Spring boot 源码分享—配置文件的读取

1.简介

Spring boot 中引用配置文件中的值可以通过注解@ConfigurationProperties，加在某个类或者某个引用了注解@Configuration 的类的某个引用了注解@Bean 的方法上，可以绑定配置文件中的值。在使用的时，在需要获取的类上添加@EnableConfigurationProperties({Config.class})的注解，并通过@autoWired 的注解来取得相应的值。

和配置相关的代码主要集中在包 org.springframework.boot.context.properties 下。

Spring boot 里的实例：

```
@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)
public class ServerProperties {

    /**
     * Server HTTP port.
     */
    private Integer port;

    /**
     * Network address to which the server should bind.
     */
    private InetAddress address;

    @NestedConfigurationProperty
    private final ErrorProperties error = new ErrorProperties();
```

2.源码分析

2.1 注解 ConfigurationProperties

可以加在类或者类的方法上，通过指定前缀来获取指定文件上的相应的值。

```
// 说明这个注解可以被用在类/接口上和方法上
@Target({ ElementType.TYPE, ElementType.METHOD })
// 注解的保留时间，RUNTIME 说明一直到运行时都保留，可以通过反射来获取注解信息
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ConfigurationProperties {
   // 配置文件中的前缀申明，在配置文件中以 prefix 开头的参数会被读取
   @AliasFor("prefix")
   String value() default "";
   @AliasFor("value")
   String prefix() default "";
// 标志着配置里遇到非法类型是否需要忽略(比如 Integer 类型传入无法解析为数字的字符串,设置为 false 则在启动时报错,设置为 true 则会忽略无法转换的值）
   boolean ignoreInvalidFields() default false;
   // 标志着配置中未知的变量是否需要忽略
   boolean ignoreUnknownFields() default true;
}
```

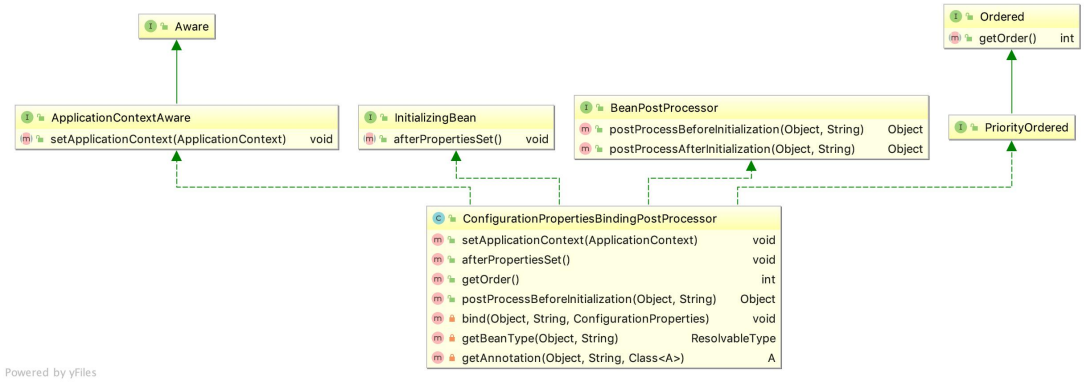
2.2 类 ConfigurationPropertiesBindingPostProcessor

实现了接口 BeanPostProcessor 的 postProcessBeforeInitialization 方法，在方法里尝试获取注解 ConfigurationProperties 并对其进行解析。postProcessBeforeInitialization 方法会在 bean 的初始化回调之前被调用，可以在这个方法里对 bean 进行 warp 操作。另外，并没有实现 postProcessAfterInitialization 方法，而是沿用了接口对默认实现，即返回原始的 bean。

还实现了接口 InitializingBean 的 afterPropertiesSet 方法，afterPropertiesSet 方法会在 BeanFactory 设置完它对全部属性后被调用。

在调用顺序上，方法 postProcessBeforeInitialization 要早于 afterPropertiesSet 方法，而 afterPropertiesSet 方法要早于方法 postProcessAfterInitialization。

