Spring源码专题

前提：Spring基础。

1，Spring源码开篇

为什么要学习Spring？源码

优秀框架，优雅。时间检验。

面试、设计模式，分层设计，架构体系。

构建起我们自己的知识体系，思考的质量，解决问题的速度。

1+1=2 --> 1+1+1=3 --> 1*3=3 --->33d9乘法口诀

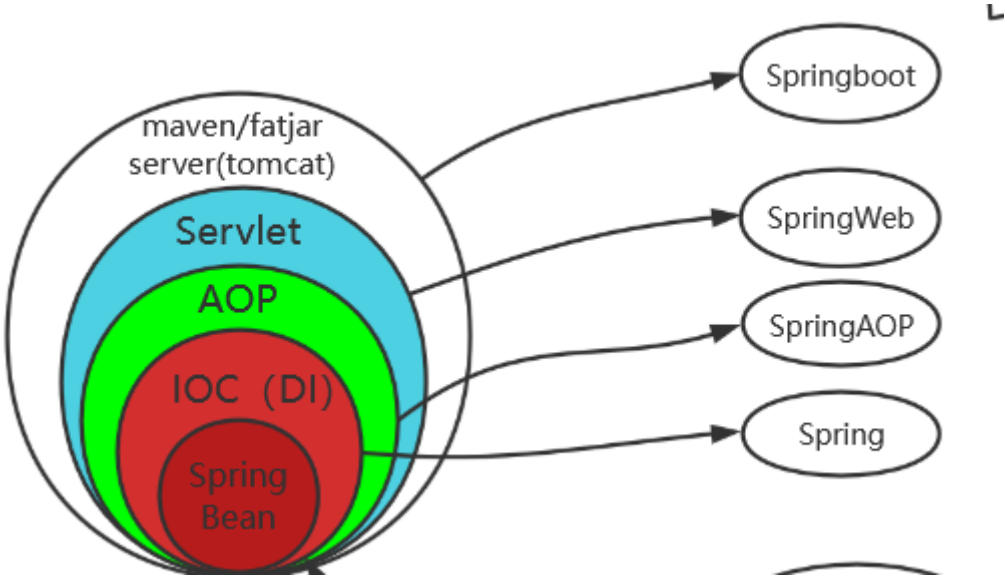
3*3 = 9

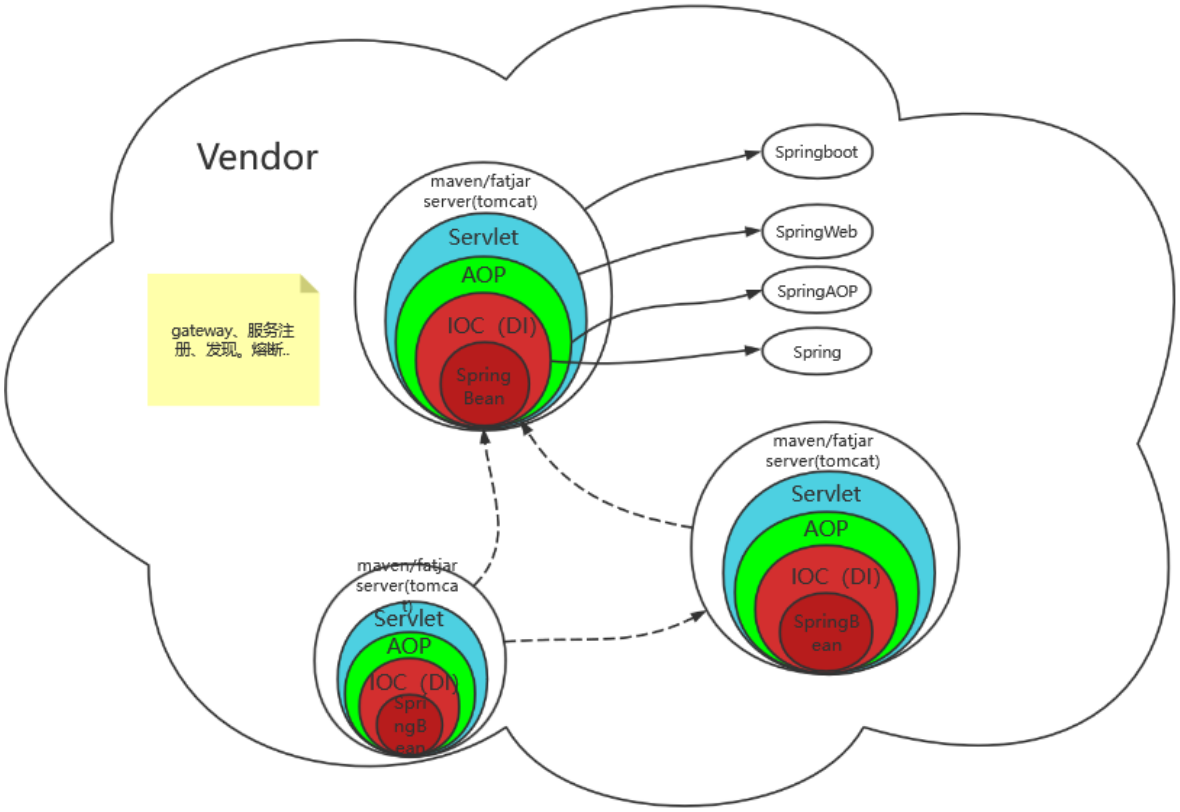
Spring-->Spring生态：Spring、SpringBoot、SpringCloud。

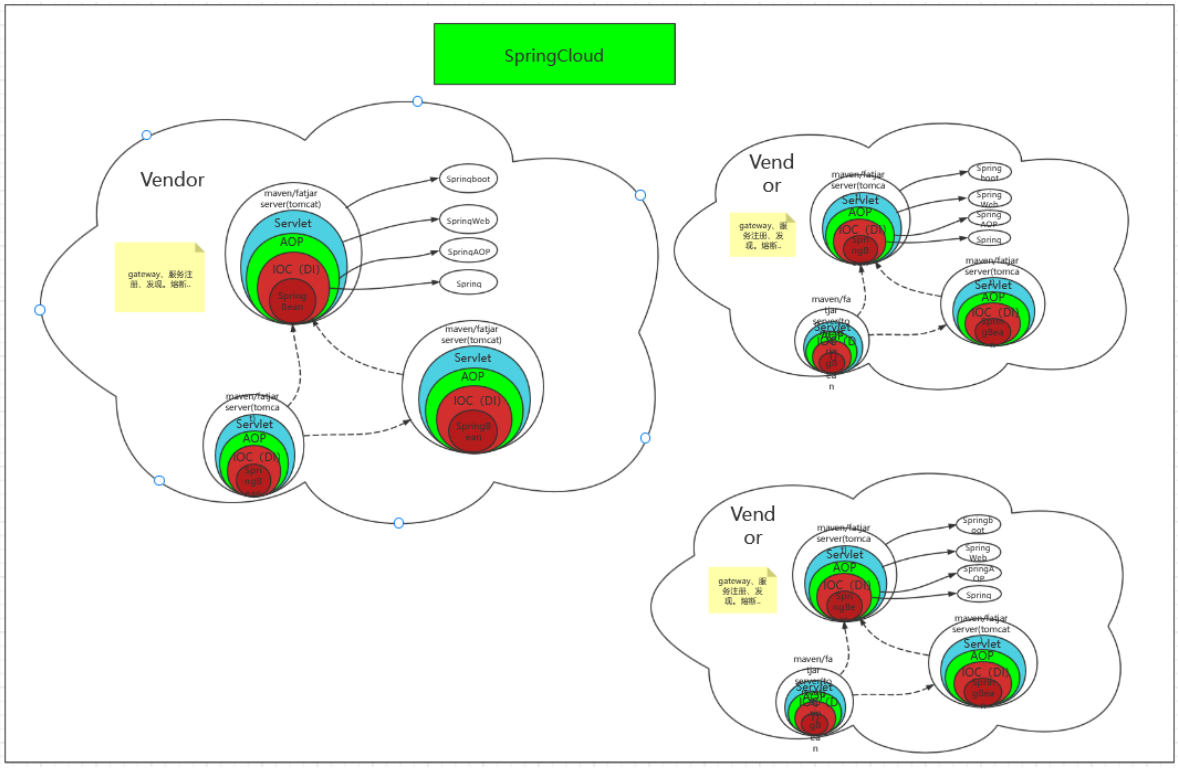
？

Spring生态全局观

对于spring生态的整体认识。见视频讲解。







理解Spring：

微观：具有黏合能力的框架（使用了IOC理念设计的SpringBean）

宏观：Spring生态。

2，快速回顾Spring用法

目的：熟悉Spring的用法、搭建起一个实验环境

2.1 依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.0.RELEASE</version>
    </dependency>
</dependencies>
```

2.2 相关类

```
/**
 * 主配置类
 */
@Configuration
@ComponentScan("com.myflx")//todo 自行改路径
public class AppConfig {
}

/**
 * 业务类
 */

@Repository
public class OrderDao {
    public void hello() {
        System.out.println("OrderDao hello...");
    }
}

@Service
//@Component
public class OrderService {
    @Autowired
    private OrderDao orderDao;

    public void hello() {
        System.out.println("OrderService hello...");
        orderDao.hello();
    }
}

/**
 * 入口类
 */
public class Bootstrap {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext annotationConfigApplicationContext =
new AnnotationConfigApplicationContext(AppConfig.class);
        OrderService orderService = (OrderService)
annotationConfigApplicationContext.getBean("orderService");
        orderService.hello();
    }
}
```

启动Bootstrap#main方法就可以 看到预期效果。简单的几个类就跑通了Spring。

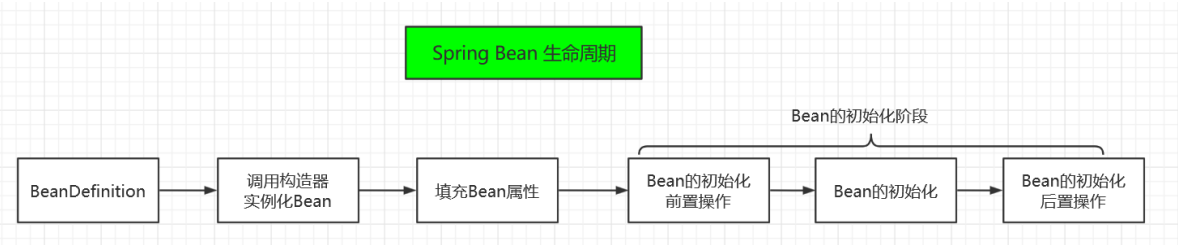
3，手写精简版Spring

1，前提要求

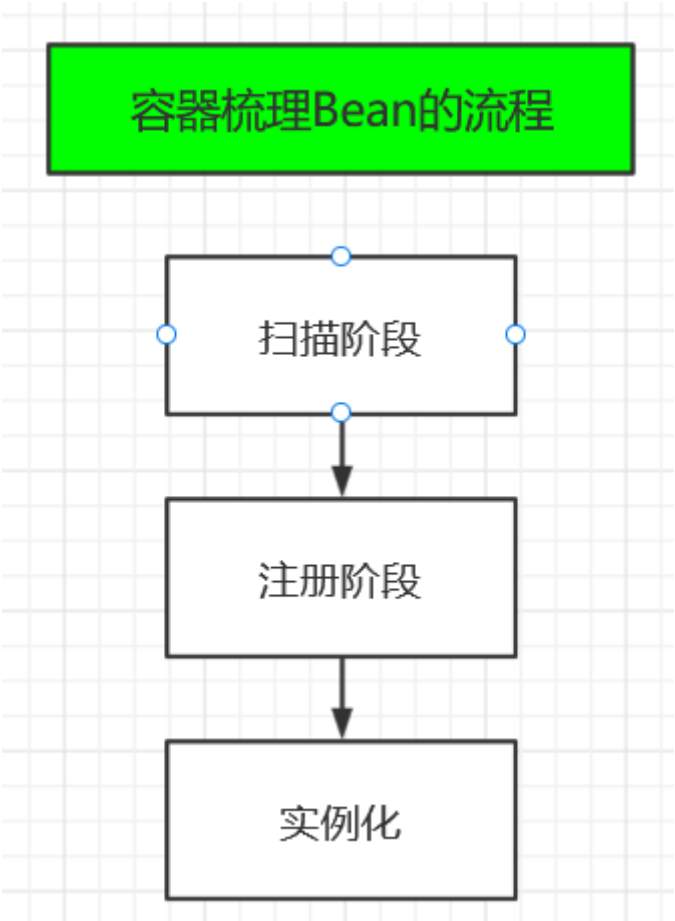
spring的两大核心：AOP、IOC。基于spring的bean，spring bean是核心中的核心。

Spring Bean生命周期

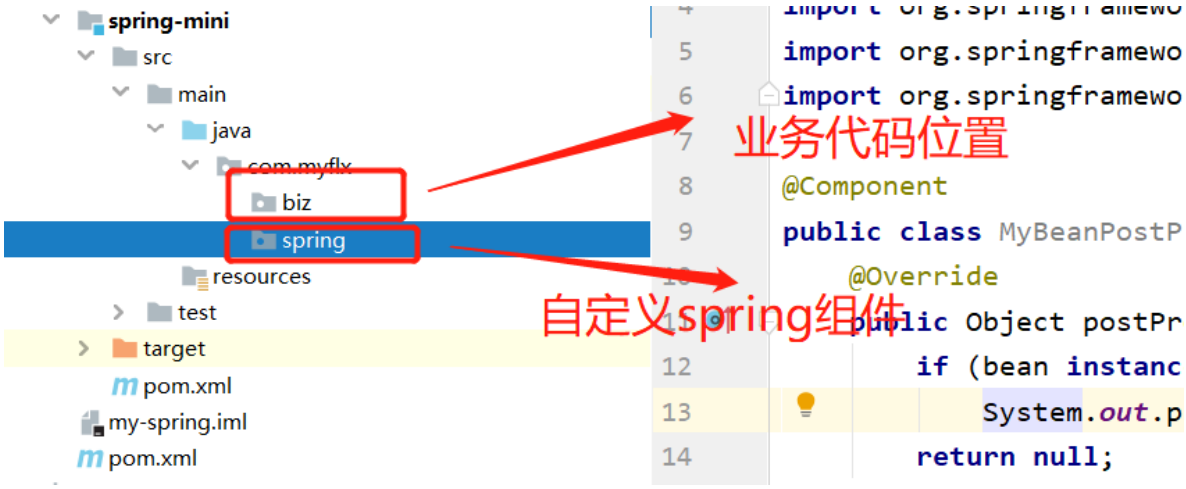
```
OrderService postProcessBeforeInitialization
OrderService initializing1...
OrderService postProcessAfterInitialization
orderService hello..
```



容器处理Bean的流程



2，项目结构



3，自定义Spring组件

自定义注解：`@Component`，`@ComponentScan`，`@Scope`，`@Autowired`

上下文：`AnntationConfigAplicationContext#getBean`

Bean相关组件：`BeanDefinition`，`InitializingBean`，`BeanPostProcessor`

4，代码实现

实现入口

