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 ApplicationListener 。

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 ApplicationListener 。

org.springframework.context.ApplicationListener ，应用事件监听器接口。代码如下：

```
@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

	void onApplicationEvent(E event);

}
```

- 如果对这块不了解的胖友，可以看看 《Spring 5 源码解析 —— Spring 框架中的事件和监听器》 文章。

2.1 SmartApplicationListener

org.springframework.context.event.SmartApplicationListener 接口，实现 ApplicationListener、Ordered 接口，是 Spring3.0 新增的接口，提供了事件类型和来源的判断接口方法。代码如下：

```
public interface SmartApplicationListener extends ApplicationListener<ApplicationEvent>, Ordered {

	boolean supportsEventType(Class<? extends ApplicationEvent> eventType);

	default boolean supportsSourceType(@Nullable Class<?> sourceType) {
		return true;
	}

	@Override
	default int getOrder() {
		return LOWEST_PRECEDENCE;
	}

}
```

2.1 GenericApplicationListener

org.springframework.context.event.GenericApplicationListener ，继承 ApplicationListener、Ordered 接口，是 Spring4.2 新增的接口，它增强了对泛型的支持， #supportsEventType(ResolvableType) 方法的参数采用的是可解析类型 ResolvableType 。代码如下：

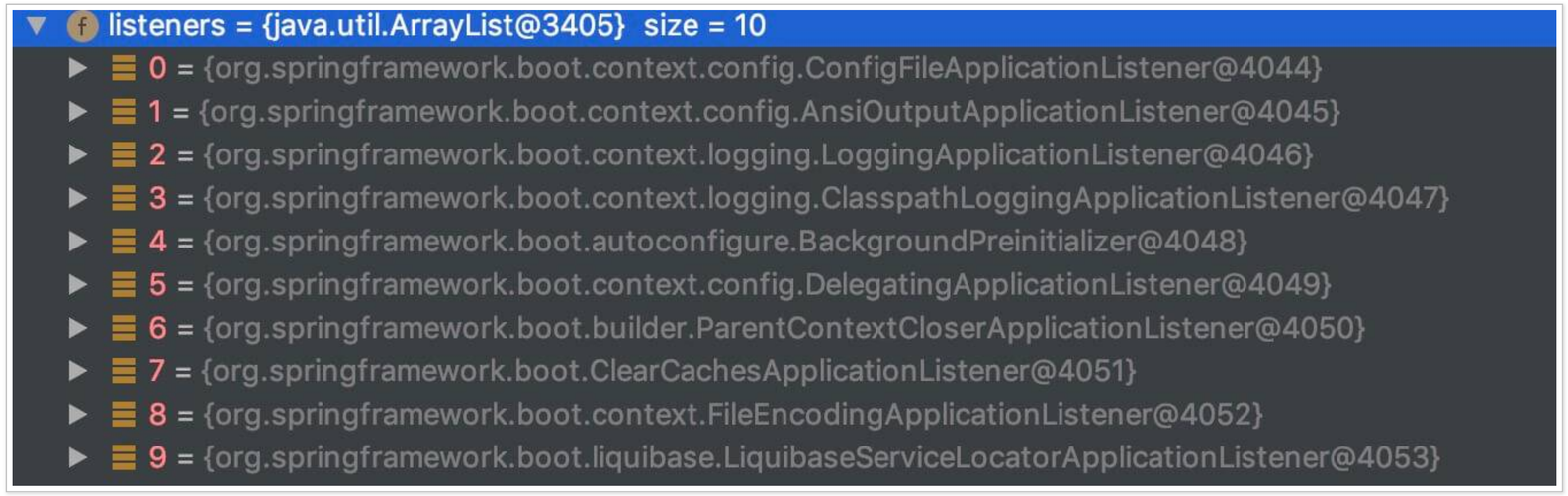
> ResolvableType 是 Spring4 提供的泛型操作支持类，通过它可以很容易地获得泛型的实际类型信息，比如类级、字段级等等泛型信息。在 Spring4 的框架中，很多核心类内部涉及的泛型操作大都使用 ResolvableType 类进行处理。

```
public interface GenericApplicationListener extends ApplicationListener<ApplicationEvent>, Ordered {



	boolean supportsEventType(ResolvableType eventType);



	default boolean supportsSourceType(@Nullable Class<?> sourceType) {
		return true;
	}



	@Override
	default int getOrder() {
		return LOWEST_PRECEDENCE;
	}

}
```

