**ELK&Beats**

## 一、Elasticsearch

**1 Elasticsearch面向文档(document oriented)**

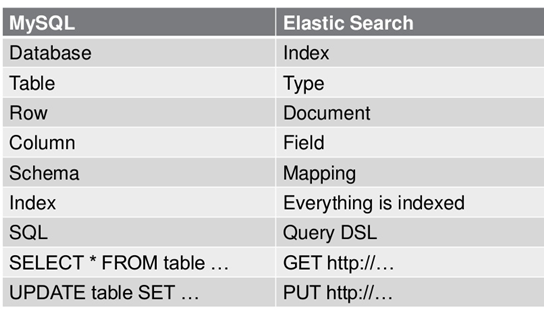
这意味着它可以存储整个对象或文档(document)。然而它不仅仅是存储，还会索引(index)每个文档的内容使之可以被搜索。在Elasticsearch中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。这种理解数据的方式与以往完全不同，这也是Elasticsearch能够执行复杂的全文搜索的原因之一。我们来看一个实际的例子，假设有如下的数据：



这里每一行是一个document。每个document都有一个docid。那么给这些document建立的倒排索引就是：

**可以看到，倒排索引是per field的，一个字段由一个自己的倒排索引。18,20这些叫做 term，而[1,3]就是posting list。Posting list就是一个int的数组，存储了所有符合某个term的文档id。**

**（1）mysql和ES**



**（2） ELK=elasticsearch+Logstash+kibana**  
elasticsearch：后台分布式存储以及全文检索   
logstash: 日志加工、“搬运工”   
kibana：数据可视化展示。   
ELK架构为数据分布式存储、可视化查询和日志解析创建了一个功能强大的管理链。 三者相互配合，取长补短，共同完成分布式大数据处理工作。

**2 思考：大规模数据如何检索？**

如：当系统数据量上了10亿、100亿条的时候，我们在做系统架构的时候通常会从以下角度去考虑问题：

1）用什么数据库好？(mysql、sybase、oracle、mongodb、hbase…)

2）如何解决单点故障；(lvs、F5、A10、Zookeep、MQ)

3）如何保证数据安全性；(热备、冷备、异地多活)

4）如何解决检索难题；(数据库代理中间件：mysql-proxy、Cobar、MaxScale等;)

5）如何解决统计分析问题；(离线、近实时)

**3 传统数据库的应对解决方案**

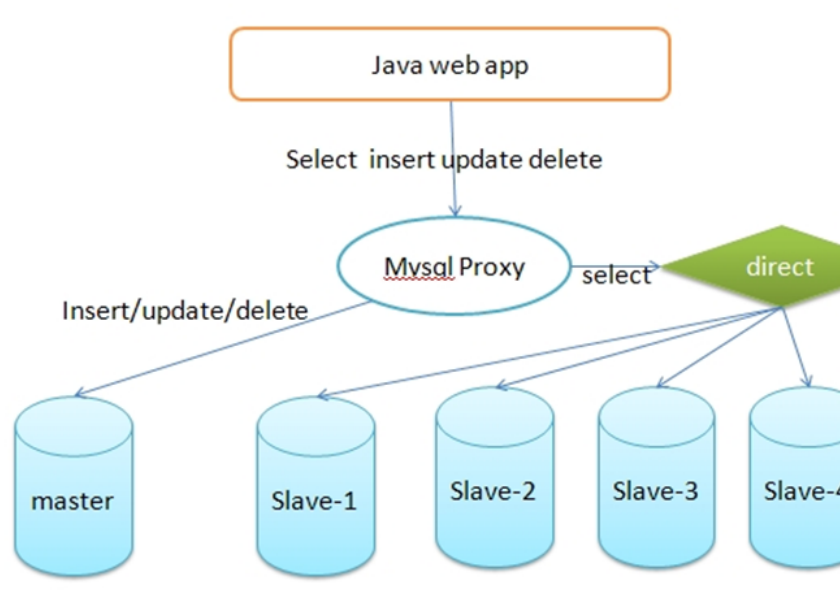
对于关系型数据，我们通常采用以下或类似架构去解决查询瓶颈和写入瓶颈：

解决要点：

1）通过主从备份解决数据安全性问题；

2）通过数据库代理中间件心跳监测，解决单点故障问题；

3）通过代理中间件将查询语句分发到各个slave节点进行查询，并汇总结果



**4 非关系型数据库的解决方案~~~~~~**

对于Nosql数据库，以mongodb为例，其它原理类似：

解决要点：

1）通过副本备份保证数据安全性；

2）通过节点竞选机制解决单点问题；

3）先从配置库检索分片信息，然后将请求分发到各个节点，最后由路由节点合并汇总结果

**另辟蹊径——完全把数据放入内存怎么样？**

我们知道，完全把数据放在内存中是不可靠的，实际上也不太现实，当我们的数据达到PB级别时，按照每个节点96G内存计算，在内存完全装满的数据情况下，我们需要的机器是：**1PB=1024T=1048576G**

**节点数=1048576/96=10922个**

实际上，考虑到数据备份，节点数往往在2.5万台左右。成本巨大决定了其不现实！

从前面讨论我们了解到，把数据放在内存也好，不放在内存也好，都不能完完全全解决问题。

全部放在内存速度问题是解决了，但成本问题上来了。

**5 为解决以上问题，从源头着手分析，通常会从以下方式来寻找方法：**

1、存储数据时按有序存储；

2、将数据和索引分离；

3、压缩数据；

这就引出了Elasticsearch。

**6 Elasticsearch基本概念**

ES=elaticsearch简写， Elasticsearch是一个开源的高扩展的分布式全文检索引擎，它可以近乎实时的存储、检索数据；本身扩展性很好，可以扩展到上百台服务器，处理PB级别的数据。 Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。

1）Lucene只是一个库。想要使用它，你必须使用Java来作为开发语言并将其直接集成到你的应用中，更糟糕的是，Lucene非常复杂，你需要深入了解检索的相关知识来理解它是如何工作的。

2）Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。

**可以把es看做是面向文档的数据库，它与关系型数据库的名词对照关系如下：**

**Relational DB => Databases => Tables => Rows => Columns**

**Elasticsearch => Index=> doc Types => Documents => Fields**

**（1）Index 索引**

索引（index）是Elasticsearch对逻辑数据的逻辑存储，所以它可以分为更小的部分。你可以把索引看成关系型数据库的表。Elasticsearch可以把索引存放在一台机器或者分散在多台服务器上，每个索引有一或多个分片（shard），每个分片可以有多个副本（replica）。

**（2）doc Types 文档类型**

在Elasticsearch中，一个索引对象可以存储很多不同用途的对象。例如，一个博客应用程序可以保存文章和评论。文档类型让我们轻易地区分单个索引中的不同对象。每个文档可以有不同的结构，但在实际部署中，将文件按类型区分对数据操作有很大帮助。

**（3） Document 文档**

存储在Elasticsearch中的主要实体叫文档（document）。用关系型数据库来类比的话，一个文档相当于数据库表中的一行记录。从客户端的角度看，文档是一个JSON对象。每个文档存储在一个索引中并有一个Elasticsearch自动生成的唯一标识符和文档类型。

**（4）field**

文档由多个field组成，从客户端的角度看，就是json对象中的多个kv节点。

**（5）Cluster：集群。**

ES可以作为一个独立的单个搜索服务器。不过，为了处理大型数据集，实现容错和高可用性，ES可以运行在许多互相合作的服务器上。这些服务器的集合称为集群。

**（6）Node：节点。**

形成集群的每个服务器称为节点。

**（7）Shard：分片。**

当有大量的文档时，由于内存的限制、磁盘处理能力不足、无法足够快的响应客户端的请求等，一个节点可能不够。这种情况下，数据可以分为较小的分片。每个分片放到不同的服务器上。 当你查询的索引分布在多个分片上时，ES会把查询发送给每个相关的分片，并将结果组合在一起，而应用程序并不知道分片的存在。即：这个过程对用户来说是透明的。

**（8）Replia：副本。**

为提高查询吞吐量或实现高可用性，可以使用分片副本。 副本是一个分片的精确复制，每个分片可以有零个或多个副本。ES中可以有许多相同的分片，其中之一被选择更改索引操作，这种特殊的分片称为主分片。 当主分片丢失时，如：该分片所在的数据不可用时，集群将副本提升为新的主分片。

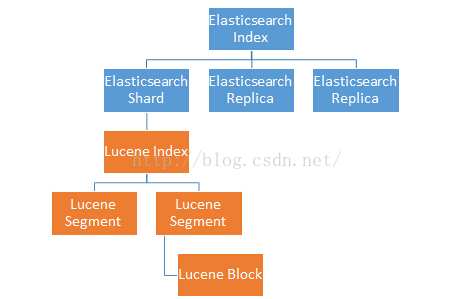
**（9）全文检索。**

全文检索就是对一篇文章进行索引，可以根据关键字搜索，类似于mysql里的like语句。 全文索引就是把内容根据词的意义进行分词，然后分别创建索引，例如”你们的激情是因为什么事情来的” 可能会被分词成：“你们“，”激情“，“什么事情“，”来“ 等token，这样当你搜索“你们” 或者 “激情” 都会把这句搜出来。

**7 基本架构**

**（1）shard分片**

实际上，index仅仅只是一个命名空间来指向一个或多个实际的物理分片(shard)。具体的物理分布粒度关系如下：



一个Elasticsearch Index相当于一个MySQL里的表，不同Index的数据是物理上隔离开来的。Elasticsearch的Index会分成多个Shard存储，一部分Shard是Replica备份。一个Shard是一份本地的存储（一个本地磁盘上的目录），也就是一个Lucene的Index。不同的Shard可能会被分配到不同的主机节点上。一个Lucene Index会存储很多的doc，为了好管理，Lucene把Lucene Index再拆成了Segment存储（子目录）。Segment内的doc数量上限是2的31次方，这样doc id就只需要一个int就可以存储。Segment对应了一些列文件存储索引（倒排表等）和主存储（DocValues等），这些文件内部又分为小的Block进行压缩。

一个shard实际上是一个Lucene实例，在它的能力范围内拥有完整的搜索功能(在处理它自己拥有的数据时有所有的功能)。我们所有文档的索引indexed(动词)和存储工作都是在shard上，但这是透明的，我们不需要直接和shard通信，而是和我们创建的index(名词)通信。

shards是ES将数据分布式在你的集群的关键。想象下shards是数据的容器，文档存储在shards里，而shards被分配在集群的每一个节点Node里。当你的集群规模增长和降低时，ES会自动的在Nodes间迁移shards以保持集群的负载均衡。

**（2）备份**

shard可分为primary shard和replica shard。 在一个index里的每一个文档都属于一个单独的primary shard，所以primary shard的数量决定了你最大能存储的数据量(对应于一个index)。

注意：shard是归属与index的，而不是cluster的。

replica shard是primary shard的拷贝。replica有两个作用： 1.冗余容灾 2.提供读请求服务，例如搜索或读取文档primary shard的数量在索引创建时确定后不能修改，replica可以在任何时候修改。

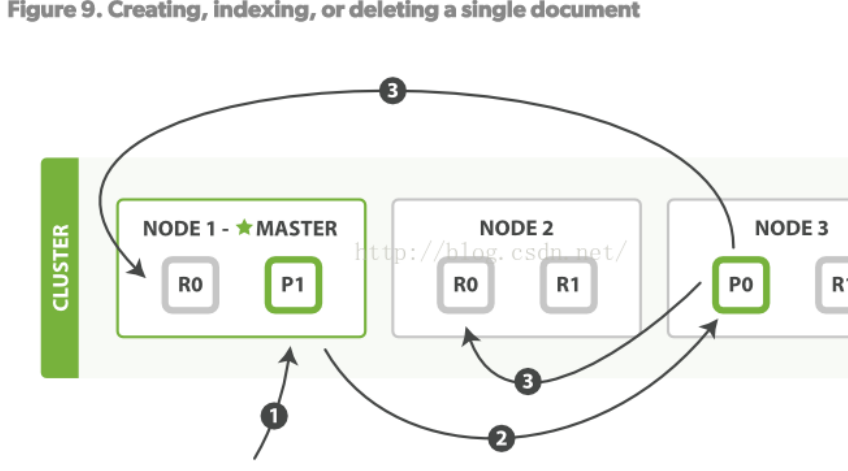
**（3）Shards文档路由**

当你对一个文档建立索引时，它仅存储在一个primary shard上。ES是怎么知道一个文档应该属于哪个shard？当你创建一个新的文档时，ES是怎么知道应该把它存储至shard1还是shard2？ 这个过程不能随机无规律的，因为以后我们还要将它取出来。它的路由算法是：

shard = hash(routing) % numberofprimary\_shards routing的值可以是文档的id，也可以是用户自己设置的一个值。hash将会根据routing算出一个数值然后%primaryshards的数量。这也是为什么primary\_shards在index创建时就不能修改的原因。我们可以向这个集群的任何一台NODE发送请求，每一个NODE都有能力处理请求。每一个NODE都知道每一个文档所在的位置所以可以直接将请求路由过去。下面的例子，我们将所有的请求都发送到NODE1。

**（4）写操作**

创建、索引、删除文档都是写操作，这些操作必须在primary shard完全成功后才能拷贝至其对应的replicas上。



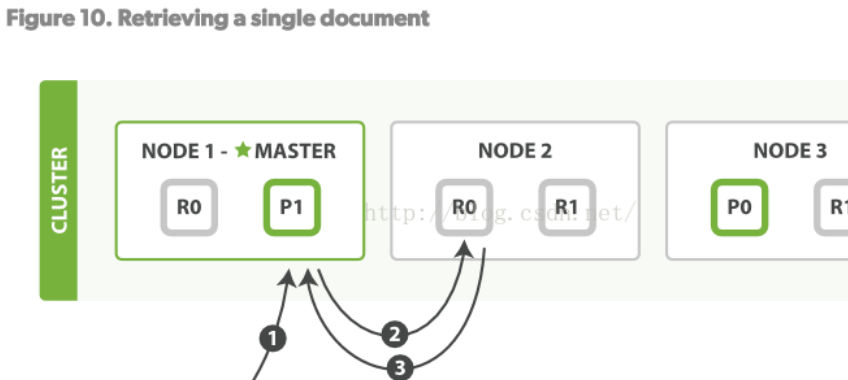
1）.客户端向Node1发送写操作的请求。

2）.Node1使用文档的\_id来决定这个文档属于shard0，然后将请求路由至NODE3，P0所在的位置。

3）.Node3在P0上执行了请求。如果请求成功，则将请求并行的路由至NODE1 NODE2的R0上。当所有的replicas报告成功后，NODE3向请求的node(NODE1)发送成功报告，NODE1再报告至Client。当客户端收到执行成功后，操作已经在Primary shard和所有的replica shards上执行成功了。当然，有一些请求参数可以修改这个逻辑。见原文。

**（5）读操作**

一个文档可以在primary shard和所有的replica shard上读取。



读操作步骤：

1）.客户端发送Get请求到NODE1。

2）.NODE1使用文档的\_id决定文档属于shard 0.shard 0的所有拷贝存在于所有3个节点上。这次，它将请求路由至NODE2。

3）.NODE2将文档返回给NODE1，NODE1将文档返回给客户端。 对于读请求，请求节点(NODE1)将在每次请求到来时都选择一个不同的replica。

**shard来达到负载均衡。使用轮询策略轮询所有的replica shards。**

**（6）更新操作**

更新操作，结合了以上的两个操作：读、写。

步骤：

1）.客户端发送更新操作请求至NODE1

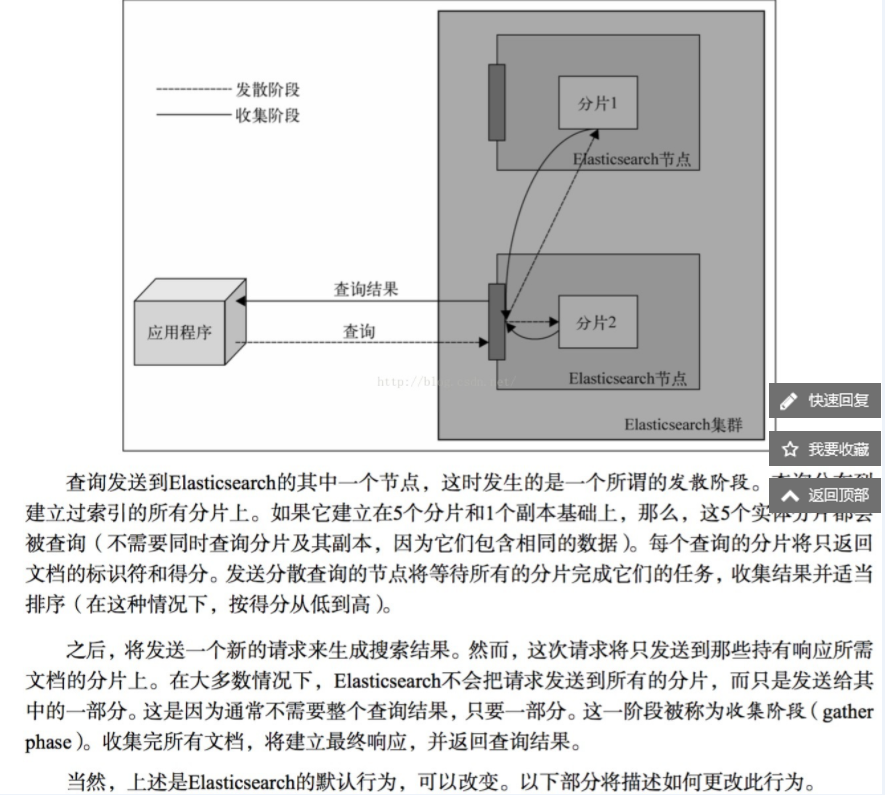
2）.NODE1将请求路由至NODE3，Primary shard所在的位置

3）.NODE3从P0读取文档，改变source字段的JSON内容，然后试图重新对修改后的数据在P0做索引。如果此时这个文档已经被其他的进程修改了，那么它将重新执行3步骤，这个过程如果超过了retryon\_conflict设置的次数，就放弃。

4）.如果NODE3成功更新了文档，它将并行的将新版本的文档同步到NODE1和NODE2的replica shards重新建立索引。一旦所有的replica

shards报告成功，NODE3向被请求的节点(NODE1)返回成功，然后NODE1向客户端返回成功。

**（7）查询操作**



**8 ES 搜索类型[query type]**

**（1）分布式搜索背景**

ES天生就是为分布式而生，但分布式有分布式的缺点。比如要搜索某个单词，但是数据却分别在5个分片（Shard)上面，这5个分片可能在5台主机上面。因为全文搜索天生就要排序（按照匹配度进行排名）,但数据却在5个分片上，如何得到最后正确的排序呢？ES是这样做的，大概分两步。

step1、ES客户端会将这个搜索词同时向5个分片发起搜索请求，这叫Scatter,

step2、这5个分片基于本Shard独立完成搜索，然后将符合条件的结果全部返回，这一步叫Gather。

客户端将返回的结果进行重新排序和排名，最后返回给用户。也就是说，ES的一次搜索，是一次scatter/gather过程（这个跟mapreduce也很类似）.

**（2）然而这其中有两个问题。**

**第一、数量问题。**比如，用户需要搜索"双黄连"，要求返回最符合条件的前10条。但在5个分片中，可能都存储着双黄连相关的数据。所以ES会向这5个分片都发出查询请求，并且要求每个分片都返回符合条件的10条记录。当ES得到返回的结果后，进行整体排序，然后取最符合条件的前10条返给用户。这种情况，ES5个shard最多会收到10\*5=50条记录，这样返回给用户的结果数量会多于用户请求的数量。

**第二、排名问题。**上面搜索，每个分片计算分值都是基于自己的分片数据进行计算的。计算分值使用的词频率和其他信息都是基于自己的分片进行的，而ES进行整体排名是基于每个分片计算后的分值进行排序的，这就可能会导致排名不准确的问题。如果我们想更精确的控制排序，应该先将计算排序和排名相关的信息（词频率等）从5个分片收集上来，进行统一计算，然后使用整体的词频率去每个分片进行查询。

**（3）这两个问题，估计ES也没有什么较好的解决方法，最终把选择的权利交给用户，方法就是在搜索的时候指定query type。**

**1）、query and fetch**

向索引的所有分片（shard）都发出查询请求，各分片返回的时候把元素文档（document）和计算后的排名信息一起返回。这种搜索方式是最快的。因为相比下面的几种搜索方式，这种查询方法只需要去shard查询一次。但是各个shard返回的结果的数量之和可能是用户要求的size的n倍。

**2）、query then fetch（默认的搜索方式）**

如果你搜索时，没有指定搜索方式，就是使用的这种搜索方式。这种搜索方式，大概分两个步骤，第一步，先向所有的shard发出请求，各分片只返回排序和排名相关的信息（注意，不包括文档document)，然后按照各分片返回的分数进行重新排序和排名，取前size个文档。然后进行第二步，去相关的shard取document。这种方式返回的document与用户要求的size是相等的。

**3）、DFS query and fetch**

这种方式比第一种方式多了一个初始化散发(initial scatter)步骤，有这一步，据说可以更精确控制搜索打分和排名。

**4）、DFS query then fetch**

比第2种方式多了一个初始化散发(initial scatter)步骤。

DSF是什么缩写？初始化散发是一个什么样的过程？从es的官方网站我们可以指定，初始化散发其实就是在进行真正的查询之前，先把各个分片的词频率和文档频率收集一下，然后进行词搜索的时候，各分片依据全局的词频率和文档频率进行搜索和排名。显然如果使用DFS\_QUERY\_THEN\_FETCH这种查询方式，效率是最低的，因为一个搜索，可能要请求3次分片。但，使用DFS方法，搜索精度应该是最高的。至于DFS是什么缩写，没有找到相关资料，这个D可能是Distributed，F可能是frequency的缩写，至于S可能是Scatter的缩写，整个单词可能是分布式词频率和文档频率散发的缩写。

**总结一下，从性能考虑QUERY\_AND\_FETCH是最快的，DFS\_QUERY\_THEN\_FETCH是最慢的。从搜索的准确度来说，DFS要比非DFS的准确度更高。**

**9 java基本操作 api【配置集群对象信息；创建客户端；查看集群信息】**

**（1）集群名称**

默认集群名为elasticsearch，如果集群名称和指定的不一致则在使用节点资源时会报错。

**（2）嗅探功能**

通过client.transport.sniff启动嗅探功能，这样只需要指定集群中的某一个节点(不一定是主节点)，然后会加载集群中的其他节点，这样只要程序不停即使此节点宕机仍然可以连接到其他节点。

**（3）查询类型SearchType.QUERY\_THEN\_FETCH**

**1）QUERY\_AND\_FETCH：**

主节点将查询请求分发到所有的分片中，各个分片按照自己的查询规则即词频文档频率进行打分排序，然后将结果返回给主节点，主节点对所有数据进行汇总排序然后再返回给客户端，此种方式只需要和es交互一次。这种查询方式存在数据量和排序问题，主节点会汇总所有分片返回的数据这样数据量会比较大，二是各个分片上的规则可能不一致。

**2）QUERY\_THEN\_FETCH：**

主节点将请求分发给所有分片，各个分片打分排序后将数据的id和分值返回给主节点，主节点收到后进行汇总排序再根据排序后的id到对应的节点读取对应的数据再返回给客户端，此种方式需要和es交互两次。**这种方式解决了数据量问题但是排序问题依然存在而且是es的默认查询方式。**

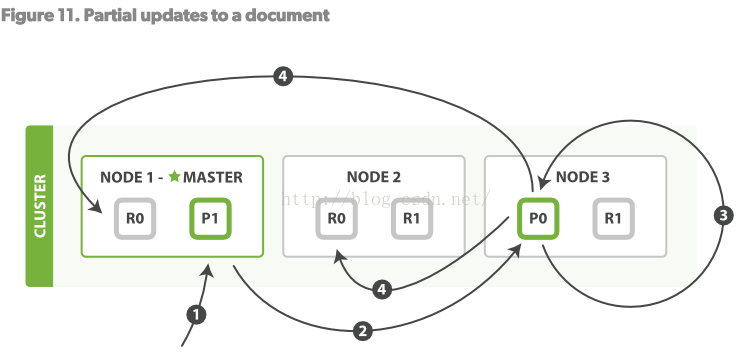
**3）DFS\_QUERY\_AND\_FETCH和DFS\_QUERY\_THEN\_FETCH：**

这两种方式和前面两种的区别在于将各个分片的规则统一起来进行打分。解决了排序问题但是DFS\_QUERY\_AND\_FETCH仍然存在数据量问题，**DFS\_QUERY\_THEN\_FETCH两种问题都解决但是效率是最差的。**

**4）特点：**一个交互两次，一个交互一次；一个统一打分规则一个不统一；一个分片返回详细数据一个分片返回id。

**（4）分页压力**

我们通过curl和java查询时都可以指定分页，但是页数越往后服务器的压力会越大。大多数搜索引擎都不会提供非常大的页数搜索，原因有两个一是用户习惯一般不会看页数大的搜索结果因为越往后越不准确，二是服务器压力。

比如分片是5分页单位是10查询第10000到10010条记录，es需要在所有分片上进行查询，每个分片会产生10010条排序后的数据然后返回给主节点，主节点接收5个分片的数据一共是50050条然后再进行汇总最后再取其中的10000到10010条数据返回给客户端，这样一来看似只请求了10条数据但实际上es要汇总5万多条数据，所以页码越大服务器的压力就越大。

**（5）超时timeout**

查询时如果数据量很大，可以指定超时时间即到达此时间后无论查询的结果是什么都会返回并且关闭连接，这样用户体验较好缺点是查询出的数据可能不完整，Java和curl都可以指定超时时间。

**（6）maven依赖**

<dependency>

<groupId>org.elasticsearch</groupId>

<artifactId>elasticsearch</artifactId>

<version>1.4.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.1.3</version>

</dependency>

**（7）java代码**

package elasticsearch;

import java.io.IOException;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.concurrent.ExecutionException;

import online.elasticsearch.bean.Student;

import org.elasticsearch.ElasticsearchException;

import org.elasticsearch.action.bulk.BulkItemResponse;

import org.elasticsearch.action.bulk.BulkRequestBuilder;

import org.elasticsearch.action.bulk.BulkResponse;

import org.elasticsearch.action.delete.DeleteRequest;

import org.elasticsearch.action.delete.DeleteResponse;

import org.elasticsearch.action.get.GetResponse;

import org.elasticsearch.action.index.IndexRequest;

import org.elasticsearch.action.index.IndexResponse;

import org.elasticsearch.action.search.SearchResponse;

import org.elasticsearch.action.search.SearchType;

import org.elasticsearch.action.update.UpdateRequest;

import org.elasticsearch.action.update.UpdateResponse;

import org.elasticsearch.client.transport.TransportClient;

import org.elasticsearch.cluster.node.DiscoveryNode;

import org.elasticsearch.common.collect.ImmutableList;

import org.elasticsearch.common.settings.ImmutableSettings;

import org.elasticsearch.common.settings.Settings;

import org.elasticsearch.common.text.Text;

import org.elasticsearch.common.transport.InetSocketTransportAddress;

import org.elasticsearch.common.transport.TransportAddress;

import org.elasticsearch.common.xcontent.XContentBuilder;

import org.elasticsearch.common.xcontent.XContentFactory;

import org.elasticsearch.index.query.FilterBuilders;

import org.elasticsearch.index.query.MatchQueryBuilder.Operator;

import org.elasticsearch.index.query.QueryBuilders;

import org.elasticsearch.search.SearchHit;

import org.elasticsearch.search.SearchHits;

import org.elasticsearch.search.aggregations.Aggregation;

import org.elasticsearch.search.aggregations.AggregationBuilders;

import org.elasticsearch.search.aggregations.Aggregations;

import org.elasticsearch.search.aggregations.bucket.terms.Terms;

import org.elasticsearch.search.aggregations.bucket.terms.Terms.Bucket;

import org.elasticsearch.search.aggregations.metrics.sum.Sum;

import org.elasticsearch.search.highlight.HighlightField;

import org.elasticsearch.search.sort.SortOrder;

import org.junit.Before;

import org.junit.Test;

import com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

public class elastaicTest {

TransportClient transportClient;

