11.5 如何实时将应用 Error 日志告警?

大数据时代，随着公司业务不断的增长，数据量自然也会跟着不断的增长，那么业务应用和集群服务器的的规模也会逐渐扩大，几百台服务器在一般的公司已经是很常见的了。那么将应用服务部署在如此多的服务器上，对开发和运维人员来说都是一个挑战。一个优秀的系统运维平台是需要将部署在这么多服务器上的应用监控信息汇总成一个统一的数据展示平台，方便运维人员做日常的监测、提升运维效率，还可以及时反馈应用的运行状态给应用开发人员。举个例子，应用的运行日志需要按照时间排序做一个展示，并且提供日志下载和日志搜索等服务，这样如果应用出现问题开发人员首先可以根据应用日志的错误信息进行问题的排查。那么该如何实时的将应用的 Error 日志推送给应用开发人员呢，接下来我们将讲解日志的处理方案。

11.5.1 日志处理方案的演进

日志处理的方案也是有一个演进的过程，要想弄清楚整个过程，我们先来看下日志的介绍。

什么是日志?

日志是带时间戳的基于时间序列的数据，它可以反映系统的运行状态，包括了一些标识信息（应用所在服务器集群名、集群机器 IP、机器设备系统信息、应用名、应用 ID、应用所属项目等）

日志处理方案演进

日志处理方案的演进过程：

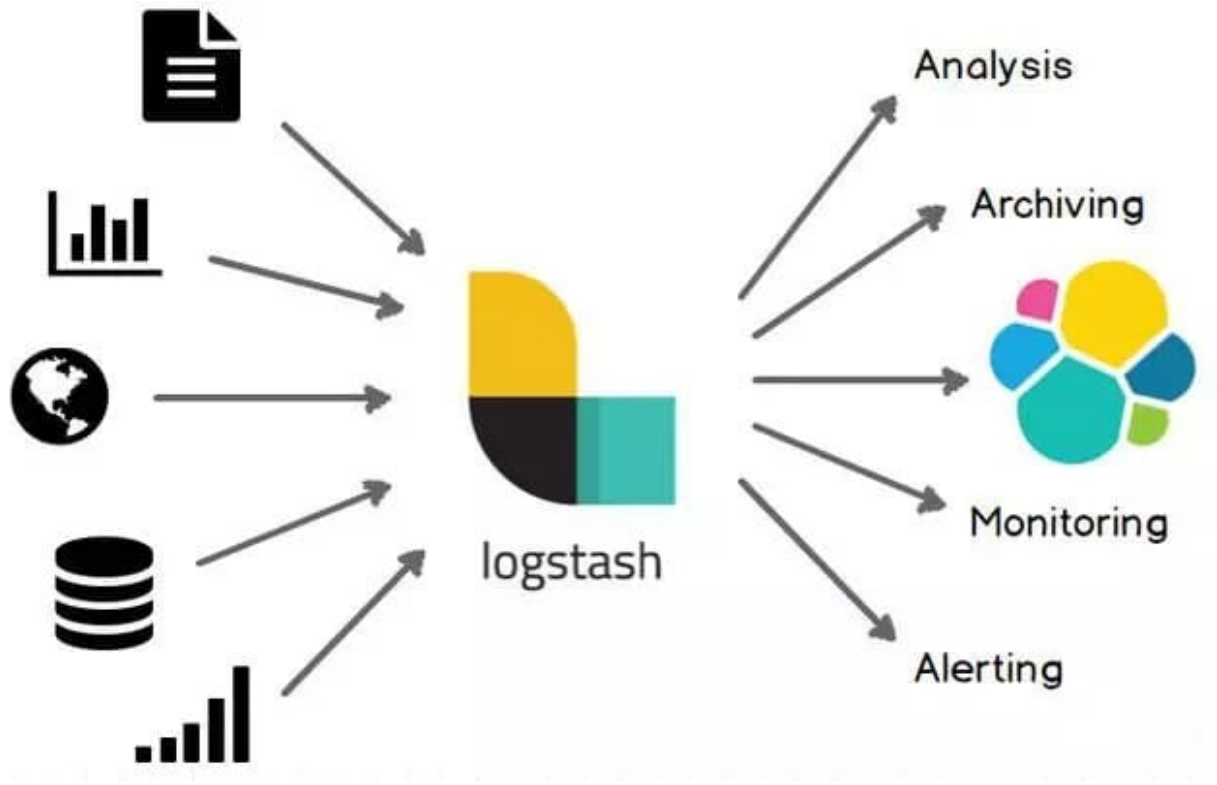
- 日志处理 v1.0: 应用日志分布在很多机器上，需要人肉手动去机器查看日志信息。
- 日志处理 v2.0: 利用离线计算引擎统一的将日志收集，形成一个日志搜索分析平台，提供搜索让用户根据关键字进行搜索和分析，缺点就是及时性比较差。
- 日志处理 v3.0: 利用 Agent 实时的采集部署在每台机器上的日志，然后统一发到日志收集平台做汇总，并提供实时日志分析和搜索的功能，这样从日志产生到搜索分析出结果只有简短的延迟（在用户容忍时间范围之内），优点是快，但是日志数据量大的情况下带来的挑战也大。

