芋道源码 —— 知识星球

2021-02-01
Spring Boot

精尽 Spring Boot 源码分析 —— 日志系统

1. 概述

在使用 Spring Boot 时，默认就已经提供了日志功能，使用 Logback 作为默认的日志框架。本文，我们就来一起研究下，Spring Boot 是如何自动初始化好日志系统的。

不了解 Spring Boot 日志功能的胖友，可以先看看 《一起来学 SpringBoot 2.x | 第三篇：SpringBoot 日志配置》 文章。

2. LoggingApplicationListener

Spring Boot 提供日志功能，关键在于 LoggingApplicationListener 类。在 《精尽 Spring Boot 源码分析 —— ApplicationListener》 中，我们已经简单介绍过它：

> 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 。

2.1 supportsEventType

实现 #supportsEventType(ResolvableType resolvableType) 方法，判断是否是支持的事件类型。代码如下：

```
// LoggingApplicationListener.java

private static final Class<?>[] EVENT_TYPES = { ApplicationStartingEvent.class,
        ApplicationEnvironmentPreparedEvent.class, ApplicationPreparedEvent.class,
        ContextClosedEvent.class, ApplicationFailedEvent.class };

@Override
public boolean supportsEventType(ResolvableType resolvableType) {
 return isAssignableFrom(resolvableType.getRawClass(), EVENT_TYPES);
}

private boolean isAssignableFrom(Class<?> type, Class<?>... supportedTypes) {
 if (type != null) {
     for (Class<?> supportedType : supportedTypes) {
         if (supportedType.isAssignableFrom(type)) {
             return true;
            }
        }
    }
 return false;
}
```

2.2 supportsSourceType

实现 `#supportsSourceType(Class<?> sourceType)` 方法，判断是否是支持的事件来源。代码如下：

```
// LoggingApplicationListener.java

private static final Class<?>[] SOURCE_TYPES = { SpringApplication.class,
        ApplicationContext.class };

@Override
public boolean supportsSourceType(Class<?> sourceType) {
 return isAssignableFrom(sourceType, SOURCE_TYPES);
}
```

2.3 onApplicationEvent

实现 `#onApplicationEvent(ApplicationEvent event)` 方法，处理事件。代码如下：

```
// LoggingApplicationListener.java

@Override
public void onApplicationEvent(ApplicationEvent event) {
    // 在 Spring Boot 应用启动的时候
 if (event instanceof ApplicationStartingEvent) {
        onApplicationStartingEvent((ApplicationStartingEvent) event);
    // 在 Spring Boot 的 Environment 环境准备完成的时候
    } else if (event instanceof ApplicationEnvironmentPreparedEvent) {
        onApplicationEnvironmentPreparedEvent((ApplicationEnvironmentPreparedEvent) event);
    // 在 Spring Boot 容器的准备工作已经完成（并未启动）的时候
    } else if (event instanceof ApplicationPreparedEvent) {
        onApplicationPreparedEvent((ApplicationPreparedEvent) event);
    // 在 Spring Boot 容器关闭的时候
    } else if (event instanceof ContextClosedEvent
            && ((ContextClosedEvent) event).getApplicationContext().getParent() == null) {
        onContextClosedEvent();
 // 在 Spring Boot 容器启动失败的时候
    } else if (event instanceof ApplicationFailedEvent) {
        onApplicationFailedEvent();
    }
}
```

不同的事件，对应不同的处理方法。下文，我们一一来看。

2.4 onApplicationStartingEvent

#onApplicationStartingEvent(ApplicationStartingEvent event) 方法，代码如下：

```
// LoggingApplicationListener.java

private LoggingSystem loggingSystem;

private void onApplicationStartingEvent(ApplicationStartingEvent event) {
    // <1> 创建 LoggingSystem 对象
 this.loggingSystem = LoggingSystem.get(event.getSpringApplication().getClassLoader());
 // <2> LoggingSystem 的初始化的前置处理
 this.loggingSystem.beforeInitialize();
}
```

- <1> 处，调用 LoggingSystem#get(ClassLoader classLoader) 方法，创建（获得） LoggingSystem 对象。关于这个，可以先看看 「3.1 get」 小节。
  - 通过 LoggingSystem 的抽象，对应不同日志框架对应的 LoggingSystem 实现，达到方便透明的接入不同的日志框架~
- <2> 处，调用 LoggingSystem#beforeInitialize() 方法，执行 LoggingSystem 的初始化的前置处理。关于这个，可以先看看 「3.2 beforeInitialize」 小节。

2.5 onApplicationEnvironmentPreparedEvent

#onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent event) 方法，代码如下：

```
// LoggingApplicationListener.java

private void onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent event) {
 if (this.loggingSystem == null) {
     this.loggingSystem = LoggingSystem.get(event.getSpringApplication().getClassLoader());
    }
 // 初始化
    initialize(event.getEnvironment(), event.getSpringApplication().getClassLoader());
}

protected void initialize(ConfigurableEnvironment environment, ClassLoader classLoader) {
    // <1> 初始化 LoggingSystemProperties 配置
    new LoggingSystemProperties(environment).apply();
    // <2> 初始化 LogFile
    LogFile logFile = LogFile.get(environment);
    if (logFile != null) {
        logFile.applyToSystemProperties(); // <2.1>
    }
    // <3> 初始化早期的 Spring Boot Logging 级别
    initializeEarlyLoggingLevel(environment);
    // <4> 初始化 LoggingSystem 日志系统
    initializeSystem(environment, this.loggingSystem, logFile);
    // <5> 初始化最终的 Spring Boot Logging 级别
    initializeFinalLoggingLevels(environment, this.loggingSystem);
    // <6>
    registerShutdownHookIfNecessary(environment, this.loggingSystem);
}
```

<1> 处，调用 LoggingSystemProperties#apply() 方法，初始化 LoggingSystemProperties 配置。关于这个，可以先看看 「4. LoggingSystemProperties」 小节。

<2> 处，调用 LogFile#get(environment) 方法，创建（获得）LogFile 。关于这个，可以先看看

「5. LogFile」 小节。

- ○ <2.1> 处，调用 LogFile#applyToSystemProperties() 方法，应用 LogFile.path 和 LogFile.file 到系统属性中。

<3> 处，调用 #initializeEarlyLoggingLevel(ConfigurableEnvironment environment) 方法，初始化早期的 Spring Boot Logging 级别。详细解析，见 「2.5.1 initializeEarlyLoggingLevel」 中。
<4> 处，调用 #initializeSystem(ConfigurableEnvironment environment, LoggingSystem system, LogFile logFile) 方法，初始化 LoggingSystem 日志系统。详细解析，见 「2.5.2 initializeSystem」 中。
<5> 处，调用 #initializeFinalLoggingLevels(ConfigurableEnvironment environment, LoggingSystem system) 方法，初始化最终的 Spring Boot Logging 级别。详细解析，见 「2.5.3 initializeFinalLoggingLevels」 中。
<6> 处，调用 #registerShutdownHookIfNecessary(Environment environment, LoggingSystem loggingSystem) 方法，注册 ShutdownHook 。详细解析，见 「2.5.4」 中。

2.5.1 initializeEarlyLoggingLevel

#initializeEarlyLoggingLevel(ConfigurableEnvironment environment) 方法，初始化早期的 Spring Boot Logging 级别。代码如下：

```
// LoggingApplicationListener.java

private boolean parseArgs = true;

private LogLevel springBootLogging = null;

private void initializeEarlyLoggingLevel(ConfigurableEnvironment environment) {
 if (this.parseArgs && this.springBootLogging == null) {
     if (isSet(environment, "debug")) {
         this.springBootLogging = LogLevel.DEBUG;
        }
     if (isSet(environment, "trace")) {
         this.springBootLogging = LogLevel.TRACE;
        }
    }
}

private boolean isSet(ConfigurableEnvironment environment, String property) {
    String value = environment.getProperty(property);
 return (value != null && !value.equals("false"));
}
```

可以通过在启动 jar 的时候，跟上 --debug 或 --trace 。
也可以在配置文件中，添加 debug=true 或 trace=true 。
关于日志级别，可以先看看 「6. LogLevel」 。

2.5.2 initializeSystem

#initializeSystem(ConfigurableEnvironment environment, LoggingSystem system, LogFile logFile) 方法，初始化 LoggingSystem 日志系统。代码如下：

```
// LoggingApplicationListener.java

public static final String CONFIG_PROPERTY = "logging.config";

private void initializeSystem(ConfigurableEnvironment environment, LoggingSystem system, LogFile logFile) {
    // <1> 创建 LoggingInitializationContext 对象
    LoggingInitializationContext initializationContext = new LoggingInitializationContext(environment);
    // <2> 获得日志组件的配置文件
    String logConfig = environment.getProperty(CONFIG_PROPERTY);
    // <3> 如果没配置，则直接初始化 LoggingSystem
    if (ignoreLogConfig(logConfig)) {
        system.initialize(initializationContext, null, logFile);
    // <3> 如果有配置，先尝试加载指定配置文件，然后在初始化 LoggingSystem
    } else {
        try {
            ResourceUtils.getURL(logConfig).openStream().close();
            system.initialize(initializationContext, logConfig, logFile); // <X>
        } catch (Exception ex) {
            // NOTE: We can't use the logger here to report the problem
            System.err.println("Logging system failed to initialize " + "using configuration from '" + logConfig + "'
            ex.printStackTrace(System.err);
            throw new IllegalStateException(ex);
        }
    }
}
```

<1> 处，创建 LoggingInitializationContext 对象。其中，org.springframework.boot.logging.LoggingInitializationContext ，LoggingSystem 初始化时的 Context 。代码如下：

```
// LoggingInitializationContext.java

public class LoggingInitializationContext {

 private final ConfigurableEnvironment environment;

 public LoggingInitializationContext(ConfigurableEnvironment environment) {
     this.environment = environment;
    }

 public Environment getEnvironment() {
     return this.environment;
    }

}
```

