

有点麻烦的同时，更多的是提供给我们更加广阔的平台 让我们尽情发挥

大米運維課堂

最前沿開源監控 Prometheus 專題講座

第八讲 Prometheus 命令行使用扩展

本节课内容

- prometheus命令行格式
- rate 函数的使用
- increase 函数的使用
- sum函数使用
- topk(x,)
- count()

(一) prometheus命令行格式

这次我们选用 一个新的key 来做讲解

`count_netstat_wait_connections !node_exporter` (TCP
`wait_connect` 数)

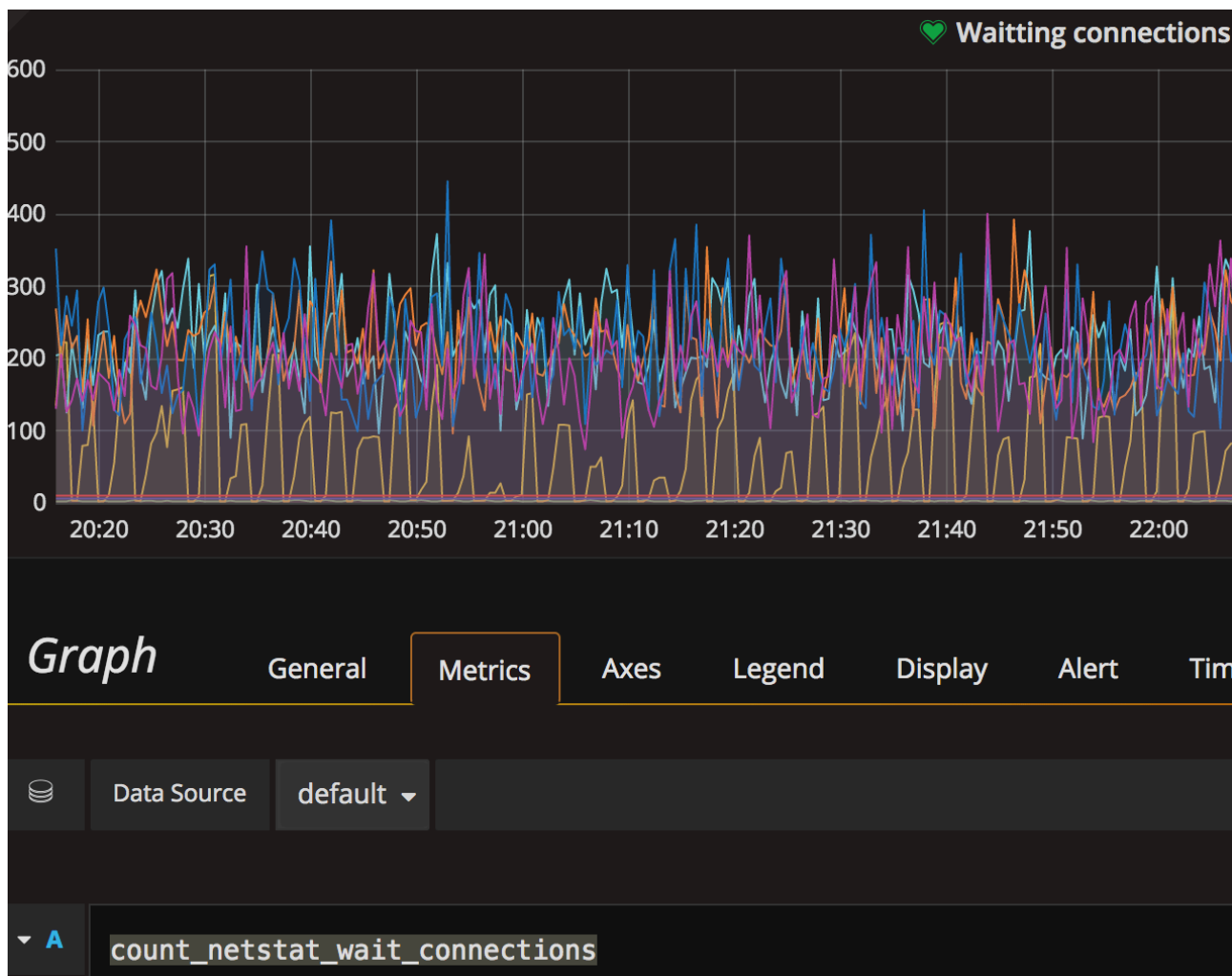
这个key 值得一说的是 并不是由我们熟悉的 `node_exporter`挖
掘而来

