**ELK学习笔记**

目录

[ELK Logstash简单配置读取Nginx配置文件 3](#_Toc498002068)

[ELK Filebeat区分不同的日志源 5](#_Toc498002069)

[ELK 监控服务器并制作线图 8](#_Toc498002070)

[ELK 通过Nginx访问ip，制作地区热力图 15](#_Toc498002071)

[ELK GeoLite2 IP解析插件操作详解 19](#_Toc498002072)

[ELK 使用谷歌地图改变视角高度最大值 24](#_Toc498002073)

[ELK Nginx访问权限配置 25](#_Toc498002074)

[ELK数据中的“主键” 29](#_Toc498002075)

[ELK 动态Mapping 29](#_Toc498002076)

[ELK X-PACK插件 32](#_Toc498002077)

[ELK集群部署 33](#_Toc498002078)

[ELK 报警插件ElastAlert 36](#_Toc498002079)

[Elasticsearch安装Head插件 39](#_Toc498002080)

[Elasticsearch 平时遇到的问题 41](#_Toc498002081)

[Elasticsearch Analyzer使用 41](#_Toc498002082)

[Elasticsearch Field Collapsing（字段折叠）+ inner\_hits 50](#_Toc498002083)

[Elasticsearch Search After分页查询 50](#_Toc498002084)

[Elasticsearch Search Template(搜索模板) 52](#_Toc498002085)

[Elasticsearch Multi Search API 53](#_Toc498002086)

[Elasticsearch Validate API 55](#_Toc498002087)

## ELK Logstash简单配置读取Nginx配置文件

今天主要熟悉了下ELK中Kibana的搜索以及图标功能。

学会了配置nginx的日志输出格式。

今天证明了一件事情，就是之前我的elasticsearch进程总被杀掉，就是因为我的虚拟机内存太小导致的，扩大虚拟机内存后，今天跑了一天，都没有挂掉。

下午试着用ELK去分析采集我的nginx日志，并且通过修改nginx的日志格式为json，然后使用logstash的json过滤器使得nginx的日志的每个字段都存储在了elasticsearch中。

首先，贴出我的ELK搭建参考帖子地址：

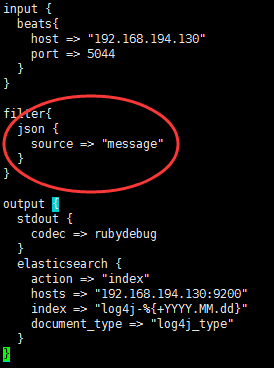
http://blog.chinaunix.net/uid-23916356-id-5764256.html

下面贴出配置文件：

Nginx的配置文件：

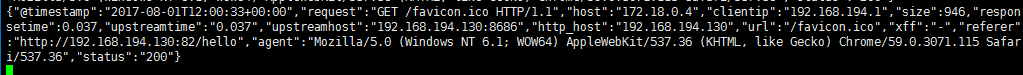
注意关键字的对应。

Logstash配置文件：



启动之前的docker-compose项目 然后访问nginx：

access.log



logstash控制台输出



发现控制台输出的logstash中，message消息中的json被提取了出来被发送到了elasticsearch，然后接下来我们就可以通过这些nginx日志的每个字段再去用kibana去做统计了。

附： elasticsearch的全部删除curl -XDELETE <http://192.168.194.130:9200/log4j-2017.08.01> [log4j-2017.08.01](http://192.168.194.130:9200/log4j-2017.08.01)是我logstash存储elasticsearch的索引，前面就是域名和端口。

今天的研究到此结束。

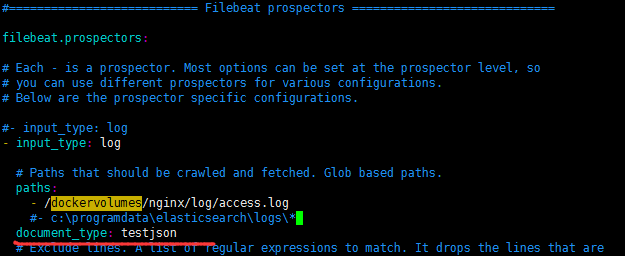
## ELK Filebeat区分不同的日志源

今天上午学习了filebeat+ELK如何区分filebeat的来源，目前在网上找到有两种方法亲测可用。

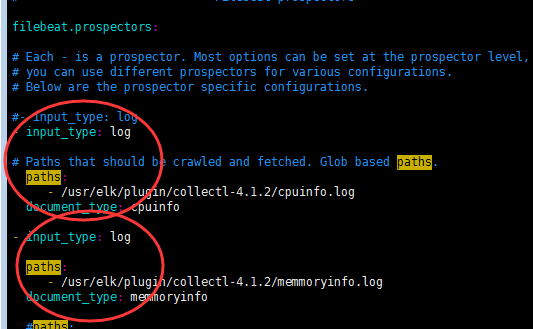
第一种，使用document\_type配置标志来源日志在logstash的input模块中的type



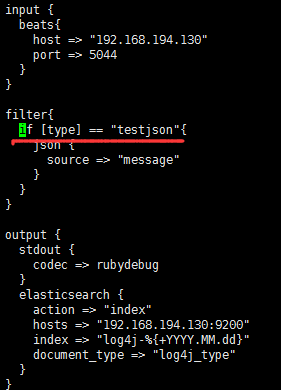
首先filebeat的配置文件：将document\_type设置为testjson



如果配置多个数据源，并且类型分别不同



然后在配置logstash的配置文件中我们去判断testjson类型



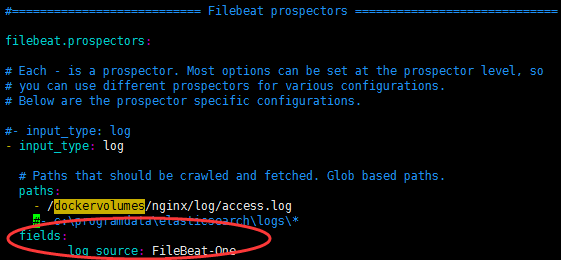
访问我们的docker-compose 中的nginx，会看到logstash控制台打印：



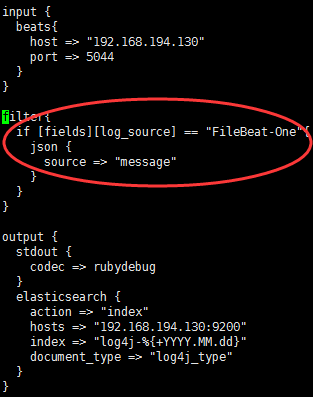
说明logstash的filter起作用了，filebeat的配置成功了。

接下来看看第二种方式：

首先filebeat中的配置如下，在paths下配置一个fields节点，并在这个节点下配置自己自定义的节点log\_source(自己写的，并非官方定义)，然后对应写一个自己的定义的值，根据这个值去判断每个来源。



然后logstash中针对filebeat的配置做过滤



然后再次访问我们的docker-componse 项目的nginx，可以再logstash的控制台输出中看到如下：



可以看到我们的nginx日志被过滤器解析了，从message中提取了出来。

## ELK 监控服务器并制作线图

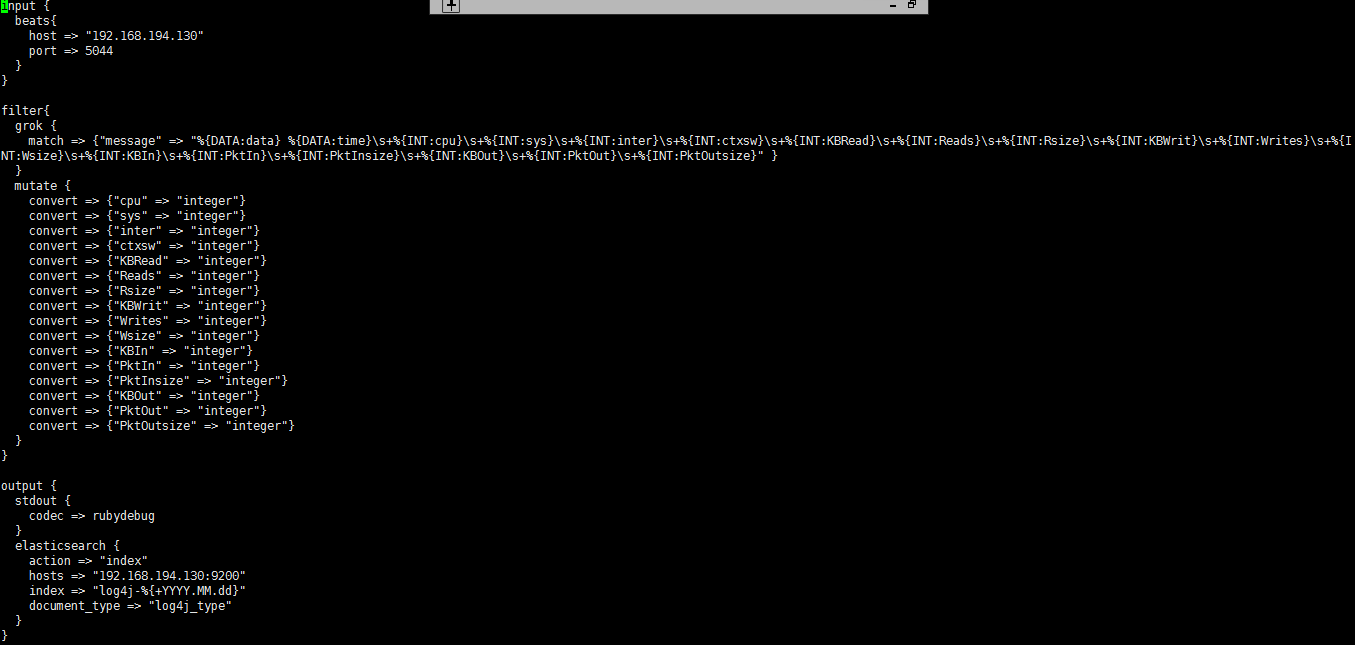
今天搞出了用ELK监控服务器的cpu参数，下面做一下详细记录。

首先我们需要用到一个linux工具，帮助我们统计服务器的性能参数，这个工具叫collectl，是一个非常好的linux服务器监控工具。通过以下命令，将其监控数据保存到一个指定的日志文件中。

./collectl -i1 -oDm --iosize > sysinfo.log

将系统分析的参数输出到sysinfo.log中，注意这个不是后台执行的命令，ctrl+c就挂掉了。

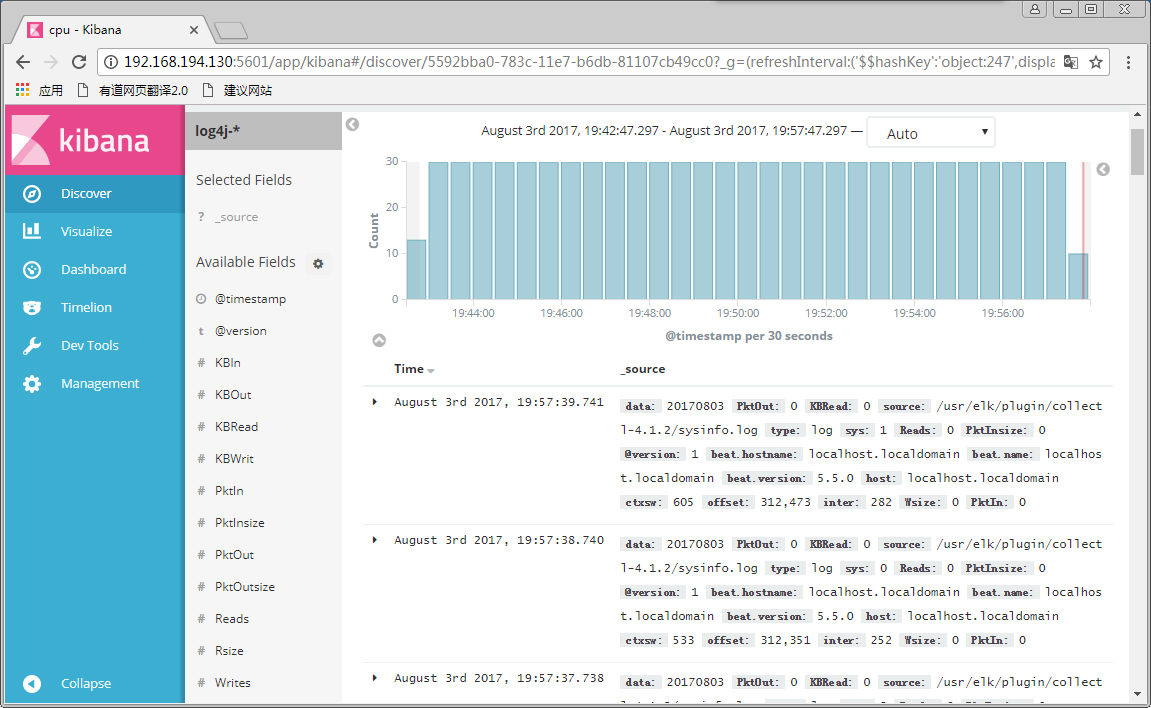
然后我们通过filebeat将这个日志发送到logstash中，logstash配置文件如下：



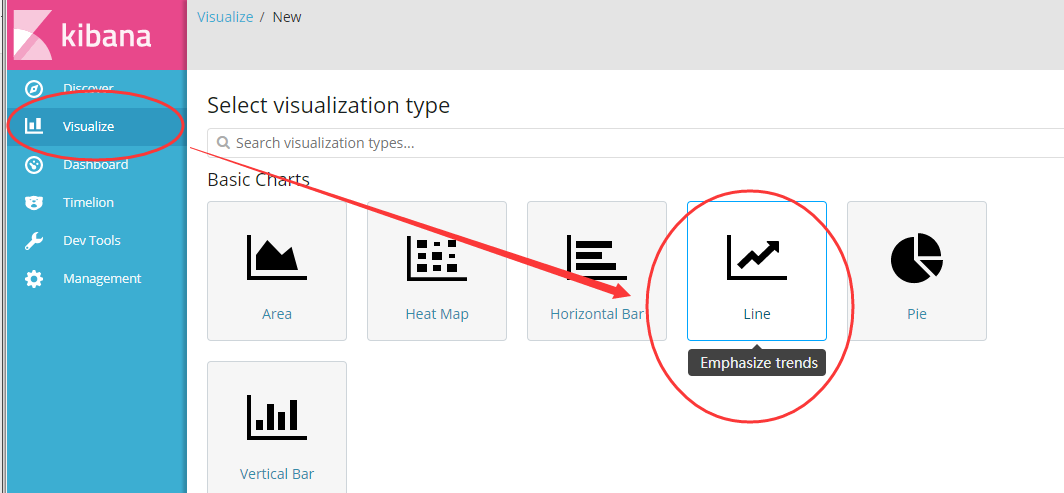
**注意：**将数据进行正则匹配了之后，我们一定不要忘记将字段进行类型转换，因为正则匹配的字符串匹配出来的数字也都是字符串的，我们需要通过logstash转换器将字符串数字转换程数字，否则Elasticsearch将不能对这些字段做运算聚合。

