

创业公司做数据分析(四)ELK 日志系统

2017-02-14Mr-Bruce



文 | Mr-Bruce

作为系列文章的第四篇，本文将重点探讨数据采集层中的 ELK 日志系统。

日志，指的是后台服务中产生的 log 信息，通常会输入到不同的文件中，比如 Django 服务下，一般会有 nginx 日志和 uWSGI 日志。这些日志分散地存储在不同的机器上，取决于服务的部署情况了。

如果我们依次登录每台机器去查阅日志，显然非常繁琐，效率也很低，而且也没法进行统计和检索。因此，我们需要对日志进行集中化管理，将所有机器上的日志信息收集、汇总到一起。完整的日志数据具有非常重要的作用：

1. **信息查找**。通过检索日志信息，定位相应的 bug，找出解决方案。
2. **服务诊断**。通过对日志信息进行统计、分析，了解服务器的负荷和服务运行状态，找出耗时请求进行优化等等。
3. **数据分析**。如果是格式化的 log，可以做进一步的数据分析，统计、聚合出有意义的信息，比如根据请求中的商品 id，找出 TOP10 用户感兴趣商品。

ELK 是一套开源的集中式日志数据管理的解决方案，由 Elasticsearch、Logstash 和 Kibana 三个系统组成。

最初我们建设 ELK 日志系统的目的是做数据分析，记得第一个需求是期望利用 nginx 的日志，从 API 请求的参数中挖掘出用户的位置分布信息。

后来该系统在追踪恶意刷量、优化耗时服务等方面都发挥了重要作用，而且随着对 Elasticsearch 的认知加深，我们将其应用到了其他方面的数据存储和分析中。

本文的重点是结合自身实践来介绍如何使用 ELK 系统、使用中的问题以及如何解决，文中涉及的 ELK 版本是：Elasticsearch 2.3、Logstash 2.3、Kibana 4。

ELK 整体方案

ELK 中的三个系统分别扮演不同的角色，组成了一个整体的解决方案。Logstash 是一个 ETL 工具，负责从每台机器抓取日志数据，对数据进行格式转换和处理后，输出到 Elasticsearch 中存储。

Elasticsearch 是一个分布式搜索引擎和分析引擎，用于数据存储，可提供实时的数据查询。Kibana 是一个数据可视化服务，根据用户的操作从 Elasticsearch 中查询数据，形成相应的分析结果，以图表的形式展现给用户。

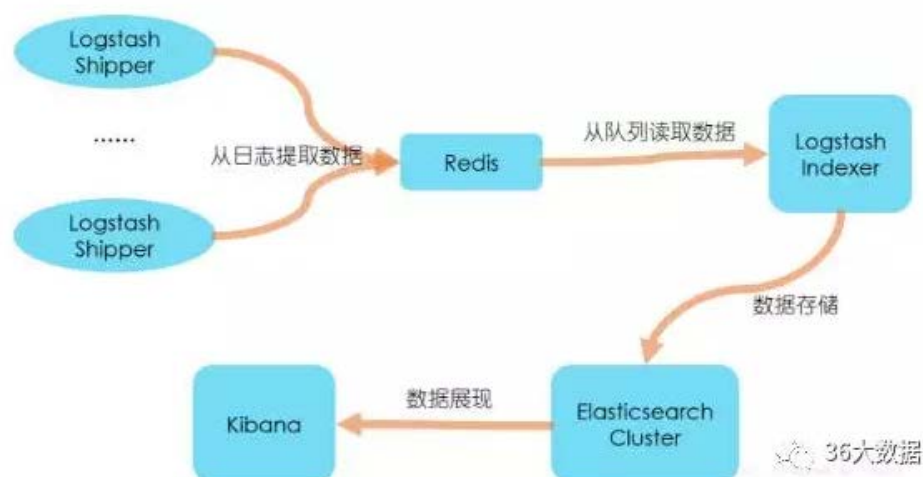
ELK 的安装很简单，可以按照“下载->修改配置文件->启动”方法分别部署三个系统，也可以使用 docker 来快速部署。具体的安装方法这里不详细介绍，我们来看一个常见的部署方案，如下图所示，部署思路是：

第一，在每台生成日志文件的机器上，部署 **Logstash**，作为 Shipper 的角色，负责从日志文件中提取数据，但是不做任何处理，直接将数据输出到 Redis 队列(list)中；