//索引库名

String index = "shb01";

//类型名称

String type = "stu";

@Before

public void before()

{

/\*\*

\* 1:通过 setting对象来指定集群配置信息

\*/

Settings setting = ImmutableSettings.settingsBuilder()

.put("cluster.name", "shb01")//指定集群名称

.put("client.transport.sniff", true)//启动嗅探功能

.build();

/\*\*

\* 2：创建客户端

\* 通过setting来创建，若不指定则默认链接的集群名为elasticsearch

\* 链接使用tcp协议即9300

\*/

transportClient = new TransportClient(setting);

TransportAddress transportAddress = new InetSocketTransportAddress("192.168.79.131", 9300);

transportClient.addTransportAddresses(transportAddress);

/\*\*

\* 3：查看集群信息

\* 注意我的集群结构是：

\* 131的elasticsearch.yml中指定为主节点不能存储数据，

\* 128的elasticsearch.yml中指定不为主节点只能存储数据。

\* 所有控制台只打印了192.168.79.128,只能获取数据节点

\*

\*/

ImmutableList<DiscoveryNode> connectedNodes = transportClient.connectedNodes();

for(DiscoveryNode node : connectedNodes)

{

System.out.println(node.getHostAddress());

}

}

/\*\*

\* 通过prepareGet方法获取指定文档信息

\*/

@Test

public void testGet() {

GetResponse getResponse = transportClient.prepareGet(index, type, "1").get();

System.out.println(getResponse.getSourceAsString());

}

/\*\*

\* prepareUpdate更新索引库中文档，如果文档不存在则会报错

\* @throws IOException

\*

\*/

@Test

public void testUpdate() throws IOException

{

XContentBuilder source = XContentFactory.jsonBuilder()

.startObject()

.field("name", "will")

.endObject();

UpdateResponse updateResponse = transportClient

.prepareUpdate(index, type, "6").setDoc(source).get();

System.out.println(updateResponse.getVersion());

}

/\*\*

\* 通过prepareIndex增加文档，参数为json字符串

\*/

@Test

public void testIndexJson()

{

String source = "{\"name\":\"will\",\"age\":18}";

IndexResponse indexResponse = transportClient

.prepareIndex(index, type, "3").setSource(source).get();

System.out.println(indexResponse.getVersion());

}

/\*\*

\* 通过prepareIndex增加文档，参数为Map<String,Object>

\*/

@Test

public void testIndexMap()

{

Map<String, Object> source = new HashMap<String, Object>(2);

source.put("name", "Alice");

source.put("age", 16);

IndexResponse indexResponse = transportClient

.prepareIndex(index, type, "4").setSource(source).get();

System.out.println(indexResponse.getVersion());

}

/\*\*

\* 通过prepareIndex增加文档，参数为javaBean

\*

\* @throws ElasticsearchException

\* @throws JsonProcessingException

\*/

@Test

public void testIndexBean() throws ElasticsearchException, JsonProcessingException

{

Student stu = new Student();

stu.setName("Fresh");

stu.setAge(22);

ObjectMapper mapper = new ObjectMapper();

IndexResponse indexResponse = transportClient

.prepareIndex(index, type, "5").setSource(mapper.writeValueAsString(stu)).get();

System.out.println(indexResponse.getVersion());

}

/\*\*

\* 通过prepareIndex增加文档，参数为XContentBuilder

\*

\* @throws IOException

\* @throws InterruptedException

\* @throws ExecutionException

\*/

@Test

public void testIndexXContentBuilder() throws IOException, InterruptedException, ExecutionException

{

XContentBuilder builder = XContentFactory.jsonBuilder()

.startObject()

.field("name", "Avivi")

.field("age", 30)

.endObject();

IndexResponse indexResponse = transportClient

.prepareIndex(index, type, "6")

.setSource(builder)

.execute().get();

//.execute().get();和get()效果一样

System.out.println(indexResponse.getVersion());

}

/\*\*

\* 通过prepareDelete删除文档

\*

\*/

@Test

public void testDelete()

{

String id = "9";

DeleteResponse deleteResponse = transportClient.prepareDelete(index,

type, id).get();

System.out.println(deleteResponse.getVersion());

//删除所有记录

transportClient.prepareDeleteByQuery(index).setTypes(type)

.setQuery(QueryBuilders.matchAllQuery()).get();

}

/\*\*

\* 删除索引库，不可逆慎用

\*/

@Test

public void testDeleteeIndex()

{

transportClient.admin().indices().prepareDelete("shb01","shb02").get();

}

/\*\*

\* 求索引库文档总数

\*/

@Test

public void testCount()

{

long count = transportClient.prepareCount(index).get().getCount();

System.out.println(count);

}

/\*\*

\* 通过prepareBulk执行批处理

\*

\* @throws IOException

\*/

@Test

public void testBulk() throws IOException

{

//1:生成bulk

BulkRequestBuilder bulk = transportClient.prepareBulk();

//2:新增

IndexRequest add = new IndexRequest(index, type, "10");

add.source(XContentFactory.jsonBuilder()

.startObject()

.field("name", "Henrry").field("age", 30)

.endObject());

//3:删除

DeleteRequest del = new DeleteRequest(index, type, "1");

//4:修改

XContentBuilder source = XContentFactory.jsonBuilder().startObject().field("name", "jack\_1").field("age", 19).endObject();

UpdateRequest update = new UpdateRequest(index, type, "2");

update.doc(source);

bulk.add(del);

bulk.add(add);

bulk.add(update);

//5:执行批处理

BulkResponse bulkResponse = bulk.get();

if(bulkResponse.hasFailures())

{

BulkItemResponse[] items = bulkResponse.getItems();

for(BulkItemResponse item : items)

{

System.out.println(item.getFailureMessage());

}

}

else

{

System.out.println("全部执行成功！");

}

}

/\*\*

\* 通过prepareSearch查询索引库

\* setQuery(QueryBuilders.matchQuery("name", "jack"))

\* setSearchType(SearchType.QUERY\_THEN\_FETCH)

\*

\*/

@Test

public void testSearch()

{

SearchResponse searchResponse = transportClient.prepareSearch(index)

.setTypes(type)

.setQuery(QueryBuilders.matchAllQuery()) //查询所有

//.setQuery(QueryBuilders.matchQuery("name", "tom").operator(Operator.AND)) //根据tom分词查询name,默认or

//.setQuery(QueryBuilders.multiMatchQuery("tom", "name", "age")) //指定查询的字段

//.setQuery(QueryBuilders.queryString("name:to\* AND age:[0 TO 19]")) //根据条件查询,支持通配符大于等于0小于等于19

//.setQuery(QueryBuilders.termQuery("name", "tom"))//查询时不分词

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

.setFrom(0).setSize(10)//分页

.addSort("age", SortOrder.DESC)//排序

.get();

SearchHits hits = searchResponse.getHits();

long total = hits.getTotalHits();

System.out.println(total);

SearchHit[] searchHits = hits.hits();

for(SearchHit s : searchHits)

{

System.out.println(s.getSourceAsString());

}

}

/\*\*

\* 多索引，多类型查询

\* timeout

\*/

@Test

public void testSearchsAndTimeout()

{

SearchResponse searchResponse = transportClient.prepareSearch("shb01","shb02").setTypes("stu","tea")

.setQuery(QueryBuilders.matchAllQuery())

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

.setTimeout("3")

.get();

SearchHits hits = searchResponse.getHits();

long totalHits = hits.getTotalHits();

System.out.println(totalHits);

SearchHit[] hits2 = hits.getHits();

for(SearchHit h : hits2)

{

System.out.println(h.getSourceAsString());

}

}

/\*\*

\* 过滤，

\* lt 小于

\* gt 大于

\* lte 小于等于

\* gte 大于等于

\*

\*/

@Test

public void testFilter()

{

SearchResponse searchResponse = transportClient.prepareSearch(index)

.setTypes(type)

.setQuery(QueryBuilders.matchAllQuery()) //查询所有

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

// .setPostFilter(FilterBuilders.rangeFilter("age").from(0).to(19)

// .includeLower(true).includeUpper(true))

.setPostFilter(FilterBuilders.rangeFilter("age").gte(18).lte(22))

.setExplain(true) //explain为true表示根据数据相关度排序，和关键字匹配最高的排在前面

.get();

SearchHits hits = searchResponse.getHits();

long total = hits.getTotalHits();

System.out.println(total);

SearchHit[] searchHits = hits.hits();

for(SearchHit s : searchHits)

{

System.out.println(s.getSourceAsString());

}

}

/\*\*

\* 高亮

\*/

@Test

public void testHighLight()

{

SearchResponse searchResponse = transportClient.prepareSearch(index)

.setTypes(type)

//.setQuery(QueryBuilders.matchQuery("name", "Fresh")) //查询所有

.setQuery(QueryBuilders.queryString("name:F\*"))

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

.addHighlightedField("name")

.setHighlighterPreTags("<font color='red'>")

.setHighlighterPostTags("</font>")

.get();

SearchHits hits = searchResponse.getHits();

System.out.println("sum:" + hits.getTotalHits());

SearchHit[] hits2 = hits.getHits();

for(SearchHit s : hits2)

{

Map<String, HighlightField> highlightFields = s.getHighlightFields();

HighlightField highlightField = highlightFields.get("name");

if(null != highlightField)

{

Text[] fragments = highlightField.fragments();

System.out.println(fragments[0]);

}

System.out.println(s.getSourceAsString());

}

}

/\*\*

\* 分组

\*/

@Test

public void testGroupBy()

{

SearchResponse searchResponse = transportClient.prepareSearch(index).setTypes(type)

.setQuery(QueryBuilders.matchAllQuery())

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

.addAggregation(AggregationBuilders.terms("group\_age")

.field("age").size(0))//根据age分组，默认返回10，size(0)也是10

.get();

Terms terms = searchResponse.getAggregations().get("group\_age");

List<Bucket> buckets = terms.getBuckets();

for(Bucket bt : buckets)

{

System.out.println(bt.getKey() + " " + bt.getDocCount());

}

}

/\*\*

\* 聚合函数,本例之编写了sum，其他的聚合函数也可以实现

\*

\*/

@Test

public void testMethod()

{

SearchResponse searchResponse = transportClient.prepareSearch(index).setTypes(type)

.setQuery(QueryBuilders.matchAllQuery())

.setSearchType(SearchType.QUERY\_THEN\_FETCH)

.addAggregation(AggregationBuilders.terms("group\_name").field("name")

.subAggregation(AggregationBuilders.sum("sum\_age").field("age")))

.get();

Terms terms = searchResponse.getAggregations().get("group\_name");

List<Bucket> buckets = terms.getBuckets();

for(Bucket bt : buckets)

{

Sum sum = bt.getAggregations().get("sum\_age");

System.out.println(bt.getKey() + " " + bt.getDocCount() + " "+ sum.getValue());

}

}

}

**10 性能结果展示**

**（1）硬件配置：**

CPU 16核 AuthenticAMD

内存 总量：32GB

硬盘 总量：500GB 非SSD

**（2）在上述硬件指标的基础上测试性能如下：**

1）平均索引吞吐量： 12307docs/s（每个文档大小：40B/docs）

2）平均CPU使用率： 887.7%（16核，平均每核：55.48%）

3）构建索引大小： 3.30111 GB

4）总写入量： 20.2123 GB

5）测试总耗时： 28m 54s.

**11 ES部署（无需安装）**

1）零配置，开箱即用

2）没有繁琐的安装配置

3）java版本要求：最低1.7

我使用的1.8

[root@laoyang config\_lhy]# echo $JAVA\_HOME

/opt/jdk1.8.0\_91

4）下载地址：

https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/zip/elasticsearch/2.3.5/elasticsearch-2.3.5.zip

5）启动

cd /usr/local/elasticsearch-2.3.5

./bin/elasticsearch

bin/elasticsearch -d(后台运行)

6）ES必要的插件

必要的Head、kibana、IK（中文分词）、graph等插件的详细安装和使用。

http://blog.csdn.net/column/details/deep-elasticsearch.html

**12 ES遇到问题怎么办？**

1）国外：https://discuss.elastic.co/

2）国内：http://elasticsearch.cn/

**13 ES Restful API**

**（1）操作类型含义**

1）GET：获取请求对象的当前状态。

2）POST：改变对象的当前状态。

3）PUT：创建一个对象。

4）DELETE：销毁对象。

5）HEAD：请求获取对象的基础信息。

**（2） 新建文档（类似mysql insert插入操作）**

http://localhost:9200/blog/ariticle/1 put

{

"title":"New version of Elasticsearch released!",

"content":"Version 1.0 released today!",

"tags":["announce","elasticsearch","release"]

}

创建成功如下显示：

{

- "\_index": "blog",

- "\_type": "ariticle",

- "\_id": "1 -d",

- "\_version": 1,

- "\_shards": {

- "total": 2,

- "successful": 1,

- "failed": 0

- },

- "created": true

}



**（3）检索文档（类似mysql search 搜索select\*操作）**

http://localhost:9200/blog/ariticle/1/ GET

检索结果如下：

{

- "\_index": "blog",

- "\_type": "ariticle",

- "\_id": "1",

- "\_version": 1,

- "found": true,

- "\_source": {

- "title": "New version of Elasticsearch released!",

- "content": "Version 1.0 released today!",

- "tags": [

- "announce"

- ,

- "elasticsearch"

- ,

- "release"

- ]

- }

}

如果未找到会提示：

{

- "\_index": "blog",

- "\_type": "ariticle",

- "\_id": "11",

- "found": false

}

查询全部文档如下：



**查询举例2：查询书名title中包含“enhance”字段，ID值为3,5,7的数据信息：**

[root@5b9dbaaa148a ~]# curl -XGET 10.200.1.121:9200/blog/ariticle/\_search?pretty -d ‘

{ "query" : {

"terms" :

{"title" : "enhance" }

{"\_id" : [ "3", "5", "7" ] }

}

}'

{

"took" : 5,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 3,

"max\_score" : 0.19245009,

"hits" : [ {

"\_index" : "blog",

"\_type" : "ariticle",

"\_id" : "5",

"\_score" : 0.19245009,

"\_source" : {

"title" : "enhance Elasticsearch for university!",

"content" : "Version 5.0!",

"tags" : [ "enhance", "elasticsearch" ]

}

}, {

"\_index" : "blog",

"\_type" : "ariticle",

"\_id" : "7",

"\_score" : 0.19245009,

"\_source" : {

"title" : "deep Elasticsearch for university!",

"content" : "Version 2.0!",

"tags" : [ "deep", "elasticsearch", "university" ]

}

}, {

"\_index" : "blog",

"\_type" : "ariticle",

"\_id" : "3",

"\_score" : 0.19245009,

"\_source" : {

"title" : "init Elasticsearch for university!",

"content" : "Version 3.0!",

"tags" : [ "initialize", "elasticsearch" ]

}

} ]

}

}

**（4）更新文档（类似mysql update操作）**

http://localhost:9200/blog/ariticle/1/\_update/ POST

{“script”:”ctx.\_source.content = \”new version 2.0 20160714\”“}

更新后结果显示：

{

“\_index”: “blog”,

“\_type”: “ariticle”,

“\_id”: “1”,

“\_version”: 2,

“\_shards”: {

”total”: 2,

“successful”: 1,

“failed”: 0

}

}

查询&验证更新后结果：（对比可知，版本号已经更新完毕）

http://localhost:9200/blog/ariticle/1/

{

- "\_index": "blog",

- "\_type": "ariticle",

- "\_id": "1",

- "\_version": 2,

- "found": true,

- "\_source": {

- "title": "New version of Elasticsearch released!",

- "content": "new version 2.0 20160714",

- "tags": [

- "announce"

- ,

- "elasticsearch"

- ,

- "release"

- ]

- }

}

注意更新文档需要在elasticsearch\_win\config\elasticsearch.yml下新增以下内容：

script.groovy.sandbox.enabled: true

script.engine.groovy.inline.search: on

script.engine.groovy.inline.update: on

script.inline: on

script.indexed: on

script.engine.groovy.inline.aggs: on

index.mapper.dynamic: true

**（5）删除文档（类似mysql delete操作）**

http://localhost:9200/blog/ariticle/8/回结果

{

- "found": true,

- "\_index": "blog",

- "\_type": "ariticle",

- "\_id": "8",

- "\_version": 2,

- "\_shards": {

- "total": 2,

- "successful": 1,

- "failed": 0

- }

}



**14 优点集合**

1）分布式实时文件存储，可将每一个字段存入索引，使其可以被检索到。

2）实时分析的分布式搜索引擎。

分布式：索引分拆成多个分片，每个分片可有零个或多个副本。集群中的每个数据节点都可承载一个或多个分片，并且协调和处理各种操作； 负载再平衡和路由在大多数情况下自动完成。

3）可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。也可以运行在单台PC上（已测试）

4）支持插件机制，分词插件、同步插件、Hadoop插件、可视化插件等。

1. **使用场景**

1）新系统开发尝试使用ES作为存储和检索服务器；

2）现有系统升级需要支持全文检索服务，需要使用ES。

3）新浪ES 如何分析处理32亿条实时日志 http://dockone.io/article/505

4）阿里ES 构建挖财自己的日志采集和分析体系 http://afoo.me/columns/tec/logging-platform-spec.html

5）有赞ES 业务日志处理 http://tech.youzan.com/you-zan-tong-ri-zhi-ping-tai-chu-tan/

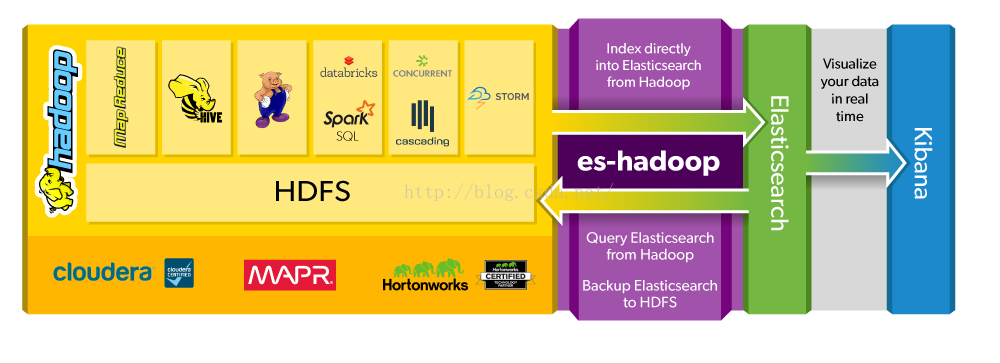
6）ES实现站内搜索 http://www.wtoutiao.com/p/13bkqiZ.html

7）检索相关数据；返回统计结果；速度要快

## 二、ES-Hadoop

**1 ES-Hadoop简介**

连接快速查询和大数据分析的桥梁，它能够无间隙的在hadoop和ElasticSearch上移动数据。ES Hadoop索引Hadoop数据到Elasticsearch，充分利用其查询速度，大量聚合能力来使它比以往更快，同时可以使用HDFS作为Elasticsearch长期存档。ES-Hadoop可以本地集成Hadoop生态系统上的很多流行组件，比如Spark、Hive、Pig、Storm、MapReduce等。官方有张图可以很好说明。

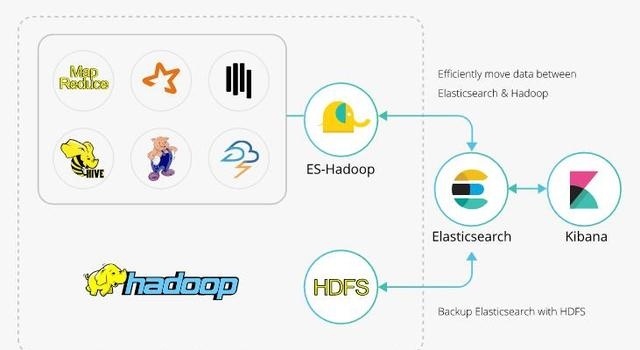


**2 Hadoop是标准的大数据生态代表**

里面有非常多的组件来处理不同类型或者场景下的数据，Hadoop的基础组件是YARN,HDFS,MapReduce，我们都知道HDFS是可靠的分布式存储系统，大多数我们都是用MapReduce来分析数据，唯一的不足之处在于速度，为了解决这种问题所以才有了Hbase，Spark，Kylin，Presto，Imapla等等许多框架。而我们的elasticsearch却恰恰相反，尤其是其定位高性能的搜索引擎，处理多维数据的检索分析非常高效，此外ES也是一个分布式的，高可靠的，可扩展的搜索框架，这些特点也决定了其处理海量数据的效率也是非常出色的。但es和hadoop属于两个不同的框架，如果想互相共享数据来处理，就需要自己来写程序把各自的数据导入需要的一方，过程非常繁琐，并且需要关注各自框架的版本，从而容易出现问题。

ES-Hadoop的出现则解决了这个问题，我们可以把它看做是ES和Hadoop大数据生态圈之间的数据桥梁，通过它，我们可以快速的分析Hadoop里面的海量数据。前面说了Hadoop的MapReduce定位是一个离线的批处理计算框架，而现在越来越多的服务，都要求是实时或者近实时的交互式分析，通过ES-Hadoop我们可以轻松的将Hadoop集群上面的数据导入到ES，从而通过使用ES来获得高性能，低延迟，并支持各种聚合，空间检索以及产品推荐的一些特性。最后还可以使用Kibana提供的可视化的数据分析一条龙服务，非常棒的组合。

**3 数据流转图**



ES-Hadoop无缝打通了ES和Hadoop两个非常优秀的框架，我们既可以把HDFS的数据导入到ES里面做分析，也可以将es数据导出到HDFS上做备份，归档，其中值得一提的是ES-Hadoop全面的支持了Spark框架，其中包括Spark，SparkStreaming，SparkSQL，此外也支持Hive，Pig，Storm，Cascading，当然还有标准的MapReduce，无论用那一个框架集成ES，都是非常简洁的。最后ES-Hadoop对各种版本的Hadoop都支持，这里面包含社区版本的ApacheHadoop，Cloudrea的CDH，MapR以及Hortonworks的HDP所以无论我们使用哪个版本的Hadoop都可以非常easy的与ES集成，从而让ES的强大性能帮助我们快速分析海量数据。

**4 java移动数据**

**（1）下面直接看一个简单的ES与Hadoop之间数据移动的实例，项目依赖的jar包如下**

<dependency>

<groupId>org.elasticsearch</groupId>

<artifactId>elasticsearch-hadoop</artifactId>

<version>2.3.2</version>

</dependency>

**（2）ElasticSearch到Hadoop最简单的实例**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

import org.elasticsearch.hadoop.mr.EsInputFormat;

import org.elasticsearch.hadoop.mr.LinkedMapWritable;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class E2HJob01 {

private static Logger LOG = LoggerFactory.getLogger(E2HJob01.class);

public static void main(String args[]) {

try {

Configuration conf = new Configuration();

conf.setBoolean("mapreduce.map.speculative", false);

conf.setBoolean("mapreduce.reduce.speculative", false);

//ElasticSearch节点

conf.set("es.nodes", "centos.host1:9200");

//ElaticSearch Index/Type

conf.set("es.resource", "job/51/");

String[] oArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (oArgs.length != 1) {

LOG.error("error");

System.exit(2);

}

Job job = Job.getInstance(conf, "51JOBE2H01");

job.setJarByClass(E2HJob01.class);

job.setInputFormatClass(EsInputFormat.class);

job.setMapperClass(E2HMapper01.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(LinkedMapWritable.class);

FileOutputFormat.setOutputPath(job, new Path(oArgs[0]));

System.out.println(job.waitForCompletion(true));

} catch (Exception e) {

LOG.error(e.getMessage(), e);

}

}

}

class E2HMapper01 extends Mapper<Text, LinkedMapWritable, Text, LinkedMapWritable> {

private static final Logger LOG = LoggerFactory.getLogger(E2HMapper01.class);

@Override

protected void setup(Context context) throws IOException, InterruptedException {

super.setup(context);

}

@Override

protected void map(Text key, LinkedMapWritable value, Context context)

throws IOException, InterruptedException {

LOG.info("key {} value {}", key, value);

context.write(key, value);

}

@Override

protected void cleanup(Context context) throws IOException, InterruptedException {

super.cleanup(context);

}

}

**（3）hadoop jar eshadoop.jar E2HJob01 /user/data/es/job/**

从hadoop上的数据文件可以看到第一列是ES的doc id，第二列是doc data

也可以添加ES查询条件，实例如下

import java.io.IOException;

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import org.apache.commons.lang.StringUtils;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.Writable;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

import org.elasticsearch.hadoop.mr.EsInputFormat;

import org.elasticsearch.hadoop.mr.LinkedMapWritable;

import org.platform.eshadoop.modules.examples.writable.JobWritable;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class E2HJob02 {

private static Logger LOG = LoggerFactory.getLogger(E2HJob02.class);

public static void main(String args[]) {

try {

Configuration conf = new Configuration();

conf.setBoolean("mapreduce.map.speculative", false);

conf.setBoolean("mapreduce.reduce.speculative", false);

conf.set("es.nodes", "centos.host1:9200");

conf.set("es.resource", "job/51/");

conf.set("es.query", "?q=高\*");

String[] oArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (oArgs.length != 1) {

LOG.error("error");

System.exit(2);

}

Job job = Job.getInstance(conf, "51JOBE2H02");

job.setJarByClass(E2HJob02.class);

job.setInputFormatClass(EsInputFormat.class);

job.setMapperClass(E2HMapper02.class);

job.setMapOutputKeyClass(NullWritable.class);

job.setMapOutputValueClass(JobWritable.class);

FileOutputFormat.setOutputPath(job, new Path(oArgs[0]));

System.out.println(job.waitForCompletion(true));

} catch (Exception e) {

LOG.error(e.getMessage(), e);

}

}

}

class E2HMapper02 extends Mapper<Text, LinkedMapWritable, NullWritable, JobWritable> {

private static final Logger LOG = LoggerFactory.getLogger(E2HMapper02.class);

@Override

protected void setup(Context context) throws IOException, InterruptedException {

super.setup(context);

}

@Override

protected void map(Text key, LinkedMapWritable value, Context context)

throws IOException, InterruptedException {

JobWritable writable = new JobWritable();

writable.setId(key);

Map<String, String> map = new HashMap<String, String>();

for (Entry<Writable, Writable> entry : value.entrySet()) {

LOG.info("key {} value {}", entry.getKey(), entry.getValue());

map.put(entry.getKey().toString(), entry.getValue().toString());