- 虽然目前只有 environment 属性。但是未来可以在后面增加新的参数，而无需改动 LoggingSystem#initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法。

<2> 处，从 environment 中获得 "logging.config" ，即获得日志组件的配置文件。一般情况下，我们无需配置。因为根据不同的日志系统，Spring Boot 按如下“约定规则”组织配置文件名加载日志配置文件：

日志框架	配置文件
Logback	logback-spring.xml, logback-spring.groovy, logback.xml, logback.groovy
Log4j	log4j-spring.properties, log4j-spring.xml, log4j.properties, log4j.xml
Log4j2	log4j2-spring.xml, log4j2.xml

日志框架	配置文件
JDK (Java Util Logging)	logging.properties

<3> 和 <4> 处，调用 LoggingSystem#initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法，初始化 LoggingSystem 日志系统。详细解析，可以先看看 「3.3 initialize」 。
<3> 和 <4> 处，差异点在于后者多了 ResourceUtils.getURL(logConfig).openStream().close() 代码块，看着有点奇怪哟？它的作用是，尝试去加载 logConfig 对应的配置文件，看看是否真的存在~

2.5.3 initializeFinalLoggingLevels

#initializeFinalLoggingLevels(ConfigurableEnvironment environment, LoggingSystem system) 方法，初始化最终的 Spring Boot Logging 级别。代码如下：

```
// LoggingApplicationListener.java

private void initializeFinalLoggingLevels(ConfigurableEnvironment environment, LoggingSystem system) {
    // <1> 如果 springBootLogging 非空，则设置到日志级别
    if (this.springBootLogging != null) {
        initializeLogLevel(system, this.springBootLogging);
    }
    // <2> 设置 environment 中配置的日志级别
    setLogLevels(system, environment);
}
```

<1> 处，如果 springBootLogging 非空，则调用 #initializeLogLevel(LoggingSystem system, LogLevel level) 方法，设置日志级别。代码如下：

```
// LoggingApplicationListener.java

private static final Map<LogLevel, List<String>> LOG_LEVEL_LOGGERS;

static {
    MultiValueMap<LogLevel, String> loggers = new LinkedMultiValueMap<>();
    loggers.add(LogLevel.DEBUG, "sql");
    loggers.add(LogLevel.DEBUG, "web");
    loggers.add(LogLevel.DEBUG, "org.springframework.boot");
    loggers.add(LogLevel.TRACE, "org.springframework");
    loggers.add(LogLevel.TRACE, "org.apache.tomcat");
    loggers.add(LogLevel.TRACE, "org.apache.catalina");
    loggers.add(LogLevel.TRACE, "org.eclipse.jetty");
    loggers.add(LogLevel.TRACE, "org.hibernate.tool.hbm2ddl");
    LOG_LEVEL_LOGGERS = Collections.unmodifiableMap(loggers);
}

protected void initializeLogLevel(LoggingSystem system, LogLevel level) {
    List<String> loggers = LOG_LEVEL_LOGGERS.get(level);
 if (loggers != null) {
     for (String logger : loggers) {
            system.setLogLevel(logger, level);
        }
    }
}
```

- 遍历的 loggers ，是 LOG_LEVEL_LOGGERS 中对应的 level 的值。

- 调用 LoggingSystem#setLogLevel(String loggerName, LogLevel level) 方法，设置指定 loggerName 的日志级别。详细解析，见 [「3.4 setLogLevel」](#) 。

<2> 处，调用 #setLogLevels(LoggingSystem system, Environment environment) 方法，设置 environment 中配置的日志级别。代码如下：

```
// LoggingApplicationListener.java

private static final ConfigurationPropertyName LOGGING_LEVEL = ConfigurationPropertyName.of("logging.level");
private static final ConfigurationPropertyName LOGGING_GROUP = ConfigurationPropertyName.of("logging.group");

private static final Bindable<Map<String, String>> STRING_STRING_MAP = Bindable.mapOf(String.class, String.clas
private static final Bindable<Map<String, String[]>> STRING_STRINGS_MAP = Bindable.mapOf(String.class, String[

protected void setLogLevels(LoggingSystem system, Environment environment) {
    if (!(environment instanceof ConfigurableEnvironment)) {
        return;
    }
    // 创建 Binder 对象
    Binder binder = Binder.get(environment);
    // <1> 获得日志分组的集合
    Map<String, String[]> groups = getGroups(); // <1.1>
    binder.bind(LOGGING_GROUP, STRING_STRINGS_MAP.withExistingValue(groups)); // <1.2>
    // <2> 获得日志级别的集合
    Map<String, String> levels = binder.bind(LOGGING_LEVEL, STRING_STRING_MAP).orElseGet(Collections::emptyMap)
    // <3> 遍历 levels 集合，逐个设置日志级别
    levels.forEach((name, level) -> {
        String[] groupedNames = groups.get(name);
        if (ObjectUtils.isEmpty(groupedNames)) {
            setLogLevel(system, name, level);
        } else {
            setLogLevel(system, groupedNames, level);
        }
    });
}
```

- <1> 处，获得日志分组的集合。
    - <1.1> 处，调用 #getGroups() 方法，获得默认的日志分组集合。代码如下：

```
// LoggingApplicationListener.java

private static final Map<String, List<String>> DEFAULT_GROUP_LOGGERS;
static {
    MultiValueMap<String, String> loggers = new LinkedMultiValueMap<>();
    loggers.add("web", "org.springframework.core.codec");
    loggers.add("web", "org.springframework.http");
    loggers.add("web", "org.springframework.web");
    loggers.add("web", "org.springframework.boot.actuate.endpoint.web");
    loggers.add("web", "org.springframework.boot.web.servlet.ServletContextInitializerBeans");
    loggers.add("sql", "org.springframework.jdbc.core");
    loggers.add("sql", "org.hibernate.SQL");
    DEFAULT_GROUP_LOGGERS = Collections.unmodifiableMap(loggers);
}

private Map<String, String[]> getGroups() {
    Map<String, String[]> groups = new LinkedHashMap<>();
```

```
            DEFAULT_GROUP_LOGGERS.forEach(
                    (name, loggers) -> groups.put(name, StringUtils.toStringArray(loggers)));
         return groups;
        }
```

        - 实际上，就是把我们日常配置的 loggerName 进行了分组。默认情况下，内置了 sql、web 分组。
    - <1.2> 处，从 environment 中读取 logging.group 配置的日志分组。举个例子，在配置文件里增加 logging.group.demo=xxx.Dog,yyy.Cat 。

- <2> 处，从 environment 中读取 logging.level 配置的日志分组。举两个例子，在配置文件里添加：

    - logging.level.web=INFO
    - logging.level.xxx.Dog=INFO

- <3> 处，遍历 levels 集合，逐个设置日志级别。涉及的方法，代码如下：

```
// LoggingApplicationListener.java

private void setLogLevel(LoggingSystem system, String[] names, String level) {
    // 遍历 names 数组
    for (String name : names) {
        setLogLevel(system, name, level);
    }
}

private void setLogLevel(LoggingSystem system, String name, String level) {
    try {
        // 获得 loggerName
        name = name.equalsIgnoreCase(LoggingSystem.ROOT_LOGGER_NAME) ? null : name;
        // 设置日志级别
        system.setLogLevel(name, coerceLogLevel(level));
    } catch (RuntimeException ex) {
        this.logger.error("Cannot set level '" + level + "' for '" + name + "'");
    }
}

/**
 * @param level 日志级别字符串
 * @return 将字符串转换成 {@link LogLevel}
 */
private LogLevel coerceLogLevel(String level) {
    String trimmedLevel = level.trim();
    if ("false".equalsIgnoreCase(trimmedLevel)) { // false => OFF
        return LogLevel.OFF;
    }
    return LogLevel.valueOf(trimmedLevel.toUpperCase(Locale.ENGLISH));
}
```

    - 比较简单，胖友瞅瞅~

2.5.4 registerShutdownHookIfNecessary

#registerShutdownHookIfNecessary(Environment environment, LoggingSystem loggingSystem) 方法，注册

ShutdownHook 。代码如下：

```
// LoggingApplicationListener.java

/**
 * The name of the Spring property that controls the registration of a shutdown hook
 * to shut down the logging system when the JVM exits.
 * @see LoggingSystem#getShutdownHandler
 */
public static final String REGISTER_SHUTDOWN_HOOK_PROPERTY = "logging.register-shutdown-hook";

private void registerShutdownHookIfNecessary(Environment environment, LoggingSystem loggingSystem) {
    // 获得 logging.register-shutdown-hook 对应的配置值
    boolean registerShutdownHook = environment.getProperty(REGISTER_SHUTDOWN_HOOK_PROPERTY, Boolean.class, false);
    // 如果开启
    if (registerShutdownHook) {
        // <x> 获得 shutdownHandler 钩子
        Runnable shutdownHandler = loggingSystem.getShutdownHandler();
        // 注册 ShutdownHook(
        if (shutdownHandler != null
                && shutdownHookRegistered.compareAndSet(false, true)) {
            registerShutdownHook(new Thread(shutdownHandler));
        }
    }
}

void registerShutdownHook(Thread shutdownHook) {
    Runtime.getRuntime().addShutdownHook(shutdownHook);
}
```

<X> 处，所注册的 ShutdownHook ，通过调用 LoggingSystem#getShutdownHandler() 方法，进行获得。详细解析，见 「3.5 getShutdownHandler」。

