@Autowired和@Resource注解的区别，大家有没有想过两者的功能差不多，那为啥Spring还提供两种依赖注入方式呢?

我们知道@Autowired注解是Spring框架提供的，所以在使用该注解时依赖于该框架，而@Resource注解是JDK自带的，不需要依赖第三方。大多数IOC框架对@Resource都做了支持，而@Autowired只能使用Spring，所以当我们使用@Resource注解后，切换了其它框架，就可能不需要去修改该注解了，也能支持其注入功能。

1.@Autowired注入过程

```
public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) {
    // 1、该方法会尝试从缓存中取当前Bean字段中使用了@Autowire的字段信息或方法信息。
    InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs);
    try {
      // 进行属性注入
      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;
  }
```

- 步骤1从缓存中取不到值时，会通过反射的方式遍历当前bean的所有字段信息和方法信息，并判断其是否使用了@Autowired注解，并封装到对应的处理器中，后面注入步骤会进行遍历进行查询处理及注入。@Autowired可以注解在方法、字段、构造器上，构造器注入在创建bean的时候进行的。
- 步骤2会根据步骤1中查询出来的信息遍历，由于字段和方法会分别封装成InjectedElement类型的不同子类实现，所以不同的类型会调用不同的子类实现方法进行处理。

```
//doResolveDependency
  @Nullable
  public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable String
beanName,
      @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter)
throws BeansException {
      //.................
      //此处省略了部分代码
      //.................

    //尝试从bean工厂中查询出需要注入类型的所有Bean对象，@Qualifier注解的解析是在这个步骤中进行的
    Map<String, Object> matchingBeans = findAutowireCandidates(beanName, type,
descriptor);
    //beanFactory工厂中查询不到时,抛出异常
    if (matchingBeans.isEmpty()) {
```

```
      //如果descriptor需要注入
      if (isRequired(descriptor)) {
        //抛出NoSuchBeanDefinitionException或BeanNotOfRequiredTypeException以解决不可 解决的
依赖关系
        raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);
      }
      //返回null，表示么有找到候选Bean对象
      return null;
    }

    //定义用于存储唯一的候选Bean名变量
    String autowiredBeanName;
    //定义用于存储唯一的候选Bean对象变量
    Object instanceCandidate;

    //如果beanFactory工厂中存在多个类型的bean时
    if (matchingBeans.size() > 1) {
      //筛选出符合要求的bean
      autowiredBeanName = determineAutowireCandidate(matchingBeans, descriptor);
      //如果autowiredBeanName为null
      if (autowiredBeanName == null) {
        //如果查询不到合适的，而且该属性并非require=true的，可以赋值为null
        if (isRequired(descriptor) || !indicatesMultipleBeans(type)) {
          //让descriptor尝试选择其中一个实例，默认实现是抛出NoUniqueBeanDefinitionException.
          return descriptor.resolveNotUnique(descriptor.getResolvableType(),
matchingBeans);
        }
        else {
          // In case of an optional Collection/Map, silently ignore a non-unique case:
          // possibly it was meant to be an empty collection of multiple regular beans
          // (before 4.3 in particular when we didn't even look for collection beans).
          // 如果是可选的Collection/Map,则静默忽略一个非唯一情况：
          // 可能是多个常规bean的空集合
          // （尤其是在4.3之前，设置在我们没有寻找collection bean的时候 ）
          return null;
        }
      }
      //获取autowiredBeanName对应的候选Bean对象
      instanceCandidate = matchingBeans.get(autowiredBeanName);
    }
    else {
      //如果查询出来只有一个时，则将该bean作为注入值
      Map.Entry<String, Object> entry = matchingBeans.entrySet().iterator().next();
      //让autowireBeanName引用该元素的候选bean名
      autowiredBeanName = entry.getKey();
      //让instanceCandidate引用该元素的候选bean对象
      instanceCandidate = entry.getValue();
    }

    //如果候选bean名不为null，
    if (autowiredBeanNames != null) {
      //将autowiredBeanName添加到autowiredBeanNames中，又添加一次
```

```
        autowiredBeanNames.add(autowiredBeanName);
      }
      //如果instanceCandidate是Class实例，对其进行实例化
      if (instanceCandidate instanceof Class) {
        //让instanceCandidate引用 descriptor对autowiredBeanName解析为该工厂的Bean实例
        instanceCandidate = descriptor.resolveCandidate(autowiredBeanName, type, this);
      }
      //定义一个result变量，用于存储最佳候选Bean对象
      Object result = instanceCandidate;
      //如果reuslt是NullBean的实例
      if (result instanceof NullBean) {
        //如果descriptor需要注入
        if (isRequired(descriptor)) {
          //抛出NoSuchBeanDefinitionException或BeanNotOfRequiredTypeException以解决不可 解决的
依赖关系
          raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);
        }
        //返回null，表示找不到最佳候选Bean对象
        result = null;
      }
      //如果result不是type的实例
      if (!ClassUtils.isAssignableValue(type, result)) {
        //抛出Bean不是必需类型异常
        throw new BeanNotOfRequiredTypeException(autowiredBeanName, type,
instanceCandidate.getClass());
      }
      //返回最佳候选Bean对象【result】
      return result;
    }
    finally {
      //设置上一个切入点对象
      ConstructorResolver.setCurrentInjectionPoint(previousInjectionPoint);
    }
  }
```

