

19 | 应用监控：如何使用埋点方式对应用监控？

2023-02-20 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 13:51 大小 12.66M



你好，我是秦晓辉。

前面几讲我们主要是介绍了常见的中间件、数据库的监控，统称为组件监控，根据 [第 9 讲](#) 中我们对监控做的分类，还有应用和业务层面的监控没有介绍，接下来我会用两讲来介绍这部分内容。

因为业务指标的生成也需要应用程序侧来实现，所以这两个层面的监控可以统称为应用监控。

在指标监控的世界里，应用和业务层面的监控有两种典型手段，一种是在应用程序里埋点，另一种是分析日志，从日志中提取指标。埋点的方式性能更好，也更灵活，只是对应用程序有一定侵入性，而分析日志的话对应用程序侵入性较小，但由于链路较长、需要做文本分析处理，

性能较差，需要更多算力支持。这一讲我们先来介绍第一种方式，使用埋点的方式做应用和业务监控。

埋点方式介绍

所谓的埋点，就是指应用程序内嵌了埋点的 SDK（一个 lib 库），然后在一些关键代码流程里调用 SDK 提供的方法，记录下各种关键指标。比如某个 HTTP 服务，提供了 10 个接口，每个接口的处理花费了多少毫秒，就可以作为指标记录下来。

你可能会疑惑，我的监控系统已经提供了 PUSH 接口了，比如 Prometheus 的 Pushgateway 组件，在应用程序里直接调用 Pushgateway 接口推数据不就行了吗？为什么还需要 SDK 呢？

核心原因是 SDK 可以帮我们封装一些通用的计算逻辑。比如有个指标是 Summary 类型（[🔗 第 2 讲](#)介绍过这个数据类型），可以提供某个接口 99 分位的延迟数据。如果没有 SDK，我们需要怎么做呢？每当这个接口响应一次请求，就记录一个延迟时间，然后存入一个内存数据结构，等到一个时间间隔，比如 10 秒钟，就把这段时间内所有的延迟数据排个序，然后取 99 分位的值，最后调用 Pushgateway 的接口推过去。很麻烦是不是？

我们需要准备这个数据结构，这个数据结构还需要线程安全。如果请求非常频繁，耗时数据很多，还得进行限制，比如这个数据结构只保存 1000 个耗时数据。然后到了时间间隔之后需要排序，还要组织成监控系统需要的数据格式，还需要调用 HTTP 接口推送数据。

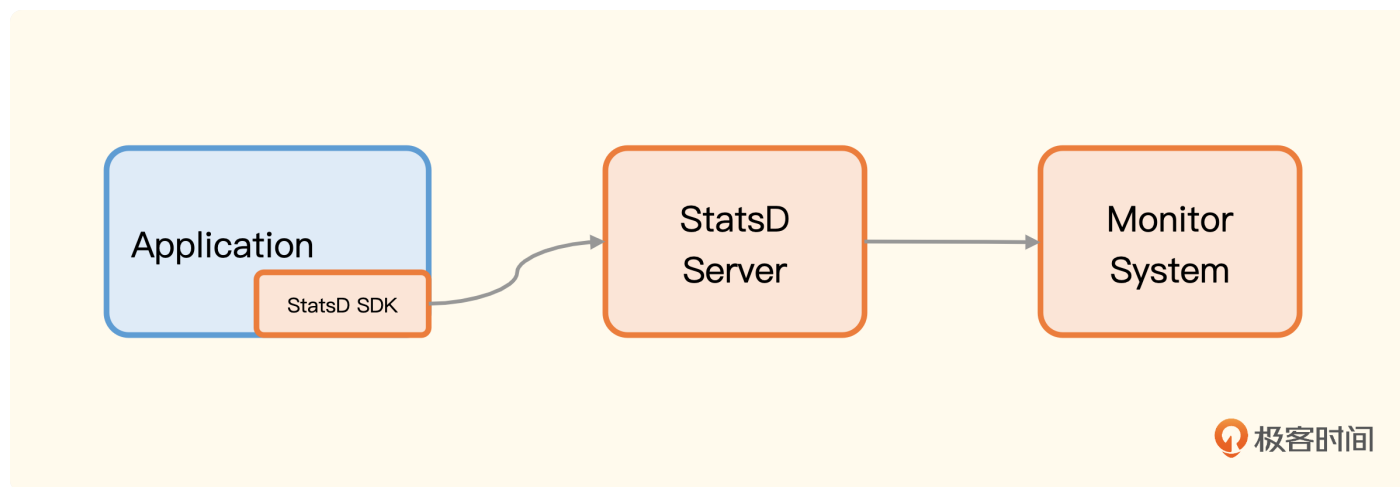
而 SDK 就是做这些通用逻辑的，应用程序要做的，就是拿到延迟数据之后，调用 SDK 的方法告知 SDK，说又有一次新的接口调用，延迟数据是多少，哥们后续就交给你了，SDK 就能完成剩余所有的事情。

