K8S之普罗米修斯

一、部署prometheus

1.1、下载

```
# 注意,此处是有版本匹配的,得匹配对应的k8s版本，具体查看git官网
https://github.com/prometheus-operator/kube-prometheus
git clone -b release-0.7 --single-branch https://github.com/coreos/kube-prometheus.git
```

1.2、安装operator

```
[root@k8s-master01 ~]# cd /root/kube-prometheus/manifests/setup
[root@k8s-master01 setup]# kubectl create -f .

# 查看是否Running
[root@k8s-master01 ~]# kubectl get pod -n monitoring
NAME                                   READY   STATUS        RESTARTS   AGE
prometheus-operator-848d669f6d-bz2tc   2/2     Running       0          4m16s
```

1.3、安装Prometheus

```
# 注意默认数据是不做持久化的,要想持久化数据需要自定义修改
[root@k8s-master01 ~]# cd /root/kube-prometheus/manifests

# 修改数据目录


[root@k8s-master01 manifests]# kubectl create -f .
```

1.4、创建ingress

```
# 创建一下Ingress代理三个service
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  generation: 1
  name: prom-ingresses
  namespace: monitoring
spec:
  rules:
  - host: alert.test.com
    http:
      paths:
      - backend:
          serviceName: alertmanager-main
          servicePort: 9093
        path: /
  - host: grafana.test.com
    http:
```

```
      paths:
      - backend:
          serviceName: grafana
          servicePort: 3000
        path: /
  - host: prome.test.com
    http:
      paths:
      - backend:
          serviceName: prometheus-k8s
          servicePort: 9090
        path: /
```

1.5、页面访问

```
# 在你windows的hosts文件添加主机映射，浏览器访问即可
192.168.1.110 krm.test.com alert.test.com grafana.test.com prome.test.com
```

1.6、卸载

```
kubectl delete --ignore-not-found=true -f manifests/ -f manifests/setup
```

1.7、部署服务简介

- prometheus-operator：prometheus-operator
- 高可用的prometheus-k8s-0：prometheus主要程序
- prometheus-adapter：聚合进apiserver，即一种custom-metrics-apiserver实现
- 高可用的alertmanager-main-0： 告警推送
- grafana： 数据展示 [官网模板下载地址：https://grafana.com/grafana/dashboards/]
- node-exporter：主机监控
- kube-state-metrics：容器监控

```
[root@k8s-master01 ~]# kubectl get po -n monitoring
NAME                                  READY   STATUS    RESTARTS   AGE
alertmanager-main-0                   2/2     Running   0          161m
grafana-f8cd57fcf-s7lkq               1/1     Running   0          161m
kube-state-metrics-587bfd4f97-nmkgr   3/3     Running   0          161m
node-exporter-26z9k                   2/2     Running   0          161m
node-exporter-fxdft                   2/2     Running   0          161m
node-exporter-m2wn7                   2/2     Running   0          161m
node-exporter-mzmx6                   2/2     Running   0          161m
node-exporter-zm699                   2/2     Running   0          161m
prometheus-adapter-69b8496df6-x47m4   1/1     Running   0          161m
prometheus-k8s-0                      2/2     Running   0          161m
prometheus-operator-7649c7454f-dnmql  2/2     Running   0          4h45m
```

1.8、Prometheus如何抓取到数据?

- 云原生应用： /metrics
  - etcd等服务自带 /metrics暴露数据
- 非云原生应用： Exporter
  - MySQL Exporter、Redis Exporter、Node Exporter等

1.8.1、ServiceMonitor是什么?