2.6 onApplicationPreparedEvent

#onApplicationPreparedEvent(ApplicationPreparedEvent event) 方法，代码如下：

```
// LoggingApplicationListener.java

/**
 * The name of the {@link LoggingSystem} bean.
 */
public static final String LOGGING_SYSTEM_BEAN_NAME = "springBootLoggingSystem";

private void onApplicationPreparedEvent(ApplicationPreparedEvent event) {
    ConfigurableListableBeanFactory beanFactory = event.getApplicationContext().getBeanFactory();
 if (!beanFactory.containsBean(LOGGING_SYSTEM_BEAN_NAME)) {
        beanFactory.registerSingleton(LOGGING_SYSTEM_BEAN_NAME, this.loggingSystem);
    }
}
```

将创建的 LoggingSystem 对象，注册到 Spring 容器中。

2.7 onContextClosedEvent

#onContextClosedEvent() 方法，代码如下：

```
// LoggingApplicationListener.java

private void onContextClosedEvent() {
 if (this.loggingSystem != null) {
     this.loggingSystem.cleanUp();
    }
}
```

调用 LoggingSystem#cleanUp() 方法，执行清理。详细解析，见 「3.6 cleanUp」 中。

2.8 onApplicationFailedEvent

#onApplicationFailedEvent() 方法，代码如下：

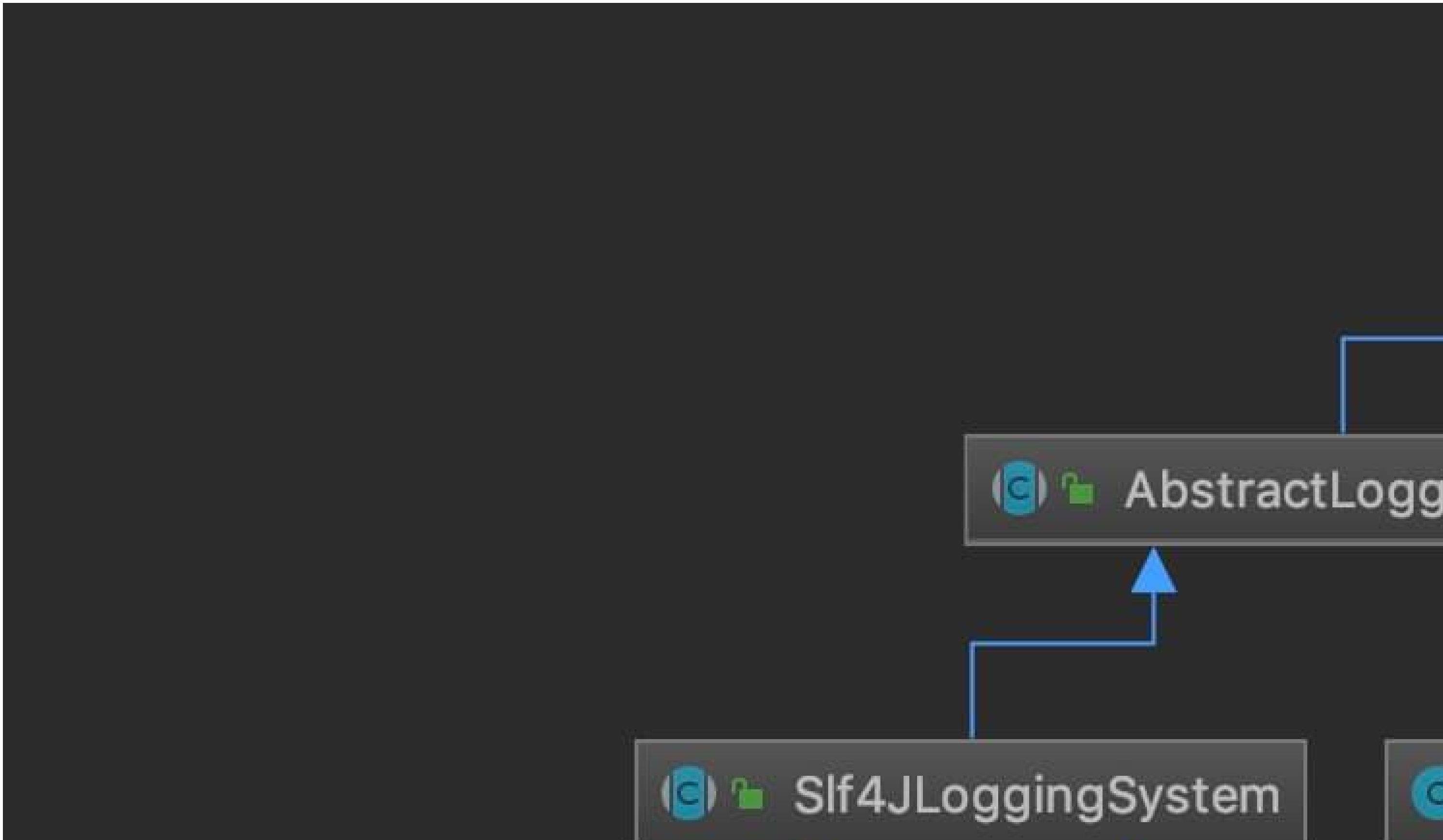
```
// LoggingApplicationListener.java

private void onApplicationFailedEvent() {
 if (this.loggingSystem != null) {
     this.loggingSystem.cleanUp();
    }
}
```

至此，我们需要来看看 LoggingSystem 的实现类。具体的，可以跳到 「7. LoggingSystem 的实现类」 中。

3. LoggingSystem

org.springframework.boot.logging.LoggingSystem ，日志系统抽象类。每个日志框架，都会对应一个实现类。如下图所示：



3.1 get

#get(ClassLoader classLoader) 方法，创建（获得） LoggingSystem 对象。代码如下：

```
// LoggingApplicationListener.java

/**
 * A System property that can be used to indicate the {@link LoggingSystem} to use.
 */
public static final String SYSTEM_PROPERTY = LoggingSystem.class.getName();

/**
 * The value of the {@link #SYSTEM_PROPERTY} that can be used to indicate that no
 * {@link LoggingSystem} should be used.
 */
public static final String NONE = "none";

private static final Map<String, String> SYSTEMS;

static {
    Map<String, String> systems = new LinkedHashMap<>();
    systems.put("ch.qos.logback.core.Appender", "org.springframework.boot.logging.logback.LogbackLoggingSystem");
    systems.put("org.apache.logging.log4j.core.impl.Log4jContextFactory", "org.springframework.boot.logging.log4j2.Lo
    systems.put("java.util.logging.LogManager", "org.springframework.boot.logging.java.JavaLoggingSystem");
    SYSTEMS = Collections.unmodifiableMap(systems);
}

public static LoggingSystem get(ClassLoader classLoader) {
    // <1> 从系统参数 org.springframework.boot.logging.LoggingSystem 获得 loggingSystem 类型
    String loggingSystem = System.getProperty(SYSTEM_PROPERTY);
    // <2> 如果非空，说明配置了
    if (StringUtils.hasLength(loggingSystem)) {
        // <2.1> 如果是 none ，则创建 NoOpLoggingSystem 对象
        if (NONE.equals(loggingSystem)) {
            return new NoOpLoggingSystem();
        }
        // <2.2> 获得 loggingSystem 对应的 LoggingSystem 类，进行创建对象
        return get(classLoader, loggingSystem);
    }
    // <3> 如果为空，说明未配置，则顺序查找 SYSTEMS 中的类。如果存在指定类，则创建该类。
    return SYSTEMS.entrySet().stream()
            .filter((entry) -> ClassUtils.isPresent(entry.getKey(), classLoader))
            .map((entry) -> get(classLoader, entry.getValue())).findFirst()
            .orElseThrow(() -> new IllegalStateException("No suitable logging system located"));
}
```

- <1> 处，从系统参数 org.springframework.boot.logging.LoggingSystem 获得 loggingSystem 类型。
- <2> 处，如果非空，说明配置了。
    - <2.1> 处，如果是 none ，则创建 NoOpLoggingSystem 对象。
    - <2.2> 处，调用 #get(ClassLoader classLoader, String loggingSystemClass) 方法，获得 loggingSystem 对应的 LoggingSystem 类，进行创建对象。代码如下：

```
// LoggingSystem.java
```

```
private static LoggingSystem get(ClassLoader classLoader, String loggingSystemClass) {
 try {
        Class<?> systemClass = ClassUtils.forName(loggingSystemClass, classLoader);
     return (LoggingSystem) systemClass.getConstructor(ClassLoader.class).newInstance(classLoader);
    } catch (Exception ex) {
     throw new IllegalStateException(ex);
    }
}
```

- systemClass 中的 VALUES ，就是 loggingSystem 对应的类。

<3> 处，如果为空，说明未配置，则顺序查找 SYSTEMS 中的类。如果存在指定类，则创建该类
。

3.2 beforeInitialize

#beforeInitialize() 抽象方法，初始化的前置方法。代码如下：

```
// LoggingSystem.java

/**
 * Reset the logging system to be limit output. This method may be called before
 * {@link #initialize(LoggingInitializationContext, String, LogFile)} to reduce
 * logging noise until the system has been fully initialized.
 */
public abstract void beforeInitialize();
```

3.3 initialize

#initialize() 方法，初始化。代码如下：

```
// LoggingSystem.java

/**
 * Fully initialize the logging system.
 * @param initializationContext the logging initialization context
 * @param configLocation a log configuration location or {@code null} if default
 * initialization is required
 * @param logFile the log output file that should be written or {@code null} for
 * console only output
 */
public void initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) {
}
```

