springboot源码(二)：自动装配原理

在之前的课程中我们讲解了springboot的启动过程，其实在面试过程中问的最多的可能是自动装配的原理，而自动装配是在启动过程中完成，只不过在刚开始的时候我们选择性的跳过了，下面详细讲解自动装配的过程。

1、在springboot的启动过程中，有一个步骤是创建上下文，如果不记得可以看下面的代码：

```
public ConfigurableApplicationContext run(String... args) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        ConfigurableApplicationContext context = null;
        Collection<SpringBootExceptionReporter> exceptionReporters = new
ArrayList<>();
        configureHeadlessProperty();
        SpringApplicationRunListeners listeners = getRunListeners(args);
        listeners.starting();
        try {
            ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
            ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
            configureIgnoreBeanInfo(environment);
            Banner printedBanner = printBanner(environment);
            context = createApplicationContext();
            exceptionReporters =
getSpringFactoriesInstances(SpringBootExceptionReporter.class,
                    new Class[] { ConfigurableApplicationContext.class },
context);
            //此处完成自动装配的过程
            prepareContext(context, environment, listeners,
applicationArguments, printedBanner);
            refreshContext(context);
            afterRefresh(context, applicationArguments);
            stopWatch.stop();
            if (this.logStartupInfo) {
                new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
            }
            listeners.started(context);
            callRunners(context, applicationArguments);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, exceptionReporters, listeners);
            throw new IllegalStateException(ex);
        }

        try {
            listeners.running(context);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, exceptionReporters, null);
            throw new IllegalStateException(ex);
        }
        return context;
    }
```

2、在prepareContext方法中查找load方法，一层一层向内点击，找到最终的load方法

```
//prepareContext方法
    private void prepareContext(ConfigurableApplicationContext context,
ConfigurableEnvironment environment,
            SpringApplicationRunListeners listeners, ApplicationArguments
applicationArguments, Banner printedBanner) {
        context.setEnvironment(environment);
        postProcessApplicationContext(context);
        applyInitializers(context);
        listeners.contextPrepared(context);
        if (this.logStartupInfo) {
            logStartupInfo(context.getParent() == null);
            logStartupProfileInfo(context);
        }
        // Add boot specific singleton beans
        ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
        beanFactory.registerSingleton("springApplicationArguments",
applicationArguments);
        if (printedBanner != null) {
            beanFactory.registerSingleton("springBootBanner", printedBanner);
        }
        if (beanFactory instanceof DefaultListableBeanFactory) {
            ((DefaultListableBeanFactory) beanFactory)

.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
        }
        if (this.lazyInitialization) {
            context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
        }
        // Load the sources
        Set<Object> sources = getAllSources();
        Assert.notEmpty(sources, "Sources must not be empty");
        //load方法完成该功能
        load(context, sources.toArray(new Object[0]));
        listeners.contextLoaded(context);
    }


    /**
     * Load beans into the application context.
     * @param context the context to load beans into
     * @param sources the sources to load
     * 加载bean对象到context中
     */
    protected void load(ApplicationContext context, Object[] sources) {
        if (logger.isDebugEnabled()) {
            logger.debug("Loading source " +
StringUtils.arrayToCommaDelimitedString(sources));
        }
        //获取bean对象定义的加载器
        BeanDefinitionLoader loader =
createBeanDefinitionLoader(getBeanDefinitionRegistry(context), sources);
        if (this.beanNameGenerator != null) {
            loader.setBeanNameGenerator(this.beanNameGenerator);
        }
        if (this.resourceLoader != null) {
            loader.setResourceLoader(this.resourceLoader);
        }
        if (this.environment != null) {
            loader.setEnvironment(this.environment);
        }
        loader.load();
    }

    /**
     * Load the sources into the reader.
     * @return the number of loaded beans
     */
    int load() {
        int count = 0;
        for (Object source : this.sources) {
            count += load(source);
        }
        return count;
    }
```

3、实际执行load的是BeanDefinitionLoader中的load方法，如下：

```
    //实际记载bean的方法
    private int load(Object source) {
        Assert.notNull(source, "Source must not be null");
        //如果是class类型，启用注解类型
        if (source instanceof Class<?>) {
            return load((Class<?>) source);
        }
        //如果是resource类型，启动xml解析
        if (source instanceof Resource) {
            return load((Resource) source);
        }
        //如果是package类型，启用扫描包，例如@ComponentScan
        if (source instanceof Package) {
            return load((Package) source);
        }
        //如果是字符串类型，直接加载
        if (source instanceof CharSequence) {
            return load((CharSequence) source);
        }
        throw new IllegalArgumentException("Invalid source type " +
source.getClass());
    }
```

