@Configuration的核心源码/原理

> @Configuration 的问题。

@Configuration 注解起到了哪些作用?
@CompentScan 和 @CompentScans 是怎么被处理的
@Import 注解又是怎么被调用解析的
@Bean 和 @Configuration 一起使用?
@Configuration 是被谁解析的?

首先，每一个注解 都有一个对应的处理类：比如 `@ComponentScan` 它的处理类 `ClassPathBeanDefinitionScanner.class` 又比如 mybatis 的 `@MapperScaner` 它的处理类 `MapperScannerRegistrar.class`

所以呢：`@Configuration` 也有它的对应处理类 `ConfigurationClassParser.class` 。它的入口在 `ConfigurationClassPostProcessor` 这个后置处理器中。

ConfigurationClassParser: 什么时候被调用?

1、由 **AnnotationConfigApplicationContext** 的无参构造 实例话 **AnnotatedBeanDefinitionReader** 的时候，去注册的 `ConfigurationClassPostProcessor`

```
public AnnotationConfigApplicationContext() {
         this.reader = new AnnotatedBeanDefinitionReader(this);
         this.scanner = new ClassPathBeanDefinitionScanner(this);
  }
```

2、**AnnotatedBeanDefinitionReader** 在构造的时候 调用 AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry); 注册处理器。

这段代码不用细看：默认注册5个后置处理器

```
public static Set registerAnnotationConfigProcessors(BeanDefinitionRegistry registry, @Nullable Object source)
{
   DefaultListableBeanFactory beanFactory = unwrapDefaultListableBeanFactory(registry);
    if (beanFactory != null) {
    // AnnotationAwareOrderComparator
        if (!(beanFactory.getDependencyComparator() instanceof AnnotationAwareOrderComparator)) {
            beanFactory.setDependencyComparator(AnnotationAwareOrderComparator.INSTANCE);
        }
        if (!(beanFactory.getAutowireCandidateResolver() instanceof ContextAnnotationAutowireCandidateResolver)) {
            beanFactory.setAutowireCandidateResolver(new ContextAnnotationAutowireCandidateResolver());
        }
    }

    Set<BeanDefinitionHolder> beanDefs = new LinkedHashSet<>(8);
    // 在这会注册 ConfigurationClassPostProcessor 后置处理器
    if (!registry.containsBeanDefinition(CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(ConfigurationClassPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME));
    }

  // AutowiredAnnotationBeanPostProcessor
    if (!registry.containsBeanDefinition(AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(AutowiredAnnotationBeanPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME));
    }

  // RequiredAnnotationBeanPostProcessor
    if (!registry.containsBeanDefinition(REQUIRED_ANNOTATION_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(RequiredAnnotationBeanPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, REQUIRED_ANNOTATION_PROCESSOR_BEAN_NAME));
    }

    // Check for JSR-250 support, and if present add the CommonAnnotationBeanPostProcessor.
    if (jsr250Present && !registry.containsBeanDefinition(COMMON_ANNOTATION_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(CommonAnnotationBeanPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, COMMON_ANNOTATION_PROCESSOR_BEAN_NAME));
    }

    // Check for JPA support, and if present add the PersistenceAnnotationBeanPostProcessor.
    if (jpaPresent && !registry.containsBeanDefinition(PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition();
        try {
            def.setBeanClass(ClassUtils.forName(PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME,
                    AnnotationConfigUtils.class.getClassLoader()));
        }
        catch (ClassNotFoundException ex) {
```

```java
            throw new IllegalStateException(
                    "Cannot load optional framework class: " + PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME, ex);
        }
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME));
    }

    if (!registry.containsBeanDefinition(EVENT_LISTENER_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(EventListenerMethodProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_PROCESSOR_BEAN_NAME));
    }

  // DefaultEventListenerFactory
    if (!registry.containsBeanDefinition(EVENT_LISTENER_FACTORY_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(DefaultEventListenerFactory.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_FACTORY_BEAN_NAME));
    }

    return beanDefs;
}
```

