



[回到首页](#)

芋道源码 —— 知识星球

我是一段不羁的公告！

记得给芳芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/onemall>

<https://github.com/YunaiV/ruoyi-vue-pro>

2019-09-27

[Spring](#)

【死磕 Spring】—— ApplicationContext 相关接口架构分析

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 <http://cmsblogs.com/?p=todo> 「小明哥」，谢谢！

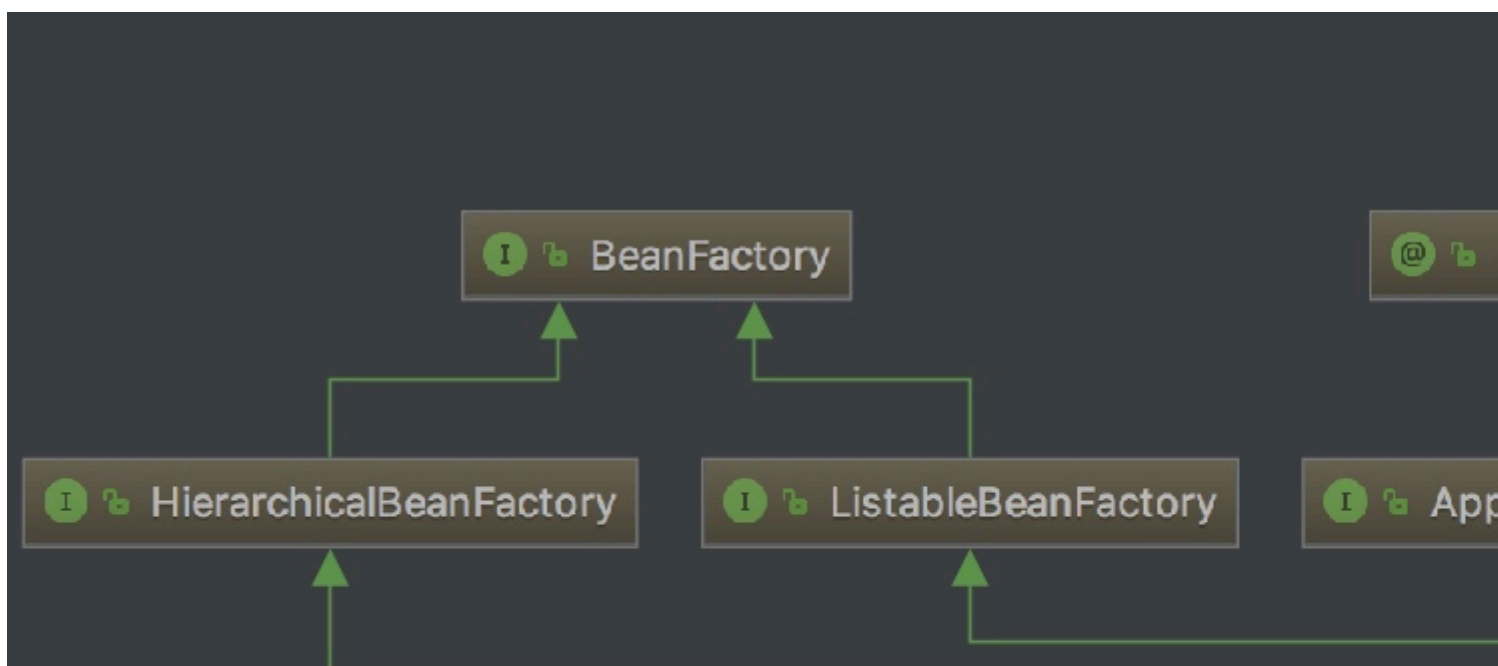
作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

在前面 40 篇博客中都是基于 BeanFactory 这个容器来进行分析的，BeanFactory 容器有点儿简单，它并不适用于我们生产环境，在生产环境我们通常会选择 ApplicationContext，相对于大多数人而言，它才是正规军，相比于 BeanFactory 这个杂牌军而言，它由如下几个区别：

