PromQL讲解与实战操作

**一、PromQL 介绍**

PromQL（Prometheus Query Language）是 Prometheus 内置的数据查询语言，它能实现对事件序列数据的查询、聚合、逻辑运算等。它并且被广泛应用在 Prometheus 的日常应用当中，包括对数据查询、可视化、告警处理当中。

简单地说，PromQL 广泛存在于以 Prometheus 为核心的监控体系中。所以需要用到数据筛选的地方，就会用到PromQL。例如：监控指标的设置、报警指标的设置等等。

当Prometheus 通过 Exporter 采集到相应的监控指标样本数据后，我们就可以通过 PromQL 对监控样本数据进行查询。

PromQL（Prometheus Query Language）为 Prometheus tsdb 的查询语言，是结合 grafana 进行数据展示和告警规则的配置的关键部分。

官方文档：<https://prometheus.io/docs/prometheus/latest/querying/basics/>



**二、四种指标类型**

在 Prometheus 中，我们所有的信息都以 Metrics（指标） 的形式存在。

Metrics 由 metric name 和 label name 组成。

<metric name>{<label name>=<label value>, ...}

例如下面的 api\_http\_requests\_total 就是 metrics name（指标名称），而 method 就是 label name（标签），POST 就是label value（标签值）。而 metric name 加上 label name 就是一个完整的 Metric。

api\_http\_requests\_total{method="POST", handler="/messages"}

Prometheus 中主要有四种不同的指标类型，用来适应不同的指标类型。

* counter（计数器）
* gauge （仪表类型）
* histogram（直方图类型）
* summary （摘要类型）

1、counter（计数器）

数据从 0 开始累计，理想状态下应该是永远增长或者是不变。

适用于例如机器开机时间、HTTP 访问量等数值。

Counter (只增不减的计数器) 类型的指标其工作方式和计数器一样，只增不减。常见的监控指标，如：

http\_requests\_total、 node\_cpu\_seconds\_total 都是 Counter 类型的监控指标。

在 node-exporter 返回的样本数据中，其注释中也包含了该样本的类型。例如：

1|*# HELP node\_cpu\_seconds\_total Seconds the cpus spent in each mode.*

2|*# TYPE node\_cpu\_seconds\_total counter*

3|node\_cpu\_seconds\_total{cpu="cpu0",mode="idle"} 362812.7890625

#HELP：解释当前指标的含义，上面表示在每种模式下 node 节点的 cpu 花费的时间，以 s 为单位。

#TYPE：说明当前指标的数据类型，上面是 counter 类型。

counter 是一个简单但又强大的工具，例如我们可以在应用程序中记录某些事件发生的次数，通过以时间序列的形式存储这些数据，我们可以轻松的了解该事件产生的速率变化。PromQL 内置的聚合操作和函数可以让用户对这些数据进行进一步的分析，例如，通过 rate() 函数获取 HTTP 请求量的增长率：

rate(http\_requests\_total[5m])

查询当前系统中，访问量前 10 的 HTTP 请求，使用 topk()函数查询：

topk(10, http\_requests\_total)

2、gauge （仪表类型）

获取一个返回值，采集回来是多少就是多少。数值可能升高，也可能降低。

适用于例如硬盘容量、CPU 内存使用率等数值。

与 Counter 不同， Gauge（可增可减的仪表盘）类型的指标侧重于反应系统的当前状态，因此这类指标的样本数据可增可减。常见指标如：node\_memory\_MemFree\_bytes（主机当前空闲的内存大小）、node\_memory\_MemAvailable\_bytes（可用内存大小）都是 Gauge 类型的监控指标。

通过 Gauge 指标，用户可以直接查看系统的当前状态：

node\_memory\_MemFree\_bytes

对于 Gauge 类型的监控指标，通过 PromQL 内置函数 delta() 可以获取样本在一段时间范围内的变化情况。例

如，计算 CPU 温度在两个小时内的差异：

delta(cpu\_temp\_celsius{host="zeus"}[2h])

还可以直接使用predict\_linear() 对数据的变化趋势进行预测。例如，预测系统磁盘空间在4 个小时之后的剩余情况：

predict\_linear(node\_filesystem\_free\_bytes[1h], 4 \* 3600)

3、histogram（直方图类型）

counter 和 gauges 反应的是数值的情况，而 histogram 则是反应数值的分布情况。

histogram 柱状图反映了样本的区间分布梳理，经常用来表示请求持续时间、响应大小等信息。

例如我们 1 分钟内有 1000 个 http 请求，我们想要知道大多数的请求耗时是多少。这时我们使用平均耗时可能不太准，但是我们使用 histogram 柱状图就可以看出这些请求大多数都是分布在哪个耗时区间。

4、Summary（摘要类型）

Summary 同样表示样本的分布情况，与 Histogram 类似，其会有总数、数量表示。但其多了一个是中位数的表示。常用来表示类似于：请求持续时间、响应大小的信息。

5、Histogram（直方图类型） 和 Summary（摘要类型）

除了 Counter 和 Gauge 类型的监控指标以外，Prometheus 还定义了 Histogram 和 Summary 的指标类型，

Histogram 和 Summary 主用用于统计和分析样本的分布情况。

在大多数情况下人们都倾向于使用某些量化指标的平均值，例如 CPU 的平均使用率、页面的平均响应时间，这种方式也有很明显的问题，以系统 API 调用的平均响应时间为例：如果大多数 API 请求都维持在 100ms的响应时间范围内，而个别请求的响应时间需要 5s，那么就会导致某些 WEB 页面的响应时间落到中位数上，而这种现象被称为长尾问题。

为了区分是平均的慢还是长尾的慢，最简单的方式就是按照请求延迟的范围进行分组。例如，统计延迟在0~10ms 之间的请求数有多少而 10~20ms 之间的请求数又有多少。通过这种方式可以快速分析系统慢的原因。Histogram 和 Summary 都是为了能够解决这样的问题存在的，通过 Histogram 和 Summary 类型的监控指标，我们可以快速了解监控样本的分布情况。

例如，指标 prometheus\_tsdb\_wal\_fsync\_duration\_seconds 的指标类型为 Summary。它记录了Prometheus Server中 wal\_fsync 的处理时间，通过访问 Prometheus Server 的 /metrics 地址，可以获取到以下监控样本数据：

1|*# HELP prometheus\_tsdb\_wal\_fsync\_duration\_seconds Duration of WAL fsync.*

