Prometheus 监控服务

接入指南

产品文档

腾讯云

文档目录

接入指南
抓取配置说明

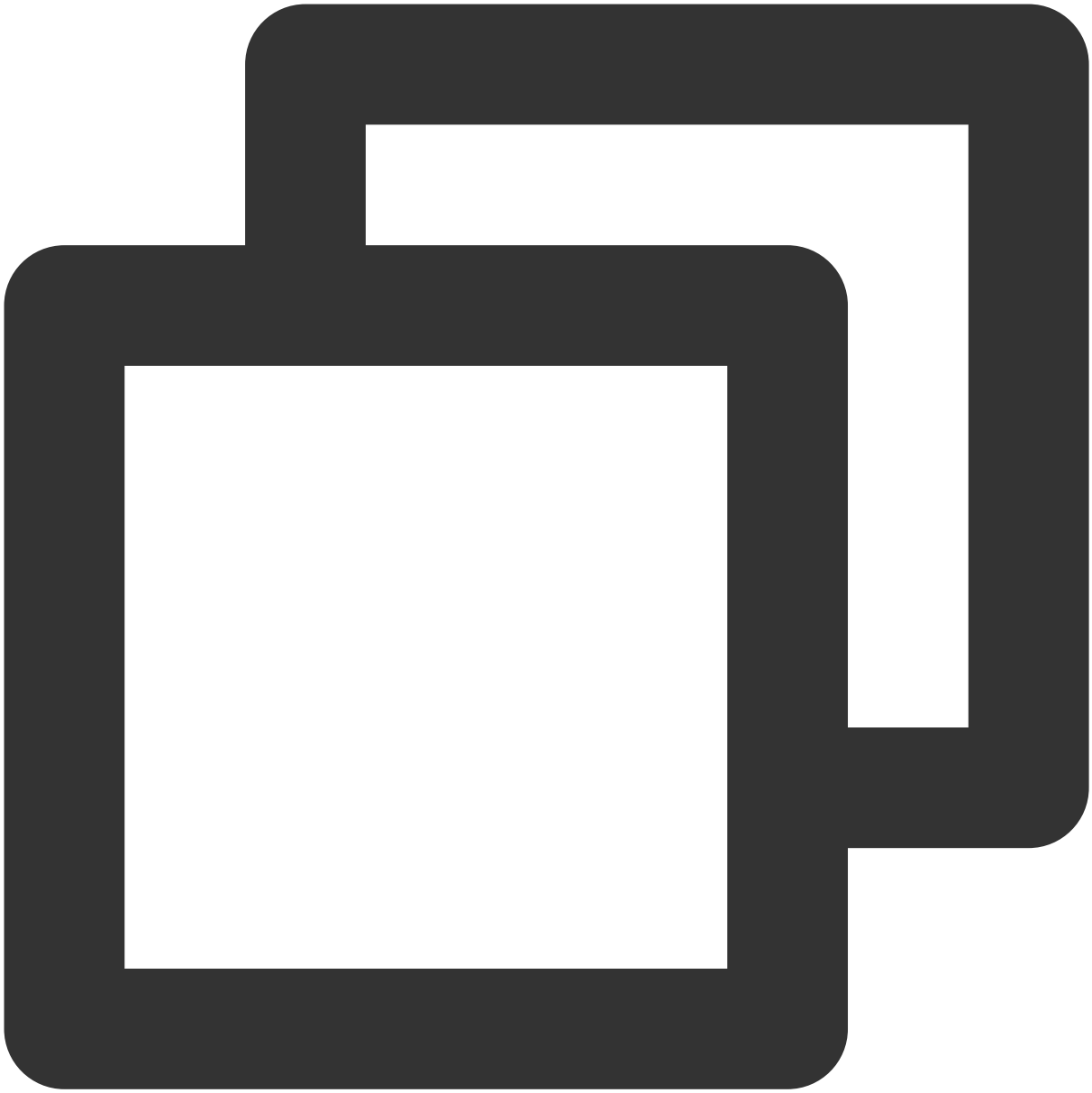
最近更新时间：2024-01-29 15:55:14

概述

Prometheus 主要通过 Pull 的方式来抓取目标服务暴露出来的监控接口，因此需要配置对应的抓取任务来请求监控数据并写入到 Prometheus 提供的存储中，目前 Prometheus 服务提供了如下几个任务的配置：
原生 Job 配置：提供 Prometheus 原生抓取 Job 的配置。
Pod Monitor：在 K8S 生态下，基于 Prometheus Operator 来抓取 Pod 上对应的监控数据。
Service Monitor：在 K8S 生态下，基于 Prometheus Operator 来抓取 Service 对应 Endpoints 上的监控数据。
说明：
`[]` 中的配置项为可选。