具体步骤总结：DBService --> MySQLDBService OracleDBService

```
1、尝试从缓存中获取被注入类型的所有Bean；
2、并对获取到Bean容器进行遍历，判断被注入类型是否有@Qualifier注解，有则进行名称匹配，匹配到时返回一个符合条件的Bean容器，否则返回所有注入类型的Bean用于后续进一步筛选。
3、如果获取不到时，判断该属性是否必须注入，如果非必须注入可以注入null；
4、如果匹配到的类型Bean只有一个时，则认为是符合要求的，返回该值后进行注入。
5、如果匹配到多个Bean时，根据匹配条件来筛选；
```

@Qualifier匹配过程见： isAutowireCandidate(BeanDefinitionHolder bdHolder, DependencyDescriptor descriptor)方法详解

```
//isAutowireCandidate
```

```
  public boolean isAutowireCandidate(BeanDefinitionHolder bdHolder, DependencyDescriptor
descriptor) {
    //判断要注入的字段是否有@Quilifier注解
    boolean match = super.isAutowireCandidate(bdHolder, descriptor);
    if (match) {
      //校验是否匹配
      match = checkQualifiers(bdHolder, descriptor.getAnnotations());
      if (match) {
        //如果是方法注入时，获取方法参数进行校验匹配
        MethodParameter methodParam = descriptor.getMethodParameter();
        if (methodParam != null) {
          Method method = methodParam.getMethod();
          if (method == null || void.class == method.getReturnType()) {
            match = checkQualifiers(bdHolder, methodParam.getMethodAnnotations());
          }
        }
      }
    }
    return match;
  }
```

多个bean的匹配过程见： determineAutowireCandidate(matchingBeans, descriptor)方法详解

```
//determineAutowireCandidate
  protected String determineAutowireCandidate(Map<String, Object> candidates,
DependencyDescriptor descriptor) {
    //通过反射的方式去查询已匹配的Bean容器中是否有@Primary注解，如果有多个则抛出异常，如果只有一个则代表
该Bean对象为符合注入条件的
    Class<?> requiredType = descriptor.getDependencyType();
    String primaryCandidate = determinePrimaryCandidate(candidates, requiredType);
    if (primaryCandidate != null) {
      return primaryCandidate;
    }
    //判断已匹配的Bean容器中是否有@Priority注解，并比较其优先级大小，挑选出符合的一个
    String priorityCandidate = determineHighestPriorityCandidate(candidates,
requiredType);
    if (priorityCandidate != null) {
      return priorityCandidate;
    }
    // Fallback
    // 这里将匹配的bean容器遍历后，挨个判断bean名称与要注入类型的名称是否相同，不同则去别名缓存中查询，
看是否有别名与要注入类型的名称一样，有则匹配
    for (Map.Entry<String, Object> entry : candidates.entrySet()) {
      String candidateName = entry.getKey();
      Object beanInstance = entry.getValue();
      if ((beanInstance != null &&
this.resolvableDependencies.containsValue(beanInstance)) ||
          matchesBeanName(candidateName, descriptor.getDependencyName())) {
        return candidateName;
```

```
      }
    }
    return null;
  }
```

具体过程总结：

1、通过反射的方式去查询已匹配的Bean容器中是否有@Primary注解，如果有多个则抛出异常，如果只有一个则代表该Bean对象为符合注入条件的

2、判断已匹配的Bean容器中是否有@Priority注解，并比较其优先级大小，挑选出符合的一个

3、这里将匹配的bean容器遍历后，挨个判断bean名称与要注入类型的名称是否相同，不同则去别名缓存中查询，看是否有别名与要注入类型的名称一样，有则匹配；

2.@Resource注入过程

```
public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) {
    //查询注入对象或者方法中是否有@Resource注解
    InjectionMetadata metadata = findResourceMetadata(beanName, bean.getClass(), pvs);
    try {
      //对其进行解析注入
      metadata.inject(bean, beanName, pvs);
    }
    catch (Throwable ex) {
      throw new BeanCreationException(beanName, "Injection of resource dependencies
failed", ex);
    }
    return pvs;
  }
```

