Spring是如何自动注入多类型

时常有个小问题围绕着我，Spring是如何给字段字符装盘，为何支持Collection、List、Map、String等这么多类型的呢？在Spring注入的过程中，有没有什么小技巧值得我们学习呢？带着这个疑惑，我们来一探究竟。

> 本文基于*SpringBoot V2.5.6，Spring V5.3.12*。不同版本可能会有不同，请注意哈

想要弄懂上面的问题，有一个小小的要求，那就是要弄懂SpringBean的生命周期（如和Get一个Bean），当然，我们也可以带着这个疑惑，一起去代码中寻找。

代码搜索

1.1 入口分析

要开始探索代码，那我们当然需要寻找一个入口，那我们从哪开始呢？当然就从启动函数开始啦。启动代码如下：

```
1 public static void main(String[] args) {
2   ConfigurableApplicationContext context =
SpringApplication.run(SpringTestApplication.class, args);
3   // 从容器中获取一个bean
4   context.getBean("fattyca1Bean");
5 }
```

我们在执行`SpringApplication.run`后可以得到一个`ApplicationContext`，那么我们就可以GetBean了，可以接着往下看GetBean。

1.2 深入其中

1.2.1

我们从getBean点进去，进入的是`org.springframework.context.support.AbstractApplicationContext#getBean(java.lang.String)`，点进去代码如下：

```
    @Override
    public Object getBean(String name) throws BeansException {
        // 获取当前的BeanFactory，然后在getBean
        return getBeanFactory().getBean(name);
    }
```

这里就是通过获取当前的BeanFactory，然后在getBean。这里我们对`getBeanFactory()`有一点点兴趣，为什么有兴趣呢？那就是我们想搞明白当前的BeanFactory是什么。话不多说，我们直接点进去。

1.2.2

点进去的时候发现`org.springframework.context.support.AbstractApplicationContext#getBeanFactory()`有两个实现类，分别是：

- `org.springframework.context.support.AbstractRefreshableApplicationContext`
- `org.springframework.context.support.GenericApplicationContext`

我们进入类中查看是如何实现BeanFactory的、

1. `org.springframework.context.support.AbstractRefreshableApplicationContext`

查看代码，有一个方法。如下：

```
protected DefaultListableBeanFactory createBeanFactory() {
  return new
DefaultListableBeanFactory(getInternalParentBeanFactory());
}
```

1. `org.springframework.context.support.GenericApplicationContext`

查看代码，构造函数：

```
public GenericApplicationContext() {
  this.beanFactory = new DefaultListableBeanFactory();
}
```

我们可以看到一个共同特点，最后实现的BeanFactory都是是`org.springframework.beans.factory.support.DefaultListableBeanFactory`，好了，到现在我们知道了，getBean最后都是通过`org.springframework.beans.factory.support.DefaultListableBeanFactory`来实现的。

不过呢，又一个疑问来了。 纳尼？ApplicationContext的GetBean竟然是通过组合`org.springframework.beans.factory.support.DefaultListableBeanFactory`来实现的，那ApplicationContext和`org.springframework.beans.factory.support.DefaultListableBeanFactory`有啥关系呢？又有啥区别呢？这个问题留在这，哈哈。

1.2.3

按着Spring的老规矩，xxx()是方法，doXxx()是真正实现的方法。我们一路点进去，从getBean -> doGetBean -> createBean -> doCreateBean();

在doCreateBean有如下代码：

```
protected Object doCreateBean(String beanName,
RootBeanDefinition mbd, @Nullable Object[] args){
        ...略
        // Initialize the bean instance.
        Object exposedObject = bean;
        try {
      // 填充Bean
            populateBean(beanName, mbd, instanceWrapper);
        }
        ...略
}
```

有一个`populateBean`方法，熟悉spring生命周期的同学知道，这里是在Bean初始化完成后，对Bean属性值就行填充的地方，当然，我们从方法注释也可以看出来哈。

> *Populate the bean instance in the given BeanWrapper with the property values from the bean definition.*

1.2.4

进去到populateBean方法内部。代码如下：

```
protected void populateBean(String beanName,
RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
      ...略
        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;
        }

        ...略
}
```

如我们所愿，在这里我们看到了两个全部大写的`AUTOWIRE_BY_NAME`&`AUTOWIRE_BY_TYPE`，这两个不就是自动注入的类型吗？看来是要到关键点了。那就点进去看看呗

