1. 前言
2. 服务器离线
2.1. 字段详解
2.2. lables 在消息模版中的使用方法
2.3. annotations 在消息模版中的使用方法
3. CPU
3.1. CPU 5分钟平均负载高
3.2. CPU 5分钟平均 iowait 过高
3.3 CPU 5分钟的平均负载高于CPU核数的 2 倍
4. 内存
5. 磁盘
5.1. 完整告警规则:
5.2. 语法回顾
5.3. 表达式拆解
5.3.1. 分区可用率部分
5.3.2. 获取节点名部分
5.3.2. 向量匹配运算符部分

1. 前言

通过学习之前的文章，相信你已经对 Prometheus 这套监控体系有了基本的了解和认识。也想把所学到的知识运用到实践中，解决自己企业所遇到的问题。特别是想通过实践来更深入的理解和灵活运用之前学到的告警规则。

那么从这篇文章开始，将会进行企业告警规则的实战，会对企业中实际运用到的每条告警规则进行详细的解读。

并对这些告警规则进行拆解，学习每条规则中的包含的原子表达式。再通过对这些表达式进行组合，来更好的理解和运用 PromQL 中的语法，以便有能力编写出更适合企业需求的告警规则。

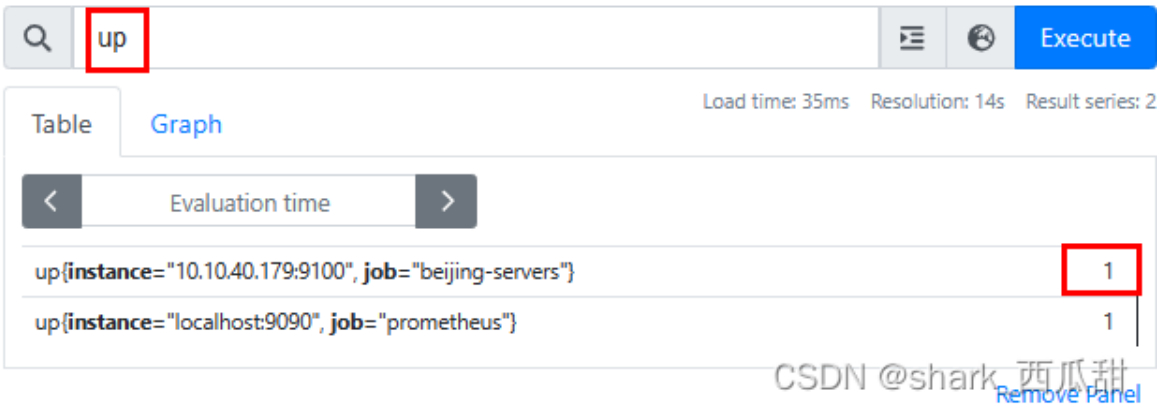
接下来我们就从第一个 exporter 所涉及到的告警规则开始。

2. 服务器离线

几乎在每个 exporter 中都有一个表示被监控对象是否在线的 **metrics**(指标) `[xxx_]up`。
比如 **node_exporter** 中表示服务器是否在线的指标是 `up`, **mysqld_exporter** 中表示mysql服务是否在线的指标是 `mysql_up`。
他们的值: `1` 表示正常在线，`0` 表示离线。

```yaml
groups:
- name: node-exporter
  interval: 5s
  rules:
    # 服务器节点不可用
  - alert: NodeDown
    expr: up == 0
    for: 5s
    labels:
      severity: critical
    annotations:
      summary: "{{ $labels.instance }}: down"
```

```
      description: "{{ $labels.instance }} has been down for more than 3m"
      value: "{{ $value }}"
```



CSDN @shark_西瓜甜

作为告警规则实战的第一篇的第一个规则，我将详细介绍每个配置字段及其值的含义，作为对规则书写语法的回顾，后面的的规则实战中我将会以分析讲解规则表达式为主。

2.1. 字段详解

以上配置中:

- `name` 给这组规则指定一个名字，不能和其他规则名称相同。
- `interval` 指定这组规则的检查时间间隔，就是每多长时间检查一次。
- `rules` 其值是一个列表，列表中的每个值都是一个告警规则。

- `alert` 此条警报的名称，不支持中文字符，这个会出现在警报通知的内容中。

- `expr` 此条警报的表达式，这个是最主要的。其值最终计算的结果必须是一个布尔值 `true` 或者 `false`。 `==` 是比较运算符，`==` 左侧是一个表达式，这里`up` 就是可以在 Prometheus 中已经存在的指标名称，当然也可以书写更复杂的计算表达式；`0` 是一个用于比较的值，这个值会和 `==` 左侧的表达式进行比较运算，整个表达式的计算结果为 `true`,则会触发此条警报，并立刻进入 **Pending** 状态，此状态不会发送警报给 Alertmanager。
- `for` 其值是一个时间单位，表示 `expr` 表达式的结果为 `true` 的状态持续这个时间后，此条警报进入 `Firing` 状态，并会发送警报给 Alertmanager。
- `lables` 其值是添加新的标签键值对，也可以覆盖已有标签的值。后续可以在消息模版中使用，具体使用方法见下方。
- `annotations` 这个值可以自定义的添加一些关于此警报的描述信息。价值对儿方式，键必须是英文，值可以含有模版和中文字符。
  - `summary` 自定义的有语义的键，表示简介，建议写简单扼要的内容。
  - `description` 自定义的有语义的键，表示描述信息，可以写更多详细的内容。
  - `value` 自定义的有语义的键，其值一般写此条警报中指标实际查询到的值，可以使用模版变量 `{{ $value }}` 获取。