目前是个空方法，需要子类来实现。
我们先不着急看子类的实现，等后面继续看。

3.4 setLogLevel

#setLogLevel(String loggerName, LogLevel level) 方法，设置指定 loggerName 的日志级别。代码如下：

```
// LoggingSystem.java

/**
 * Sets the logging level for a given logger.
 * @param loggerName the name of the logger to set ({@code null} can be used for the
 * root logger).
 * @param level the log level ({@code null} can be used to remove any custom level for
 * the logger and use the default configuration instead)
 */
public void setLogLevel(String loggerName, LogLevel level) {
 throw new UnsupportedOperationException("Unable to set log level");
}
```

目前是个空方法，需要子类来实现。
我们先不着急看子类的实现，等后面继续看。

3.5 getShutdownHandler

#getShutdownHandler() 方法，获得 ShutdownHook 的 Runnable 对象。代码如下：

```
// LoggingSystem.java

/**
 * Returns a {@link Runnable} that can handle shutdown of this logging system when the
 * JVM exits. The default implementation returns {@code null}, indicating that no
 * shutdown is required.
 * @return the shutdown handler, or {@code null}
 */
public Runnable getShutdownHandler() {
 return null;
}
```

目前是个空方法，需要子类来实现。
我们先不着急看子类的实现，等后面继续看。

3.6 cleanUp

#cleanUp() 方法，清理。代码如下：

```
// LoggingSystem.java

/**
 * Clean up the logging system. The default implementation does nothing. Subclasses
 * should override this method to perform any logging system-specific cleanup.
 */
public void cleanUp() {
}
```

目前是个空方法，需要子类来实现。
我们先不着急看子类的实现，等后面继续看。

4. LoggingSystemProperties

org.springframework.boot.logging.LoggingSystemProperties ，LoggingSystem 的配置类。

4.1 构造方法

```
// LoggingSystemProperties.java

private final Environment environment;

public LoggingSystemProperties(Environment environment) {
    Assert.notNull(environment, "Environment must not be null");
 this.environment = environment;
}
```

4.2 apply

#apply() 方法，解析 environment 的配置变量到系统属性中。代码如下：

```
// LoggingSystemProperties.java

public void apply() {
    apply(null);
}

public void apply(LogFile logFile) {
    // <1> 获得 PropertyResolver 对象
    PropertyResolver resolver = getPropertyResolver();
    // <2> 解析配置文件到系统属性中
    setSystemProperty(resolver, EXCEPTION_CONVERSION_WORD, "exception-conversion-word");
    setSystemProperty(PID_KEY, new ApplicationPid().toString()); // 应用进程编号
    setSystemProperty(resolver, CONSOLE_LOG_PATTERN, "pattern.console");
    setSystemProperty(resolver, FILE_LOG_PATTERN, "pattern.file");
    setSystemProperty(resolver, FILE_MAX_HISTORY, "file.max-history");
    setSystemProperty(resolver, FILE_MAX_SIZE, "file.max-size");
    setSystemProperty(resolver, LOG_LEVEL_PATTERN, "pattern.level");
    setSystemProperty(resolver, LOG_DATEFORMAT_PATTERN, "pattern.dateformat");
    // <3> 如果 logFile 非空，则应用配置
    if (logFile != null) {
        logFile.applyToSystemProperties();
    }
}
```

<1> 处，调用 #getPropertyResolver() 方法，获得 PropertyResolver 对象。代码如下：

```
// LoggingSystemProperties.java

private PropertyResolver getPropertyResolver() {
 if (this.environment instanceof ConfigurableEnvironment) {
        PropertySourcesPropertyResolver resolver = new PropertySourcesPropertyResolver(((ConfigurableEnvironmen
        resolver.setIgnoreUnresolvableNestedPlaceholders(true);
```

```
     return resolver;
    }
 return this.environment;
}
```

<2> 处，调用 #setSystemProperty(PropertyResolver resolver, String systemPropertyName, String propertyName) 方法，解析配置文件到系统属性中。代码如下：

```
// LoggingSystemProperties.java

public static final String PID_KEY = "PID";
public static final String EXCEPTION_CONVERSION_WORD = "LOG_EXCEPTION_CONVERSION_WORD";
public static final String LOG_FILE = "LOG_FILE";
public static final String LOG_PATH = "LOG_PATH";
public static final String FILE_LOG_PATTERN = "FILE_LOG_PATTERN";
public static final String FILE_MAX_HISTORY = "LOG_FILE_MAX_HISTORY";
public static final String FILE_MAX_SIZE = "LOG_FILE_MAX_SIZE";
public static final String LOG_LEVEL_PATTERN = "LOG_LEVEL_PATTERN";
public static final String LOG_DATEFORMAT_PATTERN = "LOG_DATEFORMAT_PATTERN";

private void setSystemProperty(PropertyResolver resolver, String systemPropertyName, String propertyName) {
    setSystemProperty(systemPropertyName, resolver.getProperty("logging." + propertyName)); // <X>
}

private void setSystemProperty(String name, String value) {
 if (System.getProperty(name) == null && value != null) {
        System.setProperty(name, value);
    }
}
```

- <X> 处，读取的是 environment 中的 logging. 开头的配置属性。

5. LogFile

org.springframework.boot.logging.LogFile ，日志文件。

5.1 构造方法

```
// LogFile.java

/**
 * 文件名
 */
private final String file;
/**
 * 文件路径
 */
private final String path;
```

5.2 applyToSystemProperties

#applyToSystemProperties() 方法，应用 file、path 到系统属性。代码如下：

```
// LogFile.java

public static final String FILE_NAME_PROPERTY = "logging.file.name";
public static final String FILE_PATH_PROPERTY = "logging.file.path";

public void applyToSystemProperties() {
    applyTo(System.getProperties());
}

public void applyTo(Properties properties) {
    put(properties, LoggingSystemProperties.LOG_PATH, this.path);
    put(properties, LoggingSystemProperties.LOG_FILE, toString());
}
```

#toString() 方法，返回文件名。代码如下：

```
// LogFile.java

@Override
public String toString() {
 if (StringUtils.hasLength(this.file)) {
     return this.file;
    }
 return new File(this.path, "spring.log").getPath();
}
```

#put(Properties properties, String key, String value) 方法，添加属性值到系统属性。代码如下：

```
// LogFile.java

private void put(Properties properties, String key, String value) {
 if (StringUtils.hasLength(value)) {
        properties.put(key, value);
    }
}
```

5.3 get

#get(PropertyResolver propertyResolver) 方法，获得（创建）LogFile 对象。代码如下：

```
// LogFile.java

public static LogFile get(PropertyResolver propertyResolver) {
    // <1> 获得 file 和 path 属性
    String file = getLogFileProperty(propertyResolver, FILE_NAME_PROPERTY, FILE_PROPERTY);
    String path = getLogFileProperty(propertyResolver, FILE_PATH_PROPERTY, PATH_PROPERTY);
 // <2> 创建 LogFile 对象
 if (StringUtils.hasLength(file) || StringUtils.hasLength(path)) {
     return new LogFile(file, path);
    }
 return null;
}
```

<1> 处，调用 #getLogFileProperty(PropertyResolver propertyResolver, String propertyName, String deprecatedPropertyName) 方法，获得 file 和 path 属性。代码如下：

```
// LogFile.java

private static String getLogFileProperty(PropertyResolver propertyResolver, String propertyName, String depreca
    String property = propertyResolver.getProperty(propertyName);
  if (property != null) {
     return property;
    }
 return propertyResolver.getProperty(deprecatedPropertyName);
}
```

<2> 处，创建 LogFile 对象。

6. LogLevel

org.springframework.boot.logging.LogLevel ，Spring Boot 日志枚举类。代码如下：

```
// LogLevel.java

public enum LogLevel {

    TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

}
```

每个日志框架，都有其日志级别。通过 LogLevel 枚举类，和它们映射。

7. LoggingSystem 的实现类

7.1 NoOpLoggingSystem

NoOpLoggingSystem ，是 LoggingSystem 的内部静态类，继承 LoggingSystem 类，空操作的 LoggingSystem 实现类，用于禁用日志系统的时候。代码如下：

```
// LoggingSystem#NoOpLoggingSystem.java

static class NoOpLoggingSystem extends LoggingSystem {

 @Override
 public void beforeInitialize() {

    }

 @Override
 public void setLogLevel(String loggerName, LogLevel level) {

    }

 @Override
 public List<LoggerConfiguration> getLoggerConfigurations() {
     return Collections.emptyList();
    }

 @Override
 public LoggerConfiguration getLoggerConfiguration(String loggerName) {
     return null;
    }

}
```

7.2 AbstractLoggingSystem

org.springframework.boot.logging.AbstractLoggingSystem ，继承 LoggingSystem 抽象类，是 LoggingSystem 的抽象基类。

7.2.1 构造方法

```
// AbstractLoggingSystem.java

private final ClassLoader classLoader;

public AbstractLoggingSystem(ClassLoader classLoader) {
 this.classLoader = classLoader;
}
```

7.2.2 initialize

实现 #initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法，提供模板化的初始化逻辑。代码如下：

```
// AbstractLoggingSystem.java

@Override
public void initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) {
    // <1> 有自定义的配置文件，则使用指定配置文件进行初始化
    if (StringUtils.hasLength(configLocation)) {
        initializeWithSpecificConfig(initializationContext, configLocation, logFile);
        return;
    }
    // <2> 无自定义的配置文件，则使用约定配置文件进行初始化
    initializeWithConventions(initializationContext, logFile);
}
```

<1> 处，有自定义的配置文件，则调用 #initializeWithSpecificConfig(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法，使用指定配置文件进行初始化。详细解析，见 「7.2.2.1 initializeWithSpecificConfig」 。

<2>

处，无自定义的配置文件，则调用 #initializeWithConventions(LoggingInitializationContext initializationContext, LogFile logFile) 方法，使用约定配置文件进行初始化。详细解析，见 7.2.2.2 initializeWithConventions 。