```
public AnnotationConfigApplicationContext(Class<?> configClass) {
    this.configClass = configClass;
    //1.扫描。扫描包路径、解析所有文件信息。做判断
    //2.注册。如果说是符合条件的Bean（@Component 注解的类），将相关的Bean的信息转换成
BeanDefinition
    doScan();
    //3.实例化。单例对象（非懒加载的对象）
    initializeNotLazyBean();
}
```

```
public AnnotationConfigApplicationContext(Class<?> configClass) {
    this.configClass = configClass;
    //1.扫描。扫描包路径、解析所有文件信息。做判断
    //2.注册。如果说是符合条件的Bean（@Component 注解的类），将相关的Bean的信息转换成BeanDefinition
    doScan();
    //3.实例化。单例对象（非懒加载的对象）
    initializeNotLazyBean();
}
/** 扫描+注册 ...*/
private void doScan() {...}

/** 实例化-单例对象 ...*/
private void initializeNotLazyBean() {...}

/** 带缓存的Bean创建 ...*/
private Object createBean(BeanDefinition definition) {...}

/** 直接创建Bean ...*/
private Object doCreateBean(BeanDefinition definition) {...}
private Object initializing(Object instance, BeanDefinition definition) throws Exception {...}
private Object postProcessAfterInitialization(Object instance, BeanDefinition definition) {...}
private Object postProcessBeforeInitialization(Object instance, BeanDefinition definition) {...}
private void populateBean(Object instance, BeanDefinition definition) {...}

public Object getBean(String beanName) {...}
```

5，总结

感谢支持！

精简版的Spring

为什么要写精简版的Spring?

见视频讲解。

为什么要学习Spring?

开篇：提高自己能力

心理准备、认可------> 把事情想明白、搞清楚。

Spring怎么学?

方法论：效率、效果。

- 构建一个实验环境。**环境、准备工作。**
- 了解Spring的用法，用过Spring。**入门。**
- 确定一个核心，要研究的主题。SpringBean。SpringWeb、AOP、接口定义、设计模式、资源及其加载、上下文、事件。**抓住重点。**
- 关键因子：核心概念、生命周期、主脉络、主流程。**深入了解。**
- 模仿、学以致用的阶段。学习他的方法方式，学习他的设计。走一遍Spring走的过路，对他了解更深。**模仿，学以致用，举一反三。**

长期的过程

思考--->独立思考。

4，Spring源码解读说明

Spring版本：`5.2.0RELEASE`。

解读方式说明：

1. 直接在项目中依赖、下载源码包的方式，结合源码的注解官方文档。

GitHub下载源码----->安装gradle ---->处理依赖和报错。

耗费了大量时间，精力。激情磨灭。

2. 记录方式，直接复制关键代码，在文档中注释。

如果采用直接在源码中注释的方式：调试-->行数跳动。打开项目，代码量极大。

3. 边调试边注释的方式。
4. 抓住主线。
5. 带着问题。猜测+验证。

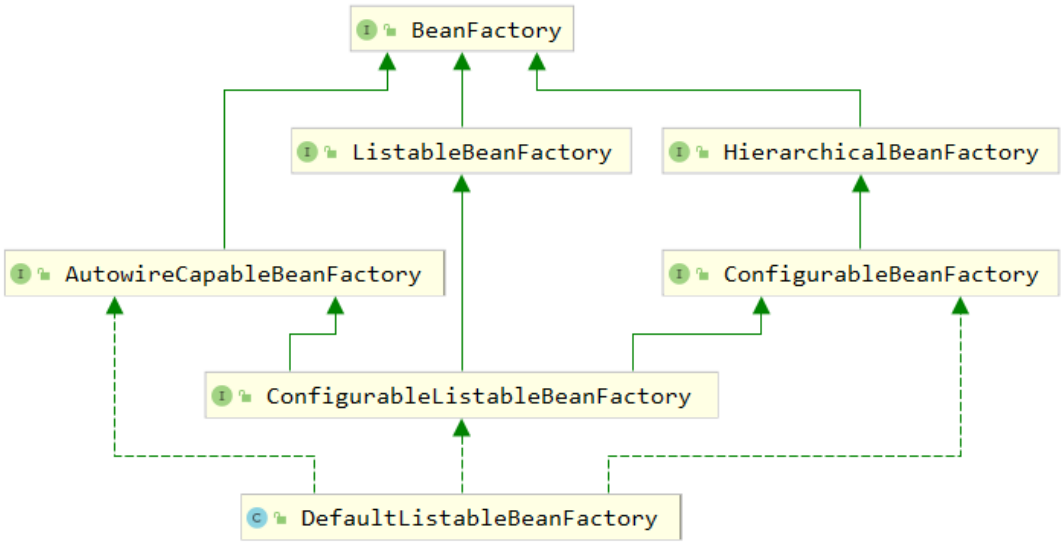
5，Spring源码解读之类体系结构说明

接口是一种规范。

学习：spring的接口设计，利用接口区分功能、隔离功能。

1，`BeanFactory`结构体系

逐个对相关的接口进行说明



BeanFactory

它是所有SpringBean容器的根接口，提供了一系列获取Bean的重载方法（支持懒加载ObjectProvider）,提供了判断是不是单例、原型、类型是不是匹配这样的接口。

```
The root interface for accessing a Spring bean container.
This is the basic client view of a bean container;
```

ListableBeanFactory

是`BeanFactory`的扩展接口。主要提供了获取BeanName数组、以BeanName做Key的Map，获取指定注解的BeanName数组或Map，通过bean和注解的类型获取指定注解。

HierarchicalBeanFactory

Hierarchical：分等级的、分层级的。

```
public interface HierarchicalBeanFactory extends BeanFactory {
    //获取父工厂
    //@see ConfigurableBeanFactory#setParentBeanFactory
    BeanFactory getParentBeanFactory();
    //判断当前工厂是否存在这个Bean。
    boolean containsLocalBean(String name);
}
```

ConfigurableBeanFactory

是BeanFactory体系里边的配置Bean工厂，提供了工厂里边的基础设施的配置方法集。为框架内部提供了一种可插拔的一种使用方式（plug'n'play），通过提供了一系列的配置。

例子：ConfigurableBeanFactory#setParentBeanFactory

AutowireCapableBeanFactory

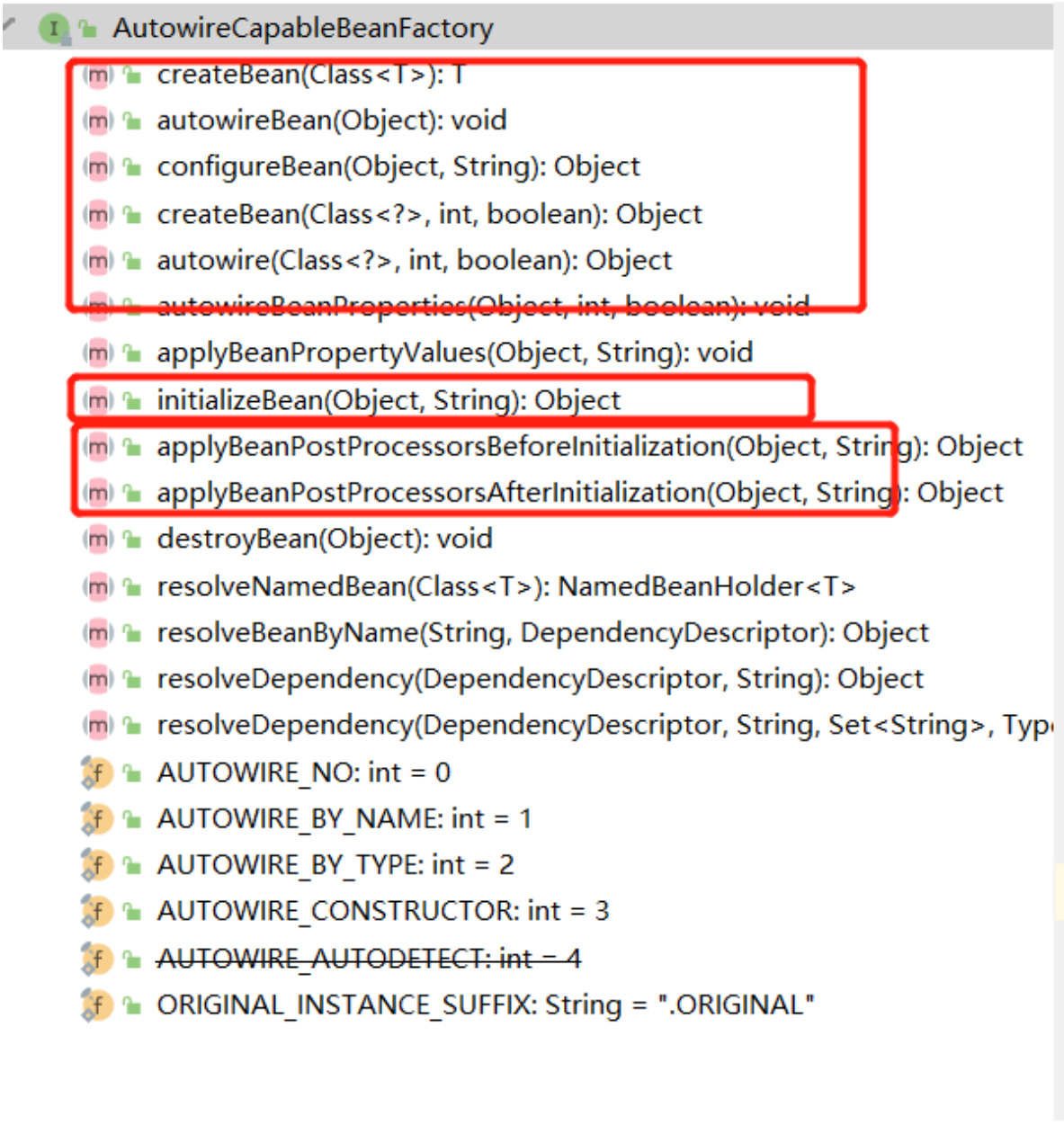
具有自动注入能力的工厂。主要是将Spring本身这种管理Bean生命周期的能力给暴露出来，给外部框架使用。

暴露了一些系列管理Bean生命周期的接口。

`ApplicationContext`没有继承该接口，通过方法`ApplicationContext#getAutowireCapableBeanFactory()`来暴露该能力。

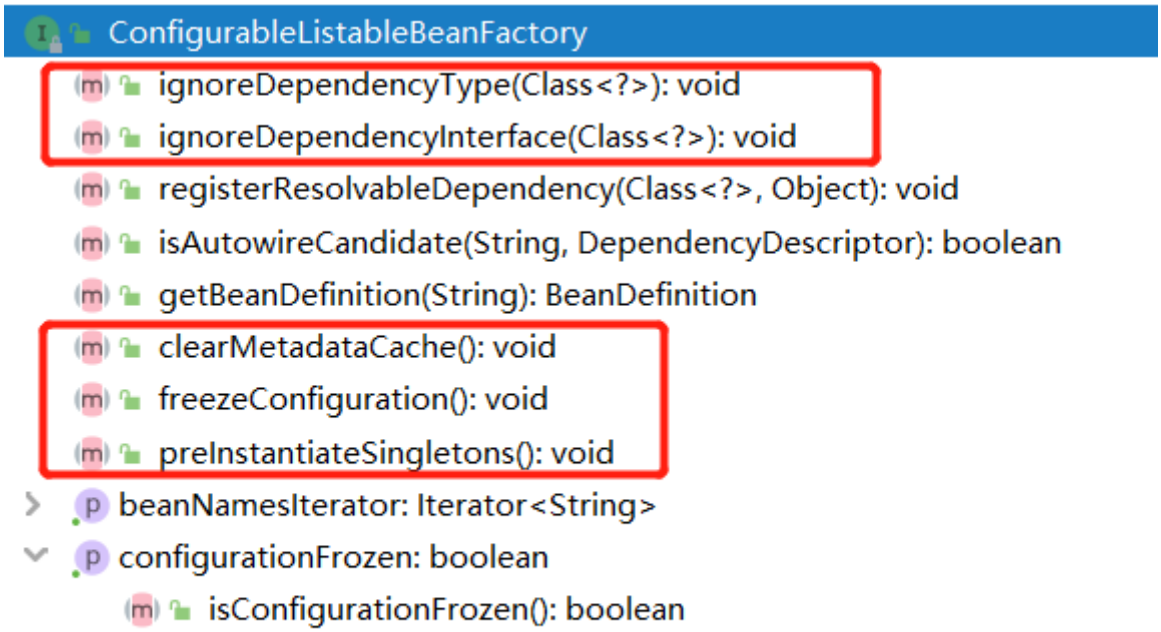
同时可以使用`BeanFactoryAware`的方式来暴露他的能力。

