就完成了一个自定义启动的定义和引用。

2、配置文件解析

spring或者springboot中配置文件解析都是一个很大的模块，这里我们就着重的分析在spring和springboot中配置文件是如何解析的。

2.1、spring配置文件解析

我们先来看看在spring如果要解析一个配置文件应该怎么做

我们必须要引入配置文件的解析类，把它配置进spring容器，如下：

```
@Component
public class BeanConfig implements ResourceLoaderAware {

    private ResourceLoader resourceLoader;

    @Bean
    public PropertySourcesPlaceholderConfigurer getPropertySourcesPlaceholderConfigurer() {
        PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer = new
PropertySourcesPlaceholderConfigurer();

 propertySourcesPlaceholderConfigurer.setLocation(resourceLoader.getResource("application.p
roperties"));
        return propertySourcesPlaceholderConfigurer;
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
}
```

或者加上**@PropertySource(value = "classpath:application.properties")**注解

2.2、springboot配置文件解析

我们重点来分析一下springboot是如何解析配置文件的。

前面我们看到了spring如果要添加配置文件解析的功能，必须要添加解析类进spring容器，那么springboot我们前面知道它有自动配置能力，那么配置文件解析是不是也是自动配置中的一项呢，答案是肯定的。

2.2.1、自动配置类

在springboot中配置文件解析的自动配置类：**PropertyPlaceholderAutoConfiguration**

```
@Configuration(proxyBeanMethods = false)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
public class PropertyPlaceholderAutoConfiguration {

   @Bean
   @ConditionalOnMissingBean(search = SearchStrategy.CURRENT)
   public static PropertySourcesPlaceholderConfigurer
propertySourcesPlaceholderConfigurer() {
      return new PropertySourcesPlaceholderConfigurer();
   }

}
```

可以看到，这个类跟spring中引入的解析类是一样的，这样就把自动配置类加入到了spring容器了。接下来我们就从源码的角度来分析该类是如何解析配置文件的。

2.2.2、源码分析

PropertySourcesPlaceholderConfigurer类最终是实现了spring中的BeanFactoryPostProcessor接口的，所以当spring容器启动的时候会掉用到postProcessBeanFactory方法，我们来看看这个方法

```
@Override
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws
BeansException {
   if (this.propertySources == null) {
      this.propertySources = new MutablePropertySources();
      if (this.environment != null) {
         //把environment对象封装成的PropertySource对象加入到
         this.propertySources.addLast(
               //把environment对象封装成PropertySource对象MutablePropertySources中的list中
            new PropertySource<Environment>(ENVIRONMENT_PROPERTIES_PROPERTY_SOURCE_NAME,
this.environment) {
               @Override
               @Nullable
               //source就是environment对象
               public String getProperty(String key) {
                  return this.source.getProperty(key);
               }
            }
         );
      }
      try {
         //加载本地配置文件中的属性值包装成properties对象后，最终包装成PropertySource对象
         PropertySource<?> localPropertySource =
               new PropertiesPropertySource(LOCAL_PROPERTIES_PROPERTY_SOURCE_NAME,
mergeProperties());
         //加入到MutablePropertySources中的list中
         if (this.localOverride) {
            this.propertySources.addFirst(localPropertySource);
         }
         else {
            this.propertySources.addLast(localPropertySource);
```

```
            }
        }
        catch (IOException ex) {
            throw new BeanInitializationException("Could not load properties", ex);
        }
    }

    processProperties(beanFactory, new
 PropertySourcesPropertyResolver(this.propertySources));
    this.appliedPropertySources = this.propertySources;
 }
```

这个方法中有一个environment对象，我们来看看这个environment对象是如何初始化的

2.2.2.1、environment初始化

在springboot的run方法中有一行

```
 ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
```

在这里完成了environment对象的初始化

```
 ConfigurableEnvironment environment = getOrCreateEnvironment();
```

```
public AbstractEnvironment() {
    customizePropertySources(this.propertySources);
}
```

```
@Override
protected void customizePropertySources(MutablePropertySources propertySources) {
    propertySources.addLast(new StubPropertySource(SERVLET_CONFIG_PROPERTY_SOURCE_NAME));
    propertySources.addLast(new StubPropertySource(SERVLET_CONTEXT_PROPERTY_SOURCE_NAME));
    if (JndiLocatorDelegate.isDefaultJndiEnvironmentAvailable()) {
        propertySources.addLast(new JndiPropertySource(JNDI_PROPERTY_SOURCE_NAME));
    }
    super.customizePropertySources(propertySources);
}
```

```
@Override
protected void customizePropertySources(MutablePropertySources propertySources) {
    //这里加载 System.getProperties();
    propertySources.addLast(
            new PropertiesPropertySource(SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME,
getSystemProperties()));
    //这里加载 System.getenv();
    propertySources.addLast(
            new SystemEnvironmentPropertySource(SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME,
getSystemEnvironment()));
}
```

从上面的代码可以看到，environment对象本身就加载了系统环境变量里面的信息的，最终又把environment对象设置到了spring上下文对象中了。

2.2.2.2、PropertySources说明

在environment中我们看到了一个PropertySources对象，具体PropertySources是一个接口，该接口定义了对PropertySource对象的操作方法，那么PropertySources怎么理解呢？ 你可以把这个接口理解为属性源，比如整个工程中属性的来源有很多，比如配置文件、环境变量、远程配置中心等等。那么实现了PropertySources接口的类就可以理解为存储这些配置属性来源的集合类，比如整个环境有，分布式配置数据，本地配置文件数据，环境变量数据等等，这些数据都会封装成PropertySource对象，然后把对象存放到实现了PropertySources接口的类的某个集合中。实现了PropertySources接口的类就是**MutablePropertySources**，那么在该类中就有一个存储PropertySource对象的集合

```
 */
public class MutablePropertySources implements PropertySources {

    private final List<PropertySource<?>> propertySourceList = new CopyOnWriteArrayList<>();
```

2.2.2.3、PropertySource说明

PropertySource可以理解为一种配置属性的来源，比如一个application.properties配置内容就会存储到一个PropertySource对象中，所以PropertySource对象肯定会有存储，获取属性值的方法。

```
public abstract class PropertySource<T> {

    protected final Log logger = LogFactory.getLog(getClass());

    protected final String name;

    protected final T source;
```

