

# k8s- Prometheus

## Prometheus介绍

[Prometheus](#)是一个开源监控系统，它前身是SoundCloud的警告工具包。从2012年开始，许多公司和组织开始使用Prometheus。

该项目的开发人员和用户社区非常活跃，越来越多的开发人员和用户参与到该项目中。

目前它是一个独立的开源项目，且不依赖与任何公司。为了强调这点和明确该项目治理结构，Prometheus在2016年继Kubernetes之后，加入了Cloud Native Computing Foundation。

## 特征

Prometheus的主要特征有：

1. 多维度数据模型
2. 灵活的查询语言
3. 不依赖分布式存储，单个服务器节点是自主的
4. 以HTTP方式，通过pull模型拉去时间序列数据
5. 也通过中间网关支持push模型
6. 通过服务发现或者静态配置，来发现目标服务对象
7. 支持多种多样的图表和界面展示，grafana也支持它

## 组件

Prometheus生态包括了很多组件，它们中的一些是可选的：

1. 主服务Prometheus Server负责抓取和存储时间序列数据
2. 客户端负责检测应用程序代码
3. 支持短生命周期的PUSH网关
4. 基于Rails/SQL仪表盘构建器的GUI
5. 多种导出工具，可以支持Prometheus存储数据转化为HAProxy、StatsD、Graphite等工具所需要的数据存储格式
6. 警告管理器
7. 命令行查询工具
8. 其他各种支撑工具

多数Prometheus组件是Go语言写的，这使得这些组件很容易编译和部署。

每个被监控的主机都可以通过专用的 `exporter` 程序提供输出监控数据的接口，并等待 Prometheus 服务器周期性的进行数据抓取。如果存在告警规则，则抓取到数据之后会根据规则进行计算，满足告警条件则会生成告警，并发送到 `Alertmanager` 完成告警的汇总和分发。当被监控的目标有主动推送数据的需求时，可以以 `Pushgateway` 组件进行接收并临时存储数据，然后等待 Prometheus 服务器完成数据的采集。

任何被监控的目标都需要事先纳入到监控系统中才能进行时序数据采集、存储、告警和展示，监控目标可以通过配置信息以静态形式指定，也可以让Prometheus通过服务发现的机制进行动态管理。下面是组件的一些解析：

- 监控代理程序：如node\_exporter：收集主机的指标数据，如平均负载、CPU、内存、磁盘、网络等等多个维度的指标数据。
- kubelet (cAdvisor)：收集容器指标数据，也是K8S的核心指标收集，每个容器的相关指标数据包括：CPU使用率、限额、文件系统读写限额、内存使用率和限额、网络报文发送、接收、丢弃速率等等。
- API Server：收集API Server的性能指标数据，包括控制队列的性能、请求速率和延迟时长等等
- etcd：收集etcd存储集群的相关指标数据
- kube-state-metrics：该组件可以派生出k8s相关的多个指标数据，主要是资源类型相关的计数器和元数据信息，包括制定类型的对象总数、资源限额、容器状态以及Pod资源标签系列等。

Prometheus 能够直接把 Kubernetes API Server 作为服务发现系统使用进而动态发现和监控集群中的所有可被监控的对象。

这里需要特别说明的是，Pod 资源需要添加下列注解信息才能被 Prometheus 系统自动发现并抓取其内建的指标数据。

- 1) prometheus.io/ scrape：用于标识是否需要被采集指标数据，布尔型值，true 或 false。
- 2) prometheus.io/ path：抓取指标数据时使用的 URL 路径，一般为/ metrics。
- 3) prometheus.io/ port：抓取指标数据时使用的套接字端口，如 8080。

另外，仅期望 Prometheus 为后端生成自定义指标时仅部署 Prometheus 服务器即可，它甚至也不需要数据持久功能。但若要配置完整功能的监控系统，管理员还需要在每个主机上部署 node\_exporter、按需部署其他特有类型的 exporter 以及 Alertmanager。

Prometheus服务，可以直接通过目标拉取数据，或者间接地通过中间网关拉取数据。它在本地存储抓取的所有数据，并通过一定规则进行清理和整理数据，并把得到的结果存储到新的时间序列中，PromQL和其他API可视化地展示收集的数据

## 适用场景

Prometheus在记录纯数字时间序列方面表现非常好。它既适用于面向服务器等硬件指标的监控，也适用于高动态的面向服务架构的监控。对于现在流行的微服务，Prometheus的多维度数据收集和数据筛选查询语言也是非常的强大。

Prometheus是为服务的可靠性而设计的，当服务出现故障时，它可以使你快速定位和诊断问题。它的搭建过程对硬件和服务没有很强的依赖关系。

## 不适用场景

Prometheus，它的价值在于可靠性，甚至在很恶劣的环境下，你都可以随时访问它和查看系统服务各种指标的统计信息。如果你对统计数据需要100%的精确，它并不适用，例如：它不适用于实时计费系统

更对详细介绍参考文章

<https://www.kancloud.cn/cdh0805010118/prometheus/719339>

<https://github.com/prometheus>

<https://github.com/yunlzheng/prometheus-book>

## Prometheus安装

---

# Operator方式

prometheus-operator: <https://github.com/prometheus-operator/prometheus-operator>

kube-prometheus: <https://github.com/prometheus-operator/kube-prometheus>

## 1.1、下载

```
git clone -b release-0.7 --single-branch https://github.com/coreos/kube-prometheus.git
```

## 1.2、安装operator

```
[root@k8s-master01 ~]# cd /root/kube-prometheus/manifests/setup
[root@k8s-master01 setup]# kubectl create -f .

# 查看是否Running
[root@k8s-master01 ~]# kubectl get pod -n monitoring
```

NAME	READY	STATUS	RESTARTS	AGE
prometheus-operator-848d669f6d-bz2tc	2/2	Running	0	4m16s

## 1.3、安装Prometheus

```
[root@k8s-master01 ~]# cd /root/kube-prometheus/manifests
[root@k8s-master01 manifests]# kubectl create -f .
```

## 1.4、创建ingress

```
# 创建一下Ingress代理三个service
# 创建一下Ingress代理三个service
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  generation: 1
  name: prom-ingresses
  namespace: monitoring
spec:
  rules:
  - host: alert.test.com
    http:
      paths:
      - backend:
          serviceName: alertmanager-main
          servicePort: 9093
        path: /
```

```
- host: grafana.test.com
  http:
    paths:
      - backend:
          serviceName: grafana
          servicePort: 3000
          path: /
- host: prome.test.com
  http:
    paths:
      - backend:
          serviceName: prometheus-k8s
          servicePort: 9090
          path: /
```