上面这部分源代码：默认的Spring中会有 5个后置处理器分别是：

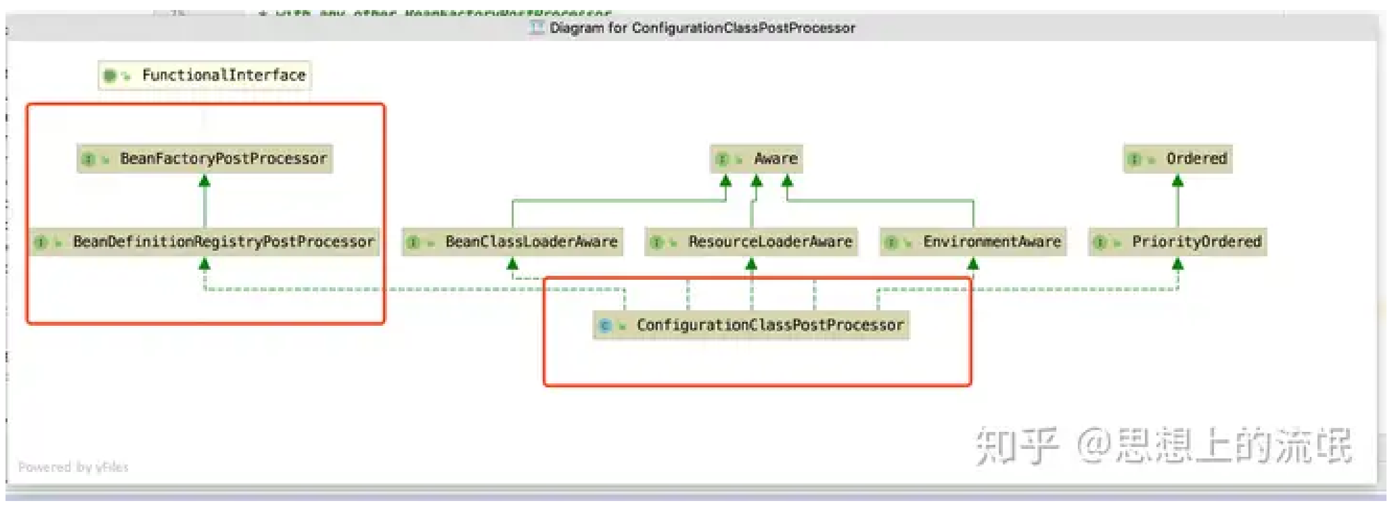
ConfigurationClassPostProcessor：处理 **@Configuration**
AutowiredAnnotationBeanPostProcessor： 处理 **@Autowoired**
RequiredAnnotationBeanPostProcessor： 处理 **@Required** @Autowoired 的是否必须进行检查
CommonAnnotationBeanPostProcessor：处理 **@Resource**
EventListenerMethodProcessor: 处理 **@EventListener**
ConfigurationClassPostProcessor.java 看源码得知他是一个后置处理器，它实现了BeanDefinitionRegistryPostProcessor 而它又实现了 BeanFactoryPostProcessor。所以他就是一个后置处理器。

```java
// ConfigurationClassPostProcessor
public class ConfigurationClassPostProcessor implements BeanDefinitionRegistryPostProcessor,
   PriorityOrdered, ResourceLoaderAware, BeanClassLoaderAware, EnvironmentAware {...}


// BeanDefinitionRegistryPostProcessor
public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {
        void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException;
}


// BeanFactoryPostProcessor
public interface BeanFactoryPostProcessor {
  void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;
}
```

继承关系结构图：



后置处理器会在bean初始化前被调用执行， 入口在于：

AbstractApplicationContext.java 的 refresh() 方法中的 invokeBeanFactoryPostProcessors(beanFactory) 方法，这个方法就是在上下文中调用注册为bean的工厂处理器。就是在bean 实例化之前调用执行。

此后置处理器 被调用 方法 `postProcessBeanDefinitionRegistry(...)` 后执行了该类的 `processConfigBeanDefinitions` 方法：源码如下：

```java
 */
    public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {

        // 存储我们自定义@Configuration的类
        List<BeanDefinitionHolder> configCandidates = new ArrayList<>();

        // 获取注册的bean 这些Bean 就是 AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
        // 方法初始化那几个后置处理器的bean
        String[] candidateNames = registry.getBeanDefinitionNames();

        for (String beanName : candidateNames) {
            BeanDefinition beanDef = registry.getBeanDefinition(beanName);
            if (ConfigurationClassUtils.isFullConfigurationClass(beanDef) ||
                    ConfigurationClassUtils.isLiteConfigurationClass(beanDef)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Bean definition has already been processed as a configuration class: " + beanDef);
                }
            }
            // 记录 @Configoration的候选类
            else if (ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef, this.metadataReaderFactory)) {
                configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));
            }
        }

        // Return immediately if no @Configuration classes were found
        if (configCandidates.isEmpty()) {
            return;
        }

        // Sort by previously determined @Order value, if applicable
        // 做一个排序，实现了@Order
        configCandidates.sort((bd1, bd2) -> {
            int i1 = ConfigurationClassUtils.getOrder(bd1.getBeanDefinition());
            int i2 = ConfigurationClassUtils.getOrder(bd2.getBeanDefinition());
            return Integer.compare(i1, i2);
        });

        // Detect any custom bean name generation strategy supplied through the enclosing application context
        .....



        // Parse each @Configuration class
        // 构造 @Configuration 的处理类 ， 关键入口
        ConfigurationClassParser parser = new ConfigurationClassParser(
                this.metadataReaderFactory, this.problemReporter, this.environment,
                this.resourceLoader, this.componentScanBeanNameGenerator, registry);

        Set<BeanDefinitionHolder> candidates = new LinkedHashSet<>(configCandidates);
        Set<ConfigurationClass> alreadyParsed = new HashSet<>(configCandidates.size());
        do {
            // 关键入口，开始解析 我们的配置类（debug 发现springboot启动流程candidates只包含了启动类）
            parser.parse(candidates);

      // 省略
      // ......
      // 将上面 parser.parse(candidates); 处理的结果的相关类都放到了 一个 Config Hash中，然后将其进行
      // 注册到IOC容器中
      Set<ConfigurationClass> configClasses = new LinkedHashSet<>(parser.getConfigurationClasses());
      // ...
      this.reader.loadBeanDefinitions(configClasses);
        }
        while (!candidates.isEmpty());
        // 省略 ...
    }
```

ConfigurationClassParser.java

调用方法 pares() ----> processConfigurationClass()---->doProcessConfigurationClass(),

1、pares()

