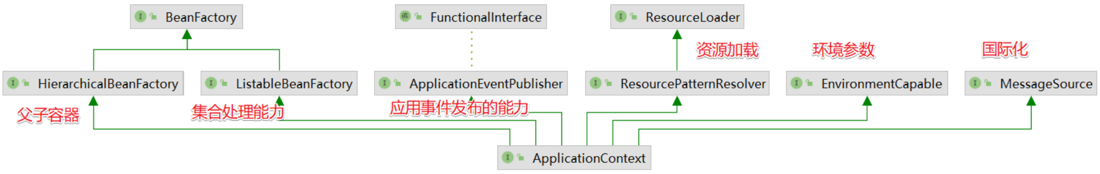
Spring源码篇-ApplicationContext

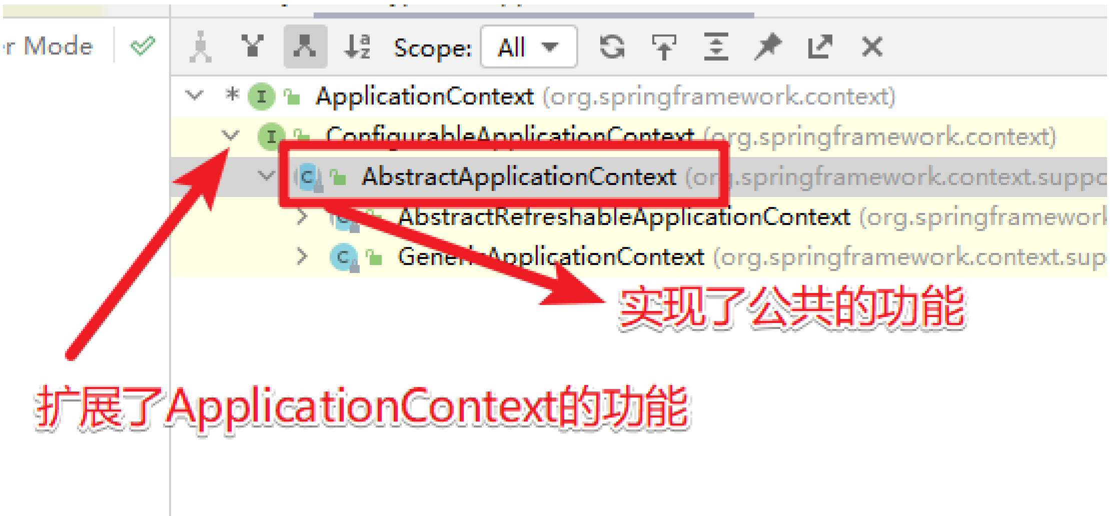
前面通过手写IoC，DI、AOP和Bean的配置。到最后ApplicationContext的门面处理，对于Spring相关的核心概念应该会比较清楚了。接下来我们就看看在Spring源码中，对于的核心组件是如何实现的。

一、ApplicationContext

ApplicationContext到底是什么？字面含义是应用的上下文。这块我们需要看看ApplicationContext的具体的结构。



通过ApplicationContext实现的相关接口来分析，ApplicationContext接口在具备BeanFactory的功能的基础上还扩展了 `应用事件发布`,`资源加载`,`环境参数` 和 `国际化` 的能力。然后我们来看看ApplicationContext接口的实现类的情况。

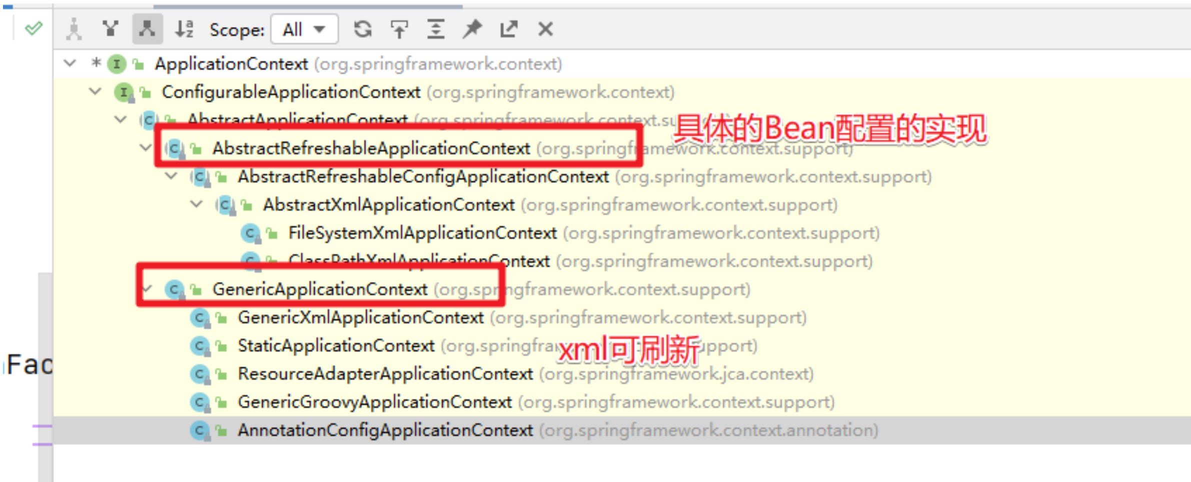


```
r Mode   Scope: All
∨ * ApplicationContext (org.springframework.context)
  ∨ ConfigurableApplicationContext (org.springframework.context)
    ∨ AbstractApplicationContext (org.springframework.context.suppo
      > AbstractRefreshableApplicationContext (org.springframework
      > GenericApplicationContext (org.springframework.context.sup
```

实现了公共的功能

扩展了ApplicationContext的功能

在ApplicationContext的实现类中有两个比较重要的分支 `AbstractRefreshableApplicationContext` 和 `GenericApplicationContext`.