接下来我们看一下kibana的可视化配置：

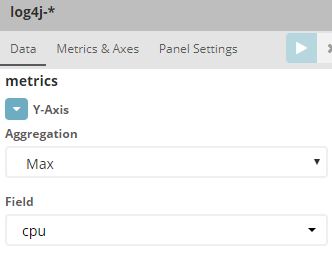
数据展示如下。



我们选择line chart线图

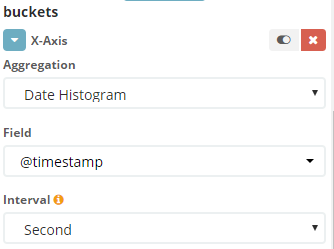


Y轴配置

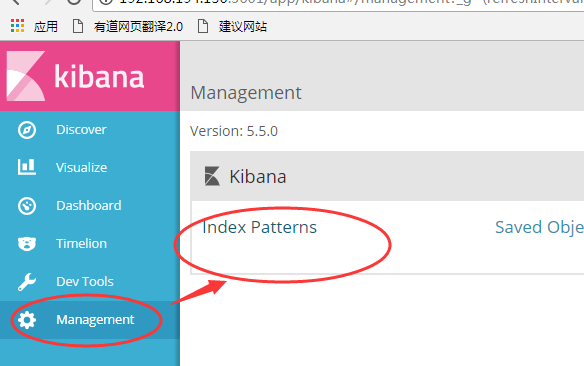


这里选择去最大值是因为我的X轴准备选择时间，并且以秒为单位，而我的后台collectl统计也是按照每秒统计，所以这里取最大值就相当于取原值了。

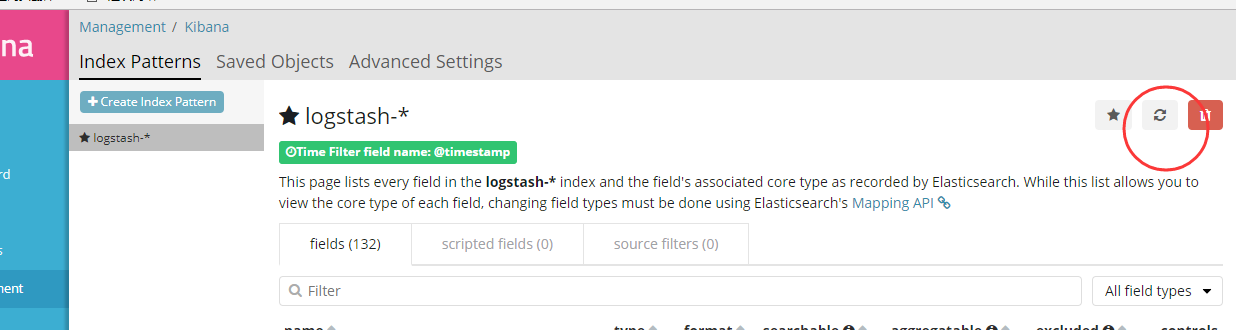
X轴配置

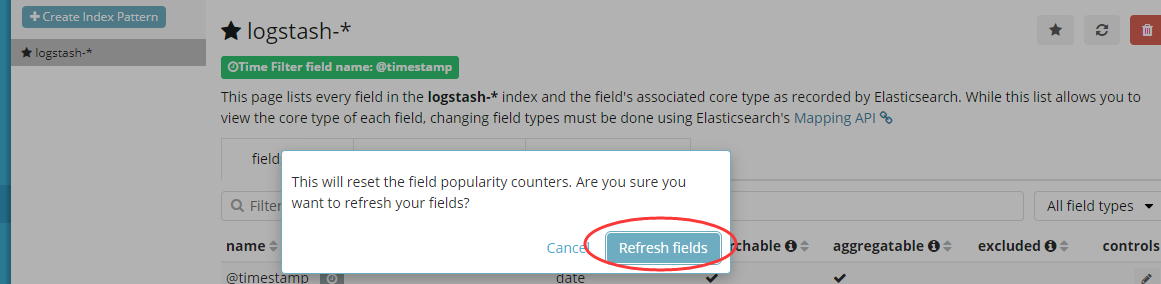
x轴的定义为时间戳，统计单位为每秒

**注意：如果你在作图过程当中发现找不到你需要统计的字段，这时你应该先确认grok表达式是否将你的message中的数据提炼出了相应字段，如果grok正常，你需要去kibana中的Management中去刷新数据索引，如果有新的格式的数据进来，不说新字段索引，作图统计的收还是一样找不到字段。**



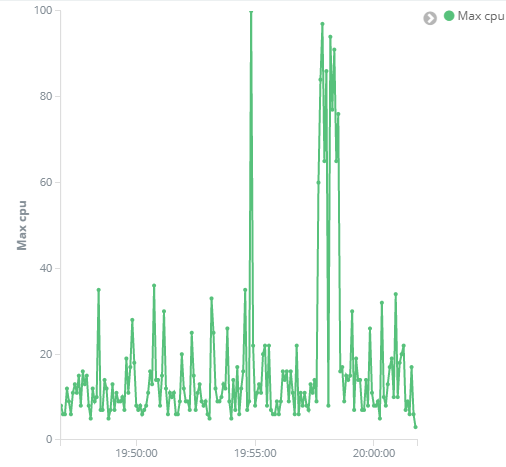
**然后我们刷新数据字段索引**





**此时发现，下面的字段索引将多出新添加的，再回到刚在的可视化界面，字段就会出现了。**

结果如图：

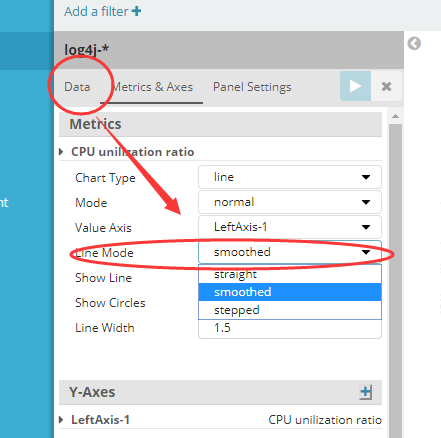


注意：如果logstash中没有转换参数的格式，Y轴就统计不到cpu字段，因为Y轴的聚合都是运算，所以如果字符串数字不做转换，不能参与运算。

正式当中最好用这个命令运行collectl，能够获取所有参数，并且后台运行

./collectl -i1 -oDm --all --iosize > sysinfo.log &

补充：如何让线图显示的更加平滑好看，成为曲线线图？这里设置。

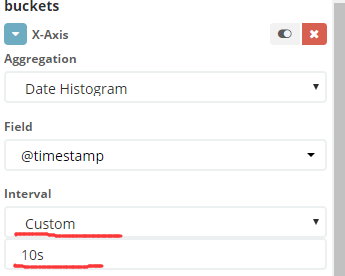


感觉更加漂亮了。



如果x轴的时间统计想自己定义间隔怎么办？上面的线图是每秒一统计，我们想自定义比如说10一统计？

看这里：

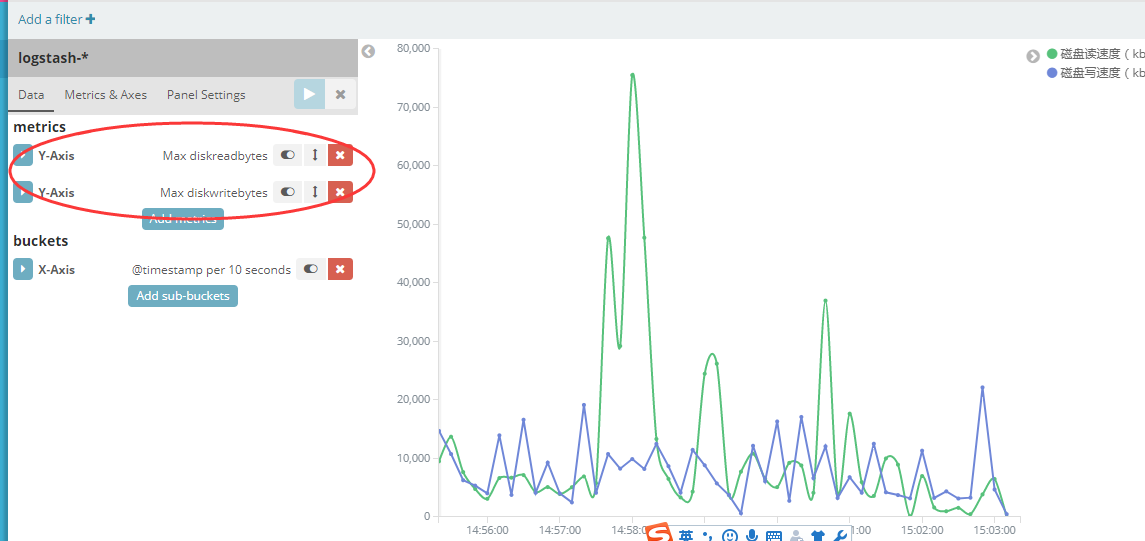
在x轴的interval处选择Custom然后下面就是自定义的时间。s(秒) m(分) h(小时)

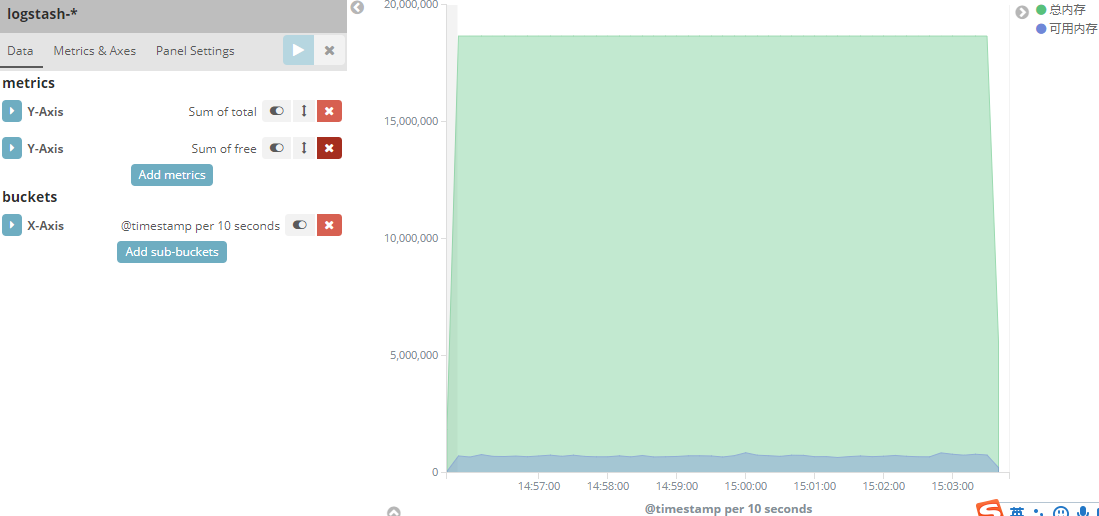
那么要想做多条线在一个图上怎么办？

有两种情况：

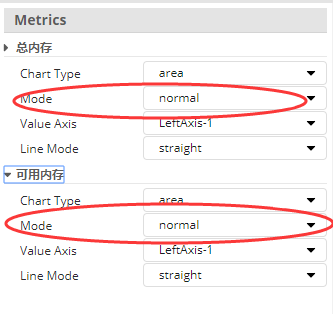
（一）统计一条数据中的两个字段参数

这种情况只要设定多个Y轴，就会有多个统计图显示在一个XY轴上。



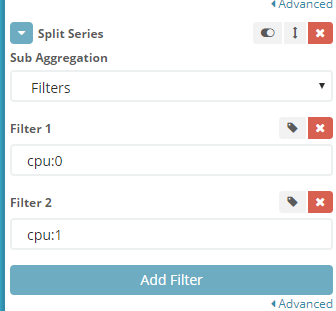


注意：设置多个Y轴的话需要配置下Y轴的形式为normal，否则多个统计图不能重叠，而是上下叠加设置如下。



（二）几条数据中，每条数据代表一个独立的数据

这种情况不能设定多个Y轴，需要用**Split Series**中的**filter**去分组。例如：一个多核cpu服务器，日志每秒每次统计的cpu参数是多条，每条代表一个cpu的参数，filter配置如下：



此时，我们将日志数据分组了，通过cpu参数将每个cpu的参数分组统计出了每个cpu的可视化图。

## ELK 通过Nginx访问ip，制作地区热力图

学习了kibana的ip地图分布统计。

首先，我们需要下载一个IP地址定位的资源库里面包含了所有公网地址的IP段，能够通过IP地址锁定经纬度，从而进行地图定位。这个插件叫做GeoLite（目前使用的是这个2版本GeoLite2-City.mmdb）。

大体逻辑为，我们通过手机nginx的日志信息，然后获取到访问的客户端ip地址，然后通过logstash的filter中的geoip插件（专门用于GeoLite分析的插件）来解析ip地址的地理位置（经度和纬度）最后提取出出相应的字段保存到ES中，再由kibana展示。

例如：nginx中的一条记录。（本例为nginx的默认日志格式，使用grok正则分析，实际情况不建议这么做，建议将nginx的日志定义成json格式，再由logstash的json插件解析，这样性能会高很多，grok性能较低，并且耗费资源）

www.58qf.com 115.159.54.244 - - [03/Jul/2017:21:07:51 +0800] http \"HEAD /www.58qf.comwwwwroot.rar HTTP/1.1\" 200 241 \"-\" \"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0)\" \"-\" \"-\" 0.002 172.16.6.85:28080 200 0.002

