

20 | 应用监控：如何使用日志来监控应用？

2023-02-22 秦晓辉 来自北京

《运维监控系统实战笔记》

课程介绍 >



讲述：秦晓辉

时长 15:02 大小 13.74M



你好，我是秦晓辉。

上一讲我们介绍了应用埋点监控，对于自研的软件，在一开始就建立可观测能力是非常好的选择，但是很多软件可能无法修改源代码，比如一些外采的软件，那就只能用一些外挂式的手段，比如在请求链路上插入一些代理逻辑，或者读取分析应用日志。

典型的代理方式是 Nginx，如果是 HTTP 服务，从 Nginx 的 Access 日志中可以获取很多信息，比如访问的是哪个接口，用的什么 HTTP 方法，返回的状态码是什么，耗时多久等等。这些信息对应用的监控很有帮助。

除此之外，我们也可以使用 eBPF 技术为网络包增加一些过滤分析逻辑，不过 eBPF 要求的内核版本较高。而通过日志对应用做监控，显然是相对直观和廉价的方式，这一讲我们就来看看怎么从日志中提取指标。

提取指标的典型做法

根据提取规则运行的位置可以分为两类做法，一个是在中心端，一个是在日志端。

中心端就是把要处理的所有机器的日志都统一传到中心，比如通过 **Kafka** 传输，最终落到 **Elasticsearch**，指标提取规则可以作为流计算任务插到 **Kafka** 通道上，性能和实时性都相对更好。或者直接写个定时任务，调用 **Elasticsearch** 的接口查询日志，同时给出聚合计算函数，让 **Elasticsearch** 返回指标数据，然后写入时序库，实时性会差一些，但也基本够用。

日志端处理是指提取规则直接运行在产生日志的机器上，流式读取日志，匹配正则表达式。对于命中的日志，提取其中的数字部分作为指标上报，或者不提取任何数字，只统计一下命中的日志行数有时也很有价值，比如统计一下 **Error** 或 **Exception** 关键字出现的次数，我们就知道系统是不是报错了。

中心端处理的方式，我没有找到开源解决方案，如果你知道有好用的开源方案可以在评论区留言分享。日志端处理的方式，倒是有很多开源方案，比较有名的是 [mtail](#) 和 [grok_exporter](#)，**mtail** 发布时间更久，3400 star，**grok_exporter** 发布时间晚一些，700 star，不过它们原理上是类似的，所以我们重点介绍 **mtail** 的用法。

快速上手 mtail

你应该用过 **Linux** 下的 **tail** 命令吧？**mtail**、**grok_exporter** 等工具就像是对日志文件执行 **tail -f**，然后每收到一条日志，就去匹配预定义的正则表达式，如果匹配成功，就执行某些动作，否则跳过等待下一条日志。

下面我们安装一下 **mtail**，统计一下 **/var/log/messages** 中 **Out of memory** 关键字出现的次数，作为一个重要的监控指标上报。

mtail 最新版本是 [3.0.0-rc50](#)，虽然是 rc 版本，不过不用怕，**mtail** 一直在发 rc 版就是不发正式版，我在生产环境用过没遇到什么问题，我们就拿这个版本举例，下载和你的 **OS** 匹配的发布包，解压缩，可以看到 **mtail** 的二进制。

我想统计 **/var/log/messages** 中 **Out of memory** 关键字出现的次数，那我得通过某种机制告诉 **mtail** 正则表达式是什么，提取规则是什么，这个规则文件我们叫做 **program**，一般命名为 **xyz.mtail**，我给出了一个样例，你可以看一下。

```

1 # 在 mtail 二进制同级目录下创建 progs 目录，下面放置 syslogs 子目录
2 # syslogs 子目录用于放置系统日志文件对应的提取规则
3 mkdir -p progs/syslogs
4
5 # 用于统计 Out of memory 关键字的 mtail 规则文件内容如下(我命名为
6 # syslogs.mtail):
7 counter oom_total
8 /Out of memory/ {
9     oom_total++
10 }