1.2.5

1.2.5.1 autowireByName

```
// Fill in any missing property values with references to other beans in this factory if autowire is set to "byName".
protected void autowireByName(
            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {
    // 获取属性的名称

```

```
        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
        for (String propertyName : propertyNames) {
      // 是否是bean
            if (containsBean(propertyName)) {
                Object bean = getBean(propertyName);
        // 将propertyName和Bean对应起来
                pvs.add(propertyName, bean);
        // 将属性和Bean关联起来
                registerDependentBean(propertyName, beanName);
            }
        }
}
```

在方法中我们看到了是名称注入是通过geatBean获取关联bean来注入的。点进去方法发现，是个套娃。getBean代码如下：

```
@Override
public Object getBean(String name) throws BeansException {
  return doGetBean(name, null, null, false);
}
```

这兜兜转转不又回到原点了吗？那我们就去另外一个方法看起来咯

1.2.5.2 autowireByType

进入放大，代码如下：

```
protected void autowireByType(
            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {
        ...略
```

```
4           Set<String> autowiredBeanNames = new LinkedHashSet<>
(4);
5           String[] propertyNames =
unsatisfiedNonSimpleProperties(mbd, bw);
6           for (String propertyName : propertyNames) {
7               try {
8           // 反射属性描述
9                   PropertyDescriptor pd =
bw.getPropertyDescriptor(propertyName);
10                  // Don't try autowiring by type for type
Object: never makes sense,
11                  // even if it technically is a unsatisfied,
non-simple property.
12                  if (Object.class != pd.getPropertyType()) {
13              // 获取属性的setter方法
14                      MethodParameter methodParam =
BeanUtils.getWriteMethodParameter(pd);
15                      // Do not allow eager init for type
matching in case of a prioritized post-processor.
16              // 是否饥饿？判断是否懒加载
17                      boolean eager = !(bw.getWrappedInstance()
instanceof PriorityOrdered);
18              // 属性描述
19                      DependencyDescriptor desc = new
AutowireByTypeDependencyDescriptor(methodParam, eager);
20              // 解析依赖(关键)
21                      Object autowiredArgument =
resolveDependency(desc, beanName, autowiredBeanNames,
converter);
22                      if (autowiredArgument != null) {
23                          pvs.add(propertyName,
autowiredArgument);
24                      }
25                      for (String autowiredBeanName :
autowiredBeanNames) {
26              // 关联属性和对应Bean
27
 registerDependentBean(autowiredBeanName, beanName);
```

```
                    }
                    autowiredBeanNames.clear();
                }
            }
        ...略
        }
    }
```

这里的代码就比较清晰了，Spring做了好几步操作，分别是：

1.2.5.1.1 反射获取属性

我们通过名称也可以看出来获取了`PropertyDescriptor`，这个类主要是获取属性的Get和Setter方法（writeMethod和readMethod），然后通过方法参数构建了一个`DependencyDescriptor`，记录一些参数信息，具体的可以看一下看。

1.2.5.1.2 解析依赖（关键）

我们进到具体的方法里面。代码如下：

```
1    public Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,
2            @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansException {
3
4
 descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());
5    // 是不是Optional类
6        if (Optional.class == descriptor.getDependencyType()) {
7            return createOptionalDependency(descriptor, requestingBeanName);
8        }
9    // 是不是ObjectFacotry，ObjectProvider
```

```
        else if (ObjectFactory.class == descriptor.getDependencyType() ||
                ObjectProvider.class == descriptor.getDependencyType()) {
            return new DependencyObjectProvider(descriptor, requestingBeanName);
        }
        else if (javaxInjectProviderClass == descriptor.getDependencyType()) {
            return new Jsr330Factory().createDependencyProvider(descriptor, requestingBeanName);
        }
        else {
            Object result = getAutowireCandidateResolver().getLazyResolutionProxyIfNecessary(
                    descriptor, requestingBeanName);
            if (result == null) {
        // 真正的解析
                result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);
            }
            return result;
        }
}
```

