

# PromQL 从入门到精通

对于 Prometheus 生态的监控系统，PromQL 是必备技能，本文着重点讲解这个查询语言，掺杂一些生产实践场景，希望你有所帮助。

📌 本文作者：秦晓辉，Open-Falcon、Nightingale 等开源项目创始人之一，极客时间《运维监控系统实战笔记》作者；当前在创业（快猫星云联创），为客户提供监控/可观测性产品方案，有需求的朋友欢迎联系我的微信 picobyte。

## 数据类型

Prometheus 有四种数据类型：Gauge、Counter、Histogram、Summary，其中最关键的是 Gauge 和 Counter，Histogram 和 Summary 只是为了上报监控数据的 Client 侧的便利，可以看做是组合使用了 Gauge 和 Counter。所以我们重点就来讲解 Gauge 和 Counter 类型。

## Gauge 类型

Gauge 类型的值表示当前的状态，可大可小、可负可正，比如某个虚拟机实例挂了，用 0 表示，如果实例存活，用 1 表示；再比如内存使用率，这个时刻采集是 33.7%，下个周期采集可能就变成了 25.8%；还有像机器最近 5 分钟的 load、正在运行的进程数量等等，都使用 Gauge 类型来表示。这种类型的值，我们非常关注当前值。

## Counter 类型

Counter 类型是单调递增的值，比如机器上某块网卡收到的数据包的总量，是从操作系统启动之后，就持续递增的，对于这种类型的值，我们通常关注的不是当前值是多少，而是关注增量和变化率。我们在机器上执行 ifconfig 命令：

```
1 eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
2     inet 10.206.0.16 netmask 255.255.240.0 broadcast 10.206.15.255
3     inet6 fe80::5054:ff:fed2:a180 prefixlen 64 scopeid 0x20<link>
4     ether 52:54:00:d2:a1:80 txqueuelen 1000 (Ethernet)
5     RX packets 457952401 bytes 125894899868 (117.2 GiB)
6     RX errors 0 dropped 0 overruns 0 frame 0
7     TX packets 518040495 bytes 276312546157 (257.3 GiB)
8     TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

RX packets 后面的值是 OS 启动以来收到的总的包量，TX packets 后面的值是 OS 启动以来发出去的总的包量，都是很大的值，我们通常不太关注这个值当前是多少，更关注的是最近 1 分钟收到/发出多少包，或者每秒收到/发出多少包。

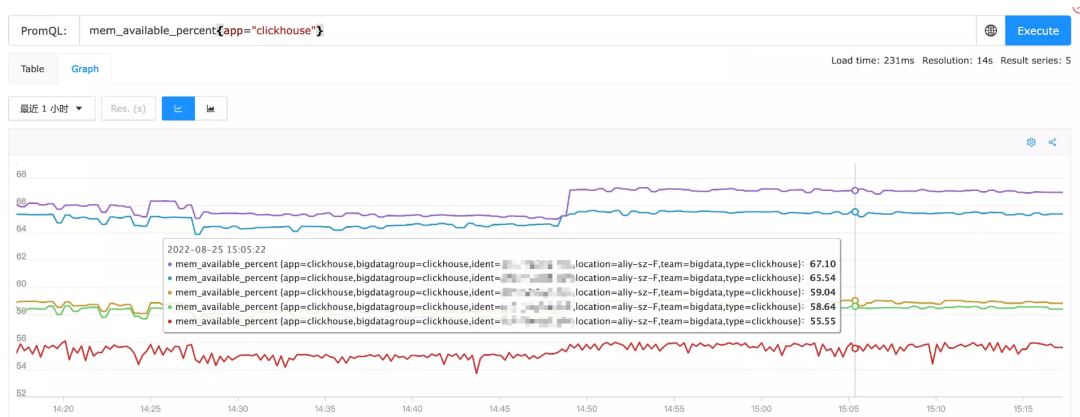
而对于监控数据采集器而言，一般是周期性运行的，比如每 10 秒采集一次，每次采集网卡收到/发出的包这个数据的时候，都只能采集到当前的值，就像执行 ifconfig 命令，每 10 秒执行一次，每次都看到一个巨大的当前值，而且一次比一次大。如果采集器不做计算，把这个值原封不动上报给监控服务端，那计算增量、计算速率这个需求，就要放到服务端来实现了，所以服务端必须要能对这种类型的数据建模抽象，也就是所谓的 Counter 类型。

## 时序数据

PromQL 就是查询时序数据的一种 Query Language，要想对 PromQL 有了解，得先搞清楚时序数据。

## 认识时序数据

我们先来看一张图，图上是 5 台机器的内存可用率：



每个机器的内存可用率数据，体现为图上的一条线，我们称为 **series**，某个机器在某一时刻的内存可用率数据，我们称为**数据点**，比如上图，2022-08-25 15:05:22 这个时刻，每个机器都有一个可用率数据点，共计 5 个数据点。

上面的图是查询的最近一小时的，我们切换到 Table 视图，得到如下结果：

mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	Value
mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	65.0578604593
mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	58.3830158265
mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	66.5530628961
mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	58.1171058705
mem_available_percent(app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse")	55.0013633864

这个表格的内容，是这 5 台机器在当前这个时间点的最新值，当前我做查询的时刻是：2022-08-25 15:48:03 用 Chrome 开发者工具可以看到发的请求参数：

Name	Value
query?query=mem_available_percent%7Bapp%3D%22clickhouse%22%7D&time=1661413683	query: mem_available_percent{app="clickhouse"} time: 1661413683

但是，监控数据是周期性上报的，比如每 10 秒上报一次，在 2022-08-25 15:48:03 这个时刻，未必恰好有监控数据啊，那这个 Table 中的数据是哪里来的？

