无

> 1. 概述本文，我们来补充 《精尽 Spring Boot 源码分析 —— SpringApplication》 文章，并未详细解析的 ApplicationContextInitializer 。

本文，我们来补充 《精尽 Spring Boot 源码分析 —— SpringApplication》 文章，并未详细解析的 ApplicationContextInitializer 。

`org.springframework.context.ApplicationContextInitializer` ，ApplicationContext 初始化接口。代码如下：

```
public interface ApplicationContextInitializer<C extends ConfigurableApplicationContext> {



    void initialize(C applicationContext);

}
```

- 1、ApplicationContextInitializer 是 Spring Framework 3.1 版本开始提供的接口。而本文，我们是来分享 Spring Boot 中，几个 ApplicationContextInitializer 实现类。
- 2、【作用】ApplicationContextInitializer 是一个回调接口，用于 Spring ConfigurableApplicationContext 容器执行 `#refresh()` 方法进行初始化之前，提前走一些自定义的初始化逻辑。
- 3、【场景】它的使用场景，例如说 Web 应用中需要注册属性，或者激活 Profiles 。
- 4、【排序】它支持 Spring 的 Ordered 接口、 `@Order` 注解，来对多个 ApplicationContextInitializer 实例进行排序，从而实现，ApplicationContextInitializer 按照顺序调用 `#initialize(C applicationContext)` 方法，进行初始化。

3.1 初始化 ApplicationContextInitializer 集合

在 SpringApplication 构造方法中，会调用 `#getSpringFactoriesInstances(Class<T> type)` 方法，获得 ApplicationContextInitializer 集合。代码如下：

```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type) {
    return getSpringFactoriesInstances(type, new Class<?>[] {});
}

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
        Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();


    Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader));

    List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names);

    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

- `<1>` 处，加载指定 ApplicationContextInitializer 类型对应的，在 `META-INF/spring.factories` 里的类名的数组。
- `<2>` 处，创建对象们。
- `<3>` 处，排序对象们。这个就是在 「2. ApplicationContextInitializer」 提到的【排序】。

3.2 prepareContext

在 `#prepareContext(...)` 方法中，即在 Spring IoC 容器初始化之前，会调用 `#applyInitializers()` 方法，逐个调用 ApplicationContextInitializer 的初始化方法。代码如下：

```
protected void applyInitializers(ConfigurableApplicationContext context) {

    for (ApplicationContextInitializer initializer : getInitializers()) {

        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(initializer.getClass(), ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");

        initializer.initialize(context);
    }
}
```

- 比较简单，就是调用 ApplicationContextInitializer#initialize(context) 方法，进行初始化。

下面，我们来逐个看看 Spring Boot 对 ApplicationContextInitializer 的实现类们。

org.springframework.boot.context.config.DelegatingApplicationContextInitializer ，实现 ApplicationContextInitializer、Ordered 接口，根据环境变量配置的 context.initializer.classes 配置的 ApplicationContextInitializer 类们，交给它们进行初始化。

4.1 构造方法

```
private static final String PROPERTY_NAME = "context.initializer.classes";



private int order = 0;



@Override
public int getOrder() {
        return this.order;
}
```

- 优先级为 0 ，在 Spring Boot 默认的 ApplicationContextInitializer 实现类中，是排在最前面的。

4.2 initialize

实现 #initialize(ConfigurableApplicationContext context) 方法，代码如下：

```
@Override
public void initialize(ConfigurableApplicationContext context) {

    ConfigurableEnvironment environment = context.getEnvironment();
    List<Class<?>> initializerClasses = getInitializerClasses(environment);

    if (!initializerClasses.isEmpty()) {
        applyInitializerClasses(context, initializerClasses);
    }
}
```

- <1> 处，调用 #getInitializerClasses(ConfigurableEnvironment env) 方法，获得环境变量配置的 ApplicationContextInitializer 集合们。代码如下：

```
private List<Class<?>> getInitializerClasses(ConfigurableEnvironment env) {

    String classNames = env.getProperty(PROPERTY_NAME);