2|*# TYPE prometheus\_tsdb\_wal\_fsync\_duration\_seconds summary*

3|prometheus\_tsdb\_wal\_fsync\_duration\_seconds{quantile="0.5"} 0.012352463

4|prometheus\_tsdb\_wal\_fsync\_duration\_seconds{quantile="0.9"} 0.014458005

5|prometheus\_tsdb\_wal\_fsync\_duration\_seconds{quantile="0.99"} 0.017316173

6|prometheus\_tsdb\_wal\_fsync\_duration\_seconds\_sum 2.888716127000002

7|prometheus\_tsdb\_wal\_fsync\_duration\_seconds\_count 216

从上面的样本中可以得知当前 Prometheus Server 进行 wal\_fsync 操作的总次数为 216 次，耗时2.888716127000002s。其中中位数（quantile=0.5）的耗时为 0.012352463，9 分位数（quantile=0.9）的耗时为 0.014458005s。

在 Prometheus Server 自身返回的样本数据中，我们还能找到类型为 Histogram 的监控指标prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket：

1|# *HELP prometheus\_tsdb\_compaction\_chunk\_range\_seconds Final time range of chunks on their first compaction*

2|# *TYPE prometheus\_tsdb\_compaction\_chunk\_range\_seconds histogram*

3|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="100"} 71

4|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="400"} 71

5|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="1600"} 71

6|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="6400"} 71

7|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="25600"} 405

8|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="102400"} 25690

9prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="409600"} 71863

10|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="1.6384e+06"} 115928

11prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="6.5536e+06"} 2.5687892e+07

12|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="2.62144e+07"} 2.5687896e+07

13|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_bucket{le="+Inf"} 2.5687896e+07

14|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_sum 4.7728699529576e+13

15|prometheus\_tsdb\_compaction\_chunk\_range\_seconds\_count 2.5687896e+07

与 Summary 类型的指标相似之处在于 Histogram 类型的样本同样会反应当前指标的记录的总数(以 \_count作为后缀)以及其值的总量（以\_sum 作为后缀）。不同在于Histogram 指标直接反应了在不同区间内样本的个数，区间通过标签 le 进行定义。

Histogram 指标直接反应了在不同区间内样本的个数，区间通过标签 len 进行定义。而 summary 则是使用中位数反映样本的情况。

**三、表达式四种数据类型**

PromQL 查询语句即表达式，实现的四种数据类型：

* 瞬时向量（Instant vector）：一组时间序列，每个时间序列包含单个样本，它们共享相同的时间戳。也就是说，表达式的返回值中只会包含该时间序列中最新的一个样本值。
* 区间向量（Range vector）：一组时间序列，每个时间序列包含一段时间范围内的样本数据。
* 标量（Scalar）：一个浮点型的数据值，没有时序。可以写成[-]（digits）[.（digits）]的形式。需要注意的是，使用表达式 count（http\_requests\_total）返回的数据类型依然是瞬时向量，用户可以通过内置函数scalar()将单个瞬时向量转换为标量。
* 字符串（String）：一个简单的字符串值。字符串可以用单引号、双引号或反引号来指定。

**1、瞬时向量（Instant vector）**

Instance vector（瞬时向量）表示一个时间序列的集合，但是每个时序只有最近的一个点，而不是线。

通过类似表达式查询时间序列时，同一个指标同一标签只会返回一条数据。这样的表达式我们称之为瞬间向量表达式，而返回的结果称之为瞬间向量。

**2、区间向量（Range vector）**

Range vector（范围向量）表示一段时间范围里的时序，每个时序可包含多个点 。

如果我们想查询一段时间范围内的样本数据，那么我们就需要用到区间向量表达式，其查询出来的结果称之为区间

向量。时间范围通过时间范围选择器 [] 进行定义。

**3、标量（Scalar）**

Scalar（标量）通常为数值，可以将只有一个时序的 Instance vector 转换成 Scalar。

**4、字符串（String）**

一个简单的字符串值。字符串可以用单引号、双引号或反引号来指定。

四、时间序列（向量）

按照时间顺序记录系统、设备状态变化的数据，每个数据成为一个样本。

* 数据采集以特定的时间周期进行，因而，随着时间流逝，将这些样本数据记录下来，将生成一个离散的样本数据序列。
* 该序列也称为向量（Vector），以时间轴为横坐标、序列为纵坐标，这些数据点连接起来就会形成一个矩阵。

**1、时间序列的构成**

每条时间序列（Time Series）是通过指标名称（Metrics name）和一组标签集（Label set）来命名的。

如果 time 相同，但是指标名称或者标签集不同，那么时间序列也不同。

**2、样本构成**

矩阵中每一个点都可称为一个样本（Sample），样本主要由 3 方面构成。

* 指标（Metrics）：包括指标名称（Metrics name）和一组标签集（Label set）名称，如

request\_total{path=“/status”，method=“GET”}。

* 时间戳（TimeStamp）：这个值默认精确到毫秒。
* 样本值（Value）：这个值默认使用 Float64 浮点类型。

时间序列的指标（Metrics）存储格式为 key-value。

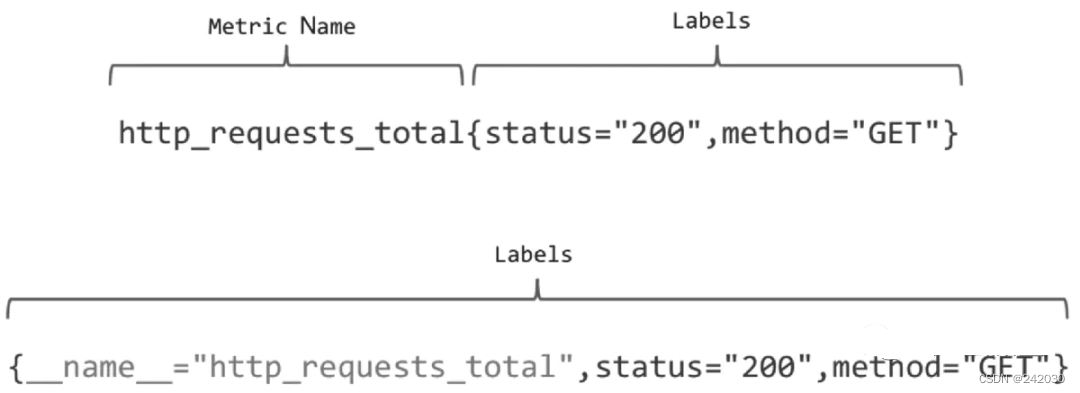
http\_request\_total{status="200"，method="GET"}@1434417560938=>94355 为例，在 Key-Value 关系中，94355 作为 Value（也就是样本值 Sample Value），前面的

http\_request\_total{status="200"，method="GET"}@1434417560938 一律作为 Key。

**3、key 的组成**

* Metric Name：指标名（例子中的 http\_request\_total）
* Label：标签（例子中的{status=“200”，method=“GET”}）
* Timestamp：时间戳（例子中的@1434417560938）

Prometheus Metrics 两种表现形式：



指标名称只能由ASCII字符、数字、下划线以及冒号组成并必须符合正则表达式[a-zA-Z\_:][a-zA-Z0-9\_:]，冒号用来表示用户自定义的记录规则，不能在Exporter中或监控对象直接暴露的指标中使用冒号来定义指标名称。