原生 Job 配置

相应配置项说明如下：



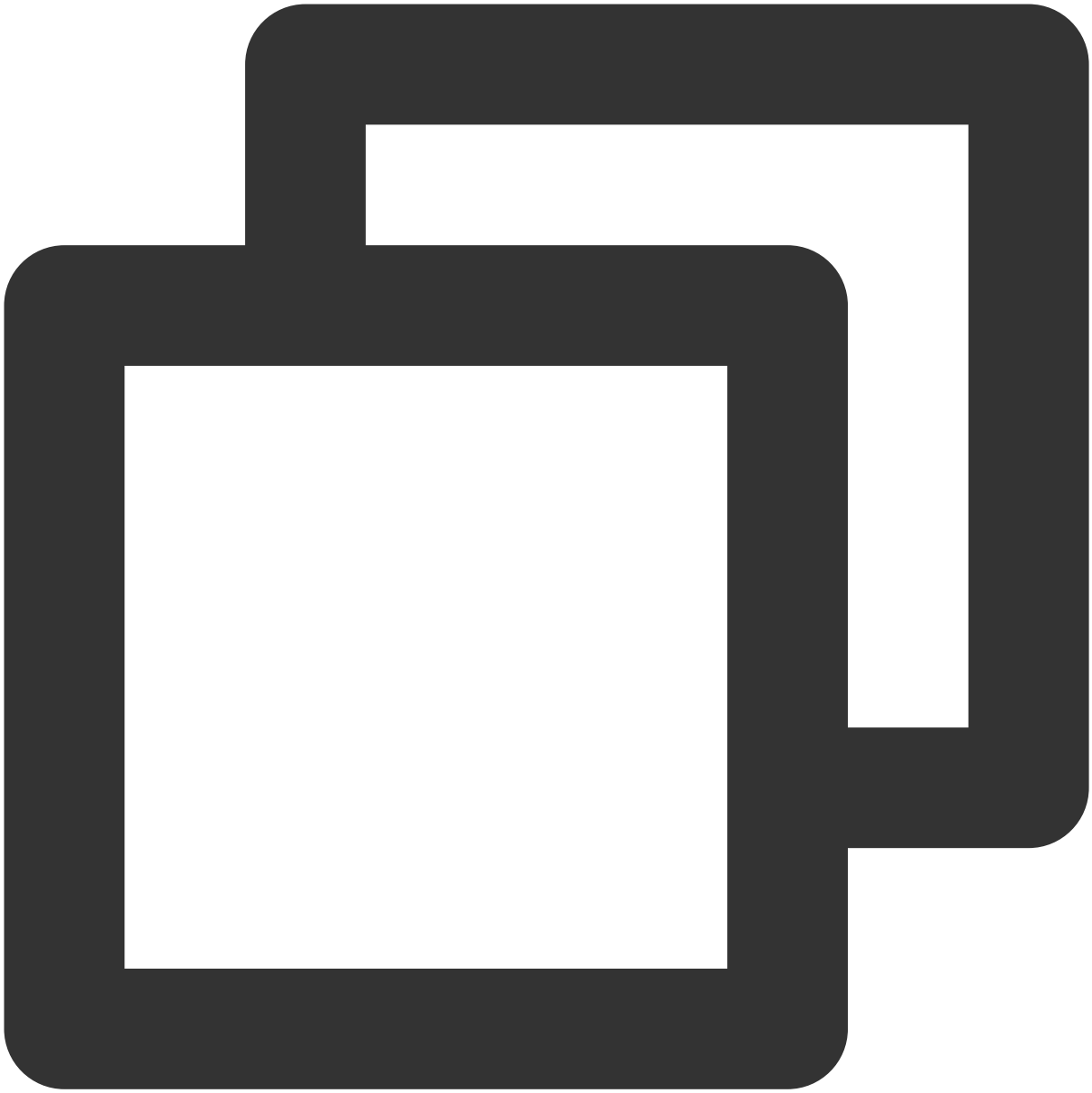
```
# 抓取任务名称，同时会在对应抓取的指标中加了一个 label(job=job_name)
job_name: <job_name>

# 抓取任务时间间隔
[ scrape_interval: <duration> | default = <global_config.scrape_interval> ]

# 抓取请求超时时间
[ scrape_timeout: <duration> | default = <global_config.scrape_timeout> ]

# 抓取任务请求 URI 路径
```

```
[ metrics_path: <path> | default = /metrics ]

# 解决当抓取的 label 与后端 Prometheus 添加 label 冲突时的处理。
# true: 保留抓取到的 label，忽略与后端 Prometheus 冲突的 label；
# false: 对冲突的 label，把抓取的 label 前加上 exported_<original-label>，添加后端 Promet
[ honor_labels: <boolean> | default = false ]

# 是否使用抓取到 target 上产生的时间。
# true: 如果 target 中有时间，使用 target 上的时间；
# false: 直接忽略 target 上的时间；
[ honor_timestamps: <boolean> | default = true ]

# 抓取协议：http 或者 https
[ scheme: <scheme> | default = http ]

# 抓取请求对应 URL 参数
params:
  [ <string>: [<string>, ...] ]

# 通过 basic auth 设置抓取请求头中 `Authorization` 的值，password/password_file 互斥，优先
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# 通过 bearer token 设置抓取请求头中 `Authorization` bearer_token/bearer_token_file 互斥
[ bearer_token: <secret> ]

# 通过 bearer token 设置抓取请求头中 `Authorization` bearer_token/bearer_token_file 互斥
[ bearer_token_file: <filename> ]

# 抓取连接是否通过 TLS 安全通道，配置对应的 TLS 参数
tls_config:
  [ <tls_config> ]

# 通过代理服务来抓取 target 上的指标，填写对应的代理服务地址。
[ proxy_url: <string> ]

# 通过静态配置来指定 target，详见下面的说明。
static_configs:
  [ - <static_config> ... ]

# CVM 服务发现配置，详见下面的说明。
cvm_sd_configs:
  [ - <cvm_sd_config> ... ]

# 在抓取数据之后，把 target 上对应的 label 通过 relabel 的机制进行改写，按顺序执行多个 relabe
```

```
# relabel_config 详见下面说明。
relabel_configs:
  [ - <relabel_config> ... ]

# 数据抓取完成写入之前，通过 relabel 机制进行改写 label 对应的值，按顺序执行多个 relabel 规则。
# relabel_config 详见下面说明。
metric_relabel_configs:
  [ - <relabel_config> ... ]

# 一次抓取数据点限制，0：不作限制，默认为 0
[ sample_limit: <int> | default = 0 ]

# 一次抓取 Target 限制，0：不作限制，默认为 0
[ target_limit: <int> | default = 0 ]
```