}

String jobName = map.get("jobName");

if (StringUtils.isNotBlank(jobName)) {

writable.setJobName(new Text(jobName));

}

String jobUrl = map.get("jobUrl");

if (StringUtils.isNotBlank(jobUrl)) {

writable.setJobUrl(new Text(jobUrl));

}

String companyName = map.get("companyName");

if (StringUtils.isNotBlank(companyName)) {

writable.setCompanyName(new Text(companyName));

}

String companyUrl = map.get("companyUrl");

if (StringUtils.isNotBlank(companyUrl)) {

writable.setCompanyUrl(new Text(companyUrl));

}

String salary = map.get("salary");

if (StringUtils.isNotBlank(salary)) {

writable.setSalary(new Text(salary));

}

String workPlace = map.get("workPlace");

if (StringUtils.isNotBlank(workPlace)) {

writable.setWorkPlace(new Text(workPlace));

}

String contact = map.get("contact");

if (StringUtils.isNotBlank(contact)) {

writable.setContact(new Text(contact));

}

String welfare = map.get("welfare");

if (StringUtils.isNotBlank(welfare)) {

writable.setWelfare(new Text(welfare));

}

context.write(NullWritable.get(), writable);

}

@Override

protected void cleanup(Context context) throws IOException, InterruptedException {

super.cleanup(context);

}

}

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.Writable;

public class JobWritable implements Writable, Cloneable {

private Text id = null;

private Text jobName = null;

private Text jobUrl = null;

private Text companyName = null;

private Text companyUrl = null;

private Text salary = null;

private Text workPlace = null;

private Text contact = null;

private Text welfare = null;

public JobWritable() {

id = new Text();

jobName = new Text();

jobUrl = new Text();

companyName = new Text();

companyUrl = new Text();

salary = new Text();

workPlace = new Text();

contact = new Text();

welfare = new Text();

}

public void readFields(DataInput dataInput) throws IOException {

id.readFields(dataInput);

jobName.readFields(dataInput);

jobUrl.readFields(dataInput);

companyName.readFields(dataInput);

companyUrl.readFields(dataInput);

salary.readFields(dataInput);

workPlace.readFields(dataInput);

contact.readFields(dataInput);

welfare.readFields(dataInput);

}

public void write(DataOutput dataOutput) throws IOException {

id.write(dataOutput);

jobName.write(dataOutput);

jobUrl.write(dataOutput);

companyName.write(dataOutput);

companyUrl.write(dataOutput);

salary.write(dataOutput);

workPlace.write(dataOutput);

contact.write(dataOutput);

welfare.write(dataOutput);

}

public Text getId() {

return id;

}

public void setId(Text id) {

this.id = id;

}

public Text getJobName() {

return jobName;

}

public void setJobName(Text jobName) {

this.jobName = jobName;

}

public Text getJobUrl() {

return jobUrl;

}

public void setJobUrl(Text jobUrl) {

this.jobUrl = jobUrl;

}

public Text getCompanyName() {

return companyName;

}

public void setCompanyName(Text companyName) {

this.companyName = companyName;

}

public Text getCompanyUrl() {

return companyUrl;

}

public void setCompanyUrl(Text companyUrl) {

this.companyUrl = companyUrl;

}

public Text getSalary() {

return salary;

}

public void setSalary(Text salary) {

this.salary = salary;

}

public Text getWorkPlace() {

return workPlace;

}

public void setWorkPlace(Text workPlace) {

this.workPlace = workPlace;

}

public Text getContact() {

return contact;

}

public void setContact(Text contact) {

this.contact = contact;

}

public Text getWelfare() {

return welfare;

}

public void setWelfare(Text welfare) {

this.welfare = welfare;

}

@Override

public String toString() {

return id + ":" + jobName + ":" + jobUrl + ":" + companyName + ":" + companyUrl +

":" + salary + ":" + workPlace + ":" + contact + ":" + welfare;

}

}

**（4）下面这个实例是每行直接以json格式存储在hadoop上**

import java.io.IOException;

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.Writable;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

import org.elasticsearch.hadoop.mr.EsInputFormat;

import org.elasticsearch.hadoop.mr.LinkedMapWritable;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.google.gson.Gson;

public class E2HJob03 {

private static Logger LOG = LoggerFactory.getLogger(E2HJob03.class);

public static void main(String args[]) {

try {

Configuration conf = new Configuration();

conf.setBoolean("mapreduce.map.speculative", false);

conf.setBoolean("mapreduce.reduce.speculative", false);

conf.set("es.nodes", "centos.host1:9200");

conf.set("es.resource", "job/51/");

String[] oArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (oArgs.length != 1) {

LOG.error("error");

System.exit(2);

}

Job job = Job.getInstance(conf, "51JOBE2H03");

job.setJarByClass(E2HJob03.class);

job.setInputFormatClass(EsInputFormat.class);

job.setMapperClass(E2HMapper03.class);

job.setMapOutputKeyClass(NullWritable.class);

job.setMapOutputValueClass(Text.class);

FileOutputFormat.setOutputPath(job, new Path(oArgs[0]));

System.out.println(job.waitForCompletion(true));

} catch (Exception e) {

LOG.error(e.getMessage(), e);

}

}

}

class E2HMapper03 extends Mapper<Text, LinkedMapWritable, NullWritable, Text> {

private static final Logger LOG = LoggerFactory.getLogger(E2HMapper02.class);

private Gson gson = null;

@Override

protected void setup(Context context) throws IOException, InterruptedException {

super.setup(context);

gson = new Gson();

}

@Override

protected void map(Text key, LinkedMapWritable value, Context context)

throws IOException, InterruptedException {

JobInfo jobInfo = new JobInfo();

jobInfo.setId(key.toString());

Map<String, String> map = new HashMap<String, String>();

for (Entry<Writable, Writable> entry : value.entrySet()) {

LOG.info("key {} value {}", entry.getKey(), entry.getValue());

map.put(entry.getKey().toString(), entry.getValue().toString());

}

jobInfo.setJobName(map.get("jobName"));

jobInfo.setJobUrl(map.get("jobUrl"));

jobInfo.setCompanyName(map.get("companyName"));

jobInfo.setCompanyUrl(map.get("companyUrl"));

jobInfo.setSalary(map.get("salary"));

jobInfo.setWorkPlace(map.get("workPlace"));

jobInfo.setContact(map.get("contact"));

jobInfo.setWelfare(map.get("welfare"));

context.write(NullWritable.get(), new Text(gson.toJson(jobInfo)));

}

@Override

protected void cleanup(Context context) throws IOException, InterruptedException {

super.cleanup(context);

}

}

class JobInfo {

private String id = null;

private String jobName = null;

private String jobUrl = null;

private String companyName = null;

private String companyUrl = null;

private String salary = null;

private String workPlace = null;

private String contact = null;

private String welfare = null;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getJobName() {

return jobName;

}

public void setJobName(String jobName) {

this.jobName = jobName;

}

public String getJobUrl() {

return jobUrl;

}

public void setJobUrl(String jobUrl) {

this.jobUrl = jobUrl;

}

public String getCompanyName() {

return companyName;

}

public void setCompanyName(String companyName) {

this.companyName = companyName;

}

public String getCompanyUrl() {

return companyUrl;

}

public void setCompanyUrl(String companyUrl) {

this.companyUrl = companyUrl;

}

public String getSalary() {

return salary;

}

public void setSalary(String salary) {

this.salary = salary;

}

public String getWorkPlace() {

return workPlace;

}

public void setWorkPlace(String workPlace) {

this.workPlace = workPlace;

}

public String getContact() {

return contact;

}

public void setContact(String contact) {

this.contact = contact;

}

public String getWelfare() {

return welfare;

}

public void setWelfare(String welfare) {

this.welfare = welfare;

}

}

**（5）接下来的实例是将hadoop上的数据移动到ElasticSearch上索引，这里直接用上面存储的JSON数据试验**

执行hadoop jar eshadoop.jar H2EJob /user/data/es/job后，可以在ES上看到数据已经索引过来。

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.Writable;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

import org.elasticsearch.hadoop.mr.EsOutputFormat;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class H2EJob {

private static Logger LOG = LoggerFactory.getLogger(H2EJob.class);

public static void main(String args[]) {

try {

Configuration conf = new Configuration();

conf.setBoolean("mapreduce.map.speculative", false);

conf.setBoolean("mapreduce.reduce.speculative", false);

conf.set("es.nodes", "centos.host1:9200");

conf.set("es.resource", "job1/51");

//Hadoop上的数据格式为JSON,可以直接导入

conf.set("es.input.json", "yes");

String[] oArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (oArgs.length != 1) {

LOG.error("error");

System.exit(2);

}

Job job = Job.getInstance(conf, "51JOBH2E");

job.setJarByClass(H2EJob.class);

job.setMapperClass(H2EMapper.class);

job.setMapOutputKeyClass(NullWritable.class);

job.setMapOutputValueClass(Text.class);

job.setOutputFormatClass(EsOutputFormat.class);

FileInputFormat.addInputPath(job, new Path(oArgs[0]));

System.out.println(job.waitForCompletion(true));

} catch (Exception e) {

LOG.error(e.getMessage(), e);

}

}

}

class H2EMapper extends Mapper<LongWritable, Text, NullWritable, Text> {

@Override

protected void setup(Context context) throws IOException, InterruptedException {

super.setup(context);

}

@Override

public void run(Context context) throws IOException, InterruptedException {

super.run(context);

}

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

context.write(NullWritable.get(), value);

}

@Override

protected void cleanup(Context context) throws IOException,InterruptedException {

super.cleanup(context);

}

}

## 三、Logstash

**1 简要说明**

logstash是一个数据分析软件，主要目的是分析log日志。整一套软件可以当作一个MVC模型，logstash是controller层，Elasticsearch是一个model层，kibana是view层。首先将数据传给logstash，它将数据进行过滤和格式化（转成JSON格式），然后传给Elasticsearch进行存储、建搜索的索引，kibana提供前端的页面再进行搜索和图表可视化，它是调用Elasticsearch的接口返回的数据进行可视化。logstash和Elasticsearch是用Java写的，kibana使用node.js框架。Logstash是一个接收，处理，转发日志的工具。支持系统日志，webserver日志，错误日志，应用日志，总之包括所有可以抛出来的日志类型。

**2 下载并启动服务**

**（1）下载命令**

curl -O https://download.elasticsearch.org/logstash/logstash/logstash-{logstash\_version}.tar.gz

tar -zxvf logstash-1.5.0.tar.gz

cd logstash-{logstash\_version}

**（2）启动服务，并启用控制台输入模式，输出方式为控制台**

bin/logstash -e 'input { stdin { } } output { stdout {} }'

在控制台看到的结果是：

hello world

2015-05-28T01:22:14.405+0000 0.0.0.0 hello world

**（3）通过Ctrl+C退出程序**

执行以下命令：

启动服务，并启用控制台输入，输出方式为RubyDebug。

bin/logstash -e 'input { stdin { } } output {

stdout { codec => rubydebug }

}'

返回的结果是这样的

输入：goodnight moon

输出：{

"message" => "goodnight moon",

"@timestamp" => "2015-05-28T23:48:05.335Z",

"@version" => "1",

"host" => "my-laptop"

}

当输出方式为RubyDebug情况下，Logstash会自动输出一个JSON格式的结果。并自动添加了当前的系统时间，版本号，及Host信息。

在大多数使用Logstash的场景下，都会选择使用ElasticSearch作为数据存储。

那么如何配置Logstash的数据输出的ElasticSearch中呢？非常的简单，

执行以下命令，并输入任何数据，就可以在ElasticSearch中查看数据了。

bin/logstash -e 'input { stdin { } } output { elasticsearch { host => localhost } }'

通过浏览器访问地址，查看数据。

http://localhost:9200/\_search?pretty

配置多种Logstash数据方式,如在控制台打印并输出到ElasticSearch中。

bin/logstash -e 'input { stdin { } } output { elasticsearch { host => localhost } stdout { } }'

bin/logstash -e 'input { stdin { } } output { elasticsearch { host => localhost } stdout { codec => rubydebug } }'

**3 配置总结**

**（1）配置的组成**

Inputs,Outputs,Codecs,Filters构成了Logstash的核心配置项。Logstash通过建立一条事件处理的管道，从你的日志提取出数据保存到Elasticsearch中，为高效的查询数据提供基础。为了让你快速的了解Logstash提供的多种选项，让我们先讨论一下最常用的一些配置。更多的信息，请参考 Logstash事件管道 。

**（2）Inputs**

input 及输入是指日志数据传输到Logstash中。其中常见的配置如下：

file：从文件系统中读取一个文件，很像UNIX命令 "tail -0a"

syslog：监听514端口，按照RFC3164标准解析日志数据

redis：从redis服务器读取数据，支持channel(发布订阅)和list模式。redis一般 在Logstash消费集群中 作为"broker"角色，保存events队列共Logstash消费。

lumberjack：使用lumberjack协议来接收数据，目前已经改为 logstash-forwarder。

input {

#file可以多次使用，也可以只写一个file而设置它的path属性配置多个文件实现多文件监控

file {

#type是给结果增加了一个属性叫type值为"<xxx>"的条目。这里的type，对应了ES中index中的type，即如果输入ES时，没有指定type，那么这里的type将作为ES中index的type。

type => "apache-access"

path => "/apphome/ptc/Windchill\_10.0/Apache/logs/access\_log\*"

#start\_position可以设置为beginning或者end，beginning表示从头开始读取文件，end表示读取最新的，这个也要和ignore\_older一起使用。

start\_position => beginning

#sincedb\_path表示文件读取进度的记录，每行表示一个文件，每行有两个数字，第一个表示文件的inode，第二个表示文件读取到的位置（byteoffset）。默认为$HOME/.sincedb\*

sincedb\_path => "/opt/logstash-2.3.1/sincedb\_path/access\_progress"

#ignore\_older表示了针对多久的文件进行监控，默认一天，单位为秒，可以自己定制，比如默认只读取一天内被修改的文件。

ignore\_older => 604800

#add\_field增加属性。这里使用了${HOSTNAME}，即本机的环境变量，如果要使用本机的环境变量，那么需要在启动命令上加--alow-env。

add\_field => {"log\_hostname"=>"${HOSTNAME}"}

#这个值默认是\n 换行符，如果设置为空""，那么后果是每个字符代表一个event

delimiter => ""

#这个表示关闭超过（默认）3600秒后追踪文件。这个对于multiline来说特别有用。... 这个参数和logstash对文件的读取方式有关，两种方式read tail，如果是read

close\_older => 3600

coodec => multiline {

pattern => "^\s"

#这个negate是否定的意思，意思跟pattern相反，也就是不满足patter的意思。

# negate => ""

#what有两个值可选 previous和next，举例说明，java的异常从第二行以空格开始，这里就可以pattern匹配空格开始，what设置为previous意思是空格开头这行跟上一行属于同一event。另一个例子，有时候一条命令太长，当以\结尾时表示这行属于跟下一行属于同一event，这时需要使用negate=>true，what=>'next'。

what => "previous"

auto\_flush\_interval => 60

}

}

file {

type => "methodserver-log"

path => "/apphome/ptc/Windchill\_10.0/Windchill/logs/MethodServer-1604221021-32380.log"

start\_position => beginning

sincedb\_path => "/opt/logstash-2.3.1/sincedb\_path/methodserver\_process"

# ignore\_older => 604800

}

}

**（3）Filters**

Fillters 在Logstash处理链中担任中间处理组件。他们经常被组合起来实现一些特定的行为来，处理匹配特定规则的事件流。常见的filters如下：

**grok：**解析无规则的文字并转化为有结构的格式。Grok 是目前最好的方式来将无结构的数据转换为有结构可查询的数据。有120多种匹配规则，会有一种满足你的需要。

**mutate：**mutate filter 允许改变输入的文档，你可以从命名，删除，移动或者修改字段在处理事件的过程中。

**drop：**丢弃一部分events不进行处理，例如：debug events。

**clone：**拷贝 event，这个过程中也可以添加或移除字段。

**geoip：**添加地理信息(为前台kibana图形化展示使用)

filter{

#执行ruby程序，下面例子是将日期转化为字符串赋予daytag

ruby {

code => "event['daytag'] = event.timestamp.time.localtime.strftime('%Y-%m-%d')"

}

# if [path] =~ "access" {} else if [path] =~ "methodserver" {} else if [path] =~ "servermanager" {} else {} 注意语句结构

if [path] =~ "MethodServer" { #z这里的=~是匹配正则表达式

grok {

patterns\_dir => ["/opt/logstash-2.3.1/patterns"] #自定义正则匹配

# Tue 4/12/16 14:24:17: TP-Processor2: hirecode---->77LS

match => { "message" => "%{DAY:log\_weekday} %{DATE\_US:log\_date} %{TIME:log\_time}: %{GREEDYDATA:log\_data}"}

}

#mutage是做转换用的

mutate {

replace => { "type" => "apache" } #替换属性值

convert => { #类型转换

"bytes" => "integer" #例如还有float

"duration" => "integer"

"state" => "integer"

}

#date主要是用来处理文件内容中的日期的。内容中读取的是字符串，

通过date将它转换为@timestamp。参考https://www.elastic.co/guide/en/logstash/current/plugins-filters-date.html#plugins-filters-date-match

# date {

# match => [ "logTime" , "dd/MMM/yyyy:HH:mm:ss Z" ]

# }

}else if [type] in ['tbg\_qas','mbg\_pre'] { # if ... else if ... else if ... else结构

}else {

drop{} # 将event丢弃

}

}

**（4）Outputs**

outputs是logstash处理管道的最末端组件。一个event可以在处理过程中经过多重输出，但是一旦所有的outputs都执行结束，这个event也就完成生命周期。一些常用的outputs。

**elasticsearch：**如果你计划将高效的保存数据，并且能够方便和简单的进行查询...Elasticsearch是一个好的方式。

**file：**将event数据保存到文件中。

**graphite：**将event数据发送到图形化组件中，一个很流行的开源存储图形化展示的组件。 http://graphite.wikidot.com/ 。

**statsd：**statsd是一个统计服务，比如技术和时间统计，通过udp通讯，聚合一个或者多个后台服务，如果你已经开始使用statsd，该选项对你应该很有用。

output {

stdout{ codec=>rubydebug} # 直接输出，调试用起来方便

# 输出到redis

redis {

host => '10.120.20.208'

data\_type => 'list'

key => '10.99.201.34:access\_log\_2016-04'

}

# 输出到ES

elasticsearch {

hosts =>"192.168.0.15:9200"

index => "%{sysid}\_%{type}"

document\_type => "%{daytag}"

}

}

**（5）Codecs**

codecs 是基于数据流的过滤器，它可以作为input，output的一部分配置。Codecs可以帮助你轻松的分割发送过来已经被序列化的数据。流行的codecs包括 json,msgpack,plain(text)。

**json：**使用json格式对数据进行编码/解码

**multiline：**将汇多个事件中数据汇总为一个单一的行。比如：java异常信息和堆栈信息

**5 过滤器**

filters是一个行处理机制将提供的为格式化的数据整理成你需要的数据，让我们看看下面的一个例子，叫grok filter的过滤器。

input { stdin { } }

filter {

grok {

match => { "message" => "%{COMBINEDAPACHELOG}" }

}

date {

match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]

}

}

output {

elasticsearch { host => localhost }

stdout { codec => rubydebug }

}

执行Logstash按照如下参数：

**bin/logstash -f logstash-filter.conf**

现在粘贴下面一行信息到你的终端(当然Logstash就会处理这个标准的输入内容)：

127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] "GET /xampp/status.php HTTP/1.1" 200 3891 "http://cadenza/xampp/navi.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0) Gecko/20100101 Firefox/25.0"你将看到类似如下内容的反馈信息：

{

"message" => "127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] \"GET /xampp/status.php HTTP/1.1\" 200 3891 \"http://cadenza/xampp/navi.php\" \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0) Gecko/20100101 Firefox/25.0\"",

"@timestamp" => "2013-12-11T08:01:45.000Z",

"@version" => "1",

"host" => "cadenza",

"clientip" => "127.0.0.1",

"ident" => "-",

"auth" => "-",

"timestamp" => "11/Dec/2013:00:01:45 -0800",

"verb" => "GET",

"request" => "/xampp/status.php",

"httpversion" => "1.1",

"response" => "200",

"bytes" => "3891",

"referrer" => "\"http://cadenza/xampp/navi.php\"",

"agent" => "\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0) Gecko/20100101 Firefox/25.0\""

}

正像你看到的那样，Logstash(使用了grok过滤器)能够将一行的日志数据(Apache的"combined log"格式)分割设置为不同的数据字段。这一点对于日后解析和查询我们自己的日志数据非常有用。比如：HTTP的返回状态码，IP地址相关等等，非常的容易。很少有匹配规则没有被grok包含，所以如果你正尝试的解析一些常见的日志格式，或许已经有人为了做了这样的工作。如果查看详细匹配规则，参考 logstash grok patterns 。

另外一个过滤器是date filter。这个过滤器来负责解析出来日志中的时间戳并将值赋给timestame字段(不管这个数据是什么时候收集到logstash的)。你也许注意到在这个例子中@timestamp字段是设置成December 11, 2013, 说明logstash在日志产生之后一段时间进行处理的。这个字段在处理日志中回添到数据中的，举例来说... 这个值就是logstash处理event的时间戳。

**5实用的例子**

Apache 日志(从文件获取)

现在，让我们使用一些非常实用的配置... apache2访问日志！我们将从本地读取日志文件，并且通过条件设置处理满足我们需要的event。首先，我们创建一个文件名是logstash-apache.conf的配置文件，内容如下(你可以根据实际情况修改你的文件名和路径)：

input {

file {

path => "/tmp/access\_log"

start\_position => beginning

}

}

filter {

if [path] =~ "access" {

mutate { replace => { "type" => "apache\_access" } }

grok {

match => { "message" => "%{COMBINEDAPACHELOG}" }

}

}

date {

match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]

}

}

output {

elasticsearch {

host => localhost

}

stdout { codec => rubydebug }

}

接下来，我们按照上面的配置创建一个文件(在例子中是"/tmp/access.log"),可以将下面日志信息作为文件内容(也可以用你自己的webserver产生的日志)：

71.141.244.242 - kurt [18/May/2011:01:48:10 -0700] "GET /admin HTTP/1.1" 301 566 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3"

134.39.72.245 - - [18/May/2011:12:40:18 -0700] "GET /favicon.ico HTTP/1.1" 200 1189 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; InfoPath.2; .NET4.0C; .NET4.0E)"

98.83.179.51 - - [18/May/2011:19:35:08 -0700] "GET /css/main.css HTTP/1.1" 200 1837 "http://www.safesand.com/information.htm" "Mozilla/5.0 (Windows NT 6.0; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1"现在使用-f参数来执行一下上面的例子：

bin/logstash -f logstash-apache.conf

你可以看到apache的日志数据已经导入到ES中了。这里logstash会按照你的配置读取，处理指定的文件，任何后添加到文件的内容也会被捕获处理最后保存到ES中。此外，数据中type的字段值会被替换成"apache\_access"(这个功能在配置中已经指定)。

这个配置只是让Logstash监控了apache access\_log，但是在实际中往往并不够用可能还需要监控error\_log，只要在上面的配置中改变一行既可以实现，如下：

input {

file {

path => "/tmp/\*\_log"

...现在你可以看到logstash处理了error日志和access日志。然而，如果你检查了你的数据(也许用elasticsearch-kopf )，你将发现access\_log日志被分成不同的字段，但是error\_log确没有这样。这是因为我们使用了“grok”filter并仅仅配置匹配combinedapachelog日志格式，这样满足条件的日志就会自动的被分割成不同的字段。我们可以通过控制日志按照它自己的某种格式来解析日志，不是很好的吗？对吧。

此外，你也许还会发现Logstash不会重复处理文件中已经处理过得events。因为Logstash已经记录了文件处理的位置，这样就只处理文件中新加入的行数。漂亮！

条件判断

我们利用上一个例子来介绍一下条件判断的概念。这个概念一般情况下应该被大多数的Logstash用户熟悉掌握。你可以像其他普通的编程语言一样来使用if,else if和else语句。让我们把每个event依赖的日志文件类型都标记出来(access\_log,error\_log其他以log结尾的日志文件)。

input {

file {

path => "/tmp/\*\_log"

}

}

filter {

if [path] =~ "access" {

mutate { replace => { type => "apache\_access" } }

grok {

match => { "message" => "%{COMBINEDAPACHELOG}" }

}

date {

match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]

}

} else if [path] =~ "error" {

mutate { replace => { type => "apache\_error" } }

} else {

mutate { replace => { type => "random\_logs" } }

}

}

output {

elasticsearch { host => localhost }

stdout { codec => rubydebug }

}

我想你已经注意到了，我们使用"type"字段来标记每个event,但是我们实际上没有解析"error"和”random"类型的日志... 而实际情况下可能会有很多很多类型的错误日志，如何解析就作为练习留给各位读者吧，你可以依赖已经存在的日志。

**6 Syslog**

Ok,现在我们继续了解一个很实用的例子：syslog。Syslog对于Logstash是一个很长用的配置，并且它有很好的表现(协议格式符合RFC3164)。Syslog实际上是UNIX的一个网络日志标准，由客户端发送日志数据到本地文件或者日志服务器。在这个例子中，你根本不用建立syslog实例;我们通过命令行就可以实现一个syslog服务，通过这个例子你将会看到发生什么。首先，让我们创建一个简单的配置文件来实现logstash+syslog，文件名是 **logstash-syslog.conf**

input {

tcp {

port => 5000

type => syslog

}

udp {

port => 5000

type => syslog

}

}

filter {

if [type] == "syslog" {

grok {

match => { "message" => "%{SYSLOGTIMESTAMP:syslog\_timestamp} %{SYSLOGHOST:syslog\_hostname} %{DATA:syslog\_program}(?:\[%{POSINT:syslog\_pid}\])?: %{GREEDYDATA:syslog\_message}" }

add\_field => [ "received\_at", "%{@timestamp}" ]

add\_field => [ "received\_from", "%{host}" ]

}

syslog\_pri { }

date {

match => [ "syslog\_timestamp", "MMM d HH:mm:ss", "MMM dd HH:mm:ss" ]

}

}

}

output {

elasticsearch { host => localhost }

stdout { codec => rubydebug }

}

执行logstash：

**bin/logstash -f logstash-syslog.conf**

通常，需要一个客户端链接到Logstash服务器上的5000端口然后发送日志数据。在这个简单的演示中我们简单的使用telnet链接到logstash服务器发送日志数据(与之前例子中我们在命令行标准输入状态下发送日志数据类似)。首先我们打开一个新的shell窗口，然后输入下面的命令：

telnet localhost 5000

你可以复制粘贴下面的样例信息(当然也可以使用其他字符，不过这样可能会被grok filter不能正确的解析)：

Dec 23 12:11:43 louis postfix/smtpd[31499]: connect from unknown[95.75.93.154]

Dec 23 14:42:56 louis named[16000]: client 199.48.164.7#64817: query (cache) 'amsterdamboothuren.com/MX/IN' denied

Dec 23 14:30:01 louis CRON[619]: (www-data) CMD (php /usr/share/cacti/site/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)

Dec 22 18:28:06 louis rsyslogd: [origin software="rsyslogd" swVersion="4.2.0" x-pid="2253" x-info="http://www.rsyslog.com"] rsyslogd was HUPed, type 'lightweight'.

