云原生监控体系建设

秦晓辉　快猫星云 联合创始人

个人介绍

秦晓辉，常用网名龙渊秦五、UlricQin，山东人，12年毕业自山东大学，10年经验一直是在运维研发相关方向，是Open-Falcon、Nightingale、Categraf 等开源软件的核心研发，快猫星云联合创始人，当前在创业，为客户提供稳定性保障相关的产品

个人主页：https://ulricqin.github.io/

快猫星云

大纲

- 云原生之后监控需求的变化
- 从Kubernetes架构来看要监控的组件
- Kubernetes所在宿主的监控
- Kubernetes Node组件监控
- Kubernetes控制面组件监控
- Kubernetes资源对象的监控
- Pod内的业务应用的监控
- 业务应用依赖的中间件的监控

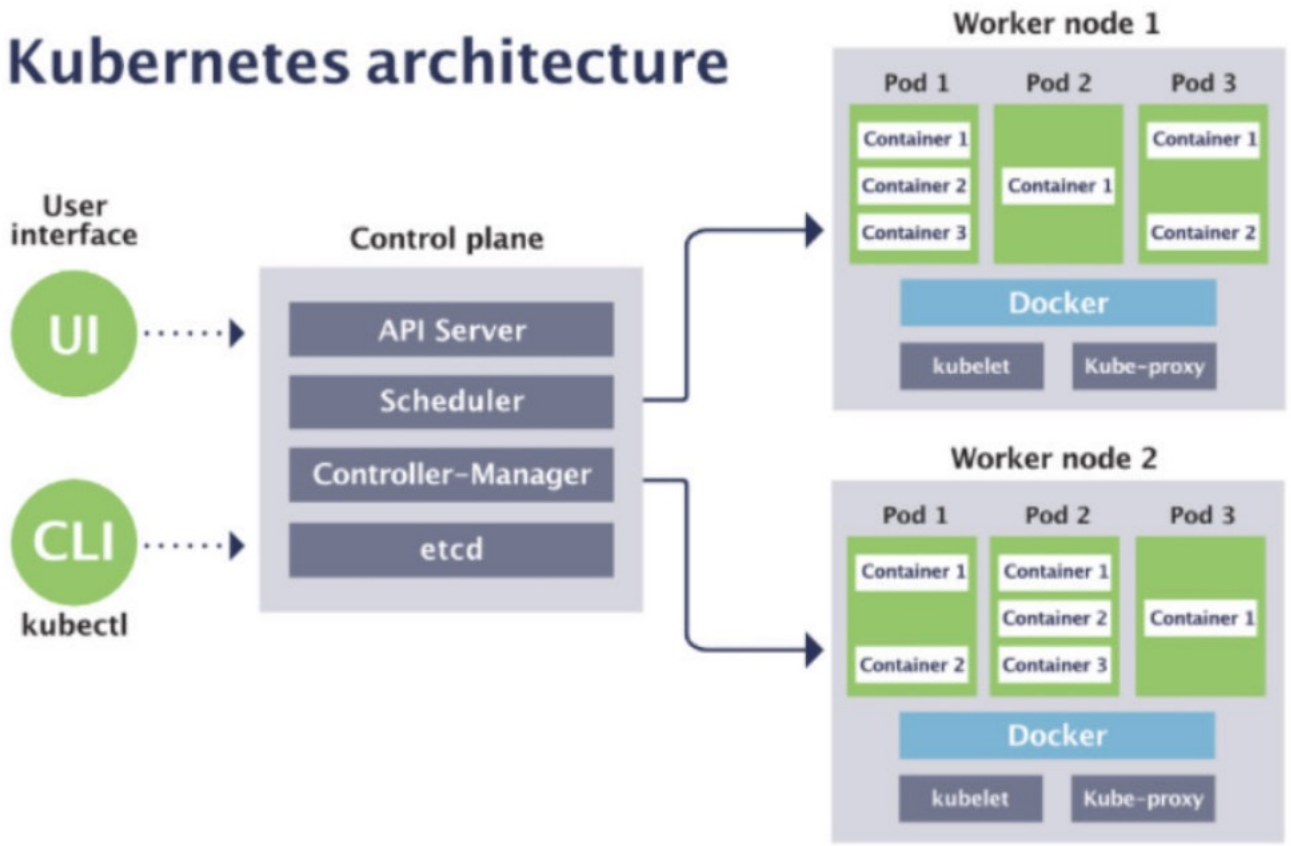
云原生之后监控需求的变化

云原生之后监控需求的变化

指标生命周期变短	•相比物理机虚拟机时代，基础设施动态化，Pod销毁重建非常频繁 •原来使用资产视角管理监控对象的系统不再适用 •要么使用注册中心来自动发现，要么就是采集器和被监控对象通过sidecar模式捆绑一体
指标数量大幅增长	•微服务的流行，要监控的服务数量大幅增长，是之前的指标数量十倍都不止 •广大研发工程师也更加重视可观测能力的建设，更愿意埋点 •各种采集器层出不穷，都是本着可采尽采的原则，一个中间件实例动辄采集几千个指标
指标维度更为丰富	•老一代监控系统更多的是关注机器、交换机、中间件的监控，每个监控对象一个标识即可，没有维度的设计 •新一代监控系统更加关注应用侧的监控，没有维度标签玩不转，每个指标动辄几个、十几个标签
平台侧自身复杂度变高，监控难度加大	•Kubernetes体系庞大，组件众多，涉及underlay、overlay两层网络，容器内容器外两个namespace，搞懂需要花些时间 •Kubernetes的监控，缺少体系化的文档指导，关键指标是哪些？最佳实践是什么？不是随便搜索几个yaml文件能搞定的

从 Kubernetes 架构来看要监控的组件

Kubernetes架构



- 服务端组件，控制面：API Server、Scheduler、Controller-Manager、ETCD
- 工作负载节点，最核心就是监控Pod容器和节点本身，也要关注 kubelet 和 kube-proxy
- 业务程序，即部署在容器中的业务程序的监控，这个其实是最重要的

随着 Kubernetes 越来越流行，几乎所有云厂商都提供了托管服务，这就意味着，服务端组件的可用性保障交给云厂商来做了，客户主要关注工作负载节点的监控即可。如果公司上云了，建议采用这种托管方式，不要自行搭建 Kubernetes，毕竟，复杂度真的很高，特别是后面还要涉及到升级维护的问题。既然负载节点更重要，我们讲解监控就从工作负载节点开始。

Kubernetes 所在宿主的监控

Kubernetes所在宿主的监控

宿主的监控，比较常规和简单，无非就是 CPU、Mem、Disk、DiskIO、Net、Netstat、Processes、System、Conntrack、Vmstat 等等。原理就是读取 OS 的数据（通过 /proc 和 syscall 等）做一些简单计算。有很多采集器可以选择：

Telegraf

Grafana-agent

Datadog-agent

node-exporter

Categraf

