基于Spark的公安大数据实时运维技术实践

公安行业存在数以万计的前后端设备，前端设备包括相机、检测器及感应器，后端设备包括各级中心机房中的服务器、应用服务器、网络设备及机房动力系统，数量巨大、种类繁多的设备给公安内部运维管理带来了巨大挑战。传统通过ICMP/SNMP、Trap/Syslog等工具对设备进行诊断分析的方式已不能满足实际要求，由于公安内部运维管理的特殊性，现行通过ELK等架构的方式同样也满足不了需要。为寻求合理的方案，我们将目光转向开源架构，构建了一套适用于公安行业的实时运维管理平台。

实时运维平台整体架构

数据采集层：Logstash+Flume，负责在不同场景下收集、过滤各类前后端硬件设备输出的Snmp Trap、Syslog日志信息以及应用服务器自身产生的系统和业务日志；

数据传输层：采用高吞吐的分布式消息队列Kafka集群，保证汇聚的日志、消息的可靠传输；

数据处理层：由Spark实时Pull Kafka数据，通过Spark Streaming以及RDD操作进行数据流的处理以及逻辑分析；

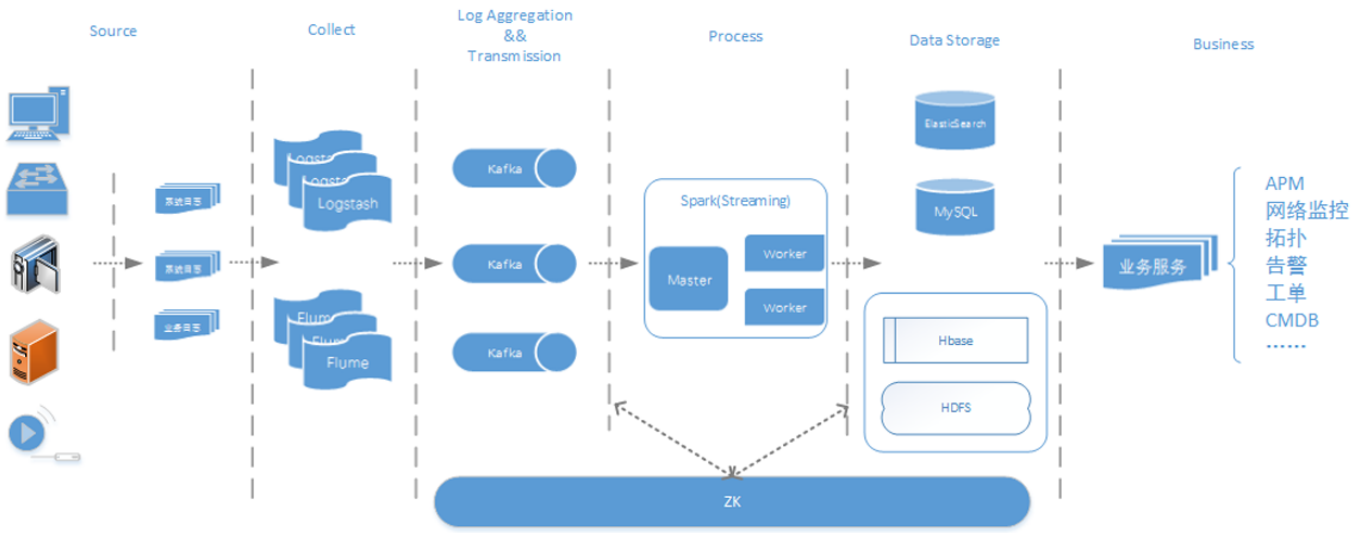
数据存储层：实时数据存入MySQL中便于实时的业务应用和展示；全量数据存入ES以及HBase中便于后续的检索分析；