**五、标签过滤器 4 种运算符**

* =：与字符串匹配
* !=：与字符串不匹配
* =~：与正则匹配
* !~：与正则不匹配

PromQL 支持户根据时间序列的标签匹配模式来对时间序列进行过滤，目前主要支持两种匹配模式：完全匹配和正则匹配。

PromQL 支持使用 = 和！= 两种完全匹配模式。

* 等于：通过使用 label=value 可以选择那些标签满足表达式定义的时间序列。
* 不等于：通过使用 label!=value 则可以根据标签匹配排除时间序列。

PromQL 还可以使用正则表达式作为匹配条件，并且可以使用多个匹配条件。

* 正向匹配：使用 label=~regx 表示选择那些标签符合正则表达式定义的时间序列。
* 反向匹配：使用 label!~regx 进行排除。

PromQL 查询语句中的正则表达式匹配使用 RE2 语法：

https://github.com/google/re2/wiki/Syntax

**1、匹配器（Matcher）**

匹配器是作用于标签上的，标签匹配器可以对时间序列进行过滤，Prometheus 支持完全匹配和正则匹配两种模式：完全匹配和正则表达式匹配。

**2、完全匹配**

**2.1 相等匹配器（=）**

相等匹配器（Equality Matcher），用于选择与提供的字符串完全相同的标签。下面介绍的例子中就会使用相等匹配器按照条件进行一系列过滤。

node\_cpu\_seconds\_total{instance="ydzs-master"}

**2.2 不相等匹配器（!=）**

不相等匹配器（Negative Equality Matcher），用于选择与提供的字符串不相同的标签。它和相等匹配器是完全相反的。举个例子，如果想要查看 job 并不是 HelloWorld 的 HTTP 请求总数，可以使用如下不相等匹配器。

node\_cpu\_seconds\_total{job!="helloworld"}

**3、正则表达式匹配**

**3.1 正则表达式匹配器（=~）**

正则表达式[匹配器（Regular Expression Matcher），用于选择与提供的字符串进行正则运算后所得结果相匹配的标签。Prometheus 的正则运算是强指定的，比如正则表达式 a 只会匹配到字符串 a，而并不会匹配到 ab 或者ba 或者 abc。如果你不想使用这样的强指定功能，可以在正则表达式的前面或者后面加上.\*。比如下面的例子

表示 job 是所有以 Hello 开头的 HTTP 请求总数。

node\_cpu\_seconds\_total{job=~"hello.\*", mode="idle"}

node\_cpu\_seconds\_total直接等效于{\_\_name\_\_="node\_cpu\_seconds\_total"}，后者也可以使用和前者一样的 4 种匹配器（=，!=，=，!）。比如下面的案例可以表示所有以 Hello 开头的指标：

{\_\_name\_\_="node\_cpu\_seconds\_total",job=~"hello.\*", mode="idle"}

**3.2 正则表达式相反匹配器（!~）**

正则表达式相反匹配器（Negative Regular Expression Matcher），用于选择与提供的字符串进行正则运算后所得结果不匹配的标签。因为 PromQL 的正则表达式基于 RE2 的语法，但是 RE2 不支持向前不匹配表达式，所以!~的出现是作为一种替代方案，以实现基于正则表达式排除指定标签值的功能。在一个选择器当中，可以针对同一个标签来使用多个匹配器。比如下面的例子，可以实现查找 job 名是 node 且安装在/prometheus 目录下，

但是并不在/prometheus/user 目录下的所有文件系统并确定其大小。

node\_filesystem\_size\_bytes{job="node",mountpoint=~"/prometheus/.\*", mountpoint!~ "/prometheus/user/.\*"}

**六、范围选择器**

我们可以通过将时间范围选择器 range vector selectors 附加到查询语句中，指定为每个返回的区间向量样本值中提取多长的时间范围。每个时间戳的值都是按时间倒序记录在时间序列中的，该值是从时间范围内的时间戳获取的对应的值。

时间范围通过数字来表示，单位可以使用以下其中之一的时间单位：

* ms - 毫秒
* s - 秒
* m - 分钟
* h - 小时
* d - 天
* w - 周
* y - 年

比如node\_cpu\_seconds\_total{instance="ydzs-master",mode="idle"}这个查询语句，如果添加上[1m]

这个时间范围选择器，则我们可以得到如下所示的信息：

node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="idle"}[1m]

这是因为现在每一个时间序列中都有多个时间戳多个值，所以没办法渲染，必须是标量或者瞬时向量才可以绘制图形。

不过通常区间向量都会应用一个函数后变成可以绘制的瞬时向量，Prometheus 中对瞬时向量和区间向量有很多操作的函数，不过对于区间向量来说最常用的函数并不多，使用最频繁的有如下几个函数：

* rate()：计算整个时间范围内区间向量中时间序列的每秒平均增长率。
* irate()：仅使用时间范围中的最后两个数据点来计算区间向量中时间序列的每秒平均增长率， irate 只能用于绘制快速变化的序列，在长期趋势分析或者告警中更推荐使用 rate 函数。
* increase()：计算所选时间范围内时间序列的增量，它基本上是速率乘以时间范围选择器中的秒数。

**七、PromQL 运算符**

**1、数学运算符**

数学运算符比较简单，就是简单的加减乘除等。

例如：我们通过prometheus\_http\_response\_size\_bytes\_sum可以查询到 Prometheus 这个应用的 HTTP 响应字节总和。但是这个单位是字节，我们希望用 MB 显示。那么我们可以这么设置：

prometheus\_http\_response\_size\_bytes\_sum/8/1024

PromQL 支持的所有数学运算符如下所示：

+ (加法)

- (减法)

\* (乘法)

/ (除法)

% (求余)

^ (幂运算)

**2、布尔运算符**

布尔运算符支持用户根据时间序列中样本的值，对时间序列进行过滤。

例如：我们可以通过prometheus\_http\_requests\_total查询出每个接口的请求次数，但是如果我们想筛选出请求次数超过 20 次的接口呢？

此时我们可以用下面的 PromQL 表达式：

prometheus\_http\_requests\_total > 20

从上面的图中我们可以看到，value 的值还是具体的数值。但如果我们希望对符合条件的数据，value 变为 1。不符合条件的数据，value 变为 0。那么我们可以使用 bool 修饰符。