而是我们自行定义 并且 使用 `bash` 脚本 + `pushgateway`的方法
推送到 `prometheus server`采集

类型 CPU counter

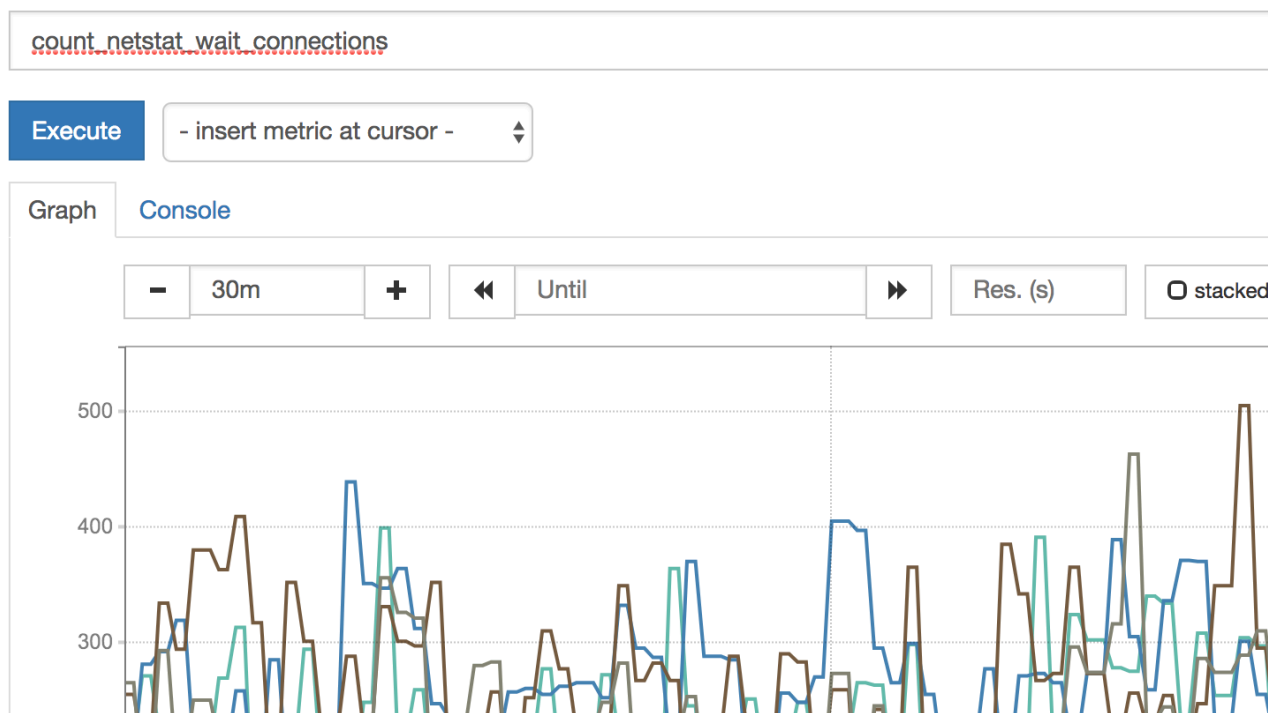
类型 gauge

`gauge`类型的数据 属于随机变化数值，并不像`counter`那样 是
持续增长



把一个key 直接输入 命令行之后 得到的是 最原始的 数据输出

count_netstat_wait_connections



counter类型的数据 使用起来 相对容易的多 CPU counter 不需要任何 increase() rate() 之类的函数去计算 单位时间段的增量 直接输入后 就可以看到 已经成型的 有确实意义的曲线图

我就使用这个key 来进行我们的命令行 正式的学习

count_netstat_wait_connections

默认输入后 会把所有 安装了这个采集项的 服务器数据都显示出来

我们来学习 命令行的 过滤

```
count_netstat_wait_connections{exported_instance="log",exported_job="pushgateway1",instance="localhost:9092",job="pushgateway"}
```

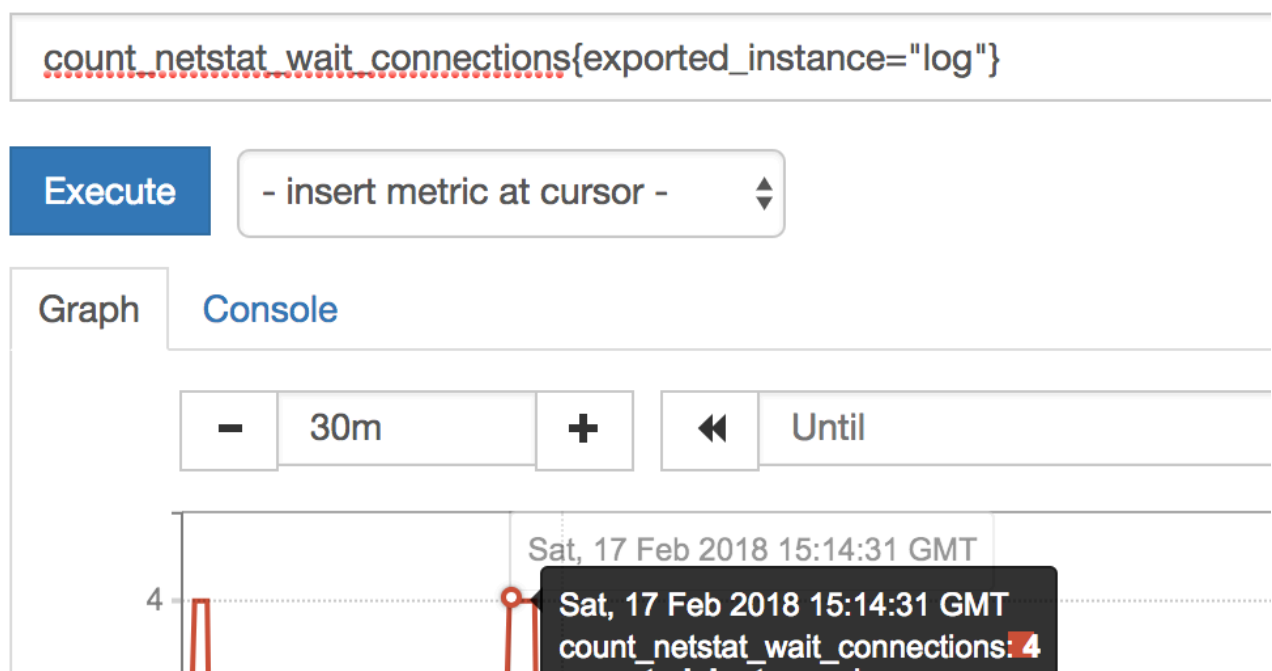
看下上面这张图的显示

后面的{.} 的部分 属于标签

标签：也是来自于采集数据，可以自定义也可以直接使用默认的exporter提供的标签项

上面的 这几个标签中 最重要的是 **exported_instance** 指明是那台被监控服务器 “log” 是一台 日志服务器的机器名

命令行的查询 在原始输入的基础上 先使用{} 进行第一步过滤
`count_netstat_wait_connections{exported_instance="log"}`



之后 就只显示 这一台服务器的 数据

过滤除了精确匹配 还有 模糊匹配

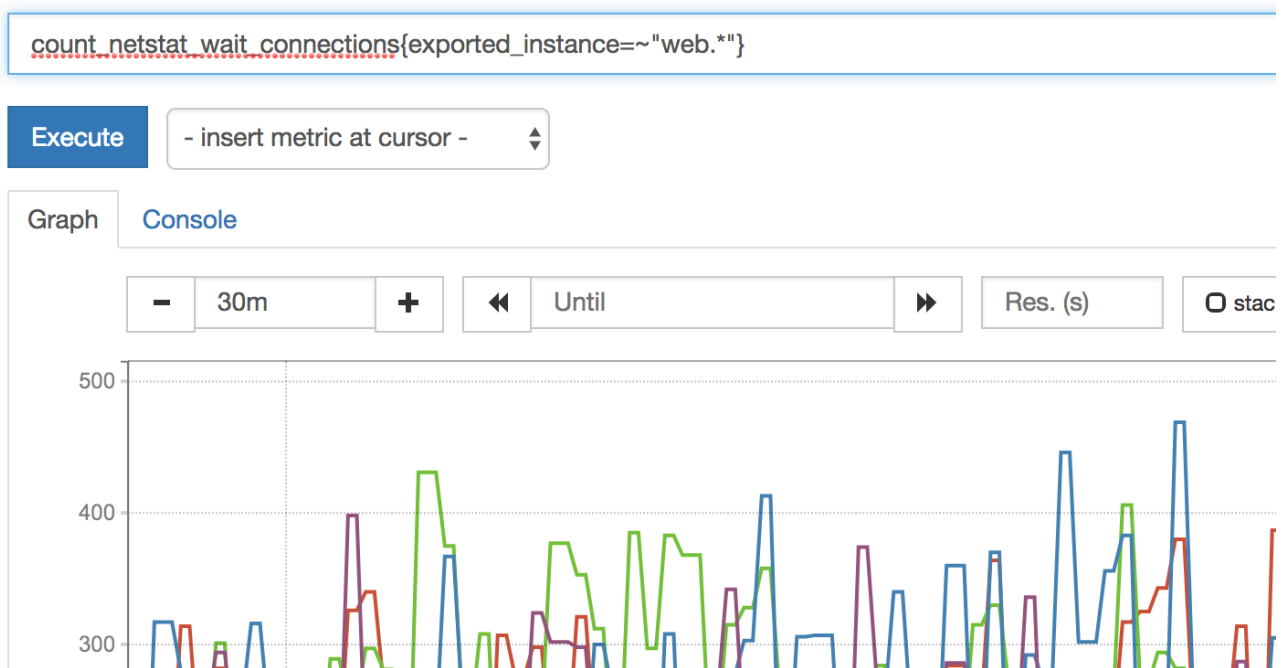
`count_netstat_wait_connections{exported_instance=~"web.*"}`
(看图演示)