7.2.2.1 initializeWithSpecificConfig

#initializeWithSpecificConfig(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法，使用指定配置文件进行初始化。代码如下：

```
// AbstractLoggingSystem.java

private void initializeWithSpecificConfig(LoggingInitializationContext initializationContext, String configLocation,
    // <1> 获得配置文件（可能有占位符）
    configLocation = SystemPropertyUtils.resolvePlaceholders(configLocation);
    // <2> 加载配置文件
    loadConfiguration(initializationContext, configLocation, logFile);
}
```

<1> 处，获得配置文件（可能有占位符）。
<2> 处，调用 #loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile)) 抽象方法，加载配置文件。代码如下：

```
// AbstractLoggingSystem.java

/**
 * Load a specific configuration.
 * @param initializationContext the logging initialization context
 * @param location the location of the configuration to load (never {@code null})
 * @param logFile the file to load or {@code null} if no log file is to be written
 */
protected abstract void loadConfiguration(LoggingInitializationContext initializationContext, String location,
```

7.2.2.2 initializeWithConventions

#initializeWithConventions(LoggingInitializationContext initializationContext, LogFile logFile) 方法，使用约定配置文件进行初始化。代码如下：

```
// AbstractLoggingSystem.java

private void initializeWithConventions(LoggingInitializationContext initializationContext, LogFile logFile) {
    // <1> 获得约定配置文件
    String config = getSelfInitializationConfig();
    // <2> 如果获取到，结果 logFile 为空，则重新初始化
    if (config != null && logFile == null) {
        // self initialization has occurred, reinitialize in case of property changes
        reinitialize(initializationContext);
        return;
    }
    // <3> 如果获取不到，则尝试获得约定配置文件（带 spring 后缀）
    if (config == null) {
        config = getSpringInitializationConfig();
    }
    // <4> 如果获取到，则加载配置文件
    if (config != null) {
        loadConfiguration(initializationContext, config, logFile);
        return;
    }
    // <5> 如果获取不到，则加载默认配置
    loadDefaults(initializationContext, logFile);
}
```

<1> 处，调用 #getSelfInitializationConfig() 方法，获得约定配置文件。代码如下：

```
// AbstractLoggingSystem.java

protected String getSelfInitializationConfig() {
 return findConfig(getStandardConfigLocations());
}

protected abstract String[] getStandardConfigLocations();

private String findConfig(String[] locations) {
    // 遍历 locations 数组，逐个判断是否存在。若存在，则返回
    for (String location : locations) {
        ClassPathResource resource = new ClassPathResource(location, this.classLoader);
        if (resource.exists()) {
            return "classpath:" + location;
        }
    }
    return null;
}
```

- #getStandardConfigLocations() 抽象方法，获得约定的配置文件。例如说：LogbackLoggingSystem 返回的是 "logback-test.groovy"、"logback-test.xml"、"logback.groovy"、"logback.xml" 。

<2> 处，如果获取到，结果 logFile 为空，则调用 #reinitialize(LoggingInitializationContext initializationContext) 方法，重新初始化。代码如下：

```
// AbstractLoggingSystem.java

/**
 * Reinitialize the logging system if required. Called when
 * {@link #getSelfInitializationConfig()} is used and the log file hasn't changed. May
 * be used to reload configuration (for example to pick up additional System
 * properties).
 * @param initializationContext the logging initialization context
 */
protected void reinitialize(LoggingInitializationContext initializationContext) {
}
```

- 一般情况下，logFile 非空~

<3> 处，如果获取不到，则调用 #getSpringInitializationConfig() 方法，尝试获得约定配置文件（带 -spring 后缀）。代码如下：

```
// AbstractLoggingSystem.java

/**
 * Return any spring specific initialization config that should be applied. By default
 * this method checks {@link #getSpringConfigLocations()}.
 * @return the spring initialization config or {@code null}
 */
protected String getSpringInitializationConfig() {
 return findConfig(getSpringConfigLocations());
}

/**
 * Return the spring config locations for this system. By default this method returns
 * a set of locations based on {@link #getStandardConfigLocations()}.
 * @return the spring config locations
 * @see #getSpringInitializationConfig()
 */
protected String[] getSpringConfigLocations() {
    String[] locations = getStandardConfigLocations();
 for (int i = 0; i < locations.length; i++) {
        String extension = StringUtils.getFilenameExtension(locations[i]);
     // 在文件名和后缀之间，拼接一个
        locations[i] = locations[i].substring(0, locations[i].length() - extension.length() - 1)
                + "-spring." + extension;
    }
 return locations;
}
```

- 例如说：LogbackLoggingSystem 返回的是 "logback-test-spring.groovy"、"logback-test-spring.xml"、 "logback-spring.groovy"、"logback-spring.xml" 。

<4> 处，如果获取到，则调用 #loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile)) 抽象方法，加载配置文件。

<5> 处，如果获取不到，则调用 #loadDefaults(LoggingInitializationContext initializationContext, LogFile logFile) 抽象方法，加载默认配置。代码如下：

```
// AbstractLoggingSystem.java

/**
 * Load sensible defaults for the logging system.
 * @param initializationContext the logging initialization context
 * @param logFile the file to load or {@code null} if no log file is to be written
 */
protected abstract void loadDefaults(LoggingInitializationContext initializationContext, LogFile logFile);
```

7.2.3 LogLevels

LogLevels ，是 AbstractLoggingSystem 的内部静态类，用于 Spring Boot LogLevel 和日志框架的 LogLevel 做映射。代码如下：

```
// AbstractLoggingSystem#LogLevels.java

/**
```

```
 * Maintains a mapping between native levels and {@link LogLevel}.
 *
 * @param <T> the native level type
 */
protected static class LogLevels<T> {

 private final Map<LogLevel, T> systemToNative;

 private final Map<T, LogLevel> nativeToSystem;

 public LogLevels() {
     this.systemToNative = new EnumMap<>(LogLevel.class);
     this.nativeToSystem = new HashMap<>();
    }

 public void map(LogLevel system, T nativeLevel) {
     if (!this.systemToNative.containsKey(system)) {
         this.systemToNative.put(system, nativeLevel);
        }
     if (!this.nativeToSystem.containsKey(nativeLevel)) {
         this.nativeToSystem.put(nativeLevel, system);
        }
    }

 public LogLevel convertNativeToSystem(T level) {
     return this.nativeToSystem.get(level);
    }

 public T convertSystemToNative(LogLevel level) {
     return this.systemToNative.get(level);
    }

 public Set<LogLevel> getSupported() {
     return new LinkedHashSet<>(this.nativeToSystem.values());
    }

}
```

7.3 Slf4JLoggingSystem

org.springframework.boot.logging.Slf4JLoggingSystem ，继承 AbstractLoggingSystem 抽象类，基于 Slf4J 的 LoggingSystem 的抽象基类。

7.3.1 beforeInitialize

重写 #beforeInitialize() 方法，代码如下：

```
// Slf4JLoggingSystem.java

@Override
public void beforeInitialize() {
    // 父方法
    super.beforeInitialize();
    // <1> 配置 JUL 的桥接处理器
    configureJdkLoggingBridgeHandler();
}
```

因为艿艿没有特别完整的了解过日志框架，所以下面的解释，更多凭的是“直觉”！如果有错误的地方，给艿艿星球留言哈~

<1> 处，调用 #configureJdkLoggingBridgeHandler() 方法，配置 JUL 的桥接处理器。详细解析，见「7.3.1.1 configureJdkLoggingBridgeHandler」 。

7.3.1.1 configureJdkLoggingBridgeHandler

#configureJdkLoggingBridgeHandler() 方法，配置 JUL 的桥接处理器。代码如下：

```
// Slf4JLoggingSystem.java

private void configureJdkLoggingBridgeHandler() {
    try {
        // <1> 判断 JUL 是否桥接到 SLF4J 了
        if (isBridgeJulIntoSlf4j()) {
            // <2> 移除 JUL 桥接处理器
            removeJdkLoggingBridgeHandler();
            // <3> 重新安装 SLF4JBridgeHandler
            SLF4JBridgeHandler.install();
        }
    } catch (Throwable ex) {
        // Ignore. No java.util.logging bridge is installed.
    }
}
```

<1> 处，调用 #isBridgeJulIntoSlf4j() 方法，判断 JUL 是否桥接到 SLF4J 了。代码如下：

```
// Slf4JLoggingSystem.java

private static final String BRIDGE_HANDLER = "org.slf4j.bridge.SLF4JBridgeHandler";

/**
 * Return whether bridging JUL into SLF4J or not.
 * @return whether bridging JUL into SLF4J or not
 * @since 2.0.4
 */
protected final boolean isBridgeJulIntoSlf4j() {
 return isBridgeHandlerAvailable() // 判断是否存在 SLF4JBridgeHandler 类
            && isJulUsingASingleConsoleHandlerAtMost(); // 判断是否 JUL 只有 ConsoleHandler 处理器被创建了
}

protected final boolean isBridgeHandlerAvailable() {
 return ClassUtils.isPresent(BRIDGE_HANDLER, getClassLoader());
}

private boolean isJulUsingASingleConsoleHandlerAtMost() {
    Logger rootLogger = LogManager.getLogManager().getLogger("");
    Handler[] handlers = rootLogger.getHandlers();
 return handlers.length == 0
            || (handlers.length == 1 && handlers[0] instanceof ConsoleHandler);
}
```