在 postProcessBeforeInitialization 方法中，如果 bean 含有注解 ConfigurationProperties ，那么则会调用 bind 函数进行操作，bind 函数又进一步调用类 ConfigurationPropertiesBinder 的 bind 函数。



```
public class ConfigurationPropertiesBindingPostProcessor implements
BeanPostProcessor,PriorityOrdered, ApplicationContextAware, InitializingBean
{
@Override
public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
   ConfigurationProperties annotation = getAnnotation(bean, beanName,
         ConfigurationProperties.class);
   if (annotation != null) {
      bind(bean, beanName, annotation);
   }
   return bean;
}

private void bind(Object bean, String beanName, ConfigurationProperties
annotation) {
   ResolvableType type = getBeanType(bean, beanName);
   Validated validated = getAnnotation(bean, beanName, Validated.class);
   Annotation[] annotations = (validated != null)
         ? new Annotation[] { annotation, validated }
         : new Annotation[] { annotation };
   // 转换成能被binder 处理的对象
   Bindable<?> target = Bindable.of(type).withExistingValue(bean)
         .withAnnotations(annotations);
   try {
      this.configurationPropertiesBinder.bind(target);
   }
   catch (Exception ex) {
      throw new ConfigurationPropertiesBindException(beanName, bean,
annotation, ex);
   }
}
```

2.3 类 ConfigurationPropertiesBinder

处理注解 ConfigurationProperties 的 binder 类。会根据注解 ConfigurationProperties 中 ignoreInvalidFields 和 ignoreUnknownFields 两个值的不同来确定不同的 BinderHandler。确定完 binderhanlder 后调用 Binder 的 bind 函数进行绑定。

```
public void bind(Bindable<?> target) {
   ConfigurationProperties annotation = target
         .getAnnotation(ConfigurationProperties.class);
   Assert.state(annotation != null,
         () -> "Missing @ConfigurationProperties on " + target);
   List<Validator> validators = getValidators(target);
   BindHandler bindHandler = getBindHandler(annotation, validators);
   getBinder().bind(annotation.prefix(), target, bindHandler);
}
private BindHandler getBindHandler(ConfigurationProperties annotation,
      List<Validator> validators) {
   BindHandler handler = new IgnoreTopLevelConverterNotFoundBindHandler();
   if (annotation.ignoreInvalidFields()) {
      handler = new IgnoreErrorsBindHandler(handler);
   }
   if (!annotation.ignoreUnknownFields()) {
      UnboundElementsSourceFilter filter = new UnboundElementsSourceFilter();
      handler = new NoUnboundElementsBindHandler(handler, filter);
   }
   if (!validators.isEmpty()) {
      handler = new ValidationBindHandler(handler,
            validators.toArray(new Validator[0]));
   }
   for (ConfigurationPropertiesBindHandlerAdvisor advisor :
getBindHandlerAdvisors()) {
      handler = advisor.apply(handler);
   }
   return handler;
}
```

2.4 接口 BinderHandler

被用在处理绑定的逻辑回调上。接口一共有四个方法，分别是 onStart、onSuccess、onFailure 和 onFinish。

```
public interface BindHandler {
   // 默认实现的什么都不做的handler
   BindHandler DEFAULT = new BindHandler() {
   };
    // 在绑定开始但是结果还未决定之前被调用
   default <T> Bindable<T> onStart(ConfigurationPropertyName name,
Bindable<T> target, BindContext context) {
      return target;
   }
    // 在绑定成功后被调用
   default Object onSuccess(ConfigurationPropertyName name, Bindable<?>
target, BindContext context, Object result) {
      return result;
   }
    // 在绑定过程中发生任何错误都会被调用
   default Object onFailure(ConfigurationPropertyName name, Bindable<?>
target,
         BindContext context, Exception error) throws Exception {
      throw error;
   }
    // 在绑定结束后被调用，无论结果如何
   default void onFinish(ConfigurationPropertyName name, Bindable<?>
target,
         BindContext context, Object result) throws Exception {
   }

}
```

三个主要的实现类

2.4.1 类 IgnoreTopLevelConverterNotFoundBindHandler

忽略顶级的 ConverterNotFoundException 错误，实现方法 onFailure，当发生顶级 ConverterNotFoundException 错误时，返回 null，其他情况依然抛出异常。默认情况下被使用，即 ignoreInvalidFields 为 false，ignoreUnknownFields 为 true 时。

```
@Override
public Object onFailure(ConfigurationPropertyName name, Bindable<?>
target,
      BindContext context, Exception error) throws Exception {
   if (context.getDepth() == 0 && error instanceof
ConverterNotFoundException) {
      return null;
   }
   throw error;
}
```