1. 继承 MessageSource，提供国际化的标准访问策略。
2. 继承 ApplicationEventPublisher，提供强大的事件机制。
3. 扩展 ResourceLoader，可以用来加载多个 Resource，可以灵活访问不同的资源。
4. 对 Web 应用的支持。

1. ApplicationContext

下图是 ApplicationContext 结构类图：



BeanFactory: Spring 管理 Bean 的顶层接口，我们可以认为他是一个简易版的 Spring 容器。ApplicationContext 继承 BeanFactory 的两个子类：HierarchicalBeanFactory 和 ListableBeanFactory。HierarchicalBeanFactory 是一个具有层级关系的 BeanFactory，拥有属性 parentBeanFactory。ListableBeanFactory 实现了枚举方法可以列举出当前 BeanFactory 中所有的 bean 对象而不必根据 name 一个一个的获取。

ApplicationEventPublisher: 用于封装事件发布功能的接口，向事件监听器（Listener）发送事件消息。

ResourceLoader: Spring 加载资源的顶层接口，用于从一个源加载资源文件。

ApplicationContext 继承 ResourceLoader 的子类 ResourcePatternResolver，该接口是将 location 解析为 Resource 对象的策略接口。

MessageSource: 解析 message 的策略接口，用不支撑国际化等功能。

EnvironmentCapable: 用于获取 Environment 的接口。

2. ApplicationContext 的子接口

ApplicationContext 有两个直接子类：WebApplicationContext 和 ConfigurableApplicationContext。

2.1 WebApplicationContext

```
// WebApplicationContext.java

public interface WebApplicationContext extends ApplicationContext {

    ServletContext getServletContext();

}
```

该接口只有一个 #getServletContext() 方法，用于给 Servlet 提供上下文信息。

2.2 ConfigurableApplicationContext

```
// ConfigurableApplicationContext.java

public interface ConfigurableApplicationContext extends ApplicationContext, Lifecycle, Closeable {

    // 为 ApplicationContext 设置唯一 ID
    void setId(String id);

    // 为 ApplicationContext 设置 parent
    // 父类不应该被修改：如果创建的对象不可用时，则应该在构造函数外部设置它
    void setParent(@Nullable ApplicationContext parent);

    // 设置 Environment
    void setEnvironment(ConfigurableEnvironment environment);

    // 获取 Environment
    @Override
    ConfigurableEnvironment getEnvironment();

    // 添加 BeanFactoryPostProcessor
```

```

void addBeanFactoryPostProcessor (BeanFactoryPostProcessor postProcessor);

// 添加 ApplicationListener
void addApplicationListener (ApplicationListener<?> listener);

// 添加 ProtocolResolver
void addProtocolResolver (ProtocolResolver resolver);

// 加载或者刷新配置
// 这是一个非常重要的方法
void refresh() throws BeansException, IllegalStateException;

// 注册 shutdown hook
void registerShutdownHook();

// 关闭 ApplicationContext
@Override
void close();

// ApplicationContext 是否处于激活状态
boolean isActive();

// 获取当前上下文的 BeanFactory
ConfigurableListableBeanFactory getBeanFactory() throws IllegalStateException;
}

```

从上面代码可以看到 `ConfigurableApplicationContext` 接口提供的方法都是对 `ApplicationContext` 进行配置的，例如 `#setEnvironment(ConfigurableEnvironment environment)`、`#addBeanFactoryPostProcessor (BeanFactoryPostProcessor postProcessor)`，同时它还继承了如下两个接口：

Lifecycle: 对 context 生命周期的管理，它提供 `#start()` 和 `#stop()` 方法启动和暂停组件。
Closeable: 标准 JDK 所提供的一个接口，用于最后关闭组件释放资源等。