正则表达式如下：

%{NOTSPACE:nginx\_request} %{IP:nginx\_clientip} - %{USER:nginx\_auth} \[%{HTTPDATE:nginx\_timestamp}\] %{WORD:nginx\_scheme} \"%{WORD:nginx\_method} %{URIPATHPARAM:nginx\_uriPath} %{WORD:nginx\_http}/%{NUMBER:nginx\_version}\" %{INT:nginx\_status} %{INT:nginx\_bytesent} \"%{NOTSPACE:nginx\_httpreferer}\" %{QS:nginx\_useragent} %{QS:nginx\_xforwardedfor} %{QS:nginx\_gzipratio} (?:%{NUMBER:nginx\_requesttime}|-) (?:%{URIHOST:nginx\_upstreamaddr}|-) (?:%{INT:nginx\_upstreamstatus}|-) (?:%{NUMBER:nginx\_responsetime}|-)

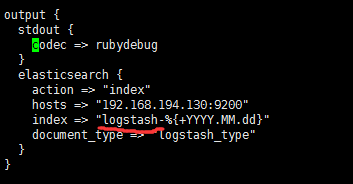
logstash的filter配置如下：



其中，geoip模块为配置GeoLite2-City.mmdb插件的模块，并且需要注意的是geoip中source字段对应这你grok中解析客户端访问ip地址的字段。



**另一个需要严重注意的地方**

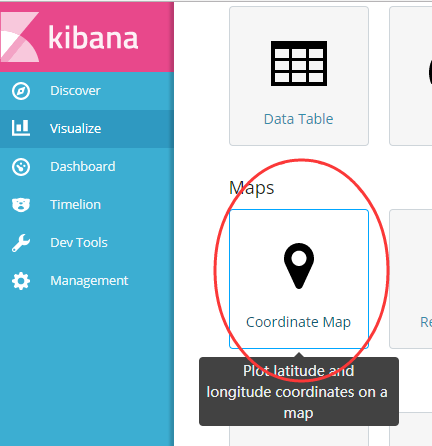


这里的索引必须配置成logstash-\*的模式，原因是地、可视化地图默认的索引能够使用它的location分析位置的模板，如果不用这个索引需要自己写模板比较麻烦。具体的原因请看：http://blog.csdn.net/yanggd1987/article/details/50469113

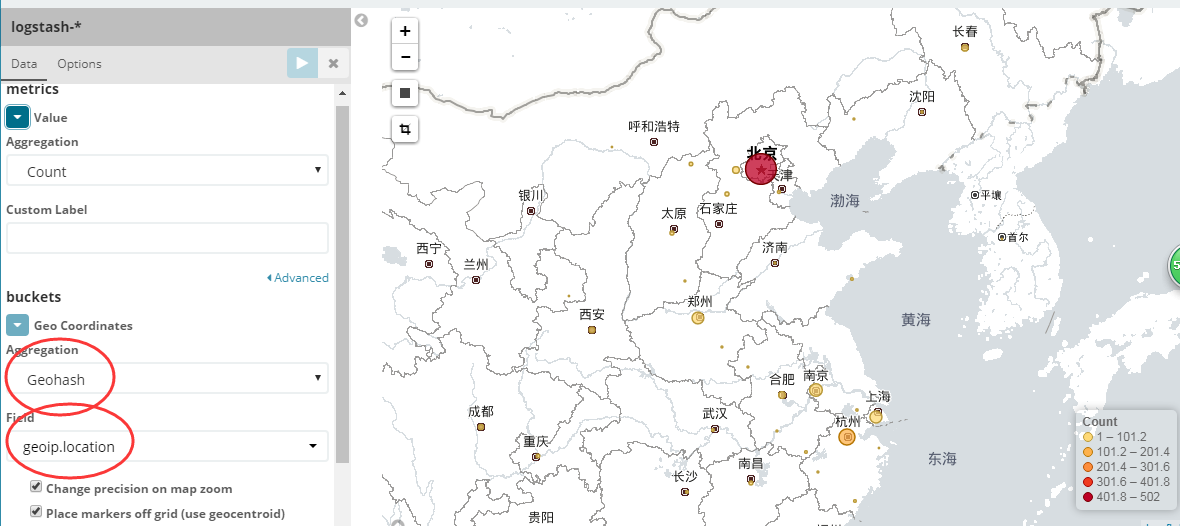
到这里logstash配置完成了，开始接收数据。

然后我们回到kibana，首先刷新你的数据字段索引原因看前面。

然后选择可视化：



这里配置就比较简单了，只需要配置两处。如果在logstash中的索引没有按照默认配置，就不会用到可用的location模板，你将不会看到geoip.location这个字段，也就不能完成这个可视化了。



另外还有一点，GeoLite2只存了世界上的所有公网ip对应的地址信息，如果nginx中记录的是内网IP地址，GeoLite2将不会解析出地址数据。

记录结束。

## ELK GeoLite2 IP解析插件操作详解

今天学习到了两点，一个是linux上ip地址查询插件GeoLite2的更新下载插件的使用，另一个就是直接通过录入经纬度数据来制作kibana的coordinates map可视化。

首先说下GeoLite2的更新下载插件，需求来源是，免费的GeoLite2的数据精度是不太准确的，比商业版的精准度低。但是由于免费版的也在不停地更新数据，所以最新版的免费版GeoLite2还是比之前的版本精度高，也就有了自动更新的需要。

更新插件的下载地址：

<https://github.com/maxmind/geoipupdate/releases>

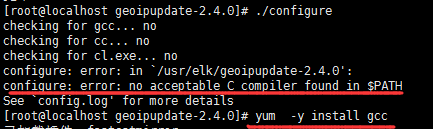
下载完成之后，在linux中进行解压。

C:\Users\Administrator\Desktop\记录\1.png

然后在执行安装程序前需要下载一个支持插件

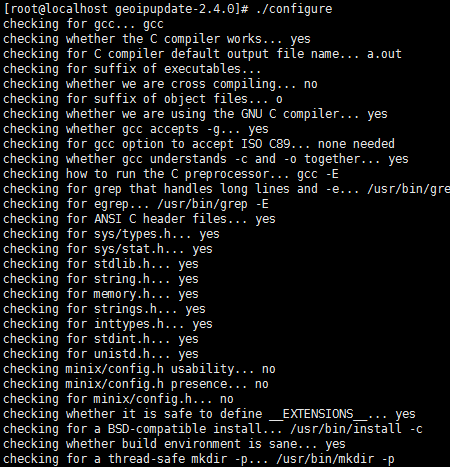
C:\Users\Administrator\Desktop\记录\2.png

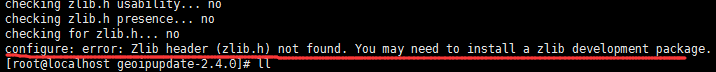
然后执行安装程序



此处报错，原因是没有C语言解析器。然后下载安装一个c语言解析器

再次安装



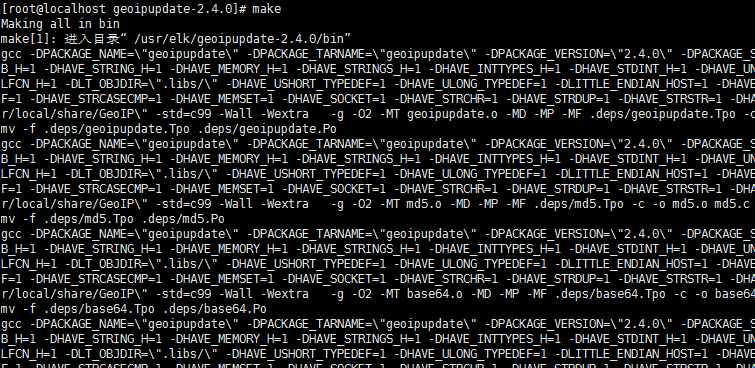


发现还缺少一个组件，然后还是下载安装

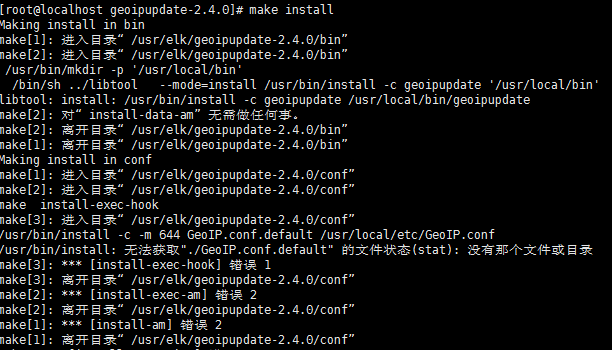
C:\Users\Administrator\Desktop\记录\6.png

再次执行安装命令./configure这次成功!

然后执行make



还有 make install



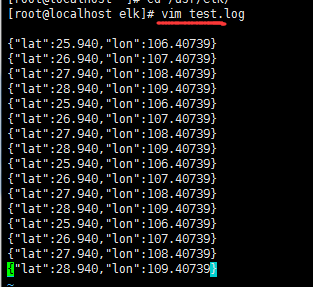
上面的错误是由于我在试安装的时候更改了配置文件GeoIP.conf.defaule的名字,导致程序找不到这个文件报的错，改回来再次执行就可以了。

最后执行命令 ./geoipupdate -d /home/testu/ -v将ip解析地质数据库插件下载到我指定的目录。完成！

现在还没有写定时任务自动下载更新，有时间不上。

接下来介绍下，通过logstash接受经纬度数据来在kibana上绘制coordinates map可视化。

首先，我们准备一些测试数据，test.log 里面有一些经纬度的json数据。



然后，配置logstash.conf



通过filebeat来传输测试日志数据。然后在filter中用json去解析字段，并将经纬度数据添加到新字段lonlat中，再将message字段移除（这里移除message字段是不得已的，以为何message的数据存到es中总是报错，所以为了下面的顺利进项想将他移除，以后再探究竟），最后将lonlat中的数据全部转换成float型（不转换就会是字符串，存到es中将不会被经纬度类型解析）

然后，由于我们需要用到经纬度，所以我们需要将我们保存经纬度的字段类型转换成经纬度类型geo\_point

由于我们新建立的索引geo-\*，所以不会使用到默认索引logstash中的类型转换，所以我们需要设置我们的数据在es中的模板（mapping），设置我们的经纬度字段lonlat为geo\_point类型，不设置将会成为默认类型（String/number）,这样就用不了coordinates map的可视化了。

在开始添加数据之前我们先在es中创建我们logstash中的索引geo-\*由于\*是一个当天的时间戳，所以我们今天的索引为geo-2017.08.10

创建索引

curl -XPUT <http://192.168.194.130:9200/geo-2017.08.10/>

添加lonlat字段类型为geo\_point类型

curl -XPUT “<http://192.168.194.130:9200/geo-2017.08.10/>” ‘{

"mappings": {

"geo\_type": {

"\_all": { "enabled": false },

"properties": {

"lonlat": {

"type": "geo\_point"

},

"@timestamp": {

"type": "date"

},

"@version": {

"type": "text"

}

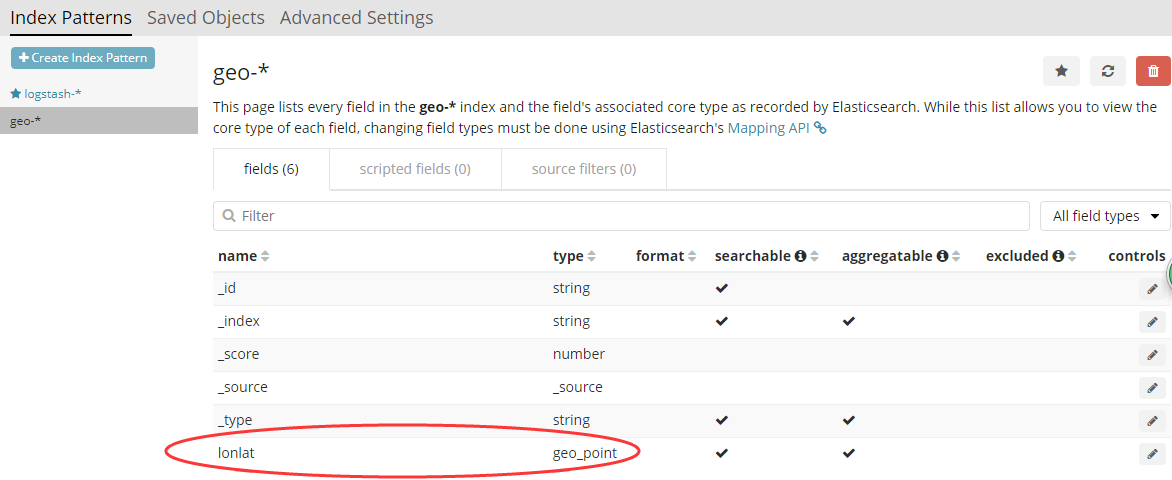
}

}

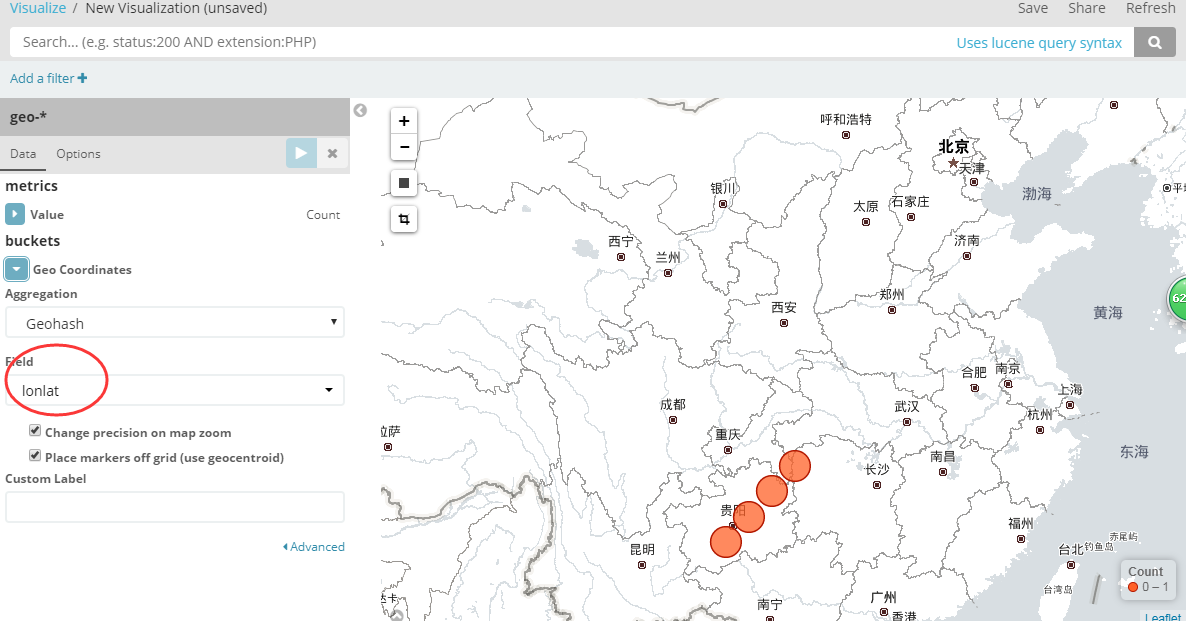
}

} ‘

然后回到kibana界面创建geo-\*索引，开启filebeat 和 logstash开始接收数据。然后刷新索引，将会看到一下效果。



我们添加的字段已经生效，成为经纬度字段类型，接下来就简单了开始制作coordinates map的可视化。



现有的问题：

（1）为什么日志的message添加到es会报错（前面logstash中溢出的字段）。

（2）es mapping的书写（由于只写了一个字段，kibana上很多东西都看不到了）。

（3）为什么在logstash中我已经将经纬度字段解析成fload类型了，添加到新建字段后又变回字符串了？

（4）mutate 过滤插件中add\_field的用法还需了解。

## ELK 使用谷歌地图改变视角高度最大值

今天突然有了个疑问，为什么kibana上的高德地图显示的分辨率不高，总觉得如果只能显示到那种程度不合理，肯定可以更精细。

所以，今天好好研究了一番，发现这个分辨率是有一个参数z来控制的，控制着视野的大小。

直接上kibana的配置文件：



[http://webrd02.is.autonavi.com/appmaptile?lang=zh\_cn&size=1&scale=1&style=7&x={x}&y={y}&z={z}](http://webrd02.is.autonavi.com/appmaptile?lang=zh_cn&size=1&scale=1&style=7&x=%7bx%7d&y=%7by%7d&z=%7bz%7d)