```
public class OrderService implements BeanFactoryAware {
    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("是不是AutowireCapableBeanFactory:" + (beanFactory
instanceof AutowireCapableBeanFactory));
    }
}
//结果输出：是不是AutowireCapableBeanFactory:true
```



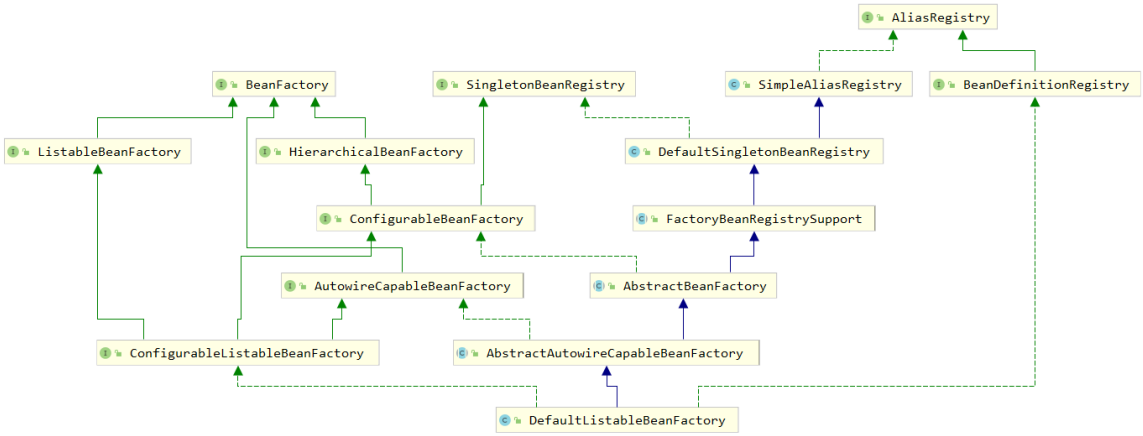
`ConfigurableListableBeanFactory`

是BeanFactory体系里边的配置接口，是对`ConfigurableBeanFactory`的扩展，主要是提供了对于分析、修改BeanDefinition的基础设置、对于**单例的预加载**。

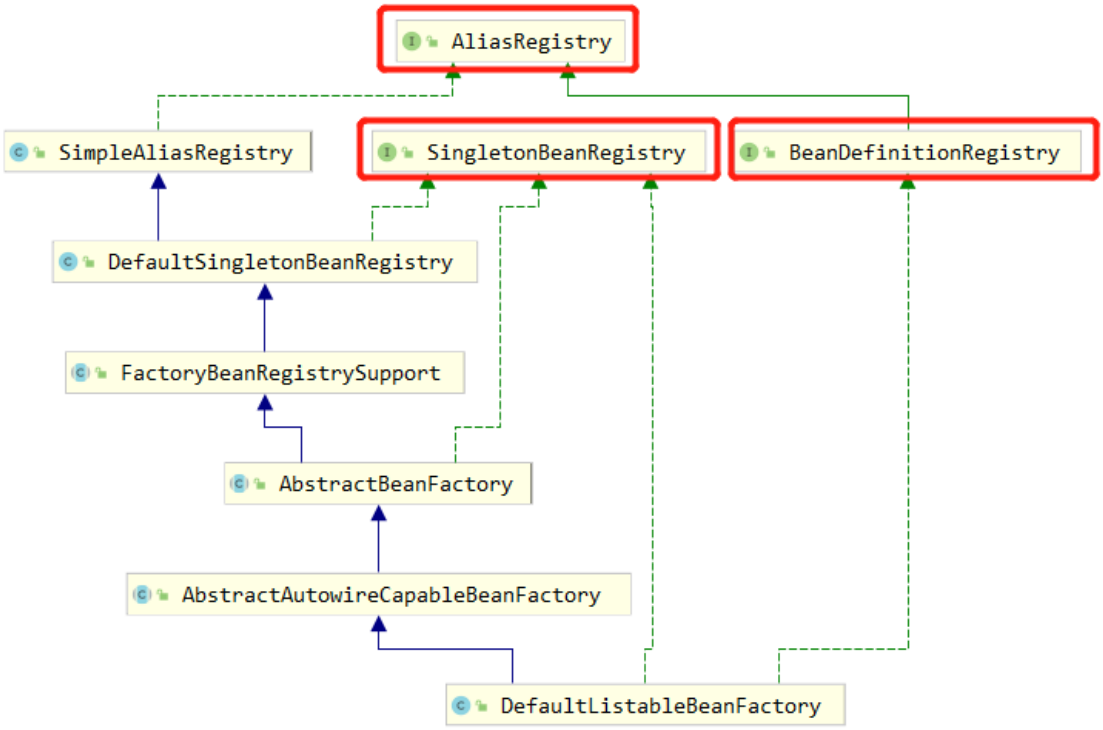


`DefaultListableBeanFactory`

```
//关键代码
/** Map of bean definition objects, keyed by bean name. */
private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<>(256);
```



2，注册系接口类结构体系



`AliasRegistry`

用于别名的注册，关键实现类：`SimpleAliasRegistry`

`SingletonBeanRegistry`

用于单例注册相关的接口，默认实现类：`DefaultSingletonBeanRegistry`

```
public interface SingletonBeanRegistry {
    void registerSingleton(String beanName, Object singletonObject);

    Object getSingleton(String beanName);

    boolean containsSingleton(String beanName);
    String[] getSingletonNames();

    //单例数量
    int getSingletonCount();

    //返回一个锁
    Object getSingletonMutex();

}
```

`DefaultSingletonBeanRegistry`：

```
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements
SingletonBeanRegistry {

    /** Cache of singleton objects: bean name to bean instance. */
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>
(256);
```

```
    /** Cache of singleton factories: bean name to ObjectFactory. */
    private final Map<String, ObjectFactory<?>> singletonFactories = new
HashMap<>(16);

    /** Cache of early singleton objects: bean name to bean instance. */
    private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

    /** Set of registered singletons, containing the bean names in registration
order. */
    private final Set<String> registeredSingletons = new LinkedHashSet<>(256);

    /** Names of beans that are currently in creation. */
    private final Set<String> singletonsCurrentlyInCreation =
            Collections.newSetFromMap(new ConcurrentHashMap<>(16));

    /** Names of beans currently excluded from in creation checks. */
    private final Set<String> inCreationCheckExclusions =
            Collections.newSetFromMap(new ConcurrentHashMap<>(16));

    /** List of suppressed Exceptions, available for associating related causes.
*/
    @Nullable
    private Set<Exception> suppressedExceptions;

    /** Flag that indicates whether we're currently within destroySingletons. */
    private boolean singletonsCurrentlyInDestruction = false;

    /** Disposable bean instances: bean name to disposable instance. */
    private final Map<String, Object> disposableBeans = new LinkedHashMap<>();

    /** Map between containing bean names: bean name to Set of bean names that
the bean contains. */
    private final Map<String, Set<String>> containedBeanMap = new
ConcurrentHashMap<>(16);

    /** Map between dependent bean names: bean name to Set of dependent bean
names. */
    private final Map<String, Set<String>> dependentBeanMap = new
ConcurrentHashMap<>(64);

    /** Map between depending bean names: bean name to Set of bean names for the
bean's dependencies. */
    private final Map<String, Set<String>> dependenciesForBeanMap = new
ConcurrentHashMap<>(64);
    ///.......//
}
```

`BeanDefinitionRegistry`

用于BeanDefinition的注册,`DefaultListableBeanFactory` 对其进行了实现。

3, `ApplicationContext`的结构体系

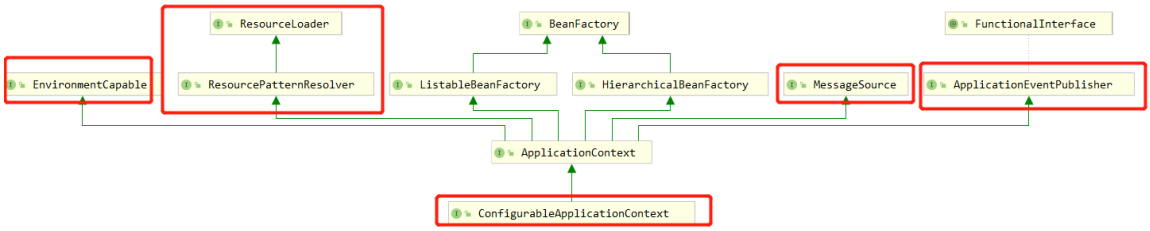
仅仅是一个门面类，集成了丰富的功能和功能入口。他集成了功能接口但是不对接口进行功能实现。

具有工厂的只读性质。

关键方法：

```
public interface ApplicationContext extends EnvironmentCapable,
ListableBeanFactory, HierarchicalBeanFactory,
        MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
            //具有层级结构
    ApplicationContext getParent();
            //暴露Spring体系中具有自动注入能力的工厂
    AutowireCapableBeanFactory getAutowireCapableBeanFactory() throws
IllegalStateException;

}
```

类结构体系图：



`EnvironmentCapable`

增加了环境的配置功能。

`ResourceLoader`

增加了资源加载的功能，重点关注。策略接口。

`MessageSource`

主要提供了国际化 的功能，策略接口。

`ApplicationEventPublisher`

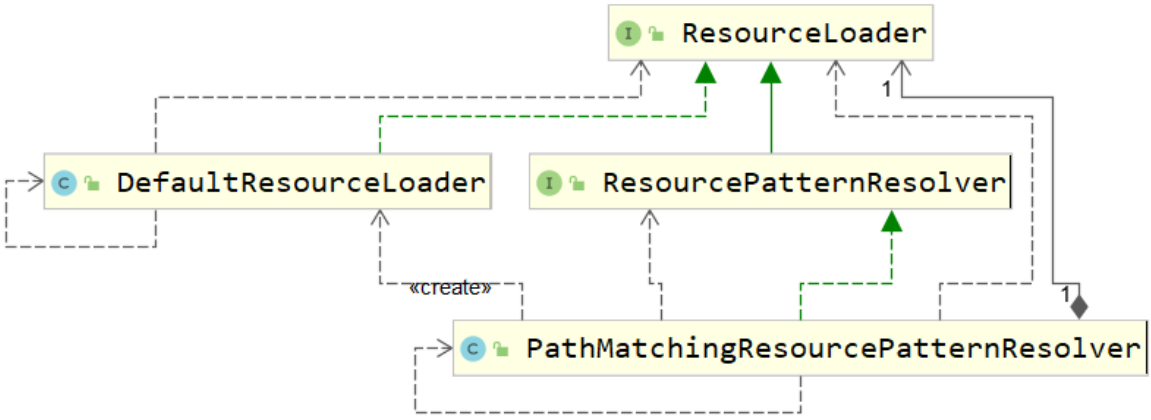
```
public interface ConfigurableApplicationContext extends ApplicationContext,
Lifecycle, Closeable {
    //。。。前面忽略//
    void setParent(@Nullable ApplicationContext parent);
    void setEnvironment(ConfigurableEnvironment environment);
    ConfigurableEnvironment getEnvironment();
    void addBeanFactoryPostProcessor(BeanFactoryPostProcessor postProcessor);
    void addApplicationListener(ApplicationListener<?> listener);
    //可以增加自定义协议
    void addProtocolResolver(ProtocolResolver resolver);
    void refresh() throws BeansException, IllegalStateException;
    void registerShutdownHook();
    ConfigurableListableBeanFactory getBeanFactory() throws
IllegalStateException;
}
```

4，Spring Resource结构体系

1，`ResourceLoader`结构体系

```java
public interface ResourceLoader {

    /** Pseudo URL prefix for loading from the class path: "classpath:". */
    //classpath协议的前缀
    String CLASSPATH_URL_PREFIX = "classpath:";

    //通过传入的资源路径返回一个资源
    Resource getResource(String location);

    //获取一个 ClassLoader
    ClassLoader getClassLoader();
}


public interface ResourcePatternResolver extends ResourceLoader {

    //classpath*协议的前缀。获取所有jar里边classpath下面的资源问题
    String CLASSPATH_ALL_URL_PREFIX = "classpath*:";

    //通过正则的资源路径获取到一个资源集合。
    Resource[] getResources(String locationPattern) throws IOException;
}

public interface ConfigurableApplicationContext extends ApplicationContext,
Lifecycle, Closeable {
    //省略其他 增加一个自定义协议处理器
    void addProtocolResolver(ProtocolResolver resolver);
}
```

类图:



2, `Reource`结构体系

什么是`Reource`?

`Reource`是Spring对于底层资源的抽象，定义了一系列的通用操作和属性，屏蔽了底层资源的操作细节。

Reource资源对象:

--->InputStream-->资源文件的内容--->解析内容，业务操作

--->URL 统一资源定位。

---->其他通用操作。

关键方法：

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
public interface Resource extends InputStreamSource {
    boolean exists();

    default boolean isReadable() {
        return exists();
    }

    default boolean isOpen() {
        return false;
    }

    default boolean isFile() {
        return false;
    }
    //资源定位符URL
    URL getURL() throws IOException;

    URI getURI() throws IOException;

    File getFile() throws IOException;

    default ReadableByteChannel readableChannel() throws IOException {
        return Channels.newChannel(getInputStream());
    }

    long contentLength() throws IOException;

    long lastModified() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

Java里是如何处理资源的呢?

URL？？需要补充关于URL的知识点。TODO

--->resource location ---->URL ---->

URLStreamHandler(协议的前缀：file、jar、war、http、https、ftp):

----->openConnetion()----->getInputStream();

ClassLoader是如何获取资源的?