2.2. lables 在消息模版中的使用方法

方式一：循环，其中 .Name 就是标签中的 键, .Value 就是标签中的值

```
  {{ range .Labels.SortedPairs }} - {{ .Name }}: {{ .Value | markdown | html }}
{{ end }}
```

方式二：单独获取指定的值

```
{{ range .Alerts.Firing }}
{{ .Labels.标签键名 }}
{{ end }}
```

示例：

```
{{ range .Alerts.Firing }}
{{ .Labels.severity }}
{{ end }}
```

2.3. annotations 在消息模版中的使用方法

方式一：循环，其中 .Name 就是每个键值对儿中的键, .Value 就是每个键值对儿中的值。

```
 {{- range .Annotations.SortedPairs }}
 - {{ .Name }}: {{ .Value | markdown | html }}
 {{ end }}{{ end }}{{ end }}
```

方式二：单独获取指定的值

```
{{ range .Alerts.Firing }}
{{ .Annotations.键名 }}
{{ end }}
```

示例：

```
{{ range .Alerts.Firing }}
{{ .Annotations.summary }}
{{ end }}
```

3. CPU

平时我们常用 `top` 命令查看 CPU 的使用状态。

下面用一条真实的数据样本来详细介绍CPU 的几种状态。

```
%Cpu(s):  8.7 us,  3.5 sy,  0.0 ni, 85.4 id,  2.5 wa,  0.0 hi,  0.0 si,  0.0 st
```

CSDN @shark 西瓜甜

)

- **8.7 us**
  这告诉我们，处理器花费 8.7%的时间运行用户空间进程。用户空间程序是指不属于内核的任何进程。shell、编译器、数据库、web服务器以及与桌面相关的程序都是用户空间进程。如果处理器没有空闲，那么CPU的大部分时间应该花在运行用户空间进程上是很正常的。
- **3.5 sy**
  这是CPU运行内核所花费的时间。所有进程和系统资源都由Linux内核处理。当用户空间进程需要来自系统的东西时，例如，当它需要分配内存、执行一些I/O或需要创建一个子进程时，内核运行

时。事实上，决定下一个进程运行的调度器本身就是内核的一部分。在内核中花费的时间应该尽可能少。在当前情况下，分配给不同进程的时间只有 3.5% 花在内核中。这个数字的峰值可能会高得多，尤其是当发生大量 **I/O** 时。

- **0.0 ni**
  ni 是 **nice** 的简称，意思是优先级，用户空间进程的优先级可以通过调整其精细度来调整，即调整优先级的值，这个值范围是 `-20` 到 `19`，每个进程的优先级初始值都是 `0`。ni 统计值显示了CPU运行经过优化的用户空间进程所花费的时间。在一个没有任何进程被优化的系统中，这个数字将是 `0`。
- **85.4 id**
  跳过其他一些统计数据，就在一瞬间，**id** 统计数据告诉我们，在上次采样期间，处理器空闲的时间刚好占整个采样期间时间的 85.4%。正常情况下，用户空间百分比-us、niced百分比-ni和空闲百分比-id的总和应该 `接近` 100%。当前情况就是这样。如果CPU在其他状态下（比如 **wa**）花费了更多的时间，那么可能出现了问题——请参阅下面的故障排除部分。
- **2.5 wa**
  与CPU的速度相比，读取或写入磁盘等输入和输出操作较慢。尽管与日常人类活动相比，这些操作发生得非常快，但与CPU的性能相比，它们仍然很慢。有时，处理器启动了读或写操作，然后不得不等待结果，但同时又没有其他事情可做。换句话说，是它在等待I/O操作完成时处于空闲状态。CPU在这种状态下花费的时间由 **wa** 统计数据显示。
- **0.0 hi** & **0.0 si**
  这两个统计数据显示了处理器为中断服务所花费的时间。**hi** 表示硬件中断，**si** 表示软件中断。硬件中断是从磁盘和网络接口等各种外围设备发送到CPU的物理中断。软件中断来自系统上运行的进程。硬件中断实际上会导致CPU停止它正在做的事情，并处理中断。软件中断不是发生在CPU级别，而是发生在内核级别。
  这个值过高-这可能是导致大量硬件中断的外围设备损坏的指示，或者是发出大量软件中断的进程的指示。
- **0..0 st**
  最后一个仅适用于虚拟机。当Linux作为虚拟机在系统管理程序上运行时，st（stolen 被盗的缩写）统计数据显示虚拟CPU等待系统管理程序为运行在不同虚拟机上的另一个虚拟CPU提供服务所花费的时间。由于在现实世界中，这些虚拟处理器共享相同的物理处理器，因此有时虚拟机想要运行，但系统管理程序却安排了另一个虚拟机。
  这个值过高-基本上这意味着运行系统管理程序的主机系统太忙。如果可能，请检查系统管理程序上运行的其他虚拟机，并且 / 或 将虚拟机迁移到另一台主机。