其中url的z这个数是一个视角高度，它的数值越大代表你所在的位置距离地面越近看的更清楚，越小距离越远，看的更模糊。

然后我们可以修改kibana coordinates map可视化的最大视角距离，使我们能够看的更清楚，只需要在kibana配置文件后面加一个配置。



完成！

## ELK Nginx访问权限配置

下面介绍，通过nginx配置kibana的访问权限。

首先，我们下载安装nginx，解压



然后进入目录编译，可能需要下面的几个插件的支持，去过缺少就安装

yum install gcc-c++

yum -y install pcre-devel

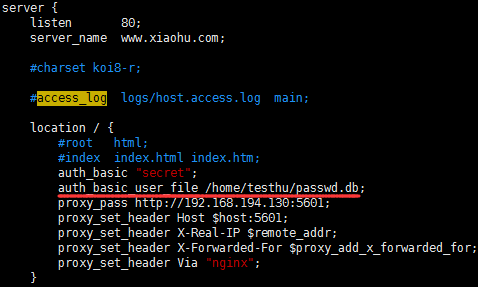
yum -y install openssl openssl-devel

yum -y install zlib-devel

执行./configure 然后make make install

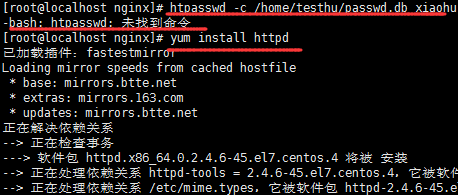
如果没有指定目录，默认安装到了/usr/local/nginx/目录下

配置文件：



标红处为我们要生成的认证文件的位置。

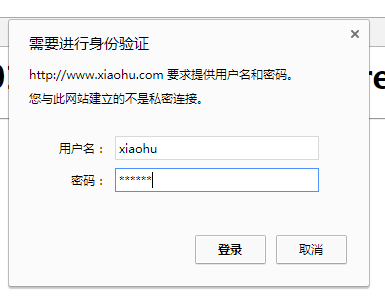
然后我们通过命令生成登录认证文件。



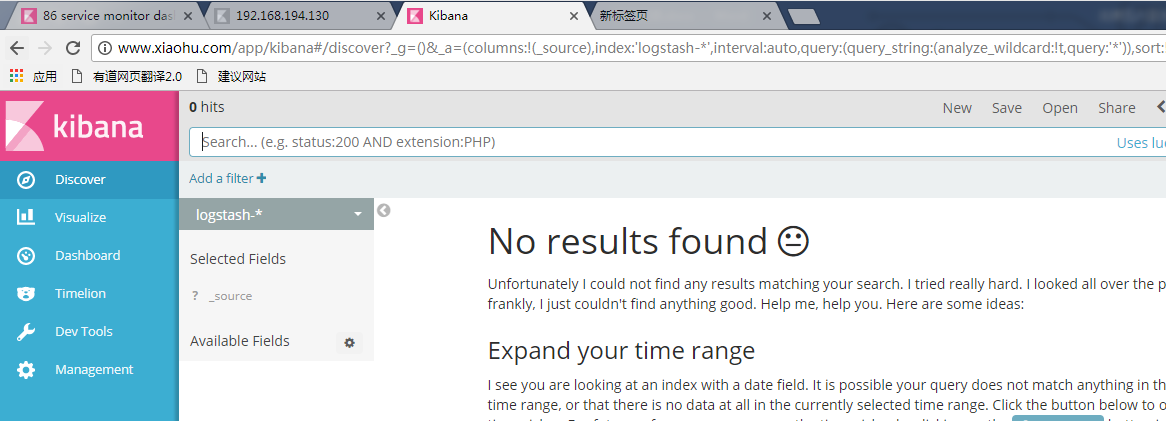
执行htpasswd命令生成认证文件，如果提示找不到命令，需要安装httpd插件。



这里我们使用的root用户取舍瓮城的文件，所以我们或者修改认证文件的访问权限或者修改nginx.conf中的nginx执行权限，否则会因为权限问题导致认证文件访问不了，造成无法登陆。



输入用户名密码

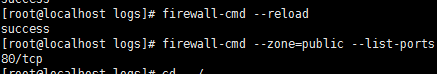
成功访问

最后我们需要开启防火墙，防止通过外网ip直接访问kibana，我的虚拟机是centos7所以防火墙用了firewall。

首先，我们开启防火墙，添加80端口访问权限：

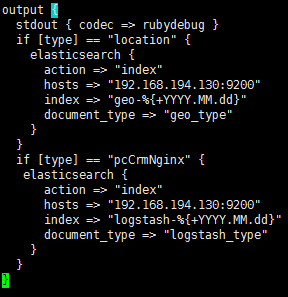


然后重启防火墙，查看防火墙已开启端口：



然后再次通过nginx访问，正常。通过IP+端口访问，失败。

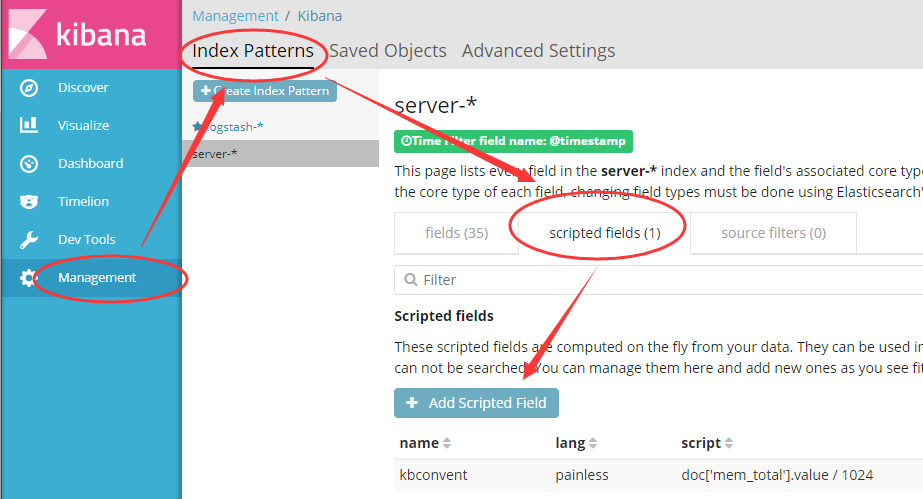
然后，logstash配置多个索引输出的方式，和filter一样，在out处判断type即可。

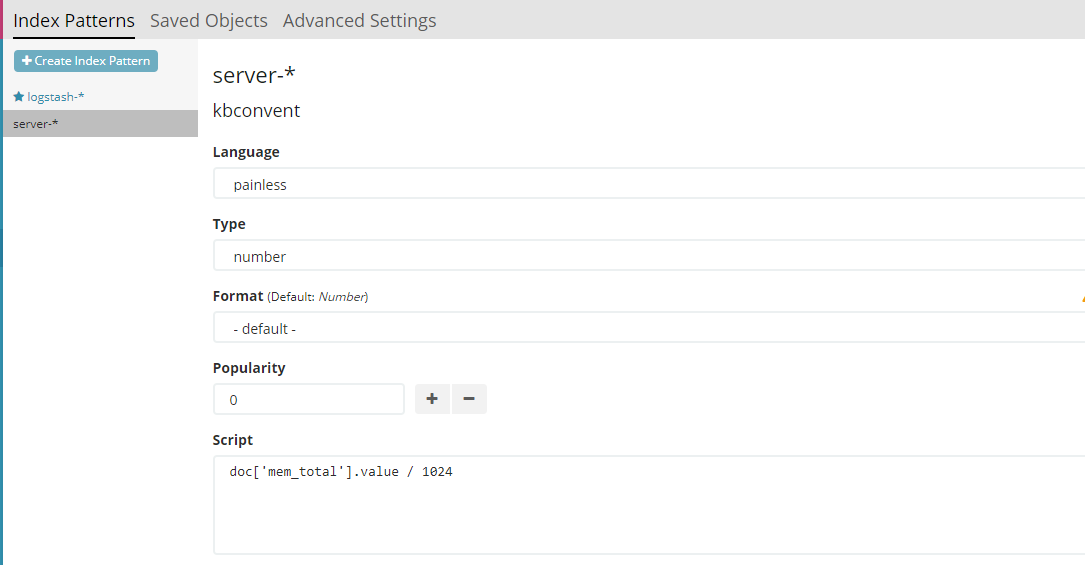


今天研究了一下Kibana的脚本过滤器。

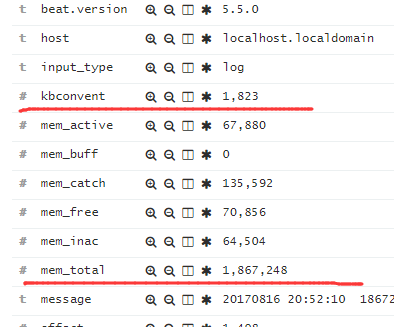
需求来源与内存统计中，我将所有的输出全都变成了kb单位，但是kibana中只有针对bytes的转换，为了能够完成转换，就退而求其次找到了Kibana的Scripted fields功能。

目标为mem\_total字段，它的单位为kb我想将他显示出它的mb值，需要在他的值的基础上除以1024.





点击保存，完成获取查看数据。



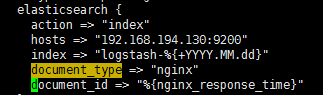
这个功能比较影响kibana的性能不太建议使用

## ELK数据中的“主键”

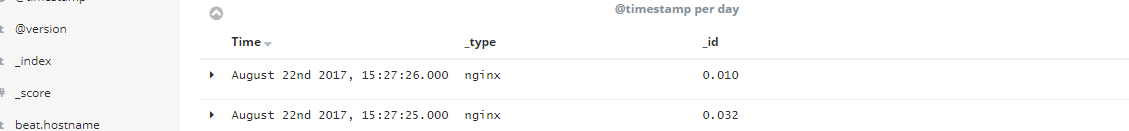
今天遇到一个问题，就是如果ELK正在收集状态会发生改变的数据，我们需要去更新或者覆盖原来的数据，否则将造成ES的数据冗余，所以看了一下ES中如何区分每条数据。

原来在ES 中也有和数据库一样的一个主键，叫做”\_id”这个字段，如果ES查询不指定“\_id”的数值，则系统会生成一个id数值。但是我们是通过logstash收集日志，并不通过ES的REST接口，通过查找官方文档，发现logstash的output中的Elasticsearch插件中 有一个document\_id字段，用来指定logstash解析出来的某个字段的值为ES中的”\_id”。这样就可以通过”\_id”去覆盖原来的数据了，从而达到更新的目的。

示例：



这里指定以nginx\_response\_time字段的值作为ES数据的id。



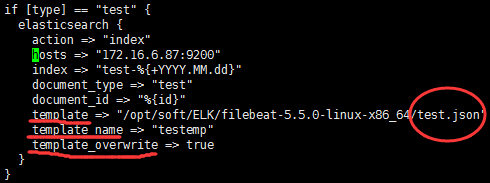
**坑：同一id但不同索引下，不会发生覆盖修改操作，如果index按照日期去做分割，跨天的相同id数据将不会被修改。**

## ELK 动态Mapping

好久没有记录一些ELK学习计划了，最近比较有进展的，就是关于es的index按时间日期分开的时候，es的自定义mapping如何能够保持一直指向我们不停变化的index。

首先环境上为filebeat+logstash+elasticsearch+kibana。在filebeat的安装目录下，有elasticsearch的默认mapping文件filebeat.template.json。但是我们这里不用他，我们来自己定义我们的mapping。

然后logstash需要在output做一些配置,具体配置如下：



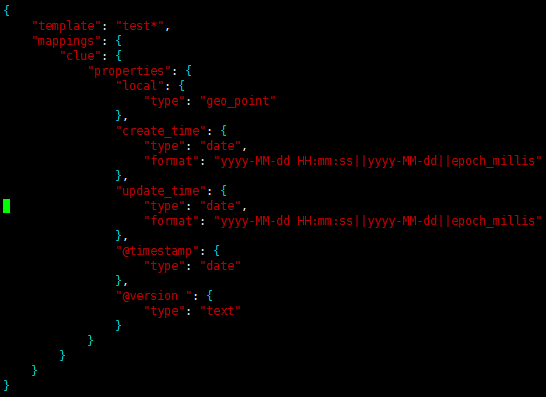
这里有3个参数之前是陌生的。

template 用来指定我们的mapping文件的路径，这里我们的mapping文件为test.json

template\_name 用来指定我们的mapping文件在es中的template名字。

Template\_overwrite 默认为false，此选项将会覆盖easticsearch指示得模板，并使用logstash中template指定的模板。

我们的模板mapping文件test.json



{

"template": "test\*",

"mappings": {

"clue": {

"properties": {

"local": {

"type": "geo\_point"

