Prometheus面试题

资料整理自网络，仅作免费交流分享，侵权删！

更多面试资料请关注微信公众号：程序员面试吧

几点原则：

- 监控是基础设施，目的是为了解决问题，不要只朝着大而全去做，尤其是不必要的指标采集，浪费人力和存储资源（To B商业产品例外）。
- 需要处理的告警才发出来，发出来的告警必须得到处理。
- 简单的架构就是最好的架构，业务系统都挂了，监控也不能挂。Google Sre 里面也说避免使用 Magic 系统，例如机器学习报警阈值、自动修复之类。这一点见仁见智吧，感觉很多公司都在搞智能 AI 运维。

一、版本的选择

Prometheus 当前最新版本为 2.16，Prometheus 还在不断迭代，因此尽量用最新版，1.X版本就不用考虑了。

2.16 版本上有一套实验 UI，可以查看 TSDB 的状态，包括Top 10的 Label、Metric.

Top 10 series count by metric names

Name	Count
kube_pod_container_status_waiting_reason	40264
apiserver_request_latencies_bucket	34760
kube_pod_container_status_last_terminated_reason	28760
kube_pod_container_status_terminated_reason	28760
kube_pod_status_phase	27165
apiserver_response_sizes_bucket	25056
kube_pod_status_scheduled	16215
kube_pod_status_ready	16179
grpc_server_handled_total	16065
apiserver_request_count	11791

二、Prometheus 的局限

- Prometheus 是基于 Metric 的监控，不适用于日志（Logs）、事件(Event)、调用链(Tracing)。
- Prometheus 默认是 Pull 模型，合理规划你的网络，尽量不要转发。

- 对于集群化和水平扩展，官方和社区都没有银弹，需要合理选择 Federate、Cortex、Thanos等方案。
- 监控系统一般情况下可用性大于一致性，容忍部分副本数据丢失，保证查询请求成功。这个后面说 Thanos 去重的时候会提到。
- Prometheus 不一定保证数据准确，这里的不准确一是指 rate、histogram_quantile 等函数会做统计和推断，产生一些反直觉的结果，这个后面会详细展开。二来查询范围过长要做降采样，势必会造成数据精度丢失，不过这是时序数据的特点，也是不同于日志系统的地方。

三、K8S 集群中常用的 exporter

Prometheus 属于 CNCF 项目，拥有完整的开源生态，与 Zabbix 这种传统 agent 监控不同，它提供了丰富的 exporter 来满足你的各种需求。你可以在这里看到官方、非官方的 exporter。如果还是没满足你的需求，你还可以自己编写 exporter，简单方便、自由开放，这是优点。

但是过于开放就会带来选型、试错成本。之前只需要在 zabbix agent里面几行配置就能完成的事，现在你会需要很多 exporter 搭配才能完成。还要对所有 exporter 维护、监控。尤其是升级 exporter 版本时，很痛苦。非官方exporter 还会有不少 bug。这是使用上的不足，当然也是 Prometheus 的设计原则。

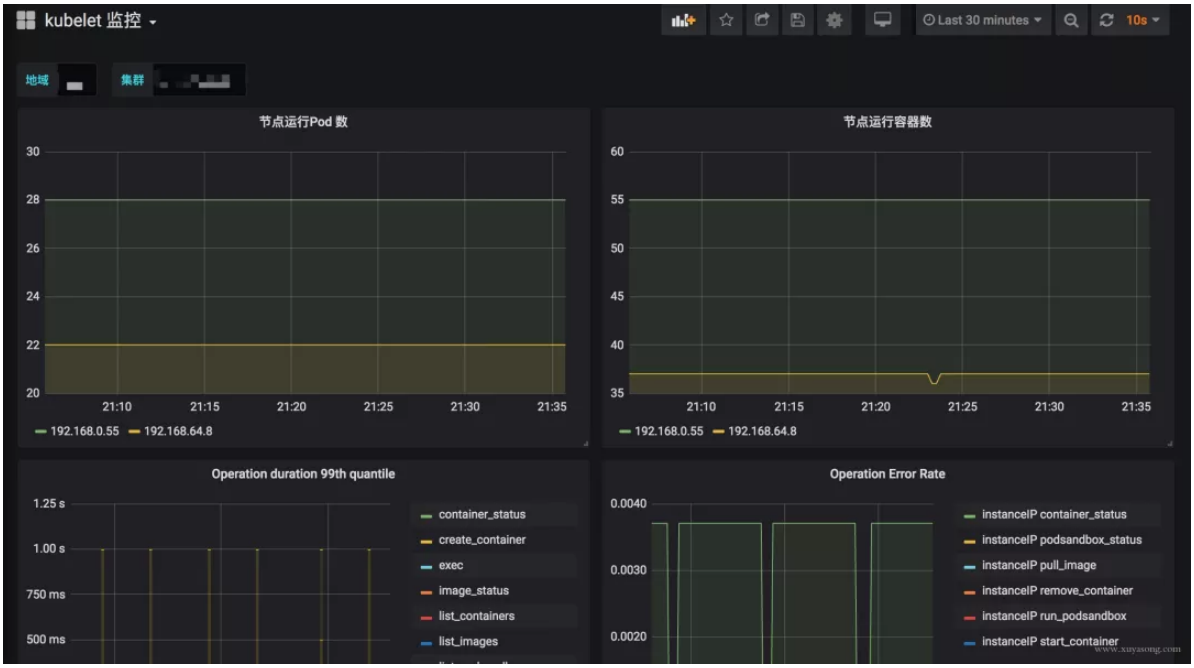
K8S 生态的组件都会提供/metric接口以提供自监控，这里列下我们正在使用的：