Kubernetes Node 组件的监控

Kubernetes Node - 容器负载监控 抓取方案

- Pod或者容器的负载情况，是一个需要关注的点，容器层面主要关注CPU和内存使用情况，Pod 层面主要关注网络IO的情况，因为多个容器共享Pod的net namespace，Pod内多个容器的网络数据相同
- 容器的监控数据可以直接通过 docker 引擎的接口读取到，也可以直接读取 cAdvisor 的接口，Kubelet 里内置了cAdvisor，cAdvisor 不管是 docker 还是 containerd 都可以采集到，推荐

```yaml
scrape_configs:
  - job_name: kubernetes-nodes-cadvisor
    scrape_interval: 10s
    scrape_timeout: 10s
    scheme: https  # remove if you want to scrape metrics on insecure port
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
      - role: node
    relabel_configs:
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
      # Only for Kubernetes ^1.7.3.
      # See: https://github.com/prometheus/prometheus/issues/2916
      - target_label: __address__
        replacement: kubernetes.default.svc:443
      - source_labels: [__meta_kubernetes_node_name]
        regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
```

{ 抓取方案一 }

- 左侧这个配置大家在网上比较容易搜到，通过kubernetes_sd_configs做服务发现，查找所有node，通过 Kubernetes apiserver 的 proxy 接口，抓取各个node（即kubelet）的 /metrics/cadvisor 接口的 prometheus 协议的数据
- 这个抓取器只需要部署一个实例，调用 apiserver 的接口即可，维护较为简单，采集频率可以调的稍大，比如30s或60s
- 所有的拉取请求都走 apiserver，如果是几千个node的大集群，对 apiserver 可能会有较大压力

Kubernetes Node - 容器负载监控 抓取方案

```
interval = 15

[[instances]]
urls = [
    "https://127.0.0.1:10250/metrics/cadvisor"
]

bearer_token_file = "/var/run/secrets/kubernetes.io/serviceaccount/token"

url_label_key = "instance"
url_label_value = "{{.Host}}"

labels = {instance="-"}

use_tls = true
insecure_skip_verify = true
```