11.5.2 日志采集工具对比

上面提到的日志采集，其实现在已经有很多开源的组件支持去采集日志，比如 Logstash、Filebeat、Fluentd、Logagent 等，这里简单做个对比。

Logstash

Logstash 是一个开源数据收集引擎，具有实时管道功能。Logstash 可以动态地将来自不同数据源的数据统一起来，并将数据标准化到你所选择的目的地。如下图所示，Logstash 将采集到的数据用作分析、监控、告警等。

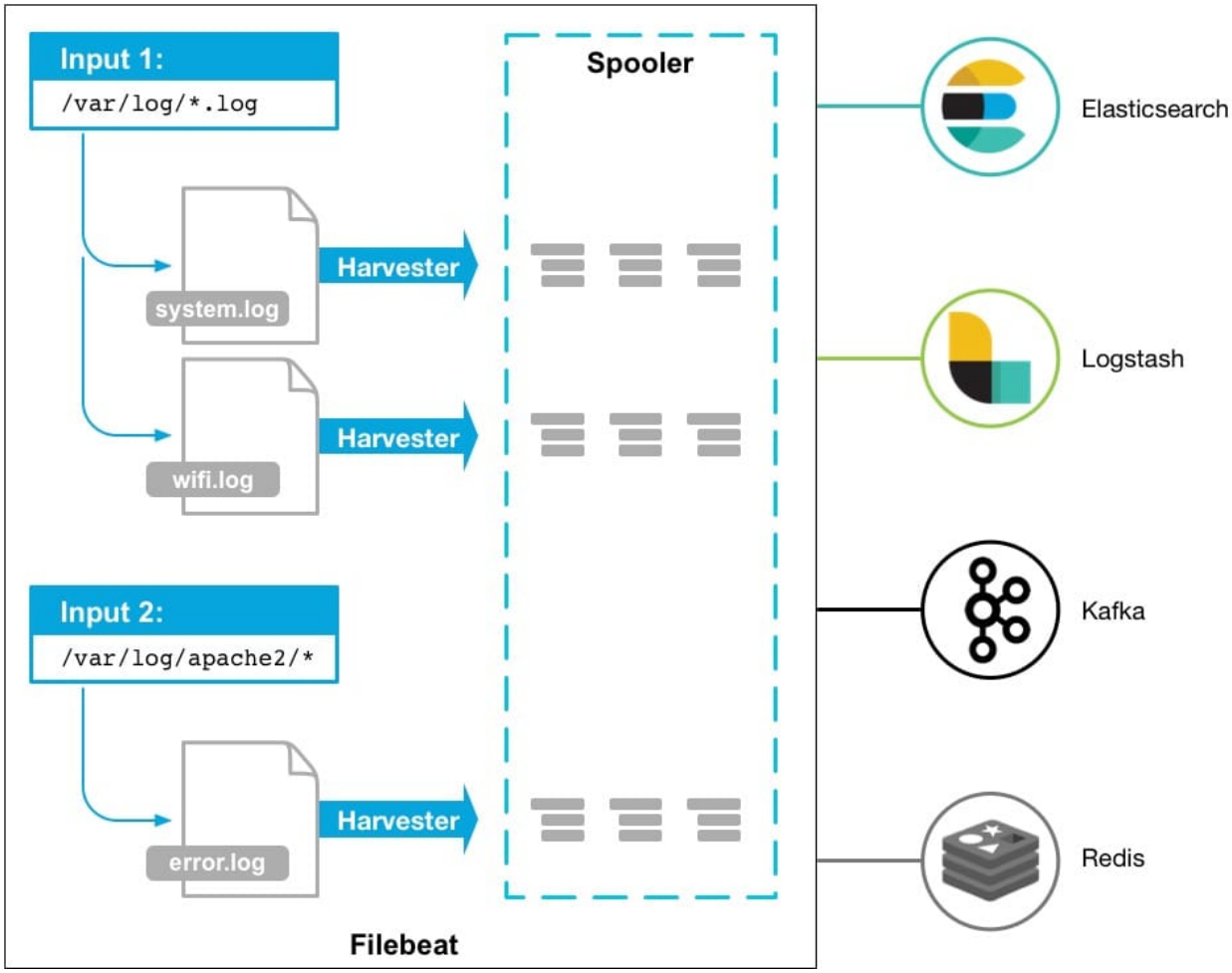


优势：Logstash 主要的优点就是它的灵活性，它提供很多插件，详细的文档以及直白的配置格式让它可以在多种场景下应用。而且现在 ELK 整个技术栈在很多公司应用的比较多，所以基本上可以在往上找到很多相关的学习资源。

劣势：Logstash 致命的问题是它的性能以及资源消耗(默认的堆大小是 1GB)。尽管它的性能在近几年已经有很大提升，与它的替代者们相比还是要慢很多的，它在大数据量的情况下会是个问题。另一个问题是它目前不支持缓存，目前的典型替代方案是将 Redis 或 Kafka 作为中心缓冲池：

Filebeat