## 1.5、页面访问

```
# 在你Windows的hosts文件添加主机映射，浏览器访问即可
10.4.7.107 alert.test.com grafana.test.com prome.test.com
```

# Prometheus Metrics类型

Prometheus 的客户端库中提供了四种核心的指标类型。但这些类型只是在客户端库（客户端可以根据不同的数据类型调用不同的 API 接口）和在线协议中，实际在 Prometheus server 中并不对指标类型进行区分，而是简单地把这些指标统一视为无类型的时间序列

## Metrics指标类型

### 2.1、Counter（计数器）

**Counter** 类型代表一种样本数据单调递增的指标，即只增不减，除非监控系统发生了重置。

例如，你可以使用 counter 类型的指标来表示服务的请求数、已完成的任务数、错误发生的次数等。counter 主要有两个方法：

```
//将counter值加1.
Inc()

// 将指定值加到counter值上，如果指定值<0 会panic.
Add(float64)
```

Counter 类型数据可以让用户方便的了解事件产生的速率的变化，在 PromQL 内置的相关操作函数可以提供相应的分析，比如以 HTTP 应用请求量来进行说明：

```
//通过rate()函数获取HTTP请求量的增长率
rate(http_requests_total[5m])

//查询当前系统中，访问量前10的HTTP地址
topk(10, http_requests_total)
```

不要将 counter 类型应用于样本数据非单调递增的指标，例如：当前运行的进程数量（应该用 Guage 类型）。

## 2.2、Guage（仪表盘）

**Guage** 类型代表一种样本数据可以任意变化的指标，即可增可减。guage 通常用于像温度或者内存使用率这种指标数据，也可以表示能随时增加或减少的“总数”，例如：当前并发请求的数量。

对于 Gauge 类型的监控指标，通过 PromQL 内置函数 [delta()] 可以获取样本在一段时间内的变化情况，例如，计算 CPU 温度在两小时内的差异：

```
delta(cpu_temp_celsius{host="zeus"}[2h])
```

你还可以通过 PromQL 内置函数 [predict\_linear()] 基于简单线性回归的方式，对样本数据的变化趋势做出预测。例如，基于 2 小时的样本数据，来预测主机可用磁盘空间在 4 个小时之后的剩余情况：

```
predict_linear(node_filesystem_free{job="node"}[2h], 4 * 3600) < 0
```

## 2.3、Histogram（直方图）

在大多数情况下人们都倾向于使用某些量化指标的平均值，例如 CPU 的平均使用率、页面的平均响应时间。这种方式的问题很明显，以系统 API 调用的平均响应时间为例：如果大多数 API 请求都维持在 100ms 的响应时间范围内，而个别请求的响应时间需要 5s，那么就会导致某些 WEB 页面的响应时间落到中位数的情况，而这种现象被称为长尾问题。

为了区分是平均的慢还是长尾的慢，最简单的方式就是按照请求延迟的范围进行分组。例如，统计延迟在 0~10ms 之间的请求数有多少而 10~20ms 之间的请求数又有多少。通过这种方式可以快速分析系统慢的原因。Histogram 和 Summary 都是为了能够解决这样问题的存在，通过 Histogram 和 Summary 类型的监控指标，我们可以快速了解监控样本的分布情况。

Histogram 在一段时间范围内对数据进行采样（通常是请求持续时间或响应大小等），并将其计入可配置的存储桶（bucket）中，后续可通过指定区间筛选样本，也可以统计样本总数，最后一般将数据展示为直方图。

Histogram 类型的样本会提供三种指标（假设指标名称为 <basename>）：

- 样本的值分布在 bucket 中的数量，命名为 <basename>\_bucket{le="<上边界>"}。解释的更通俗易懂一点，这个值表示指标值小于等于上边界的所有样本数量。

```
// 在总共2次请求当中。http 请求响应时间 <=0.005 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",code="200",le="0.005",} 0.0
// 在总共2次请求当中。http 请求响应时间 <=0.01 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",code="200",le="0.01",} 0.0
// 在总共2次请求当中。http 请求响应时间 <=0.025 秒 的请求次数为0
```

```

    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.025",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.05",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.075",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.1",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.25",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.5",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="0.75",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="1.0",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="2.5",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="5.0",} 0.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="7.5",} 2.0
    // 在总共2次请求当中。http 请求响应时间 <=10 秒 的请求次数为 2
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="10.0",} 2.0
    io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET"
,code="200",le="+Inf",} 2.0

```

- 所有样本值的大小总和，命名为 `<basename>_sum`。

```

// 实际含义： 发生的2次 http 请求总的响应时间为 13.107670803000001 秒
io_namespace_http_requests_latency_seconds_histogram_sum{path="/",method="GET",co
de="200",} 13.107670803000001

```

- 样本总数，命名为 `<basename>_count`。值和 `<basename>_bucket{le="+Inf"}` 相同。

```

// 实际含义： 当前一共发生了 2 次 http 请求
io_namespace_http_requests_latency_seconds_histogram_count{path="/",method="GET",
code="200",} 2.0

```

## 注意

bucket 可以理解为是对数据指标值域的一个划分，划分的依据应该基于数据值的分布。注意后面的采样点是包含前面的采样点的，假设 `xxx_bucket{...,le="0.01"}` 的值为 10，而

`xxx_bucket{...,le="0.05"}` 的值为 30，那么意味着这 30 个采样点中，有 10 个是小于 10 ms 的，其余 20 个采样点的响应时间是介于 10 ms 和 50 ms 之间的。

可以通过 [histogram\\_quantile\(\)](#) 函数来计算 Histogram 类型样本的[分位数](#)。分位数可能不太好理解，你可以理解为分割数据的点。我举个例子，假设样本的 9 分位数（quantile=0.9）的值为 x，即表示小于 x 的采样值的数量占总体采样值的 90%。Histogram 还可以用来计算应用性能指标值（[Apdex score](#)）。

## 2.4、Summary（摘要）

与 Histogram 类型类似，用于表示一段时间内的数据采样结果（通常是请求持续时间或响应大小等），但它直接存储了分位数（通过客户端计算，然后展示出来），而不是通过区间来计算。

Summary 类型的样本也会提供三种指标（假设指标名称为）：

- 样本值的分位数分布情况，命名为 `<basename>{quantile="<φ>"}`。

```
// 含义：这 12 次 http 请求中有 50% 的请求响应时间是 3.052404983s
io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.5"}, 3.052404983
// 含义：这 12 次 http 请求中有 90% 的请求响应时间是 8.003261666s
io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.9"}, 8.003261666
```

- 所有样本值的大小总和，命名为 `<basename>_sum`。

```
// 含义：这12次 http 请求的总响应时间为 51.029495508s
io_namespace_http_requests_latency_seconds_summary_sum{path="/",method="GET",code="200"}, 51.029495508
```

- 样本总数，命名为 `<basename>_count`。

```
// 含义：当前一共发生了 12 次 http 请求
io_namespace_http_requests_latency_seconds_summary_count{path="/",method="GET",code="200"}, 12.0
```