通常对于 CPU 的指标，我们主要关注常见的 4 个指标：

- **CPU使用率** CPU使用率是除了CPU空闲率 (**id** 统计的数据) 的值。 在繁忙的服务器或上，您可以预期CPU空闲的时间很短。然而，如果一个系统很少有空闲时间，那么它要么是a）过载（你需要一个更好配置的服务器），要么是b）出了问题。

- **us** 如果这个值很高，那么这可能意味着一个进程已经疯了，占用了所有的CPU时间。

```
top - 11:36:29 up  3:30,  4 users,  load average: 1.50, 0.77, 0.52
Tasks: 158 total,   3 running, 155 sleeping,   0 stopped,   0 zombie
%Cpu(s): 95.9 us,  2.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  2.1 si,  0.0 st
KiB Mem:   1018256 total,   792524 used,   225732 free,        0 buffers
KiB Swap:  2129916 total,    15632 used,  2114284 free.   185216 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6859 gary      20   0    9208   2540    492 R  97.4  0.2   0:47.95 matho-primes
 6520 gary      20   0  135684   2136    940 S   1.0  0.2   0:00.24 sshd
 6860 gary      20   0  123648   1608   1132 R   1.0  0.2   0:00.62 top
    1 root      20   0   52848   4856   2860 S   0.0  0.5   0:04.58 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:02.40 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/u2:0
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcuob/0
   10 root      20   0       0      0      0 S   0.0  0.0   0:12.51 rcu_sched
   11 root      20   0       0      0      0 R   0.0  0.0   0:32.25 rcuos/0
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.63 watchdog/0
   13 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 khelper
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   15 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
```

CSDN @shark-西瓜甜

- **sy** 高内核使用率，有时这是可以接受的。例如，一个执行大量控制台I/O的程序可能会导致内核使用率飙升。然而，如果它在很长一段时间内（一般我们看平均 5分钟）保持较高水平，那么这可能表明有些事情不对劲。这种峰值的一个可能原因可能是驱动程序/内核模块的问题。
- **wa** I/O等待时间长，这意味着系统上运行一些密集的I/O任务，不会占用太多CPU时间。如果这个数字对于除了短突发之外的任何事情都很高，那么这意味着任务执行的I/O效率非常低，或者数据被传输到速度非常慢的设备，或者硬盘存在需要很长时间来处理读写的潜在问题。

3.1. CPU 5分钟平均负载高

这个值是减去CPU空闲率的值。
这里可以根据需要，调整为 5分钟、10分钟或15分钟。

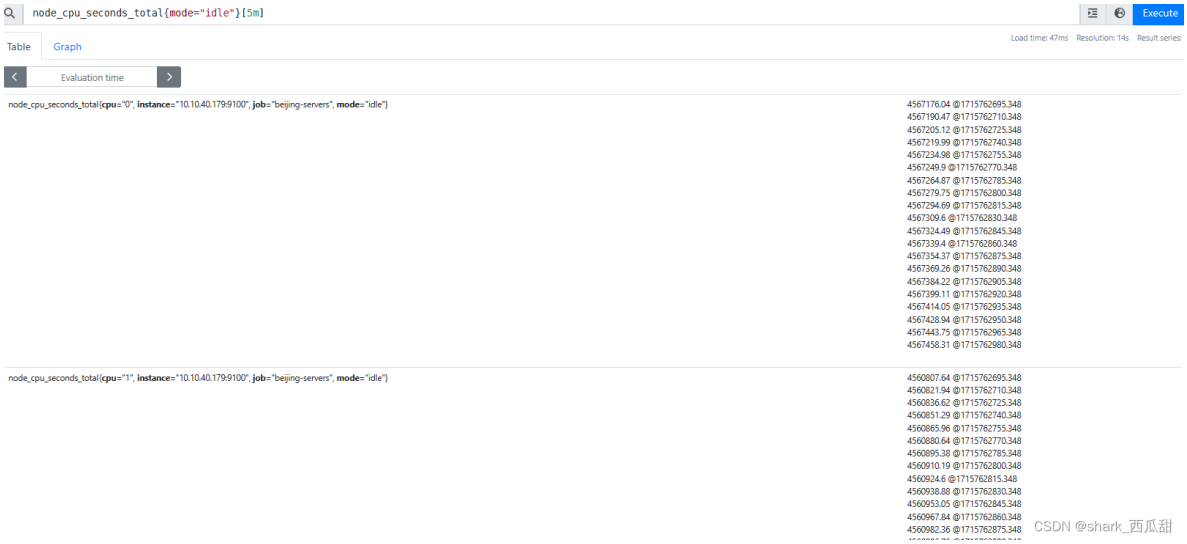
完整的告警规则如下：

```
      - alert: NodeCPUHigh
        # 节点 CPU  5 分钟的平均负载 过高，大于 75%
        expr: (1 - avg(rate(node_cpu_seconds_total{mode="idle"}[5m]))
by(instance)) * 100 > 75
        for: 5m
        labels:
          severity:
        annotations:
          summary: "{{$labels.instance}}: 高 CPU 使用率"
          description: "{{$labels.instance}}: CPU 使用率高于 75%"
          value: "{{ $value }}"
```