2.4.2 类 IgnoreErrorsBindHandler

忽略所有的错误，实现方法 onFailure，当绑定发生错误时，返回 null。注解的 ignoreInvalidFields 的值为 true 时被启用。

```
public class IgnoreErrorsBindHandler extends AbstractBindHandler {
   @Override
   public Object onFailure(ConfigurationPropertyName name, Bindable<?>
target,
         BindContext context, Exception error) throws Exception {
      return (target.getValue() != null) ? target.getValue().get() : null;
   }
}
```

2.4.3 NoUnboundElementsBindHandler

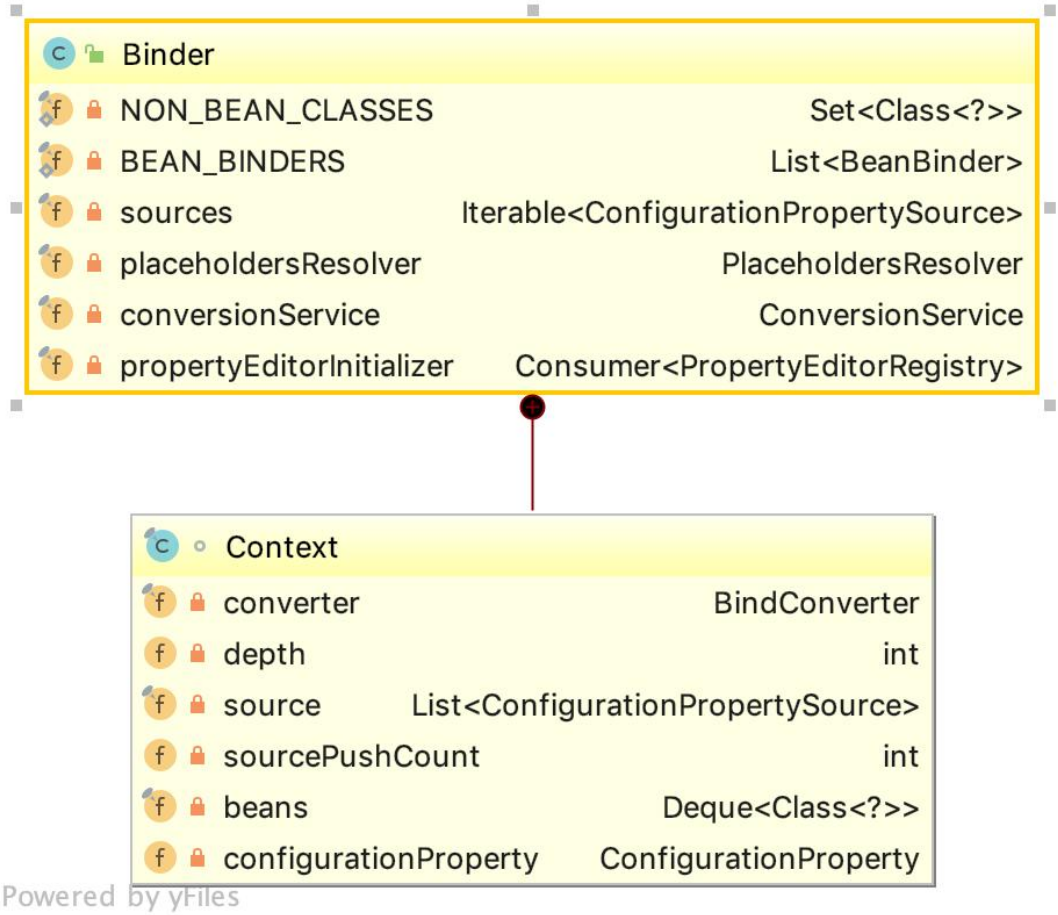
强制要求配置文件中的项都要有匹配都属性，否则会报错。重写类方法 onSuccess 和方法 onFinish。在 ignoreUnknownFields 为 false 时被启用，但是如果同时 ignoreInvalidFields 为 true，则实际上不起作用，因为此时 onFailure 会忽略所有都错误。

```
@Override
public Object onSuccess(ConfigurationPropertyName name, Bindable<?> target,
BindContext context, Object result) {
   this.boundNames.add(name);
   return super.onSuccess(name, target, context, result);
}

@Override
public void onFinish(ConfigurationPropertyName name, Bindable<?> target,
      BindContext context, Object result) throws Exception {
   if (context.getDepth() == 0) {
      checkNoUnboundElements(name, context);
   }
}
```