实际上，Prometheus 有个启动参数，`--query.lookback-delta=2m` 来控制这个行为，如果配置为 2m，就表示，Prometheus 会查询 2022-08-25 15:46:03 ~ 2022-08-25 15:48:03 这 2 分钟之间的数据，然后返回最新的那个。

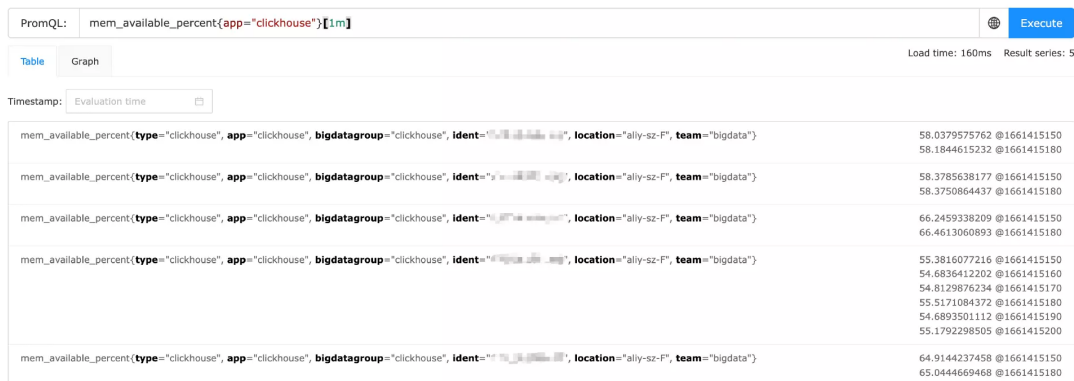
## 查询类型

上例中的 `mem_available_percent{app="clickhouse"}` 称为查询表达式，不同的表达式，会返回不同的内容，返回的内容总共有 4 种格式，分别是：Instant vector（瞬时向量）、Range vector（范围向量）、Scalar（标量）、String（字符串）。返回瞬时向量的查询表达式，我们称为 Instant Query，返回范围向量的查询表达式，我们称为 Range Query。

上例中的 `mem_available_percent{app="clickhouse"}` 既可以用于展示 Table 视图，又可以用于展示 Graph 视图，是典型的 Instant Query。

如果在表达式后面加上一个时间范围，比如 1 分钟：

`mem_available_percent{app="clickhouse"}[1m]` 这个查询表达式就变成了 Range Query，Range Query 里有个时间范围，其 Table 视图的截图如下：



Labels	Value
<code>mem_available_percent{type="clickhouse", app="clickhouse", bigdatagroup="clickhouse", ident="...", location="aly-sz-F", team="bigdata"}</code>	58.0379575762 @1661415150 58.1844615232 @1661415180
<code>mem_available_percent{type="clickhouse", app="clickhouse", bigdatagroup="clickhouse", ident="...", location="aly-sz-F", team="bigdata"}</code>	58.3785638177 @1661415150 58.3750864437 @1661415180
<code>mem_available_percent{type="clickhouse", app="clickhouse", bigdatagroup="clickhouse", ident="...", location="aly-sz-F", team="bigdata"}</code>	66.2459338209 @1661415150 66.4613060893 @1661415180
<code>mem_available_percent{type="clickhouse", app="clickhouse", bigdatagroup="clickhouse", ident="...", location="aly-sz-F", team="bigdata"}</code>	55.3816077216 @1661415150 54.6836412202 @1661415160 54.8129876234 @1661415170 55.5171084372 @1661415180 54.6893501112 @1661415190 55.1792298505 @1661415200
<code>mem_available_percent{type="clickhouse", app="clickhouse", bigdatagroup="clickhouse", ident="...", location="aly-sz-F", team="bigdata"}</code>	64.9144237458 @1661415150 65.0444669468 @1661415180

第 4 台机器相比其他的机器，返回了更多数据，是因为那个机器的监控数据采集频率是 10s，而其他的机器采集频率是 30s。

📌 通过 range query + Table 视图，可以让我们直观看到原始上报的监控数据以及上报的具体时刻（对于排查监控数据采集相关的问题尤为有用），如果在 Graph 视图，返回的数据取决于 step 参数，查询时传给时序库的 `step = 10`，返回的图形就是每 10s 一个点，`step = 20` 就是每 20s 一个点，返回的数据的时间间隔取决于 step 参数而非原始数据的上报间隔。

Range Query 理论上是没法绘制 Graph 的（当然有些时序库可能会做容错处理），因为从原理上说不通。绘图的时候，我们要选择一个时间范围，比如最近一小时，然后传给后端一个 step 参数用于控制分辨率，即数据间隔，比如 `step=60`，即表示希望每个 series 每分钟返回一个点，但如果是 Range Query，相当于在某个时刻返回多个点，这就无所适从了。

Prometheus 文档中有一个章节专门介绍函数，各个函数的介绍中，都会写明是用于 instant-vector，还是用于 range-vector，如果不理解查询类型，就无法很好的应用这些函数。

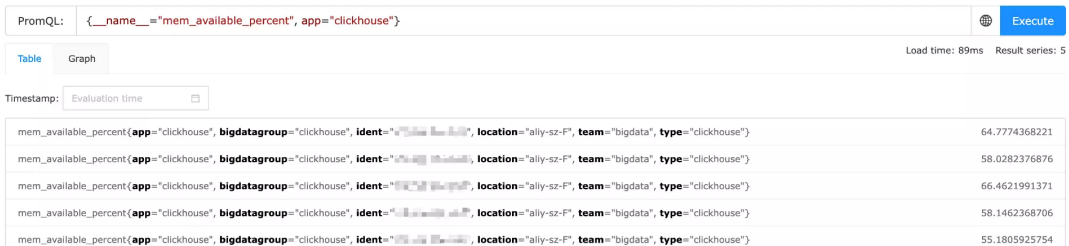
## 查询选择器

PromQL大括号里的部分是 selector，查询选择器，用于从一大堆监控数据中，过滤出真正关心的数据，在 Prometheus 生态里，时序数据的标识，就是一堆标签集合，所以这里的过滤，就是针对标签做过滤，支持四类操作符：

- = : 完全匹配，比如 `app="clickhouse"`
- != : 完全不匹配，比如 `app!="clickhouse"`
- =~ : 正则匹配，比如 `app=~"n9e-.*"`
- !~ : 正则不匹配，比如 `app!~"n9e-.*"`

指标名称，通常放到大括号之外，但实际上，指标名称也是一个标签，其标签Key是 `__name__`，所以之前查询的例子，可以这么写：