之后你可以在你之前运行Logstash的窗口中看到输出结果，信息被处理和解析！

{

"message" => "Dec 23 14:30:01 louis CRON[619]: (www-data) CMD (php /usr/share/cacti/site/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)",

"@timestamp" => "2013-12-23T22:30:01.000Z",

"@version" => "1",

"type" => "syslog",

"host" => "0:0:0:0:0:0:0:1:52617",

"syslog\_timestamp" => "Dec 23 14:30:01",

"syslog\_hostname" => "louis",

"syslog\_program" => "CRON",

"syslog\_pid" => "619",

"syslog\_message" => "(www-data) CMD (php /usr/share/cacti/site/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)",

"received\_at" => "2013-12-23 22:49:22 UTC",

"received\_from" => "0:0:0:0:0:0:0:1:52617",

"syslog\_severity\_code" => 5,

"syslog\_facility\_code" => 1,

"syslog\_facility" => "user-level",

"syslog\_severity" => "notice"

}

恭喜各位，看到这里你已经成为一个合格的Logstash用户了。你将可以轻松的配置，运行Logstash，还可以发送event给Logstash，但是这个过程随着使用还会有很多值得深挖的地方。

## 四、Kinaba

**1 Kibana是一个开源的分析与可视化平台**

设计出来用于和Elasticsearch一起使用的。你可以用kibana搜索、查看、交互存放在Elasticsearch索引里的数据，使用各种不同的图表、表格、地图等kibana能够很轻易地展示高级数据分析与可视化。Kibana让我们理解大量数据变得很容易。它简单、基于浏览器的接口使你能快速创建和分享实时展现Elasticsearch查询变化的动态仪表盘。安装Kibana非常快，你可以在几分钟之内安装和开始探索你的Elasticsearch索引数据—-—-不需要写任何代码，没有其他基础软件依赖。（注意：这里只介绍Kibana如何使用，更多关于Kibana 4.3版本更新查看https://www.elastic.co/guide/en/kibana/current/releasenotes.html）

**2 一些基础**

@基础：

**metrics 度量**

**Y轴 平均数 最大值 最小值 总数 唯一的数量 百分比**

**出现多列 则统计多个值，下面有table形式**

**Search可以搜索关键字筛选，筛选出的结果集进行聚合**

**buckets 桶 X-Axis X轴一般是时间，柱状图也可以是terms**

**一般统计的是所有的频道，如果要分开：**

**Split Series:按照系列切分，一个图表出现多条线。**

**Split Chart:按照图表切分，分为行和列，行切分出现三行图表，每个设备独占一个。**

**行切分出现三列图表，每个设备独占一个。**

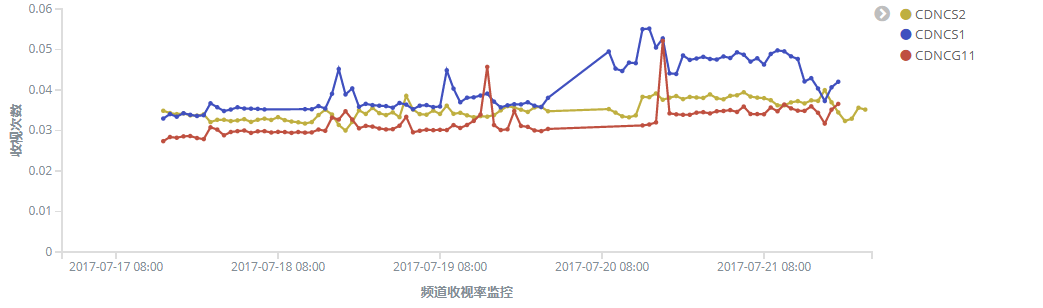
@Line线图,area 图：

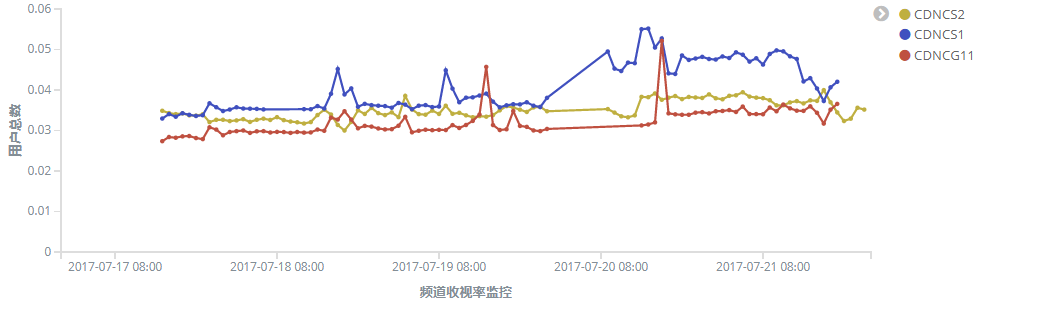
多条线：

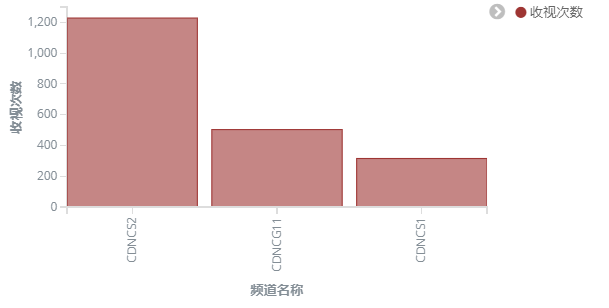
可以是统计所有设备的CPU平均律，内存平均律，2条线

可以是统计各个设备的CPU平均律，3条线

可以是统计各个设备的CPU平均律，内存平均律 一共6条线







**3 Kibana4 安装方式**

可以在几分钟内安装好 Kibana 然后开始探索你的 Elasticsearch 索引。只需要预备：

Elasticsearch 1.4.4 或者更新的版本

一个现代浏览器 - 支持的浏览器列表.

有关你的 Elasticsearch 集群的信息：

你想要连接 Elasticsearch 实例的 URL

你想搜索哪些 Elasticsearch 索引

**4 安装并启动 kibana**

要安装启动 Kibana:

下载对应平台的 Kibana 4 二进制包

解压 .zip 或 tar.gz 压缩文件

在安装目录里运行: bin/kibana (Linux/MacOSX) 或 bin\kibana.bat (Windows)

完毕！Kibana 现在运行在 5601 端口了。

**5 让 kibana 连接到 elasticsearch**

在开始用 Kibana 之前，你需要告诉它你打算探索哪个 Elasticsearch 索引。第一次访问 Kibana 的时候，你会被要求定义一个 index pattern 用来匹配一个或者多个索引名。好了。这就是你需要做的全部工作。以后你还可以随时从 Settings 标签页添加更多的 index pattern。默认情况下，Kibana 会连接运行在 localhost 的 Elasticsearch。要连接其他 Elasticsearch 实例，修改 kibana.yml 里的 Elasticsearch URL，然后重启 Kibana。如何在生产环境下使用 Kibana。要从 Kibana 访问的 Elasticsearch 索引的配置方法：从浏览器访问 Kibana 界面。也就是说访问比如 localhost:5601 或者 <http://YOURDOMAIN.com:5601。>

制定一个可以匹配一个或者多个 Elasticsearch 索引的 index pattern 。默认情况下，Kibana 认为你要访问的是通过 Logstash 导入 Elasticsearch 的数据。这时候你可以用默认的 logstash-\* 作为你的 index pattern。通配符(\*) 匹配索引名中零到多个字符。如果你的 Elasticsearch 索引有其他命名约定，输入合适的 pattern。pattern 也开始是最简单的单个索引的名字。

选择一个包含了时间戳的索引字段，可以用来做基于时间的处理。Kibana 会读取索引的映射，然后列出所有包含了时间戳的字段(译者注：实际是字段类型为 date 的字段，而不是“看起来像时间戳”的字段)。如果你的索引没有基于时间的数据，关闭 Index contains time-based events 参数。如果一个新索引是定期生成，而且索引名中带有时间戳，选择 Use event times to create index names 选项，然后再选择 Index pattern interval。这可以提高搜索性能，Kibana 会至搜索你指定的时间范围内的索引。在你用 Logstash 输出数据给 Elasticsearch 的情况下尤其有效。点击 Create 添加 index pattern。第一个被添加的 pattern 会自动被设置为默认值。如果你有多个 index pattern 的时候，你可以在 Settings > Indices 里设置具体哪个是默认值。

**6生产环境部署**

Kibana4 是是一个完整的 web 应用。使用时，你需要做的只是打开浏览器，然后输入你运行 Kibana 的机器地址然后加上端口号。比如说：localhost:5601 或者 http://YOURDOMAIN.com:5601。但是当你准备在生产环境使用 Kibana4 的时候，比起在本机运行，就需要多考虑一些问题：

@在哪运行 kibana

@是否需要加密 Kibana 出入的流量

@是否需要控制访问数据的权限

**7 Nginx 代理配置**

因为 Kibana4 不再是 Kibana3 那种纯静态文件的单页应用，所以其服务器端是需要消耗计算资源的。因此，如果用户较多，Kibana4 确实有可能需要进行多点部署，这时候，就需要用 Nginx 做一层代理了。和 Kibana3 相比，Kibana4 的 Nginx 代理配置倒是简单许多，因为所有流量都是统一配置的。下面是一段包含入口流量加密、简单权限控制的 Kibana4 代理配置：

upstream kibana4 {

server 127.0.0.1:5601 fail\_timeout=0;

}

server {

listen \*:80;

server\_name kibana\_server;

access\_log /var/log/nginx/kibana.srv-log-dev.log;

error\_log /var/log/nginx/kibana.srv-log-dev.error.log;

ssl on;

ssl\_certificate /etc/nginx/ssl/all.crt;

ssl\_certificate\_key /etc/nginx/ssl/server.key;

location / {

root /var/www/kibana;

index index.html index.htm;

}

location ~ ^/kibana4/.\* {

proxy\_pass http://kibana4;

rewrite ^/kibana4/(.\*) /$1 break;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

auth\_basic "Restricted";

auth\_basic\_user\_file /etc/nginx/conf.d/kibana.myhost.org.htpasswd;

}

}

如果用户够多，当然你可以单独跑一个 kibana4 集群，然后在 upstream 配置段中添加多个代理地址做负载均衡。

**8 配置 Kibana 和 shield 一起工作**

Nginx 只能加密和管理浏览器到服务器端的请求，而 Kibana4 到 ELasticsearch 集群的请求，就需要由 Elasticsearch 方面来完成了。如果你在用 Shield 做 Elasticsearch 用户认证，你需要给 Kibana 提供用户凭证，这样它才能访问 .kibana 索引。Kibana 用户需要由权限访问 .kibana 索引里以下操作：

'.kibana':

- indices:admin/create

- indices:admin/exists

- indices:admin/mapping/put

- indices:admin/mappings/fields/get

- indices:admin/refresh

- indices:admin/validate/query

- indices:data/read/get

- indices:data/read/mget

- indices:data/read/search

- indices:data/write/delete

- indices:data/write/index

- indices:data/write/update

- indices:admin/create

更多配置 Shield 的内容，请阅读官网的 Shield with Kibana 4。

要配置 Kibana 的凭证，设置 kibana.yml 里的 kibana\_elasticsearch\_username 和kibana\_elasticsearch\_password 选项即可：

# If your Elasticsearch is protected with basic auth:

kibana\_elasticsearch\_username: kibana4

kibana\_elasticsearch\_password: kibana4

**9 开启 ssl**

Kibana 同时支持对客户端请求以及 Kibana 服务器发往 Elasticsearch 的请求做 SSL 加密。要加密浏览器(或者在 Nginx 代理情况下，Nginx 服务器)到 Kibana 服务器之间的通信，配置 kibana.yml里的 ssl\_key\_file 和 ssl\_cert\_file 参数：

# SSL for outgoing requests from the Kibana Server (PEM formatted)

ssl\_key\_file: /path/to/your/server.key

ssl\_cert\_file: /path/to/your/server.crt

如果你在用 Shield 或者其他提供 HTTPS 的代理服务器保护 Elasticsearch，你可以配置 Kibana 通过 HTTPS 方式访问 Elasticsearch，这样 Kibana 服务器和 Elasticsearch 之间的通信也是加密的。要做到这点，你需要在 kibana.yml 里配置 Elasticsearch 的 URL 时指明是 HTTPS 协议：

**elasticsearch: "https://<your\_elasticsearch\_host>.com:9200"**

如果你给 Elasticsearch 用的是自己签名的证书，请在 kibana.yml 里设定 ca 参数指明 PEM 文件位置，这也意味着开启了 verify\_ssl 参数：

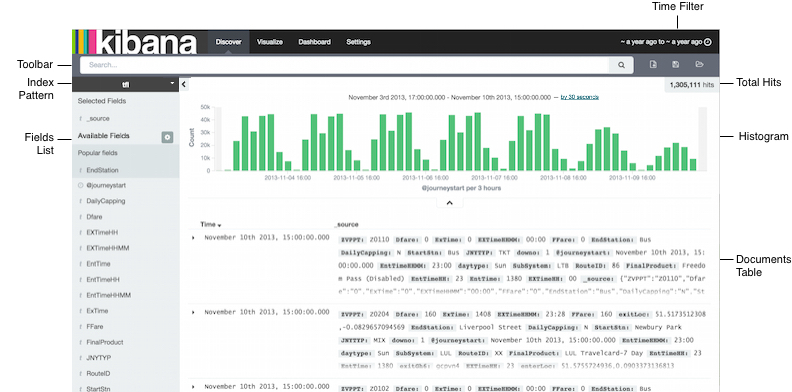
# If you need to provide a CA certificate for your Elasticsarech instance, put

# the path of the pem file here.

ca: /path/to/your/ca/cacert.pem

**10 discover 功能**

Discover 标签页用于交互式探索你的数据。你可以访问到匹配得上你选择的索引模式的每个索引的每条记录。你可以提交搜索请求，过滤搜索结果，然后查看文档数据。你还可以看到匹配搜索请求的文档总数，获取字段值的统计情况。如果索引模式配置了时间字段，文档的时序分布情况会在页面顶部以柱状图的形式展示出来。

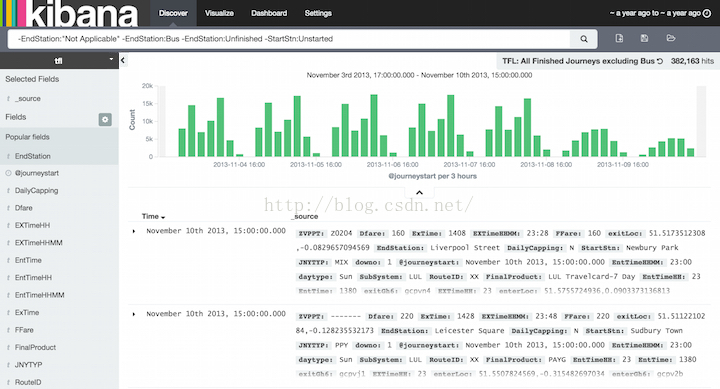


**11 基本使用【NMS 】**

首先让我们来看下你将如何使用Kibana来发掘与可视化数据。我们假设已经为一些数据建立好了索引，这些数据来源于伦敦交通局（TFL）显示最近一个星期牡蛎卡（类似一通卡）的使用情况。 在Kibana的发现页面我们可以提交查询、过滤结果、检查返回的文档数据，比如我们能获取所有

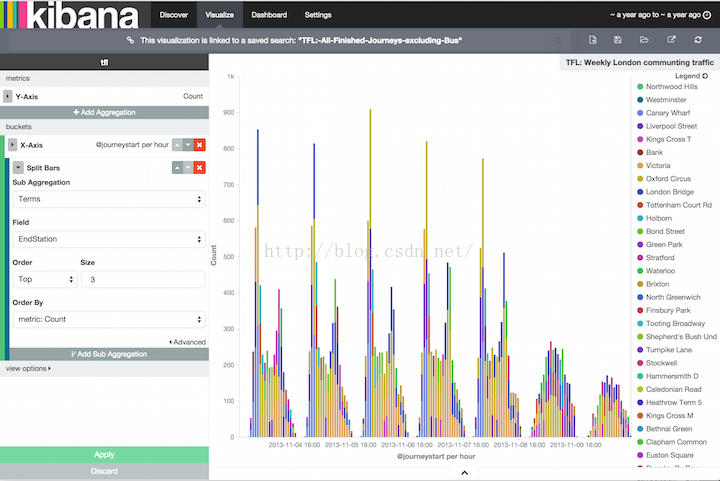
通过地铁完成的完整旅程

通过排除不完整的旅程和使用公交车完成的旅程。

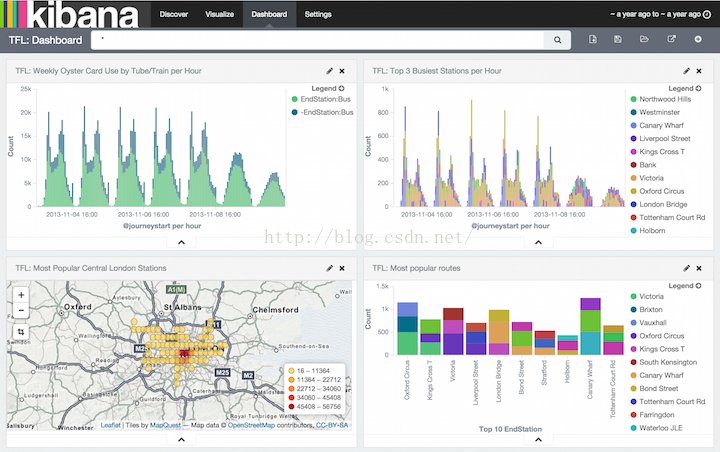


现在，我们能在柱状图中看出上班的早晚高峰。默认地，Discover页面显示前500条查询匹配到的实体，你能改变时间过滤器、交互柱状图来深入了解数据、查看某一文档的详细信息。更多关于在Discover页面探索发掘的你数据信息，查看Discover.章节。

你能够在Visualization页面构建查询结果的可视化，每一个可视化界面都是和一个查询一一对应的。比如，我们可以通过上一个查询来展示一个伦敦每周通过地铁上下班的柱状图，y轴表示旅程的数量，x轴显示星期和时间。通过追加一个子聚合，我们可以看到每小时top3的目的地站。

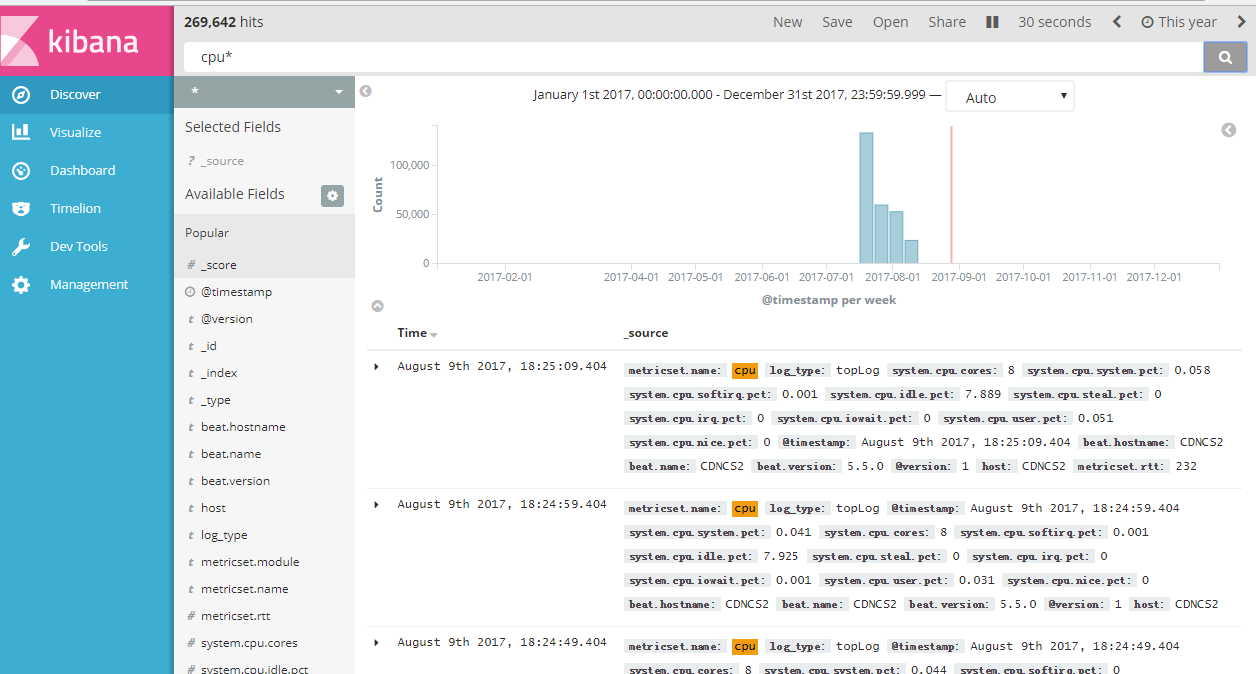


你可以保存和分享这些形象化的图表，并且能将之结合到dashboards中，使它更容易地和某些相关的信息相互关联。比如，我们可以创建一个dashboard用于展示一些伦敦交通局数据的图表。