- 第一个方法，调用后面的两个方法判断~

<2> 处，调用 #removeJdkLoggingBridgeHandler() 方法，移除 JUL 桥接处理器。代码如下：

```
// Slf4JLoggingSystem.java

private void removeJdkLoggingBridgeHandler() {
 try {
     // 移除 JUL 的 ConsoleHandler
        removeDefaultRootHandler();
     // 卸载 SLF4JBridgeHandler
        SLF4JBridgeHandler.uninstall();
    } catch (Throwable ex) {
     // Ignore and continue
    }
}

private void removeDefaultRootHandler() {
 try {
        Logger rootLogger = LogManager.getLogManager().getLogger("");
        Handler[] handlers = rootLogger.getHandlers();
     if (handlers.length == 1 && handlers[0] instanceof ConsoleHandler) {
            rootLogger.removeHandler(handlers[0]);
        }
    } catch (Throwable ex) {
     // Ignore and continue
    }
}
```

- 移除 JUL 的 ConsoleHandler ，卸载 SLF4JBridgeHandler 。

<3> 处，会重新安装 SLF4JBridgeHandler。

7.3.2 cleanUp

重写 #cleanUp() 方法，代码如下：

```
// Slf4JLoggingSystem.java

@Override
public void cleanUp() {
    // 判断 JUL 是否桥接到 SLF4J 了
    if (isBridgeHandlerAvailable()) {
        // 移除 JUL 桥接处理器
        removeJdkLoggingBridgeHandler();
    }
}
```

7.3.3 loadConfiguration

重写 #loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile) 方法，代码如下：

```
// Slf4JLoggingSystem.java

@Override
protected void loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile
    Assert.notNull(location, "Location must not be null");
```

```
  if (initializationContext != null) {
        applySystemProperties(initializationContext.getEnvironment(), logFile);
    }
}
```

调用 #applySystemProperties(Environment environment, LogFile logFile) 方法，应用 environment 和 logFile 的属性，到系统属性种。在 「4.2 apply」 中，已经详细解析。
不过有一点，搞不懂，为什么这么实现。

7.4 LogbackLoggingSystem

org.springframework.boot.logging.logback.LogbackLoggingSystem ，继承 Slf4JLoggingSystem 抽象类，基于 Logback 的 LoggingSystem 实现类。

7.4.1 beforeInitialize

重写 #beforeInitialize() 方法，代码如下：

```
// LogbackLoggingSystem.java

@Override
public void beforeInitialize() {
    // <1.1> 获得 LoggerContext 对象
    LoggerContext loggerContext = getLoggerContext();
    // <1.2> 如果已经初始化过，则直接返回
    if (isAlreadyInitialized(loggerContext)) {
        return;
    }
    // <2> 调用父方法
    super.beforeInitialize();
    // <3> 添加 FILTER 到其中
    loggerContext.getTurboFilterList().add(FILTER);
}
```

<1.1> 处，调用 #getLoggerContext() 方法，获得 LoggerContext 对象。代码如下：

```
// LogbackLoggingSystem.java

private LoggerContext getLoggerContext() {
    ILoggerFactory factory = StaticLoggerBinder.getSingleton().getLoggerFactory();
    Assert.isInstanceOf(LoggerContext.class, factory,
            String.format(
                 "LoggerFactory is not a Logback LoggerContext but Logback is on "
                           + "the classpath. Either remove Logback or the competing "
                           + "implementation (%s loaded from %s). If you are using "
                           + "WebLogic you will need to add 'org.slf4j' to "
                           + "prefer-application-packages in WEB-INF/weblogic.xml",
                   factory.getClass(), getLocation(factory)));
 return (LoggerContext) factory;
}
```

<1.2> 处，调用 #isAlreadyInitialized(LoggerContext loggerContext) 方法，判断如果已经初始化过，则直接返回。代码如下：

```
// LogbackLoggingSystem.java

private boolean isAlreadyInitialized(LoggerContext loggerContext) {
 return loggerContext.getObject(LoggingSystem.class.getName()) != null;
}
```

<2> 处，调用父方法。

<3> 处，添加 FILTER 到 loggerContext 其中。代码如下：

```
// LogbackLoggingSystem.java

private static final TurboFilter FILTER = new TurboFilter() {

 @Override
 public FilterReply decide(Marker marker, ch.qos.logback.classic.Logger logger,
           Level level, String format, Object[] params, Throwable t) {
     return FilterReply.DENY;
    }

};
```

- 因为此时，Logback 并未初始化好，所以全部返回 FilterReply.DENY 。即，先不打印日志。

7.4.2 getStandardConfigLocations

实现 #getStandardConfigLocations() 方法，获得约定的配置文件的数组。代码如下：

```
// LogbackLoggingSystem.java

@Override
protected String[] getStandardConfigLocations() {
 return new String[] { "logback-test.groovy", "logback-test.xml", "logback.groovy",
        "logback.xml" };
}
```

7.4.3 initialize

重写 #initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) 方法，代码如下：

```
// LogbackLoggingSystem.java

private static final String CONFIGURATION_FILE_PROPERTY = "logback.configurationFile";

@Override
public void initialize(LoggingInitializationContext initializationContext, String configLocation, LogFile logFile) {
    // <1> 如果已经初始化，则返回
    LoggerContext loggerContext = getLoggerContext();
    if (isAlreadyInitialized(loggerContext)) {
        return;
    }
    // <2> 调用父方法
    super.initialize(initializationContext, configLocation, logFile);
    // <3> 移除 FILTER
    loggerContext.getTurboFilterList().remove(FILTER);
    // <4> 标记已经初始化
    markAsInitialized(loggerContext);
    // <5> 如果配置了 logback.configurationFile ，则打印日志
    if (StringUtils.hasText(System.getProperty(CONFIGURATION_FILE_PROPERTY))) {
        getLogger(LogbackLoggingSystem.class.getName()).warn("Ignoring '" + CONFIGURATION_FILE_PROPERTY + "' system p
    }
}
```

<1> 处，如果已经初始化，则返回。
<2> 处，调用父方法，进行初始化。
<3> 处，从 loggerContext 中，移除 FILTER 。 如果不移除，就一直打印不出日志列。

<4> 处，调用 #markAsInitialized(LoggerContext loggerContext) 方法，标记已经初始化。代码如下：

```
// LogbackLoggingSystem.java

private void markAsInitialized(LoggerContext loggerContext) {
    loggerContext.putObject(LoggingSystem.class.getName(), new Object());
}
```

<5> 处，如果配置了 "logback.configurationFile" ，则打印日志。

7.4.3.1 loadConfiguration

实现 #loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile) 方法，代码如下：

```
// LogbackLoggingSystem.java

@Override
protected void loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile
    // <1> 调用父方法
    super.loadConfiguration(initializationContext, location, logFile);
    // <2> 重置
    LoggerContext loggerContext = getLoggerContext();
    stopAndReset(loggerContext);
    // <3> 读取配置文件，并进行配置
    try {
        configureByResourceUrl(initializationContext, loggerContext, ResourceUtils.getURL(location));
    } catch (Exception ex) {
        throw new IllegalStateException("Could not initialize Logback logging from " + location, ex);
    }
    // <4> 判断是否发生错误。如果有，则抛出 IllegalStateException 异常
    List<Status> statuses = loggerContext.getStatusManager().getCopyOfStatusList();
    StringBuilder errors = new StringBuilder();
    for (Status status : statuses) {
        if (status.getLevel() == Status.ERROR) {
            errors.append((errors.length() > 0) ? String.format("%n") : "");
            errors.append(status.toString());
        }
    }
    if (errors.length() > 0) {
        throw new IllegalStateException(String.format("Logback configuration error detected: %n%s", errors));
    }
}
```

<1> 处，调用父方法。

<2> 处，调用 #stopAndReset(LoggerContext loggerContext) 方法，重置。代码如下：

```
// LogbackLoggingSystem.java

private void stopAndReset(LoggerContext loggerContext) {
    // 停止
    loggerContext.stop();
    // 重置
    loggerContext.reset();
    // 如果是 SLF4J 桥接
    if (isBridgeHandlerInstalled()) {
        // 添加 LevelChangePropagator
        addLevelChangePropagator(loggerContext);
    }
}

private boolean isBridgeHandlerInstalled() {
    if (!isBridgeHandlerAvailable()) {
        return false;
    }
    java.util.logging.Logger rootLogger = LogManager.getLogManager().getLogger("");
    Handler[] handlers = rootLogger.getHandlers();
    // 判断有 SLF4JBridgeHandler 唯一元素
    return handlers.length == 1 && handlers[0] instanceof SLF4JBridgeHandler;
}

private void addLevelChangePropagator(LoggerContext loggerContext) {
    // 创建 LevelChangePropagator 对象（见 https://cloud.tencent.com/developer/ask/174323 说明）
    LevelChangePropagator levelChangePropagator = new LevelChangePropagator();
    // 设置属性
    levelChangePropagator.setResetJUL(true);
    levelChangePropagator.setContext(loggerContext);
    // 添加 LevelChangePropagator 到 loggerContext 中
    loggerContext.addListener(levelChangePropagator);
}
```

- 通过阅读 https://cloud.tencent.com/developer/ask/174323 文章，我们能弄懂这里为什么要使用 LevelChangePropagator ，以及 「7.3.1.1 configureJdkLoggingBridgeHandler」 处的原因。

<3> 处，调用 #configureByResourceUrl(LoggingInitializationContext initializationContext, LoggerContext loggerContext, URL url) 方法，读取配置文件，并进行配置。代码如下：

```
// LogbackLoggingSystem.java

private void configureByResourceUrl(LoggingInitializationContext initializationContext, LoggerContext loggerCo
    // <X> 如果是 xml 配置格式，则使用 SpringBootJoranConfigurator
    if (url.toString().endsWith("xml")) {
        JoranConfigurator configurator = new SpringBootJoranConfigurator(initializationContext);
        configurator.setContext(loggerContext);
        configurator.doConfigure(url);
    // 如果是其它格式，则使用 ContextInitializer
    } else {
        new ContextInitializer(loggerContext).configureByResource(url);
    }
}
```