业界比较知名的跨语言埋点工具是 [🔗 StatsD](#) 和 [🔗 Prometheus](#)。当然，有些语言有自己生态的常用工具，比如 Java 生态的 Micrometer，不过一般公司都会使用多种语言，这些跨语言的埋点方案通常使用频率会更高。所以，这一讲我会分别介绍这两个方式，让你有个全面的认识，我们先从 StatsD 开始。

StatsD

StatsD 开始被熟知是因为 [🔗 《Measure Anything, Measure Everything》](#) 这篇文章，Etsy 的工程师使用 NodeJS 开源了这个 [🔗 项目](#)。

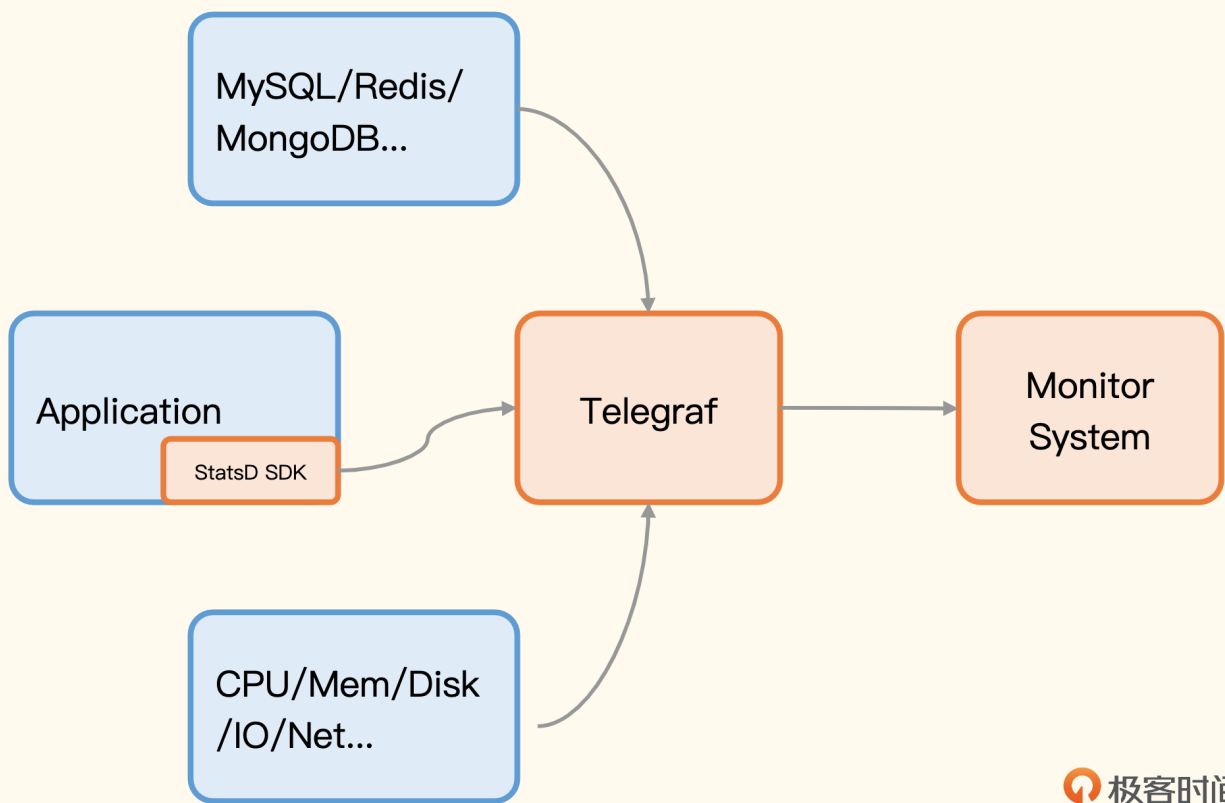
StatsD 有个很大的特点是使用 UDP 传输协议，大部分计算逻辑都挪到了 StatsD 的 Server，SDK 层面的工作非常轻量。



StatsD 架构图

StatsD SDK 与 StatsD Server 之间通信使用的是 UDP 协议，UDP 协议是 fire-and-forget，无需建立连接，即使 StatsD Server 挂了，也不影响应用程序，而对于延迟分布区间这样的计算逻辑，是在 StatsD Server 里计算的，也不会影响应用程序，所以整个 StatsD 的设计是非常轻量的，对应用程序基本没有影响。