},

"create\_time": {

"type": "date",

"format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch\_millis"

},

"update\_time": {

"type": "date",

"format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch\_millis"

},

"@timestamp": {

"type": "date"

},

"@version ": {

"type": "text"

}

}

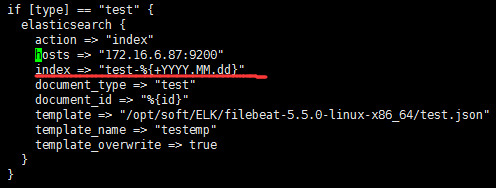
}

}

}

这里指定了local为geo\_point类型，create\_time和update\_time指定为之间戳，并设定格式化匹配方式（三种），@timestamp为时间类型，@version为字符串类型。

匹配原理：



我们的es的index名字为test并按时间戳分隔，如果我们按照es的REST接口去设置mapping，只能设置一天的，当日期改变时，我们的mapping将失效，我以我们在此采用logstash来配置我们的es的mapping。

我们的index为test-%{+YYYY.MM.dd}，并且我们需要在mapping文件test.json中匹配我们的index，即使我们的index的日期改变我们的mapping也能够匹配到我们的index

在es的mapping中，有一个template参数，用来匹配我们的index，由于我们的index会所这日期发生改变，所以我们这里=将其设置为test\*（对应index test-%{+YYYY.MM.dd}）



然后我们插入数据，即使index随时间改变，mapping也能都对应上，不会失效。

## ELK X-PACK插件

介绍ELK x-pack插件安装以及运行。

首先，ELK各个部分下载x-pack插件

bin/elasticsearch-plugin install x-pack

bin/kibana-plugin install x-pack

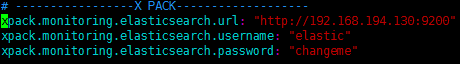
bin/logstash-plugin install x-pack

基本上各个组件的默认配置就能满足使用，但是有些地方还是需要简单配置。

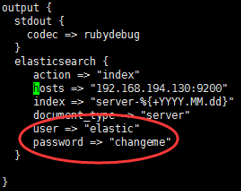
Kibana中，添加如下配置：配置kibana访问elasticsearch的用户名以及密码



Logstash中添加如下配置：



logstash.conf 配置文件：



然后分别启动运行。

x-pack默认只有一个月的有效期，需要我们去官方网站购买，或者注册申请免费使用，每次申请免费使用一年，过期可以再次申请。这个申请只能使用x-pack的ELK集群监控功能。

<https://register.elastic.co/xpack_register>

之后，会在你填写的邮箱发给你下载注册的json文件。

得到之后改下名字 改成license.json

修改license

curl -XPUT -u elastic 'http://192.168.194.130:9200/\_xpack/license?acknowledge=true' -H "Content-Type: application/json" -d @license.json

查看license

curl -XGET -u elastic:changeme 'http://192.168.194.130:9200/\_license'

## ELK集群部署

今天针对elasticsearch的集群进行了简单部署

集群节点为二，分别在两台虚拟机上进行部署。

首先节点一的配置文件如下：

cluster.name: hycTest

node.name: node-1

node.master: true

node.data: true

path.data: /usr/elk/elkbackup/ESdata

path.logs: /usr/elk/elkbackup/ESlogs

network.host: 0.0.0.0

http.port: 9200

transport.tcp.port: 9300

discovery.zen.ping.unicast.hosts: ["192.168.194.130"]

discovery.zen.ping\_timeout: 30s

discovery.zen.minimum\_master\_nodes: 1

然后节点二配置如下：

node.name: node-2

node.master: false

node.data: true

path.data: /usr/elk/elasticsearch-5.5.0/data

path.logs: /usr/elk/elasticsearch-5.5.0/logs

network.host: 0.0.0.0

http.port: 9200

transport.tcp.port: 9300

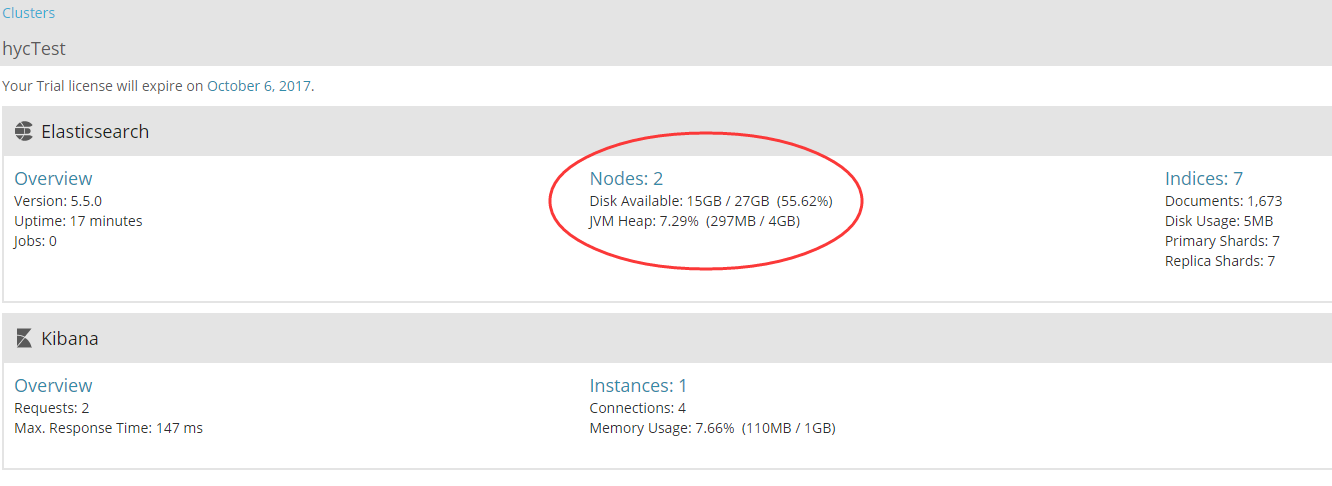
discovery.zen.ping.unicast.hosts: ["192.168.194.130"]

discovery.zen.ping\_timeout: 30s

discovery.zen.minimum\_master\_nodes: 1

然后分别启动节点一，节点二

启动kibana观察节点数：



节点数为二表示成功。

需要注意的地方：

关于集群主节点数量的配置，为了防止脑裂现象，官方的推荐值是(N/2)+1，其中N是具有master资格的节点的数量，具有master节点资格的，也就是需要配置node.master为true以及在各个节点的discovery.zen.ping.unicast.hosts配置中间加上它的IP。

踩过的坑：

2017-09-06T10:58:40,809][INFO ][o.e.d.z.ZenDiscovery ] [node-2] failed to send join request to master [{node-1}{YHKfuvf5SO-vAYf6kje3WA}{L6akxU9iTO2FpoeyXaoP5A}{192.168.194.130}{192.168.194.130:9300}{ml.enabled=true}], reason [RemoteTransportException[[node-1][192.168.194.130:9300][internal:discovery/zen/join]]; nested: ConnectTransportException[[node-2][192.168.194.131:9300] connect\_timeout[30s]]; nested: IOException[没有到主机的路由: 192.168.194.131/192.168.194.131:9300]; nested: IOException[没有到主机的路由]; ]

查看两台主句是否能够telnet互通，检查防火墙是否关闭。

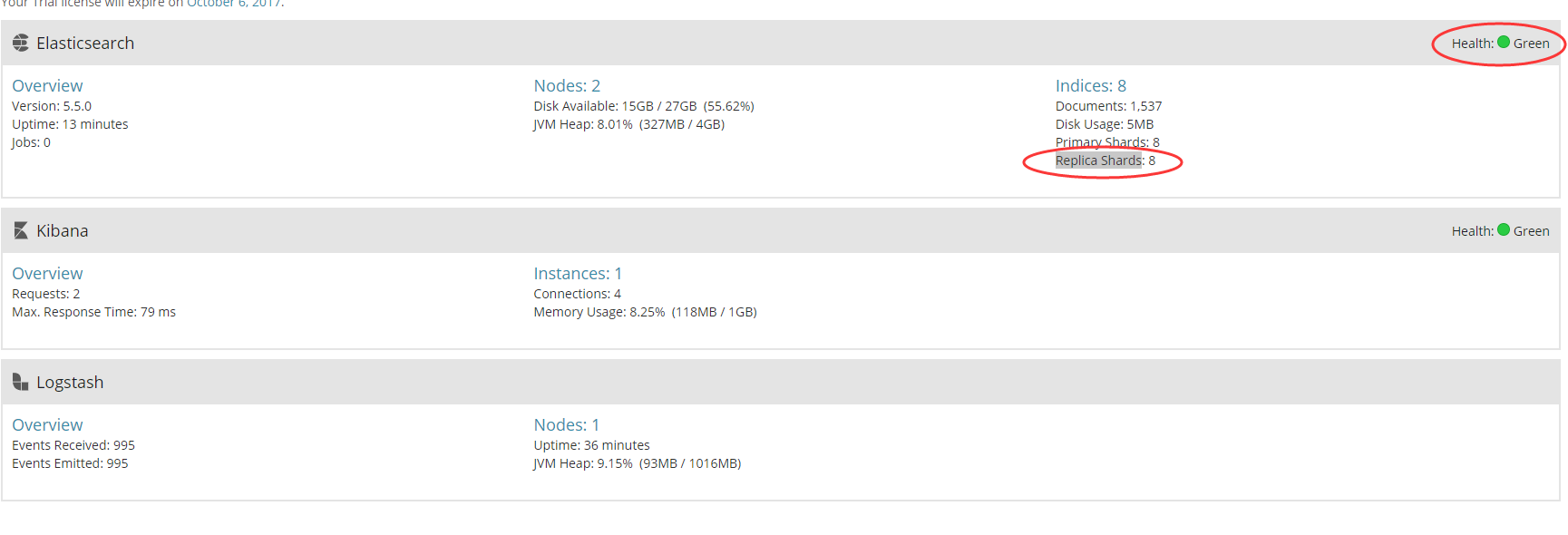
还有当node.data 为true的节点不大于1的时候，集群的状态始终将为黄色，原因如下（官方解释）：

The cluster health status is: green, yellow or red. On the shard level, a red status indicates that the specific shard is not allocated in the cluster, yellow means that the primary shard is allocated but replicas are not, and green means that all shards are allocated. The index level status is controlled by the worst shard status. The cluster status is controlled by the worst index status.

根据上面所说，我们知道了，集群状态为yellow的状况为集群没有副本分片（Replica Shards）

但是为什么集群node.data 为true的节点不大于1的时候集群无法创建副本分片？个人猜想，原因可能是：如果集群数据节点为1，当发生宕机情况时，没有数据节点在可以备份，此时状态和单机情况一样，这也是单机情况健康状态始终未yellow的原因。

最后我将主节点的node.data 设置为true，使主节点页充当数据节点，集群状态最后变成green并且创建了副本分片



这样一个最简单的集群就搭建好了，那么问题来了，如何增加节点。

增加一个数据节点到现有的集群。

再来一个新的虚拟机，装好jdk1.8和elasticsearch，ES配置如下：

cluster.name: testHu

node.name: node-3

node.master: false

node.data: true

path.data: /usr/elk/elasticsearch-5.5.0/data

path.logs: /usr/elk/elasticsearch-5.5.0/logs

network.host: 0.0.0.0

http.port: 9200

transport.tcp.port: 9300

discovery.zen.ping.unicast.hosts: ["192.168.194.130"]

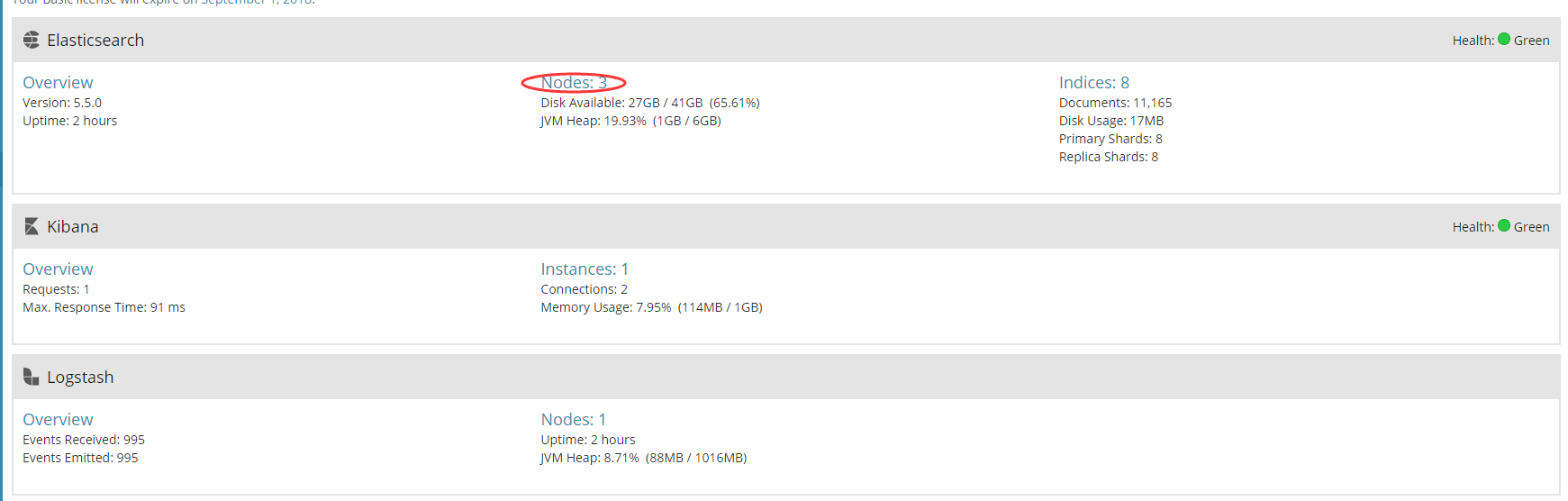
discovery.zen.ping\_timeout: 30s

discovery.zen.minimum\_master\_nodes: 1

**配置和前面两个节点配置差不多，然后需要注意的是，如果之前的节点安装了x-pack插件，我们在启动新的节点之前，也需要安装x-pack，否则启动将会失败。**

**安装好x-pack插件之后，再次启动ES，发现我们的新的ES节点的license自动和集群的license同步了，并且如果我们的集群主节点的license集群的所有节点的license将全部更新。**

**启动之后我们去kibana的Monitoring中观看下集群情况。**



ELK的集群自动化做的非常好，完全傻瓜式配置，值得学习。

## ELK 报警插件ElastAlert

今天研究了一下ElastAlert一个用于监控elasticsearch数据以及报警的应用。由于ES中的报警插件集成在了x-pack中，并且这个功能不是免费的，所以我们需要其他途径来实现报警。于是我找到了ElastAlert