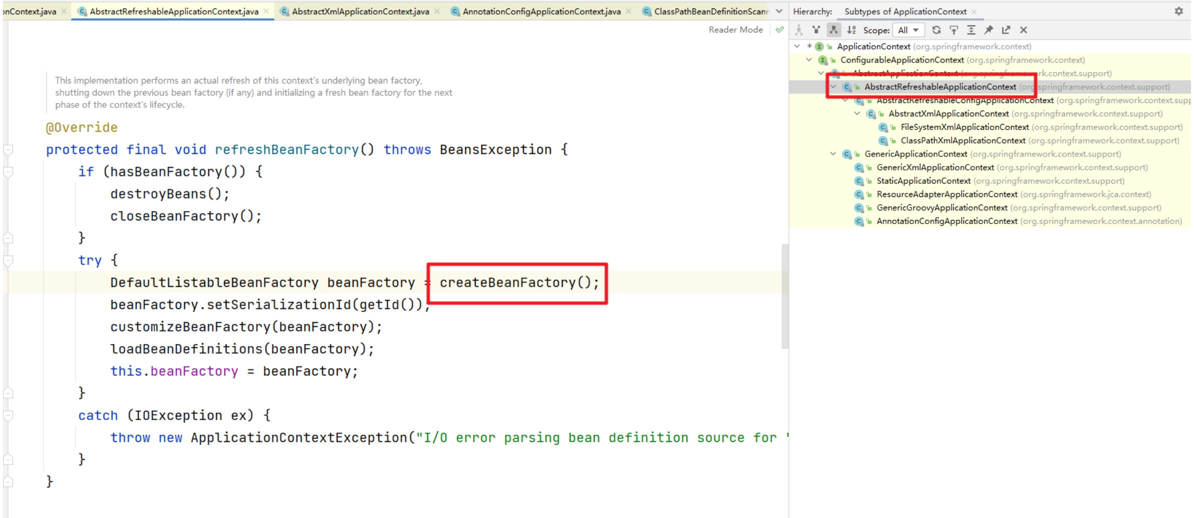
```
Scope: All
∨ * ApplicationContext (org.springframework.context)
  ∨ ConfigurableApplicationContext (org.springframework.context)
    ∨ AbstractApplicationContext (org.springframework.context.su
      ∨ AbstractRefreshableApplicationContext (org.springframework.context.support)
        ∨ AbstractRefreshableConfigApplicationContext (org.springframework.context.support)
          ∨ AbstractXmlApplicationContext (org.springframework.context.support)
              FileSystemXmlApplicationContext (org.springframework.context.support)
              ClassPathXmlApplicationContext (org.springframework.context.support)
      ∨ GenericApplicationContext (org.springframework.context.support)
          GenericXmlApplicationContext (org.springframework.context.support)
          StaticApplicationContext (org.springfra                  upport)
          ResourceAdapterApplicationContext (org.springframework.jca.context)
          GenericGroovyApplicationContext (org.springframework.context.support)
          AnnotationConfigApplicationContext (org.springframework.context.annotation)
```

具体的Bean配置的实现

xml可刷新

二、BeanFactory

上面分析了 `ApplicationContext` 接口的结构。然后我们来看看 `BeanFactory` 在ApplicationContext中具体的实现是怎么样的

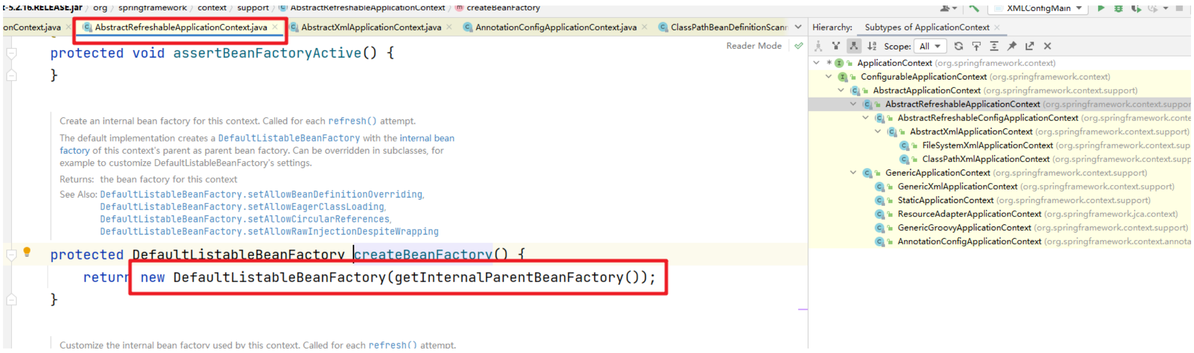


onContext.java
AbstractRefreshableApplicationContext.java
AbstractXmlApplicationContext.java
AnnotationConfigApplicationContext.java
ClassPathBeanDefinitionScann
Reader Mode
Hierarchy:
Subtypes of ApplicationContext
Scope:
All

This implementation performs an actual refresh of this context's underlying bean factory, shutting down the previous bean factory (if any) and initializing a fresh bean factory for the next phase of the context's lifecycle.

```java
@Override
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        loadBeanDefinitions(beanFactory);
        this.beanFactory = beanFactory;
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition source for '
    }
}
```

```
ApplicationContext (org.springframework.context)
  ConfigurableApplicationContext (org.springframework.context)
    AbstractApplicationContext (org.springframework.context.support)
      AbstractRefreshableApplicationContext (org.springframework.context.support)
        AbstractRefreshableConfigApplicationContext (org.springframework.context.supp
          AbstractXmlApplicationContext (org.springframework.context.support)
            FileSystemXmlApplicationContext (org.springframework.context.support)
            ClassPathXmlApplicationContext (org.springframework.context.support)
      GenericApplicationContext (org.springframework.context.support)
        GenericXmlApplicationContext (org.springframework.context.support)
        StaticApplicationContext (org.springframework.context.support)
        ResourceAdapterApplicationContext (org.springframework.jca.context)
        GenericGroovyApplicationContext (org.springframework.context.support)
        AnnotationConfigApplicationContext (org.springframework.context.annotation)
```



t-5.2.16.RELEASE.jar
org
springframework
context
support
AbstractRefreshableApplicationContext
createBeanFactory
XMLConfigMain
onContext.java
AbstractRefreshableApplicationContext.java
AbstractXmlApplicationContext.java
AnnotationConfigApplicationContext.java
ClassPathBeanDefinitionScann
Reader Mode
Hierarchy:
Subtypes of ApplicationContext
Scope:
All

```java
protected void assertBeanFactoryActive() {
}
```

Create an internal bean factory for this context. Called for each `refresh()` attempt.

The default implementation creates a `DefaultListableBeanFactory` with the internal bean factory of this context's parent as parent bean factory. Can be overridden in subclasses, for example to customize DefaultListableBeanFactory's settings.

Returns: the bean factory for this context

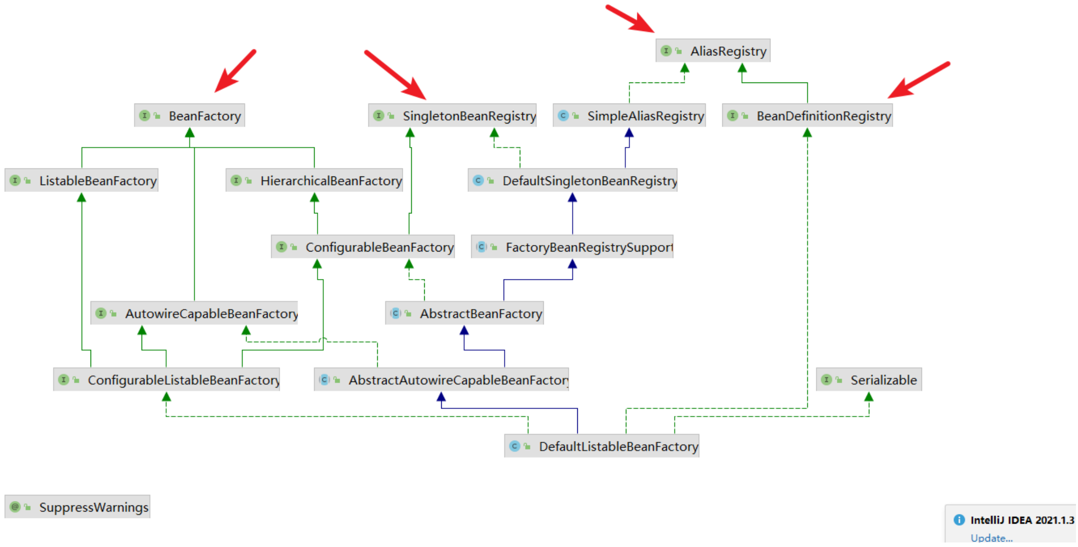
See Also: `DefaultListableBeanFactory.setAllowBeanDefinitionOverriding`, `DefaultListableBeanFactory.setAllowEagerClassLoading`, `DefaultListableBeanFactory.setAllowCircularReferences`, `DefaultListableBeanFactory.setAllowRawInjectionDespiteWrapping`

```java
protected DefaultListableBeanFactory createBeanFactory() {
    return new DefaultListableBeanFactory(getInternalParentBeanFactory());
}
```