把所有 机器名中 带有 web的 机器都显示出来

. * 属于正则表达式

模糊匹配 = ~

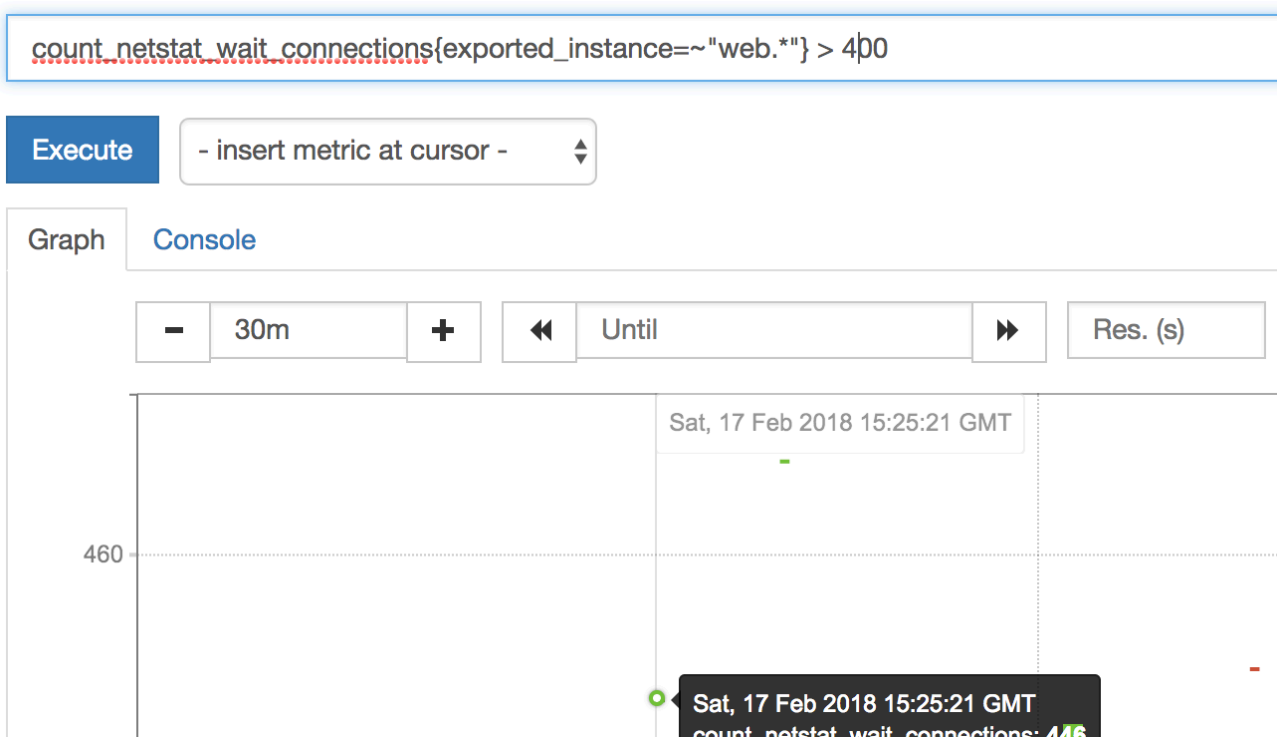
模糊不匹配 !~



标签过滤之后 就是数值的过滤

比如 我们只想找出 wait_connection数量 大于200的

`count_netstat_wait_connections{exported_instance=~"web.*"} > 400`



这时候 发现图上 看不到曲线了 只看到 很小的点

是因为 我们 > 400 的过滤 在 30 分钟之内 只有很少的时间点 达到了这个数值

--

(二) rate 函数的使用

rate 函数可以说 是 prometheus 提供的 最重要的函数之一

rate()

`rate(v range-vector)` calculates the per-second average rate of increase of the time series in the range vector. Breaks in monotonicity (such as counter resets due to target restarts) are automatically adjusted for. Also, the calculation extrapolates to the ends of the time range,

allowing for missed scrapes or imperfect alignment of scrape cycles with the range's time period.

The following example expression returns the per-second rate of HTTP requests as measured over the last 5 minutes, per time series in the range vector:

```
rate(http_requests_total{job="api-server"}[5m])
```

上面这一段 是官网给提供的解释

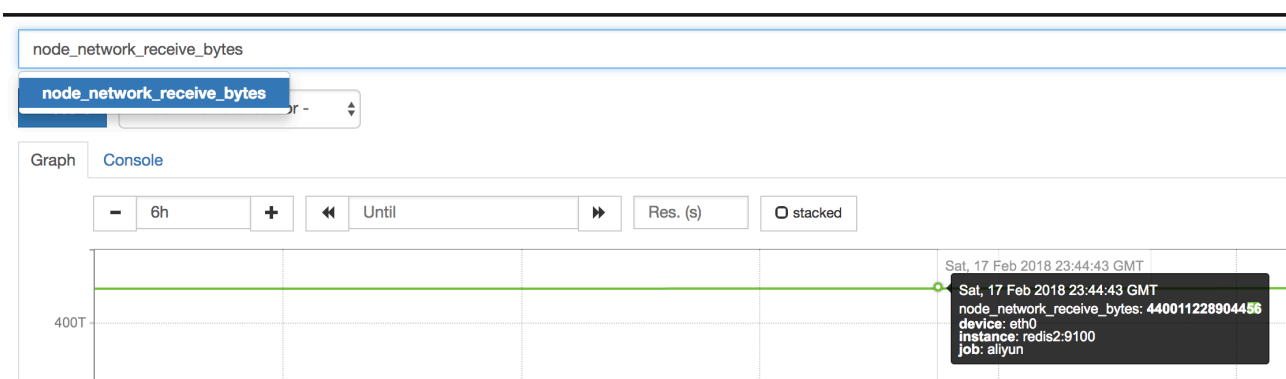
我们来简单翻译一下

gauge

rate(.) 函数 是专门搭配**counter**类型数据使用的函数

它的功能 是按照设置一个时间段，取**counter**在这个时间段中的 平均**每秒**的增量

这么说可能还是有点抽象 咱们来举个例子吧



这个例子 使用的 `node_exporter`
key **`node_network_receive_bytes`**
`rate(node_network_receive_bytes[1m])`