2.3 ConfigurableWebApplicationContext

`WebApplicationContext` 接口和 `ConfigurableApplicationContext` 接口有一个共同的子类接口 `ConfigurableWebApplicationContext`，该接口将这两个接口进行合并，提供了一个可配置、可管理、可关闭的 `WebApplicationContext`，同时该接口还增加了 `#setServletContext(ServletContext servletContext)`，`setServletConfig(ServletConfig servletConfig)` 等方法，用于装配 `WebApplicationContext`。代码如下：

```

// ConfigurableWebApplicationContext.java

public interface ConfigurableWebApplicationContext extends WebApplicationContext, ConfigurableApplicationContext {

    void setServletContext(@Nullable ServletContext servletContext);

    void setServletConfig(@Nullable ServletConfig servletConfig);
    ServletConfig getServletConfig();

    void setNamespace(@Nullable String namespace);
    String getNamespace();

    void setConfigLocation(String configLocation);
}

```

}

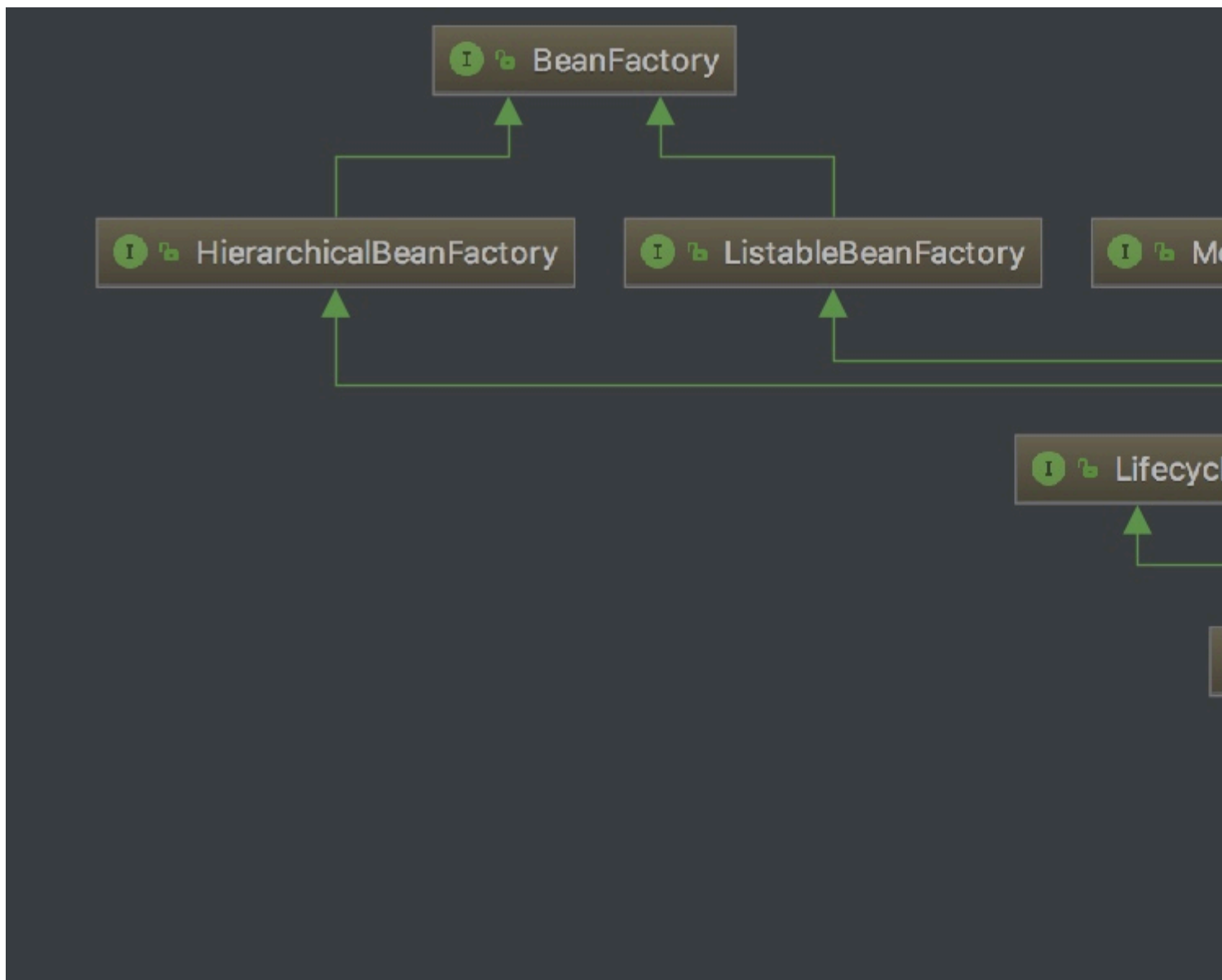
ClassPathXmlApplicationContext 来分析 ApplicationContext。

