Spring 初始化过程

1. prepareRefresh 这一步创建和准备了 Environment 对象，

- Environment 的作用之一是为后续@Value，值注入时提供键值

2. obtainFreshBeanFactory 创建或获取 BeanFactory

- BeanFactory 的作用是负责 bean 的创建、依赖注入和初始化
- BeanDefinition 作为 bean 的设计蓝图，规定了 bean 的特征，如单例多例、依赖关系、初始销毁方法等
- BeanDefinition 的来源有多种多样，可以是通过 xml 获得、通过配置类获得、通过组件扫描获得，也可以是编程添加

3.prepareBeanFactory 准备 BeanFactory

- StandardBeanExpressionResolver 来解析 SpEL
- ResourceEditorRegistrar 会注册类型转换器，并应用 ApplicationContext 提供的 Environment 完成 ${ } 解析
- 特殊 bean 指 beanFactory 以及 ApplicationContext，通过 registerResolvableDependency 来注册它们
- ApplicationContextAwareProcessor 用来解析 Aware 接口
- ApplicationListenerDetector 用来识别容器中 ApplicationListener 类型的 bean

4. postProcessBeanFactor 这一步是空实现，留给子类扩展

- 一般 Web 环境的 ApplicationContext 都要利用它注册新的 Scope，完善 Web 下的 BeanFactory
- 体现的是模板方法设计模式

5. invokeBeanFactoryPostProcessors 后处理器扩展 BeanFactory

- beanFactory 后处理器，充当 beanFactory 的扩展点，可以用来补充或修改 BeanDefinition
- ConfigurationClassPostProcessor – 解析 @Configuration、@Bean、@Import、@PropertySource 等
- PropertySourcesPlaceHolderConfigurer – 替换 BeanDefinition 中的 ${ }
- MapperScannerConfigurer – 补充 Mapper 接口对应的 BeanDefinition

6. registerBeanPostProcessors 准备 Bean 后处理器

- bean 后处理器，充当 bean 的扩展点，可以工作在 bean 的实例化、依赖注入、初始化阶段
- AutowiredAnnotationBeanPostProcessor 功能有：解析@Autowired，@Value 注解
- CommonAnnotationBeanPostProcessor 功能有：解析@Resource，@PostConstruct，@PreDestroy
- AnnotationAwareAspectJAutoProxyCreator 功能有：为符合切点的目标 bean 自动创建代理

7. initMessageSource 为 ApplicationContext 提供国际化功能

- 实现国际化
- 容器中一个名为 messageSource 的 bean，如果没有，则提供空的 MessageSource 实现

8. initApplicationEventMulticaster 为 ApplicationContext 提供事件发布器

- 用来发布事件给监听器

- 可以从容器中找名为 applicationEventMulticaster 的 bean 作为事件广播器，若没有，也会新建默认的事件广播器
- 可以调用 ApplicationContext.publishEvent(事件对象) 来发布事件

9. onRefresh 这一步是空实现，留给子类扩展

- SpringBoot 中的子类可以在这里准备 WebServer，即内嵌 web 容器
- 体现的是模板方法设计模式

10. registerListeners 为 ApplicationContext 准备监听器

- 用来接收事件
- 一部分监听器是事先编程添加的、另一部分监听器来自容器中的 bean、还有一部分来自于@EventListener 的解析
- 实现 ApplicationListener 接口，重写其中 onApplicationEvent(E e)方法即可

11. finishBeanFactoryInitialization 初始化单例 Bean，执行 Bean 后处理器扩展

- conversionService 也是一套转换机制，作为对 PropertyEditor 的补充
- 内嵌值解析器用来解析 @Value 中的 ${ }，借用的是 Environment 的功能
- 单例池用来缓存所有单例对象，对象的创建都分三个阶段，每一阶段都有不同的 bean 后处理器参与进来，扩展功能

12. finishRefresh 准备生命周期管理器，发布 ContextRefreshed 事件

- 用来控制容器内需要生命周期管理的 bean
- 如果容器中有名称为 lifecycleProcessor 的 bean 就用它，否则创建默认的生命周期管理器
- 调用 context 的 start，即可触发所有实现 LifeCycle 接口 bean 的 start

- 调用 context 的 stop，即可触发所有实现 LifeCycle 接口 bean 的 stop

```
{
    //step1:prepareRefresh 上下文刷新前准备工作：设置 ConfigurableWebApplicationContext 上下文实例对象 wac 的启动日期和活动标志、加载属性源配置以及判断必填项是否都完整。
    prepareRefresh();

    //step2: obtainFreshBeanFactory 通知子类刷新内部 bean 工厂工作：创建 BeanFactory，如果已有就销毁，没有就创建；核心工作就是解析 XML 以及扫描注解 将扫描到的 Bean 配置属性封装到 BeanDefinition 对象中，并对它 beanName(key)，BeanDefinition(v) 保存到一个 Map 中。
    ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

    //step3:prepareBeanFactory 对 bean factory 进行一些上下文标准化配置：设置 factory 的类加载器、bean 表达式解释器、资源编辑注册器、应用上下文自动注入后处理器、配置在自动装配（通过 beans 标签 default-autowire 属性来依赖注入）的时候的需要忽略
```

的类

(如 ApplicationContextAwareProcessor、EnvironmentAware、ResourceLoaderAware、EmbeddedValueResolverAware、ApplicationEventPublisherAware、MessageSourceAware、ApplicationContextAware）、注册可以自动装配的 Bean（如 BeanFactory）以及注册默认系统变量配置等。

prepareBeanFactory(beanFactory);

//step4: postProcessBeanFactory 修改 BeanFactory 的后处理器配置：在标准初始化之后修改应用程序上下文的内部 bean 工厂初始化配置，

允许注册特殊 BeanPostProcessors->即 ServletContextAwareProcessor 后处理器以及设置自动装配忽略接口类（ServletContextAware、ServletConfigAware）以及注册应用上下文的 request/session 范围。 这一阶段所有 bean 定义都已加载，但没有 bean 将被实例化。

postProcessBeanFactory(beanFactory);

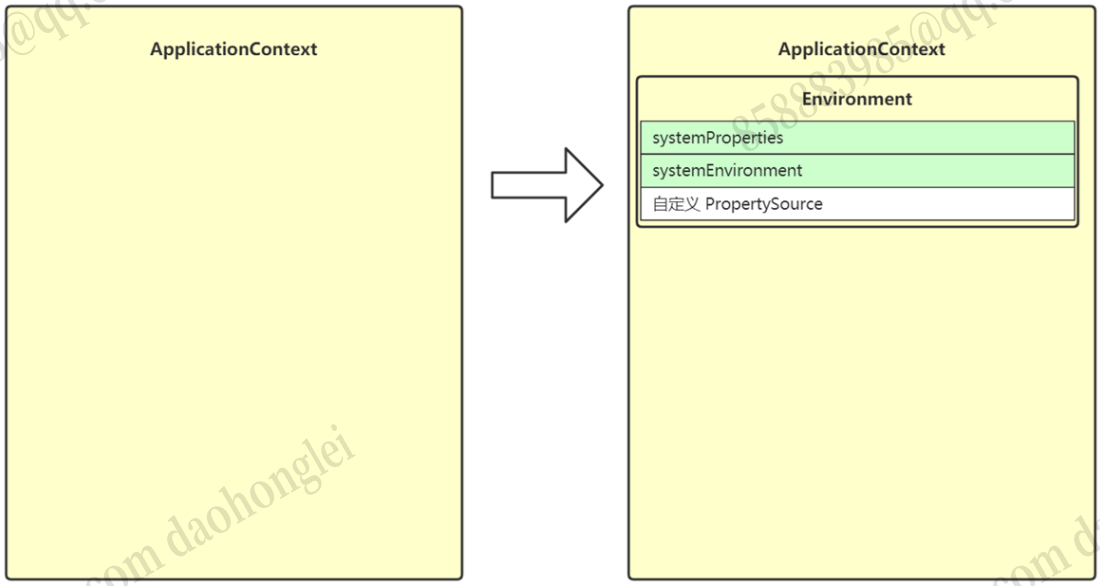
```
    //step5:invokeBeanFactoryPostProcessors 工作：通过显示顺序方式调用手动注册的 BeanFactory 后处理器，先实例化 spring 框架涉及到的后处理器，在调用。

invokeBeanFactoryPostProcessors(beanFactory);

    //step6:registerBeanPostProcessors 注册拦截 bean 创建的 bean 处理器-实例化：
    registerBeanPostProcessors(beanFactory);

    //step7:initMessageSource 初始化消息源：MessageSource 接口类用于支持信息的国际化和包含参数的信息的替换。
    initMessageSource();

    //step8: initApplicationEventMulticaster 初始化事件广播器：如果上下文中没有定义则使用默认广播器：SimpleApplicationEventMulticaster。
    initApplicationEventMulticaster();

    //step9: onRefresh 初始化其他特定的 bean，由
```

```
具体子类实现。
    onRefresh();

    //step10:registerListeners 注册监听事件：在容器中将所有项目里面的 ApplicationListener 注册进来，大体过程如下：
    获取所有的事件，并添加到事件派发器中 -> 监听事件进行派发广播。
    registerListeners();

    //step11: finishBeanFactoryInitialization 初始化所有剩下的单实例 Bean(没有配置赖加载的 lazy!=true)
    finishBeanFactoryInitialization(beanFactory);

    //step12:finishRefresh 完成 BeanFactory 的初始化创建工作：
    finishRefresh();
}
```

1. prepareRefresh

这一步创建和准备了 Environment 对象，

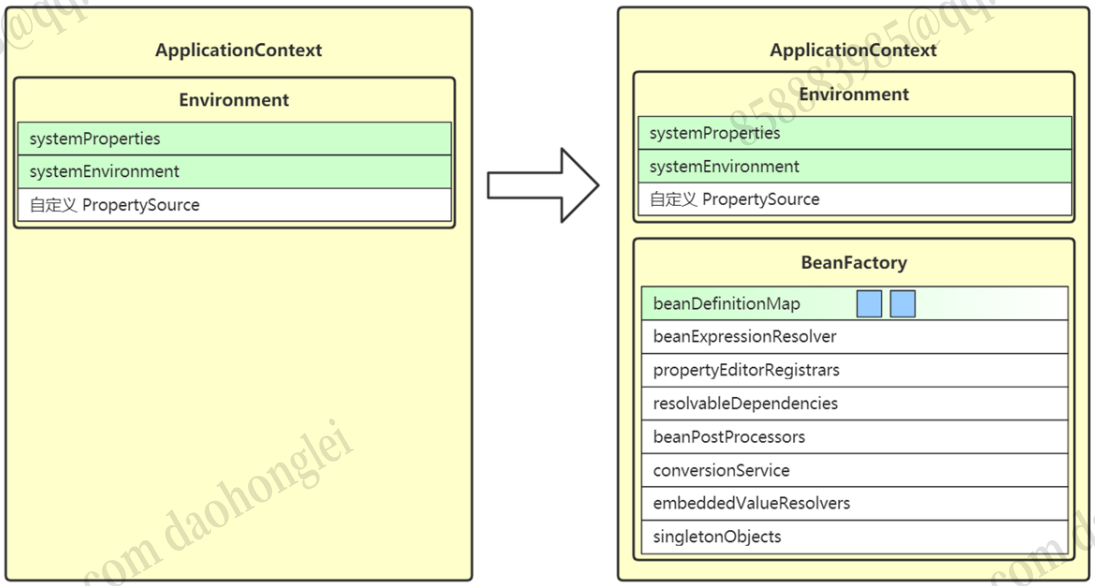
- Environment 的作用之一是为后续@Value，值注入时提供键值



2. obtainFreshBeanFactory

创建或获取 BeanFactory

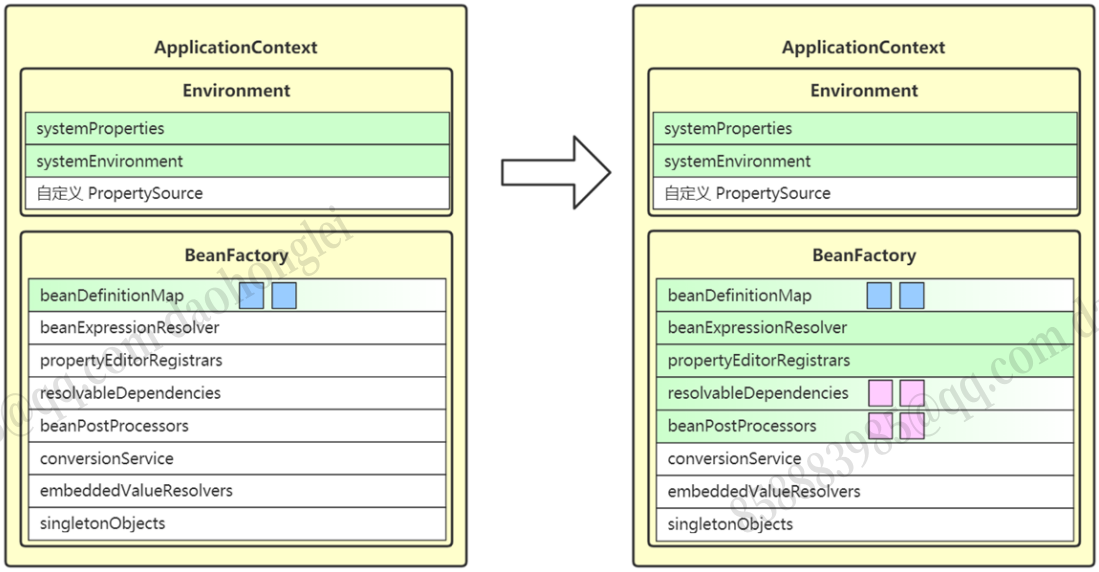
- BeanFactory 的作用是负责 bean 的创建、依赖注入和初始化
- BeanDefinition 作为 bean 的设计蓝图，规定了 bean 的特征，如单例多例、依赖关系、初始销毁方法等
- BeanDefinition 的来源有多种多样，可以是通过 xml 获得、通过配置类获得、通过组件扫描获得，也可以是编程添加