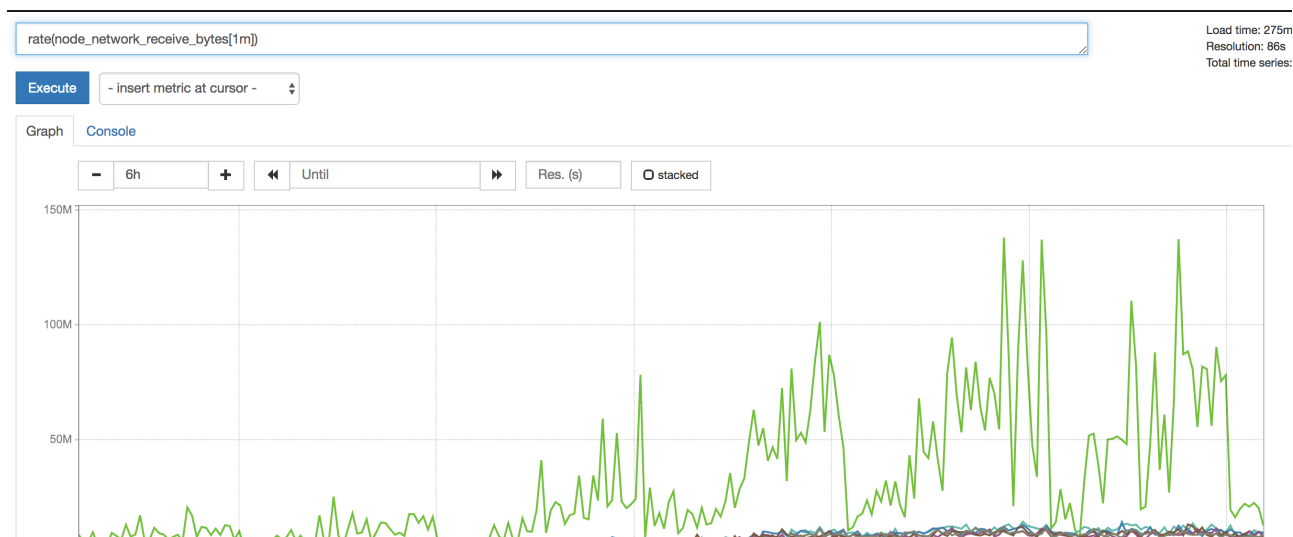
`node_network_receive_bytes` 本身是一个**counter**类型 字面意思 也很好理解
网络接收字节数

咱们之前也学习过了 对于这种 持续增长的 **counter**数据，直接输入**key** 是没有任何意义的
我们必须要以获取单位时间内 增量的方式 来进行加工 才有意义

那么 对于**counter**数据， 进行第一步的初始化的 增量获取 加工
通常的使用发放 就是直接用 **`rate()`** 包上。（**`increase()`** 也是可以的 我之后会讲到）

`node_network_receive_bytes` 被**`rate(. [1m])`**包上以后

就可以获取到 在1分钟时间内，平均每秒钟的 增量

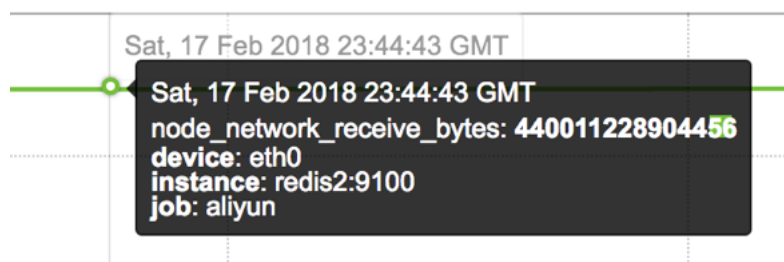


这样以来 数据就变得有意义了

强调

所以说 我们以后在使用任何counter数据类型的时候，永远记得 别的先不做 先给它加上一个 `rate()` 或者 `increase()`

接下来我们把`rate()`做的事情 更加细化的 咱们来解释一下



比如上面这个图， 网络接收字节数 一直不停的累加

从 23:44开始 到 23:45

比如累积量 从440011229804456 到了 -> 440011229805456
1分钟内 增加了 1000bytes （假设）

从 23:45开始 到 23:50

比如累积量 从440011229805456 到了 -> 440011229810456

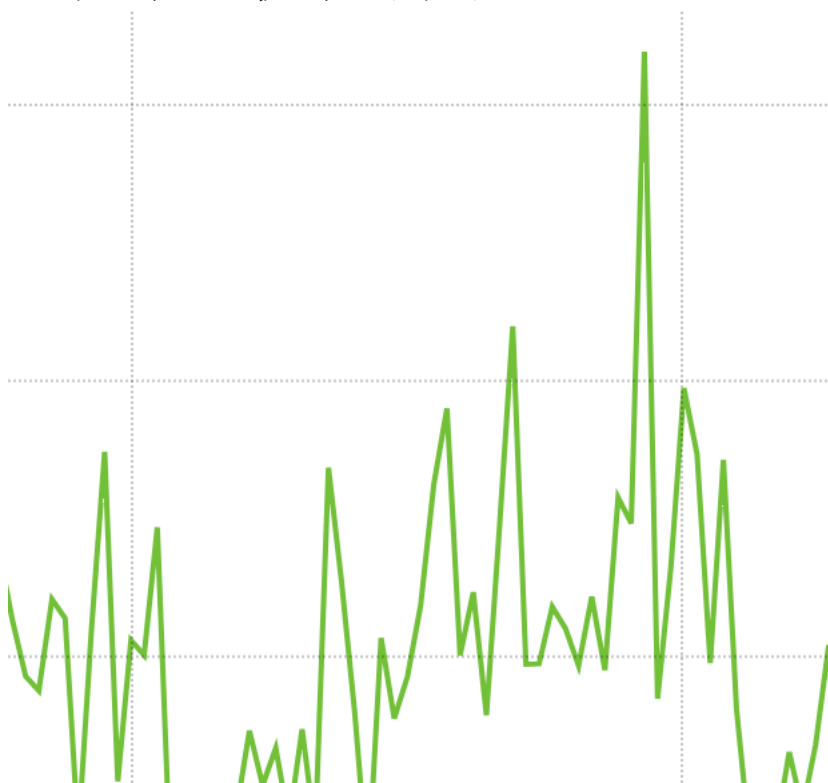
5分钟内 增加了 5000bytes (假设)