现在可以总结一下 Histogram 与 Summary 的异同：

- 它们都包含了 `<basename>_sum` 和 `<basename>_count` 指标
- Histogram 需要通过 `<basename>_bucket` 来计算分位数，而 Summary 则直接存储了分位数的值。

关于 Summary 与 Histogram 的详细用法，请参考 [histograms and summaries](#)。

不同语言关于 Summary 的客户端库使用文档：

## 数据模型

Prometheus 所有采集的监控数据均以指标（metric）的形式保存在内置的时间序列数据库当中（TSDB）：属于同一指标名称，同一标签集合的、有时间戳标记的数据流。除了存储的时间序列，Prometheus 还可以根据查询请求产生临时的、衍生的时间序列作为返回结果。



### 3.1、指标名称和标签

每一条时间序列由指标名称（Metrics Name）以及一组标签（键值对）唯一标识。其中指标的名称（metric name）可以反映被监控样本的含义（例如，`http_requests_total` 表示当前系统接收到的 HTTP 请求总量），指标名称只能由 ASCII 字符、数字、下划线以及冒号组成，同时必须匹配正则表达式 `[a-zA-Z_:][a-zA-Z0-9_:]*`。

（时间序列[唯一标识]）=指标名称+标签

注意：

冒号用来表示用户自定义的记录规则，不能在 `exporter` 中或监控对象直接暴露的指标中使用冒号来定义指标名称。

通过使用标签，Prometheus 开启了强大的多维数据模型：对于相同的指标名称，通过不同标签列表的集合，会形成特定的度量维度实例（例如：所有包含度量名称为 `/api/tracks` 的 http 请求，打上 `method=POST` 的标签，就会形成具体的 http 请求）。该查询语言在这些指标和标签列表的基础上进行过滤和聚合。改变任何度量指标上的任何标签值（包括添加或删除指标），都会创建新的时间序列。

标签的名称只能由 ASCII 字符、数字以及下划线组成并满足正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*`。其中以 `_` 作为前缀的标签，是系统保留的关键字，只能在系统内部使用。标签的值则可以包含任何 `Unicode` 编码的字符。

### 3.2、样本（sample）

在时间序列中的每一个点称为一个样本（sample），样本由以下三部分组成：

- 指标（metric）：指标名称和描述当前样本特征的 labelsets；
- 时间戳（timestamp）：一个精确到毫秒的时间戳；
- 样本值（value）：一个 float64 的浮点型数据表示当前样本的值。

### 3.3、表示方式

通过如下表达方式表示指定指标名称和指定标签集合的时间序列：

```
<metric name>{<label name>=<label value>, ...}
```

例如，指标名称为 `api_http_requests_total`，标签为 `method="POST"` 和 `handler="/messages"` 的时间序列可以表示为：

```
api_http_requests_total{method="POST", handler="/messages"}
```

## PromQL

参考：<https://fuckcloudnative.io/prometheus/3-prometheus/basics.html>

瞬时向量：包含该时间序列中最新的一个样本值。

区间向量：一段时间范围内的数据。



# Prometheus监控etcd集群

## 1.1、查看接口信息

```
[root@k8s-master01 ~]# curl --cert /etc/etcd/ssl/etcd.pem --key /etc/etcd/ssl/etcd-key.pem https://192.168.1.201:2379/metrics -k
# 这样也行
curl -L http://localhost:2379/metrics
```

## 1.2、创建service和Endpoints

# 创建ep和svc代理外部的etcd服务，其他自带metrics接口的服务也是如此！

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    app: etcd-k8s
    name: etcd-k8s
    namespace: kube-system
subsets:
- addresses:      # etcd节点对应的主机ip，有几台就写几台
  - ip: 192.168.1.201
  - ip: 192.168.1.202
  - ip: 192.168.1.203
  ports:
  - name: etcd-port
    port: 2379    # etcd端口
    protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: etcd-k8s
    name: etcd-k8s
    namespace: kube-system
spec:
  ports:
  - name: etcd-port
    port: 2379
    protocol: TCP
    targetPort: 2379
  type: ClusterIP
```

### 1.3、测试是否代理成功

#再次curl, 把IP换成svc的IP测试, 输出相同内容即创建成功

```
[root@k8s-master01 ~]# kubectl get svc -n kube-system etcd-k8s
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
etcd-ep	ClusterIP	10.103.53.103	<none>	2379/TCP	8m54s

# 再次请求接口

```
[root@k8s-master01 ~]# curl --cert /etc/etcd/ssl/etcd.pem --key /etc/etcd/ssl/etcd-key.pem https://10.111.200.116:2379/metrics -k
```

### 1.4、创建secret

# 1、这里我们k8s-master01节点进行创建,ca为k8sca证书, 剩下2个为etcd证书, 这是我证书所在位置

```
cert-file: '/etc/kubernetes/pki/etcd/etcd.pem'
```

```
key-file: '/etc/kubernetes/pki/etcd/etcd-key.pem'
```

```
trusted-ca-file: '/etc/kubernetes/pki/etcd/etcd-ca.pem'
```

# 2、接下来我们需要创建一个secret, 让prometheus pod节点挂载

```
kubectl create secret generic etcd-ssl --from-file=/etc/kubernetes/pki/etcd/etcd-ca.pem  
--from-file=/etc/kubernetes/pki/etcd/etcd.pem --from-  
file=/etc/kubernetes/pki/etcd/etcd-key.pem -n monitoring
```

# 3、创建完成后可以检查一下

```
[root@k8s-master01 prometheus-down]# kubectl describe secrets -n monitoring etcd-ssl
```

```
Name:          etcd-ssl  
Namespace:     monitoring  
Labels:        <none>  
Annotations:   <none>
```

```
Type: Opaque
```

```
Data
```

```
====
```

```
etcd-ca.pem: 1367 bytes
```

```
etcd-key.pem: 1679 bytes
```

```
etcd.pem: 1509 bytes
```

### 1.5、编辑prometheus, 把证书挂载进去

```
# 1、通过edit直接编辑prometheus
[root@k8s-master01 ~]# kubectl edit prometheus k8s -n monitoring
# 在replicas底下加上secret名称
replicas:2
secrets:
- etcd-ssl #添加secret名称

# 进入容器查看，就可以看到证书挂载进去了
[root@k8s-master01 prometheus-down]# kubectl exec -it -n monitoring prometheus-k8s-0
/bin/sh

# 查看文件是否存在
/prometheus $ ls /etc/prometheus/secrets/etcd-ssl/
etcd-ca.pem  etcd-key.pem  etcd.pem
```