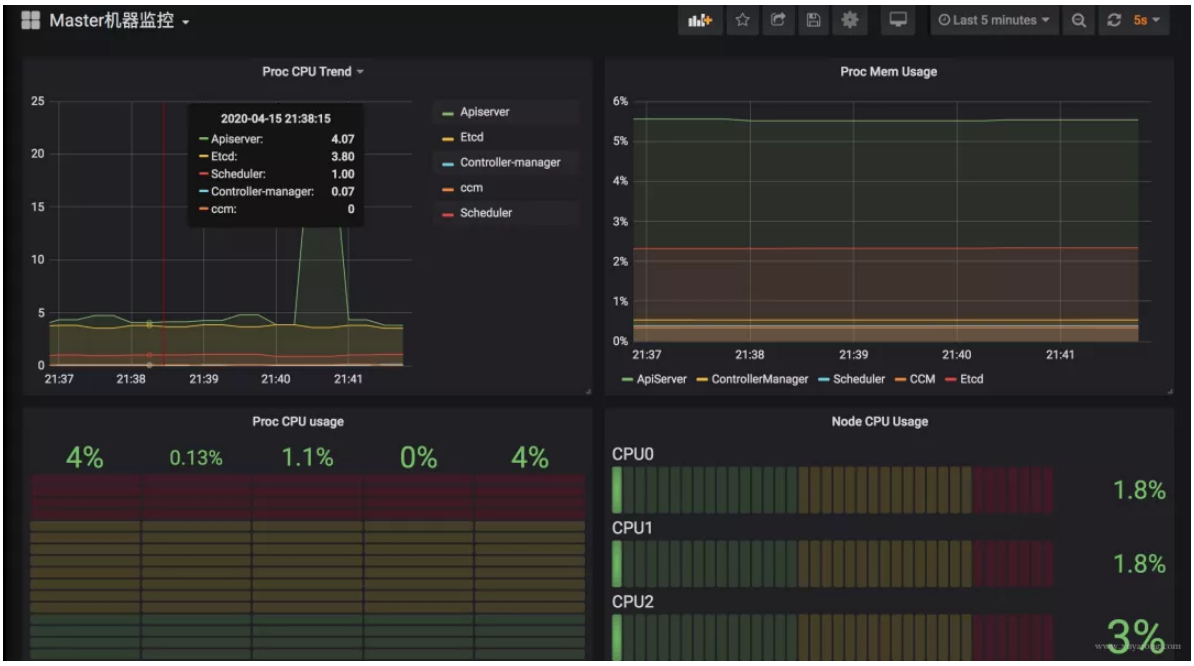
- cadvisor: 集成在 Kubelet 中。
- kubelet: 10255为非认证端口，10250为认证端口。
- apiserver: 6443端口，关心请求数、延迟等。
- scheduler: 10251端口。
- controller-manager: 10252端口。
- etcd: 如etcd 写入读取延迟、存储容量等。
- docker: 需要开启 experimental 实验特性，配置 metrics-addr，如容器创建耗时等指标。
- kube-proxy: 默认 127 暴露，10249端口。外部采集时可以修改为 0.0.0.0 监听，会暴露：写入 iptables 规则的耗时等指标。
- kube-state-metrics: K8S 官方项目，采集pod、deployment等资源的元信息。
- node-exporter: Prometheus 官方项目，采集机器指标如 CPU、内存、磁盘。
- blackbox_exporter: Prometheus 官方项目，网络探测，dns、ping、http监控
- process-exporter: 采集进程指标
- nvidia exporter: 我们有 gpu 任务，需要 gpu 数据监控
- node-problem-detector: 即 npd，准确的说不是 exporter，但也会监测机器状态，上报节点异常打 taint
- 应用层 exporter: mysql、nginx、mq等，看业务需求。

还有各种场景下的自定义 exporter，如日志提取后面会再做介绍。

四、K8S 核心组件监控与 Grafana 面板

k8s 集群运行中需要关注核心组件的状态、性能。如 kubelet、apiserver 等，基于上面提到的 exporter 的指标，可以在 Grafana 中绘制如下图表：





模板可以参考dashboards-for-kubernetes-administrators，根据运行情况不断调整报警阈值。

这里提一下 Grafana 虽然支持了 templates 能力，可以很方便地做多级下拉框选择，但是不支持 templates 模式下配置报警规则，相关issue

官方对这个功能解释了一堆，可最新版本仍然没有支持。借用 issue 的一句话吐槽下：

```
It would be grate to add templates support in alerts. Otherwise the feature
looks useless a bit.
```

关于 Grafana 的基础用法，可以看这个文章

五、采集组件 All IN One

Prometheus 体系中 Exporter 都是独立的，每个组件各司其职，如机器资源用 Node-Exporter，Gpu有Nvidia Exporter等等。但是 Exporter 越多，运维压力越大，尤其是对 Agent做资源控制、版本升级。我们尝试对一些Exporter进行组合，方案有二：

1. 通过主进程拉起N个 Exporter 进程，仍然可以跟着社区版本做更新、bug fix。
2. 用Telegraf来支持各种类型的 Input，N 合 1。

另外，Node-Exporter 不支持进程监控，可以加一个Process-Exporter，也可以用上边提到的 Telegraf，使用 procstat 的 input来采集进程指标。

六、合理选择黄金指标

采集的指标有很多，我们应该关注哪些？Google 在“Sre Handbook”中提出了“四个黄金信号”：延迟、流量、错误数、饱和度。实际操作中可以使用 Use 或 Red 方法作为指导，Use 用于资源，Red 用于服务。

- Use 方法：Utilization、Saturation、Errors。如 Cadvisor 数据
- Red 方法：Rate、Errors、Duration。如 Apiserver 性能指标

Prometheus 采集中常见的服务分三种：

1. 在线服务：如 Web 服务、数据库等，一般关心请求速率，延迟和错误率即 RED 方法
2. 离线服务：如日志处理、消息队列等，一般关注队列数量、进行中的数量，处理速度以及发生的错误即 Use 方法
3. 批处理任务：和离线任务很像，但是离线任务是长期运行的，批处理任务是按计划运行的，如持续集成就是批处理任务，对应 K8S 中的 job 或 cronjob， 一般关注所花的时间、错误数等，因为运行周期短，很可能还没采集到就运行结束了，所以一般使用 Pushgateway，改拉为推。

对 Use 和 Red 的实际示例可以参考容器监控实践—K8S常用指标分析这篇文章。

七、K8S 1.16中 Cadvisor 的指标兼容问题

在 K8S 1.16版本，Cadvisor 的指标去掉了 pod_Name 和 container_name 的 label，替换为了pod 和 container。如果你之前用这两个 label 做查询或者 Grafana 绘图，需要更改下 Sql 了。因为我们一直支持多个 K8S 版本，就通过 relabel配置继续保留了原来的**_name。

```
metric_relabel_configs:- source_labels: [container]
  regex: (.+)
  target_label: container_name
  replacement: $1
  action: replace
- source_labels: [pod]
  regex: (.+)
  target_label: pod_name
  replacement: $1
  action: replace
```

注意要用 metric_relabel_configs，不是 relabel_configs，采集后做的replace。

八、Prometheus 采集外部 K8S 集群、多集群

Prometheus 如果部署在K8S集群内采集是很方便的，用官方给的Yaml就可以，但我们因为权限和网络需要部署在集群外，二进制运行，采集多个 K8S 集群。

以 Pod 方式运行在集群内是不需要证书的（In-Cluster 模式），但集群外需要声明 token之类的证书，并替换**address**，即使用 Apiserver Proxy采集，以 Cadvisor采集为例，Job 配置为：

```yaml
- job_name: cluster-cadvisor
  honor_timestamps: true
  scrape_interval: 30s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: https
  kubernetes_sd_configs:
  - api_server: https://xx:6443
    role: node
    bearer_token_file: token/cluster.token
    tls_config:
      insecure_skip_verify: true
  bearer_token_file: token/cluster.token
  tls_config:
    insecure_skip_verify: true
  relabel_configs:
  - separator: ;
    regex: __meta_kubernetes_node_label_(.+)
    replacement: $1
    action: labelmap
  - separator: ;
    regex: (.*)
    target_label: __address__
    replacement: xx:6443
    action: replace
  - source_labels: [__meta_kubernetes_node_name]
    separator: ;
    regex: (.+)
    target_label: __metrics_path__
    replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
    action: replace
  metric_relabel_configs:
  - source_labels: [container]
    separator: ;
    regex: (.+)
    target_label: container_name
    replacement: $1
    action: replace
  - source_labels: [pod]
    separator: ;
    regex: (.+)
    target_label: pod_name
    replacement: $1
    action: replace
```

bearer_token_file 需要提前生成，这个参考官方文档即可。记得 base64 解码。

对于 cadvisor 来说，`__metrics_path__`可以转换为`/api/v1/nodes/{1}/proxy/metrics/cadvisor`，代表Apiserver proxy 到 Kubelet，如果网络能通，其实也可以直接把 Kubelet 的10255作为 target，可以直接写为：{1}:10255/metrics/cadvisor，代表直接请求Kubelet，规模大的时候还减轻了 Apiserver 的压力，即服务发现使用 Apiserver，采集不走 Apiserver