- serviceMonitor就是定义的一个CRD资源
- 会关联到svc
- 会自动生成Prometheus.yaml配置文件

```
[root@k8s-master01 ~]# kubectl get servicemonitor -n monitoring
NAME                      AGE
alertmanager              107m
coredns                   107m
grafana                   107m
kube-apiserver            107m
kube-controller-manager   107m
kube-scheduler            107m
kube-state-metrics        107m
kubelet                   107m
node-exporter             107m
prometheus                107m
prometheus-adapter        107m

[root@k8s-master01 ~]# kubectl get servicemonitor -n monitoring grafana -oyaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor  # 自定义资源
metadata:
  generation: 1
    manager: kubectl-create
    operation: Update
  name: grafana
  namespace: monitoring
spec:
  endpoints:
  - interval: 15s
    port: http
  selector:
    matchLabels:
      app: grafana
```

- Prometheus读取配置文件的方式
  - 通过prometheus.yal加载配置的部署方式
    - 二进制部署
    - 容器部署
    - Helm部署
  - 通过ServiceMonitor加载配置的部署方式
    - Prometheus Operator
    - kube-Prometheus Stack

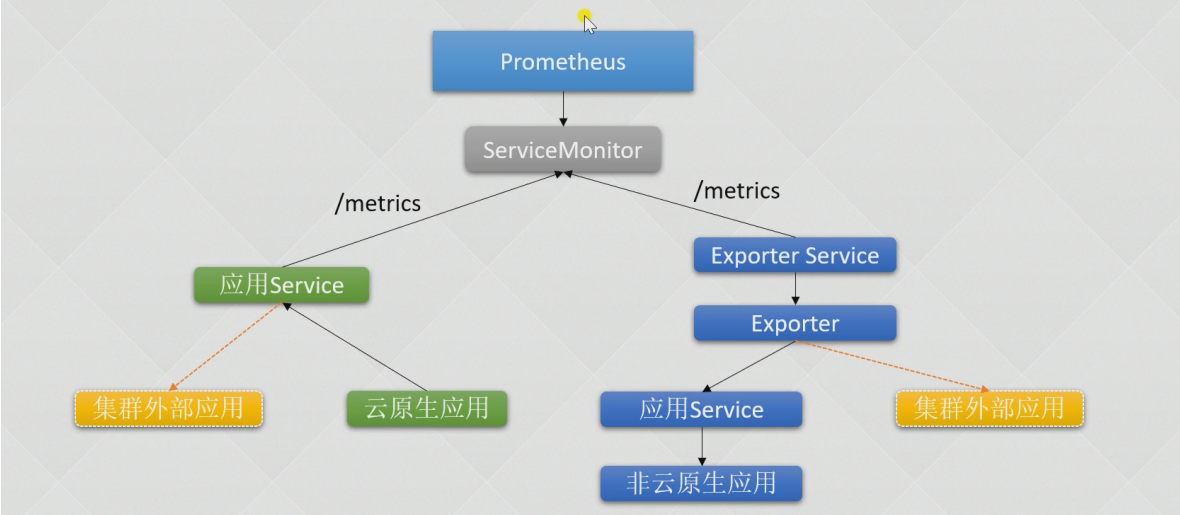
1.8.1、ServiceMonitor配置文件解析

- interval：指定Prometheus对当前endpoints采集的周期。单位为秒，在本次示例中设定为15s
- path： 指定Prometheus的采集路径。在本次示例中，指定为`/actuator/prometheus`
- port： 指定采集数据需要通过的端口，设置的端口为创建Service时端口所设置的`name`
- selector： 监控目标svc的标签
- scheme：Metrics接口的协议
- namespaceSelector： 监控目标svc所在的ns

```
apiVersion: monitoring.coreos.com/v1
```

```yaml
kind: ServiceMonitor
metadata:
  name: micrometer-demo
  namespace: default
spec:
  endpoints:
    - interval: 15s
      path: /actuator/prometheus
      port: metrics
      scheme: http
  namespaceSelector:
    any: true    # any ：有且仅有一个值true，当该字段被设置时，将监听所有符合Selector过滤条件的Service的变动
    # matchNames:  # 数组值，指定需要监听的namespace的范围
    # - default
    # - arms-prom
  selector:
    matchLabels:
      micrometer-prometheus-discovery: 'true'
```

1.9、Prometheus监控流程



二、Metrics指标类型

Prometheus 的客户端库中提供了四种核心的指标类型。但这些类型只是在客户端库（客户端可以根据不同的数据类型调用不同的 API 接口）和在线协议中，实际在 Prometheus server 中并不对指标类型进行区分，而是简单地把这些指标统一视为无类型的时间序列