```

文件内容看起来很简单，只有 4 行，第一行是声明了一个变量，类型是 **counter**，变量名是 **oom_total**，第二行是在 `//` 之间定义了一个正则表达式，用来匹配 **Out of memory**，如果匹配成功，就执行大括号里的内容，对 **oom_total** 变量加 1。

接下来，我们把 **mtail** 运行起来，看看效果如何。

启动命令：

```
1 ./mtail -progs ./progs/syslogs/ -logs /var/log/messages
```

通过 **-progs** 参数指定 **mtail** 文件所在目录，当然，指定具体的文件也可以，通过 **-logs** 参数指定要提取的日志文件是哪个，支持 **glob** 匹配，也支持传入多次 **-logs** 参数。**mtail** 启动之后会默认监听在 **3903** 端口，请求 **3903** 的 **/metrics** 接口就能拿到 **Prometheus** 协议的监控数据。

```

1 [root@fc-demo-02 qinxiaohui]# ss -tlnp|grep mtail
2 LISTEN      0      128      [::]:3903      [::]:*
3 [root@fc-demo-02 qinxiaohui]# curl -s localhost:3903/metrics | grep oom_total
4 # HELP oom_total defined at syslogs.mtail:1:9-17
5 # TYPE oom_total counter
6 oom_total{prog="syslogs.mtail"} 0

```

上面的例子里，我使用 **grep** 命令做了过滤，只展示了 **oom_total** 相关的内容，实际上 **mtail** 会输出很多指标，你可以自己测试一下。有了这个 **/metrics** 接口，怎么和监控系统对接就很明显

了，直接由抓取器来这个地址抓取数据就可以了。下面我们继续讲解 mtail 本身的使用。

例子里 mtail 自动加了一个 prog 标签，把 mtail 文件名作为标签加上了，对于一些 access.log 类型的日志，经常用于统计接口的吞吐、延迟等，需要把接口路径、method、statuscode 等作为标签，应该如何配置呢？这里我以 Nginx 的 access 日志作为样例来演示，你可以看一下我的 Nginx 的 logformat。

 复制代码

```
1 log_format main '$remote_addr - $remote_user [$time_local] "$request" '
2 '$status $body_bytes_sent "$http_referer" '
3 '"$http_user_agent" "$http_x_forwarded_for"';
```

对应的几条样例日志如下：

 复制代码

```
1 # tail -n 5 /var/log/nginx/access.log
2 119.45.249.92 - - [17/Dec/2022:22:35:39 +0800] "GET / HTTP/1.1" 200 14849 "-" "
3 119.45.249.92 - - [17/Dec/2022:22:35:40 +0800] "GET / HTTP/1.1" 200 14849 "-" "
4 119.45.249.92 - - [17/Dec/2022:22:35:40 +0800] "GET / HTTP/1.1" 200 14849 "-" "
5 119.45.249.92 - - [17/Dec/2022:22:35:41 +0800] "GET / HTTP/1.1" 200 14849 "-" "
6 119.45.249.92 - - [17/Dec/2022:22:35:41 +0800] "GET / HTTP/1.1" 200 14849 "-" "
```

这个 logformat 着实简单，连响应延迟都没有打印，这是 Nginx 默认的 logformat，我们也先维持现状，统计一下请求数量以及响应体的大小，下面是具体的 mtail 文件内容。

 复制代码

```
1 counter request_total by method, url, code
2 counter body_bytes_sent_total by method, url, code
3
4 /"(?P<method>\S+) (?P<url>\S+) HTTP\/1.1" (?P<code>\d+) (?P<body_bytes_sent>\d+
5 request_total[$method][$url][$code]++
6 body_bytes_sent_total[$method][$url][$code] += $body_bytes_sent
7 }
```

这个正则看起来就复杂多了，比如获取 statuscode 的地方写的正则 (?P<code>\d+)，这个叫做**命名捕获**，核心的正则就是 \d+，但是后面想用这个内容，就给它设置了一个变量叫 code，而 method、url、body_bytes_sent 都是同样的道理。