3. `ClassPathXmlApplicationContext`

个接触的 Spring 容器就是它，包括小编自己，下面代码我想很多人依然还记得吧。

```
StudentService studentService = (StudentService)ac.getBean("studentService");
```

下图是 `ClassPathXmlApplicationContext` 的结构类图:



主要的类层级关系如下：

```
org.springframework.context.support.AbstractApplicationContext
    org.springframework.context.support.AbstractRefreshableApplicationContext
        org.springframework.context.support.AbstractRefreshableConfigApplicationContext
            org.springframework.context.support.AbstractXmlApplicationContext
                org.springframework.context.support.ClassPathXmlApplicationContext
```

这种设计是模板方法模式典型的应用，AbstractApplicationContext 实现了 ConfigurableApplicationContext 这个全家桶接口，其子类 AbstractRefreshableConfigApplicationContext 又实现了 BeanNameAware 和 InitializingBean 接口。所以 ClassPathXmlApplicationContext 设计的顶级接口有：

BeanFactory: Spring 容器 Bean 的管理
MessageSource: 管理 message，实现国际化等功能
ApplicationEventPublisher: 事件发布
ResourcePatternResolver: 资源加载
EnvironmentCapable: 系统 Environment (profile + Properties) 相关
Lifecycle: 管理生命周期
Closable: 关闭，释放资源
InitializingBean: 自定义初始化
BeanNameAware: 设置 beanName 的 Aware 接口

下面就这些接口来一一分析。

3.1 MessageSource

MessageSource 定义了获取 message 的策略方法 #getMessage(...)。

在 ApplicationContext 体系中，该方法由 AbstractApplicationContext 实现。

在 AbstractApplicationContext 中，它持有一个 MessageSource 实例，将 #getMessage(...) 方法委托给该实例来实现，代码如下：

```
// AbstractApplicationContext.java

private MessageSource messageSource;

// 实现 getMessage()
public String getMessage(String code, @Nullable Object[] args, @Nullable String defaultMessage, Locale locale) {
    // 委托给 messageSource 实现
    return getMessageSource().getMessage(code, args, defaultMessage, locale);
}

private MessageSource getMessageSource() throws IllegalStateException {
    if (this.messageSource == null) {
        throw new IllegalStateException("MessageSource not initialized - " + "call 'refresh' before accessing message");
    }
    return this.messageSource;
}
```

真正实现逻辑，是在 AbstractMessageSource 中，代码如下：

```
// AbstractMessageSource.java
```

```

    public final String getMessage(String code, @Nullable Object[] args, @Nullable String defaultMessage, Locale locale) {
        String msg = getMessageInternal(code, args, locale);
        if (msg != null) {
            return msg;
        }
        if (defaultMessage == null) {
            return getDefaultMessage(code);
        }
        return renderDefaultMessage(defaultMessage, args, locale);
    }

```

- 具体的实现这里就不分析了，有兴趣的小伙伴可以自己去深入研究。

3.2 ApplicationEventPublisher

`ApplicationEventPublisher`，用于封装事件发布功能的接口，向事件监听器（`Listener`）发送事件消息。

该接口提供了一个 `#publishEvent(Object event, ...)` 方法，用于通知在此应用程序中注册的所有的监听器。该方法在 `AbstractApplicationContext` 中实现。

```

// AbstractApplicationContext.java

@Override
public void publishEvent(ApplicationEvent event) {
    publishEvent(event, null);
}

@Override
public void publishEvent(Object event) {
    publishEvent(event, null);
}

protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");

    // Decorate event as an ApplicationEvent if necessary
    ApplicationEvent applicationEvent;
    if (event instanceof ApplicationEvent) {
        applicationEvent = (ApplicationEvent) event;
    } else {
        applicationEvent = new PayloadApplicationEvent<>(this, event);
        if (eventType == null) {
            eventType = ((PayloadApplicationEvent) applicationEvent).getResolvableType();
        }
    }

    // Multicast right now if possible - or lazily once the multicaster is initialized
    if (this.earlyApplicationEvents != null) {
        this.earlyApplicationEvents.add(applicationEvent);
    } else {
        getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
    }

    // Publish event via parent context as well...