    List<Class<?>> classes = new ArrayList<>();
    if (StringUtils.hasLength(classNames)) {
        for (String className : StringUtils.tokenizeToStringArray(classNames, ",")) {
            classes.add(getInitializerClass(className));
        }
    }
    return classes;
}

private Class<?> getInitializerClass(String className) throws LinkageError {
        try {
```

```
                Class<?> initializerClass = ClassUtils.forName(className, ClassUtils.getDefaultClassLoader());
                Assert.isAssignable(ApplicationContextInitializer.class, initializerClass);
                return initializerClass;
        } catch (ClassNotFoundException ex) {
                throw new ApplicationContextException("Failed to load context initializer class [" + className + "]", ex);
        }
}
```

- <2> 处，调用 #applyInitializerClasses(ConfigurableApplicationContext context, List<Class<?>> initializerClasses) 方法，执行初始化。代码如下：

```
private void applyInitializerClasses(ConfigurableApplicationContext context, List<Class<?>> initializerClasses) {
    Class<?> contextClass = context.getClass();

    List<ApplicationContextInitializer<?>> initializers = new ArrayList<>();
    for (Class<?> initializerClass : initializerClasses) {
        initializers.add(instantiateInitializer(contextClass, initializerClass));
    }

    applyInitializers(context, initializers);
}


private ApplicationContextInitializer<?> instantiateInitializer(Class<?> contextClass, Class<?> initializerClass) {

    Class<?> requireContextClass = GenericTypeResolver.resolveTypeArgument(initializerClass, ApplicationContextInitializer.class);
    Assert.isAssignable(requireContextClass, contextClass, String.format(
                    "Could not add context initializer [%s]"
                            + " as its generic parameter [%s] is not assignable "
                            + "from the type of application context used by this "
                            + "context loader [%s]: ",
                    initializerClass.getName(), requireContextClass.getName(), contextClass.getName()));

    return (ApplicationContextInitializer<?>) BeanUtils.instantiateClass(initializerClass);
}


@SuppressWarnings({ "unchecked", "rawtypes" })
private void applyInitializers(ConfigurableApplicationContext context, List<ApplicationContextInitializer<?>> initializers) {

    initializers.sort(new AnnotationAwareOrderComparator());

    for (ApplicationContextInitializer initializer : initializers) {
        initializer.initialize(context);
    }
}
```

    - 虽然代码有点长，但是简单的。

org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer ，实现 ApplicationContextInitializer、Ordered 接口，它会创建一个用于在 ConfigurationClassPostProcessor 和 Spring Boot 间共享的 CachingMetadataReaderFactory Bean 对象。

简化代码如下：

```
public static final String BEAN_NAME = "org.springframework.boot.autoconfigure."
        + "internalCachingMetadataReaderFactory";

@Override
public void initialize(ConfigurableApplicationContext applicationContext) {
    applicationContext.addBeanFactoryPostProcessor(new CachingMetadataReaderFactoryPostProcessor());
}

@Override
public int getOrder() {
    return 0;
}
```

- 看不懂这个类的具体用途，暂时先不去深究。😈 等真的需要它时，咱们在细细的撸它。

org.springframework.boot.context.ContextIdApplicationContextInitializer ，实现 ApplicationContextInitializer、Ordered 接口，负责生成 Spring 容器的编号。

6.1 构造方法

```
private int order = Ordered.LOWEST_PRECEDENCE - 10;

@Override
public int getOrder() {
        return this.order;
}
```

6.2 initialize

实现 #initialize(ConfigurableApplicationContext context) 方法，代码如下：

```
@Override
public void initialize(ConfigurableApplicationContext applicationContext) {

    ContextId contextId = getContextId(applicationContext);

    applicationContext.setId(contextId.getId());

    applicationContext.getBeanFactory().registerSingleton(ContextId.class.getName(), contextId);
}
```

- ContextId ，是 ContextIdApplicationContextInitializer 的内部类，Spring 容器编号的封装。代码如下：