## 1.6、创建ServiceMonitor

```
[root@k8s-master01 ~]# cat etcd-servicemonitor.yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: etcd-k8s
  namespace: monitoring
  labels:
    app: etcd-k8s
spec:
  jobLabel: app
  endpoints:
    - interval: 30s
      port: etcd-port # 这个port对应 Service.spec.ports.name
      scheme: https
      tlsConfig:
        caFile: /etc/prometheus/secrets/etcd-ssl/etcd-ca.pem #证书路径 (在prometheus pod
里路径)
        certFile: /etc/prometheus/secrets/etcd-ssl/etcd.pem
        keyFile: /etc/prometheus/secrets/etcd-ssl/etcd-key.pem
        insecureSkipVerify: true # 关闭证书校验
  selector:
    matchLabels:
      app: etcd-k8s # 跟scv的lables保持一致
  namespaceSelector:
    matchNames:
      - kube-system # 跟svc所在namespace保持一致

# 匹配Kube-system这个命名空间下面具有app=etcd-k8s这个label标签的Serve，job label用于检索job任务
名称的标签。由于证书serverName和etcd中签发的证书可能不匹配，所以添加了insecureSkipVerify=true将不
再对服务端的证书进行校验
```

## 1.7、页面查看三个etcd节点都获取到数据

此处数据获取有点慢，需要等待一下

PrometheusAlertsGraphStatus▼HelpClassic UI

Targets

AllUnhealthy

monitoring/alertmanager/0 (3/3 up)show more

monitoring/etcd-k8s/0 (3/3 up)show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://192.168.1.201:2379/metrics	UP	endpoint="etcd-prot"instance="192.168.1.201:2379"job="etcd-k8s"namespace="kube-system"service="etcd-k8s"	18.861s	39.356ms	
https://192.168.1.202:2379/metrics	UP	endpoint="etcd-prot"instance="192.168.1.202:2379"job="etcd-k8s"namespace="kube-system"service="etcd-k8s"	24.655s	25.796ms	
https://192.168.1.203:2379/metrics	UP	endpoint="etcd-prot"instance="192.168.1.203:2379"job="etcd-k8s"namespace="kube-system"service="etcd-k8s"	15.321s	20.426ms	

monitoring/grafana/0 (1/1 up)show more

monitoring/kube-apiserver/0 (1/1 up)show more

monitoring/kube-state-metrics/0 (1/1 up)show more

monitoring/kube-state-metrics/1 (1/1 up)show more

monitoring/kubelet/0 (5/5 up)show more

## 1.8、grafana模板导入

数据采集完成后，接下来可以在grafana中导入dashboard


# 打开官网来的如下图所示，点击下载Jso文件

grafana官网：<https://grafana.com/grafana/dashboards/3070>

中文版ETCD集群插件：<https://grafana.com/grafana/dashboards/9733>

Grafana LabsGrafanaProductsOpen SourceLearnDownloadsLoginContact us

All dashboards » Etcd by Prometheus



Etcd by Prometheus

by Vincent Brouillet

DASHBOARD

Etcd Dashboard for Prometheus metrics scraper

Last updated: 3 years ago


Downloads: 228925

Reviews: 1

★★★★☆

Add your review!

OverviewRevisionsReviews



Tested with:

- kube-aws 0.9.7
- Kubernetes 1.6
- Prometheus Operator 0.11.1
- Prometheus 1.7.0

Get this dashboard:

3070

Copy ID to Clipboard

Download JSON

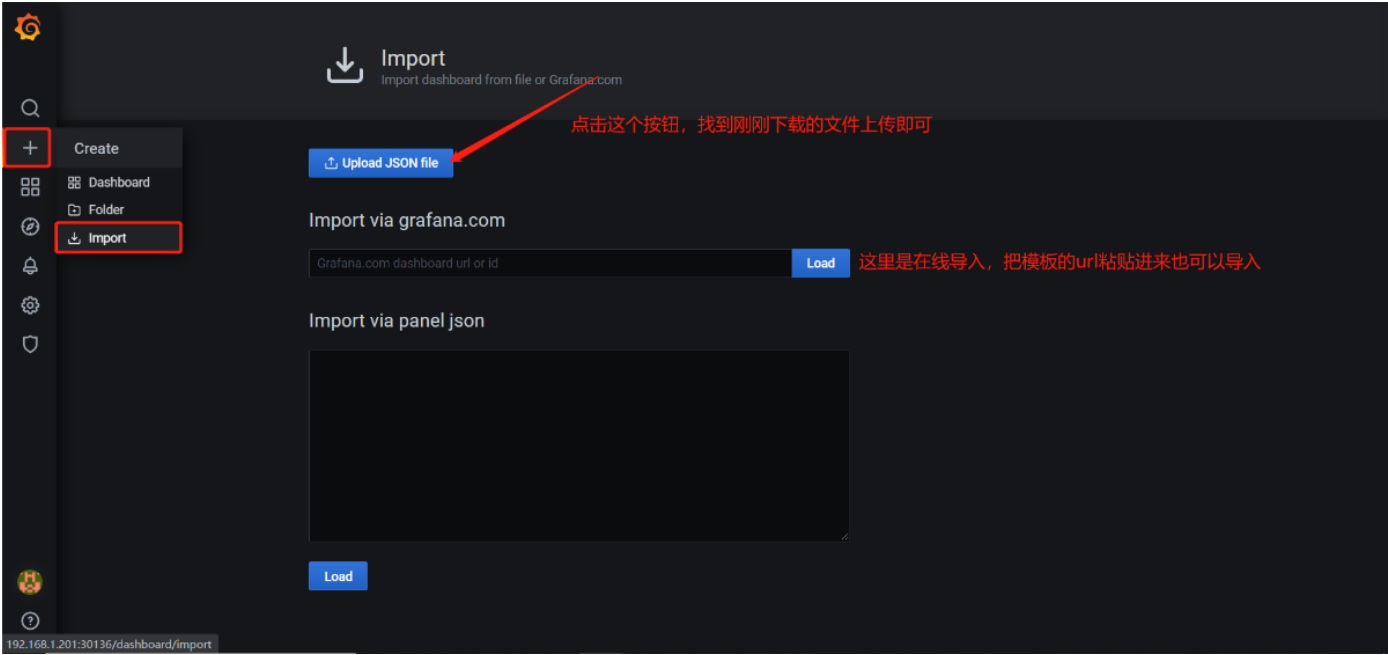
How do I import this dashboard?