```

```

        if (this.parent != null) {
            if (this.parent instanceof AbstractApplicationContext) {
                ((AbstractApplicationContext) this.parent).publishEvent(event, eventType);
            } else {
                this.parent.publishEvent(event);
            }
        }
    }
}

```

如果指定的事件不是 `ApplicationEvent`，则它将包装在 `PayloadApplicationEvent` 中。如果存在父级 `ApplicationContext`，则同样要将 `event` 发布给父级 `ApplicationContext`。

3.3 ResourcePatternResolver

`ResourcePatternResolver` 接口继承 `ResourceLoader` 接口，为将 `location` 解析为 `Resource` 对象的策略接口。他提供的 `#getResources(String locationPattern)` 方法，在 `AbstractApplicationContext` 中实现，在 `AbstractApplicationContext` 中他持有一个 `ResourcePatternResolver` 的实例对象。代码如下：

```

// AbstractApplicationContext.java

/** ResourcePatternResolver used by this context. */
private ResourcePatternResolver resourcePatternResolver;

public Resource[] getResources(String locationPattern) throws IOException {
    return this.resourcePatternResolver.getResources(locationPattern);
}

```

如果小伙伴对 Spring 的 `ResourceLoader` 比较熟悉的话，你会发现最终是在 `PathMatchingResourcePatternResolver` 中实现，该类是 `ResourcePatternResolver` 接口的实现者。

3.4 EnvironmentCapable

提供当前系统环境 `Environment` 组件。提供了一个 `#getEnvironment()` 方法，用于返回 `Environment` 实例对象。该方法在 `AbstractApplicationContext` 实现。代码如下：

```

// AbstractApplicationContext.java
public ConfigurableEnvironment getEnvironment() {
    if (this.environment == null) {
        this.environment = createEnvironment();
    }
    return this.environment;
}

```

如果持有的 `environment` 实例对象为空，则调用 `#createEnvironment()` 方法，创建一个。代码如下：

```

// AbstractApplicationContext.java

```

```
protected ConfigurableEnvironment createEnvironment() {
    return new StandardEnvironment();
}
```

- StandardEnvironment 是一个适用于非 WEB 应用的 Environment。

3.5 Lifecycle

Lifecycle，一个用于管理声明周期的接口。

在 AbstractApplicationContext 中存在一个 LifecycleProcessor 类型的实例对象 lifecycleProcessor，AbstractApplicationContext 中关于 Lifecycle 接口的实现都是委托给 lifecycleProcessor 实现的。代码如下：

```
// AbstractApplicationContext.java

/** LifecycleProcessor for managing the lifecycle of beans within this context. */
@Nullable
private LifecycleProcessor lifecycleProcessor;

@Override
public void start() {
    getLifecycleProcessor().start();
    publishEvent(new ContextStartedEvent(this));
}

@Override
public void stop() {
    getLifecycleProcessor().stop();
    publishEvent(new ContextStoppedEvent(this));
}

@Override
public boolean isRunning() {
    return (this.lifecycleProcessor != null && this.lifecycleProcessor.isRunning());
}
```

在启动、停止的时候会分别发布 ContextStartedEvent 和 ContextStoppedEvent 事件。

3.6 Closable

Closable 接口用于关闭和释放资源，提供了 #close() 方法，以释放对象所持有的资源。在 ApplicationContext 体系中由 AbstractApplicationContext 实现，用于关闭 ApplicationContext 销毁所有 Bean，此外如果注册有 JVM shutdown hook，同样要将其移除。代码如下：

```
// AbstractApplicationContext.java

public void close() {
    synchronized (this.startupShutdownMonitor) {
        doClose();
        // If we registered a JVM shutdown hook, we don't need it anymore now:
    }
}
```



```

        // We've already explicitly closed the context.
        if (this.shutdownHook != null) {
            try {
                Runtime.getRuntime().removeShutdownHook(this.shutdownHook);
            } catch (IllegalStateException ex) {
                // ignore - VM is already shutting down
            }
        }
    }
}