我们使用下面的 PromQL 表达式：

prometheus\_http\_requests\_total > bool 20

目前，Prometheus 支持以下布尔运算符如下：

* ==（相等）
* !=（不相等）
* >（大于）
* <（小于）
* >=（大于或等于）
* <=（小于或等于）

**3、集合运算符**

通过集合运算，可以在两个瞬时向量与瞬时向量之间进行相应的集合操作。目前，Prometheus 支持以下集合运算符：

* and 与操作
* or 或操作
* unless 排除操作

**1、and 与操作**

vector1 and vector2 进行一个与操作，会产生一个新的集合。该集合中的元素同时在 vector1 和 vector2 中都存在。

例如：我们有 vector1 为 A B C，vector2 为 B C D，那么 vector1 and vector2 的结果为：B C。

**2、or 或操作**

vector1 and vector2 进行一个或操作，会产生一个新的集合。该集合中包含 vector1 和 vector2 中的所有元素。

例如：我们有 vector1 为 A B C，vector2 为 B C D，那么 vector1 or vector2 的结果为：A B C D。

**3、unless 排除操作**

vector1 and vector2 进行一个或操作，会产生一个新的集合。该集合首先取 vector1 集合的所有元素，然后排除掉所有在 vector2 中存在的元素。

例如：我们有 vector1 为 A B C，vector2 为 B C D，那么 vector1 unless vector2 的结果为：A。

**4、操作符优先级**

在 PromQL 操作符中优先级由高到低依次为：

* ^
* \*, /, %
* +, -
* ==, !=, <=, <, >=, >
* and, unless
* or

**八、PromQL 内置函数**

Prometheus 提供了其它大量的内置函数，可以对时序数据进行丰富的处理。某些函数有默认的参数，例如：year(v=vector(time()) instant-vector)。其中参数 v 是一个瞬时向量，如果不提供该参数，将使用默认值

vector(time())。instant-vector 表示参数类型。

**1、abs()**

1|*# 返回输入向量的所有样本的绝对值*

2|abs(v instant-vector)

abs(node\_cpu\_seconds\_total)

**2、absent()**

absent(v instant-vector)，如果传递给它的向量参数具有样本数据，则返回空向量；如果传递的向量参数没有样本数据，则返回不带度量指标名称且带有标签的时间序列，且样本值为 1。

当监控度量指标时，如果获取到的样本数据是空的， 使用 absent 方法对告警是非常有用的。例如：

1|*# 这里提供的向量有样本数据*

2|absent(http\_requests\_total{method="get"}) => no data

3|absent(sum(http\_requests\_total{method="get"})) => no data

4|# *由于不存在度量指标nonexistent,所以返回不带度量指标名称且带有标签的时间序列,且样本值为1*

5|absent(nonexistent{job="myjob"}) => {job="myjob"} 1

6|# *正则匹配的instance不作为返回labels中的一部分*

7|absent(nonexistent{job="myjob",instance=~".\*"}) => {job="myjob"} 1

8|# sum函数返回的时间序列不带有标签,且没有样本数据

9|absent(sum(nonexistent{job="myjob"})) => {} 1

**3、ceil()**

ceil(v instant-vector) 将 v 中所有元素的样本值向上四舍五入到最接近的整数。例如：

node\_cpu\_seconds\_total{ mode="softirq"} # 结果为 615.18

ceil(node\_cpu\_seconds\_total{ mode="softirq"}) # 结果为 616

**4、changes()**

changes(v range-vector) 输入一个区间向量， 返回这个区间向量内每个样本数据值变化的次数（瞬时向量）。例如：

1|# *如果样本数据值没有发生变化,则返回结果为 1*

2|changes(node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"}[3m]) # 结果为 2

**5、clamp\_max()**

clamp\_max(v instant-vector, max scalar)函数，输入一个瞬时向量和最大值，样本数据值若大于 max，则改为 max，否则不变。例如：

1|node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"} # 结果为 615.21

2|clamp\_max(node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"}, 700) # 结果为615.21

3|clamp\_max(node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"}, 600) # 结果为600

**6、clamp\_min()**

clamp\_min(v instant-vector, min scalar)函数，输入一个瞬时向量和最小值，样本数据值若小于 min，则改为 min，否则不变。例如：

1|node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"} # 结果为 615.21

2|amp\_min(node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"}, 700) # 结果为700

3|amp\_min(node\_cpu\_seconds\_total{instance="192.168.58.195:8088",mode="softirq"}, 600) # 结果为615.21

**7、day\_of\_month()**

day\_of\_month(v=vector(time()) instant-vector)函数，返回被给定 UTC 时间所在月的第几天，返回值范围：1~31。

day\_of\_month() # 结果 {} 28

**8、day\_of\_week()**

day\_of\_week(v=vector(time()) instant-vector)函数，返回被给定 UTC 时间所在周的第几天，返回值范围：0~6，0 表示星期天。

day\_of\_week() # 结果 {} 2

**9、days\_in\_month()**

days\_in\_month(v=vector(time()) instant-vector)函数，返回当月一共有多少天，返回值范围：28~31。

days\_in\_month() # 结果 {} 28

**10、delta()**

delta(v range-vector)的参数是一个区间向量，返回一个瞬时向量。它计算一个区间向量 v 的第一个元素和最后一个元素之间的差值。由于这个值被外推到指定的整个时间范围，所以即使样本值都是整数，你仍然可能会得到一个非整数值。

delta(node\_cpu\_seconds\_total{}[100m])

**11、deriv()**

deriv(v range-vector)的参数是一个区间向量，返回一个瞬时向量。它使用简单的线性回归计算区间向量 v 中各个时间序列的导数。这个函数一般只用在 Gauge 类型的时间序列上。deriv(node\_cooling\_device\_max\_state{name="0",type="Processor"}[10m])

**12、exp()**

exp(v instant-vector) 函数，输入一个瞬时向量，返回各个样本值的 e 的指数值，即 e 的 N 次方。当 N 的值足够大时会返回 +Inf。特殊情况为：

1|Exp(+Inf) = +Inf

2|Exp(NaN) = NaN

1|node\_cooling\_device\_max\_state{name="0",type="Processor"} # 结果为 10

2|exp(node\_cooling\_device\_max\_state{name="0",type="Processor"}) # 结果为 22026.465794806718

**13、floor()**

floor(v instant-vector)函数与 ceil() 函数相反，将 v 中所有元素的样本值向下四舍五入到最接近的整数。

1|node\_cpu\_seconds\_total{ mode="softirq"} # 结果为 615.18

2|floor(node\_cpu\_seconds\_total{ mode="softirq"}) # 结果为 615

**14、histogram\_quantile()**

histogram\_quantile(φ float, b instant-vector)从 bucket 类型的向量 b 中计算 φ (0 ≤ φ ≤ 1) 分位数（百分位数的一般形式）的样本的最大值。（有关 φ 分位数的详细说明以及直方图指标类型的使用，请参阅直方图和摘要）。向量 b 中的样本是每个 bucket 的采样点数量。每个样本的 labels 中必须要有 le 这个 label 来表示每个bucket 的上边界，没有 le 标签的样本会被忽略。直方图指标类型自动提供带有 \_bucket 后缀和相应标签的时间序列。可以使用 rate() 函数来指定分位数计算的时间窗口。