```
1 {__name__="mem_available_percent", app="clickhouse"}
```



The screenshot shows a Prometheus query interface. The PromQL query is `{__name__="mem_available_percent", app="clickhouse"}`. The results are displayed in a table with columns for the metric name, labels, and the value. The table contains five rows of data for the `mem_available_percent` metric.

mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	Value
mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	64.7774368221
mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	58.0282376876
mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	66.4621991371
mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	58.1462368706
mem_available_percent	app="clickhouse", bigdatagroup="clickhouse", ident="...", location="ally-sz-F", team="bigdata", type="clickhouse"	55.1805925754

仍然可以达成相同的效果。有时采集的监控数据格式设计的不好，一些本该用 label 的信息，放到了 metric 名称中了，此时就可以用 `__name__` 做一些正则匹配。

## Offset

监控系统里，经常会有同环比的需求，比如，当前的值相比一周之前，是否有巨大变化，那怎么才能获取历史数据呢？可以使用 `offset` 关键字。

`offset` 后面跟一个时间段，比如 `5m`、`1d`、`7d`、`1w`，`offset` 要紧跟查询选择器，比如：

```
1 sum(http_requests_total{method="GET"} offset 1d)
```

## 运算符

PromQL 支持基本的算术运算符和比较运算符，可以对不同的即时向量做运算，这为监控系统带来了巨大的进步，算术运算符让很多计算不需要在采集端做了，可以轻易挪到服务端，而比较运算符则为告警逻辑提供了支撑。

### 算术运算符

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulo)
- `^` (power/exponentiation)

举一个例子来演示真实环境下的算术运算符的应用，比如之前的例子，对于内存可用率的指标 `mem_available_percent` 这个指标是采集器直接计算好的，如果采集器没有计算，而是上报了原始指标 `mem_available` 和 `mem_total`，我们仍然可以使用 `promql` 计算出可用率指标：

The screenshot shows the Prometheus PromQL query editor. The query is `mem_available{app="clickhouse"}/mem_total{app="clickhouse"}*100`. Below the query, there are tabs for 'Table' and 'Graph'. The 'Table' view is active, showing a table with 5 rows of data. Each row contains a JSON-like selector string and a numerical value.

Selector	Value
<code>{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.160", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	64.79533526493334
<code>{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.161", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	57.76330016149392
<code>{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.162", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	65.67076029506653
<code>{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.163", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	57.85702608365623
<code>{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.164", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	54.68382203483751

逻辑上，是先根据 `mem_available{app="clickhouse"}` 找到相关指标数据，会找到5条，再根据 `mem_total{app="clickhouse"}` 也能找到5条，二者相除的逻辑姑且可以理解为，循环遍历 `mem_available` 的5条记录，对于每一条，去 `mem_total` 的5条记录中找标签相同的记录，进行除法运算。除法运算得到5条结果（0~1之间的数字），然后跟100相乘（得到百分比大小），100这个数字称为标量，5条结果和标量计算，会把每一条结果分别乘以100，得到最终的结果，这个最终结果其实就是 `mem_available_percent`。

如果分子和分母对应的selector查到的数据标签不同，就没法做除法运算了，比如 `net_bytes_recv` 比内存相关的指标多了一个 `interface` 的标签（标明网卡），二者是没法做运算的，结果为空：

```
1 net_bytes_recv{app="clickhouse"}/mem_total{app="clickhouse"}
```

## 比较运算符

- `==` (equal)
- `!=` (not-equal)
- `>` (greater-than)
- `<` (less-than)
- `>=` (greater-or-equal)
- `<=` (less-or-equal)

比较运算符的最典型用法，就是一个 `instant-vector` 和一个标量之间的比较，比如 `mem_available_percent{app="clickhouse"} < 60` 的结果：

The screenshot shows the Prometheus PromQL query editor. The query is `mem_available_percent{app="clickhouse"} < 60`. Below the query, there are tabs for 'Table' and 'Graph'. The 'Table' view is active, showing a table with 5 rows of data. Each row contains a JSON-like selector string and a numerical value.

Selector	Value
<code>mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.160", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	64.9310300255
<code>mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.161", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	57.7962503192
<code>mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.162", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	65.8632220275
<code>mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.163", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	57.6751640573
<code>mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.164", location="ally-sz-F", team="bigdata", type="clickhouse"}</code>	54.334051579

如果我们认为内存可用率小于60就是有问题的，想找出所有有问题的数据，只要在 `promql` 中拼上 `< 60` 即可：

PromQL: mem_available_percent{app="clickhouse"} < 60		Execute
Table	Graph	Load time: 66ms Result series: 3
Timestamp: Evaluation time		
mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.161", location="ally-sz-F", team="bigdata", type="clickhouse"}		57.7493487697
mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.163", location="ally-sz-F", team="bigdata", type="clickhouse"}		57.9273982535
mem_available_percent{app="clickhouse", bigdatagroup="clickhouse", ident="172.18.204.164", location="ally-sz-F", team="bigdata", type="clickhouse"}		54.1945200203

如上的方法，其实就是告警引擎的核心逻辑。告警规则里会要求用户配置promql以及执行频率，告警引擎就会根据执行频率周期性执行，每次执行的时候就是拿着promql去查询，promql中带有阈值，即上例中的 <60，所以如果所有机器的内存可用率都很高，比如维持在80~90，那这个promql是不会返回查询结果的，此时监控系统就认为一切正常。如果返回了结果，比如上例中返回了3条结果，告警引擎就会认为有异常产生，生成3个告警事件。

当然，有的时候，偶尔一次触发了阈值我们认为不算啥事，希望连续触发多次才告警，此时就要使用 prometheus alerting rule 的 for 关键字，或者夜莺中的持续时长的配置，表示在一个时间范围内多次执行，每次都触发了才告警。

像上例触发了3个告警事件，如果后面继续周期性使用promql查询查不到数据了，就说明最新的mem\_available\_percent数据不再小于60，即告警恢复。

## 逻辑/集合运算符

相关运算符有三个：and、or、unless 用于 instant-vector 之间的运算。首先来解释一下各个运算符的行为。

### and

vector1 and vector2，其结果是一个由vector1的元素组成的向量，对于这些元素，vector2中存在着完全匹配的标签集，其他元素被删除。metric的名称和值从左边的向量转移过来。

用于什么场景？先经过 vector1 做过滤得到一批监控数据，可能里边有一些是不想要的，可以用 and 操作符，再加一个条件，用另一个 metric 的值做一些二次过滤。举例：