第二，需要一台机器部署 **Logstash**，作为 Indexer 的角色，负责从 Redis 中取出数据，对数据进行格式化和相关处理后，输出到 Elasticsearch 中存储；

第三，部署 **Elasticsearch** 集群，当然取决于你的数据量了，数据量小的话可以使用单台服务，如果做集群的话，最好是有 3 个以上节点，同时还需要部署相关的监控插件；

第四，部署 **Kibana** 服务，提供 Web 服务。



在前期部署阶段，主要工作是 **Logstash** 节点和 **Elasticsearch** 集群的部署，而在后期使用阶段，主要工作就是 **Elasticsearch** 集群的监控和使用 **Kibana** 来检索、分析日志数据了，当然也可以直接编写程序来消费 **Elasticsearch** 中的数据。

在上面的部署方案中，我们将 **Logstash** 分为 **Shipper** 和 **Indexer** 两种角色来完成不同的工作，中间通过 **Redis** 做数据管道，为什么要这样做？为什么不是直接在每台机器上使用 **Logstash** 提取数据、处理、存入 **Elasticsearch**？

首先，采用这样的架构部署，有三点优势：

第一，降低对日志所在机器的影响，这些机器上一般都部署着反向代理或应用服务，本身负载就很重了，所以尽可能的在这些机器上少做事；

第二，如果有很多台机器需要做日志收集，那么让每台机器都向 **Elasticsearch** 持续写入数据，必然会对 **Elasticsearch** 造成压力，因此需要对数据进行缓冲，同时，这样的缓冲也可以一定程度的保护数据不丢失；

第三，将日志数据的格式化与处理放到 **Indexer 中统一做**，可以在一处修改代码、部署，避免需要到多台机器上去修改配置。

其次，我们需要做的是将数据放入一个消息队列中进行缓冲，所以 **Redis** 只是其中一个选择，也可以是 **RabbitMQ**、**Kafka** 等等，在实际生产中，**Redis** 与 **Kafka** 用的比较多。

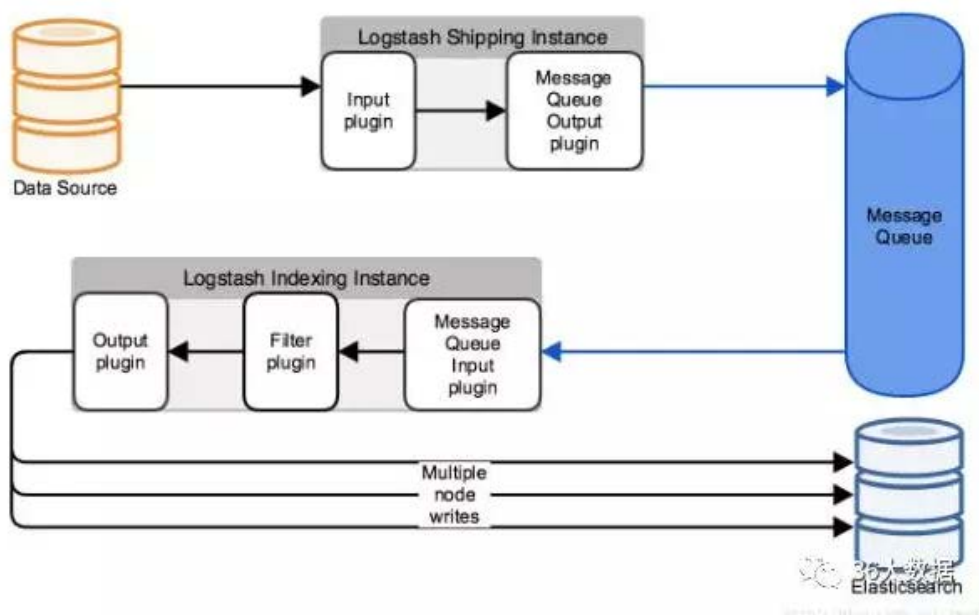
由于 **Redis** 集群一般都是通过 **key** 来做分片，无法对 **list** 类型做集群，在数据量大的时候必然不合适了，而 **Kafka** 天生就是分布式的消息队列系统。

Logstash

在官方文档中，**Deploying and Scaling Logstash** 一文详细介绍了各种 **Logstash** 的部署架构，下图是与我们上述方案相吻合的架构。