2.1、Counter （计数器）

Counter 类型代表一种样本数据单调递增的指标，即只增不减，除非监控系统发生了重置。

例如，你可以使用 counter 类型的指标来表示**服务的请求数、已完成的任务数、错误发生的次数**等。counter 主要有两个方法：

```
//将counter值加1.
Inc()

// 将指定值加到counter值上，如果指定值<0 会panic.
Add(float64)
```

Counter 类型数据可以让用户方便的了解事件产生的速率的变化，在 PromQL 内置的相关操作函数可以提供相应的分析，比如以 HTTP 应用请求量来进行说明：

```
//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地址
topk(10, http_requests_total)
```

不要将 counter 类型应用于样本数据非单调递增的指标，**例如：当前运行的进程数量（应该用 Guage 类型）。**

2.2、Guage（仪表盘）

Guage 类型代表一种样本数据可以任意变化的指标，即可增可减。guage 通常用于像温度或者内存使用率这种指标数据，也可以表示能随时增加或减少的“总数”，**例如：当前并发请求的数量。**

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内的变化情况，**例如，计算 CPU 温度在两小时内的差异：**

```
dalta(cpu_temp_celsius{host="zeus"}[2h])
```

你还可以通过PromQL 内置函数 [predict_linear()]基于简单线性回归的方式，**对样本数据的变化趋势做出预测。**例如，基于 2 小时的样本数据，**来预测主机可用磁盘空间在 4 个小时之后的剩余情况：**

```
predict_linear(node_filesystem_free{job="node"}[2h], 4 * 3600) < 0
```

2.3、 Histogram（直方图）

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。这种方式的问题很明显，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的情况，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问题的存在，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。

Histogram 在一段时间范围内对数据进行采样（通常是请求持续时间或响应大小等），并将其计入可配置的存储桶（bucket）中，后续可通过指定区间筛选样本，也可以统计样本总数，最后一般将数据展示为直方图。

Histogram 类型的样本会提供三种指标（假设指标名称为 `<basename>`）：

- 样本的值分布在 bucket 中的数量，命名为 `<basename>_bucket{le="<上边界>"}`。解释的更通俗易懂一点，这个值表示指标值小于等于上边界的所有样本数量。
- ```
  // 在总共2次请求当中。http 请求响应时间 <=0.005 秒 的请求次数为0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
  d="GET",code="200",le="0.005",} 0.0
    // 在总共2次请求当中。http 请求响应时间 <=0.01 秒 的请求次数为0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
  d="GET",code="200",le="0.01",} 0.0
    // 在总共2次请求当中。http 请求响应时间 <=0.025 秒 的请求次数为0
  ```
  ```

```
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.025",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.05",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.075",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.1",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.25",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.5",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="0.75",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="1.0",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="2.5",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="5.0",} 0.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="7.5",} 2.0
  // 在总共2次请求当中。http 请求响应时间 <=10 秒 的请求次数为 2
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="10.0",} 2.0
  io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",metho
d="GET",code="200",le="+Inf",} 2.0
```

- 所有样本值的大小总和，命名为 `<basename>_sum`。

```
  // 实际含义：  发生的2次 http 请求总的响应时间为 13.107670803000001 秒
  io_namespace_http_requests_latency_seconds_histogram_sum{path="/",method="
GET",code="200",} 13.107670803000001
```

- 样本总数，命名为 `<basename>_count`。值和 `<basename>_bucket{le="+Inf"}` 相同。