可以看到该类中有两个比较重要的属性，一个是name，一个是source，name是标识属性来源的名称，根据name可以找到一个PropertySource对象，source是用来存储属性key和value的结构，比如source可以是一个map，可以是一个properties。PropertySource中有一个getProperty方法，这个方法就可以根据key获取到这个key对应的属性值。

2.2.2.4、属性解析

```
@Override
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws
BeansException {
   if (this.propertySources == null) {
      this.propertySources = new MutablePropertySources();
      if (this.environment != null) {
         //把environment对象封装成的PropertySource对象加入到
         this.propertySources.addLast(
               //把environment对象封装成PropertySource对象MutablePropertySources中的list中
            new PropertySource<Environment>(ENVIRONMENT_PROPERTIES_PROPERTY_SOURCE_NAME,
this.environment) {
               @Override
```

```
               @Nullable
               //source就是environment对象
               public String getProperty(String key) {
                  return this.source.getProperty(key);
               }
            }
         );
      }
      try {
         //加载本地配置文件中的属性值包装成properties对象后，最终包装成PropertySource对象
         PropertySource<?> localPropertySource =
               new PropertiesPropertySource(LOCAL_PROPERTIES_PROPERTY_SOURCE_NAME,
mergeProperties());
         //加入到MutablePropertySources中的list中
         if (this.localOverride) {
            this.propertySources.addFirst(localPropertySource);
         }
         else {
            this.propertySources.addLast(localPropertySource);
         }
      }
      catch (IOException ex) {
         throw new BeanInitializationException("Could not load properties", ex);
      }
   }

   processProperties(beanFactory, new
PropertySourcesPropertyResolver(this.propertySources));
   this.appliedPropertySources = this.propertySources;
}
```

上面把一个environment对象封装到了propertySources对象中了

接着我们来看看**processProperties**方法

```
protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess,
      final ConfigurablePropertyResolver propertyResolver) throws BeansException {

   //设置占位符的前缀后缀 ${
   propertyResolver.setPlaceholderPrefix(this.placeholderPrefix);
    //   }
   propertyResolver.setPlaceholderSuffix(this.placeholderSuffix);
   //设分割符 :  ":"
   propertyResolver.setValueSeparator(this.valueSeparator);

   //重点是这个匿名对象 @value的依赖注入会掉过来
   StringValueResolver valueResolver = strVal -> {
      String resolved = (this.ignoreUnresolvablePlaceholders ?
            propertyResolver.resolvePlaceholders(strVal) :
            propertyResolver.resolveRequiredPlaceholders(strVal));
      if (this.trimValues) {
         resolved = resolved.trim();
      }
```

```
      return (resolved.equals(this.nullValue) ? null : resolved);
   };
   //核心流程。把占位符${xxx}替换成真正的值
   doProcessProperties(beanFactoryToProcess, valueResolver);
}
```

我们来看看doProcessProperties的逻辑

```
protected void doProcessProperties(ConfigurableListableBeanFactory beanFactoryToProcess,
      StringValueResolver valueResolver) {

   BeanDefinitionVisitor visitor = new BeanDefinitionVisitor(valueResolver);

   String[] beanNames = beanFactoryToProcess.getBeanDefinitionNames();
   for (String curName : beanNames) {
      // Check that we're not parsing our own bean definition,
      // to avoid failing on unresolvable placeholders in properties file locations.
      if (!(curName.equals(this.beanName) && beanFactoryToProcess.equals(this.beanFactory)))
{
         BeanDefinition bd = beanFactoryToProcess.getBeanDefinition(curName);
         try {
            //把beanDefinition中属性的值，例如：${xx.xx.name}解析成真正的配置属性值
            visitor.visitBeanDefinition(bd);
         }
         catch (Exception ex) {
            throw new BeanDefinitionStoreException(bd.getResourceDescription(), curName,
ex.getMessage(), ex);
         }
      }
   }

   // New in Spring 2.5: resolve placeholders in alias target names and aliases as well.
   beanFactoryToProcess.resolveAliases(valueResolver);

   // New in Spring 3.0: resolve placeholders in embedded values such as annotation
attributes.
    //@Value的时候有用，把解析器对象设置到BeanFactory中了
   beanFactoryToProcess.addEmbeddedValueResolver(valueResolver);
}
```

这个代码是重点代码，是一个值的解析器对象，@Value的时候会掉进来

```
   //重点是这个匿名对象 @Value的依赖注入会掉过来
   StringValueResolver valueResolver = strVal -> {
      String resolved = (this.ignoreUnresolvablePlaceholders ?
            propertyResolver.resolvePlaceholders(strVal) :
            propertyResolver.resolveRequiredPlaceholders(strVal));
      if (this.trimValues) {
         resolved = resolved.trim();
      }
      return (resolved.equals(this.nullValue) ? null : resolved);
   };
```