Dependencies:

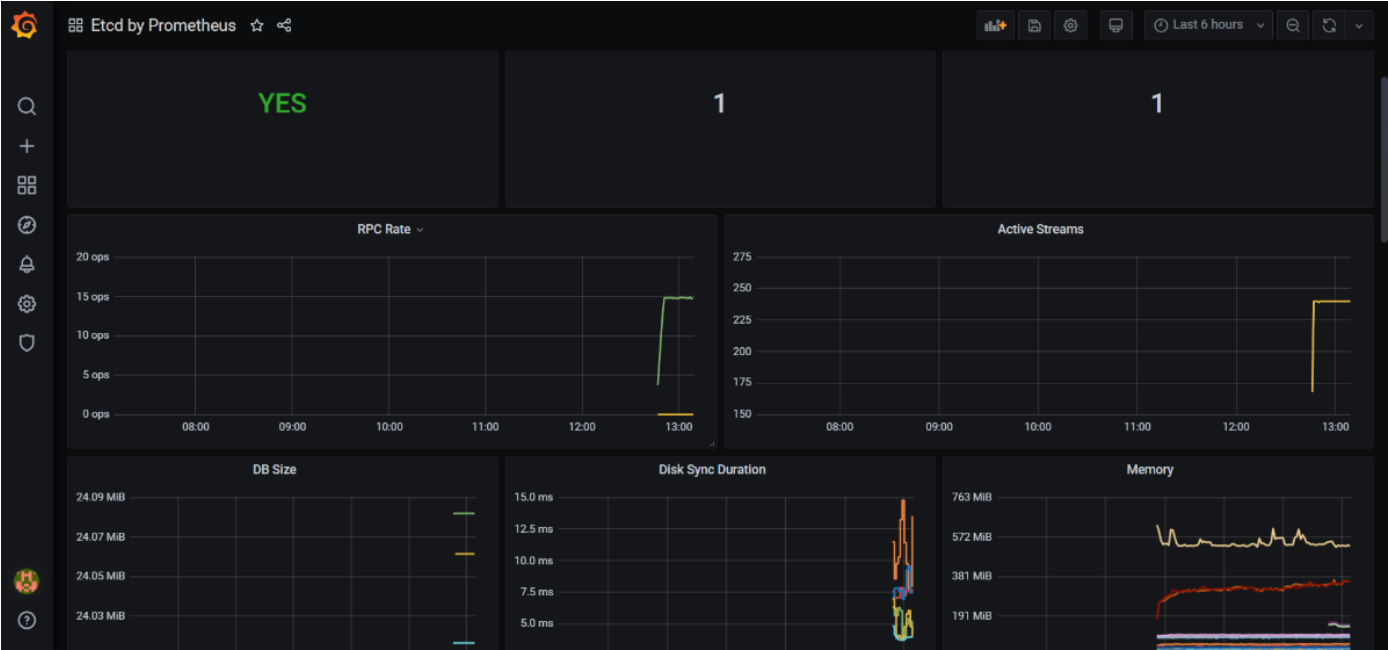
GRAFANA 4.4.1

The Etcd cluster is running outside of Kubernetes on EC2 instances. I've used Prometheus Operator ServiceMonitor to scrape the

点击HOME->导入模板



导入后页面展示



# Promethues之node-exporter

在 Kubernetes 下，Promethues 通过与 Kubernetes API 集成，主要支持5中服务发现模式，分别是：Node、Service、Pod、Endpoints、Ingress。

我们通过 kubectl 命令可以很方便的获取到当前集群中的所有节点信息：

```
[root@master1 ~]# kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
master1	Ready	master	40d	v1.19.3
node1	Ready	<none>	40d	v1.19.3
node2	Ready	<none>	40d	v1.19.3

但是要让 Prometheus 也能够获取到当前集群中的所有节点信息的话，我们就需要利用 Node 的服务发现模式，同样的，在 `prometheus.yml` 文件中配置如下的 job 任务即可：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
    - role: node

# 然后apply这个cm
```

其实现在到页面上查看会报错，查看Promethues 日志：

```
orbidden: User \"system:serviceaccount:kube-mon:default\" cannot list resource
\"nodes\" in API group \"\" at the cluster scope"
level=error ts=2021-02-12T14:15:17.104Z caller=klog.go:94 component=k8s_client_runtime
func=ErrorDepth msg="/app/discovery/kubernetes/kubernetes.go:335: Failed to list
*v1.Node: nodes is forbidden: User \"system:serviceaccount:kube-mon:default\" cannot
list resource \"nodes\" in API group \"\" at the cluster scope"
level=error ts=2021-02-12T14:15:18.108Z caller=klog.go:94 component=k8s_client_runtime
func=ErrorDepth msg="/app/discovery/kubernetes/kubernetes.go:335: Failed to list
*v1.Node: nodes is forbidden: User \"system:serviceaccount:kube-mon:default\" cannot
list resource \"nodes\" in API group \"\" at the cluster scope"
level=error ts=2021-02-12T14:15:19.112Z caller=klog.go:94 component=k8s_client_runtime
func=ErrorDepth msg="/app/discovery/kubernetes/kubernetes.go:335: Failed to list
*v1.Node: nodes is forbidden: User \"system:serviceaccount:kube-mon:default\" cannot
list resource \"nodes\" in API group \"\" at the cluster scope"
```

实际上就是没权限，所以我们得给他加权限

```
# 默认的sa是没有权限的
[root@master1 prometheus]# kubectl get sa default -n kube-mon
```

NAME	SECRETS	AGE
default	1	6h37m

## 步骤一

在创建Prometheus的deployment中加上

```
# 位置: Deployment.spec.template.spec下面
serviceAccountName: prometheus # 指定自己创建的sa

"""
```

```

template:
  metadata:
    labels:
      app: prometheus # 这个labels要跟selector的一致, 才能匹配上
  spec:
    serviceAccountName: prometheus # 指定自己创建的sa
    volumes:           # 真正挂载, 这里需要用到底下声明挂载的name
    - name: config      # 这里需要跟底下声明挂载的name(config)保持一致
      configMap:
        name: prometheus-config # configmap name
"""

```

## 步骤二

创建SA,创建文件 `prometheus-sa.yaml`

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus # 名字要跟deployment中的那个serviceAccountName保持一致
  namespace: kube-mon

```

## 步骤三

创建集群角色, 且做好绑定。文件 `prometheus-rbac.yaml`

```

# 先创建一个集群角色, 给定以下一堆权限
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/proxy
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - "extensions"
  resources:
  - ingresses
  verbs:

```



```

- get
- list
- watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
---
# 然后做集群角色绑定，把集群角色跟sa绑定在一起即可
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io # 使用这个API
  kind: ClusterRole # 指定绑定的ClusterRole名字是prometheus
  name: prometheus
subjects: # 指定绑定的kind为ServiceAccount, name is prometheus, namespace is kube-mon
- kind: ServiceAccount
  name: prometheus
  namespace: kube-mon
# 所以现在这个namespace下的Pod就有以上权限了

```

## 步骤四

然后重新apply这几个文件