一个外部入入口

```java
  public void parse(Set<BeanDefinitionHolder> configCandidates) {
        this.deferredImportSelectors = new LinkedList<>();
        // 在Spring中，会将注册的bean都包装成 BeanDefinitionHolder
        // BeanDefinition 是一个接口，自然就会有不通的子类
        // 此处就是根据不同的BeanDefinition做个分支处理
        // 然后又都会走 processConfigurationClass 方法
        for (BeanDefinitionHolder holder : configCandidates) {
            BeanDefinition bd = holder.getBeanDefinition();
            try {
                if (bd instanceof AnnotatedBeanDefinition) {
                    parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());
         //parse()方法 processConfigurationClass(new ConfigurationClass(metadata, beanName));
         // 将其封装成 ConfigurationClass（这里会把启动类下所有扫描到的类封装成ConfigurationClass）
                }
                else if (bd instanceof AbstractBeanDefinition && ((AbstractBeanDefinition) bd).hasBeanClass()) {
         // processConfigurationClass(new ConfigurationClass(clazz, beanName));

                    parse(((AbstractBeanDefinition) bd).getBeanClass(), holder.getBeanName());
                }
                else {
         //MetadataReader reader = this.metadataReaderFactory.getMetadataReader(className);
                    //processConfigurationClass(new ConfigurationClass(reader, beanName));
                    parse(bd.getBeanClassName(), holder.getBeanName());
                }
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to parse configuration class [" + bd.getBeanClassName() + "]", ex);
            }
        }
        /** 这里会导入自动装配的类，包括springmvc的处理器适配器和处理器映射器（知道处理器映射器为啥能在@Controller修饰的
        类实例化后再实例化了吧，因为它注册的时机就晚了一步），（注意版本不同，这里的代码会有细微的差别，有的版本是
        this.deferredImportSelectorHandler.process();看命名就是处理延迟导入，spingmvc就是延迟导入的一种了，方法命名多规范
        啊）*/
        processDeferredImportSelectors();
    }
```

2、processConfigurationClass()

```
 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) {
            // 如果已经处理过了这个配置类
            if (configClass.isImported()) {
                if (existingClass.isImported()) {
                    // 覆盖 属性
                    // Otherwise ignore new imported config class; existing non-imported class overrides it.
                    existingClass.mergeImportedBy(configClass);
                }
                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 {
            sourceClass = doProcessConfigurationClass(configClass, sourceClass);
        }
        while (sourceClass != null);

        // 记录已经处理过的配置类，下面会拿出这些类，将其注册到IOC中
        this.configurationClasses.put(configClass, configClass);
    }
```

3、关键看：doProcessConfigurationClass()

```
 protected final SourceClass doProcessConfigurationClass(ConfigurationClass configClass, SourceClass sourceClass)
            throws IOException {

        // Recursively process any member (nested) classes first
        // 处理内部类
        processMemberClasses(configClass, sourceClass);

        // Process any @PropertySource annotations
        // 处理 PropertySources 注解 加载资源文件
        for (AnnotationAttributes propertySource : AnnotationConfigUtils.attributesForRepeatable(
                sourceClass.getMetadata(), PropertySources.class,
                org.springframework.context.annotation.PropertySource.class)) {
            if (this.environment instanceof ConfigurableEnvironment) {
                processPropertySource(propertySource);
            }
```

```java
            else {
                logger.warn("Ignoring @PropertySource annotation on [" + sourceClass.getMetadata().getClassName() +
                        "]. Reason: Environment must implement ConfigurableEnvironment");
            }
        }

        // Process any @ComponentScan annotations
        // 处理包扫描注解 ComponentScans  ComponentScan 交给 ComponentScanAnnotationParser 去解析，然后再交给 扫描类去扫描包
        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
                // ComponentScanAnnotationParser 去解析@ComponentScan 注解，然后交给 ClassPathBeanDefinitionScanner 扫描器去扫描注册b
                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
                // 这里会做一步递归解析，检查 是否还存在有 @Configation 的注解类
                for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
                    BeanDefinition bdCand = holder.getBeanDefinition().getOriginatingBeanDefinition();
                    if (bdCand == null) {
                        bdCand = holder.getBeanDefinition();
                    }
                    // 是否是配置类，是 做一递归
                    if (ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand, this.metadataReaderFactory)) {
                        parse(bdCand.getBeanClassName(), holder.getBeanName());
                    }
                }
            }
        }

        // Process any @Import annotations
        // 处理 @Import 注解，加载某个类，将其假如IOC容器中
        processImports(configClass, sourceClass, getImports(sourceClass), true);

        // Process any @ImportResource annotations
        // 处理加载 第三方自定义的资源文件
        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，将实例加载到IOC容器中
        Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(sourceClass);
        for (MethodMetadata methodMetadata : beanMethods) {
            configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
        }

        // Process default methods on interfaces
        // 处理接口上的方法
        processInterfaces(configClass, sourceClass);

        // Process superclass, if any
        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
                // 存在父类 继续循环
                return sourceClass.getSuperClass();
            }
        }

        // No superclass -> processing is complete
    // 返回空，退出循环
        return null;
    }
```

4、看一下 @Import 注解的处理

```java
// Process any @Import annotations
// 处理 @Import 注解，加载某个类，将其假如IOC容器中 擦数： 当前的配置类，当前的源码类，导入的类
processImports(configClass, sourceClass, getImports(sourceClass), true);
```

