4af1b934328048e4237c~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3078&h=522&s=186753&e=png&b=282c34)

首先是`addInstanceRules`，这是一个抽象方法，我们刚刚不是看过`JoranConfigurator`嘛，它实现了这个方法:

![image-20231205085901875.png](https://p1-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/b470c48664984ff6beabee3ce9a88ade~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3054&h=1676&s=701695&e=png&b=282c34)

怎么样，熟悉吗？这不就是我们在`logback.xml`里用到的一些标签嘛！看类名，`**Rules`、`**Action`、`ElementSelector`，**所以`addInstanceRules`的作用就是将`logback.xml`的不同标签匹配路径和对应的动作进行绑定，然后保存到`RuleStore`中，这就是模式匹配！**

我们顺便看一眼这些`**Action`的类，你会发现它们其实都继承了`Action`这个类:

![image-20231205090447171.png](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/756cdf7ce7974d9fb1c62430686d3828~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3054&h=1682&s=550114&e=png&b=292d35)

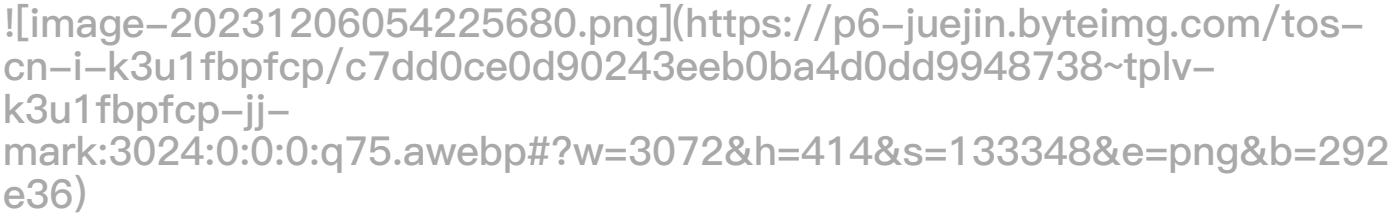
所有继承了`Action`的类都要实现`begin`和`end`方法，这很重要！然后还在`implicitActions`中增加了处理其它标签的模式匹配规则`NestedComplexPropertyIA`和`NestedBasicPropertyIA`:

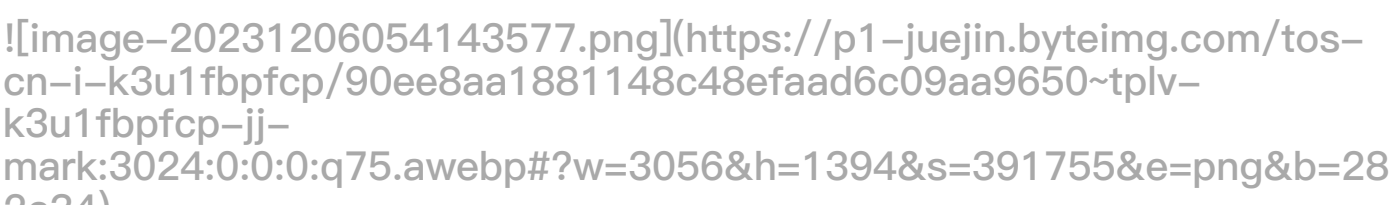
![image-20231206090243460.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/b62aedd412924011819d85284870c548~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3104&h=644&s=245363&e=png&b=292d35)

我们上面的`addInstanceRules`方法里，其实是没有添加类似`<encode>`和`<file>`等等这些标签的匹配规则，这是因为这些标签都属于嵌套标签，它们都与`NestedComplexPropertyIA`或`NestedBasicPropertyIA`进行绑定。

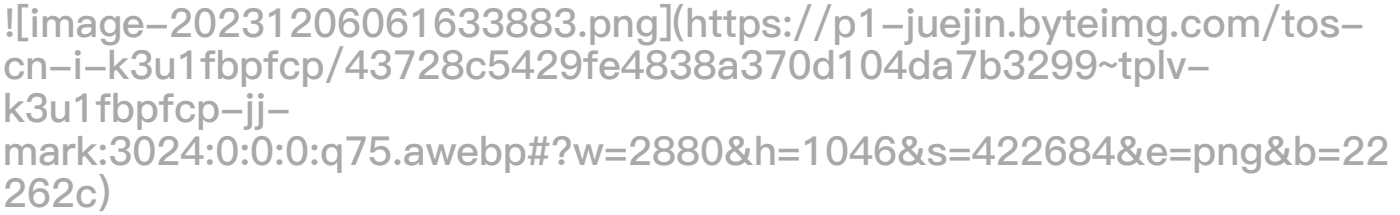
3.1.4.`joran`之`play`

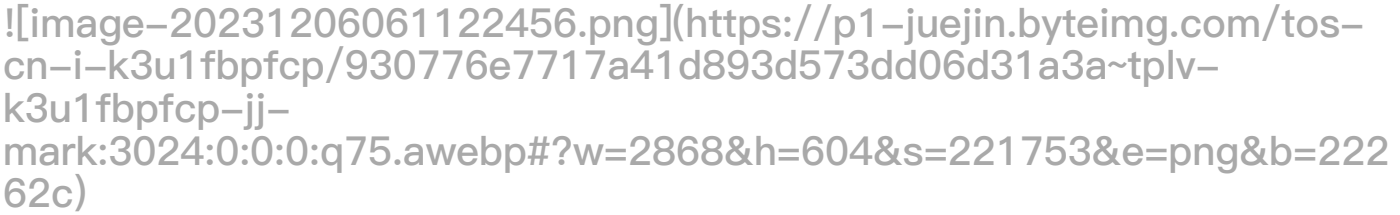
我们现在代码追踪到了`EventPlayer.play`：





`play`的主要逻辑很直观：遍历事件列表`List<SaxEvent>`，处理每一个事件，事件包括 3 种类型：`StartEvent`、`BodyEvent`和`EndEvent`。我们先来看一下这个`List<SaxEvent>`都有什么内容（一共有 61 个元素，我只截取了一部分）：



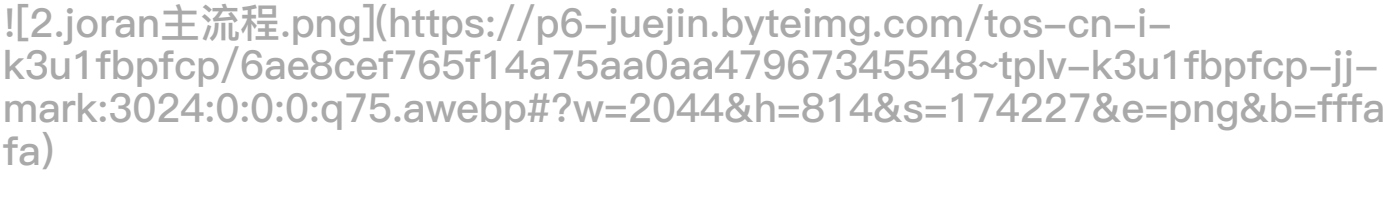


就是我们配置文件中定义的各个标签，而且我们发现了 2 个规律：

1. 标签和事件的映射关系如下：`<pattern>`-->`StartEvent`、`context`-->`BodyEvent`、`</pattern>`-->`EndEvent`；
2. 标签的嵌套关系与`List<SaxEvent>`顺序完全一致；
3. 如果标签中没有内容，就不会映射`BodyEvent`事件。

3.1.5.总结

1. `logback`使用`JDK`提供的`XML`解析工具类`SAXParser`将`logback.xml`解析为自己定义的`SaxEvent`子类，包括`StartEvent`、`BodyEvent`和`EndEvent`；
2. 同时构造解析器，加载支持所有的模式匹配规则到`RuleStore`，默认匹配规则到`implicitActions`；
3. 遍历事件列表，根据模式匹配规则找到对应的`Action`，回调事件对应的方法，`StartEvent`回调`Action.begin()`，`BodyEvent`回调`Action.body()`，`EndEvent`回调`Action.end()`。