3. prepareBeanFactory

准备 BeanFactory

- StandardBeanExpressionResolver 来解析 SpEL
- ResourceEditorRegistrar 会注册类型转换器，并应用 ApplicationContext 提供的 Environment 完成 ${ } 解析
- 特殊 bean 指 beanFactory 以及 ApplicationContext，通过 registerResolvableDependency 来注册它们
- ApplicationContextAwareProcessor 用来解析 Aware 接口
- ApplicationListenerDetector 用来识别容器中 ApplicationListener 类型的 bean



4. postProcessBeanFactory

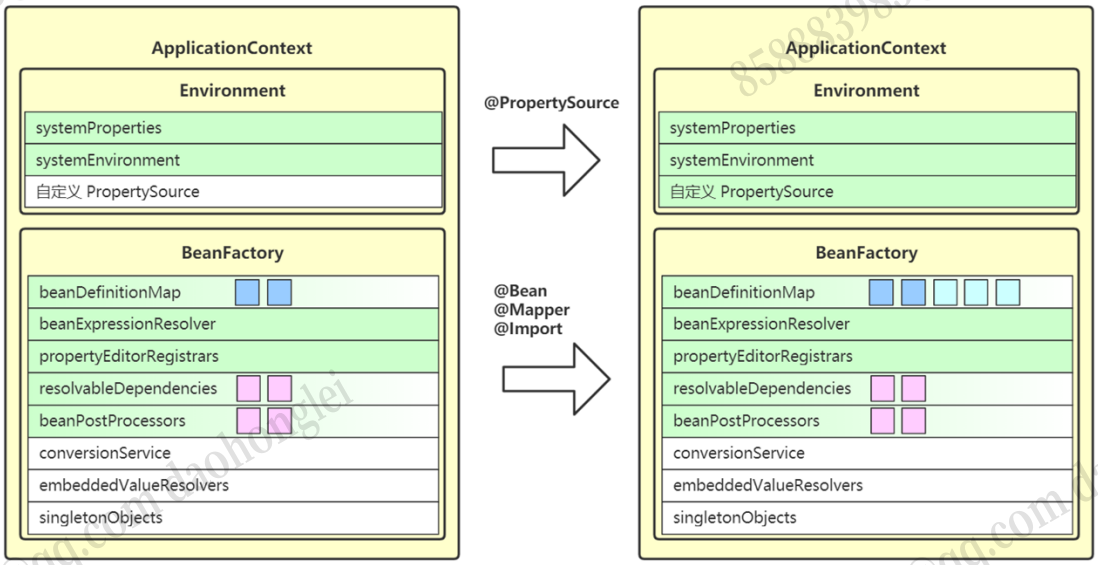
这一步是空实现，留给子类扩展

- 一般 Web 环境的 ApplicationContext 都要利用它注册新的 Scope，完善 Web 下的 BeanFactory
- 体现的是模板方法设计模式

5. invokeBeanFactoryPostProcessors

后处理器扩展 BeanFactory

- beanFactory 后处理器，充当 beanFactory 的扩展点，可以用来补充或修改 BeanDefinition
- ConfigurationClassPostProcessor – 解析 @Configuration、@Bean、@Import、@PropertySource 等
- PropertySourcesPlaceHolderConfigurer – 替换 BeanDefinition 中的 ${}
- MapperScannerConfigurer – 补充 Mapper 接口对应的 BeanDefinition



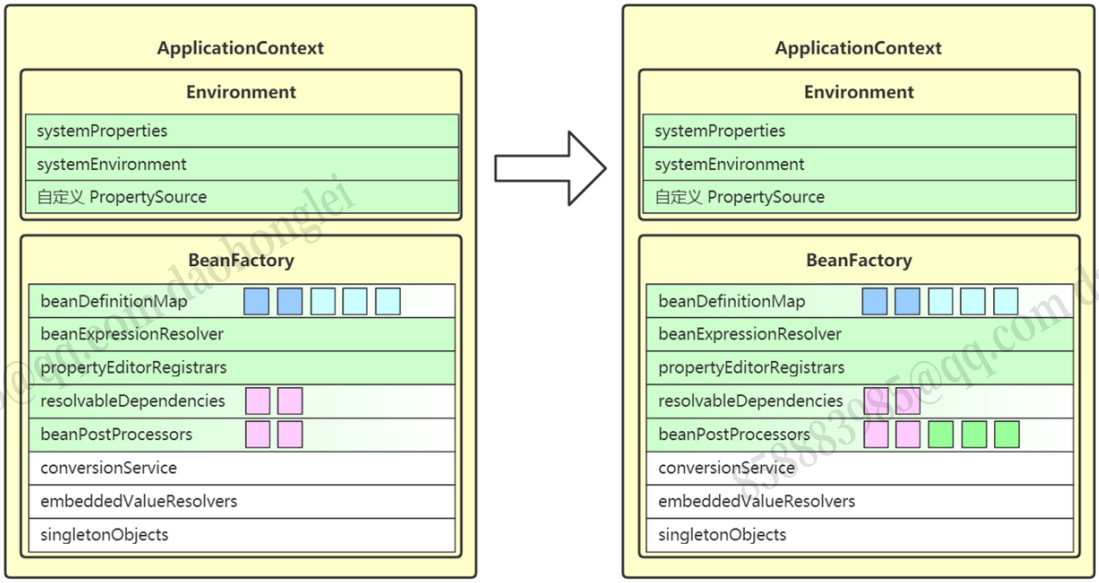
6. registerBeanPostProcessors

准备 Bean 后处理器

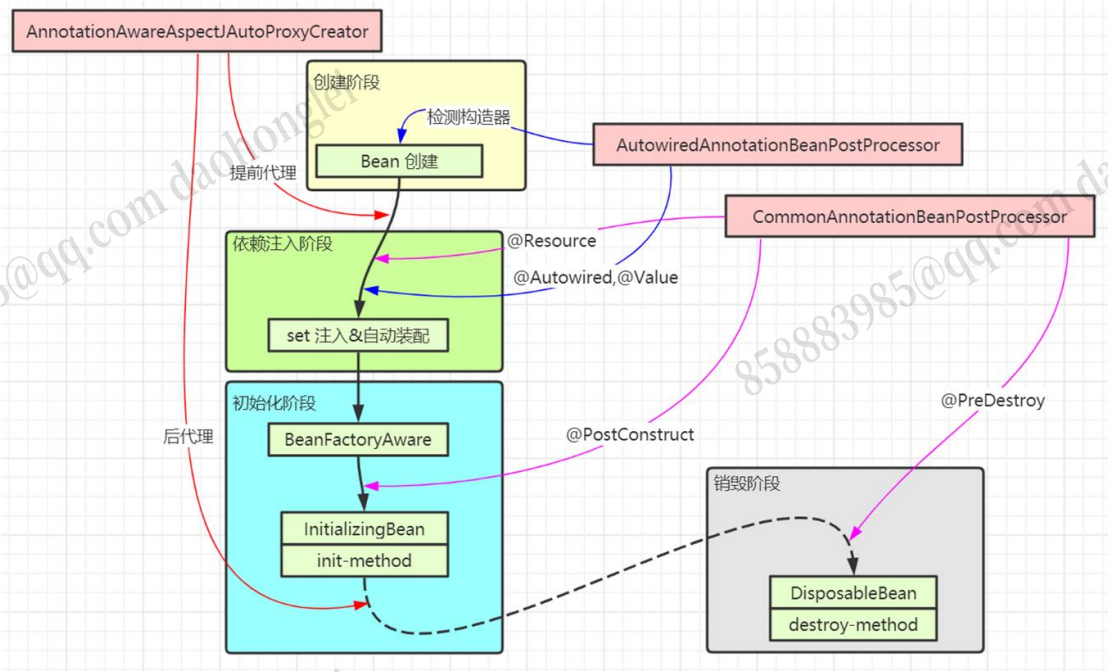
- bean 后处理器，充当 bean 的扩展点，可以工作在 bean 的实例化、依赖注入、初始化阶段
- AutowiredAnnotationBeanPostProcessor 功能有：解析@Autowired，@Value 注解
- CommonAnnotationBeanPostProcessor 功能有：解析@Resource，

@PostConstruct，@PreDestroy

- AnnotationAwareAspectJAutoProxyCreator 功能有：为符合切点的目标 bean 自动创建代理



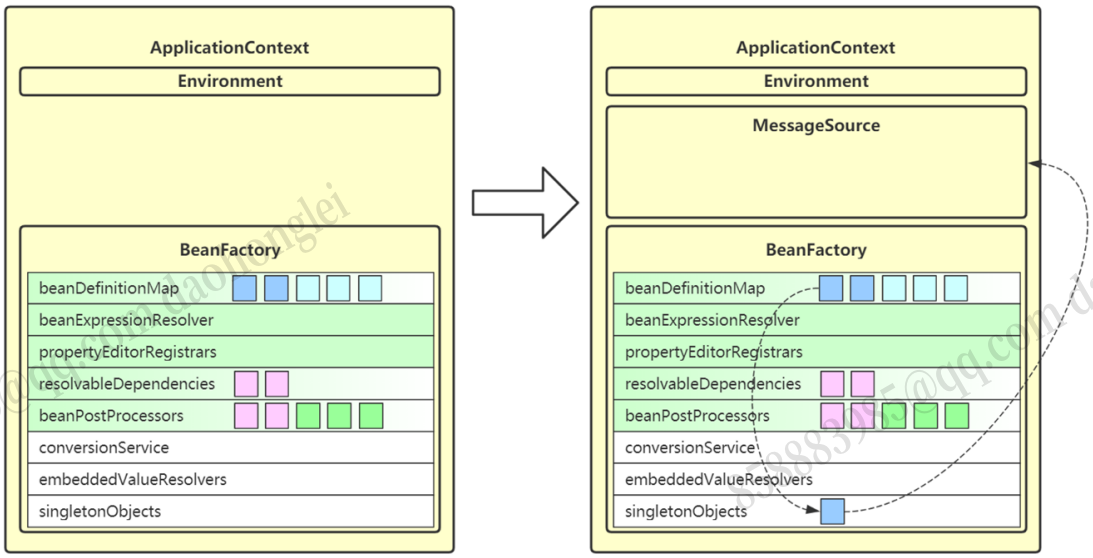
常见后处理器的执行时机



7. initMessageSource

为 ApplicationContext 提供国际化功能

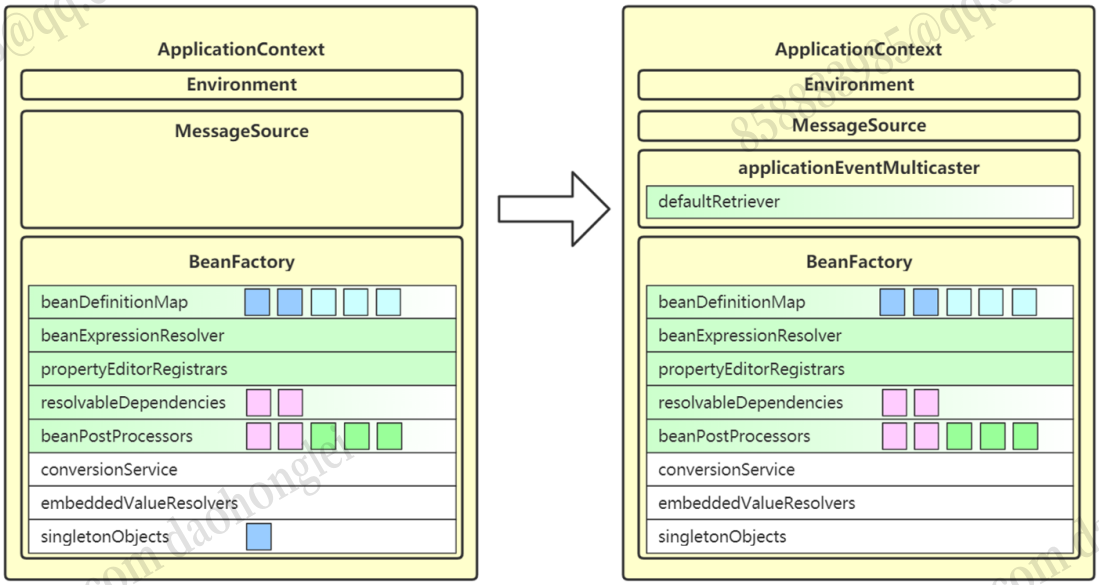
- 实现国际化
- 容器中一个名为 messageSource 的 bean，如果没有，则提供空的 MessageSource 实现



8. initApplicationEventMulticaster

为 ApplicationContext 提供事件发布器

- 用来发布事件给监听器
- 可以从容器中找名为 applicationEventMulticaster 的 bean 作为事件广播器，若没有，也会新建默认的事件广播器
- 可以调用 ApplicationContext.publishEvent(事件对象) 来发布事件