先看一下 @Import注解的源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Import {

    /**
     * {@link Configuration}, {@link ImportSelector}, {@link ImportBeanDefinitionRegistrar}
     * or regular component classes to import.
     *
     *  这个value值 有三种类型:
     *      -   其它的常规类型，就相当于Configuration配置累解析
     *      -   ImportSelector
     *      -   ImportBeanDefinitionRegistrar
     *
     *
     */
    Class<?>[] value();

}
```

在看一下 **processImports(configClass, sourceClass, getImports(sourceClass), true);**

```
// 处理 @Import 注解，加载某个类，将其假如IOC容器中 擦数:  当前的配置类，当前的源码类，导入的类
    private void processImports(ConfigurationClass configClass, SourceClass currentSourceClass,
            Collection<SourceClass> importCandidates, boolean checkForCircularImports) {

        if (importCandidates.isEmpty()) {
            return;
        }

        if (checkForCircularImports && isChainedImportOnStack(configClass)) {
            this.problemReporter.error(new CircularImportProblem(configClass, this.importStack));
        }
        else {
            this.importStack.push(configClass);
            try {
                // @Import 注解 可以配置多个类没循环遍历
                // 该注解上面说了 有三种类型，遍历做分支处理
                for (SourceClass candidate : importCandidates) {

                    if (candidate.isAssignable(ImportSelector.class)) {
                        // Candidate class is an ImportSelector -> delegate to it to determine imports
                        Class<?> candidateClass = candidate.loadClass();
                        ImportSelector selector = BeanUtils.instantiateClass(candidateClass, ImportSelector.class);
                        ParserStrategyUtils.invokeAwareMethods(
                                selector, this.environment, this.resourceLoader, this.registry);

                        //延迟导入处理
                        if (this.deferredImportSelectors != null && selector instanceof DeferredImportSelector) {

                            this.deferredImportSelectors.add(
                                    new DeferredImportSelectorHolder(configClass, (DeferredImportSelector) selector));
                        }
                        else {

            // 执行 ImportSelector 接口的方法
                            String[] importClassNames = selector.selectImports(currentSourceClass.getMetadata());
            // 根据返回的类 使用递归
                            Collection<SourceClass> importSourceClasses = asSourceClasses(importClassNames);
                            processImports(configClass, currentSourceClass, importSourceClasses, false);
                        }
                    }
                    else if (candidate.isAssignable(ImportBeanDefinitionRegistrar.class)) {
                        // Candidate class is an ImportBeanDefinitionRegistrar ->
                        // delegate to it to register additional bean definitions
                        Class<?> candidateClass = candidate.loadClass();
                        ImportBeanDefinitionRegistrar registrar =
                                BeanUtils.instantiateClass(candidateClass, ImportBeanDefinitionRegistrar.class);
                        ParserStrategyUtils.invokeAwareMethods(
                                registrar, this.environment, this.resourceLoader, this.registry);
                        configClass.addImportBeanDefinitionRegistrar(registrar, currentSourceClass.getMetadata());
                    }
                    else {
                        // Candidate class not an ImportSelector or ImportBeanDefinitionRegistrar ->
                        // process it as an @Configuration class
                        // 当作配置类解析
                        this.importStack.registerImport(
                                currentSourceClass.getMetadata(), candidate.getMetadata().getClassName());
                        processConfigurationClass(candidate.asConfigClass(configClass));
                    }
                }
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to process import candidates for configuration class [" +
                        configClass.getMetadata().getClassName() + "]", ex);
            }
            finally {
                this.importStack.pop();
            }
        }
    }
```

归类总结:

> **doProcessConfigurationClass** 对该配置累做了不同的配置处理:

处理内部类: @Compent 注解的 例如

```
        @Configuration
        static class AppConfig
        {
                @Component
                class Apple{}
        }
```

处理 @PropertySources 注解 加载资源文件
处理包扫描注解 ComponentScans. ComponentScan 交给 ComponentScanAnnotationParser 去解析，然后再交给 扫描类去扫描包。对扫描包后得到的结果，进行遍历，是否还存在有配置类，否则进行递归，解析Config类。
处理 @Import 注解。
处理加载 @ImportResource 第三方自定义的资源文件
处理 @Bean注解
处理接口的方法

还继续检查是否存在相关父类，返回空就退出循环

在上面的步骤走完之后，会将其相关解析出来的类，放到了 ConfigurationClass 类中，然后又将ConfigurationClass类 放到了 configurationClasses 的hash中（processConfigurationClass() 在此方法上的操作）。

```
  parser.parse(candidates); // 该方法就是上面的操作步骤的入口 处理完后就开始处理 解析的结果类。
```

// 处理解析后的结果:

```
 Set<ConfigurationClass> configClasses = new LinkedHashSet<>(parser.getConfigurationClasses());
```

// 注册Bean

```
 public Set<ConfigurationClass> getConfigurationClasses() {
    return this.configurationClasses.keySet();
 }
```

