- 在 Springboot 源码分析—— 总纲 中介绍了 Springboot 启动的总过程，本文将详细解析其中的 prepareEnvironment() 方法。
- refreshContext() 方法主要是为了解析配置文件，加载业务 bean，启动 tomcat 等。

1 refreshContext()：解析配置文件，加载业务 `bean`，启动 `tomcat` 等

```
//SpringApplication：
private void refreshContext(ConfigurableApplicationContext context) {
    this.refresh(context);
    //9.14 注册shutdown后的逻辑
    context.registerShutdownHook();
}
//ServletWebServerApplicationContext中的方法
public final void refresh() throws BeansException, IllegalStateException {
    super.refresh();
}
//AbstractApplicationContext类中的方法
public void refresh() throws BeansException, IllegalStateException {
    Object var1 = this.startupShutdownMonitor;
    synchronized(this.startupShutdownMonitor) {
        //初始化一些配置属性，验证配置文件
        this.prepareRefresh();
        //简单的获取beanFactory
        ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();
        //将context中的一些属性设置到beanFactory中
        this.prepareBeanFactory(beanFactory);
        //注册Scope相关的类
        this.postProcessBeanFactory(beanFactory);
        //1.1 解析配置文件、生成所有的beanDefinitions
```

```
            this.invokeBeanFactoryPostProcessors(beanFactory);
            //1.2 分类、排序、注册（注入）所有的BeanPostProcessors
            this.registerBeanPostProcessors(beanFactory);
            //国际化
            this.initMessageSource();
            //初始化多播事件
            this.initApplicationEventMulticaster();
            //主要创建并初始化容器
            this.onRefresh();
            //向多播事件注册监听者
            this.registerListeners();
            //1.3 主要是初始化非懒加载单例
            this.finishBeanFactoryInitialization(beanFactory);
            //主要是开启Web容器
            this.finishRefresh();
            //清除一些缓存
            this.resetCommonCaches();
        }
    }
```