作为 Beats 家族的一员，Filebeat 是一个轻量级的日志传输工具，它的存在正弥补了 Logstash 的缺点，Filebeat 作为一个轻量级的日志传输工具可以将日志推送到 Kafka、Logstash、ElasticSearch、Redis。它的处理流程如下图所示：

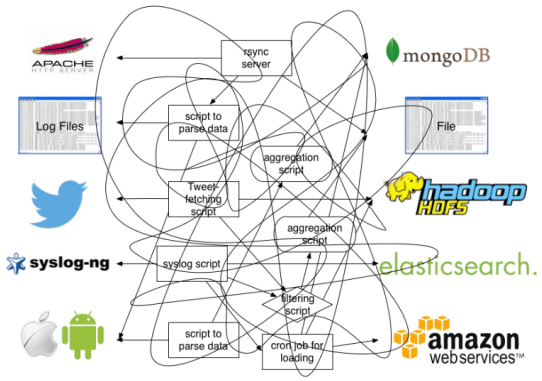


优势：Filebeat 只是一个二进制文件没有任何依赖。它占用资源极少，尽管它还十分年轻，正式因为它简单，所以几乎没有什么可以出错的地方，所以它的可靠性还是很高的。它也为我们提供了很多可以调节的点，例如：它以何种方式搜索新的文件，以及当文件有一段时间没有发生变化时，何时选择关闭文件句柄。

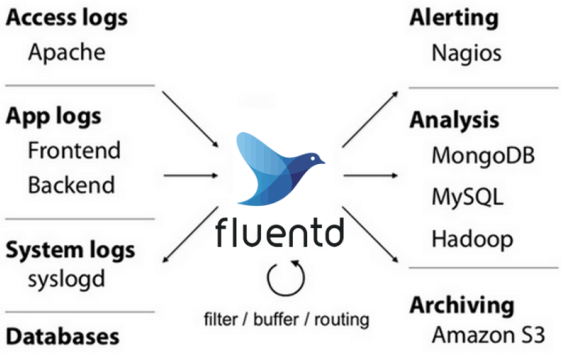
劣势：Filebeat 的应用范围十分有限，所以在某些场景下我们会碰到问题。例如，如果使用 Logstash 作为下游管道，我们同样会遇到性能问题。正因为如此，Filebeat 的范围在扩大。开始时，它只能将日志发送到 Logstash 和 Elasticsearch，而现在它可以将日志发送给 Kafka 和 Redis，在 5.x 版本中，它还具备过滤的能力。