加入`rate(. [1m])` 之后

会把 1000bytes 除以 $1m \times 60$ 秒, $\approx 16\text{bytes/s}$

就是这样 计算出 在这1分钟内, 平均每秒钟增加 16bytes

这个还是比较好理解的



接下来 咱们修改 $1m \Rightarrow 5m$

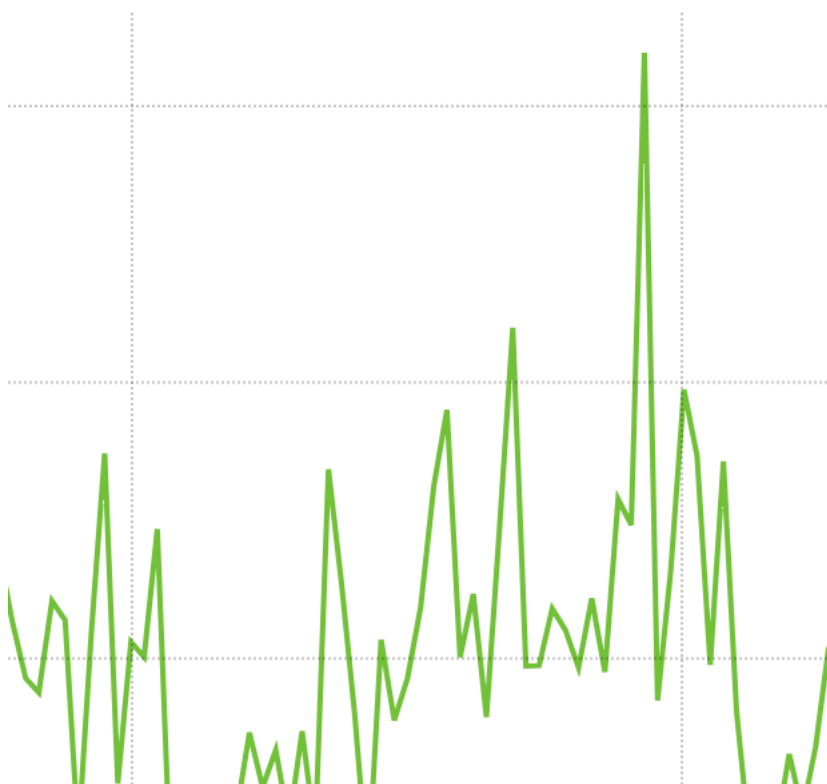
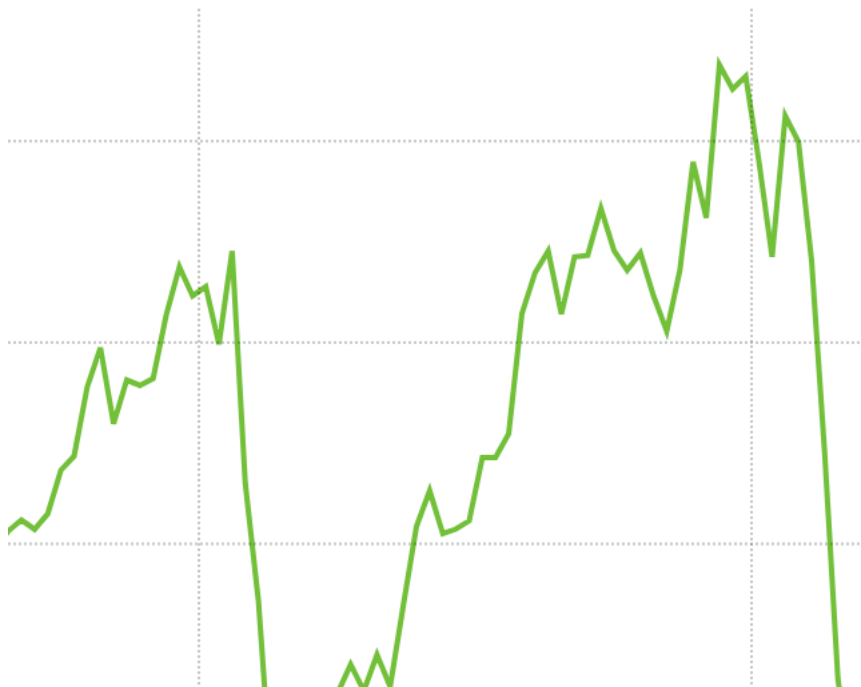
变成这样 `rate(. [5m])`

这样就变成 把5分钟内的增量 除以 $5m \times 60$

5分钟的增量 假如是 5000, 那么除以300 以后 也还是约等于 $\approx 16\text{bytes/s}$

感觉好像是一模一样的?

那么我们来看下输出 `rate[5]`



啊？感觉图形发生了一定的变化 怎么回事？



--

明明是一样的 平均 16/s 啊 怎么图形变了呢?