4、下面方法将用来判断是否资源的类型，是使用groovy加载还是使用注解的方式

```
    private int load(Class<?> source) {
        //判断使用groovy脚本
        if (isGroovyPresent() &&
GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
```

```
            // Any GroovyLoaders added in beans{} DSL can contribute beans here
            GroovyBeanDefinitionSource loader =
BeanUtils.instantiateClass(source, GroovyBeanDefinitionSource.class);
            load(loader);
        }
        //使用注解加载
        if (isComponent(source)) {
            this.annotatedReader.register(source);
            return 1;
        }
        return 0;
    }
```

5、下面方法判断启动类中是否包含@Component注解，但是会神奇的发现我们的启动类中并没有该注解，继续更进发现MergedAnnotations类传入了一个参数SearchStrategy.TYPE_HIERARCHY，会查找继承关系中是否包含这个注解，@SpringBootApplication-->@SpringBootConfiguration-->@Configuration-->@Component,当找到@Component注解之后，会把该对象注册到AnnotatedBeanDefinitionReader对象中

```
private boolean isComponent(Class<?> type) {
   // This has to be a bit of a guess. The only way to be sure that this type is
   // eligible is to make a bean definition out of it and try to instantiate it.
   if (MergedAnnotations.from(type,
SearchStrategy.TYPE_HIERARCHY).isPresent(Component.class)) {
      return true;
   }
   // Nested anonymous classes are not eligible for registration, nor are groovy
   // closures
   return !type.getName().matches(".*\\$_.*closure.*") &&
!type.isAnonymousClass()
         && type.getConstructors() != null && type.getConstructors().length !=
0;
}


    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations.
     * 从给定的bean class中注册一个bean对象，从注解中找到相关的元数据
     */
    private <T> void doRegisterBean(Class<T> beanClass, @Nullable String name,
            @Nullable Class<? extends Annotation>[] qualifiers, @Nullable
Supplier<T> supplier,
            @Nullable BeanDefinitionCustomizer[] customizers) {

        AnnotatedGenericBeanDefinition abd = new
AnnotatedGenericBeanDefinition(beanClass);
        if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
            return;
        }

        abd.setInstanceSupplier(supplier);
        ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(abd);
        abd.setScope(scopeMetadata.getScopeName());
        String beanName = (name != null ? name :
this.beanNameGenerator.generateBeanName(abd, this.registry));
```

```
        AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
        if (qualifiers != null) {
            for (Class<? extends Annotation> qualifier : qualifiers) {
                if (Primary.class == qualifier) {
                    abd.setPrimary(true);
                }
                else if (Lazy.class == qualifier) {
                    abd.setLazyInit(true);
                }
                else {
                    abd.addQualifier(new AutowireCandidateQualifier(qualifier));
                }
            }
        }
        if (customizers != null) {
            for (BeanDefinitionCustomizer customizer : customizers) {
                customizer.customize(abd);
            }
        }

        BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd,
beanName);
        definitionHolder =
AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
        BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
this.registry);
    }

    /**
     * Register the given bean definition with the given bean factory.
     * 注册主类，如果有别名可以设置别名
     */
    public static void registerBeanDefinition(
            BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry
registry)
            throws BeanDefinitionStoreException {

        // Register bean definition under primary name.
        String beanName = definitionHolder.getBeanName();
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

        // Register aliases for bean name, if any.
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String alias : aliases) {
                registry.registerAlias(beanName, alias);
            }
        }
    }

//@SpringBootApplication
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
```

```
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {}

//@SpringBootConfiguration
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {}

//@Configuration
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {}
```

当看完上述代码之后，只是完成了启动对象的注入，自动装配还没有开始，下面开始进入到自动装配。

6、自动装配入口，从刷新容器开始

```
@Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                // 此处是自动装配的入口
                invokeBeanFactoryPostProcessors(beanFactory);
            }
```

7、在invokeBeanFactoryPostProcessors方法中完成bean的实例化和执行

```
/**
     * Instantiate and invoke all registered BeanFactoryPostProcessor beans,
     * respecting explicit order if given.
     * <p>Must be called before singleton instantiation.
     */
    protected void
invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
        //开始执行beanFactoryPostProcessor对应实现类,需要知道的是
beanFactoryPostProcessor是spring的扩展接口，在刷新容器之前，该接口可以用来修改bean元数据
信息

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory,
getBeanFactoryPostProcessors());

        // Detect a LoadTimeWeaver and prepare for weaving, if found in the
meantime
        // (e.g. through an @Bean method registered by
ConfigurationClassPostProcessor)
        if (beanFactory.getTempClassLoader() == null &&
beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
            beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));
            beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
        }
    }
```