由于 StatsD 相当知名，所以很多采集器都实现了 StatsD 的协议，比如 Telegraf、Datadog-agent，也就是说，图上的 StatsD Server 是可以换成 Telegraf 或 Datadog-agent 的。这样就不用部署太多进程，一个采集器包打天下，就拿 Telegraf 来说吧，替换后架构就变成了这样。



下面我来演示一下，用一个小程序做个埋点，让你有个感性的认识。我们就把 **Telegraf** 当做 **StatsD Server** 来接收埋点数据，**Telegraf** 是 **InfluxData** 主导开源的监控采集器，我之前做过一次调研，调研笔记我也放在 [这里](#)，供你参考。

Telegraf 提供 **StatsD** 的 **input** 插件，在配置文件中搜索 **inputs.statsd** 就能找到相关的配置段了，里边有详尽的注释，我就不再赘述了，直接上配置样例。

复制代码

```
1 [[inputs.statsd]]
2 protocol = "udp"
3 service_address = ":8125"
4 percentiles = [50.0, 90.0, 99.0, 99.9, 99.95, 100.0]
5 metric_separator = "_"
```

Telegraf 重启之后就会在 **8125** 开启 **UDP** 监听，我们写的程序就可以往这个地址推送监控数据了。因为我对 **Go** 语言比较熟悉，所以这里我给你一个 **Go** 埋点的例子。

复制代码

```
1 package main
2 import (
3     "fmt"
```

```

4     "math/rand"
5     "net/http"
6     "time"
7     "github.com/smira/go-statsd"
8 )
9 var client *statsd.Client
10 func homeHandler(w http.ResponseWriter, r *http.Request) {
11     start := time.Now()
12     // random sleep
13     num := rand.Int31n(100)
14     time.Sleep(time.Duration(num) * time.Millisecond)
15     fmt.Fprintf(w, "duration: %d", num)
16     client.Incr("requests.counter,page=home", 1)
17     client.PrecisionTiming("requests.latency,page=home", time.Since(start))
18 }
19 func main() {
20     // init client
21     client = statsd.NewClient("localhost:8125",
22         statsd.TagStyle(statsd.TagFormatInfluxDB),
23         statsd.MaxPacketSize(1400),
24         statsd.MetricPrefix("http."),
25         statsd.DefaultTags(statsd.StringTag("service", "n9e-webapi"), statsd.St
26     )
27     defer client.Close()
28     http.HandleFunc("/", homeHandler)
29     http.ListenAndServe(":8000", nil)
30 }

```

这个 Web 服务只有一个根路径，逻辑也很简单，就是随机 sleep 几十个毫秒当做业务处理时间。整体逻辑是这样的：首先，我们要通过 `statsd.NewClient` 初始化一个 StatsD 的客户端，参数中指定了 StatsD 的 Server 地址（在这个例子里就是 Telegraf 的 8125），指定了所有监控指标的前缀是 `http.`，还指定了两个全局 Tag，一个是 `service=n9e-webapi`，另一个是 `region=bj`。通过 `TagStyle` 指定要发送的是 InfluxDB 样式的标签。

然后，在请求的具体处理逻辑里上报了两个监控指标，一个是 `requests.counter`，另一个是 `requests.latency`，并为它们指定了一个指标级别的标签 `page=home`，整体看起来还是比较简单的。

Telegraf 支持多种 output 插件，可以使用比较快捷的 `outputs.file` 插件，把生成的指标写到 `stdout`，来验证效果。

 复制代码

1 `[[outputs.file]]`

请求数量和请求延时都是典型的应用层面的监控指标，如果我们想要做业务指标监控，比如订单量监控，应该怎么做呢？其实逻辑是完全一样的，仿照 `requests.counter` 的写法，做一个订单量的指标，每次创建一个新订单的时候，就给订单量指标 +1。一般一个服务会部署多个实例，每个实例都统计了自己的订单量，要计算最近一分钟的订单数量的话，只需要在服务端使用 PromQL 做二次汇总计算。

StatsD 的埋点方式我们就介绍这么多，下面我们介绍一下 Prometheus 的埋点方式。

Prometheus

Prometheus 的埋点方式跟 StatsD 很像，对于请求数量和延迟这样的监控指标，也是在请求处理完成之后，调用 SDK 的方法进行记录的。不过，如果每个方法都要加这么几行代码就显得太冗余了，最好还是通过 AOP 的方式做一些切面逻辑，Nightingale 的 Webapi 模块就是这么干的，我直接用这个 [🔗 代码](#) 给你讲解。

Webapi 的职能是提供一系列 HTTP 接口给 JavaScript 调用，我们需要监控这些接口的调用量、成功率、延迟数据。埋点之前我们先规划一下标签，我们给每个 HTTP 接口规划 4 个标签。

