# Kubernetes集群监控

## 概述

本文描述了Kubernetes集群的监控方案，本文主要描述了以下内容：

1. 整体架构，Prometheus与数据平台的关系。
2. Kubernetes集群内Prometheus的部署和容量规划。
3. 告警方案
4. 接口和数据展示
5. 自定义exporter

适合对象：Kubernetes集群管理员，数据平台管理员，应用部署人员，应用开发人员(内置metrics接口)。

## 名词解释

|  |  |
| --- | --- |
| 名词 | 说明 |
| Kubernetes | 用于自动部署，扩展和管理容器化应用程序的开源系统 |
| Prometheus | 一套开源的监控，报警和时间序列数据库的组合 |
| metrics | Prometheus的监控指标，格式为<metric name>{<label name>=<label value>, ...} |
| exporter | 采集监控数据并通过Prometheus监控规范对外提供数据的组件 |
| Cortex | 开源的Prometheus即服务平台 |

## 架构

Prometheus的各个组件和Kubernetes，以及数据平台的关系如下图所示：

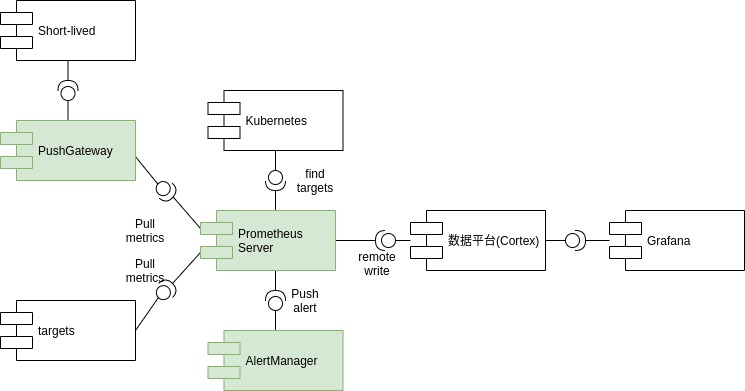


图2-1 整体架构

1. 上图中绿色部分属于Prometheus部署范围。Prometheus组件部署在Kubernetes集群内部。
2. Prometheus Server通过Kubernetes提供的接口来获取监控目标。Prometheus Server拉取监控目标的监控数据，根据规则将告警信息推送给Alertmanager。
3. AlertManager将告警推送个预先配置的目标(邮件，企业微信，webhook)。
4. 随这集群规模增加，可以在几个集群内部署多个Prometheus Server实例，前期按照管理和业务的维度来拆分，随着业务pod数量增加，可以按照namespace的维度继续拆分。Prometheus Server的容量规划参见【容量规划】章节。
5. 集群内部的Prometheus将监控数据推送给数据平台的cortex，remote\_write的url和认证方式由数据平台提前分配。
6. 数据通过grafana展示，grafana的数据源配置为数据平台提供的url。dashboard按需自定义，通过PromQL可以参考或者直接使用grafana官方提供的dashboard。

## 部署配置

### 集群内Prometheus-server对接cortex

* 从数据平台获取remote\_write的url和认证信息，在Prometheus配置文件添加remote\_write段，并添加从数据平台获取的信息。

serverFiles:

Prometheus.yml:

# 将数据通过remote\_write发送到数据平台网关, 目前需要手动选择就近的网关地址

remote\_write:

- url: <数据平台提供的url>

- basic\_auth:

username: <数据平台提供的url>

password: <数据平台提供的url>

* 在global下面配置extenal\_labels段，来制定采集到的metrics的所属site和集群信息。

global:

external\_labels:

[ site: aliyun / tenxun... ]

[ cluster: prod / pre-prod... ]

### 部署平台使用Kubernetes集群

部署平台在Kubernetes部署的pod如果要采集metrics，要满足两个条件:

1. pod提供metrics接口。
2. pod的annotation需要加上指定的简直对。

* 必须项

Prometheus.io/scrape: "true"