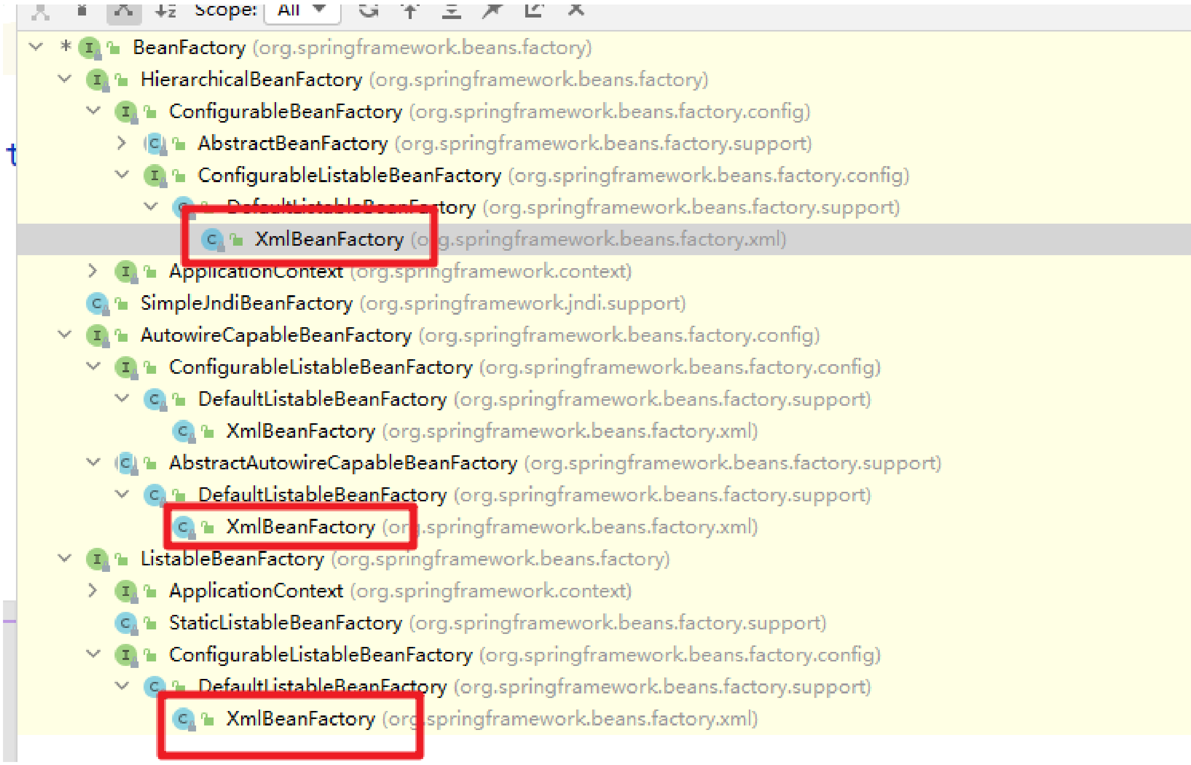
Customize the internal bean factory used by this context. Called for each `refresh()` attempt.

```
ApplicationContext (org.springframework.context)
  ConfigurableApplicationContext (org.springframework.context)
    AbstractApplicationContext (org.springframework.context.support)
      AbstractRefreshableApplicationContext (org.springframework.context.suppor
        AbstractRefreshableConfigApplicationContext (org.springframework.conte
          AbstractXmlApplicationContext (org.springframework.context.support)
            FileSystemXmlApplicationContext (org.springframework.context.sup
            ClassPathXmlApplicationContext (org.springframework.context.supp
      GenericApplicationContext (org.springframework.context.support)
        GenericXmlApplicationContext (org.springframework.context.support)
        StaticApplicationContext (org.springframework.context.support)
        ResourceAdapterApplicationContext (org.springframework.jca.context)
        GenericGroovyApplicationContext (org.springframework.context.support)
        AnnotationConfigApplicationContext (org.springframework.context.annota
```

可以看到具体的实现是 DefaultListableBeanFactory .然后我们来看看他的体系结构



BeanFactory的继承体系



```
BeanFactory (org.springframework.beans.factory)
  HierarchicalBeanFactory (org.springframework.beans.factory)
    ConfigurableBeanFactory (org.springframework.beans.factory.config)
      AbstractBeanFactory (org.springframework.beans.factory.support)
      ConfigurableListableBeanFactory (org.springframework.beans.factory.config)
        DefaultListableBeanFactory (org.springframework.beans.factory.support)
          XmlBeanFactory (org.springframework.beans.factory.xml)
    ApplicationContext (org.springframework.context)
  SimpleJndiBeanFactory (org.springframework.jndi.support)
  AutowireCapableBeanFactory (org.springframework.beans.factory.config)
    ConfigurableListableBeanFactory (org.springframework.beans.factory.config)
      DefaultListableBeanFactory (org.springframework.beans.factory.support)
        XmlBeanFactory (org.springframework.beans.factory.xml)
    AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
      DefaultListableBeanFactory (org.springframework.beans.factory.support)
        XmlBeanFactory (org.springframework.beans.factory.xml)
  ListableBeanFactory (org.springframework.beans.factory)
    ApplicationContext (org.springframework.context)
    StaticListableBeanFactory (org.springframework.beans.factory.support)
    ConfigurableListableBeanFactory (org.springframework.beans.factory.config)
      DefaultListableBeanFactory (org.springframework.beans.factory.support)
        XmlBeanFactory (org.springframework.beans.factory.xml)
```