- **service**: 服务名称，要求全局唯一，可以和其他服务区分开。
- **code**: HTTP 返回状态码，可以根据这个信息得知 4xx 的比例是多少，5xx 的比例是多少，计算成功率。
- **path**: 接口路径，比如 `/api/v1/users`，有时候我们会在接口路径中放置 URL 参数，比如 `/api/v1/user/23`、`/api/v1/user/12` 是请求 id 为 23 和 12 的用户信息。这个时候不能直接把这个 URL 作为接口路径的标签值，否则这个指标颗粒度就太细了，应该把接口路径的标签值设置成 `/api/v1/user/:id`。
- **method**: HTTP 方法，GET、POST、DELETE、PUT 等。

Prometheus 埋点需要提前把指标变量定义出来，我们使用一个单独的包来放置，你可以看一下 [🔗 相关代码](#)。

```

1 package stat
2
3
4 import (
5     "time"
6
7     "github.com/prometheus/client_golang/prometheus"
8 )
9
10 const Service = "n9e-webapi"
11
12 var (
13     labels = []string{"service", "code", "path", "method"}
14
15     uptime = prometheus.NewCounterVec(
16         prometheus.CounterOpts{
17             Name: "uptime",
18             Help: "HTTP service uptime.",
19         }, []string{"service"},
20     )
21
22     RequestCounter = prometheus.NewCounterVec(
23         prometheus.CounterOpts{
24             Name: "http_request_count_total",
25             Help: "Total number of HTTP requests made.",
26         }, labels,
27     )
28
29     RequestDuration = prometheus.NewHistogramVec(
30         prometheus.HistogramOpts{
31             Buckets: []float64{.01, .1, 1, 10},
32             Name:     "http_request_duration_seconds",
33             Help:     "HTTP request latencies in seconds.",
34         }, labels,
35     )
36 )
37
38 func Init() {
39     // Register the summary and the histogram with Prometheus's default registry.
40     prometheus.MustRegister(
41         uptime,
42         RequestCounter,
43         RequestDuration,
44     )
45
46     go recordUptime()
47 }
48
49 // recordUptime increases service uptime per second.
50 func recordUptime() {
51     for range time.Tick(time.Second) {
52         uptime.WithLabelValues(Service).Inc()
53     }
54 }

```


uptime 变量是顺手为之，统计进程启动了多长时间，不用太关注，RequestCounter 和 RequestDuration，分别统计请求流量和请求延迟。Init 方法是在 Webapi 模块进程初始化的时候调用，所以进程一起，就会自动注册好。

然后我们写一个 middleware，在请求进来的时候拦截一下，省得每个请求都要去统计，你可以看一下 middleware 方法的 [代码](#)。

 复制代码

```
1 func stat() gin.HandlerFunc {
2     return func(c *gin.Context) {
3         start := time.Now()
4         c.Next()
5
6         code := fmt.Sprintf("%d", c.Writer.Status())
7         method := c.Request.Method
8         labels := []string{promstat.Service, code, c.FullPath(), method}
9
10        promstat.RequestCounter.WithLabelValues(labels...).Inc()
11        promstat.RequestDuration.WithLabelValues(labels...).Observe(float64(time.
12    })
13 }
```

有了这个 middleware 之后，new 出 gin 的 engine 的时候，就立马 Use 一下。

 复制代码

```
1 ...
2 r := gin.New()
3 r.Use(stat())
4 ...
```

最后，监控数据要通过 /metrics 接口暴露出去，我们要暴露这个请求端点。

 复制代码

```
1 import (
2     ...
3     "github.com/prometheus/client_golang/prometheus/promhttp"
4 )
5 func configRoute(r *gin.Engine, version string) {
6     ...
```