1-1 invokeBeanFactoryPostProcessors()：解析配置文件、生成所有的 beanDefinitions

- 解析配置文件，生成 `beanDefinitions`
- 替换 `BeanDefinitions` 中的占位符
- 遍历 `BeanDefinitions` 设置 `beansFactoryMetadata`

```
//PostProcessorRegistrationDelegate：遍历调用3种PostProcessors
//1 SharedMetadataReaderFactoryContextInitializer 2 ConfigFileApplicationListener 3
ConfigurationWarningsApplicationContextInitializer
public static void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory
beanFactory, List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {

    if (beanFactory instanceof BeanDefinitionRegistry) {
        BeanDefinitionRegistry registry = (BeanDefinitionRegistry)beanFactory;
        //这里有3个
        Iterator var6 = beanFactoryPostProcessors.iterator();
        while(var6.hasNext()) {
            BeanFactoryPostProcessor postProcessor =
(BeanFactoryPostProcessor)var6.next();
            if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
                BeanDefinitionRegistryPostProcessor registryProcessor =
(BeanDefinitionRegistryPostProcessor)postProcessor;
                //注册registry
                registryProcessor.postProcessBeanDefinitionRegistry(registry);
                //这里有2个：SharedMetadataReaderFactoryContextInitializer、
ConfigurationWarningsApplicationContextInitializer
                registryProcessors.add(registryProcessor);
            } else {
                //这里有1个 ：ConfigFileApplicationListener
```

```
                regularPostProcessors.add(postProcessor);
            }
        }

        currentRegistryProcessors = new ArrayList();
        //第1次获取-------------------
        //从7个beanDefinitionNames中获取匹配的，只有一个为：
internalConfigurationAnnotationProcessor
        postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
        String[] var16 = postProcessorNames;
        var9 = postProcessorNames.length;
        int var10;
        String ppName;
        for(var10 = 0; var10 < var9; ++var10) {
            ppName = var16[var10];
            //只有满足PriorityOrdered的才会被放入processedBeans
            if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                processedBeans.add(ppName);
            }
        }
        //排序currentRegistryProcessors：只有一个元素：configurationClassPostProcessor
        sortPostProcessors(currentRegistryProcessors, beanFactory);
        registryProcessors.addAll(currentRegistryProcessors);
        //第1次解析-------------------满足PriorityOrdered
        //internalConfigurationAnnotationProcessor满足PriorityOrdered
        //currentRegistryProcessors为：configurationClassPostProcessor
        //2 最重要的一步：解析配置文件！！第2次、第3次获取都没有进入
        invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
        //清除
        currentRegistryProcessors.clear();

        //第2次获取-------------------
        postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
        var16 = postProcessorNames;
        var9 = postProcessorNames.length;
        for(var10 = 0; var10 < var9; ++var10) {
            ppName = var16[var10];
            //如果ppName对应的不是PriorityOrdered，就不存在
            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);
        //第2次解析------------------满足Ordered
        //currentRegistryProcessors=0，没有作用
        invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
        currentRegistryProcessors.clear();

        boolean reiterate = true;
        while(reiterate) {
            reiterate = false;
            //第3次获取------------------
            postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true,
false);
            String[] var19 = postProcessorNames;
            var10 = postProcessorNames.length;

            for(int var26 = 0; var26 < var10; ++var26) {
                String ppName = var19[var26];
                //不为PriorityOrdered和Ordered就不存在
                if (!processedBeans.contains(ppName)) {
                    currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                    processedBeans.add(ppName);
                    //如果有的话，while不退出
                    reiterate = true;
                }
            }
            sortPostProcessors(currentRegistryProcessors, beanFactory);
            registryProcessors.addAll(currentRegistryProcessors);
            //第3次解析------------------不满足Ordered和PriorityOrdered
            //currentRegistryProcessors=0，没有作用
            invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
            currentRegistryProcessors.clear();
        }

        //每次获取postProcessorNames都会将解析出的currentRegistryProcessors放入
registryProcessors
        //我的程序这里最终仅仅是注册了一个 ImportAwareBeanPostProcessor 处理器：
        //基本上所有的 aware处理器功能都比较简单，仅仅是设置一些值
        invokeBeanFactoryPostProcessors((Collection)registryProcessors,
(ConfigurableListableBeanFactory)beanFactory);
        //这里仅仅是去除了 `defalut` profiles
        invokeBeanFactoryPostProcessors((Collection)regularPostProcessors,
(ConfigurableListableBeanFactory)beanFactory);
    } else {
        //如果不是：beanFactory instanceof BeanDefinitionRegistry
        invokeBeanFactoryPostProcessors((Collection)beanFactoryPostProcessors,
(ConfigurableListableBeanFactory)beanFactory);
    }

    //第1次获取------------------BeanFactoryPostProcessor!
```

```
    //这里获取的是之前获取的父集因为：BeanDefinitionRegistryPostProcessor extends
BeanFactoryPostProcessor
    //这里有5个：
    String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);
    regularPostProcessors = new ArrayList();
    registryProcessors = new ArrayList();
    currentRegistryProcessors = new ArrayList();
    postProcessorNames = postProcessorNames;
    int var20 = postProcessorNames.length;

    String ppName;
    for(var9 = 0; var9 < var20; ++var9) {
        ppName = postProcessorNames[var9];
        //除去之前BeanDefinitionRegistryPostProcessor加入的
        if (!processedBeans.contains(ppName)) {
            if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                regularPostProcessors.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
            } else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                registryProcessors.add(ppName);
            } else {
                currentRegistryProcessors.add(ppName);
            }
        }
    }
    sortPostProcessors(regularPostProcessors, beanFactory);
    //a 解析regularPostProcessors----------一个：PropertySourcesPlacehoderConfigurer
    //替换174BeanDefinition中的占位符：
https://blog.csdn.net/jy02268879/article/details/88062673
    invokeBeanFactoryPostProcessors((Collection)regularPostProcessors,
(ConfigurableListableBeanFactory)beanFactory);

    List<BeanFactoryPostProcessor> orderedPostProcessors = new ArrayList();
    Iterator var21 = registryProcessors.iterator();
    while(var21.hasNext()) {
        String postProcessorName = (String)var21.next();
        orderedPostProcessors.add(beanFactory.getBean(postProcessorName,
BeanFactoryPostProcessor.class));
    }
    sortPostProcessors(orderedPostProcessors, beanFactory);
    //b 解析registryProcessors也即orderedPostProcessors----------没有
    invokeBeanFactoryPostProcessors((Collection)orderedPostProcessors,
(ConfigurableListableBeanFactory)beanFactory);

    List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList();
    Iterator var24 = currentRegistryProcessors.iterator();
    while(var24.hasNext()) {
        ppName = (String)var24.next();
        nonOrderedPostProcessors.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
    }
    //c 主要调用ConfigurationBeanFactoryMetadata
```

```
    //遍历 所有 beanDefinition 设置beansFactoryMetadata属性（即设置目标类中 @Bean 注解的方法）
    invokeBeanFactoryPostProcessors((Collection)nonOrderedPostProcessors,
(ConfigurableListableBeanFactory)beanFactory);
    //清楚缓存
    beanFactory.clearMetadataCache();
}
```