完整配置可以参考：

https://github.com/flashcatcloud/categraf/blob/main/k8s/daemonset.yaml

{ 抓取方案二 }

- 直接调用 kubelet 的接口 /metrics/cadvisor，不走 apiserver 这个 proxy，避免对 apiserver 的请求压力
- 采用 Daemonset 的方式部署
- 但是，缺少了对 __meta_kubernetes_node_label_(.+) 的 labelmap，即：通过这种方式采集的数据，我们无法得到 node 上的 label 数据。node 上的 label 数据最核心的就是标识了 node 的 IP
- 我们可以通过环境变量为抓取器注入NODEIP，比如用 Categraf 的话，就会自动把本机 IP 作为标签带上去，设置 config.toml 中的 hostname="$ip" 即可

Kubernetes Node - 容器负载监控 关键指标

CPU使用率，分子是每秒内容器用了多少CPU时间，分母是每秒内被限制使用多少CPU时间

```
sum(
irate(container_cpu_usage_seconds_total[3m])
) by (pod,id,namespace,container,ident,image)
/
sum(
container_spec_cpu_quota/container_spec_cpu_period
) by (pod,id,namespace,container,ident,image)
```

内存使用量除以内存限制量，就是使用率，但是后面跟了 and container_spec_memory_limit_bytes != 0 是因为有些容器没有配置 limit 的内存大小

```
container_memory_usage_bytes
/
container_spec_memory_limit_bytes
and
container_spec_memory_limit_bytes != 0
```

CPU被限制的时间比例

```
increase(container_cpu_cfs_throttled_periods_total[1m])
/
increase(container_cpu_cfs_periods_total[1m]) * 100
```

Pod网络出入向流量

```
irate(container_network_transmit_bytes_total[1m]) * 8
irate(container_network_receive_bytes_total[1m]) * 8
```

Pod硬盘IO读写流量

```
irate(container_fs_reads_bytes_total[1m])
irate(container_fs_writes_bytes_total[1m])
```

Kubernetes Node - Kubelet监控

- kubelet_running_pods：运行的Pod的数量，gauge类型
- kubelet_running_containers：运行的容器的数量，gauge类型，container_state标签来区分容器状态
- volume_manager_total_volumes：volume的数量，gauge类型，state标签用于区分是actual还是desired
- kubelet_runtime_operations_total：通过kubelet执行的各类操作的数量，counter类型
- kubelet_runtime_operations_errors_total：这个指标很关键，通过kubelet执行的操作失败的次数，counter类型
- kubelet_pod_start_duration_seconds*：histogram类型，描述pod从pending状态进入running状态花费的时间
- go_goroutines：kubelet的goroutine的数量
- kubelet_pleg_relist_duration_seconds*：histogram类型，pleg是Pod Lifecycle Event Generator，
- 如果这个时间花费太大，会对 Kubernetes中的pod状态造成影响
- kubelet_pleg_relist_interval_seconds*：histogram类型，relist的频率间隔

- Kubelet 在 /metrics/cadvisor 暴露的是 cAdvisor 的监控数据（prometheus协议），在 /stats/summary 暴露的是容器的概要监控数据（普通json协议），在 /metrics 暴露的是自身的监控数据（prometheus协议）
- **Kubelet 的核心职责就是管理本机的 Pod 和容器，典型的比如创建 Pod、销毁 Pod，显然我们应该关注这些操作的成功率和耗时**
- Categraf 的仓库中 inputs/kubernetes/kubelet-metrics-dash.json 就是 Kubelet 的大盘文件

Kubernetes Node – kube-proxy 监控

- up

关注 kube-proxy 的存活性，应该和 node 节点的数量相等

- rest_client_request_duration_seconds

针对 apiserver 的请求延迟的指标

- rest_client_requests_total

针对 apiserver 的请求量的指标

- kubeproxy_sync_proxy_rules_duration_seconds

同步网络规则的延迟指标

以及通用的进程相关的指标，进程的 CPU 内存 文件句柄等指标

- kube-proxy 在 /metrics 暴露监控数据，可以直接拉取
- kube-proxy 的核心职责是同步网络规则，修改 iptables 或者 ipvs。**所以要重点关注 sync_proxy_rules 相关的指标**
- Categraf 的仓库中 inputs/kubernetes/kube-proxy-dash.json 就是 kube-proxy 的大盘文件

Kubernetes 控制面组件的监控

Kubernetes控制面 apiserver的监控

- apiserver_request_total
请求量的指标，可以统计每秒请求数、成功率

- apiserver_request_duration_seconds
请求延迟统计

- process_cpu_seconds_total
进程使用的CPU时间

- process_resident_memory_bytes
进程的内存使用量

- apiserver 通过 /metrics 接口暴露监控数据，直接拉取即可
- apiserver 在 Kubernetes 架构中，是负责各种 API 调用的总入口，**重点关注的是吞吐、延迟、错误率这些黄金指标**
- apiserver 也会缓存很多数据到内存里，所以进程占用的内存，所在机器的内存使用率都应该要关注
- 采集方式可以参考 categraf 仓库的 k8s/deployment.yaml，大盘可以参考 k8s/apiserver-dash.json