* 可选项

Prometheus.io/path: 默认路径是/metrics，可按需更改路径。

Prometheus.io/port: 默认不指定，可以按需更改port。

## 监控对象

Prometheus针对Kubernetes集群的监控，利用了kubernetes\_sd\_configs服务发现机制，动态的发现集群内的资源对象并进行监控。我们主要关注node和pod的指标，除此之外kubernetes\_sd\_configs还支持service, endpoints, ingress这3这角色的指标采集。

### 主机监控

利用node\_exporter来完成数据指标的采集工作，node\_exporter支持常见的监控指标，比如CPU，内测，网络，文件系统，磁盘I/O，连接数等。node\_exporter详细监控项参考5.3节。

### 容器监控

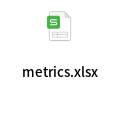
kubelet内置的cadvisor提供容器监控指标。主要提供了下面四类指标：

1. CPU：指标以container\_cpu开头，比如container\_cpu\_user\_seconds\_total表示容器用户台CPU时间总和。
2. 内测：指标以container\_memory开头，比如container\_memory\_usage\_byters表示内测总占用量。
3. 磁盘IO：指标以container\_fs开头，比如container\_fs\_writes\_bytes\_total表示磁盘写总量。
4. 网络：指标以container\_network开头，比如container\_network\_recive\_bytes\_total表示接收的字节总数。

Kubernetes的资源对象的指标都可以在容器的基础上做聚合来得到。

### 监控指标列表

监控指标详细列表如下：

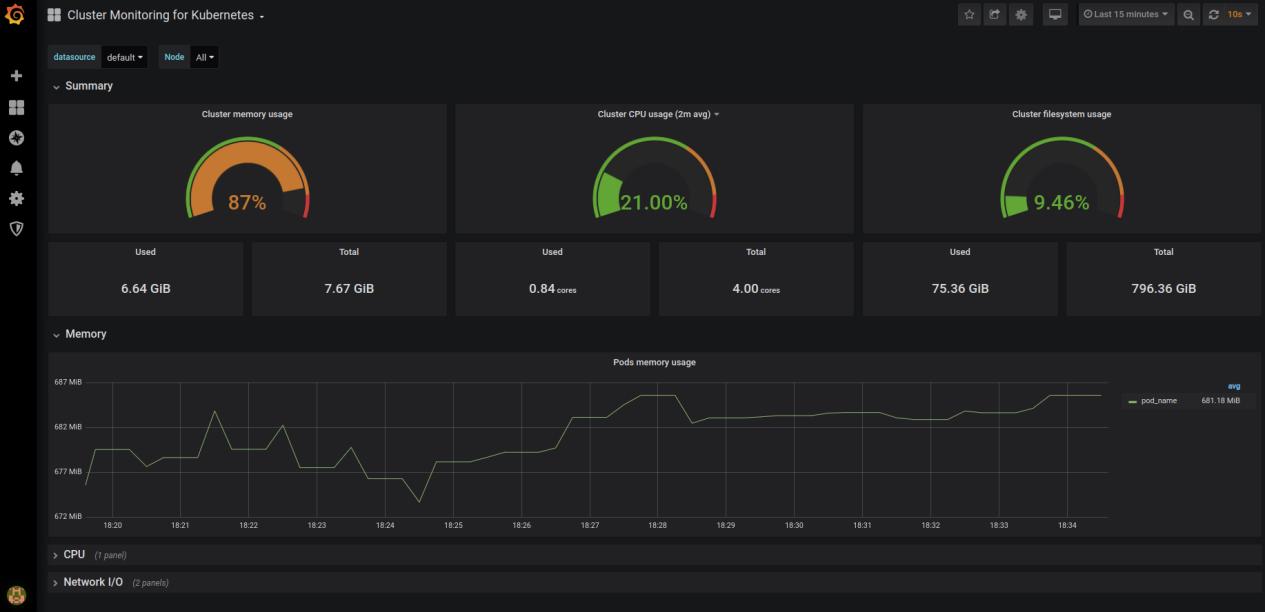


## 数据展示

利用grafana来做数据展示，grafana的数据源来是数据平台提供的url。

可以根据自己的需求新建dashboard，也可以从grafana的官网下载Kubernetes相关的dashboard，根据需求通过修改PromQL来做数据汇聚和展示。以下是官方dashboard的例子：