```
  // 实际含义：  当前一共发生了 2 次 http 请求
  io_namespace_http_requests_latency_seconds_histogram_count{path="/",method
="GET",code="200",} 2.0
```

> **注意**
>
> bucket 可以理解为是对数据指标值域的一个划分，划分的依据应该基于数据值的分布。注意后面的采样点是包含前面的采样点的，假设 `xxx_bucket{...,le="0.01"}` 的值为 10，而 `xxx_bucket{...,le="0.05"}` 的值为 30，那么意味着这 30 个采样点中，有 10 个是小于 10 ms 的，其余 20 个采样点的响应时间是介于 10 ms 和 50 ms 之间的。

可以通过 histogram_quantile() 函数来计算 Histogram 类型样本的分位数。分位数可能不太好理解，你可以理解为分割数据的点。我举个例子，假设样本的 9 分位数（quantile=0.9）的值为 x，即表示小于 x 的采样值的数量占总体采样值的 90%。Histogram 还可以用来计算应用性能指标值（Apdex score）。

2.4、Summary（摘要）

与 Histogram 类型类似，用于表示一段时间内的数据采样结果（通常是请求持续时间或响应大小等），但它直接存储了分位数（通过客户端计算，然后展示出来），而不是通过区间来计算。

Summary 类型的样本也会提供三种指标（假设指标名称为 ）：

- 样本值的分位数分布情况，命名为 `<basename>{quantile="<φ>"}`。

```
  // 含义：这 12 次 http 请求中有 50% 的请求响应时间是 3.052404983s
  io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.5",} 3.052404983
  // 含义：这 12 次 http 请求中有 90% 的请求响应时间是 8.003261666s
  io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.9",} 8.003261666
```

- 所有样本值的大小总和，命名为 `<basename>_sum`。

```
  // 含义：这12次 http 请求的总响应时间为 51.029495508s
  io_namespace_http_requests_latency_seconds_summary_sum{path="/",method="GET",code="200",} 51.029495508
```

- 样本总数，命名为 `<basename>_count`。

```
  // 含义：当前一共发生了 12 次 http 请求
  io_namespace_http_requests_latency_seconds_summary_count{path="/",method="GET",code="200",} 12.0
```

现在可以总结一下 Histogram 与 Summary 的异同：

- 它们都包含了 `<basename>_sum` 和 `<basename>_count` 指标
- Histogram 需要通过 `<basename>_bucket` 来计算分位数，而 Summary 则直接存储了分位数的值。

关于 Summary 与 Histogram 的详细用法，请参考 histograms and summaries。

不同语言关于 Summary 的客户端库使用文档：

三、数据模型

Prometheus 所有采集的监控数据均以指标（metric）的形式**保存在内置的时间序列数据库当中（TSDB）**：属于同一指标名称，同一标签集合的、有时间戳标记的数据流。除了存储的时间序列，Prometheus 还可以根据查询请求产生临时的、衍生的时间序列作为返回结果。

3.1、指标名称和标签

每一条时间序列由指标名称（Metrics Name）以及一组标签（键值对）唯一标识。其中指标的名称（metric name）可以反映被监控样本的含义（例如，`http_requests_total` — 表示当前系统接收到的 HTTP 请求总量），指标名称只能由 ASCII 字符、数字、下划线以及冒号组成，同时必须匹配正则表达式 `[a-zA-Z_:][a-zA-Z0-9_:]*`。

`（时间序列[唯一标识]）=指标名称+标签）`

> 注意：
> 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。

通过使用标签，Prometheus 开启了强大的多维数据模型：对于相同的指标名称，通过不同标签列表的集合，会形成特定的度量维度实例（例如：所有包含度量名称为 `/api/tracks` 的 http 请求，打上 `method=POST` 的标签，就会形成具体的 http 请求）。该查询语言在这些指标和标签列表的基础上进行过滤和聚合。改变任何度量指标上的任何标签值（包括添加或删除指标），都会创建新的时间序列。

标签的名称只能由 ASCII 字符、数字以及下划线组成并满足正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*`。其中以 `__` 作为前缀的标签，是系统保留的关键字，只能在系统内部使用。标签的值则可以包含任何 `Unicode` 编码的字符。

3.2、样本（sample）

在时间序列中的每一个点称为一个样本（sample），样本由以下三部分组成：

- 指标（metric）：指标名称和描述当前样本特征的 labelsets；
- 时间戳（timestamp）：一个精确到毫秒的时间戳；
- 样本值（value）： 一个 folat64 的浮点型数据表示当前样本的值。

3.3、表示方式

通过如下表达方式表示指定指标名称和指定标签集合的时间序列：

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

例如，指标名称为 `api_http_requests_total`，标签为 `method="POST"` 和 `handler="/messages"` 的时间序列可以表示为：