2.5 类 Binder

使用一个或多个 ConfigurationPropertySource 对目标进行绑定。



2.5.1 bind 方法

检查参数并创建 context 对象，然后进一步调用 bind 方法，最后将 bind 方法的返回值转化为 BindResult 对象并返回。类 Context 是 Binder 的内部类，实现了接口 BindContext。

```
public <T> BindResult<T> bind(ConfigurationPropertyName name, Bindable<T> target,
BindHandler handler) {
   Assert.notNull(name, "Name must not be null");
   Assert.notNull(target, "Target must not be null");
   handler = (handler != null) ? handler : BindHandler.DEFAULT;
   Context context = new Context();
   T bound = bind(name, target, handler, context, false);
   return BindResult.of(bound);
}
```

先将 context 中的 configurationProperty 变量设为 null，然后调用 handler 的 onStart 方法，onStart 方法不做其他操作直接返回 target。然后调用方法 bindObject，最后调用 handleBindResult 方法，并返回方法返回值。

```
protected final <T> T bind(ConfigurationPropertyName name, Bindable<T> target,
      BindHandler handler, Context context, boolean allowRecursiveBinding) {
   context.clearConfigurationProperty();
   try {
      target = handler.onStart(name, target, context);
      if (target == null) {
         return null;
      }
      Object bound = bindObject(name, target, handler, context,
            allowRecursiveBinding);
      return handleBindResult(name, target, handler, context, bound);
   }
   catch (Exception ex) {
      return handleBindError(name, target, handler, context, ex);
   }
}
```

2.5.2 handleBindResult 方法

bindObject 方法比较复杂，所以先看一下 handleBindResult 方法：

这个方法主要是对 bindObject 返回的对象进行校验，如果不为 null，则调用 handle 的 onSuccess 方法，然后尝试把 result 转化成 target 类型。之后会调用 handle 的 onFinish 方法，然后再一次对结果进行转换。最后返回结果。

```
private <T> T handleBindResult(ConfigurationPropertyName name, Bindable<T>
target, BindHandler handler, Context context, Object result) throws Exception
{
   if (result != null) {
      result = handler.onSuccess(name, target, context, result);
      result = context.getConverter().convert(result, target);
   }
   handler.onFinish(name, target, context, result);
   return context.getConverter().convert(result, target);
}
```

2.5.3 bindObject 方法

首先尝试获取 propertyName 对应的 property，如果为 null 并且这个属性是配置里的最后一项，那么返回 null；然后检查是否为集合类型，如果是，则调用 bindAggregate 方法并返回；如果 property 不为 null，则调用 bindProperty 方法并返回，为 null 则调用 bindBean 方法并返回。

```
private <T> Object bindObject(ConfigurationPropertyName name, Bindable<T> target,
BindHandler handler, Context context, boolean allowRecursiveBinding) {
   ConfigurationProperty property = findProperty(name, context);
   if (property == null && containsNoDescendantOf(context.getSources(), name)) {
      return null;
   }
   AggregateBinder<?> aggregateBinder = getAggregateBinder(target, context);
   if (aggregateBinder != null) {
      return bindAggregate(name, target, handler, context, aggregateBinder);
   }
   if (property != null) {
      try {
         return bindProperty(target, context, property);
      }
      catch (ConverterNotFoundException ex) {
         // We might still be able to bind it as a bean
         Object bean = bindBean(name, target, handler, context,
               allowRecursiveBinding);
         if (bean != null) {
            return bean;
         }
         throw ex;
      }
   }
   return bindBean(name, target, handler, context, allowRecursiveBinding);
}
```