Kubernetes控制面 controller-manager的监控

- rest_client_request_duration_seconds
请求 apiserver 的耗时分布，histogram类型，按照 url + verb 统计

- workqueue_adds_total
各个 controller 已处理的任务总数

- workqueue_depth
各个 controller 的队列深度，表示一个 controller 中的任务的数量，值越大表示越繁忙

- process_cpu_seconds_total
进程使用的CPU时间的总量，rate 之后就是 CPU 使用率

- controller-manager 通过 /metrics 接口暴露监控数据，直接拉取即可
- controller-manager 在 Kubernetes 架构中，是负责监听对象状态，并与期望状态做对比，如果状态不一致则进行调谐，**重点关注的是各个controller的运行情况，比如任务数量，队列深度**
- controller-manager出问题的概率相对较小，进程层面没问题大概率就没问题
- 采集方式可以参考 categraf 仓库的 k8s/deployment.yaml，大盘可以参考 k8s/cm-dash.json

Kubernetes控制面 scheduler的监控

- rest_client_request_duration_seconds

请求 apiserver 的耗时分布，histogram类型，按照 url + verb 统计

- scheduler_framework_extension_point_duration_seconds

调度框架的扩展点延迟分布，按 extension_point 统计

- scheduler_pending_pods

调度 pending 的 pod 数量，按照 queue type 分别统计

- scheduler_schedule_attempts_total

按照调度结果统计的调度重试次数。“unschedulable”表示无法调度，“error”表示调度器内部错误

- scheduler 通过 /metrics 接口暴露监控数据，直接拉取即可
- scheduler 在 Kubernetes 架构中，是负责调度对象到合适的node上，会有一系列的规则计算和筛选。**重点关注调度这个动作的相关指标**
- 采集方式可以参考 categraf 仓库的 k8s/deployment.yaml，大盘可以参考 k8s/scheduler-dash.json

Kubernetes控制面 etcd 的监控

- etcd_server_has_leader

etcd 是否有 leader

- etcd_server_leader_changes_seen_total

偶尔切主问题不大，频繁切主就要关注了

- etcd_server_proposals_failed_total

提案失败次数

- etcd_disk_backend_commit_duration_seconds
- etcd_disk_wal_fsync_duration_seconds

etcd 强依赖硬盘做数据持久化，一定要用IO性能高的盘，要特别关注硬盘相关的写入指标

- ETCD 也是直接暴露 /metrics 接口，可以直接抓取
- ETCD 可以考虑使用 sidecar 模式来抓，对于运维比较简单一些，不需要先部署 ETCD 再去配置抓取规则
- **ETCD 强依赖硬盘做持久化，所以要特别关注硬盘相关的指标，尽量用 SSD 或 NVME 的盘**
- 采集方式可以参考 categraf 仓库的 k8s/deployment.yaml，如果是 sidecar 模式，就直接使用 categraf prometheus 插件即可，大盘可以参考 k8s/etcd-dash.json