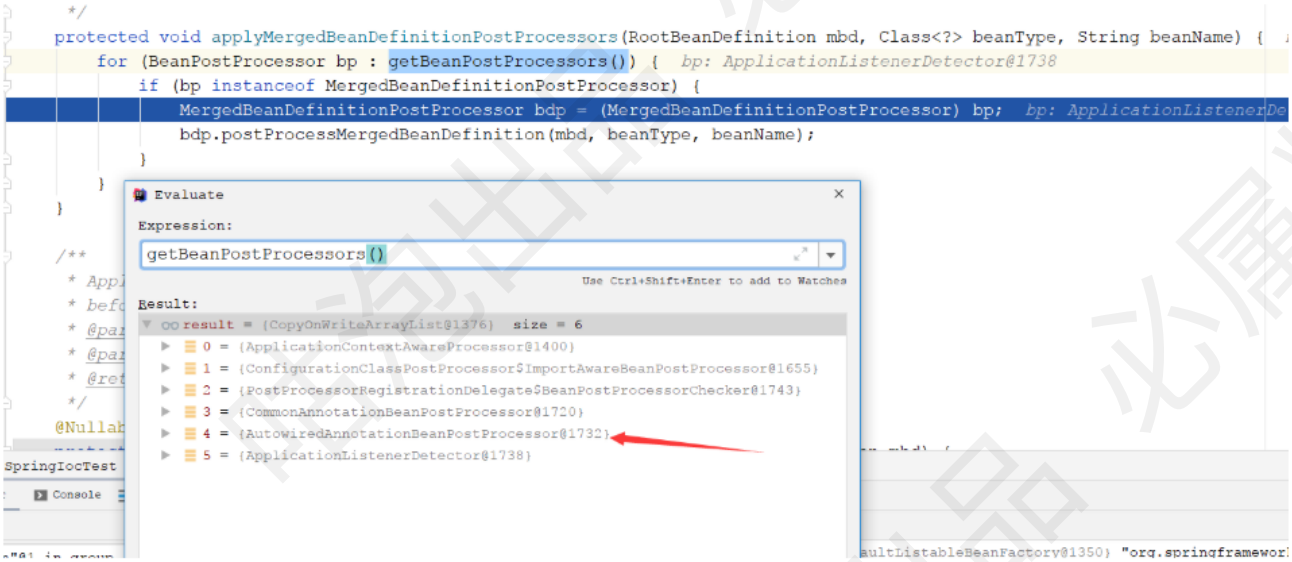
2.2.2.5、@Value的依赖注入

2.2.2.5.1、@Value注解的收集

注解收集的核心流程

```
// Allow post-processors to modify the merged bean definition.
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        try {
            //CommonAnnotationBeanPostProcessor  支持了@PostConstruct，@PreDestroy,@Resou
            //AutowiredAnnotationBeanPostProcessor 支持 @Autowired,@Value注解
            //BeanPostProcessor接口的典型运用，这里要理解这个接口
            //对类中注解的装配过程
            //重要程度5，必须看
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
    }
}
```

对应的接口类型MergedBeanDefinitionPostProcessor，这个类型的BeanPostProcessor的埋点就是用来处理注解收集这个功能点的，只有关系该功能点的类才会对该接口的方法进行实现，其他不关心的可以不管MergedBeanDefinitionPostProcessor该接口的方法实现，也就是方法可以直接是一个空方法。



AutowiredAnnotationBeanPostProceessor类就是用来对@Autowired和@Value注解进行支持的。该类的注册是在spring上下文对象的构造函数的完成注册的。

```
@Override
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?>
beanType, String beanName) {
   //注解收集核心逻辑
   InjectionMetadata metadata = findAutowiringMetadata(beanName, beanType, null);
   metadata.checkConfigMembers(beanDefinition);
}
```

```
private InjectionMetadata findAutowiringMetadata(String beanName, Class<?> clazz, @Nullable
PropertyValues pvs) {
   // Fall back to class name as cache key, for backwards compatibility with custom callers.
   String cacheKey = (StringUtils.hasLength(beanName) ? beanName : clazz.getName());
   // Quick check on the concurrent map first, with minimal locking.
    //先从缓存拿结果
   InjectionMetadata metadata = this.injectionMetadataCache.get(cacheKey);
   if (InjectionMetadata.needsRefresh(metadata, clazz)) {
      synchronized (this.injectionMetadataCache) {
         metadata = this.injectionMetadataCache.get(cacheKey);
         if (InjectionMetadata.needsRefresh(metadata, clazz)) {
            if (metadata != null) {
               metadata.clear(pvs);
            }
            //主要看这个方法
             //收集的核心逻辑
            metadata = buildAutowiringMetadata(clazz);
            this.injectionMetadataCache.put(cacheKey, metadata);
         }
      }
   }
   return metadata;
}
```

```
//其实这里就是拿到类上所有的field，然后判断field上是否有@Autowired注解，如果有则封装成对象
//寻找field上面的@Autowired注解并封装成对象
ReflectionUtils.doWithLocalFields(targetClass, field -> {
   MergedAnnotation<?> ann = findAutowiredAnnotation(field);
   if (ann != null) {
      if (Modifier.isStatic(field.getModifiers())) {
         if (logger.isInfoEnabled()) {
            logger.info("Autowired annotation is not supported on static fields: " + field);
         }
         return;
      }
      boolean required = determineRequiredStatus(ann);
      currElements.add(new AutowiredFieldElement(field, required));
   }
});
```

注解的收集就完成了，可以看到最终把有@Value的注解的属性包装成了AutowiredFieldElement对象。

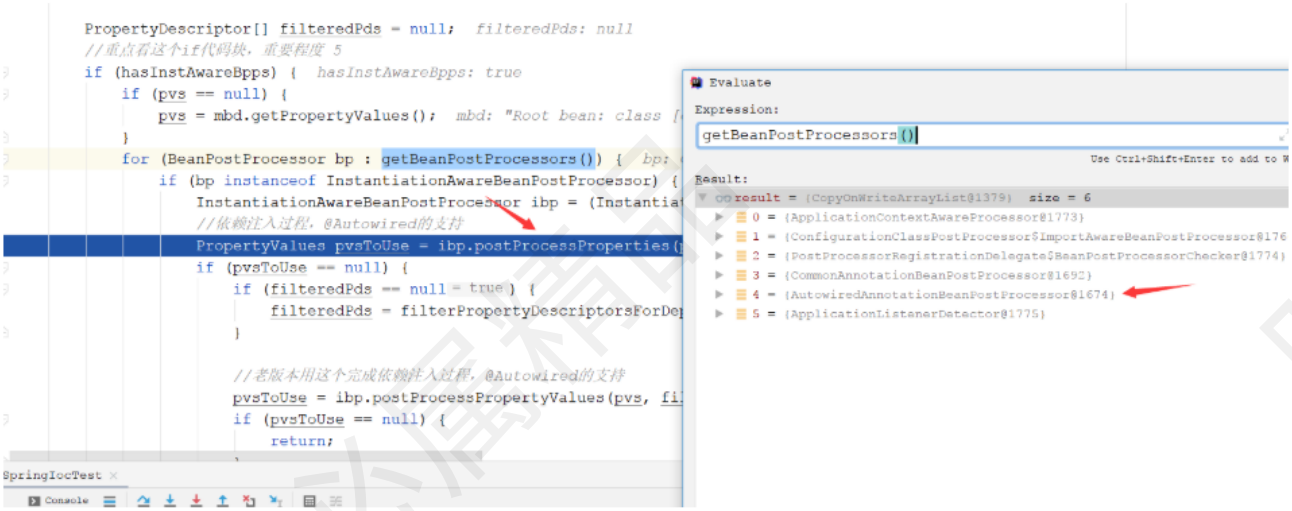
2.2.2.5.2、@Value的依赖注入

依赖注入的核心方法：

```
//ioc di，依赖注入的核心方法，该方法必须看，重要程度：5
populateBean(beanName, mbd, instanceWrapper);
```