Fluentd

Fluentd 创建的初衷主要是尽可能的使用 JSON 作为日志输出，所以传输工具及其下游的传输线不需要猜测子字符串里面各个字段的类型。这样它为几乎所有的语言都提供库，这也意味着可以将它插入到自定义的程序中。它的处理流程如下图所示：



Before Fluentd



After Fluentd

优势：和多数 Logstash 插件一样，Fluentd 插件是用 Ruby 语言开发的非常易于编写维护。所以它数量很多，几乎所有的源和目标存储都有插件(各个插件的成熟度也不太一样)。这也意味这可以用 Fluentd 来串联所有的东西。

劣势：因为在多数应用场景下得到 Fluentd 结构化的数据，它的灵活性并不好。但是仍然可以通过正则表达式来解析非结构化的数据。尽管性能在大多数场景下都很好，但它并不是最好的，它的缓冲只存在与输出端，单线程核心以及 Ruby GIL 实现的插件意味着它大的节点下性能是受限的。

Logagent

Logagent 是 Sematext 提供的传输工具，它用来将日志传输到 Logsene(一个基于 SaaS 平台的 Elasticsearch API)，因为 Logsene 会暴露 Elasticsearch API，所以 Logagent 可以很容易将数据推送到 Elasticsearch 。

优势：可以获取 /var/log 下的所有信息，解析各种格式的日志，可以掩盖敏感的数据信息。它还可以基于 IP 做 GeoIP 丰富地理位置信息。同样，它轻量又快速，可以将其置入任何日志块中。Logagent 有本地缓冲，所以在数据传输目的地不可用时不会丢失日志。

劣势：没有 Logstash 灵活。

11.5.3 日志结构设计

前面介绍了日志和对比了常用日志采集工具的优势和劣势，通常在不同环境，不同机器上都会部署日志采集工具，然后采集工具会实时的将新的日志采集发送到下游，因为日志数据量毕竟大，所以建议发到 MQ 中，比如 Kafka，这样再想怎么处理这些日志就会比较灵活。假设我们忽略底层采集具体是哪种，但是规定采集好的日志结构化数据如下：

```java
public class LogEvent {
    private String type;//日志的类型(应用、容器、...)
    private Long timestamp;//日志的时间戳
    private String level;//日志的级别(debug/info/warn/error)
    private String message;//日志内容
    //日志的标识(应用 ID、应用名、容器 ID、机器 IP、集群名、...)
    private Map<String, String> tags = new HashMap<>();
}
```

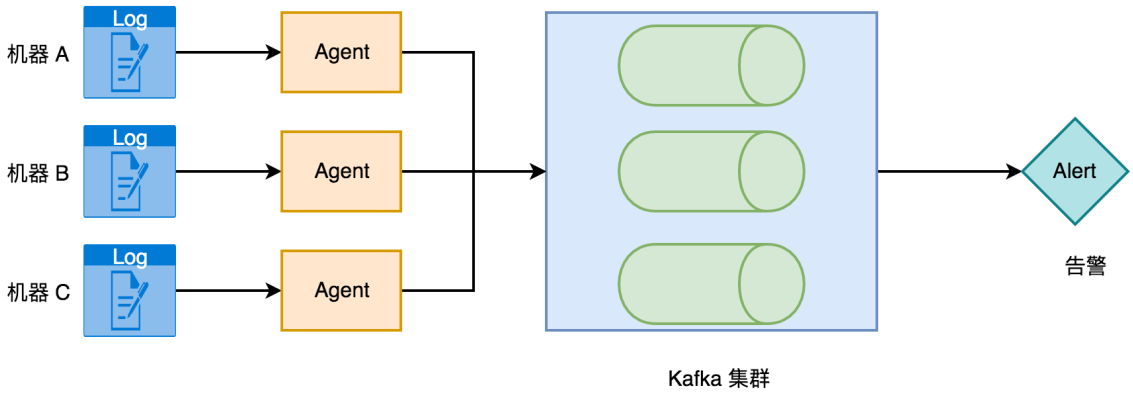
然后上面这种 LogEvent 的数据（假设采集发上来的是这种结构数据的 JSON 串，所以需要在 Flink 中做一个反序列化解析）就会往 Kafka 不断的发送数据，样例数据如下：

```json
{
  "type": "app",
  "timestamp": 1570941591229,
  "level": "error",
  "message": "Exception in thread \"main\" java.lang.NoClassDefFoundError: org/apache/flink/api/common/ExecutionConfig$GlobalJobParameters",
  "tags": {
    "cluster_name": "zhisheng",
    "app_name": "zhisheng",
    "host_ip": "127.0.0.1",
    "app_id": "21"
  }
}
```