在这个方法里判断了好几种类型。Optional、ObjectFactory、ObjectProvider、Java.Inject.Provider、普通类等。不同的类有不同的处理方式。当然，按照老规矩，我们还是进入到doResolveDependency是真正具体的解析操作，我们进去瞧一瞧。

```
public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable String beanName,
```

```
            @Nullable Set<String> autowiredBeanNames,
@Nullable TypeConverter typeConverter) throws BeansException
{

        try {
            ...略

            Class<?> type = descriptor.getDependencyType();
      // 自动注入的解析器获取值，默认实现，返回值为null
            Object value =
getAutowireCandidateResolver().getSuggestedValue(descriptor);
            if (value != null) {
        // 字符类型判断
                if (value instanceof String) {
                    String strVal =
resolveEmbeddedValue((String) value);
                    BeanDefinition bd = (beanName != null &&
containsBean(beanName) ?
                            getMergedBeanDefinition(beanName)
: null);
          // 解析值
                    value =
evaluateBeanDefinitionString(strVal, bd);
                }
        // 获取转换器
                TypeConverter converter = (typeConverter !=
null ? typeConverter : getTypeConverter());
                try {
          // 将类型转换对应的类型的值
                    return
converter.convertIfNecessary(value, type,
descriptor.getTypeDescriptor());
                }
                catch (UnsupportedOperationException ex) {
                    // A custom TypeConverter which does not
support TypeDescriptor resolution...
                    return (descriptor.getField() != null ?
```

```
28
     converter.convertIfNecessary(value, type,
    descriptor.getField()) :
29
     converter.convertIfNecessary(value, type,
    descriptor.getMethodParameter()));
30                      }
31                  }
32                  // 多依赖Bean
33                  Object multipleBeans =
    resolveMultipleBeans(descriptor, beanName,
    autowiredBeanNames, typeConverter);
34                  if (multipleBeans != null) {
35                      return multipleBeans;
36                  }
37                  ...略
38  }
```

看代码，首先是通过`getAutowireCandidateResolver().getSuggestedValue(descriptor);`获取了一波值，但是跟进一下代码，`getAutowireCandidateResolver()`的默认实现是：`org.springframework.beans.factory.support.SimpleAutowireCandidateResolver`，其`getSuggestedValue`的返回值为null。

```
1  public Object getSuggestedValue(DependencyDescriptor
   descriptor) {
2    return null;
3  }
```

，接着我们往下看，到了`resolveMultipleBeans`，这个一看名字可能就是解析有多个Bean的方法，有点那味了，多个Bean解析就有可能使我们要找的，我们接着看。

```
```

```
private Object resolveMultipleBeans(DependencyDescriptor descriptor, @Nullable String beanName,
            @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) {

        Class<?> type = descriptor.getDependencyType();

        if (descriptor instanceof StreamDependencyDescriptor) {
            ...略
        }
        else if (type.isArray()) {
            ...略
        }
        else if (Collection.class.isAssignableFrom(type) && type.isInterface()) {
            ...略
        }
        else if (Map.class == type) {
            ...略
        }
        else {
            return null;
        }
    }
```

一点进来，好家伙，这代码，if..else if ..else，在看看这判断，不就是我们心心念念的Collection、Map..类型注入的吗？那我们找一个具体的方法看看呗，比如说Map。代码如下：

```
else if (Map.class == type) {
        // 获取泛型类型
            ResolvableType mapType = descriptor.getResolvableType().asMap();
            Class<?> keyType = mapType.resolveGeneric(0);
        // 判断key的类型是不是String
```

```
            if (String.class != keyType) {
                return null;
            }
            Class<?> valueType = mapType.resolveGeneric(1);
            if (valueType == null) {
                return null;
            }
        // 找到符合条件的Bean，并返回Map<BeanName,Bean>类型
            Map<String, Object> matchingBeans =
findAutowireCandidates(beanName, valueType,
                    new MultiElementDescriptor(descriptor));
            if (matchingBeans.isEmpty()) {
                return null;
            }
            if (autowiredBeanNames != null) {

 autowiredBeanNames.addAll(matchingBeans.keySet());
            }
        // 返回结果
            return matchingBeans;
}
```

通过反射获取需要注入类型的泛型(ResolvableType是Spring中提供反射的，注释上有使用说明，可以自行看一下)。然后判断key的类型。这里有一个小问题，如果KeyType不是String类型的，将会直接返回Null。这个是我们使用注册Bean的时候需要的注意点。

然后是判断valueType，接着使用`findAutowireCandidates`方法找到Class的所有Bean类型，并且直接封装成了Map类型的结构，然后直接返回了。

至此我们知道了，在我们自动装配Spring帮我们做了太多的事情了，设计的Spring的初始化，在注入时自动帮忙组装成Map、List、Array等，Spring还是细心啊~