三、BeanDefinition

然后我们来了解下ApplicationContext是如何来加载Bean定义的。具体代码我们需要分为XML配置文件和基于注解的两种方式来看。

1.基于XML方式

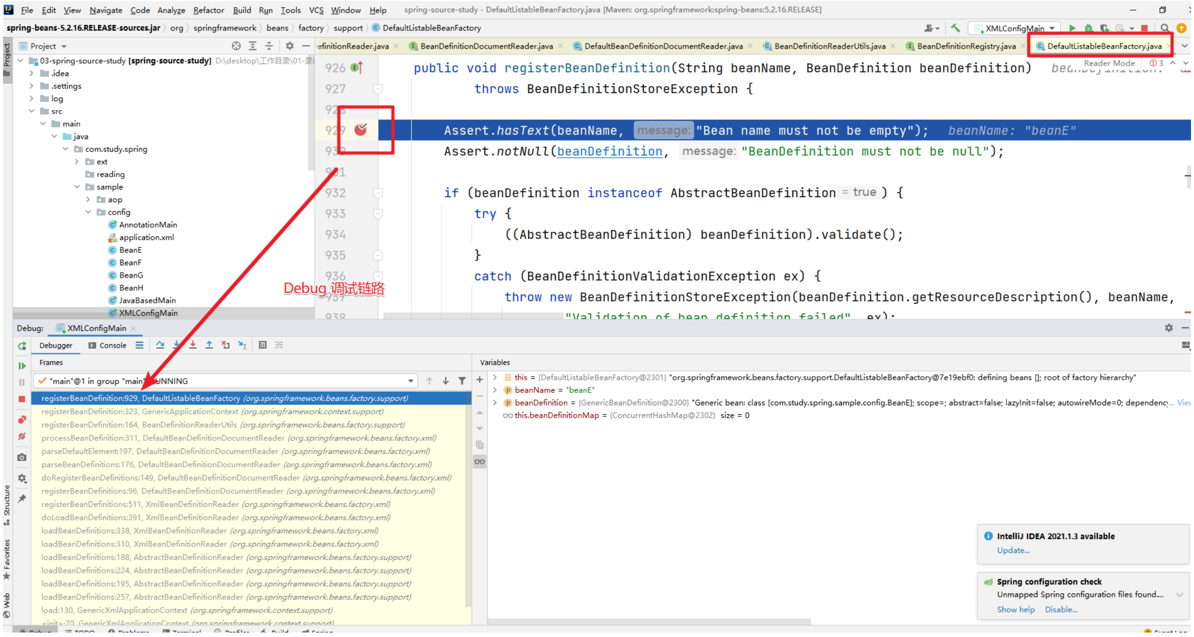
我们先定义对应的application.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="beanE" class="com.study.spring.sample.config.BeanE" />

    <bean id="beanF" class="com.study.spring.sample.config.BeanF" ></bean>

    <context:annotation-config/>

    <context:component-scan base-package="com.study.spring.sample.config" >
</context:component-scan>

</beans>
```

然后我们的测试类代码

```
public class XMLConfigMain {

    public static void main(String[] args) {
        ApplicationContext context = new GenericXmlApplicationContext(
                "classpath:com/study/spring/sample/config/application.xml");
        BeanF bf = context.getBean(BeanF.class);
        bf.do1();
    }
}
```

处理的过程 解析XML --> BeanDefinition --> BeanDefinitionRegistry --> BeanFactory



spring-source-study - DefaultListableBeanFactory.java [Maven: org.springframework:spring-beans:5.2.16.RELEASE]

```
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
        throws BeanDefinitionStoreException {

    Assert.hasText(beanName, message: "Bean name must not be empty");   beanName: "beanE"
    Assert.notNull(beanDefinition, message: "BeanDefinition must not be null");

    if (beanDefinition instanceof AbstractBeanDefinition = true ) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                    "Validation of bean definition failed", ex);
```

Debug 调试链路

```
"main"@1 in group "main": RUNNING
registerBeanDefinition:929, DefaultListableBeanFactory (org.springframework.beans.factory.support)
registerBeanDefinition:323, GenericApplicationContext (org.springframework.context.support)
registerBeanDefinition:164, BeanDefinitionReaderUtils (org.springframework.beans.factory.support)
processBeanDefinition:311, DefaultBeanDefinitionDocumentReader (org.springframework.beans.factory.xml)
parseDefaultElement:197, DefaultBeanDefinitionDocumentReader (org.springframework.beans.factory.xml)
parseBeanDefinitions:176, DefaultBeanDefinitionDocumentReader (org.springframework.beans.factory.xml)
doRegisterBeanDefinitions:149, DefaultBeanDefinitionDocumentReader (org.springframework.beans.factory.xml)
registerBeanDefinitions:96, DefaultBeanDefinitionDocumentReader (org.springframework.beans.factory.xml)
registerBeanDefinitions:511, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
doLoadBeanDefinitions:391, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
loadBeanDefinitions:338, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
loadBeanDefinitions:310, XmlBeanDefinitionReader (org.springframework.beans.factory.xml)
loadBeanDefinitions:188, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
loadBeanDefinitions:224, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
loadBeanDefinitions:195, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
loadBeanDefinitions:257, AbstractBeanDefinitionReader (org.springframework.beans.factory.support)
load:130, GenericXmlApplicationContext (org.springframework.context.support)
<init>:70, GenericXmlApplicationContext (org.springframework.context.support)
```

```
this = {DefaultListableBeanFactory@2301} "org.springframework.beans.factory.support.DefaultListableBeanFactory@7e19ebf0: defining beans []; root of factory hierarchy"
beanName = "beanE"
beanDefinition = {GenericBeanDefinition@2300} "Generic bean: class [com.study.spring.sample.config.BeanE]; scope=; abstract=false; lazyInit=false; autowireMode=0; dependenc... View
this.beanDefinitionMap = {ConcurrentHashMap@2302} size = 0
```

IntelliJ IDEA 2021.1.3 available
Update...

Spring configuration check
Unmapped Spring configuration files found...
Show help Disable...

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
spring-beans-5.2.16.RELEASE-sources.jar org springframework beans factory support DefaultListableBeanFactory
XMLConfigMain
BeanDefinitionDocumentReader.java
DefaultBeanDefinitionDocumentReader.java
BeanDefinitionReaderUtils.java
BeanDefinitionRegistry.java
DefaultListableBeanFactory.java
Reader Mode
Project
03-spring-source-study [spring-source-study]
.idea
.settings
log
src
main
java
com.study.spring
ext
reading
sample
aop
config
AnnotationMain
application.xml
BeanE
BeanF
BeanG
BeanH
JavaBasedMain
XMLConfigMain
Debug: XMLConfigMain
Debugger
Console
Frames
Variables
Structure
Favorites
Web

2.基于注解方式

然后来看看基于注解方式的使用的情况。首先是我们的配置类

```
@Configuration
@ComponentScan("com.study.spring.sample.config")
public class JavaBasedMain {

    @Bean
    public BeanH getBeanH() {
        return new BeanH();
    }

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(JavaBasedMain.class);

        BeanH bh = context.getBean(BeanH.class);
        bh.doH();
    }
}
```

然后是我们的测试类

```java
public class AnnotationMain {

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext("com.study.spring.sample.config");