```
public URL getResource(String name) {
    URL url;
    if (parent != null) {
        url = parent.getResource(name);
    } else {
        url = getBootstrapResource(name);
    }
    if (url == null) {
        url = findResource(name);
    }
    return url;
}
```

java里边并没有对于资源进行统一的一个抽象，对外暴露出来资源实际是一个URL对象。

为什么Spring还需要这样的Resource接口?

- Java没有对资源做一个很好的抽象，满足不了对于底层资源的操作的需求。例如：描述、可读性，是否打开...
- Java对于自定义协议具有一定的复杂性。

Spring的内建资源（Build-In Resource）

`UrlResource`

对于java URL 实现。

`ClassPathResource`

针对classpath的资源

`FileSystemResource`

文件系统资源

`ByteArrayResource`

`InputStreamResource`

默认是打开的，存在。

`ServletContextResource`

针对于`ServletContext`对应资源的封装。

5，BeanDefinition体系解读

1，什么是BeanDefinition?

Definition：定义（名词）

`BeanDefinition`是针对于由开发人员提供的**配置元数据（Configuration metadata）**的统一抽象。配置元数据包括：bean相关的基础属性（ClassName、beanName..）,bean行为相关的属性（scope、生命周期相关的回调方法，自动注入的模式。。。），bean的依赖属性。

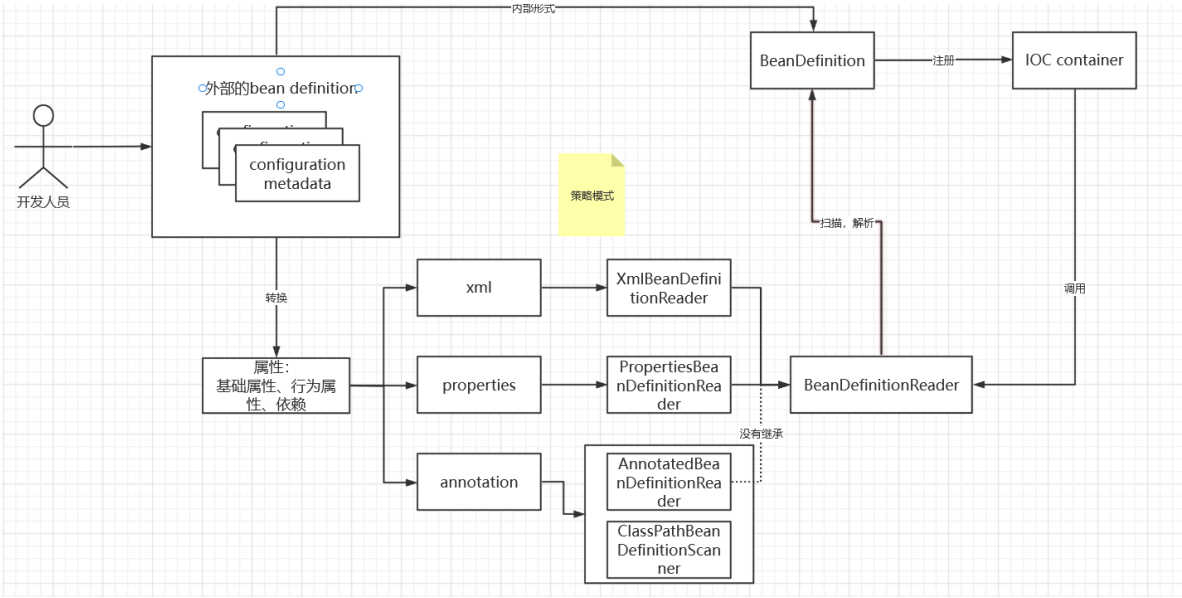
参考官网

Table 1. The bean definition

Property	Explained in...
Class--->String beanClassName	Instantiating Beans
Name-->BeanDefinitionHolder#beanName	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

2，BeanDefinition流程关系图

解读BeanDefinition是如何在Spring容器中发挥作用，怎么交互。



3，BeanDefinition扩展

BeanDefinition扩展：`BeanFactoryPostProcessor`

Spring容器三大扩展点：`BeanPostProcessor`，`BeanFactoryPostProcessor`，`FactoryBean`

5，Spring源码走读

1，确定主脉络

程序入口

```
public AnnotationConfigApplicationContext(Class<?>... componentClasses) {
    this();
    //通过AnnotatedBeanDefinitionReader 直接注册程序主配置类.
    register(componentClasses);
    //应用上下文启动
    refresh();
}
```

构造方法调用流程

```
//1-资源加载器层初始化了类加载器
public DefaultResourceLoader() {
    this.classLoader = ClassUtils.getDefaultClassLoader();
}

//2-初始化扩展的资源加载器
public AbstractApplicationContext() {
    //通过一个钩子方法（子类可以对其进行重写）初始化父类的扩展资源加载器。
    this.resourcePatternResolver = getResourcePatternResolver();//钩子
}

//3-初始化实际用来注册的Spring IOC 容器
public GenericApplicationContext() {
    this.beanFactory = new DefaultListableBeanFactory();
}


//4.1-BeanDefinitionReader相关的初始化
public AnnotationConfigApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader(this);
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}

//4.2-最后调用什么的构造方法
public AnnotationConfigApplicationContext(Class<?>... componentClasses) {
    this();
    //通过AnnotatedBeanDefinitionReader 直接注册程序主配置类.
    register(componentClasses);
    //启动应用上下文
    refresh();
}
```

获取默认类加载器

`org.springframework.util.ClassUtils#getDefaultClassLoader`

```
public static ClassLoader getDefaultClassLoader() {
    ClassLoader cl = null;
    try {
        cl = Thread.currentThread().getContextClassLoader();
    }
    catch (Throwable ex) {
        // Cannot access thread context ClassLoader - falling back...
    }
    if (cl == null) {
        // No thread context class loader -> use class loader of this class.
```

```
        cl = ClassUtils.class.getClassLoader();
        if (cl == null) {
            // getClassLoader() returning null indicates the bootstrap
ClassLoader
            try {
                cl = ClassLoader.getSystemClassLoader();
            }
            catch (Throwable ex) {
                // Cannot access system ClassLoader - oh well, maybe the caller
can live with null...
            }
        }
    }
    return cl;
}
```

2，注册程序主配置类

`AnnotationConfigApplicationContext#doRegisterBean`

```
private <T> void doRegisterBean(Class<T> beanClass, @Nullable String name,
            @Nullable Class<? extends Annotation>[] qualifiers, @Nullable
Supplier<T> supplier,
            @Nullable BeanDefinitionCustomizer[] customizers) {
    //构建BeanDefinition
    AnnotatedGenericBeanDefinition abd = new
AnnotatedGenericBeanDefinition(beanClass);
    //判断定义是否符合条件
    if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
        return;
    }

    abd.setInstanceSupplier(supplier);
    //解析Scope元数据
    ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(abd);
    abd.setScope(scopeMetadata.getScopeName());
    //构建BeanName
    String beanName = (name != null ? name :
this.beanNameGenerator.generateBeanName(abd, this.registry));
    //一些注解属性的设置
    AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
    if (qualifiers != null) {
        for (Class<? extends Annotation> qualifier : qualifiers) {
            if (Primary.class == qualifier) {
                abd.setPrimary(true);
            }
            else if (Lazy.class == qualifier) {
                abd.setLazyInit(true);
            }
            else {
                abd.addQualifier(new AutowireCandidateQualifier(qualifier));
            }
        }
    }
```

```
    if (customizers != null) {
        for (BeanDefinitionCustomizer customizer : customizers) {
            customizer.customize(abd);
        }
    }
    //封装BeanDefinitionHolder
    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd,
beanName);
    //应用代理模式，如果需要代理就注入一个 scopedTarget.appConfig 定义
    definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, this.registry);
    //注册定义
    BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
this.registry);
}
```

3，应用上下文启动

接口方法定义:`ConfigurableApplicationContext#refresh`

方法实现：`AbstractApplicationContext#refresh`

主脉络:

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        //启动前的准备工作：属性变量的初始化、校验。
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        //获取ConfigurableListableBeanFactory，如果工厂不存在就进行创建。
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        //工厂的准备阶段：一些属性、核心组件的设置，关键BeanPostProcessor注册，单例对象的注
册。
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            //子类应用上下文对于工厂的准备阶段（postPrepareBeanFactory）
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            //调用BeanFactoryPostProcessor
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            //注册一些核心BeanPostProcessor：有哪些？什么用？
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            //初始化MessageSource
            initMessageSource();
```

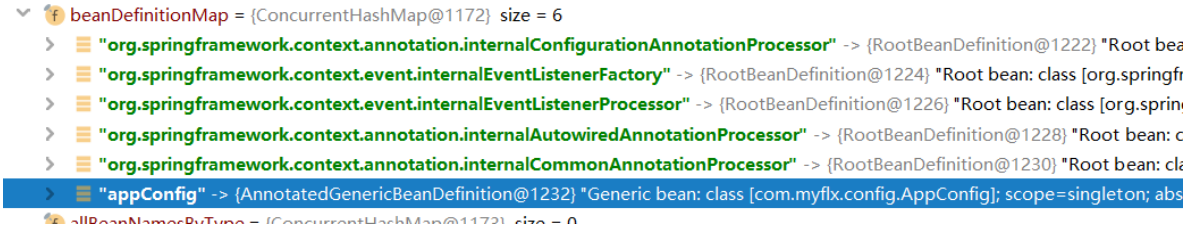
```
            // Initialize event multicaster for this context.
            //初始化应用事件广播器：ApplicationEventMulticaster
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            //留给子类的初始化方法
            onRefresh();

            // Check for listener beans and register them.
            //注册应用事件监听器
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            //初始化完成阶段：做了非懒加载对象的实例化。
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            //应用上下文启动完成，发布事件。
            finishRefresh();
        }
        catch (BeansException ex) {
            //....
        }
        finally {
            // Reset common introspection caches in Spring's core, since we
            // might not ever need metadata for singleton beans anymore...
        }
    }
}
```

4，`BeanDefinition`扫描&注册

疑问？默认的定义是在哪个地方注册的？

在`AnnotatedBeanDefinitionReader`的构造方法进行了spring内置定义的注册。

有什么用？结合具体的功能。

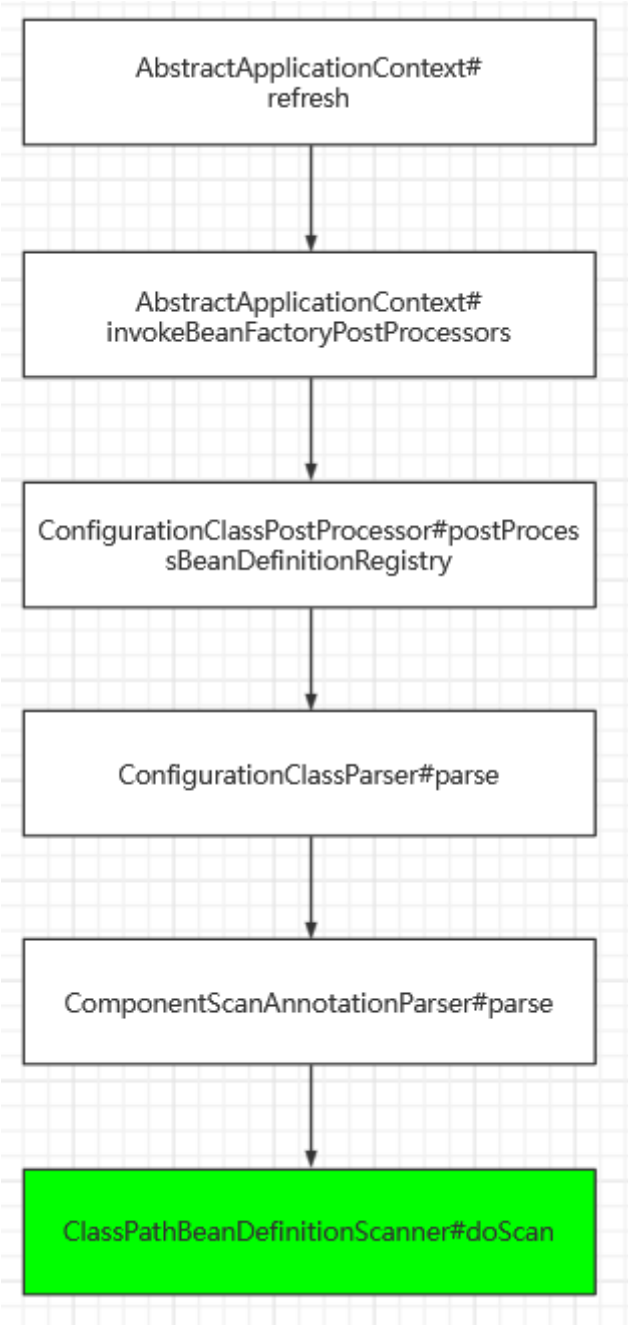


1，internalConfigurationAnnotationProcessor

`ConfigurationClassPostProcessor`实现了接口:`BeanDefinitionRegistryPostProcessor`同时是实现 了接口`BeanFactoryPostProcessor`，作为容器的扩展点植入容器，成为了`BeanDefinition`进行扫描注册的入口。

1，`BeanDefinition`扫描的前置流程

扫描发生在上下文启动的`AbstractApplicationContext#invokeBeanFactoryPostProcessors`阶段



`ConfigurationClassPostProcessor` 实现了接口: `BeanDefinitionRegistryPostProcessor`

```
public interface BeanDefinitionRegistryPostProcessor extends
BeanFactoryPostProcessor {
    /**
     * Modify the application context's internal bean definition registry after
its
     * standard initialization. All regular bean definitions will have been
loaded,
     * but no beans will have been instantiated yet. This allows for adding
further
     * bean definitions before the next post-processing phase kicks in.
     * @param registry the bean definition registry used by the application
context
     * @throws org.springframework.beans.BeansException in case of errors
     */
    void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException;
}
```

2，`BeanDefinition`扫描&注册解读

扫描的主脉络解读

```
protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages, "At least one base package must be
specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
    for (String basePackage : basePackages) {
        //具体扫描包路径并获取定义结合.重点。
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            //解析作用域的元数据
            ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            //构造BeanName
            String beanName = this.beanNameGenerator.generateBeanName(candidate,
this.registry);
            //做了BeanDefinition的基础属性设置。
            if (candidate instanceof AbstractBeanDefinition) {
                postProcessBeanDefinition((AbstractBeanDefinition) candidate,
beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinitio
n) candidate);
            }
            //检查定义是否符合条件
            if (checkCandidate(beanName, candidate)) {
                //包装成BeanDefinitionHolder
                BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder(candidate, beanName);
                //应用作用域的代理模式
                definitionHolder =
                    AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, this.registry);
                beanDefinitions.add(definitionHolder);
                //将定义注册到容器中。
                registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}
```

扫描之获取BeanDefinition集合

```
public Set<BeanDefinition> findCandidateComponents(String basePackage) {
    if (this.componentsIndex != null && indexSupportsIncludeFilters()) {
       //走索引的方式
        return addCandidateComponentsFromIndex(this.componentsIndex,
basePackage);
    }
    else {
        //正常扫描 classpath路径下的扫描。
        return scanCandidateComponents(basePackage);
    }
}
```

```
使用component-index索引
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-indexer</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

扫描之实际进行扫描的方法