Logstash 由 **input**、**filter** 和 **output** 三部分组成，**input** 负责从数据源提取数据，**filter** 负责解析、处理数据，**output** 负责输出数据，每部分都有提供丰富的插件。

Logstash 的设计思路也非常值得借鉴，以插件的形式来组织功能，通过配置文件来描述需要插件做什么。我们以 **nginx** 日志为例，来看看如何使用一些常用插件。



1. 配置 nginx 日志格式

首先需要将 nginx 日志格式规范化，便于做解析处理。在 nginx.conf 文件中设置：

```
log_format main '$remote_addr' '$time_iso8601' '$request' '$status' '$body_bytes_sent'
'$http_user_agent' '$http_referer' '$http_x_forwarded_for' '$request_time'
'$upstream_response_time' '$http_cookie' '$http_authorization' '$http_token';

access_log /var/log/nginx/example.access.log main;
```

2. nginx 日志->>Logstash->>消息队列

这部分是 Logstash Shipper 的工作，涉及 input 和 output 两种插件。input 部分，由于需要提取的是日志文件，一般使用 file 插件，该插件常用的几个参数是：


1. path，指定日志文件路径。
2. type，指定一个名称，设置 type 后，可以在后面的 filter 和 output 中对不同的 type 做不同的处理，适用于需要消费多个日志文件的场景。
3. start_position，指定起始读取位置，“beginning”表示从文件头开始，“end”表示从文件尾开始(类似 tail -f)。
4. sincedb_path，与 Logstash 的一个坑有关。通常 Logstash 会记录每个文件已经被读取到的位置，保存在 sincedb 中，如果 Logstash 重启，那么对于同一个文件，会继续从上次记录的位置开始读取。如果想重新从头读取文件，需要删除 sincedb 文件，sincedb_path 则是指定了该文件的路径。为了方便，我们可以根据需要将其设置为“/dev/null”，即不保存位置信息。

```

input {
  file {
    type => "example_nginx_access"
    path => ["/var/log/nginx/example.access.log"]

    start_position => "beginning"
    sincedb_path => "/dev/null"
  }
}

```


 36大数据

output 部分，将数据输出到消息队列，以 redis 为例，需要指定 redis server 和 list key 名称。另外，在测试阶段，可以使用 stdout 来查看输出信息。

```

# 输出到redis
output {
  if [type] == "example_nginx_access" {
    redis {
      host => "127.0.0.1"
      port => "6379"
      data_type => "list"
      key => "logstash:example_nginx_access"
    }
    # stdout {codec => rubydebug}
  }
}

```

 36大数据

3. 消息队列->>Logstash->>Elasticsearch


这部分是 Logstash Indexer 的工作，涉及 input、filter 和 output 三种插件。在 input 部分，我们通过 redis 插件将数据从消息队列中取出来。在 output 部分，我们通过 elasticsearch 插件将数据写入 Elasticsearch。

```

# 从redis输入数据
input {
  redis {
    host => "127.0.0.1"
    port => "6379"
    data_type => "list"
    key => "logstash:example_nginx_access"
  }
}

output {
  elasticsearch {
    index => "logstash-example-nginx-%{+YYYY.MM}"
    hosts => ["127.0.0.1:9200"]
  }
}

```

 36大数据

这里，我们重点关注 **filter** 部分，下面列举几个常用的插件，实际使用中根据自身需求从官方文档中查找适合自己业务的插件并使用即可，当然也可以编写自己的插件。

grok，是 **Logstash** 最重要的一个插件，用于将非结构化的文本数据转化为结构化的数据。

grok 内部使用正则语法对文本数据进行匹配，为了降低使用复杂度，其提供了一组 **pattern**，我们可以直接调用 **pattern** 而不需要自己写正则表达式，参考源码 **grok-patterns**。

grok 解析文本的语法格式是 **%{SYNTAX:SEMANTIC}**，**SYNTAX** 是 **pattern** 名称，**SEMANTIC** 是需要生成的字段名称，使用工具 **Grok Debugger** 可以对解析语法进行调试。