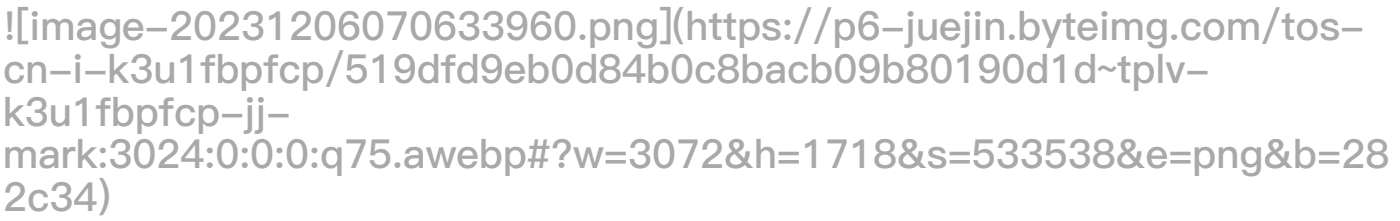
3.2.简单嵌套标签的事件执行流程

下面我们就以`<file>`标签为例，分别来分析简单嵌套标签的事件执行流程。然后在此基础上，去对比分析`<encode>`和`<appender>`等标签。

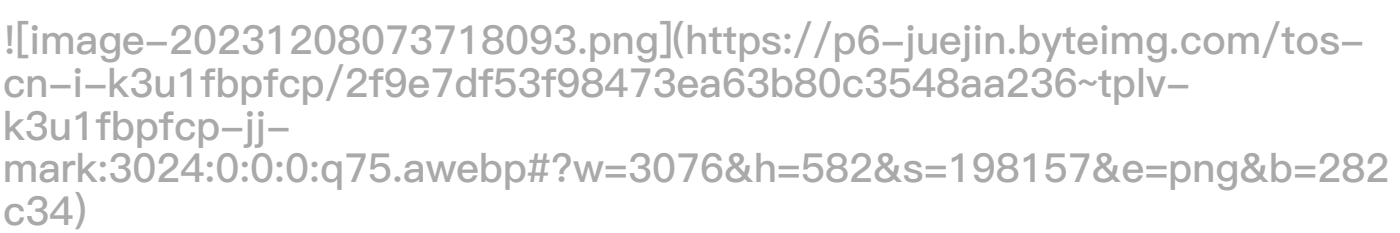
3.2.1.`<file>`之`StartEvent`

只有 2 行，调用`startElement`和`fireInPlay`：

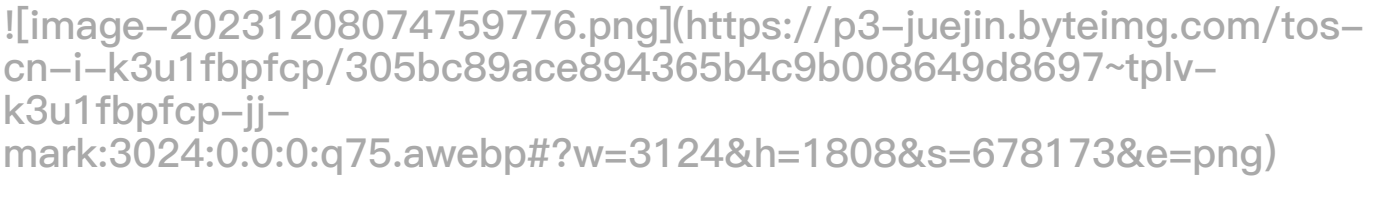
`startElement`获取到标签绑定的`Action`对象（*回顾上面的`addInstanceRules`*），先将它们塞到`actionListStack`当中，供后面的`BodyEvent`使用，然后调用`Action`对象的`begin()`方法：



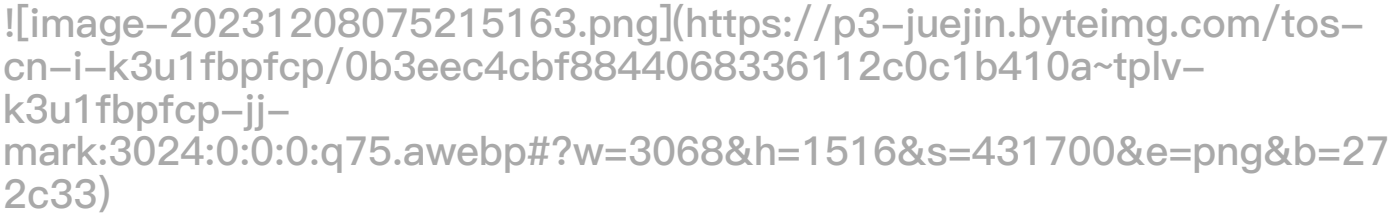
首先获取标签对应的`Action`对象：



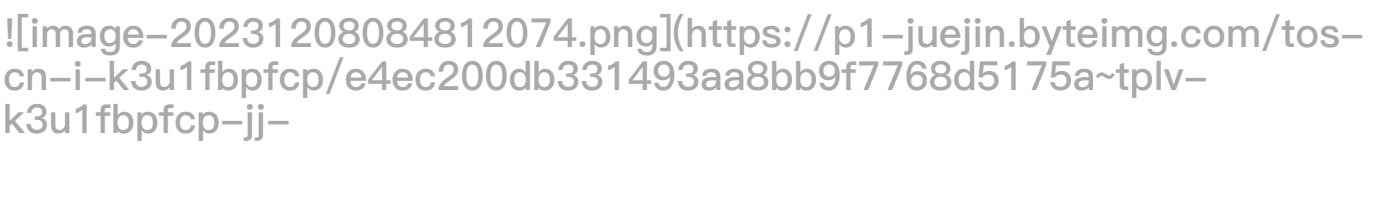
先到`ruleStore`里边找，这里存放的都是我们一开始`JoranConfigurator`类的`addInstanceRules`方法写入的所有模式匹配规则：



在这里找不到，继续通过`lookupImplicitAction`到`implicitActions`找（一开始`JoranConfigurator`类的`addImplicitRules`写入的 2 个对象）：

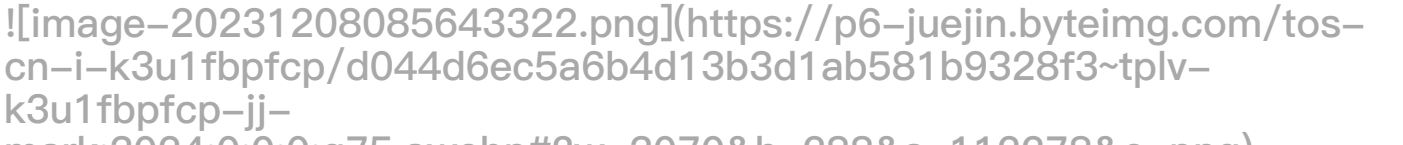


会顺序遍历这两个对象**（`NestedComplexPropertyIA`在前`NestedBasicPropertyIA`在后）**，调用它们的`isApplicable`方法判断该标签是否适用当前的`Action`。首先是`NestedComplexPropertyIA`的`isApplicable`方法：



mark:3024:0:0:0:q75.awebp#?w=3076&h=1620&s=463888&e=png&b=292d35)

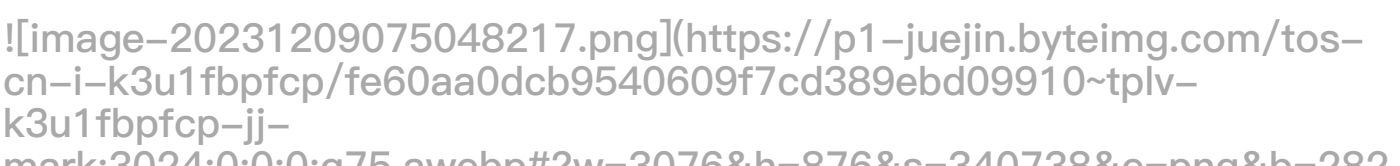
`parentBean.computeAggregationType`方法计算结果为`AS_COMPLEX_*`的标签，会使用`NestedComplexPropertyIA`。计算`AggregationType`前会先获取父类 Bean，我们以`logback.xml`中的`<file>`标签为例，它的父类就是`<appender>`标签指定的`class`参数`ch.qos.logback.core.rolling.RollingFileAppender`：