1. 集群状况展示



1. Node状况展示



1. Pod状况展示



## 告警

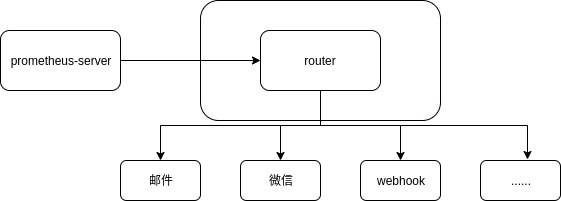


图6-1 alertmanager工作流程

### 告警规则

Prometheus-server的配置文件里添加需要告警的规则。最关键的是是用PromQL来定义告警规则。

下面定义的节点和服务组件相关的告警规则，后续可以根据需要扩展。

|  |  |  |
| --- | --- | --- |
| 节点 | 利用率 | cpu |
| 内存 |
| filesystem |
| 网络 |
| 饱和度 | 节点负载 |
| 错误 | 节点状态 |
| 服务组件 | 请求数量 | apiserver请求数量 |
| 错误数量 | apiserver错误请求量 |
| 响应时间 | apiserver请求响应时间 |

label会在alertmanager里被使用到，receiver在alertmanager里定义。

groups:

- name: node

rules:

- alert: NodeFilesystemUsage

expr: (node\_filesystem\_size\_bytes{mountpoint="/"} - node\_filesystem\_free\_bytes{mountpoint="/"} ) / node\_filesystem\_size\_bytes{mountpoint="/"} \* 100> 80

for: 5m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: High Filesystem usage detected"

description: "{{$labels.instance}}: Filesystem usage is above 80% (current value is: {{ $value }}"

- alert: NodeMemoryUsage

expr: (node\_memory\_MemTotal\_bytes - (node\_memory\_MemFree\_bytes+node\_memory\_Buffers\_bytes+node\_memory\_Cached\_bytes )) / node\_memory\_MemTotal\_bytes \* 100 > 80

for: 5m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: High Memory usage detected"

description: "{{$labels.instance}}: Memory usage is above 80% (current value is: {{ $value }}"

- alert: NodeCPUUsage

expr: (100 - avg by (instance) (irate(node\_cpu\_seconds\_total{mode="idle"}[5m])) \* 100) > 80

for: 5m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: High CPU usage detected"

description: "{{$labels.instance}}: CPU usage is above 80% (current value is: {{ $value }}"

- alert: NodeNICUsage

expr: ((rate(node\_network\_receive\_bytes\_total{device=~"en.."}[5m]) \* 8 / 1024 /1024 /1000) \* 100) > 70

for: 5m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: High network card usage detected"

description: "{{$labels.instance}}: Networ card usage is above 70% (current value is: {{ $value }}"

- alert: NodeLoadAverager

expr: (node\_load1{job="kubernetes-service-endpoints"} / count without (cpu) (count without (mode) (node\_cpu\_seconds\_total{job="kubernetes-service-endpoints"}))) > 1

for: 2m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: High load average detected"

description: "{{$labels.instance}}: Load average is too high (current value is: {{ $value }}"

- alert: Node\_exporter

expr: (up{job="kubernetes-nodes"}) != 1

for: 5m

labels:

team: node

annotations:

summary: "{{$labels.instance}}: Node status abnormal detected"

description: "{{$labels.instance}}: Node status is abnormal."

- name: component

rules:

- alert: DockerDaemon

expr: kube\_node\_spec\_taint{effect="NoExecute"} == 1

for: 5m

labels:

team: component

annotations:

summary: "{{$labels.node}}: Docker daemon is down"

description: "{{$labels.node}}: Docker daemon is down"

- alert: APIServerRequestsTotal

expr: increase(apiserver\_request\_total[5m]) > 10000

for: 5m

labels:

team: component

annotations:

summary: "APIServer HTTP requests abnormal detected"

description: "{{$labels.client}}: HTTP Requests in the 5 minute is above 10000 (current value is: {{ $value }}"

- alert: APIServerRequestsTotal

expr: increase(apiserver\_request\_count{code!~"2.."}[5m]) > 100

for: 5m

labels:

team: component

annotations:

summary: "{{$labels.client}}: HTTP request error detected"

description: "{{$labels.client}}: More than 100 HTTP request errors in the past 5 minutes (current value is: {{ $value }}"