1、遍历当前Bean的属性及其方法，判断是否有@Resource注解，并封装成对应的包装类

2、对其进行查找注入

```
//autowireResource
protected Object autowireResource(BeanFactory factory, LookupElement element, @Nullable
String requestingBeanName)
      throws NoSuchBeanDefinitionException {
// 自动装配的对象
  Object resource;
  // 自动装配的名字
  Set<String> autowiredBeanNames;
  // 依赖的属性名
  String name = element.name;
  //默认BeanFacory是AutowireCapableBeanFactory工厂类型
  if (factory instanceof AutowireCapableBeanFactory) {
```

```
    AutowireCapableBeanFactory beanFactory = (AutowireCapableBeanFactory) factory;
    // 创建依赖描述
    DependencyDescriptor descriptor = element.getDependencyDescriptor();
    //这里的factory.containsBean(name)会根据需要注入的属性名称去BeanFactory工厂中进行查询，如果查询到了直接获取后进行注入，如果查询不到时，调用beanFactory.resolveDependency方法进行查找，该方法流程与@Autowire一样。
    if (this.fallbackToDefaultTypeMatch && element.isDefaultName &&
!factory.containsBean(name)) {
      //如果容器中还没有此bean，则会使用resolveDependency()方法将符合bean type的bean definetion调用一次getBean()
      // 从这些bean选出符合requestingBeanName的bean
      autowiredBeanNames = new LinkedHashSet<>();
      resource = beanFactory.resolveDependency(descriptor, requestingBeanName,
autowiredBeanNames, null);
      if (resource == null) {
        throw new NoSuchBeanDefinitionException(element.getLookupType(), "No resolvable resource object");
      }
    }
    else {
      //如果容器中有此bean则取出这个bean对象作为属性值
      resource = beanFactory.resolveBeanByName(name, descriptor);
      autowiredBeanNames = Collections.singleton(name);
    }
  }
  else {
    resource = factory.getBean(name, element.lookupType);
    autowiredBeanNames = Collections.singleton(name);
  }

  //.......
  //.......

  return resource;
}
```

3.@Resouce解析与@Autowired解析异同：

- @Resouce会先尝试使用beanName名称去beanFactory工厂中查询是否有该定义信息，如果有则直接取出。
- 如果上一步骤不满足时，走后面按类型匹配的步骤，该步骤与@Autowired调用的方法一致，所以流程是一样的；

4.总结

@Autowired解析步骤：

尝试从缓存中获取被注入类型的所有Bean；

并对获取到Bean容器进行遍历，判断被注入类型是否有@Qualifier注解，有则进行名称匹配，匹配到时返回一个符合条件的Bean容器，否则返回所有注入类型的Bean用于后续进一步筛选。

如果获取不到时，判断该属性是否必须注入，如果非必须注入可以注入null;

如果匹配到的类型Bean只有一个时，则认为是符合要求的，返回该值后进行注入。

如果匹配到多个Bean时，根据匹配条件来筛选：

通过反射的方式去查询已匹配的Bean容器中是否有@Primary注解，如果有多个则抛出异常，如果只有一个则代表该Bean对象为符合注入条件的

判断已匹配的Bean容器中是否有@Priority注解，并比较其优先级大小，挑选出符合的一个，数值越低优选级越高

这里将匹配的bean容器遍历后，挨个判断bean名称与要注入类型的名称是否相同，不同则去别名缓存中查询，看是否有别名与要注入类型的名称一样，有则匹配；

@Resouce解析步骤：

根据beanName去工厂中查询是否有该定义信息，有则获取Bean对象，没有则进行类型匹配操作；

尝试从缓存中获取被注入类型的所有Bean；

并对获取到Bean容器进行遍历，判断被注入类型是否有@Qualifier注解，有则进行名称匹配，匹配到时返回一个符合条件的Bean容器，否则返回所有注入类型的Bean用于后续进一步筛选。

如果获取不到时，判断该属性是否必须注入，如果非必须注入可以注入null;

如果匹配到的类型Bean只有一个时，则认为是符合要求的，返回该值后进行注入。

如果匹配到多个Bean时，根据匹配条件来筛选：

通过反射的方式去查询已匹配的Bean容器中是否有@Primary注解，如果有多个则抛出异常，如果只有一个则代表该Bean对象为符合注入条件的

判断已匹配的Bean容器中是否有@Priority注解，并比较其优先级大小，挑选出符合的一个，数值越低优选级越高

这里将匹配的bean容器遍历后，挨个判断bean名称与要注入类型的名称是否相同，不同则去别名缓存中查询，看是否有别名与要注入类型的名称一样，有则匹配；