因为 cadvisor 是暴露主机端口，配置相对简单，如果是 kube-state-metric 这种 Deployment，以 endpoint 形式暴露，写法应该是：

```
- job_name: cluster-service-endpoints
  honor_timestamps: true
  scrape_interval: 30s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: https
  kubernetes_sd_configs:
  - api_server: https://xxx:6443
    role: endpoints
    bearer_token_file: token/cluster.token
    tls_config:
      insecure_skip_verify: true
  bearer_token_file: token/cluster.token
  tls_config:
    insecure_skip_verify: true
  relabel_configs:
  - source_labels:
[__meta_kubernetes_service_annotation_prometheus_io_scrape]
    separator: ;
    regex: "true"
    replacement: $1
    action: keep
  - source_labels:
[__meta_kubernetes_service_annotation_prometheus_io_scheme]
    separator: ;
```

```
    regex: (https?)
    target_label: __scheme__
    replacement: $1
    action: replace
  - separator: ;
    regex: (.*)
    target_label: __address__
    replacement: xxx:6443
    action: replace
  - source_labels: [__meta_kubernetes_namespace,
__meta_kubernetes_endpoints_name,
      __meta_kubernetes_service_annotation_prometheus_io_port]
    separator: ;
    regex: (.+);(.+);(.*)
    target_label: __metrics_path__
    replacement: /api/v1/namespaces/${1}/services/${2}:${3}/proxy/metrics
    action: replace
  - separator: ;
    regex: __meta_kubernetes_service_label_(.+)
    replacement: $1
    action: labelmap
  - source_labels: [__meta_kubernetes_namespace]
    separator: ;
    regex: (.*)
    target_label: kubernetes_namespace
    replacement: $1
    action: replace
  - source_labels: [__meta_kubernetes_service_name]
    separator: ;
    regex: (.*)
    target_label: kubernetes_name
    replacement: $1
    action: replace
```

对于 endpoint 类型，需要转换`__metrics_path__`为`/api/v1/namespaces/{1}/services/{2}:${3}/proxy/metrics`，需要替换 namespace、svc 名称端口等，这里的写法只适合接口为/metrics的exporter，如果你的 exporter 不是/metrics接口，需要替换这个路径。或者像我们一样统一约束都使用这个地址。

这里的`__meta_kubernetes_service_annotation_prometheus_io_port`来源就是 exporter 部署时写的那个 annotation，大多数文章中只提到`prometheus.io/scrape: 'true'`，但也可以定义端口、路径、协议。以方便在采集时做替换处理。

其他的一些 relabel 如kubernetes_namespace 是为了保留原始信息，方便做 promql 查询时的筛选条件。

如果是多集群，同样的配置多写几遍就可以了，一般一个集群可以配置三类job：

- role:node 的，包括 cadvisor、 node-exporter、kubelet 的 summary、kube-proxy、docker 等指标
- role:endpoint 的，包括 kube-state-metric 以及其他自定义 Exporter
- 普通采集：包括Etcd、Apiserver 性能指标、进程指标等。

九、GPU 指标的获取

nvidia-smi可以查看机器上的 GPU 资源，而Cadvisor 其实暴露了Metric来表示容器使用 GPU 情况，

```
container_accelerator_duty_cycle
container_accelerator_memory_total_bytes
container_accelerator_memory_used_bytes
```

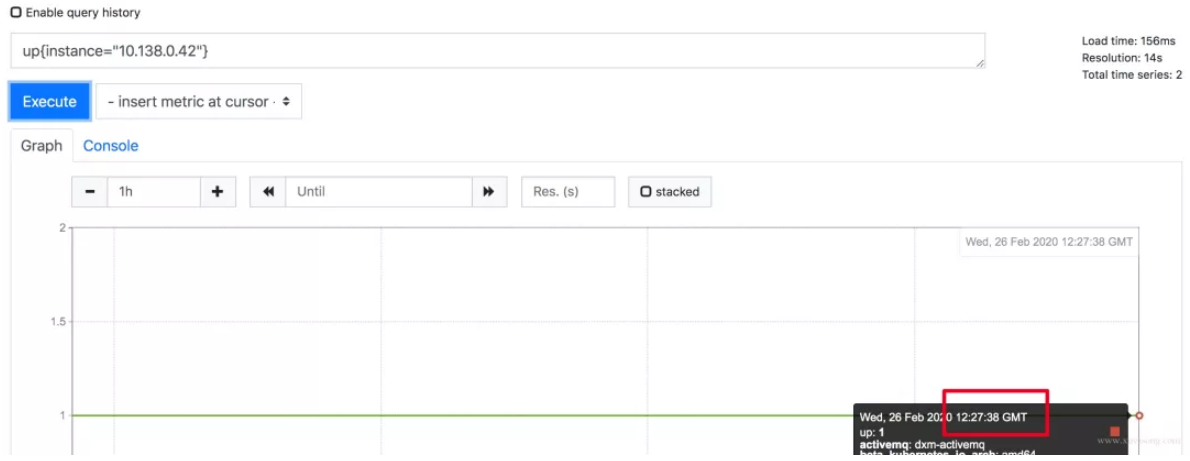
如果要更详细的 GPU 数据，可以安装dcgm exporter，不过K8S 1.13 才能支持。

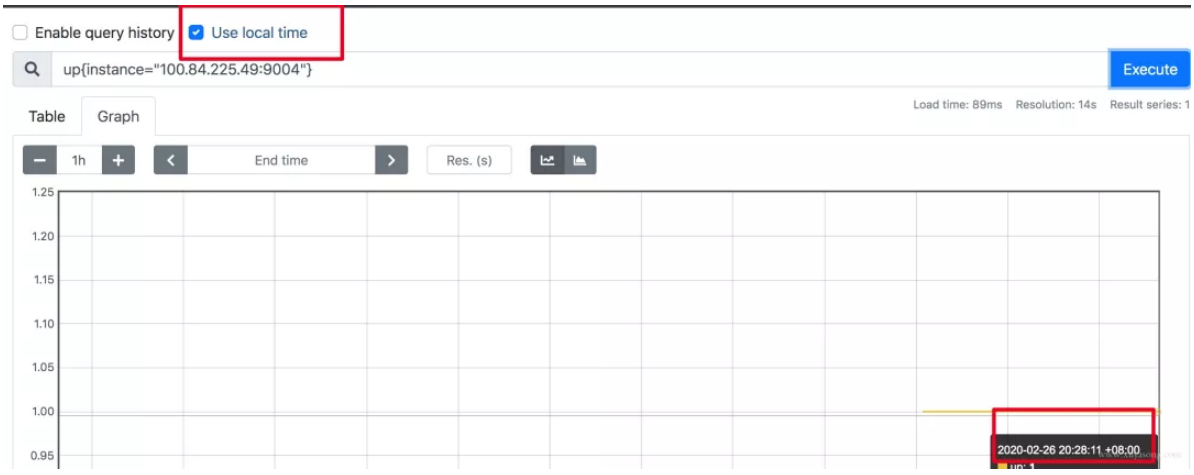
十、更改 Prometheus 的显示时区

Prometheus 为避免时区混乱，在所有组件中专门使用 Unix Time 和 Utc 进行显示。不支持在配置文件中设置时区，也不能读取本机 /etc/timezone 时区。

其实这个限制是不影响使用的：

- 如果做可视化，Grafana是可以做时区转换的。
- 如果是调接口，拿到了数据中的时间戳，你想怎么处理都可以。
- 如果因为 Prometheus 自带的 UI 不是本地时间，看着不舒服，2.16 版本的新版 Web UI已经引入了Local Timezone 的选项，区别见下图。
- 如果你仍然想改 Prometheus 代码来适应自己的时区，可以参考这篇文章。





关于 timezone 的讨论，可以看这个issue。

十一、如何采集 LB 后面的 RS 的 Metric

假如你有一个负载均衡 LB，但网络上 Prometheus 只能访问到 LB 本身，访问不到后面的 RS，应该如何采集 RS 暴露的 Metric?