8、查看invokeBeanFactoryPostProcessors的具体执行方法

```
    public static void invokeBeanFactoryPostProcessors(
            ConfigurableListableBeanFactory beanFactory,
List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {

        // Invoke BeanDefinitionRegistryPostProcessors first, if any.
        Set<String> processedBeans = new HashSet<>();

        if (beanFactory instanceof BeanDefinitionRegistry) {
            BeanDefinitionRegistry registry = (BeanDefinitionRegistry)
beanFactory;
            List<BeanFactoryPostProcessor> regularPostProcessors = new
ArrayList<>();
            List<BeanDefinitionRegistryPostProcessor> registryProcessors = new
ArrayList<>();
            //开始遍历三个内部类，如果属于BeanDefinitionRegistryPostProcessor子类，加
入到bean注册的集合，否则加入到regularPostProcessors
            for (BeanFactoryPostProcessor postProcessor :
beanFactoryPostProcessors) {
                if (postProcessor instanceof
BeanDefinitionRegistryPostProcessor) {
                    BeanDefinitionRegistryPostProcessor registryProcessor =
                            (BeanDefinitionRegistryPostProcessor) postProcessor;

registryProcessor.postProcessBeanDefinitionRegistry(registry);
                    registryProcessors.add(registryProcessor);
                }
                else {
                    regularPostProcessors.add(postProcessor);
                }
            }

            // Do not initialize FactoryBeans here: We need to leave all regular
beans
            // uninitialized to let the bean factory post-processors apply to
them!
```

```
            // Separate between BeanDefinitionRegistryPostProcessors that
implement
            // PriorityOrdered, Ordered, and the rest.
            List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors
= new ArrayList<>();

            // First, invoke the BeanDefinitionRegistryPostProcessors that
implement PriorityOrdered.
            //通过BeanDefinitionRegistryPostProcessor获取到对应的处理
类“org.springframework.context.annotation.internalConfigurationAnnotationProcess
or”，但是需要注意的是这个类在springboot中搜索不到，这个类的完全限定名在
AnnotationConfigEmbeddedWebApplicationContext中，在进行初始化的时候会装配几个类，在创建
AnnotatedBeanDefinitionReader对象的时候会将该类注册到bean对象中，此处可以看到
internalConfigurationAnnotationProcessor为bean名称，容器中真正的类是
ConfigurationClassPostProcessor
            String[] postProcessorNames =

beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
            //首先执行类型为PriorityOrdered的BeanDefinitionRegistryPostProcessor
            //PriorityOrdered类型表明为优先执行
            for (String ppName : postProcessorNames) {
                if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                    //获取对应的bean
                    currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                    //用来存储已经执行过的BeanDefinitionRegistryPostProcessor
                    processedBeans.add(ppName);
                }
            }
            sortPostProcessors(currentRegistryProcessors, beanFactory);
            registryProcessors.addAll(currentRegistryProcessors);
            //开始执行装配逻辑

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
            currentRegistryProcessors.clear();

            // Next, invoke the BeanDefinitionRegistryPostProcessors that
implement Ordered.
            //其次执行类型为Ordered的BeanDefinitionRegistryPostProcessor
            //Ordered表明按顺序执行
            postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
            for (String ppName : postProcessorNames) {
                if (!processedBeans.contains(ppName) &&
beanFactory.isTypeMatch(ppName, Ordered.class)) {
                    currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                    processedBeans.add(ppName);
                }
            }
            sortPostProcessors(currentRegistryProcessors, beanFactory);
            registryProcessors.addAll(currentRegistryProcessors);

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
            currentRegistryProcessors.clear();
            // Finally, invoke all other BeanDefinitionRegistryPostProcessors
until no further ones appear.
            //循环中执行类型不为PriorityOrdered，Ordered类型的
BeanDefinitionRegistryPostProcessor
            boolean reiterate = true;
            while (reiterate) {
                reiterate = false;
                postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
                for (String ppName : postProcessorNames) {
                    if (!processedBeans.contains(ppName)) {

currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                        processedBeans.add(ppName);
                        reiterate = true;
                    }
                }
                sortPostProcessors(currentRegistryProcessors, beanFactory);
                registryProcessors.addAll(currentRegistryProcessors);

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
                currentRegistryProcessors.clear();
            }

            // Now, invoke the postProcessBeanFactory callback of all processors
handled so far.
            //执行父类方法，优先执行注册处理类
            invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
            //执行有规则处理类
            invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);
        }

        else {
            // Invoke factory processors registered with the context instance.
            invokeBeanFactoryPostProcessors(beanFactoryPostProcessors,
beanFactory);
        }

        // Do not initialize FactoryBeans here: We need to leave all regular
beans
        // uninitialized to let the bean factory post-processors apply to them!
        String[] postProcessorNames =
                beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class,
true, false);

        // Separate between BeanFactoryPostProcessors that implement
PriorityOrdered,
        // Ordered, and the rest.
        List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new
ArrayList<>();
        List<String> orderedPostProcessorNames = new ArrayList<>();
        List<String> nonOrderedPostProcessorNames = new ArrayList<>();
        for (String ppName : postProcessorNames) {
            if (processedBeans.contains(ppName)) {
                // skip - already processed in first phase above
            }
            else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, 
BeanFactoryPostProcessor.class));
            }
            else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                orderedPostProcessorNames.add(ppName);
            }
            else {
                nonOrderedPostProcessorNames.add(ppName);
            }
        }

        // First, invoke the BeanFactoryPostProcessors that implement 
PriorityOrdered.
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
        invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors, 
beanFactory);

        // Next, invoke the BeanFactoryPostProcessors that implement Ordered.
        List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList<>
(orderedPostProcessorNames.size());
        for (String postProcessorName : orderedPostProcessorNames) {
            orderedPostProcessors.add(beanFactory.getBean(postProcessorName, 
BeanFactoryPostProcessor.class));
        }
        sortPostProcessors(orderedPostProcessors, beanFactory);
        invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);

        // Finally, invoke all other BeanFactoryPostProcessors.
        List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new 
ArrayList<>(nonOrderedPostProcessorNames.size());
        for (String postProcessorName : nonOrderedPostProcessorNames) {
            nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName, 
BeanFactoryPostProcessor.class));
        }
        invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

        // Clear cached merged bean definitions since the post-processors might 
have
        // modified the original metadata, e.g. replacing placeholders in 
values...
        beanFactory.clearMetadataCache();
    }
```