这里最关键的一步是将父类`RollingFileAppender`的`get`、`set`和`add`方法分别放入`propertyNameToGetter`、`propertyNameToGetter`和`propertyNameToGetter` Map 当中。在判别是使用哪个`Action`时会用到该信息：

先去`RollingFileAppender`类中找是否存在`addFile`方法，如果没有，再去找`setFile`，后者是有的，所以要通过`computeRawAggregationType(setter)`判别：

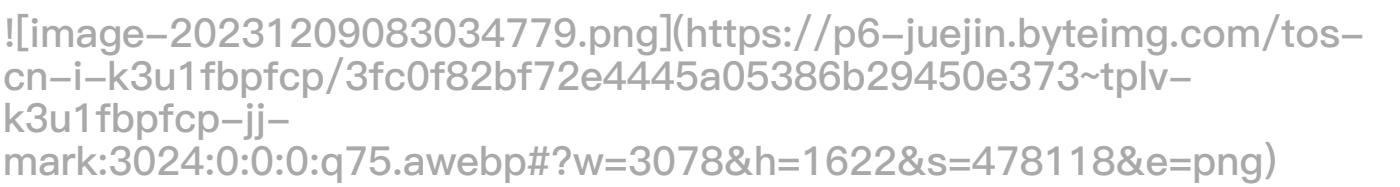
先获取入参类型，然后判断入参类型是简单类型还是复杂类型，如果没有入参类型或者是简单类型就用`NestedBasicPropertyIA`，如果是复杂类型就用`NestedComplexPropertyIA`。`setFile`的第一个入参类型为`java.lang.String`，不为空，所以继续判断入参类型：



类型判断共有 5 种情况：

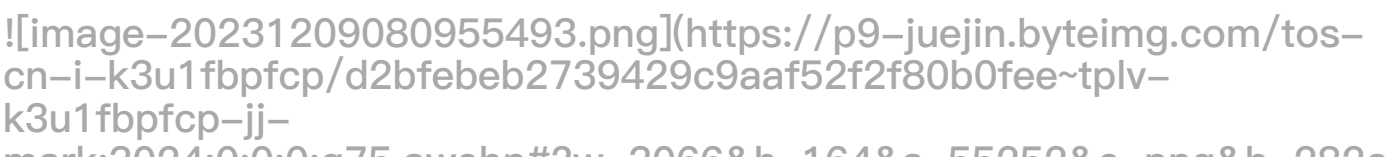
1. 是否是JDK基本类型，包括`Boolean`、`Character`、`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`和`Void`；
2. 是否是`java.lang`包中的类**（为什么单独把`java.lang`拿了出来？）**；
3. 是否是静态类；
4. 是否是枚举；
5. 是否是`Charset`类型，或其子类。

这里命中了第 3 种情况，所以`NestedComplexPropertyIA.isApplicable`最终返回的是`AS_BASIC_PROPERTY`，这个类型自然不能使用复杂`Action`了，那接着会判断能否使用简单的`Action`，调用`NestedBasicPropertyIA.isApplicable`：



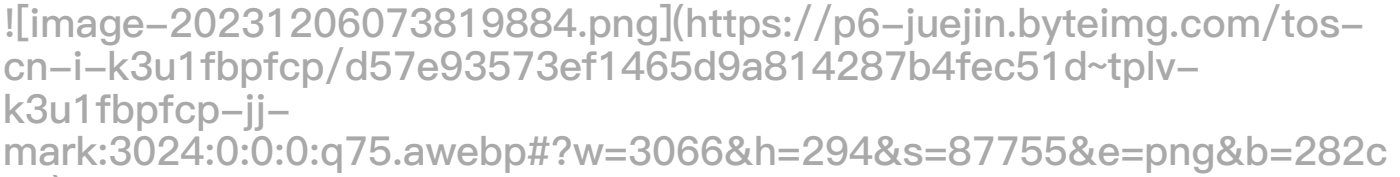
可以看到主体逻辑和复杂`Action`大差不差！唯一区别，就是如果是简单类型，那此刻要将父类`Bean`等信息构建成的`IADataForBasicProperty`添加到`actionDataStash`当中（你回上文看一下，会发现复杂类型也有类似操作）。

然后我们去看一下这个简单的`Action`，它的`begin()`方法干了什么：

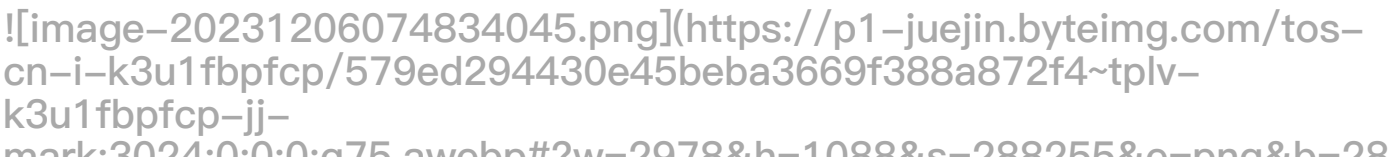


空实现，好吧，还真是简单……

调用完`begin()`方法后，就这就是`fireInPlay`方法，主要用来执行注册的监听器：

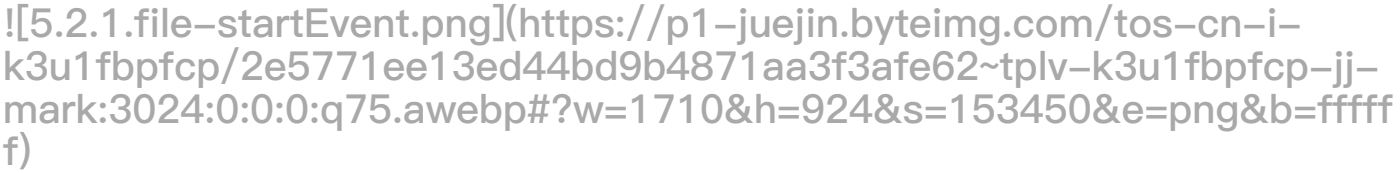


只有配置文件中有`<else>`等条件标签以及`<sift>`等日志隔离的标签，才会在`listenerList`中注册监听器，例如`<then>`标签对应的`ThenAction`的父类`ThenOrElseActionBase`：