static_config 配置

相应配置项说明如下：



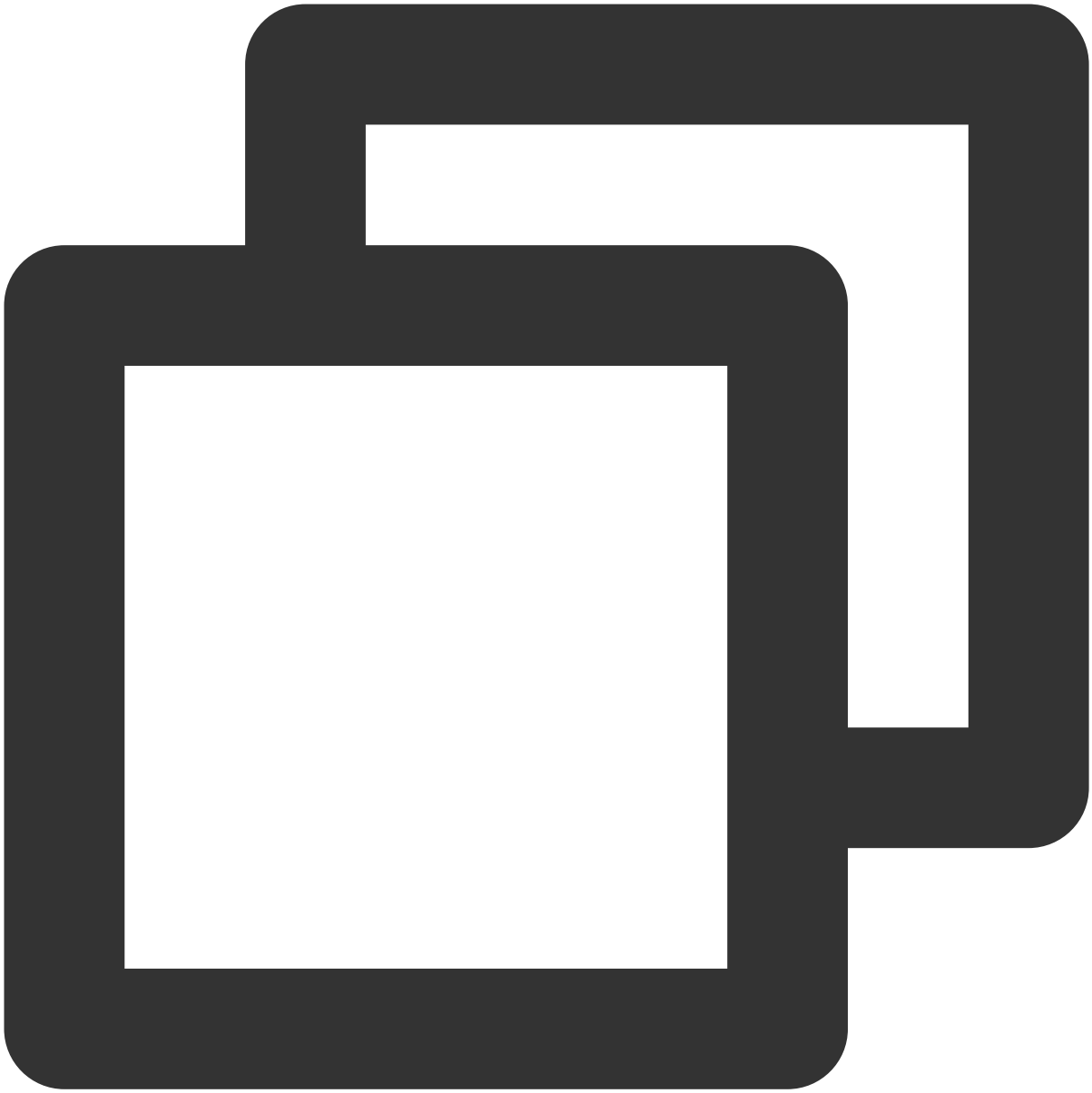
```
# 指定对应 target host 的值，如ip:port。
targets:
  [ - '<host>' ]

# 在所有 target 上加上对应的 label，类似全局 label 的概念。
labels:
  [ <labelname>: <labelvalue> ... ]
```

cvm_sd_config 配置

CVM 服务发现利用腾讯云 API 自动获取 CVM 实例列表，默认使用 CVM 的私网 IP。服务发现产生以下元标签，这些标签可以在 relabel 配置中使用。

标签	说明
__meta_cvm_instance_id	实例 ID
__meta_cvm_instance_name	实例名
__meta_cvm_instance_state	实例状态
__meta_cvm_instance_type	实例机型
__meta_cvm_OS	实例操作系统
__meta_cvm_private_ip	私网 IP
__meta_cvm_public_ip	公网 IP
__meta_cvm_vpc_id	网络 ID
__meta_cvm_subnet_id	子网 ID
__meta_cvm_tag_<tagkey>	实例标签值
__meta_cvm_region	实例所在区域
__meta_cvm_zone	实例的可用区

CVM 服务发现配置说明：



```
# 腾讯云的地域，地域列表见文档 https://www.tencentcloud.com/document/product/213/31574。
region: <string>

# 自定义 endpoint。
[ endpoint: <string> ]

# 访问腾讯云 API 的的凭证信息。如果不设置，取环境变量 TENCENT_CLOUD_SECRET_ID 和 TENCENT_CL
# 如使用集成中心的 CVM 抓取任务进行配置，则无需填写。
[ secret_id: <string> ]
[ secret_key: <secret> ]
```