匹配了正则之后，做了两个动作，`request_total` 变量加一，相当于在统计请求次数，`body_bytes_sent_total` 变量加上日志这行提取的 `$body_bytes_sent` 变量的值，是在统计响应总大小。

这里 `request_total` 和 `body_bytes_sent_total` 这两个指标都是带有标签的，且都是 3 个标签：`method`、`url`、`code`，声明之后就可以使用，通过命名捕获的方式给的变量名也可以在后面使用，非常灵活。下面是测试输出。

 复制代码

```
1 # 通过下面的命令加载 nginx 的 mtail, 指定 nginx 的 access.log
2 ./mtail -progs ./progs/nginx/ -logs /var/log/nginx/access.log
3
4 # 下面请求一下 /metrics 接口，看看是否采集成功
5 [root@fc-demo-02 qinxiaohui]# curl -s localhost:3903/metrics | grep -P 'request
6 # HELP body_bytes_sent_total defined at nginx.mtail:2:9-29
7 # TYPE body_bytes_sent_total counter
8 body_bytes_sent_total{code="200",method="GET",prog="nginx.mtail",url="/"} 1.143
9 # HELP request_total defined at nginx.mtail:1:9-21
10 # TYPE request_total counter
11 request_total{code="200",method="GET",prog="nginx.mtail",url="/"} 77
```

看起来一切正常，上面这些标签都是从日志中直接提取的，如果我想附加一些静态标签应该怎么做呢？比如把机房信息作为标签附到时序数据上，你可以看一下样例。

 复制代码

```
1 hidden text zone
2 zone = "beijing"
3
4 counter request_total by method, url, code, zone
5 counter body_bytes_sent_total by method, url, code, zone
6
7 /"(?P<method>\S+) (?P<url>\S+) HTTP\/1.1" (?P<code>\d+) (?P<body_bytes_sent>\d+
8   request_total[$method][$url][$code][zone]++
9   body_bytes_sent_total[$method][$url][$code][zone] += $body_bytes_sent
10 }
```

这里加了一个全局的 `zone="beijing"` 的标签，写法一目了然，不过多解释。唯一需要注意的是在引用 `zone` 变量的时候，前面不要加 `$` 符号。这里千万别写顺手了，看到其他变量都加 `$` 就把 `zone` 也加上了，加上就识别不了了。

上面两个例子演示的都是 Counter 类型的变量，其实 mtail 还支持 Gauge 和 Histogram 类型，并且在仓库的 [examples](#) 目录下提供了很多样例，可以直接拿来使用，注意 Histogram 类型在定义的时候要给出 Bucket 的分布范围，要不然 mtail 不知道如何放置统计数据，这个应该也容易理解，如果有疑问可以返回前面 [第 2 讲](#)，再回顾一下 Histogram 类型的讲解。

下面我们来看一个生产级的例子，用于分析 lighttpd 的访问日志，我们从中可以学到更多生产级的写法。

 复制代码

```
1 # Copyright 2010 Google Inc. All Rights Reserved.
2 # This file is available under the Apache license.
3
4 # mtail module for a lighttpd server
5
6 counter request by status
7 counter time_taken by status
8 counter bytes_out by subtotal, status
9 counter bytes_in by status
10 counter requests by proxy_cache
11
12
13
14 const ACCESSLOG_RE // +
15     /(?P<proxied_for>\S+) (?P<request_ip>\S+) (?P<authuser>\S+)/ +
16     / \[(?P<access_time>[^\]]+)\] "(?P<http_method>\S+) (?P<url>.+?) / +
17     /(?P<protocol>\S+)" (?P<status>\d+) (?P<bytes_body>\d+) (?P<bytes_in>\d+)/
18     / (?P<bytes_out>\d+) (?P<time_taken>\d+) "(?P<referer>[^\"]+)" / +
19     /"(?P<agent>[^\"]+)" /
20
21
22
23 # /var/log/lighttpd/access.log
24 getfilename() =~ /lighttpd.access.log/ {
25     // + ACCESSLOG_RE {
26         # Parse an accesslog entry.
27         $url == "/healthz" {
28             # nothing
29         }
30         otherwise {
31             strptime($access_time, "02/Jan/2006:15:04:05 -0700")
32
33             request[$status]++
34             time_taken[$status] += $time_taken
35             bytes_out["resp_body", $status] += $bytes_body
36             bytes_out["resp_header", $status] += $bytes_out - $bytes_body
37             bytes_in[$status] += $bytes_in
38
39             $proxied_for != "-" {
```



```
40     requests[$request_ip]++
41   }
42 }
43 }
44 }
```