例如，在下面的配置中，我们先使用 **grok** 对输入的原始 **nginx** 日志信息(默认以 **message** 作为字段名)进行解析，并添加新的字段 **request_path_with_verb**(该字段的值是 **verb** 和 **request_path** 的组合)，然后对 **request_path** 字段做进一步解析。

kv，用于将某个字段的值进行分解，类似于编程语言中的字符串 **Split**。在下面的配置中，我们将 **request_args** 字段值按照“&”进行分解，分解后的字段名称以“**request_args_**”作为前缀，并且丢弃重复的字段。

geoip，用于根据 **IP** 信息生成地理位置信息，默认使用自带的一份 **GeoLiteCity database**，也可以自己更换为最新的数据库，但是需要数据格式需要遵循 **Maxmind** 的格式(参考 **GeoLite**)，似乎目前只能支持 **legacy database**，数据类型必须是 **.dat**。

下载 **GeoLiteCity.dat.gz** 后解压， 并将文件路径配置到 **source** 中即可。

translate，用于检测某字段的值是否符合条件，如果符合条件则将其翻译成新的值，写入一个新的字段，匹配 **pattern** 可以通过 **YAML** 文件来配置。例如，在下面的配置中，我们对 **request_api** 字段翻译成更加易懂的文字描述。


```

filter {
  grok {
    match => {"message" => "%{IPORHOST:client_ip}
    \"%{TIMESTAMP_ISO8601:timestamp}\" \"%{WORD:verb}%{NOTSPACE:request_path}
    HTTP/%{NUMBER:httpversion}\" \"%{NUMBER:response_status:int} %{NUMBER:response
    _body_bytes:int}\" \"%{DATA:user_agent}\" \"%{DATA:http_referer}\" \"%{NOTSPACE:
    http_x_forwarder_for}\" \"%{NUMBER:request_time:float}\" \"%{DATA:upstream_re
    sponse_time}\" \"%{DATA:http_cookie}\" \"%{DATA:http_authentication}\"
    \"%{DATA:http_token}\""}
    add_field => {"request_path_with_verb" => "%{verb} %{request_path}"}
  }

  grok {
    match => {"request_path" => "%{URIPATH:request_api} (?:\?%{NOTSPACE:request_args} |)"}
    add_field => {"request_annotation" => "%{request_api}"}
  }

  kv {
    prefix => "request_args_"
    field_split => "&"
    source => "request_args"
    allow_duplicate_values => false
  }

  geoip {
    source => "client_ip"
    database => "/home/elktest/geoip_data/GeoLiteCity.dat"
  }

  translate {
    field => request_path
    destination => request_annotation
    regex => true
    exact => true
    dictionary_path => "/home/elktest/api_annotation.yaml"
    override => true
  }
}

```

Elasticsearch

Elasticsearch 承载了数据存储和查询的功能，其基础概念和使用方法可以参考另一篇博文 [Elasticsearch 使用总结](#)，这里主要介绍些实际生产中的问题和方法：

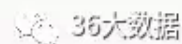
关于集群配置，重点关注三个参数：

第一，`discovery.zen.ping.unicast.hosts`，Elasticsearch 默认使用 Zen Discovery 来做节点发现机制，推荐使用 `unicast` 来做通信方式，在该配置项中列举出 Master 节点。

第二，`discovery.zen.minimum_master_nodes`，该参数表示集群中可工作的具有 Master 节点资格的最小数量，默认值是 1。为了提高集群的可用性，避免脑裂现象(所谓脑裂，就是同一个集群中的不同节点，对集群的状态有不一致的理解。)，官方推荐设置为 $(N/2)+1$ ，其中 N 是具有 Master 资格的节点的数量。

第三，`discovery.zen.ping_timeout`，表示节点在发现过程中的等待时间，默认值是 3 秒，可以根据自身网络环境进行调整，一定程度上提供可用性。

```
discovery.zen.ping.unicast.hosts: ["master1", "master2", "master3"]
discovery.zen.minimum_master_nodes: 2
discovery.zen.ping_timeout: 10
```



关于集群节点：

第一，节点类型包括：候选 Master 节点、数据节点和 Client 节点。通过设置两个配置项 `node.master` 和 `node.data` 为 `true` 或 `false`，来决定将一个节点分配为什么类型的节点。

第二，尽量将候选 Master 节点和 Data 节点分离开，通常 Data 节点负载较重，需要考虑单独部署。

关于内存，Elasticsearch 默认设置的内存是 1GB，对于任何一个业务部署来说，这个都太小了。通过指定 `ES_HEAP_SIZE` 环境变量，可以修改其堆内存大小，服务进程在启动时候会读取这个变量，并相应的设置堆的大小。