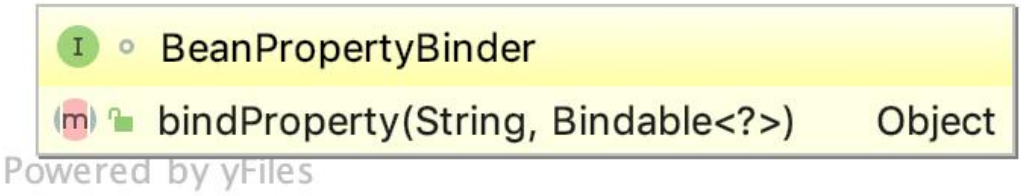
2.5.4 bindProperty 方法

把 context 的 **configurationProperty** 变量设为 proeprty，然后从 property 中拿出 value，处理一下其中的占位符，然后将其转化成 target 的类型并返回。

```
private <T> Object bindProperty(Bindable<T> target, Context context,
      ConfigurationProperty property) {
   context.setConfigurationProperty(property);
   Object result = property.getValue();
   result = this.placeholdersResolver.resolvePlaceholders(result);
   result = context.getConverter().convert(result, target);
   return result;
}
```

2.5.5 bindBean 方法

先检查是否能绑定为 bean，不能则直接返回 null。接着实例化了一个接口 BeanPropertyBinder 的对象，接口 BeanPropertyBinder 包含一个方法：bindProperty。实例 propertyBinder 以 Binder 的 bind 方法来实现 BeanPropertyBinder 接口的 bindProperty 方法。



最后调用了 context 的 withBean 方法。

```
private Object bindBean(ConfigurationPropertyName name, Bindable<?> target,
BindHandler handler, Context context, boolean allowRecursiveBinding) {
   if (containsNoDescendantOf(context.getSources(), name)
         || isUnbindableBean(name, target, context)) {
      return null;
   }
   BeanPropertyBinder propertyBinder = (propertyName, propertyTarget) ->
bind( name.append(propertyName), propertyTarget, handler, context, false);
   Class<?> type = target.getType().resolve(Object.class);
   if (!allowRecursiveBinding && context.hasBoundBean(type)) {
      return null;
   }
   return context.withBean(type, () -> {
      Stream<?> boundBeans = BEAN_BINDERS.stream()
            .map((b) -> b.bind(name, target, context, propertyBinder));
      return boundBeans.filter(Objects::nonNull).findFirst().orElse(null);
   });
}
```

Context 的 withBean 方法最终会调用 supplier 的 get 方法，也就是传入参数中的第二个 lambada 表达式。而 lambada 表达式中又调用了 BeanBinder 接口的 bind 方法。

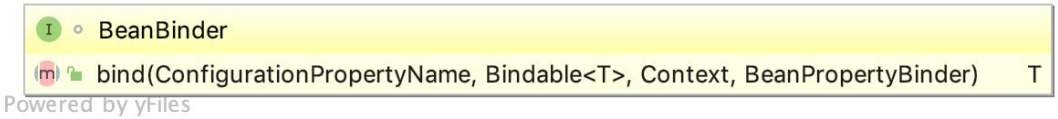
```
private <T> T withBean(Class<?> bean, Supplier<T> supplier) {
   this.beans.push(bean);
   try {
      return withIncreasedDepth(supplier);
   }
   finally {
      this.beans.pop();
   }
}
```

```
private <T> T withIncreasedDepth(Supplier<T> supplier) {
   increaseDepth();
   try {
      return supplier.get();
   }
   finally {
      decreaseDepth();
   }
}
```