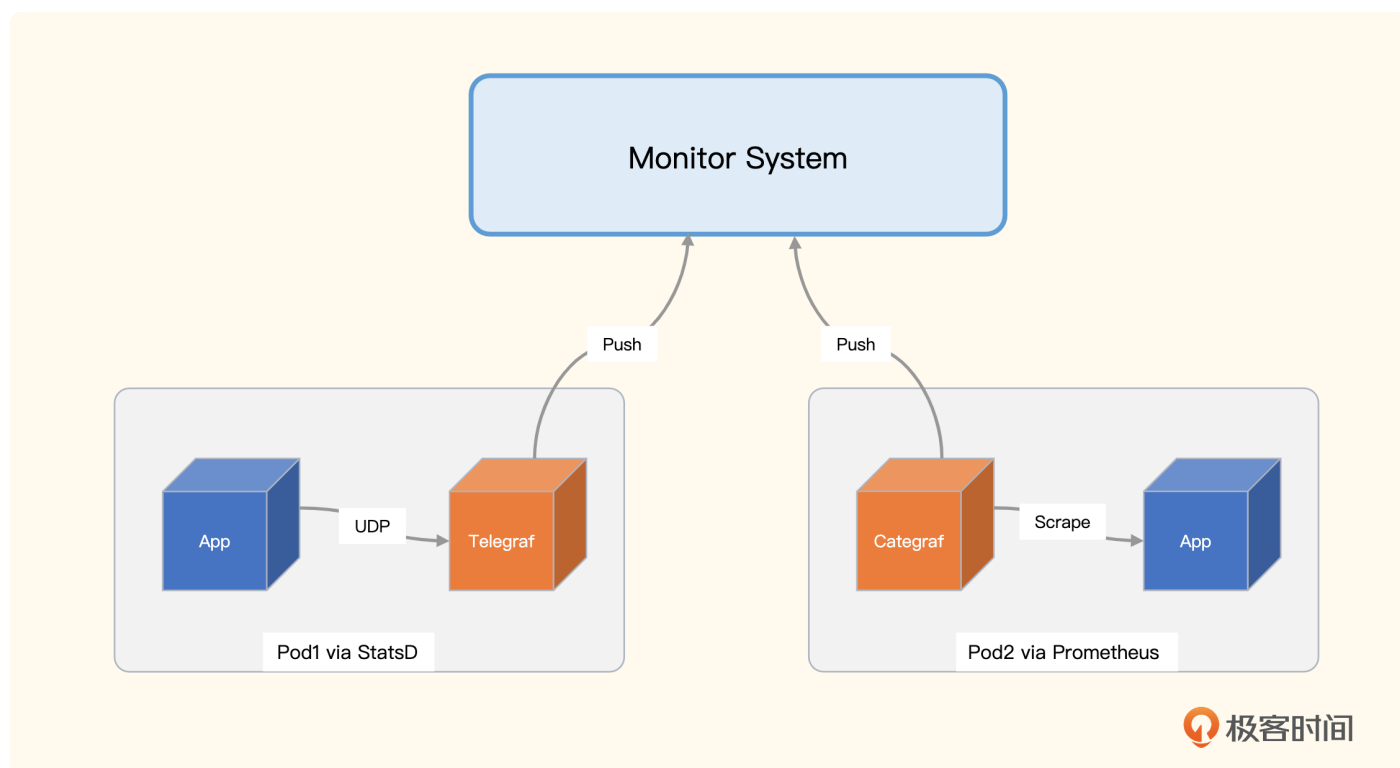
```
7 r.GET("/metrics", gin.WrapH(promhttp.Handler()))
8 }
```

这样，每个 Webapi 接口的流量和成功率都可以监控到了。如果你也部署了 **Nightingale**，请求 **Webapi**（默认是 **18000**）的 **/metrics** 接口看看返回的内容吧。

无论是 **StatsD**，还是使用 **Prometheus**，总之通过埋点我们采集到了监控指标。接下来这些数据如何进入监控服务端呢？我们看一下这个通路典型的构建方式。

数据传输通路

使用 **StatsD** 的埋点方式，数据通过 **UDP** 推给 **Telegraf**，**Telegraf** 推给后端监控系统。如果是通过 **Prometheus** 的方式来埋点，就是暴露 **/metrics** 接口，等待监控系统来拉。如果应用是部署在物理机或虚拟机上，直接通过本地的监控 **agent** 来拉取即可。如果应用是部署在 **Kubernetes** 的 **Pod** 里，则有两种办法来拉取数据，一个是 **Sidecar** 模式，一个是中心端服务发现的模式。下面这个示意图展示的是 **Sidecar** 模式。