```
 this.reader.loadBeanDefinitions(configClasses);
```

@Configuration 注解起到了哪些作用?

该注解，可以通过api的方式启动加载Spring，作为启动Spring的入口，那它还要提供发现其它相关配置的功能，比如发现 @CompentScan @Bean @Import...这些相关的配置，然后交给对应的类去调用处理。

@CompentScan 和 @CompentScans 是怎么被处理的?

该注解会被@Configuration 的注解类，去发现，然后交给 `ComponentScanAnnotationParser` 去解析@CompenScan ,前期填充好一些扫描的规则：比如是否是懒加载啊，扫描的时候是否需要排除掉某些类（接口、抽象）、扫描的包啊.。然后在将其交给 `ClassPathBeanDefinitionScanner` 去扫描注册该包下相关类。

@Import 注解又是怎么被调用解析的

@Import 导入的类 有三种，分别是：ImportSelector、 ImportBeanDefinitionRegistrar、一种是普通各类，会当作为配置类去处理。根据这三种类做分支去处理。如上。

@Configuration 是被谁解析的?

ConfigurationClassParser.class 由它解析并发现其它相关配置。
至此，@Configuration的核心源码 分析结束

5、@Configuration的使用

1、测试程序

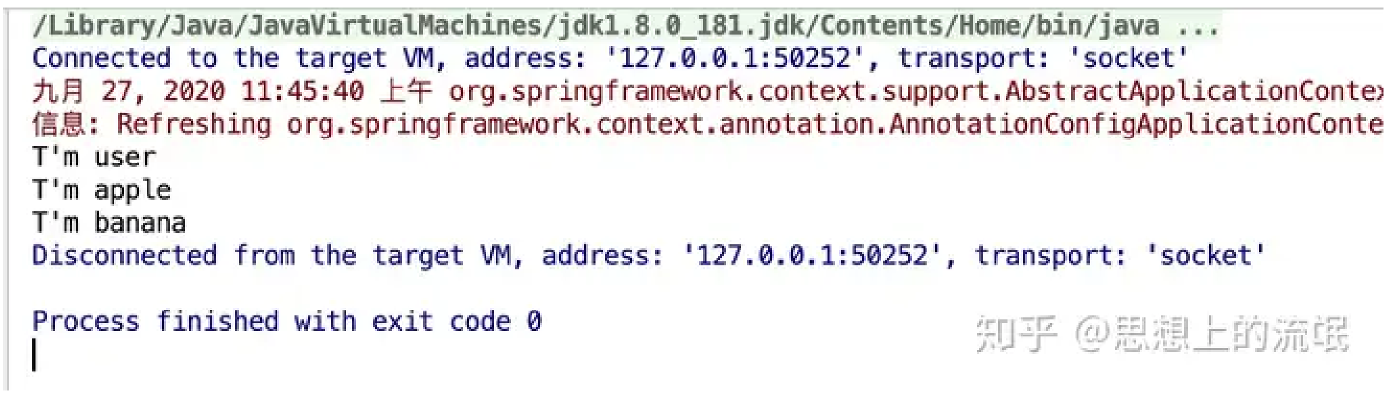
```java
public class App
{
    public static void main(String[] args)
    {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        context.getBean(User.class).who();
        context.getBean(AppConfig.Apple.class).who();
        context.getBean(Banana.class).who();
    }

    @Configuration
    @Import({Banana.class,MyImportSelector.class,MyImportBeanDefinitionRegistrar.class})
    //@ComponentScan("org.spring.demo.beans_cycle")
    static class AppConfig
    {
        @Component
        class Apple{
            public void who(){
                System.out.println("T'm apple");
            }
        }

        @Bean
        public User user()
        {
            return new User();
        }
    }

    static class User
    {
        public void who(){
            System.out.println("T'm user");
        }
    }
    static class Banana
    {
        public void who(){
            System.out.println("T'm banana");
        }
    }
}
```

> 使用AnnotationConfigApplicationContext，启动加载Spring的上下文，且省去了xml 的配置。



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:50252', transport: 'socket'
九月 27, 2020 11:45:40 上午 org.springframework.context.support.AbstractApplicationConte
信息: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationConte
T'm user
T'm apple
T'm banana
Disconnected from the target VM, address: '127.0.0.1:50252', transport: 'socket'

Process finished with exit code 0
```

本文讲的是自定义@Configutation的原理，注意和自动配置类的注册的区别，比如spring mvc的RequestMappingHandlerMapping需要在我们自定义的bean实例化后才实例化（因为需要扫描@Controller相关bean），那么它的注册时机肯定晚于我们自定义的bean,而处理器映射器是通过WebMvcAutoConfiguration这个配置类的@bean 实例化的，而自动配置类的注册时候晚于我们自定义的bean,我在parse方法里已经注释了。

Spring源码之@Configuration原理

总结

1. @Configuration注解的Bean，在BeanDefinition加载注册到IOC容器之后，进行postProcessBeanFactory处理时会进行CGLIB动态代理
2. 将@PropertySource、@ComponentScan、@Import、@ImportResource、@Bean等直接注解的类的BeanDefinition，是在ConfigurationClassParser#parse()中直接进行加载注册
3. 通过ConfigurationClassBeanDefinitionReader#loadBeanDefinitions()开始将@Configuration注解类内部@Import、@Bean进行BeanDefinition的加载注册

简单例子

```
@Configuration
public class ConfigTest {
    @Bean
    public ConfigBean configBean() {
        return  new ConfigBean();
    }
}
```

```
public class ConfigBean {......}
```

对@Configuration的注解类进行CGLIB动态代理

调用链：

> AbstractApplicationContext#refresh() --> AbstractApplicationContext#invokeBeanFactoryPostProcessors() --> PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors() --> PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors() --> ConfigurationClassPostProcessor#postProcessBeanFactory() --> ConfigurationClassPostProcessor#enhanceConfigurationClasses()