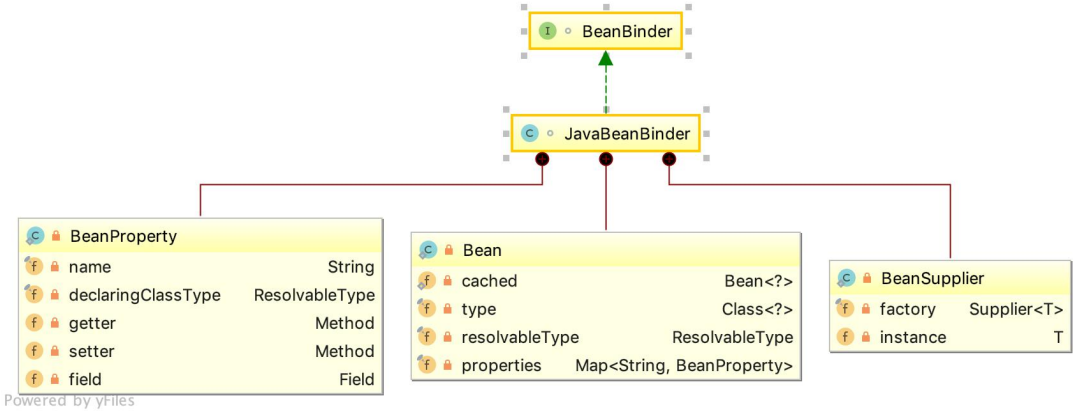
2.6 接口 BeanBinder

2.6.1 接口申明

如图所示，包含方法 bind，将返回绑定好的 bean：



2.6.2 实现类 JavaBeanBinder



bind 方法

第一个 bind 方法：

获取 Bean 和 BeanSupplier 的实例，这里 Bean 和 BeanSupplier 都是 JavaBeanBinder 的内部类。

然后通过进一步调用 bind 方法，把所有属性都放置在 BeanSupplier 的 instance 里。

```
@Override
public <T> T bind(ConfigurationPropertyName name, Bindable<T> target, Context
context, BeanPropertyBinder propertyBinder) {
   boolean hasKnownBindableProperties = hasKnownBindableProperties(name, context);
   Bean<T> bean = Bean.get(target, hasKnownBindableProperties);
   if (bean == null) {
      return null;
   }
   BeanSupplier<T> beanSupplier = bean.getSupplier(target);
   boolean bound = bind(propertyBinder, bean, beanSupplier);
   return (bound ? beanSupplier.get() : null);
}
```

第二个 bind 方法：

从 Bean 中获取全部的属性，并进一步调用 bind 方法对每一个属性进行配置。只要有任何一个属性的值不为 null，都会返回 true。

```
private <T> boolean bind(BeanPropertyBinder propertyBinder, Bean<T> bean,
      BeanSupplier<T> beanSupplier) {
   boolean bound = false;
   for (BeanProperty beanProperty : bean.getProperties().values()) {
      bound |= bind(beanSupplier, propertyBinder, beanProperty);
   }
   return bound;
}
```

第三个 bind 方法：

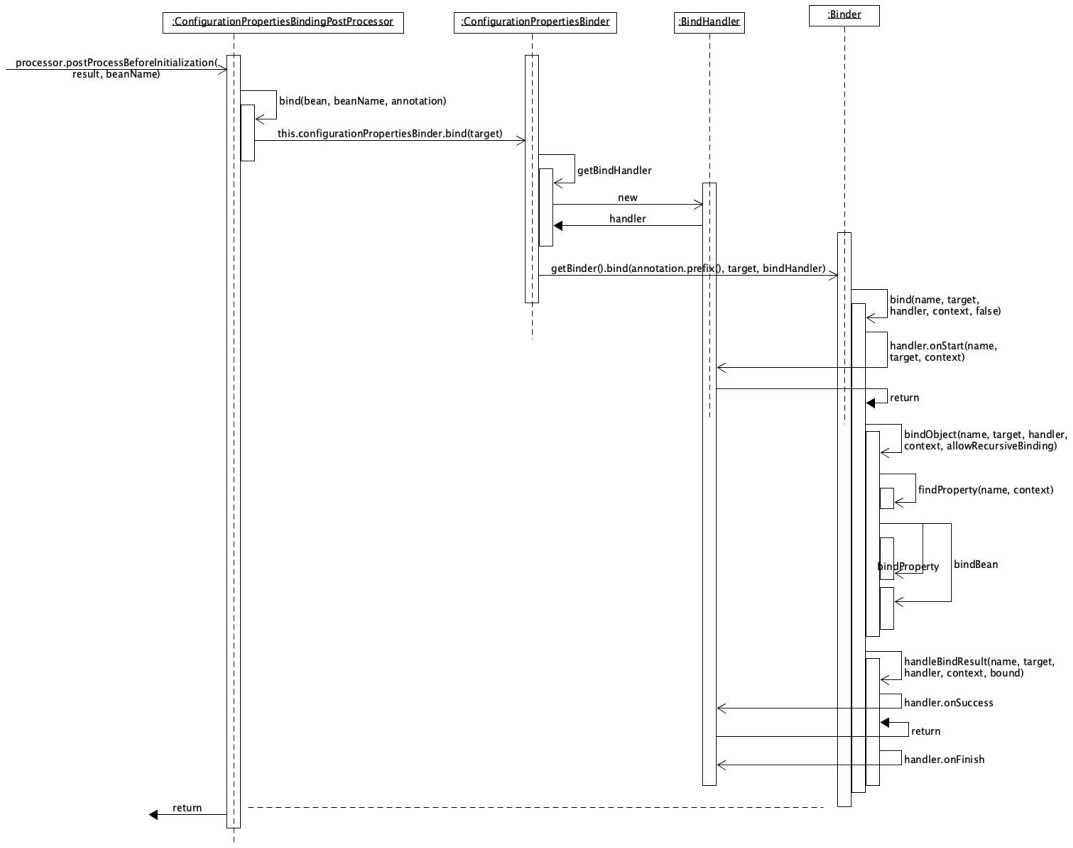
通过 BeanPropertyBinder 接口的 bindProperty 方法获取到了配置文件里的值，其中传入的 propertyBinder 正是之前在 Binder 里生成的 BeanPropertyBinder 对象，它的 bindProperty 方法的实现是调用了 Binder 的 bind 方法，所以它又回到了 Binder 的 bind 方法中去，再次判断新的 propertyName 需要调用哪个方法，是返回 null，还是返回 bindAggregate 的结果，还是返回 bindProperty 的结果，还是再次调用 bindBean 方法。