事实是这样的

如果 我们按照 `rate(1m)` 这样来取, 那么是取1分钟内的增量 除以秒数

如果 我们按照 `rate(5m)` 这样来取, 那么是取5分钟内的增量 除以秒数

而这种取法 是一种平均的取法 而且是假设的

刚才我们说 counter 在一分钟 5分钟 之内的增量 1000 和 5000 其实是一种假设的理想状态

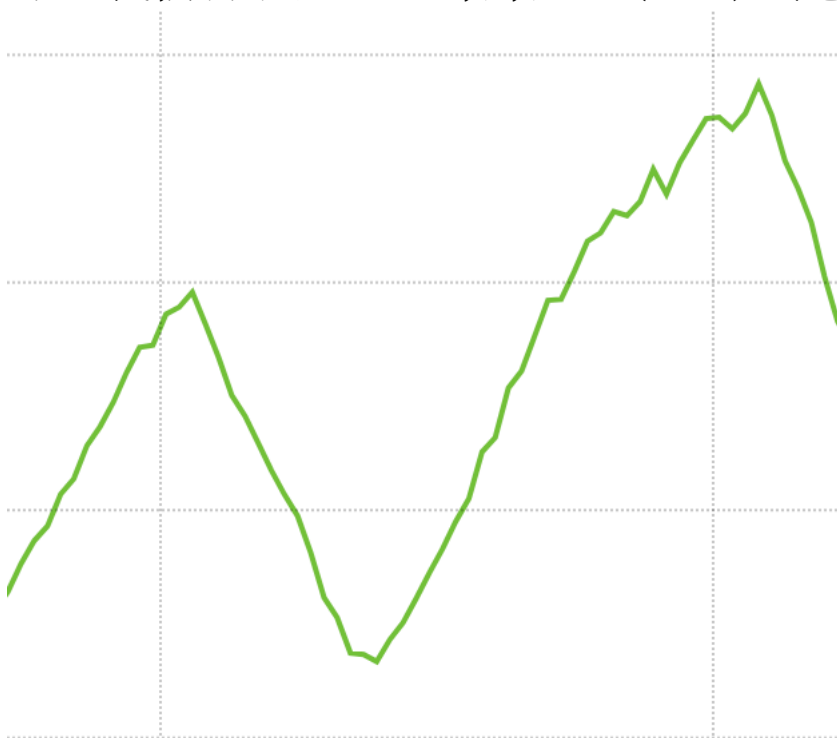
事实上 生产环境 网络数据接收量 可不是这么平均的

有可能在 第一分钟内 增加了 1000 , 到 第二分钟 就变成增加了 2500

所以 `rate(1m)` 这样的取值方法 比起 `rate(5m)` , 因为它取的时间段短, 所以 任何某一瞬间的凸起或者降低 在成图的时候 会体现的更细致 更敏感

而 `rate(5m)` 把整个5分钟内的 都一起平均了，那么当发生瞬时凸起的时候，会显得图平缓了一些（因为取的时间段长 把波峰波谷 都给平均消下去了）

那么我们再放大一些 看看 `rate(20m)` 会怎么样



哈哈 😄 是不是就更平缓了

在我们的工作中 取1m 还是取5m

这个取决于 我们对于监控数据的敏感性程度来挑选

说到这里 我们对`rate()` 函数 应该也有了一定的了解了

接下来 咱们继续学习 第二个重要的函数

(三) `increase`函数 使用

`increase` 函数 其实和`rate()` 的概念及使用方法 非常相似

rate(1m) 是取一段时间增量的平均每秒数量
increase(1m) 则是 取一段时间增量的总量

比如

increase(**node_network_receive_bytes**[1m])
取的是 1分钟内的 增量总量

和

rate(node_network_receive_bytes[1m])

取的是 1分钟内的增量 除以 60秒 每秒数量

```
increase(node_network_receive_bytes[1m])
```

Execute

- insert metric at cursor -

Graph

Console

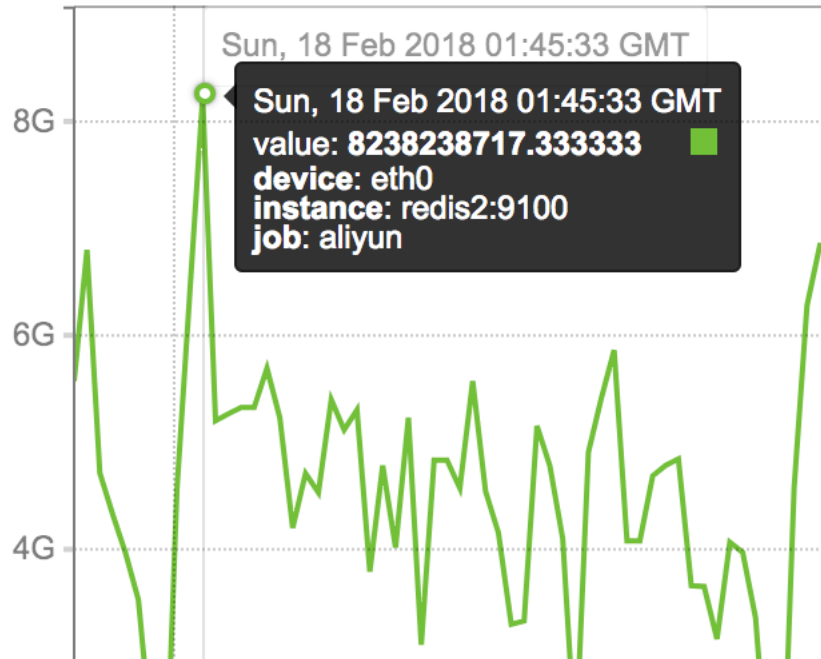
-

1h

+

◀

Until



```
rate(node_network_receive_bytes[1m])
```

Execute

- insert metric at cursor -

Graph

Console

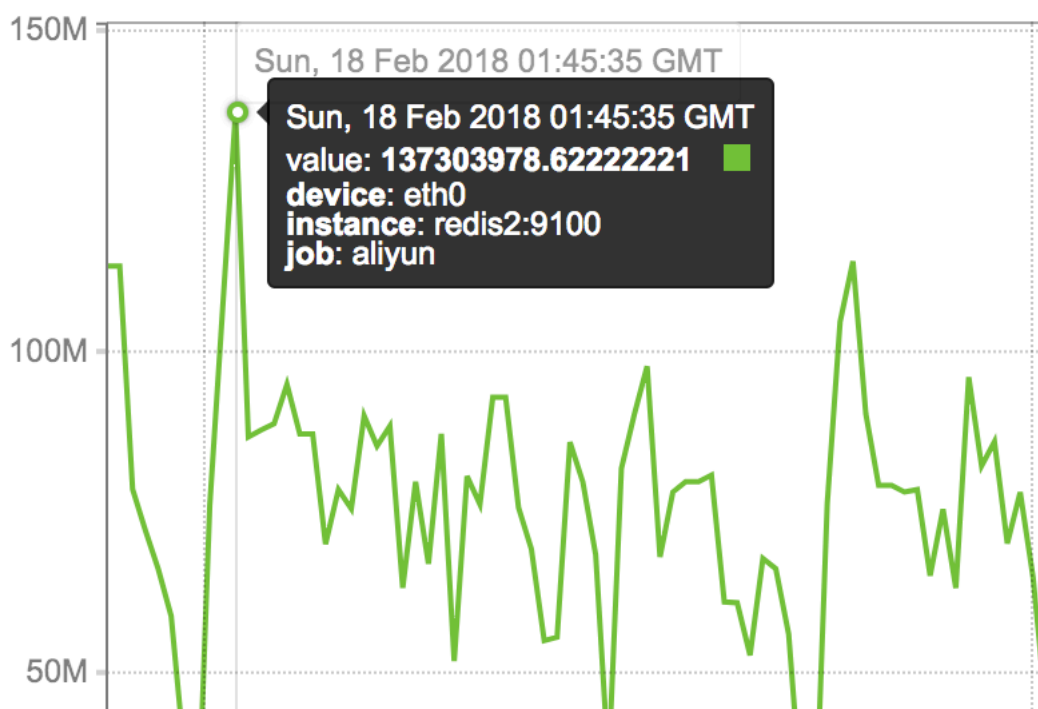
-

1h

+

◀

Until



从这两个图 我们可以看到 其实曲线的走势 基本是一样的
但是 显示出来的数量级bytes 可不一样

$137303978 * 60 = 8238238680$ (发现)