表达式拆解：
计算CPU使用率是拿 1 减去CPU 空闲率。

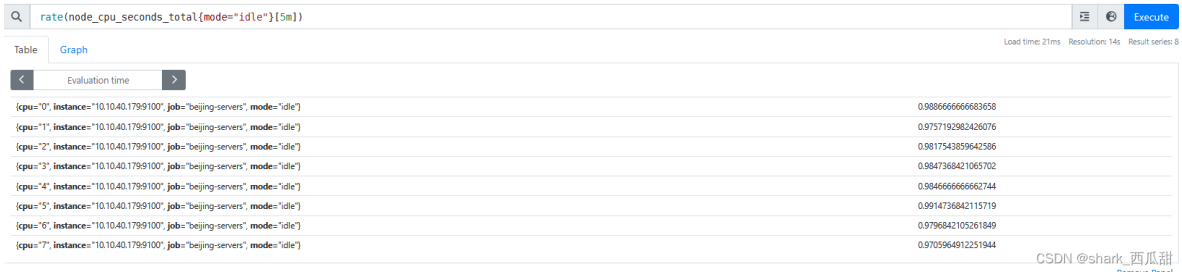
`node_cpu_seconds_total{mode="idle"}` 获取当前的CPU 空闲率。

由于我们这里计算的是 5m中内的，最后加上时间范围 `node_cpu_seconds_total{mode="idle"}[5m]`，可以获取最近5分钟内每个时间戳获取到的值。



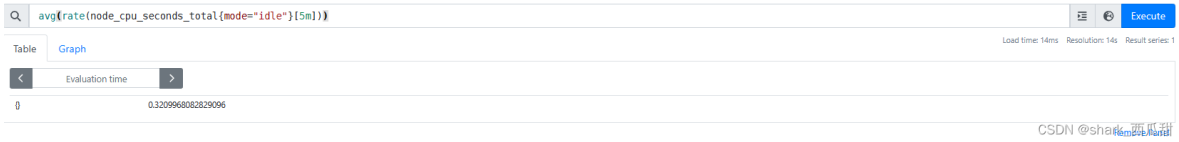
计算一段时间的增长率，我需要使用函数 `rate(v range-vector)` ,此函数是计算范围向量中时间序列的每秒平均增长率。

因此表达式是 `rate(node_cpu_seconds_total{mode="idle"}[5m])`



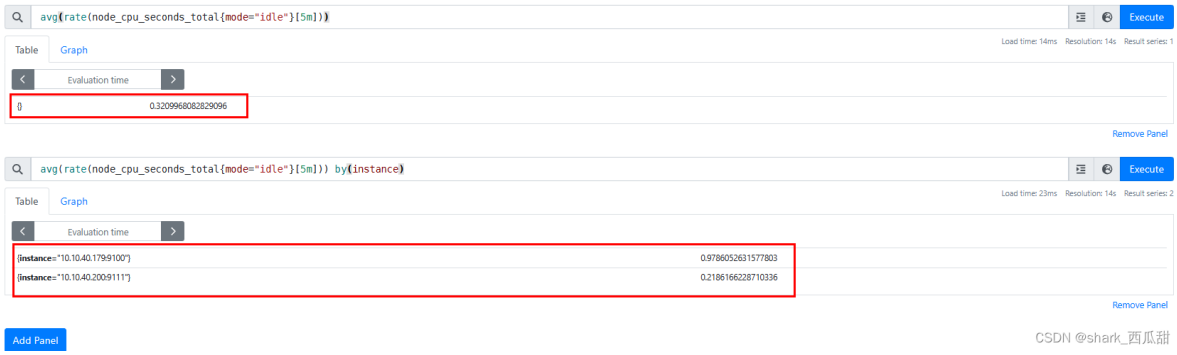
从上图中可以看到，我们得到的是CPU每个核的空闲率，实际上我们关系的是服务器CPU总体平均的空闲率。
因此我们需要使用函数 `avg` 统计几个CPU核的平均值。

表达式是 `avg(rate(node_cpu_seconds_total{mode="idle"}[5m]))`



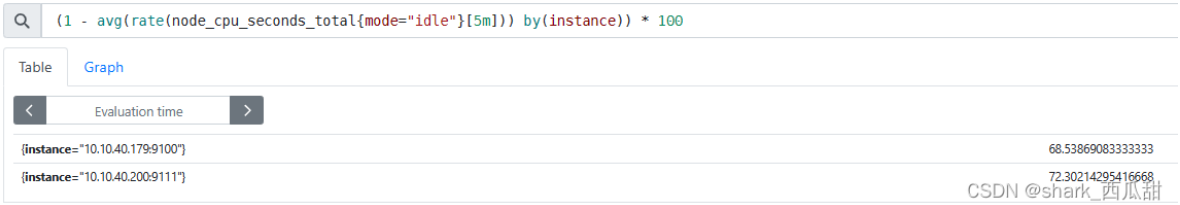
同时，我们需要获取的是每个被监控实例的CPU平均值，而不是获取此指标所对应的总体数据的平均值(比如此指标对应了多个被监控实例)。那我们就可以使用 `by` 指定标签 `instance`