这个 `program` 一开始，定义了很多 `counter` 类型的变量，这里没有什么新知识，略过。然后定义了一个常量 `ACCESSLOG_RE`，这个正则很复杂，对于这类复杂的正则，可以拆成很多个小的部分，相互之间用 `+` 连接，这种做法既容易阅读，又容易为每个片段增加注释，便于后期维护，后面介绍的 `grok_exporter` 则更进一步，把这些正则片段直接做成 `pattern` 单独维护了。

继续往下看，`getfilename()` 是个内置函数，获取日志文件的路径，对这个内容做了一个正则判断，如果匹配才去走核心逻辑，这里是不是有些多此一举了呢？我猜测，这个写法的初衷是觉得这段内容可能会和其他的提取规则混在一起，同时提取多个日志文件时，为了避免这段逻辑跑在一些无关的日志上，就加了这么一句判断，考虑得很周全。不过，如果我们在使用的时候，可以保证这段逻辑只用于处理 `lighttpd` 的访问日志，这个判断就是可以去掉的。

`getfilename()` 的判断通过之后，就开始校验主正则了。主正则匹配就开始判断请求的 `url` 是不是 `/healthz`，如果是就什么都不干（空逻辑），因为这个是健康检查的接口，没必要提取指标。否则进行主逻辑处理，“否则”的关键词是 `otherwise`，相当于 `else`。主逻辑部分你应该一眼就能明白是什么意思，核心点是这个 `strptime` 函数，它其实是在告诉 `mtail` 用什么时间格式来转换时间戳，第二个参数是 Go 写法的 `format pattern`，这些其实都好理解，比较容易掉坑里的是**时区问题**。

如果日志里的时间戳没有打印时区信息，`mtail` 在处理的时候会把它统一当做 `UTC` 时间来对待，这在其他时区的场景显然是错误的，这个时候我们就要手工指定时区，比如通过 `-override_timezone=Asia/Shanghai` 启动参数可以让 `mtail` 使用东八区。

当然，如果我们压根就不在 `/metrics` 接口中暴露时间戳信息，那抓取器抓取数据的时候只能使用抓取的时间，时区这个参数有没有都无所谓了，但如果我们在 `/metrics` 接口中返回时间戳信息，就一定要在启动参数中控制时区，是否在 `/metrics` 接口中返回时间戳信息，也是通过一个启动参数来控制的，`emit_metric_timestamp` 设置为 `true` 的时候才会返回时间戳。

最后说一下 **mtail** 的部署，如果一个机器上有 5 个应用程序都要用 **mtail** 来提取指标，各个应用的日志格式又不一样，建议启动 5 个 **mtail** 进程分别来处理。虽然管理起来麻烦，但是性能好，相互之间没有影响。

如果把所有提取规则都放到一个目录下，然后通过多次 **-logs** 参数的方式同时指定这多个应用的日志路径，一个 **mtail** 进程也能处理，但是对于每一行日志，**mtail** 要把所有提取规则都跑一遍，十分浪费性能，而且正则提取，速度本来就不快。另外有些指标可能是所有应用都可以复用的，如果放在一起处理，还容易相互干扰，导致统计数据不准。从这两点来看，尽量还是要拆开分别处理，虽然管理起来麻烦一些，但也是值得的。