```
private Set<BeanDefinition> scanCandidateComponents(String basePackage) {
        Set<BeanDefinition> candidates = new LinkedHashSet<>();
    try {
        //构建一个classpath*协议字符串
        String packageSearchPath =
ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX +
            resolveBasePackage(basePackage) + '/' + this.resourcePattern;
        //通过容器的资源加载器获取资源数组
        Resource[] resources =
getResourcePatternResolver().getResources(packageSearchPath);
        for (Resource resource : resources) {
            if (resource.isReadable()) {
                try {
                //通过MetadataReaderFactory接受资源对象获取了元数据读取器。重点关注如何
获取元数据
                    MetadataReader metadataReader =
getMetadataReaderFactory().getMetadataReader(resource);
                    //校验是否component是否符合条件。主要针对filter相关的校验
                    if (isCandidateComponent(metadataReader)) {
                        ScannedGenericBeanDefinition sbd = new
ScannedGenericBeanDefinition(metadataReader);
                        sbd.setResource(resource);
                        sbd.setSource(resource);
                        //继续进行定义合法性的校验
                        if (isCandidateComponent(sbd)) {
                            candidates.add(sbd);
                        }
                    }
    return candidates;
}
```

扫描之注册前校验

```
protected boolean checkCandidate(String beanName, BeanDefinition beanDefinition)
throws IllegalStateException {
    //判断是否已经注册进去了
        if (!this.registry.containsBeanDefinition(beanName)) {
            return true;
        }
    //如果注册进去了
        BeanDefinition existingDef = this.registry.getBeanDefinition(beanName);
        BeanDefinition originatingDef =
existingDef.getOriginatingBeanDefinition();
        if (originatingDef != null) {
            existingDef = originatingDef;
        }
    //判断兼容性
        if (isCompatible(beanDefinition, existingDef)) {
            return false;
        }
        throw new ...
    }
```

5，Bean的预加载

应用上下文启动过程中，非懒加载的单例对象的实例化

`AbstractApplicationContext#finishBeanFactoryInitialization`

上下文BeanFactory初始化的完成阶段

```
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {
    // Initialize conversion service for this context.
    if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
        beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class)) {
        beanFactory.setConversionService(
        beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class));
    }

    // Register a default embedded value resolver if no bean post-processor
    // (such as a PropertyPlaceholderConfigurer bean) registered any before:
    // at this point, primarily for resolution in annotation attribute values.
    if (!beanFactory.hasEmbeddedValueResolver()) {
        beanFactory.addEmbeddedValueResolver(strVal ->
getEnvironment().resolvePlaceholders(strVal));
    }

    // Initialize LoadTimeWeaverAware beans early to allow for registering their
transformers early.
    String[] weaverAwareNames =
beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
    for (String weaverAwareName : weaverAwareNames) {
        getBean(weaverAwareName);
    }

    // Stop using the temporary ClassLoader for type matching.
    beanFactory.setTempClassLoader(null);
```

```
    // Allow for caching all bean definition metadata, not expecting further
changes.
    //冻结配置（BeanDefinition），给了一个冻结标记。应用上下文已经启动。
    beanFactory.freezeConfiguration();

    // Instantiate all remaining (non-lazy-init) singletons.
    beanFactory.preInstantiateSingletons();
}
```

1，配置冻结

给了一个冻结标记、将定义的列表转为数组。

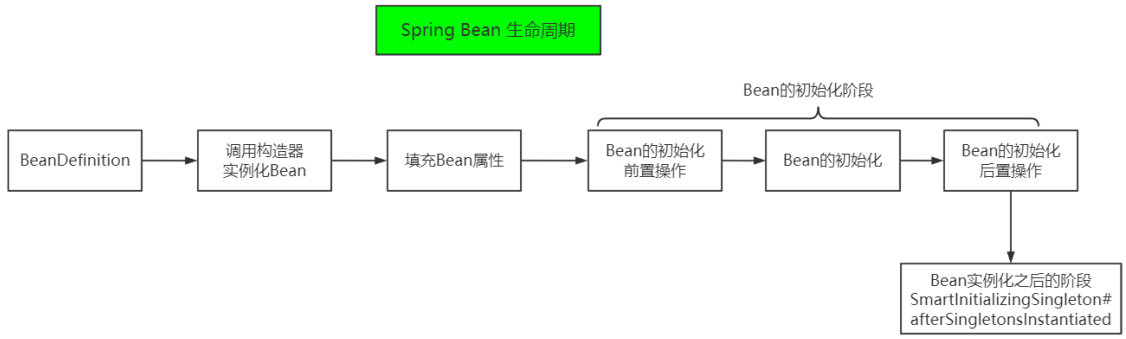
```
//DefaultListableBeanFactory#freezeConfiguration
@Override
public void freezeConfiguration() {
    this.configurationFrozen = true;
    this.frozenBeanDefinitionNames =
StringUtils.toStringArray(this.beanDefinitionNames);
}
```

2，预实例化非懒加载单例

```
@Override
public void preInstantiateSingletons() throws BeansException {
    List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

    // Trigger initialization of all non-lazy singleton beans...
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        //非抽象的、是单例的、非懒加载的
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            //判断是否为FactoryBean
            if (isFactoryBean(beanName)) {
                Object bean = getBean(FACTORY_BEAN_PREFIX + beanName);
                if (bean instanceof FactoryBean) {
                    final FactoryBean<?> factory = (FactoryBean<?>) bean;
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory
instanceof SmartFactoryBean) {
                        isEagerInit =
AccessController.doPrivileged((PrivilegedAction<Boolean>)

((SmartFactoryBean<?>) factory)::isEagerInit,

 getAccessControlContext());
                    }
                    else {
                        isEagerInit = (factory instanceof SmartFactoryBean &&
                                       ((SmartFactoryBean<?>)
factory).isEagerInit());
                    }
                    if (isEagerInit) {
                        getBean(beanName);
                    }
```

```
                }
            }
            else {
                //实例化单例的bean。重点。。
                getBean(beanName);
            }
        }
    }
    //bean的生命周期多了一个节点。。
    // Trigger post-initialization callback for all applicable beans...
    for (String beanName : beanNames) {
        Object singletonInstance = getSingleton(beanName);
        if (singletonInstance instanceof SmartInitializingSingleton) {
            final SmartInitializingSingleton smartSingleton =
(SmartInitializingSingleton) singletonInstance;
            if (System.getSecurityManager() != null) {
                AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                    smartSingleton.afterSingletonsInstantiated();
                    return null;
                }, getAccessControlContext());
            }
            else {
                smartSingleton.afterSingletonsInstantiated();
            }
        }
    }
}
```

生命周期补充



3，实例化：getBean(beanName)

```
doGetBean(name, null, null, false)
```

实际实例化Bean的位置：

`AbstractBeanFactory#doGetBean`

```
protected <T> T doGetBean(final String name, @Nullable final Class<T>
requiredType,
            @Nullable final Object[] args, boolean typeCheckOnly) throws
BeansException {
    final String beanName = transformedBeanName(name);
    Object bean;
    // Eagerly check singleton cache for manually registered singletons.
    //从单例池中获取单例对象。重点。
```

```
    Object sharedInstance = getSingleton(beanName);
    if (sharedInstance != null && args == null) {
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }else {
        // Fail if we're already creating this bean instance:
        // We're assumably within a circular reference.
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }
        //从父工厂中获取Bean。
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            // Not found -> check parent.
            String nameToLookup = originalBeanName(name);
            if (parentBeanFactory instanceof AbstractBeanFactory) {
                return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                    nameToLookup, requiredType, args, typeCheckOnly);
            }
            else if (args != null) {
                // Delegation to parent with explicit args.
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            }
            else if (requiredType != null) {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
            else {
                return (T) parentBeanFactory.getBean(nameToLookup);
            }
        }

        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        try {
            final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
            checkMergedBeanDefinition(mbd, beanName, args);

            // Guarantee initialization of beans that the current bean depends
on.
            //实例化依赖的对象
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    if (isDependent(beanName, dep)) {
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                                              "Circular depends-on
relationship between '" + beanName + "' and '" + dep + "'");
                    }
                    registerDependentBean(dep, beanName);
                    try {
                        getBean(dep);
                    }
                    catch (NoSuchBeanDefinitionException ex) {
```

```
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                                                "'" + beanName + "'
depends on missing bean '" + dep + "'", ex);
                    }
                }
            }
            // Create bean instance.
            //针对于单例对象的实例化
            if (mbd.isSingleton()) {
                sharedInstance = getSingleton(beanName, () -> {
                    try {
                        return createBean(beanName, mbd, args);
                    }
                    catch (BeansException ex) {destroySingleton(beanName);
                        throw ex;
                    }
                });
                bean = getObjectForBeanInstance(sharedInstance, name, beanName,
mbd);
            }
            //针对于原型对象的实例化
            else if (mbd.isPrototype()) {
                // It's a prototype -> create a new instance.
                Object prototypeInstance = null;
                try {
                    beforePrototypeCreation(beanName);
                    prototypeInstance = createBean(beanName, mbd, args);
                }
                finally {
                    afterPrototypeCreation(beanName);
                }
                bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
            }
            //针对于自定义作用域对象的实例化
            else {
                String scopeName = mbd.getScope();
                final Scope scope = this.scopes.get(scopeName);
                if (scope == null) {
                    throw new IllegalStateException("No Scope registered for
scope name '" + scopeName + "'");
                }
                try {
                    Object scopedInstance = scope.get(beanName, () -> {
                        beforePrototypeCreation(beanName);
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        finally {
                            afterPrototypeCreation(beanName);
                        }
                    });
                    bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
                }
                catch (IllegalStateException ex) {//
                }
```

```
            }
        }
        catch (BeansException ex) {
            cleanupAfterBeanCreationFailure(beanName);
            throw ex;
        }
    }

    // Check if required type matches the type of the actual bean instance.
    if (requiredType != null && !requiredType.isInstance(bean)) {
        try {
            T convertedBean = getTypeConverter().convertIfNecessary(bean,
requiredType);
            if (convertedBean == null) {
                throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
            }
            return convertedBean;
        }
        catch (TypeMismatchException ex) {
            throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
        }
    }
    return (T) bean;
}
```

4，从单例池中获取对象：getSingleton(beanName)

```
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    //从一级缓存中获取对象
    Object singletonObject = this.singletonObjects.get(beanName);
    //一级缓存中没有并且当前的单例对象正在创建中
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {//lock
            //从二级缓存中获取对象
            singletonObject = this.earlySingletonObjects.get(beanName);
            //二级缓存中没有并且当前运行获取提前暴露的对象
            if (singletonObject == null && allowEarlyReference) {
                //从提前暴露（暴露的是对象工厂）的三级缓存中获取
                ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

5，getSingleton(beanName, ObjectFactory)

创建单例核心逻辑的around逻辑。lock--》double check，前后置的创建中占位逻辑，缓存操作。

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(beanName, "Bean name must not be null");
    synchronized (this.singletonObjects) {
        //double check
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null) {
            if (this.singletonsCurrentlyInDestruction) {
                throw ,,,,,
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Creating shared instance of singleton bean '" +
beanName + "'");
            }
            //创建单例核心逻辑的前置操作
            beforeSingletonCreation(beanName);
            boolean newSingleton = false;
            boolean recordSuppressedExceptions = (this.suppressedExceptions ==
null);
            if (recordSuppressedExceptions) {
                this.suppressedExceptions = new LinkedHashSet<>();
            }
            try {
                //创建单例核心逻辑
                singletonObject = singletonFactory.getObject();
                newSingleton = true;
            }
            catch (IllegalStateException ex) {
                , , , ,
            }
            finally {
                if (recordSuppressedExceptions) {
                    this.suppressedExceptions = null;
                }
                //创建单例核心逻辑的后置操作
                afterSingletonCreation(beanName);
            }
            if (newSingleton) {
                //添加单例到响应的缓存池中。
                addSingleton(beanName, singletonObject);
            }
        }
        return singletonObject;
    }
}


protected void addSingleton(String beanName, Object singletonObject) {
    synchronized (this.singletonObjects) {
        this.singletonObjects.put(beanName, singletonObject);
        this.singletonFactories.remove(beanName);
        this.earlySingletonObjects.remove(beanName);
        this.registeredSingletons.add(beanName);
    }
```

```
}
```

6，createBean(beanName, mbd, args)

创建单例核心逻辑的前置逻辑。类的加载、实例化的前置操作。

```
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable
Object[] args){
    RootBeanDefinition mbdToUse = mbd;
    //对bean的类型进行处理，主要是加载bean的类型。
    Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
    ///.....////
    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the
target bean instance.
        //应用bean实例化前的前置操作，通过BeanPostProcessors的方式留了口子，运行外界通过：
InstantiationAwareBeanPostProcessor#postProcessBeforeInstantiation的来创建可以投产
的bean。
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbdToUse.getResourceDescription(),
beanName,
                                        "BeanPostProcessor before instantiation
of bean failed", ex);
    }

    try {
        ///bean创建的核心逻辑。
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName +
"'");
        }
        return beanInstance;
    }
}
```

7，doCreateBean

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition
mbd, final @Nullable Object[] args)
            throws BeanCreationException {
    // Instantiate the bean.
    //bean 的实例化
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
```

```
            instanceWrapper = createBeanInstance(beanName, mbd, args);
        }
        final Object bean = instanceWrapper.getWrappedInstance();
        Class<?> beanType = instanceWrapper.getWrappedClass();
        if (beanType != NullBean.class) {
            mbd.resolvedTargetType = beanType;
        }

        // Allow post-processors to modify the merged bean definition.
        //应用MergedBeanDefinitionPostProcessor来处理元数据：Autowired、init、
destroy。。。
        synchronized (mbd.postProcessingLock) {
            if (!mbd.postProcessed) {
                try {
                    applyMergedBeanDefinitionPostProcessors(mbd, beanType,
beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
                                                    "Post-processing of merged bean
definition failed", ex);
                }
                mbd.postProcessed = true;
            }
        }

        // Eagerly cache singletons to be able to resolve circular references
        // even when triggered by lifecycle interfaces like BeanFactoryAware.
        //判断是否运行循环依赖：单例的、运行循环依赖、正在创建中。
        boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
                                          isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            //解决循环依赖的关键点：提前暴露对象，ObjectFactory为了getEarlyBeanReference
            addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd,
bean));
        }

        // Initialize the bean instance.
        Object exposedObject = bean;
        try {
            //bean的属性填充。。
            populateBean(beanName, mbd, instanceWrapper);
            //bean的初始化逻辑。初始化的阶段运行对对象进行替换。
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
        catch (Throwable ex) {
            ```
        }
        if (earlySingletonExposure) {
            //从单例池中获取对象（不允许从提前暴露的对象池中获取）
            Object earlySingletonReference = getSingleton(beanName, false);
            if (earlySingletonReference != null) {//一定发生了循环依赖。。
                if (exposedObject == bean) {
                    //循环依赖的场景下。正常场景下，初始化之后返回的bean和实例化的Bean是一样的
                    exposedObject = earlySingletonReference;
                }
```
```