9、开始执行自动配置逻辑（启动类指定的配置，非默认配置），可以通过debug的方式一层层向里进行查找，会发现最终会在ConfigurationClassParser类中，此类是所有配置类的解析类，所有的解析逻辑在parser.parse(candidates)中

```
public void parse(Set<BeanDefinitionHolder> configCandidates) {
        for (BeanDefinitionHolder holder : configCandidates) {
            BeanDefinition bd = holder.getBeanDefinition();
            try {
                //是否是注解类
                if (bd instanceof AnnotatedBeanDefinition) {
                    parse(((AnnotatedBeanDefinition) bd).getMetadata(), 
holder.getBeanName());
                }
                else if (bd instanceof AbstractBeanDefinition &&
((AbstractBeanDefinition) bd).hasBeanClass()) {
                    parse(((AbstractBeanDefinition) bd).getBeanClass(),
holder.getBeanName());
                }
                else {
                    parse(bd.getBeanClassName(), holder.getBeanName());
                }
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to parse configuration class [" +
bd.getBeanClassName() + "]", ex);
            }
        }
        //执行配置类
        this.deferredImportSelectorHandler.process();
    }
-------------------
        protected final void parse(AnnotationMetadata metadata, String beanName)
throws IOException {
        processConfigurationClass(new ConfigurationClass(metadata, beanName));
    }
-------------------
    protected void processConfigurationClass(ConfigurationClass configClass)
throws IOException {
        if (this.conditionEvaluator.shouldSkip(configClass.getMetadata(),
ConfigurationPhase.PARSE_CONFIGURATION)) {
            return;
        }

        ConfigurationClass existingClass =
this.configurationClasses.get(configClass);
        if (existingClass != null) {
            if (configClass.isImported()) {
                if (existingClass.isImported()) {
                    existingClass.mergeImportedBy(configClass);
                }
                // Otherwise ignore new imported config class; existing non-
imported class overrides it.
                return;
            }
            else {
                // Explicit bean definition found, probably replacing an import.
                // Let's remove the old one and go with the new one.
                this.configurationClasses.remove(configClass);
                this.knownSuperclasses.values().removeIf(configClass::equals);
            }
        }

        // Recursively process the configuration class and its superclass
hierarchy.
        SourceClass sourceClass = asSourceClass(configClass);
        do {
            //循环处理bean,如果有父类，则处理父类，直至结束
            sourceClass = doProcessConfigurationClass(configClass, sourceClass);
        }
        while (sourceClass != null);

        this.configurationClasses.put(configClass, configClass);
    }
```