在@Configuration注解的类attributes中有<org.springframework.context.annotation.ConfigurationClassPostProcessor.configurationClass, <org.springframework.context.annotation.ConfigurationClassPostProcessor.configurationClass, full>>值（具体怎么通过反射从class文件获取@Configuration attributes，详见前文《Spring源码之注解的原理》）

Object configClassAttr = beanDef.getAttribute(ConfigurationClassUtils.CONFIGURATION_CLASS_ATTRIBUTE);的值为full

```
public void enhanceConfigurationClasses(ConfigurableListableBeanFactory beanFactory) {
        StartupStep enhanceConfigClasses = this.applicationStartup.start("spring.context.config-classes.enhance");
        Map<String, AbstractBeanDefinition> configBeanDefs = new LinkedHashMap<>();
        for (String beanName : beanFactory.getBeanDefinitionNames()) {
                BeanDefinition beanDef = beanFactory.getBeanDefinition(beanName);
                // 执行到@Configuration注解类, configClassAttr的value为full
                Object configClassAttr = beanDef.getAttribute(ConfigurationClassUtils.CONFIGURATION_CLASS_ATTRIBUTE);
                MethodMetadata methodMetadata = null;
                if (beanDef instanceof AnnotatedBeanDefinition) {
                        methodMetadata = ((AnnotatedBeanDefinition) beanDef).getFactoryMethodMetadata();
                }
                if ((configClassAttr != null || methodMetadata != null) && beanDef instanceof AbstractBeanDefinition) {
                        // Configuration class (full or lite) or a configuration-derived @Bean method
                        // -> resolve bean class at this point...
                        AbstractBeanDefinition abd = (AbstractBeanDefinition) beanDef;
                        if (!abd.hasBeanClass()) {
                                try {
                                        abd.resolveBeanClass(this.beanClassLoader);
                                }
                                catch (Throwable ex) {
                                        throw new IllegalStateException(
                                                        "Cannot load configuration class: " + beanDef.getBeanClassName(), ex);
                                }
                        }
                }
                //true,执行下面put逻辑
                if (ConfigurationClassUtils.CONFIGURATION_CLASS_FULL.equals(configClassAttr)) {
                        if (!(beanDef instanceof AbstractBeanDefinition)) {
                                throw new BeanDefinitionStoreException("Cannot enhance @Configuration bean definition '" +
                                                beanName + "' since it is not stored in an AbstractBeanDefinition subclass");
                        }
                        else if (logger.isInfoEnabled() && beanFactory.containsSingleton(beanName)) {
                                logger.info("Cannot enhance @Configuration bean definition '" + beanName +
                                                "' since its singleton instance has been created too early. The typical cause " +
                                                "is a non-static @Bean method with a BeanDefinitionRegistryPostProcessor " +
                                                "return type: Consider declaring such methods as 'static'.");
                        }
                        configBeanDefs.put(beanName, (AbstractBeanDefinition) beanDef);
                }
        }
        if (configBeanDefs.isEmpty()) {
                // nothing to enhance -> return immediately
                enhanceConfigClasses.end();
                return;
        }
        if (IN_NATIVE_IMAGE) {
                throw new BeanDefinitionStoreException("@Configuration classes need to be marked as " +
                                "proxyBeanMethods=false. Found: " + configBeanDefs.keySet());
        }

    // 进行CGLIB动态代理
        ConfigurationClassEnhancer enhancer = new ConfigurationClassEnhancer();
        for (Map.Entry<String, AbstractBeanDefinition> entry : configBeanDefs.entrySet()) {
                AbstractBeanDefinition beanDef = entry.getValue();
                // If a @Configuration class gets proxied, always proxy the target class
                beanDef.setAttribute(AutoProxyUtils.PRESERVE_TARGET_CLASS_ATTRIBUTE, Boolean.TRUE);
                // Set enhanced subclass of the user-specified bean class
                Class<?> configClass = beanDef.getBeanClass();
                Class<?> enhancedClass = enhancer.enhance(configClass, this.beanClassLoader);
                if (configClass != enhancedClass) {
                        if (logger.isTraceEnabled()) {
                                logger.trace(String.format("Replacing bean definition '%s' existing class '%s' with " +
                                                "enhanced class '%s'", entry.getKey(), configClass.getName(), enhancedClass.getName()));
                        }
                        beanDef.setBeanClass(enhancedClass);
                }
        }
        enhanceConfigClasses.tag("classCount", () -> String.valueOf(configBeanDefs.keySet().size())).end();
}
```

ConfigurationClassParser#parse()扫描出configClasses

springboot启动时，在AbstractApplicationContext#refresh()中invokeBeanFactoryPostProcessors(beanFactory)会将@PropertySource、@ComponentScan、@Import、@ImportResource、@Bean注解的类进行生成BeanDefinition，并加载注册。并将ConfigurationClass进行缓存

```
private final Map<ConfigurationClass, ConfigurationClass> configurationClasses = new LinkedHashMap<>();

protected void processConfigurationClass(ConfigurationClass configClass, Predicate<String> filter) throws IOException {
    this.configurationClasses.put(configClass, configClass);
}
```

详见前文《Spring源码之IOC容器创建、BeanDefinition加载和注册和IOC容器依赖注入》

loadBeanDefinitions处理

调用链:

> AbstractApplicationContext#refresh() --> AbstractApplicationContext#invokeBeanFactoryPostProcessors() --> PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors() --> PostProcessorRegistrationDelegate#invokeBeanDefinitionRegistryPostProcessors() --> ConfigurationClassPostProcessor#postProcessBeanDefinitionRegistry () --> ConfigurationClassPostProcessor#processConfigBeanDefinitions() --> ConfigurationClassBeanDefinitionReader#loadBeanDefinitions() --> ConfigurationClassBeanDefinitionReader#loadBeanDefinitionsForConfigurationClass()

```
private void loadBeanDefinitionsForConfigurationClass(
                ConfigurationClass configClass, TrackedConditionEvaluator trackedConditionEvaluator) {

        if (trackedConditionEvaluator.shouldSkip(configClass)) {
                String beanName = configClass.getBeanName();
                if (StringUtils.hasLength(beanName) && this.registry.containsBeanDefinition(beanName)) {
                        this.registry.removeBeanDefinition(beanName);
                }
                this.importRegistry.removeImportingClass(configClass.getMetadata().getClassName());
                return;
        }

    //@Configuration注解内部@Import注解方法处理
        if (configClass.isImported()) {
                registerBeanDefinitionForImportedConfigurationClass(configClass);
        }
        //@Configuration注解内部@Bean注解方法处理
        for (BeanMethod beanMethod : configClass.getBeanMethods()) {
                loadBeanDefinitionsForBeanMethod(beanMethod);
        }

        loadBeanDefinitionsFromImportedResources(configClass.getImportedResources());
        loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars());
}
```

处理Bean的一系列属性后，向IOC容器中开始注册。this.registry.registerBeanDefinition()，注册逻辑可详见前文。

```java
/**
 * Read the given {@link BeanMethod}, registering bean definitions
 * with the BeanDefinitionRegistry based on its contents.
 */
@SuppressWarnings("deprecation")  // for RequiredAnnotationBeanPostProcessor.SKIP_REQUIRED_CHECK_ATTRIBUTE
private void loadBeanDefinitionsForBeanMethod(BeanMethod beanMethod) {
	ConfigurationClass configClass = beanMethod.getConfigurationClass();
	MethodMetadata metadata = beanMethod.getMetadata();
	String methodName = metadata.getMethodName();

	// Do we need to mark the bean as skipped by its condition?
	if (this.conditionEvaluator.shouldSkip(metadata, ConfigurationPhase.REGISTER_BEAN)) {
		configClass.skippedBeanMethods.add(methodName);
		return;
	}
	if (configClass.skippedBeanMethods.contains(methodName)) {
		return;
	}

	AnnotationAttributes bean = AnnotationConfigUtils.attributesFor(metadata, Bean.class);
	Assert.state(bean != null, "No @Bean annotation attributes");

	// Consider name and any aliases
	List<String> names = new ArrayList<>(Arrays.asList(bean.getStringArray("name")));
	String beanName = (!names.isEmpty() ? names.remove(0) : methodName);

	// Register aliases even when overridden
	for (String alias : names) {
		this.registry.registerAlias(beanName, alias);
	}

	// Has this effectively been overridden before (e.g. via XML)?
	if (isOverriddenByExistingDefinition(beanMethod, beanName)) {
		if (beanName.equals(beanMethod.getConfigurationClass().getBeanName())) {
			throw new BeanDefinitionStoreException(beanMethod.getConfigurationClass().getResource().getDescription(),
					beanName, "Bean name derived from @Bean method '" + beanMethod.getMetadata().getMethodName() +
					"' clashes with bean name for containing configuration class; please make those names unique!");
		}
		return;
	}

	ConfigurationClassBeanDefinition beanDef = new ConfigurationClassBeanDefinition(configClass, metadata, beanName);
	beanDef.setSource(this.sourceExtractor.extractSource(metadata, configClass.getResource()));

	if (metadata.isStatic()) {
		// static @Bean method
		if (configClass.getMetadata() instanceof StandardAnnotationMetadata) {
			beanDef.setBeanClass(((StandardAnnotationMetadata) configClass.getMetadata()).getIntrospectedClass());
		}
		else {
			beanDef.setBeanClassName(configClass.getMetadata().getClassName());
		}
		beanDef.setUniqueFactoryMethodName(methodName);
	}
	else {
		// instance @Bean method
		beanDef.setFactoryBeanName(configClass.getBeanName());
		beanDef.setUniqueFactoryMethodName(methodName);
	}

	if (metadata instanceof StandardMethodMetadata) {
		beanDef.setResolvedFactoryMethod(((StandardMethodMetadata) metadata).getIntrospectedMethod());
	}

	beanDef.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_CONSTRUCTOR);
	beanDef.setAttribute(org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor.
			SKIP_REQUIRED_CHECK_ATTRIBUTE, Boolean.TRUE);

	AnnotationConfigUtils.processCommonDefinitionAnnotations(beanDef, metadata);

	Autowire autowire = bean.getEnum("autowire");
	if (autowire.isAutowire()) {
		beanDef.setAutowireMode(autowire.value());
	}

	boolean autowireCandidate = bean.getBoolean("autowireCandidate");
	if (!autowireCandidate) {
		beanDef.setAutowireCandidate(false);
	}

	String initMethodName = bean.getString("initMethod");
	if (StringUtils.hasText(initMethodName)) {
		beanDef.setInitMethodName(initMethodName);
```

```
        }

        String destroyMethodName = bean.getString("destroyMethod");
        beanDef.setDestroyMethodName(destroyMethodName);

        // Consider scoping
        ScopedProxyMode proxyMode = ScopedProxyMode.NO;
        AnnotationAttributes attributes = AnnotationConfigUtils.attributesFor(metadata, Scope.class);
        if (attributes != null) {
                beanDef.setScope(attributes.getString("value"));
                proxyMode = attributes.getEnum("proxyMode");
                if (proxyMode == ScopedProxyMode.DEFAULT) {
                        proxyMode = ScopedProxyMode.NO;
                }
        }

        // Replace the original bean definition with the target one, if necessary
        BeanDefinition beanDefToRegister = beanDef;
        if (proxyMode != ScopedProxyMode.NO) {
                BeanDefinitionHolder proxyDef = ScopedProxyCreator.createScopedProxy(
                                new BeanDefinitionHolder(beanDef, beanName), this.registry,
                                proxyMode == ScopedProxyMode.TARGET_CLASS);
                beanDefToRegister = new ConfigurationClassBeanDefinition(
                                (RootBeanDefinition) proxyDef.getBeanDefinition(), configClass, metadata, beanName);
        }

        if (logger.isTraceEnabled()) {
                logger.trace(String.format("Registering bean definition for @Bean method %s.%s()",
                                configClass.getMetadata().getClassName(), beanName));
        }
        this.registry.registerBeanDefinition(beanName, beanDefToRegister);
}
```