```
api_http_requests_total{method="POST", handler="/messages"}
```

四、PromQL简介

4.1、PromQL简介

Prometheus 提供了一种功能表达式语言 `PromQL`，允许用户实时选择和汇聚时间序列数据。表达式的结果可以在浏览器中显示为图形，也可以显示为表格数据，或者由外部系统通过 HTTP API 调用。

4.2、表达式语言数据类型

在 Prometheus 的表达式语言中，表达式或子表达式包括以下四种类型之一：

- **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的最 新的一个样本值。而相应的这样的表达式称之为**瞬时向量表达式。**
- **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。
- **字符串（String）**
  - 一个简单的字符串值。

根用户输入的表达式返回的数据类型是否合法取决于用例的不同，例如：瞬时向量表达式返回的数据类型是唯一可以直接绘制成图表的数据类型。

4.3、字面量

4.3.1、字符串

字符串可以用单引号、双引号或反引号指定为文字常量。

PromQL 遵循与 Go 相同的转义规则。在单引号或双引号中，用反斜杠来表示转义序列，后面可以跟 `a`, `b`, `f`, `n`, `r`, `t`, `v` 或 `\`。特殊字符可以使用八进制（`\nnn`）或者十六进制（`\xnn`，`\unnnn` 和 `\Unnnnnnnn`）。

与 Go 不同，**Prometheus 不会对反引号内的换行符进行转义。**

例如：

```
"this is a string"
'these are unescaped: \n \\ \t'
`these are not unescaped: \n ' " \t`
```

4.3.2、字符串

标量浮点值可以字面上写成 `[-](digits)[.(digits)]` 的形式。

```
-2.43
```

4.4、时间序列过滤器

4.4.1、瞬时向量过滤器

瞬时向量过滤器允许在指定的时间戳内选择一组时间序列和每个时间序列的单个样本值。在最简单的形式中，近指定指标（metric）名称。这将生成包含此指标名称的所有时间序列的元素的瞬时向量。

例如：选择指标名称为 `http_requests_total` 的所有时间序列：

```
http_requests_total
```

可以通过向花括号（`{}`）里附加一组标签来进一步过滤时间序列。

例如：选择指标名称为 `http_requests_total`，`job` 标签值为 `prometheus`，`group` 标签值为 `canary` 的时间序列：

```
http_requests_total{job="prometheus",group="canary"}
```

PromQL 还支持用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。总共有以下几种标签匹配运算符：

- `=`：选择与提供的字符串完全相同的标签。
- `!=`：选择与提供的字符串不相同的标签。
- `=~`：选择正则表达式与提供的字符串（或子字符串）相匹配的标签。
- `!~`：选择正则表达式与提供的字符串（或子字符串）不匹配的标签。

例如：选择指标名称为 `http_requests_total`，环境为 `staging`、`testing` 或 `development`，HTTP 方法为 `GET` 的时间序列：

```
http_requests_total{environment=~"staging|testing|development",method!="GET"}
```

没有指定标签的标签过滤器会选择该指标名称的所有时间序列。

所有的 PromQL 表达式必须至少包含一个指标名称，或者一个不会匹配到空字符串的标签过滤器。

以下表达式是非法的（因为会匹配到空字符串）：

```
{job=~".*"} # 非法!
```

以下表达式是合法的：

```
{job=~".+"}               # 合法!
{job=~".*",method="get"} # 合法!
```

除了使用 `<metric name>{label=value}` 的形式以外，我们还可以使用内置的 `__name__` 标签来指定监控指标名称。例如：表达式 `http_requests_total` 等效于 `{__name__="http_requests_total"}`。也可以使用除 `=` 之外的过滤器（`=`，`=~`，`~`）。以下表达式选择指标名称以 `job:` 开头的所有指标：