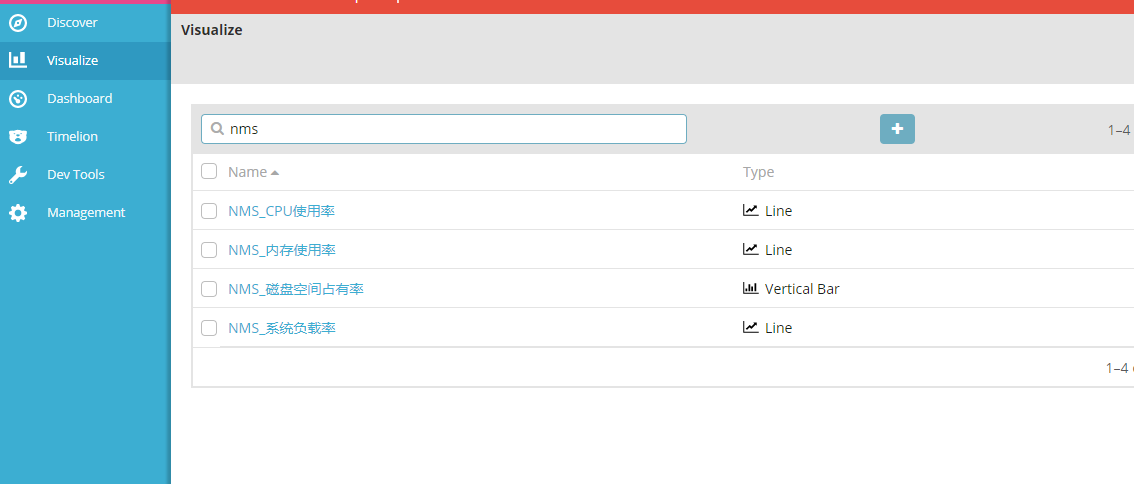


**12 实际操作图表**

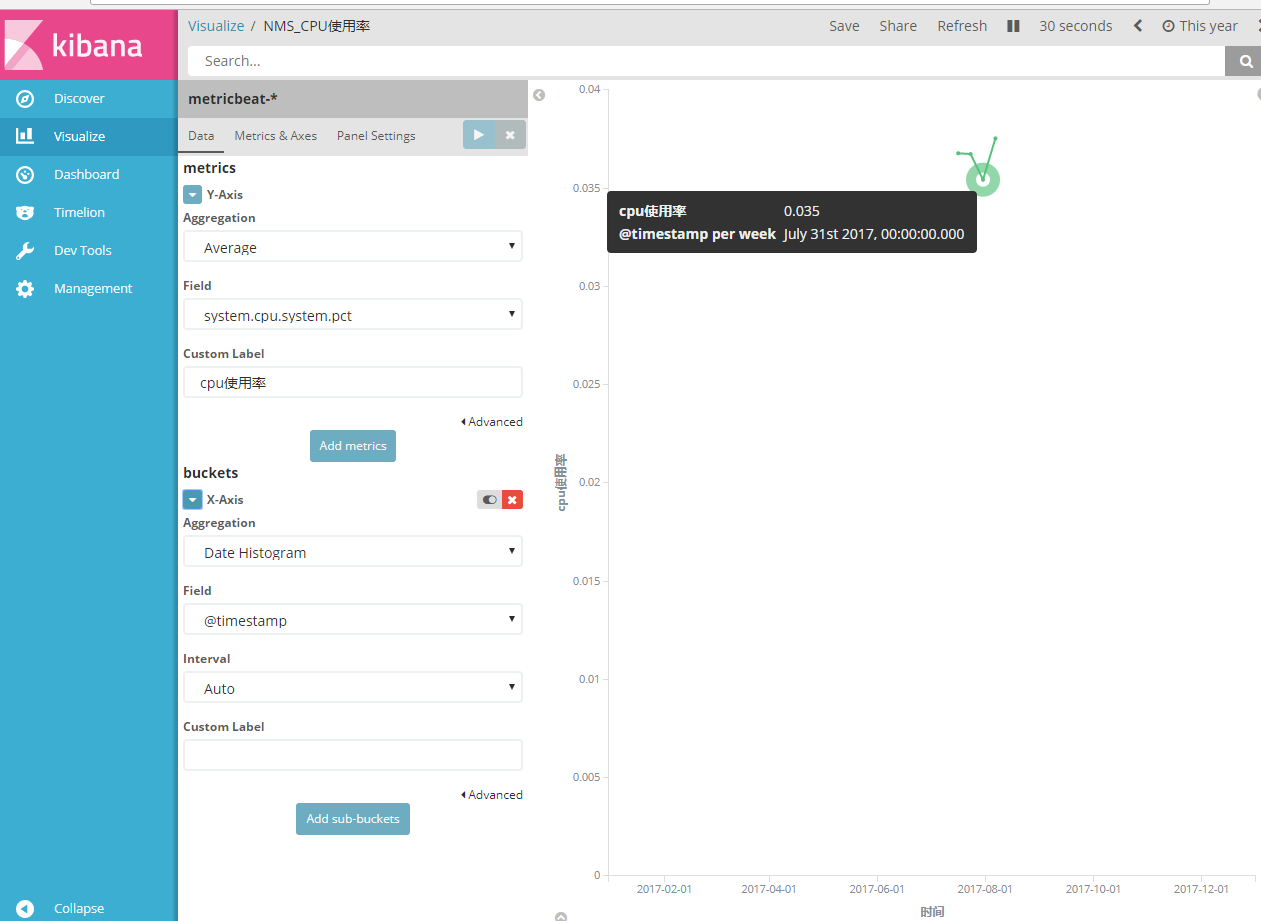




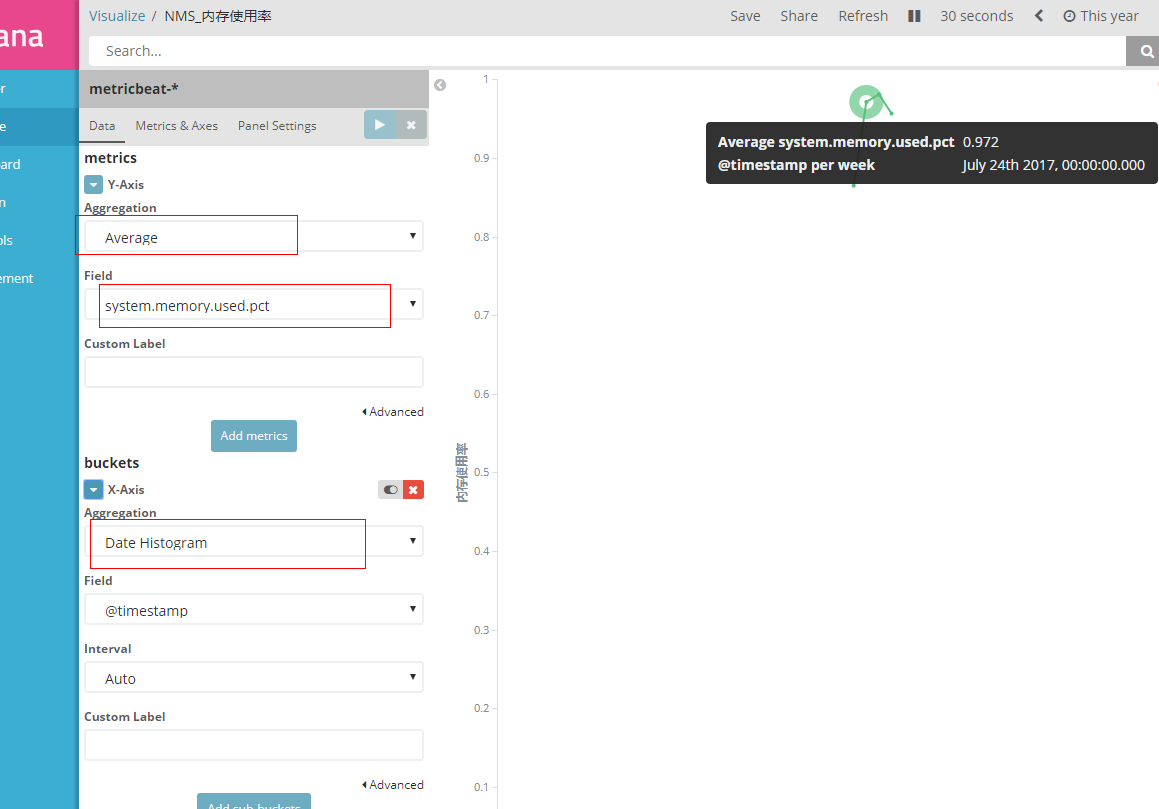
搜索数据



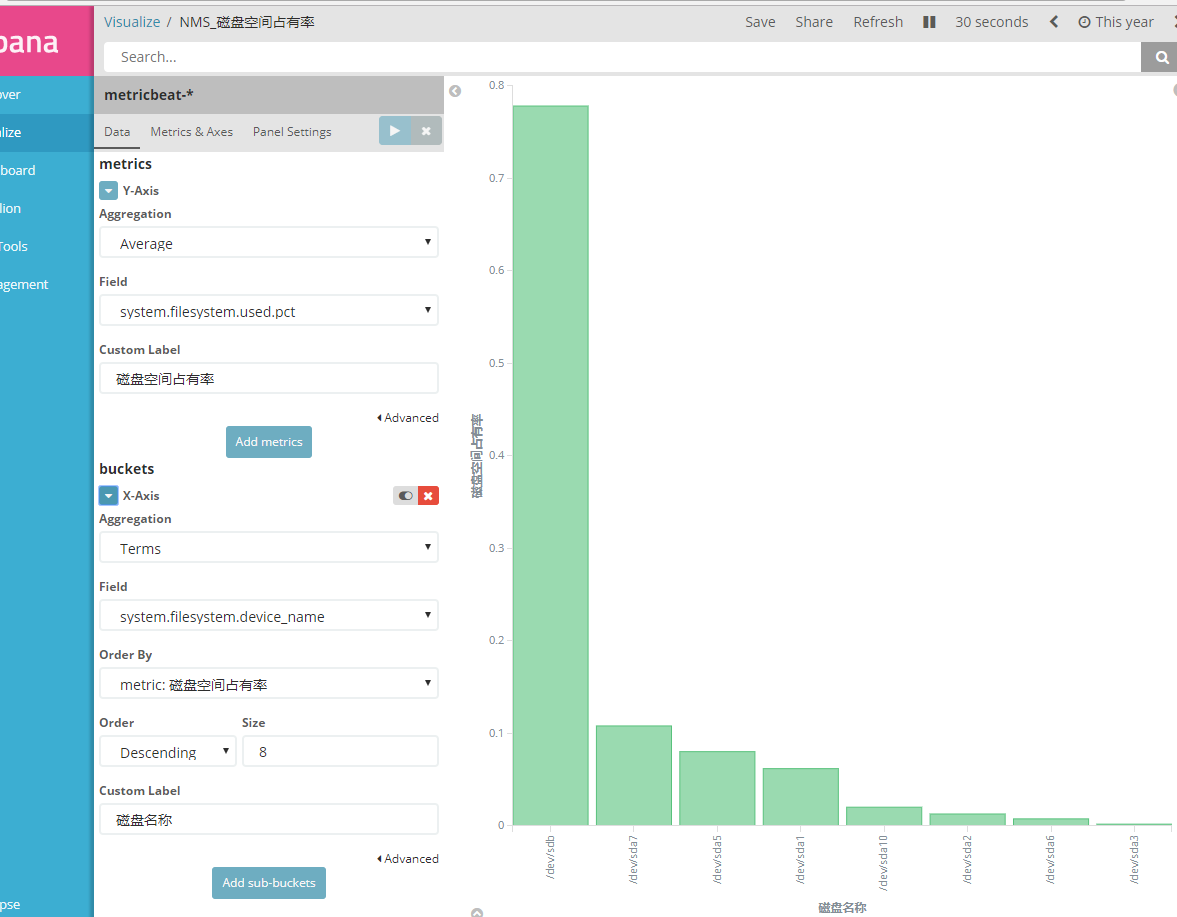
选择已有的视图

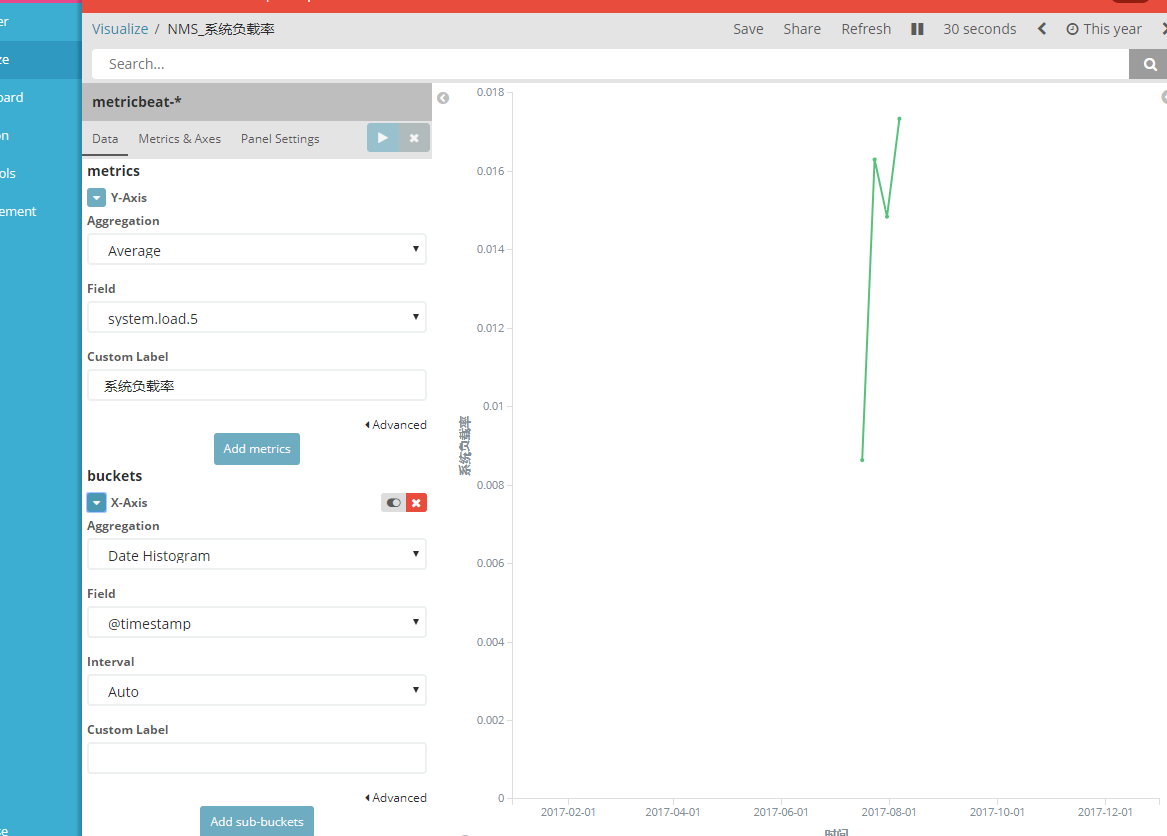


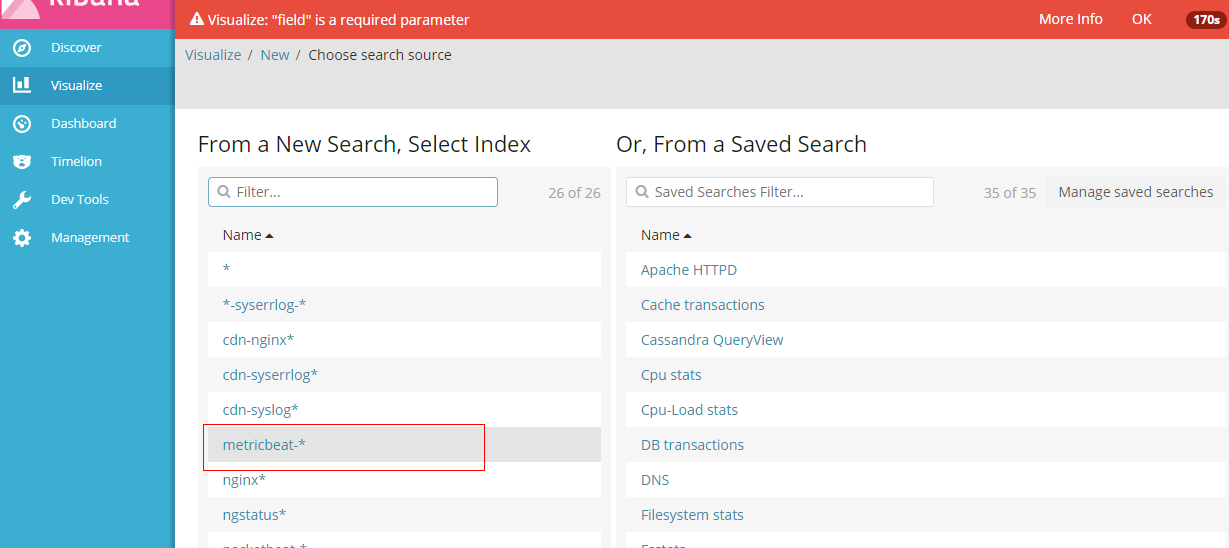
CPU使用率的line图

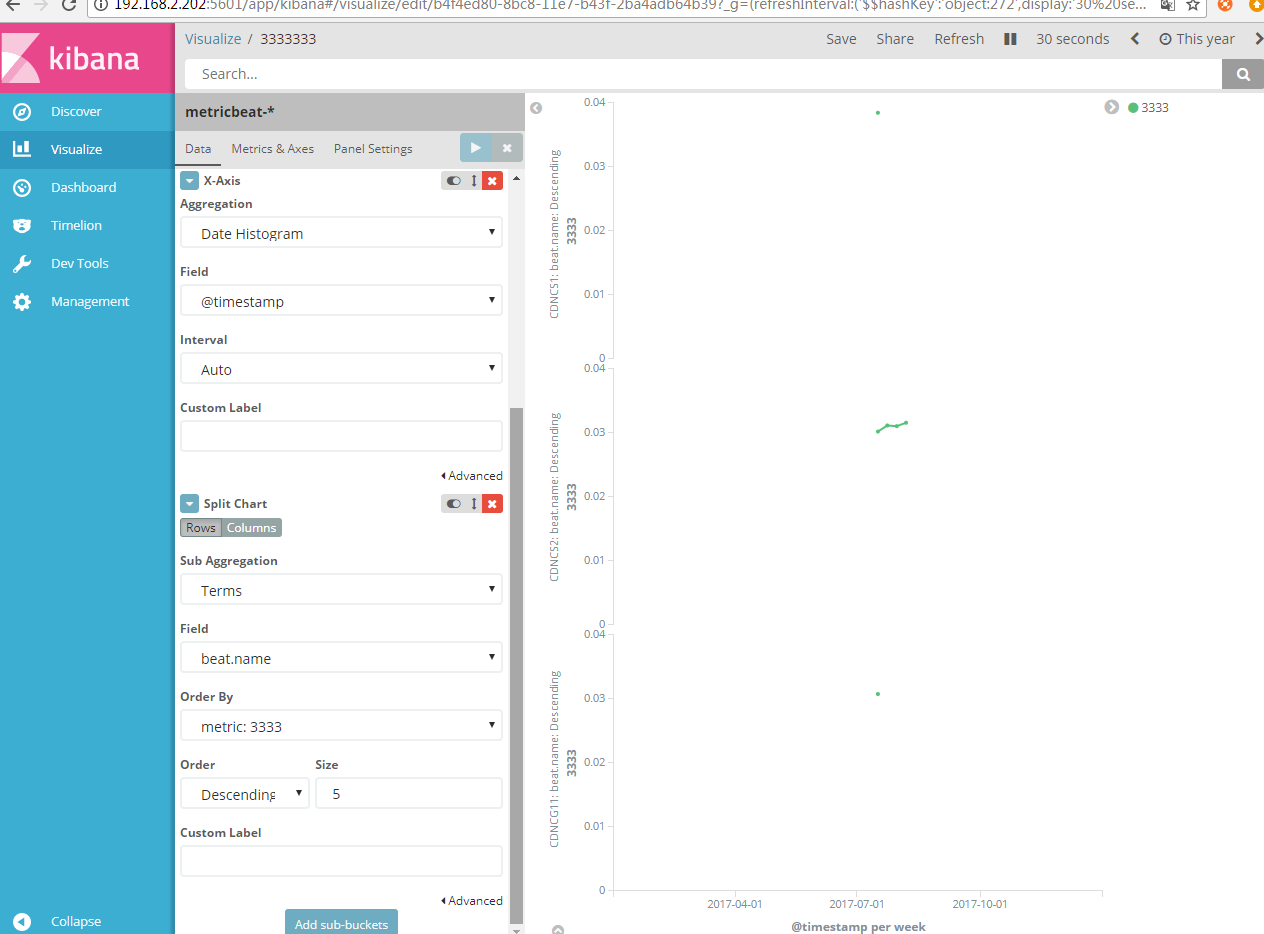


内存使用率





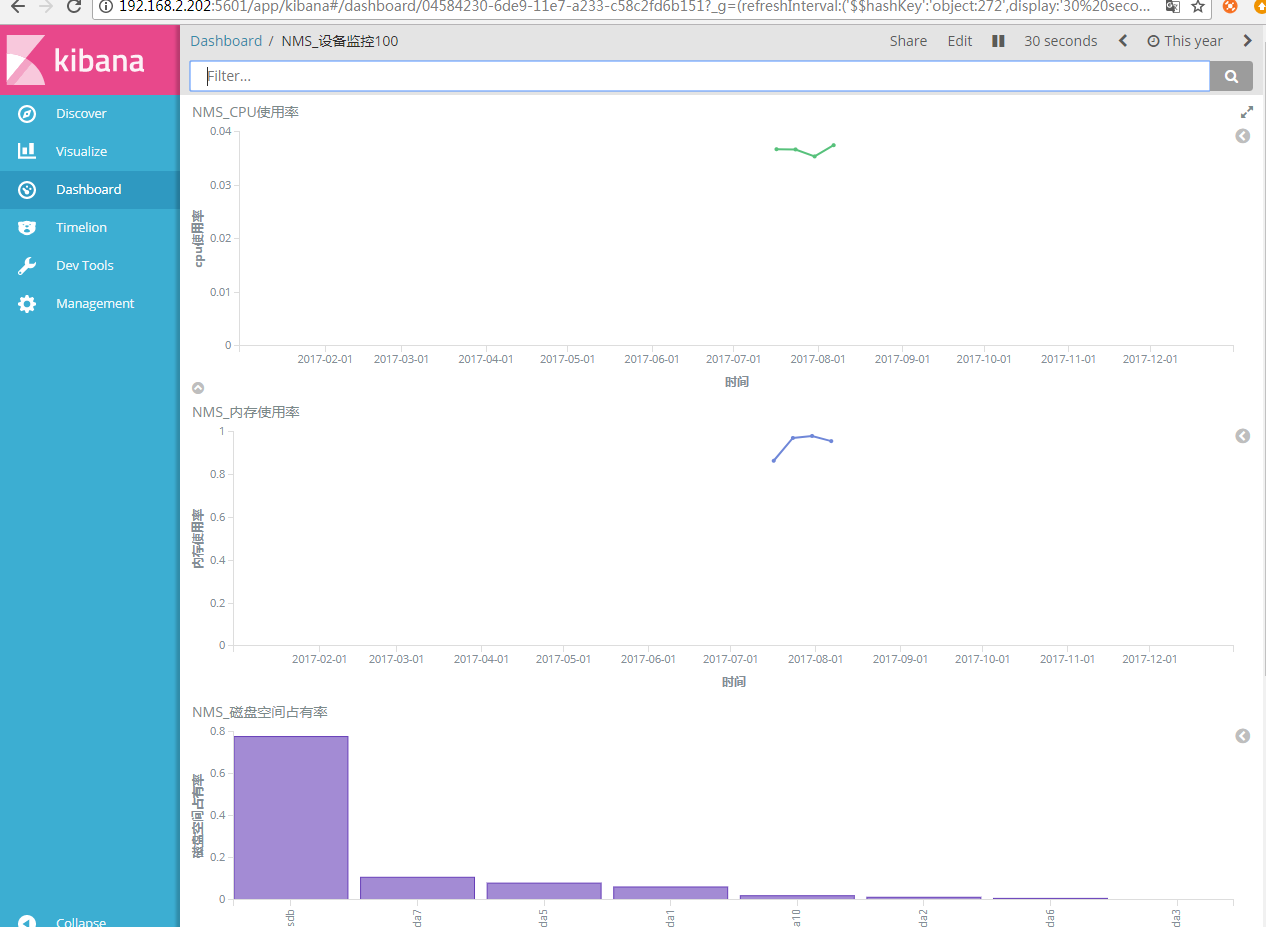




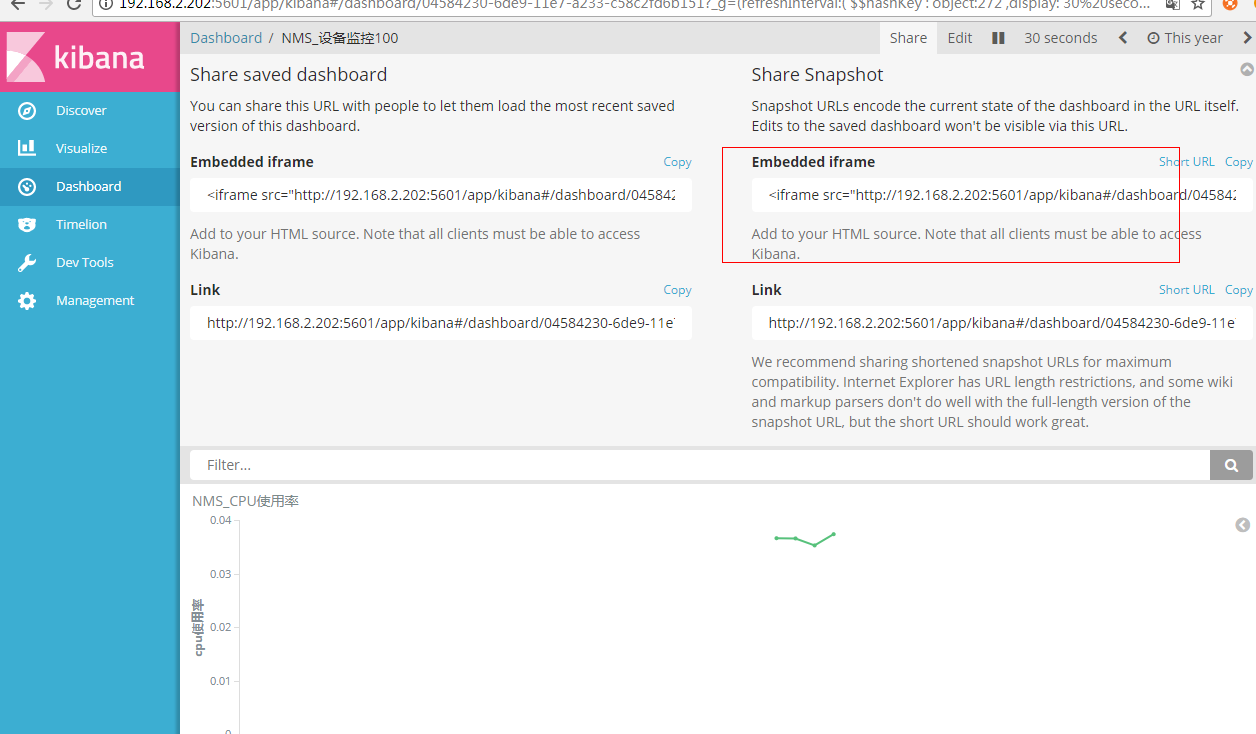
**X坐标是时间连续的片**

**Y坐标是CPU平均使用率**

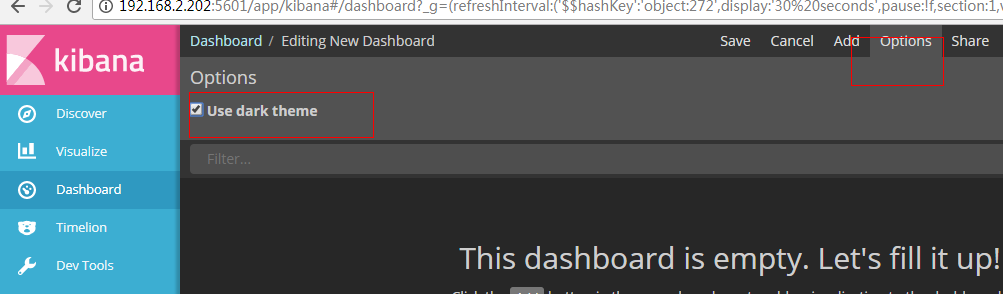
**以机器分组**



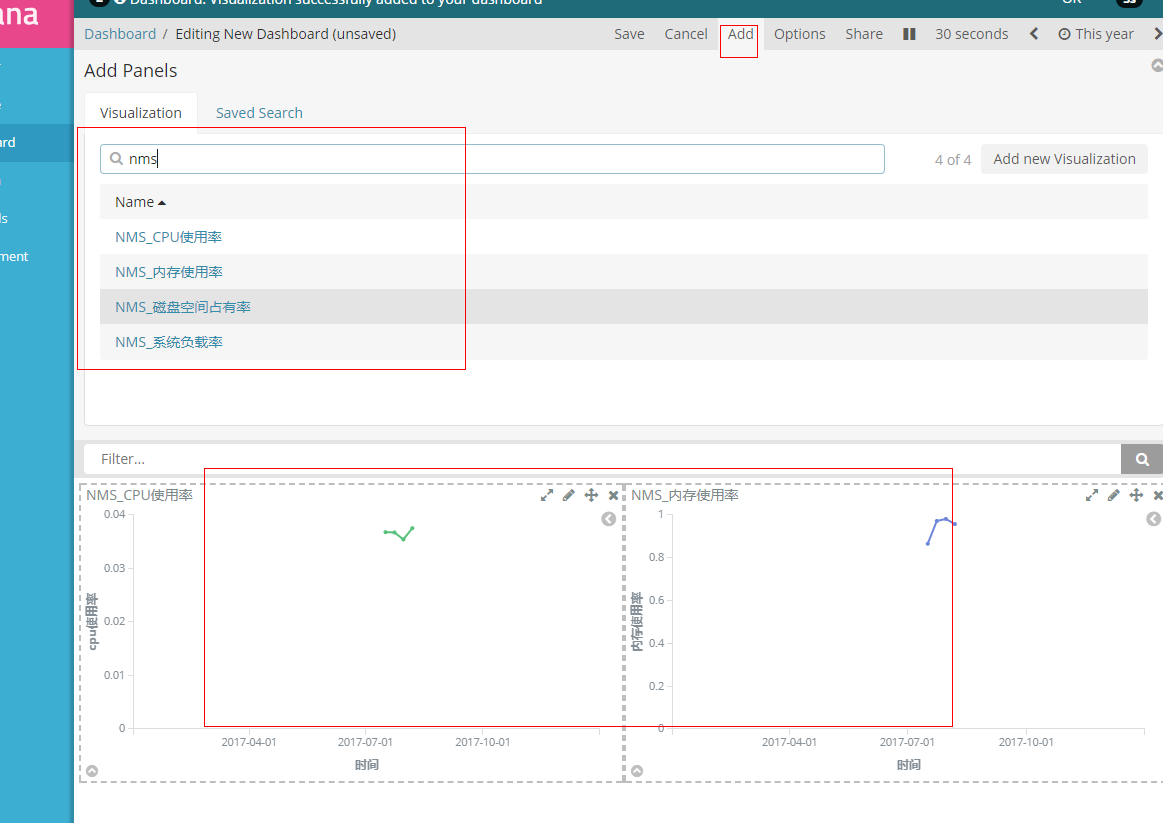
仪表盘包括4个视图，宽度100%，



嵌入页面的URL



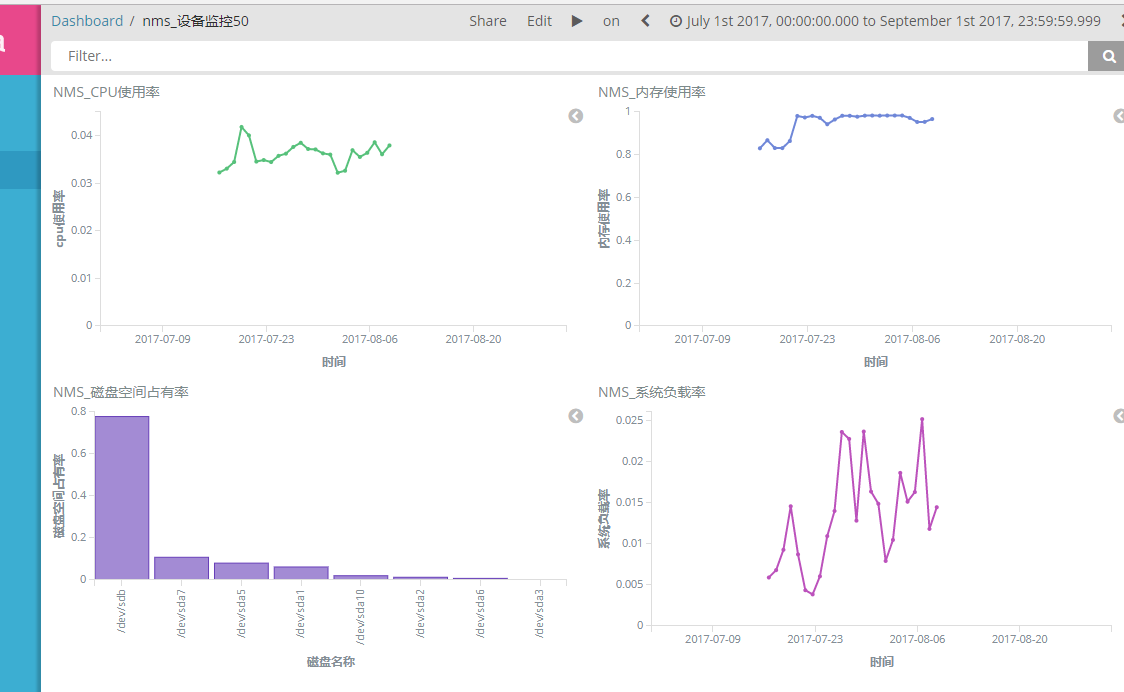
添加仪表盘第一步，选择 背景颜色



添加视图过程



调节宽度

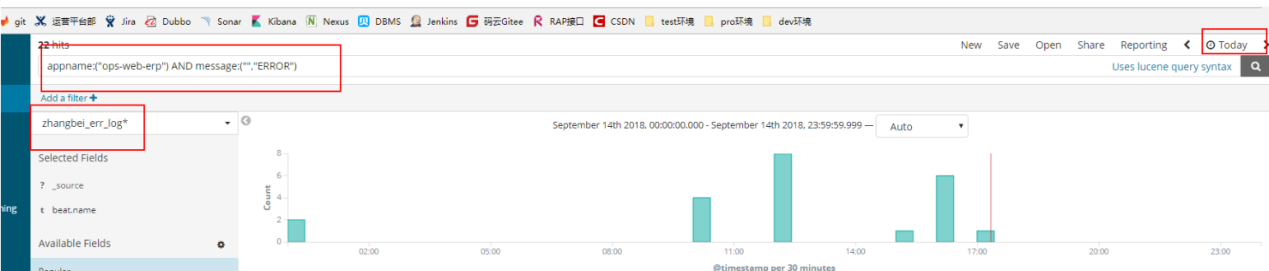


50%的比例

[http://elk.ops.yunnex.com](http://elk.ops.yunnex.com/)

