1 问题来源

- 对于 @ConditionOnBean 注解的匹配成功的条件不是很清晰
- 随着项目的复杂化，各个 bean 直接的关联也越来越复杂，这样极有可能在使用 @ConditionOnBean 注解时，并不能达到预期的目的
- 预要清晰的知道使用的 @ConditionOnBean 注解的预期效果，必须了解清楚它的**匹配的时机和匹配的成功的条件**

2 @ConditionOnBean 注解匹配的时机

- 在 Springboot 源码分析 —— refreshContext() 解析 一文中，详细介绍了配置类的解析过程
- 在其中可以看出在 解析 ConfigurationClass 和 加载 BeanDefinition 这两个阶段都使用了**条件注解进行配置类过滤**
- 过滤的关键是 **ConditionEvluator#shouldSkip** 方法，源码分析如下
    - 在 解析 ConfigurationClass 阶段，phase 参数是 PARSE_CONFIGURATION
    - 在 加载 BeanDefinition 阶段，phase 参数是 REGISTER_BEAN
- 通过源码分析可知：@ConditionOnBean 注解是在 **加载 BeanDefinition 阶段**，才去匹配的

```
public boolean shouldSkip(@Nullable AnnotatedTypeMetadata metadata, @Nullable
ConfigurationPhase phase) {
   //判断类是否被 @Conditional 注解，如果不是，返回 false，表示不需要跳过该类
   if (metadata == null || !metadata.isAnnotated(Conditional.class.getName())) {return
false;}
   //如果 phase 为 null，判断类是否是配置类，如果是旧是在 PARSE_CONFIGURATION 阶段
   if (phase == null) {
      if (metadata instanceof AnnotationMetadata &&
            ConfigurationClassUtils.isConfigurationCandidate((AnnotationMetadata)
metadata)) {
         return shouldSkip(metadata, ConfigurationPhase.PARSE_CONFIGURATION);
      }
      return shouldSkip(metadata, ConfigurationPhase.REGISTER_BEAN);
   }
   //获取所有的 conditions
   List<Condition> conditions;//具体代码略
```

```
        AnnotationAwareOrderComparator.sort(conditions);//排序
        for (Condition condition : conditions) {
            ConfigurationPhase requiredPhase = null;
            //这里很关键：@ConditionOnBean 注解对应的 条件类就是 实现了 ConfigurationCondition 接口
            if (condition instanceof ConfigurationCondition) {
                //并且 @ConditionOnBean 注解 得到的 requiredPhase 为 REGISTER_BEAN
                requiredPhase = ((ConfigurationCondition) condition).getConfigurationPhase();
            }
            //第一阶段 phase 为 PARSE_CONFIGURATION，这时 @ConditionOnBean 注解 就会被忽略
            //第二阶段 phase 为 REGISTER_BEAN，这时 @ConditionOnBean 注解 就会生效
            if ((requiredPhase == null || requiredPhase == phase) &&
!condition.matches(this.context, metadata)) {
                return true;
            }
        }
        return false;
    }
```

3 @ConditionOnBean 注解匹配的条件

- 跟踪源码得到匹配的关键代码，由以下代码可以看出注解匹配的条件是：**目前** beanDefinitionNames 中是否存在该类，如果存在则匹配成功
- 注：当 @ConditionOnBean(用到是 name) 时，只会去 beanDefinitionNames 中找对应的 name 字符串，不会去匹配类型。而@ConditionOnBean(**.class) 时，则会去匹配类型

```
public Set<String> getNamesForType(Class<?> type, TypeExtractor typeExtractor) {
    //将beanDefinitionNames 和 manualSingletonNames 添加到 beanTypes 中
    updateTypesIfNecessary();
    return this.beanTypes.entrySet().stream().filter((entry) -> {
        Class<?> beanType = extractType(entry.getValue(), typeExtractor);
        //如果 name 对应的 type 在 beanType 中则匹配到
        return beanType != null && type.isAssignableFrom(beanType);
    }).map(Map.Entry::getKey).collect(Collectors.toCollection(LinkedHashSet::new));
}
```