        BeanG bg = context.getBean(BeanG.class);
        bg.dog();
    }
}
```

注解使用有两种方法：

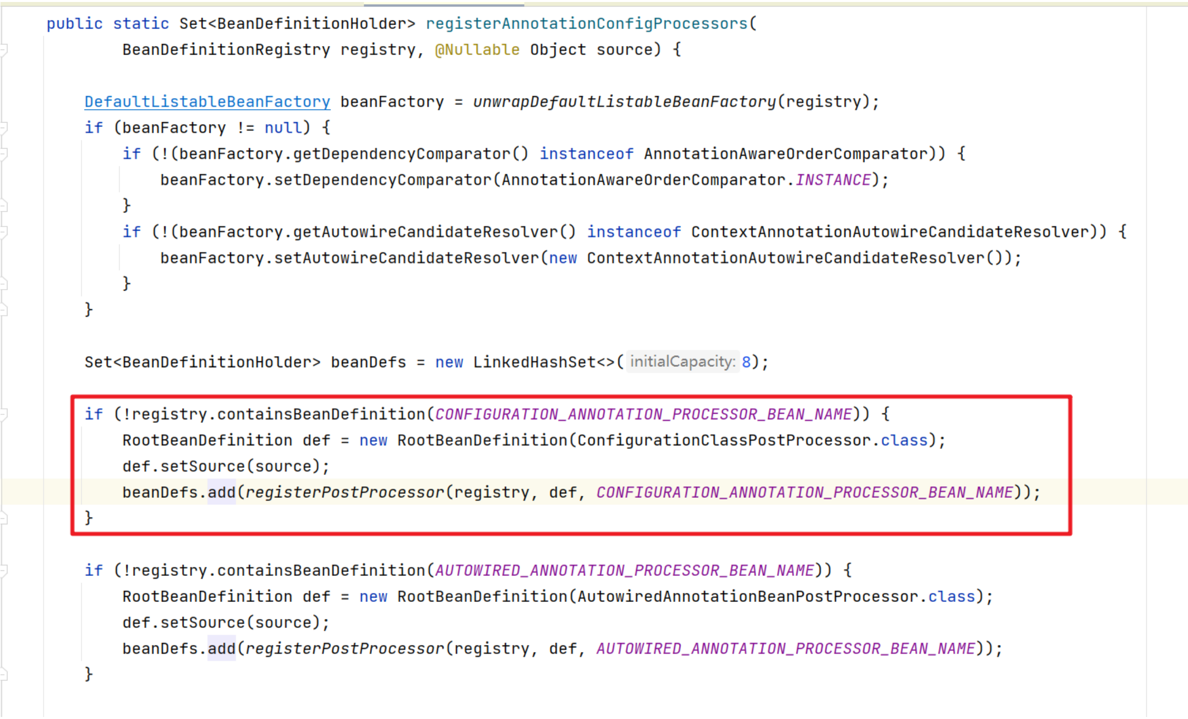
1. 配置扫描路径
2. 配置@Configuration的注解类

2.1 this构造方法

在this的构造方法中会完成相关的配置处理。

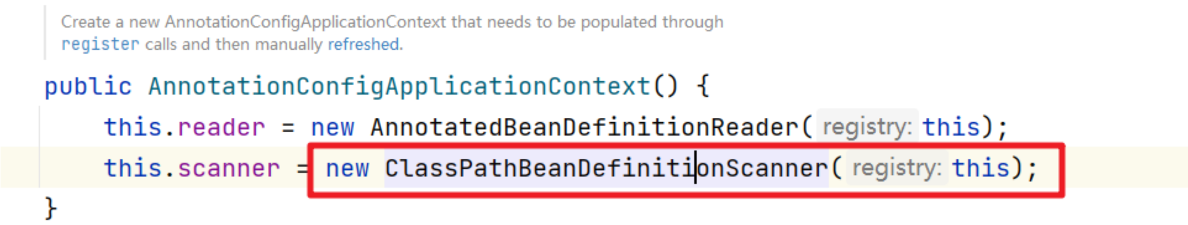
```java
    public AnnotationConfigApplicationContext() {
        this.reader = new AnnotatedBeanDefinitionReader(this);
        this.scanner = new ClassPathBeanDefinitionScanner(this);
    }
```

首先是AnnotatedBeanDefinitionReader(this)方法。会完成核心的ConfigurationClassPostProcessor的注入。ConfigurationClassPostProcessor 会完成@Configuration相关的注解的解析



```java
public static Set<BeanDefinitionHolder> registerAnnotationConfigProcessors(
        BeanDefinitionRegistry registry, @Nullable Object source) {

    DefaultListableBeanFactory beanFactory = unwrapDefaultListableBeanFactory(registry);
    if (beanFactory != null) {
        if (!(beanFactory.getDependencyComparator() instanceof AnnotationAwareOrderComparator)) {
            beanFactory.setDependencyComparator(AnnotationAwareOrderComparator.INSTANCE);
        }
        if (!(beanFactory.getAutowireCandidateResolver() instanceof ContextAnnotationAutowireCandidateResolver)) {
            beanFactory.setAutowireCandidateResolver(new ContextAnnotationAutowireCandidateResolver());
        }
    }

    Set<BeanDefinitionHolder> beanDefs = new LinkedHashSet<>( initialCapacity: 8);

    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));
    }

    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));
    }
```

this.scanner其实就是创建了一个对应的扫描器

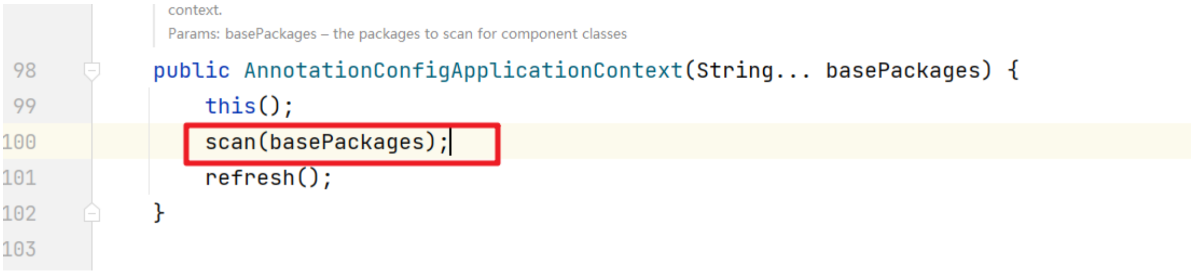


```
Create a new AnnotationConfigApplicationContext that needs to be populated through
register calls and then manually refreshed.
public AnnotationConfigApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader( registry: this);
    this.scanner = new ClassPathBeanDefinitionScanner( registry: this);
}
```

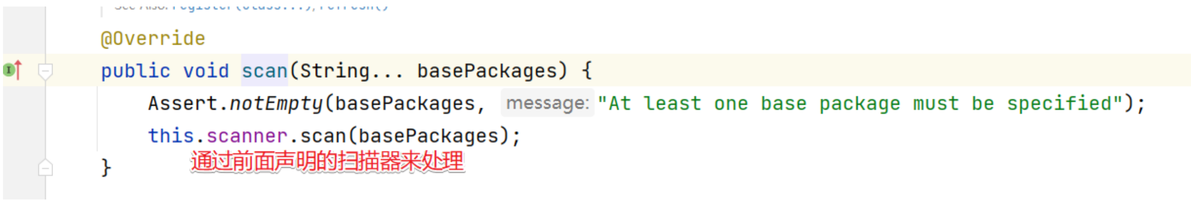
创建了一个扫描器

2.2 扫描实现

扫描就需要进入到scan方法中。

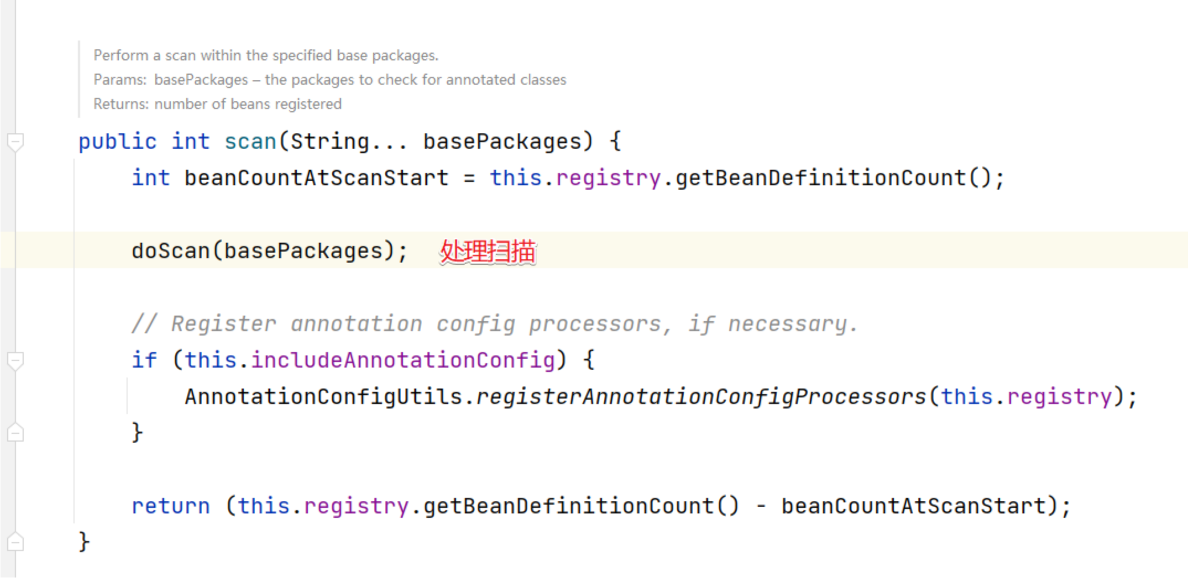