在容器场景中就没有这个问题，容器场景直接使用 **sidecar** 部署就好了，每个 **Pod** 必然只有一个应用，伴生的 **mtail** 就专注去处理这个应用的日志就好了。

延伸讨论：物理机大概率会有混部 5 个甚至 50 个服务的场景，容器又必然是一个服务一个 **Pod**，那虚拟机呢？做成大规格的好，还是小规格的好呢？是有混部好还是没有混部好呢？欢迎留言分享你的见解。

mtail 基本用法我们就介绍这么多，不知道你有没有感受到，得写这么多正则，太麻烦了。有没有一些工具可以复用这些正则表达式呢？毕竟有很多相同的正则需求，没必要重复造轮子。的确有，除了刚才介绍的 **mtail** 自带的 [examples](#) 之外，**grok_exporter** 把每个正则都拆散了，复用性更好，下面我们看一下 **grok_exporter** 是如何使用的。

快速上手 **grok_exporter**

grok_exporter 的核心逻辑和 **mtail** 一样，就是通过正则从日志中提取指标，我们之前已经介绍过 **mtail** 的核心逻辑了，所以关于 **grok_exporter** 的介绍会相对简明一些。


grok_exporter 显然是用到了 **Grok**，**Grok** 在 **logstash** 中被重度使用，内置了 100 多个预定义的正则（叫做 **pattern**），在 **grok_exporter** 的代码仓库里直接作为 **submodule** 的方式引用了 [logstash-patterns-core](#)，预定义的正则放在了 [patterns](#) 目录下，你可以点击查阅。

从 **grok_exporter** 的 [releases](#) 页面下载发布包，解压缩，直接运行就可以。

 复制代码

```
1 ./grok_exporter -config ./example/config.yml
```


grok_exporter 默认监听在 9144 端口，我们看下访问测试效果。

 复制代码

```
1 [flashcat@fc-demo-02 ~]$ curl -s 10.100.0.7:9144/metrics | head -n 6
2 # HELP exim_rejected_rcpt_total Total number of rejected recipients, partitione
3 # TYPE exim_rejected_rcpt_total counter
4 exim_rejected_rcpt_total{error_message="Sender verify failed",logfile="exim-rej
5 exim_rejected_rcpt_total{error_message="Unrouteable address",logfile="exim-reje
6 exim_rejected_rcpt_total{error_message="relay not permitted",logfile="exim-reje
7 # HELP go_gc_duration_seconds A summary of the pause duration of garbage collec
```

通过代码，我们可以看到 grok_exporter 可以正常拿到监控数据了。下面我们搞一下测试数据，把它放到 example 目录下，保存为 login.log。

 复制代码

```
1 12.12.2022 04:33:03 10.1.2.1 user=Ulric message="logged in"
2 12.12.2022 06:47:03 10.1.2.2 user=Qin message="logged failed"
3 12.12.2022 06:55:03 10.1.2.2 user=Qin message="logged in"
4 12.12.2022 07:03:03 10.1.2.3 user=Sofia message="logged in"
5 12.12.2022 07:37:03 10.1.2.1 user=Ulric message="logged out"
6 12.12.2022 08:47:03 10.1.2.2 user=Qin message="logged out"
7 12.12.2022 14:34:03 10.1.2.3 user=Sofia message="logged out"
```

之后修改 config.yml 来解析 login.log。

 复制代码

```
1 global:
2   config_version: 3
3 input:
4   type: file
5   path: ./example/login.log
6   readall: true # Read from the beginning of the file? False means we start at
7 imports:
8 - type: grok_patterns
9   dir: ./patterns
10 metrics:
11 - type: counter
12   name: user_activity
13   help: Counter metric example with labels.
14   match: '%{DATE} %{TIME} %{HOSTNAME:instance} user=%{USER:user} message="%{GRE
15   labels:
16     user: '{{.user}}'
17     logfile: '{{base .logfile}}'
18 server:
```

```
19 protocol: http
20 port: 9144
```