9. onRefresh

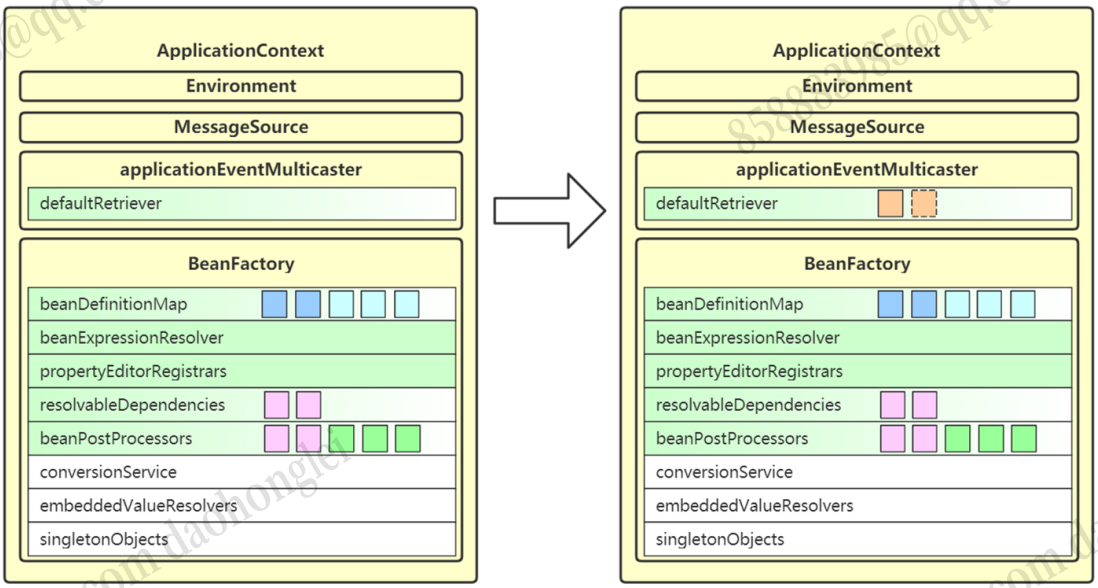
这一步是空实现，留给子类扩展

- SpringBoot 中的子类可以在这里准备 WebServer，即内嵌 web 容器
- 体现的是模板方法设计模式

10. registerListeners

为 ApplicationContext 准备监听器

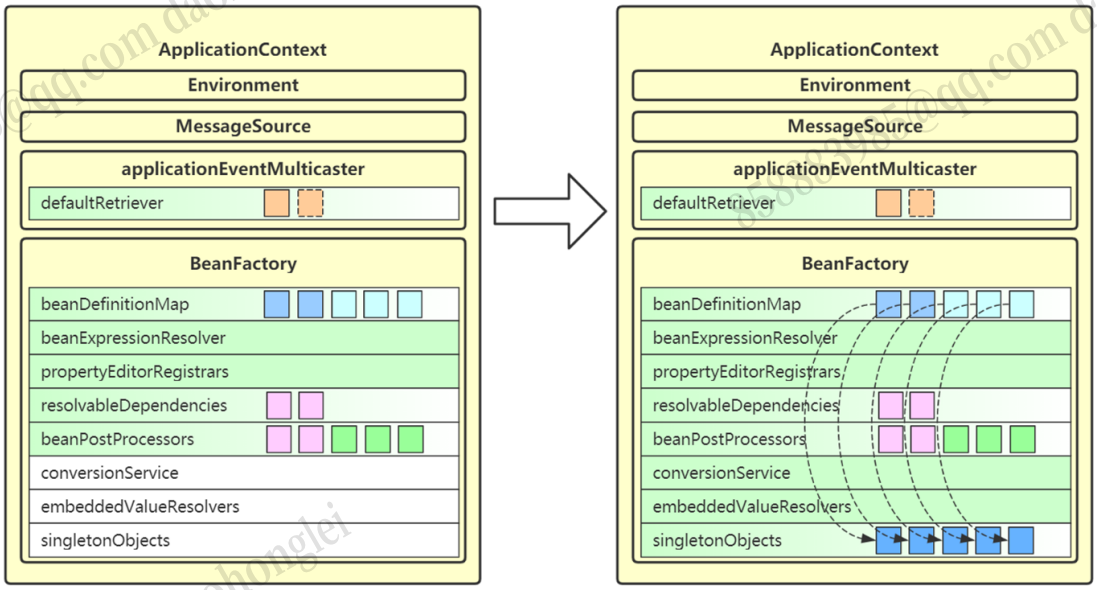
- 用来接收事件
- 一部分监听器是事先编程添加的、另一部分监听器来自容器中的 bean、还有一部分来自于@EventListener 的解析
- 实现 ApplicationListener 接口，重写其中 onApplicationEvent(E e)方法即可



11. finishBeanFactoryInitialization

初始化单例 Bean，执行 Bean 后处理器扩展

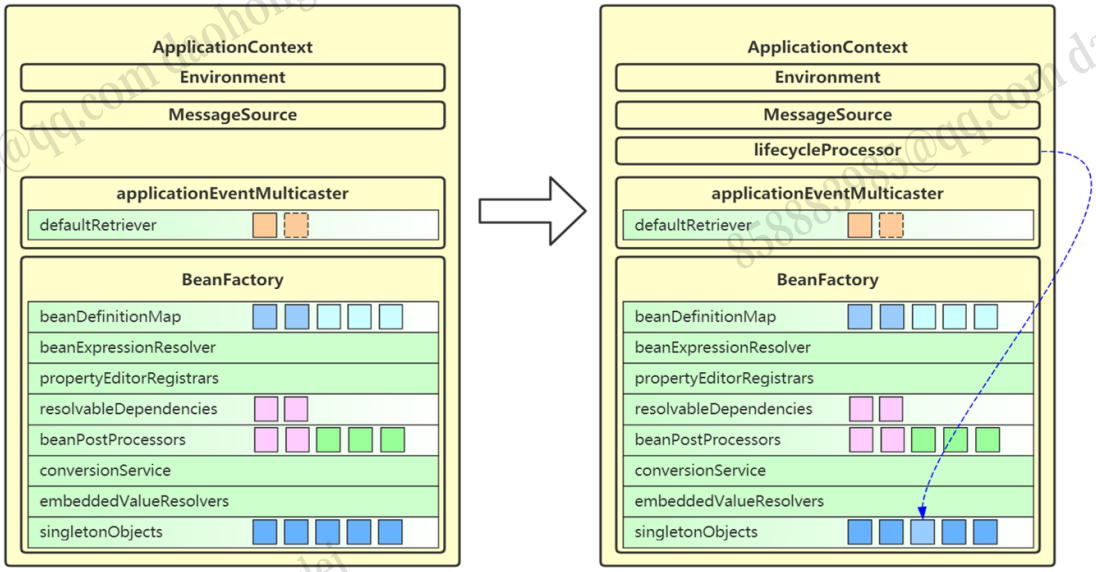
- conversionService 也是一套转换机制，作为对 PropertyEditor 的补充
- 内嵌值解析器用来解析 @Value 中的 ${}，借用的是 Environment 的功能
- 单例池用来缓存所有单例对象，对象的创建都分三个阶段，每一阶段都有不同的 bean 后处理器参与进来，扩展功能



12. finishRefresh

准备生命周期管理器，发布 ContextRefreshed 事件

- 用来控制容器内需要生命周期管理的 bean
- 如果容器中有名称为 lifecycleProcessor 的 bean 就用它，否则创建默认的生命周期管理器
- 调用 context 的 start，即可触发所有实现 LifeCycle 接口 bean 的 start
- 调用 context 的 stop，即可触发所有实现 LifeCycle 接口 bean 的 stop



SpringBoot 的启动指定参数

```
#!/bin/bash

nohup java -Dserver.tomcat.threads.min-spare=100 -Xmx500m -Xss64m -

XX:+PrintGCDetails  -jar demo.jar > log.txt & tail -f log.txt
```

SpringBoot 的启动流程

1. 创建一个新的实例，这个应用程序的上下文将要从指定的来源加载 Bean
   a) 推断当前 WEB 应用类型，一共有三种：NONE,SERVLET,REACTIVE
   b) 设置监听器,从"META-INF/spring.factories"读取 ApplicationListener 类的实例名称集合并去重，并进行 set 去重。（一共 11 个）

c) 推断主入口应用类，通过当前调用栈，获取 Main 方法所在类，并赋值给 mainApplicationClass

2. 创建并启动计时监控类
3. 声明应用上下文和异常报告集合
4. 设置系统属性“java.awt.headless”的值，默认为 true，用于运行 headless 服务器，进行简单的图像处理，多用于在缺少显示屏、键盘或者鼠标时的系统配置，很多监控工具如 jconsole 需要将该值设置为 true
5. 创建所有 spring 运行监听器并发布应用启动事件，简单说的话就是获取 SpringApplicationRunListener 类型的实例（EventPublishingRunListener 对象），并封装进 SpringApplicationRunListeners 对象，然后返回这个 SpringApplicationRunListeners 对象。说的再简单点，getRunListeners 就是准备好了运行时监听器 EventPublishingRunListener。
6. 初始化默认应用参数类
7. 根据运行监听器和应用参数来准备 spring 环境
8. 创建 banner 打印类
9. 创建应用上下文，可以理解为创建一个容器
10. 准备异常报告器，用来支持报告关于启动的错误
11. 准备应用上下文，该步骤包含一个非常关键的操作，将启动类注入容器，为后续开启自动化提供基础
12. 刷新应用上下文
13. 应用上下文刷新后置处理，做一些扩展功能
14. 停止计时监控类

15. 打印日志记录执行主类名、时间信息

16. 发布应用上下文刷新完成监听事件

17. 执行所有的 Runner 运行器

18. 发布应用上下文就绪事件

19. 返回应用上下文

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
    //资源初始化资源加载器，默认为 null
    this.resourceLoader = resourceLoader;
    //断言主要加载资源类不能为 null，否则报错
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //初始化主要加载资源类集合并去重
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    //推断当前 WEB 应用类型，一共有三种：NONE,SERVLET,REACTIVE
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    //设置应用上线文初始化器,从"META-INF/spring.factories"读取
ApplicationContextInitializer 类的实例名称集合并去重，并进行 set 去重。（一共 7 个）
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
    //设置监听器,从"META-INF/spring.factories"读取 ApplicationListener 类的实例名称
集合并去重，并进行 set 去重。（一共 11 个）
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
```

```
    //推断主入口应用类，通过当前调用栈，获取 Main 方法所在类，并赋值给
mainApplicationClass
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

springboot 启动的运行方法，可以看到主要是各种运行环境的准备工作

```
public ConfigurableApplicationContext run(String... args) {
        //1、创建并启动计时监控类
        StopWatch stopWatch = new StopWatch();
stopWatch.start();
        //2、声明化应用上下文和异常报告集合
        ConfigurableApplicationContext context = null;
        Collection<SpringBootExceptionReporter> exceptionReporters = new
ArrayList<>();
        //3、设置系统属性“java.awt.headless”的值，默认为 true，用于运行 headless
服务器，进行简单的图像处理，多用于在缺少显示屏、键盘或者鼠标时的系统配置，很多
监控工具如 jconsole 需要将该值设置为 true
        configureHeadlessProperty();
        //4、创建所有 spring 运行监听器并发布应用启动事件，简单说的话就是获取
SpringApplicationRunListener 类型的实例（EventPublishingRunListener 对象），并
封装进 SpringApplicationRunListeners 对象，
        //然后返回这个 SpringApplicationRunListeners 对象。说的再简单点，
getRunListeners 就是准备好了运行时监听器 EventPublishingRunListener。
```

```
        SpringApplicationRunListeners listeners = getRunListeners(args);
listeners.starting();
 try {
            //5、初始化默认应用参数类
ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
            //6、根据运行监听器和应用参数来准备 spring 环境
ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
            //将要忽略的 bean 的参数打开
configureIgnoreBeanInfo(environment);
            //7、创建 banner 打印类
Banner printedBanner = printBanner(environment);
            //8、创建应用上下文，可以理解为创建一个容器
context = createApplicationContext();
            //9、准备异常报告器，用来支持报告关于启动的错误
exceptionReporters =
getSpringFactoriesInstances(SpringBootExceptionReporter.class,
new Class[] { ConfigurableApplicationContext.class }, context);
            //10、准备应用上下文，该步骤包含一个非常关键的操作，将启动类注入容器，
为后续开启自动化提供基础
prepareContext(context, environment, listeners, applicationArguments,
```

```
printedBanner);
            //11、刷新应用上下文
refreshContext(context);
            //12、应用上下文刷新后置处理，做一些扩展功能
afterRefresh(context, applicationArguments);
            //13、停止计时监控类
stopWatch.stop();
            //14、打印日志记录执行主类名、时间信息
if (this.logStartupInfo) {
new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
stopWatch);
}
            //15、发布应用上下文刷新完成监听事件
listeners.started(context);
            //16、执行所有的 Runner 运行器
callRunners(context, applicationArguments);
 }catch (Throwable ex) {
handleRunFailure(context, ex, exceptionReporters, listeners);
throw new IllegalStateException(ex);
 }
 try {
            //17、发布应用上下文就绪事件
```

```
listeners.running(context);
        }catch (Throwable ex) {
handleRunFailure(context, ex, exceptionReporters, null);
throw new IllegalStateException(ex);
 }
        //18、返回应用上下文
 return context;
}
```