Kubernetes 资源对象的监控

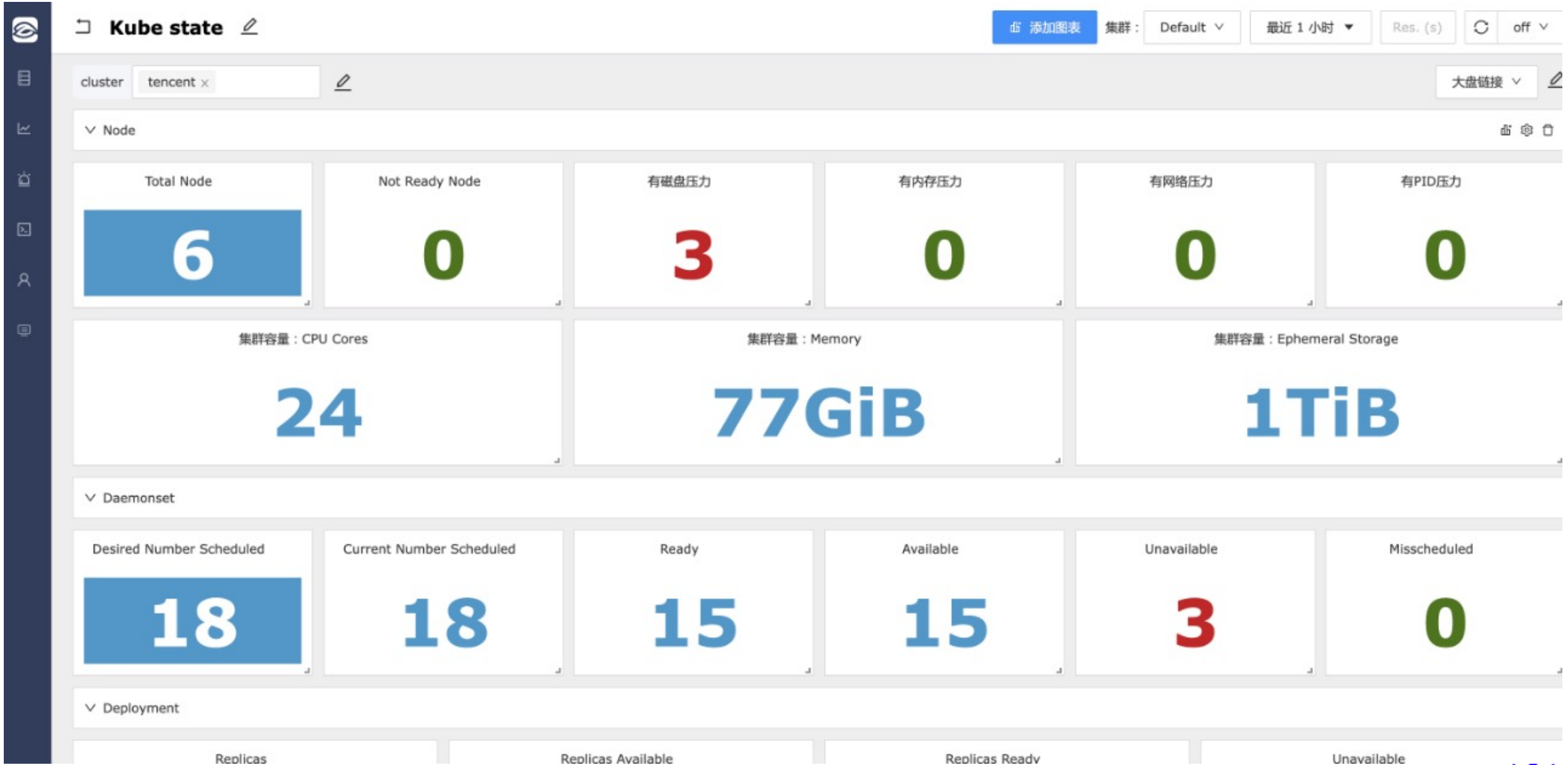
Kubernetes资源对象的监控 kube-state-metrics

- 所谓的资源对象，就是指Deployment、StatefulSet、Secret等，比如针对Deployment，我们希望知道有多少Deployment，调度了多少副本，可用的副本有多少，多少个Pod是running、stopped、terminated状态，等等
- 资源对象的监控使用 kube-state-metrics，这个开源项目是基于 client-go 开发，轮询 Kubernetes API，并将 Kubernetes 的结构化信息转换为 metrics
- 支持右侧罗列的相关资源对象的指标
- 比如Pod的指标，会有 info、owner、status_phase、status_ready、status_scheduled、container_status_waiting、container_status_terminated_reason 等指标

- CronJob Metrics
- DaemonSet Metrics
- Deployment Metrics
- Job Metrics
- LimitRange Metrics
- Node Metrics
- PersistentVolume Metrics
- PersistentVolumeClaim Metrics
- Pod Metrics
- Pod Disruption Budget Metrics
- ReplicaSet Metrics
- ReplicationController Metrics
- ResourceQuota Metrics
- Service Metrics
- StatefulSet Metrics
- Namespace Metrics
- Horizontal Pod Autoscaler Metrics
- Endpoint Metrics
- Secret Metrics
- ConfigMap Metrics

Kubernetes资源对象的监控 ksm部署

- ksm的镜像：k8s.gcr.io/kube-state-metrics/kube-state-metrics
- kube-state-metrics 也是部署到 Kubernetes 集群中即可，通常作为一个单副本的 Deployment
- 可以为 ksm 分配一个 ClusterIP，当做一个普通服务配置静态目标地址即可，也可以不分配，不分配就是用 PodIP 采集，容器迁移 PodIP 会变，所以只能走 kubernetes 的服务发现机制
- ksm 采集的监控指标数据量很大，请求其 /metrics 接口可能要拉取十几秒甚至几十秒，对于一些不关注的资源，我们可以不采集，典型的手段是通过 –resources 参数来控制，比如 –resources=deamonsets,deployments
- 对于某个具体的资源类型，可以做更细粒度的控制，比如屏蔽某个指标：--metric-denylist=kube_deployment_spec_.* 支持正则的写法，对于几千个node的大集群，几十万个 pod，每个小小的优化都很值得
- 最后，附赠大家一个我们做好的监控大盘，在 Categraf 仓库的：inputs/kube_state_metrics 目录下