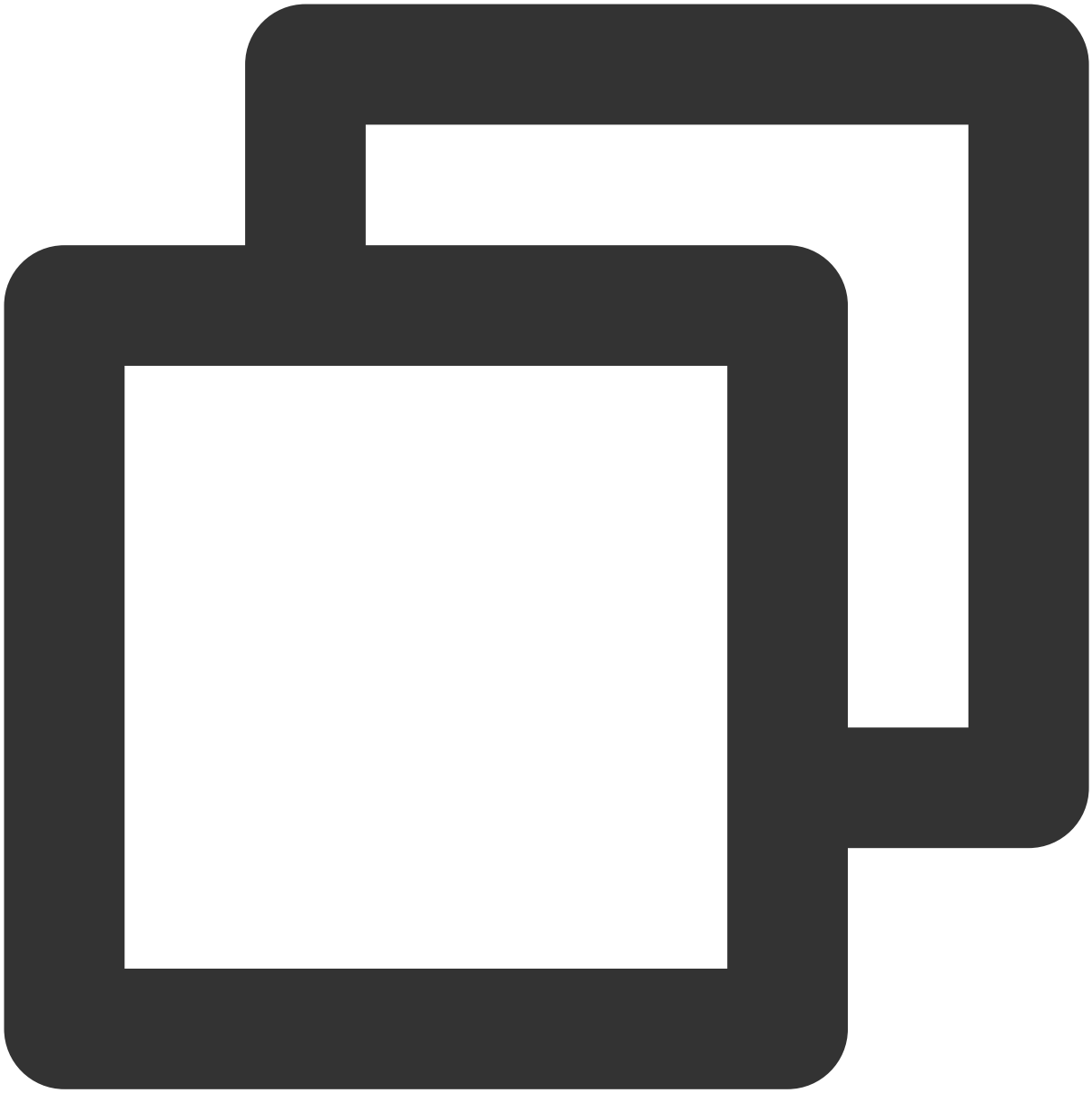
```
# CVM 列表的刷新周期。
[ refresh_interval: <duration> | default = 60s ]

# 抓取 metrics 的端口。
ports:
  - [ <int> | default = 80 ]

# CVM 列表的过滤规则。支持的过滤条件见文档 https://www.tencentcloud.com/document/product/
filters:
  [ - name: <string>
      values: <string>, [...] ]
```

说明：

使用集成中心的 CVM 抓取任务配置 cvm_sd_configs 时，该集成自动使用服务预设角色授权确保安全性，无需您手动填写如下参数：secret_id、secret_key、endpoint。