例如：一个直方图指标名称为 employee\_age\_bucket\_bucket，要计算过去 10 分钟内 第 90 个百分位数，请使用

以下表达式：

histogram\_quantile(0.9, rate(employee\_age\_bucket\_bucket[10m]))

返回：

{instance="10.0.86.71:8080",job="prometheus"} 35.714285714285715

这表示最近 10 分钟之内 90% 的样本的最大值为 35.714285714285715。

**15、holt\_winters()**

holt\_winters(v range-vector, sf scalar, tf scalar)函数基于区间向量 v，生成时间序列数据平滑值。平滑因子 sf 越低, 对旧数据的重视程度越高。趋势因子 tf 越高，对数据的趋势的考虑就越多。其中，0< sf, tf<=1。holt\_winters 仅适用于 Gauge 类型的时间序列。

**16、hour()**

hour(v=vector(time()) instant-vector)函数返回被给定 UTC 时间的当前第几个小时，时间范围：0~23。

hour() # 结果 {} 5

**17、idelta()**

idelta(v range-vector)的参数是一个区间向量， 返回一个瞬时向量。它计算最新的 2 个样本值之间的差值。

这个函数一般只用在 Gauge 类型的时间序列上。

idelta(prometheus\_notifications\_queue\_capacity [5m])

**18、increase()**

increase(v range-vector)函数获取区间向量中的第一个和最后一个样本并返回其增长量, 它会在单调性发生变化时(如由于采样目标重启引起的计数器复位)自动中断。由于这个值被外推到指定的整个时间范围，所以即使样本值都是整数，你仍然可能会得到一个非整数值。

例如：以下表达式返回区间向量中每个时间序列过去 5 分钟内 HTTP 请求数的增长数：

increase(prometheus\_http\_requests\_total[5m])

increase 的返回值类型只能是计数器类型，主要作用是增加图表和数据的可读性。使用 rate 函数记录规则的使用率，以便持续跟踪数据样本值的变化。

**19、irate()**

irate(v range-vector)函数用于计算区间向量的增长率，但是其反应出的是瞬时增长率。irate 函数是通过区间向量中最后两个两本数据来计算区间向量的增长速率，它会在单调性发生变化时(如由于采样目标重启引起的计数器复位)自动中断。这种方式可以避免在时间窗口范围内的“长尾问题”，并且体现出更好的灵敏度，通过 irate 函数绘制的图标能够更好的反应样本数据的瞬时变化状态。

例如：以下表达式返回区间向量中每个时间序列过去 5 分钟内最后两个样本数据的 HTTP 请求数的增长率：

irate(prometheus\_http\_requests\_total[5m])

**20、label\_join()**

label\_join(v instant-vector, dst\_label string, separator string, src\_label\_1 string,src\_label\_2 string, ...)函数可以将时间序列 v 中多个标签 src\_label 的值，通过 separator 作为连接符写入到一个新的标签 dst\_label 中，可以有多个 src\_label 标签。

1|label\_join(up,"instancejob","-","instance","job")

2|*# 结果*

3|up{instance="192.168.58.195:8088", instancejob="192.168.58.195:8088-node", job="node"}

4|up{instance="192.168.58.195:9090", instancejob="192.168.58.195:9090-prometheus", job="prometheus"} 1

**21、label\_replace()**

为了能够让客户端的图标更具有可读性，可以通过 label\_replace 函数为时间序列添加额外的标签。label\_replace的具体参数如下：

label\_replace(v instant-vector, dst\_label string, replacement string, src\_label string, regex string)

该函数会依次对 v 中的每一条时间序列进行处理，通过 regex 匹配 src\_label 的值，并将匹配部分 relacement 写入到 dst\_label 标签中。如下所示：

1|up

2|*# 结果*

3|up{instance="192.168.58.195:8088", job="node"} 1

4|up{instance="192.168.58.195:9090", job="prometheus"} 1

1|label\_replace(up, "host", "$1", "instance", "(.\*):.\*")

2*|# 结果*

3|up{host="192.168.58.195", instance="192.168.58.195:8088", job="node"} 1

4|up{host="192.168.58.195", instance="192.168.58.195:9090", job="prometheus"} 1

函数处理后，时间序列将包含一个 host 标签，host 标签的值为 Exporter 实例的 IP 地址。

1|label\_replace(up, "port", "$1", "instance", ".\*:(.\*)")

2|*# 结果*

3|up{instance="192.168.58.195:8088", job="node", port="8088"} 1

4|up{instance="192.168.58.195:9090", job="prometheus", port="9090"} 1

函数处理后，时间序列将包含一个 port 标签，port 标签的值为 Exporter 实例的 port。

**22、ln()**

ln(v instant-vector) 计算瞬时向量 v 中所有样本数据的自然对数。特殊情况：

* ln(+Inf) = +Inf
* ln(0) = -Inf
* ln(x < 0) = NaN
* ln(NaN) = NaN

ln(prometheus\_http\_requests\_total)

**23、log2()**

log2(v instant-vector)函数计算瞬时向量 v 中所有样本数据的二进制对数。特殊情况同上。

log2(prometheus\_http\_requests\_total)

**24、log10()**

log10(v instant-vector)计算瞬时向量 v 中所有样本数据的十进制对数。特殊情况同上。

log10(prometheus\_http\_requests\_total)

**25、minute()**

minute(v=vector(time()) instant-vector)函数返回给定 UTC 时间当前小时的第多少分钟，结果范围：0~59。

minute() # 结果 {} 38

**26、month()**

month(v=vector(time()) instant-vector)函数返回给定 UTC 时间当前属于第几个月，结果范围：0~12。

month() # 结果 {} 2

**27、predict\_linear()**

predict\_linear(v range-vector, t scalar)函数可以预测时间序列 v 在 t 秒后的值。它基于简单线性回归的方式，对时间窗口内的样本数据进行统计，从而可以对时间序列的变化趋势做出预测。该函数的返回结果不带有度量指标，只有标签列表。

例如，基于 2 小时的样本数据，来预测主机可用磁盘空间的是否在 4 个小时候被占满，可以使用如下表达式：

1|*# 小于0的代表占满*

2|predict\_linear(node\_filesystem\_free{job="node"}[2h], 4 \* 3600) < 0

**28、rate()**

rate(v range-vector)函数可以直接计算区间向量 v 在时间窗口内平均增长速率，它会在单调性发生变化时(如由于采样目标重启引起的计数器复位)自动中断。该函数的返回结果不带有度量指标，只有标签列表。

例如，以下表达式返回区间向量中每个时间序列过去 5 分钟内 HTTP 请求数的每秒增长率：

rate(prometheus\_http\_requests\_total[5m])

rate()函数返回值类型只能用计数器，在长期趋势分析或者告警中推荐使用这个函数。

注意：

当将 rate() 函数与聚合运算符（例如 sum()）或随时间聚合的函数（任何以 \_over\_time 结尾的函数）一起使用时，必须先执行 rate 函数，然后再进行聚合操作，否则当采样目标重新启动时 rate() 无法检测到计数器是否被重置。