1-1.1 invokeBeanDefinitionRegistryPostProcessors()：解析配置类，生成 beanDefinitions

- ConfigurationClassPostProcessor：这个类就是专门解析配置类的类
- 从 main 对应的那个配置类开始解析
- 经过 parser.parse(candidates) 解析 main 类后，得到了更多的 candidates ---------- 重要 a
- 通过 loadBeanDefinitions(configClasses) 加载这些类的定义 ----------重要 b
- 继续使用 parser.parse(candidates) 继续新得到的 candidates，知道不在产生新的 candidates 时，解析配置文件结束

```
//ConfigurationClassPostProcessor：关键类
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
    this.processConfigBeanDefinitions(registry);//正式解析
}
//以下所有的基础类就是：ConfigurationClassPostProcessor，是通过这个类解析的
public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {
    List<BeanDefinitionHolder> configCandidates = new ArrayList();
    //1 获取已经注册的bean名称：7个
    String[] candidateNames = registry.getBeanDefinitionNames();
    String[] var4 = candidateNames;
    int var5 = candidateNames.length;

    //  ----------------------获取一个configCandidates
    for(int var6 = 0; var6 < var5; ++var6) {
        String beanName = var4[var6];
        BeanDefinition beanDef = registry.getBeanDefinition(beanName);
        // 如果BeanDefinition 中的configurationClass 属性为full 或者lite ,则意味着已经处理过
了,直接跳过
        if (!ConfigurationClassUtils.isFullConfigurationClass(beanDef) &&
!ConfigurationClassUtils.isLiteConfigurationClass(beanDef)) {
            //3.1 判断对应bean是否为配置类
            if (ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef,
this.metadataReaderFactory)) {
                // 2 判断对应bean是否为配置类,如果是,则加入到configCandidates(关键！)
                //7个bean中只有一个是配置类：就是main对于的类
                configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));
            }
        } else if (this.logger.isDebugEnabled()) {
            this.logger.debug("Bean definition has already been processed as a
configuration class: " + beanDef);
        }
    }

    //走到这里，configCandidates中只有一个main对应的类
```

```
    //这个if一直到最后
    if (!configCandidates.isEmpty()) {
        //3 对configCandidates 进行 排序,按照@Order 配置的值进行排序
        configCandidates.sort((bd1, bd2) -> {
            int i1 = ConfigurationClassUtils.getOrder(bd1.getBeanDefinition());
            int i2 = ConfigurationClassUtils.getOrder(bd2.getBeanDefinition());
            return Integer.compare(i1, i2);
        });
        // 如果BeanDefinitionRegistry 是SingletonBeanRegistry 子类的话,由于我们当前传入的是
DefaultListableBeanFactory,是SingletonBeanRegistry 的子类。
        //因此会将registry强转为SingletonBeanRegistry
        SingletonBeanRegistry sbr = null;
        if (registry instanceof SingletonBeanRegistry) {
            sbr = (SingletonBeanRegistry)registry;
            if (!this.localBeanNameGeneratorSet) {
                BeanNameGenerator generator =
(BeanNameGenerator)sbr.getSingleton("org.springframework.context.annotation.internalCon
figurationBeanNameGenerator");
                if (generator != null) {
                    this.componentScanBeanNameGenerator = generator;
                    this.importBeanNameGenerator = generator;
                }
            }
        }
        if (this.environment == null) {
            this.environment = new StandardEnvironment();
        }
        // ----------------------实例化ConfigurationClassParser 为了解析 各个配置类
        ConfigurationClassParser parser = new
ConfigurationClassParser(this.metadataReaderFactory, this.problemReporter,
this.environment, this.resourceLoader, this.componentScanBeanNameGenerator, registry);

          // 实例化2个set
         //candidates：放置需要解析的类
        // alreadyParsed：放置已经解析过的类
        Set<BeanDefinitionHolder> candidates = new LinkedHashSet(configCandidates);
        HashSet alreadyParsed = new HashSet(configCandidates.size());

        //------------------------ 进行递归解析
        do {
            //3.2 解析一个main类，并得到其他的配置类
            parser.parse(candidates);//解析
            parser.validate();
            //将解析过的配置类加入到configClasses
            Set<ConfigurationClass> configClasses = new
LinkedHashSet(parser.getConfigurationClasses());
            //将configClasses去重已经处理过的,以防止重复加载
            configClasses.removeAll(alreadyParsed);
            if (this.reader == null) {
                this.reader = new ConfigurationClassBeanDefinitionReader(registry,
this.sourceExtractor, this.resourceLoader, this.environment,
this.importBeanNameGenerator, parser.getImportRegistry());
            }
```

```
            //调用ConfigurationClassBeanDefinitionReader#loadBeanDefinitions 进行加载
            //3.3
            this.reader.loadBeanDefinitions(configClasses);
            //---------------将已经解析过的bean加入alreadyParsed
            alreadyParsed.addAll(configClasses);
            candidates.clear();
            //如果registry中注册的bean的数量 大于 之前获得的数量,则意味着在解析过程中又新加入了很
多,那么就需要对其进行解析
            //candidateNames一开始是7个，随着递归解析会越来越多
            if (registry.getBeanDefinitionCount() > candidateNames.length) {
                //解析后的bean
                String[] newCandidateNames = registry.getBeanDefinitionNames();
                //解析前的bean，和解析后的bean比较后，得出多出来的bean
                Set<String> oldCandidateNames = new
HashSet(Arrays.asList(candidateNames));

                //设置alreadyParsedClasses
                Set<String> alreadyParsedClasses = new HashSet();
                Iterator var12 = alreadyParsed.iterator();
                //将所有 register 中已经有的 beanDefinition 转换成 class，放入
alreadyParsedClasses 中
                while(var12.hasNext()) {
                    ConfigurationClass configurationClass =
(ConfigurationClass)var12.next();

 alreadyParsedClasses.add(configurationClass.getMetadata().getClassName());
                }

                //遍历所有的现有的BeanDefinitionNames
                //如果不是旧的oldCandidateNames且没有解析过，加入candidates
                String[] var23 = newCandidateNames;//目前已经有的 beanDefinitionNames
                int var24 = newCandidateNames.length;
                for(int var14 = 0; var14 < var24; ++var14) {
                    String candidateName = var23[var14];
                    //过滤出解析后得出来的新bean，加入下一轮的candidates
                    if (!oldCandidateNames.contains(candidateName)) {
                        BeanDefinition bd = registry.getBeanDefinition(candidateName);
                        //如果是配置类，且beanDefinitionNames 不包含 configurationClass
                        //alreadyParsedClasses 是目前扫描到的 configurationClass
                        //candidateName 是目前通过 configurationClass 解出来的
beanDefinitionNames
                        //什么情况呢：一个类中通过 @Bean 注入另一个类，并且这个类正好没有被之前解
析到，这样就进去了
                        if
(ConfigurationClassUtils.checkConfigurationClassCandidate(bd,
this.metadataReaderFactory) && !alreadyParsedClasses.contains(bd.getBeanClassName())) {
                            //很罕见
                            candidates.add(new BeanDefinitionHolder(bd,
candidateName));
                        }
                    }
                }
```

```
                candidateNames = newCandidateNames;//设置下一轮解析前的bean为这一轮解析后的
bean
            }
        } while(!candidates.isEmpty());

        //sbr = (SingletonBeanRegistry)registry;，如果sbr中不存在
IMPORT_REGISTRY_BEAN_NAME，注册一个
        if (sbr != null && !sbr.containsSingleton(IMPORT_REGISTRY_BEAN_NAME)) {
            sbr.registerSingleton(IMPORT_REGISTRY_BEAN_NAME,
parser.getImportRegistry());
        }
        //清除缓存
        if (this.metadataReaderFactory instanceof CachingMetadataReaderFactory) {
            ((CachingMetadataReaderFactory)this.metadataReaderFactory).clearCache();
        }
    }
}
```