准备应用上下文

```
prepareContext(context, environment, listeners, applicationArguments,
printedBanner);


private void prepareContext(ConfigurableApplicationContext context,
ConfigurableEnvironment environment,
SpringApplicationRunListeners listeners, ApplicationArguments
applicationArguments, Banner printedBanner) {
    //应用上下文的 environment
context.setEnvironment(environment);
    //应用上下文后处理
postProcessApplicationContext(context);
    //为上下文应用所有初始化器，执行容器中的
applicationContextInitializer(spring.factories 的实例)，将所有的初始化对象放置到
```

```
context 对象中
applyInitializers(context);
    //触发所有 SpringApplicationRunListener 监听器的 ContextPrepared 事件方法。添
加所有的事件监听器
listeners.contextPrepared(context);
    //记录启动日志
if (this.logStartupInfo) {
logStartupInfo(context.getParent() == null);
logStartupProfileInfo(context);
}
// 注册启动参数 bean，将容器指定的参数封装成 bean，注入容器
ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
beanFactory.registerSingleton("springApplicationArguments",
applicationArguments);
    //设置 banner
if (printedBanner != null) {
beanFactory.registerSingleton("springBootBanner", printedBanner);
}
if (beanFactory instanceof DefaultListableBeanFactory) {
((DefaultListableBeanFactory) beanFactory)
.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
}
```

```
if (this.lazyInitialization) {
context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
}
// 加载所有资源，指的是启动器指定的参数
Set<Object> sources = getAllSources();
Assert.notEmpty(sources, "Sources must not be empty");
    //将 bean 加载到上下文中
load(context, sources.toArray(new Object[0]));
    //触发所有 springapplicationRunListener 监听器的 contextLoaded 事件方法，
listeners.contextLoaded(context);
}
-------------------
//这里没有做任何的处理过程，因为 beanNameGenerator 和 resourceLoader 默认为空，
可以方便后续做扩展处理
protected void postProcessApplicationContext(ConfigurableApplicationContext
context) {
if (this.beanNameGenerator != null) {
context.getBeanFactory().registerSingleton(AnnotationConfigUtils.CONFIGURATIO
N_BEAN_NAME_GENERATOR,
this.beanNameGenerator);
}
```

```
if (this.resourceLoader != null) {
if (context instanceof GenericApplicationContext) {
((GenericApplicationContext) context).setResourceLoader(this.resourceLoader);
}
if (context instanceof DefaultResourceLoader) {
((DefaultResourceLoader)
context).setClassLoader(this.resourceLoader.getClassLoader());
}
}
if (this.addConversionService) {
context.getBeanFactory().setConversionService(ApplicationConversionService.getS
haredInstance());
}
}
----------
//将启动器类加载到 spring 容器中，为后续的自动化配置奠定基础，之前看到的很多注解
也与此相关
protected void load(ApplicationContext context, Object[] sources) {
if (logger.isDebugEnabled()) {
logger.debug("Loading source " +
StringUtils.arrayToCommaDelimitedString(sources));
}
```

```
BeanDefinitionLoader loader =
createBeanDefinitionLoader(getBeanDefinitionRegistry(context), sources);
if (this.beanNameGenerator != null) {
loader.setBeanNameGenerator(this.beanNameGenerator);
}
if (this.resourceLoader != null) {
loader.setResourceLoader(this.resourceLoader);
}
if (this.environment != null) {
loader.setEnvironment(this.environment);
}
loader.load();
}
----------------------
//springboot 会优先选择 groovy 加载方式，找不到在选择 java 方式
private int load(Class<?> source) {
if (isGroovyPresent() &&
GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
// Any GroovyLoaders added in beans{} DSL can contribute beans here
GroovyBeanDefinitionSource loader = BeanUtils.instantiateClass(source,
GroovyBeanDefinitionSource.class);
load(loader);
```

```
}
if (isComponent(source)) {
this.annotatedReader.register(source);
return 1;
}
return 0;
}
```

刷新应用上下文

```
refreshContext(context);


private void refreshContext(ConfigurableApplicationContext context) {
refresh(context);
if (this.registerShutdownHook) {
try {
context.registerShutdownHook();
}
catch (AccessControlException ex) {
// Not allowed in some environments.
}
}
}
```

```
-----------
public void refresh() throws BeansException, IllegalStateException {
   synchronized (this.startupShutdownMonitor) {
      // Prepare this context for refreshing.
      //刷新上下文环境，初始化上下文环境，对系统的环境变量或者系统属性进行准备和
校验
prepareRefresh();

// Tell the subclass to refresh the internal bean factory.
      //初始化 beanfactory，解析 xml，相当于之前的 xmlBeanfactory 操作
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();


       // Prepare the bean factory for use in this context.
       //为上下文准备 beanfactory，对 beanFactory 的各种功能进行填充，如
@autowired，设置 spel 表达式解析器，设置编辑注册器，添加
applicationContextAwareprocessor 处理器等等
 prepareBeanFactory(beanFactory);
      try {
// Allows post-processing of the bean factory in context subclasses.
            //提供子类覆盖的额外处理，即子类处理自定义的 beanfactorypostProcess
postProcessBeanFactory(beanFactory);
```

```
// Invoke factory processors registered as beans in the context.
            //激活各种 beanfactory 处理器
invokeBeanFactoryPostProcessors(beanFactory);


       // Register bean processors that intercept bean creation.
            //注册拦截 bean 创建的 bean 处理器，即注册 beanPostProcessor
registerBeanPostProcessors(beanFactory);

// Initialize message source for this context.
            //初始化上下文中的资源文件如国际化文件的处理
initMessageSource();


      // Initialize event multicaster for this context.
            //初始化上下文事件广播器
initApplicationEventMulticaster();


// Initialize other special beans in specific context subclasses.
            //给子类扩展初始化其他 bean
onRefresh();


// Check for listener beans and register them.
            //在所有的 bean 中查找 listener bean,然后 注册到广播器中
```

```
registerListeners();


// Instantiate all remaining (non-lazy-init) singletons.
            //初始化剩余的非懒惰的 bean，即初始化非延迟加载的 bean
finishBeanFactoryInitialization(beanFactory);


// Last step: publish corresponding event.
            //发完成刷新过程，通知声明周期处理器刷新过程，同时发出
ContextRefreshEvent 通知别人
finishRefresh();
        }catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
"cancelling refresh attempt: " + ex);
            }
    // Destroy already created singletons to avoid dangling resources.
    destroyBeans();
    // Reset 'active' flag.
    cancelRefresh(ex);
    // Propagate exception to caller.
     throw ex;
}finally {
```

```
// Reset common introspection caches in Spring's core, since we
// might not ever need metadata for singleton beans anymore...
resetCommonCaches();
}
    }
}
```

BeanDefinitionRegistryPostProcessor 后置处理器

BeanDefinitionRegistryPostProcessor 接口继承 BeanFactoryPostProcessor 接口，在所有 bean 定义信息将要被加载，bean 实例还没有创建的时候执行。

BeanDefinitionRegistry 是 bean 定义信息的保存中心，以后 BeanFactory 就是按照 BeanDefinitionRegistry 里面保存的每一个 bean 定义信息创建 bean 实例

BeanDefinitionRegistryPostProcessor 会在 BeanFactoryPostProcessor 之前执行，可以利用 BeanDefinitionRegistryPostProcessor 给容器中添加一些其它组件

自定义 MyBeanDefinitionRegistryPostProcessor 组件，实现 BeanDefinitionRegistryPostProcessor 接口

```
@Component
public class MyBeanDefinitionRegistryPostProcessor implements
BeanDefinitionRegistryPostProcessor {
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException {
```

```
System.out.println("MyBeanDefinitionRegistryPostProcessor...postProcessBeanDefinitionRegistry 方法 " +
                "Bean 的数量: " + registry.getBeanDefinitionCount());

        //手动注册一些 bean 定义信息
        RootBeanDefinition beanDefinition = new RootBeanDefinition(Girl.class);
        registry.registerBeanDefinition("helloGirl",beanDefinition);

        AbstractBeanDefinition beanDefinition1 = BeanDefinitionBuilder.rootBeanDefinition(Girl.class).getBeanDefinition();
        registry.registerBeanDefinition("byeGirl",beanDefinition1);
    }

    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {

System.out.println("MyBeanDefinitionRegistryPostProcessor...postProcessBeanFactory 方法 " +
                "Bean 的数量: " + beanFactory.getBeanDefinitionCount());
```

```
    }
}
```

Spring Bean 的生命周期：

1）、指定初始化和销毁方法； 通过@Bean 指定 init-method 和 destroy-method；

2）、通过让 Bean 实现 InitializingBean（定义初始化逻辑），DisposableBean（定义销毁逻辑）；

3）、可以使用 JSR250；

@PostConstruct：在 bean 创建完成并且属性赋值完成；来执行初始化方法

@PreDestroy：在容器销毁 bean 之前通知我们进行清理工作

4）、BeanPostProcessor【interface】：bean 的后置处理器；

在 bean 初始化前后进行一些处理工作；

postProcessBeforeInitialization:在初始化之前工作

postProcessAfterInitialization:在初始化之后工作

创建

a) Bean 的建立， 由 BeanFactory 读取 Bean 定义文件，并生成各个实例

b) Setter 注入，执行 Bean 的属性依赖注入

初始化

c) BeanNameAware 的 setBeanName(), 如果实现该接口，则执行其 setBeanName 方法

d) BeanFactoryAware 的 setBeanFactory()，如果实现该接口，则执行其 setBeanFactory 方法

e) BeanPostProcessor 的 processBeforeInitialization()，如果有关联的 processor，则在 Bean 初始化之前都会执行这个实例的 processBeforeInitialization()方法

f) InitializingBean 的 afterPropertiesSet()，如果实现了该接口，则执行其 afterPropertiesSet()方法

g) Bean 定义文件中定义 init-method

h) BeanPostProcessors 的 processAfterInitialization()，如果有关联的 processor，则在 Bean 初始化之前都会执行这个实例的 processAfterInitialization()方法

销毁

i) DisposableBean 的 destroy()，在容器关闭时，如果 Bean 类实现了该接口，则执行它的 destroy()方法

j) Bean 定义文件中定义 destroy-method，在容器关闭时，可以在 Bean 定义文件中使用“destory-method”定义的方法

简单回答 springbean 生命周期：

（1）实例化（必须的）构造函数构造对象

（2）装配（可选的）为属性赋值

（3）回调（可选的）（容器-控制类和组件-回调类）

（4）初始化(init-method=" ")

（5）就绪

（6）销毁（destroy-method=" "）

```
public class LiftTest implements BeanNameAware , BeanFactoryAware ,
InitializingBean , DisposableBean , BeanPostProcessor {
    public void initMethod(){
        System.out.println("initMethod");
    }
    public void destroyMethod(){
        System.out.println("destroyMethod");
    }

    @Override
    public void setBeanName(String name) {
        System.out.println(name);
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println(beanFactory.toString());
    }

    @Override
    public void afterPropertiesSet() throws Exception {
```

```
        System.out.println("afterPropertiesSet");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("destroy");
    }

    @PostConstruct
    public void postConstruct(){
        System.out.println("postConstruct");
    }

    @PreDestroy
    public void preDestroy(){
        System.out.println("preDestroy");
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        return BeanPostProcessor.super.postProcessBeforeInitialization(bean,
```

```
beanName);
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        return BeanPostProcessor.super.postProcessAfterInitialization(bean,
beanName);
    }
}


liftTest
org.springframework.beans.factory.support.DefaultListableBeanFactory@30ea8c23:
defining beans
[org.springframework.context.annotation.internalConfigurationAnnotationProcess
or,org.springframework.context.annotation.internalAutowiredAnnotationProcessor
,org.springframework.context.annotation.internalCommonAnnotationProcessor,or
g.springframework.context.event.internalEventListenerProcessor,org.springframew
ork.context.event.internalEventListenerFactory,demo2Application,org.springframe
work.boot.autoconfigure.internalCachingMetadataReaderFactory,liftTest,org.sprin
gframework.boot.autoconfigure.AutoConfigurationPackages,org.springframework.
aop.config.internalAutoProxyCreator,org.springframework.boot.autoconfigure.con
text.PropertyPlaceholderAutoConfiguration,propertySourcesPlaceholderConfigure
```

r,org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAut
oConfiguration$TomcatWebSocketConfiguration,websocketServletWebServerCust
omizer,org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServ
letAutoConfiguration,org.springframework.boot.autoconfigure.web.servlet.Servlet
WebServerFactoryConfiguration$EmbeddedTomcat,tomcatServletWebServerFacto
ry,org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryA
utoConfiguration,servletWebServerFactoryCustomizer,tomcatServletWebServerFac
toryCustomizer,org.springframework.boot.context.properties.ConfigurationPropert
iesBindingPostProcessor,org.springframework.boot.context.internalConfigurationP
ropertiesBinderFactory,org.springframework.boot.context.internalConfigurationPr
opertiesBinder,org.springframework.boot.context.properties.BoundConfigurationP
roperties,org.springframework.boot.context.properties.EnableConfigurationProper
tiesRegistrar.methodValidationExcludeFilter,server-
org.springframework.boot.autoconfigure.web.ServerProperties,webServerFactoryC
ustomizerBeanPostProcessor,errorPageRegistrarBeanPostProcessor,org.springfram
ework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration$Dispat
cherServletConfiguration,dispatcherServlet,spring.mvc-
org.springframework.boot.autoconfigure.web.servlet.WebMvcProperties,org.sprin
gframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration$
DispatcherServletRegistrationConfiguration,dispatcherServletRegistration,org.spri
ngframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,tas