```
{__name__=~"job:.*"}
```

Prometheus 中的所有正则表达式都使用 RE2语法。

4.4.2、区间向量过滤器

区间向量与瞬时向量的工作方式类似，唯一的差异在于在区间向量表达式中我们需要定义时间选择的范围，时间范围通过时间范围选择器 `[]` 进行定义，以指定应为每个返回的区间向量样本值中提取多长的时间范围。

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：

- `s` - 秒
- `m` - 分钟
- `h` - 小时
- `d` - 天
- `w` - 周
- `y` - 年

例如：选择在过去 5 分钟内指标名称为 `http_requests_total`，`job` 标签值为 `prometheus` 的所有时间序列：

```
http_requests_total{job="prometheus"}[5m]
```

4.4.3、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：

```
http_request_total{} # 瞬时向量表达式，选择当前最新的数据
http_request_total{}[5m] # 区间向量表达式，选择以当前时间为基准，5分钟内的数据
```

而如果我们想查询，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`。

例如，以下表达式返回相对于当前查询时间过去 5 分钟的 `http_requests_total` 值：

```
http_requests_total offset 5m
```

注意： `offset` 关键字需要紧跟在选择器（`{}`）后面。以下表达式是正确的：

```
sum(http_requests_total{method="GET"} offset 5m) // GOOD.
```

下面的表达式是不合法的：

```
sum(http_requests_total{method="GET"}) offset 5m // INVALID.
```

该操作同样适用于区间向量。以下表达式返回指标 `http_requests_total` 一周前的 5 分钟之内的 HTTP 请求量的增长率：

```
rate(http_requests_total[5m] offset 1w)
```

五、Prometheus 操作符

5.1、二元运算符

Prometheus 的查询语言支持基本的逻辑运算和算术运算。对于两个瞬时向量, 匹配行为可以被改变。

5.1.1、算术二元运算符

在 Prometheus 系统中支持下面的二元算术运算符：

- `+` 加法
- `-` 减法
- `*` 乘法
- `/` 除法
- `%` 模
- `^` 幂等

二元运算操作符支持 `scalar/scalar(标量/标量)`、`vector/scalar(向量/标量)`、和 `vector/vector(向量/向量)` 之间的操作。

在两个**标量**之间进行数学运算，得到的结果也是**标量。**

在**向量和标量之间**，这个运算符会作用于这个向量的每个样本值上。例如：如果一个时间序列瞬时向量除以 2，**操作结果也是一个新的瞬时向量**，且度量指标名称不变, 它是原度量指标瞬时向量的每个样本值除以 2。

如果是瞬时向量与瞬时向量之间进行数学运算时，过程会相对复杂一点，运算符会依次找到与左边向量元素匹配（标签完全一致）的右边向量元素进行运算，如果没找到匹配元素，则直接丢弃。同时新的时间序列将不会包含指标名称。

例如，如果我们想根据 `node_disk_bytes_written` 和 `node_disk_bytes_read` 获取主机磁盘IO的总量，可以使用如下表达式：

```
node_disk_bytes_written + node_disk_bytes_read
```

该表达式返回结果的示例如下所示：

```
{device="sda",instance="localhost:9100",job="node_exporter"}=>1634967552@1518146
427.807 + 864551424@1518146427.807
{device="sdb",instance="localhost:9100",job="node_exporter"}=>0@1
```

5.1.2、布尔运算符

目前，Prometheus 支持以下布尔运算符：

- `==` (相等)
- `!=` (不相等)
- `>` (大于)
- `<` (小于)
- `>=` (大于等于)
- `<=` (小于等于)

布尔运算符被应用于 `scalar/scalar（标量/标量）`、`vector/scalar（向量/标量）`，和 `vector/vector（向量/向量）`。默认情况下布尔运算符只会根据时间序列中样本的值，对时间序列进行过滤。我们可以通过在运算符后面使用 `bool` 修饰符来改变布尔运算的默认行为。使用 bool 修改符后，布尔运算不会对时间序列进行过滤，而是直接依次瞬时向量中的各个样本数据与标量的比较结果 `0` 或者 `1`。

在两个标量之间进行布尔运算，必须提供 bool 修饰符，得到的结果也是标量，即 `0`（`false`）或 `1`（`true`）。例如：