- RS 的服务加 Sidecar Proxy，或者本机增加 Proxy 组件，保证 Prometheus 能访问到。
- LB 增加 /backend1 和 /backend2请求转发到两个单独的后端，再由 Prometheus 访问 LB 采集。

十二、Prometheus 大内存问题

随着规模变大，Prometheus 需要的 CPU 和内存都会升高，内存一般先达到瓶颈，这个时候要么加内存，要么集群分片减少单机指标。这里我们先讨论单机版 Prometheus 的内存问题。

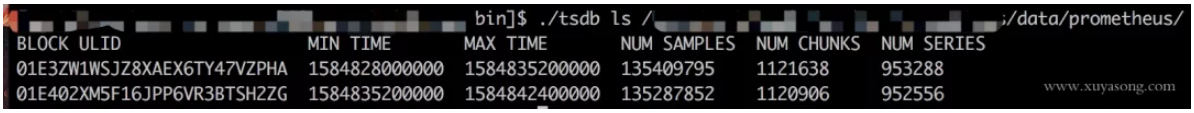
原因：

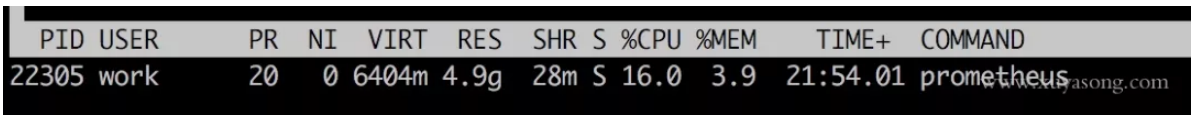
- Prometheus 的内存消耗主要是因为每隔2小时做一个 Block 数据落盘，落盘之前所有数据都在内存里面，因此和采集量有关。
- 加载历史数据时，是从磁盘到内存的，查询范围越大，内存越大。这里面有一定的优化空间。
- 一些不合理的查询条件也会加大内存，如 Group 或大范围 Rate。

我的指标需要多少内存：

- 作者给了一个计算器，设置指标量、采集间隔之类的，计算 Prometheus 需要的理论内存值：计算公式

以我们的一个 Prometheus Server为例，本地只保留 2 小时数据，95 万 Series，大概占用的内存如下：





有什么优化方案：

- Sample 数量超过了 200 万，就不要单实例了，做下分片，然后通过 Victoriametrics，Thanos，Trickster 等方案合并数据。
- 评估哪些 Metric 和 Label 占用较多，去掉没用的指标。2.14 以上可以看 Tsdb 状态
- 查询时尽量避免大范围查询，注意时间范围和 Step 的比例，慎用 Group。
- 如果需要关联查询，先想想能不能通过 Relabel 的方式给原始数据多加个 Label，一条Sql 能查出来的何必用Join，时序数据库不是关系数据库。

Prometheus 内存占用分析：

- 通过 pprof分析：https://www.robustperception.io/optimising-prometheus-2-6-0-memory-usage-with-pprof
- 1.X 版本的内存：https://www.robustperception.io/how-much-ram-does-my-prometheus-need-for-ingestion

相关 issue：

- https://groups.google.com/forum/#!searchin/prometheus-users/memory%7Csort:date/prometheus-users/q4oiVGU6Bxo/uifpXVw3CwAJ
- https://github.com/prometheus/prometheus/issues/5723
- https://github.com/prometheus/prometheus/issues/1881

十三、Prometheus 容量规划

容量规划除了上边说的内存，还有磁盘存储规划，这和你的 Prometheus 的架构方案有关。

- 如果是单机Prometheus，计算本地磁盘使用量。
- 如果是 Remote-Write，和已有的 Tsdb 共用即可。
- 如果是 Thanos 方案，本地磁盘可以忽略（2H），计算对象存储的大小就行。

Prometheus 每2小时将已缓冲在内存中的数据压缩到磁盘上的块中。包括Chunks、Indexes、Tombstones、Metadata，这些占用了一部分存储空间。一般情况下，Prometheus中存储的每一个样本大概占用1-2字节大小（1.7Byte）。可以通过Promql来查看每个样本平均占用多少空间：

```
 rate(prometheus_tsdb_compaction_chunk_size_bytes_sum[1h])/
  rate(prometheus_tsdb_compaction_chunk_samples_sum[1h])
{instance="0.0.0.0:8890", job="prometheus"}  1.252747585939941
```

如果大致估算本地磁盘大小，可以通过以下公式：

```
磁盘大小=保留时间*每秒获取样本数*样本大小
```

保留时间(retention_time_seconds)和样本大小(bytes_per_sample)不变的情况下，如果想减少本地磁盘的容量需求，只能通过减少每秒获取样本数(ingested_samples_per_second)的方式。

查看当前每秒获取的样本数：

```
rate(prometheus_tsdb_head_samples_appended_total[1h])
```

有两种手段，一是减少时间序列的数量，二是增加采集样本的时间间隔。考虑到 Prometheus 会对时间序列进行压缩，因此减少时间序列的数量效果更明显。

举例说明：

- 采集频率 30s，机器数量1000，Metric种类6000，1000*600026*024 约 200 亿，30G 左右磁盘。
- 只采集需要的指标，如 match[], 或者统计下最常使用的指标，性能最差的指标。

以上磁盘容量并没有把 wal 文件算进去，wal 文件(Raw Data)在 Prometheus 官方文档中说明至少会保存3个 Write-Ahead Log Files，每一个最大为128M(实际运行发现数量会更多)。

因为我们使用了 Thanos 的方案，所以本地磁盘只保留2H 热数据。Wal 每2小时生成一份Block文件，Block文件每2小时上传对象存储，本地磁盘基本没有压力。

关于 Prometheus 存储机制，可以看这篇。

十四、对 Apiserver 的性能影响

如果你的 Prometheus 使用了 kubernetes_sd_config 做服务发现，请求一般会经过集群的 Apiserver，随着规模的变大，需要评估下对 Apiserver性能的影响，尤其是Proxy失败的时候，会导致CPU 升高。当然了，如果单K8S集群规模太大，一般都是拆分集群，不过随时监测下 Apiserver 的进程变化还是有必要的。

在监控Cadvisor、Docker、Kube-Proxy 的 Metric 时，我们一开始选择从 Apiserver Proxy 到节点的对应端口，统一设置比较方便，但后来还是改为了直接拉取节点，Apiserver 仅做服务发现。