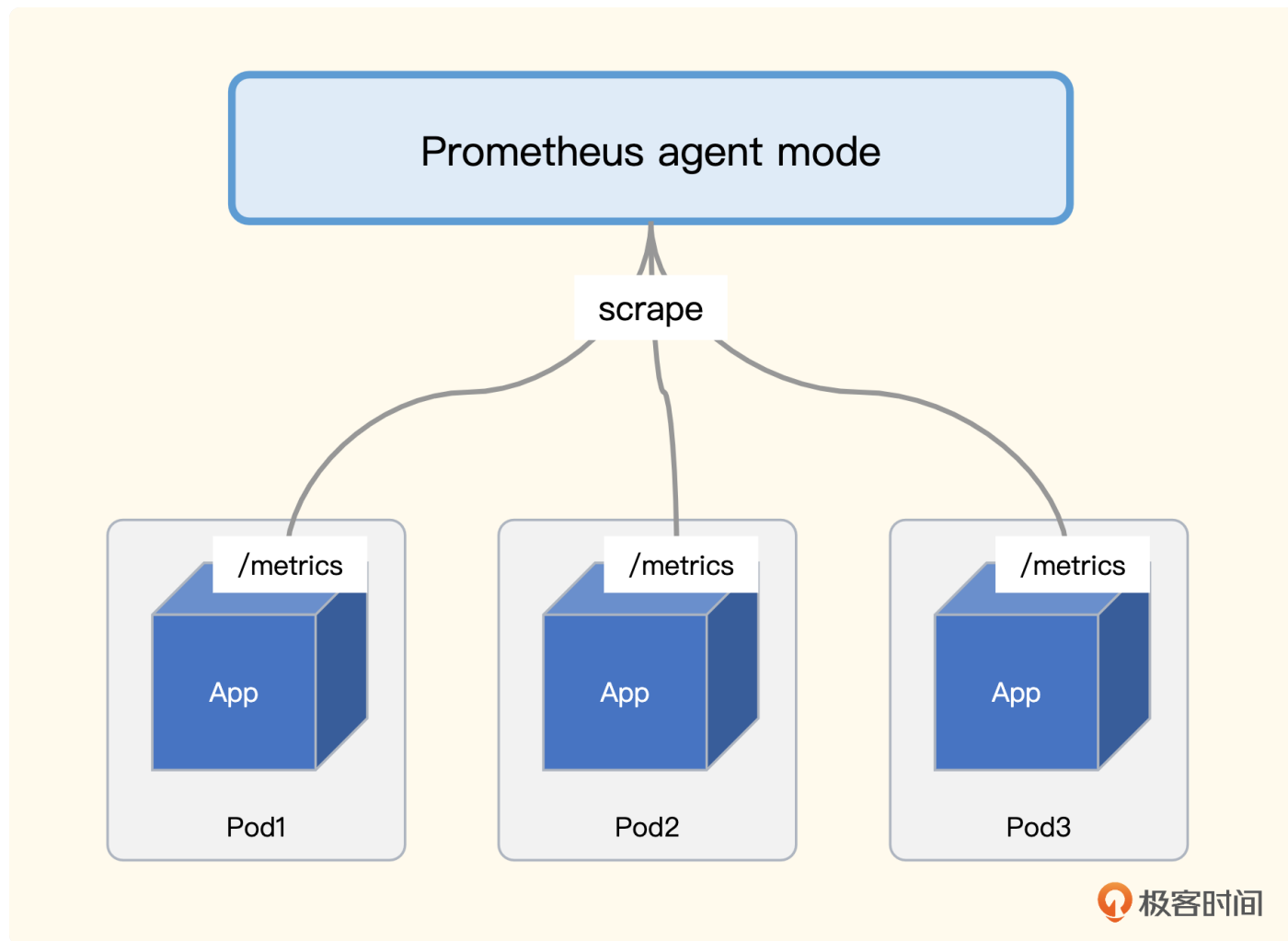


Sidecar 模式

左侧 **Pod1** 里有两个容器，**App** 通过 **StatsD** 埋点，然后通过 **UDP** 推给 **Telegraf**，**Telegraf** 接收到数据之后做二次计算，把结果推给监控服务端；右侧 **Pod2** 里也有两个容器，**App** 通过 **Prometheus SDK** 埋点，暴露 **/metrics** 接口，**Categraf** 通过这个接口拉取数据，然后推给监控服务端。

这种方式的优点是比较灵活，Pod 内怎么做应用自己说了算。即使给 `/metrics` 接口增加一些认证鉴权、指标过滤、扩展标签的逻辑，都不影响其他的 Pod。数据是推给监控服务端的，监控服务端接收数据的组件可以做成无状态集群，前面架设负载均衡，整个架构非常简单、扩展性也很好。当然缺点也很明显，每个 Pod 里都伴生 Sidecar agent，浪费资源。

StatsD 埋点的方式只能使用 Sidecar 模式传输数据，而 Prometheus SDK 埋点还有第二种数据抓取方式：中心端服务发现模式。



中心端服务发现模式

每个应用容器都统一暴露 `/metrics` 接口，监控抓取器通过 Kubernetes 服务发现机制，找到所有的 Pod，分别抓取即可。不过，并不是所有的 Pod 都暴露指标数据，即使暴露了，接口路径也未必一定是 `/metrics`，这就需要有一个机制和规范，通过这个机制告诉指标抓取器，选择哪个类型的 Pod 抓数据，抓数据的时候使用哪个端口、哪个接口路径来抓。

一般我们使用 Pod 注解来承接这个规范，比如我们制定标准：所有想要被抓取监控数据的 Pod 都统一暴露如下注解。

```

1 prometheus.io/scrape=true
2 prometheus.io/metric_path=<path>
3 prometheus.io/scrape_port=<port>