业务服务层：基于存储层，后续的整体业务应用涵盖了APM、网络监控、拓扑、告警、工单、CMDB等。

整体系统涉及的主要开源框架情况如下：

开源框架	版本	开源框架	版本
Apache Spark	2.0.1	Apache Kafka	2.10-0.10.0.1
Apache Flume	1.6.0	Apache Hadoop	2.7.3
Logstash	2.3.4	Zookeeper	3.4.8

另外，整体环境基于JDK 8以及Scala 2.10.4。公安系统设备种类繁多，接下来将以交换机Syslog日志为例，详细介绍日志处理分析的整体流程。下图为公安实时运维平台整体架构



如果想
及时了解Spar
k、Flink、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Flume+Logstash日志收集

Flume是Cloudera贡献的一个分布式、可靠及高可用的海量日志采集系统，支持定制各类Source（数据源）用于数据收集，同时提供对数据的简单处理以及通过缓存写入Sink（数据接收端）的能力。

Flume中，Source、Channel及Sink的配置如下：

```
# 为该Flume Agent的source、channel以及sink命名
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# 配置Syslog源
a1.sources.r1.type = syslogtcp
a1.sources.r1.port = 5140
a1.sources.r1.host = localhost

# Kafka Sink的相关配置
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.topic = syslog-kafka
a1.sinks.k1.brokerList = gtcluster-slave01:9092
a1.sinks.k1.requiredAcks = 1
a1.sinks.k1.batchSize = 20
a1.sinks.k1.channel = c1

# Channel基于内存作为缓存
```

```
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# 将Source以及Sink绑定至Channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

该配置通过syslog source配置localhost tcp 5140端口来接收网络设备发送的Syslog信息，event缓存在内存中，再通过KafkaSink将日志发送到kafka集群中名为 syslog-kafka 的topic中。

Logstash来自Elastic公司，专为收集、分析和传输各类日志、事件以及非结构化的数据所设计。它有三个主要功能：事件输入（Input）、事件过滤器（Filter）以及事件输出（Output），在后缀为.conf的配置文件中设置，本例中Syslog配置如下：

```
# syslog.conf
input {
  Syslog {
    port => "514"
  }
}
filter {
}
output {
  kafka {
    bootstrap_servers => "192.168.8.65:9092"
    topic_id => "syslog-kafka"
    compression_type => "snappy"
    codec => plain {
      format => "%{host} %{@timestamp} %{message}"
    }
  }
}
```

Input（输入）插件用于指定各种数据源，本例中的Logstash通过udp 514端口接收Syslog信息；

Filter（过滤器）插件虽然在本例中不需要配置，但它的功能非常强大，可以进行复杂的逻辑处理，包括正则表达式处理、编解码、k/v切分以及各种数值、时间等数据处理，具体可根据实际场景设置；

Output（输出）插件用于将处理后的事件数据发送到指定目的地，指定了Kafka的位置、topic以及压缩类型。在最后的Codec编码插件中，指定来源主机的IP地址（host）、Logstash处理的时间戳（@timestamp）作为前缀并整合原始的事件消息（message），方便在事件传输过程中判断Syslog信息来源。单条原始Syslog信息流样例如下：

147>12164: Oct 9 18:04:10.735: %LINK-3-UPDOWN: Interface GigabitEthernet0/16, changed state to down

Logstash Output插件处理后的信息流变成为：

19.1.1.12 2016-10-13T10:04:54.520Z <147>12164: Oct 9 18:04:10.735: %LINK-3-UPDOWN: Interface GigabitEthernet0/16, changed state to down

其中 19.1.1.12 2016-10-13T10:04:54.520Z 就是codec编码插件植入的host以及timestamp信息。处理后的Syslog信息会发送至Kafka集群中进行消息的缓存。

Kafka日志缓冲

Kafka是一个高吞吐的分布式消息队列，也是一个订阅/发布系统。Kafka集群中每个节点都有一个被称为broker的实例，负责缓存数据。Kafka有两类客户端，Producer（消息生产者的）和Consumer（消息消费者）。Kafka中不同业务系统的消息可通过topic进行区分，每个消息都会被分区，用以分担消息读写负载，每个分区又可以有多个副本来防止数据丢失。消费者在具体消费某个topic消息时，指定起始偏移量。Kafka通过Zero-Copy、Exactly Once等技术语义保证了消息传输的实时、高效、可靠以及容错性。

Kafka集群中某个broker的配置文件server.properties的部分配置如下：