Kube state
添加图表
集群：
Default
最近 1 小时
Res. (s)
off
cluster
tencent
大盘链接
Node
Total Node
6
Not Ready Node
0
有磁盘压力
3
有内存压力
0
有网络压力
0
有PID压力
0
集群容量：CPU Cores
24
集群容量：Memory
77GiB
集群容量：Ephemeral Storage
1TiB
Daemonset
Desired Number Scheduled
18
Current Number Scheduled
18
Ready
15
Available
15
Unavailable
3
Misscheduled
0
Deployment
Replicas
Replicas Available
Replicas Ready
Unavailable

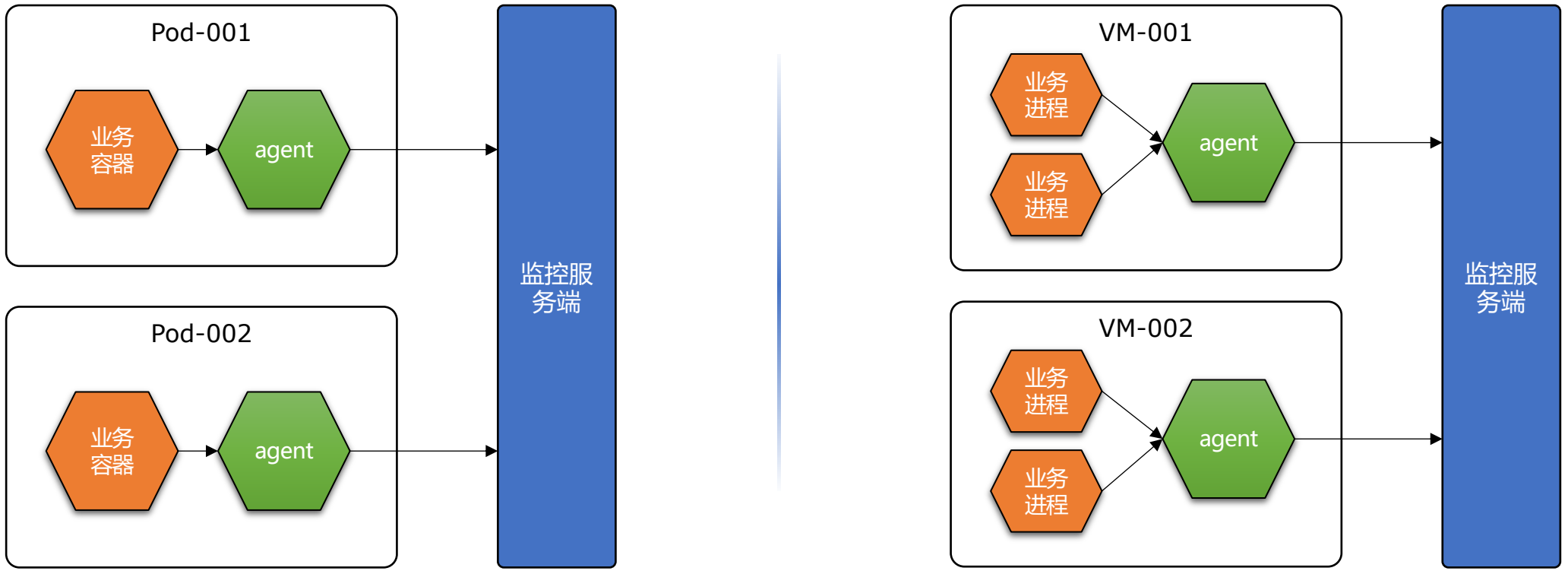
Kubernetes Pod 内的业务应用的监控

Pod内的业务应用的监控 - 两种埋点方式

- Pod 内的业务应用，有两种典型的埋点方案，statsd 和 prometheus sdk，当然，也可以用日志的方式，但是成本比价高，处理起来比较麻烦，如果业务程序是自己研发团队写的，可控，尽量就别用日志来暴露监控指标
- statsd 出现的时间比较久了，各个语言都有 sdk，很完善，业务程序内嵌 statsd 的 sdk，截获请求之后通过 UDP 推送给兼容 statsd 协议的 agent（比如telegraf、datadog-agent），这些 agent 在内存里做指标计算聚合，然后把结果数据推给服务端。因为是 UDP 协议，fire-and-forget，即使 agent 挂了，对业务也没啥影响
- prometheus sdk 作为另一种埋点方式，聚合计算逻辑是在 sdk 里完成，即在业务进程的内存里完成，对此介意者慎用，然后把指标通过 /metrics 接口暴露，交由监控系统来抓取
- statsd 和 prometheus sdk 都是通用方案，此外，不同语言也会有一些习惯性使用的埋点方案，比如 Java 的 micrometer
- 埋点方案尽量要全公司一套，规范统一，在代码框架层面内置，减轻各个研发团队的使用成本