# *两者的结果等价*

increase(node\_cpu\_seconds\_total[2m]) / 120

rate(node\_cpu\_seconds\_total[2m])

需要注意的是使用 rate 或者 increase 函数去计算样本的平均增长速率，容易陷入「长尾问题」当中，其无法反应在时间窗口内样本数据的突发变化。

为了解决该问题，PromQL 提供了另外一个灵敏度更高的函数 irate(v range-vector)。irate 同样用于计算区间向量的计算率，但是其反应出的是瞬时增长率。irate 函数是通过区间向量中最后两个样本数据来计算区间向量的增长速率。

这种方式可以避免在时间窗口范围内的「长尾问题」，并且体现出更好的灵敏度，通过 irate 函数绘制的图标能够更好的反应样本数据的瞬时变化状态。

irate(node\_cpu\_seconds\_total[2m])

irate函数相比于rate函数提供了更高的灵敏度，不过当需要分析长期趋势或者在告警规则中，irate 的这种灵敏度反而容易造成干扰。因此在长期趋势分析或者告警中更推荐使用 rate函数。

**29、resets()**

resets(v range-vector)的参数是一个区间向量。对于每个时间序列，它都返回一个计数器重置的次数。两个连续样本之间的值的减少被认为是一次计数器重置。

这个函数一般只用在计数器类型的时间序列上。

**30、round()**

round(v instant-vector, to\_nearest=1 scalar) 函数与ceil和floor函数类似，返回向量中所有样本值的最接近的整数。to\_nearest参数是可选的，默认为 1，表示样本返回的是最接近 1 的整数倍的值。你也可以将该参数指定为任意值（也可以是小数），表示样本返回的是最接近它的整数倍的值。

round(prometheus\_http\_requests\_total{})

**31、scalar()**

scalar(v instant-vector)函数的参数是一个单元素的瞬时向量，它返回其唯一的时间序列的值作为一个标量。如果度量指标的样本数量大于 1 或者等于 0, 则返回 NaN。

scalar(prometheus\_http\_requests\_total{handler="/targets"})

**32、sort()**

sort(v instant-vector)函数对向量按元素的值进行升序排序，返回结果：key: value = 度量指标：样本值升序排列。

sort(prometheus\_http\_requests\_total)

**33、sort\_desc()**

sort(v instant-vector) 函数对向量按元素的值进行降序排序，返回结果：key: value = 度量指标：样本值[降序排列]。

sort\_desc(prometheus\_http\_requests\_total)

**34、sqrt()**

sqrt(v instant-vector) 函数计算向量 v 中所有元素的平方根。

sqrt(prometheus\_http\_requests\_total)

**35、time()**

time()函数返回从 1970-01-01 到现在的秒数，注意：它不是直接返回当前时间，而是时间戳。

time() # 结果 scalar 1677562178.602

**36、timestamp()**

timestamp(v instant-vector)函数返回向量 v 中的每个样本的时间戳（从 1970-01-01 到现在的秒数）。该函数从 Prometheus 2.0 版本开始引入。

**37、vector()**

vector(s scalar)函数将标量 s 作为没有标签的向量返回，即返回结果为：key: value= {}, s。

vector(20) # 结果 {} 20

**38、year()**

year(v=vector(time()) instant-vector)函数返回被给定 UTC 时间的当前年份。

year() # 结果 {} 2023

**39、\_over\_time()**

下面的函数列表允许传入一个区间向量，它们会聚合每个时间序列的范围，并返回一个瞬时向量：

1|avg\_over\_time(range-vector) : 区间向量内每个度量指标的平均值。

2|min\_over\_time(range-vector) : 区间向量内每个度量指标的最小值。

3|max\_over\_time(range-vector) : 区间向量内每个度量指标的最大值。

4|sum\_over\_time(range-vector) : 区间向量内每个度量指标的求和。

5|count\_over\_time(range-vector) : 区间向量内每个度量指标的样本数据个数。

6|quantile\_over\_time(scalar, range-vector) : 区间向量内每个度量指标的样本数据值分位数,φ-quantile (0 ≤ φ ≤ 1)。

7|stddev\_over\_time(range-vector) : 区间向量内每个度量指标的总体标准差。

8|stdvar\_over\_time(range-vector) : 区间向量内每个度量指标的总体标准方差。

1|avg\_over\_time(prometheus\_http\_requests\_total{}[10m])

注意：

即使区间向量内的值分布不均匀，它们在聚合时的权重也是相同的。

更多的函数请参考：

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

**八、PromQL 聚合操作**

Prometheus 还提供了聚合操作符，这些操作符作用于瞬时向量。可以将瞬时表达式返回的样本数据进行聚合，形成一个新的时间序列。目前支持的聚合函数有：

* sum (求和)
* min (最小值)
* max (最大值)
* avg (平均值)
* stddev (标准差)
* stdvar (标准方差)
* count (计数)
* count\_values (计算具有相同值的元素个数)
* bottomk (后 n 条时序)
* topk (前 n 条时序)
* group(分组)
* quantile(分位数计算 φ-quantile (0 ≤ φ ≤ 1))

表达式语法：

1|<aggr-op> [without|by (<label list>)] ([parameter,] <vector expression>)

2|或

3|<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]

聚合运算符可通过 without、by根据标签扩展：sum、min、max、avg。

label list：是未加引号的标签列表，比如(code，method)等。

without：用于从计算结果中移除列举的标签，而保留其它标签。

by：和without的作用想法，表示用于从计算结果中保留列举的标签，而移除其它标签。

parameter：只能在count\_values、quantile、topk和bottomk中使用。

count\_values：用于时间序列中每一个样本值出现的次数。count\_values 会为每一个唯一的样本值输出一个时间序列，并且每一个时间序列包含一个额外的标签。这个标签的名字由聚合参数指定，同时这个标签值是唯一的样本值。

quantile：用于计算当前样本数据值的分布情况 quantile(φ, express) ，其中 0 ≤ φ ≤ 1。

**1、sum求和**

用于对记录的 value 值进行求和。

例如：sum(prometheus\_http\_requests\_total) 表示统计所有 HTTP 请求的次数。

sum(prometheus\_http\_requests\_total)

**2、min最小值**

返回所有记录的最小值。

例如：min(prometheus\_http\_requests\_total) 表示获取数据集合中的最小值。

min(prometheus\_http\_requests\_total)

**3、max最大值**

返回所有记录的最大值。

例如：maxmetheus\_http\_requests\_total) 表示获取数据集合中的最大值。

max(prometheus\_http\_requests\_total)

**4、avg平均值**

avg 函数返回所有记录的平均值。

例如：avg(metheus\_http\_requests\_total) 表示获取数据集合中的平均值。

avg(prometheus\_http\_requests\_total)