十五、Rate 的计算逻辑

Prometheus 中的 Counter 类型主要是为了 Rate 而存在的，即计算速率，单纯的 Counter 计数意义不大，因为 Counter 一旦重置，总计数就没有意义了。

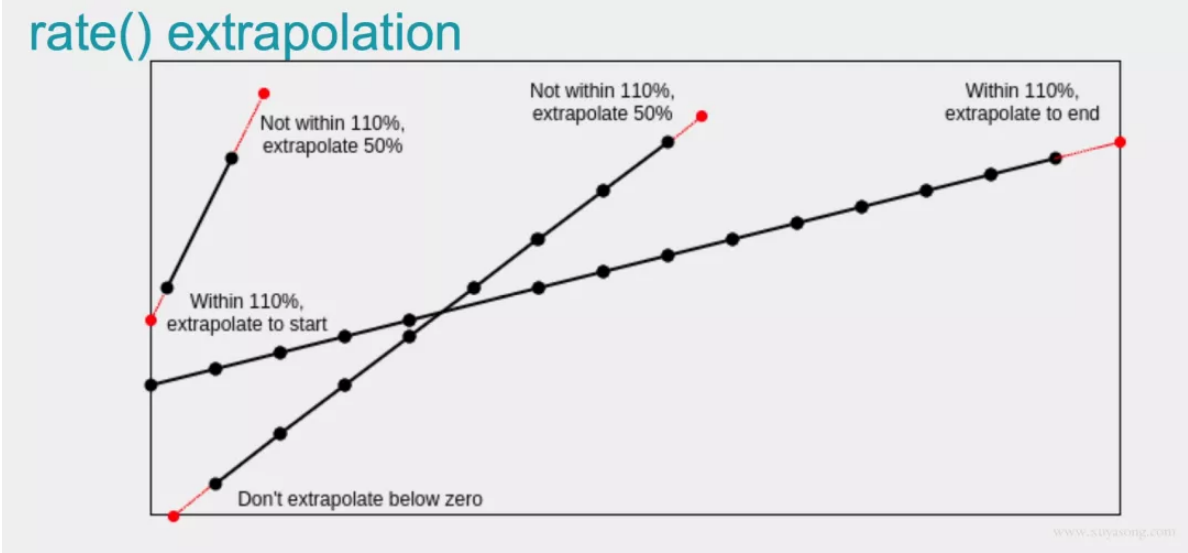
Rate 会自动处理 Counter 重置的问题，Counter 一般都是一直变大的，例如一个 Exporter 启动，然后崩溃了。本来以每秒大约10的速率递增，但仅运行了半个小时，则速率（x_total [1h]）将返回大约每秒5的结果。另外，Counter 的任何减少也会被视为 Counter 重置。例如，如果时间序列的值为[5,10,4,6]，则将其视为[5,10,14,16]。

Rate 值很少是精确的。由于针对不同目标的抓取发生在不同的时间，因此随着时间的流逝会发生抖动，query_range 计算时很少会与抓取时间完美匹配，并且抓取有可能失败。面对这样的挑战，Rate 的设计必须是健壮的。

Rate 并非想要捕获每个增量，因为有时候增量会丢失，例如实例在抓取间隔中挂掉。如果 Counter 的变化速度很慢，例如每小时仅增加几次，则可能会导致【假象】。比如出现一个 Counter 时间序列，值为100，Rate 就不知道这些增量是现在的值，还是目标已经运行了好几年并且才刚刚开始返回。

建议将 Rate 计算的范围向量的时间至少设为抓取间隔的四倍。这将确保即使抓取速度缓慢，且发生了一次抓取故障，您也始终可以使用两个样本。此类问题在实践中经常出现，因此保持这种弹性非常重要。例如，对于1分钟的抓取间隔，您可以使用4分钟的 Rate 计算，但是通常将其四舍五入为5分钟。

如果 Rate 的时间区间内有数据缺失，他会基于趋势进行推测，比如：



十六、Prometheus 重启慢与热加载

Prometheus 重启的时候需要把 Wal 中的内容 Load 到内存里，保留时间越久、Wal 文件越大，重启的实际越长，这个是 Prometheus 的机制，没得办法，因此能 Reload 的就不要重启，重启一定会导致短时间的不可用，而这个时候Prometheus高可用就很重要了。

Prometheus 也曾经对启动时间做过优化，在 2.6 版本中对于Wal的 Load 速度就做过速度的优化，希望重启的时间不超过 1 分钟

Prometheus 提供了热加载能力，不过需要开启`web.enable-lifecycle`配置，更改完配置后，curl 下 reload 接口即可。prometheus-operator 中更改了配置会默认触发 reload，如果你没有使用 operator，又希望可以监听 configmap 配置变化来 reload 服务，可以试下这个简单的脚本

```
#!/bin/sh
FILE=$1
URL=$2
HASH=$(md5sum $(readlink -f $FILE))while true; do
   NEW_HASH=$(md5sum $(readlink -f $FILE))
   if [ "$HASH" != "$NEW_HASH" ]; then
     HASH="$NEW_HASH"
     echo "[$(date +%s)] Trigger refresh"
     curl -sSL -X POST "$2" > /dev/null
   fi
   sleep 5done
```

使用时和 prometheus 挂载同一个 configmap，传入如下参数即可：

```
args:
    - /etc/prometheus/prometheus.yml
    - http://prometheus.kube-system.svc.cluster.local:9090/-/reloadargs:
    - /etc/alertmanager/alertmanager.yml
    - http://prometheus.kube-system.svc.cluster.local:9093/-/reload
```

十七、你的应用需要暴露多少指标

当你开发自己的服务的时候，你可能会把一些数据暴露 Metric出去，比如特定请求数、Goroutine 数等，指标数量多少合适呢?

虽然指标数量和你的应用规模相关，但也有一些建议(Brian Brazil),

比如简单的服务如缓存等，类似 Pushgateway，大约 120 个指标，Prometheus 本身暴露了 700 左右的指标，如果你的应用很大，也尽量不要超过 10000 个指标，需要合理控制你的 Label。

十八、node-exporter 的问题

- node-exporter 不支持进程监控，这个前面已经提到了。
- node-exporter 只支持 unix 系统，windows机器 请使用 wmi_exporter。因此以 yaml 形式不是 node-exporter 的时候，node-selector 要表明os类型。
- 因为node_exporter是比较老的组件，有一些最佳实践并没有merge进去，比如符合Prometheus命名规范，因此建议使用较新的0.16和0.17版本。

一些指标名字的变化

```
* node_cpu ->  node_cpu_seconds_total
* node_memory_MemTotal -> node_memory_MemTotal_bytes
* node_memory_MemFree -> node_memory_MemFree_bytes
* node_filesystem_avail -> node_filesystem_avail_bytes
* node_filesystem_size -> node_filesystem_size_bytes
* node_disk_io_time_ms -> node_disk_io_time_seconds_total
* node_disk_reads_completed -> node_disk_reads_completed_total
* node_disk_sectors_written -> node_disk_written_bytes_total
* node_time -> node_time_seconds
* node_boot_time -> node_boot_time_seconds
* node_intr -> node_intr_total
```