```
context.
Params: basePackages – the packages to scan for component classes
public AnnotationConfigApplicationContext(String... basePackages) {
    this();
    scan(basePackages);
    refresh();
}
```



```
See Also: register(Class...), refresh()
@Override
public void scan(String... basePackages) {
    Assert.notEmpty(basePackages, message: "At least one base package must be specified");
    this.scanner.scan(basePackages);
}
```

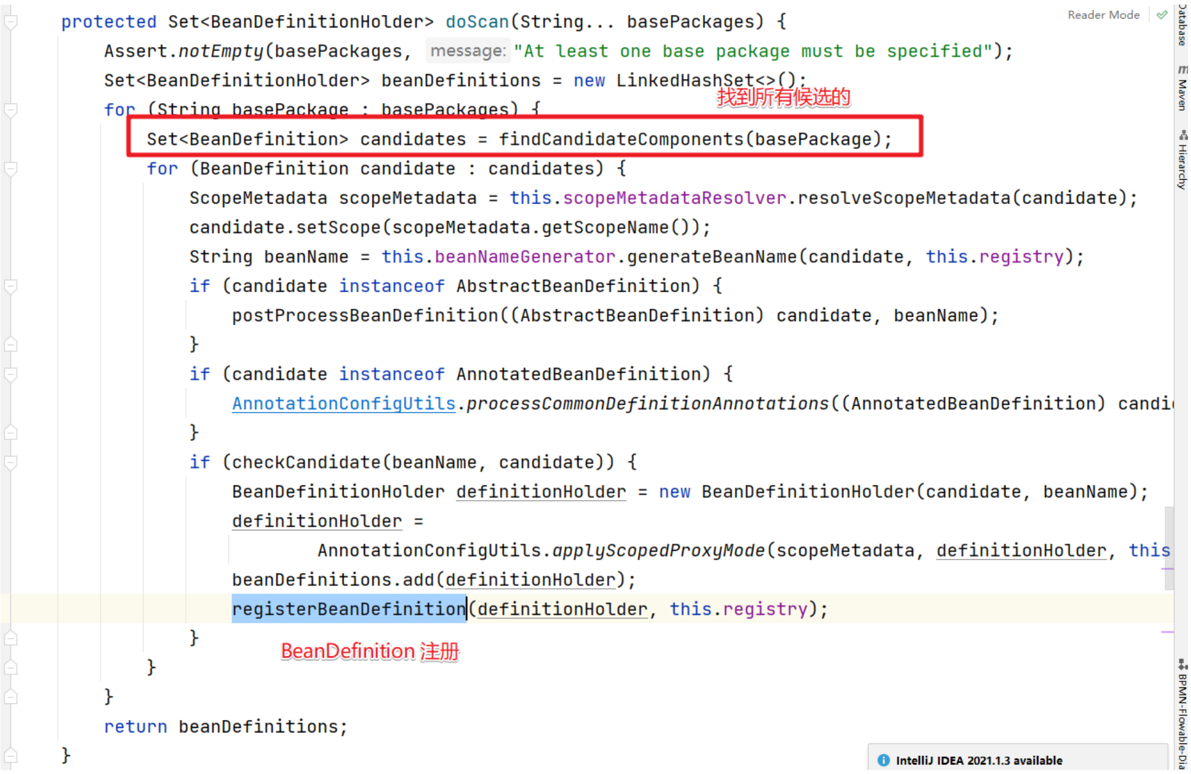
通过前面声明的扫描器来处理



```
Perform a scan within the specified base packages.
Params: basePackages – the packages to check for annotated classes
Returns: number of beans registered
public int scan(String... basePackages) {
    int beanCountAtScanStart = this.registry.getBeanDefinitionCount();

    doScan(basePackages);

    // Register annotation config processors, if necessary.
    if (this.includeAnnotationConfig) {
        AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
    }

    return (this.registry.getBeanDefinitionCount() - beanCountAtScanStart);
}
```

处理扫描

完成相关的注册



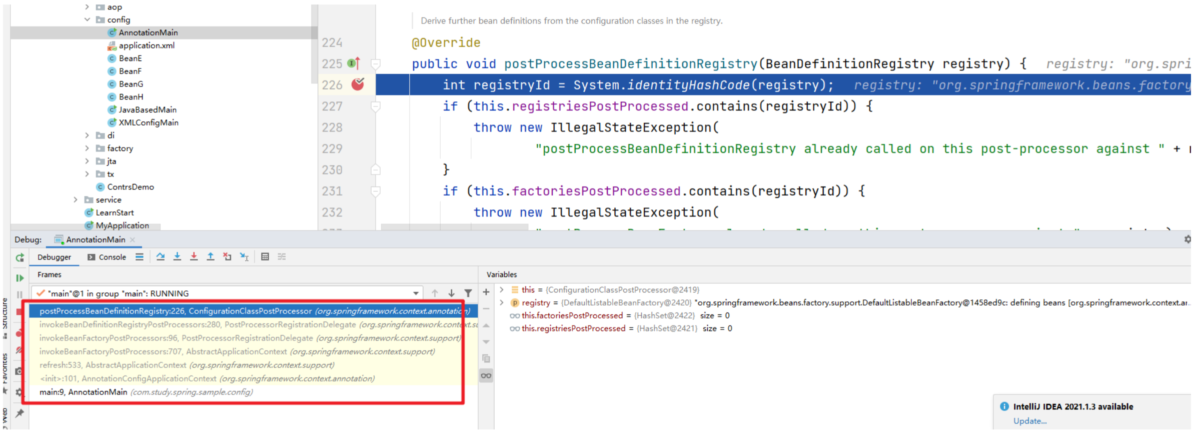
```
protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages,  message: "At least one base package must be specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
    for (String basePackage : basePackages) {
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
                postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
                AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candi
            }
            if (checkCandidate(beanName, candidate)) {
                BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);
                definitionHolder =
                        AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this
                beanDefinitions.add(definitionHolder);
                registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}
```

找到所有候选的

BeanDefinition 注册

2.3 @Configuration

@Configuration的解析其实是在refresh方法中来实现的。



Derive further bean definitions from the configuration classes in the registry.