获取到配置的值以后，通过 BeanProperty 把值放在 BeanSupplier 中。

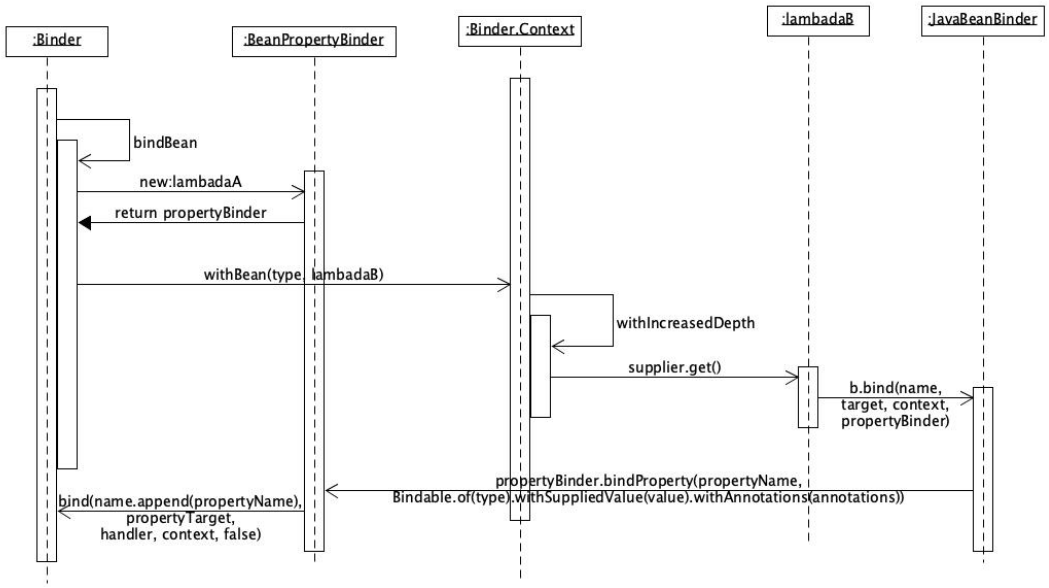
```
private <T> boolean bind(BeanSupplier<T> beanSupplier,
      BeanPropertyBinder propertyBinder, BeanProperty property) {
   String propertyName = property.getName();
   ResolvableType type = property.getType();
   Supplier<Object> value = property.getValue(beanSupplier);
   Annotation[] annotations = property.getAnnotations();
   Object bound = propertyBinder.bindProperty(propertyName,
Bindable.of(type).withSuppliedValue(value).withAnnotations(annotations));
   if (bound == null) {
      return false;
   }
   if (property.isSettable()) {
      property.setValue(beanSupplier, bound);
   }
   else if (value == null || !bound.equals(value.get())) {
      throw new IllegalStateException(
            "No setter found for property: " + property.getName());
   }
   return true;
}
```