使用这个新的配置文件做个测试，下面是返回内容。

 复制代码

```
1 [flashcat@fc-demo-02 ~]$ curl -s 10.100.0.7:9144/metrics | grep user_activity
2 grok_exporter_line_processing_errors_total{metric="user_activity"} 0
3 grok_exporter_lines_matching_total{metric="user_activity"} 7
4 grok_exporter_lines_processing_time_microseconds_total{metric="user_activity"}
5 # HELP user_activity Counter metric example with labels.
6 # TYPE user_activity counter
7 user_activity{logfile="login.log",user="Qin"} 3
8 user_activity{logfile="login.log",user="Sofia"} 2
9 user_activity{logfile="login.log",user="Ulric"} 2
```

看起来 **grok_exporter** 要比 **mtail** 的方式更易用，不过和 **mtail** 一样，如果要对多个应用程序分别进行日志分析处理，就要启动多个 **grok_exporter** 实例，这点还是不太方便。当然，在运算方面，**grok_exporter** 没有 **mtail** 这种类语言式的处理来得灵活方便。至于选用哪个，尺有所短寸有所长，都学会，具体使用场景具体决策。这一讲我就给你介绍这么多，下面我们做一个总结。

小结

我们这一讲介绍的手段，从标题上来看是服务于应用监控，不过实际上也可以用于操作系统、中间件、数据库等其他监控场景，而应用监控本身，反倒不推荐日志监控方式，而是更推荐上一讲介绍的埋点监控方式。这一点，请你一定要注意，毕竟相比埋点方式，日志方式链路又长、性能又差，算是一个不得已而为之的方式。

指标提取的几种方式，总体上来看就是中心端和日志端两种，由于中心端的处理方式多见于商业软件，没有看到开源解决方案，所以我们重点介绍的是日志端的处理方式，日志端的处理核心逻辑都是一样的，通过类似 **tail -f** 的方式不断读取日志内容，然后对每行日志做正则匹配提取，由于日志格式不固定，很难有结构化的处理手段，所以这些工具都是选择使用正则的方式来提取过滤指标。

mtail 和 **grok_exporter** 是日志端处理工具的佼佼者，**mtail** 直接写正则，虽然可阅读性上稍微差了点儿，但是胜在逻辑处理能力，可以对提取的变量做运算，就像一门小语言，所以 **mtail**

把这些提取规则的文件叫做 `program`。`grok_exporter` 可以使用预定义的 `pattern` 名称配置匹配规则，更易读、易维护，运算方面则显得稍弱。

日志监控

指标抓取典型做法

- 中心端
 - 流式处理，流计算任务插入 Kafka 通道上
 - 定时查询，直接查 ES 接口，传入聚合计算函数由 ES 做聚合计算
- 日志端——常用工具：`mtail`、`grok_exporter`

mtail

- 默认在 3903 端口暴露 Prometheus 协议格式的 metrics 指标数据
- 支持 Gauge、Counter、Histogram 三种类型
- `mtail` 文件的典型写法是正则匹配规则 + 大括号里的 action
- 使用 `(?P<NAME>RE)` 作为命名捕获，后面的逻辑就可以使用 `NAME` 作为变量
- 支持自定义静态标签，比如使用 `hidden text`，引用此类变量时小心别加 `$` 符号
- `getfilename()` 可以拿到当前文件名
- 支持类似 `if else` 的结构，而且支持嵌套
- `strptime` 函数用于定义日志中的时间 `format` 格式
- `override_timezone` 用于指定时区，不指定默认使用 UTC
- `emit_metric_timestamp` 用于设置时间戳是否在 metrics 接口内容中返回
- 一般一类日志用一个 `mtail` 进程来处理

grok_exporter

- 预定义了 100 多个正则片段，可以复用
- 在一些场景下更易用一些，不过计算方面不太好用，不如 `mtail` 灵活



互动时刻

由于 `mtail` 和 `grok_exporter` 都是通过正则提取的方式来处理非结构化的日志数据的，性能是个比较关键的问题，如果日志量很大，可能会侵蚀较多的机器算力，甚至影响上面运行的服务。有没有什么实践方式可以提升性能呢？欢迎在留言区分享你的想法，也欢迎你把今天的内容分享给你身边的朋友，邀他一起学习。我们下一讲再见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 19 | 应用监控：如何使用埋点方式对应用监控？