由于ElastAlert是基于python开发的，所以我们的服务器上需要python环境，但是幸运的是，我用的是Centos7系统所以里面自带python2.7+的环境，而ElastAlert近支持运行在python2.6以及2.7中。

安装步骤大概如下：

#安装git，用来将ElastAlert用git上克隆源码

yum -y install git

#拉取ElastAlert源码

git clone https://github.com/Yelp/elastalert.git

#安装python的pip安装命令

cd elastalert

yum -y install python-pip

#安装相关插件

pip install --upgrade pip

pip install "setuptools>=11.3"

#安装yaml模块

wget http://pyyaml.org/download/libyaml/yaml-0.1.5.tar.gz

cd yaml-0.1.5

./configure --prefix=/usr/local/

maker&&make install

#编译ElastAlert代码

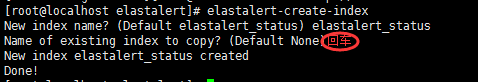
python setup.py install

#安装ElastAlert的命令

pip install -r requirements.txt

然后，我们启动ELK，用来调试ElastAlert。

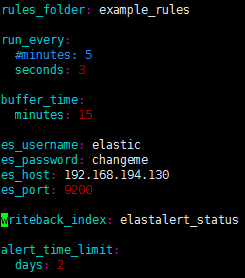
ELK启动后，我们来创建ElastAlert在elasticsearch中的索引。



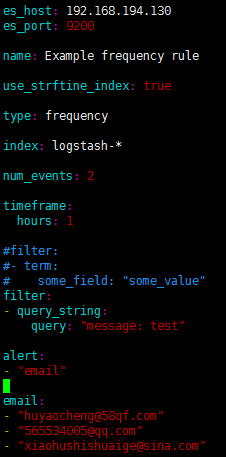
期间有很多次失败或者操作失误，没关系，我们可以去elasticsearch中删除索引，重新创建。

接下来我们需要配置两个文件，一个是主配置文件config.yaml 一个是rule.yaml

config.yaml配置详情：



rule.yaml配置详情：



alert：为email表示我将用邮件的形式报警通知

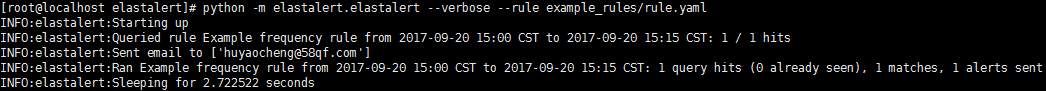
email：表示要发送的邮箱

配置完成之后我们可以测试下我们的配置：



如果没有问题，我们就可以启动ElastAlert了，在conf.yaml所在目录下，执行命令：

python -m elastalert.elastalert --verbose --rule example\_rules/rule.yaml



然后我们可以开始测试了。

注意：一开始刚刚编译好的程序，是没有conf.yaml文件的，但是有一个config.yaml.example，这个是配置文件的模板，我们将他复制一份并且更名为conf.yaml然后在其中做配置即可。

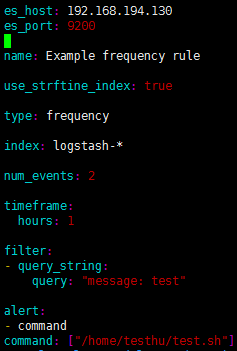
rule.yaml是我们自定义的查询报警规则配置文件，它的模板文件是example\_rules/example\_frequency.yaml

rule.yam配置文件中，我们查询的条件在filter中，报警的阈值是num\_events是2，也就是说，当我们想elasticsearch发送两次test时，ElastAlert将会向我们指定的邮箱发送邮件。

有可能邮件已经发送但是邮箱没有显示，可能是邮箱将其拦截，可配置你说使用的邮箱，将ElastAlert的邮件地址添加白名单即可。

commend发送：

比较灵活的方式，可以执行脚本。



这里我自己写了一个测试脚本，下本内容如下。



很简单，就是执行一个jar包，这里需要注意，声明一个shell一要在首行添加

#!/bin/bash

## Elasticsearch安装Head插件

Elasticsearch5.0以后的Head插件安装还是非常麻烦的，下面贴出安装步骤。

Linux系统：

首先安装git: yum -y install git

然后从Git上将Head源码拉下来:

git clone git://github.com/mobz/elasticsearch-head.git

然后安装node.js，由于node.js下载的包是xz的，所以我们需要安装xz解析器

yum -y install xz

然后我们将下载的node.js包解压，并配置环境变量

xz -d node\*.tar.xz

tar -xvf node\*.tar

编辑/etc/profile,添加

export NODE\_HOME=/usr/elk/node-v6.9.1-linux-x64

export PATH=$PATH:$NODE\_HOME/bin

source /etc/profile

校验一下是否安装成功 node -v npm -v

然后开始安装grunt

#建立软连接,否则root找不到npm

npm init

#生成package.json否则报错

sudo npm install -g grunt-cli

检验一下是否成功：grunt -version

目录：head/Gruntfile.js 第93行

connect: {

server: {

options: {

port: 9100,

hostname: '\*',

base: '.',

keepalive: true

}

}

}

最后在Head目录下执行

npm install

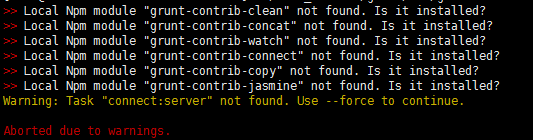
然后在elasticsearch.yml中增加新的参数，这样head插件可以访问es http.cors.enabled: true

http.cors.allow-origin: "\*"

修改完后重启es，然后启动Head插件

grunt server

有可能报已下错：



这是由于这些是Gruntfile.js中引用的，分别下载。

npm install grunt-contrib-copy

npm install grunt-contrib-concat

npm install grunt-contrib-jasmine(这个一直下不成功，报错。但是不下也能启动)

npm install grunt-contrib-clean

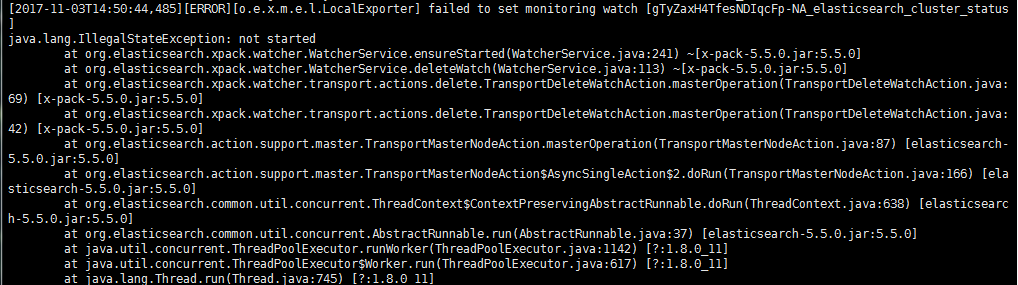
npm install grunt-contrib-connect

npm install grunt-contrib-watch

elasticsearch5.0以后head插件不能放在es的plugins目录下了，否则es将不能启动。

## Elasticsearch 平时遇到的问题

今天启动ES集群报错



x-pack插件中的监控模块不能启动了

之后我将ES中的所有历史监测数据删除之后，就可以正常启动了，具体原因还找不到。

后来我发现我的Linux虚拟机的时间比本地时间晚了3天，我猜也许是这个原因导致的，将时间设定正常后，可以正常启动了

## Elasticsearch Analyzer使用

今天主要学习了Elasticsearch的Analyzer的使用以及配置，并且针对这个功能自己做了一个小小的需求，根据拼音分词器（pinyin）或者IK分词器（中文分词）来做到使用trem的快速查询以及拼音搜索自动补齐的功能。

首先需要了解什么是Analyzer：Analyzer是将文本（如任何电子邮件的正文）转换 为添加到倒排索引以进行搜索的标记或术语的过程。

Analyzer分为三个模块，或者说当有一段数据需要进行分词的时候，Analyzer都做了些什么？

Analyzer 按顺序做三件事：

1.使用 CharacterFilter 过滤字符（此处的过滤器的任务是将待分词的文本进行特殊字符或者HTML标签以及数字等等一些字符进行过滤）

2.使用 Tokenizer 分词（讲接受的文本按照指定的方式进行分词操作，ES默认我分词器是Standard）

3.使用 TokenFilter 过滤词（用来对分词器处理完的词流再次进行处理，如[lowercase](https://www.elastic.co/guide/en/elasticsearch/reference/5.3/analysis-lowercase-tokenfilter.html)令牌过滤器将所有令牌转换为小写这种后续操作）

要想做到trem查询的拼音自动补齐，主要需要做的步骤是在index中自定义我们的Analyzer，并在mapping中为我们需要Analyzer处理的字段，配置我们的自定义分词器。

首先，我们需要在我们的index中自定义我们的Analyzer，Analyzer得配置模板如下：

PUT my\_index

{

"index": {

"analysis": {

"analyzer":{

"my-analyzer ":{

"type":"custom",

"char\_filter":"myCharFilter" ,

"tokenizer":"myTokenizer" ,

"filter":["myFilter"]

}

},

"filter": {

"myFilter":{...}

},

"tokenizer": {

"myTokenizer":{...}

},

"char\_filter": {

"myCharFilter":{...}

}

}

}

}

然后我们需要配置我们的mapping，将index的mapping中的指定分词的字段，配置成为我们自定义的分词器：

PUT /my\_index/my\_type/\_mapping

{

"type":{

"\_all": {

"enabled": false

},

"properties":{

"title":{

"analyzer": "my-analyzer ",

"search\_analyzer": "my-analyzer ",

"type":"text"

},

"age":{

"type":"long"

}

}

}

}

上面的两步也可以合并为一步：

PUT my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_ analyzer ": { …}

}

}

},

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "text",

"analyzer": "my-analyzer ",

"search\_analyzer": "my-analyzer ",

}

}

}

}

}

这样自定义分词器analyzer以及mapping中的待分词字段就设置完成了，接下来我们需要查看分词结果了，我们该如何做？

GET my\_index/\_analyze

{

"analyzer": "my\_analyzer",

"text": "我是需要分词的"

}

这样我们就能看到我们分词器分词后的效果了，接下来就是添加数据，开始搜索了。

那么，定义讲完了，接下来我们该实战了。

首先，我们为我们的index自定义一个分词器，该分词器需要将我们传输的文本，进行拼音分词以及中文分词，并且为了达到自动补齐的体验效果，我们需要用到Edge NGram（edge\_ngram）过滤器，该过滤器能够将每个分词进行从min\_gram长度到max\_gram长度的切分，形成行的分词（如[abc]，将会分成[a,b,c,ab,bc,abc]）,用户在输入拼音或者文字的时候，会根据输入的字符进行自动补齐，当用户输入的字符越多，也就越能返回用户想要查询的结果。

PUT <http://192.168.194.130:9200/index/>

{

"index": {

"analysis": {

"analyzer":{

"my-pinyin":{

"type":"custom",

"tokenizer":"keyword",

"filter":[

"autocomplete\_filter",

"pinyin\_first\_letter\_and\_full\_pinyin\_filter"

]

}

},

"filter": {

"pinyin\_first\_letter\_and\_full\_pinyin\_filter" : {

"type" : "pinyin",

"keep\_first\_letter" : true,

"keep\_full\_pinyin" : false,

"keep\_none\_chinese" : true,

"keep\_original" : true,

"keep\_joined\_full\_pinyin": true,

"limit\_first\_letter\_length" : 16,

"lowercase" : true,

"trim\_whitespace" : true,

"keep\_none\_chinese\_in\_first\_letter" : true

},

"autocomplete\_filter":{

"type":"edge\_ngram",

"min\_gram":1,

"max\_gram":15

}

}

}

}

}

第二步，设置index的mapping，将带分词的字段设置为我们自定义的分词器，其中analyzer为ES在插入数据创建倒排索引的时候用的那个分词器。而search\_analyzer为用户进行搜索的时候（如match查询）针对搜索跳进进行分词使用的分词器，默认search\_analyzer与analyzer相同，如果不相同，查询将不准确。

PUT <http://192.168.194.130:9200/index/type/_mapping>

{

"type":{

"\_all": {

"enabled": false

},

"properties":{

"title":{

"analyzer": "my-pinyin",

"search\_analyzer": "my-pinyin",

"type":"text"

},

"name":{

"analyzer": "my-pinyin",

"search\_analyzer": "my-pinyin",

"type":"text"

},

"age":{

"type":"long"

}

}

}

}

第三步，验证分词效果：

GET <http://192.168.194.130:9200/index/_analyze>

{

"text":"小胡是帅哥",

"analyzer":"my-pinyin"

}

返回结果如下：

{

"tokens": [

{

"token": "小",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "小胡",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "小胡是",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "小胡是帅",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "小胡是帅哥",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "x",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xi",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xia",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiao",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaoh",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohu",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohus",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohush",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushi",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushis",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushish",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushishu",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushishua",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushishuai",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xiaohushishuaig",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "x",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xh",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xhs",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xhss",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

},

{

"token": "xhssg",

"start\_offset": 0,

"end\_offset": 5,

"type": "word",

"position": 0

}

]

}

第四部，插入数据：

PUT <http://192.168.194.130:9200/index/type/1>

{

"title":"小胡是帅哥",

"name":"胡耀成",

"age":26

}

第五步，trem查询：

GET <http://192.168.194.130:9200/index/type/_search>

{

"query": {

"term":{

"title":"胡是"

}

}

}

**扩展：可以添加一个属性"profile":true用来查看查询效率等明细。**

**扩展：trem查询与Match查询的区别，trem查询为精确查询，match查询为匹配查询，trem查询在查询过程中不会对查询内容进行分词，因此输入什么就回到倒排索引中检所什么，因此trem查询又叫精确查询，而match查询会在查询过程中先对查询内容进行分词，然后再对每个分词做查询，因此trem查询在效率上会好一点。**