我们这里没有，直接跳过。

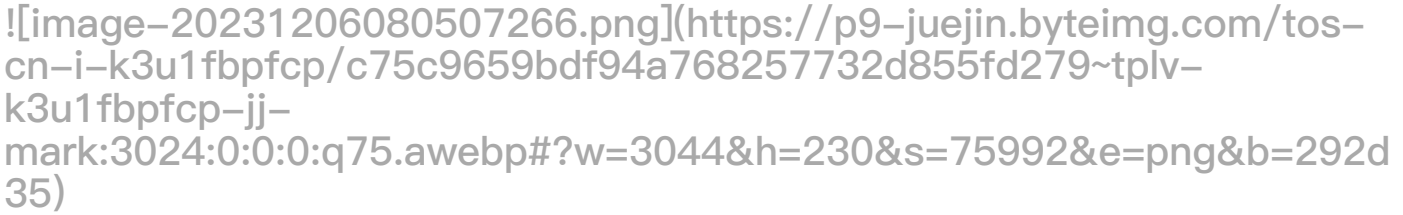
总结一下`<file>`标签的`StartEvent`处理流程：



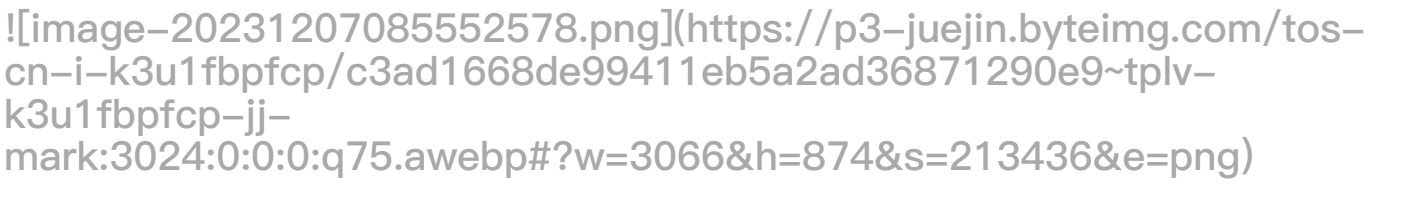
1. 查找标签对应的`Action`类型，先到`RuleStore`中找，没找到去`ImplicitActions`找，找到后将`Action`推送到`actionListStack`；
2. 如果是`ImplicitAction`类型，则先调用`NestedComplexPropertyIA`的`isApplicable`方法判断能否使用该类型处理标签；
3. 如果不行则继续调用`ImplicitActions`中的下一个类型的`isApplicable`方法，即`NestedBasicPropertyIA`；
4. 找到`ImplicitAction`处理类后，将该类型推送到`actionDataStack`中，然后调用事件对应的`play`方法。

3.2.2.`<file>`之`BodyEvent`

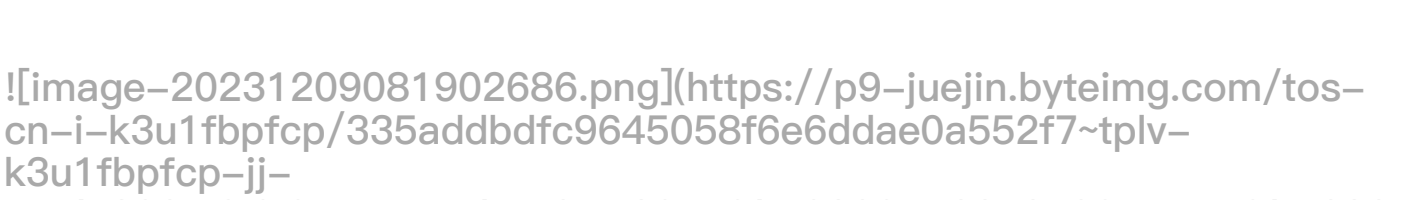
`<file>`标签在执行完`BeginEvent`之后，就该执行`BodyEvent`了。前面入口的细节我们就不列了，直接分析一下重点。首先，不是每个标签的`StartEvent`后面都有`BodyEvent`，只有像`<file>`标签这种标签内有内容才会有`BodyEvent`：



`BodyEvent`的主要处理流程：



`actionListStack`是我们上面在执行`StartEvent`的时候塞好的，上面`StartEvent`已经分析过了，直接调用`NestedBasicPropertyIA`的`body()`方法即可：



cn-i-k3u1fbpfcp/a020a3f5a09a4255bdbbbc0a54bef3de~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3068&h=870&s=225254&e=png&b=292d35)

可以看到，这里直接把`body`的内容作为`setFile`的入参，调用`setFile`：

![image-20231209082256381.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/07f12a3a4a174a028d84248942f436df~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3068&h=532&s=165189&e=png)

`setFile`内部调用了父类`FileAppender`的`setFile`方法：

![image-20231209221730795.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/785e7f73650c4f9ab42f32148b809d93~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=3104&h=530&s=117843&e=png)

`setFile`方法只是把日志文件的完整路径保存到`fileName`字段当中。

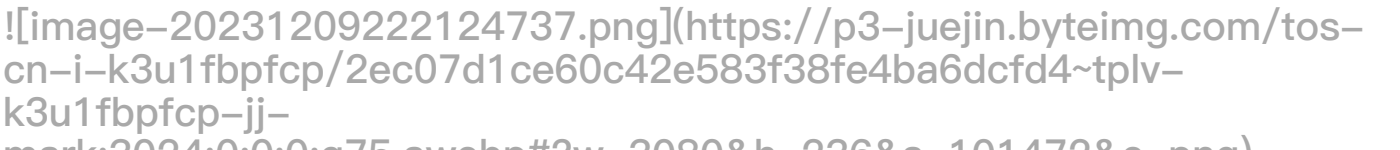
总结一下`BodyEvent`的执行流程：

![5.2.2.file-bodyEvent.png](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/009f6a5f18ad4101a29b691a25f07180~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1254&h=1274&s=129377&e=png&b=ffffff)

1. 解析器直接读取`actionListStack`中的`Action`类型为`NestedBasicPropertyIA`，调用`NestedBasicPropertyIA`的`body()`方法；
2. `body()`内部获取父类`Bean`（这里是`RollingFileAppender`）的`setFile`方法，将内容赋值给`fileName`。

3.2.3.`<file>`之`EndEvent`

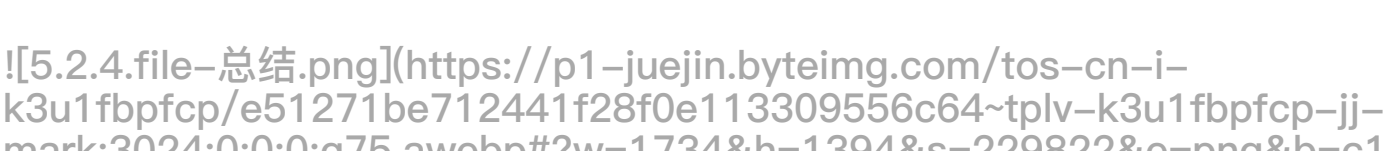
调用完`BodyEvent`事件之后，就该接着执行`<file>`标签的`EndEvent`事件了，同样，还是调用`NestedBasicPropertyIA`的`end()`方法：



可以看到`end()`方法非常简单，只是单纯把`actionDataStack`中塞进去的和`<file>`标签相关的上下文清除。到此为止我们对`<file>`标签的全部解析过程就分析完了。

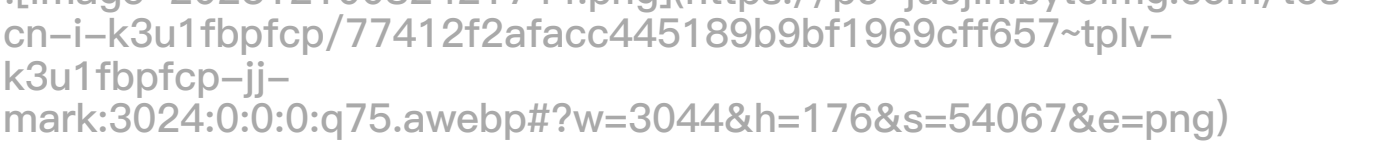
3.2.4.总结

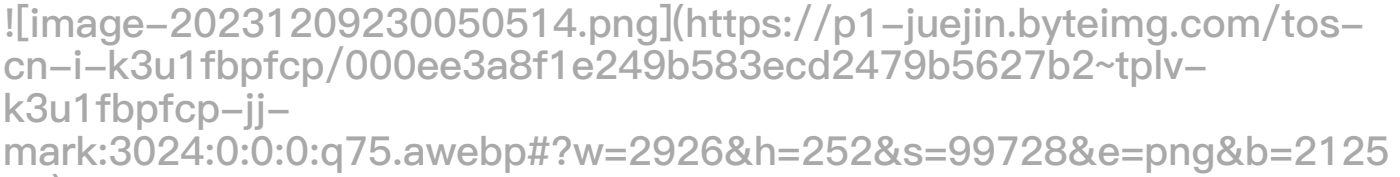
`<file>`标签执行的所有事件流程整理如下：



3.3.复杂嵌套标签的事件执行流程

`<file>`标签对应的 3 个事件的执行逻辑比较简单，下面我们分析一个比较复杂的标签`<encoder>`，这也是`logback`最核心的功能交汇处，控制日志的输出格式和方式。我们先来回顾一下`<encoder>`相关的事件都有哪些：