a parser.parse(candidates)：解析配置类

```
//ConfigurationClassParser
public void parse(configCandidates) {
    //a.1 main对应的类走这里，我的程序没有走其它两个分支
    this.parse(((AnnotatedBeanDefinition)bd).getMetadata(), holder.getBeanName());

    //只有完整走完 a.1 方法才会走到 a.2，在 a.1 内循环解析
    //a.2 处理deferredImportSelect---一般为AutoConfiguration类中的import内类
    //在这之前，configurationClasses里只有简单的几个业务扫描得到的bean
    //这个完了之后，很多autoconfiguration bean 加入了 configurationClasses
    this.deferredImportSelectorHandler.process();
}
```

a.1 解析常规业务类

- 如果类被 Conditional 注解，且 matches 方法不匹配，则不解析这个类，跳过
- 递归解析类中的内部类
- 解析 `@PropertySources`
- 解析 `@ComponentScan/s,@ComponentScan`
- 解析 `@Import`
- 解析 `@ImportResource`
- 解析 `@Bean`
- 解析 类接口类中的 `@Bean`
- 如果类有父类，则返回父类，否则返回 null

```
//ConfigurationClassParser：a 中的 this.parse(...)，最后走到这里
protected void processConfigurationClass(ConfigurationClass configClass) throws
IOException {
    //如果类被Conditional注解(conditionOnClass 等)，且matches方法不匹配，则不解析这个类，跳过，
注意：这里并不处理 ConditionOnBean/MissingBean 注解，这要到全部解析完后，在 loadBedifinition 中
再做判断
    //这个if一直到最后
```

```
    if (!this.conditionEvaluator.shouldSkip(configClass.getMetadata(),
ConfigurationPhase.PARSE_CONFIGURATION)) {
        //将configclass包装成SourceClass
        ConfigurationClassParser.SourceClass sourceClass =
this.asSourceClass(configClass);
        //递归调用进行解析
        do {
            sourceClass = this.doProcessConfigurationClass(configClass, sourceClass);
        } while(sourceClass != null);
        this.configurationClasses.put(configClass, configClass);
    }
}

//ConfigurationClassParser
protected final ConfigurationClassParser.SourceClass
doProcessConfigurationClass(ConfigurationClass configClass,
ConfigurationClassParser.SourceClass sourceClass) throws IOException {
    if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
        //递归解析类中的内部类
        this.processMemberClasses(configClass, sourceClass);
    }

    // 查找该类有没有@PropertySources注解,有的话做处理
    // 查找该参数在类及其父类找中对应的注解
    Iterator var3 =
AnnotationConfigUtils.attributesForRepeatable(sourceClass.getMetadata(),
PropertySources.class, PropertySource.class).iterator();
    AnnotationAttributes importResource;
    while(var3.hasNext()) {
        importResource = (AnnotationAttributes)var3.next();
        //a.1.1 处理@PropertySources注解
        this.processPropertySource(importResource);
    }

    //处理@ComponentScan/s,@ComponentScan
    Set<AnnotationAttributes> componentScans =
AnnotationConfigUtils.attributesForRepeatable(sourceClass.getMetadata(),
ComponentScans.class, ComponentScan.class);
    if (!componentScans.isEmpty() &&
!this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN)) {
        Iterator var13 = componentScans.iterator();

        while(var13.hasNext()) {
            AnnotationAttributes componentScan = (AnnotationAttributes)var13.next();
            //a.1.2 解析basePackages并扫描，这里扫描出了PersonController/personServiceImpl
            Set<BeanDefinitionHolder> scannedBeanDefinitions =
this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());
            Iterator var7 = scannedBeanDefinitions.iterator();
            while(var7.hasNext()) {
                //递归解析扫描出来的类：回到a.1进行解析
                this.parse(bdCand.getBeanClassName(), holder.getBeanName());
```

```
            }
        }
    }

    //a.1.3 处理Import注解，其中getImports()：递归得到了目标类的所有import类
    this.processImports(configClass, sourceClass, this.getImports(sourceClass), true);

    //处理@ImportResource：@ImportResource(“classpath:config.xml”)，加入当前类的
ImportedResource中，留待以后处理