在 SpringApplication 构造方法中，会调用 `#getSpringFactoriesInstances(Class<T> type)` 方法，获得 ApplicationListener 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 ApplicationListener 类型对应的，在 `META-INF/spring.factories` 里的类名的数组。
  - 假设只在 Spring MVC 的环境下， `listeners` 属性的结果如下图：



`listeners` 属性
    - 芮芮整理了 Spring Boot 中，ApplicationContextInitializer 的实现类们，非常多。本文，我们就分享上述的 10 个。
- `<2>` 处，创建对象们。
- `<3>` 处，排序对象们。

`org.springframework.boot.context.config.ConfigFileApplicationListener` ，实现 SmartApplicationListener、Ordered、EnvironmentPostProcessor 接口，实现 Spring Boot 配置文件的加载。

考虑到它非常重要，且复杂，我们单独在 《精尽 Spring Boot 源码分析 —— 配置加载》 详细解析。

`org.springframework.boot.context.config.AnsiOutputApplicationListener` ，实现 ApplicationListener、Ordered 接口，在 Spring Boot 环境变量（`environment`）准备完成以后运行，如果你的终端支持 ANSI，设置彩色输出会让日志更具可读性。

> 不了解 “彩色输出” 的胖友，可以看看 《Spring Boot 日志管理》 文章。

代码如下：

```
public class AnsiOutputApplicationListener
        implements ApplicationListener<ApplicationEnvironmentPreparedEvent>, Ordered {

    @Override
    public void onApplicationEvent(ApplicationEnvironmentPreparedEvent event) {


        ConfigurableEnvironment environment = event.getEnvironment();

        Binder.get(environment).bind("spring.output.ansi.enabled", AnsiOutput.Enabled.class)
        .ifBound(AnsiOutput::setEnabled);
        AnsiOutput.setConsoleAvailable(environment.getProperty("spring.output.ansi.console-available", Boolean.class));
    }

    @Override
    public int getOrder() {

        return ConfigFileApplicationListener.DEFAULT_ORDER + 1;
    }

}
```

- `<1>` 处，我们可以知道，监听的是 ApplicationEnvironmentPreparedEvent 事件。
- `<2>` 处，根据环境变量 `"spring.output.ansi.enabled"` 的值，设置 `AnsiOutput.enabled` 属性。这块的逻辑，卡了芮芮很久，一点一点来说。
    - 首先，因为芮芮并没有细看 Binder 的实现代码，所以这块就卡了一会。简单来说， `Binder.get(environment).bind("spring.output.ansi.enabled", AnsiOutput.Enabled.class)` 代码块的意思是，从 `environment` 读取 `"spring.output.ansi.enabled"` 对应的值，并转换成 AnsiOutput.Enabled 类型。其中，AnsiOutput.Enabled 的枚举值如下：

```
public enum Enabled {


    DETECT,


    ALWAYS,
```

```
        NEVER
    }
```

- 貌似也没什么问题。但是，让艿艿闷逼的是，为什么结果会是 `AnsiOutput.Enabled.ALWAYS` ，在 IDEA 环境中。后来，在 `environment` 中，一个名字是 `"systemProperties"` 的 MapPropertySource 属性源，里面提供了 `"spring.output.ansi.enabled=always"` 的配置。
  - 各种全文检索代码，貌似除了测试用例里，没有地方强制赋值了 `"spring.output.ansi.enabled"` 。
  - 后来发现， `"systemProperties"` 这个 MapPropertySource 属性源，读取的是 `System#getProperties()` 方法，但是为啥里面会有 `"spring.output.ansi.enabled=always"` 呢？目前的猜测是，IDEA 判断在 Spring Boot 环境下，自动添加进去的！
    - 然后， `.ifBound(AnsiOutput::setEnabled)` 代码段，应该翻译成如下的代码，可能比较易懂：

```
Binder.get(environment).bind("spring.output.ansi.enabled", AnsiOutput.Enabled.class)
.ifBound(new Consumer<Enabled>() {
    @Override
    public void accept(Enabled enabled) {
        AnsiOutput.setEnabled(enabled);
    }
});
```

- 即， `Binder.get(environment).bind("spring.output.ansi.enabled", AnsiOutput.Enabled.class)` 返回的是 BindResult 对象，然后调用 `BindResult#ifBound(Consumer)` 方法，将解析到的属性值，赋值到 `AnsiOutput.enabled` 属性。
  - 😈 真的是有点绕噢。

- `<3>` 处，根据环境变量 `"spring.output.ansi.console-available"` 的值，设置 `AnsiOutput.consoleAvailable` 属性。