```
2 > bool 1 # 结果为 1
```

瞬时向量和标量之间的布尔运算，这个运算符会应用到某个当前时刻的每个时序数据上，如果一个时序数据的样本值与这个标量比较的结果是 `false`，则这个时序数据被丢弃掉，如果是 `true`，则这个时序数据被保留在结果中。如果提供了 bool 修饰符，那么比较结果是 `0` 的时序数据被丢弃掉，而比较结果是 `1` 的时序数据被保留。例如：

```
http_requests_total > 100 # 结果为 true 或 false
http_requests_total > bool 100 # 结果为 1 或 0
```

瞬时向量与瞬时向量直接进行布尔运算时，同样遵循默认的匹配模式：依次找到与左边向量元素匹配（标签完全一致）的右边向量元素进行相应的操作，如果没找到匹配元素，或者计算结果为 false，则直接丢弃。如果匹配上了，则将左边向量的度量指标和标签的样本数据写入瞬时向量。如果提供了 bool 修饰符，那么比较结果是 `0` 的时序数据被丢弃掉，而比较结果是 `1` 的时序数据（只保留左边向量）被保留。

5.1.3、集合运算符

使用瞬时向量表达式能够获取到一个包含多个时间序列的集合，我们称为瞬时向量。 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` (并且)
- `or` (或者)
- `unless` (排除)

vector1 and vector2 会产生一个由 `vector1` 的元素组成的新的向量。该向量包含 vector1 中完全匹配 `vector2` 中的元素组成。

vector1 or vector2 会产生一个新的向量，该向量包含 `vector1` 中所有的样本数据，以及 `vector2` 中没有与 `vector1` 匹配到的样本数据。

vector1 unless vector2 会产生一个新的向量，新向量中的元素由 `vector1` 中没有与 `vector2` 匹配的元素组成。

5.2、匹配模式

向量与向量之间进行运算操作时会基于默认的匹配规则：**依次找到与左边向量元素匹配（标签完全一致）的右边向量元素进行运算，如果没找到匹配元素，则直接丢弃。**

接下来将介绍在 PromQL 中有两种典型的匹配模式：**一对一（one-to-one），多对一（many-to-one）或一对多（one-to-many）。**

5.2.1、一对一匹配

一对一匹配模式会从操作符两边表达式获取的瞬时向量依次比较并找到唯一匹配(标签完全一致)的样本值。默认情况下，使用表达式：

```
vector1 <operator> vector2
```

在操作符两边表达式标签不一致的情况下，可以使用 `on(label list)` 或者 `ignoring(label list)` 来修改便签的匹配行为。使用 `ignoreing` 可以在匹配时忽略某些便签。而 `on` 则用于将匹配行为限定在某些便签之内。

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

例如当存在样本：

```
method_code:http_errors:rate5m{method="get", code="500"}  24
method_code:http_errors:rate5m{method="get", code="404"}  30
method_code:http_errors:rate5m{method="put", code="501"}  3
method_code:http_errors:rate5m{method="post", code="500"} 6
method_code:http_errors:rate5m{method="post", code="404"} 21

method:http_requests:rate5m{method="get"}  600
method:http_requests:rate5m{method="del"}  34
method:http_requests:rate5m{method="post"} 120
```

使用 PromQL 表达式：

```
method_code:http_errors:rate5m{code="500"} / ignoring(code)
method:http_requests:rate5m
```

该表达式会返回在过去 5 分钟内，HTTP 请求状态码为 500 的在所有请求中的比例。如果没有使用 `ignoring(code)`，操作符两边表达式返回的瞬时向量中将找不到任何一个标签完全相同的匹配项。

因此结果如下：

```
{method="get"}  0.04            //  24 / 600
{method="post"} 0.05            //   6 / 120
```

同时由于 method 为 `put` 和 `del` 的样本找不到匹配项，因此不会出现在结果当中。

5.2.2、多对一和一对多

多对一和一对多两种匹配模式指的是“一”侧的每一个向量元素可以与"多"侧的多个元素匹配的情况。在这种情况下，必须使用 group 修饰符：`group_left` 或者 `group_right` 来确定哪一个向量具有更高的基数（充当“多”的角色）。