```
1 disk_used_percent{app="clickhouse"} > 70
2 and
3 disk_total{app="clickhouse"}/1024/1024/1024 < 500
```

磁盘利用率大于70%就告警，对于盘不大的情况是适用的，如果盘太大，比如16T一块盘，使用率70%还有非常大的余量，所以这里我们使用and附加一个条件，限制一下disk\_total，即磁盘总大小，磁盘总大小小于500GB，才适用磁盘利用率大于70%这个规则。

### or

vector1 or vector2，其结果是一个向量，包含vector1的所有原始元素（标签集+值）以及vector2中所有在vector1中没有匹配标签集的元素。

举一个例子，比如系统负载，有最近1分钟、最近5分钟、最近15分钟的负载，需求是：最近1分钟的负载大于8或者最近5分钟的负载大于8，就告警，promql写法：

```
1 system_load1{app="clickhouse"} > 8
2 or
```

```
3 system_load5{app="clickhouse"} > 8
```

## unless

vector1 unless vector2，结果是一个由vector1中的元素组成的向量，在vector2中没有完全匹配标签集的元素，两个vector中的所有匹配元素都被删除。姑且可以理解为一个减法，vector1 - vector2。

举个例子，还是磁盘利用率的问题，对于超过1个T的大盘，剩余量小于300G就告警，promql怎么写？

```
1 disk_free{app="clickhouse"}/1024/1024/1024 < 300
2 unless
3 disk_total{app="clickhouse"}/1024/1024/1024 < 1024
```

使用 unless 排除掉小于1个T的盘，剩下的就只剩大于1个T的大盘了，效果达成。

## 向量匹配

向量之间的操作试图在右侧的向量中为左侧向量的每个条目找到一个匹配的元素，匹配行为分为：one-to-one、many-to-one、one-to-many。

## on 和 ignoring

上面演示 and、or、unless 的例子，两个vector的标签集都是一样的，那如果有些标签略微有些差异，可以使用 on 和 ignoring 关键字来限制用于做匹配的标签集。举例：

```
1 mysql_slave_status_slave_sql_running == 0
2 and ON (instance)
3 mysql_slave_status_master_server_id > 0
```

这个promql想表达的意思是如果这个mysql实例是个slave ( master\_server\_id>0 )，则检查其 slave\_sql\_running的值，如果slave\_sql\_running==0表示slave sql线程没有在运行。

但是mysql\_slave\_status\_slave\_sql\_running和mysql\_slave\_status\_master\_server\_id这两个metric的标签可能并非完全一致，不过好在二者都有个instance标签，且相同instance标签的数据从语义上来看就表示一个实例的多个指标数据，那就可以用on关键字，指定只使用instance标签做匹配，忽略其他标签。

与on相反的是ignoring关键字，顾名思义，ignoring是忽略掉某些标签，用剩下的标签来做匹配。我们拿 Prometheus 文档中的例子来说明：

```
1 ## example series
2 method_code:http_errors:rate5m{method="get", code="500"} 24
3 method_code:http_errors:rate5m{method="get", code="404"} 30
4 method_code:http_errors:rate5m{method="put", code="501"} 3
5 method_code:http_errors:rate5m{method="post", code="500"} 6
```

```

6 method_code:http_errors:rate5m{method="post", code="404"} 21
7 method:http_requests:rate5m{method="get"} 600
8 method:http_requests:rate5m{method="del"} 34
9 method:http_requests:rate5m{method="post"} 120
10
11 ## promql
12 method_code:http_errors:rate5m{code="500"}
13 / ignoring(code)
14 method:http_requests:rate5m
15
16 ## result
17 {method="get"} 0.04 // 24 / 600
18 {method="post"} 0.05 // 6 / 120

```

## group\_left 和 group\_right

这两个关键词用于 one-to-many 和 many-to-one 的匹配场景，left、right 指向高基数的那一侧的 vector。还是拿上面的 method\_code:http\_errors:rate5m 和 method:http\_requests:rate5m 这两指标来做例子，使用 group\_left 的 promql 和结果如下：

```

1 ## promql
2 method_code:http_errors:rate5m
3 / ignoring(code) group_left
4 method:http_requests:rate5m
5
6 ## result
7 {method="get", code="500"} 0.04 // 24 / 600
8 {method="get", code="404"} 0.05 // 30 / 600
9 {method="post", code="500"} 0.05 // 6 / 120
10 {method="post", code="404"} 0.175 // 21 / 120

```

比如针对 method="get" 的条目，右侧的vector中只有一个记录，但是左侧的vector中有两个记录，所以高基数的一侧是左侧，故而使用 group\_left。

另外举一个例子，说明 group\_left group\_right 的一个常见用法，比如我们使用 kube-state-metrics 来采集 Kubernetes 各个对象的指标数据，其中针对 pod 有个指标是 kube\_pod\_labels，会把 pod 的一些信息放到这个指标的标签里，指标值是1，相当于一个元信息，比如：