kExecutorBuilder,applicationTaskExecutor,spring.task.execution-
org.springframework.boot.autoconfigure.task.TaskExecutionProperties,org.springfr
amework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$Whitel
abelErrorViewConfiguration,error,beanNameViewResolver,org.springframework.bo
ot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorViewR
esolverConfiguration,conventionErrorViewResolver,spring.web-
org.springframework.boot.autoconfigure.web.WebProperties,spring.resources-
org.springframework.boot.autoconfigure.web.ResourceProperties,org.springframe
work.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,errorAttribu
tes,basicErrorController,errorPageCustomizer,preserveErrorControllerTargetClassP
ostProcessor,org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoC
onfiguration$EnableWebMvcConfiguration,requestMappingHandlerAdapter,reque
stMappingHandlerMapping,welcomePageHandlerMapping,localeResolver,themeR
esolver,flashMapManager,mvcConversionService,mvcValidator,mvcContentNegoti
ationManager,mvcPatternParser,mvcUrlPathHelper,mvcPathMatcher,viewControlle
rHandlerMapping,beanNameHandlerMapping,routerFunctionMapping,resourceHa
ndlerMapping,mvcResourceUrlProvider,defaultServletHandlerMapping,handlerFun
ctionAdapter,mvcUriComponentsContributor,httpRequestHandlerAdapter,simpleC
ontrollerHandlerAdapter,handlerExceptionResolver,mvcViewResolver,mvcHandler
MappingIntrospector,viewNameTranslator,org.springframework.boot.autoconfigur
e.web.servlet.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter,def
aultViewResolver,viewResolver,requestContextFilter,org.springframework.boot.aut

oconfigure.web.servlet.WebMvcAutoConfiguration,formContentFilter,org.springfra
mework.boot.autoconfigure.jmx.JmxAutoConfiguration,mbeanExporter,objectNam
ingStrategy,mbeanServer,org.springframework.boot.autoconfigure.admin.SpringA
pplicationAdminJmxAutoConfiguration,springApplicationAdminRegistrar,org.sprin
gframework.boot.autoconfigure.aop.AopAutoConfiguration$AspectJAutoProxying
Configuration$CglibAutoProxyConfiguration,org.springframework.boot.autoconfig
ure.aop.AopAutoConfiguration$AspectJAutoProxyingConfiguration,org.springfram
ework.boot.autoconfigure.aop.AopAutoConfiguration,org.springframework.boot.a
utoconfigure.availability.ApplicationAvailabilityAutoConfiguration,applicationAvail
ability,org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
$Jackson2ObjectMapperBuilderCustomizerConfiguration,standardJacksonObjectM
apperBuilderCustomizer,spring.jackson-
org.springframework.boot.autoconfigure.jackson.JacksonProperties,org.springfra
mework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMa
pperBuilderConfiguration,jacksonObjectMapperBuilder,org.springframework.boot.
autoconfigure.jackson.JacksonAutoConfiguration$ParameterNamesModuleConfig
uration,parameterNamesModule,org.springframework.boot.autoconfigure.jackson.
JacksonAutoConfiguration$JacksonObjectMapperConfiguration,jacksonObjectMa
pper,org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,j
sonComponentModule,org.springframework.boot.autoconfigure.context.Configur
ationPropertiesAutoConfiguration,org.springframework.boot.autoconfigure.contex
t.LifecycleAutoConfiguration,lifecycleProcessor,spring.lifecycle-

org.springframework.boot.autoconfigure.context.LifecycleProperties,org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration$StringHttpMessageConverterConfiguration,stringHttpMessageConverter,org.springframework.boot.autoconfigure.http.JacksonHttpMessageConvertersConfiguration$MappingJackson2HttpMessageConverterConfiguration,mappingJackson2HttpMessageConverter,org.springframework.boot.autoconfigure.http.JacksonHttpMessageConvertersConfiguration,org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,messageConverters,org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,spring.info-
org.springframework.boot.autoconfigure.info.ProjectInfoProperties,org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration,spring.sql.init-
org.springframework.boot.autoconfigure.sql.init.SqlInitializationProperties,org.springframework.boot.sql.init.dependency.DatabaseInitializationDependencyConfigurer$DependsOnDatabaseInitializationPostProcessor,org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,scheduledBeanLazyInitializationExcludeFilter,taskSchedulerBuilder,spring.task.scheduling-
org.springframework.boot.autoconfigure.task.TaskSchedulingProperties,org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,restTemplateBuilderConfigurer,restTemplateBuilder,org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration$TomcatWebServerFactoryCustomizerConfiguration,tomcatWebServerFactoryCustomizer,org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServe

```
rFactoryCustomizerAutoConfiguration,org.springframework.boot.autoconfigure.w
eb.servlet.HttpEncodingAutoConfiguration,characterEncodingFilter,localeCharsetM
appingsCustomizer,org.springframework.boot.autoconfigure.web.servlet.Multipart
AutoConfiguration,multipartConfigElement,multipartResolver,spring.servlet.multip
art-org.springframework.boot.autoconfigure.web.servlet.MultipartProperties]; root
of factory hierarchy
postConstruct
afterPropertiesSet
initMethod


preDestroy
destroy
destroyMethod
```

BeanFactoryPostProcessor 和 BeanPostProcessor 区别

Spring 中 BeanFactoryPostProcessor 和 BeanPostProcessor 都是 Spring 初始化 bean 时对外暴露的扩展点。两个接口从名字看起来很相似，但是作用及使用场景却不同。

BeanFactoryPostProcessor

Spring IoC 容器允许 BeanFactoryPostProcessor 在容器实例化任何 bean 之前读取 bean 的定义(配置元数据)，并可以修改它。同时可以定义多个 BeanFactoryPostProcessor，通过设置'order'属性来确定各个

BeanFactoryPostProcessor 执行顺序。

注册一个 BeanFactoryPostProcessor 实例需要定义一个 Java 类来实现 BeanFactoryPostProcessor 接口，并重写该接口的 postProcessorBeanFactory 方法。通过 beanFactory 可以获取 bean 的定义信息，并可以修改 bean 的定义信息。这点是和 BeanPostProcessor 最大区别

BeanPostProcessor

1：后置处理器的 postProcessorBeforeInitailization 方法是在 bean 实例化，依赖注入之后及自定义初始化方法(例如：配置文件中 bean 标签添加 init-method 属性指定 Java 类中初始化方法、 @PostConstruct 注解指定初始化方法，Java 类实现 InitailztingBean 接口)之前调用

2：后置处理器的 postProcessorAfterInitailization 方法是在 bean 实例化、依赖注入及自定义初始化方法之后调用

注意： BeanFactory 和 ApplicationContext 两个容器对待 bean 的后置处理器稍微有些不同。ApplicationContext 容器会自动检测 Spring 配置文件中那些 bean 所对应的 Java 类实现了 BeanPostProcessor 接口，并自动把它们注册为后置处理器。在创建 bean 过程中调用它们，所以部署一个后置处理器跟普通的 bean 没有什么太大区别。

FactoryBean 与 ObjectFactory 以及其区别

```
public interface FactoryBean<T> {
```

```
    //获取对象的工厂方法
    @Nullable
    T getObject() throws Exception;
    //对象类型
    @Nullable
    Class<?> getObjectType();
    //是否是单例，这个属性决定了 spring 是否基于缓存来将它维护成一个单例对象。
    default boolean isSingleton() {
        return true;
    }
}
@FunctionalInterface
public interface ObjectFactory<T> {
    //获取对象的工厂方法。
    T getObject() throws BeansException;
}
```

FactoryBean 在 BeanFacotry 的实现中有着特殊的处理，如果一个对象实现了 FactoryBean 那么通过它 get 出来的对象实际是 factoryBean.getObject() 得到的对象，如果想得到 FactoryBean 必须通过在 '&' + beanName 的方式获取

ObjectFactory 则只是一个普通的对象工厂接口。在查看 AbstractBeanFacotry 的 doGetBean(..) 部分的源码时，可以看到 spring 对 ObjectFactory 的应用之一就是，将创建对象的步骤封装到 ObjectFactory 中 交给自定义的 Scope 来选择是否需要创建对象来

灵活的实现 scope。

Spring 是怎么解决 Bean 之间的循环依赖的

一级缓存里面是完整的 Bean,是当一个 Bean 完全创建后才 put

二级缓存是对三级缓存的易用性处理，只不过是通过 getObject()方法从三级缓存的 BeanFactory 中取出 Bean

三级缓存是不完整的 BeanFactory,是当一个 Bean 在 new 之后就 put(没有属性填充、初始化)

循环依赖解决

Spring 在创建 bean 的时候并不是等它完全完成，而是在创建过程中将创建中的 bean 的 ObjectFactory 提前曝光（即加入到 singletonFactories 三级缓存中），这样一旦下一个 bean 创建的时候需要依赖 bean ，则从三级缓存中获取

最后来描述之前那个循环依赖 Spring 解决的过程：

首先 A 完成初始化第一步并将自己提前曝光出来（通过 三级缓存将自己提前曝光），在初始化的时候，发现自己依赖对象 B，此时就会去尝试 get(B)，这个时候发现 B 还没有被创建出来，然后 B 就走创建流程，在 B 初始化的时候，同样发现自己依赖 C，C 也没有被创建出来，这个时候 C 又开始初始化进程，但是在初始化的过程中发现自己依赖 A，于是尝试 get(A)，这个时候由于 A 已经添加至缓存中（三级缓存 singletonFactories ），通过 ObjectFactory 提前曝光，所以可以通过 ObjectFactory#getObject() 方法来拿到 A 对象，C 拿到 A 对象后顺利完成初始化，然后将自己添加到一级缓存中，回到 B ，B 也可以拿到 C 对象，完成初始化，A 可以顺利拿到 B 完成初始化，到这里整个链路就已经完成了初始化过程了

为什么多例模式不能解决循环依赖呢?

因为多例模式下每次 new 的 Bean 都不是一个，如果按照这样存到缓存中，就变成单例了

BeanFactory 与 FactoryBean 的区别

一、这两个概念，命名比较相似，刚接触源码的时候，困惑了一票程序猿

1）BeanFactory 是接口，提供了 IOC 容器最基本的形式，给具体的 IOC 容器的实现提供了规范，顶层接口。

2）FactoryBean 也是接口，为 IOC 容器中 Bean 的实现提供了更加灵活的方式，FactoryBean 在 IOC 容器的基础上给 Bean 的实现加上了一个简单的工厂模式和装饰模式，我们可以在 getObject()方法中灵活配置.

区别：FactoryBean 是个 Bean 在 Spring 中，所有的 Bean 都是由 BeanFactory(也就是 IOC 容器)来进行管理的。但对 FactoryBean 而言，这个 Bean 不是简单的 Bean，而是一个能生产或者修饰对象生成的工厂 Bean,它的实现与设计模式中的工厂模式和修饰器模式类似 .

二、深入分析：

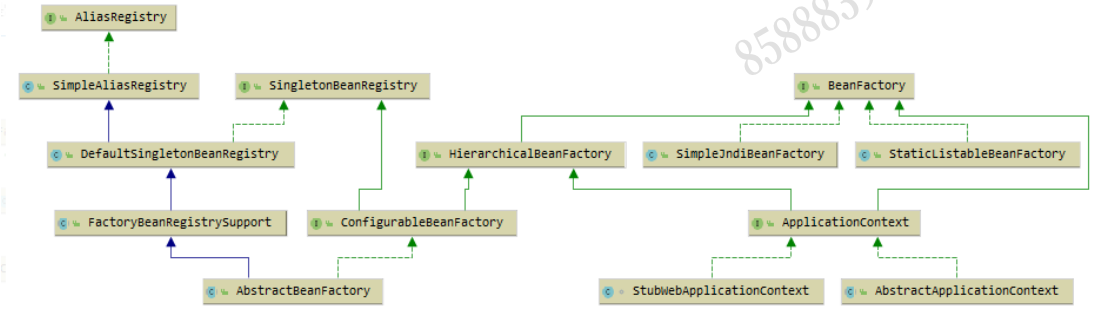
BeanFactory:

1）负责生产和管理 Spring 中 bean 的一个工厂；

2）IOC 容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象的依赖。

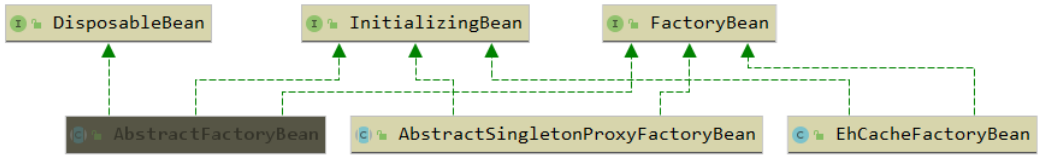
3）多种实现：如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext 等，其中 XmlBeanFactory 就是常用的一个，该实现将以 XML

方式描述组成应用的对象及对象间的依赖关系。



FactoryBean

FactoryBean 是一个接口，当在 IOC 容器中的 Bean 实现了 FactoryBean 后，通过 getBean(String BeanName)获取到的 Bean 对象并不是 FactoryBean 的实现类对象，而是这个实现类中的 getObject()方法返回的对象。要想获取 FactoryBean 的实现类，就要 getBean(&BeanName)，在 BeanName 之前加上&。



BeanFactory 和 ApplicationContext 的区别总结

BeanFactory：

是 Spring 里面最底层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

ApplicationContext：

应用上下文，继承 BeanFactory 接口，它是 Spring 的一各更高级的容器，提供了更多的有用的功能；

- 1) 国际化（MessageSource）
- 2) 访问资源，如 URL 和文件（ResourceLoader）
- 3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层
- 4) 消息发送、响应机制（ApplicationEventPublisher）

- 5) AOP（拦截器）

两者装载 bean 的区别

BeanFactory：BeanFactory 在启动的时候不会去实例化 Bean，当有从容器中拿 Bean 的时候才会去实例化；

ApplicationContext：ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了。它还可以为 Bean 配置 lazy-init=true 来让 Bean 延迟实例化；