    importResource = AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(),
ImportResource.class);
    if (importResource != null) {
        // 遍历配置的locations,加入到configClass 中的ImportedResource
        String[] resources = importResource.getStringArray("locations");
        Class<? extends BeanDefinitionReader> readerClass =
importResource.getClass("reader");
        for(int var22 = 0; var22 < var21; ++var22) {
            String resource = var19[var22];
            String resolvedResource =
this.environment.resolveRequiredPlaceholders(resource);
            configClass.addImportedResource(resolvedResource, readerClass);
        }
    }

    //处理@Bean注解：获取类内部被@Bean注解修饰的方法，然后添加到配置类的beanMethods属性中，这里并没
有处理 conditional 注解
    //a.1.4
    Set<MethodMetadata> beanMethods = this.retrieveBeanMethodMetadata(sourceClass);
    Iterator var17 = beanMethods.iterator();
    while(var17.hasNext()) {
        MethodMetadata methodMetadata = (MethodMetadata)var17.next();
        configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
    }
    //处理类实现接口的类被@Bean注解的方法
    this.processInterfaces(configClass, sourceClass);

    //如果有父类的话,则返回父类进行进一步的解析
    if (sourceClass.getMetadata().hasSuperClass()) {
        String superclass = sourceClass.getMetadata().getSuperClassName();
        if (superclass != null && !superclass.startsWith("java") &&
!this.knownSuperclasses.containsKey(superclass)) {
            this.knownSuperclasses.put(superclass, configClass);
            return sourceClass.getSuperClass();//return不为null，则不跳过最外层的循环，继续解
析父类！
        }
    }
    return null;//走到这里，才跳过外层循环
}
```

a.1.1 解析 @PropertySources 注解

- 解析 name,encoding,locations,ignoreResourceNotFound
- 解析 factory ,用来根据前面的解析值创建 propertySource

- 对 location 进行 SPEL 表达式的解析
- 将资源加入 environment 中的 propertySource 中

```
private void processPropertySource(AnnotationAttributes propertySource) throws
IOException {
    //解析name
    String name = propertySource.getString("name");
    if (!StringUtils.hasLength(name)) {
        name = null;
    }
    //解析encoding
    String encoding = propertySource.getString("encoding");
    if (!StringUtils.hasLength(encoding)) {
        encoding = null;
    }
    //解析路径
    String[] locations = propertySource.getStringArray("value");
    Assert.isTrue(locations.length > 0, "At least one @PropertySource(value) location
is required");
    //解析ignoreResourceNotFound属性，如果为true，找不到资源不报错
    boolean ignoreResourceNotFound =
propertySource.getBoolean("ignoreResourceNotFound");

    //解析factory,用来根据前面的解析值创建propertySource
    Class<? extends PropertySourceFactory> factoryClass =
propertySource.getClass("factory");
    //如果该值没有配置，默认为PropertySourceFactory则直接实例化DefaultPropertySourceFactory
类，否则开始实例化自定义的类
    PropertySourceFactory factory = factoryClass == PropertySourceFactory.class ?
DEFAULT_PROPERTY_SOURCE_FACTORY :
(PropertySourceFactory)BeanUtils.instantiateClass(factoryClass);

    //遍历路径
    String[] var8 = locations;
    for(int var10 = 0; var10 < var9; ++var10) {
        String location = var8[var10];
        //对location进行SPEL表达式的解析。比如当前的配置环境中有一个属性为app=shareniu，我们配置的
location为${app}最终值为shareniu。通过这里的处理逻辑可以知道location支持多环境的切换以及表达式的配
置
        String resolvedLocation =
this.environment.resolveRequiredPlaceholders(location);
        //加载资源
        Resource resource = this.resourceLoader.getResource(resolvedLocation);
        //将资源加入environment中的propertySource中！！
        this.addPropertySource(factory.createPropertySource(name, new
EncodedResource(resource, encoding)));
    }
}
```