```
class ContextId {


        private final AtomicLong children = new AtomicLong(0);

        private final String id;

        ContextId(String id) {
                this.id = id;
        }



        ContextId createChildId() {
                return new ContextId(this.id + "-" + this.children.incrementAndGet());
        }

        String getId() {
                return this.id;
        }

}
```

- <1> 处，调用 #getContextId(ConfigurableApplicationContext applicationContext) 方法，获得（创建） ContextId 对象。代码如下：

```
private ContextId getContextId(ConfigurableApplicationContext applicationContext) {

    ApplicationContext parent = applicationContext.getParent();

    if (parent != null && parent.containsBean(ContextId.class.getName())) {
        return parent.getBean(ContextId.class).createChildId();
    }

    return new ContextId(getApplicationId(applicationContext.getEnvironment()));
}

private String getApplicationId(ConfigurableEnvironment environment) {
    String name = environment.getProperty("spring.application.name");
    return StringUtils.hasText(name) ? name : "application";
}
```

  - 一般情况下，使用 "spring.application.name" 环境变量，作为 ContextId 对象的 id 属性。

- `<2>` 处，设置到 `applicationContext.id` 中。
- `<3>` 处，注册到 `contextId` 到 Spring 容器中。这样，后续就可以拿到了。

> 芳芳：对于这个类，选择性了解即可。

`org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer` ，实现 ApplicationContextInitializer 接口，用于检查配置，报告错误的配置。如下是其类上的注释：

7.1 initialize

实现 `#initialize(ConfigurableApplicationContext applicationContext)` 方法，代码如下：

```
@Override
public void initialize(ConfigurableApplicationContext context) {

	context.addBeanFactoryPostProcessor(new ConfigurationWarningsPostProcessor(getChecks()));
}
```

- 注册 ConfigurationWarningsPostProcessor 到 Spring 容器中。关于 ConfigurationWarningsPostProcessor 类，在 「7.2 ConfigurationWarningsPostProcessor」 中，详细解析。
- 其中， `#getChecks()` 方法，返回 Check 数组。代码如下：

  ```
  protected Check[] getChecks() {
  	return new Check[] { new ComponentScanPackageCheck() };
  }
  ```

  - 返回的数组，只有一个 ComponentScanPackageCheck 对象。
- Check ，是 ConfigurationWarningsApplicationContextInitializer 的内部接口，校验器。代码如下：

  ```
  @FunctionalInterface
  protected interface Check {

  	String getWarning(BeanDefinitionRegistry registry);

  }
  ```

- 看到此处，胖友可能有点懵逼，不着急。在 「7.2 ConfigurationWarningsPostProcessor」 中，会串起来滴。

7.2 ConfigurationWarningsPostProcessor

ConfigurationWarningsPostProcessor ，是 ConfigurationWarningsApplicationContextInitializer 的内部静态类，实现 PriorityOrdered、BeanDefinitionRegistryPostProcessor 接口，代码如下：

```
protected static final class ConfigurationWarningsPostProcessor
		implements PriorityOrdered, BeanDefinitionRegistryPostProcessor {

    private Check[] checks;

    public ConfigurationWarningsPostProcessor(Check[] checks) {
        this.checks = checks;
    }

    @Override
    public int getOrder() {
        return Ordered.LOWEST_PRECEDENCE - 1;
    }
```

```
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {}

    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {

        for (Check check : this.checks) {
            String message = check.getWarning(registry);
            if (StringUtils.hasLength(message)) {
                warn(message);
            }
        }

    }

    private void warn(String message) {
        if (logger.isWarnEnabled()) {
            logger.warn(String.format("%n%n** WARNING ** : %s%n%n", message));
        }
    }