```

调用 `#doClose()` 方法，发布 `ContextClosedEvent` 事件，销毁所有 Bean（单例），关闭 `BeanFactory`。代码如下：

```

// AbstractApplicationContext.java

protected void doClose() {
    // ... 省略部分代码
    try {
        // Publish shutdown event.
        publishEvent(new ContextClosedEvent(this));
    } catch (Throwable ex) {
        logger.warn("Exception thrown from ApplicationListener handling ContextClosedEvent", ex);
    }

    // ... 省略部分代码
    destroyBeans();
    closeBeanFactory();
    onClose();

    this.active.set(false);
}

```

3.7 InitializingBean

`InitializingBean` 为 Bean 提供了初始化方法的方式，它提供的 `#afterPropertiesSet()` 方法，用于执行初始化动作。在 `ApplicationContext` 体系中，该方法由 `AbstractRefreshableConfigApplicationContext` 实现，代码如下：

```

// AbstractRefreshableConfigApplicationContext.java
public void afterPropertiesSet() {
    if (!isActive()) {
        refresh();
    }
}

```

执行 `refresh()` 方法，该方法在 `AbstractApplicationContext` 中执行，执行整个 Spring 容器的初始化过程。该方法将在下篇文章进行详细分析说明。

3.8 BeanNameAware

`BeanNameAware`，设置 Bean Name 的接口。接口在

AbstractRefreshableConfigApplicationContext 中实现。

```
// AbstractRefreshableConfigApplicationContext.java
public void setBeanName(String name) {
    if (!this.setCalled) {
        super.setBeanName(name);
        setDisplayName("ApplicationContext '" + name + "'");
    }
}
```