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper
bw) {
   if (bw == null) {
      if (mbd.hasPropertyValues()) {
         throw new BeanCreationException(
               mbd.getResourceDescription(), beanName, "Cannot apply property values to null
instance");
      }
      else {
         // Skip property population phase for null instance.
         return;
      }
   }

   // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
   // state of the bean before properties are set. This can be used, for example,
   // to support styles of field injection.
   //这里很有意思，写接口可以让所有类都不能依赖注入
   if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
      for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor)
bp;
            if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
               return;
            }
         }
      }
   }

   PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

   int resolvedAutowireMode = mbd.getResolvedAutowireMode();
   if (resolvedAutowireMode == AUTOWIRE_BY_NAME || resolvedAutowireMode == AUTOWIRE_BY_TYPE)
{
      MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
      // Add property values based on autowire by name if applicable.
      if (resolvedAutowireMode == AUTOWIRE_BY_NAME) {
         autowireByName(beanName, mbd, bw, newPvs);
      }
      // Add property values based on autowire by type if applicable.
      if (resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
         autowireByType(beanName, mbd, bw, newPvs);
      }
      pvs = newPvs;
   }
```

```
   boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
   boolean needsDepCheck = (mbd.getDependencyCheck() !=
AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);

   PropertyDescriptor[] filteredPds = null;
   //重点看这个if代码块，重要程度 5
   if (hasInstAwareBpps) {
      if (pvs == null) {
         pvs = mbd.getPropertyValues();
      }
      for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor)
bp;
            //依赖注入过程，@Autowired的支持。。这里完成了依赖注入
            PropertyValues pvsToUse = ibp.postProcessProperties(pvs,
bw.getWrappedInstance(), beanName);
            if (pvsToUse == null) {
               if (filteredPds == null) {
                  filteredPds = filterPropertyDescriptorsForDependencyCheck(bw,
mbd.allowCaching);
               }

               //老版本用这个完成依赖注入过程，@Autowired的支持
               pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
               if (pvsToUse == null) {
                  return;
               }
            }
            pvs = pvsToUse;
         }
      }
   }
   if (needsDepCheck) {
      if (filteredPds == null) {
         filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
      }
      checkDependencies(beanName, mbd, filteredPds, pvs);
   }

   //这个方法很鸡肋了，建议不看，是老版本用<property name="username" value="Jack"/>
   //标签做依赖注入的代码实现，复杂且无用
   if (pvs != null) {
      applyPropertyValues(beanName, mbd, bw, pvs);
   }
}
```



这里又会调用到AutowiredAnnotationBeanPostProceessor类的方法进行依赖注入。

```
public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) {
   InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs);
   try {
      metadata.inject(bean, beanName, pvs);
   }
   catch (BeanCreationException ex) {
      throw ex;
   }
   catch (Throwable ex) {
      throw new BeanCreationException(beanName, "Injection of autowired dependencies
failed", ex);
   }
   return pvs;
}
```

接下来我把@Value的获取值的流程的核心代码贴出来

```
value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter);
```

```
result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames,
typeConverter);
```

```
//获取@Value中的值，值类似于${enjoy.name}
Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor);
```

```
//根据${cn.username}的value值获取解析后的值
String strVal = resolveEmbeddedValue((String) value);
```

```
@Override
@Nullable
```

```
public String resolveEmbeddedValue(@Nullable String value) {
   if (value == null) {
      return null;
   }
   String result = value;
   for (StringValueResolver resolver : this.embeddedValueResolvers) {
      result = resolver.resolveStringValue(result);
      if (result == null) {
         return null;
      }
   }
   return result;
}
```

这里对应的embeddedValueResolvers集合的值，是**PropertySourcesPlaceholderConfigurer**类中设置进去的。设置代码如下：

```
//@Value的时候有用，把解析器对象设置到BeanFactory中了
beanFactoryToProcess.addEmbeddedValueResolver(valueResolver);
```

可以看到最终调到这里来了

```
   //重点是这个匿名对象 @Value的依赖注入会掉过来
   StringValueResolver valueResolver = strVal -> {  strVal: "${cn.username:jack}"
      String resolved = (this.ignoreUnresolvablePlaceholders ?
            propertyResolver.resolvePlaceholders(strVal) :
            propertyResolver.resolveRequiredPlaceholders(strVal));
      if (this.trimValues) {
          resolved = resolved.trim();
      }
      return (resolved.equals(this.nullValue) ? null : resolved);
   };
```

接下来我们来看看值是如何解析的?

解析的逻辑是这行

```
propertyResolver.resolveRequiredPlaceholders(strVal)
```

接下来

```
helper.replacePlaceholders(text, this::getPropertyAsRawString);
```

关键的地方都写了注释，大家看一下注释

```
protected String parseStringValue(
      String value, PlaceholderResolver placeholderResolver, @Nullable Set<String>
visitedPlaceholders) {

   int startIndex = value.indexOf(this.placeholderPrefix);
   if (startIndex == -1) {
      return value;
```