如果你之前用的旧版本 exporter，在绘制 grafana 的时候指标名称就会有差别，解决方法有两种：

- 一是在机器上启动两个版本的node-exporter，都让prometheus去采集。
- 二是使用指标转换器,他会将旧指标名称转换为新指标

十九、kube-state-metric 的问题

kube-state-metric 的使用和原理可以先看下这篇

除了文章中提到的作用，kube-state-metric还有一个很重要的使用场景，就是和 cadvisor 指标组合，原始的 cadvisor 中只有 pod 信息，不知道属于哪个 deployment 或者 sts，但是和kube-state-metric 中的 kube_pod_info 做 join 查询之后就可以显示出来，kube-state-metric的元数据指标，在扩展 cadvisor 的 label 中起到了很多作用，prometheus-operator 的很多 record rule 就使用了 kube-state-metric 做组合查询。

kube-state-metric 中也可以展示 pod 的 label 信息，可以在拿到 cadvisor 数据后更方便地做 group by，如按照 pod 的运行环境分类。但是 kube-state-metric 不暴露 pod 的 annotation，原因是下面会提到的高基数问题，即 annotation 的内容太多，不适合作为指标暴露。

二十、relabel_configs 与 metric_relabel_configs

relabel_config 发生在采集之前，metric_relabel_configs 发生在采集之后，合理搭配可以满足很多场景的配置。

如：

```
metric_relabel_configs:
  - separator: ;
    regex: instance
    replacement: $1
    action: labeldrop
```

```
- source_labels: [__meta_kubernetes_namespace,
__meta_kubernetes_endpoints_name,
      __meta_kubernetes_service_annotation_prometheus_io_port]
    separator: ;
    regex: (.+);(.+);(.*)
    target_label: __metrics_path__
    replacement: /api/v1/namespaces/${1}/services/${2}:${3}/proxy/metrics
    action: replace
```

二十一、找到最大的 metric 或 job

top10的 metric 数量：按 metric 名字分

```
topk(10, count by (__name__)({__name__=~".+"}))

apiserver_request_latencies_bucket{}  62544
apiserver_response_sizes_bucket{}   44600
```

top10的 metric 数量：按 job 名字分

```
topk(10, count by (__name__, job)({__name__=~".+"}))

{job="master-scrape"}   525667
{job="xxx-kubernetes-cadvisor"}  50817
{job="yyy-kubernetes-cadvisor"}   44261
```

二十二、反直觉的 P95 统计

histogram_quantile 是 Prometheus 常用的一个函数，比如经常把某个服务的 P95 响应时间来衡量服务质量。不过它到底是什么意思很难解释得清，特别是面向非技术的同学，会遇到很多“灵魂拷问”。

我们常说 P95（P99,P90都可以） 响应延迟是 100ms，实际上是指对于收集到的所有响应延迟，有 5% 的请求大于 100ms，95% 的请求小于 100ms。Prometheus 里面的 histogram_quantile 函数接收的是 0-1 之间的小数，将这个小数乘以 100 就能很容易得到对应的百分位数，比如 0.95 就对应着 P95，而且还可以高于百分位数的精度，比如 0.9999。

当你用 histogram_quantile 画出响应时间的趋势图时，可能会被问：为什么P95大于或小于我的平均值?

正如中位数可能比平均数大也可能比平均数小，P99 比平均值小也是完全有可能的。通常情况下 P99 几乎总是比平均值要大的，但是如果数据分布比较极端，最大的 1% 可能大得离谱从而拉高了平均值。一种可能的例子：

```
1, 1, ... 1, 901 // 共 100 条数据，平均值=10，P99=1
```

服务 X 由顺序的 A，B 两个步骤完成，其中 X 的 P99 耗时 100Ms，A 过程 P99 耗时 50Ms，那么推测 B 过程的 P99 耗时情况是?

直觉上来看，因为有 X=A+B，所以答案可能是 50Ms，或者至少应该要小于 50Ms。实际上 B 是可以大于 50Ms 的，只要 A 和 B 最大的 1% 不恰好遇到，B 完全可以有很大的 P99：

```
A = 1, 1, ... 1,  1,  1,  50,  50 // 共 100 条数据，P99=50
B = 1, 1, ... 1,  1,  1,  99,  99 // 共 100 条数据，P99=99
X = 2, 2, ... 1, 51, 51, 100, 100 // 共 100 条数据，P99=100
```

```
如果让 A 过程最大的 1% 接近 100Ms，我们也能构造出 P99 很小的 B：
A = 50, 50, ... 50,  50,  99 // 共 100 条数据，P99=50
B =  1,  1, ...  1,   1,  50 // 共 100 条数据，P99=1
X = 51, 51, ... 51, 100, 100 // 共 100 条数据，P99=100
```

所以我们从题目唯一能确定的只有 B 的 P99 应该不能超过 100ms，A 的 P99 耗时 50Ms 这个条件其实没啥用。

类似的疑问很多，因此对于 histogram_quantile 函数，可能会产生反直觉的一些结果，最好的处理办法是不断试验调整你的 Bucket 的值，保证更多的请求时间落在更细致的区间内，这样的请求时间才有统计意义。

二十三、慢查询问题

Promql 的基础知识看这篇文章

Prometheus 提供了自定义的 Promql 作为查询语句，在 Graph 上调试的时候，会告诉你这条 Sql 的返回时间，如果太慢你就要注意了，可能是你的用法出现了问题。

评估 Prometheus 的整体响应时间，可以用这个默认指标:

```
prometheus_engine_query_duration_seconds{}
```

一般情况下响应过慢都是Promql 使用不当导致，或者指标规划有问题，如：

- 大量使用 join 来组合指标或者增加 label，如将 kube-state-metric 中的一些 meta label和 node-exporter 中的节点属性 label加入到 cadvisor容器数据里，像统计 pod 内存使用率并按照所属节点的机器类型分类，或按照所属 rs 归类。
- 范围查询时，大的时间范围 step 值却很小，导致查询到的数量过大。
- rate 会自动处理 counter 重置的问题，最好由 promql 完成，不要自己拿出来全部元数据在程序中自己做 rate 计算。
- 在使用 rate 时，range duration要大于等于step，否则会丢失部分数据
- prometheus 是有基本预测功能的，如`deriv`和`predict_linear`(更准确)可以根据已有数据预测未来趋势
- 如果比较复杂且耗时的sql，可以使用 record rule 减少指标数量，并使查询效率更高，但不要什么指标都加 record，一半以上的 metric 其实不太会查询到。同时 label 中的值不要加到 record rule 的 name 中。

二十四、高基数问题 Cardinality

高基数是数据库避不开的一个话题，对于 Mysql 这种 DB 来讲，基数是指特定列或字段中包含的唯一值的数量。基数越低，列中重复的元素越多。对于时序数据库而言，就是 tags、label 这种标签值的数量多少。

比如 Prometheus 中如果有一个指标 `http_request_count{method="get",path="/abc",originIP="1.1.1.1"}` 表示访问量，method 表示请求方法，originIP 是客户端 IP，method的枚举值是有限的，但 originIP 却是无限的，加上其他 label 的排列组合就无穷大了，也没有任何关联特征，因此这种高基数不适合作为 Metric 的 label，真要的提取originIP，应该用日志的方式，而不是 Metric 监控

时序数据库会为这些 Label 建立索引，以提高查询性能，以便您可以快速找到与所有指定标签匹配的值。如果值的数量过多，索引是没有意义的，尤其是做 P95 等计算的时候，要扫描大量 Series 数据