通过这样的方式，后续在 IDEA 中，我们就可以发现日志打印出来的，是带有彩色输出的。如果胖友是个调皮的人，可以尝试打开 `<X>` 处的三行注释，然后重新运行，就神奇的发现，彩色输出不见了，嘿嘿嘿。

`org.springframework.boot.context.logging.LoggingApplicationListener` ，实现 GenericApplicationListener 接口，实现根据配置初始化日志系统 Logger 。

考虑到它非常重要，且复杂，我们单独在 《精尽 Spring Boot 源码分析 —— 日志系统》 详细解析。

`org.springframework.boot.context.logging.ClasspathLoggingApplicationListener` ，实现 GenericApplicationListener 接口，程序启动时，将 classpath 打印到 debug 日志，启动失败时 classpath 打印到 debug 日志。代码如下：

```
public final class ClasspathLoggingApplicationListener implements GenericApplicationListener {



	private static final int ORDER = LoggingApplicationListener.DEFAULT_ORDER + 1;

	private static final Log logger = LogFactory.getLog(ClasspathLoggingApplicationListener.class);

	@Override
	public void onApplicationEvent(ApplicationEvent event) {
		if (logger.isDebugEnabled()) {

			if (event instanceof ApplicationEnvironmentPreparedEvent) {
				logger.debug("Application started with classpath: " + getClasspath());

			} else if (event instanceof ApplicationFailedEvent) {
				logger.debug("Application failed to start with classpath: " + getClasspath());
			}
		}
	}

	@Override
	public int getOrder() {
		return ORDER;
	}

	@Override
	public boolean supportsEventType(ResolvableType resolvableType) {
		Class<?> type = resolvableType.getRawClass();
		if (type == null) {
			return false;
		}
```

```
                return ApplicationEnvironmentPreparedEvent.class.isAssignableFrom(type)
                                || ApplicationFailedEvent.class.isAssignableFrom(type);
        }



        private String getClasspath() {
                ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
                if (classLoader instanceof URLClassLoader) {
                        return Arrays.toString(((URLClassLoader) classLoader).getURLs());
                }

                return "unknown";
        }

}
```

org.springframework.boot.autoconfigure.BackgroundPreinitializer ，实现 ApplicationListener 接口，实现后台提前执行耗时的初始化任务。代码如下：

```
@Order(LoggingApplicationListener.DEFAULT_ORDER + 1)
public class BackgroundPreinitializer implements ApplicationListener<SpringApplicationEvent> {



    public static final String IGNORE_BACKGROUNDPREINITIALIZER_PROPERTY_NAME = "spring.backgroundpreinitializer.ignore";


    private static final AtomicBoolean preinitializationStarted = new AtomicBoolean(false);


    private static final CountDownLatch preinitializationComplete = new CountDownLatch(1);

    @Override
    public void onApplicationEvent(SpringApplicationEvent event) {



        if (!Boolean.getBoolean(IGNORE_BACKGROUNDPREINITIALIZER_PROPERTY_NAME)
                && event instanceof ApplicationStartingEvent
                && multipleProcessors()
                && preinitializationStarted.compareAndSet(false, true)) {

            performPreinitialization();
        }

        if ((event instanceof ApplicationReadyEvent
                || event instanceof ApplicationFailedEvent)
                && preinitializationStarted.get()) {

            try {
                preinitializationComplete.await();
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
    }



    private boolean multipleProcessors() {
        return Runtime.getRuntime().availableProcessors() > 1;
    }

}
```

- 几个变量，胖友直接看代码注释。

- `<1>` 处，满足如下的四个条件（每一行注释，对应一个条件），调用 `#performPreinitialization()` 方法，启动线程，后台执行预初始化任务。关于 `#performPreinitialization()` 方法，在 「8.1 performPreinitialization」 详细解析。
- `<2>` 处，如果是 ApplicationReadyEvent 或 ApplicationFailedEvent 事件，说明应用启动成功后失败，则等待预初始化任务完成。
- 总结来说， `<1>` 处，启动后台任务， `<2>` 处，保证后台任务执行完成。

8.1 performPreinitialization

`#performPreinitialization()` 方法，启动线程，后台执行预初始化任务。代码如下：

```
private void performPreinitialization() {
    try {

        Thread thread = new Thread(new Runnable() {

            @Override
            public void run() {

                runSafely(new ConversionServiceInitializer());
                runSafely(new ValidationInitializer());
                runSafely(new MessageConverterInitializer());
                runSafely(new MBeanFactoryInitializer());
                runSafely(new JacksonInitializer());
                runSafely(new CharsetInitializer());

                preinitializationComplete.countDown();
            }

            public void runSafely(Runnable runnable) {
                try {
                    runnable.run();
                } catch (Throwable ex) {

                }
            }

        }, "background-preinit");

        thread.start();
    } catch (Exception ex) {



        preinitializationComplete.countDown();
    }
}
```