}
```

- 核心就是 `#postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)` 方法。在其内部，遍历 Check 数组，执行校验。若有错，则打印 warn 日志。在上文中，我们看到目前 `checks` 只有一个 ComponentScanPackageCheck 元素。关于它，我们在 「7.3 ComponentScanPackageCheck」 详细解析。

7.3 ComponentScanPackageCheck

ComponentScanPackageCheck ，实现 Check 接口，检查是否使用了 `@ComponentScan` 注解，扫描了指定扫描的包。

7.3.1 构造方法

```
private static final Set<String> PROBLEM_PACKAGES;

static {
    Set<String> packages = new HashSet<>();
    packages.add("org.springframework");
    packages.add("org");
    PROBLEM_PACKAGES = Collections.unmodifiableSet(packages);
}
```

- 即禁止扫描 `"org.springframework"` 和 `"org"` 包。因为 `"org.springframework"` 包下，有非常多的 Bean ，这样扫描，会错误的注入很多 Bean 。

7.3.2 getWarning

实现 `#getWarning(BeanDefinitionRegistry registry)` 方法，代码如下：

```
@Override
public String getWarning(BeanDefinitionRegistry registry) {

    Set<String> scannedPackages = getComponentScanningPackages(registry);

    List<String> problematicPackages = getProblematicPackages(scannedPackages);

    if (problematicPackages.isEmpty()) {
        return null;
    }

    return "Your ApplicationContext is unlikely to "
            + "start due to a @ComponentScan of "
            + StringUtils.collectionToDelimitedString(problematicPackages, ", ")
            + ".";
}
```

- `<1>` 处，调用 `#getComponentScanningPackages(BeanDefinitionRegistry registry)` 方法，获得要扫描的包。代码如下：

```
protected Set<String> getComponentScanningPackages(BeanDefinitionRegistry registry) {
```

```
    Set<String> packages = new LinkedHashSet<>();

    String[] names = registry.getBeanDefinitionNames();
    for (String name : names) {

        BeanDefinition definition = registry.getBeanDefinition(name);
        if (definition instanceof AnnotatedBeanDefinition) {
            AnnotatedBeanDefinition annotatedDefinition = (AnnotatedBeanDefinition) definition;

            addComponentScanningPackages(packages, annotatedDefinition.getMetadata());
        }
    }
    return packages;
}

private void addComponentScanningPackages(Set<String> packages, AnnotationMetadata metadata) {

    AnnotationAttributes attributes = AnnotationAttributes.fromMap(metadata.getAnnotationAttributes(ComponentScan.class.getName(), true));

    if (attributes != null) {
        addPackages(packages, attributes.getStringArray("value"));
        addPackages(packages, attributes.getStringArray("basePackages"));
        addClasses(packages, attributes.getStringArray("basePackageClasses"));
        if (packages.isEmpty()) {
            packages.add(ClassUtils.getPackageName(metadata.getClassName()));
        }
    }
}

private void addPackages(Set<String> packages, String[] values) {
    if (values != null) {
        Collections.addAll(packages, values);
    }
}

private void addClasses(Set<String> packages, String[] values) {
    if (values != null) {
        for (String value : values) {
            packages.add(ClassUtils.getPackageName(value));
        }
    }
}
```

    - 虽然很长，但是比较简单。就是找 `@ComponentScan` 注解所扫描的包。

- `<2>` 处，调用 `#getProblematicPackages(Set<String> scannedPackages)` 方法，获得要扫描的包中，有问题的包。代码如下：

```
private List<String> getProblematicPackages(Set<String> scannedPackages) {

    List<String> problematicPackages = new ArrayList<>();
    for (String scannedPackage : scannedPackages) {

        if (isProblematicPackage(scannedPackage)) {
            problematicPackages.add(getDisplayName(scannedPackage));
        }
    }
    return problematicPackages;
}

private boolean isProblematicPackage(String scannedPackage) {
    if (scannedPackage == null || scannedPackage.isEmpty()) {
        return true;
    }
    return PROBLEM_PACKAGES.contains(scannedPackage);
}