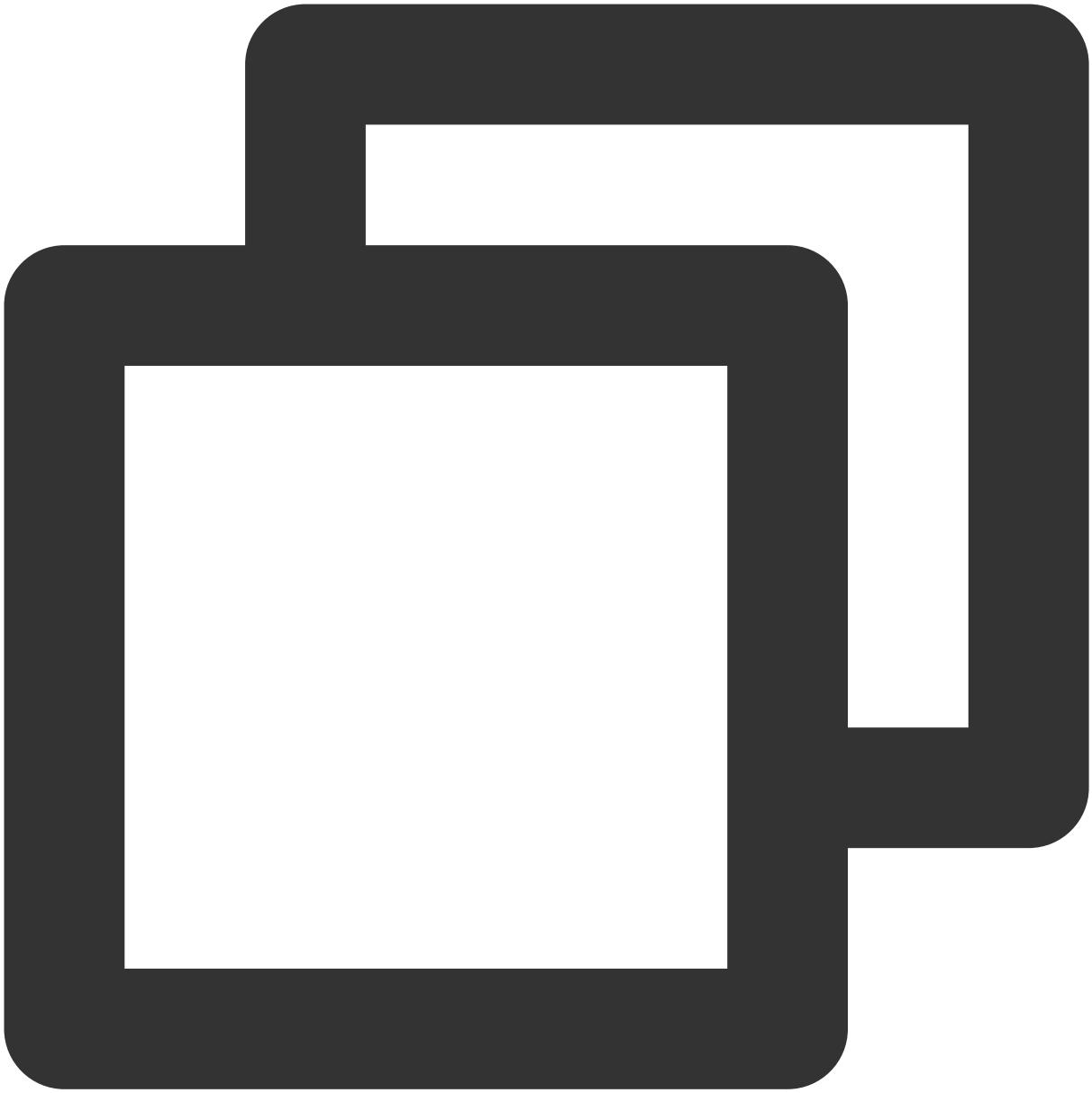
例子

静态配置



```
job_name: prometheus
scrape_interval: 30s
static_configs:
- targets:
  - 127.0.0.1:9090
```

CVM 服务发现配置

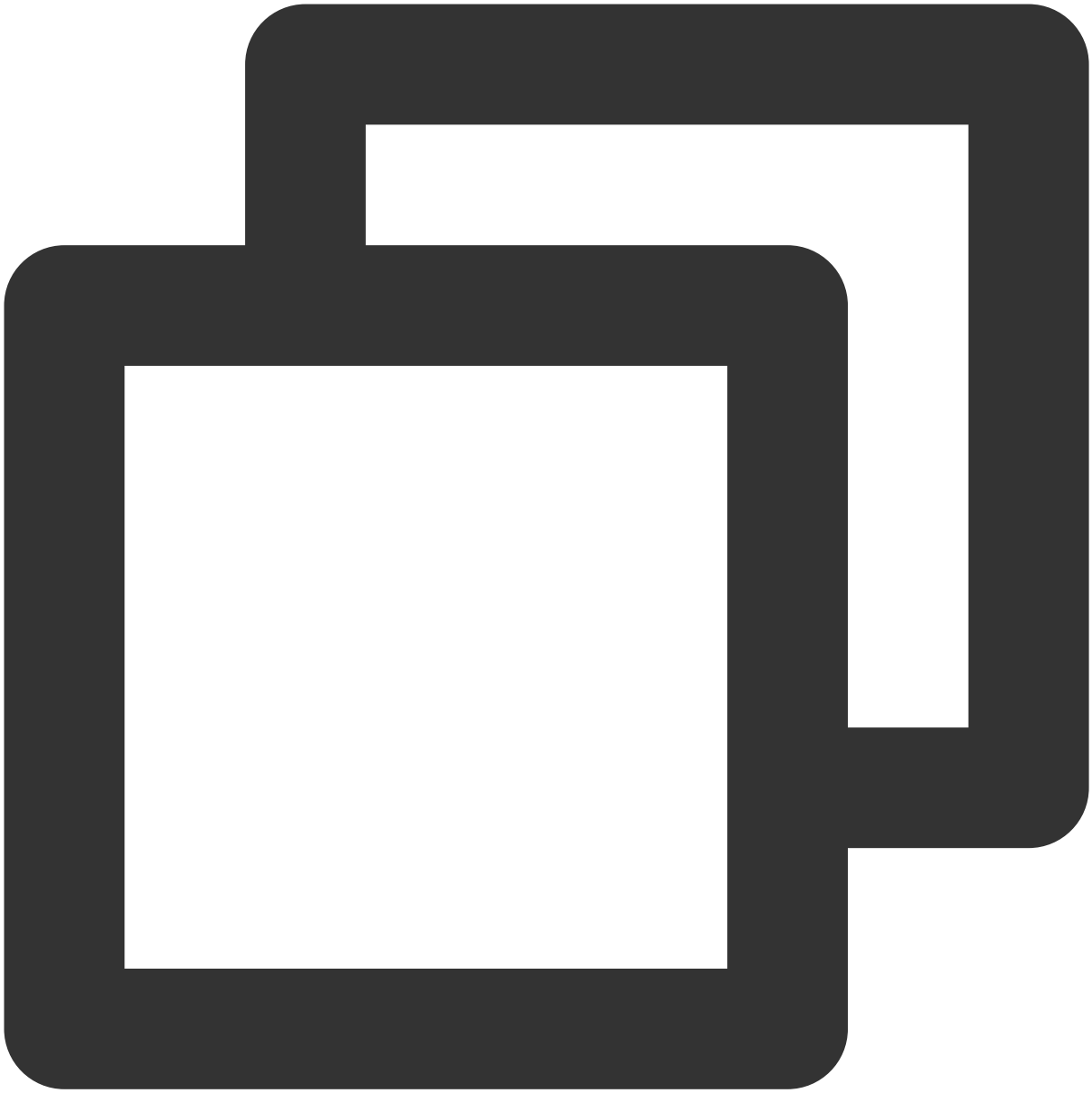


```
job_name: demo-monitor
cvm_sd_configs:
- region: ap-guangzhou
  ports:
  - 8080
  filters:
  - name: tag:service
    values:
    - demo
relabel_configs:
- source_labels: [__meta_cvm_instance_state]
```

```
    regex: RUNNING
    action: keep
  - regex: __meta_cvm_tag_(.*)
    replacement: $1
    action: labelmap
  - source_labels: [__meta_cvm_region]
    target_label: region
    action: replace
```