```
            //如果不允许忽略包过的对象并且有对象正在依赖当前的对象。===>报错。。
            else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
                String[] dependentBeans = getDependentBeans(beanName);
                Set<String> actualDependentBeans = new LinkedHashSet<>
(dependentBeans.length);
                for (String dependentBean : dependentBeans) {
                    if
(!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                        actualDependentBeans.add(dependentBean);
                    }
                }
                if (!actualDependentBeans.isEmpty()) {
                    throw new BeanCurrentlyInCreationException
                }
            }
        }
    }

    // Register bean as disposable.
    try {
        //注册disposable bean
        registerDisposableBeanIfNecessary(beanName, bean, mbd);
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Invalid destruction
signature", ex);
    }
    return exposedObject;
}
```

8，populateBean

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable
BeanWrapper bw) {
    //...//
    // Give any InstantiationAwareBeanPostProcessors the opportunity to modify
the
    // state of the bean before properties are set. This can be used, for
example,
    // to support styles of field injection.
    //给了开关支持外界自定义属性填充逻辑
    boolean continueWithPropertyPopulation = true;
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
beanName)) {
                    continueWithPropertyPopulation = false;
                    break;
                }
            }
        }
    }
```

```
    if (!continueWithPropertyPopulation) {
        return;
    }

    PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() :
null);

    int resolvedAutowireMode = mbd.getResolvedAutowireMode();
    if (resolvedAutowireMode == AUTOWIRE_BY_NAME || resolvedAutowireMode ==
AUTOWIRE_BY_TYPE) {
        MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
        // Add property values based on autowire by name if applicable.
        if (resolvedAutowireMode == AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
        }
        // Add property values based on autowire by type if applicable.
        if (resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
        }
        pvs = newPvs;
    }

    boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
    boolean needsDepCheck = (mbd.getDependencyCheck() !=
AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);

    PropertyDescriptor[] filteredPds = null;
    if (hasInstAwareBpps) {
        if (pvs == null) {
            pvs = mbd.getPropertyValues();
        }
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                PropertyValues pvsToUse = ibp.postProcessProperties(pvs,
bw.getWrappedInstance(), beanName);
                if (pvsToUse == null) {
                    if (filteredPds == null) {
                        filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                    }
                    //应用属性的填充。
                    pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
                    if (pvsToUse == null) {
                        return;
                    }
                }
                pvs = pvsToUse;
            }
        }
    }
    if (needsDepCheck) {
        if (filteredPds == null) {
            filteredPds = filterPropertyDescriptorsForDependencyCheck(bw,
mbd.allowCaching);
```

```
            }
            checkDependencies(beanName, mbd, filteredPds, pvs);
        }

        if (pvs != null) {
            applyPropertyValues(beanName, mbd, bw, pvs);
        }
    }
```

`org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor#postProcessProperties` 实现了属性的填充。

9，initializeBean

```
protected Object initializeBean(final String beanName, final Object bean,
@Nullable RootBeanDefinition mbd) {
    //invokeAwareMethods
    invokeAwareMethods(beanName, bean);

    //调用初始化前置操作
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);
    }

    try {
        //调用初始化方法
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            (mbd != null ? mbd.getResourceDescription() : null),
            beanName, "Invocation of init method failed", ex);
    }
   //调用初始化后置操作
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
    }
    return wrappedBean;
}
```

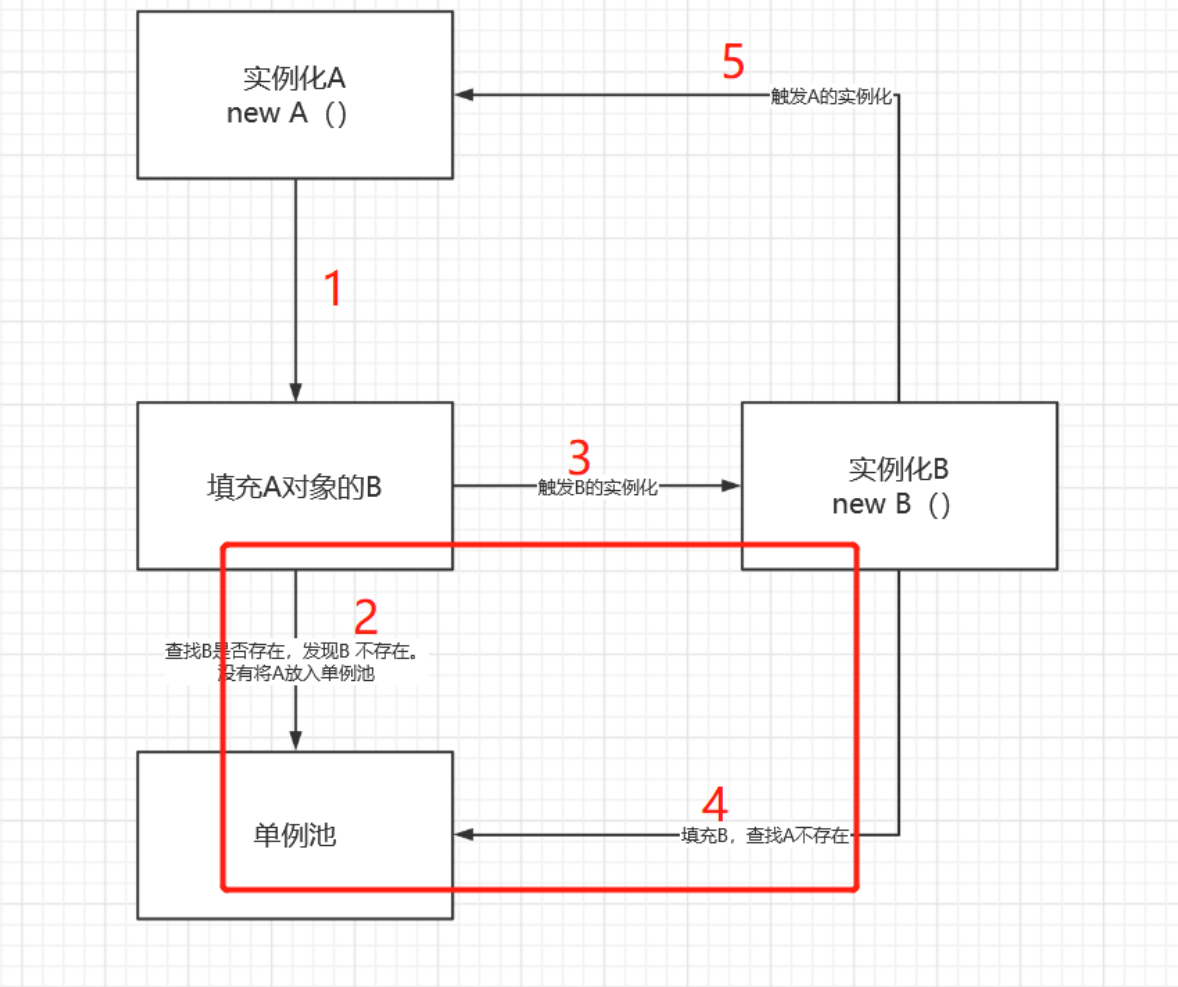
6，Spring循环依赖

提出问题---->分析问题、找出关键点----->解决问题

1，什么是循环依赖?

```
class A{
    B b = new B();
}
class B{
    A a = new A();
```

```
}

//IOC 控制实例化
class A{
    B b;
}
class B{
    A a;
}
```



2，循环依赖问题的关键点?

对象的取用发生在对象存放之前。

解决问题的关键在于：**提前暴露对象**

3，如何解决循环依赖?

解决问题思路

- 不让发生问题

不允许发生循环依赖 springboot 2.6，Spring中的参数控制

- 直接解决问题。

  解决问题的范围。单例的对象并且是非构造方法注入的方式（要有无参的构造方法）

问题场景

非IOC的场景

方式一：

通过在构造方法的引用传递，进行循环依赖。

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

方式二：

实例化之后，互相的set。

IOC的场景

参考手写精简版spring-->spring-mini

4，Spring是如何解决循环依赖的?

解决方式：三级缓存

三级缓存初步认识

```
//一级缓存。完整的（属性完全填充，初始化完成）单例对象
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>
(256);
//二级缓存。不完整的单例对象（属性没有完全填充，对象没有被初始化）
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
//三级缓存。缓存内容是 ObjectFactory
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>
(16);
```

5，流程梳理

从缓存中获取单例对象

4，从单例池中获取对象：getSingleton(beanName)

加锁，缓存的double-check，创建中占位、创建单例、放入缓存

5，getSingleton(beanName, ObjectFactory)

类加载、实例化的前置操作

6，createBean(beanName, mbd, args)

实例化步骤

实例化

元数据处理（MergedBeanDefinitionPostProcessor）

提前暴露对象（对象工厂）

Bean填充

Bean的初始化

返回提前暴露的对象

6，场景源码走读

场景一

场景二，加入循环依赖

场景三，加入动态代理

场景四，仅仅有动态代理（动态代理发生在Bean的初始化阶段）

7，问题汇总

什么是循环依赖?

解决循环依赖的关键点?

如何解决循环依赖问题？分场景

基于构造方法的依赖注入能不能解决循环依赖?

IOC场景下（spring中）原型模式下能不能解决循环依赖?

循环依赖需要几级缓存才能解决问题?

Spring解决循环依赖的三级缓存都有什么作用?

8，画图总结

自行完成

7，Spring Bean生命周期

0，如何接管Bean的实例化?

```
@Component
public class MyBeanPostProcessor implements InstantiationAwareBeanPostProcessor
{
    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String
beanName) throws BeansException {
        //如果此处返回了一个非空对象，说明应用层接管了Bean的实例化
        return null;
    }
}
```

1，如何接管SpringBean的属性填充阶段?

```
public class MyBeanPostProcessor implements InstantiationAwareBeanPostProcessor
{
    @Override
    public boolean postProcessAfterInstantiation(Object bean, String beanName)
throws BeansException {
        //控制开关，告诉spring是否有IOC容器进行属性填充
        //false=属性由外界完成
        //对于bean的属性填充可以此处完成
        return false;
    }
}
```

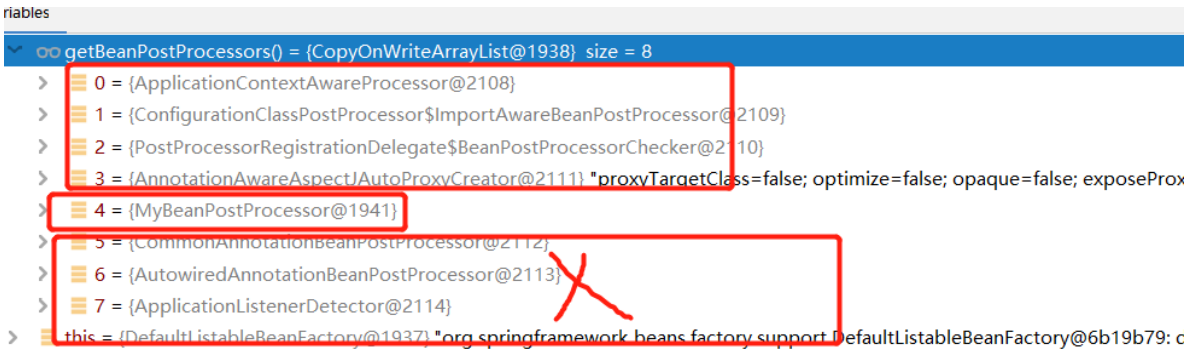
2，如何扩展SpringBean属性的填充?

```
public class MyBeanPostProcessor implements InstantiationAwareBeanPostProcessor
{
    @Override
    public boolean postProcessAfterInstantiation(Object bean, String beanName)
throws BeansException {
        //控制开关，告诉spring是否有IOC容器进行属性填充
        //false=属性填充由外界完成
        //对于bean的属性填充可以此处完成
        return true;
    }

    @Override
    public PropertyValues postProcessProperties(PropertyValues pvs, Object bean,
String beanName) throws BeansException {
        //执行自定义的属性填充实现
        if (bean instanceof OrderService){
            ((OrderService) bean).setTestPopulate("test");
        }
        return pvs;
    }
}
```

3，为什么`@PostConstruct`会失效?

```
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        if (bean instanceof OrderService)
            System.out.println("OrderService postProcessBeforeInitialization");
        //此处返回了空对象，导致BeanPostProcessor的处理链断掉
        return null;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        if (bean instanceof OrderService)
            System.out.println("OrderService postProcessAfterInitialization");
        return null;
    }
}
```

Spring允许自定义的BeanPostProcessor接管基于注解的初始化操作。



4，如何实现一个XXXAware方法?

```
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        if (bean instanceof XXXAware){
            //setXXX
        }
        return bean;
    }
}
```