- alert: APIServerRequestsDuration

expr: histogram\_quantile(0.99, sum(rate(apiserver\_request\_duration\_seconds\_bucket{job="kubernetes-apiservers",subresource!="log",verb!~"LIST|WATCH|WATCHLIST|PROXY|CONNECT"}[5m])) without(instance, pod)) > 4

for: 5m

labels:

team: component

annotations:

summary: "HTTP request latency detected"

description: "{{$labels.group}} {{$labels.resoure}}: HTTP request latency is greater than 4s in the past 5 minutes (current value is: {{ $value }}"

### 告警通知

Prometheus根据配置的规则产生告警并推算给alertmanager，alertmanager根据配置将告警推送给目标。

#### 邮件通知例子

global:  
 resolve\_timeout: 5m  
 smtp\_smarthost: 'smtp.163.com:465'  
 smtp\_from: 'abc@163.com'  
 smtp\_auth\_username: 'abc@163.com'  
 smtp\_auth\_password: '<password>'  
 smtp\_require\_tls: false   
  
# 所有报警信息进入后的根路由，用来设置报警的分发策略  
route:  
 group\_by: ['chentest'] # 这里的标签列表是接收到报警信息后的重新分组标签  
 group\_wait: 10s # 第一次等待多久时间发送一组警报的通知  
 group\_interval: 10s # 在发送新警报前的等待时间  
 repeat\_interval: 1m # 如果一个报警信息已经发送成功了，等待'repeat\_interval'时间来重新发送他们  
 receiver: 'xxx-email' # 发送警报的接收者的名称，与receivers name的名称相同  
  
receivers:  
- name: 'xxx-email'  
 email\_configs: # 邮箱配置  
 - send\_resolved: true # 告警解决是否通知，默认是不通知  
 to: 'abc@xxx.com'

#### 微信通知例子

global:

resolve\_timeout: 2m

wechat\_api\_url: 'https://qyapi.weixin.qq.com/cgi-bin/'

wechat\_api\_secret: 'xxx'

wechat\_api\_corp\_id: 'xxx'

route:

group\_by: ['alertname']

group\_wait: 10s

group\_interval: 10s

repeat\_interval: 1h

receiver: 'wechat'receivers:

- name: 'wechat'

wechat\_configs:

- send\_resolved: true

to\_party: '1'

agent\_id: '1000002'

#### webhook

使用webhook可以灵活的配置多个receiver，但是需要额外提供服务。

global:

resolve\_timeout: 5m

route:

receiver: webhook

group\_wait: 30s

group\_interval: 5m

repeat\_interval: 4h

group\_by: [alertname]

routes:

- receiver: webhook

group\_wait: 10s

match:

team: node

receivers:

- name: webhook

webhook\_configs:

- url: 'http://xxx/hooks/wechat/'

send\_resolved: true

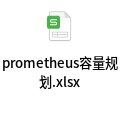
- url: 'http://xxx/hooks/mail/'

send\_resolved: true

## 容量规划

按照默认抓取metrics的间隔(15s)和保留时间(15天)，每个node上跑100个pod，可以估算Prometheus需要的内存和磁盘容量。100个节点大致需要的内容为20G，硬盘容量100G。

具体的计算公式可以参考下面的excel文件。



Prometheus作为管理面工具，部署在单独的资源池，避免和业务容器发生资源争抢。

## API

Prometheus当前稳定的HTTP API可以通过/api/v1访问。

### API响应格式

API响应的格式是JSON，每个成功的API请求都会返回2xx状态代码。  
API处理程序的无效请求将返回JSON错误对象以及以下HTTP响应代码之一：

1. 404 Bad Request：当参数错误或者缺失时。
2. 422 Unprocessable Entity 当表达式无法执行时。
3. 503 Service Unavailiable 当请求超时或者被中断时。

所有的API请求均使用以下的JSON格式：

{

"status": "success" | "error",

"data": <data>,

// Only set if status is "error". The data field may still hold

// additional data.