```
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {   registry: "org.spri
    int registryId = System.identityHashCode(registry);   registry: "org.springframework.beans.factory
    if (this.registriesPostProcessed.contains(registryId)) {
        throw new IllegalStateException(
                "postProcessBeanDefinitionRegistry already called on this post-processor against " + r
    }
    if (this.factoriesPostProcessed.contains(registryId)) {
        throw new IllegalStateException(
```

```
"main"@1 in group "main": RUNNING
postProcessBeanDefinitionRegistry:226, ConfigurationClassPostProcessor (org.springframework.context.annotation)
invokeBeanDefinitionRegistryPostProcessors:280, PostProcessorRegistrationDelegate (org.springframework.context.su
invokeBeanFactoryPostProcessors:96, PostProcessorRegistrationDelegate (org.springframework.context.support)
invokeBeanFactoryPostProcessors:707, AbstractApplicationContext (org.springframework.context.support)
refresh:533, AbstractApplicationContext (org.springframework.context.support)
<init>:101, AnnotationConfigApplicationContext (org.springframework.context.annotation)
main:9, AnnotationMain (com.study.spring.sample.config)
```

```
this = {ConfigurationClassPostProcessor@2419}
registry = {DefaultListableBeanFactory@2420} "org.springframework.beans.factory.support.DefaultListableBeanFactory@1458ed9c: defining beans [org.springframework.context.ar
this.factoriesPostProcessed = {HashSet@2422}  size = 0
this.registriesPostProcessed = {HashSet@2421}  size = 0
```

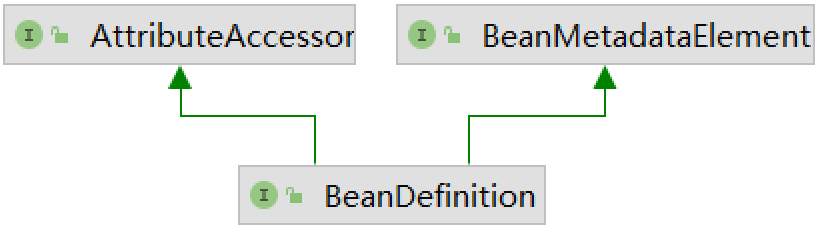
3.小结

通过上面的分析其实我们已经对Bean定义的扫描，解析和注册过程有了一定的了解。归纳为：

1. reader解析XML，完成xml方法配置的bean定义
2. scanner扫描指定包下的类，找出带有@Component注解的类，注册成Bean定义
3. 通过ConfigurationClassPostProcessor对带有@Configuration注解的类进行处理，解析它上面的注解，以及类中带有@Bean 注解，加入这些的Bean的定义。

4.BeanDefinition

;然后我们来看看BeanDefinition的继承结构



继承属性访问器和元数据接口，增加了Bean定义操作，实现了数据和操作解耦。属性访问器和元数据接口接着往下看。

4.1 BeanMetadataElement

BeanMetadataElement提供了获取数据源的方式，也就是可以指导Bean是来自哪个类。

```
public interface BeanMetadataElement {

    /**
     * Return the configuration source {@code Object} for this metadata element
     * (may be {@code null}).
     */
    @Nullable
    default Object getSource() {
        return null;
    }

}
```

4.2 BeanMetadataAttribute元数据属性

实现了元数据接口，增加了属性的名字和值。。

```

public class BeanMetadataAttribute implements BeanMetadataElement {

    private final String name;

    @Nullable
    private final Object value;

    @Nullable
    private Object source;

}
```

4.3 AttributeAccessor属性访问器

AttributeAccessor用来给Bean定义了增删改查属性的功能

```
public interface AttributeAccessor {

    /**
     * Set the attribute defined by {@code name} to the supplied {@code value}.
     * If {@code value} is {@code null}, the attribute is {@link #removeAttribute removed}.
     * <p>In general, users should take care to prevent overlaps with other
     * metadata attributes by using fully-qualified names, perhaps using
     * class or package names as prefix.
     * @param name the unique attribute key
     * @param value the attribute value to be attached
     */
    void setAttribute(String name, @Nullable Object value);

    /**
     * Get the value of the attribute identified by {@code name}.
     * Return {@code null} if the attribute doesn't exist.
     * @param name the unique attribute key
     * @return the current value of the attribute, if any
     */
    @Nullable
    Object getAttribute(String name);

    /**
     * Remove the attribute identified by {@code name} and return its value.
     * Return {@code null} if no attribute under {@code name} is found.
     * @param name the unique attribute key
     * @return the last value of the attribute, if any
     */
    @Nullable
    Object removeAttribute(String name);

    /**
     * Return {@code true} if the attribute identified by {@code name} exists.
     * Otherwise return {@code false}.
     * @param name the unique attribute key
     */
    boolean hasAttribute(String name);

    /**
     * Return the names of all attributes.
     */
    String[] attributeNames();

}
```

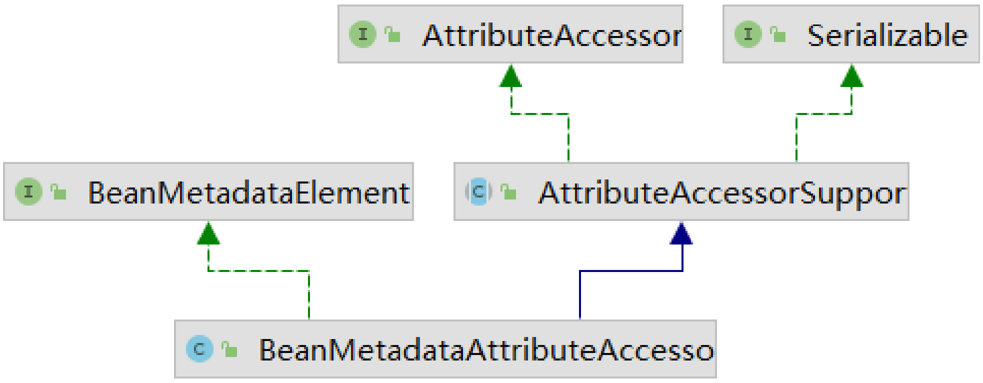
4.4 AttributeAccessorSupport属性访问抽象实现类

内部定义了1个map来存放属性。

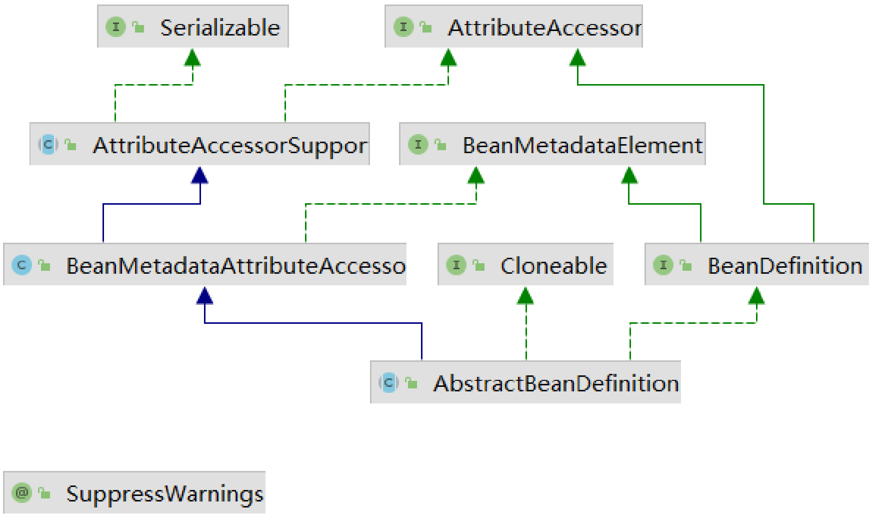
```
public abstract class AttributeAccessorSupport implements AttributeAccessor, Serializable {