- <X> 处，如果是 Logback xml 配置格式，则使用 SpringBootJoranConfigurator 类。
- 至此，Logback 配置文件，就已经被读完落。

<4> 处，判断是否发生错误。如果有，则抛出 IllegalStateException 异常。

7.4.3.1.1 SpringBootJoranConfigurator

org.springframework.boot.logging.logback.SpringBootJoranConfigurator ，继承 JoranConfigurator 类，增加 Spring Boot 自定义的标签。代码如下：

```
// SpringBootJoranConfigurator.java

class SpringBootJoranConfigurator extends JoranConfigurator {

 private LoggingInitializationContext initializationContext;

    SpringBootJoranConfigurator(LoggingInitializationContext initializationContext) {
     this.initializationContext = initializationContext;
    }

 @Override
 public void addInstanceRules(RuleStore rs) {
        // 调用父方法
     super.addInstanceRules(rs);
        Environment environment = this.initializationContext.getEnvironment();
        rs.addRule(new ElementSelector("configuration/springProperty"), new SpringPropertyAction(environment));
        rs.addRule(new ElementSelector("*/springProfile"), new SpringProfileAction(environment));
        rs.addRule(new ElementSelector("*/springProfile/*"), new NOPAction());
    }

}
```

不了解的胖友，可以先看看 《SpringBoot 中 logback.xml 使用 application.yml 中属性》 文章。

org.springframework.boot.logging.logback.SpringProfileAction ，处理 <springProfile /> 标签。
org.springframework.boot.logging.logback.SpringPropertyAction ，处理 <springProperty /> 标签。

7.4.3.2 reinitialize

实现 #reinitialize(LoggingInitializationContext initializationContext) 方法，代码如下：

```
// LogbackLoggingSystem.java
```

```
@Override
protected void reinitialize(LoggingInitializationContext initializationContext) {
 // <1> 重置
    getLoggerContext().reset();
 // <2> 清空 StatusManager
    getLoggerContext().getStatusManager().clear();
 // <3> 加载配置
    loadConfiguration(initializationContext, getSelfInitializationConfig(), null);
}
```

<1> 处，重置。
<2> 处，清空 StatusManager 。
<3> 处，调用 #loadConfiguration(LoggingInitializationContext initializationContext, String location, LogFile logFile) 方法，加载配置。此时，使用的是约定的 Logback 配置文件。

7.4.3.3 loadDefaults

实现 #loadDefaults(LoggingInitializationContext initializationContext, LogFile logFile) 方法，代码如下：

```
// LogbackLoggingSystem.java

@Override
protected void loadDefaults(LoggingInitializationContext initializationContext, LogFile logFile) {
    // <1> 重置
    LoggerContext context = getLoggerContext();
    stopAndReset(context);
    // <2> 创建 LogbackConfigurator 对象
    LogbackConfigurator configurator = new LogbackConfigurator(context);
    // <3> 从 environment 读取变量，设置到 context 中。
    Environment environment = initializationContext.getEnvironment();
    context.putProperty(LoggingSystemProperties.LOG_LEVEL_PATTERN, environment.resolvePlaceholders("${logging.pattern
    context.putProperty(LoggingSystemProperties.LOG_DATEFORMAT_PATTERN, environment.resolvePlaceholders("${logging.pa
    // <4> 创建 DefaultLogbackConfiguration 对象，设置到 configurator 中
    new DefaultLogbackConfiguration(initializationContext, logFile).apply(configurator);
    // <5> 设置日志文件，按天滚动
    context.setPackagingDataEnabled(true);
}
```

<1> 处，调用 #stopAndReset(LoggerContext loggerContext) 方法，重置。
<2> 处，创建 LogbackConfigurator 对象。详细解析，见 「7.4.3.3.1 LogbackConfigurator」 。
<3> 处，从 environment 读取变量，设置到 context 中。
<4> 处，创建 DefaultLogbackConfiguration 对象，后调用 DefaultLogbackConfiguration#apply(LogbackConfigurator) 方法，设置到 configurator 中。详细解析，见 「7.4.3.3.2 DefaultLogbackConfiguration」 。
<5> 处，调用 LoggerContext#setPackagingDataEnabled(boolean packagingDataEnabled) 方法，设置日志文件，按天滚动。

7.4.3.3.1 LogbackConfigurator

org.springframework.boot.logging.logback.LogbackConfigurator ，Logback 配置器，提供一些工具方法，方便配置 Logback 。

因为 LogbackConfigurator 提供的方法，都是被 DefaultLogbackConfiguration 所调用。所以我们先跳到 「7.4.3.3.2 DefaultLogbackConfiguration」 中。

7.4.3.3.1.1 conversionRule

#conversionRule(String conversionWord, Class<? extends Converter> converterClass) 方法，添加转换规则。代码如下：

```
// LogbackConfigurator.java

public void conversionRule(String conversionWord, Class<? extends Converter> converterClass) {
    Assert.hasLength(conversionWord, "Conversion word must not be empty");
    Assert.notNull(converterClass, "Converter class must not be null");
    // 获得注册表
    Map<String, String> registry = (Map<String, String>) this.context.getObject(CoreConstants.PATTERN_RULE_REGISTRY);
    // 如果注册表为空，则进行注册
    if (registry == null) {
        registry = new HashMap<>();
        this.context.putObject(CoreConstants.PATTERN_RULE_REGISTRY, registry);
    }
    // 添加转换规则，到注册表中
    registry.put(conversionWord, converterClass.getName());
}
```

比较简单，胖友自己瞅瞅。

目前有三个转换规则，分别是：

- org.springframework.boot.logging.logback.ColorConverter ，实现 ANSI 颜色转换器。
- org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter ，在异常堆栈的打印过程中添加一些空格。
- org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter ，在异常堆栈的打印过程中添加一些空格。
- org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter ，在异常堆栈的打印过程中添加一些空格。【同上】
- 就不详细解析啦，胖友自己瞅瞅就明白列。

7.4.3.3.1.2 logger

#logger(String name, Level level) 方法，添加 Logger 。代码如下：

```
// LogbackConfigurator.java

public void logger(String name, Level level) {
    logger(name, level, true);
}

public void logger(String name, Level level, boolean additive) {
    logger(name, level, additive, null);
}

public void logger(String name, Level level, boolean additive, Appender<ILoggingEvent> appender) {
    // 获得 Logger 对象
    Logger logger = this.context.getLogger(name);
    // 设置 level
    if (level != null) {
        logger.setLevel(level);
    }
    // 设置 additive
    logger.setAdditive(additive);
    // 设置 appender
    if (appender != null) {
        logger.addAppender(appender);
    }
}
```

7.4.3.3.1.3 appender

#appender(String name, Appender<?> appender) 方法，启动 Appender 。代码如下：

```
// LogbackConfigurator.java

public void appender(String name, Appender<?> appender) {
 // 设置 name
    appender.setName(name);
 // 启动 Appender
    start(appender);
}

public void start(LifeCycle lifeCycle) {
    // 设置 context
    if (lifeCycle instanceof ContextAware) {
        ((ContextAware) lifeCycle).setContext(this.context);
    }
    // 启动
    lifeCycle.start();
}
```

7.4.3.3.1.4 root

#root(Level level, Appender<ILoggingEvent>... appenders) 方法，设置 appender 到 ROOT Logger 。代码如下：

```
// LogbackConfigurator.java

public final void root(Level level, Appender<ILoggingEvent>... appenders) {
    // 获得 Root Logger 对象
    Logger logger = this.context.getLogger(org.slf4j.Logger.ROOT_LOGGER_NAME);
    // 设置 level
    if (level != null) {
        logger.setLevel(level);
    }
    // 添加 appender 到 logger 中
    for (Appender<ILoggingEvent> appender : appenders) {
        logger.addAppender(appender);
    }
}
```

7.4.3.3.2 DefaultLogbackConfiguration

org.springframework.boot.logging.logback.DefaultLogbackConfiguration ，默认的 Logback 配置类。代码如下：

相当于代码生成 logback.xml 的效果。

7.4.3.3.2.1 构造方法

```
// DefaultLogbackConfiguration.java

/**
 * PropertyResolver 对象。提供从 environment 解析配置
 */
private final PropertyResolver patterns;

private final LogFile logFile;

DefaultLogbackConfiguration(LoggingInitializationContext initializationContext, LogFile logFile) {
    this.patterns = getPatternsResolver(initializationContext.getEnvironment());
    this.logFile = logFile;
}

private PropertyResolver getPatternsResolver(Environment environment) {
    // 创建 PropertySourcesPropertyResolver 对象，无 environment
    if (environment == null) {
        return new PropertySourcesPropertyResolver(null);
    }
    // 创建 PropertySourcesPropertyResolver 对象，有 environment
    if (environment instanceof ConfigurableEnvironment) {
        PropertySourcesPropertyResolver resolver = new PropertySourcesPropertyResolver(((ConfigurableEnvironment) env
        resolver.setIgnoreUnresolvableNestedPlaceholders(true);
        return resolver;
    }
    // 直接返回 environment
    return environment;
}
```

7.4.3.3.2.2 apply

#apply(LogbackConfigurator config) 方法，应用配置。代码如下：

```
// DefaultLogbackConfiguration.java

public void apply(LogbackConfigurator config) {
    // <1> 锁
    synchronized (config.getConfigurationLock()) {
        // <2> 设置基础属性
        base(config);
        // <3> 创建 console Appender
        Appender<ILoggingEvent> consoleAppender = consoleAppender(config);
        // <4> 如果 logFile 非空，则创建 file Appender
        if (this.logFile != null) {
            Appender<ILoggingEvent> fileAppender = fileAppender(config, this.logFile.toString());
            // <5> 设置 appender 到 ROOT Logger
            config.root(Level.INFO, consoleAppender, fileAppender);
        } else {
            // <5> 设置 appender 到 ROOT Logger
            config.root(Level.INFO, consoleAppender);
        }
    }
}
```