```

[root@master1 prometheus]# ll
total 24
-rw-r--r-- 1 root root 1926 Feb 12 21:54 prome-node-exporter.yaml
-rw-r--r-- 1 root root 968 Feb 12 20:37 prome-redis.yaml
-rw-r--r-- 1 root root 574 Feb 12 22:11 prometheus-cm.yaml
-rw-r--r-- 1 root root 2247 Feb 12 22:52 prometheus-deploy.yaml
-rw-r--r-- 1 root root 861 Feb 12 17:30 prometheus-rbac.yaml
-rw-r--r-- 1 root root 451 Feb 12 16:28 prometheus-svc.yaml
[root@master1 prometheus]# kubectl apply -f .

```

步骤五、

现在页面查看还是有问题的

Prometheus Alerts Graph Status Help

All Unhealthy

coredns (2/2 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.0.3:9153/metrics	UP	instance="10.244.0.3:9153" job="coredns"	8.276s ago	6.091ms	
http://10.244.2.153:9153/metrics	UP	instance="10.244.2.153:9153" job="coredns"	10.96s ago	6.901ms	

kubernetes-nodes (0/3 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.1.11:10250/metrics	DOWN	instance="master1" job="kubernetes-nodes"	8.72s ago	3.021ms	server returned HTTP status 400 Bad Request
http://192.168.1.22:10250/metrics	DOWN	instance="node1" job="kubernetes-nodes"	7.946s ago	2.631ms	server returned HTTP status 400 Bad Request
http://192.168.1.33:10250/metrics	DOWN	instance="node2" job="kubernetes-nodes"	1.984s ago	1.875ms	server returned HTTP status 400 Bad Request

prometheus (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	5.195s ago	11.37ms	


redis (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
				Scrape	

我们可以看到上面的 `kubernetes-nodes` 这个 job 任务已经自动发现了我们5个 node 节点，但是在获取数据的时候失败了，出现了类似于下面的错误信息：

```
server returned HTTP status 400 Bad Request
```

这个是因为 prometheus 去发现 Node 模式的服务的时候，访问的端口默认是10250，而默认是需要认证的 https 协议才有权访问的，但实际上我们并不是希望让去访问10250端口的 `/metrics` 接口，而是 `node-exporter` 绑定到节点的 9100 端口，所以我们应该将这里的 `10250` 替换成 `9100`，但是应该怎样替换呢？

这里我们就需要使用到 Prometheus 提供的 `relabel_configs` 中的 `replace` 能力了，`relabel` 可以在 Prometheus 采集数据之前，通过 Target 实例的 `Metadata` 信息，动态重新写入 Label 的值。除此之外，我们还能根据 Target 实例的 `Metadata` 信息选择是否采集或者忽略该 Target 实例。比如我们这里就可以去匹配 `__address__` 这个 Label 标签，然后替换掉其中的端口，如果你不知道有哪些 Label 标签可以操作的话，可以将鼠标  移动到 Targets 的标签区域，其中显示的 `Before relabeling` 区域都是我们可以操作的标签：

Prometheus Alerts Graph Status ▾ Help						
All Unhealthy						
coredns (2/2 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://10.244.0.3:9153/metrics">http://10.244.0.3:9153/metrics</a>	UP	instance="10.244.0.3:9153" job="coredns"	8.276s ago	6.091ms		
<a href="http://10.244.2.153:9153/metrics">http://10.244.2.153:9153/metrics</a>	UP	instance="10.244.2.153:9153" job="coredns"	10.96s ago	6.901ms		
kubernetes-nodes (0/3 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://192.168.1.11:10250/metrics">http://192.168.1.11:10250/metrics</a>	DOWN	instance="master1" job="kubernetes-nodes"	8.72s ago	3.021ms	server returned HTTP status 400 Bad Request	
<a href="http://192.168.1.22:10250/metrics">http://192.168.1.22:10250/metrics</a>	DOWN	instance="node1" job="kubernetes-nodes"	7.946s ago	2.631ms	server returned HTTP status 400 Bad Request	
<a href="http://192.168.1.33:10250/metrics">http://192.168.1.33:10250/metrics</a>	DOWN	instance="node2" job="kubernetes-nodes"	1.984s ago	1.875ms	server returned HTTP status 400 Bad Request	
prometheus (1/1 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://localhost:9090/metrics">http://localhost:9090/metrics</a>	UP	instance="localhost:9090" job="prometheus"	8.195s ago	11.37ms		
redis (1/1 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://localhost:9090/metrics">http://localhost:9090/metrics</a>	UP	instance="localhost:9090" job="prometheus"	8.195s ago	11.37ms		

现在我们来替换掉端口，修改 ConfigMap：（此处忽略，用下面的最终版本）

```

- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
  - role: node
    relabel_configs:
    - source_labels: [__address__]
      regex: '(.*):10250'
      replacement: '${1}:9100'
      target_label: __address__
      action: replace

```

这里就是一个正则表达式，去匹配 `__address__` 这个标签，然后将 host 部分保留下来，port 替换成了 9100，现在我们重新更新配置文件，执行 reload 操作，然后再去看 Prometheus 的 Dashboard 的 Targets 路径下面 kubernetes-nodes 这个 job 任务是否正常了：

Prometheus Alerts Graph Status ▾ Help						
kubernetes-nodes (3/3 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://192.168.1.11:9100/metrics">http://192.168.1.11:9100/metrics</a>	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="master1" job="kubernetes-nodes" kubernetes_io_arch="amd64" kubernetes_io_hostname="master1" kubernetes_io_os="linux"	10.699s ago	349.3ms		
<a href="http://192.168.1.22:9100/metrics">http://192.168.1.22:9100/metrics</a>	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="node1" job="kubernetes-nodes" kubernetes_io_arch="amd64" kubernetes_io_hostname="node1" kubernetes_io_os="linux"	5.22s ago	248ms		
<a href="http://192.168.1.33:9100/metrics">http://192.168.1.33:9100/metrics</a>	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" com="youdianzishi" instance="node2" job="kubernetes-nodes" kubernetes_io_arch="amd64" kubernetes_io_hostname="node2" kubernetes_io_os="linux" monitor="prometheus"	8.587s ago	399.8ms		
prometheus (1/1 up) <a href="#">show less</a>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
<a href="http://localhost:9090/metrics">http://localhost:9090/metrics</a>	UP	instance="localhost:9090" job="prometheus"	6.05s ago	11ms		

我们可以看到现在已经正常了，但是还有一个问题就是我们采集的指标数据 Label 标签就只有一个节点的 hostname，这对于我们在进行监控分组分类查询的时候带来了很不方便的地方，要是我们能够将集群中 Node 节点的 Label 标签也能获取到就很好了。这里我们可以通过 `labelmap` 这个属性来将 Kubernetes 的 Label 标签添加为 Prometheus 的指标数据的标签：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
  - role: node
  relabel_configs:
  - source_labels: [__address__]
    regex: '(.*):10250'
    replacement: '${1}:9100'
    target_label: __address__
    action: replace
  - action: labelmap # 这个
    regex: __meta_kubernetes_node_label_(.+)
# 热更新, ip是prometheusPod的Ip
[root@master1 ~]# curl -X POST "http://10.244.1.139:9090/-/reload"
```

