

有道云链接: <http://note.youdao.com/noteshare?id=478b1de14614674acde6e331b09d792b&sub=24A246DE39194295878C28F0F8A3C548> (复制链接到浏览器的时候注意转行的空格)

上节课我们讲了Spring中的自动注入(byName,byType)和@Autowired注解的工作原理以及源码分析,那么今天这节课,我们来分析还没讲完的,剩下的核心的方法:

```
@Nullable
Object resolveDependency(DependencyDescriptor descriptor, @Nullable String
    requestingBeanName,
    @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter)
    throws BeansException;
```

该方法表示,传入一个依赖描述(DependencyDescriptor),该方法会根据该依赖描述从BeanFactory中找出对应的唯一的一个Bean对象。

下面来分析一下DefaultListableBeanFactory中**resolveDependency()**方法的具体实现,具体流程图: <https://www.processon.com/view/link/5f8d3c895653bb06ef076688>

findAutowireCandidates()实现

根据类型找beanName的底层流程:

<https://www.processon.com/view/link/6135bb430e3e7412ecd5d1f2> 对应执行流程图为:

<https://www.processon.com/view/link/5f8fdfa8e401fd06fd984f20>

1. 找出BeanFactory中类型为type的所有的Bean的名字,注意是名字,而不是Bean对象,因为我们可以根据BeanDefinition就能判断和当前type是不是匹配,不用生成Bean对象
2. 把resolvableDependencies中key为type的对象找出来并添加到result中
3. 遍历根据type找出的beanName,判断当前beanName对应的Bean是不是能够被自动注入
4. 先判断beanName对应的BeanDefinition中的autowireCandidate属性,如果为false,表示不能用来进行自动注入,如果为true则继续进行判断
5. 判断当前type是不是泛型,如果是泛型是会把容器中所有的beanName找出来的,如果是这种情况,那么在这一步中就要获取到泛型的真正类型,然后进行匹配,如果当前beanName和当前泛型对应的真实类型匹配,那么则继续判断
6. 如果当前DependencyDescriptor上存在@Qualifier注解,那么则要判断当前beanName上是否定义了Qualifier,并且是否和当前DependencyDescriptor上的Qualifier相等,相等则匹配
7. 经过上述验证之后,当前beanName才能成为一个可注入的,添加到result中

关于依赖注入中泛型注入的实现

首先在Java反射中,有一个Type接口,表示类型,具体分类为:

1. raw types: 也就是普通Class
2. parameterized types: 对应ParameterizedType接口,泛型类型

- 3. array types: 对应GenericArrayType, 泛型数组
- 4. type variables: 对应TypeVariable接口, 表示类型变量, 也就是所定义的泛型, 比如T、K
- 5. primitive types: 基本类型, int、boolean

大家可以好好看看下面代码所打印的结果:

```
public class TypeTest<T> {

    private int i;
    private Integer it;
    private int[] iarray;
    private List list;
    private List<String> slist;
    private List<T> tlist;
    private T t;
    private T[] tarray;

    public static void main(String[] args) throws NoSuchFieldException {

        test(TypeTest.class.getDeclaredField("i"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("it"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("iarray"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("list"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("slist"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("tlist"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("t"));
        System.out.println("=====");
        test(TypeTest.class.getDeclaredField("tarray"));

    }

    public static void test(Field field) {

        if (field.getType().isPrimitive()) {
            System.out.println(field.getName() + "是基本数据类型");
        } else {
            System.out.println(field.getName() + "不是基本数据类型");
        }

        if (field.getGenericType() instanceof ParameterizedType) {
            System.out.println(field.getName() + "是泛型类型");
        } else {
            System.out.println(field.getName() + "不是泛型类型");
        }

        if (field.getType().isArray()) {
            System.out.println(field.getName() + "是普通数组");
        } else {
            System.out.println(field.getName() + "不是普通数组");
        }
    }
}
```

```

    }

    if (field.getGenericType() instanceof GenericArrayType) {
        System.out.println(field.getName() + "是泛型数组");
    } else {
        System.out.println(field.getName() + "不是泛型数组");
    }

    if (field.getGenericType() instanceof TypeVariable) {
        System.out.println(field.getName() + "是泛型变量");
    } else {
        System.out.println(field.getName() + "不是泛型变量");
    }

}

}

```

Spring中，但注入点是一个泛型时，也是会进行处理的，比如：

```

@Component
public class UserService extends BaseService<OrderService, StockService> {

    public void test() {
        System.out.println(o);
    }

}

public class BaseService<O, S> {

    @Autowired
    protected O o;

    @Autowired
    protected S s;

}

```

1. Spring扫描时发现UserService是一个Bean
2. 那就取出注入点，也就是BaseService中的两个属性o、s
3. 接下来需要按注入点类型进行注入，但是o和s都是泛型，所以Spring需要确定o和s的具体类型。
4. 因为当前正在创建的是UserService的Bean，所以可以通过
`userService.getClass().getGenericSuperclass().getTypeName()` 获取到具体的泛型信息，比如
`com.zhouyu.service.BaseService<com.zhouyu.service.OrderService, com.zhouyu.service.StockService>`
5. 然后再拿到UserService的父类BaseService的泛型变量：
`for (TypeVariable<? extends Class<?>> typeParameter : userService.getClass().getSuperclass().getTypeParameters()) {
 System.out.println(typeParameter.getName()); }`

6. 通过上面两段代码，就能知道，o对应的具体就是OrderService，s对应的具体类型就是StockService
7. 然后再调用 oField.getGenericType() 就知道当前field使用的是哪个泛型，就能知道具体类型了

@Qualifier的使用

定义两个注解：

```
@Target({ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier("random")
public @interface Random {
}
```

```
@Target({ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier("roundRobin")
public @interface RoundRobin {
}
```

定义一个接口和两个实现类，表示负载均衡：

```
public interface LoadBalance {
    String select();
}
```

```
@Component
@Random
public class RandomStrategy implements LoadBalance {

    @Override
    public String select() {
        return null;
    }
}
```

```
@Component
@RoundRobin
public class RoundRobinStrategy implements LoadBalance {

    @Override
    public String select() {
        return null;
    }
}
```

使用:

```
@Component
public class UserService {

    @Autowired
    @RoundRobin
    private LoadBalance loadBalance;

    public void test() {
        System.out.println(loadBalance);
    }
}
```

@Resource

@Resource注解底层工作流程图:

<https://www.processon.com/view/link/5f91275f07912906db381f6e>