```
   }

   StringBuilder result = new StringBuilder(value);
   while (startIndex != -1) {
      int endIndex = findPlaceholderEndIndex(result, startIndex);
      if (endIndex != -1) {
         String placeholder = result.substring(startIndex + this.placeholderPrefix.length(),
endIndex);
         String originalPlaceholder = placeholder;
         if (visitedPlaceholders == null) {
            visitedPlaceholders = new HashSet<>(4);
         }
         if (!visitedPlaceholders.add(originalPlaceholder)) {
            throw new IllegalArgumentException(
                  "Circular placeholder reference '" + originalPlaceholder + "' in property
definitions");
         }
         //递归解析 ${${}}
         // Recursive invocation, parsing placeholders contained in the placeholder key.
         placeholder = parseStringValue(placeholder, placeholderResolver,
visitedPlaceholders);
         // Now obtain the value for the fully resolved key...
         //掉到 this::getPropertyAsRawString  ${enjoy.name:jack}
         String propVal = placeholderResolver.resolvePlaceholder(placeholder);
         if (propVal == null && this.valueSeparator != null) {
            int separatorIndex = placeholder.indexOf(this.valueSeparator);
            if (separatorIndex != -1) {
               String actualPlaceholder = placeholder.substring(0, separatorIndex);
               //获取 ":"号后面的默认值
               String defaultValue = placeholder.substring(separatorIndex +
this.valueSeparator.length());
               //":"前面的参数解析
               propVal = placeholderResolver.resolvePlaceholder(actualPlaceholder);
               //如果解析不到，则用默认值
               if (propVal == null) {
                  propVal = defaultValue;
               }
            }
         }
         if (propVal != null) {
            // Recursive invocation, parsing placeholders contained in the
            // previously resolved placeholder value.
            propVal = parseStringValue(propVal, placeholderResolver, visitedPlaceholders);
            result.replace(startIndex, endIndex + this.placeholderSuffix.length(), propVal);
            if (logger.isTraceEnabled()) {
               logger.trace("Resolved placeholder '" + placeholder + "'");
            }
            startIndex = result.indexOf(this.placeholderPrefix, startIndex +
propVal.length());
         }
         else if (this.ignoreUnresolvablePlaceholders) {
            // Proceed with unprocessed value.
            startIndex = result.indexOf(this.placeholderPrefix, endIndex +
```

```
this.placeholderSuffix.length());
        }
        else {
           throw new IllegalArgumentException("Could not resolve placeholder '" +
                 placeholder + "'" + " in value \"" + value + "\"");
        }
        visitedPlaceholders.remove(originalPlaceholder);
     }
     else {
        startIndex = -1;
     }
  }
  return result.toString();
}
```

从这里我们可以看到，其实就是解析${cn.username:jack}把里面的 cn.username解析出来，然后从PropertySources对象中用cn.username作为key去拿值，这里就可以拿到wy这个值了。这里根据cn.username拿值的过程，我们来分析一下，代码就是从这里进去的，根据cn.username获取值

```
    //":"前面的参数解析
    propVal = placeholderResolver.resolvePlaceholder(actualPlaceholder);  propVal: null
    //如果解析不到，则用默认值
    if (propVal == null) {
        propVal = defaultValue;
    }
```

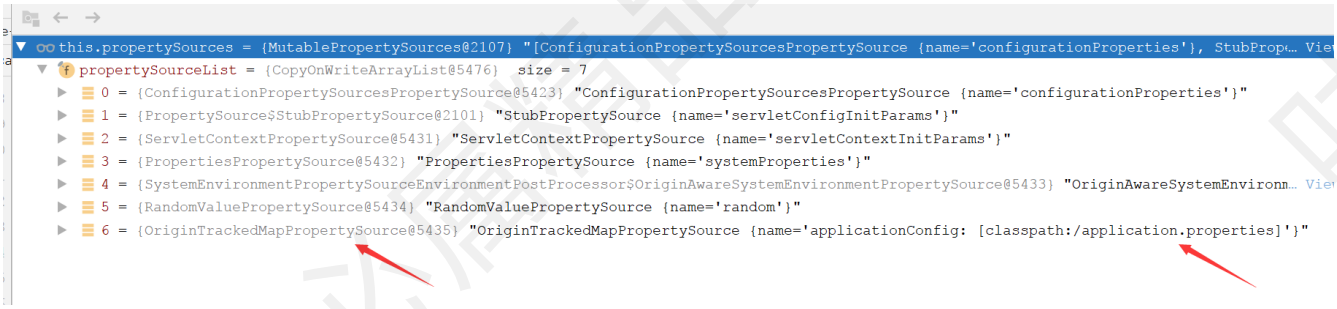
+ "cn.username"

下面我们看看后去值的核心代码：

```
@Nullable
protected <T> T getProperty(String key, Class<T> targetValueType, boolean
resolveNestedPlaceholders) {
   //其实就是从MutablePropertySources中的list中获取每一个PropertySource对象然后调用getProperty方法
   if (this.propertySources != null) {
      for (PropertySource<?> propertySource : this.propertySources) {
         if (logger.isTraceEnabled()) {
            logger.trace("Searching for key '" + key + "' in PropertySource '" +
                  propertySource.getName() + "'");
         }
         //掉用getProperty方法，属性来源有environment和配置文件
         Object value = propertySource.getProperty(key);
         if (value != null) {
            if (resolveNestedPlaceholders && value instanceof String) {
               value = resolveNestedPlaceholders((String) value);
            }
            logKeyFound(key, propertySource, value);
            //参数转换
            return convertValueIfNecessary(value, targetValueType);
         }
      }
   }
   if (logger.isTraceEnabled()) {
      logger.trace("Could not find key '" + key + "' in any property source");
   }
```

```
    return null;
}
```

可以看到就是循环PropertySources对象，然后根据PropertySource.getProperty(key)方法来获取key对应的值的。cn.username的值就是从如下的PropertySource对象中获取的；



那么，application.properties的配置文件是如何被包装成了PropertySource对象的呢？？

2.2.2.6、application.properties

下面我们重点来介绍一下application.properties的加载流程

application.properties配置文件包装，这个是我们必须要分析的，接下来我们就来分析一下在springboot中，application.properties，application.yml等这种配置文件是如何包装成PropertySource对象的。

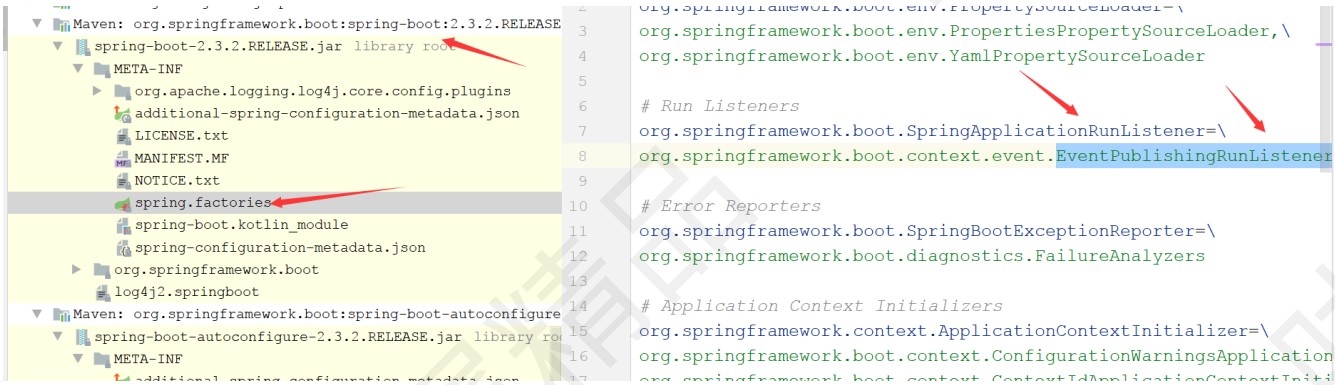
首先我们还是从SpringApplication.run方法看起，看看run方法中的这行代码