账号：elastic/EelJMkJi3f3Sg8VB

 appname:("ops-web-erp") AND message:("","ERROR")  ,选择zhangbei\_err\_log\*, 选择 today ;  点击搜索



## 五、ELK Cooperation

**1安装完ELK不知从哪下手？**

拿到数据样本不知道怎么分解数据

导入到elasticsearch中奇怪为什么搜不出来

搜到结果后，不知道它还能干什么

本篇就以一个完整的流程介绍下，数据从 读取-->分析-->检索-->应用 的全流程处理。在阅读本篇之前，需要先安装ELK,可以参考之前整理安装文档：ELK5.0部署教程

**2 在利用ELK做数据分析时，大致为下面的流程：**

1） 基于logstash分解字段

2） 基于字段创建Mapping

3） 查看分词结果

4） 检索

5） 聚合

6） 高亮

**3 基于logstash分解字段**

在使用logstash前，需要对它有一定的了解。logstash的组件其实很简单，主要包括input、filter、output、codec四个部分。

**input 用于**读取内容，常用的有stdin(直接从控制台输入)、file(读取文件)等，另外还提供了对接redis、kafka等的插件

**filter 用于**对输入的文本进行处理，常用的有grok(基于正则表达式提取字段)、kv(解析键值对形式的数据)、csv、xml等，另外还提供了了一个ruby插件，这个插件如果会用的话，几乎是万能的。

**output 用于**把fitler得到的内容输出到指定的接收端，常用的自然是elasticsearch(对接ES)、file(输出到文件)、stdout(直接输出到控制台)

codec 它用于格式化对应的内容，可以再Input和output插件中使用，比如在output的stdout中使用rubydebug以json的形式输出到控制台

理解上面的内容后，再看看logstash的使用方法。

首先需要定义一个配置文件，配置文件中配置了对应的input,filter,output等，至少是一个input,output。

如我的配置文件：

input {

file {

path => "C:\Users\Documents\workspace\elk\page.csv"

start\_position => "beginning"

}

}

filter {

grok {

match => {

"message" => "%{NOTSPACE:url}\s\*%{NOTSPACE:date}\s\*%{NOTSPACE:pvs}\s\*%{NOTSPACE:uvs}\s\*%{NOTSPACE:ips}\s\*%{NOTSPACE:mems}\s\*%{NOTSPACE:new\_guests}\s\*%{NOTSPACE:quits}\s\*%{NOTSPACE:outs}\s\*%{NOTSPACE:stay\_time}"

}

}

}

output {

stdout{codec => dots}

elasticsearch {

document\_type => "test"

index => "page"

hosts => ["1.1.1.1:9200"]

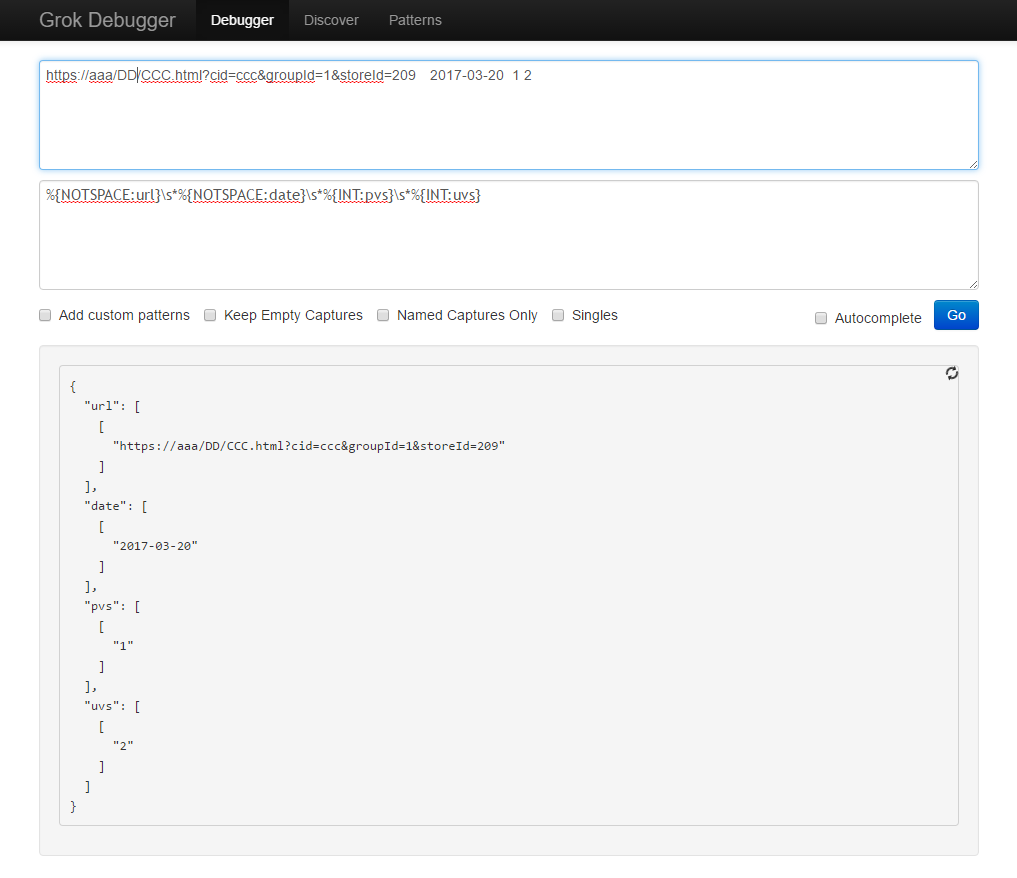
}

}

上面的配置最不容易理解的就是Grok,其实它就是个正则表达式而已，你可以把它理解成是一段正则表达式的占位。至于grok都有哪些关键字，这些关键字对应的正则都是什么，可以直接参考logstash的源码,目录的位置为：

logstash-5.2.2\vendor\bundle\jruby\1.9\gems\logstash-patterns-core-4.0.2\patterns

如果提供的话，可以直接在grokdebug上面进行测试：



另外一个技巧就是，如果开启stdout并且codec为rubydebug，会把数据输出到控制台，因此使用.代替，即可省略输出，又能检测到现在是否有数据正在处理。而且每个.是一个字符，如果把它输出到文件，也可以直接通过文件的大小，判断处理了多少条。

**4 基于字段创建Mapping**

虽然说Es是一个文档数据库，但是它也是有模式的概念的。文档中的每个字段仍然需要定义字段的类型，使用者经常会遇到明明是数字，在kibana却做不了加法；或者明明是IP，kibana里面却不认识。这都是因为Mapping有问题导致的。

在Elasticsearch中其实是有动态映射这个概念的，在字段第一次出现时，ES会自动检测你的字段是否属于数字或者日期或者IP，如果满足它预定义的格式，就按照特殊格式存储。一旦格式设置过了，之后的数据都会按照这种格式存储。举个例子，第一条数据进入ES时，字段检测为数值型；第二条进来的时候，却是一个字符串，结果可能插不进去，也可能插进去读不出来（不同版本处理的方式不同）。因此，我们需要事先就设定一下字段的Mapping,这样之后使用的时候才不会困惑。

另外，Mapping里面不仅仅有字段的类型，还有这个字段的分词方式，比如使用标准standard分词器，还是中文分词器，或者是自定义的分词器，这个也是很关键的一个概念，稍后再讲。

创建Mapping有两种方式：

**第一种，直接创建索引并创建映射**

创建索引时，可以直接指定它的配置和Mapping:

PUT index\_name

{

"settings" : {

"number\_of\_shards" : 1

},

"mappings" : {

"type\_name" : {

"properties" : {

"field\_name" : { "type" : "text" }

}

}

}

}

**第二种，先创建索引，再创建映射**

# 先创建索引

PUT index\_name

{}

# 然后创建Mapping

PUT /index\_name/\_mapping/type\_name

{

"properties": {

"ip": {

"type": "ip"

}

}

}

# 最后查询创建的Mapping

GET /index\_name/\_mapping/type\_name

比如我们上面的URL场景，可以这么建立索引：

PUT url/\_mapping/test

{

"properties": {

"url": {

"type": "string",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"date": {

"type": "date"

},

"pvs": {

"type": "integer"

},

"uvs": {

"type": "integer"

}

}

}

PS,在上面的例子中，url需要有两个用途，一个是作为聚合的字段；另一个是需要做全文检索。在ES中全文检索的字段是不能用来做聚合的，因此使用嵌套字段的方式，新增一个url.keyword字段，这个字段设置成keyword类型，不采用任何分词（这是5.0的新特性，如果使用以前版本，可以直接设置string对应的index属性即可）；然后本身的url字段则采用默认的标准分词器进行分词。这样，以后在搜索的时候可以直接以query string的方式检索url，聚合的时候则可以直接使用url.keyword

查看分词结果

如果字段为https://www.elastic.co/guide/en/elasticsearch/reference/5.2,使用standard标准分词器，输入elastic却收不到任何结果，是不是有点怀疑人生。

我们做个小例子，首先创建一个空的索引：

PUT test1/test1/1

{

"text":"https://www.elastic.co/guide/en/elasticsearch/reference/5.2"

}

然后查询这个字段被分解成了什么鬼？

GET /test1/test1/1/\_termvectors?fields=text

得到的内容如下：

{

"\_index": "test1",

"\_type": "test1",

"\_id": "1",

"\_version": 1,

"found": true,

"took": 1,

"term\_vectors": {

"text": {

"field\_statistics": {

"sum\_doc\_freq": 7,

"doc\_count": 1,

"sum\_ttf": 7

},

"terms": {

"5.2": {

"term\_freq": 1,

"tokens": [

{

"position": 6,

"start\_offset": 56,

"end\_offset": 59

}

]

},

"elasticsearch": {

"term\_freq": 1,

"tokens": [

{

"position": 4,

"start\_offset": 32,

"end\_offset": 45

}

]

},

"en": {

"term\_freq": 1,

"tokens": [

{

"position": 3,

"start\_offset": 29,

"end\_offset": 31

}

]

},

"guide": {

"term\_freq": 1,

"tokens": [

{

"position": 2,

"start\_offset": 23,

"end\_offset": 28

}

]

},

"https": {

"term\_freq": 1,

"tokens": [

{

"position": 0,

"start\_offset": 0,

"end\_offset": 5

}

]

},

"reference": {

"term\_freq": 1,

"tokens": [

{

"position": 5,

"start\_offset": 46,

"end\_offset": 55

}

]

},

"www.elastic.co": {

"term\_freq": 1,

"tokens": [

{

"position": 1,

"start\_offset": 8,

"end\_offset": 22

}

]

}

}

}

}

}

看到了吧，没有elastic这个词，自然是搜不出来的。如果你不理解这句话，回头看看倒排索引的原理吧！或者看看我的这篇文章:分词器的作用

那么你可能很郁闷，我就是要搜elastic怎么办！没关系，换个分词器就行了~比如elasticsearch为我们提供的simple分词器，就可以简单的按照符号进行切分:

POST \_analyze

{

"analyzer": "simple",

"text": "https://www.elastic.co/guide/en/elasticsearch/reference/5.2"

}

得到的结果为：

{

"tokens": [

{

"token": "https",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "www",

"start\_offset": 8,

"end\_offset": 11,

"type": "word",

"position": 1

},

{

"token": "elastic",

"start\_offset": 12,

"end\_offset": 19,

"type": "word",

"position": 2

},

{

"token": "co",

"start\_offset": 20,

"end\_offset": 22,

"type": "word",

"position": 3

},

{

"token": "guide",

"start\_offset": 23,

"end\_offset": 28,

"type": "word",

"position": 4

},

{

"token": "en",

"start\_offset": 29,

"end\_offset": 31,

"type": "word",

"position": 5

},

{

"token": "elasticsearch",

"start\_offset": 32,

"end\_offset": 45,

"type": "word",

"position": 6

},

{

"token": "reference",

"start\_offset": 46,

"end\_offset": 55,

"type": "word",

"position": 7

}

]

}

这样你就可以搜索elastic了，但是前提是需要在Mapping里面为该字段指定使用simple分词器，方法为:

PUT url/\_mapping/test

{

"properties": {

"url": {

"type": "string",

"analyzer": "simple",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"date": {

"type": "date"

},

"pvs": {

"type": "integer"

},

"uvs": {

"type": "integer"

}

}

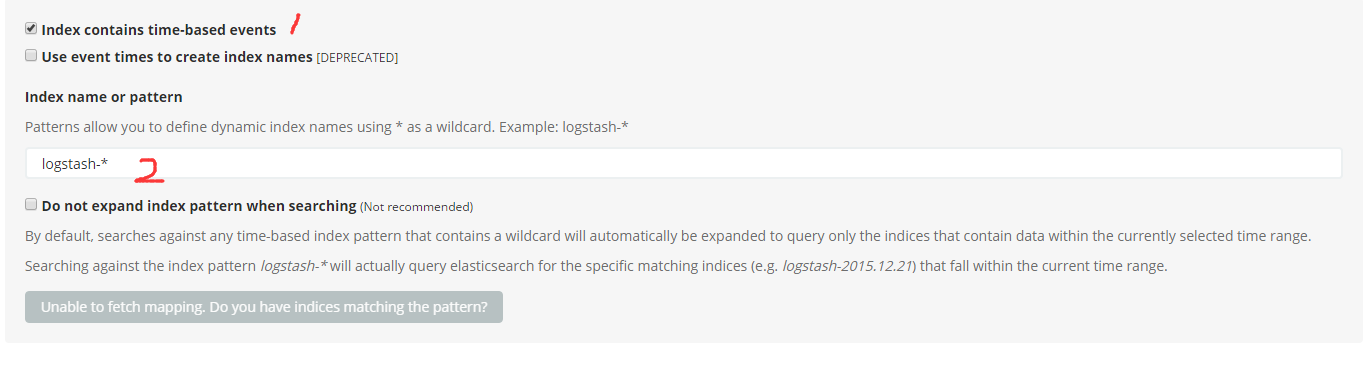
修改Mapping前，需要先删除索引，然后重建索引。删除索引的命令为：

DELETE url

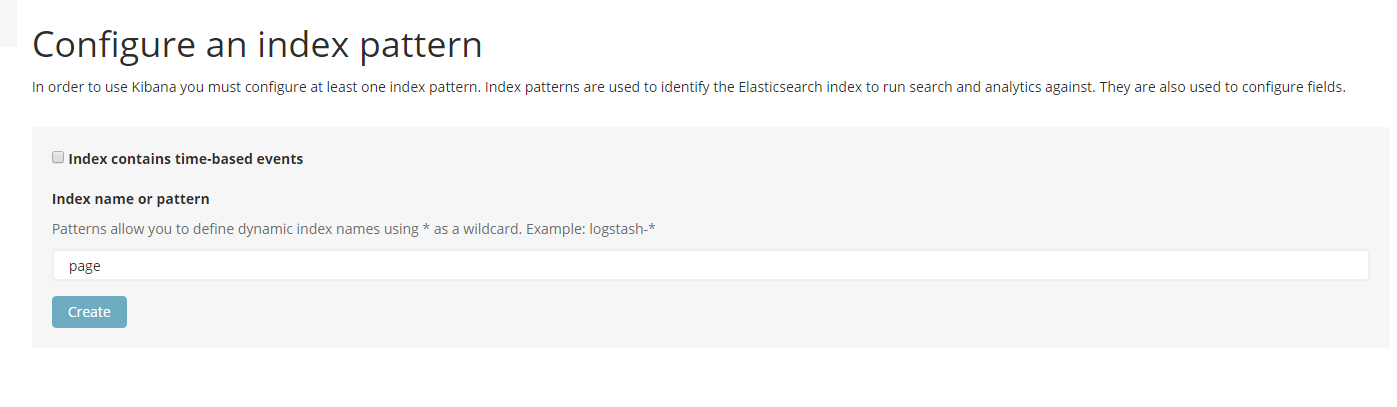
不想删除索引，只想改变Mapping？想得美....你当ES是孙悟空会72变？不过，你可以创建一个新的索引，然后把旧索引的数据导入到新索引就行了，这也不失为一种办法。如果想这么搞，可以参考reindex api，如果版本是5.0之前，那么你倒霉了！自己搞定吧！

**5 检索**

ES里面检索是一个最基础的功能了，很多人其实这个都是一知半解。由于内容太多，我就结合Kibana讲讲其中的一小部分吧。很多人安装完kibana之后，登陆后不知道该干啥。如果你的elasticsearch里面已经有数据了，那么此时你需要在Kiban新建对应的索引。



如果你的es的索引是name-2017-03-19,name-2017-03-20这种名字+时间后缀的，那么可以勾选1位置的选项，它会自动聚合这些索引。这样在这一个索引中就可以查询多个索引的数据了，其实他是利用了索引的模式匹配的特性。如果你的索引仅仅是一个简单的名字，那么可以不勾选1位置的选项，直接输入名字，即可。



然后进入Kibana的首页，在输入框里面就可以任意输入关键字进行查询了。

查询的词，需要是上面\_termvectors分析出来的词，差一个字母都不行！！！！！

这个搜索框其实就是elasticsearch中的query string,因此所有的lucene查询语法都是支持的！如果想要了解更多的查询语法，也可以参考我之前整理的文章，Lucene查询语法

另外，这个输入框，其实也可以输入ES的DSL查询语法，只不过写法过于蛋疼，就不推荐了。

**6 自定义查询语法**

如果不使用kibana，想在自己的程序里面访问es操作，也可以直接以rest api的方式查询。

比如查询某个索引的全部内容，默认返回10个:

GET /page/test/\_search?pretty

再比如，增加一个特殊点的查询:

GET /page/test/\_search?pretty

{

"query": {

"query\_string" : {

"default\_field" : "url",

"query" : "颜色"

}

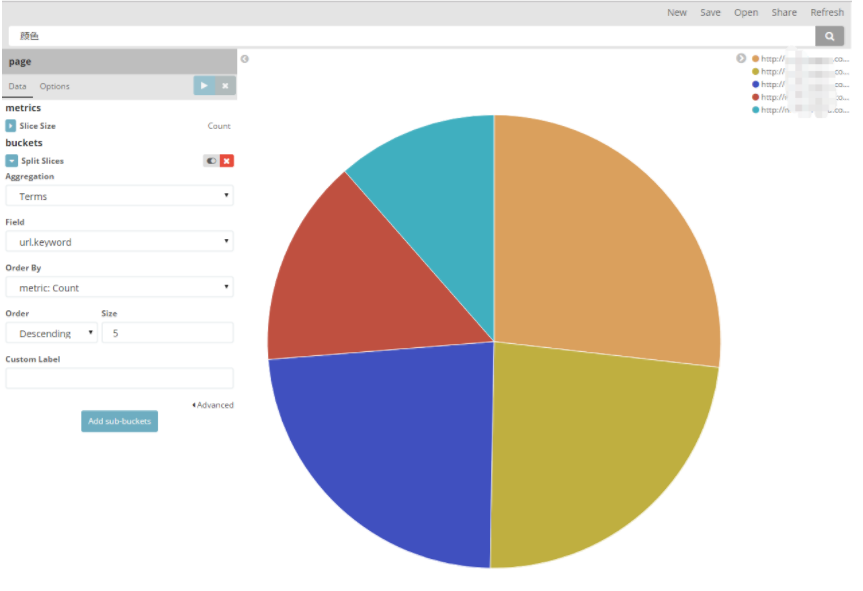
},

"size": 10,

}

**7 聚合**

在es中一个很重要的亮点，就是支持很多的聚合语法，如果没有它，我想很多人会直接使用lucene吧。在ES中的聚合，大体上可以为两类聚合方法，metric和bucket。metic可以理解成avg、sum、count、max、min，bucket可以理解为group by 。有了这两种聚合方法，就可以对ES中的数据做很多处理了。比如在kibana中，做一个最简单的饼图：



其实它在后台发送的请求，就是这个样子的:

{

"size": 0,

"query": {

"query\_string": {

"query": "颜色",

"analyze\_wildcard": true

}

},

"\_source": {

"excludes": []

},

"aggs": {

"2": {

"terms": {

"field": "url.keyword",

"size": 5,

"order": {

"\_count": "desc"

}

}

}

}

}

如果不适用kibana，自己定义聚合请求，那么可以这样写：

GET /page/test/\_search?pretty

{

"query": {

"query\_string" : {

"default\_field" : "url",

"query" : "颜色"

}

},

"size": 0,

"aggs" : {

"agg1" : {

"terms" : {

"field" : "url.keyword",

"size" : 10

},

"aggs" : {

"pvs" : { "sum" : { "field" : "pvs" } },

"uvs" : { "sum" : { "field" : "uvs" } }

}

}

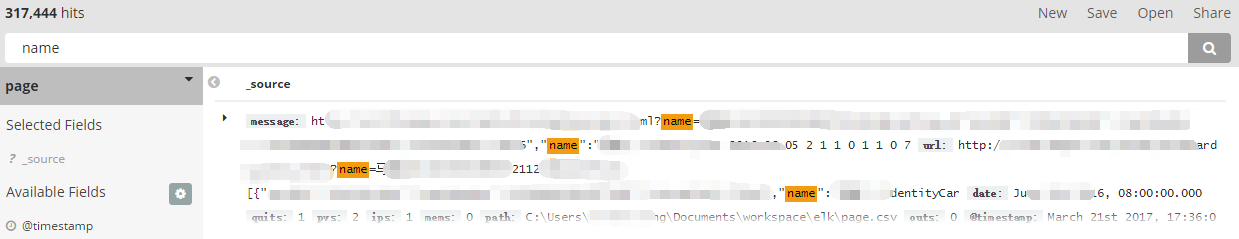
}

}

另外，聚合也支持嵌套聚合，就是跟terms或者sum等agg并列写一个新的aggs对象就行。

**8 高亮**

如果是自己使用elasticsearch，高亮也是一个非常重要的内容，它可以帮助最后的使用者快速了解搜索的结果。



后台的原理，是利用ES提供的highlight API，针对搜索的关键字，返回对应的字段。该字段中包含了一个自定义的标签，前端可以基于这个标签高亮着色。

举个简单的例子：

GET /\_search

{

"query" : {

"match": { "content": "kimchy" }

},

"highlight" : {

"fields" : {

"content" : {}

}

}

}

上面的请求会针对content字段搜索kimchy。并且返回对应的字段，比如原来的字段内容时hello kimchy，经过高亮后，会再搜索结果的hits中返回:

{

"took": 3,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 30,

"max\_score": 13.945707,

"hits": [

{

"\_index": "page",

"\_type": "test",

"\_id": "AVrvHh\_kvobeDQC6Q5Sg",

"\_score": 13.945707,

"\_source": {

"date": "2016-03-14",

"pvs": "3",

"url": "hello kimchy",

"@timestamp": "2017-03-21T04:29:07.187Z",

"uvs": "1",

"@version": "1"

},

"highlight": {

"url": [

"hello <em>kimchy</em>"

]

}

}

]

}

}

这样就可以直接利用highlight中的字段做前端的显示了。

另外，上面的<em>标签可以自定义,比如：

GET /\_search

{

"query" : {

"match": { "user": "kimchy" }

},

"highlight" : {

"pre\_tags" : ["<tag1>"],

"post\_tags" : ["</tag1>"],

"fields" : {

"\_all" : {}

}

}

}

**9 搭建一个简单的例子**

**（1）搭建一个日志系统**

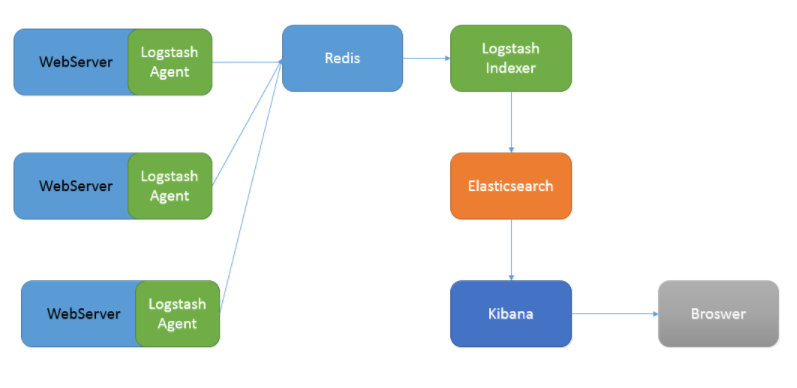
在微服务开发过程中，一般都会利用多台服务器做分布式部署，如何能够把分散在各个服务器中的日志归集起来做分析处理，是一个微服务服务需要考虑的一个因素。搭建一个日志系统需要考虑一下一些因素：利用什么技术，是自己实现还利用现成的组件；日志需要定义统一的格式；日志需要拥有一个锚点来进行全局跟踪

第一个问题，针对我们小公司来说，基本没有自己的研发能力，绝对是选用第三方开源的组件了。ELK配置比较简单，有现成的UI界面，容易检索日志信息，是首选。

第二个问题，利用log4j2定义好统一的日志格式，利用logstash过滤日志内容。

第三个问题，全局跟踪的ID有几种生产方式，一种是利用UUID或者生成随机数，一种是利用数据库来生成sequence number，还可以通过自定义一个id生成服务来获取。考虑到自身服务的需要，这里选用生成随机数来实现。