那么在 Flink 中如何将应用异常或者错误的日志做实时告警呢?

11.5.4 异常日志实时告警项目架构

整个异常日志实时告警项目的架构如下图所示。



应用日志散列在不同的机器，然后每台机器都有部署采集日志的 Agent（可以是上面的 Filebeat、Logstash 等），这些 Agent 会实时的将分散在不同机器、不同环境的应用日志统一的采集发到 Kafka 集群中，然后告警这边是有一个 Flink 作业去实时的消费 Kafka 数据做一个异常告警计算处理。如果还想做日志的搜索分析，可以起另外一个作业去实时的将 Kafka 的日志数据写入进 ElasticSearch，再通过 Kibana 页面做搜索和分析。

11.5.5 日志数据发送到 Kafka

上面已经讲了日志数据 LogEvent 的结构和样例数据，因为要在服务器部署采集工具去采集应用日志数据对于本地测试来说可能稍微复杂，所以在这里就只通过代码模拟构造数据发到 Kafka 去，然后在 Flink 作业中去实时消费 Kafka 中的数据，下面演示构造日志数据发到 Kafka 的工具类，这个工具类主要分两块，构造 LogEvent 数据和发送到 Kafka。

```
@Slf4j
public class BuildLogEventDataUtil {
    //Kafka broker 和 topic 信息
    public static final String BROKER_LIST = "localhost:9092";
    public static final String LOG_TOPIC = "zhisheng_log";

    public static void writeDataToKafka() {
        Properties props = new Properties();
        props.put("bootstrap.servers", BROKER_LIST);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        KafkaProducer producer = new KafkaProducer<String, String>(props);

        for (int i = 0; i < 10000; i++) {
            //模拟构造 LogEvent 对象
            LogEvent logEvent = new LogEvent().builder()
                    .type("app")
                    .timestamp(System.currentTimeMillis())
                    .level(logLevel())
                    .message(message(i + 1))
                    .tags(mapData())
                    .build();
//            System.out.println(logEvent);
            ProducerRecord record = new ProducerRecord<String, String>
(LOG_TOPIC, null, null, GsonUtil.toJson(logEvent));
            producer.send(record);
        }
        producer.flush();
    }

    public static void main(String[] args) {
        writeDataToKafka();
    }

    public static String message(int i) {
        return "这是第 " + i + " 行日志！";
    }

    public static String logLevel() {
        Random random = new Random();
        int number = random.nextInt(4);
        switch (number) {
            case 0:
                return "debug";
            case 1:
                return "info";
```

```
            case 2:
                return "warn";
            case 3:
                return "error";
            default:
                return "info";
        }
    }

    public static String hostIp() {
        Random random = new Random();
        int number = random.nextInt(4);
        switch (number) {
            case 0:
                return "121.12.17.10";
            case 1:
                return "121.12.17.11";
            case 2:
                return "121.12.17.12";
            case 3:
                return "121.12.17.13";
            default:
                return "121.12.17.10";
        }
    }

    public static Map<String, String> mapData() {
        Map<String, String> map = new HashMap<>();
        map.put("app_id", "11");
        map.put("app_name", "zhisheng");
        map.put("cluster_name", "zhisheng");
        map.put("host_ip", hostIp());
        map.put("class", "BuildLogEventDataUtil");
        map.put("method", "main");
        map.put("line", String.valueOf(new Random().nextInt(100)));
        //add more tag
        return map;
    }
}
```