```

然后我们就可以写抓取 **Job** 来抓取这类数据了，下面我给出一个样例。

```

1 - job_name: "kubernetes-pods"
2   kubernetes_sd_configs:
3     - role: pod
4   relabel_configs:
5     # "prometheus.io/scrape = true" annotation.
6     - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
7       action: keep
8       regex: true
9     # "prometheus.io/metric_path = <metric path>" annotation.
10    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_metric_pat
11      action: replace
12      target_label: __metrics_path__
13      regex: (.+)
14    # "prometheus.io/scrape_port = <port>" annotation.
15    - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_
16      action: replace
17      regex: ([^:]+)(?::\d+)?;(\d+)
18      replacement: $1:$2
19      target_label: __address__
20    - action: labelmap
21      regex: __meta_kubernetes_pod_label_(.+)
22    - source_labels: [__meta_kubernetes_namespace]
23      action: replace
24      target_label: namespace
25    - source_labels: [__meta_kubernetes_pod_name]
26      action: replace
27      target_label: pod

```

我们分析一下中心端服务发现机制的优缺点。优点很明显，不需要 **sidecar** 的 **agent** 了，大幅节省资源，缺点是所有的抓取规则都是统一的，不好为每个 **Pod** 单独指定一些认证机制、过滤规则等。不是说不能干，而是要做的话就得修改中心端这个统一的抓取规则，非常不方便，稍有不慎还容易改错，影响到别的 **Pod**。

当然了，一般来说不提供这个灵活性也不是太大的问题，大家遵照一个统一的规范也挺好的，如果真的有某个 **Pod** 需要额外做一些灵活配置，大不了就不开那些注解，仍使用 **sidecar** 模式