我们该用 BeanFactory 还是 ApplicationContent

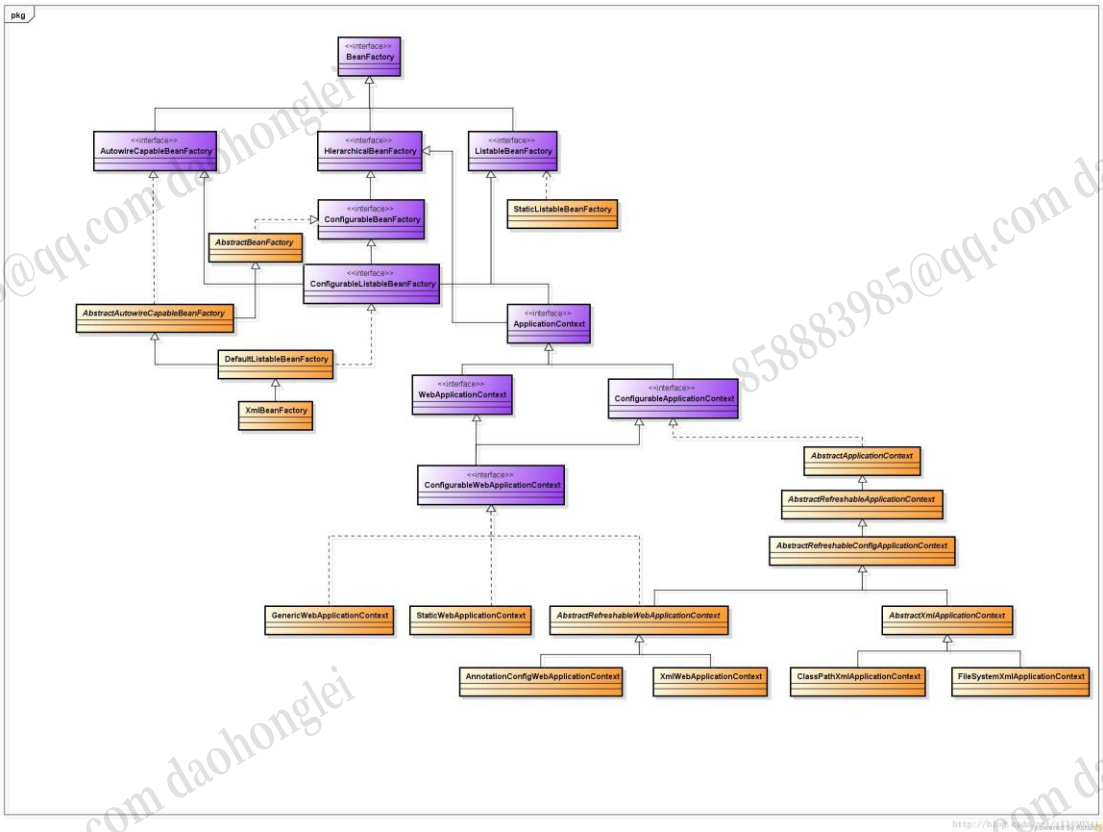
延迟实例化的优点：（BeanFactory）

- 应用启动的时候占用资源很少；对资源要求较高的应用，比较有优势；

不延迟实例化的优点：（ApplicationContext）

- 1. 所有的 Bean 在启动的时候都加载，系统运行的速度快；
- 2. 在启动的时候所有的 Bean 都加载了，我们就能在系统启动的时候，尽早的发现系统中的配置问题
- 3. 建议 web 应用，在启动的时候就把所有的 Bean 都加载了。（把费时的操作放到系统启动中完成）

BeanFactory 类关系继承图



@Lazy 的作用

1 延迟加载 bean 对象，减少容器启动时间

2 解决循环依赖问题

解释一下什么是 aop?

AOP（Aspect-Oriented Programming，面向方面编程），可以说是 OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP 则显得无能为力。也就是说，OOP 允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在 OOP 设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而 AOP 技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP 代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的

切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如 Avanade 公司的高级方案构架师 Adam Magee 所说，AOP 的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

Spring aop 通知方法

- 前置通知(@Before)：logStart：在目标方法(div)运行之前运行
- 后置通知(@After)：logEnd：在目标方法(div)运行结束之后运行（无论方法正常结束还是异常结束）
- 返回通知(@AfterReturning)：logReturn：在目标方法(div)正常返回之后运行
- 异常通知(@AfterThrowing)：logException：在目标方法(div)出现异常以后运行
- 环绕通知(@Around)：动态代理，手动推进目标方法运行（joinPoint.procced()）

Cglib 和 jdk 动态代理的区别

JDK 和 Cglib 都是在运行期生成字节码

1、Jdk 动态代理：利用拦截器（必须实现 InvocationHandler）加上反射机制生成一个代理接口的匿名类，在调用具体方法前调用 InvokeHandler 来处理

2、 Cglib 动态代理：利用 ASM 框架，对代理对象类生成的 class 文件加载进来，通过修改其字节码生成子类来处理

什么时候用 cglib 什么时候用 jdk 动态代理?

1、目标对象生成了接口 默认用 JDK 动态代理

2、如果目标对象使用了接口，可以强制使用 cglib

3、如果目标对象没有实现接口，必须采用 cglib 库，Spring 会自动在 JDK 动态代理和 cglib 之间转换

JDK 动态代理和 cglib 字节码生成的区别?

1、JDK 动态代理只能对实现了接口的类生成代理，而不能针对类

2、Cglib 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法的增强，但是因为采用的是继承，和 private final 的关系如下。

a.被代理的类在出现 final 修饰时编译会通过但会在运行报错，

b.由于 final 方法无法被继承，所以编译运行不会报错，但是方法无法被代理

Cglib 比 JDK 快?

1、cglib 底层是 ASM 字节码生成框架，但是字节码技术生成代理类，在 JDK1.6 之前比使用 java 反射的效率要高

2、在 jdk6 之后逐步对 JDK 动态代理进行了优化，在调用次数比较少时效率高于 cglib 代理效率，只有在大量调用的时候 cglib 的效率高

3、在 1.8 的时候 JDK 的效率已高于 cglib

Spring 支持几种 bean 的作用域?

当通过 spring 容器创建一个 Bean 实例时，不仅可以完成 Bean 实例的实例化，还可以为 Bean 指定特定的作用域。Spring 支持如下 5 种作用域:

- singleton：单例模式，在整个 Spring IoC 容器中，使用 singleton 定义的 Bean 将只有一个实例
- prototype：原型模式，每次通过容器的 getBean 方法获取 prototype

定义的 Bean 时，都将产生一个新的 Bean 实例

- request：对于每次 HTTP 请求，使用 request 定义的 Bean 都将产生一个新实例，即每次 HTTP 请求将会产生不同的 Bean 实例。只有在 Web 应用中使用 Spring 时，该作用域才有效
- session：对于每次 HTTP Session，使用 session 定义的 Bean 豆浆产生一个新实例。同样只有在 Web 应用中使用 Spring 时，该作用域才有效
- globalsession：每个全局的 HTTP Session，使用 session 定义的 Bean 都将产生一个新实例。典型情况下，仅在使用 portlet context 的时候有效。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

其中比较常用的是 singleton 和 prototype 两种作用域。对于 singleton 作用域的 Bean，每次请求该 Bean 都将获得相同的实例。容器负责跟踪 Bean 实例的状态，负责维护 Bean 实例的生命周期行为；如果一个 Bean 被设置成 prototype 作用域，程序每次请求该 id 的 Bean，Spring 都会新建一个 Bean 实例，然后返回给程序。在这种情况下，Spring 容器仅仅使用 new 关键字创建 Bean 实例，一旦创建成功，容器不在跟踪实例，也不会维护 Bean 实例的状态。

如果不指定 Bean 的作用域，Spring 默认使用 singleton 作用域。Java 在创建 Java 实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，prototype 作用域 Bean 的创建、销毁代价比较大。而 singleton 作用域的 Bean 实例一旦创建成功，可以重复使用。因此，除非必要，否则尽量避免将 Bean 被设置成 prototype 作用域。

Springmvc 生命周期：

1.用户向服务器发送请求，请求被 Spring 前端控制 Servelt DispatcherServlet 捕获

2.DispatcherServlet 对请求 URL 进行解析，得到请求资源标识符（URI）。然后根据该 URI，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回

3.DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter。（附注：如果成功获得 HandlerAdapter 后，此时将开始执行拦截器的 preHandler(...)方法)

4.提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（Controller)。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：

HttpMessageConveter： 将请求消息（如 Json、xml 等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如 String 转换成 Integer、Double 等

数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

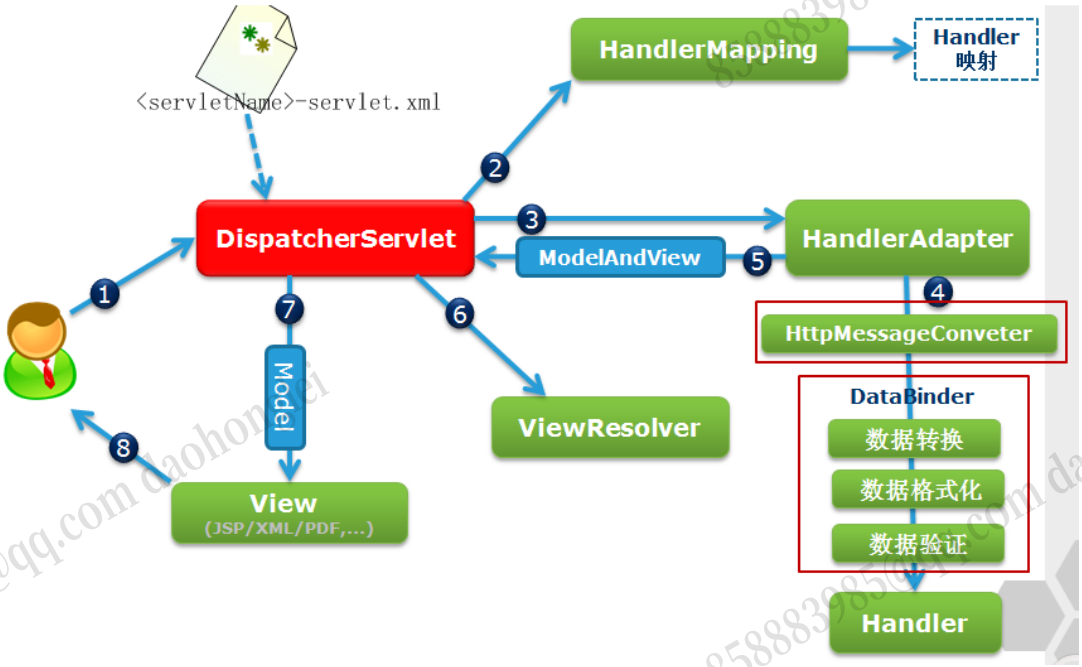
数据验证： 验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 Error 中

5.Handler 执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象

6.根据返回的 ModelAndView，选择一个适合的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver)返回给 DispatcherServlet

7.ViewResolver 结合 Model 和 View，来渲染视图

8.将渲染结果返回给客户端

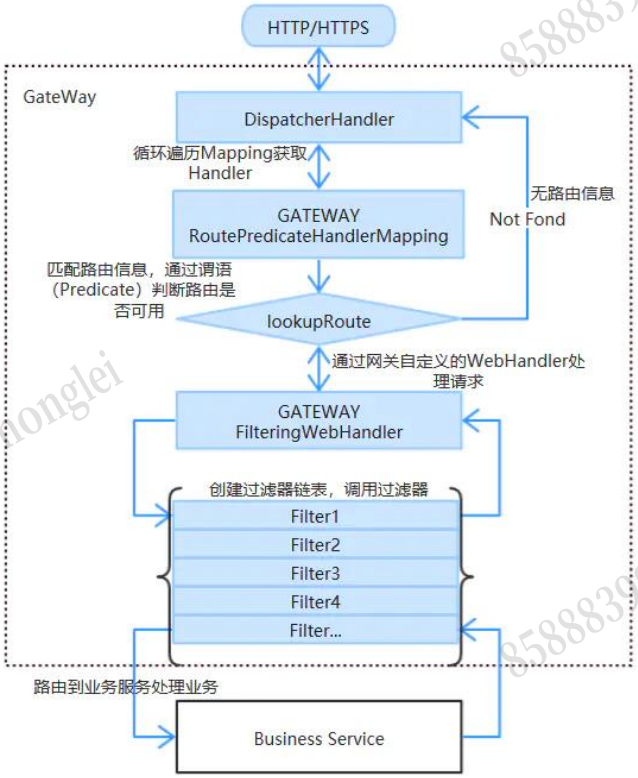


spring mvc 有哪些组件?