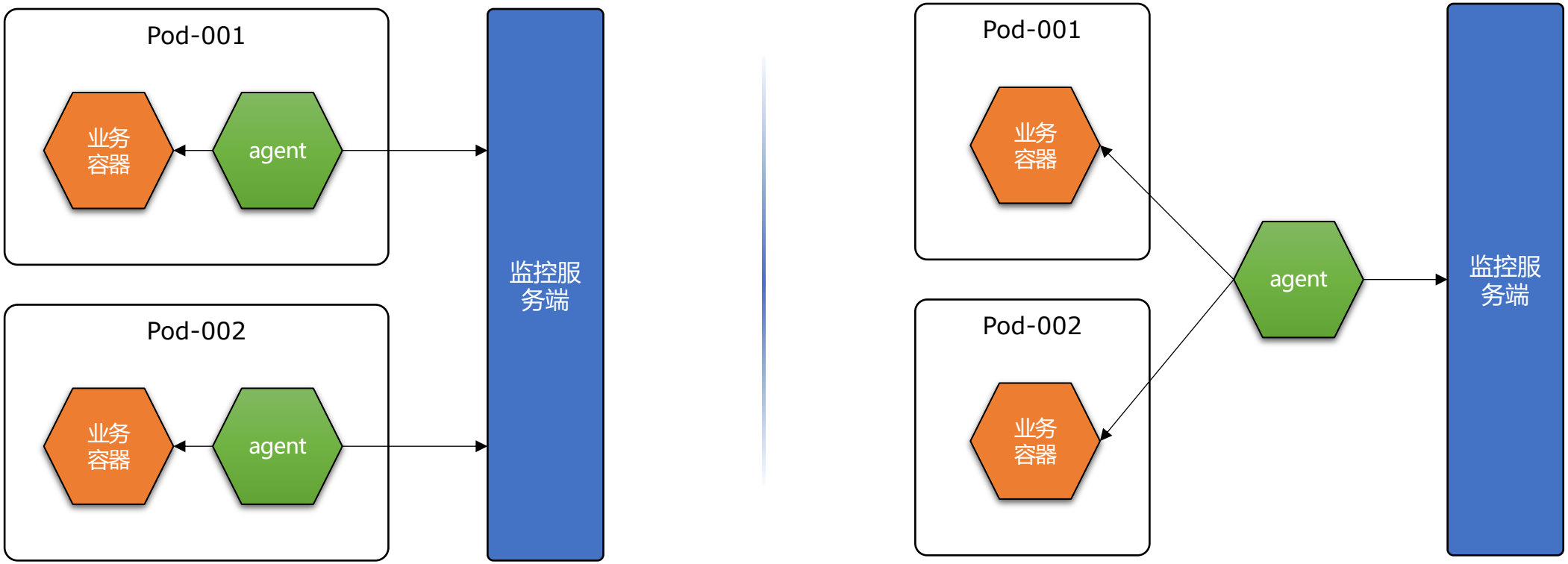
Pod内的业务应用的监控 - statsd 数据流向

- 推荐做法：如果是容器环境，Pod 内 sidecar 的方式部署 statsd；如果是物理机虚拟机环境，每个机器上部署一个 statsd 的 agent，接收到数据之后统一推给服务端