a.1.2 解析 @ComponentScan/s,@ComponentScan 得到路径下的类

- 扫描注解指定的包路径得到包下的类
- 解析类上的注解属性，注入到 `beanDefinition` 中

- 递归从 `a.1` 开始解析扫描到的类

```
public Set<BeanDefinitionHolder> parse(AnnotationAttributes componentScan, final String
declaringClass) {
    //注册扫描器
    //获取nameGenerator：一般是内置的BeanNameGenerator，也可以自定义（在@ComponentScan注解中指
定）
    //设置resourcePattern
    //设置includeFilters
    //设置excludeFilters：默认有两个：TypeExcludeFilter，AutoConfigurationExcludeFilter
    //设置lazyInit
    //扫描包路径：默认当前路径
    //添加一个ExcludeFilter
    //得到扫描路径，开始解析
    return scanner.doScan(StringUtils.toStringArray(basePackages));
}
//ClassPathBeanDefinitionScanner
//basePackages一般默认为当前路径，doScan 会将扫描到的类，先行注册到 beandefinitionNames 中！
protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    for(int var5 = 0; var5 < var4; ++var5) {
        String basePackage = var3[var5];
        //找到basePackage下的类：personController,personServiceImpl，这里会调用
shouldSkip() 来处理 Conditional 注解！在 isCandidateComponent() 方法中调用
isConditionMatch() 方法做过滤
        Set<BeanDefinition> candidates = this.findCandidateComponents(basePackage);
        Iterator var8 = candidates.iterator();
        while(var8.hasNext()) {
            BeanDefinition candidate = (BeanDefinition)var8.next();
            // 解析scope属性：一般为singleton
            ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            //beanNameGenerator生成BeanName
            String beanName = this.beanNameGenerator.generateBeanName(candidate,
this.registry);
            // 普通的BeanDefinition
            if (candidate instanceof AbstractBeanDefinition) {
                //设置candidate的beanDefinition的默认属性：isLazyInit、getAutowireMode等
                this.postProcessBeanDefinition((AbstractBeanDefinition)candidate,
beanName);
            }
             // 注解的BeanDefinition，处理注解@Primary、@DependsOn等Bean注解
            //如果是AnnotatedBeanDefinition类型,设置正真属性：setLazyInit、setPrimary、
setDependsOn等
            if (candidate instanceof AnnotatedBeanDefinition) {

 AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition)cand
idate);
            }
            // 检查当前bean是否已经注册（BeanFactory中是否包含此BeanDefinition）
            if (this.checkCandidate(beanName, candidate)) {
                //如果没有
```

```
                BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder(candidate, beanName);
                //如果当前bean是用于生成代理的bean那么需要进一步处理，取出类的注解信息！！！
                definitionHolder =
AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
                //加入beanDefinitions
                beanDefinitions.add(definitionHolder);
                //将beanDefinition信息注册到相关map中和 beanDefinitionNames 中！！！！！！
                this.registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}
```