另外由于 kubelet 也自带了一些监控指标数据，就上面我们提到的 10250 端口，所以我们这里也把 kubelet 的监控任务也一并配置上：

```
- job_name: 'kubernetes-kubelet'
  kubernetes_sd_configs:
  - role: node
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    insecure_skip_verify: true
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  relabel_configs:
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)

```

但是这里需要特别注意的是这里必须使用 `https` 协议访问，这样就必然需要提供证书，我们这里是通过配置 `insecure_skip_verify: true` 来跳过了证书校验，但是除此之外，要访问集群的资源，还必须要有对应的权限才可以，也就是对应的 ServiceAccount 的权限允许才可以，我们这里部署的 prometheus 关联的 ServiceAccount 对象前面我们已经提到过了，这里我们只需要将 Pod 中自动注入的 `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` 和 `/var/run/secrets/kubernetes.io/serviceaccount/token` 文件配置上，就可以获取到对应的权限了。

现在我们去更新下配置文件，执行 reload 操作，让配置生效，然后访问 Prometheus 的 Dashboard 查看 Targets 路径：

到这里我们就把 Kubernetes 集群节点使用 Prometheus 监控起来了，接下来我们再来和大家学习下怎样监控 Pod 或者 Service 之类的资源对象。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/port: "9153" # metrics 接口的端口
    prometheus.io/scrape: "true" # 这个注解可以让prometheus自动发现
```

## Promethues之黑盒监控

**白盒监控：**监控一些内部的数据，topic的监控数据，Redis key的大小。内部暴露的指标被称为白盒监控。比较关注的是原因。

**黑盒监控：**站在用户的角度看到的東西。网站不能打开，网站打开的比较慢。比较关注现象，表示正在发生的问题，正在发生的告警。

### 一、部署exporter

黑盒监控官网：

```
https://github.com/prometheus/blackbox_exporter
https://github.com/prometheus/blackbox_exporter/blob/master/blackbox.yml
https://grafana.com/grafana/dashboards/5345
```

# 1、创建ConfigMap，通过ConfigMap形式挂载进容器里

```
apiVersion: v1
data:
  blackbox.yml: |-
    modules:
      http_2xx:
        prober: http
      http_post_2xx:
        prober: http
      http:
        method: POST
      tcp_connect:
        prober: tcp
      pop3s_banner:
        prober: tcp
      tcp:
        query_response:
          - expect: "^+OK"
        tls: true
        tls_config:
          insecure_skip_verify: false
      ssh_banner:
        prober: tcp
      tcp:
```

```

        query_response:
          - expect: "^SSH-2.0-"
    irc_banner:
      prober: tcp
      tcp:
        query_response:
          - send: "NICK prober"
          - send: "USER prober prober prober :prober"
          - expect: "PING :([^\s]+)"
            send: "PONG ${1}"
          - expect: "^[^\s]+ 001"
    icmp:
      prober: icmp
kind: ConfigMap
metadata:
  name: blackbox.conf
  namespace: monitoring

```

```

---
# 2、创建Service、deployment

```

```

---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: blackbox-exporter
  name: blackbox-exporter
  namespace: monitoring
spec:
  ports:
    - name: container-1-web-1
      port: 9115
      protocol: TCP
      targetPort: 9115
  selector:
    app: blackbox-exporter
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}

```

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: blackbox-exporter
  name: blackbox-exporter
  namespace: monitoring

```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: blackbox-exporter
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: blackbox-exporter
    spec:
      affinity: {}
      containers:
      - args:
        - --config.file=/mnt/blackbox.yml
        env:
        - name: TZ
          value: Asia/Shanghai
        - name: LANG
          value: C.UTF-8
        image: prom/blackbox-exporter:master
        imagePullPolicy: IfNotPresent
        lifecycle: {}
        name: blackbox-exporter
        ports:
        - containerPort: 9115
          name: web
          protocol: TCP
        resources:
          limits:
            cpu: 260m
            memory: 395Mi
          requests:
            cpu: 10m
            memory: 10Mi
        securityContext:
          allowPrivilegeEscalation: false
          capabilities: {}
          privileged: false
          procMount: Default
          readOnlyRootFilesystem: false
          runAsNonRoot: false
        volumeMounts:
        - mountPath: /usr/share/zoneinfo/Asia/Shanghai
```



```

    name: tz-config
  - mountPath: /etc/localtime
    name: tz-config
  - mountPath: /etc/timezone
    name: timezone
  - mountPath: /mnt
    name: config
dnsPolicy: ClusterFirst
restartPolicy: Always
securityContext: {}
volumes:
  - hostPath:
      path: /usr/share/zoneinfo/Asia/Shanghai
      type: ""
    name: tz-config
  - hostPath:
      path: /etc/timezone
      type: ""
    name: timezone
  - configMap:
      name: blackbox.conf
    name: config
# 查看pod状态
[root@k8s-master01 ~]# kubectl get pod -n monitoring blackbox-exporter-78bb74fd9d-z5xdq

```

NAME	READY	STATUS	RESTARTS	AGE
blackbox-exporter-78bb74fd9d-z5xdq	1/1	Running	0	67s

## 二、additional传统监控

```

# 测试exporter是否正常
# 查看svc的IP
[root@k8s-master01 ~]# kubectl get svc -n monitoring blackbox-exporter

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
blackbox-exporter	ClusterIP	10.100.9.18	<none>	9115/TCP	30m

```

# curl一下exporter的svc
[root@k8s-master01 ~]# curl "http://10.100.9.18:9115/probe?target=baidu.com&module=http_2xx"

```

### 2.1、添加个监控测试

```

[root@k8s-master01 prometheus-down]# vim prometheus-additional.yaml
- job_name: "blackbox"
  metrics_path: /probe
  params:
    module: [http_2xx] # Look for a HTTP 200 response.
  static_configs:

```

```

- targets:
  - http://prometheus.io
  - https://prometheus.io
  - http://www.baidu.com
relabel_configs:
- source_labels: [__address__]
  target_label: __param_target
- source_labels: [__param_target]
  target_label: instance
- target_label: __address__
  replacement: blackbox-exporter:9115 # exporter的svc name

```

# Then you will need to make a secret out of this configuration.

```

[root@k8s-master01 prometheus-down]# kubectl create secret generic additional-scrape-
configs --from-file=prometheus-additional.yaml --dry-run -oyaml > additional-scrape-
configs.yaml