正好是 60倍

也就很好理解了, `increase()` 是不会取一秒的平均值的

`inraese()` `rate()`

监控 获取采集数据源

频率

5m => rate()

形成图 断链

increase(5m)

rate() -> CPU 内存 硬盘 IO 网络流量

counter

(四) **sum()**函数的学习

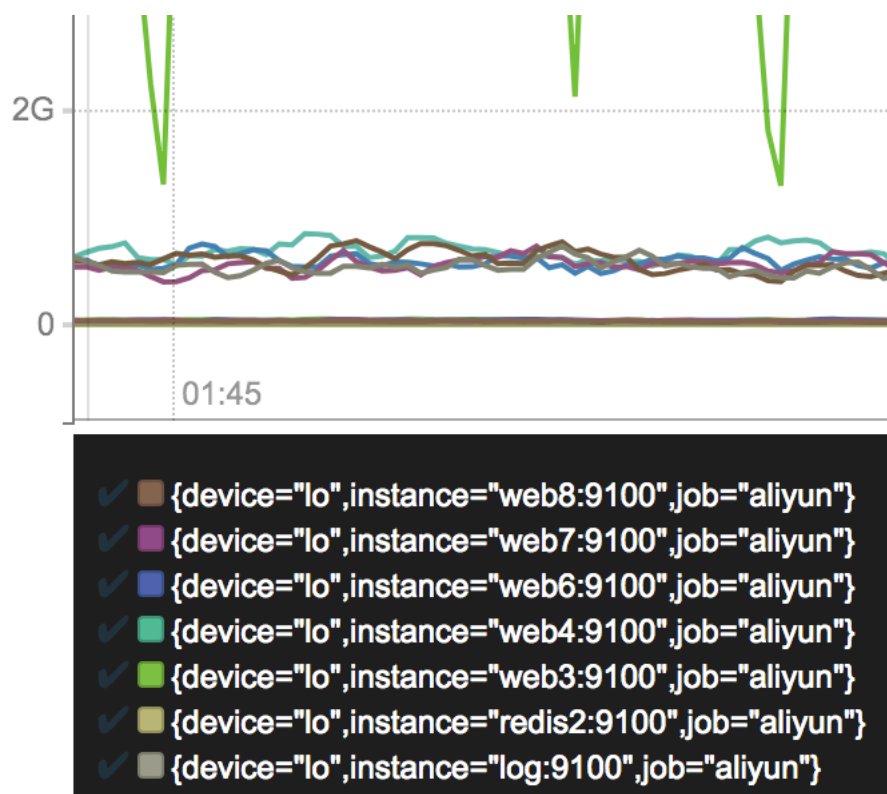
sum()函数的使用 , 我们在上节课 讲CPU拆分公式的时候 已经讲过了

sum就是 总取合

sum 会把结果集的输出 进行总加合

比如

rate(node_network_receive_bytes[1m]) 显示的结果集 会包含如下内容



从标签 我们就不难看出，有好多台服务器 都返回了 这个监控数据

当我们使用 `sum()`包起来以后

`sum(rate(node_network_receive_bytes[1m]))` 就变成下面这样了

```
sum(rate(node_network_receive_bytes[1m]))
```

Execute

- insert metric at cursor -

Graph

Console

-

1h

+

◀

Until

150M

100M

变成一条线了 ...

等于是 给出了 所有机器的 每秒请求量..

我们之前也说过，如果要进行下一层的拆分
需要在 `sum()` 的后面 加上 `by (instance)`

才可以按照 机器名 拆分出一层来

sum() 加合 其实还有更多巧妙使用

instance

sum () by (cluster_name)

如果是 by instance 那么 其实跟 不加sum()的输出结果是一样的

本来 rate(node_network_receive_bytes[1m]) 就已经是按照每台机器 返回了

但是如果 我们希望 按集群总量输出呢?