@Configuration CGLIB增强的功能

> https://www.cnblogs.com/fnlingnzb-learner/p/10762905.html
> https://blog.csdn.net/weixin_42997554/article/details/104578710

例子:

```
@Component
public class ConfigTest {
    @Bean
    public ConfigBean configBean() {

        ConfigBean configBean = new ConfigBean();
        System.out.println(configBean + "-----------@Component");
        return configBean;
    }

    @Bean ConfigBean2 configBean2() {
        return new ConfigBean2(configBean());
    }
}
```

```
public class ConfigBean {}
public class ConfigBean2 {
    public ConfigBean2(ConfigBean configBean) {
        System.out.println(configBean + "-------configBean2");
    }
}
```

打印:

```
com.java.study.spring.bean.configuration.ConfigBean@1d8e2eea-----------@Component
com.java.study.spring.bean.configuration.ConfigBean@240139e1-----------@Component
com.java.study.spring.bean.configuration.ConfigBean@240139e1-------configBean2
```

换成@Configuration注解时:

```
@Configuration
public class ConfigTest {
    @Bean
    public ConfigBean configBean() {

        ConfigBean configBean = new ConfigBean();
        System.out.println(configBean + "-----------@Component");
        return configBean;
    }

    @Bean ConfigBean2 configBean2() {
        return new ConfigBean2(configBean());
    }
}
```

打印：

```
com.java.study.spring.bean.configuration.ConfigBean@1d81e101-----------@Component
com.java.study.spring.bean.configuration.ConfigBean@1d81e101-------configBean2
```

结论：
@Configuration通过CGLIB进行增强时，方法里面@Bean的对象都会和@Configuration注解的类scope一样，是单例的。@Component则会创建多个对象。

源码

在AbstractAutowireCapableBeanFactory#createBeanInstance时

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
        if (mbd.getFactoryMethodName() != null) {
                return instantiateUsingFactoryMethod(beanName, mbd, args);
        }
}
```

会调用ConfigurationClassEnhancer的内部类BeanMethodInterceptor的intercept拦截

```
public Object intercept(Object enhancedConfigInstance, Method beanMethod, Object[] beanMethodArgs,
                        MethodProxy cglibMethodProxy) throws Throwable {

        ConfigurableBeanFactory beanFactory = getBeanFactory(enhancedConfigInstance);
        String beanName = BeanAnnotationHelper.determineBeanNameFor(beanMethod);

        if (isCurrentlyInvokedFactoryMethod(beanMethod)) {
                // The factory is calling the bean method in order to instantiate and register the bean
                // (i.e. via a getBean() call) -> invoke the super implementation of the method to actually
                // create the bean instance.
                // 第一次创建
                return cglibMethodProxy.invokeSuper(enhancedConfigInstance, beanMethodArgs);
        }

        return resolveBeanReference(beanMethod, beanMethodArgs, beanFactory, beanName);
}
```

第二次实例化ConfigBean时，isCurrentlyInvokedFactoryMethod(beanMethod)为false，走入resolveBeanReference方法

通过getBean直接从容器中获取

```
private Object resolveBeanReference(Method beanMethod, Object[] beanMethodArgs,ConfigurableBeanFactory beanFactory, String beanName) {

        Object beanInstance = (useArgs ? beanFactory.getBean(beanName, beanMethodArgs) :
                        beanFactory.getBean(beanName));
}
```

Spring会为@Configuration注解修饰的类生成一个cglib代理类，当调用 @Bean修饰的方法时会被代理类的拦截器拦截，如果是第一次调用则生成新的bean，如果是第二次调用则直接从ioc容器里获取，不走新建流程，从而保证单例。而使用@Component修饰的类中的@Bean方法是不能保证单例的，每调用一次就生成新的实例。