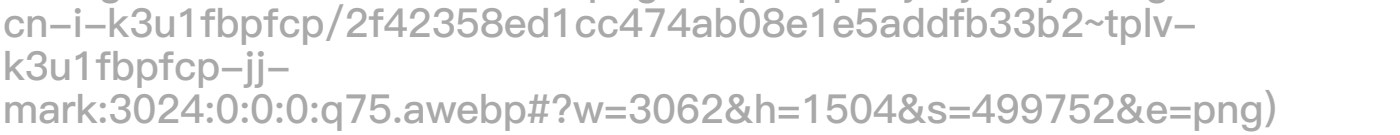
可以看到事件顺序和配置文件当中的标签嵌套关系是一致的。我们如果回去看一下`JoranConfigurator`的`addInstanceRules`方法，会发现并没有往`RuleStore`当中添加和`<encoder>`或`<pattern>`相关的模式匹配规则，所以

它们必然还是调用`NestedBasicPropertyIA`或`NestedComplexPropertyIA`的方法。

`<encoder>`标签指定的`bean`为`PatternLayoutEncoder`，这是`logback`目前唯一有用且默认的 encoder 。

3.3.1.`<encoder>`之`StartEvent`

首先是`<encoder>`标签的`StartEvent`事件，还是先后通过`NestedComplexPropertyIA.isApplicable`和`NestedBasicPropertyIA.isApplicable`判别使用哪个`Action`：



`encoder`和`file`对应的是同一个父类`RollingFileAppender`，所以这里要去判断该父类有没有实现`addEncoder`或`setEncoder`方法，实际上其父类`OutputStreamAppender`实现了，然后通过判断最终会调用到`NestedComplexPropertyIA.begin()`。

哇塞，这个复杂`Action`的`begin`我们还没看过哦：

`begin`主要完成了 3 件事：

1. 将`PatternLayoutEncoder`记录到`NestedComplexProperty`；
2. 将上下文`context`记录到`NestedComplexProperty.context`；
3. 将`NestedComplexProperty`（`Object`类型，真实类型为`PatternLayoutEncoder`）推送到`objectStack`。

3.3.2.`<pattern>`之`StartEvent`

执行完`<encoder>`的`StartEvent`之后，就要执行`<pattern>`标签的`StartEvent`了。`<pattern>`标签在`NestedComplexPropertyIA.isApplicable`进行判断时，通过`ic.peekObject()`获取到的栈顶`bean`是刚刚推送进去的`PatternLayoutEncoder`：

5.`SpringBoot`整合`Logback`

如果我们使用的是`SpirngBoot` 项目，如何打印日志呢？`SpringBoot`已经整合了所有主流的日志框架，当然包括我们上面介绍的`Logback`和`Log4j2`。

5.1.依赖和配置

`SpringBoot`项目默认使用的日志框架是`Logback`，也就是说只要你引入任意一个`SpringBoot`的`starter`依赖，它默认会引入`Logback`日志框架。例如：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.12</version>
</dependency>
```

执行`mvn dependency:tree`获取的`Maven`依赖树如下：

```
[INFO] com.combat:spring-logback:jar:1.0-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter-web:jar:2.7.12:compile
[INFO]    +- org.springframework.boot:spring-boot-starter:jar:2.7.12:compile
[INFO]    |  +- org.springframework.boot:spring-boot:jar:2.7.12:compile
[INFO]    |  +- org.springframework.boot:spring-boot-autoconfigure:jar:2.7.12:compile
```

```
[INFO]    |  +- org.springframework.boot:spring-boot-starter-logging:jar:2.7.12:compile
[INFO]    |  |  +- ch.qos.logback:logback-classic:jar:1.2.12:compile
[INFO]    |  |  |  +- ch.qos.logback:logback-core:jar:1.2.12:compile
[INFO]    |  |  |  \- org.slf4j:slf4j-api:jar:1.7.32:compile
[INFO]    |  |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.17.2:compile
[INFO]    |  |  |  \- org.apache.logging.log4j:log4j-api:jar:2.17.2:compile
[INFO]    |  |  \- org.slf4j:jul-to-slf4j:jar:1.7.36:compile
```

可以看到`spring-boot-starter-web`已经添加了依赖`logback-classic`、`log4j-to-slf4j`和`jul-to-slf4j`。

> 由此看出，`SpringBoot`通过`spring-boot-starter-logging`不仅整合了`logback`，还整合了`log4j`。

如果你还不确定用哪些`SpringBoot`的`start`，那就直接使用`spring-boot-starter-logging`即可：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
    <version>2.7.12</version>
</dependency>
```

配置文件的读取略有区别，初了可以自动读取`logback.xml`，`SpringBoot`还支持下面的格式：

* *logback-spring.xml*
* *logback.xml*
* *logback-spring.groovy*
* *logback.groovy*

官方更建议使用`logback-spring.xml`。

配置文件的格式没有变化。

如果我们需要修改日志等级，可以修改`logback-spring.xml`配置文件，也可以在我们项目的配置文件`application.properties`中指定：

```
logger.level.root = INFO
```

`application.properties`中的优先级高于`logback-spring.xml`。

5.2.源码分析

`SpringBoot`通过事件的发布订阅模式（即观察者模式）完成`Logback`日志框架的初始化。我们首先简单回顾一下`SpringBoot`的事件发布和订阅（这部分详细分析属于`SpringBoot`源码范畴，这里不做太深入的介绍，大家自行网上搜资料学习），然后看`SpringBoot`如果完成对`Logback`的初始化。

5.2.1.`SpringBoot`事件机制

`SpringBoot`的事件机制是对`JDK`的事件机制的扩展。`JDK`中定义了事件和监听者：

* `JDK`事件

```
package java.util;

public class EventObject implements java.io.Serializable {
    private static final long serialVersionUID = 5516075349620653480L;
    protected transient Object  source;

    public EventObject(Object source) {
        if (source == null)
            throw new IllegalArgumentException("null source");
        this.source = source;
    }
```

```
    public Object getSource() {
        return source;
    }

    public String toString() {
        return getClass().getName() + "[source=" + source + "]";
    }
}

```

* `JDK`监听者

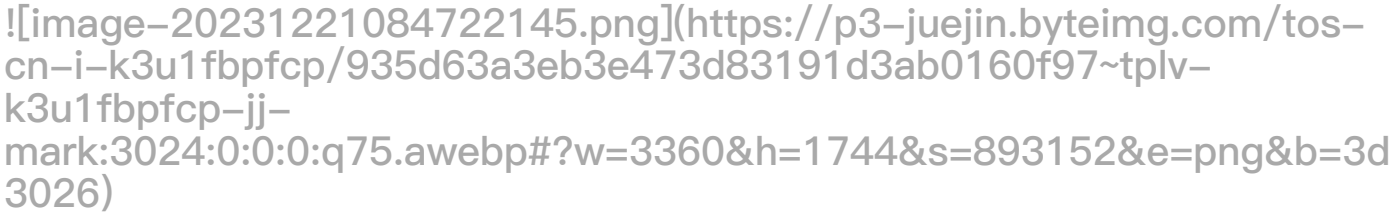
```
package java.util;

/**
 * A tagging interface that all event listener interfaces must extend.
 * @since JDK1.1
 */
public interface EventListener {
}

```

`SpringBoot`中定义了很多种事件，这些事件的基类是`ApplicationEvent`，而`ApplicationEvent`则继承自`JDK`的`EventObject`：