官方文档中对于Label 的建议

```
CAUTION: Remember that every unique combination of key-value label pairs
represents a new time series, which can dramatically increase the amount of data
stored. Do not use labels to store dimensions with high cardinality (many
different label values), such as user IDs, email addresses, or other unbounded
sets of values.
```

如何查看当前的Label 分布情况呢，可以使用 Prometheus提供的Tsdb工具。可以使用命令行查看，也可以在 2.16 版本以上的 Prometheus Graph 查看

```
[work@xxx bin]$ ./tsdb analyze ../data/prometheus/
Block ID: 01E41588AJNGM31SPGHYA3XSXG
Duration: 2h0m0s
Series: 955372
Label names: 301
Postings (unique label pairs): 30757
Postings entries (total label pairs): 10842822
....
```

TSDB Status

Head Cardinality Stats

Top 10 label names with value count

Name	Count
id	3612
resource_version	3539
name	3368
__name__	1826
container_id	1811
uid	1297
pod	1277
secret	1112
pod_name	1015
pod_ip	981

Top 10 series count by label value pairs

Name	Count
job=master-scrape	631242
beta_kubernetes_io_os=linux	316649
beta_kubernetes_io_arch=amd64	316639

top10 高基数的 metric

```
Highest cardinality metric names:87176 apiserver_request_latencies_bucket
59968 apiserver_response_sizes_bucket
39862 apiserver_request_duration_seconds_bucket
37555 container_tasks_state
....
```

高基数的 label

```
Highest cardinality labels:4271 resource_version
3670 id3414 name
1857 container_id
1824 __name__
1297 uid
1276 pod
...
```

二十五、Prometheus 的预测能力

场景1：你的磁盘剩余空间一直在减少，并且降低的速度比较均匀，你希望知道大概多久之后达到阈值，并希望在某一个时刻报警出来。

场景2：你的 Pod 内存使用率一直升高，你希望知道大概多久之后会到达 Limit 值，并在一定时刻报警出来，在被杀掉之前上去排查。

Prometheus 的 Deriv 和 Predict_Linear 方法可以满足这类需求， Promtheus 提供了基础的预测能力，基于当前的变化速度，推测一段时间后的值。

以 mem_free 为例，最近一小时的 free 值一直在下降。

`mem_free仅为举例，实际内存可用以mem_available为准`

deriv函数可以显示指标在一段时间的变化速度:

predict_linear方法是预测基于这种速度，最后可以达到的值:

```
predict_linear(mem_free{instanceIP="100.75.155.55"}[1h], 2*3600)/1024/1024
```

你可以基于设置合理的报警规则，如小于 10 时报警:

```
rule: predict_linear(mem_free{instanceIP="100.75.155.55"}[1h], 2*3600)/1024/1024
<10
```

predict_linear 与 deriv 的关系: 含义上约等于如下表达式，不过 predict_linear 稍微准确一些。

```
  deriv(mem_free{instanceIP="100.75.155.55"}[1h]) * 2 * 3600+
  mem_free{instanceIP="100.75.155.55"}[1h]
```

如果你要基于 Metric做模型预测，可以参考下forecast-prometheus

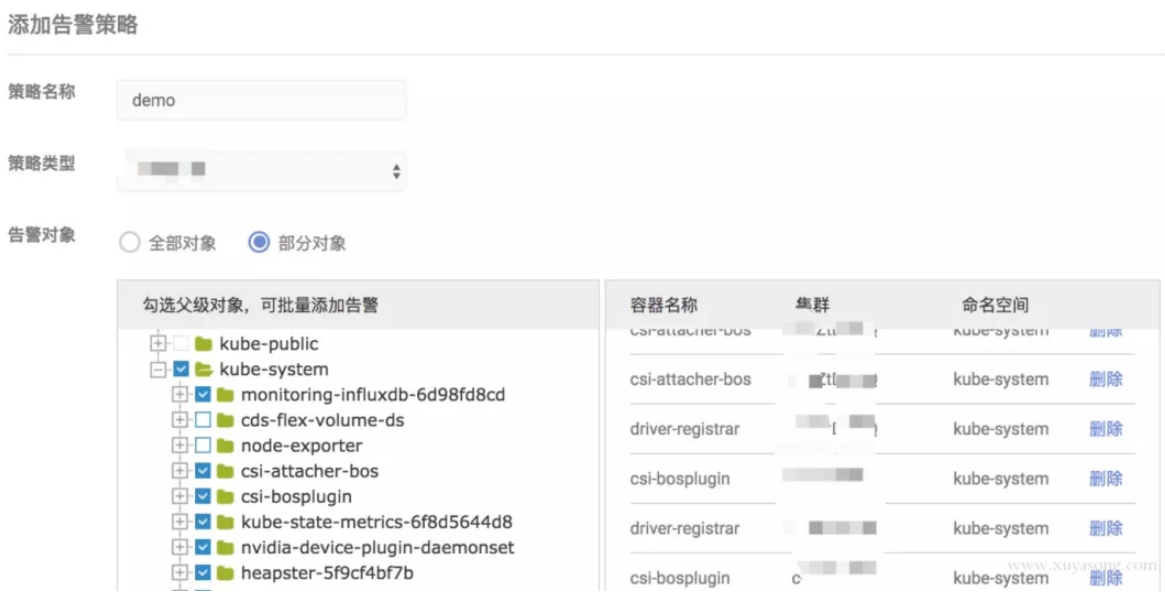
二十六、alertmanager 的上层封装

Prometheus 部署之后很少会改动，尤其是做了服务发现，就不需要频繁新增 target。但报警的配置是很频繁的，如修改阈值、修改报警人等。alertmanager 拥有丰富的报警能力如分组、抑制等，但如果你要想把他给业务部门使用，就要做一层封装了，也就是报警配置台。用户喜欢表单操作，而非晦涩的 yaml，同时他们也并不愿意去理解 promql。而且大多数公司内已经有现成的监控平台，也只有一份短信或邮件网关，所以最好能使用 webhook 直接集成。

例如: 机器磁盘使用量超过 90% 就报警，rule 应该写为：`disk_used/disk_total > 0.9`

如果不加 label 筛选，这条报警会对所有机器生效，但如果你想去掉其中几台机器，就得在disk_used和disk_total后面加上{instance != “”}。这些操作在 promql 中是很简单的，但是如果放在表单里操作，就得和内部的 cmdb 做联动筛选了。

- 对于一些简单的需求，我们使用了 Grafana 的报警能力，所见即所得，直接在图表下面配置告警即可，报警阈值和状态很清晰。不过 Grafana 的报警能力很弱，只是实验功能，可以作为调试使用。
- 对于常见的 pod 或应用监控，我们做了一些表单化，如下图所示：提取了 CPU、内存、磁盘 IO 等常见的指标作为选择项，方便配置。



- 使用 webhook 扩展报警能力，改造 alertmanager， 在 send message 时做加密和认证，对接内部已有报警能力，并联动用户体系，做限流和权限控制。
- 调用 alertmanager api 查询报警事件，进行展示和统计。

对于用户来说，封装 alertmanager yaml 会变的易用，但也会限制其能力，在增加报警配置时，研发和运维需要有一定的配合。如新写了一份自定义的 exporter，要将需要的指标供用户选择，并调整好展示和报警用的 promql。还有报警模板、原生 promql 暴露、用户分组等，需要视用户需求做权衡。