4. 小结

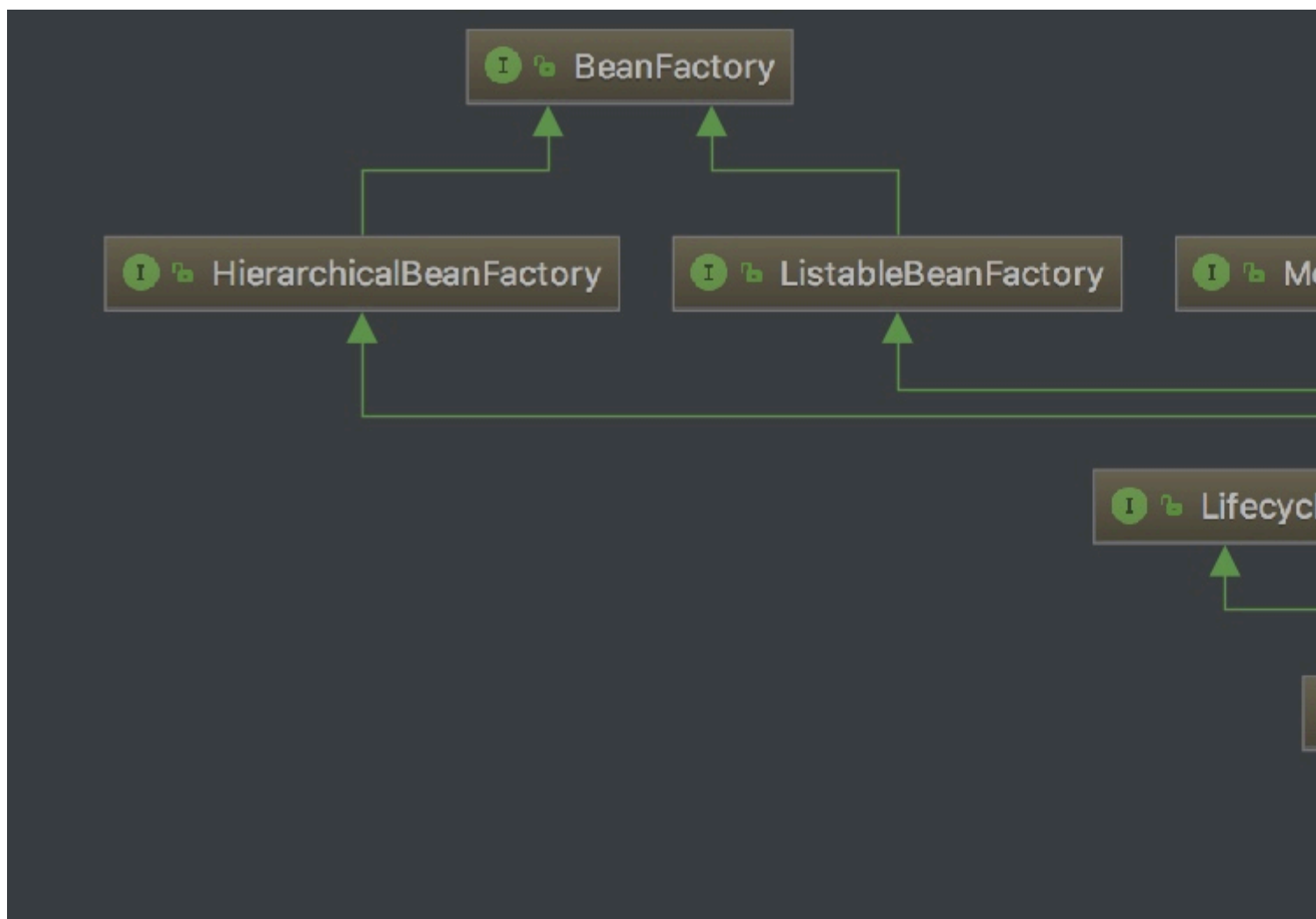
由于篇幅问题，再加上大部分接口小编都已经在前面文章进行了详细的阐述，所以本文主要是以 Spring Framework 的 ApplicationContext 为中心，对其结构和功能的实现进行了简要的说明。

这里不得不说 Spring 真的是一个非常优秀的框架，具有良好的结构设计和接口抽象，它的每一个接口职能单一，且都是具体功能到各个模块的高度抽象，且几乎每套接口都提供了一个默认的实现（defaultXXX）。

对于 ApplicationContext 体系而言，他继承 Spring 中众多的核心接口，能够为客户端提供一个相对完整的 Spring 容器，接口 ConfigurableApplicationContext 对 ApplicationContext 接口再次进行扩展，提供了生命周期的管理功能。

抽象类 ApplicationContext 对整套接口提供了大部分的默认实现，将其中“不易变动”的部分进行了封装，通过“组合”的方式将“容易变动”的功能委托给其他类来实现，同时利用模板方法模式将一些方法的实现开放出去由子类实现，从而实现“对扩展开放，对修改封闭”的设计原则。

最后我们再来领略下图的风采：



文章目录

1. [1. 1. ApplicationContext](#)
2. [2. 2. ApplicationContext 的子接口](#)
 1. [2. 1. 2. 1 WebApplicationContext](#)
 2. [2. 2. 2. 2 ConfigurableApplicationContext](#)
 3. [2. 3. 2. 3 ConfigurableWebApplicationContext](#)
3. [3. 3. ClassPathXmlApplicationContext](#)
 1. [3. 1. 3. 1 MessageSource](#)
 2. [3. 2. 3. 2 ApplicationEventPublisher](#)
 3. [3. 3. 3. 3 ResourcePatternResolver](#)
 4. [3. 4. 3. 4 EnvironmentCapable](#)
 5. [3. 5. 3. 5 Lifecycle](#)
 6. [3. 6. 3. 6 Closable](#)
 7. [3. 7. 3. 7 InitializingBean](#)
 8. [3. 8. 3. 8 BeanNameAware](#)
4. [4. 4. 小结](#)

2014 - 2023 芋道源码 |
总访客数 次 && 总访问量 次
[回到首页](#)