**（2）日志系统架构**



从左边看起，每一台webserver上都会部署一个logstash-agent,它的作用是用类似tailf的方式监听日志文件，然后把新添加的日志发送到redis队列里面，logstash-indexer负责从redis相应的队列里面取出日志，对日志进进行加工后输出到elasticsearch中，elasticsearch会根据要求对日志进行索引归集，最后用户可以通过kibana来查看和分析日志。

**（3）准备工作**

@安装JDK8

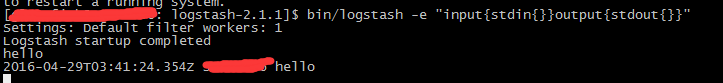
@安装Redis2.8

**（4）在webserver上安装logstash-agent**

@.解压 tar -zxvf logstash2.3.1

@.检验安装是否成功 bin/logstash -e "input{stdin{}}output{stdout{}}",然后在终端输入hello，看看有没有内容返回

安装logstash



@.编写配置文件logstash\_agent.conf

在logstash安装目录下新建conf文件夹，在里面新建配置文件logstash\_agent.conf

input {

file {

type => "customer\_service"

#需要收集的日志文件

path => ["/home/java/logs/cust/customer-service-\*.log"]

tags => ["customer-service", "log4j2"]

#

codec => multiline { #

pattern => "^%{TIMESTAMP\_ISO8601}"

negate => true

what => "previous"

}

}

}

output {

redis {

host => "192.168.235.195"

data\_type => "list"

key => "logstash:redis:customer"

}

}

@.后台启动

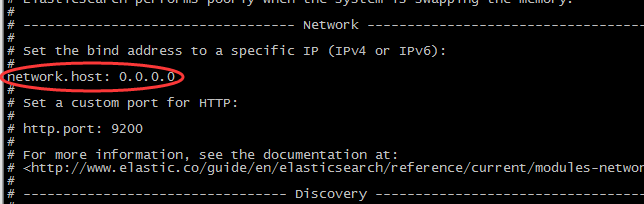
**nohup ./bin/logstash -f conf/logstash\_agent.conf &**

**（5）在日志server上安装elasticsearch**

1）.解压tar -zxvf elasticsearch-2.3.1.tar.gz

2）.修改安装目录下的配置文件config/elasticsearch.yml

把network.host字段给反注释掉，把地址改为0.0.0.0（官方并没明确说要去改这配置，默认配置应该就可以了，不过实测的时候发现如果不做这修改，elasticsearch访问不了）



3）.后台启动elasticsearch

nohup ./bin/elasticsearch &

**（6）在日志server上安装logstash-indexer**

与在webserver上安装logstash-agent一节基本相同，只是配置文件不一样，这里使用logstash-indexer.conf

input {

redis {

host => "localhost" #redis地址

data\_type => "list"

key => "logstash:redis:customer"

type => "redis-input"

}

}

filter {

grok {

match => {

"message" => "%{TIMESTAMP\_ISO8601:date} \[(?<thread\_name>.+?)\] (?<log\_level>\w+)\s\*(?<content>.\*)"

}

}

date {

match => ["timestamp", "dd/MMM/YYYY:HH:mm:ss Z"]

}

}

output {

if[type] == "customer\_service" {

elasticsearch {

#embedded => false

#protocol => "http"

hosts => "localhost:9200"

index => "customer-%{+YYYY.MM.dd}"

}

}else if[type] == "其他类型" {

elasticsearch {

#embedded => false

#protocol => "http"

hosts => "localhost:9200"

index => "其他类型索引名字-%{+YYYY.MM.dd}"

}

}else {

//做其他处理

}

stdout{ #输出到标准输出，可以去掉

}

}

启动logstash-indexer

**nohup ./bin/logstash -f conf/logstash-indexer.conf &**

**（7）在日志server上安装kibana**

1）.解压tar -zxvf kibana-4.5.0-linux-x64.gz

2）.后台启动kibana nohup ./bin/kibana &

这里没有特别去配置kibana要访问的elasticsearch地址，默认它会找本地的，如果需要，可以到conf/kibana.yml修改

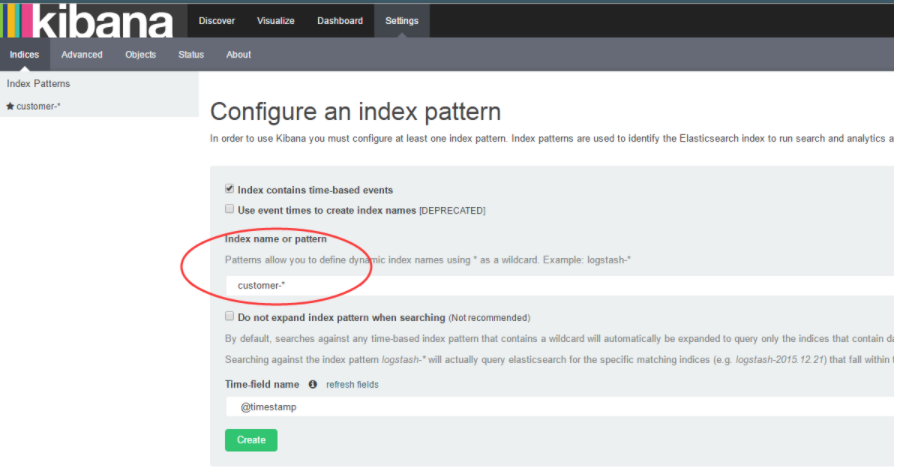
配置log4j2的日志格式

<PatternLayout pattern="%d{ISO8601} %t %level %msg%n"/>

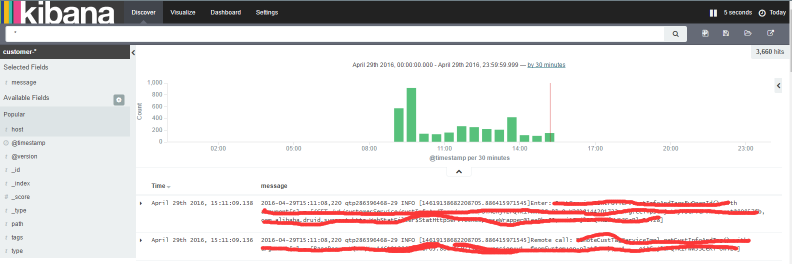
访问kibana主页

1）.打开kibana主页http://your-kibana-ip:5601/

2）.输入索引customer-\*，并点击create



3）.在Discover页面观察日志



**（8）ToDo:**

需要了解elasticsearch日志存放在哪里，定期清理日志

清理Elasticsearch索引

索引放久了需要清理，清理所以可以使用Elasticsearch的API。

例如我需要删除某个索引2016年5月份的所有索引，可以使用下面的命令：44

**grep -rn "CDNCS1" \* (包含CDNCS1当前目录所有文件)**

\* : 表示当前目录所有文件，也可以是某个文件名

-r 是递归查找

-n 是显示行号

-R 查找所有文件包含子目录

-i 忽略大小写

**find / -name 'nohup.out'**

**hostname修改文件：**

**sysconfig/network:2:HOSTNAME=CDNCS1**

**sysconfig/networking/profiles/default/network:1:HOSTNAME=CDNCS1**

**/usr/local/mysql-tomcat-cms-8081/logs/debug**

**tail -f -n 2000 LOG.LOG**

**netstat -ano**

**netstat -aon|findstr "8080"**

**tasklist|findstr "1220"**

**taskkill /IM java.exe**

**TASKKILL /PID 1230**

**TASKKILL /F(强制结束) /IM（镜像名称） java.exe /T(子进程)**

**netstat -anp|grep 8081**

**ps -ef|grep 12006**

**kill -9 8978**

**ps -ef|grep iscg**

**kill -9 8978**

**ps -ef |grep 5601 |awk '{print $2}'|xargs kill -9**

**击杀动态进程**

**fuser -n tcp 5601**

**kill -9 XXXX**

**./bin/kibana**

**vi rrr.txt**

**esc insert**

**:wq**

**:q!**

**elasticsearch:11.158,11.159**

**logstash:11.158**

**filebeat:11.158**

**kibana:11.158**

**es: http://192.168.11.158:9200/\_cat/health**

**logstash:http://192.168.11.158:9600**

**kibana:http://192.168.11.158:5601**

**./bin/elasticsearch**

**./bin/kibana**

**./bin/logstash -f logstash.conf**

**./filebeat -e -c filebeat.yml**

**10 在ES5.0安装 X-PACK**

bin/elasticsearch-plugin install x-pack

注意需要关闭ES服务器，如果已经启动的话，否则安装失败。

安装成功之后再启动ES服务器即可

如果不需要该插件，也可以通过非常简单的方式删除：

bin/elasticsearch-plugin

remove x-pack

重启ElasticSearch

bin/kibana-plugin

remove x-pack

重启Kibana

以上安装之后，就会发现ES的访问和Kibana的访问都受到限制，那么默认的用户名和密码是多少呢

ElasticSearch：

用户名：elastic

密码：changeme

**adduser avit002**

**passwd avit002**

**chown -R avit002 elasticsearch-5.4.2**

**http://grokdebug.herokuapp.com/?#**

**192.168.2.203 GET /index.html 15824 0.043**

**%{IPORHOST:clientip}\s%{WORD:method}\s%{URIPATHPARAM:request}\s%{NUMBER:bytes}\s%{NUMBER:duration}**

**ES权威：**

**https://fuxiaopang.gitbooks.io/learnelasticsearch/getting\_started/what\_is\_it.html**

**kibana:2.202**

**logstash:2.202,2.203**

**elasticsearch:2.202,2.203,2.212**

**metricbeat:11.67,11.68,6.253**

**filebeat:11.67,6.253**

**packetbeat:11.67**

**hadoop:192.168.2.182**

tar zxvf logstash-1.4.2.tar.gz

cd logstash-1.4.2

现在我们来运行一下：

bin/logstash -e 'input { stdin { } } output { stdout {} }'

查看网卡信息 ifconfig

增加网卡：

ifconfig eth0:c 192.168.2.190 netmask 255.255.255.0 up

ifconfig eth0:z 192.168.2.191 netmask 255.255.255.0 up

ifconfig eth0:j 192.168.2.192 netmask 255.255.255.0 up

#ifconfg eth0:0 down

#ifconfg eth0:1 down

#ifconfg eth0:2 down

ip地址：/etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=eth0

TYPE=Ethernet

UUID=b67469db-4107-4209-9bb6-5f8e2005d8e7

ONBOOT=yes

NM\_CONTROLLED=yes

BOOTPROTO=none

HWADDR=00:0C:29:23:90:45

IPADDR=192.168.11.158

PREFIX=24

GATEWAY=192.168.11.1

DEFROUTE=yes

IPV4\_FAILURE\_FATAL=yes

IPV6INIT=no

NAME="System eth0"

**重启网络服务：service network restart**

## 六、Metricbeat

**1 Metricbeat是elastic下的项目**

在5.1及之后的版本中支持对Docker的监控，需与EK配合使用能在界面上显示，也可直接将数据导入kafka中。

**2 Metricbeat安装**

@使用版本:

elasticsearch-5.2.0-1.noarch（用于输出显示）

kibana-5.2.0-1.x86\_64（用于输出显示）

metricbeat-5.2.0-x86\_64

操作系统是centos7

Java：yum -y installjava-1.8.0-openjdk （elasticsearch 使用Java，centos自带yum源即可安装）

@配置yum源

name=Elasticsearch repository for 5.xpackages

baseurl=https://artifacts.elastic.co/packages/5.x/yum

gpgcheck=1

gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch

enabled=1

autorefresh=1

type=rpm-md

@安装elasticsearch和kibana：

yum -y install elasticsearch kibana

@下载metric rpm包：

curl -L -Ohttps://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-5.2.0-x86\_64.rpm

下载后直接用rpm安装。

**3 .配置文件**

**（1）配置elasticsearch：**

vi /etc/elasticsearch/elasticsearch.yml

network.host: 192.168.1.100

http.port: 9200

**（2）配置kibana：**

vi /etc/kibana/kibana.yml

server.port: 5601

server.host: "192.168.1.100"

**（3）配置metricbeat：**

vi /etc/metricbeat/metricbeat.yml

- module: docker

metricsets: ["cpu", "info", "memory","network", "diskio", "Container"]

hosts:["unix:///var/run/docker.sock"]

enabled: true

period: 1s

model：要监控的模块

metricset：监控指标，要保证是模块能够提供的

enabled：是否使能对该模块的监控，默认true

period：每次执行取数据的时间间隔，如果系统无效，则每个周期会返回一个错误信息

hosts：从哪些主机列表中取数据（不是所有模块都需要配置，如system）

field：数据的附加信息，会在输出文档中进行分组，字典格式配置

fields: {project: “myproject”, instance-id: “574734885120952459”}

tag：为数据做标记，便于搜索

tags: [“my-service”, “hardware”, “test”]

metricbeat支持的model有:

Apache,couchbase,docker,HAProxy,kafka,MongoDB,MySQL,Nginx,PostgreSQL,Prometheus,Redis,System,ZooKeeper

默认model是system,在后面追加docker部分即可,metricsets设置要监控的项。

System可监控的项为:

Core,cpu,diskio,filesystem,fsstat,load,memory,network,process,socket

Core:提供每个cpu的load statistics

Fsstat:所有文件系统的信息统计.

Docker可监控的项为:

Container,cpu,diskio,info,memory,network

Container:运行状态的容器信息。

Output：

Elasticsearch；Logstash；kafka；redis；file；console，输出到文件配置如下：

File：

output.file:

path: "/tmp/metricbeat"

filename: metricbeat

#rotate\_every\_kb: 10000

#number\_of\_files: 7

**4 启动服务**

systemctl enable elasticsearch

systemctl start elasticsearch

systemctl enable kibana

systemctl start kibana

systemctl enable metricbeat

systemctl start metricbeat

检查Elasticsearch是否正常运行：

curl -XGET '192.168.1.100:9200/?pretty'

{

"name" : "ZAWwiQz",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "PoBTufBIRSuql1ZidCiWUA",

"version" : {

"number" : "5.2.0",

"build\_hash" : "24e05b9",

"build\_date" : "2017-01-24T19:52:35.800Z",

"build\_snapshot" : false,

"lucene\_version" : "6.4.0"

},

"tagline" : "You Know, for Search"

}

检查metricbeat是否正常运行：

curl -XGET'http://192.168.1.100:9200/metricbeat-\*/\_search?pretty'

{

"took" : 9,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

}

**5 加载kibana**

cd /usr/share/metricbeat&& ./scripts/import\_dashboards -es http://192.168.1.100:9200

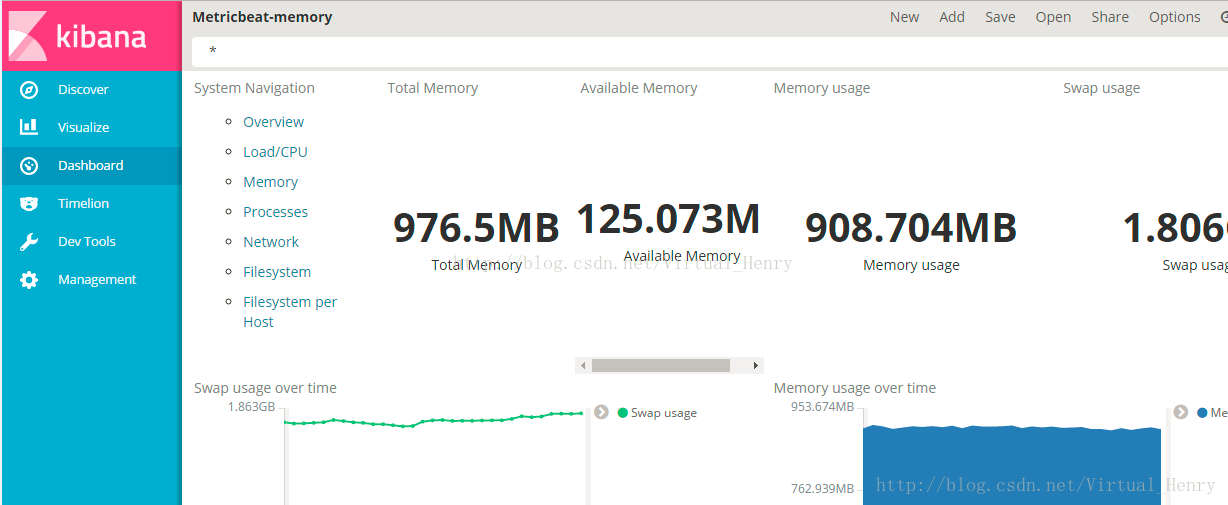
**6 显示输出**

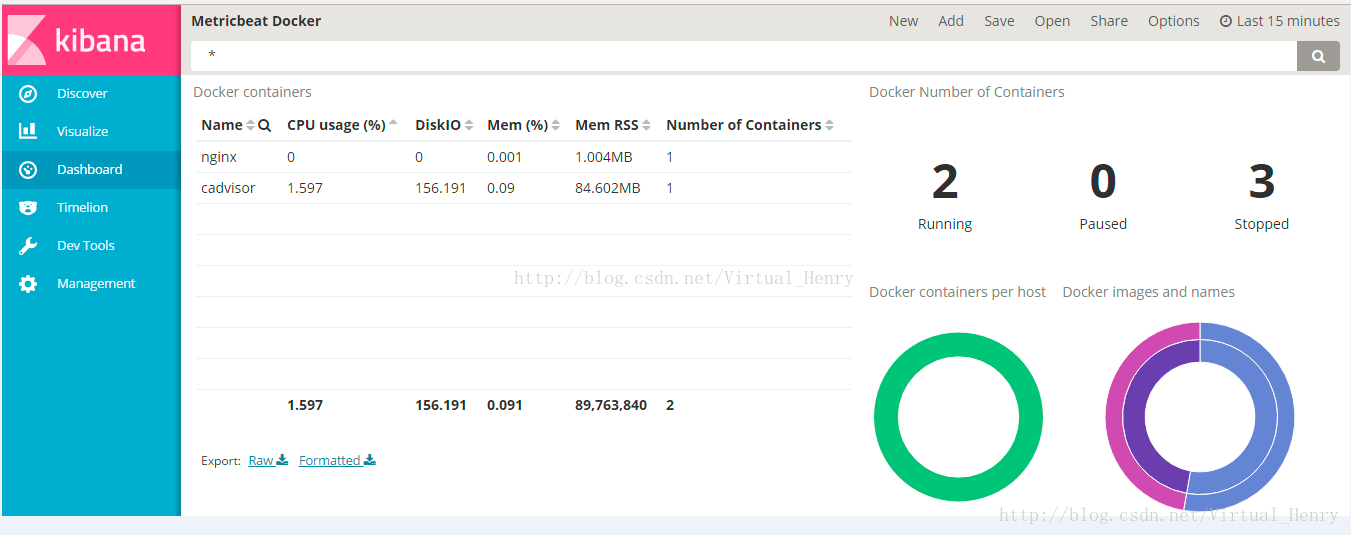
浏览器中访问http://192.168.1.100:5601/

在discover项选择metricbeat-\*查看metric的数据；

在dashboard项点击open选择要查看的数据，如

**7 查看Metricbeat-memory：**



**8 查看metricdocker：**  


**9 metricbeat输出到kafka**

**（1）在metricbeat配置文件中配置output为kafka**

output.kafka:

# initial brokers for reading cluster metadata

hosts: ["kafka1:9092", "kafka2:9092", "kafka3:9092"]

# message topic selection + partitioning

topic: '%{[type]}'

partition.round\_robin:

reachable\_only: false

required\_acks: 1

compression: gzip

max\_message\_bytes: 1000000

配置说明：

  Username：连接kafka的用户名（如果kafka有设置）

  Password：连接kafka的密码

  Topic：kafka中的topic设置

  Key：发送消息时可以指定 key，producer根据此key和partition策略来选择存入哪个partition（可用于hash计算消息发布到哪个分区）

  Partition：kafka的分发策略，有random（随机），round\_robin（轮询），hash（哈希），默认是hash

Random.group\_events:分发消息时对每个partition发送event的数量，默认为1（partition主要是实现负载均衡，让消息均匀的发送到多个partition中，设置为1则表示对一个partition发送一个event后开始对下一个partition发送event）

  Round\_robin.group\_events:同上

  Hash.hash：fields的列表，用于计算partition的hash值

Hash.random如果没有hash则采用随机分发的方式到partition

  Client\_id：theconfigurable ClientID used for logging, debugging, and auditing purposes. Thedefault is “beats”.

  Worker：Thenumber of concurrent load-balanced Kafka output workers.

  Metadata：kafka元数据更新设置（Topic/Partion与broker的映射关系：每一个Topic的每一个Partion，得知道其对应的broker列表是什么，其中leader是谁，follower是谁）

  Refresh\_frequency:刷新频率，默认10分钟

  Retry.max：尝试更新次数，默认3

  Retry.backoff:每次尝试更新之间的间隔，默认250ms

  Max\_retries:消息发布失败后重新尝试的次数，默认3（beat->kafka）

  bulk\_max\_size:单个kafka请求里面的最大事件数，默认2048

The maximum number of events to bulk in a singleKafka request. The default is 2048

  timeout：等待kafka broker响应的时间，默认30s

  broker\_timeout：kafka broker等待请求的最大时长，默认10s

  channel\_buffer\_size:每个kafka broker在输出管道中的消息缓存数，默认256

Per Kafka broker number of messages buffered inoutput pipeline. The default is 256.

  keep\_alive：网络连接的保活时间，默认为0，不开启保活机制

  compression：输出压缩码，可选项有none，snappy，gzip，默认为gzip

（kafka支持的压缩，数据会先被压缩，然后被生产者发送，并且在服务端也是保持压缩状态，只有在最终的消费者端才会被解压缩）。

  max\_message\_bytes：允许的最大json消息大小，默认为1000000，超出的会被丢弃，应该小于broker的  message.max.bytes（broker能接收消息的最大字节数）

  required\_acks:kafka的响应返回值，0位无等待响应返回，继续发送下一条消息；1表示等待本地提交（leader broker已经成功写入，但follower未写入），-1表示等待所有副本的提交，默认为1

flush\_interval:在producer api调用时，等待新事件的时间秒数

The number of seconds to wait for new eventsbetween two producer API calls.

ssl：ssl的配置项，如对kafka连接的root CA等，具体配置，参考点这里

**（2）选择配置：**

vi /etc/metricbeat/metricbeat.yml

output.kafka:

#Array of hosts to connect to.

hosts: ["192.168.1.100:9092"]

#message topic

topic: "test2topic"

**（3）然后重启metricbeat服务即可。**

在kafka中查看topic中保存的消息：

{"@timestamp":"2017-02-20T02:48:31.499Z","beat":{"hostname":"centos-78","name":"centos-78","version":"5.2.0"},"metricset":{"module":"system","name":"process","rtt":72121},"system":{"process":{"cmdline":"pickup-l -t unix -u","cpu":{"start\_time":"2017-02-20T02:42:40.000Z","total":{"pct":0.000000}},"fd":{"limit":{"hard":4096,"soft":4096},"open":12},"memory":{"rss":{"bytes":4059136,"pct":0.001400},"share":3035136,"size":91222016},"name":"pickup","pgid":2139,"pid":4847,"ppid":2139,"state":"sleeping","username":"postfix

## 七、Packetbeat

**1 packetbeat概念**

使用logstash来收集mysql的慢查询日志，然后通过kibana以web的方式展示出来，但在生产环境中，需求会更复杂一些，而且通过logstash写正则，实在是个费时费劲的事。例如在生产环境中会有要求分析某个时间段mysql或者mongodb的慢查询日志情况；还有I/O吞吐量；这个时间段内经常执行的查询语句，http访问情况等信息；然后将分析出来的结果以图表的形式展现出来。听起来是不是有点头晕，有点高大上的感觉，其实通过packetbeat，一切将变得简单高效。本文介绍使用packetbeat，elasticsearch,kibana实现这个需求。

操作系统版本：centos6.6 64bit

Elasticsearch版本：elasticsearch-2.1.0.tar.gz

Kibana版本：Kibana 4.2.1

Packetbeat版本：packetbeat-1.0.0-1.x86\_64

Topbeat版本：topbeat-1.0.0-x86\_64 （topbeat其实是用来收集操作系统信息的）

在前两篇文章中未介绍如果安装elasticsearch和kibana，这个其实很简单，基本下载下来解压一下，稍微修改一下配置文件即可运行起来，所有就忽略了，如果有问题，可以自行百度或者bing一下。目前packetbeat支持的网络协议有http,mysql,postgresql,redis,mongodb和thrift。Packetet支持pcap，pf\_ring等抓包方式，采用哪种方式进行抓包，则需要安装相应的依赖包。

**2 下载并安装packetbeat**

# yum -y install libpcap

# rpm -ivh https://download.elastic.co/beats/packetbeat/packetbeat-1.0.0-x86\_64.rpm

# rpm -ivh https://download.elastic.co/beats/topbeat/topbeat-1.0.0-x86\_64.rpm

**3 向elasticsearch导入packetbeat模板**

# curl -XPUT

'http://192.168.1.226:9200/\_template/packetbeat' -d@/etc/packetbeat/packetbeat.template.json

**4 修改packetbeat配置文件**

**# cat /etc/packetbeat/packetbeat.yml --server15**

shipper:

name: server15

tags: ["server15"]

interfaces:

device: any

type: pcap

buffer\_size\_mb: 100

protocols:

mysql:

ports: [3306]

output:

elasticsearch:

host: 192.168.1.207

port: 9200

enabled: true

**# cat /etc/packetbeat/packetbeat.yml --server226**

shipper:

name: server226

tags: ["server226"]

interfaces:

device: eth0

type: pcap

buffer\_size\_mb: 100

protocols:

mongodb:

ports: [37017, 38017]

send\_request: true # index the request payload

send\_response: true # index the response payload

max\_docs: 10 # maximum number of documents to index per request/response

max\_doc\_length: 1024 # maximum document size to index

protocols:

mysql:

ports: [3306]

protocols:

redis:

ports: [6379]

output:

elasticsearch:

enabled: true

host: 192.168.1.207

port: 9200

**5 启动packetbeat服务**

# /etc/init.d/packetbeat start

**6 导入packetbeat-dashboards**

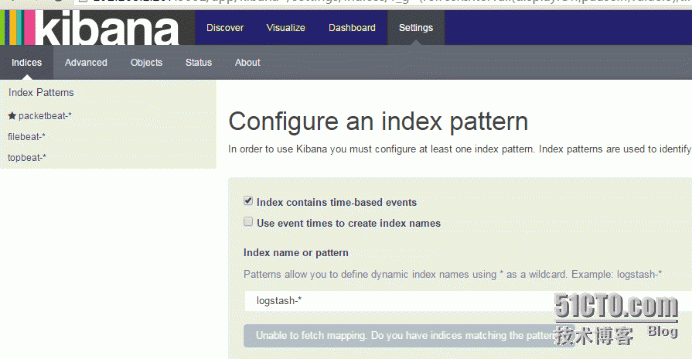
# git clone https://github.com/elastic/packetbeat-dashboards