如果之前 Kafka 中没有 zhisheng_log 这个 topic，运行这个工具类之后也会自动创建这个 topic 了。

11.5.6 Flink 实时处理日志数据

在 3.7 章中已经讲过如何使用 Flink Kafka connector 了，接下来就直接写代码去消费 Kafka 中的日志数据，作业代码如下：

```
public class LogEventAlert {
    public static void main(String[] args) throws Exception {
        final ParameterTool parameterTool =
ExecutionEnvUtil.createParameterTool(args);
        StreamExecutionEnvironment env =
ExecutionEnvUtil.prepare(parameterTool);
        Properties properties =
KafkaConfigUtil.buildKafkaProps(parameterTool);
```

```
        FlinkKafkaConsumer011<LogEvent> consumer = new
FlinkKafkaConsumer011<>(
                parameterTool.get("log.topic"),
                new LogSchema(),
                properties);
        env.addSource(consumer)
                .print();
        env.execute("log event alert");
    }
}
```

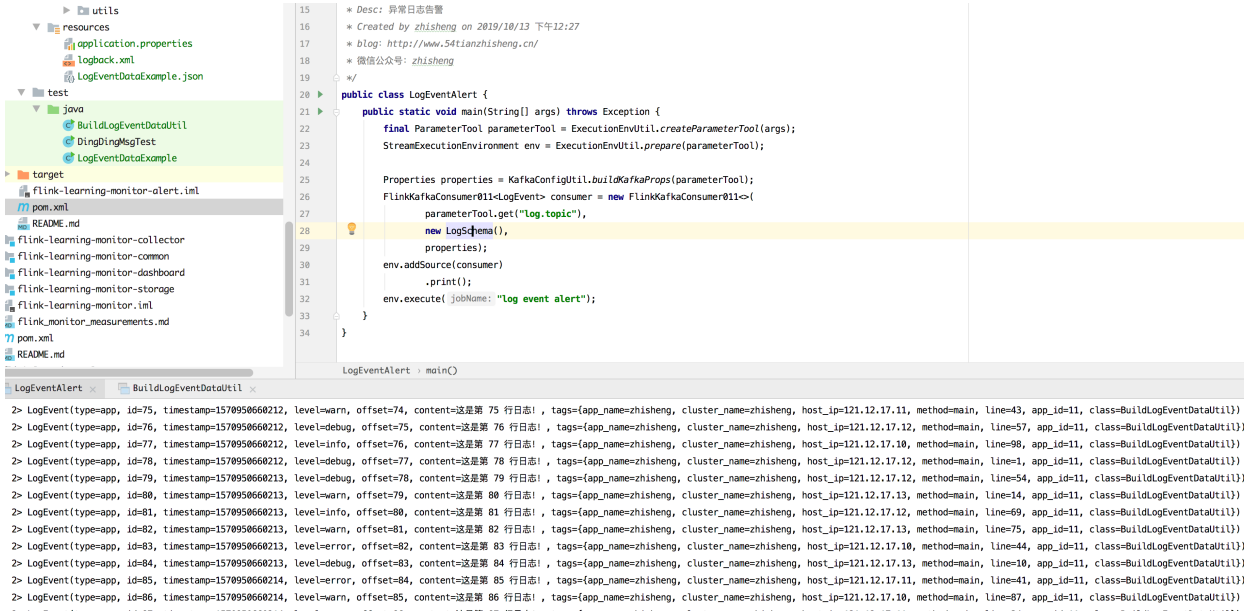
因为 Kafka 的日志数据是 JSON 的，所以在消费的时候需要额外定义 Schema 来反序列化数据，定义的 LogSchema 如下：

```
public class LogSchema implements DeserializationSchema<LogEvent>,
SerializationSchema<LogEvent> {

    private static final Gson gson = new Gson();

    @Override
    public LogEvent deserialize(byte[] bytes) throws IOException {
        return gson.fromJson(new String(bytes), LogEvent.class);
    }

    @Override
    public boolean isEndOfStream(LogEvent logEvent) {
        return false;
    }

    @Override
    public byte[] serialize(LogEvent logEvent) {
        return gson.toJson(logEvent).getBytes(Charset.forName("UTF-8"));
    }

    @Override
    public TypeInformation<LogEvent> getProducedType() {
        return TypeInformation.of(LogEvent.class);
    }
}
```