```

1 kube_pod_labels{
2   [...]
3   label_name="frontdoor",
4   label_version="1.0.1",
5   label_team="blue"
6   namespace="default",
7   pod="frontdoor-xxxxxxxx-xxxxxx",
8 } = 1

```



假设某个 Pod 是接入层的，统计了很多 HTTP 请求相关的指标，我们想统计 5xx 的请求数量，希望能按 Pod 的 version 画一个饼图。这里有个难点：接入层这个 Pod 没有 version 标签，version 信息只是出现在 kube\_pod\_labels 中，如何让二者联动呢？上答案：

```
1 sum(  
2   rate(http_request_count{code=~"^(?:5..)$"}[5m])) by (pod)  
3 *  
4 on (pod) group_left(label_version) kube_pod_labels
```

我们来掰开揉碎这个 promql 看一下具体的意思，乘号前面的部分，是一个典型的统计每秒 5xx 数量的语法，group by pod。

然后我们乘以 kube\_pod\_labels，这个值是1，所以对整体数值没有影响，而 kube\_pod\_labels 有多个标签，而且和sum语句的结果vector的标签不一致，所以通过 on(pod) 的语法指定只是按照pod标签来做对应关系。

最后，利用 group\_left(label\_version) 把 label\_version 附加到了结果向量中，高基数的部分显然是sum的部分，所以是group\_left而非group\_right。

## 聚合运算

针对单个指标的多个 series，比如100台机器的 mem\_available\_percent，可能会有一些聚合需求，比如想查看这100台机器的平均内存可用率，或者排个序，取数值最小的10台。这种需求使用promql内置的聚合函数来做。

- `sum` (calculate sum over dimensions)
- `min` (select minimum over dimensions)
- `max` (select maximum over dimensions)
- `avg` (calculate the average over dimensions)
- `group` (all values in the resulting vector are 1)
- `stddev` (calculate population standard deviation over dimensions)
- `stdvar` (calculate population standard variance over dimensions)
- `count` (count number of elements in the vector)
- `count_values` (count number of elements with the same value)
- `bottomk` (smallest k elements by sample value)
- `topk` (largest k elements by sample value)
- `quantile` (calculate  $\varphi$ -quantile ( $0 \leq \varphi \leq 1$ ) over dimensions)

比如取平均值和取最小的2个可用率数据，其对应的 promql 如下：

```
1 avg(mem_available_percent{app="clickhouse"})  
2 bottomk(2, mem_available_percent{app="clickhouse"})
```

另外，我们有时会有分组统计的需求，比如我想分别统计clickhouse和canal的机器的内存可用率，可以使用by关键字指定分组统计的维度（与by相反的是without）：

```
1 avg(mem_available_percent{app=~"clickhouse|canal"}) by (app)
```

## 函数

Prometheus 函数非常多，具体文档参考：<https://prometheus.io/docs/prometheus/latest/querying/functions/> 这一节我们举例说明一些常用的函数。

### absent\_over\_time

接收一个 range-vector，如果range-vector是空，则返回1，表示absent，如果range-vector有内容，则什么都不返回。

这个特性在生产环境下可以用作nodata告警，比如：

```
1 absent_over_time(system_load_norm_1{ident="tt-fc-dev02.nj"}[5m])
```

这个promql表示，tt-fc-dev02.nj 这个机器在最近5m内如果上报过system\_load\_norm\_1指标，即 tt-fc-dev02.nj 机器存活，则什么都不返回，如果机器挂了，不再上报监控数据了，即指标在最近5m内不存在了，即可判断机器失联。

这种方法有个弊端，就是得把指标的所有标签都写上，比如我们的需求可能是，100台机器，任何一台失联了就告警，想当然的我们可能会这么写：

```
1 absent_over_time(system_load_norm_1[5m])
```

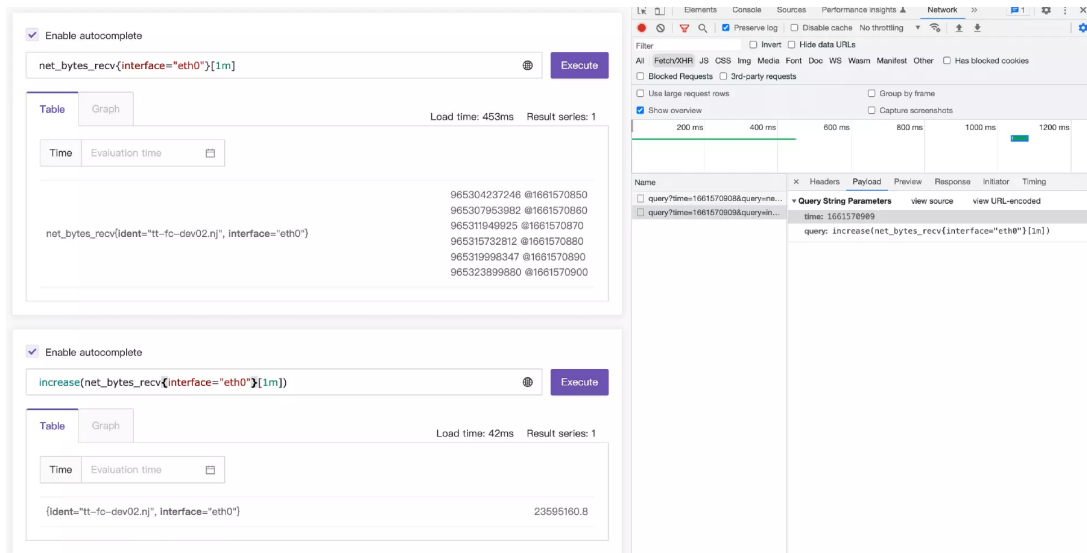
很遗憾，这个结果不符合预期，只要任一台机器有在上报监控数据，这个promql就返回空，即使已经有99台机器挂了，还剩最后一台机器在上报监控数据，这个promql也仍然返回空。

所以实际上，如果我们想要对100台机器使用absent\_over\_time做失联告警，就要配置100条告警规则，每个规则里的promql都要把机器标识信息写上。

🔴 对于拉模式的监控系统，比如 Prometheus，很容易判断机器失联，因为 pull 不到数据了，就知道 target 挂了，通过 up 指标就可以告警；对于推模式的监控系统，比如 Open-Falcon、Datadog、Nightingale，就不好搞了。所以夜莺的告警规则里专门做了一个机器告警类型，用于机器失联告警。

### increase

这个函数很常用，但是其计算结果可能会出乎意料，这一节详细讲解，打消各位的疑问。字面意思上，表示求取一个增量，接收一个 range-vector，range-vector 显然是会返回多个 value+timestamp 的组合，我们直观理解就是，直接把时间范围内最后一个值减去第一个值，不就可以得到增量了吗？非也！如下图：



这个图上的一些关键信息，我们摘录出文本，具体如下：