a.1.3 解析 @Import 注解

- 通过 getImport() 递归得到的目标类上所有的 @import 中的类的集合
- 遍历以上结合，按照集合中类的类型，分情况讨论
- 如果是 ImportSelector 类：1）如果又是 DeferredImportSelector，留到 `a.2` 处理；2）否则调用 selectImports 得到要解析的类字符串，并递归进行 processImports()
- 如果是 ImportBeanDefinitionRegistrar 类，解析属性，加入到目标类属性中
- 如果是其它，说明 @Import 中的类是一个配置类，递归回到 `a.1` 解析配置类

```
private void
processImports(configClass,currentSourceClass,importCandidates,checkForCircularImports)
{
    //importCandidates这个属性，是通过getImport()递归得到的目标类上所有的 import 类集合
    //如果有循环 import,且checkForCircularImports=true，直接报错

    this.importStack.push(configClass);//将Import注解的目标类放入栈中
    //遍历这些@Import注解内部的属性类集合
    Iterator var5 = importCandidates.iterator();
    while (var5.hasNext()) {
        ConfigurationClassParser.SourceClass candidate =
(ConfigurationClassParser.SourceClass) var5.next();
        Class candidateClass;
        //如果这个类是个ImportSelector接口的实现类
        if (candidate.isAssignable(ImportSelector.class)) {
            candidateClass = candidate.loadClass();
            //实例化这个ImportSelector
            ImportSelector selector = (ImportSelector)
BeanUtils.instantiateClass(candidateClass, ImportSelector.class);
            //如果 selcector 继承了某个 aware 接口，就调用相关方法，设置一些属性
            ParserStrategyUtils.invokeAwareMethods(selector, this.environment,
this.resourceLoader, this.registry);
            //如果这个类也是DeferredImportSelector接口的实现类，留到a.2处理
            //加入ConfigurationClassParser的deferredImportSelectors
            if (selector instanceof DeferredImportSelector) {
                this.deferredImportSelectorHandler.handle(configClass,
(DeferredImportSelector) selector);
```

```
            } else {
                // 否则调用ImportSelector的selectImports方法得到需要Import的类
                // 然后对这些类递归做@Import注解的处理
                String[] importClassNames =
selector.selectImports(currentSourceClass.getMetadata());
                //包装成类
                Collection<ConfigurationClassParser.SourceClass> importSourceClasses =
this.asSourceClasses(importClassNames);
                //递归解析Import注解
                this.processImports(configClass, currentSourceClass,
importSourceClasses, false);
            }
        }
        // 如果这个类是ImportBeanDefinitionRegistrar接口的实现类
        else if (candidate.isAssignable(ImportBeanDefinitionRegistrar.class)) {
            candidateClass = candidate.loadClass();
            ImportBeanDefinitionRegistrar registrar = (ImportBeanDefinitionRegistrar)
BeanUtils.instantiateClass(candidateClass, ImportBeanDefinitionRegistrar.class);
            ParserStrategyUtils.invokeAwareMethods(registrar, this.environment,
this.resourceLoader, this.registry);
            // 设置到配置类的importBeanDefinitionRegistrars属性中
            configClass.addImportBeanDefinitionRegistrar(registrar,
currentSourceClass.getMetadata());
        } else {
            // 其它情况下，如果Import注解中的类就是一个配置类
            //把这个类入队到ConfigurationClassParser的importStack(队列)属性中
            this.importStack.registerImport(currentSourceClass.getMetadata(),
candidate.getMetadata().getClassName());
            // 然后把这个类当成是@Configuration注解修饰的类递归从 a.1 开始解析这个类！！
            this.processConfigurationClass(candidate.asConfigClass(configClass));
        }
    }
}
```

a.1.4 解析 @Bean 注解，得到注解方法，加入目标类属性中

```
private Set<MethodMetadata> retrieveBeanMethodMetadata(sourceClass) {
    AnnotationMetadata original = sourceClass.getMetadata();
    //获取被@Bean注解的方法
    Set<MethodMetadata> beanMethods =
original.getAnnotatedMethods(Bean.class.getName());
    //这个if我的程序没有进去过，应该就行动态代理一样，DynamicJDK只能基于接口，如果没有接口就要使用
asm 技术实现
    //这里的意义可能是通过 asm 技术得到 注解方法及注解信息吧
    if (((Set)beanMethods).size() > 1 && original instanceof
StandardAnnotationMetadata) {
    }
    //直接返回注解方法（包含所有的该方法上的注解信息）
    return (Set)beanMethods;
}
```

a.2 解析 autoconfigution 类

- 创建解析器
- 遍历调用 `processGroupImports()` 获取配置文件

```
public void process() {
    //获取所有的 deferredImports
    List<ConfigurationClassParser.DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
    this.deferredImportSelectors = null;
    try {
        if (deferredImports != null) {
            //创建解析器，注意这里有个 group 概念，如果有 group 执行的就是 group 类中的
selectImport了
            ConfigurationClassParser.DeferredImportSelectorGroupingHandler handler =
ConfigurationClassParser.this.new DeferredImportSelectorGroupingHandler();
            //排序
            deferredImports.sort(ConfigurationClassParser.DEFERRED_IMPORT_COMPARATOR);
            //简单注册
            deferredImports.forEach(handler::register);
            //这里获取autoconfiguration,有group可能就走group类中的selectImport方法了
            //从spring.factories中获取 autoconfiguration
            handler.processGroupImports();
        }
    } finally {
        this.deferredImportSelectors = new ArrayList();
    }
}
public void processGroupImports() {
    Iterator var1 = this.groupings.values().iterator();
    while(var1.hasNext()) {
        ConfigurationClassParser.DeferredImportSelectorGrouping grouping =
(ConfigurationClassParser.DeferredImportSelectorGrouping)var1.next();
        grouping.getImports().forEach((entry) -> {
            //getImports()后，这里有30个entry了
            ConfigurationClass configurationClass =
(ConfigurationClass)this.configurationClasses.get(entry.getMetadata());

            //这里继续解析在spring.factoies中的得到的配置类！！！
            ConfigurationClassParser.this.processImports(configurationClass,
ConfigurationClassParser.this.asSourceClass(configurationClass),
ConfigurationClassParser.this.asSourceClasses(entry.getImportClassName()), false);
        });
    }
}
//在AutoConfigurationImportSelector中的这个方法中，获取了所有可能的自动配置类！！
protected AutoConfigurationImportSelector.AutoConfigurationEntry
getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata,
AnnotationMetadata annotationMetadata) {
{
     AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
     //在spring.factories中加载所有的autoconfiguration类！！
     List<String> configurations = this.getCandidateConfigurations(annotationMetadata,
attributes);
     configurations = this.removeDuplicates(configurations);//去重
```

```
        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);//过滤
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = this.filter(configurations, autoConfigurationMetadata);//这个过滤了
绝大多数的配置类
        this.fireAutoConfigurationImportEvents(configurations, exclusions);
        return new AutoConfigurationImportSelector.AutoConfigurationEntry(configurations,
exclusions);
    }
```