```
########## Server Basics ###########
# 为每一个broker设置独立的数字作为id
broker.id=1

###### Socket Server Settings ######
# socket监听端口
port=9092

########### Zookeeper ###############
# Zookeeper的连接配置
zookeeper.connect=gtcluster-slave02:2181,gtcluster-slave03:2181,gtcluster-slave04:2181
```

```
# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=3000
```

其中需指定集群里不同broker的id，此台broker的id为1，默认监听9092端口，然后配置Zookeeper（后续简称zk）集群，再启动broker即可。

Kafka集群名为 syslog-kafka 的topic：

```
bin/kafka-topics.sh
--create
--zookeeper gtcluster-slave02:2181,gtcluster-slave03:2181,gtcluster-slave04:2181
--replication-factor 3 --partitions 3
--topic syslog-kafka
```

Kafka集群的topic以及partition等信息也可以通过登录zk来观察。然后再通过下列命令查看Kafka接收到的所有交换机日志信息：

```
bin/kafka-console-consumer.sh--zookeeper gtcluster-slave02:2181--from-beginning
--topic Syslog-kafka
```

部分日志样例如下：

```
10.1.1.10 2016-10-18T05:23:04.015Z <163>5585: Oct 18 13:22:45: %LINK-3-UPDOWN: Interface FastEthernet0/9, changed state to down
19.1.1.113 2016-10-18T05:24:04.425Z <149>10857: Oct 18 13:25:23.019 cmt: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet1/0/3, changed state to down
19.1.1.113 2016-10-18T05:24:08.312Z <149>10860: Oct 18 13:25:27.935 cmt: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet1/0/3, changed state to up
```

Spark日志处理逻辑

Spark是一个为大规模数据处理而生的快速、通用的引擎，在速度、效率及通用性上表现极为优异。

在Spark主程序中，通过Scala的正则表达式解析Kafka Source中名为 syslog-kafka 的topic中的所有Syslog信息，再将解析后的有效字段封装为结果对象，最后通过MyBatis近实时地写入MySQL中，供前端应用进行实时地可视化展示。另外，全量数据存储进入HBase及ES中，为后续海量日志的检索分析及其它更高级的应用提供支持。主程序示例代码如下：

```
object SwSyslogProcessor {
  def main(args: Array[String]): Unit = {
    // 初始化SparkContext，批处理间隔5秒
    val sparkConf: SparkConf = new SparkConf().setAppName("SwSyslogProcessorApp ")
      .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    val ssc = new StreamingContext(sparkConf, Seconds(5))
    // 定义topic
    val topic = Set("syslog-kafka")
    // 定义kafka的broker list地址
    val brokers = "192.168.8.65:9092,192.168.8.66:9092,192.168.8.67:9092"
    val kafkaParams = Map[String, String]("metadata.broker.list" -> brokers, "serializer.class" -
> "kafka.serializer.StringDecoder")
    // 通过topic以及brokers，创建从kafka获取的数据流
    val swSyslogDstream = KafkaUtils.createDirectStream[String, String, StringDecoder, String
Decoder](
      ssc, kafkaParams, topic)
    val totalcounts = ssc.sparkContext.accumulator(0L, "Total count")
    val lines = swSyslogDstream.map(x => x._2)
    // 将一行一行数据映射成SwSyslog对象
    lines.filter(x => !x.isEmpty && x.contains("%LIN") && x.contains("Ethernet")
      .map(
        x => {
          SwSyslogService.encapsulateSwSyslog(x) // 封装并返回SwSyslog
        }).foreachRDD((s: RDD[SwSyslog], time: Time) => {
      // 遍历DStream中的RDD
      if (!s.isEmpty()) {
        // 遍历RDD中的分区记录
        s.foreachPartition {
          records => {
            if (!records.isEmpty) records.toSet.foreach {
              r: SwSyslog => // 统计当前处理的记录总数
                totalcounts.add(1L) // 保存SwSyslog信息到MySQL
                SwSyslogService.saveSwSyslog(r)
            }
          }
        }
      }
    })