## Elasticsearch Field Collapsing（字段折叠）+ inner\_hits

使用 term agg+ top hits agg组合两种聚和的方式可以实现Field Collapsing（字段折叠）+ inner\_hits的功能，不过也有一些局限性，比如，不能分页；结果不够精确（top term+top hits，es 的聚合实现选择了牺牲精度来提高速度）；数据量大的情况下，聚合比较慢，影响搜索体验。  
   
而新的的字段折叠的方式是怎么实现的的呢，有这些要点：

1.折叠+取 inner\_hits 分两阶段执行（组合聚合的方式只有一个阶段），所以 top hits 永远是精确的。

2.字段折叠只在 top hits 层执行，不需要每次都在完整的结果集上对为每个折叠主键计算实际的 doc values 值，只对 top hits 这小部分数据操作就可以，和 term agg 相比要节省很多内存。

3.因为只在 top hits 上进行折叠，所以相比组合聚合的方式，速度要快很多。

折叠 top docs 不需要使用全局序列（global ordinals）来转换 string，相比 agg 这也节省了很多内存。

4.分页成为可能，和常规搜索一样，具有相同的局限，先获取 from+size 的内容，再合并。

search\_after 和 scroll 暂未实现，不过具备可行性。

5. 折叠只影响搜索结果，不影响聚合，搜索结果的 total 是所有的命中纪录数，去重的结果数未知（无法计算）。

## Elasticsearch Search After分页查询

结果的分页可以通过使用这个来完成from，size但是当达到深度分页时，成本会变得过高。在index.max\_result\_window缺省10000是一种保障，搜索请求采取堆内存和时间成正比from + size。该[滚动](https://www.elastic.co/guide/en/elasticsearch/reference/5.5/search-request-scroll.html) API被推荐用于高效深滚动滚动，但上下文昂贵，因此不建议使用它的实时用户的请求。该search\_after参数通过提供实时光标来绕过这个问题。这个想法是使用前一页的结果来帮助检索下一页。

还有一个和翻页相关的问题，就是深度分页 ，是个老大难的问题，因为需要全局排序（ number\_of\_shards \* (from + size) ），所以需要消耗大量内存，以前的 es 没有限制，有些同学翻到几千页发现 es 直接内存溢出挂了，后面 elasticsearch 加上了限制， from+size 不能超过 1w 条，并且如果需要深度翻页，建议使用 scroll 来做。

　　但是 scroll 有几个问题，第一个是没有顺序，直接从底层 segment 进行遍历读取，第二个实时性没法保证， scroll 操作有状态， es 会维持 scroll 请求的上下文一段时间，超时后才释放，另外你在 scroll 过程中对索引数据进行了修改了，这个时候 scroll接口是拿不到的，灵活性较差， 现在有一个新的 Search After 机制，其实和 scroll 类似，也是游标的机制，它的原理是对文档按照多个字段进行排序，然后利用上一个结果的最后一个文档作为起始值，拿 size 个文档，一般我们建议使用 \_uid 这个字段，它的值是唯一的 id 。

例子：

首先在查询的时候我们需要指定几个字段去排序，这些排序结果将会作为search\_after的下一页的搜索参数：

GET http://192.168.194.130:9200/bank/account/\_search

{

"query":{

"match":{

"address":"Terrace"

}

},

"sort":[

{"balance":"asc"},

{"\_uid":"desc"}

]，

“size”：1

}

为了返回结果不会太长，便于查看，我们在此只返回一条数据

结果如下：

{

"took": 109,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 16,

"max\_score": null,

"hits": [

{

"\_index": "bank",

"\_type": "account",

"\_id": "411",

"\_score": null,

"\_source": {

"account\_number": 411,

"balance": 1172,

"firstname": "Guzman",

"lastname": "Whitfield",

"age": 22,

"gender": "M",

"address": "181 Perry Terrace",

"employer": "Springbee",

"email": "guzmanwhitfield@springbee.com",

"city": "Balm",

"state": "IN"

},

"sort": [

1172,

"account#411"

]

}

]

}

}

然后，当我们查询下一页的时候，我们将使用search\_after，其中search\_after中的两个参数为前一次查询中sort返回的两个参数，这将会在search\_after中座位查询下一页使用

GET http://192.168.194.130:9200/bank/account/\_search

{

"query":{

"match":{

"address":"Terrace"

}

},

"search\_after":[1172,"account#411"],

"sort":[

{"balance":"asc"},

{"\_uid":"desc"}

] ，

“size”：10

}

## Elasticsearch Search Template(搜索模板)

简单来说就是定义一套固定的搜索条件的搜索，将有变化的部分参数化，并保存为模板，当使用时，直接调用模板传入参数即可。

具体的规格语法看官网，这里只关注如何保存模板以及调用模板。

创建模板的API：

POST \_search/template/<templatename>

{

"template": {

"query": {

"match": {

"title": "{{query\_string}}"

}

}

}

}

调用模板API：

GET \_search/template

{

"id": "<templateName>",

"params": {

"query\_string": "使用模板查询传入的查询参数"

}

}

删除模板API：

DELETE \_search/template/<templatename>

**扩展：查询模板在创建的时候不能够指定index和type，但是在调用查询模板的时候我们可以指定indez和type，这样避免了多余的索引以及分片的检索，提高了查询效率。**

**查询时调用模板时指定index和type的用例：**

**curl -XGET 'localhost:9200/index/type/\_search/template?pretty' -H 'Content-Type: application/json' -d'**

**{**

**"id": "test",**

**"params": {**

**"query\_string": "小"**

**}**

**}**

**'**

## Elasticsearch Multi Search API

Multi Search API允许在相同的API中执行多个搜索请求。它的终点是\_msearch。

请求的格式类似于批量API格式，并使用换行符分隔的JSON（NDJSON）格式。结构如下（如果特定的搜索结束重定向到另一个节点，则该结构被特别优化以减少解析）：

header\n

body\n

header\n

body\n

报头部分包括index/indices来搜索，可选的（映射）类型有，search\_type，preference，和routing。主体包括典型的搜索体请求（包括query，aggregations，from，size，等等）。这里是一个例子：

$ cat requests //编辑请求体文件

{"index" : "test"}

{"query" : {"match\_all" : {}}, "from" : 0, "size" : 10}

{"index" : "test", "search\_type" : "dfs\_query\_then\_fetch"}

{"query" : {"match\_all" : {}}}

{}

{"query" : {"match\_all" : {}}}

{"query" : {"match\_all" : {}}}

{"search\_type" : "dfs\_query\_then\_fetch"}

{"query" : {"match\_all" : {}}}

//发送请求文件

$ curl -H "Content-Type: application/x-ndjson" -XGET localhost:9200/\_msearch --data-binary "@requests"; echo

注意，上面包括了一个空头的例子（也可以是没有任何内容的），它也被支持。

响应返回一个responses数组，其中包含每个搜索请求的搜索响应和状态码，以匹配原始多重搜索请求中的顺序。如果该特定搜索请求完全失败，error则会返回带有消息和相应状态码的对象，而不是实际的搜索响应。

端点还允许在URI本身中对index/indices和type/types进行搜索，在这种情况下，它将被用作默认值，除非在头中另外明确定义。例如：

GET twitter/\_msearch

{}

{"query" : {"match\_all" : {}}, "from" : 0, "size" : 10}

{}

{"query" : {"match\_all" : {}}}

{"index" : "twitter2"}

{"query" : {"match\_all" : {}}}

以上将twitter针对所有未定义索引的请求的索引执行搜索，最后一个将针对twitter2索引执行。

该search\_type可以以类似的方式被设置为适用于全球所有搜索请求。

msearch的max\_concurrent\_searches请求参数可以用来控制多搜索api将执行的最大并发搜索数量。此默认值基于数据节点的数量和默认搜索线程池大小。

模板支持

就像\_search资源的搜索模板中所述，\_msearch也提供对模板的支持。提交他们如下所示：

GET \_msearch/template

{"index" : "twitter"}

{ "inline" : "{ \"query\": { \"match\": { \"message\" : \"{{keywords}}\" } } } }", "params": { "query\_type": "match", "keywords": "some message" } }

{"index" : "twitter"}

{ "inline" : "{ \"query\": { \"match\_{{template}}\": {} } }", "params": { "template": "all" } }

您也可以创建搜索模板：

POST /\_search/template/my\_template\_1

{

"template": {

"query": {

"match": {

"message": "{{query\_string}}"

}

}

}

}

POST /\_search/template/my\_template\_2

{

"template": {

"query": {

"term": {

"{{field}}": "{{value}}"

}

}

}

}

并稍后在\_msearch中使用它们：

GET \_msearch/template

{"index" : "main"}

{ "id": "my\_template\_1", "params": { "query\_string": "some message" } }

{"index" : "main"}

{ "id": "my\_template\_2", "params": { "field": "user", "value": "test" } }

## Elasticsearch Validate API

Validate API是Elasticsearch用于校验我们的搜索渔具是否正确，能够允许用户在不执行的情况下验证潜在的昂贵查询，语法同search语法，但是应用的关键子是\_validate/query而不是\_search

基本语法如下，可以使用url查询，或者dsl查询

GET my\_index/my\_type/\_validate/query

URL参数有以下几个：

explain=true 当查询失败的时候，提供了更为详细的失败原因

rewrite=true 显示更详细的将要执行的实际Lucene的查询。

all\_shards=true 默认情况下，请求只在一个单独的分片上执行，这是随机选择的，获得所有可用分片的响应。

实际用例：

GET http://192.168.194.130:9200/twitter/tweet/\_validate/query?explain=true&rewrite=true&all\_shards=true

{

"valid": true,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"explanations": [

{

"index": "twitter",

"shard": 0,

"valid": true,

"explanation": "+ConstantScore(user:kimchy) #ConstantScore(MatchNoDocsQuery(\"empty BooleanQuery\"))"

},

{

"index": "twitter",

"shard": 1,

"valid": true,

"explanation": "+ConstantScore(user:kimchy) #ConstantScore(MatchNoDocsQuery(\"empty BooleanQuery\"))"

},

{

"index": "twitter",

"shard": 2,

"valid": true,

"explanation": "+ConstantScore(user:kimchy) #ConstantScore(MatchNoDocsQuery(\"empty BooleanQuery\"))"

},

{

"index": "twitter",

"shard": 3,

"valid": true,

"explanation": "ConstantScore(user:kimchy)"

},

{

"index": "twitter",

"shard": 4,

"valid": true,

"explanation": "+ConstantScore(user:kimchy) #ConstantScore(MatchNoDocsQuery(\"empty BooleanQuery\"))"

}

]

}

## Elasticsearch Bulk API

**注意：调用\_bluk 接口时 最后一行一定要回车换一行，不然最后一条数据会不能录入ES或不能生效。例如下面的例子，批量导入数据：**

**curl -XPOST 'localhost:9200/\_bulk?pretty' -H 'Content-Type: application/json' -d'**

**{"create":{"\_index":"test","\_type":"test","\_id":1}}**

**{"title": "华硕笔记本","content": "华硕笔记本"}**

**{"create":{"\_index":"test","\_type":"test","\_id":2}}**

**{"title": "江南笔记本","content": "江南笔记本"}**

**{"create":{"\_index":"test","\_type":"test","\_id":3}}**

**{"title": "话说笔记本电脑","content": "话说笔记本电脑"}**

**{"create":{"\_index":"test","\_type":"test","\_id":4}}**

**{"title": "小跳蛙水彩笔","content": "小跳蛙水彩笔"}**

**{"create":{"\_index":"test","\_type":"test","\_id":5}}**

**{"title": "华硕平板电脑","content": "华硕平板电脑"}**

**' //此处注意，最后一条数据要添加回车换行到此处**

## Elasticsearch 多字段配合分词权重进行查询

今天看了几篇文章，因此有了想了解这方面知识的想法：

https://www.cnblogs.com/leixingzhi7/p/6903938.html

https://elasticsearch.cn/question/297

通过这两篇技术文章，也享受动的实现一个能够做到模糊、建议、纠错的搜索，尝试了一下Elasticsearch的suggesters，但是针对中文的支持还是很坑的，所以就绕路而行试试其他的方式，感觉通过match查询中的multi\_match多字段查询，效果也是不错的。

首先重中之重，还是需要配置分词器的使用，由于我们中文搜索不仅要考虑文字上的匹配，还要考虑错别字纠错，同音字纠错，所以这里我们分词方面不仅需要选择IK中文分词器，还要选择pinyin拼音分词器。

由于在中文中，词的概念通常最短距离为2，拼音也一样，为了保证查询多音字词时的准确性，我们添加了一个分词长度多虑器length，将分词后的词长度哦控制在了2以上，保证了查询的准确性。

另外考虑到多音字的错别字纠错概念，引入了拼音分词器，但是中文博大精深，同音情况下可能有很多同音词，因此这里考虑到了字段分词权重的功能，在mapping中将同一文本字段，另存为使用不同分词结果的属性，并且保留原来字段。

Analyzer与mapping设置如下：

{

"settings": {

"index": {

"analysis": {

"analyzer":{

"my\_analyzer":{

"type":"custom",

"tokenizer":"ik\_max\_word",

"filter":[

"en\_len\_filter",

"pinyin\_first\_letter\_and\_full\_pinyin\_filter"

]

},

"my\_ik":{

"type":"custom",

"tokenizer":"ik\_max\_word",

"filter":[

"cn\_len\_filter"

]

}

},

"filter": {

"pinyin\_first\_letter\_and\_full\_pinyin\_filter" : {

"type" : "pinyin",

"keep\_first\_letter" : true,

"keep\_full\_pinyin" : false,

"keep\_none\_chinese" : true,

"keep\_original" : true,

"keep\_joined\_full\_pinyin": true,

"limit\_first\_letter\_length" : 16,

"lowercase" : true,

"trim\_whitespace" : true,

"keep\_none\_chinese\_in\_first\_letter" : true

},

"cn\_len\_filter":{

"type":"length",

"min":2

},

"en\_len\_filter":{

"type":"length",

"min":2

}

}

}

}

},

"mappings": {

"\_default\_":{

"\_all": { "enabled": false }

},

"test" : {

"properties" : {

"title" : {

"type": "text",

"fields":{

"cn":{

"type":"text",

"analyzer": "my\_ik",

"search\_analyzer": "my\_ik"

},

"en":{

"type":"text",

"analyzer": "my\_analyzer",

"search\_analyzer": "my\_analyzer"