```

# 查看Secret

```

[root@k8s-master01 prometheus-down]# cat additional-scrape-configs.yaml

```

```

apiVersion: v1

```

```

data:

```

```

  prometheus-additional.yaml:

```

```

LSBqb2JfbmFtZTogImJsYWNrYm94IgotIG1ldHJpY3NfcGF0aDogL3Byb2JlCiAgcGFyYW1zOgogICAgbW9kdWx
l0iBbaHR0cF8yeHhdICAjIExvb2sgZm9yIGEgSFRUUCAYMDAgcmVzcG9uc2UuCiAgc3RhZGljX2NvbmZpZ3M6Ci
AgICAtIHRhcmldHM6CiAgICAgIC0gaHR0cDovL3Byb2JldGhldXMuaW8gICAgIyBUYXJnZXQgdG8gcHJvYmUgd
2l0aCBodHRwLgogICAgICAtIGH0dHBzOi8vcHJvbWV0aGVlcy5pbyAgICMgVGZyZ2V0IHRvIHByb2JlIHdpdGgg
aHR0cHMuCiAgICAgIC0gaHR0cDovL3d3dy5iYWlkdS5jb20gICAgIyBUYXJnZXQgdG8gcHJvYmUgd2l0aCBodHR
wIG9uIHVcnQgODA4MC4KICByZWxhYmVsX2NvbmZpZ3M6CiAgICAtIHNvdXJjZV9sYWJlbHM6IFtfx2FkZHZHJlc3
NfX10KICAgICAgdGFyZ2V0X2xhYmVsOiBfX3BhcmFtX3RhcmldAogICAgLSBzb3VyY2VfbGFzWxzOiBbX19wY
XJhbV90YXJnZXRdCiAgICAgIHRhcmldfF9sYWJlbDogaw5zdGFuY2UKICAgIC0gdGFyZ2V0X2xhYmVsOiBfX2Fk
ZHZHJlc3NfXwogICAgICByZXBsYWNlbWVudDogYmxhY2tib3gtZXhwb3J0ZXI6OTExNSAgIyBleHBvcnRlcueahHN
2YyBuYW1lCg==

```

```

kind: Secret

```

```

metadata:

```

```

  name: additional-scrape-configs

```

# 创建Secret

```

[root@k8s-master01 prometheus-down]# kubectl apply -f additional-scrape-configs.yaml
-n monitoring
secret/additional-scrape-configs created

```

# 进到manifests目录, 编辑

```

[root@k8s-master01 manifests]# vim prometheus-prometheus.yaml

```

```

apiVersion: monitoring.coreos.com/v1

```

```

kind: Prometheus

```

```

metadata:

```

```

  name: prometheus

```

```

  labels:

```

```

    prometheus: prometheus

```

spec:

replicas: 2

... 加上下面3行

additionalScrapeConfigs:

name: additional-scrape-configs

key: prometheus-additional.yaml

...

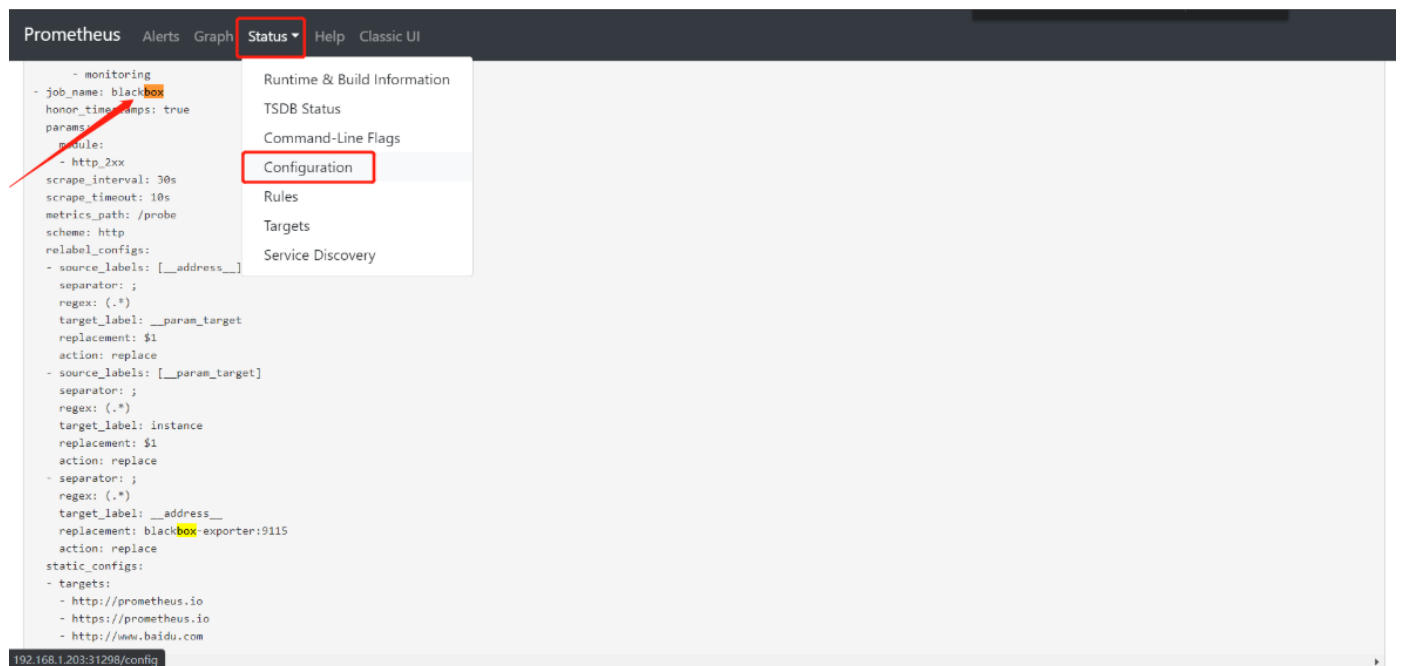
# replace刚刚修改的文件

```
[root@k8s-master01 manifests]# kubectl replace -f prometheus-prometheus.yaml -n monitoring
```

# 手动删除pod、使之重新构建

```
[root@k8s-master01 manifests]# kubectl delete po prometheus-k8s-0 prometheus-k8s-1 -n monitoring
```

查看是否成功加载配置：



数据查看：

☒ Enable query history

☒ Use local time

☒ Enable autocomplete

Q

probe\_duration\_seconds

probe开头的

Execute

Table

Graph

Load time: 11ms Resolution: 14s Result series: 3

<

2021-01-17 18:19:27

×

>

probe_duration_seconds(instance="http://prometheus.io", job="blackbox")	0.005242288
probe_duration_seconds(instance="http://www.baidu.com", job="blackbox")	0.282851118
probe_duration_seconds(instance="https://prometheus.io", job="blackbox")	0.046233361

Remove Panel

Add Panel