5，单例加载完成的后置操作

`SmartInitializingSingleton#afterSingletonsInstantiated`

在单例加载完成之后，会统一遍历所有单例，如果是`SmartInitializingSingleton`类型的就会回调该方法。

6，Bean的销毁

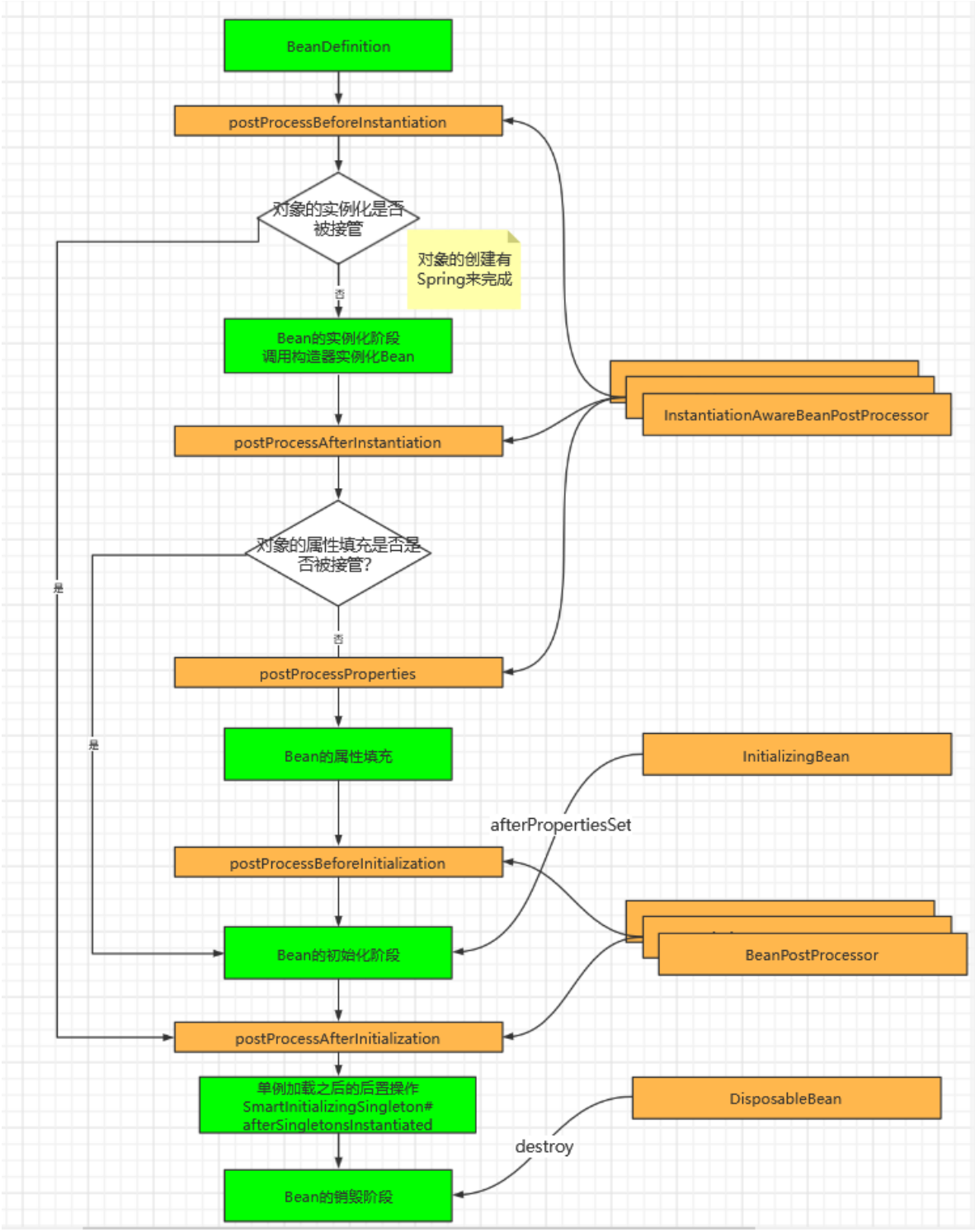
如何使用：

- 实现接口：`org.springframework.beans.factory.DisposableBean`
- 使用注解：`@PreDestroy`

销毁的入口：
`org.springframework.context.support.AbstractApplicationContext#registerShutdownHook`

```
@Override
public void registerShutdownHook() {
    if (this.shutdownHook == null) {
        // No shutdown hook registered yet.
        this.shutdownHook = new Thread(SHUTDOWN_HOOK_THREAD_NAME) {
            @Override
            public void run() {
                synchronized (startupShutdownMonitor) {
                    doClose();
                }
            }
        };
        Runtime.getRuntime().addShutdownHook(this.shutdownHook);
    }
}
```

7，Spring Bean生命周期的完整梳理



8，Spring-AOP

1，建立对于Spring-AOP的全局认识

Spring-AOP 和 AOP的关系

Spring-AOP是Spring对于AOP理念的实现，仅仅是AOP其他实现（AspectJ）的子集实现。

Spring-AOP的核心架构

基于代理的架构。围绕着代理的实现。具有丰富功能的一种代理模式。

Spring中代理的实现：JDK动态代理、CGLIB动态代理。

Spring-AOP架构的逻辑划分

分为两个部分： 第一部分：spring-aop的内核。spring-aop的api集合。

第二部分：spring-aop提供一系列的框架服务集合。提供方便应用程序使用spring-aop 的框架服务。例如；事务管理器。

2，相关概念梳理

基础概念

代理：代理的这种做事情的方式

代理模式：在大量使用代理的方式时形成一种套路、方法论。

aop的概念：面向切面编程（aspect-oriented programming）（专注于切面的编程方式）

aop vs oop : oop 是基础、aop是OOP的补充，没有竞争关系。

AOP中的概念

目标对象（Target）

被代理的对象

代理对象（Proxy）

代理目标对象的对象，实际提供服务的对象（在spring中就是暴露个spring-ioc容器的对象）

切面（Aspect）

属于aop的基本单位，需要开发人员编写。包括需要织入的代码逻辑，以及织入位置。

织入（weaver）

按照织入的时机分为三种：

编译期的织入、加载期（LTW）的织入、运行时的织入。

连接点（Joinpoint）

是应用执行期间明确定义一个点。

通知（Advice）

在连接点处理执行的代码逻辑就是通知。

切入点（Pointcut）

一系列织入代码逻辑的连接点集合。

引入（Introduction）

顾问（Advisor）

由spring提供的概念。持有Advice和一些适应性过滤逻辑。可以理解为spring的切面的体现。控制逻辑的织入。

Spring-AspectJ实例化模型

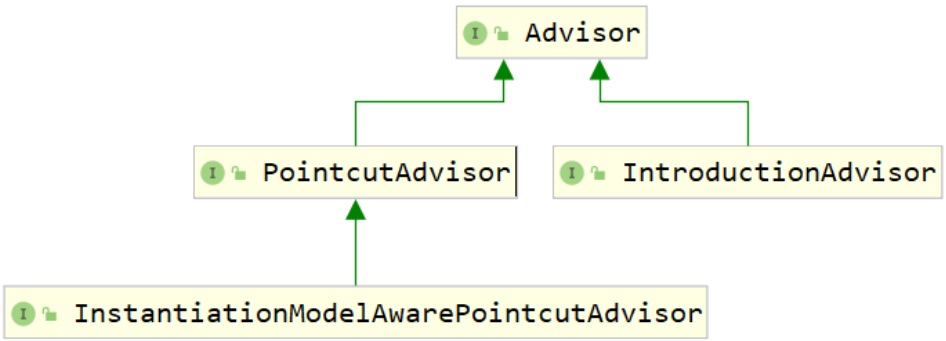
AspectJ实例化模型

3，接口定义以及关键类的解读

org.springframework.aop.Advisor

直接将其理解为Spring中的AOP切面。

```
public interface Advisor {
    Advice getAdvice();
}
```

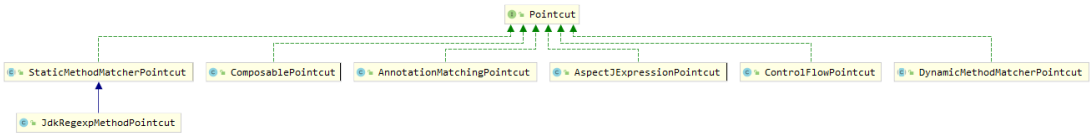


org.springframework.aop.PointcutAdvisor

```
public interface PointcutAdvisor extends Advisor {
    Pointcut getPointcut();
}
```

org.springframework.aop.Pointcut

Spring的内置Pointcut实现类，spring通过提供大量的 内置切入点的实现，大大简化了开发人员的工作。




```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}
```

org.springframework.aop.ClassFilter

目标对象的类型过滤器

org.springframework.aop.MethodMatcher

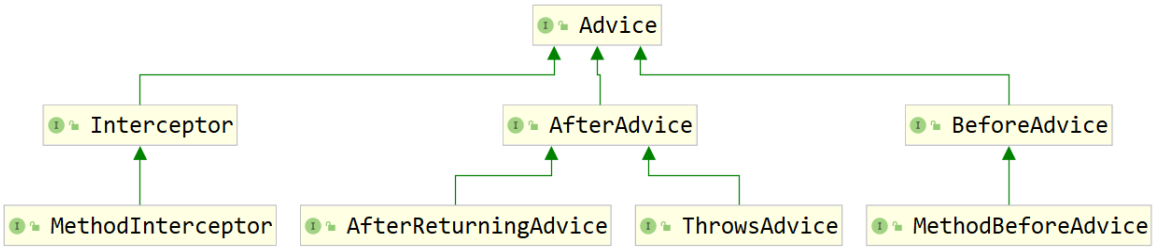
目标对象的目标方法匹配器

方法匹配器的类型：动态的、静态的。

```
public interface MethodMatcher {
    boolean matches(Method method, Class<?> targetClass);
    /**
     * Is this MethodMatcher dynamic, that is, must a final call be made on the
```

```
     * {@link #matches(java.lang.reflect.Method, Class, Object[])} method at
     * runtime even if the 2-arg matches method returns {@code true}?
     * <p>Can be invoked when an AOP proxy is created, and need not be invoked
     * again before each method invocation,
     * @return whether or not a runtime match via the 3-arg
     * {@link #matches(java.lang.reflect.Method, Class, Object[])} method
     * is required if static matching passed
     */
    boolean isRuntime();
    boolean matches(Method method, Class<?> targetClass, Object... args);
}
```

org.aopalliance.aop.Advice



org.aopalliance.intercept.Interceptor

所有的Interceptor都是通知Advice。是环绕通知。

org.aspectj.lang.JoinPoint

连接点，AspectJ中支持丰富的连接点，Spring中仅支持方法调用的连接点。这个是spring的一个简化点。

```
public interface Joinpoint {
    Object proceed() throws Throwable;
    Object getThis();
    AccessibleObject getStaticPart();

}
```

所有的`org.aopalliance.intercept.Invocation`都是一个`JoinPoint`

关键实现：`org.aopalliance.intercept.MethodInvocation`

org.springframework.aop.framework.ProxyFactory

获取代理对象的关键类。

org.springframework.aop.framework.AopProxy

spring的具体的AOP代理的 顶层接口。

关键实现类：

`org.springframework.aop.framework.ObjenesisCglibAopProxy`

`org.springframework.aop.framework.JdkDynamicAopProxy`

org.springframework.aop.framework.AopProxyFactory

封装了具体AopProxy选取的逻辑，负责创建AopProxy的具体实现。

`org.springframework.aop.framework.ProxyFactoryBean`

通过的是spring FactoryBean的方式来创建代理类。

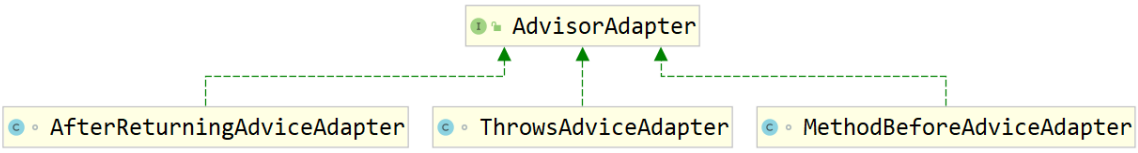
`org.springframework.aop.SpringProxy`

Spring中所有的代理对象都会实现了接口。

4，基于Spring-AOP API的编程体验

Spring在内部AOP代理的实现中仅仅使用了MethodInterceptor，他是同适配接口：`AdvisorAdapter`：

```
public interface AdvisorAdapter {
    boolean supportsAdvice(Advice advice);
    MethodInterceptor getInterceptor(Advisor advisor);
}
```




```
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {
    @Override
    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }

    @Override
    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
        return new MethodBeforeAdviceInterceptor(advice);
    }
}
```

创建一个Advice：

```
public class SimpleAroundAdvice implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println(invocation.getMethod() + "开始执行。。。");
        Object proceed = invocation.proceed();
        System.out.println(invocation.getMethod() + "执行结果：" + proceed);
        return proceed;
    }
}

public class SimpleThrowsAdvice implements ThrowsAdvice {
    public void afterThrowing(Exception e) {
        System.out.println("出现异常1：" + e);
    }
    public void afterThrowing(NullPointerException e) {
        System.out.println("出现异常2：" + e);
```

```
    }

    public void afterThrowing(Method method, Object args, Object target,
Exception e) {
        System.out.println("出现异常3: " + e);
    }
    public void afterThrowing(Method method, Object args, Object target,
NullPointerException e) {
        System.out.println("出现异常4: " + e);
    }
}
```

创建目标类:

```
public class TargetBean {
    public String hello() {
        System.out.println("TargetBean hello...");
        return "hello";
    }

    private String sing() {
        System.out.println("TargetBean sing...");
        return "sing";
    }

    public String throwNPE() {
        throw new NullPointerException("throwNPE");
    }
    public String throwE() throws Exception {
        throw new Exception("throwE");
    }
}
```

注意:

要区分`org.springframework.cglib.proxy.MethodInterceptor`和`org.aopalliance.intercept.MethodInterceptor`，都是属于Springframework中类。

1，基于Advice

```
@Test
public void testAdvice() {
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.setTarget(new TargetBean());
    proxyFactory.addAdvice(new SimpleAroundAdvice());

    TargetBean targetBean = (TargetBean) proxyFactory.getProxy();
    targetBean.hello();
    //执行结果。。
    //        public java.lang.String com.myflx.advice.TargetBean.hello()开始执
行。。。
    //        TargetBean hello...
    //        public java.lang.String com.myflx.advice.TargetBean.hello()执行结
果：hello
}
```