[下一篇](#) 21 | 事件管理（上）：事件降噪的几个典型手段

精选留言 (6)

[写留言](#)

刘涛

2023-02-23 来自广东

中心处理，filebeat kafka flink



2



那一刻

2023-03-01 来自北京

讨论：物理机大概率会有混部 5 个甚至 50 个服务的场景，容器又必然是一个服务一个 Pod，那虚拟机呢？做成大规格的好，还是小规格的好呢？是有混部好还是没有混部好呢？

我觉得对于虚拟机，大规格适合混部，小规格适合单独部署。大规格混部的话，可以最大化利用资源。不过，从监控角度来说，混部会对于数据监控带来干扰因素，因为混部破坏了隔离性。

思考题：由于 mtail 和 grok_exporter 都是通过正则提取的方式来处理非结构化的日志数据的，性能是个比较关键的问题，如果日志量很大，可能会侵蚀较多的机器算力，甚至影响上面运行的服务

我没有过多的使用经验，谈谈我的想法。对于日志量大，可以考虑分段处理，可以先把日志切分成多段，然后每段分别处理，减少一次处理的数据量。另外，为了控制 mtail 和 grok_exporter 侵蚀较多的算力，可以通过 cgroup 的方式来控制 max cpu 使用率。

问题：请问老师 Telegraf 的 plugin logparser 和 tail 可以读取 log 文件，同时也有 prometheus_client，实际工作中有应用么？

作者回复: 读取日志到中心, 大都还是采用EFK的生态; Telegraf 采集数据通过 prometheus_client 暴露, 没有看到哪个公司这么用, 通过 remotewrite 写数据到后端存储的倒是不少, 如果是pull的方式, 大都还是使用node-exporter居多



Kevin

2023-02-24 来自北京

看了下目录, 这是指标搜集的最后一章了。想问下, categraf没有做etcd的指标采集吗? 看conf目录下没有input.etcd目录

作者回复: etcd直接暴露prometheus协议的监控数据, 使用input.prometheus直接抓就好了, Kubernetes监控章节其实介绍过如何采集etcd的数据了



林龍

2023-02-22 来自广东

由于项目中已经搭建了opentracing链路, 把数据加载到prometheus, 请问这种方案有没有什么缺点



peter

2023-02-22 来自北京

请教老师几个问题:

Q1: 应用保存日志还有用吗?

既然对于应用的监控推荐使用埋点方式, 不推荐使用日志方式。那么, 对于应用, 还有必要打印、保存日志吗? 尤其是线上环境。

Q2: 用云服务的话, 一般是虚拟机, categraf怎么部署到机器上? 机器是虚拟的, 是不确定的实体, 怎么把categraf部署到特定的机器上啊。

Q3: 注册用户100万的网站, 适合用什么监控?

通过前面的学习, 感觉Prometheus适合比较大的规模的网站。那么, 对于注册用户100万的网站, 是不是有更合适的监控方案? (注1: 对于网站规模大小, 我不很清楚; 100万用户的规模, 算大还是小, 不清楚, 只是个人臆测; 注2: 也许Prometheus也适合小规模网站)

作者回复: 1, 需要保存, 指标是统计数据, 日志是细节

2, 虚拟机对于用户来说跟正常的机器没啥太大区别。。。。

3, 没法通过注册用户量衡量, 根据监控目标衡量





Tangzen

2023-02-22 来自北京

同类产品有loki+grafana,sls