Spring MVC 的核心组件：

1. DispatcherServlet：中央控制器，把请求给转发到具体的控制类
2. Controller：具体处理请求的控制器
3. HandlerMapping：映射处理器，负责映射中央处理器转发给 controller 时的映射策略
4. ModelAndView：服务层返回的数据和视图层的封装类
5. ViewResolver：视图解析器，解析具体的视图
6. Interceptors ：拦截器，负责拦截我们定义的请求然后做处理工作

SpringCloud gateway 原理分析



1. 项目启动时路由的配置转换为 RouteDefinition

2.有请求后将请求交给 DispatcherHandler 处理

3.获取请求对应的路由规则, 将 RouteDefinition 转换为 Route

- RouterFunctionMapping:匹配 WebFlux 的 router functions。
- RequestMappingHandlerMapping:匹配@RequestMapping 标注的请求。
- RoutePredicateHandlerMapping:匹配 Spring Cloud Gateway 中路由里的 断言集合。
- SimpleUrlHandlerMapping: 匹配静态资源等请求。

4.执行 Predicate 判断是否符合路由, 以及执行相关的过滤(全局过滤器以及路由过滤器)

5.创建过滤器链表对其进行链式调用

a.RouteToRequestUrlFilter：根据匹配的 Route ， 计算请求的地址

b.LoadBalancerClientFilter：负责将请求中的 serviceId 转换为具体的服务实例

lp

c.NettyRoutingFilter：根据 http:// 或 https:// 前缀(Scheme)过滤处理，使用基于 Netty 实现的 HttpClient 请求后端 Http 服务。

d.NettyWriteResponseFilter：与 NettyRoutingFilter 成对使用的网关过滤器。其将 NettyRoutingFilter 请求后端 Http 服务的响应写回客户端。

gateway 配置

uri 配置方式

1. 第一种：ws(websocket)方式: uri: ws://localhost:9000
2. 第二种：http 方式: uri: http://localhost:8130/
3. 第三种：lb(注册中心中服务名字)方式: uri: lb://brilliance-consumer

其中 ws 和 http 方式不容易出错，因为 http 格式比较固定，但是 lb 方式比较灵活自由。不考虑网关，只考虑服务时，服务名命名时比较自由，都能启动被访问，被注册到注册中心，但是如果提供给 gateway 使用时，就会对服务命名方式有特殊要求了。

https://blog.csdn.net/weixin_40579395/article/details/124450221

Route Predicate Factory

1. After 作用：请求在指定时间之后才匹配

```
spring:
  cloud:
    gateway:
      routes:
```

```
- id: USER-CENTER #路由的 ID
    uri:  http://localhost:8888/ #匹配后路由地址
    predicates:
        - After=2022-04-27T16:35:04.030+08:00[Asia/Shanghai]
```

2. Before 作用：请求在指定时间之前才匹配

```
spring:
 cloud:
  gateway:
   routes:
    - id: USER-CENTER #路由的 ID
      uri:  http://localhost:8888/ #匹配后路由地址
      predicates:
          - Before=2022-04-27T16:35:04.030+08:00[Asia/Shanghai]
```

3. Between 作用：请求在指定时间区间之内才匹配，

第一个时间需要小于第二个时间。

```
spring:
 cloud:
  gateway:
   routes:
    - id: USER-CENTER #路由的 ID
      uri:  http://localhost:8888/ #匹配后路由地址
      predicates:
```

```
        - Between=2022-04-22T16:00:00.020+08:00[Asia/Shanghai],2022-04-
22T16:30:00.020+08:00[Asia/Shanghai]
```

4. Cookie 作用：请求携带指定 Cookie 才匹配

只有当请求 Cookie 中带有 name=yellowDuck 才可以匹配到此路由

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
          predicates:
            - Cookie=name,yellowDuck
```

5. Header 作用：请求携带指定 Header 才匹配

只有当请求 Header 种中带有 X-User-Id:001 才可以匹配到此路由，其中\d+为校验数字正则表达式，可以根据需要自己定制。

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
```

```
      predicates:
        - Header=X-User-Id,\d+
```

6. Host 作用：请求携带指定 Host 才匹配

```
spring:
  cloud:
    gateway:
      routes:
      - id: USER-CENTER #路由的 ID
        uri:  http://localhost:8888/ #匹配后路由地址
        predicates:
          - Host=**.yellowDuck.com
```

7. Method 作用：请求指定 Mehtod 请求方式才匹配

只有 GET,POST,DELETE 请求才可以访问

```
spring:
  cloud:
    gateway:
      routes:
       - id: USER-CENTER #路由的 ID
         uri:  http://localhost:8888/ #匹配后路由地址
         predicates:
           - Method=GET,POST,DELETE
```

8. Path 作用：请求路径匹配

只有包含配置的路径才可以匹配 也可以支持/duck/{color}参数形式

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
          predicates:
            - Path=/duck/**
```

9. Query 作用：请求参数包含才匹配

请求中必须有 duck 这个参数才可以访问

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
          predicates:
            - Query=duck
```

10. RemoteAddr 作用：请求的 IP/IP 段一致才可以访问

```
spring:
  cloud:
```

```
  gateway:
    routes:
     - id: USER-CENTER #路由的 ID
       uri:  http://localhost:8888/ #匹配后路由地址
       predicates:
         - RemoteAddr=127.0.0.1/24
```

11. Weight 作用：根据权重的分配路由到相应的请求

group 分组，wight 权重，是一个 int 值，下面代表有 90%的请求会分配到 http://localhost:8888/有 10%请求会分配到 http://localhost:8887

```
spring:
 cloud:
  gateway:
    routes:
     - id: USER-CENTER #路由的 ID
       uri:  http://localhost:8888/ #匹配后路由地址
       predicates:
         - Weight=group1,9
     - id: DUCK-CENTER #路由的 ID
       uri: http://localhost:8887/ #匹配后路由地址
       predicates:
         - Weight=group2,1
```

Route Filters

过滤器 Filter 将会对请求和响应进行修改处理，路由过滤器只能指定路由进行使用。

1. AddRequestParameter 作用：对请求添加参数

如下对 GET 请求添加 duck=yellow 相当于 ?duck=yellow

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
          filters:
            - AddRequestParameter=duck,yellow
          predicates:
            - Method=GET
```

2. StripPrefix GatewayFilter 作用：对指定数量的路径进行删除过滤

比如 http://localhost:8888/yellow/big/duck 过滤前两个路径后会变成 http://localhost:8888/duck

```
spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
          uri:  http://localhost:8888/ #匹配后路由地址
```

```
      filters:
        - Path=/duck/**
      predicates:
        - StripPrefix=2
```

3. PrefixPath GatewayFilter 作用：对指定的路径进行增加

比如 http://localhost:8888/duck 过滤增加路径后会变

成 http://localhost:8888/yellow/duck

```
spring:
 cloud:
  gateway:
   routes:
    - id: USER-CENTER #路由的 ID
      uri:  http://localhost:8888/ #匹配后路由地址
      filters:
        - Method=POST
      predicates:
        - PrefixPath=/yellow
```

4. Hystrix GatewayFilter 作用：提供了熔断和降级功能

a) 首先需要添加 Hystrix 依赖，Hystrix 提供了熔断和降级。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
```

```
</dependency>
```

b) 编写服务降级的处理类

```
package com.gostop.cloud.gateway.config;


import com.gostop.cloud.gateway.handle.HystrixFallbackHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.server.RequestPredicates;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;



@Configuration
public class GatewayFallbackConfiguration {

    @Autowired
    private HystrixFallbackHandler hystrixFallbackHandler;

    @Bean
    public RouterFunction routerFunction() {
```

```
        return RouterFunctions.route(
                RequestPredicates.GET("/defaultfallback")
                        .and(RequestPredicates.accept(MediaType.TEXT_PLAIN)),
hystrixFallbackHandler);
    }
}
package com.gostop.cloud.gateway.handle;

import lombok.extern.slf4j.Slf4j;
import org.springframework.cloud.gateway.support.ServerWebExchangeUtils;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.BodyInserters;
import org.springframework.web.reactive.function.server.HandlerFunction;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;


@Slf4j
@Component
public class HystrixFallbackHandler implements HandlerFunction<ServerResponse>
```

```
{


    @Override
    public Mono<ServerResponse> handle(ServerRequest serverRequest) {


serverRequest.attribute(ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_
URL_ATTR)
            .ifPresent(originalUrls -> log.error("=====网关执行请求:{}失败,服务降级处
理=====", originalUrls));


        return ServerResponse
                .status(HttpStatus.INTERNAL_SERVER_ERROR)
                .contentType(MediaType.APPLICATION_JSON_UTF8)
                .body(BodyInserters.fromValue("网络繁忙！"));
    }
}


spring:
  cloud:
    gateway:
      routes:
        - id: USER-CENTER #路由的 ID
```

```
      uri:  http://localhost:8888/ #匹配后路由地址
      filters:
        - name: Hystrix
          args:
            name: fallback
            fallbackUri: forward:/defaultfallback
      predicates:
        - Method=GET
```

5. RequestRateLimiter GatewayFilter 作用：用于限流

引入 Redis 限流

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>


spring:
 cloud:
   gateway:
     routes:
       - id: USER-CENTER #路由的 ID
         uri:  http://localhost:8888/ #匹配后路由地址
         filters:
```

```
        - name: RequestRateLimiter
          args:
            #每秒允许处理的请求数量
            redis-rate-limiter.replenishRate: 10
            #每秒最大处理的请求数量
            redis-rate-limiter.burstCapacity: 20
            redis-rate-limiter.requestedTokens: 1
        predicates:
          - Method=GET,POST
```

拦截器（Interceptor）和过滤器（Filter）的区别

Spring 的 Interceptor(拦截器)与 Servlet 的 Filter 有相似之处，比如二者都是 AOP 编程思想的体现，都能实现权限检查、日志记录等。不同的是：

Filter	Interceptor	Summary
Filter 接口定义在 javax.servlet 包中 Filter 定义在 web.xml 中	接口 HandlerInterceptor 定义在 org.springframework.web.servlet 包中	
Filter 在只在 Servlet 前后起作用。Filters 通常将 请求和响应（request/response） 当做黑盒子，Filter 通常不考虑 servlet 的实现。	拦截器能够深入到方法前后、异常抛出前后等，因此拦截器的使用具有更大的弹性。允许用户介入（hook into）请求的生命周期，在请求过程中获取信息，Interceptor 通常和请求更加耦合。	在 Spring 构架的程序中，要优先使用拦截器。几乎所有 Filter 能够做的事情，

		interceptor 都能够轻松的实现
Filter 是 Servlet 规范规定的。	而拦截器既可以用于 Web 程序，也可以用于 Application、Swing 程序中。	使用范围不同
Filter 是在 Servlet 规范中定义的，是 Servlet 容器支持的。	而拦截器是在 Spring 容器内的，是 Spring 框架支持的。	规范不同
Filter 不能够使用 Spring 容器资源	拦截器是一个 Spring 的组件，归 Spring 管理，配置在 Spring 文件中，因此能使用 Spring 里的任何资源、对象，例如 Service 对象、数据源、事务管理等，通过 IoC 注入到拦截器即可	Spring 中使用 interceptor 更容易
Filter 是被 Server(like Tomcat) 调用	Interceptor 是被 Spring 调用	因此 Filter 总是优先于 Interceptor 执行

Spring 声明式事务原理

1）、@EnableTransactionManagement

利用 TransactionManagementConfigurationSelector 给容器中 导入两个组件

AutoProxyRegistrar

ProxyTransactionManagementConfiguration

2）、AutoProxyRegistrar：

给容器中注册一个 InfrastructureAdvisorAutoProxyCreator 组件；

InfrastructureAdvisorAutoProxyCreator 利用后置处理器机制在对象创建以后，包装对象，返回一个代理对象（增强器），代理对象执行方法利用拦截器链进行调用；

3）、ProxyTransactionManagementConfiguration(配置文件) 做了什么?

给容器中注册事务增强器；

1）、事务增强器要用事务注解的信息:

AnnotationTransactionAttributeSource 解析事务注解

2）、事务拦截器：TransactionInterceptor；保存了事务属性信息，事务管理器；

他是一个 MethodInterceptor；

在目标方法执行的时候；

执行拦截器链；

事务拦截器：

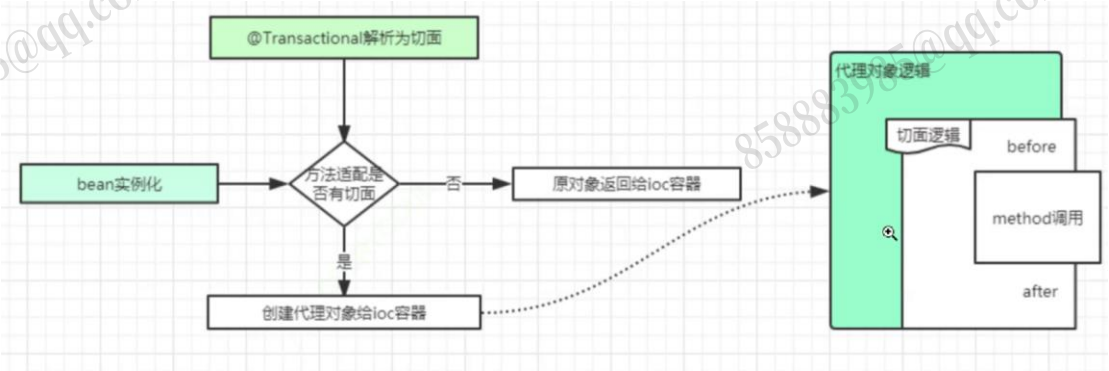
1）、先获取事务相关的属性

2）、再获取 PlatformTransactionManager，如果事先没有添加指定任何 transactionmanger，最终会从容器中按照类型获取一个 PlatformTransactionManager；

3）、执行目标方法

如果异常，获取到事务管理器，利用事务管理回滚操作；

如果正常，利用事务管理器，提交事务



Spring 事务没回滚

1.事务方法必须是 public，如果定义成 protected、private 或者默认可见性,则无法调用

2.注意关键词 “动态代理”，这意味着要生成一个代理类，那么我们就不能在一个类内直接调用事务方法，否则无法代理

3.默认情况下，当程序发生 RuntimeException 和 Error 的这两种异常的时候事务会回滚，但是如果发生了 checkedExceptions ，如 fileNotfundException 则不会回滚，所以 rollbackFor = Exception.class 这个一定要加

4.在 a 方法中调用 b 方法，不管 a 和 b 是不是在同一个类中，只要 a 方法中没有事务，则发生异常的时候不会回滚，即：当 a 无事务时，则 a 和 b 均没有事务，当 a 有事务时，b 如果有事务，则 b 事务会加到 a 事务中，二者为同一事务

1. 抛出检查异常导致事务不能正确回滚

   原因：Spring 默认只会回滚非检查异常 RuntimeException 和 Error

   解法：配置 rollbackFor 属性

2. 业务方法内自己 try-catch 异常导致事务不能正确回滚

   原因：事务通知只有捉到了目标抛出的异常，才能进行后续的回滚处理，如果目标自己处理掉异常，事务通知无法知悉

   解法 1：异常原样抛出

解法 2：手动设置 TransactionStatus.setRollbackOnly()

3. aop 切面顺序导致事务不能正确回滚

   原因：事务切面优先级最低，但如果自定义的切面优先级和他一样，则还是自定义切面在内层，这时若自定义切面没有正确抛出异常...

   解法 1：异常原样抛出

   解法 2：手动设置 TransactionStatus.setRollbackOnly()

   解法 3：指定事务优先级

4. 非 public 方法导致的事务失效

   原因：Spring 为方法创建代理、添加事务通知、前提条件都是该方法是 public 的

   解法：改为 public 方法

5. 父子容器导致的事务失效　Spring MVC

   原因：子容器扫描范围过大，把未加事务配置的 service 扫描进来

   解法 1：各扫描各的，不要图简便

   解法 2：不要用父子容器，所有 bean 放在同一容器

6. 调用本类方法导致传播行为失效

   原因：本类方法调用不经过代理，因此无法增强

   解法 1：依赖注入自己（代理）来调用

   - 解法 2：通过 AopContext 拿到代理对象，来调用

   解法 3：通过 CTW，LTW 实现功能增强

7. @Transactional 没有保证原子行为

   原因：事务的原子性仅涵盖 insert、update、delete、select ... for update 语句，select 方法并不阻塞

8. @Transactional 方法导致的 synchronized 失效

   原因： synchronized 保证的仅是目标方法的原子性，环绕目标方法的还有 commit 等操作，它们并未处于 sync 块内

   解法 1：synchronized 范围应扩大至代理方法调用

   解法 2：使用 select ... for update 替换 select

Spring 事务的隔离级别

spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致

- ISOLATION_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
- ISOLATION_ READUNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读；
- ISOLATION_READCOMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
- ISOLATION_REPEATABLEREAD：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
- ISOLATION_ SERIALIZABLE：序列化，代价最高最可靠的隔离级别，不允许读写并发操作，写执行时，读必须等待,该隔离级别能防止脏读、不可重复读、幻读。

Spring 事务传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在 TransactionDefinition 定义中包括了如下几个表示传播行为的常量：

- TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。
- TransactionDefinition.PROPAGATION_REQUIRES_NEW：如果当前存在事务，则把当前事务挂起，创建一个新的事务。
- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED：如果当前存在事务，则把当前事务挂起，以非事务方式运行。
- TransactionDefinition.PROPAGATION_NEVER：如果当前存在事务，则抛出异常，以非事务方式运行。
- TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
- TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

Spring 手动开启事务