**5、stddev标准差**

标准差（Standard Deviation）常用来描述数据的波动大小。

例如：统计出不同 HTTP 请求的数量波动情况。

stddev(prometheus\_http\_requests\_total)

**6、count计数**

count 函数返回所有记录的计数。

例如：count(prometheus\_http\_requests\_total) 表示统计所有 HTTP 请求的次数。

count(prometheus\_http\_requests\_total)

**7、count\_values**

1|# *计算每个value的数量*

2|count\_values("value",prometheus\_http\_requests\_total)

**8、bottomk后几条**

bottomk 用于对样本值进行排序，返回当前样本值后 N 位的时间序列。

例如：获取 HTTP 请求量后 5 位的请求，可以使用表达式：

bottomk(5, prometheus\_http\_requests\_total)

**9、topk前几条**

topk 用于对样本值进行排序，返回当前样本值前 N 位的时间序列。

例如：获取 HTTP 请求量前 5 位的请求，可以使用表达式：

topk(5, prometheus\_http\_requests\_total)

**10、group**

group by(handler)(prometheus\_http\_requests\_total)

**11、quantile**

quantile(0.95, prometheus\_http\_requests\_total)

**九、时间位移操作**

在瞬时向量表达式或者区间向量表达式中，都是以当前时间为基准：

1|#*瞬时向量表达式,选择当前最新的数据*

2|Prometheus\_http\_requests\_total{}

3|#*区间向量表达式,选择以当前时间为基准,5分钟内的数据*

4|prometheus\_http\_requests\_total{}[5m]

如果我们想查询 5 分钟前的瞬时样本数据，或昨天一天的区间内的样本数据呢？这个时候我们就可以使用位移操作，位移操作的关键字为 offset。

1*|#查询5分钟前的最新数据*

2|prometheus\_http\_requests\_total{} offset 5m

3|*#同样支持查询range vector*

4|*#往前移动1天,查询1天前的数据*

5|*#例如现在是2020-10-07 00:00:00*

6|*#那么这个表达式查询的数据是2020-10-05至2020-10-06的数据*

7|prometheus\_http\_requests\_total{}[1d] offset 1d

@ 修饰符，还可以通过 @ 直接跳转到某个uinx时间戳，需开启启动参数：

--enable-feature=promql-at-modifier

prometheus\_http\_requests\_total{code="200"} @ 1646089826

**十、标量和字符串**

**1、标量**

在 PromQL 中，标量是一个浮点型的数字值，没有时序，例如：10。

需要注意的是，当使用表达式 count(http\_requests\_total)，返回的数据类型，依然是瞬时向量。用户可以通过内置函数 scalar () 将单个瞬时向量转换为标量。

scalar(sum(prometheus\_http\_requests\_total))

我们将 sum 操作的值用 scalar 转换了一下，最终的结果就是一个标量了。

**2、字符串**

在 PromQL 中，字符串是一个简单的字符串值。直接使用字符串作为 PromQL 表达式，则会直接返回字符串。

"this is a string"

使用字符串 "this is a string" 直接作为 PromQL 查询表达式，结果返回的是一个字符串。

**十一、任务 (Job）和实例（Instance）**

在 Prometheus 中抓取数据的应用叫做实例（Instance），而几个为了同个目的的实例组合起来称之为任务

（Job）。例如下面是一个有 4 个实例的服务工作：

1|*#名为api-server的job*

2|job: api-server

3| instance 1: 1.2.3.4:5670

4| instance 2: 1.2.3.4:5671

5| instance 3: 5.6.7.8:5670

6| instance 4: 5.6.7.8:5671

1|*# Prometheus的配置例如*

2|*# 指标采集配置*

3|scrape\_configs:

4| *# 任务名称*

5| - job\_name: "prometheus"

6| # 实例列表

7| static\_configs:

8| - targets: ["127.0.0.1:9090"]

9| *# 任务名称*

10| - job\_name: "node-exporter"

11| *# 实例列表*

12| static\_configs:

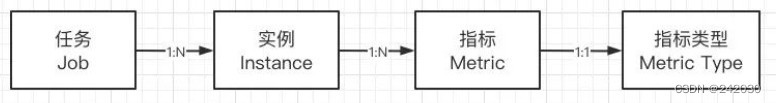
13| - targets:

14| - "192.168.200.101:9100"

15| - "192.168.200.102:9100"

一个任务（Job）可以有多个实例（Instance），一个实例上可以有多个指标（Metric），一个指标只会有一个指标类型（Metric Type）。

它们之间的关系如下图所示：



**十二、数据存储**

Prometheus 2.x 默认将时间序列数据库保存在本地磁盘中，当然，我们也可以将数据保存到第三方的存储服务中。

**1、本地存储**

Prometheus 按照两个小时为一个时间窗口，将两小时内产生的数据存储在一个块（Block）中，每个块都是一个单独的目录，里面包含了对应时间窗口内的所有样本数据（chunks），元数据文件（meta.json）以及索引文件（index）。其中索引文件会将指标名称和标签索引到样板数据的时间序列中。此期间如果通过 API 删除时间序列，删除记录会保存在单独的逻辑文件 tombstone 当中。而样本数据所在的块则会被直接保存在内存中，不会持久化到磁盘中。为了确保 Prometheus 发生崩溃或重启时能够恢复数据，Prometheus 启动时会通过预写日志（write-ahead-log(WAL)）重新记录，从而恢复数据。预写日志文件保存在 wal 目录中，每个文件大小为 128MB。wal 文件包括还没有被压缩的原始数据，所以比常规的块文件大得多。一般情况下，Prometheus 会保留三个 wal 文件，但如果有些高负载服务器需要保存两个小时以上的原始数据，wal 文件的数量就会大于 3 个。

**2、远程存储**

受限于可拓展性和持久性，Prometheus 的本地存储仅限于单个节点，所以 Prometheus 并没有提供集群的存储解决方案，而是提供了一系列的接口，以便和远程存储系统相结合，比如当我们在 prometheus.yml 配置文件里配置了 Remote Write(远程写) 的 URL 地址后，Prometheus 就会将采集到的样本数据通过 HTTP 的形式发送给适配器，后续用户就可以在适配器里对接外部服务了。外部服务可以是真正的存储系统，也可以是云存储、消息队列等。

和 Remote Write(远程写) 一样，当我们在配置文件里配置了 Remote Read(远程读) 的 URL 地址后，就会通过HTTP 的形式到 Adaptor 里查询数据，然后 Adaptor 再去第三方存储服务里获取数据转发回来。

**十三、PromQL 非法总结**

由于所有的 PromQL 表达式必须至少包含一个指标名称，或者至少有一个不会匹配到空字符串的标签过滤器，因此结合 Prometheus 官方文档，可以梳理出如下非法示例。

1|*# .\*表示任意一个字符,这就包括空字符串,且还没有指标名称*

2|{job=~".\*"} # 非法！

3|{job=""} # 非法！

4|{job!=""} # 非法！

相反，如下表达式是合法的。

1|{job=~".+"} # 合法！.+表示至少一个字符

2|{job=~".\*",method="get"} # 合法！.\*表示任意一个字符

3|{job="",method="post"} # 合法！存在一个非空匹配

4|{job=~".+",method="post"} # 合法！存在一个非空匹配

关于 Prometheus PromQL 讲解就先到这里了，其实官网介绍的很清楚，也不难，多使用就很容易掌握了。

**十四、向量匹配**

向量之间的运算尝试为左侧的每个条目在右侧向量中找到匹配的元素。

匹配行为有两种基本类型：一对一、一对多、多对一。

**1、基础知识**

1、向量之间的匹配是在2个即时向量之间进行匹配，不能是区间向量。

2、那么2个向量如何算匹配上呢?