- `<1>` 处，创建一个线程，用于后台调用 `#runSafely(Runnable runnable)` 方法，“安全”（即发生异常，不进行抛出去）执行每个初始化任务。
- `<2>` 处，启动线程。
- `<3>` 处，在所有初始化任务都执行完成后，通过标记 `preinitializationComplete` 完成，从而实现在 `#onApplicationEvent(SpringApplicationEvent even)` 的 `<2>` 处的阻塞等待被通知完成。

8.2 初始化任务

- ConversionServiceInitializer
- ValidationInitializer
- MessageConverterInitializer
- MBeanFactoryInitializer

- JacksonInitializer
- CharsetInitializer

org.springframework.boot.context.config.DelegatingApplicationListener ，实现 ApplicationListener、Ordered 接口，和 「DelegatingApplicationContextInitializer」 是类似的，只是读取的是环境变量 "context.listener.classes" 对应的 ApplicationContextInitializer 实现类们。

org.springframework.boot.builder.ParentContextCloserApplicationListener ，实现 ApplicationListener、ApplicationContextAware、Ordered 接口，容器关闭时发出通知，如果父容器关闭，那么自容器也一起关闭。代码如下：

```
public class ParentContextCloserApplicationListener implements ApplicationListener<ParentContextAvailableEvent>, ApplicationContextAware, Ordered {



        private int order = Ordered.LOWEST_PRECEDENCE - 10;

        private ApplicationContext context;

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

        @Override
        public void setApplicationContext(ApplicationContext context) throws BeansException {
                this.context = context;
        }

        @Override
        public void onApplicationEvent(ParentContextAvailableEvent event) {
                maybeInstallListenerInParent(event.getApplicationContext());
        }

}
```

- 在 <1> 处，我们可以看到当接收到 ParentContextAvailableEvent 事件后，会调用 #maybeInstallListenerInParent(ConfigurableApplicationContext child) 方法，向父容器添加监听器，监听父容器的关闭事件。详细解析，见 「10.1 maybeInstallListenerInParent」 。
- ParentContextAvailableEvent 事件的发布，依赖 org.springframework.boot.builder.ParentContextApplicationContextInitializer 初始化类。默认情况下，ParentContextApplicationContextInitializer 类并未使用，所以吧，😈 ParentContextCloserApplicationListener 基本无法生效列。

10.1 maybeInstallListenerInParent

#maybeInstallListenerInParent(ConfigurableApplicationContext child) 方法，向父容器添加监听器，监听父容器的关闭事件。代码如下：

```
private void maybeInstallListenerInParent(ConfigurableApplicationContext child) {

    if (child == this.context

            && child.getParent() instanceof ConfigurableApplicationContext) {

        ConfigurableApplicationContext parent = (ConfigurableApplicationContext) child.getParent();
        parent.addApplicationListener(createContextCloserListener(child));
    }
}
```

- 在 <1> 处，会调用 #createContextCloserListener(ConfigurableApplicationContext child) 方法，创建 ContextCloserListener 对象。代码如下：

```
protected ContextCloserListener createContextCloserListener(ConfigurableApplicationContext child) {
        return new ContextCloserListener(child);
}
```

  - 关于 ContextCloserListener 类，在 「10.2 ContextCloserListener」 中，详细解析。

- `<1>` 处，创建后的 ContextCloserListener 对象，向父容器 `parent` 中注册。

10.2 ContextCloserListener

ContextCloserListener ，是 ParentContextCloserApplicationListener 的内部类，实现 ApplicationListener 接口，监听父容器关闭时，关闭自己（容器）。代码如下：

```
protected static class ContextCloserListener implements ApplicationListener<ContextClosedEvent> {


    private WeakReference<ConfigurableApplicationContext> childContext;

    public ContextCloserListener(ConfigurableApplicationContext childContext) {
        this.childContext = new WeakReference<>(childContext);
    }

    @Override
    public void onApplicationEvent(ContextClosedEvent event) {
        ConfigurableApplicationContext context = this.childContext.get();
        if ((context != null)
                && (event.getApplicationContext() == context.getParent())
                && context.isActive()) {

            context.close();
        }
    }
}
```