<1> 处，锁。代码如下：

```
// LogbackConfigurator.java

private LoggerContext context;

public Object getConfigurationLock() {
 return this.context.getConfigurationLock();
}
```

<2> 处，调用 #base(LogbackConfigurator config) 方法，设置基础属性。代码如下：

```
// LogbackConfigurator.java

private void base(LogbackConfigurator config) {
 // <2.1> Converter
    config.conversionRule("clr", ColorConverter.class);
    config.conversionRule("wex", WhitespaceThrowableProxyConverter.class);
    config.conversionRule("wEx", ExtendedWhitespaceThrowableProxyConverter.class);
 // <2.2> 默认的 logger
    config.logger("org.apache.catalina.startup.DigesterFactory", Level.ERROR);
    config.logger("org.apache.catalina.util.LifecycleBase", Level.ERROR);
    config.logger("org.apache.coyote.http11.Http11NioProtocol", Level.WARN);
    config.logger("org.apache.sshd.common.util.SecurityUtils", Level.WARN);
    config.logger("org.apache.tomcat.util.net.NioSelectorPool", Level.WARN);
    config.logger("org.eclipse.jetty.util.component.AbstractLifeCycle", Level.ERROR);
    config.logger("org.hibernate.validator.internal.util.Version", Level.WARN);
}
```

- <2.1> 处，调用 LogbackConfigurator#conversionRule(String conversionWord, Class<? extends Converter> converterClass) 方法，添加转换规则。详细解析，见 「 7.4.3.3.1.1 conversionRule」 。
- <2.2> 处，调用 LogbackConfigurator#logger(String name, Level level) 方法，默认的 logger 。详细解析，见 「7.4.3.3.1.2 logger」 。

<3> 处，调用 #consoleAppender(LogbackConfigurator config) 方法，创建 console Appender 对象。代码如下：

```
// LogbackConfigurator.java

private static final String CONSOLE_LOG_PATTERN = "%clr(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}})
        + "%clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} "
        + "%clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} "
        + "%clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}";

private Appender<ILoggingEvent> consoleAppender(LogbackConfigurator config) {
    ConsoleAppender<ILoggingEvent> appender = new ConsoleAppender<>();
    PatternLayoutEncoder encoder = new PatternLayoutEncoder();
    String logPattern = this.patterns.getProperty("logging.pattern.console", CONSOLE_LOG_PATTERN); // <X>
    encoder.setPattern(OptionHelper.substVars(logPattern, config.getContext()));
    config.start(encoder);
    appender.setEncoder(encoder);
    config.appender("CONSOLE", appender); // <Y>
 return appender;
}
```

  - <X> 处，从 environment 中，读取 "logging.pattern.console" 作为格式。如果找不到，使用 CONSOLE_LOG_PATTERN 。
  - <Y> 处，调用 LogbackConfigurator#appender(String name, Appender<?> appender) 方法，启动 Appender 。详细解析，见 「7.4.3.3.1.3 appender」 。

<4> 处，如果 logFile 非空，则调用 #fileAppender(LogbackConfigurator config, String logFile) 方法，创建 file Appender。代码如下：

```
// LogbackConfigurator.java

private static final String FILE_LOG_PATTERN = "%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}} "
+ "${LOG_LEVEL_PATTERN:-%5p} ${PID:- } --- [%t] %-40.40logger{39} : %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"

private static final String MAX_FILE_SIZE = "10MB";

private Appender<ILoggingEvent> fileAppender(LogbackConfigurator config, String logFile) {
    RollingFileAppender<ILoggingEvent> appender = new RollingFileAppender<>();
    PatternLayoutEncoder encoder = new PatternLayoutEncoder();
    String logPattern = this.patterns.getProperty("logging.pattern.file", FILE_LOG_PATTERN);
    encoder.setPattern(OptionHelper.substVars(logPattern, config.getContext()));
    appender.setEncoder(encoder);
    config.start(encoder);
    appender.setFile(logFile);
 // 滚动策略
    setRollingPolicy(appender, config, logFile);
    config.appender("FILE", appender);
 return appender;
}

private void setRollingPolicy(RollingFileAppender<ILoggingEvent> appender, LogbackConfigurator config, String
    SizeAndTimeBasedRollingPolicy<ILoggingEvent> rollingPolicy = new SizeAndTimeBasedRollingPolicy<>();
    rollingPolicy.setFileNamePattern(logFile + ".%d{yyyy-MM-dd}.%i.gz");
 // 单文件最大值
    setMaxFileSize(rollingPolicy, this.patterns.getProperty("logging.file.max-size", MAX_FILE_SIZE));
    rollingPolicy.setMaxHistory(this.patterns.getProperty("logging.file.max-history", Integer.class, CoreCons
    appender.setRollingPolicy(rollingPolicy);
    rollingPolicy.setParent(appender);
    config.start(rollingPolicy);
}

private void setMaxFileSize(SizeAndTimeBasedRollingPolicy<ILoggingEvent> rollingPolicy, String maxFileSize) {
 try {
        rollingPolicy.setMaxFileSize(FileSize.valueOf(maxFileSize));
    } catch (NoSuchMethodError ex) {
     // Logback < 1.1.8 used String configuration
        Method method = ReflectionUtils.findMethod(SizeAndTimeBasedRollingPolicy.class, "setMaxFileSize", Str
        ReflectionUtils.invokeMethod(method, rollingPolicy, maxFileSize);
```

```
            }
        }
```

<5> 处，调用 LogbackConfigurator#root(Level level, Appender<ILoggingEvent>... appenders) 方法，设置 appender 到 ROOT Logger 。详细解析，见 「7.4.3.3.1.4 root」 。

7.4.4 setLogLevel

实现 #setLogLevel(String loggerName, LogLevel level) 方法，代码如下：

```
// LogbackLoggingSystem.java

private static final LogLevels<Level> LEVELS = new LogLevels<>();

static {
    LEVELS.map(LogLevel.TRACE, Level.TRACE);
    LEVELS.map(LogLevel.TRACE, Level.ALL);
    LEVELS.map(LogLevel.DEBUG, Level.DEBUG);
    LEVELS.map(LogLevel.INFO, Level.INFO);
    LEVELS.map(LogLevel.WARN, Level.WARN);
    LEVELS.map(LogLevel.ERROR, Level.ERROR);
    LEVELS.map(LogLevel.FATAL, Level.ERROR);
    LEVELS.map(LogLevel.OFF, Level.OFF);
}

@Override
public void setLogLevel(String loggerName, LogLevel level) {
 // <1> 获得 Logger 对象
    ch.qos.logback.classic.Logger logger = getLogger(loggerName);
 // <2> 设置日志级别
 if (logger != null) {
        logger.setLevel(LEVELS.convertSystemToNative(level));
    }
}
```

<1> 处，调用 #getLogger(String name) 方法，获得 Logger 对象。代码如下：

```
// LogbackLoggingSystem.java

private ch.qos.logback.classic.Logger getLogger(String name) {
    LoggerContext factory = getLoggerContext();
 if (StringUtils.isEmpty(name) || ROOT_LOGGER_NAME.equals(name)) {
        name = Logger.ROOT_LOGGER_NAME;
    }
 return factory.getLogger(name);

}
```

<2> 处，设置日志级别。

7.4.4 cleanUp

重写 #cleanUp() 方法，代码如下：

```
// LogbackLoggingSystem.java

@Override
public void cleanUp() {
 // 标记为未初始化
    LoggerContext context = getLoggerContext();
    markAsUninitialized(context);
 // 调用父方法
 super.cleanUp();
 // 清空 StatusManager
    context.getStatusManager().clear();
 // 移除 FILTER
    context.getTurboFilterList().remove(FILTER);
}

private void markAsUninitialized(LoggerContext loggerContext) {
    loggerContext.removeObject(LoggingSystem.class.getName());
}
```

7.4.5 getShutdownHandler

实现 `#getShutdownHandler()` 方法，代码如下：

```
// LogbackLoggingSystem.java

@Override
public Runnable getShutdownHandler() {
 return new ShutdownHandler();
}

private final class ShutdownHandler implements Runnable {

 @Override
 public void run() {
        getLoggerContext().stop(); // 停止
    }

}
```

7.5 Log4J2LoggingSystem

`org.springframework.boot.logging.log4j2.Log4J2LoggingSystem` ，继承 Slf4JLoggingSystem 抽象类，基于 Log4J2 的 LoggingSystem 实现类。

就暂时不解析了，基本类似。感兴趣的胖友，可以看看 《spring boot 源码解析28-Log4J2LoggingSystem》 。

7.6 JavaLoggingSystem

`org.springframework.boot.logging.java.JavaLoggingSystem` ，继承 AbstractLoggingSystem 抽象类，基于 JUL 的 LoggingSystem 实现类。

就暂时不解析了，基本类似。感兴趣的胖友，可以看看 《spring boot 源码解析27-JavaLoggingSystem及LoggingSystem生命周期详解》 的 「LoggingSystem」 部分。

666. 彩蛋

Spring Boot 的文章，基本都短不了~咋说呢？虽然长了一些吧，总体还是比较简单和顺畅的。

参考和推荐如下文章：

> 一个努力的码农 《spring boot 源码解析29-LogbackLoggingSystem》
> oldflame-Jm 《Spring boot源码分析-log日志系统（6）》

文章目录

回到首页