表达式 `avg by(instance)(rate(node_cpu_seconds_total{mode="idle"}[5m]))` 或者 `avg(rate(node_cpu_seconds_total{mode="idle"}[5m])) by(instance)`



好了，至此获取到的是每个服务器 CPU 的 5分钟平均空闲率。接下来就简单了，只要使用初中水平甚至小学水平的数学知识就可以计算出我们最终的结果了。

`(1 - avg(rate(node_cpu_seconds_total{mode="idle"}[5m])) by(instance)) * 100`



通常我们认为 CPU 平均使用率超过 75% 就说明CPU 压力大，或者程序出现了问题。

最终表达式: `(1 - avg(rate(node_cpu_seconds_total{mode="idle"}[5m])) by(instance)) * 100 > 75`

3.2. CPU 5分钟平均 iowait 过高

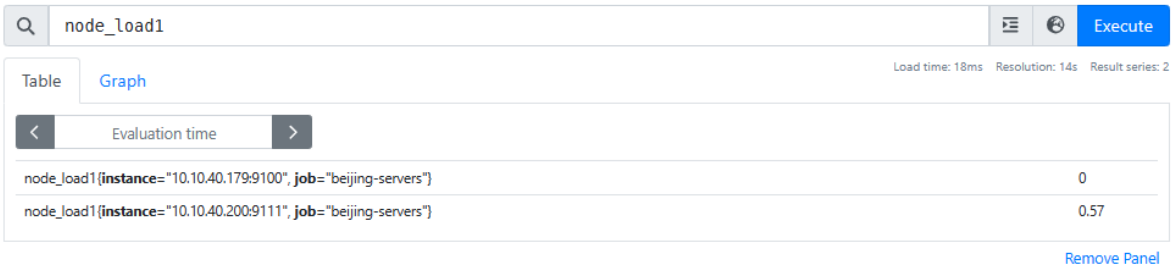
`node_exporter` 中直接提供了 `iowait` 的数据。所以可以使用指标名称加标签即可得到结果，再使用和上个 CPU 使用率的计算方式一样，以被监控实例为组分别计算 5 分钟的平均值（当然也可以使用其他的时间范围，如 10分钟，15分钟）。通常我们希望这个数值在 50以内是可以接受的。

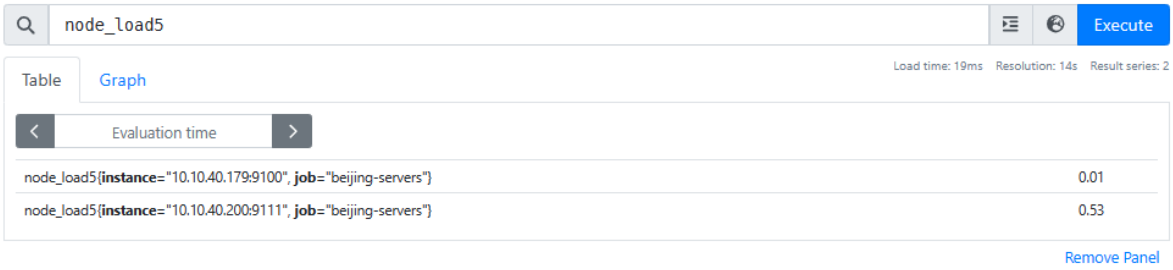
完整的告警规则如下：

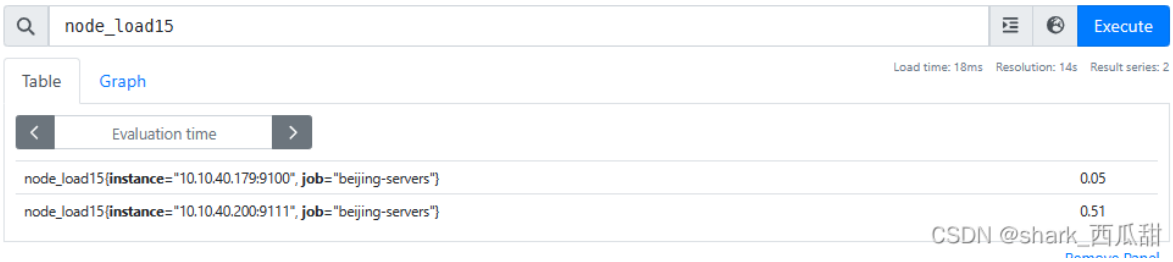
```
      - alert: NodeCPUIowaitHigh
        # 节点 5 分钟内的CPU iowait 过高，大于  50
        expr: avg(rate(node_cpu_seconds_total{mode="iowait"}[5m])) by(instance)
* 100 > 50
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "{{$labels.instance}}: 高 CPU iowait 使用率"
          description: "{{$labels.instance}}:CPU iowait使用率高于 50%"
          value: "{{ $value }}"
```