Pod Monitor

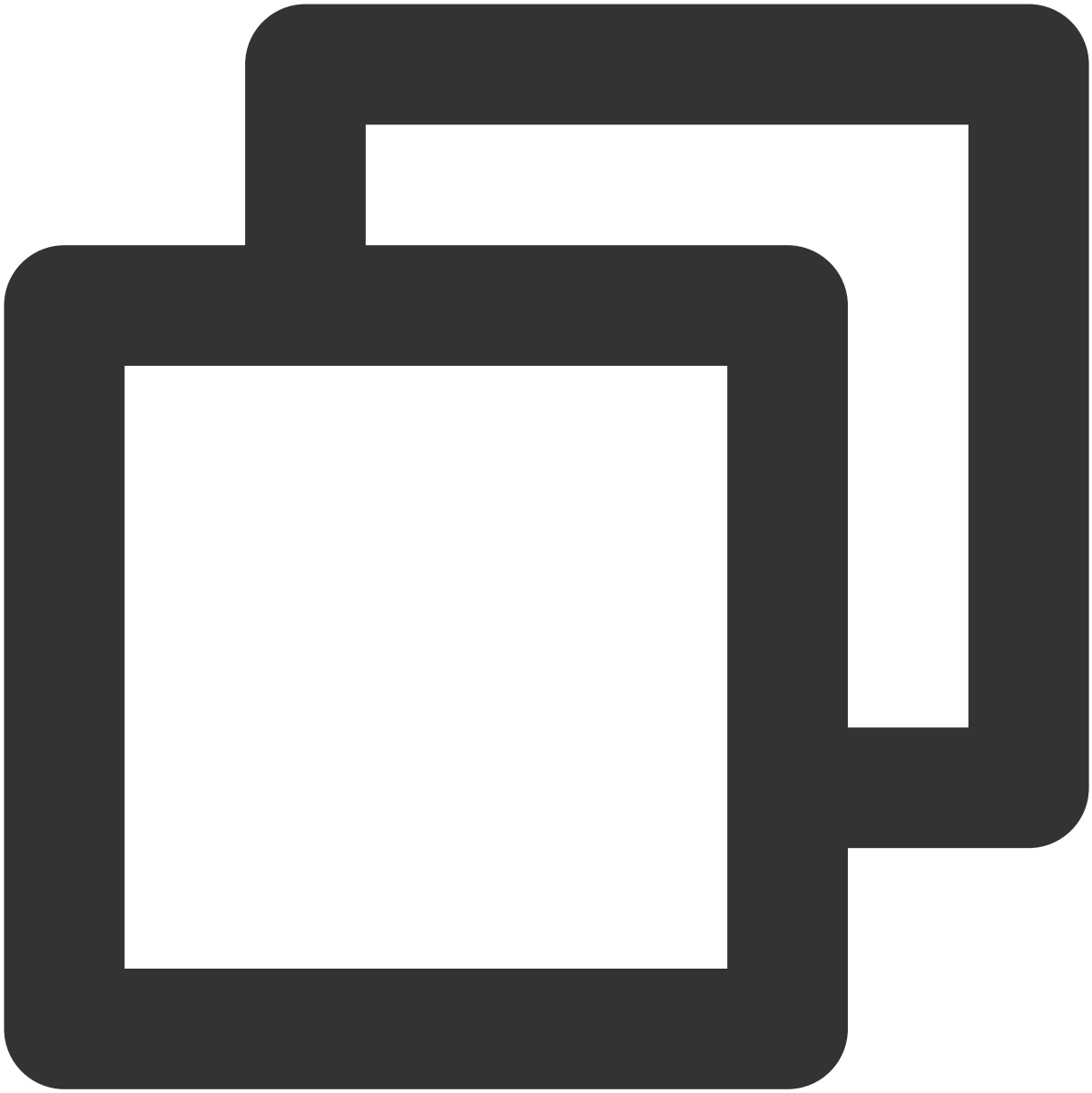
相应配置项说明如下：



```
# Prometheus Operator CRD 版本
apiVersion: monitoring.coreos.com/v1
# 对应 K8S 的资源类型，这里面 Pod Monitor
kind: PodMonitor
# 对应 K8S 的 Metadata，这里只用关心 name，如果没有指定 jobLabel，对应抓取指标 label 中 job
metadata:
  name: redis-exporter # 填写一个唯一名称
  namespace: cm-prometheus  # namespace固定，不需要修改
# 描述抓取目标 Pod 的选取及抓取任务的配置
spec:
  # 填写对应 Pod 的 label，pod monitor 会取对应的值作为 job label 的值。
```

```
  # 如果查看的是 Pod Yaml，取 pod.metadata.labels 中的值。
  # 如果查看的是 Deployment/Daemonset/Statefulset，取 spec.template.metadata.labels。
  [ jobLabel: string ]
  # 把对应 Pod 上的 Label 添加到 Target 的 Label 中
  [ podTargetLabels: []string ]
  # 一次抓取数据点限制，0：不作限制，默认为 0
  [ sampleLimit: uint64 ]
  # 一次抓取 Target 限制，0：不作限制，默认为 0
  [ targetLimit: uint64 ]
  # 配置需要抓取暴露的 Prometheus HTTP 接口，可以配置多个 Endpoint
  podMetricsEndpoints:
  [ - <endpoint_config> ... ] # 详见下面 endpoint 说明
  # 选择要监控 Pod 所在的 namespace，不填为选取所有 namespace
  [ namespaceSelector: ]
    # 是否选取所有 namespace
    [ any: bool ]
    # 需要选取 namespace 列表
    [ matchNames: []string ]
  # 填写要监控 Pod 的 Label 值，以定位目标 Pod  [K8S metav1.LabelSelector](https://kube
  selector:
    [ matchExpressions: array ]
      [ example: - {key: tier, operator: In, values: [cache]} ]
    [ matchLabels: object ]
      [ example: k8s-app: redis-exporter ]
```