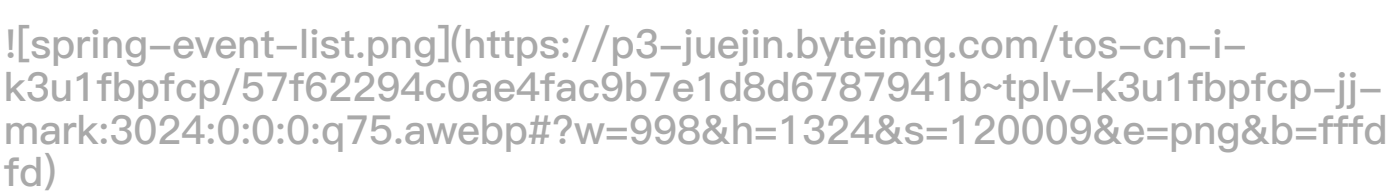
```
package org.springframework.context;

import java.time.Clock;
import java.util.EventObject;

public abstract class ApplicationEvent extends EventObject {

/** use serialVersionUID from Spring 1.2 for interoperability. */
private static final long serialVersionUID = 7099057708183571937L;
/** System time when the event happened. */
private final long timestamp;

    public ApplicationEvent(Object source) {
super(source);
this.timestamp = System.currentTimeMillis();
```

```
}

public ApplicationEvent(Object source, Clock clock) {
super(source);
this.timestamp = clock.millis();
}

public final long getTimestamp() {
return this.timestamp;
}
}
```

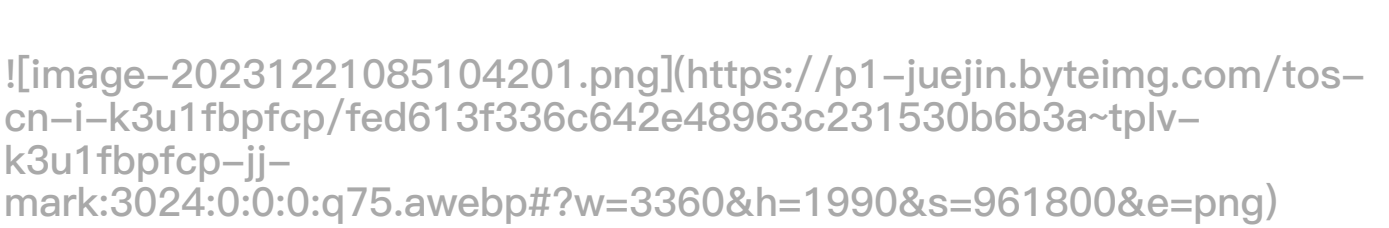
相比`JDK`的`EventObject`，`ApplicationEvent`多了`timestamp`字段。
`SpringBoot`的事件包括下面这些：



`SpringBoot`会在不同的运行阶段发布对应的事件：

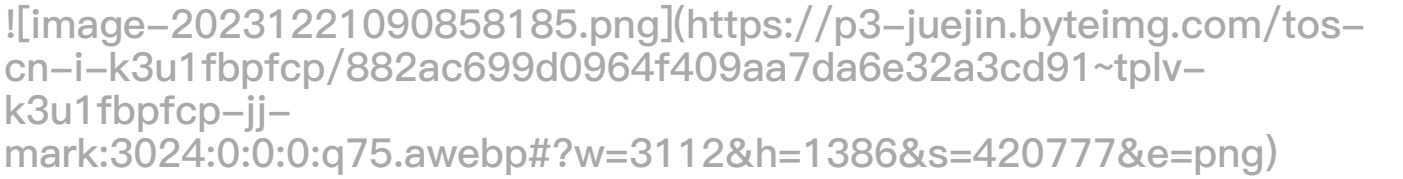


`SpringBoot`的监听者都实现了`ApplicationListener`，它又继承自`JDK`的`EventListener`：



`SpringBoot`的事件发布由`ApplicationEventPublish`接口的`publishEvent`方法完成，而`AbstractApplicationContext`唯一实现了该方法，所以有它完成事件的发布。

监听者有很多，如何让多个监听者顺序执行呢？—— 让监听者实现`Ordered`接口，然后实现`getOrder`方法，给这些监听者指定顺序。例如`SmartApplicationListener`：



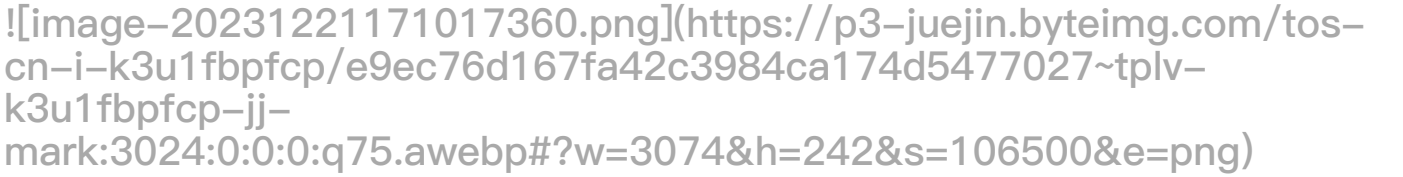
5.2.2.`SpringBoot`完成`Logback`的初始化

`SpringBoot`通过`LoggingApplicationListener`来完成`Logback`的初始化：

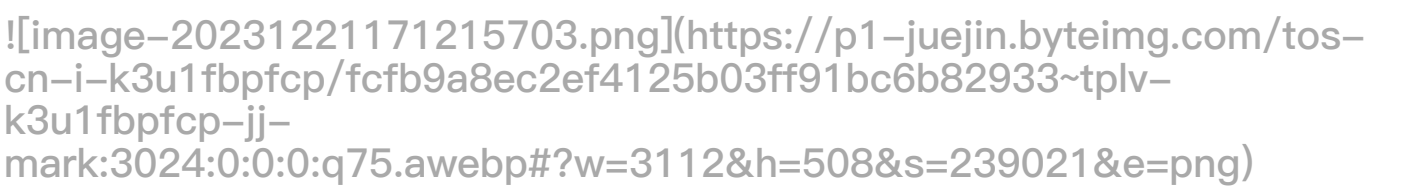
主要看一下下面这几个事件的处理逻辑：

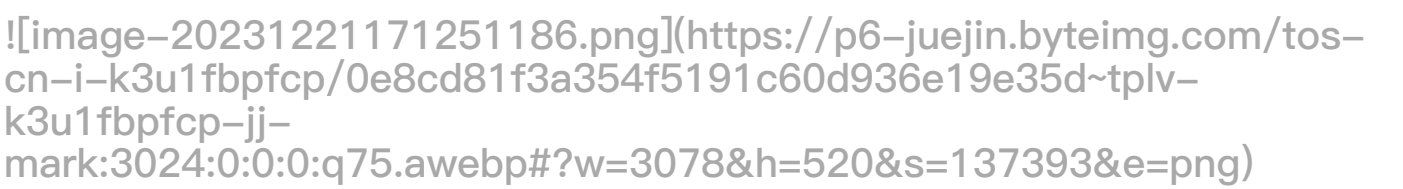
* *onApplicationStartingEvent*
* *onApplicationEnvironmentPreparedEvent*
* *onApplicationPreparedEvent*

`onApplicationStartingEvent`事件是容器刚启动时触发的，主要完成`SpringBoor`容器中的`Bean`实例化前的一些准备工作：



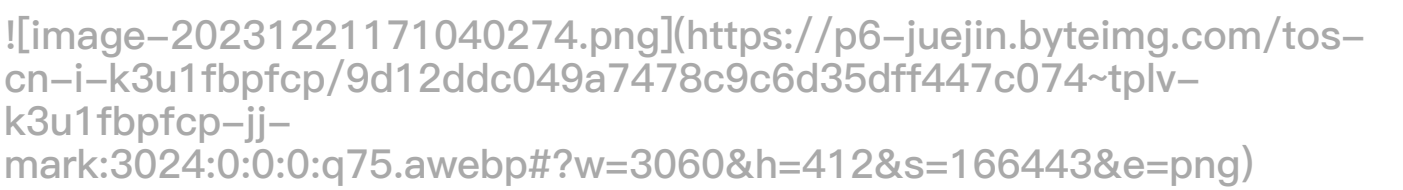
`beforeInitialize()`是一个抽象方法，`SpringBoot`分别实现了`Log4j2`、`Logback`和`Slf4j`等日志框架初始化的准备工作：