比如 我们返回了20台机器的数据

其中 有6台 属于 web server

10台属于 DB server

其他的 属于一般server

那么我们这时候 sum() by (cluster_name) 就可以帮我们实现 集群加合并分三条曲线输出了

100 - 1000 CPU

顺带一提的是 (**cluster_name**) 这个标签，默认node_exporter 是没有办法提供的

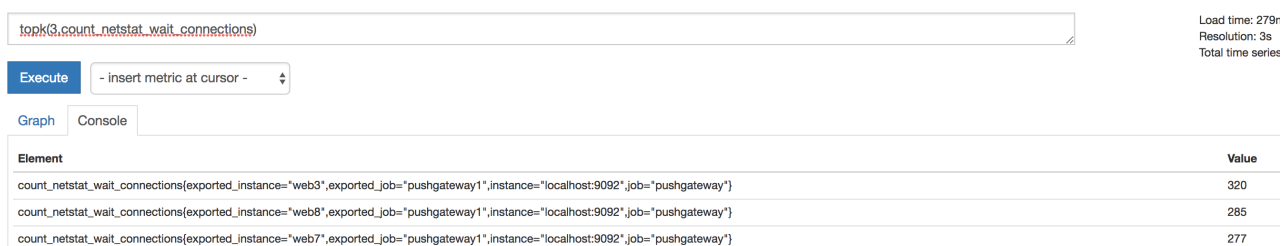
node_exporter只能按照 不同的机器名 去划分

如果希望 支持cluster_name 我们需要自行定义标签 😊

往后的课程 我们会涉及到这个内容

(五) topk()函数的学习

定义：取前几位的最高值



Gauge类型的使用

`topk(3,count_netstat_wait_connections)`

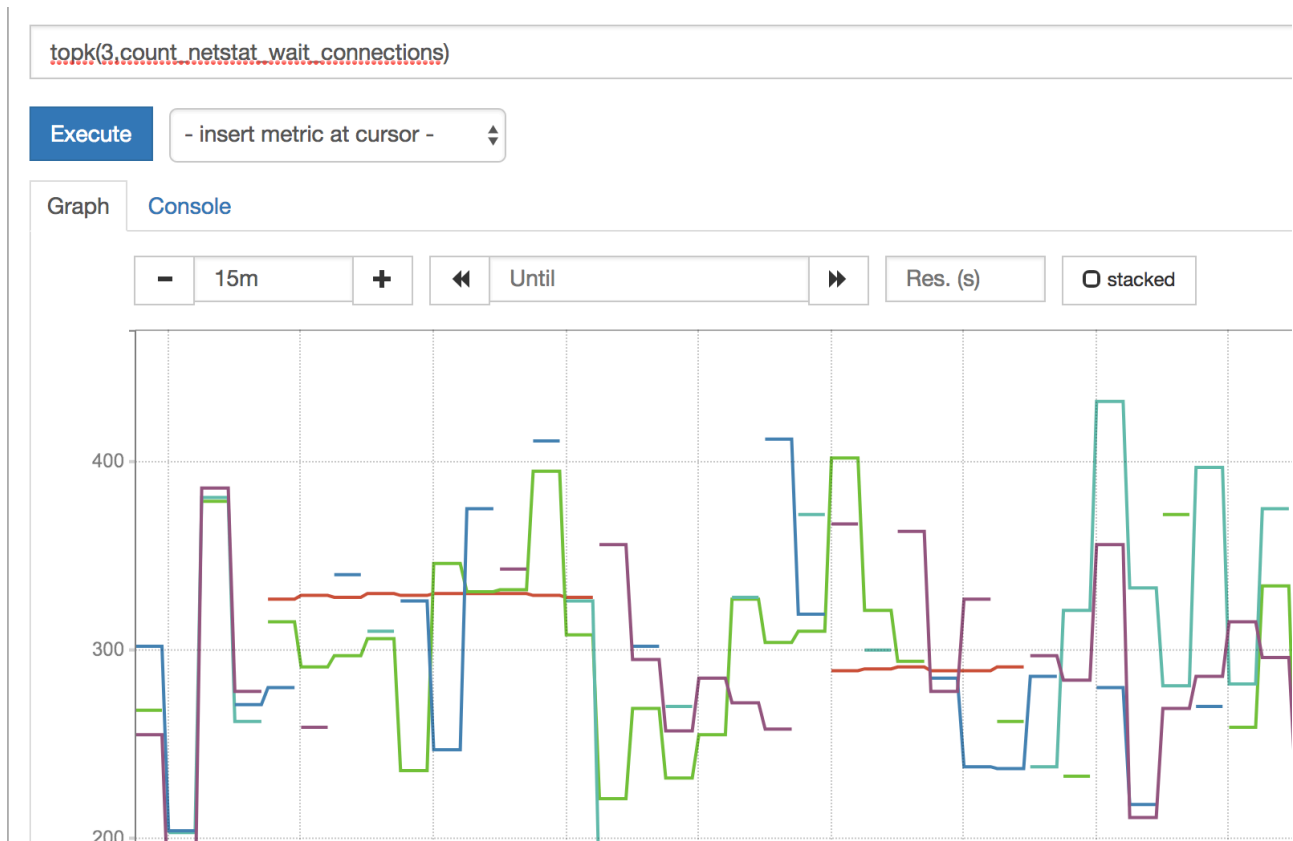
Counter类型的使用

`topk(3,rate(node_network_receive_bytes[20m]))`

这个函数 还是比较容易理解的，根据给定的数字 取数值最高 $\geq x$ 的数值

唯一需要注意的是

这个函数 一般在使用的時候 只适合于 在console查看 graph的意义不大
如下图所示



Topk因为对于每一个时间点 都只取前三高的数值
那么必然会造成 单个机器的采集数据不连贯

因为：比如 server01在在这一分钟的 wait_connection数量排在所有机器的前三，到了下一分钟 可能就排到垫底了.. 自然其曲线就会中断

实际使用的时候 一般用topk（）函数 进行瞬时报警 而不是为了观察曲线图

CPU 10 40核
400项

`topk(10, rate(CPU))`

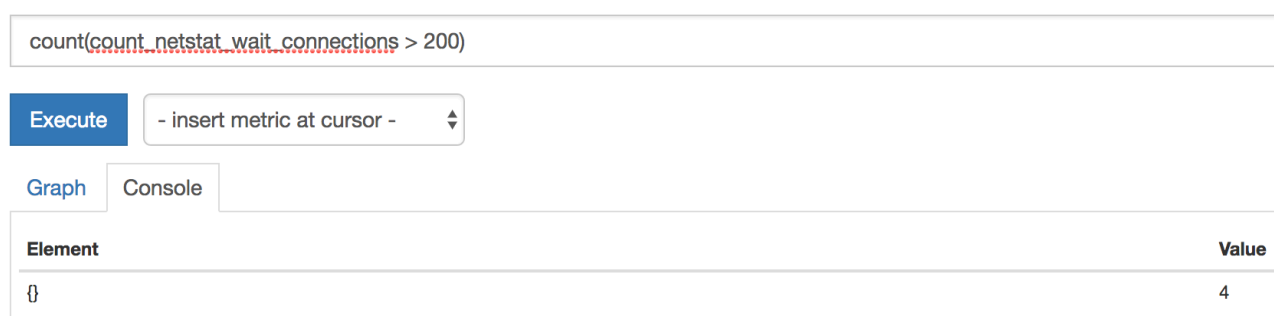
CPU 更多不看核（很自然的分布在多核 服务器 软件）

(六) count()函数的学习

定义：把数值符合条件的 输出数目进行加合

举个例子

找出当前（或者历史的）当TCP等待数大于200的 机器数量



The screenshot shows a monitoring interface. At the top, a text input field contains the query `count(count_netstat_wait_connections > 200)`. Below the input field is a blue button labeled "Execute" and a dropdown menu with the text "- insert metric at cursor -". Below these are two tabs: "Graph" and "Console". The "Console" tab is active, showing a table with two columns: "Element" and "Value". The table has one row with the value "{}" in the "Element" column and "4" in the "Value" column.

Element	Value
{}	4

`count(count_netstat_wait_connections > 200)`

这个函数在实际工作中还是很有用的

一般用它count进行一些模糊的监控判断

比如说 企业中有**100**台服务器，那么当只有10台服务器CPU高于80%的时候 这个时候不需要报警
但是 当符合80%CPU的服务器数量 超过 30台的时候 那么就会触发报警 count()

到这里 我们第八讲结束

顺便提一句 prometheus提供的函数 其实还远远不止这几个

这节课中 列出的6个函数 是作为 工作中使用频率最高的函数
(或者说 如果这6个函数 不会的话 那么监控基本没发做)

其他更多的函数 感兴趣的朋友 可以去prometheus官方网站继续学习

<https://prometheus.io/docs/prometheus/latest/querying/functions/>

也欢迎多给大米提建议和补充新函数知识点

谢谢🙏



大米運維課堂