```

1 promql: net_bytes_recv{interface="eth0"}[1m] @ 1661570908
2 965304237246 @1661570850
3 965307953982 @1661570860
4 965311949925 @1661570870
5 965315732812 @1661570880
6 965319998347 @1661570890
7 965323899880 @1661570900
8
9 promql: increase(net_bytes_recv{interface="eth0"}[1m]) @1661570909
10 23595160.8

```

监控数据是10秒上报一次，所以虽然两次 promql 查询时间不同，一次是 1661570908，一次是 1661570909，但是所查询的原始数据内容是一样的，就是 1661570850~1661570900 这几个时间点对应的数据。

直观上理解，在这几个时间点对应的数据上求取 increase，无非就是最后一个值减去第一个值，即 $965323899880 - 965304237246 = 19662634$ ，很遗憾，实际结果是23595160.8，差别有点大，显然这个直观理解的算法是错的。

实际上，increase 这个 promql 发起请求的时间是1661570909，时间范围是[1m]，相当于告诉Prometheus，我要查询1661570849 ( 1661570909-60 ) ~1661570909之间的 increase 数值。但是原始监控数据并没有 1661570849、1661570909 这两个时刻的数值，怎么办呢？Prometheus只能基于现有的数据做外推，即使用最后一个点的数值减去第一个点的数值，得到的结果除以时间差，再乘以60，即：

```

1 (965323899880.0-965304237246.0)/(1661570900.0-1661570850.0)*60=
2 23595160.8

```

上例中，我的测试数据是没有缺失数据点的，如果有缺失数据点的情况，数据外推会更复杂，具体可以参考这篇文章：<https://mp.weixin.qq.com/s/9aiqrtLTnzysV9oIMx-rzA>

## rate

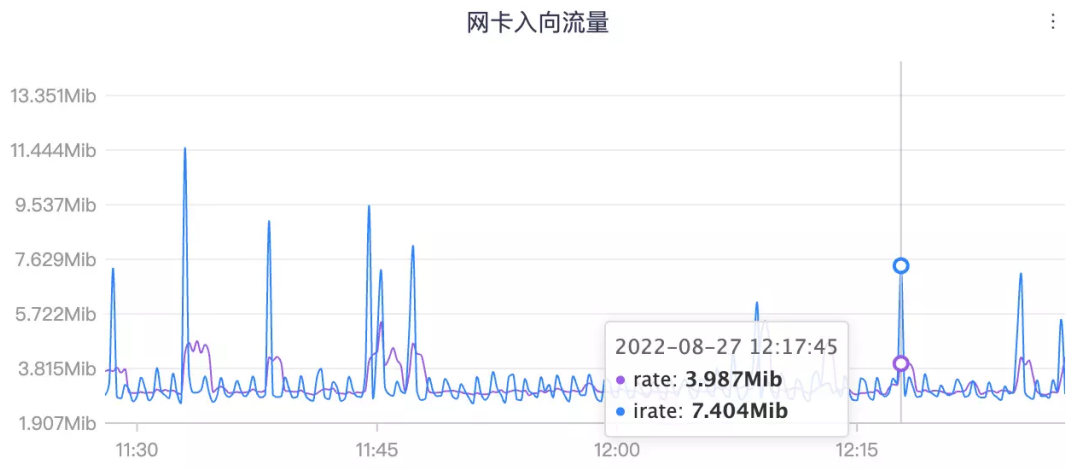
趁热打铁，说一下 rate 函数，increase 函数是求取的时间段内的增量，而且有数据外推，rate 函数则求取的每秒变化率，也有数据外推的逻辑，相当于 increase 的结果除以 range-vector 的时间段的大小，就是 rate 的值。我们用如下 promql 做验证：