例子

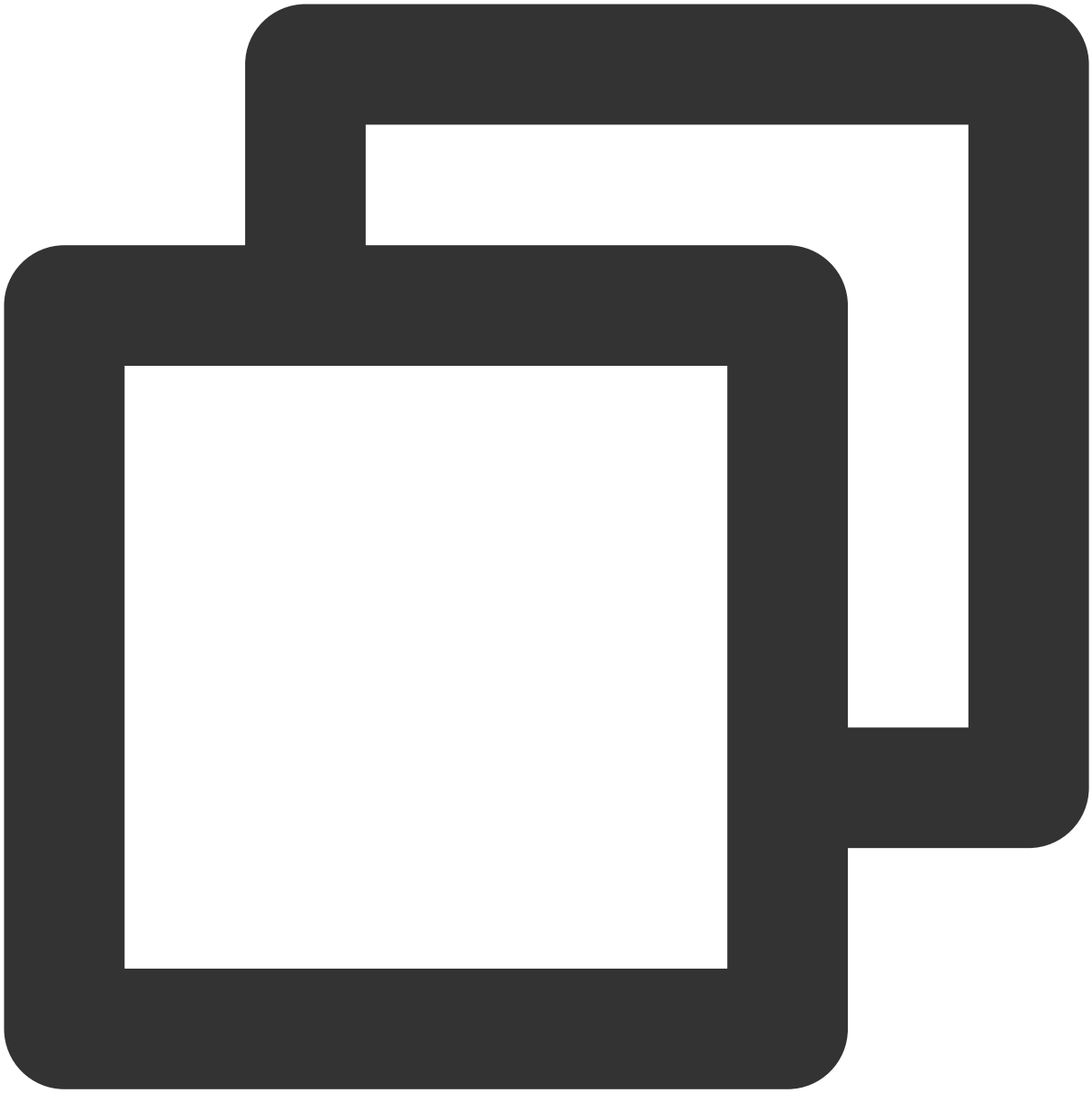


```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: redis-exporter # 填写一个唯一名称
  namespace: cm-prometheus  # namespace固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port  # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics  # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
```

```
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: instance
      replacement: 'crs-xxxxxx' # 调整成对应的 Redis 实例 ID
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: ip
      replacement: '1.x.x.x' # 调整成对应的 Redis 实例 IP
  namespaceSelector:    # 选择要监控pod所在的namespace
    matchNames:
    - redis-test
  selector:     # 填写要监控pod的Label值，以定位目标pod
    matchLabels:
      k8s-app: redis-exporter
```