3.3 CPU 5分钟的平均负载高于CPU核数的 2 倍

node_exporter 直接提供了 1 分钟，5分钟，15分钟的CPU平均负载数据，分别对应的指标名称是 `node_load1`、`node_load5`、`node_load15`。



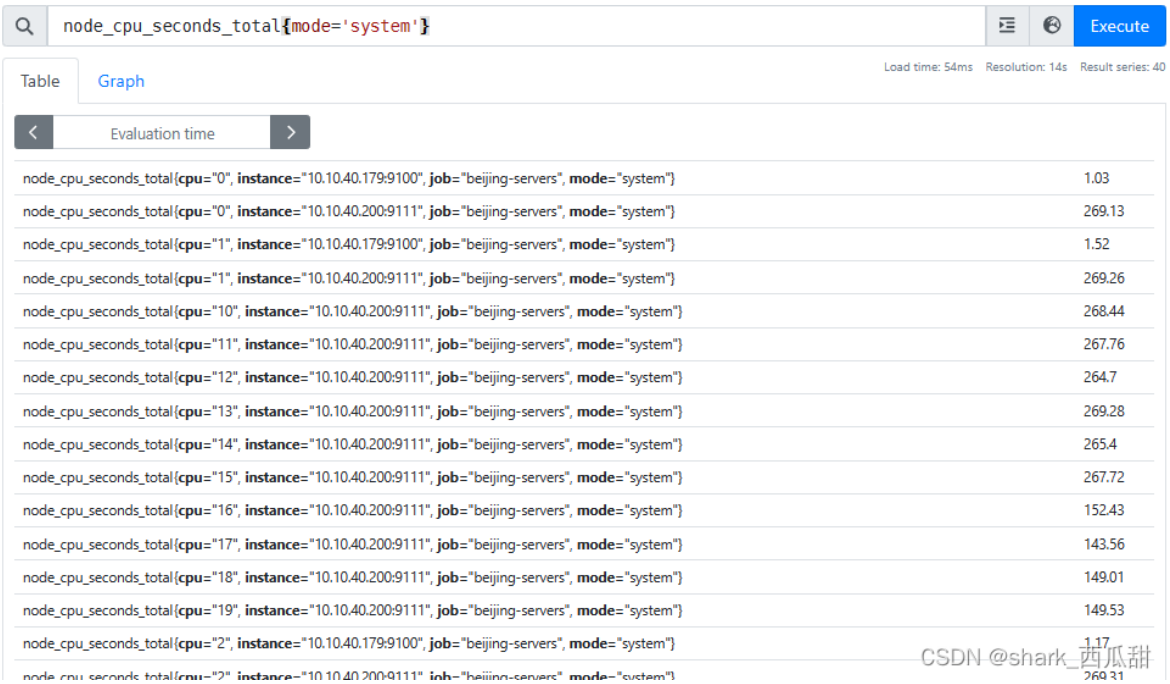




大部分时候我们会比较关心 5分钟的平均负载情况，特别是在多核CPU的服务器上，这个值大于服务器CPU核心数的2倍的时候，需要检查服务器程序运行情况。

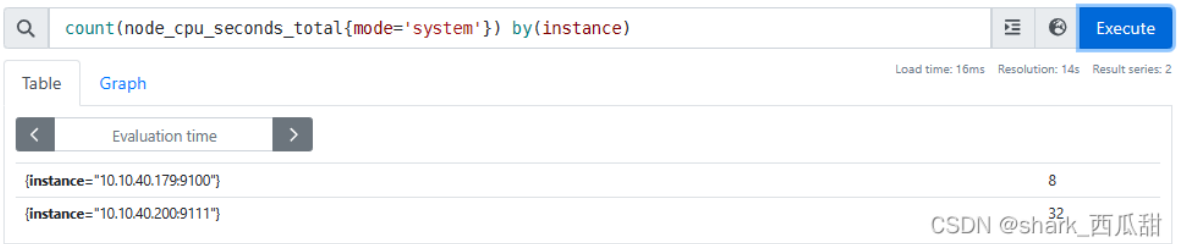
> CPU 是可压缩资源，允许偶尔短时间内的负载高，但是如果高出总核心数的2倍后，就不正常了。

`node_cpu_seconds_total{mode='system'}` 获取到 **system** 模式的所有指标数据，当然也可以使用其他的模式，这个只是为了识别筛选，保障结果中每个CPU不重复被统计。