# cd packetbeat-dashboards

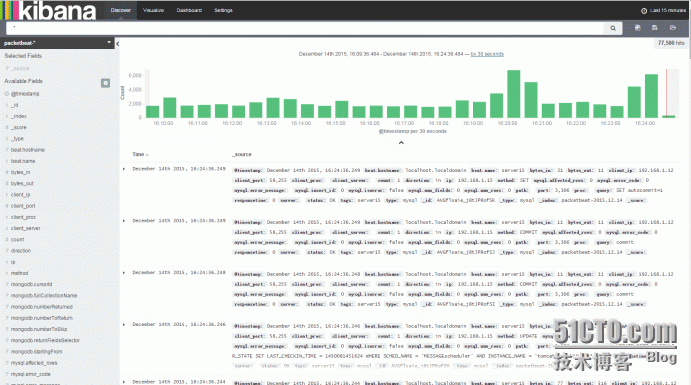
# sh load.sh -url http://192.168.1.207:9200

**7 web展示**

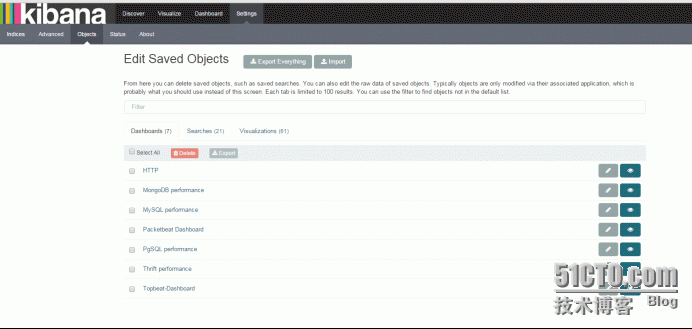
1: 配置索引，这个在执行完load.sh脚本之后，索引会自动创建



2: 查看客户端的数据推送情况



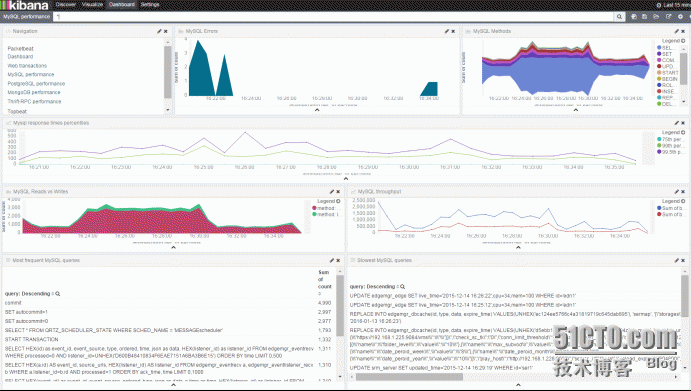
3: 查看导入的面板，可视化视图，点击setting-objects



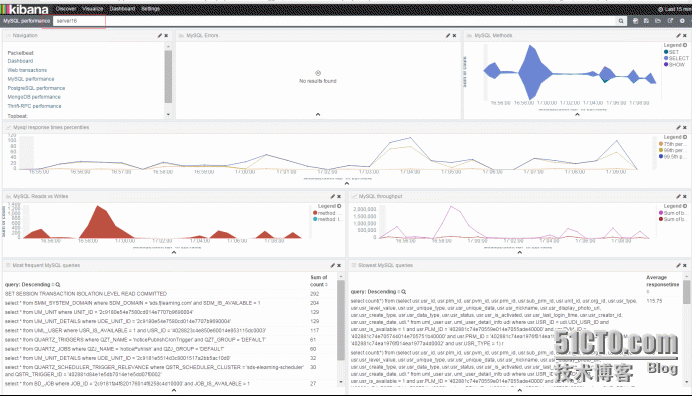
4: 图形展示,点击dashboard-load save dashboards



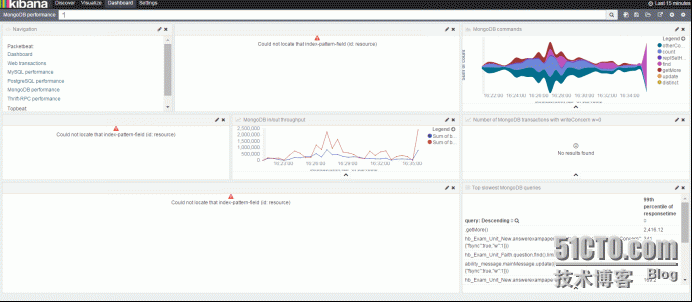
Mysql情况：



在有多台mysql服务的情况下，可以根据tags来区分，在搜索框中输入相应的tag，则只显示对应的数据



Mongodb情况



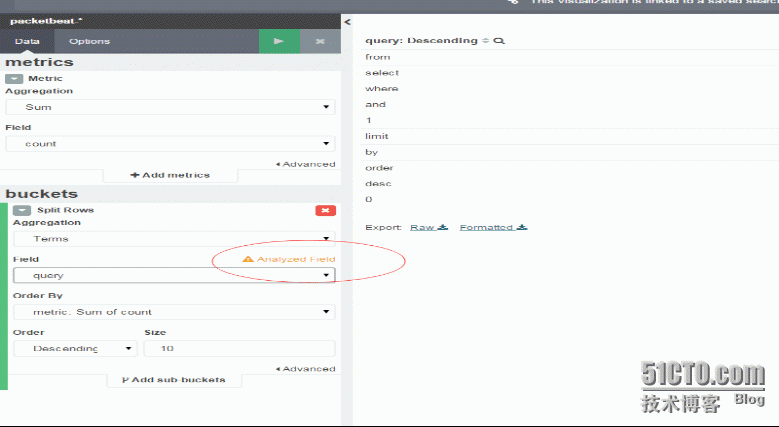
汇总情况：



7：故障排错

(1): 在测试过程中曾经发现mysql里面的most frequent Mysql queries和slowest mysql queries数据显示不全，像是被截断的样子，排查后发现其实是模板的问题，删除模板后重新导入即可.





# curl -XDELETE 'http://192.168.1.207:9200/\*'

# curl -XPUT

'http://192.168.1.207:9200/\_template/packetbeat' -d@/etc/packetbeat/packetbeat.template.json

# cd packetbeat-dashboards

# sh load.sh -url http://192.168.1.207:9200

(2): elasticsearch数据维护

搜索数据：(如果你有多个索引，可以把packetbeat-\*换成对应的索引名)：

# curl -XGET 'http://192.168.1.226:9200/packetbeat-\*/\_search?pretty'

删除数据(如果你有多个索引，可以把packetbeat-\*换成对应的索引名)：

# curl -XDELETE 'http://192.168.1.207:9200/packetbeat-\*'

## 八、Winlogbeat

**1 winlogbeat 背景**

公司人数五位数起，基于AD的域模式集中式管理。AD密码X天过期一次，用户输错Y次 锁定账号。有一天AD管理员偷偷请我吃了顿饭，暗示说到能不能在用户账号被锁的时候 就发送一条微信给用户，引导用户自助解决AD账号被锁问题。吃人的嘴软 拿人的手短，我于是满口答应了下来。现有的收集AD日志架构是通过winlogbeat 发送日志到elasticsearch

我的思路是通过winlogbeat 在发送一份日志到logstash中，通过logstash中output的exec执行Python脚本发送锁定日志给用户，并引导解锁。

**2 winlogbeat监听windows日志到kafka、hdfs,不同层级取数据**

**（1）winlogbeat.event\_logs:**

- name: Application

ignore\_older: 72h

- name: Security

- name: System

**（2）安装 Winlogbeatedit**

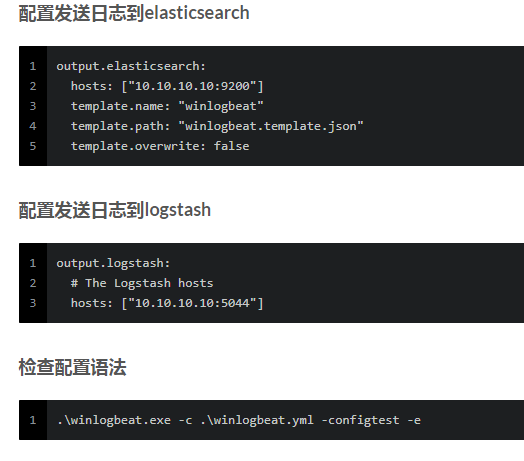
@Download the Winlogbeat zip file from the downloads page.

@Extract the contents into C:\Program Files.

@Rename the winlogbeat- directory to Winlogbeat.

@Open a PowerShell prompt as an Administrator (right-click on the PowerShell icon and select Run As Administrator). If you are running Windows XP, you may need @to download and install PowerShell.

@Run the following commands to install the service.



**3 接入kafka**

#--------------------------kafka-----------------------------------

output.kafka:

# initial brokers for reading cluster metadata

hosts: ["192.1.1.216:9092"]

topic: 'chao-beat0710'

**4 kafka查询数据命令**

bin/kafka-console-consumer.sh --zookeeper 192.1.1.216:2181 --topic chao-beat0710 --from-beginning

**5接入到hdfs**

input{

kafka{

zk\_connect=>"192.1.1.216:2181"

topic\_id=>"chao-beat0710"

}

}

filter{

mutate{

#替换windows中获取message数据中的\n以及\t为空

gsub => ["message","\n",""]

gsub => ["message","\t",""]

#获取key值的测试,可以通过map的值,比如..beat是map,则beat下面的字段是key，可以通过beat[name]去到字段值

update => {"message" => "%{activity\_id}--%{beat}--%{beat[hostname]}"}

}

}

output{

webhdfs{

host => "192.1.1.151"

port => 50070

path => "/chao/hdfs/test/kafka3/data.txt"

user => "lee"

}

stdout{

codec => rubydebug

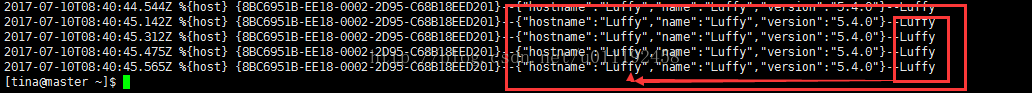
}

}

**6 数据格式例子#**



**7 取出数据在hdfs展示#**



**8 附带winlogbeat接入es，以及logstash方式**

#-------------------------- Elasticsearch output ------------------------------

#output.elasticsearch:

# Array of hosts to connect to.

#hosts: ["localhost:9200"]

# Optional protocol and basic auth credentials.

#protocol: "https"

#username: "elastic"

#password: "changeme"

#----------------------------- Logstash output --------------------------------

#output.logstash:

# The Logstash hosts

#hosts: ["localhost:5044"]

**9 最后附带官方文档API**

(

packetbeat(用于监控网络流量)、

filebeat(用于监听日志数据，可以替代logstash-input-file)、

topbeat(用于搜集进程的信息、负载、内存、磁盘等数据)、

winlogbeat(用于搜集windows事件日志)

)

## 九、Heatbeat

**1 HA概念**

(high available)高可用，又被叫做双机热备，用于关键性业务。简单理解就是，有2台机器 A 和 B，正常是 A 提供服务，B 待命闲置，当 A 宕机或服务宕掉，会切换至B机器继续提供服务。常见的实现高可用的开源软件有 heartbeat 和 keepalived。这样，一台 web 服务器一天24小时提供web服务，难免会存在 web 服务挂掉或服务器宕机宕机的情况，那么用户就访问不了服务了，这当然不是我们期望的。如果这样，有2台服务器，A对外提供 web 服务，B作为备用，如果A挂掉，那么B立刻替代A的位置去提供 web 服务，这样对用户来说是透明的。但是有个问题，服务器A的 ip 是 10.0.0.100，服务器B的 ip 是 10.0.0.101，显然向用户提供A或B的ip地址是不可行的，因为用户总不能去切换ip来访问的吧。这时heartbeat或keepalived可以提供一个虚拟IP：10.0.0.102，用户只需要访问 10.0.0.102，当A提供服务时，VIP 会设置在A服务器上，当B提供服务时，VIP会设置在B服务器上，这样就可以让用户通过访问 10.0.0.102 来获取web服务，即使A或B服务器切换也不影响用户的正常访问。下面我们使用 heartbeat 来做 HA 集群，并且把 nginx 服务作为 HA 对应的服务。

**2 Heartbaet**

一款开源高可用（Highly-Available）服务的软件，通过heartbeat, 可以将资源（ip及程序服务等资源）从一台已故障的计算机快速转移到另一台正常运转的机器上继续提供服务，一般称之为高可用服务。在实际成产应用场景中，heartbeat的功能和另一个高可用开源软件keeplived有很多相同之处，但在生产中，对应实际的业务应用也是有区别的，例如：keeplived主要是控制ip的漂移，配置、应用简单，而hearbeat则不但可以控制ip的漂移，更擅长对资源服务的控制（mysql的重启），配置，应用比较复杂。

**3 Heartbeat工作原理**

keeplived和heartbeat高可用是操作系统级别的，不是（软件级别的），可以通过简单的脚本，实现软件级别的高可用。

heartbeat的主备模式 ，通过修改heatbeat软件的配置文件，可以指定那一台heartbeat服务器为主服务器，则另一台将自动成为热备服务器。然后再热备服务器上配置Heartbeat守护程序来监听自服务器的心跳信息。如果热备服务器在指定时间内未监听到来自主服务器的心跳，就会启动故障转移程序，并取得主服务器的相关资源服务的权限，接替主服务器继续不间断的提供服务，从而达到资源及服务高可用性的目的。

heartbeat还支持主主模式（可以针对不同的业务），及两台服务器互为主备，这时他们之间会相互发送报文来告诉对方自己的当前的状态，如果在指定的时间内未收到对方发送的心跳报文，那么一方就会认为对方实效或者宕机，这时美个运行正常的主机就会启动自身的资源接管模块来接管运行在对方主机上的资源或者服务，继续为用户提供服务。一般情况下，可以较好的实现一台主机故障后，企业服务仍能够不间断的持续运行。注意，所谓的业务不间断，在故障转移期间也是需要切换时间的（例如：停止数据库及存储服务等），heartbeat的主备高可用的切换时间一般是在5-20秒左右（服务器的宕机比人工切换服务快）。

**4 Heartbeat消息类型**

**（1）心跳消息**

心跳消息为约150字节数据包，可能为串口，单播、广播或多播的方式，控制心跳频率及出现故障等待多久进行故障转移。

**（2）集群转换消息**

ip\_request和ip\_request\_resp

当主服务器恢复在线状态时，通过ip\_requset

消息要求备机释放主服务器失败时报备服务器取得资源，；备服务器释放主服务器失败时取得的资源及服务后，通过ip\_request——resp消息通知主服务器它不再拥有该资源及服务。

**（3）重传请求 rexmit－request控制重传心跳请求（不重要）**

**5准备实验环境**

服务器A：

主机名：master

操作系统：CentOS6.8 64位

eth0网卡地址：192.168.0.18

eth1网卡地址：172.16.254.18

服务器B：

主机名：slave

操作系统：CentOS6.8 64位

eth0网卡地址：192.168.0.28

eth1网卡地址：172.16.254.28

虚拟VIP：

VIP：192.168.0.38

**6 设置主机名**

master节点设置hostname

hostname master

vim /etc/sysconfig/network

编辑配置文件:

HOSTNAME=master

slave节点设置hostname

# hostname slave

# vim /etc/sysconfig/network

编辑配置文件:

HOSTNAME=slave

**7 关闭防火墙和selinux(2台节点都要操作)**

关闭iptables

# iptables -F

# service iptables save

# service iptables stop

关闭selinux：

# setenforce 0

# sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config

**8 配置hosts文件(2台节点都操作)**

# vim /etc/hosts

增加内容如下：

192.168.0.18 master

192.168.0.28 slave

**9安装epel扩展源 (2台都操作)**

# yum install -y epel-release

**10安装heartbeat (2台都操作)**

# yum install -y heartbeat\* libnet nginx

**11主master节点配置**

**（1）、拷贝配置文件：**

# cd /usr/share/doc/heartbeat-3.0.4/

# cp authkeys ha.cf haresources /etc/ha.d/

# cd /etc/ha.d

**（2）、修改authkeys**

# vim authkeys

更改或增加如下内容：

auth 3

3 md5 Hello!

然后修改其权限

# chmod 600 authkeys

**（3）、编辑haresources文件**

# vim haresources

加入下面一行：

master 192.168.0.38/24/eth0:0 nginx

说明：master为主节点hostname，192.168.0.38为vip，/24为掩码为24的网段，eth0:0为vip的设备名，nginx为heartbeat监控的服务，也是两台机器对外提供的核心服务。

**（4）、编辑ha.cf**

复制代码

# vim ha.cf

修改为如下内容：

debugfile /var/log/ha-debug

logfile /var/log/ha-log

logfacility local0

keepalive 2

deadtime 30

warntime 10

initdead 60

udpport 694

ucast eth1 172.16.254.28

auto\_failback on

node master

node slave

ping 172.16.254.1

respawn hacluster /usr/lib64/heartbeat/ipfail

复制代码

**（5）、配置说明：**

debugfile /var/log/ha-debug：该文件保存heartbeat的调试信息。

logfile /var/log/ha-log：heartbeat的日志文件。

keepalive 2：心跳的时间间隔，默认时间单位为秒s。

deadtime 30：超出该时间间隔未收到对方节点的心跳，则认为对方已经死亡。

warntime 10：超出该时间间隔未收到对方节点的心跳，则发出警告并记录到日志中。

initdead 60：在某系统上，系统启动或重启之后需要经过一段时间网络才能正常工作，该选项用于解决这种情况产生的时间间隔，取值至少为deadtime的2倍。

udpport 694：设置广播通信使用的端口，694为默认使用的端口号。

ucast eth1 172.16.254.28：设置对方机器心跳检测的网卡和IP。

auto\_failback on：heartbeat的两台主机分别为主节点和从节点。主节点在正常情况下占用资源并运行所有的服务，遇到故障时把资源交给从节点由从节点运行服务。在该选项设为on的情况下，一旦主节点恢复运行，则自动获取资源并取代从节点，否则不取代从节点。

respawn heartbeat /usr/lib/heartbeat/ipfail：指定与heartbeat一同启动和关闭的进程，该进程被自动监视，遇到故障则重新启动。最常用的进程是ipfail，该进程用于检测和处理网络故障，需要配合ping语句指定的ping node来检测网络连接。如果你的系统是64bit，请注意该文件的路径。

**12 把主节点上的三个配置文件拷贝到从节点**

# cd /etc/ha.d

# scp authkeys ha.cf haresources slave:/etc/ha.d

**13 从节点slave编辑ha.cf**

# vim /etc/ha.d/ha.cf

只需要更改一个地方如下:

ucast eth1 172.16.254.28改为ucast eth1 172.16.254.18

**14 启动heartbeat服务**

配置完毕后，先master启动，后slave启动。

# service heartbeat start

**15 检查测试**

# ifconfig

看是否有接口 eth0:0

# ps aux | grep nginx

看是否有nginx进程

**16 测试方式1**

主节点上故意禁ping

# iptables -I INPUT -p icmp -j DROP

**17 测试方式2**

主节点停止heartbeat服务

# service heartbeat stop

**18 测试脑裂**

主节点master和从节点slave都down掉eth1网卡

# ifdown eth1

**19 防止裂脑方式总结**

1）、同时使用串行电缆和以太网电缆，同时使用两条心跳线。 常用

2）、当检测到裂脑时强行关闭心跳节点。相当于程序上备节点发现心跳线故障，发送关机命令到主节点（场景使用较少，银行使用较多，需要特殊设备支持，如stonith 杀死其它节点、fence）

3）、监控报警 （依赖报警）方式1：探测备节点是否有vip，然后探测主服务是否有异常）

（ 人工介入冲裁，对网站常规业务，例如百度报警监控上下行）或者在报警在服务器接管之前，给人员处理留足够的时间。

1. 、启用磁盘锁。（使用较少）

即：正在服务的一方只在发现心跳全部断开时才启用磁盘锁，平时不上锁，此功能适用于共享场景；比如oracle

5）、增加仲裁机制（仲裁一般是网关，不可能挂）

当心跳全部断开时，两个节点各自ping参考ip，不通主动放弃竞争或自我重启，让通的一端接管服务。通过第三方软件仲裁获得资源

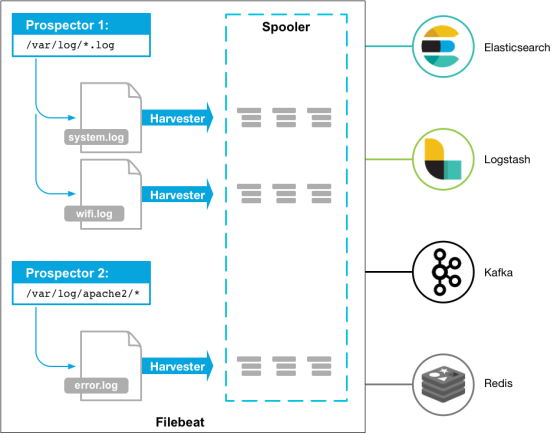
**20 使用场景**

1）.heartbeat只负责心跳监测部分和资源接管部分，应用程序自身出现异常，在2.1以前的版本是无法监控的  
2）.双网卡不绑定bond单播模式下，禁用其中一块的网卡连接，模拟网卡失效时，主/备服务器不能实现正常的故障转移， HA不切换,只有双网卡全部断开连接，HA才会切换  
3）.heartbeat程序停止 HA一定会切换 这也是该软件效果最明显的地方  
4）.系统内核异常或者OS挂起的情况下，会出现资源抢占的情况，主服务节点资源无法释放，服务异常，而此时从服务器会接管资源，同时激活haresource,可以采用watchdog服务，出现异常后重启该服务器  
5）. 使用ipfail最大的作用是检测连接情况，当存在单一网卡时，有可能出现网卡故障，网络正常，此时HA却依然切换，造成集群服务不稳定。  
针对以上可能出现的情况 可采用的解决方案：  
**heartbeat-devel-2.1.3-3.el5 + watchdog + ipfail（指向单一网关） + bound技术 （双网卡绑定冗余） + LVS ldirectord (负载均衡）**

## 十、Filebeat

**1 Filebeat简介**

是一个日志文件托运工具，在你的服务器上安装客户端后，filebeat会监控日志目录或者指定的日志文件，追踪读取这些文件（追踪文件的变化，不停的读），并且转发这些信息到elasticsearch或者logstarsh中存放。以下是filebeat的工作流程：当你开启filebeat程序的时候，它会启动一个或多个探测器（prospectors）去检测你指定的日志目录或文件，对于探测器找出的每一个日志文件，filebeat启动收割进程（harvester），每一个收割进程读取一个日志文件的新内容，并发送这些新的日志数据到处理程序（spooler），处理程序会集合这些事件，最后filebeat会发送集合的数据到你指定的地点。filebeat是一个轻量级的logstash，当你需要收集信息的机器配置或资源并不是特别多时，使用filebeat来收集日志。日常使用中，filebeat十分稳定，笔者没遇到过宕机。



**2 依赖程序**

Elasticsearch 作为存储和索引这些数据。

Kibana 作为展示平台。

Logstash（可选）去插入数据到elasticsearch。

详见 beat和elastic的入门

在安装完毕elastic集群之后，阅读接下来的选项学习怎样安装，配置，与运行filebeat。

**3 安装filebeat**

请选择你系统中的下载安装命令来下载安装filebeat。(deb for Debian/Ubuntu, rpm for Redhat/Centos/Fedora, mac for OS X, and win for Windows).如果你使用yum或者apt，你可以从我们的安装仓库中更方便的安装或更新到新版本。

deb:

curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-5.1.1-amd64.deb

sudo dpkg -i filebeat-5.1.1-amd64.deb

rpm:

curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-5.1.1-x86\_64.rpm

sudo rpm -vi filebeat-5.1.1-x86\_64.rpm

mac:

curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-5.1.1-darwin-x86\_64.tar.gz

tar xzvf filebeat-5.1.1-darwin-x86\_64.tar.gz

win:

**4 配置filebeat**

编辑配置文件来配置filebeat，对于rpm或者deb来说，配置文件是/etc/filebeat/filebeat.yml这个文件，对于MAC或者win来说，请查看你的解压文件中。

这里有一个简单的filebeat的配置文件filebeat.yml的样本，filebeat会使用很多默认的选项。

filebeat.prospectors:

- input\_type: log

paths:

- /var/log/\*.log

#- c:\programdata\elasticsearch\logs\\*

让我们来配置filebeat：

**定义你的日志文件的路径（一个或多个）**

对于大多数的基本filebeat配置，你可以定义一个单一探测器针对一个单一的路径，例如：

filebeat.prospectors:

- input\_type: log

paths：

- /var/log/\*.log

PS：若收取日志格式为json的log，请开启此配置

json.keys\_under\_root: true

在这个例子中，探测器会收集/var/log/\*.log的所有匹配文件，这意味这filebeat会手机所有的/var/log下以.log结尾的文件，此处还支持Golang Glob支持的所有模式。

在预定义级别的子目录中获取所有文件，可以使用这个配置：/var/log/\*/\*.log，这会找到/var/log下所有子目录中所有的以.log结尾的文件。但它并不会找到/var/log文件夹下的以.log结尾的文件。现在它还不能递归的在所有子目录中获取所有的日志文件。

如果你设置输出到elasticsearch中，那么你需要在filebeat的配置文件中设置elasticsearch的IP地址与端口。

output.elasticsearch:

hosts: ["192.168.1.42:9200"]

如果你设置输出到logstarsh，那么请参考第三步，配置filebeat使用logstarsh

样例：

filebeat:

prospectors:

-

paths:

- /www/wwwLog/www.lanmps.com\_old/\*.log

- /www/wwwLog/www.lanmps.com/\*.log

input\_type: log

document\_type: nginx-access-www.lanmps.com

-

paths:

- /www/wwwRUNTIME/www.lanmps.com/order/\*.log

input\_type: log

document\_type: order-www.lanmps.com

-

paths:

- /www/wwwRUNTIME/www.lanmps.com/pay/\*.log

input\_type: log

document\_type: pay-www.lanmps.com

output:

#elasticsearch:

# hosts: ["localhost:9200"]

logstash:

hosts: ["10.1.5.65:5044"]

**5 配置filebeat或elasticsearch使用logstarsh**

如果要使用logstash对filebeat收集起来的数据执行其他处理，你需要配置filebeat使用logstash。你需要编辑filebeat的配置文件，注释elasticsearch的选项，并打开logstash的配置的注释：

#----------------------------- Logstash output --------------------------------

output.logstash:

hosts: ["127.0.0.1:5044"]

hosts选项需要指明logstash服务所监听的地址和它的端口。对于这个配置，你必须手动将索引模板加载到elasticsearch中，因为自动加载模板的选项仅适用于Elasticsearch输出。

注：若要测试你的配置文件，切换到你安装好的filebeat的可执行文件的目录，然后在命令行运行以下选项：./filebeat -configtest -e ，确保您的配置文件在默认配置文件目录下，见 目录布局。在使用这个配置之前，你需要提前设置好logstash来接收数据。如果想不通过logstash直接入库到elasticsearch，可注释掉logstash入库部分，直接打开elasticsearch入库部分即可。

output.elasticsearch:

hosts: ["localhost:9200"]

**6 启动filebeat**

sudo /etc/init.d/filebeat start

现在，filebeat已经准备好读取你的日志文件并发送到你定义的输出中了！

停止:

ps -ef |grep filebeat

kill -9 id