    /** Map with String keys and Object values. */
    private final Map<String, Object> attributes = new LinkedHashMap<>();


    @Override
    public void setAttribute(String name, @Nullable Object value) {
        Assert.notNull(name, "Name must not be null");
        if (value != null) {
            this.attributes.put(name, value);
        }
        else {
            removeAttribute(name);
        }
    }
    //  ......
}
```

4.5 BeanMetadataAttributeAccessor元数据属性访问器

继承AttributeAccessorSupport具备属性访问功能，实现BeanMetadataElement具备获取元数据功能。 **AbstractBeanDefinition就继承于它，使得同时具有属性访问和元数据访问的功能 。**



结合AbstractBeanDefinition.来看看他们的类图结构



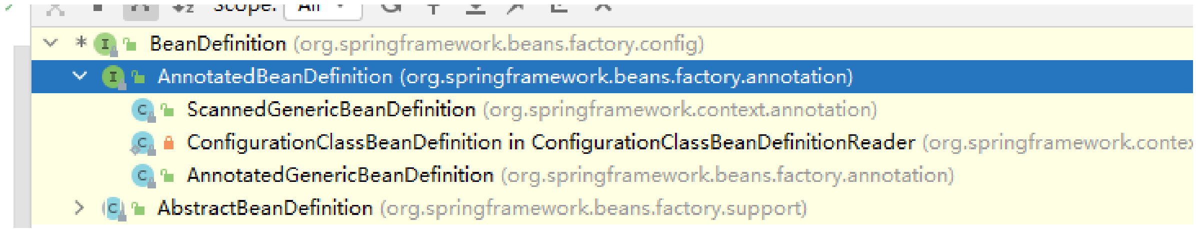
5. BeanDefinition继承体系

5.1 AnnotatedBeanDefinition

增加了2个方法，获取bean所在类的注解元数据和工厂方法元数据，这些数据在进行解析处理的时候需要用到。

```
public interface AnnotatedBeanDefinition extends BeanDefinition {

    /**
     * Obtain the annotation metadata (as well as basic class metadata)
     * for this bean definition's bean class.
     * @return the annotation metadata object (never {@code null})
     */
    AnnotationMetadata getMetadata();

    /**
     * Obtain metadata for this bean definition's factory method, if any.
     * @return the factory method metadata, or {@code null} if none
     * @since 4.1.1
     */
    @Nullable
    MethodMetadata getFactoryMethodMetadata();

}
```

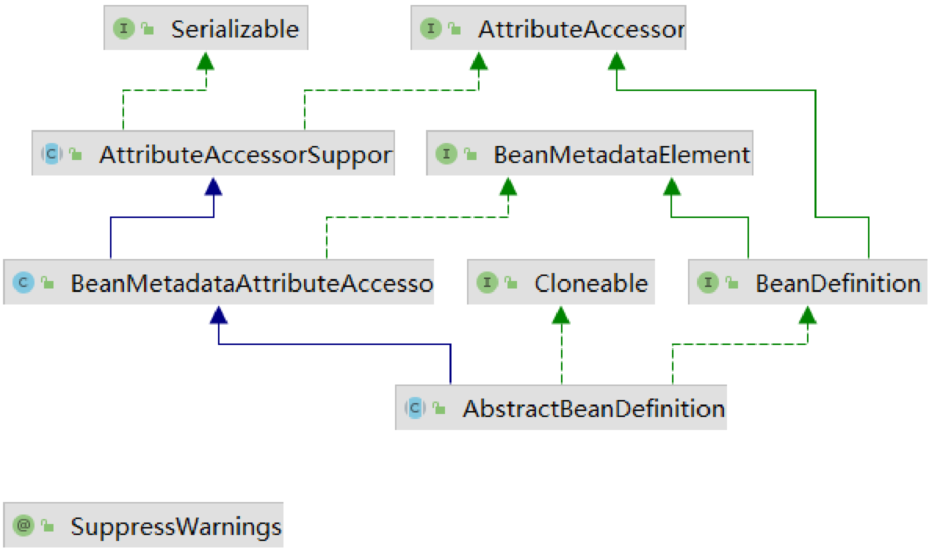
该注解有三个具体的实现。ScannedGenericBeanDefinition、AnnotatedGenericBeanDefinition、ConfigurationClassBeanDefinition。



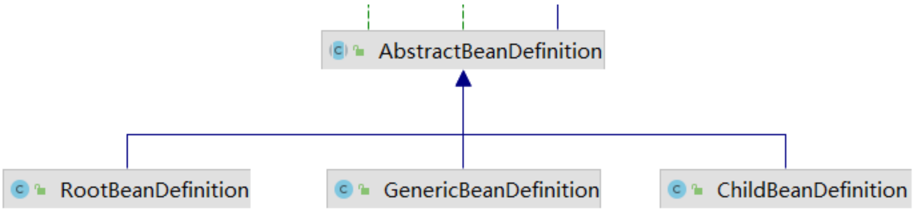
```
BeanDefinition (org.springframework.beans.factory.config)
  AnnotatedBeanDefinition (org.springframework.beans.factory.annotation)
    ScannedGenericBeanDefinition (org.springframework.context.annotation)
    ConfigurationClassBeanDefinition in ConfigurationClassBeanDefinitionReader (org.springframework.contex
    AnnotatedGenericBeanDefinition (org.springframework.beans.factory.annotation)
  AbstractBeanDefinition (org.springframework.beans.factory.support)
```

5.2 AbstractBeanDefinition模板类

AbstractBeanDefinition我们可以称之为BeanDefinition的模板类。结构我们上面其实有梳理



通过上面我们可以看到AbstractBeanDefinition 具备了 Bean元数据的获取和属性相关的操作。同时AbstractBeanDefinition的继承结构



5.3 RootBeanDefinition根bean定义

它主要用在spring内部的bean定义、把不同类型的bean定义合并成RootBeanDefinition（getMergedLocalBeanDefinition方法）。没有实现BeanDefinition接口的设置获取父bean定义方法，不支持设置父子beanDefinition。

5.4 ConfigurationClassBeanDefinition

用作ConfigurationClassPostProcessor解析过程中封装配置类的bean定义。

5.5 GenericBeanDefinition

GenericBeanDefinition通用Bean的定义。

5.6 ScannedGenericBeanDefinition

@ComponentScan扫描的bean定义使用。

5.7 AnnotatedGenericBeanDefinition