    ssc.start()
    ssc.awaitTermination()
```

```
}
```

整体的处理分析主要分为4步：

1、初始化SparkContext并指定Application的参数；
2、创建基于Kafka topic “syslog-kafka” 的DirectStream；
3、将获取的每一行数据映射为Syslog对象，调用Service进行对象封装并返回；
4、遍历RDD，记录不为空时保存或者更新Syslog信息到MySQL中。

Syslog POJO的部分基本属性如下：

```
@Table(name = "sw_syslog")
public class SwSyslog {
    /**
     * 日志ID
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * 设备IP
     */
    @Column(name = "dev_ip")
    private String devIp;

    /**
     * 服务器时间
     */
    @Column(name = "server_time")
    private String serverTime;

    /**
     * 信息序号
     */
    @Column(name = "syslog_num")
    private Long syslogNum;

    ......
}
```

SwSyslog 实体中的基本属性对应Syslog中的接口信息，注解中的name对应MySQL中的表sw_syslog以及各个字段，MyBatis完成成员属性和数据库结构的ORM（对象关系映射）。

程序中的SwSyslogService有两个主要功能：

```
public static SwSyslog encapsulateSwSyslog(String syslogInfo) {
    SwSyslog swsyslog = new SwSyslog();
    swsyslog.setDevIp(SwSyslogExtractorUtil.extractDevIp(syslogInfo));
    swsyslog.setServerTime(SwSyslogExtractorUtil.extractServerTime(syslogInfo));
    swsyslog.setSyslogNum(SwSyslogExtractorUtil.extractSyslogNum(syslogInfo));
    swsyslog.setDevTime(SwSyslogExtractorUtil.extractDevTime(syslogInfo));
    swsyslog.setSyslogType(SwSyslogExtractorUtil.extractSyslogType(syslogInfo));
    swsyslog.setInfoType(SwSyslogExtractorUtil.extractInfoType(syslogInfo));
    swsyslog.setDevInterface(SwSyslogExtractorUtil.extractDevInterface(syslogInfo));
    swsyslog.setInterfaceState(SwSyslogExtractorUtil.extractInterfaceState(syslogInfo));
    return swsyslog;
}

public static void saveSwSyslog(SwSyslog swSyslog) {
    LOGGER.debug("开始保存或更新SwSyslog", swSyslog);
    // 根据ip查询所有Syslog
    List<swsyslog> list = swSyslogMapper.queryAllByIp(swSyslog.getDevIp());
    //  如果list非空，即查到对应IP的SwSyslog
    if (list != null && !list.isEmpty()) {
        for (SwSyslog sys : list) {
            // 若IP的接口相同，则更新信息
            if (sys.getDevInterface().equals(swSyslog.getDevInterface())) {
                LOGGER.debug("有相同IP相同接口的记录，开始更新SwSyslog");
                sys.setServerTime(swSyslog.getServerTime());
                sys.setSyslogNum(swSyslog.getSyslogNum());
                sys.setDevTime(swSyslog.getDevTime());
                sys.setSyslogType(swSyslog.getSyslogType());
                sys.setInfoType(swSyslog.getInfoType());
                sys.setInterfaceState(swSyslog.getInterfaceState());
                sys.setUpdated(new Date());
                swSyslogMapper.update(sys);
            } else {
                // 若接口不同，则直接保存
                LOGGER.debug("相同IP无对应接口，保存SwSyslog");
                swSyslog.setCreated(new Date());
                swSyslog.setUpdated(swSyslog.getCreated());
                swSyslogMapper.insert(swSyslog);
            }
        }
    } else {
        // 没有对应的IP记录，直接保存信息
```