}

}

}

}

}

}

}

Mapping中属性title下有另外添加了两个属性cn、en，并且分别使用了ik分词器和pinyin分词器，这里的拼音分词器采用的是ik分词然后通过pinyin分词器处理成拼音。这样在查询时，我们可以通过同时查询title，title.cn，title.en来实现针对拼音，中文词汇，搜索条件的同时查询，并且查询时可以通过为不同字段分配不同权重，来影响你的排序结果，比如中文词汇如果命中我希望他最靠前，那就将title.cn字段的权重调的比其他字段高一点。

测试数据如下：

POST <http://192.168.194.130:9200/_bulk>

{"index":{"\_index":"test","\_type":"test","\_id":1}}

{"title": "华硕笔记本","content": "华硕笔记本"}

{"index":{"\_index":"test","\_type":"test","\_id":2}}

{"title": "江南笔记本","content": "江南笔记本"}

{"index":{"\_index":"test","\_type":"test","\_id":3}}

{"title": "话说笔记本电脑","content": "话说笔记本电脑"}

{"index":{"\_index":"test","\_type":"test","\_id":4}}

{"title": "小跳蛙水彩笔","content": "小跳蛙水彩笔"}

{"index":{"\_index":"test","\_type":"test","\_id":5}}

{"title": "华硕平板电脑","content": "华硕平板电脑"}

{"index":{"\_index":"test","\_type":"test","\_id":6}}

{"title": "华硕笔记本电脑","content": "华硕笔记本电脑"}

查询语句：

POST <http://192.168.194.130:9200/test/test/_search?pretty>

{

"query":{

"multi\_match": {

"type": "best\_fields",

"query": "笔记",

"fields": [ "title", "title.cn^10", "title.en^5" ]

}

}

}

这里我们将title.cn权重分为10，title.en权重分为5，title为默认1。搜索结果如下：

POST http://192.168.194.130:9200/test/test/\_search?pretty

{

"took": 35,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 4.7000365,

"hits": [

{

"\_index": "test",

"\_type": "test",

"\_id": "2",

"\_score": 4.7000365,

"\_source": {

"title": "江南笔记本",

"content": "江南笔记本"

}

},

{

"\_index": "test",

"\_type": "test",

"\_id": "6",

"\_score": 4.2011933,

"\_source": {

"title": "华硕笔记本电脑",

"content": "华硕笔记本电脑"

}

},

{

"\_index": "test",

"\_type": "test",

"\_id": "3",

"\_score": 2.824934,

"\_source": {

"title": "话说笔记本电脑",

"content": "话说笔记本电脑"

}

},

{

"\_index": "test",

"\_type": "test",

"\_id": "1",

"\_score": 2.565813,

"\_source": {

"title": "华硕笔记本",

"content": "华硕笔记本"

}

}

]

}

}

目前分词器还待优化，还会有很多的冗余，比如数字字母组合，单词等，这些ik与pinyin做的还不够好。

顺序也还是不够尽人意，但是我们后续可以通过修改评分规则的方式来去定义它。

针对昨天的多字段纠错、建议权重分配查询，发现了以下几点不足：

1.首先是在使用pinyin分子过滤器使发现它对于中英文混合 中文数字英文混合的粉刺效果不友好，导致许多冗余垃圾词项。

2.顺序有时达不到满意效果，例如以下的数据，我搜 “话说” 居然有 “华硕“的文档排在了话说前面

分词器，以及mapping配置如下：这里我添加了一个filter，拼音处理那块用来过滤掉费拼音的词（数字字母组合，英文），因为针对非汉字的拼音过滤效果很差，索性抛弃，毕竟拼音只是用来汉语纠错，以及搜索建议用的。

PUT http://192.168.194.130:9200/test

{

"settings": {

"index": {

"analysis": {

"analyzer":{

"my\_analyzer":{

"type":"custom",

"tokenizer":"ik\_smart",

"filter":[

"pinyin\_first\_letter\_and\_full\_pinyin\_filter",

"en\_len\_filter",

"extract\_numbers"

]

},

"my\_ik":{

"type":"custom",

"tokenizer":"ik\_smart",

"filter":[

"cn\_len\_filter"

]

}

},

"filter": {

"pinyin\_first\_letter\_and\_full\_pinyin\_filter" : {

"type" : "pinyin",

"keep\_separate\_first\_letter" : false,

"keep\_full\_pinyin" : false,

"keep\_joined\_full\_pinyin": true,

"keep\_original" : false,

"limit\_first\_letter\_length" : 16,

"lowercase" : true,

"keep\_none\_chinese":true,

"keep\_none\_chinese\_together":true,

"none\_chinese\_pinyin\_tokenize":true,

"keep\_none\_chinese\_in\_first\_letter":true,

"keep\_first\_letter":false

},

"cn\_len\_filter":{

"type":"length",

"min":2

},

"en\_len\_filter":{

"type":"length",

"min":2

},

"extract\_numbers" : {

"type" : "keep\_types",

"types" : [ "CN\_WORD" ]

}

}

}

}

},

"mappings": {

"\_default\_":{

"\_all": { "enabled": false }

},

"test" : {

"properties" : {

"title" : {

"type": "text",

"analyzer": "whitespace",

"search\_analyzer": "whitespace",

"fields":{

"cn":{

"type":"text",

"analyzer": "my\_ik",

"search\_analyzer": "my\_ik"

},

"pinyin":{

"type":"text",

"analyzer": "my\_analyzer",

"search\_analyzer": "my\_analyzer"

}

}

}

}

}

}

}

数据

POST http://192.168.194.130:9200/\_bulk

{"index":{"\_index":"test","\_type":"test","\_id":1}}

{"title": "华硕笔记本"}

{"index":{"\_index":"test","\_type":"test","\_id":2}}

{"title": "江南笔记本"}

{"index":{"\_index":"test","\_type":"test","\_id":3}}

{"title": "话说笔记本电脑"}

{"index":{"\_index":"test","\_type":"test","\_id":4}}

{"title": "小跳蛙水彩笔"}

{"index":{"\_index":"test","\_type":"test","\_id":5}}

{"title": "华硕平板电脑"}

{"index":{"\_index":"test","\_type":"test","\_id":6}}

{"title": "华硕笔记本电脑 16G Inter"}

{"index":{"\_index":"test","\_type":"test","\_id":7}}

{"title": "华硕笔记本电脑 8G Inter"}

{"index":{"\_index":"test","\_type":"test","\_id":8}}

{"title": "话说笔记本"}

{"index":{"\_index":"test","\_type":"test","\_id":9}}

{"title": "话说笔记本电脑 Inter 16G"}

{"index":{"\_index":"test","\_type":"test","\_id":10}}

{"title": "画说水彩笔"}

搜索：虽然通过调整每个词的权重来改变顺序，但这不是好的处理方式，因为搜索“话说”有一个“华硕”的评分不该那么高。

这里，我们将拼音分词的字段权重设置到1以下这样来降低拼音匹配命中的权重，从排序上让其更加靠后。

GET http://192.168.194.130:9200/test/test/\_search?pretty

{

"explain":true,

"profile":false,

"query":{

"multi\_match": {

"type": "best\_fields",

"query": "话说",

"fields": [ "titl^3", "title.cn^1.5","title.pinyin^0.3" ]

}

}

}

待解决中。。。

## Elasticsearch数据迁移

Elasticsearch数据迁移策略：

首先我们需要介绍一个ES的接口，\_alias接口。他的作用是相当于为index起一个别名，或者是一个超链接。我们为我们的一个index起一个别名test\_index 超链接到 alias\_index，当我们查询alias\_index时，他会指向test\_index索引。

假如有下面的情景，我们插入一条数据：

PUT /my\_index/my\_type/1

{

"title" : "2017-01-01"

}

之后我们来查看新建index的mapping

GET my\_index/\_mapping/my\_type

结果

{

"my\_index": {

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "date"

}

}

}

}

}

}

我们发现ES的dynamic mapping自动将日志格式的字符串给映射为了date类型，因为我们刚才传的数据被date格式解析，此时我们再插入一条数据：

PUT /my\_index/my\_type/2

{

"title" : "hello world"

}

发现报错，因为我们提交了第一条数据之后，title字段已经被设定为date类型，我们此时在插入一个字符串当然就会报错了。

而此时你可能会尝试去修改mapping中title字段的类型，你会发现还是会报错，原因是Elasticsearch不允许你修改一个已有数据的index的mapping。

所以，办法只有一个，那就是重建索引宁设定mapping，重新导入原来数据。

此时，\_alias接口的作用就来了，它会帮助你平滑的进行这方面的过度。因为你在进行数据从旧索引到新索引迁移的过程中，你需要保持你的系统还是可用的，所以我们在系统调用Elasticsearch的过程中，要考虑到这一步，在调用索引接口的时候，最好引用链接索引，然后在数据迁移完的时候，链接索引再从旧的索引指向新的索引就完成了平滑的迁移，不会影响系统的使用

为了示例效果首先，我们创建链接索引

PUT /my\_index/\_alias/good\_index

然后，新创建一个index

PUT /my\_index\_new

{

"mappings": {

"my\_type" : {

"properties": {

"title" : {

"type": "string"

}

}

}

}

}

然后，我们使用scroll API将数据批量查询出来

GET /my\_index/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": ["\_doc"]

}

然后，将查询出来的数据通过\_bulk api批量写入

POST /\_bulk

{"index": {"\_index" : "my\_index\_new", "\_type" : "my\_type", "\_id" : "3"}}

{"title" : "2017-01-01"}

反复几次，直到数据全部导入成功，然后我们在通过\_alias API将链接指向我们的新索引：

POST /\_aliases

{

"actions" : [

{"remove" : {"index" : "my\_index", "alias" : "goods\_index"}},

{"add" : {"index" : "my\_index\_new", "alias" : "goods\_index"}}

]

}

然后，我们通过新索引来查询：

GET /goods\_index/my\_type/\_search

返回结果

{

"took": 1,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "my\_index\_new",

"\_type": "my\_type",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "2017-01-01"

}

}

]

}

}

我们发现，查询链接索引此时已经指向了我们新的索引。

## Elasticsearch bouncing results（结果跳跃）处理

**preference**

该参数能够让你控制哪些分片或者节点会用来处理搜索请求。它能够接受：

\_primary

\_primary\_first

\_local

\_only\_node:xyz

\_prefer\_node:xyz

以及\_shards:2,3

这样的值。这些值的含义在搜索preference的文档中有详细解释。

但是，最常用的值是某些任意的字符串，用来避免结果跳跃问题(Bouncing Result Problem)。

**结果跳跃(Bouncing Results)**

比如当你使用一个timestamp

字段对结果进行排序，有两份文档拥有相同的timestamp。因为搜索请求是以一种循环(Round-robin)的方式被可用的分片拷贝进行处理的，因此这两份文档的返回顺序可能因为处理的分片不一样而不同，比如主分片处理的顺序和副本分片处理的顺序就可能不一样。

这就是结果跳跃问题：每次用户刷新页面都会发现结果的顺序不一样。

这个问题可以通过总是为相同用户指定同样的分片来避免：将preference

参数设置为一个任意的字符串，比如用户的会话ID(Session ID)。让每个user每次搜索的时候，都使用同一个replica shard去执行，就不会看到bouncing results了。

## Elasticsearch type底层原理详解

type，是一个index中用来区分类似的数据的，类似的数据，但是可能有不同的fields，而且有不同的属性来控制索引建立、分词器。

field的value，在底层的lucene中建立索引的时候，全部是opaque bytes类型（二进制类型），因为Lucene是没有type概念的，所以不区分类型。在document中，实际上将type作为一个document的field来存储，即\_type，es通过\_type来进行type的过滤和筛选。

一个index中的多个type实际上是放到一起存储的，所以一个index下不能有多个重名的type。

比如下面这个代码：

{

"ecommerce": {

"mappings": {

"elactronic\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"service\_period": {

"type": "string"

}

}

},

"fresh\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"eat\_period": {

"type": "string"

}

}

}

}

}

}

两个type对应的document如下：

{

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year"

}

{

"name": "aozhou dalongxia",

"price": 199.0,

"eat\_period": "one week"

}

但是在底层的存储却是下面这样

{

"ecommerce": {

"mappings": {

"\_type": {

"type": "string",

"index": "not\_analyzed"

},

"name": {

"type": "string"

}

"price": {

"type": "double"

}

"service\_period": {

"type": "string"

}

"eat\_period": {

"type": "string"

}

}

}

}

document是这样的:

{

"\_type": "elactronic\_goods",

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year",

"eat\_period": ""

}

{

"\_type": "fresh\_goods",

"name": "aozhou dalongxia",

"price": 199.0,

"service\_period": "",

"eat\_period": "one week"

}

可以看出如下知识点：

（1）实际上type是document中的一个字段\_type，且类型是string，不进行分词

（2）底层会将当前index下所有type下的所有document的field都统一存放到了一个地方。对多个type的多个document进行了merge（合并）。

（3）doc1中没有eat\_period字段，但是也会将他放入其内，只是value为空字符串，doc2同理。进一步证明了进行了merge。

（4）证明了同一个index下的不同type下的document字段名若相同，则类型一样要相同，否则会冲突，因为他在merge的时候不知道你这个字段是什么类型。

**总结：将类似结构的type放在一个index下，这些type应该有多个field是相同的。假如说，你将两个type的field完全不同放到了一个index下，那么就每条数据都至少有一半的field在底层的Lucene中是空值，会有严重的性能问题**

## Elasticsearch match与prefix查询性能对比

match性能是很高的，为什么？

因为match搜索针对搜索条件会进行分词，然后针对每个分词有如下操作：

分词 --》扫描倒排索引 --》一旦扫描到分词就可以停了。已经找到了，没必要继续搜索其他term了，因为已经找到了。

prefix性能很低，为什么？

因为前缀搜索不分词，完全按照搜索条件搜索，针对搜索条件有如下操作：

搜索条件（前缀） --> 如果搜索到，还是要继续搜索的，因为后面可能还有买足条件的文档--> 你扫描到了一个前缀匹配的term，不能停，必须继续搜索 --> 直到扫描完整个的倒排索引，才能结束。

## Elasticsearch match\_phrase\_prefix实现搜索推荐提示

用match\_phrase\_prefix实现搜索推荐提示，一定要设置max\_expansions属性。

max\_expansions：制定prefix最多匹配多少个term，超过这个数量就不继续匹配了，限定性能。

默认情况下，前缀要扫描所有的倒排索引中的term，去查找w打头的单词，但是这样性能太差。可以用max\_expansions限定，w前缀最多匹配多少个term，就不再继续搜索倒排索引了。