配置文件中设置如下：

```
kafka.brokers=localhost:9092
kafka.group.id=zhisheng
log.topic=zhisheng_log
```

接下来先启动 Kafka，然后运行 BuildLogEventDataUtil 工具类，往 Kafka 中发送模拟的日志数据，接下来运行 LogEventAlert 类，去消费将 Kafka 中的数据做一个验证，运行结果如下图所示，可以发现有日志数据打印出来了。

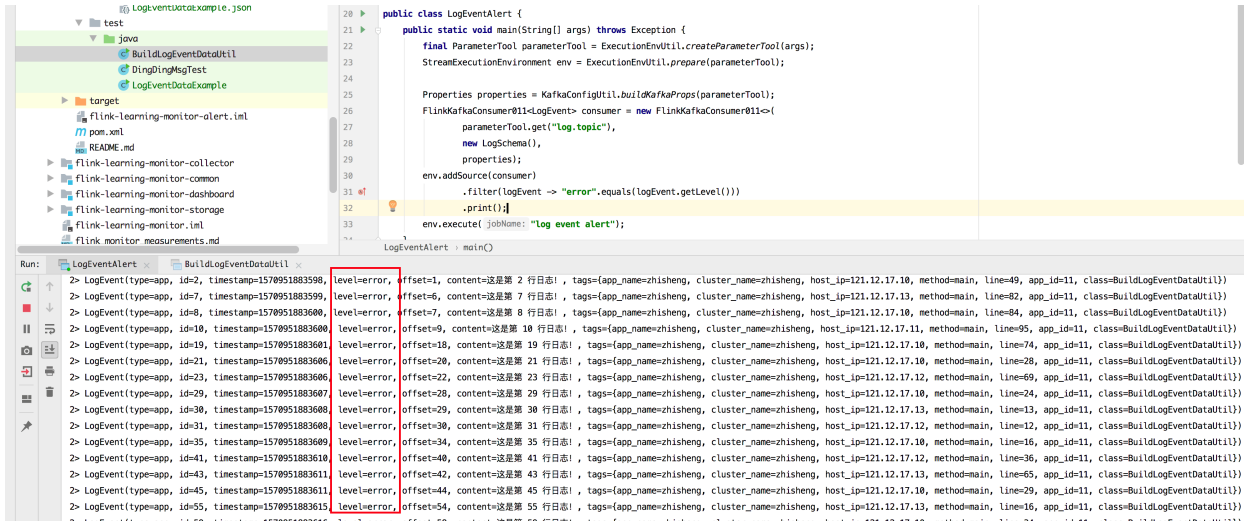


11.5.7 处理应用异常日志

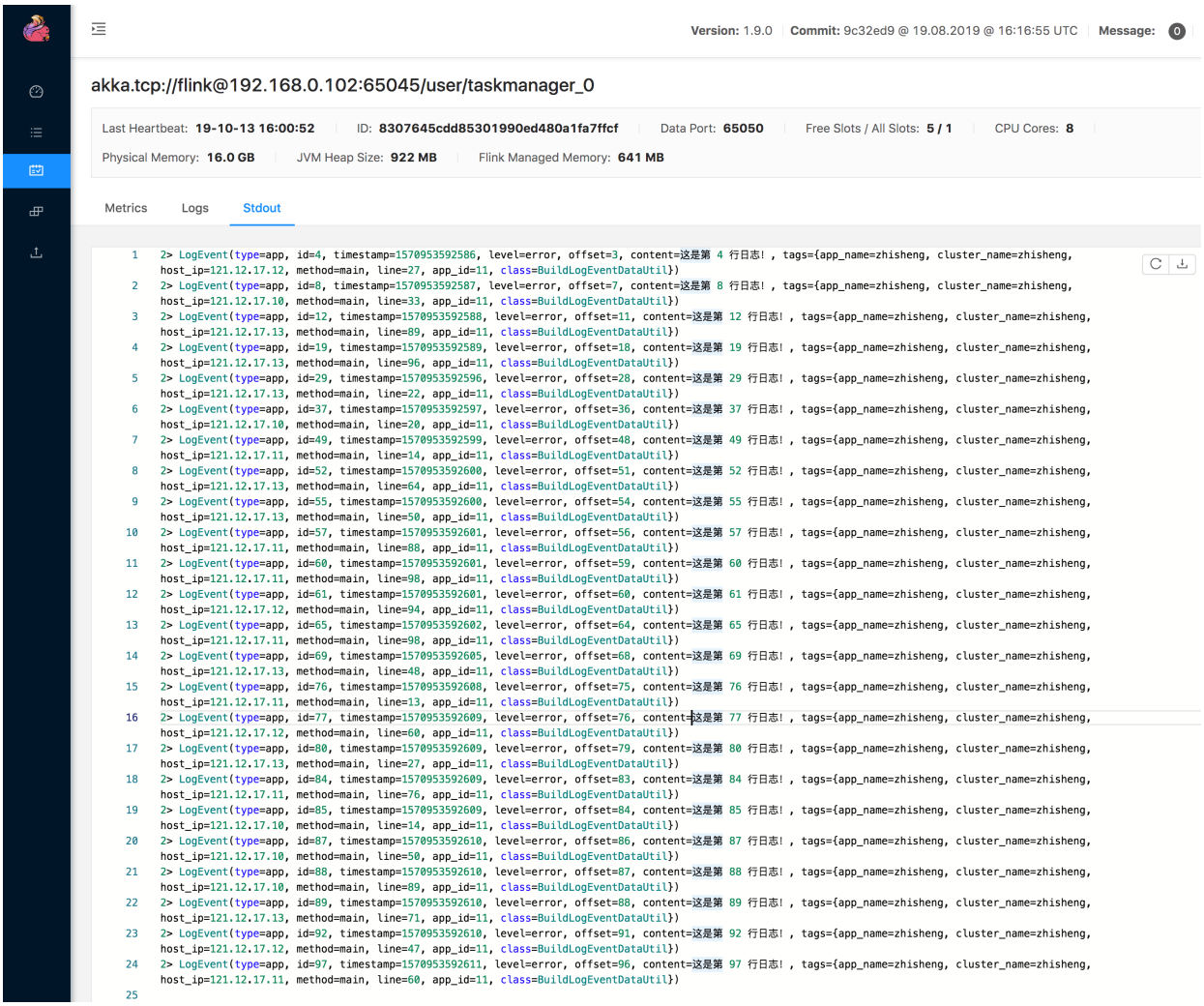
上面已经能够处理这些日志数据了，但是需求是要将应用的异常日志做告警，所以在消费到所有的数据后需要过滤出异常的日志，比如可以使用 filter 算子进行过滤。

```
1  .filter(logEvent -> "error".equals(logEvent.getLevel()))
```

过滤后只有 error 的日志数据打印出来了，如下图所示：



再将作业打包通过 UI 提交到集群运行的结果如下：



再获取到这些 Error 类型的数据后，就可以根据这个数据构造成一个新的 Event，组装成告警消息，然后在 Sink 处调用下游的通知策略进行告警通知，当然这些告警通知策略可能会很多，然后还有收敛策略。具体的通知策略和收敛策略在这节不做细讲，最后发出的应用异常日志告警消息中会携带一个链接，点击该链接可以跳转到对应的应用异常页面，这样就可以查看应用堆栈的详细日志，更加好定位问题。

11.5.8 小结与反思

本节开始讲了日志处理方案的演进，接着分析最新日志方案的实现架构，包含它的日志结构设计和异常日志实时告警的方案，然后通过模拟日志数据发送到 Kafka，Flink 实时去处理这种日志的数据进行告警。

本节涉及的代码地址：[flink-learning-monitor-alert]