建议设置系统内存的一半给 Elasticsearch，但是不要超过 32GB。参考官方文档。

关于硬盘空间，Elasticsearch 默认将数据存储在 `/var/lib/elasticsearch` 路径下，随着数据的增长，一定会出现硬盘空间不够用的情形，此时就需要给机器挂载新的硬盘，并将 Elasticsearch 的路径配置到新硬盘的路径下。

通过“`path.data`”配置项来进行设置，比如“`path.data:/data1,/var/lib/elasticsearch,/data`”。需要注意的是，同一分片下的数据只能写入到一个路径下，因此还是需要合理的规划和监控硬盘的使用。

关于 Index 的划分和分片的个数，这个需要根据数据量来做权衡了，Index 可以按时间划分，比如每月一个或者每天一个，在 Logstash 输出时进行配置，shard 的数量也需要做好控制。

关于监控，笔者使用过 head 和 marvel 两个监控插件，head 免费，功能相对有限，marvel 现在需要收费了。另外，不要在数据节点开启监控插件。

Kibana

Kibana 提供的是数据查询和显示的 Web 服务，有丰富的图表样板，能满足大部分的数据可视化需求，这也是很多人选择 ELK 的主要原因之一。UI 的操作没有什么特别需要介绍的，经常使用就会熟练，这里主要介绍经常遇到的三个问题。

1. 查询语法

在 Kibana 的 Discover 页面中，可以输入一个查询条件来查询所需的数据。查询条件的写法使用的是 Elasticsearch 的 Query String 语法，而不是 Query DSL，参考官方文档 [query-string-syntax](#)，这里列举其中部分常用的：

1. 单字段的全文检索，比如搜索 args 字段中包含 first 的文档，写作 args:first;
2. 单字段的精确检索，比如搜索 args 字段值为 first 的文档，写作 args: "first";
3. 多个检索条件的组合，使用 NOT, AND 和 OR 来组合，注意必须是大写，比如 args:("first" OR "second") AND NOT agent: "third";
4. 字段是否存在，_exists_:agent 表示要求 agent 字段存在，_missing_:agent 表示要求 agent 字段不存在;
5. 通配符：用 ? 表示单字母，* 表示任意个字母。

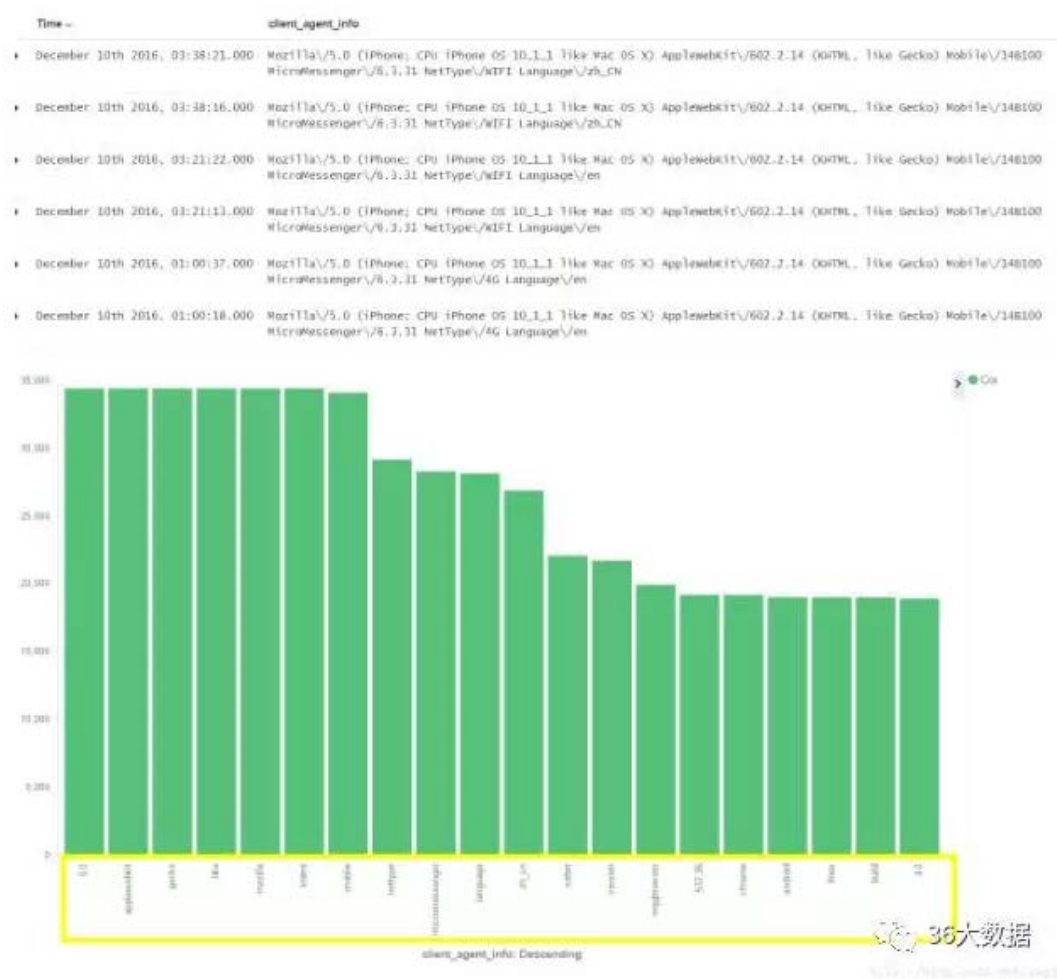
2. 错误“Discover: Request Timeout after 30000ms”