```
ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
```

application.properties文件的加载就在这行代码中

```
    private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners,
            ApplicationArguments applicationArguments) {
        // Create and configure the environment
        ConfigurableEnvironment environment = getOrCreateEnvironment();
        configureEnvironment(environment, applicationArguments.getSourceArgs());
        ConfigurationPropertySources.attach(environment);
        listeners.environmentPrepared(environment);
        bindToSpringApplication(environment);
        if (!this.isCustomEnvironment) {
            environment = new EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(env
                    deduceEnvironmentClass());
        }
        ConfigurationPropertySources.attach(environment);
        return environment;
    }
```

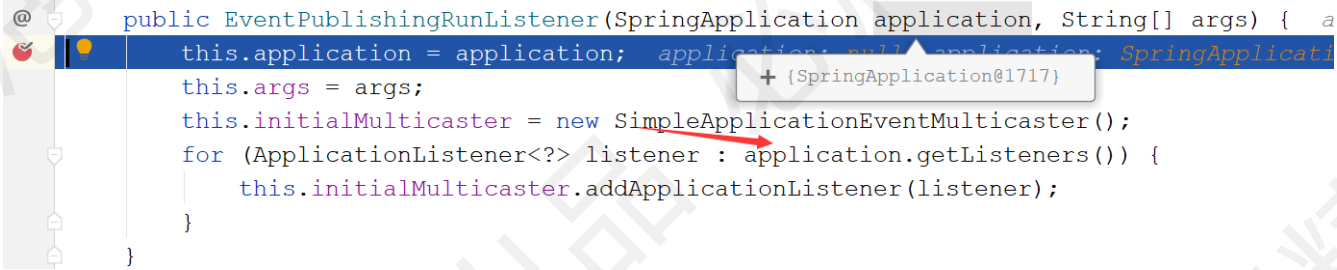
```
void environmentPrepared(ConfigurableEnvironment environment) {
    //listener实例就是EventPublishingRunListener,SPI加载进来的
    for (SpringApplicationRunListener listener : this.listeners) {
      listener.environmentPrepared(environment);
    }
}
```



接着

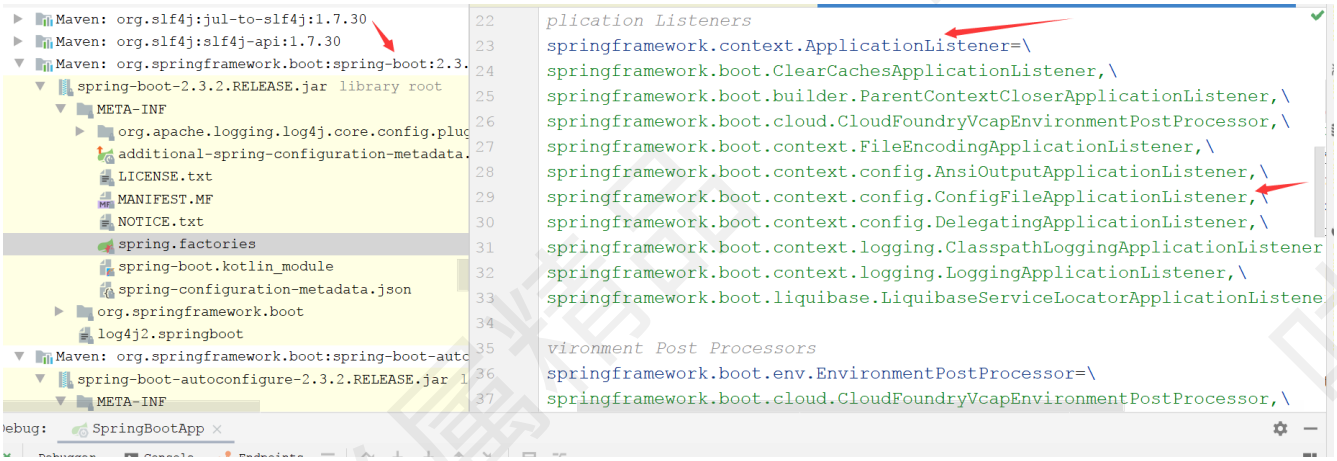
```
@Override
public void environmentPrepared(ConfigurableEnvironment environment) {
   this.initialMulticaster
         .multicastEvent(new ApplicationEnvironmentPreparedEvent(this.application,
this.args, environment));
}
```

initialMulticaster是SimpleApplicationEventMulticaster类型的对象，看看EventPublishingRunListener的构造函数，



可以看到，SimpleApplicationEventMulticaster中的Listener对象是从SpringApplication对象中获取到的，那么SpringApplication中的Listener对象是怎么来的呢？ spi的方式加载的。如下

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
   this.resourceLoader = resourceLoader;
   Assert.notNull(primarySources, "PrimarySources must not be null");
   this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
   this.webApplicationType = WebApplicationType.deduceFromClasspath();
   setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
   //这里就设置类listener对象，是从spring.factories文件中获取ApplicationListener类型的类
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
   this.mainApplicationClass = deduceMainApplicationClass();
}
```