`org.springframework.boot.ClearCachesApplicationListener` ，实现 ApplicationListener 接口，实现 ReflectionUtils 的缓存、ClassLoader 的缓存。代码如下：

```
class ClearCachesApplicationListener implements ApplicationListener<ContextRefreshedEvent> {

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {

        ReflectionUtils.clearCache();
        clearClassLoaderCaches(Thread.currentThread().getContextClassLoader());
    }

    private void clearClassLoaderCaches(ClassLoader classLoader) {
        if (classLoader == null) {
            return;
        }

        try {
            Method clearCacheMethod = classLoader.getClass().getDeclaredMethod("clearCache");
            clearCacheMethod.invoke(classLoader);
        } catch (Exception ex) {

        }

        clearClassLoaderCaches(classLoader.getParent());
    }

}
```

- 在接收到容器初始化 ContextRefreshedEvent 事件，触发 ClearCachesApplicationListener 监听器，进行清空缓存。

`org.springframework.boot.context.FileEncodingApplicationListener` ，实现 ApplicationListener、Ordered 接口，在 Spring Boot 环境准备完成以后运行，获取环境中的系统环境参数，检测当前系统环境的 `file.encoding` 和 `spring.mandatory-file-encoding` 设置的值是否一样，如果不一样则抛出异常；
如果不配置 `spring.mandatory-file-encoding` 则不检查。代码如下：

```
public class FileEncodingApplicationListener implements ApplicationListener<ApplicationEnvironmentPreparedEvent>, Ordered {

    private static final Log logger = LogFactory.getLog(FileEncodingApplicationListener.class);

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

    @Override
```

```
    public void onApplicationEvent(ApplicationEnvironmentPreparedEvent event) {
        ConfigurableEnvironment environment = event.getEnvironment();

        if (!environment.containsProperty("spring.mandatory-file-encoding")) {
            return;
        }


        String encoding = System.getProperty("file.encoding");
        String desired = environment.getProperty("spring.mandatory-file-encoding");
        if (encoding != null && !desired.equalsIgnoreCase(encoding)) {
            logger.error("System property 'file.encoding' is currently '" + encoding
                    + "'. It should be '" + desired
                    + "' (as defined in 'spring.mandatoryFileEncoding').");
            logger.error("Environment variable LANG is '" + System.getenv("LANG")
                    + "'. You could use a locale setting that matches encoding='"
                    + desired + "'.");
            logger.error("Environment variable LC_ALL is '" + System.getenv("LC_ALL")
                    + "'. You could use a locale setting that matches encoding='"
                    + desired + "'.");

            throw new IllegalStateException(
                    "The Java Virtual Machine has not been configured to use the "
                            + "desired default character encoding (" + desired + ").");
        }
    }

}
```

org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener ，实现 ApplicationListener 接口，初始化 Liquibase 的 ServiceLocator 对象。代码如下：

> 如果不了解 Liquibase 的胖友，可以看看 《一起来学 SpringBoot 2.x | 第二十四篇：数据库管理与迁移（Liquibase）》 文章。

```
public class LiquibaseServiceLocatorApplicationListener implements ApplicationListener<ApplicationStartingEvent> {

    private static final Log logger = LogFactory.getLog(LiquibaseServiceLocatorApplicationListener.class);

    @Override
    public void onApplicationEvent(ApplicationStartingEvent event) {

        if (ClassUtils.isPresent("liquibase.servicelocator.CustomResolverServiceLocator",
                event.getSpringApplication().getClassLoader())) {
            new LiquibasePresent().replaceServiceLocator();
        }
    }



    private static class LiquibasePresent {

        public void replaceServiceLocator() {

            CustomResolverServiceLocator customResolverServiceLocator = new CustomResolverServiceLocator(new SpringPackageScanClassResolver(logger));

            ServiceLocator.setInstance(customResolverServiceLocator);
        }

    }

}
```

- <1> 处，判断是否存在 liquibase.servicelocator.CustomResolverServiceLocator 类。通过该判断，可以知道是否引入了 liquibase-core 包，需要使初始化 Liquibase 功能。
- <2> 处，创建 CustomResolverServiceLocator 对象。当然，如果没有自定义的话，这个对象，返回的就是默认的 Liquibase ServiceLocator 对象。
- <3> 处，设置到 ServiceLocator 的 instance 属性。

当然，以上的逻辑，胖友选择性了解即可。哈哈哈哈~

严格来说，本文并没有写太多具体的内容。更多的是，为了后面的内容做一个铺垫，同时也让胖友知道，Spring Boot 默认提供的 ApplicationListener 实现类。

参考和推荐如下文章：

- oldflame-Jm 《Spring Boot 源码分析 - ApplicationListener 应用环境（5）》