二十七、错误的高可用设计

有些人提出过这种类型的方案，想提高其扩展性和可用性。

应用程序将 Metric 推到到消息队列如 Kafaka，然后经过 Exposer 消费中转，再被 Prometheus 拉取。产生这种方案的原因一般是有历史包袱、复用现有组件、想通过 Mq 来提高扩展性。

这种方案有几个问题：

- 增加了 Queue 组件，多了一层依赖，如果 App与 Queue 之间连接失败，难道要在 App 本地缓存监控数据?
- 抓取时间可能会不同步，延迟的数据将会被标记为陈旧数据，当然你可以通过添加时间戳来标识，但就失去了对陈旧数据的处理逻辑
- 扩展性问题：Prometheus 适合大量小目标，而不是一个大目标，如果你把所有数据都放在了 Exposer 中，那么 Prometheus 的单个 Job 拉取就会成为 CPU 瓶颈。这个和 Pushgateway 有些类似，没有特别必要的场景，都不是官方建议的方式。
- 缺少了服务发现和拉取控制，Prom 只知道一个 Exposer，不知道具体是哪些 Target，不知道他们的 UP 时间，无法使用 Scrape_* 等指标做查询，也无法用scrape_limit做限制。

如果你的架构和 Prometheus 的设计理念相悖，可能要重新设计一下方案了，否则扩展性和可靠性反而会降低。

二十八、prometheus-operator 的场景

如果你是在 K8S 集群内部署 Prometheus，那大概率会用到 prometheus-operator，他对 Prometheus 的配置做了 CRD 封装，让用户更方便的扩展 Prometheus实例，同时 prometheus-operator 还提供了丰富的 Grafana 模板，包括上面提到的 master 组件监控的 Grafana 视图，operator 启动之后就可以直接使用，免去了配置面板的烦恼。

operator 的优点很多，就不一一列举了，只提一下 operator 的局限：

- 因为是 operator，所以依赖 K8S 集群，如果你需要二进制部署你的 Prometheus，如集群外部署，就很难用上prometheus-operator了，如多集群场景。当然你也可以在 K8S 集群中部署 operator 去监控其他的 K8S 集群，但这里面坑不少，需要修改一些配置。
- operator 屏蔽了太多细节，这个对用户是好事，但对于理解 Prometheus 架构就有些 gap 了，比如碰到一些用户一键安装了operator，但 Grafana 图表异常后完全不知道如何排查，record rule 和 服务发现还不了解的情况下就直接配置，建议在使用 operator 之前，最好熟悉 prometheus 的基础用法。
- operator 方便了 Prometheus 的扩展和配置，对于 alertmanager 和 exporter 可以很方便的做到多实例高可用，但是没有解决 Prometheus 的高可用问题，因为无法处理数据不一致，operator 目前的定位也还不是这个方向，和 Thanos、Cortex 等方案的定位是不同的，下面会详细解释。

二十九、高可用方案

Prometheus 高可用有几种方案：

1. 基本 HA：即两套 Prometheus 采集完全一样的数据，外边挂负载均衡
2. HA + 远程存储：除了基础的多副本 Prometheus，还通过 Remote Write 写入到远程存储，解决存储持久化问题
3. 联邦集群：即 Federation，按照功能进行分区，不同的 Shard 采集不同的数据，由Global节点来统一存放，解决监控数据规模的问题。
4. 使用 Thanos 或者 Victoriametrics，来解决全局查询、多副本数据 Join 问题。

就算使用官方建议的多副本 + 联邦，仍然会遇到一些问题:

1. 官方建议数据做 Shard，然后通过Federation来实现高可用，
2. 但是边缘节点和Global节点依然是单点，需要自行决定是否每一层都要使用双节点重复采集进行保活。
   也就是仍然会有单机瓶颈。
3. 另外部分敏感报警尽量不要通过Global节点触发，毕竟从Shard节点到Global节点传输链路的稳定性会影响数据到达的效率，进而导致报警实效降低。
4. 例如服务Updown状态，Api请求异常这类报警我们都放在Shard节点进行报警。

本质原因是，Prometheus 的本地存储没有数据同步能力，要在保证可用性的前提下，再保持数据一致性是比较困难的，基础的 HA Proxy 满足不了要求，比如：

- 集群的后端有 A 和 B 两个实例，A 和 B 之间没有数据同步。A 宕机一段时间，丢失了一部分数据，如果负载均衡正常轮询，请求打到A 上时，数据就会异常。
- 如果 A 和 B 的启动时间不同，时钟不同，那么采集同样的数据时间戳也不同，就不是多副本同样数据的概念了
- 就算用了远程存储，A 和 B 不能推送到同一个 TSDB，如果每人推送自己的 TSDB，数据查询走哪边就是问题了。

因此解决方案是在存储、查询两个角度上保证数据的一致:

- 存储角度：如果使用 Remote Write 远程存储， A 和 B后面可以都加一个 Adapter，Adapter做选主逻辑，只有一份数据能推送到 TSDB，这样可以保证一个异常，另一个也能推送成功，数据不丢，同时远程存储只有一份，是共享数据。方案可以参考这篇文章
- 查询角度：上边的方案实现很复杂且有一定风险，因此现在的大多数方案在查询层面做文章，比如 Thanos 或者 Victoriametrics，仍然是两份数据，但是查询时做数据去重和 Join。只是 Thanos 是通过 Sidecar 把数据放在对象存储，Victoriametrics 是把数据 Remote Write 到自己的 Server 实例，但查询层 Thanos-Query 和 Victor 的 Promxy的逻辑基本一致。

我们采用了 Thanos 来支持多地域监控数据，具体方案可以看这篇文章

三十、容器日志与事件

本文主要是 Prometheus 监控内容, 这里只简单介绍下 K8S 中的日志、事件处理方案，以及和 Prometheus 的搭配。

日志处理：

- 日志采集与推送：一般是Fluentd/Fluent-Bit/Filebeat等采集推送到 ES、对象存储、kafaka，日志就该交给专业的 EFK 来做，分为容器标准输出、容器内日志。
- 日志解析转 metric：可以提取一些日志转为 Prometheus 格式的指标，如解析特定字符串出现次数，解析 Nginx 日志得到 QPS 、请求延迟等。常用方案是 mtail 或者 grok

日志采集方案：

- sidecar 方式：和业务容器共享日志目录，由 sidecar 完成日志推送，一般用于多租户场景。
- daemonset 方式：机器上运行采集进程，统一推送出去。

需要注意的点：对于容器标准输出，默认日志路径是`/var/lib/docker/containers/xxx`， kubelet 会将改日志软链到`/var/log/pods`，同时还有一份`/var/log/containers`是对`/var/log/pods`的软链。不过不同的 K8S 版本，日志的目录格式有所变化，采集时根据版本做区分：

- 1.15 及以下：/var/log/pods/{pod_uid}/
- 1.15 以上：var/log/pods/{pod_name+namespace+rs+uuid}/

事件：在这里特指 K8S Events，Events 在排查集群问题时也很关键，不过默认情况下只保留 1h，因此需要对 Events 做持久化。一般 Events 处理方式有两种：

- 使用 kube-eventer 之类的组件采集 Events 并推送到 ES
- 使用 event_exporter 之类的组件将Events 转化为 Prometheus Metric，同类型的还有谷歌云的 stackdriver 下的 event-exporter

作者：徐亚松

来源：http://www.xuyasong.com