其中**ConfigFileApplicationListener**就是加载application.properties的关键类。

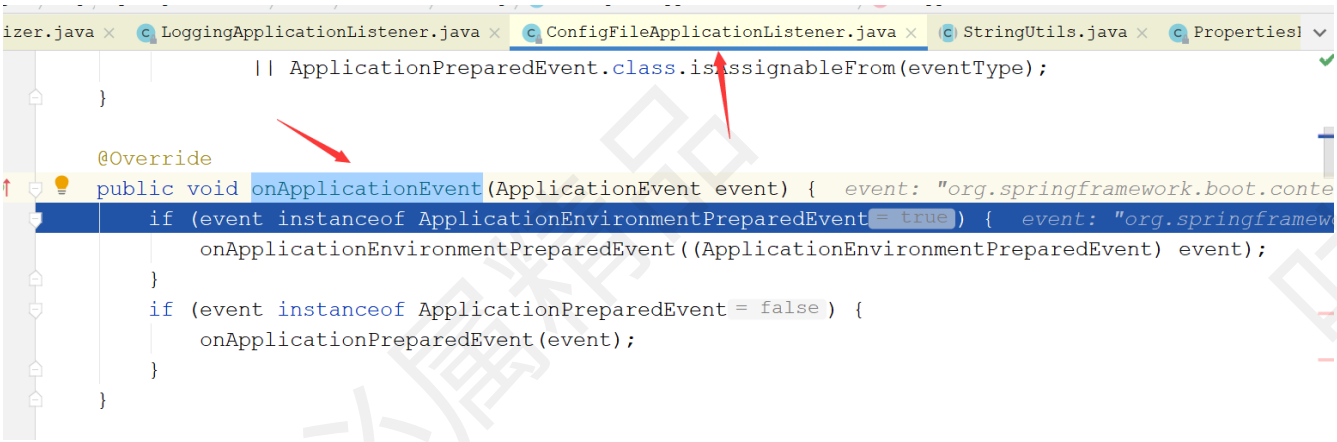
我们接着核心流程玩下走

```
@Override
public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType)
{
   ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
   Executor executor = getTaskExecutor();
    //这里的listener对象就是SpringApplication中的listener对象，从spi的方式加载进来的
   for (ApplicationListener<?> listener : getApplicationListeners(event, type)) {
      if (executor != null) {
         executor.execute(() -> invokeListener(listener, event));
      }
      else {
          //走这里
         invokeListener(listener, event);
      }
   }
}
```

```
protected void invokeListener(ApplicationListener<?> listener, ApplicationEvent event) {
   ErrorHandler errorHandler = getErrorHandler();
   if (errorHandler != null) {
      try {
         doInvokeListener(listener, event);
      }
      catch (Throwable err) {
         errorHandler.handleError(err);
      }
   }
   else {
      doInvokeListener(listener, event);
   }
}
```

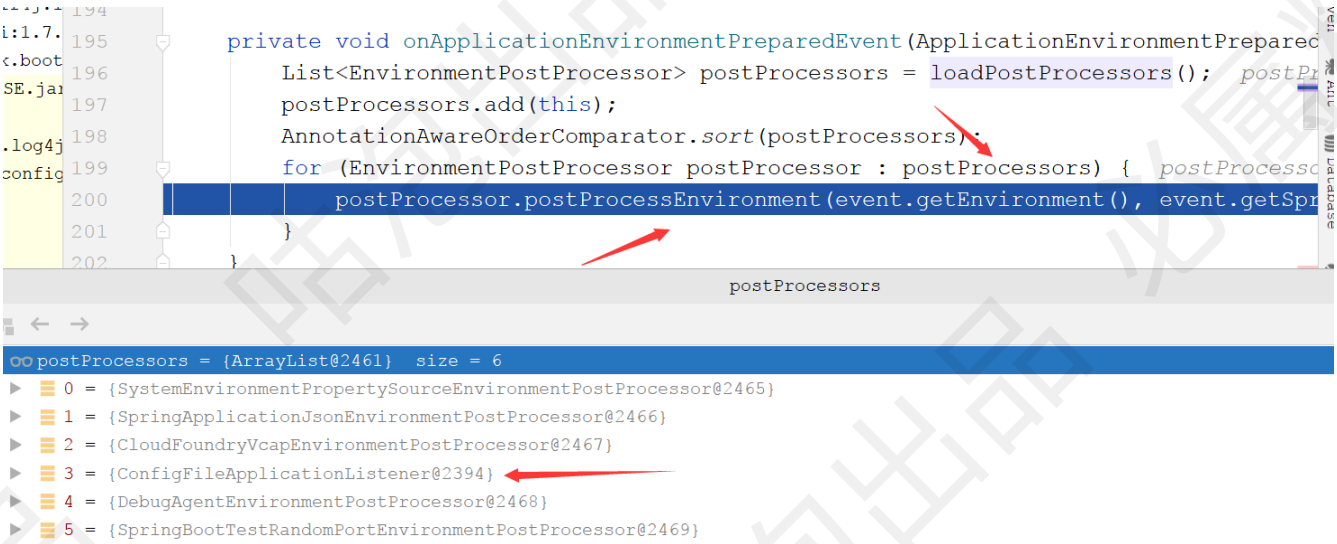
从这里我们就顺利的走到了ConfigFileApplicationListener类的onApplicationEvent方法，事件类型就是

ApplicationEnvironmentPreparedEvent




```
private void onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent
event) {
    //这里也是spi的方式获取实例
   List<EnvironmentPostProcessor> postProcessors = loadPostProcessors();
   postProcessors.add(this);
   AnnotationAwareOrderComparator.sort(postProcessors);
   for (EnvironmentPostProcessor postProcessor : postProcessors) {
      postProcessor.postProcessEnvironment(event.getEnvironment(),
event.getSpringApplication());
   }
}
```