这个就需要2个向量存在完全一致的标签值相同，比如2个向量都存在标签 {code="200"}。

完全一致的标签值：

1、2个向量具有完全相同的标签。

2、使用 on 或 ignoring 后具有相同的标签。

那如果2个向量就是标签不同，但是存在若干个相同的标签，那么怎么比较呢?

使用 on 指定使用那个标签相同，就认为匹配上。

使用 ignoring 忽略某些标签相同，就认为匹配上。

3、找不到匹配的值会被忽略，不会出现在比较的结果中。

**2、示例数据**

示例数据是从prometheus官网上抄的：

1|method\_code:http\_errors:rate5m{method="get", code="500"} 24

2|method\_code:http\_errors:rate5m{method="get", code="404"} 30

3|method\_code:http\_errors:rate5m{method="put", code="501"} 3

4|method\_code:http\_errors:rate5m{method="post", code="500"} 6

5|method\_code:http\_errors:rate5m{method="post", code="404"} 21

6|method:http\_requests:rate5m{method="get"} 600

7|method:http\_requests:rate5m{method="del"} 34

8|method:http\_requests:rate5m{method="post"} 120

**3、一对一**

一对一从操作的每一侧查找一对唯一的条目，如果两个条目具有完全相同的标签集和相应的值，则它们匹配。

ignoring：允许在匹配时忽略某些标签。

on：允许在匹配时只匹配指定的标签。

3.1 语法格式

<vector expr> <bin-op> ignoring(<label list>) <vector expr>

<vector expr> <bin-op> on(<label list>) <vector expr>

3.2 示例

1、查询语句

method\_code:http\_errors:rate5m{code="500"} / ignoring(code) method:http\_requests:rate5m

2、解释

1、method\_code:http\_errors:rate5m{code="500"} 可以查询出如下数据

method\_code:http\_errors:rate5m{method="get", code="500"} 24

method\_code:http\_errors:rate5m{method="post", code="500"} 6

2、method:http\_requests:rate5m 可以查询出如下数据

method:http\_requests:rate5m{method="get"} 600

method:http\_requests:rate5m{method="del"} 34

method:http\_requests:rate5m{method="post"} 120

3、ignoring(code)的作用

因为method\_code:http\_errors:rate5m 比 method:http\_requests:rate5m 多了一个 code 的标签，正常

匹配匹配不上，如果忽略code标签的匹配，那么那边的标签一样且标签的值也一样，所有就可以匹配上。

4、比较的结果

{method="get"} 0.04 // 24 / 600

==> method\_code:http\_errors:rate5m{method="get", code="500"} / method:http\_requests:rate5m{method="get"}

{method="post"} 0.05 // 6 / 120

==> method\_code:http\_errors:rate5m{method="post", code="500"} / method:http\_requests:rate5m{method="post"}

只有2边都能匹配上的数据才会出现，method等于put和del的样本找不到匹配项，所以没有出现在结果集中。

5、为什么没有出现 method=del 的数据

因为 {method="del"} 在 method\_code:http\_errors:rate5m 中没有查询出来，所以在结果中忽略了。即没有

在2边都存在并且相匹配上。

**4、一对多或多对一**

一对多和多对一匹配指的是：一的一侧的每个向量元素可以与多侧的多个元素进行匹配。必须使用group\_left或

group\_right修饰符显式请求，其中 left/right 确定哪个向量具有更高的基数。

4.1 语法格式

<vector expr> <bin-op> ignoring(<label list>) group\_left(<label list>) <vector expr>

<vector expr> <bin-op> ignoring(<label list>) group\_right(<label list>) <vector expr>

<vector expr> <bin-op> on(<label list>) group\_left(<label list>) <vector expr>

<vector expr> <bin-op> on(<label list>) group\_right(<label list>) <vector expr>

4.2 示例

1、查询语句

method\_code:http\_errors:rate5m / ignoring(code) group\_left method:http\_requests:rate5m

2、解释

1、ignoring(code) 表示比较的时候，忽略code标签。

2、group\_left表示，左侧即 method\_code:http\_errors:rate5m 为多的一侧，右侧为一的一侧。

3、没有匹配的直接从结果集中删除。

4、结果如下：

{method="get", code="500"} 0.04 // 24 / 600

==> method\_code:http\_errors:rate5m{method="get", code="500"} / method:http\_requests:rate5m{method="get"}

{method="get", code="404"} 0.05 // 30 / 600

==> method\_code:http\_errors:rate5m{method="get", code="404"} / method:http\_requests:rate5m{method="get"}

{method="post", code="500"} 0.05 // 6 / 120

==> method\_code:http\_errors:rate5m{method="post", code="500"} / method:http\_requests:rate5m{method="post"}

{method="post", code="404"} 0.175 // 21 / 120

==> method\_code:http\_errors:rate5m{method="post", code="404"} / method:http\_requests:rate5m{method="post"}

注意：group 修饰符只能在比较和数学运算符中使用。在逻辑运算 and，unless 和 or 操作中默认与右向量中的所

有元素进行匹配。

**十五、常见的查询**

1|sum(node\_cpu\_seconds\_total{mode="idle"}) by (instance)

2|*#结果*

3|{instance="192.168.58.195:8088"} 1085270.34

1|sum(rate(node\_cpu\_seconds\_total{mode="idle"}[5m]))by(instance)

2|*#结果*

3|{instance="192.168.58.195:8088"} 0.30349916000020105

4|*#注意:Counter计数器一般会和rate结合使用,rate一般会和sum结合使用*

1|*#多个表达式之间使用"|"进行分割*

2|http\_requests\_total{job="HelloWorld", status=~"500|501|502|503|504|505|506|507|509|510"}

1|*#查询5开头的状态码*

2|http\_requests\_total{job="HelloWorld", status=~"5.."}

1|*#匹配不以5开头的*

2|http\_requests\_total{status!~"5.."}