```
        LOGGER.debug("没有相同IP记录，直接保存SwSyslog");
        swSyslog.setCreated(new Date());
        swSyslog.setUpdated(swSyslog.getCreated());
        swSyslogMapper.insert(swSyslog);
    }
}
```

encapsulateSwSyslog() 将Spark处理后的每一行Syslog通过Scala的正则表达式解析为不同的字段，然后封装并返回Syslog对象；遍历RDD分区生成的每一个Syslog对象中都有ip以及接口信息，saveSwSyslog() 会据此判断该插入还是更新Syslog信息至数据库。另外，封装好的Syslog对象通过ORM工具MyBatis与MySQL进行互操作。

获取到的每一行Syslog信息如之前所述：

19.1.1.12 2016-10-13T10:04:54.520Z <147>12164: Oct 9 18:04:10.735: %LINK-3-UPDOWN: Interface GigabitEthernet0/16, changed state to down

这段信息需解析为设备ip、服务器时间、信息序号、设备时间、Syslog类型、属性、设备接口、接口状态等字段。Scala正则解析逻辑如下：

```
/**
  * 抽取服务器时间
  * 样例：2016-10-09T10:04:54.517Z
  * @param line
  * @return
  */
def extractServerTime(line: String): String = {
    val regex1 = "20₩₩d{2}-₩₩d{2}-₩₩d{2}".r
    val regex2 = "₩₩d{2}:₩₩d{2}:₩₩d{2}.?(₩₩d{3})?".r
    val matchedDate = regex1.findFirstIn(line)
    val matchedTime = regex2.findFirstIn(line)
    val result1 = matchedDate match {
        case Some(date) => date
        case None => " "
    }
    val result2 = matchedTime match {
        case Some(time) => time
        case None => " "
    }
    result1 + " " + result2
```

```
}

/**
  * 抽取设备时间
  * 样例：Sep 29 09:33:06
  *      Oct  9 18:04:09.733
  * @param line
  * @return
  */
def extractDevTime(line: String): String = {
    val regex = "[a-zA-Z]{3}WWs+WWd+WWsWWd{2}:WWd{2}:WWd{2}((.WWd{3})|())".r
    val matchedDevTime = regex.findFirstIn(line)
    val result = matchedDevTime match {
        case Some(devTime) => devTime
        case None => " "
    }
    result
}
```

通过正则过滤、Syslog封装以及MyBatis持久层映射，Syslog接口状态信息最终解析如下：

设备IP	19.1.1.12
服务器时间	2016-10-13 10:04:54.520
信息序号	12164
设备时间	Oct 9 18:04:10.735
Syslog类型	Link-3-UPDOWN
设备接口	Interface GigabitEthernet0/16
接口状态	down

最后，诸如APM、网络监控或者告警等业务应用便可以基于MySQL做可视化展示。

总结

本文首先对公安运维管理现状做了简要介绍，然后介绍公安实时运维平台的整体架构，再以交换机Syslog信息为例，详细介绍如何使用Flume+Logstash+Kafka+Spark Streaming进行实时日志处理分析，对处理过程中大量的技术细节进行了描述并通过代码详细地介绍整体处理步骤。本文中的示例实时地将数据写入MySQL存在一定的性能瓶颈，后期会对包含本例的相关代码重构，数据将会实时写入HBase来提高性能。

本文转载自：基于Spark的公安大数据实时运维技术实践
作者：秦海龙，杭州以数科科技有限公司大数据工程师。Java及Scala语言，Hadoop生态、Spark大数据处理技术爱好者。