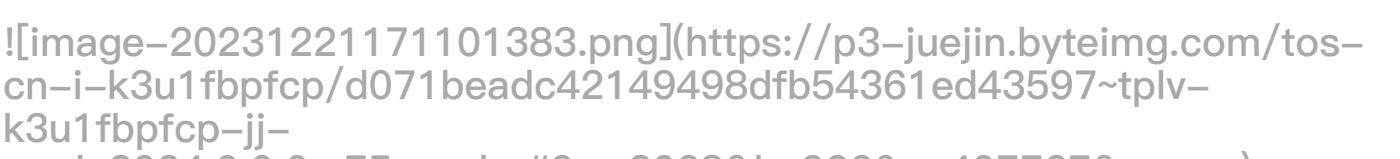
如果你认真研读了前面几章关于`Logback`源码的分析，到这里应该很熟悉了：这里直接调用了`Logback`的绑定方法`StaticLoggerBinder.getSingleton()`完成了`Logback`与`Slf4j`日志门面的绑定工作！

`onApplicationEnvironmentPreparedEvent`事件是`SpringBoot`创建好抽象环境类后发布的事件，这里监听到该事件后正式完成`Logback`日志对象的初始化：



初始化主要 2 个关键步骤，`initializeSystem`和`initializeFinalLoggingLevels`，前者用于设置日志文件路径，优先读取`application.properties`文件设的`logging.config`值，没有则去读取`logback-spring.xml`中的文件路径；后者用于设置日志级别，优先读取`application.properties`文件设的`logging.level.root`值，没有则去读取`logback-spring.xml`中的日志级别：

`onApplicationPreparedEvent`是`SpringBoot`已经构建好上下文以后发布的事件，这里主要完成日志`Bean`的注入工作：



到这里我们完整分析了`SpringBoot`是如何整合`Logback`日志框架的。当然，一些非主流链路的细节，我们这里就不分析了，大家感兴趣的话可以去调试和验证，我这里写再多也比不上你动手实践一遍的！

6.`Logback`常用扩展

我们如果只会配置`Logback`，然后让它打印规定格式的日志，很多时候还是无法满足业务要求的。就比如我想在所有业务日志中统一打印一些业务字段，但是这些字段`Logback`本身是不认识的，无法为我们解析和转换怎么办？如果我想对日志中的一些敏感信息进行脱敏处理怎么办？本节我们就列举一些常用的`Logback`扩展功能。

6.1.添加业务字段

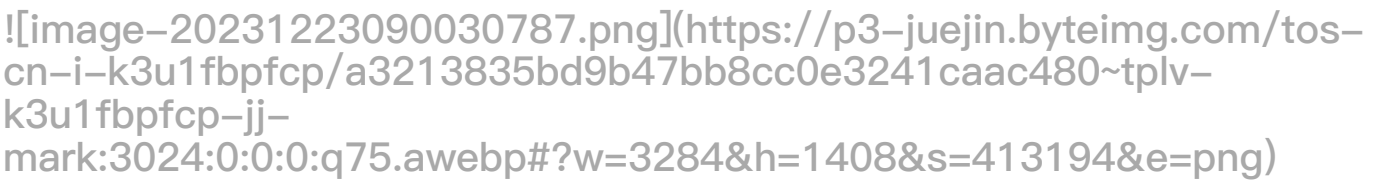
业务中最常见的需求就是在日志中打印一次请求的`RequestId`以串联起一次请求的所有日志，或者打印用户的`uid`以方便确认某个用户的行为。我们给一个如何在`SpringBoot`项目的日志中添加`RequestId`的示例。

给日志添加业务字段的核心手段是`AOP`（AOP 的概念就不在这里普及了，大家自行了解），所以我们在第 5 章`SpringBoot`项目的基础上，添加依赖：

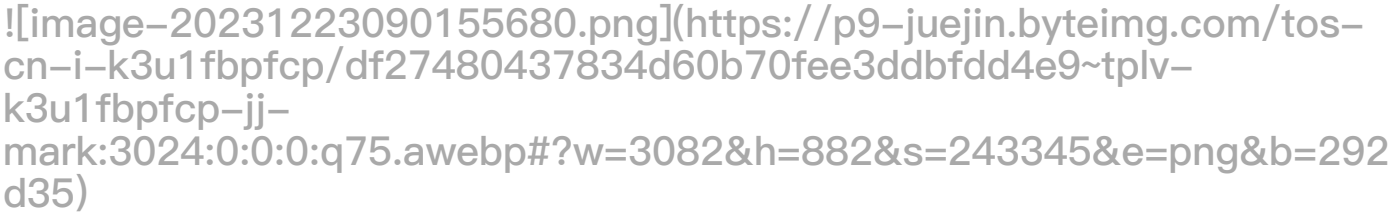
```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
    <version>2.7.12</version>
</dependency>

```

我们的项目结构如下：

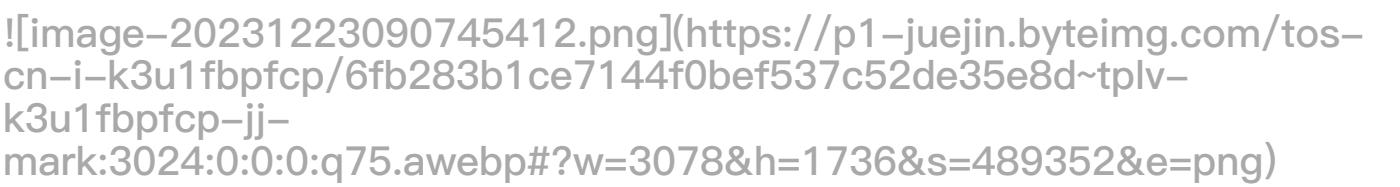


主要是添加了一个`切面类 LoggerAspect`：

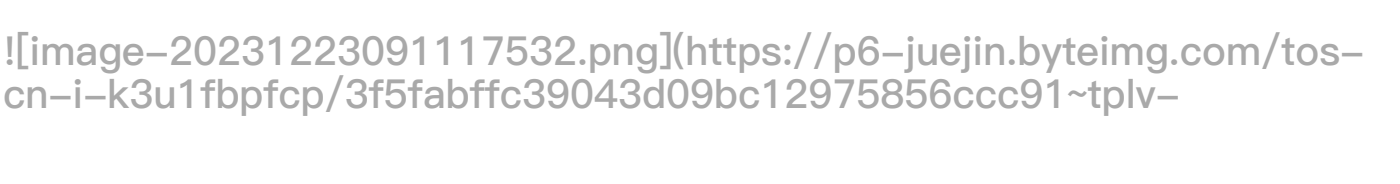


`@Component`帮助我们将切面类注入到`SpringBoot`容器中，`@Aspect`能够让`SpringBoot`识别到这是一个切面类。`@Pointcut`点指向的是`controller`包下的所有方法（这是`@Pointcut`的语法规则）。

然后定义`@Before`注解方法：

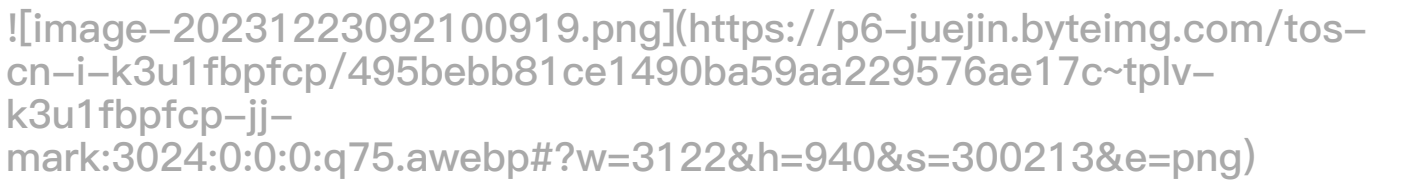


如果请求`Header`中包含`RequestId`字段，则放到`dataMap`当中，如果是请求入参当中有`RequestId`，会覆盖`Header`的值，最终会在`recoredRequestId`方法中将`RequestId`字段放到`MDC`当中，这样就完成对`RequestId`字段的添加了（如果没有值，这里通过`UUID`做默认处理）：