EnvironmentPostProcessor对象我们只看ConfigFileApplicationListener类就可以了



接下来

```
@Override
public void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication
application) {
   addPropertySources(environment, application.getResourceLoader());
}
```

```
protected void addPropertySources(ConfigurableEnvironment environment, ResourceLoader
resourceLoader) {
   RandomValuePropertySource.addToEnvironment(environment);
   //核心代码
   new Loader(environment, resourceLoader).load();
}
```

```
void load() {
   FilteredPropertySource.apply(this.environment, DEFAULT_PROPERTIES,
LOAD_FILTERED_PROPERTY,
         (defaultProperties) -> {
            this.profiles = new LinkedList<>();
            this.processedProfiles = new LinkedList<>();
            this.activatedProfiles = false;
            this.loaded = new LinkedHashMap<>();
            initializeProfiles();
            while (!this.profiles.isEmpty()) {
               Profile profile = this.profiles.poll();
               if (isDefaultProfile(profile)) {
                  addProfileToEnvironment(profile.getName());
               }
                //核心代码
               load(profile, this::getPositiveProfileFilter,
                     addToLoaded(MutablePropertySources::addLast, false));
               this.processedProfiles.add(profile);
            }
            load(null, this::getNegativeProfileFilter,
addToLoaded(MutablePropertySources::addFirst, true));
            addLoadedPropertySources();
            applyActiveProfiles(defaultProperties);
         });
}
```

```
private void load(Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer
consumer) {
    //"classpath:/,classpath:/config/,file:./,file:./config/*/,file:./config/";默认的加载路径
   getSearchLocations().forEach((location) -> {
      boolean isDirectory = location.endsWith("/");
      Set<String> names = isDirectory ? getSearchNames() : NO_SEARCH_NAMES;
      names.forEach((name) -> load(location, name, profile, filterFactory, consumer));
   });
}
```

```
private void load(PropertySourceLoader loader, String location, Profile profile,
DocumentFilter filter,
      DocumentConsumer consumer) {
   Resource[] resources = getResources(location);
   for (Resource resource : resources) {
      try {
         if (resource == null || !resource.exists()) {
            if (this.logger.isTraceEnabled()) {
               StringBuilder description = getDescription("Skipped missing config ",
location, resource,
                     profile);
               this.logger.trace(description);
            }
            continue;
         }
         if (!StringUtils.hasText(StringUtils.getFilenameExtension(resource.getFilename())))
{
            if (this.logger.isTraceEnabled()) {
               StringBuilder description = getDescription("Skipped empty config extension ",
location,
                     resource, profile);
               this.logger.trace(description);
            }
            continue;
         }
         String name = "applicationConfig: [" + getLocationName(location, resource) + "]";
          //加载配置文件，核心代码
         List<Document> documents = loadDocuments(loader, name, resource);
         if (CollectionUtils.isEmpty(documents)) {
            if (this.logger.isTraceEnabled()) {
               StringBuilder description = getDescription("Skipped unloaded config ",
location, resource,
                     profile);
               this.logger.trace(description);
            }
            continue;
         }
         List<Document> loaded = new ArrayList<>();
         for (Document document : documents) {
```

```
            if (filter.match(document)) {
               addActiveProfiles(document.getActiveProfiles());
               addIncludedProfiles(document.getIncludeProfiles());
               loaded.add(document);
            }
         }
         Collections.reverse(loaded);
         if (!loaded.isEmpty()) {
            loaded.forEach((document) -> consumer.accept(profile, document));
            if (this.logger.isDebugEnabled()) {
               StringBuilder description = getDescription("Loaded config file ", location,
resource,
                     profile);
               this.logger.debug(description);
            }
         }
      }
      catch (Exception ex) {
         StringBuilder description = getDescription("Failed to load property source from ",
location,
               resource, profile);
         throw new IllegalStateException(description.toString(), ex);
      }
   }
}
```

```
private List<Document> loadDocuments(PropertySourceLoader loader, String name, Resource
resource)
      throws IOException {
   DocumentsCacheKey cacheKey = new DocumentsCacheKey(loader, resource);
   List<Document> documents = this.loadDocumentsCache.get(cacheKey);
   if (documents == null) {
       //核心代码
      List<PropertySource<?>> loaded = loader.load(name, resource);
      documents = asDocuments(loaded);
      this.loadDocumentsCache.put(cacheKey, documents);
   }
   return documents;
}
```

```
@Override
public List<PropertySource<?>> load(String name, Resource resource) throws IOException {   name: "applicationConfi
    Map<String, ?> properties = loadProperties(resource);   resource: "class path resource [application.properties
    if (properties.isEmpty()) {
        return Collec
    }
    return Collections
            .singletonList(new OriginTrackedMapPropertySource(name, Collections.unmodifiableMap(properties),
}
```

+ {ClassPathResource@3144} "class path resource [application.properties]"

可以看到，最终是通过把application.properties加载进来了，并且最终包装成了**OriginTrackedMapPropertySource**对象，OriginTrackedMapPropertySource本身就是一个PropertySource对象，PropertySource对象前面有讲到，是一种数据源。。这里我们搞明白了，一个application.properties配置文件生成了一个PropertySource对象了，那么这个对象是如何加入到Environment对象中的？我们接着看看

```
    void load() {
        FilteredPropertySource.apply(this.environment, DEFAULT_PROPERTIES, LOAD_FILTERED_PROPERTY,
                (defaultProperties) -> {
                    this.profiles = new LinkedList<>();
                    this.processedProfiles = new LinkedList<>();
                    this.activatedProfiles = false;
                    this.loaded = new LinkedHashMap<>();
                    initializeProfiles();
                    while (!this.profiles.isEmpty()) {
                        Profile profile = this.profiles.poll();
                        if (isDefaultProfile(profile)) {
                            addProfileToEnvironment(profile.getName());
                        }
                        load(profile, this::getPositiveProfileFilter,
                                addToLoaded(MutablePropertySources::addLast, checkForExisting: false));
                        this.processedProfiles.add(profile);
                    }
                    load( profile: null, this::getNegativeProfileFilter, addToLoaded(MutablePropertySources::ac
                    addLoadedPropertySources();
                    applyActiveProfiles(defaultProperties);
                });
    }
```

在这行代码里面，把生成了PropertySource对象放入到了Environment中

```
private void addLoadedPropertySources() {
   MutablePropertySources destination = this.environment.getPropertySources();
   List<MutablePropertySources> loaded = new ArrayList<>(this.loaded.values());
   Collections.reverse(loaded);
   String lastAdded = null;
   Set<String> added = new HashSet<>();
   for (MutablePropertySources sources : loaded) {
      for (PropertySource<?> source : sources) {
         if (added.add(source.getName())) {
             //这里存到了Environment中了
            addLoadedPropertySource(destination, lastAdded, source);
            lastAdded = source.getName();
         }
      }
   }
}
```

OK，这样就完成了一个application.properties配置文件的加载了。