之后我们使用 `count` 函数和 `by` 子句配合，获取到以每个服务器进行分组统计，获取每台服务器的CPU核数。

表达式：`count(node_cpu_seconds_total{mode='system'}) by(instance)`



最后我们使用 5分钟负载数据和每台服务器CPU核心数的 2 倍进行比较。

表达式：`node_load5 > count(node_cpu_seconds_total{mode='system'}) by(instance) * 2`

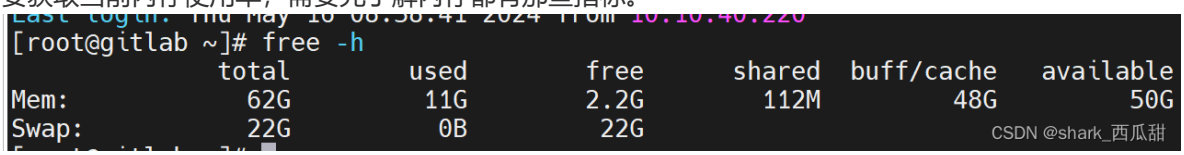
完整的告警规则如下：

```
      - alert: NodeLoad5High
        expr: (node_load5) > (count(node_cpu_seconds_total{mode='system'}) by
(instance) * 2)
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "{{$labels.instance}}: 5 m 负载高"
          description: "{{$labels.instance}}: 当前 5m 负载是CPU核数的2倍"
          value: "{{ $value }}"
```

4. 内存

内存主要关注每台服务器的内存使用率是否太高，比如，内存使用率不能高于 90%。

要获取当前内存使用率，需要先了解内存都有那些指标。



- `total` 服务器的总内存容量。
- `used` 服务器中程序实际使用的内存。
- `free` 服务器空闲的内存，没有任何程序和操作系统内部使用。
- `shared` 服务器上程序之间共享的内存。
- `buff/cache` 缓冲缓存区，操作系统主动缓存的数据使用的内存，随时可以给到需要内存的程序。

  > 从磁盘读取数据比从内存访问数据慢得多。Linux将磁盘中的部分热数据缓存到内存中。事实上，Linux使用所有空闲的RAM作为缓冲区缓存，以使读取数据尽可能高效。

> 当系统中出现一个新的进程，或者原有的进程需要更多的内存，系统会先从 `available` 中的内存分配，如果一个程序需要比 `available` 内存更多的内存，会发生什么？缓冲区缓存(buff/cache)将收缩以适应增加的内存需求。缓冲缓存的工作原理就像你最高效的同事：当事情不忙的时候，他会四处奔波，让事情更顺利。当一项重要的任务出现时，他会放弃不太重要的家务。

- `available` 可用内存，操作系统会优先从这里分配内存给程序。

  实际可用内存 = free(2.2G) + buff/cache(48G) = 50.2G

但一般我们会认为 `available` 的容量就是可用内存，因此当前服务器内存可用内存率是：

内存可用率 = 可用内存 / 总内存 * 100

内存可用率表达式使用：`node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes) * 100`

那么内存的使用率的表达式是：`1 -  node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes) * 100`

完整的告警规则如下：

```
      - alert: NodeMemoryUsageHigh
        # 节点内存使用率太高，大于 90%
        # node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes 得出当前可用率
        # 1 - 当前可用率 得出已经使用率
        # (1 - 当前可用率) * 100 得出当前已使用百分比
        expr: (1 - node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes) * 100 > 90
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "{{$labels.instance}}: High memory usage"
          description: "{{$labels.instance}}: Memory usage is above 90%"
          value: "{{ $value }}"
```

5. 磁盘

根据我之前的 [2-云原生监控体系-使用node-exporter监控Linux服务器](#) 文章中介绍的，使用启动参数 `--collector.filesystem.ignored-mount-points=^/(boot|sys|proc|dev|run)($|/)` 排除一下不必要统计的系统自带的分区。之后我们获取的数据都是需要监控的磁盘分区了，就不必写正则表达式去匹配这些分区了。

5.1. 完整告警规则：

```
      - alert: NodeDiskRootLow
        expr: ((node_filesystem_avail_bytes / node_filesystem_size_bytes)  * 100 < 25 and ON (instance, device, mountpoint) node_filesystem_readonly == 0) * on(instance) group_left (nodename) node_uname_info{nodename=~".+"}
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "{{$labels.instance}}: Low disk(the / partition) space"
          description: "{{$labels.instance}}: 根分区可用率低于 25%，当前值:{{ $value }}"
```

5.2. 语法回顾

要理解上面的表达式，需要先回归一下文章中提到的如下语法知识:

- `and`
  and

`vector1 and vector2` 返回的结果是由 `vector1` 的元素组成的向量，但是结果集的每条数据，都必须和其中 `vector2` 中的数据具有完全匹配的标签集（即标签名和其对应的标签值）。`vector1` 中不匹配的数据将删除。指标名称和值是从左侧向量中继承的。

- `on` 限定匹配标签集的范围。
- `group_left` 是 向量匹配运算中的修饰符，对于两个向量，指定左侧的向量是 "多" 的一方。

5.3. 表达式拆解

表达式 `((node_filesystem_avail_bytes / node_filesystem_size_bytes)  * 100 < 25 and ON (instance, device, mountpoint) node_filesystem_readonly == 0) * on(instance) group_left (nodename) node_uname_info{nodename=~".+"}`