"errorType": "<string>",

"error": "<string>",

// Only if there were warnings while executing the request.

// There will still be data in the data field.

"warnings": ["<string>"]

}

### 表达式查询

#### 立刻查询

GET /api/v1/query

**URL 查询参数：**

* query=<string>: Prometheus表达式查询字符串.
* time=<rfc3339 | unix\_timestamp>: 指定查询的时间戳。可选的.
* timeout=<duration>: 超时时间，可选的.

如果省略time参数，则使用当前服务器时间。

**查询结果的数据部分具有以下格式：**

{  
 "resultType": "matrix" | "vector" | "scalar" | "string",  
 "result": <value>  
}

是指查询结果数据，具有不同的格式，具体取决于resultType。

#### 范围查询

GET /api/v1/query\_range

**URL 查询参数：**

* query=<string>: Prometheus表达式查询字符串.
* start=<rfc3339 | unix\_timestamp>: 开始时间戳.
* end=<rfc3339 | unix\_timestamp>: 结束时间戳.
* step=<duration>: 查询平率步长.
* timeout=<duration>: 超时时间，可选的.

****查询结果的数据部分具有以下格式：****

{  
 "resultType": "matrix" | "vector" | "scalar" | "string",  
 "result": <value>  
}

## 附录

### Exporter

业务pod要被监控的前提是需要提供metrics接口，metrics接口代码可以和业务融合在一起，也可以利用Prometheus提供的官方依赖库定义独立的exporter。

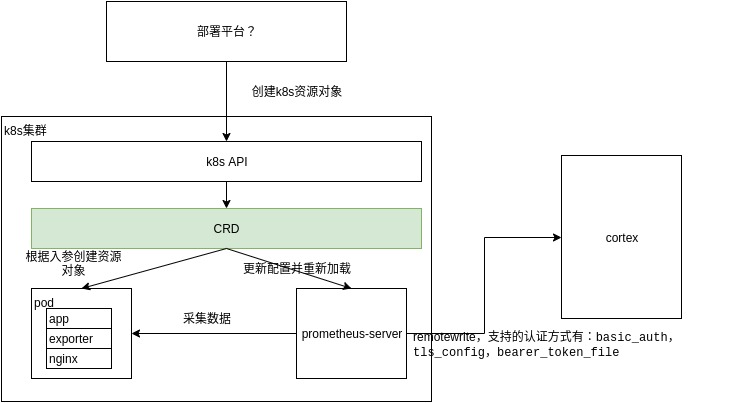
自定义的exporter可以参考<https://gitlab.oneitfarm.com/hl/myexport>实现。

Peometheus官方以及第三方都提供了各自丰富的exporter，对于常用的中间件，官方和第三方exporter可以满足大部分需求。

独立的exporter在Kubernetes以side-car的方式和业务容器部署在一个pod里。

### 拉取metrics认证

大多数exporter没有没有提供认证支持，可以借助nginx作为反向代理服务器来实现exporter的basicauth认证，在Kubernetes里通过给业务pod额外注入一个nginx容器来实现。绿色CRD部分需要单独开发。



注\*：对于metrics拉取的认证需求，也可以换一种思路通过namespace的网络隔离来实现。

### 针对哪些项目需要告警

以下是一些常用的监控告警原则

1. Google 4个黄金指标：

* 延迟：服务请求所需时间
* 通讯量：监控当前系统的流量，用于衡量服务的容量需求。
* 错误：监控当前系统所有发生的错误请求，衡量当前系统错误发生的速率。
* 饱和度：衡量当前服务的饱和度。

1. RED 原则指的是，按照如下三个维度来规划服务监控指标：

* 每秒请求数量（Rate）
* 每秒错误数量（Errors）
* 服务响应时间（Duration）

1. USE 原则指的是，按照如下三个维度来规划资源监控指标：

* 利用率（Utilization），资源被有效利用起来提供服务的平均时间占比
* 饱和度（Saturation），资源拥挤的程度，比如工作队列的长度
* 错误率（Errors），错误的数量

告警原则：

1. 作为kubernetes管理员，主要关注集群整体资源使用情况，采用USE原则。
2. 针对kubernetes的组件，采用RED原则。