```
1 rate(net_bytes_recv{interface="eth0"}[1m])
2 == bool
3 increase(net_bytes_recv{interface="eth0"}[1m])/60.0
```

这里 == 后面跟了一个 bool 修饰符，表示希望返回一个 bool 值，如果是 true 就会返回 1，如果是 false 就返回 0，我们观察结果会发现，这个表达式永远都会返回 1，即等号前后的两个 promql 语义上是相同的。

## irate

rate 函数求取的变化率，相对平滑，因为是拿时间范围内的最后一个值和第一个值做数据外推，一些毛刺现象就会被平滑掉，如果想要得到更敏感的数据，可以使用 irate 函数。irate 是拿时间范围内的最后两个值来做计算，变化就会更剧烈，我们还是拿网卡入向流量这个指标来做对比：



蓝色的更变化更剧烈的线是 irate 函数计算的，紫色的相对平滑的线是 rate 函数计算得到的。

## histogram\_quantile

要了解 histogram\_quantile 函数的用法，首先得了解 Histogram 类型的数据。Histogram 翻译过来是柱状图，设计这个数据类型，是为了描述响应延时的情况。

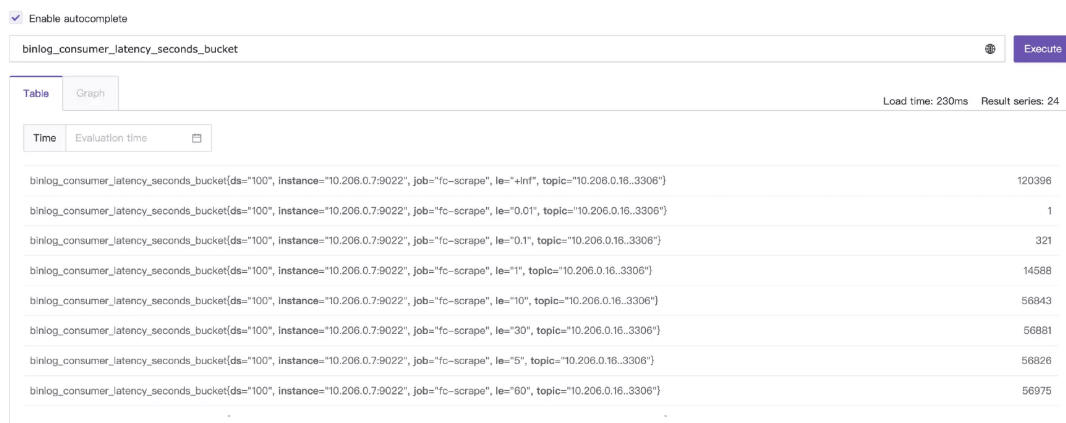
比如接口：/api/v1/query，如何度量这个接口的健康状况？最核心有两个指标，一个是成功率，一个是延迟，成功率的计算代价比较小，只需要为每个请求指标打上 statuscode 的标签即可，然后可以求取非 5xx 非 4xx 的请求占比，即可得到成功的数量，除以总量就是成功率。

而对于延迟，如果只是求取平均延迟，代价也比较小，只要把请求总量做成一个 Counter 指标，把耗时总量做成一个 Counter 指标即可。但是，平均响应时间有时并不能很好的反应长尾问题，比如最近1分钟有1万个请求，大部分请求都是1秒内返回，有200个请求是10秒返回，平均响应时间是：1.18秒，看起来还不错，导致我们忽略了这200个长尾请求，而这200个长尾请求，可能恰好是暴露问题的200个请求。

所以在看延迟数据时，我们通常会用分位值，比如99分位，90分位，50分位，所谓的分位值，就是把一段时间内的所有延迟数据从小到大排序，99分位就是看第99%位置的那个值的大小。还是上面的例子，平均响应时间是1.18秒，但是99分位时间是10秒，相差巨大，更容易暴露问题。这里所谓的99分位延迟10秒，可以理解为，99%的请求都在10秒内返回。

从监控系统角度，如何来存储和计算出99分位值呢？如果每分钟有1亿个请求，难道真的要在监控系统中存储这1亿个请求，然后排序，然后求取分位值？那这个代价就太大了。监控数据是采样数据，对准确性要求没有那么的高，有没有什么办法可以降低这个代价呢？这就是 Prometheus Histogram 的设计初衷了。

Histogram 类型，是把延迟数据分到多个桶里，比如下面的例子，我们查询一个bucket指标看看效果，虽然这个指标的桶划分不是很合理，也可以说明问题：



Time	Evaluation time	
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="+Inf", topic="10.206.0.16..3306"}		120396
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="0.01", topic="10.206.0.16..3306"}		1
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="0.1", topic="10.206.0.16..3306"}		321
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="1", topic="10.206.0.16..3306"}		14588
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="5", topic="10.206.0.16..3306"}		56843
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="10", topic="10.206.0.16..3306"}		56881
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="30", topic="10.206.0.16..3306"}		56826
binlog_consumer_latency_seconds_bucket{ds="100", instance="10.206.0.7:9022", job="fc-scrape", le="60", topic="10.206.0.16..3306"}		56975

binlog\_consumer\_latency\_seconds\_bucket 这个指标，有一个非常非常重要的标签叫 le，表示桶上限，上面的例子就表示，binlog的consume延迟数据分成了6个桶，分别统计了每个桶的总的consume次数：

- 1 延迟小于 0.01 秒的次数： 1
- 2 延迟小于 0.1 秒的次数： 321
- 3 延迟小于 1 秒的次数： 14588
- 4 延迟小于 5 秒的次数： 56826
- 5 延迟小于 10 秒的次数： 56843
- 6 延迟小于 30 秒的次数： 56881
- 7 延迟小于 60 秒的次数： 56975
- 8 所有consume总数 : 120396

假设我们统计50分位，那就是 $120396 * 0.5 = 60198.0$ ，落到了 le="+Inf" 这个桶里了，所以我们断定50分位的值一定是大于60秒的，当然，因为这个桶划分不是很合理，导致，90分位、99分位，定然也是在 le="+Inf" 桶里，即值一定是大于60秒的，因为 le="+Inf" 这个桶没有上界，导致我们无法区分这几个分位值。

下面我们假设一个指标及其数据，做一个算法演示，假设指标名是 `http_request_duration_seconds_bucket`，其各个 bucket 的值如下：

```
1 http_request_duration_seconds_bucket{job="n9e-proxy", le="0.1"} 500
2 http_request_duration_seconds_bucket{job="n9e-proxy", le="1"} 700
3 http_request_duration_seconds_bucket{job="n9e-proxy", le="10"} 850
4 http_request_duration_seconds_bucket{job="n9e-proxy", le="20"} 1000
5 http_request_duration_seconds_bucket{job="n9e-proxy", le="+Inf"} 1000
```

根据这个数据，我们可以计算出落在各个延迟区间的请求数量，如下：