10、继续跟进doProcessConfigurationClass方法，此方式是支持注解配置的核心逻辑

```
/**
     * Apply processing and build a complete {@link ConfigurationClass} by
reading the
     * annotations, members and methods from the source class. This method can
be called
     * multiple times as relevant sources are discovered.
     * @param configClass the configuration class being build
     * @param sourceClass a source class
     * @return the superclass, or {@code null} if none found or previously
processed
     */
    @Nullable
    protected final SourceClass doProcessConfigurationClass(ConfigurationClass
configClass, SourceClass sourceClass)
            throws IOException {

        //处理内部类逻辑，由于传来的参数是启动类，并不包含内部类，所以跳过
        if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
            // Recursively process any member (nested) classes first
            processMemberClasses(configClass, sourceClass);
        }

        // Process any @PropertySource annotations
        //针对属性配置的解析
        for (AnnotationAttributes propertySource :
AnnotationConfigUtils.attributesForRepeatable(
                sourceClass.getMetadata(), PropertySources.class,
                org.springframework.context.annotation.PropertySource.class)) {
            if (this.environment instanceof ConfigurableEnvironment) {
                processPropertySource(propertySource);
            }
            else {
                logger.info("Ignoring @PropertySource annotation on [" +
sourceClass.getMetadata().getClassName() +
                        "]. Reason: Environment must implement
ConfigurableEnvironment");
            }
        }

        // Process any @ComponentScan annotations
        // 这里是根据启动类@ComponentScan注解来扫描项目中的bean
        Set<AnnotationAttributes> componentScans =
AnnotationConfigUtils.attributesForRepeatable(
                sourceClass.getMetadata(), ComponentScans.class,
ComponentScan.class);
        if (!componentScans.isEmpty() &&
                !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN)) {
```

```
            for (AnnotationAttributes componentScan : componentScans) {
                // The config class is annotated with @ComponentScan -> perform
the scan immediately
                //遍历项目中的bean，如果是注解定义的bean，则进一步解析
                Set<BeanDefinitionHolder> scannedBeanDefinitions =
                        this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());
                // Check the set of scanned definitions for any further config
classes and parse recursively if needed
                for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
                    BeanDefinition bdCand =
holder.getBeanDefinition().getOriginatingBeanDefinition();
                    if (bdCand == null) {
                        bdCand = holder.getBeanDefinition();
                    }
                    if
(ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand,
this.metadataReaderFactory)) {
                        //递归解析，所有的bean,如果有注解，会进一步解析注解中包含的bean
                        parse(bdCand.getBeanClassName(), holder.getBeanName());
                    }
                }
            }
        }

        // Process any @Import annotations
        //递归解析，获取导入的配置类，很多情况下，导入的配置类中会同样包含导入类注解
        processImports(configClass, sourceClass, getImports(sourceClass), true);

        // Process any @ImportResource annotations
        //解析@ImportResource配置类
        AnnotationAttributes importResource =
                AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(),
ImportResource.class);
        if (importResource != null) {
            String[] resources = importResource.getStringArray("locations");
            Class<? extends BeanDefinitionReader> readerClass =
importResource.getClass("reader");
            for (String resource : resources) {
                String resolvedResource =
this.environment.resolveRequiredPlaceholders(resource);
                configClass.addImportedResource(resolvedResource, readerClass);
            }
        }

        // Process individual @Bean methods
        //处理@Bean注解修饰的类
        Set<MethodMetadata> beanMethods =
retrieveBeanMethodMetadata(sourceClass);
        for (MethodMetadata methodMetadata : beanMethods) {
            configClass.addBeanMethod(new BeanMethod(methodMetadata,
configClass));
        }

        // Process default methods on interfaces
        // 处理接口中的默认方法
        processInterfaces(configClass, sourceClass);
        // Process superclass, if any
        //如果该类有父类，则继续返回，上层方法判断不为空，则继续递归执行
        if (sourceClass.getMetadata().hasSuperClass()) {
            String superclass = sourceClass.getMetadata().getSuperClassName();
            if (superclass != null && !superclass.startsWith("java") &&
                    !this.knownSuperclasses.containsKey(superclass)) {
                this.knownSuperclasses.put(superclass, configClass);
                // Superclass found, return its annotation metadata and recurse
                return sourceClass.getSuperClass();
            }
        }

        // No superclass -> processing is complete
        return null;
    }
```