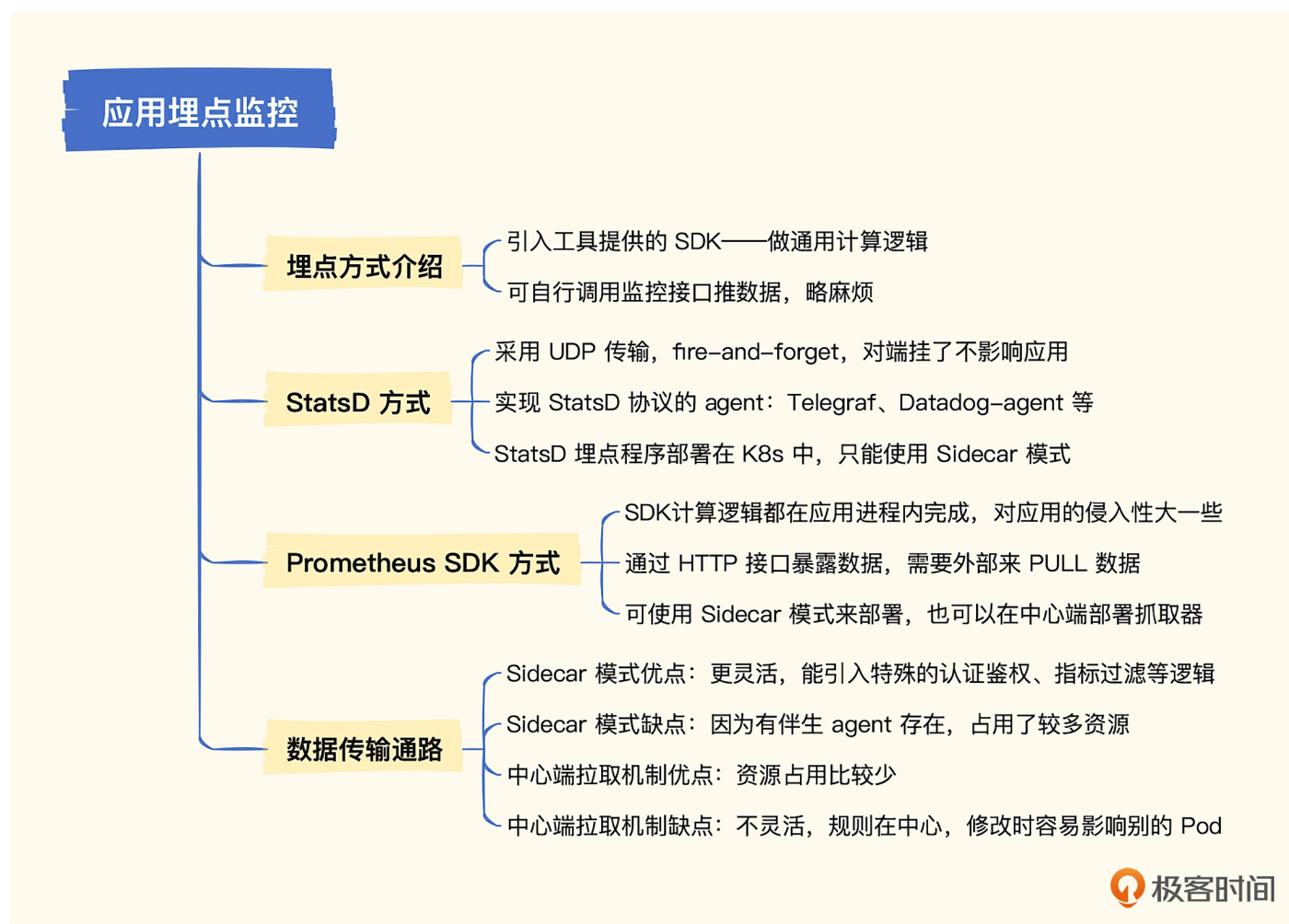
就好。这两种方式并不一定要二选一，同时并用也没有任何问题。

到这里，埋点监控的两种手段就都介绍完了，下面我们做一个小结。

小结

这一讲我们介绍了两种埋点监控应用的方式，一个是 **StatsD**，一个是 **Prometheus**，这两种方式都是跨语言的埋点方案，业界应用广泛。**StatsD** 是推模式，采用 **UDP** 协议，各类计算逻辑挪到了 **StatsD Server** 中，对应用本身不会造成影响。**Prometheus** 是拉模式，**SDK** 的逻辑跑在应用进程里，可能对应用造成一定影响，不过已经有很多公司做了生产验证，**Prometheus** 的 **SDK** 还是很稳定的，大可放心使用。


对于 **Prometheus SDK** 的埋点方案，有两种数据抓取方式，一个是 **Sidecar** 模式，灵活但是占用资源多；一个是中心端服务发现模式，不那么灵活但是资源占用得少，技术上没有非黑即白，建议你根据具体场景灵活选择。



互动时刻

对于 Prometheus SDK 埋点机制，在使用中心端抓取方式的时候，如果目标 Pod 量很大，单个抓取器就抓不过来了，这个时候就需要有个分片机制，用多个抓取器来抓，每个抓取器抓一部分指标。我们应该按照哪些维度做分片呢？具体配置样例是什么样子的？欢迎你留言分享，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！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 6

 提建议

© 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 18 | 组件监控：Kubernetes 控制面组件的关键指标与数据采集

下一篇 20 | 应用监控：如何使用日志来监控应用？

精选留言 (4)

 写留言



Geek_1a3949

2023-02-20 来自上海

尝试回答一下课后题：

分片可以使用Prometheus的hashmod，label选择尽量随机的，比如address，或者address+name等等

```
- job_name: "pods"
...
relabel_configs:
- source_labels: [__address__]
  modulus: 3
  target_label: __tmp_hash
  action: hashmod
- source_labels: [__tmp_hash]
  regex: 1
  action: keep
```



2

**CDS**

2023-03-29 来自浙江

背景

- 1 同事不愿意埋点增加代码
- 2 目前只是希望看见系统（ubuntu 18.04）内各个进程的资源（cpu 内存）的大致趋势
- 3 精度要求为秒级
- 4 只监控本机 不监控其他机器

请问如下方案是否可行

使用Prometheus SDK http方式 拉取固定一个HTTP接口（A服务上的）

A服务通过端口查询pid 在使用top命令去查询资源使用情况

如果可行 github上面有没有类似这样的项目

如果不可行 是使用ebpf进行监控吗

作者回复: categraf 的 input.procstat 插件就可以哈，指定要监控的进程是什么，就可以采集进程的cpu、mem等信息

共 2 条评论 >

**x**

2023-03-09 来自广东

应用进程里的sdk一直在计算和保存数据，长期没有prometheus去访问它的/metrics接口，会有内存问题吗

作者回复: 不会，这个大可放心

**peter**

2023-02-20 来自北京

请教老师几个问题：

Q1: Pod 中的 Sidecar agent就是指Telegraf或Categraf吗？

Q2: Sidecar方式的图中，StatsD用Telegraf;prometheus用Categraf，这是一种典型配置？还是说必须这样？（比如，Prometheus可以使用Telegraf吗？或者StatsD可以使用Categraf吗？）

Q3: 应用层埋点的Prometheus方式，是Prometheus自身具备的一个功能，不是另外的一个软件，对吗？

Q4: 老师经历的公司所用的应用层埋点方案，是用开源框架多还是自研的多？框架的话，是S

tatsD用得更多还是Prometheus用得更多？

Q5：本文的两种埋点方案适用于移动端吗？

比如安卓、iOS，可以用这两种方式吗？

作者回复: 1，是

2，catergraf目前不支持接收statsd协议的数据

3，是应用引入lib

4，开源的多，statsd和prom都挺多

5，适用，但是端上，metrics没有那么关键，更应该引入sentry之类的方案做事件监控

共 2 条评论 >