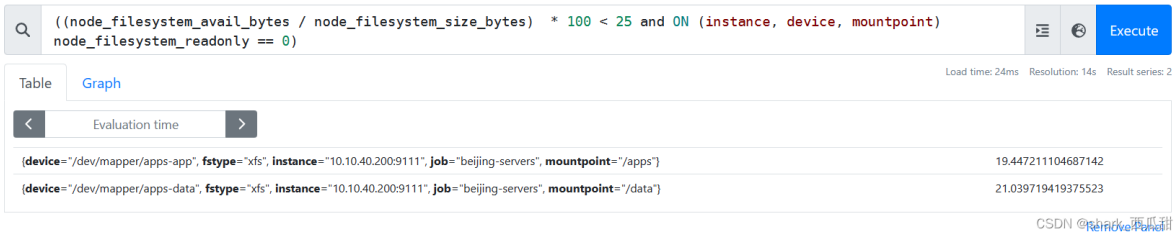
首先以最后一个 `* on(instance) group_left (nodename)` 为分界，可以分为三部分。

- 分区可用率部分
- 获取节点名称部分
- 向量运算符部分

5.3.1. 分区可用率部分

`((node_filesystem_avail_bytes / node_filesystem_size_bytes)  * 100 < 25 and ON (instance, device, mountpoint) node_filesystem_readonly == 0)`

这个表达式主要是返回分区剩余可用率小于 `25%` 的，并且挂载属性不是只读的。



通过 `and ON (instance, device, mountpoint)` 又可以拆分为如下两部分:

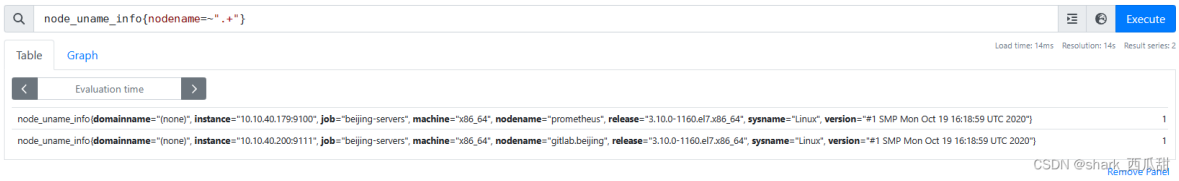
- `(node_filesystem_avail_bytes / node_filesystem_size_bytes)  * 100 < 25`
  这部分获取的是分区可用率低于 `25%` 的数据。
- `node_filesystem_readonly == 0`
  这部分获取的是分区的挂载的文件系统是读写，而非只读的数据。`1` 表示 只读，`0` 表示非只读（即读写）。

> 因为在 `(node_filesystem_avail_bytes / node_filesystem_size_bytes)` 获取的结果中，有可能会有光盘的设备或者 ISO 镜像文件被以只读的属性挂载到操作系统中，所以需要使用 `and` 逻辑运算符，并按照标签 `instance, device, mountpoint` 将这两部分进行逻辑与运算。

5.3.2. 获取节点名部分

`node_uname_info{nodename=~".+"}`

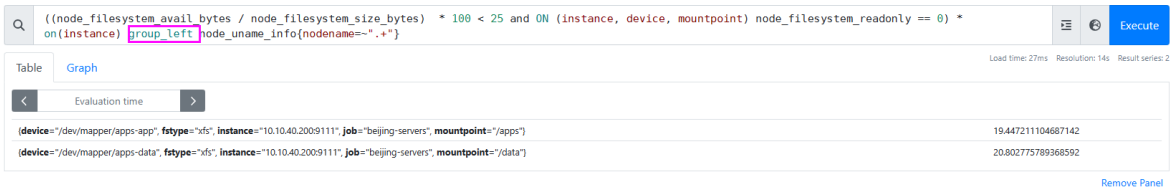
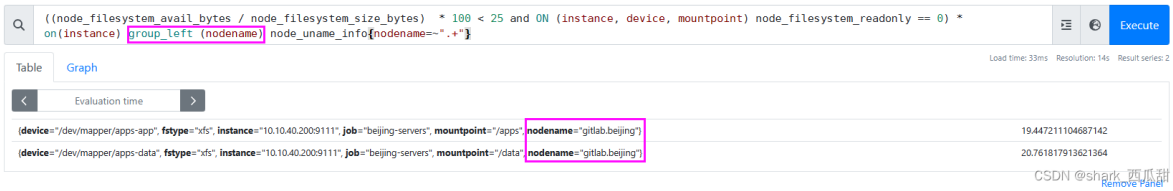
这部分主要是为了获取 `nodename` 这个标签和值，以便和分区可用率低的结果集匹配上具体的主机名。
这部分主要使用了正则表达式 `.+` 去匹配节点名称。



5.3.2. 向量匹配运算符部分

`* on(instance) group_left (nodename)`

由于 **分区可用率部分** 获取到的结果中会有同一个服务器中多个分区设备，并且不同的服务器有可能出现相同名称的分区设备，并且返回的结果中没有区分那台服务器的标签。

因此需要使用 `* on(instance) group_left (nodename)` 进行向量匹配运算，同时使用 `on(instance)` 限定只使用标签 `instance` 作为匹配条件。

进一步还使用 `group_left (nodename)` 在返回的结果中添加标签 `nodename` 以便明确的看出每条数据属于哪个节点。