3.1 注意事项

- 因为 @ConditionOnBean 注解是实时的去 beanDefinitionNames 匹配，所以该注解对应的类的 BeanDefinition 加载的顺序先后，都可能会对匹配的结果造成影响！
- 所有要清楚的了解该注解匹配的最终结果，就需要知道项目所有的 BeanDefinition 的加载顺序

4 BeanDefinition 生成的2个阶段

- 在 Springboot 源码分析 —— refreshContext() 解析 一文中通过跟踪源码，详细分析了配置类的 beanDefinition 加载过程
- 本小节主要是对上文的抽象，总结了项目中所有配置类、自动配置类的总体的 beanDefinition 的生成顺序

4.1 刷选出符合条件的 ConfigurationClass

- 这一阶段通过扫描配置类得到所有符合条件的 ConfigurationClass —— 通过 parser.parse(..) 解析得到

- **注意：** 过滤条件为 @ConditionOnClass、@ConditionalOnProperty 等静态条件(编译阶段就可以确定的)，不包含 @ConditionOnBean 等动态条件(运行后才生成的)
- **ConfigurationClass 中 class 的顺序：**
  - 第一是 @ComponentScan 中扫描包的顺序
  - 如果配置类中有 @Import 注解，那么会先加载 @Import 注解中的配置类
  - **注意：** 这里并没有 @Bean 对应的 ConfigurationClass，@Bean 对应的类是在 第二阶段解析它所在的配置类时，才直接解析为 BeanDefinition 的

4.2 通过 ConfigurationClass 解析得到所有的 BeanDefinition

- 通过 loadBeanDefinitions(ConfigurationClass) 解析得到所有的 BeanDefinition
- **注意：** 在第一阶段，所有通过 Scan 得到的配置类已经注册到了 BeanDefinitionMaps 中了！所以它们的位置是最靠前的
- 这一阶段会处理 @ConditionOnBean 等动态条件，来跳过一些配置类、移除已在 BeanDefinitionMaps 中的但不满足条件的类
- **BeanDefinitionMaps 中 BeanDefinition 的顺序：**
  - 首先是第一阶段就生成的 BeanDefinition（可能被动态条件所移除）
  - 然后基本上就是第一阶段中的 ConfigurationClass 对应的顺序：具体的顺序如下源码所示

```
private void loadBeanDefinitionsForConfigurationClass(..) {
   //处理动态条件
   if (trackedConditionEvaluator.shouldSkip(configClass)) {..}
   //如果自己是被 import 进来的，先注册自己
   if (configClass.isImported()) {
      registerBeanDefinitionForImportedConfigurationClass(configClass);
   }
   //注册 @Bean 方法
   for (BeanMethod beanMethod : configClass.getBeanMethods()) {
      loadBeanDefinitionsForBeanMethod(beanMethod);
   }
   //加载ImportedResources，如xml，解析其中的Bean，放入beanDefinition中
   loadBeanDefinitionsFromImportedResources(configClass.getImportedResources());
   //注册配置类中 @import 注解中的 Registrars 类
   loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars());
}
```

5 配置类、自动配置类的 BeanDefinition 加载顺序

- 由于自动配置类是不被扫描到的（是通过一个 实现了 deferredImportSelector 接口的类加载得到的）
- 总体来说，**配置类的 BeanDefinition 是在 自动配置类之前生成的**，由以下源码可以看出

```
    public void parse(Set<BeanDefinitionHolder> configCandidates) {
        //首先解析所有的配置类
        for (BeanDefinitionHolder holder : configCandidates) {
            BeanDefinition bd = holder.getBeanDefinition();
            parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());
        }
        //这里才解析自动配置类的。因为自动配置类是通过 实现了 DeferredImportSelector 接口的
ImportSelector 类去查找的(AutoConfigurationImportSelector)
        this.deferredImportSelectorHandler.process();
    }
```