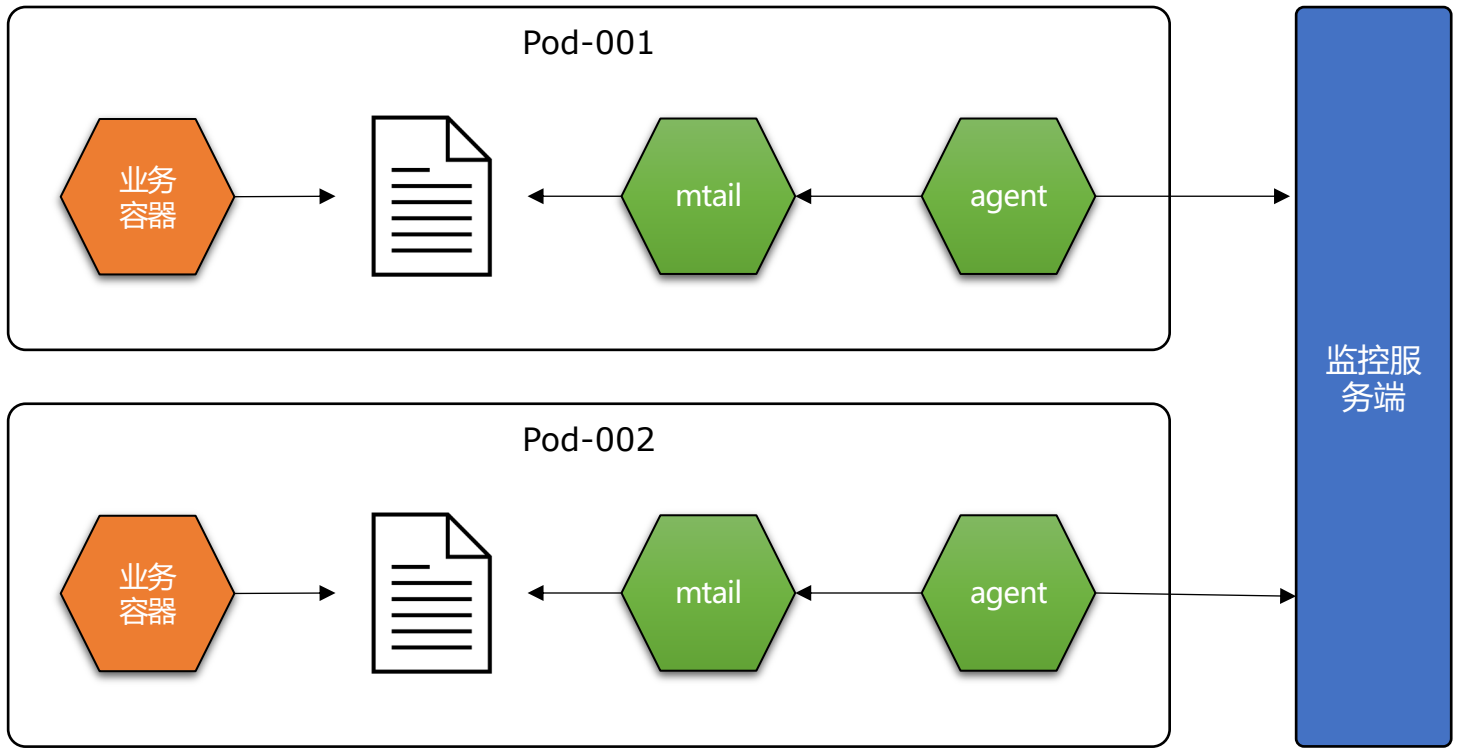
Pod内的业务应用的监控 – prom sdk 数据流向

- /metrics 接口的抓取，对于大规模集群可以考虑 sidecar 模式，自闭环更灵活，可以自定义认证、过滤规则；对于小集群，可以直接使用 Kubernetes 服务发现机制，用一个抓取器来抓



Pod内的业务应用的监控 - 日志转metrics数据流向

- 如果实在没办法埋点，通过 mtail 解析日志从中提取 metrics 也是一种方案，categraf 后面计划把 mtail 直接集成进来，这样就可以省去一个二进制



- 指标数据是性价比最高的数据类型，传输存储成本相对较低
- 日志的处理和存储成本最高，能用指标解决的尽量就用指标解决，不要用日志
- 如果是从第三方采购的产品，我们也尽量要求供应商统一暴露 prometheus 接口，也别去处理日志

业务应用依赖的中间件的监控

业务应用依赖的中间件的监控

- 典型的监控方案分3类，一类是 sidecar 方式，一类是动态改配置，最后一类是中心端统一采集
- sidecar 方式：中间件部署在容器里，比如 zookeeper 或 rabbitmq，直接暴露了 /metrics 接口，可以做一个 sidecar 模式的抓取器，与中间件一起部署、一起升级、一起下线销毁
- 动态改配置：比如中间件部署在物理机上，部署中间件的脚本，顺便创建对应的采集配置，然后对采集器 reload，下线中间件的时候，就是删除对应的采集配置，对采集器 reload
- 中心端统一采集：不同的中间件，可以分别使用不同的采集器实例（相当于根据中间件类型做抓取器的分片），每次部署了一个新的中间件实例，就来这个中心配置的地方，增加一条新的采集规则，或者使用服务发现的方式，把中间件实例注册到注册中心，由抓取器统一去注册中心拉取实例列表。这就要求，各个实例的认证信息都得一致，没有个性化配置，要不然处理起来就略麻烦了
- 中间件实例的监控数据采集其实还是次要的，关键是采集哪些指标，哪些指标要配置告警哪些要配置大盘，需要一个最佳实践，这其实就是 https://github.com/flashcatcloud/categraf 的初衷

Thank You