11、查看获取配置类的逻辑

```
processImports(configClass, sourceClass, getImports(sourceClass), true);

    /**
     * Returns {@code @Import} class, considering all meta-annotations.
     */
    private Set<SourceClass> getImports(SourceClass sourceClass) throws
IOException {
        Set<SourceClass> imports = new LinkedHashSet<>();
        Set<SourceClass> visited = new LinkedHashSet<>();
        collectImports(sourceClass, imports, visited);
        return imports;
    }
------------------
        /**
     * Recursively collect all declared {@code @Import} values. Unlike most
     * meta-annotations it is valid to have several {@code @Import}s declared
with
     * different values; the usual process of returning values from the first
     * meta-annotation on a class is not sufficient.
     * <p>For example, it is common for a {@code @Configuration} class to
declare direct
     * {@code @Import}s in addition to meta-imports originating from an {@code
@Enable}
     * annotation.
     * 看到所有的bean都以导入的方式被加载进去
     */
    private void collectImports(SourceClass sourceClass, Set<SourceClass>
imports, Set<SourceClass> visited)
            throws IOException {

        if (visited.add(sourceClass)) {
            for (SourceClass annotation : sourceClass.getAnnotations()) {
                String annName = annotation.getMetadata().getClassName();
                if (!annName.equals(Import.class.getName())) {
                    collectImports(annotation, imports, visited);
                }
            }
imports.addAll(sourceClass.getAnnotationAttributes(Import.class.getName(),
"value"));
        }
    }
```

12、继续回到ConfigurationClassParser中的parse方法中的最后一行,继续跟进该方法:

```
this.deferredImportSelectorHandler.process()
-------------
public void process() {
            List<DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
            this.deferredImportSelectors = null;
            try {
                if (deferredImports != null) {
                    DeferredImportSelectorGroupingHandler handler = new
DeferredImportSelectorGroupingHandler();
                    deferredImports.sort(DEFERRED_IMPORT_COMPARATOR);
                    deferredImports.forEach(handler::register);
                    handler.processGroupImports();
                }
            }
            finally {
                this.deferredImportSelectors = new ArrayList<>();
            }
        }
---------------
  public void processGroupImports() {
            for (DeferredImportSelectorGrouping grouping :
this.groupings.values()) {
                grouping.getImports().forEach(entry -> {
                    ConfigurationClass configurationClass =
this.configurationClasses.get(
                            entry.getMetadata());
                    try {
                        processImports(configurationClass,
asSourceClass(configurationClass),
                                asSourceClasses(entry.getImportClassName()),
false);
                    }
                    catch (BeanDefinitionStoreException ex) {
                        throw ex;
                    }
                    catch (Throwable ex) {
                        throw new BeanDefinitionStoreException(
                                "Failed to process import candidates for
configuration class [" +

configurationClass.getMetadata().getClassName() + "]", ex);
                    }
                });
            }
        }
------------
    /**
         * Return the imports defined by the group.
```

```
          * @return each import with its associated configuration class
          */
        public Iterable<Group.Entry> getImports() {
            for (DeferredImportSelectorHolder deferredImport :
this.deferredImports) {

this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                        deferredImport.getImportSelector());
            }
            return this.group.selectImports();
        }
    }
------------
    public DeferredImportSelector getImportSelector() {
            return this.importSelector;
        }
------------
    @Override
        public void process(AnnotationMetadata annotationMetadata,
DeferredImportSelector deferredImportSelector) {
            Assert.state(deferredImportSelector instanceof
AutoConfigurationImportSelector,
                    () -> String.format("Only %s implementations are supported,
got %s",

AutoConfigurationImportSelector.class.getSimpleName(),
                            deferredImportSelector.getClass().getName()));
            AutoConfigurationEntry autoConfigurationEntry =
((AutoConfigurationImportSelector) deferredImportSelector)
                    .getAutoConfigurationEntry(getAutoConfigurationMetadata(),
annotationMetadata);
            this.autoConfigurationEntries.add(autoConfigurationEntry);
            for (String importClassName :
autoConfigurationEntry.getConfigurations()) {
                this.entries.putIfAbsent(importClassName, annotationMetadata);
            }
        }
```