b loadBeanDefinitions(configClasses)：通过 class 加载 beanDefinition

- 如果是 @Import 注解中的类，调用 registerBeanDefinition() 方法
- 解析此类中的 @Bean 注解的方法，先加载这些方法
- 加载 ImportedResources，如 `xml`，解析其中的 Bean，放入 `beanDefinition` 中
- 加载 Registrars 有关的：调用其中的 registerBeanDefinitions() 方法！

```
//遍历加载所有的 configClass
public void loadBeanDefinitions(Set<ConfigurationClass> configurationModel) {
    ConfigurationClassBeanDefinitionReader.TrackedConditionEvaluator
trackedConditionEvaluator = new
ConfigurationClassBeanDefinitionReader.TrackedConditionEvaluator();
    Iterator var3 = configurationModel.iterator();

    while(var3.hasNext()) {
        ConfigurationClass configClass = (ConfigurationClass)var3.next();
        this.loadBeanDefinitionsForConfigurationClass(configClass,
trackedConditionEvaluator);
    }
}
//通过 configClass 加载beanDefinition
private void loadBeanDefinitionsForConfigurationClass(ConfigurationClass configClass,
trackedConditionEvaluator) {
    //这里才处理了 ConditionOnBean/MissingBean 注解，并且重复处理了 ConditionOnClass 注解
    if (trackedConditionEvaluator.shouldSkip(configClass)) {
        //如果应该条件，就需要删除 之前加载 configClass 操作留下的信息了
        String beanName = configClass.getBeanName();
        if (StringUtils.hasLength(beanName) &&
this.registry.containsBeanDefinition(beanName)) {
            this.registry.removeBeanDefinition(beanName);
        }


 this.importRegistry.removeImportingClass(configClass.getMetadata().getClassName());
    } else {
        //如果是被 scan 到的，就不会进去注册 beanDefinition 了，因为 scan 的时候已经注册了！！！
        if (configClass.isImported()) {
            this.registerBeanDefinitionForImportedConfigurationClass(configClass);
        }
        //解析此类中的 @Bean 注解的方法
        Iterator var3 = configClass.getBeanMethods().iterator();
```

```
        while(var3.hasNext()) {
            BeanMethod beanMethod = (BeanMethod)var3.next();
            //如果有@Bean方法，先加载它
            this.loadBeanDefinitionsForBeanMethod(beanMethod);
        }

        //加载ImportedResources，如xml，解析其中的Bean，放入beanDefinition中

 this.loadBeanDefinitionsFromImportedResources(configClass.getImportedResources());
        //加载Registrars有关的：MapperScannerRegistrar：处理@MapperScan、
AutoConfiguredMapperScanner：处理@Mapper、
DataSourceInitializationConfiguration.Registrar：注册
dataSourceInitializerPostProcessor，运行 sql 文件、AspectJAutoProxyRegistrar：处理
@EnableAspectJAutoProxy等等
        //调用其中的 registerBeanDefinitions() 方法！

 this.loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars()
);
    }
}
```

1-1.2 PropertySourcesPlacehoderConfigurer：替换占位符

- 在加载了所有的 `beanDefinitions` 后，替换其中的占位符
- spring的启动过程03.1-占位符替换过程-xml配置的参数

1-2 finishBeanFactoryInitialization()：加载 bean

- 见 Springboot 源码分析—— bean 初始化流程、beanPostProcessor 用法、循环依赖

2 相关链接

spring boot实战(第十篇)Spring boot Bean加载源码分析 spring boot 源码解析11-ConfigurationClassPostProcessor类加载解析 Spring如何加载和解析@Configuration标签 spring容器加载分析三Configuration类解析 Spring Boot启动过程（二） spring的启动过程03.1-占位符替换过程-xml配置的参数 本文 refreshContext() 解析的原始笔记(较详细)