```
@Autowired

private DataSourceTransactionManager transactionManager;



DefaultTransactionDefinition defaultTransactionDefinition = new

DefaultTransactionDefinition();

defaultTransactionDefinition.setPropagationBehavior(TransactionDefinition.PROPA

GATION_REQUIRED);

TransactionStatus transaction =

transactionManager.getTransaction(defaultTransactionDefinition);



transactionManager.commit(transaction);

transactionManager.rollback(transaction);
```

为什么要用 Spring?

1.方便解耦，便于开发（Spring 就是一个大工厂，可以将所有对象的创建和依赖关系维护都交给 spring 管理）
2.spring 支持 aop 编程（spring 提供面向切面编程，可以很方便的实现对程序进行权限拦截和运行监控等功能）
3.声明式事务的支持（通过配置就完成对事务的支持，不需要手动编程）
4.方便程序的测试，spring 对 junit4 支持，可以通过注解方便的测试 spring 程序
5.方便集成各种优秀的框架
6.降低 javaEE API 的使用难度（Spring 对 javaEE 开发中非常难用的一些 API 例如 JDBC,javaMail,远程调用等，都提供了封装，是这些 API 应用难度大大降低）

Spring 和 Spring boot 区别

二者主要区别是:

1、Spring Boot 提供极其快速和简化的操作，让 Spring 开发者快速上手。

2、Spring Boot 提供了 Spring 运行的默认配置。

3、Spring Boot 为通用 Spring 项目提供了很多非功能性特性。

一 、Spring Boot

Spring Boot 基本上是 Spring 框架的扩展，它消除了设置 Spring 应用程序所需的 XML 配置，为更快，更高效的开发生态系统铺平了道路。 以下是 Spring Boot 中的一些特点:

1：创建独立的 spring 应用

2：嵌入 Tomcat , Jetty Undertow 而且不需要部署他们

3：提供的“starters” poms 来简化 Maven 配置

4：尽可能自动配置 spring 应用

5：提供生产指标,健壮检查和外部化配置

6：绝对没有代码生成和 XML 配置要求

二、Spring

Spring 框架为开发 Java 应用程序提供了全面的基础架构支持，它包含一些很好的功能，如依赖注入和开箱即用的模块，如：Spring JDBC 、Spring MVC 、Spring Security、 Spring AOP 、Spring ORM 、Spring Test 这些模块大家应该都用过吧，这些模块缩短应用程序的开发时间，提高了应用开发的效率。例如，在 Java Web 开发的早期阶段，我们需要编写大量的代码来将记录插入到数据源中。但是通过使用 Spring JDBC 模块的 JDBCTemplate ，我们可以将这操作简化为只需配置几行代码。

SpringBoot 使用 AutoConfiguration 自定义 Starter

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-autoconfigure</artifactId>
   <version>2.1.8.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <version>2.1.8.RELEASE</version>
</dependency>
```

配置映射参数实体

我们在文章开头埋下了一个疑问，starter 是如何读取 application.properties 或者 application.yml 配置文件内需要的配置参数的呢？那么接下来我们就看看如何可以获

取自定义的配置信息。

SpringBoot 在处理这种事情上早就已经考虑到了，所以提供了一个注解 @ConfigurationProperties，该注解可以完成将 application.properties 配置文件内的有规则的配置参数映射到实体内的 field 内，不过需要提供 setter 方法，自定义配置参数实体代码如下所示：

```
import org.springframework.boot.context.properties.ConfigurationProperties;
@ConfigurationProperties(prefix = "hello")
public class HelloProperties{
   //消息内容
   private String msg = "HengYu";
   //是否显示消息内容
   private boolean show = true;



   public String getMsg() {
      return msg;
   }
   public void setMsg(String msg) {
      this.msg = msg;
   }
   public boolean isShow() {
      return show;
   }
   public void setShow(boolean show) {
      this.show = show;
   }
}
```

编写自定义业务

我们为自定义 starter 提供一个 Service，并且提供一个名为 sayHello 的方法用于返回我们配置的 msg 内容。代码如下所示：

```
public class HelloService{
   //消息内容
   private String msg;
   //是否显示消息内容
   private boolean show = true;
```

```
    public String sayHello(){
        return show ? "Hello，" + msg : "Hidden";
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
    public void setShow(boolean show) {
        this.show = show;
    }
}
```

实现自动化配置

自动化配置其实只是提供实体 bean 的验证以及初始化，我们先来看看代码：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration//开启配置
@EnableConfigurationProperties(HelloProperties.class)//开启使用映射实体对象
@ConditionalOnClass(HelloService.class)//存在 HelloService 时初始化该配置类
@ConditionalOnProperty//存在对应配置信息时初始化该配置类
    (
        prefix = "hello",//存在配置前缀 hello
        value = "enabled",//开启
        matchIfMissing = true//缺失检查
    )
public class HelloAutoConfiguration{
```

```
    @Autowired  //application.properties 配置文件映射前缀实体对象
    private HelloProperties helloProperties;


    /**
     * 根据条件判断不存在 HelloService 时初始化新 bean 到 SpringIoc
     * @return
     */
    @Bean //创建 HelloService 实体 bean
    @ConditionalOnMissingBean(HelloService.class) //缺失 HelloService 实体 bean 时，初始化 HelloService 并添加到 SpringIoc
    public HelloService helloService(){
        System.out.println(">>>The HelloService Not Found，Execute Create New Bean.");
        HelloService helloService = new HelloService();
        helloService.setMsg(helloProperties.getMsg());//设置消息内容
        helloService.setShow(helloProperties.isShow());//设置是否显示
        return helloService;
    }
}
```

自定义 spring.factories

我们在 src/main/resource 目录下创建 META-INF 目录，并在目录内添加文件

spring.factories，具体内容如下所示：

```
#配置自定义 Starter 的自动化配置
org.springframework.boot.autoconfigure.EnableAutoConfiguration=HelloAutoConfiguration
```

SpringBoot 内置条件注解

有关@ConditionalOnXxx 相关的注解这里要系统的说下，因为这个是我们配置的关键，根据名称我们可以理解为具有 Xxx 条件，当然它实际的意义也是如此，条件注解是一个系列，下面我们详细做出解释

@ConditionalOnBean：当 SpringIoc 容器内存在指定 Bean 的条件

@ConditionalOnClass：当 SpringIoc 容器内存在指定 Class 的条件

@ConditionalOnExpression：基于 SpEL 表达式作为判断条件

@ConditionalOnJava：基于 JVM 版本作为判断条件

@ConditionalOnJndi：在 JNDI 存在时查找指定的位置

@ConditionalOnMissingBean：当 SpringIoc 容器内不存在指定 Bean 的条件

@ConditionalOnMissingClass：当 SpringIoc 容器内不存在指定 Class 的条件

@ConditionalOnNotWebApplication：当前项目不是 Web 项目的条件

@ConditionalOnProperty：指定的属性是否有指定的值

@ConditionalOnResource：类路径是否有指定的值

@ConditionalOnSingleCandidate：当指定 Bean 在 SpringIoc 容器内只有一个，或者虽然有多个但是指定首选的 Bean

@ConditionalOnWebApplication：当前项目是 Web 项目的条件

以上注解都是元注解@Conditional 演变而来的，根据不用的条件对应创建以上的具体条件注解。

到目前为止我们还没有完成自动化配置 starter，我们需要了解 SpringBoot 运作原理后才可以完成后续编码。

SpringCloud FeignClient 原理

注册

1.通过@EnableFeignClients 注入 FeignClientsRegistrar，FeignClientsRegistrar 实现了 ImportBeanDefinitionRegistrar 接口。

2.通过 FeignClientsRegistrar 注册了 FeignClientFactoryBean，FeignClientFactoryBean 实现了 FactoryBean 接口。

3.**@Autowired** 会调用 FeignClientFactoryBean 的 getObject 生成代理对象，代理

对象的方法用 SynchronousMethodHandler 包装。

调用

1.调用接口是先被 ReflectiveFeign 的 invoke 拦截，然后调用 SynchronousMethodHandler 的 invoke 方法。

2.通过 clientName 得到 LoadBalancer 信息。

3.选取合适的 client 发送请求

SpringBoot 单元测试

```
@ConfigurationProperties(prefix = "user")
@Component
public class UserProperties {
    private String name;
    private Integer age;
}


@Service
public class Service1 {
    @Autowired
    private Service2 service2;
    public String test1(){
        return "test1";
    }
```

```
    public String test2(){
        return service2.test2();
    }
}


public class Service2 {
    public String test2() {
        return "test2";
    }
}


@Configuration
public class AppConfig {
    @Bean
    public Service2 service2() {
        return new Service2();
    }
}


@RestController
public class MyController {
    @Autowired
```

```
    private Service1 service1;
    @GetMapping("get")
    public String get(String param) {
        System.out.println(param);
        return "index";
    }
    @PostMapping("post")
    public User post(@RequestBody User user) {
        System.out.println(JSON.toJSONString(user));
        user.setName(service1.test1());
        user.setAge(userProperties.getAge());
        return user;
    }
}
```

```
import com.alibaba.fastjson.JSON;
import com.example.demo.dto.User;
import com.example.demo.service.Service1;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
```

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;
import org.springframework.http.MediaType;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import org.springframework.transaction.annotation.Transactional;

import java.util.Map;
import java.util.Properties;
import java.util.Set;

import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

@SpringBootTest(classes = Demo2Application.class, properties =
{"user.age=100"}, args = {"--user.age=20"})
@Import({AppConfig.class})
@AutoConfigureMockMvc
@Transactional
@Rollback(true)
```

```
class Demo2ApplicationTests {


    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private Service1 service1;


    @Test
    void test1() {
        System.out.println(service1.test1());
    }


    @Test
    void test2() {
        System.out.println(service1.test2());
    }


    @Test
    void testGet() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders
                .get("/get")
                .param("param", "hello")
```

```
                .accept(MediaType.APPLICATION_FORM_URLENCODED)
                .contentType(MediaType.APPLICATION_FORM_URLENCODED_VAL
UE)
                .header("Authorization", "Bearer ********-****-****-****-************")
            )
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andDo(print());
    }

    @Test
    void testPost() throws Exception {
        User user = new User();
        user.setName("1233");
        user.setDetail("2333");
        user.setAge(18);
        mockMvc.perform(MockMvcRequestBuilders
                .post("/post")
                .content(JSON.toJSONString(user).getBytes())
                .accept(MediaType.APPLICATION_JSON_UTF8)
                .contentType(MediaType.APPLICATION_PROBLEM_JSON_UTF8_VAL
UE)
                .header("Authorization", "Bearer ********-****-****-****-************")
```

```
            )
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andDo(print());
        Properties properties = System.getProperties();
        Set<Map.Entry<Object, Object>> entries = properties.entrySet();
        for (Map.Entry<Object, Object> entry : entries) {
            System.out.println(entry.getKey() + "=" + entry.getValue());
        }
    }
}
```

https://github.com/daohonglei/javaStereotypedWriting

https://gitee.com/daohonglei/javaStereotypedWriting