```
1 0 ~ 0.1 : 500
2 0.1 ~ 1 : 200
3 1 ~ 10 : 150
4 10 ~ 20 : 150
5 20 ~ +Inf : 0
```

总共有1000个请求，我们来计算其90分位的值，即 $1000 \times 0.9 = 900$ ，第900个请求，显然，第900个请求落在了10~20这个区间，即90分位的延迟是10秒~20秒，那具体是多少？其实是无法知晓的，不过 Prometheus 的 `histogram_quantile` 有个估计算法，它假设落在各个 bucket 的数据是均匀分布的，即10~20这个区间的150个请求，延迟最小的那个请求是10s，延迟最大的那个请求是20秒，总的第900个请求，就是这个区间的第50个请求，其延迟数据大概是：

```
1 (20-10)*(50/150)+10=13s
```

这是假设数据是均匀分布在各个桶的，假设10~20那个桶的150个请求，最大延迟的那个请求，其延迟数据是11秒，而这里算出13秒，显然与现实不符，不符也没办法，这本来就是预估，知道大概数量级就可以了，还是那句话，监控数据是采样数据，这么计算虽然不是那么准确，但是成本低。

实际上，我们基于某个指标的历史所有数据计算分位值，意义不大，通常我们是基于最近一段时间的增量数据来计算，比如基于10分钟区间的增量数据计算，就可以较为方便的知道，当前这个10分钟的延迟是多少，上一个10分钟的延迟是多少。`histogram_quantile` 接收两个参数，第一个是分位标量，第二个 `instant-vector`（这个vector的标签中一定要有 `le` 标签），举例：

```
1 histogram_quantile(0.9, rate(http_request_duration_seconds_bucket[10m]))
```

上面的例子，是会对每个请求分别做计算，假设有两个模块：`n9e-proxy`、`n9e-webapi`，都统计了 `http_request_duration_seconds_bucket`，我们可能希望以模块为颗粒度，分别计算每个模块的90分位延迟，写法是：

```
1 histogram_quantile(
2 0.9,
3 sum by (job, le) (rate(http_request_duration_seconds_bucket[10m]))
4 )
```

注意，这里通过 `job` 标签来区分模块，`le` 是计算 `histogram_quantile` 必须的，所以也要放到 `sum by` 后面，如果我们要计算全部数据的90分位值呢（虽然这大概率是个伪需求）？

```
1 histogram_quantile(
2 0.9,
```

```
3 sum by (le) (rate(http_request_duration_seconds_bucket[10m]))
4 )
```

针对分位值的计算，已经阐述清楚了，但是分位值的计算是个挺重的查询，可能会把后端时序库打爆，所以很多公司可能在业务埋点SDK中不提供histogram这种方式，只提供summary方式。

所谓的summary，也是prometheus的一种埋点数据类型，summary也可以计算90分位、99分位的值，但是这个值不是通过promql在服务端计算的，而是在应用的内存里，在SDK层面计算的，即客户端把这个分位值算好，再上报给服务端，服务端就无需通过histogram\_quantile这么重的函数做计算了，而是直接查看就好。

但是，既然是在客户端SDK层面计算，就会产生局限，这些分位值只能是实例级别（或者说进程级别，因为SDK是在应用进程里运行的）的分位值，这个是个问题？

笔者看来，这是个问题，但是这个问题不是特别严重，如果要求全局的90分位值，可以把所有实例的90分位值取个平均，虽然不是那么准确，也凑合能用。而实际上，对于一个服务部署多个实例的场景，通常这多个实例是负载均衡的，查看其中一个实例的分位值和查看总体的分位值理论上差不太多。而且，如果某个机器有问题，比如某个机器磁盘故障，导致部署在上面的实例异常，延迟变高，其他实例都是正常的，全局查看延迟数据的时候，每个实例是一条曲线，那个故障的机器，对应的曲线应该是恰好严重偏离其他曲线，正好可以借机知道具体是哪个实例/机器出了问题。

## <aggregation>\_over\_time

这类聚合函数和聚合运算章节提供的sum、avg等聚合运算符非常像，容易混淆，着重做一个说明，比如avg，参数是instant-vector，是在同一时刻，对多个series的多个值求平均，而avg\_over\_time，参数是range-vector，是根据指定的时间范围，求取时间范围内的多个值的平均。

比如 avg\_over\_time(mem\_available\_percent{ident="10.3.4.5"}[1m]) 是取10.3.4.5这个机器的内存可用率，取其最近1分钟内的多个值（如果10秒上报一次，1分钟内有6个值），求平均。

更多函数就不过多介绍了，相对容易理解，参考 Prometheus 官方文档即可。最后扩展介绍一个 MetricsQL（MetricsQL 是 VictoriaMetrics 提供的一种查询语言，兼容 PromQL 并对其做了增强，如果你的存储是 VictoriaMetrics，则可以使用这些扩展函数）中的扩展函数。

## count\_gt\_over\_time

假设原始需求：某个指标（假设指标名字是 interface\_status）每分钟上报一次，如果5分钟内有3次大于10，就报警。使用 PromQL 比较难写，使用 MetricsQL 就非常简单：

```
1 count_gt_over_time(interface_status[5m], 10) >= 3
```

看到这个写法，基本能直观理解其含义了，`count_gt_over_time(series_selector[d], gt)` 函数有两个参数，一个是 range-vector，一个是标量 `gt`，表示在 range-vector 中大于 `gt` 的个数，如果大于等于 3，就报警。除了 `count_gt_over_time` 函数之外，还有 `count_le_over_time`、`count_ne_over_time`、`count_eq_over_time` 道理相同。

## 小结

上面的知识点是 PromQL 的常规知识，尽量融入了一些生产实践的场景，当然，PromQL 还有更多函数没有介绍，大家可以阅读其文档学习。

我是来自快猫星云 (<https://flashcat.cloud/>) 的秦晓辉，在监控/可观测性道路上，伴你前行 :-)