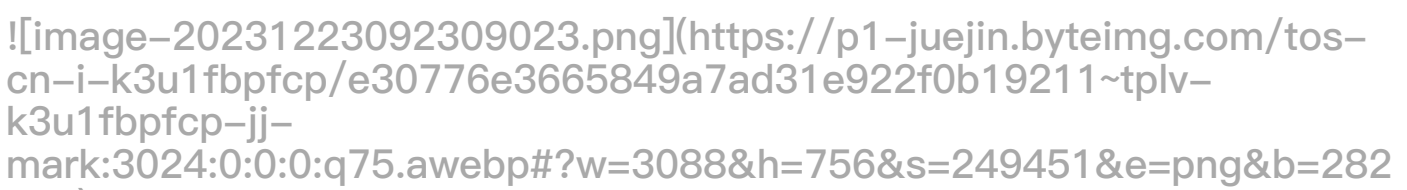


k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=2238&h=406&s=132432&e=png)

然后是`@AfterRunning`注解打印请求完成时的日志：



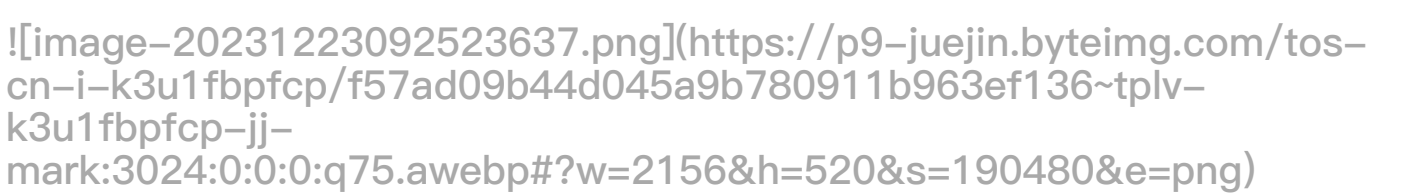
打印的日志中还包含了接口耗时信息，注意这里还用到了`ThreadLocal`。打印完请求返回日志后要进行资源清理，防止内存泄漏或者请求直接串值。我们的示例请求接口很简单：



最后别忘记在我们的`Logback`配置文件`logback-spring.xml`的日志格式中添加`requestId`字段：

```
<property name="FILE_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %X{requestId} %-5level %logger{50}: %m%n"/>
```

到此为止我们就完成业务字段`requestId`的添加了，来看一下执行效果：



6.2.日志脱敏

如果我们的业务中涉及一些账号、密码、验证码或者提取码等敏感信息的打印，为了方式日志泄漏后对用户造成不必要的损失，在业务中往往需要对日志进行脱敏处理。脱敏即去除敏感信息，例如：我们的邮箱源数据为`13312348080@163.com`，则日志中仅打印`1*********8@163.com`；而密码我们则固定打印为`******`。

`Java`项目中的脱敏手段主要有 2 中方式：

* 在需要脱敏的字段上添加注解，通过`Logback`对带有注解的字段进行统一处理，这种方式直接引入[houbb/sensitive](http://cxyroad.com/”https://github.com/houbb/sensitive”)即可；
* 直接在`Logback`配置文件中添加自定义规则，实现自定义`Converter`类。

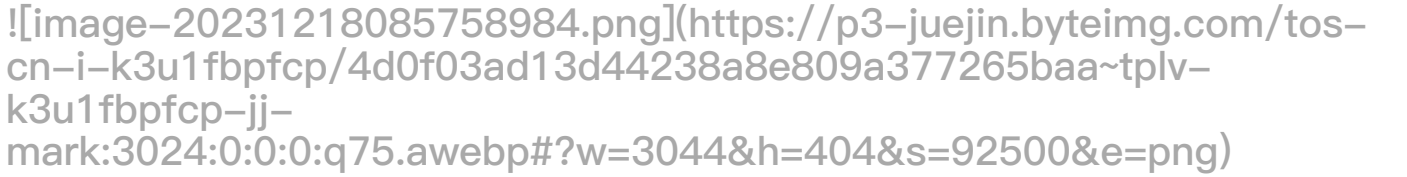
第一种方式我们暂时不在这里展开讲，如果是在业务开发早期就对脱敏有规划，那我建议优先使用这种方式，因为这种方式性能好还不会有遗漏。但是如果对于一个我们不想有任何代码改动的项目，对性能以及脱敏的结果又没有那么苛刻，我更建议使用第二种，这种方式也可以应付大多是情况了，下面我们介绍一下第 2 中方式。

首先，我们要继承`MessageConverter`类，实现自定义的日志内容转换类，在这个类里完成日志的脱敏工作：

```
package com.combat.logback.converter;

import ch.qos.logback.classic.pattern.MessageConverter;
import ch.qos.logback.classic.spi.ILoggingEvent;

import java.util.HashMap;
import java.util.Map;

public class SensitiveMessageConverter extends MessageConverter {
    private static final Map<String, String> regexMap = new
HashMap<>();

    static {
        // 手机号脱敏
        regexMap.put(”(mobile|手机号)(=|=\[|\”:\”|:|：|=’|’:’)(1)([3-
9]{2})(\d{4})(\d{4})(\]|\”|’|)”, ”$1$2$3$4****$6$7”);
        // 码脱敏
regexMap.put(”(password|pwd|PASSWORD)([=|\”|\\\\|:|：|\\[|’|\\s]+)([\\w
```

```
\\d\\s$@$!%*?&/+=]{1,128})(\\w*)", "$1$2******$4");
    }

    @Override
    public String convert(ILoggingEvent event) {
        String oriLogMsg = event.getFormattedMessage();
        String afterLogMsg = oriLogMsg;
        if (afterLogMsg == null || afterLogMsg.length() <= 0) {
            return afterLogMsg;
        }

        for (Map.Entry<String, String> regexPair : regexMap.entrySet()) {
            afterLogMsg = afterLogMsg.replaceAll(regexPair.getKey(),
regexPair.getValue());
        }
        return afterLogMsg;
    }
}
```

我们在上面分析`Logback`源码时知道`Logback`提供的`Converter`的继承类有很多，这里为什么实现类`MessageConverter`就可以呢？我们看一下该类的内容：



`MessageConverter`是日志格式化完成的最后一个`Converter`，它将格式化好的内容直接返回给`appender`，所以我们继承该类后重写`convert`方法，对日志内容进行最后的修改，即可完成脱敏。

然后，我们还需在`logback.xml`中添加一条自定义规则：

```
<conversionRule conversionWord="m"
converterClass="com.combat.logback.converter.SensitiveMessageConverter"/>
```

`conversionWord`的值要和我们的日志格式中的关键字对应（`%m`，这里还支持`%msg`和`%message`）：

```
<property name=”FILE_PATTERN” value=”%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}: %m%n”/>
```

`converterClass`指定我们自定义的脱敏类。

7.参考文献

1. `Logback`官方文档：[logback.qos.ch/manual/intr...](http://cxyroad.com/ ”https://logback.qos.ch/manual/introduction.html”)
2. `Logback`中文文档：[logbackcn.gitbook.io/logback/13-...](http://cxyroad.com/ ”https://logbackcn.gitbook.io/logback/13-di-shi-san-zhang-cong-log4j-qian-yi”)
3. `Log4j2`官网：[logging.apache.org/log4j/2.x/i...](http://cxyroad.com/ ”https://logging.apache.org/log4j/2.x/index.html”)