6 自动配置类的 BeanDefinition 加载顺序

- 在 `deferredImportSelectorHandler.process()` 解析生成了 自动配置类的 ConfigurationClass，这里的 ConfigurationClass 是排在最后的了，所以对应的 BeanDefinition 加载 也是在最后
- 通过源码跟踪得到以下结论
  - 首先得到所有的自动配置类
  - 然后，先按 字典顺序排序，再按 order 顺序排序，再按 before/after 注解排序
  - 所以说 before/after 注解优先级最高，即使 order 值很小，也可能最后加载 beanDefinition!
- 注：@AutoConfigureAfter、@AutoConfigureBefore、@AutoConfigureOrder 只对自动配置类有效

```
this.deferredImportSelectorHandler.process();
public void process() {
    //得到所有的自动配置的 selector，注：一个 @EnableAutoConfiguration 就会生成一个 自动配置
selector，多个 @EnableAutoConfiguration 会重复获取，浪费资源
    List<DeferredImportSelectorHolder> deferredImports = this.deferredImportSelectors;
    handler.processGroupImports();//关键方法
}
public void processGroupImports() {
    //selector 有一个 group 概念，暂时没有了解
    for (DeferredImportSelectorGrouping grouping : this.groupings.values()) {
        //关键方法： getImports()，获取该组下所有的自动配置类
        grouping.getImports().forEach(entry -> {
            //将每一个自动配置类当作普通配置类来处理
        });
    }
}
public Iterable<Group.Entry> getImports() {
    for (DeferredImportSelectorHolder deferredImport : this.deferredImports) {
        //处理得到
        this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                           deferredImport.getImportSelector());
    }
    return this.group.selectImports();
}
//得到候选的配置类
public void process(..) {
    //关键方法：getAutoConfigurationEntry()
    AutoConfigurationEntry autoConfigurationEntry = ((AutoConfigurationImportSelector)
deferredImportSelector)
        .getAutoConfigurationEntry(getAutoConfigurationMetadata(), annotationMetadata);
```

```
        this.autoConfigurationEntries.add(autoConfigurationEntry);
        for (String importClassName : autoConfigurationEntry.getConfigurations()) {
            this.entries.putIfAbsent(importClassName, annotationMetadata);
        }
    }
    protected AutoConfigurationEntry getAutoConfigurationEntry(...) {
        //获取所有的自动配置类
        List<String> configurations = getCandidateConfigurations(..);
        configurations = removeDuplicates(configurations);//去重
        //通过 @EnableAutoConfiguration 注解中的属性得到需要排除的配置类
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = filter(configurations, autoConfigurationMetadata);//搞不清楚的过滤一
    大波
        fireAutoConfigurationImportEvents(configurations, exclusions);//搞不清楚的事件
        return new AutoConfigurationEntry(configurations, exclusions);
    }
```

6.1 核心顺序解析方法

```
    public Iterable<Entry> selectImports() {
       Set<String> processedConfigurations = 得到待排序的配置类;
       return sortAutoConfigurations(..)..collect(Collectors.toList());;//排序
    }
    new AutoConfigurationSorter(..).getInPriorityOrder(configurations);//排序

    public List<String> getInPriorityOrder(classNames) {
        List<String> orderedClassNames = new ArrayList<>(classNames);

        Collections.sort(orderedClassNames);//先按字典顺序排序！！为什么有这一步呢，，
        //按 @AutoConfigureOrder() 的值 升序 排序，一般默认是 0
        orderedClassNames.sort((o1, o2) -> {
            int i1 = classes.get(o1).getOrder();
            int i2 = classes.get(o2).getOrder();
            return Integer.compare(i1, i2);
        });
        //最后按 @AutoConfigureAfter、@AutoConfigureBefore 来排序
        //所以说 这两个注解优先级最高，即使 order 值很小，也可能最后加载 beanDefinition!
        orderedClassNames = sortByAnnotation(classes, orderedClassNames);
        return orderedClassNames;
    }
```

参考

spring-boot-2.1.6.RELEASE