2.7 方法调用链

总体一览



bindBean 调用详情



3.实例

3.1 在 application.properties 中加入自定义的属性

创建类 AppConfig，加入注解 ConfigurationProperties，并指定相应的 prefix，那么在 applica.properties 中以 prefix 开头的属性便会被读取。如果有内部类需要申明为静态内部类的，

```
@ConfigurationProperties(prefix = "mini")
public class AppConfig {
    private String appId;
    private String secret;
    private String channel_id;
    private Open open;
    private Person person;
    ...getter/setter
    public static class Open {
        private String deviceId;
        private String secret;
        private String channel;
        ...getter/setter
    }
}
```

Application.properties 中的变量设置如下,变量名的写法可以上驼峰式也可以上用下划线或者短线分隔开：

```
#mini.app_id=wxhkshjbx
#mini.app-id=wxhkshjbx
mini.appId=wxhkshjbx
mini.secret=yu&hjh56
mini.channel_id=4332
mini.open.channel=1001
mini.open.secret=jhs7875%
mini.open.device-id=1837287387
```

在需要使用的类上加上注解@EnableConfigurationProperties({AppConfig.class})

并通过注解@Autowired 来引用在 application.properties 中设置的属性。

```
@RestController
@EnableConfigurationProperties({AppConfig.class})
public class MiniController {
    @Autowired
    AppConfig appConfig;

    @RequestMapping("/mini")
    AppConfig mini() {
        return appConfig;
    }
}
```

3.2 在其他文件中加入自定义的属性

除去上述步骤操作以外，在类 ExConfig 上还需要额外添加两个注解@Component 和@PropertySource("classpath:/ex.properties")，其中需要在 PropertySource 注解中指明要读取的文件的路径

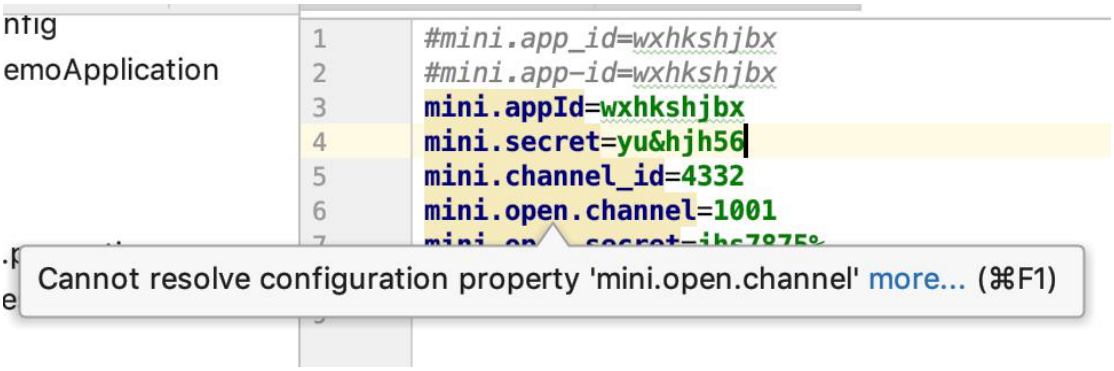
```
@Component
@PropertySource("classpath:/ex.properties")
@ConfigurationProperties(prefix = "local")
public class ExConfig {
    private String name;
    private DB db;
    ...getter/setter
    public static class DB {
        private String name;
        private String password;
        ...getter/setter
    }
}
```

3.3 可选操作：

在 pom 文件中添加 configuration-processor 的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

这样可以避免 application.properties 文件中出现不可解析的属性的提示：



还能够拥有代码提示：


#mini.app_id=wxhkshjbx
#mini.app-id=wxhkshjbx
mini.appId=wxhkshjbx
mini.secret=yu&hjh56
mini.
mini.app-id String
mini.channel-id String
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>