private String getDisplayName(String scannedPackage) {
    if (scannedPackage == null || scannedPackage.isEmpty()) {
        return "the default package";
    }
    return "'" + scannedPackage + "'";
}
```

    - 😈 就是判断 `scannedPackages` 哪些在 `PROBLEM_PACKAGES` 中。

- `<3.1>` 处，如果 `problematicPackages` 为空，说明不存在问题。

- `<3.2>` 处，如果 `problematicPackages` 非空，说明有问题，返回错误提示。

org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer ，实现 ApplicationContextInitializer、ApplicationListener 接口，监听 EmbeddedServletContainerInitializedEvent 类型的事件，然后将内嵌的 Web 服务器使用的端口给设置到 ApplicationContext 中。

8.1 initialize

实现 #initialize(ConfigurableApplicationContext applicationContext) 方法，代码如下：

```
@Override
public void initialize(ConfigurableApplicationContext applicationContext) {
        applicationContext.addApplicationListener(this);
}
```

- 将自身作为一个 ApplicationListener 监听器，添加到 Spring 容器中。

8.2 onApplicationEvent

实现 #onApplicationEvent(WebServerInitializedEvent event) 方法，当监听到 WebServerInitializedEvent 事件，进行触发。代码如下：

```
@Override
public void onApplicationEvent(WebServerInitializedEvent event) {

    String propertyName = "local." + getName(event.getApplicationContext()) + ".port";

    setPortProperty(event.getApplicationContext(), propertyName, event.getWebServer().getPort());
}
```

- <1> 处，获得属性名。其中， #getName(WebServerApplicationContext context) 方法，获得 WebServer 的名字。代码如下：

```
private String getName(WebServerApplicationContext context) {
        String name = context.getServerNamespace();
        return StringUtils.hasText(name) ? name : "server";
}
```

- <2> 处，调用 #setPortProperty(ApplicationContext context, String propertyName, int port) 方法，设置端口到 environment 的 propertyName 中。代码如下：

```
private void setPortProperty(ApplicationContext context, String propertyName, int port) {

    if (context instanceof ConfigurableApplicationContext) {
        setPortProperty(((ConfigurableApplicationContext) context).getEnvironment(), propertyName, port);
    }

    if (context.getParent() != null) {
        setPortProperty(context.getParent(), propertyName, port);
    }
}

@SuppressWarnings("unchecked")
private void setPortProperty(ConfigurableEnvironment environment, String propertyName, int port) {
    MutablePropertySources sources = environment.getPropertySources();

    PropertySource<?> source = sources.get("server.ports");
    if (source == null) {
        source = new MapPropertySource("server.ports", new HashMap<>());
        sources.addFirst(source);
    }

    ((Map<String, Object>) source.getSource()).put(propertyName, port);
}
```

  - 注意噢，设置的属性结果是， "server.ports" 中，的 KEY 为 propertyName ，VALUE 为 port 。😈

ApplicationContextInitializer 还有一些其它实现类，不是很重要，可以选择不看。

- spring-boot-test 模块
  - org.springframework.boot.test.context.ConfigFileApplicationContextInitializer 类
  - org.springframework.boot.test.context.SpringBootContextLoader 中的 ParentContextApplicationContextInitializer 类

- spring-boot-devtools 模块
    - org.springframework.boot.devtools.restart.RestartScopeInitializer 类
- spring-boot-autoconfigure 模块
    - org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener 类
- spring-boot 模块
    - org.springframework.boot.web.servlet.support.ServletContextApplicationContextInitializer 类
    - org.springframework.boot.builder.ParentContextApplicationContextInitializer 类

😈 貌似还是蛮多的。

小更一下，算是水文~ 感觉比较有收获的 ApplicationContextInitializer 的实现类是：

- 「6. ContextIdApplicationContextInitializer」
- 「8. ServerPortInfoApplicationContextInitializer」

参考和推荐如下文章：

- dm_vincent 《[Spring Boot] 5. Spring Boot 中的 ApplicationContext - 执行 ApplicationContextInitializer 初始化器》