这个错误经常发生在要查询的数据量比较大的情况下，此时 Elasticsearch 需要较长时间才能返回，导致 Kibana 发生 Timeout 报错。

解决这个问题的方法，就是在 Kibana 的配置文件中修改 elasticsearch.requestTimeout 一项的值，然后重启 Kibana 服务即可，注意单位是 ms。

3. 疑惑“字符串被分解了”

经常在 QQ 群里看到一些人在问这样一个问题：为什么查询结果的字段值是正确的，可是做图表时却发现字段值被分解了，不是想要的结果？如下图所示的 client_agent_info 字段。



得到这样一个不正确结果的原因是使用了 **Analyzed** 字段来做图表分析，默认情况下 **Elasticsearch** 会对字符串数据进行分析，建立倒排索引，所以如果对这么一个字段进行 **terms** 聚合，必然会得到上面所示的错误结果了。

那么应该怎么做才对?默认情况下，**Elasticsearch** 还会创建一个相对应的没有被 **Analyzed** 的字段，即带“.raw”后缀的字段，在这样的字段上做聚合分析即可。

又会有很多人问这样的问题：为什么我的 **Elasticsearch** 没有自动创建带“.raw”后缀的字段?然而在 **Logstash** 中输出数据时，设置 **index** 名称前缀为“logstash-”就有了这个字段。

这个问题的根源是 **Elasticsearch** 的 **dynamic template** 在捣鬼，**dynamic template** 用于指导 **Elasticsearch** 如何为插入的数据自动建立 **Schema** 映射关系。

默认情况下，**Logstash** 会在 **Elasticsearch** 中建立一个名为“logstash”的模板，所有前缀为“logstash-”的 **index** 都会参照这个模板来建立映射关系，在该模板中声明了要为每个字符串数据建立一个额外的带“.raw”后缀的字段。可以向 **Elasticsearch** 来查询你的模板，使用 **API**: **GET** http://localhost:9200/_template。

以上便是对 **ELK** 日志系统的总结介绍，还有一个重要的功能没有提到，就是如何将日志数据与自身产品业务的数据融合起来。

举个例子，在 **nginx** 日志中，通常会包含 **API** 请求访问时携带的用户 **Token** 信息，由于 **Token** 是有时效性的，我们需要及时将这些 **Token** 转换成真实的用户信息存储下来。

这样的需求通常有两种实现方式，一种是自己写一个 **Logstash filter**，然后在 **Logstash** 处理数据时调用；另一种是将 **Logstash Indexer** 产生的数据再次输出到消息队列中，由我们自己的脚本程序从消息队列中取出数据，做相应的业务处理后，输出到 **Elasticsearch** 中。目前，团队对 **ruby** 技术栈不是很熟悉，所以我们采用了第二种方案来实施。

当前，我们的数据增长相对缓慢，遇到的问题也有限，随着数据量的增加，未来一定会遇到更多的挑战，也可以进一步探索 **ELK**。

End.