```
<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>
```

多对一和一对多两种模式一定是出现在操作符两侧表达式返回的向量标签不一致的情况。因此需要使用 ignoring 和 on 修饰符来排除或者限定匹配的标签列表。

例如，使用表达式：

```
method_code:http_errors:rate5m / ignoring(code) group_left
method:http_requests:rate5m
```

该表达式中，左向量 `method_code:http_errors:rate5m` 包含两个标签 `method` 和 `code`。而右向量 `method:http_requests:rate5m` 中只包含一个标签 `method`，因此匹配时需要使用 `ignoring` 限定匹配的标签为 `code`。 在限定匹配标签后，右向量中的元素可能匹配到多个左向量中的元素 因此该表达式的匹配模式为多对一，需要使用 group 修饰符 `group_left` 指定左向量具有更好的基数。

最终的运算结果如下：

```
{method="get", code="500"}  0.04            //  24 / 600
{method="get", code="404"}  0.05            //  30 / 600
{method="post", code="500"} 0.05            //   6 / 120
{method="post", code="404"} 0.175           //  21 / 120
```

> 提醒：`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。

5.3、聚合操作

Prometheus 还提供了下列内置的聚合操作符，这些操作符作用域瞬时向量。可以将瞬时表达式返回的样本数据进行聚合，形成一个具有较少样本值的新的时间序列。

- `sum` (求和)
- `min` (最小值)
- `max` (最大值)
- `avg` (平均值)
- `stddev` (标准差)
- `stdvar` (标准差异)
- `count` (计数)
- `count_values` (对 value 进行计数)
- `bottomk` (样本值最小的 k 个元素)
- `topk` (样本值最大的k个元素)
- `quantile` (分布统计)

这些操作符被用于聚合所有标签维度，或者通过 `without` 或者 `by` 子语句来保留不同的维度。

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

其中只有 `count_values`, `quantile`, `topk`, `bottomk` 支持参数(parameter)。

`without` 用于从计算结果中移除列举的标签，而保留其它标签。`by` 则正好相反，结果向量中只保留列出的标签，其余标签则移除。通过 without 和 by 可以按照样本的问题对数据进行聚合。

例如：

如果指标 `http_requests_total` 的时间序列的标签集为 `application`, `instance`, 和 `group`，我们可以通过以下方式计算所有 instance 中每个 application 和 group 的请求总量：

```
sum(http_requests_total) without (instance)
```

等价于

```
 sum(http_requests_total) by (application, group)
```

如果只需要计算整个应用的 HTTP 请求总量，可以直接使用表达式：

```
sum(http_requests_total)
```

`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。

例如要计算运行每个构建版本的二进制文件的数量：

```
count_values("version", build_version)
```

返回结果如下：

```
{count="641"}   1
{count="3226"}  2
{count="644"}   4
```

`topk` 和 `bottomk` 则用于对样本值进行排序，返回当前样本值前 n 位，或者后 n 位的时间序列。

获取 HTTP 请求数前 5 位的时序样本数据，可以使用表达式：

```
topk(5, http_requests_total)
```

`quantile` 用于计算当前样本数据值的分布情况 quantile(φ, express)，其中 $0 \leq \phi \leq 1$。

例如，当 φ 为 0.5 时，即表示找到当前样本数据中的中位数：

```
quantile(0.5, http_requests_total)
```

返回结果如下：

```
{}   656
```

5.4、二元运算符优先级

在 Prometheus 系统中，二元运算符优先级从高到低的顺序为：

1. `^`
2. `*`, `/`, `%`
3. `+`, `-`
4. `==`, `!=`, `<=`, `<`, `>=`, `>`
5. `and`, `unless`
6. `or`

具有相同优先级的运算符是满足结合律的（左结合）。例如，`2 * 3 % 2` 等价于 `(2 * 3) % 2`。运算符 `^` 例外，`^` 满足的是右结合，例如，`2 ^ 3 ^ 2` 等价于 `2 ^ (3 ^ 2)`。