2，基于Advisor（Advice + Pointcut）

```
@Test
public void testAdvisor() throws Exception {
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.setTarget(new TargetBean());

    SimpleAroundAdvice simpleAroundAdvice = new SimpleAroundAdvice();
    NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
    pointcut.addMethodName("hello");

    DefaultPointcutAdvisor defaultPointcutAdvisor = new
DefaultPointcutAdvisor(simpleAroundAdvice);
    defaultPointcutAdvisor.setPointcut(pointcut);

    proxyFactory.addAdvisor(defaultPointcutAdvisor);
    TargetBean targetBean = (TargetBean) proxyFactory.getProxy();
    targetBean.hello();
    targetBean.hello2();
}

@Test
public void testAdvisor2() throws Exception {
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.setTarget(new TargetBean());

    SimpleAroundAdvice simpleAroundAdvice = new SimpleAroundAdvice();
    NameMatchMethodPointcutAdvisor nameMatchMethodPointcutAdvisor = new
NameMatchMethodPointcutAdvisor(simpleAroundAdvice);
    nameMatchMethodPointcutAdvisor.addMethodName("hello");

    proxyFactory.addAdvisor(nameMatchMethodPointcutAdvisor);
    TargetBean targetBean = (TargetBean) proxyFactory.getProxy();
    targetBean.hello();
    targetBean.hello2();
}
```

3，基于Aspectj+ Annotation

依赖:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>
```

使用方式（语法糖）：

```
@EnableAspectJAutoProxy
@Aspect
@Component
public class LogAspect {

    @Before("execution(* com.myflx.dao.OrderDao..*(..))")
    public void log() {
        System.out.println("before aspect.....");
    }
}
```

最终的落地代码：

```
@Test
public void testAdvisor3() throws Exception {
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.setTarget(new TargetBean());

    SimpleAroundAdvice simpleAroundAdvice = new SimpleAroundAdvice();
    AspectJExpressionPointcutAdvisor aspectJExpressionPointcutAdvisor =
        new AspectJExpressionPointcutAdvisor();
    aspectJExpressionPointcutAdvisor.setAdvice(simpleAroundAdvice);
    aspectJExpressionPointcutAdvisor.setExpression("execution(* hello2*(..))");
    proxyFactory.addAdvisor(aspectJExpressionPointcutAdvisor);
    TargetBean targetBean = (TargetBean) proxyFactory.getProxy();
    targetBean.hello();
    targetBean.hello2();
}
```

5，Spring-AOP的源码解读

Spring-AOP的源码解读

ProxyFactory的类结构体系说明

源码解读入口

`org.springframework.aop.framework.ProxyFactory#getProxy()`

```
public Object getProxy() {
    return createAopProxy().getProxy();
}
```

createAopProxy()

```
@Override
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException
{
    //ProxyConfig中允许优化:config.isOptimize()
    //需要代理目标的类型
    //没有显式的设置代理接口 ， 就会 进入优化选择。
    if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new AopConfigException("TargetSource cannot determine target
class: " +
                                         "Either an interface or a target is
required for proxy creation.");
        }
        //如果目标类型是接口或者是已经动态代理的就会继续使用jdk的动态代理
        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
            return new JdkDynamicAopProxy(config);
        }
        return new ObjenesisCglibAopProxy(config);
    }
    else {
        return new JdkDynamicAopProxy(config);
    }
}
```

创建代理对象

使用顶层接口：`org.springframework.aop.framework.AopProxy`

`org.springframework.aop.framework.JdkDynamicAopProxy`

`org.springframework.aop.framework.ObjenesisCglibAopProxy`

创建代理对象：`org.springframework.aop.framework.AopProxy#getProxy()`

Spring容器整合Spring-AOP的源码解读

1，解读入口：`@EnableAspectJAutoProxy`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {
    boolean proxyTargetClass() default false;//是否代理目标类
    boolean exposeProxy() default false;//是否将代理的对象暴露出来。AopContext
}
```

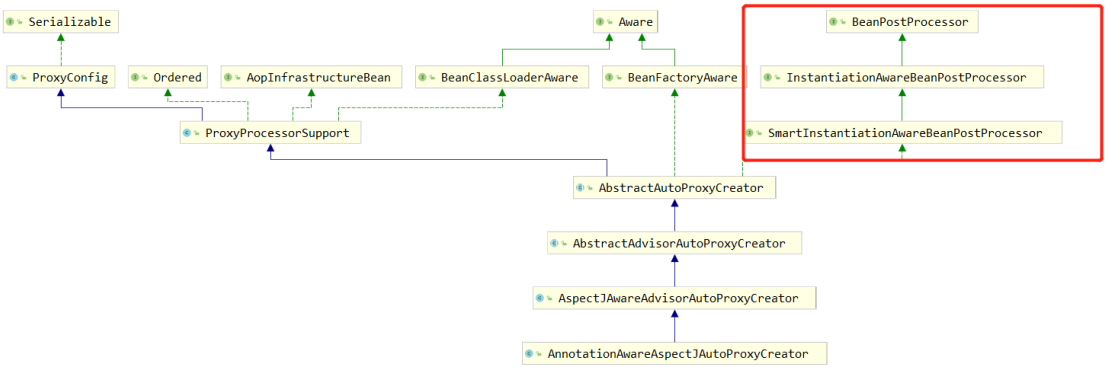
2，注册对象：`AspectJAutoProxyRegistrar`

```
@Override
public void registerBeanDefinitions(
    AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry)
{
    //注册对象AspectJAnnotationAutoProxyCreator

 AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);
    //获取注册属性并填充
    AnnotationAttributes enableAspectJAutoProxy =
        AnnotationConfigUtils.attributesFor(importingClassMetadata,
EnableAspectJAutoProxy.class);
    if (enableAspectJAutoProxy != null) {
        if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
        }
        if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
            AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
        }
    }
}

@Nullable
public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary(
    BeanDefinitionRegistry registry, @Nullable Object source) {
    //AnnotationAwareAspectJAutoProxyCreator的BeanDefinition注册。
    return
registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class,
registry, source);
}
```

3，注册对象AspectJAnnotationAutoProxyCreator

spring容器整合spring-aop的关键类。

类图：



通过BeanPostprocessor这种扩展方式整合spring容器。

基础说明：有SpringBean生命周期的基础，生成代理的核心生命节点是在bean的初始化前后置逻辑。一般情况下代理是初始化之后的后置操作。

4，AbstractAutoProxyCreator#postProcessAfterInitialization

```
@Override
public Object postProcessAfterInitialization(@Nullable Object bean, String
beanName) {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);//构造缓存key
        if (this.earlyProxyReferences.remove(cacheKey) != bean) {//提前暴露对象的代
理引用列表
            //代理的关键逻辑
            return wrapIfNecessary(bean, beanName, cacheKey);//执行代理的逻辑。
        }
    }
    return bean;
}
```

5，wrapIfNecessary

`AbstractAutoProxyCreator#wrapIfNecessary`

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey)
{
    if (StringUtils.hasLength(beanName) &&
this.targetSourcedBeans.contains(beanName)) {
        return bean;
    }
    if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
        return bean;
    }
    if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(),
beanName)) {
        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }

    // Create proxy if we have advice.
    Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
    if (specificInterceptors != DO_NOT_PROXY) {
        this.advisedBeans.put(cacheKey, Boolean.TRUE);
```

```
            /进入了创建代理的逻辑。构建了SingletonTargetSource.
            Object proxy = createProxy(
                bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
```

6，createProxy

`org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#createProxy`

```
protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
            @Nullable Object[] specificInterceptors, TargetSource targetSource)
{

    if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
        AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
this.beanFactory, beanName, beanClass);
    }

    ProxyFactory proxyFactory = new ProxyFactory();
    //复制传入的配置 ProxyConfig
    proxyFactory.copyFrom(this);

    if (!proxyFactory.isProxyTargetClass()) {
        if (shouldProxyTargetClass(beanClass, beanName)) {
            proxyFactory.setProxyTargetClass(true);
        }
        else {
            //处理代理的接口
            evaluateProxyInterfaces(beanClass, proxyFactory);
        }
    }
    //统一封装通知和Advisor为Advisor
    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    proxyFactory.addAdvisors(advisors);
    proxyFactory.setTargetSource(targetSource);
    customizeProxyFactory(proxyFactory);
    //设置代理是否冻结。
    proxyFactory.setFrozen(this.freezeProxy);
    if (advisorsPreFiltered()) {
        proxyFactory.setPreFiltered(true);
    }
    return proxyFactory.getProxy(getProxyClassLoader());
}
```

6，Spring循环依赖中的重复代理问题

1，问题描述

当对象创建时会构建一个ObjectFactory放入第三级缓存池中（a），当发生循环依赖的时候，那么当前的这个对象就会在getSingleton(b)方法中调用ObjectFactory.getObject发生代理(c)，产生一个 代理对象。那么在后续的属性填充阶段会继续进入代理的创建。如果又产生了代理对象那么就会被后续的逻辑检测到抛出异常(d)

(a):addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

(b):getSingleton

```
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    ////=======================////
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

(c) 产生代理：ObjectFactory.getObject

```
protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd,
Object bean) {
    Object exposedObject = bean;
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
                SmartInstantiationAwareBeanPostProcessor ibp =
(SmartInstantiationAwareBeanPostProcessor) bp;
                //此处会产生代理对象。
                exposedObject = ibp.getEarlyBeanReference(exposedObject,
beanName);
            }
        }
    }
    return exposedObject;
}
```

(d)检测暴露对象：

```
if (earlySingletonExposure) {
    Object earlySingletonReference = getSingleton(beanName, false);
    if (earlySingletonReference != null) {
        if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
```

```
        }
        else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<>
(dependentBeans.length);
            for (String dependentBean : dependentBeans) {
                if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                    actualDependentBeans.add(dependentBean);
                }
            }
            if (!actualDependentBeans.isEmpty()) {
                throw new BeanCurrentlyInCreationException(beanName,
                                                           "Bean with name '" +
beanName + "' has been injected into other beans [" +

StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                                                           "] in its raw version
as part of a circular reference, but has eventually been " +
                                                           "wrapped. This means
that said other beans do not use the final version of the " +
                                                           "bean. This is often
the result of over-eager type matching - consider using " +
                                                           "'getBeanNamesOfType'
with the 'allowEagerInit' flag turned off, for example.");
            }
        }
    }
}
```

2，spring-aop是如何解决重复代理的问题

增加提前暴露对象的代理缓存来解决重复代理的问题。

主要是实现了接口：
`org.springframework.beans.factory.config.SmartInstantiationAwareBeanPostProcessor#getEarlyBeanReference` 来解决该问题。

`org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#getEarlyBeanReference`

```
@Override
public Object getEarlyBeanReference(Object bean, String beanName) {
    Object cacheKey = getCacheKey(bean.getClass(), beanName);
    this.earlyProxyReferences.put(cacheKey, bean);//代理前的缓存
    return wrapIfNecessary(bean, beanName, cacheKey);
}


@Override
public Object postProcessAfterInitialization(@Nullable Object bean, String
beanName) {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (this.earlyProxyReferences.remove(cacheKey) != bean) {//说明没有被代理了
            return wrapIfNecessary(bean, beanName, cacheKey);
        }
```

```
    }
    //已经被代理了
    return bean;
}
```

接口定义来源：

```
public interface SmartInstantiationAwareBeanPostProcessor extends
InstantiationAwareBeanPostProcessor {
    @Nullable
    default Class<?> predictBeanType(Class<?> beanClass, String beanName) throws
BeansException {
        return null;
    }

    @Nullable
    default Constructor<?>[] determineCandidateConstructors(Class<?> beanClass,
String beanName)
            throws BeansException {

        return null;
    }

    default Object getEarlyBeanReference(Object bean, String beanName) throws
BeansException {
        return bean;
    }
}
```

3，重现重复代理的问题

1，创建服务构建循环依赖

```
@Service
public class AddressService {

    @Autowired
    private UserService userService;

    public void hello() {
        System.out.println("AddressService hello...");
    }
    @Async
    public void asyncHello() {
        System.out.println("AddressService async hello...");
    }
}

@Service
public class UserService {

    @Autowired
    private AddressService addressService;

    public void hello() {
        System.out.println("UserService hello...");
    }
```

```
    @Async
    public void asyncHello() {
        System.out.println("UserService async hello...");
    }
}
```

2，配置类增加异步注解和方法

`@EnableAsync`

3，增加注解，启动抛出异常

```
Exception in thread "main"
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error
creating bean with name 'addressService': Bean with name 'addressService' has
been injected into other beans [userService] in its raw version as part of a
circular reference, but has eventually been wrapped. This means that said other
beans do not use the final version of the bean. This is often the result of
over-eager type matching - consider using 'getBeanNamesOfType' with the
'allowEagerInit' flag turned off, for example.
    at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doC
reateBean(AbstractAutowireCapableBeanFactory.java:624)
```

为什么@Async放在不同类里边效果不一样?

和启动顺序有关

4，产生原因

重复代理，涉及到@Async的实现原理。

入口：`org.springframework.scheduling.aspectj.AspectJAsyncConfiguration`

5，问题解决

方式1：@Lazy

方式2：

增加依赖：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

改变模式：

`@EnableAsync(mode = AdviceMode.ASPECTJ)`

6，重复代理的问题补充

在spring发生循环依赖时，代理逻辑重复运行，导致了已经被引用的对象二次发生代理。代理逻辑发生两遍：

第一遍代理逻辑：发生在循环依赖中getSingleton()里边的ObjectFactory.getObject，获取第三级缓存中发生的。

第二遍代理逻辑：发生在对象初始化的后置操作中。

spring检测出该对象已经被其他对象引用就抛出异常。

99，阅读源码技能

关于实验环境

要求实验环境尽可能的纯粹，不受其他环境的影响。jar不要有多余的。

关于编码习惯

common sense

方法命名

在很多源码中，实际做核心工作的方法命名为`doxxx`方法，而方法`xxx`是暴露给外界调用的，往往是有很多重载的暴露方法。

关于源码走读

抓住主脉络

结合具体的功能