Service Monitor

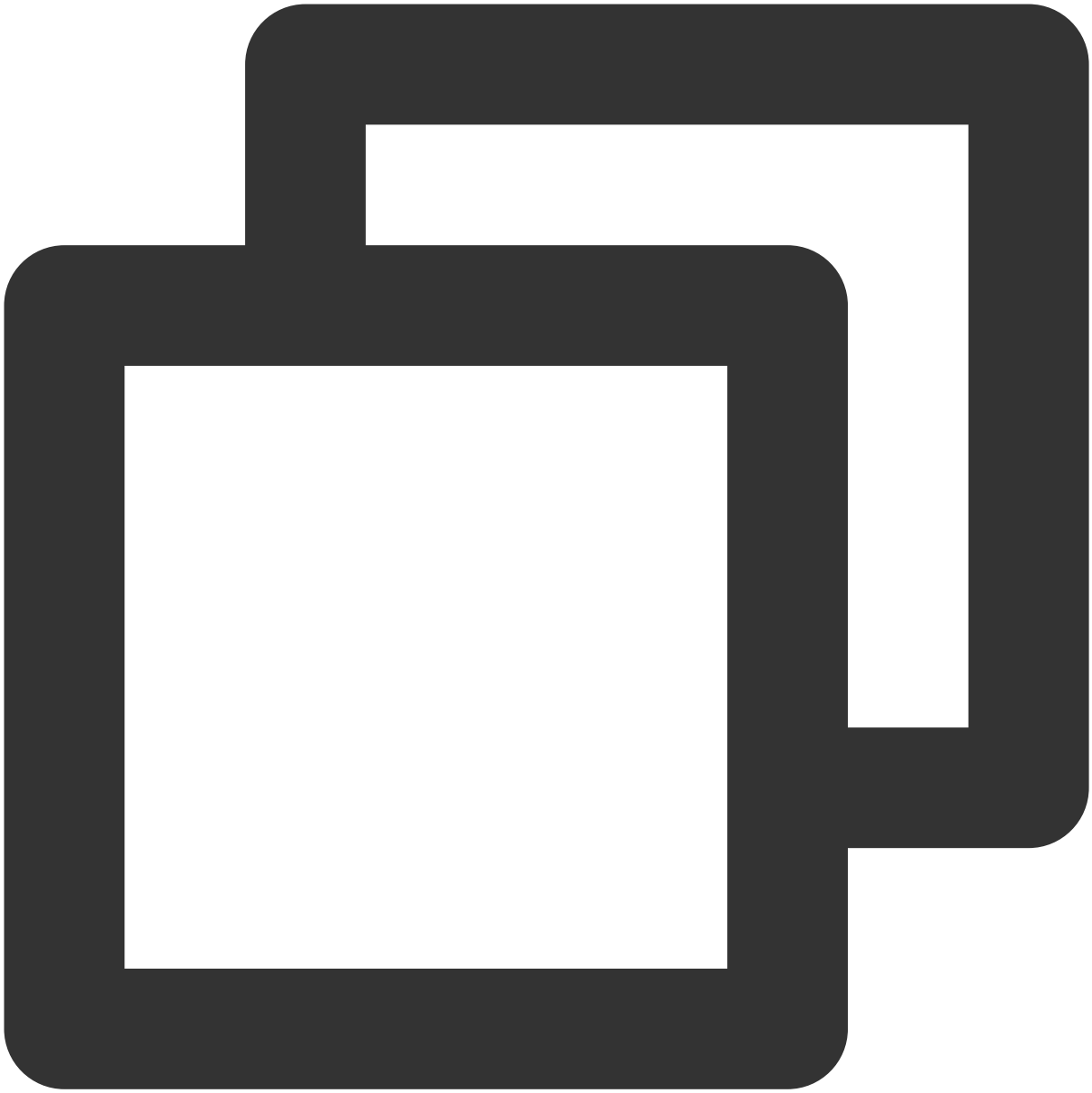
相应配置项说明如下：



```
# Prometheus Operator CRD 版本
apiVersion: monitoring.coreos.com/v1
# 对应 K8S 的资源类型，这里面 Service Monitor
kind: ServiceMonitor
# 对应 K8S 的 Metadata，这里只用关心 name，如果没有指定 jobLabel，对应抓取指标 label 中 job
metadata:
  name: redis-exporter # 填写一个唯一名称
  namespace: cm-prometheus  # namespace固定，不需要修改
# 描述抓取目标 Pod 的选取及抓取任务的配置
spec:
```

```
  # 填写对应 Pod 的 label(metadata/labels), service monitor 会取对应的值作为 job label 的
  [ jobLabel: string ]
  # 把对应 service 上的 Label 添加到 Target 的 Label 中
  [ targetLabels: []string ]
  # 把对应 Pod 上的 Label 添加到 Target 的 Label 中
  [ podTargetLabels: []string ]
  # 一次抓取数据点限制, 0：不作限制, 默认为 0
  [ sampleLimit: uint64 ]
  # 一次抓取 Target 限制, 0：不作限制, 默认为 0
  [ targetLimit: uint64 ]
  # 配置需要抓取暴露的 Prometheus HTTP 接口, 可以配置多个 Endpoint
  endpoints:
  [ - <endpoint_config> ... ] # 详见下面 endpoint 说明
  # 选择要监控 Pod 所在的 namespace, 不填为选取所有 namespace
  [ namespaceSelector: ]
    # 是否选取所有 namespace
    [ any: bool ]
    # 需要选取 namespace 列表
    [ matchNames: []string ]
  # 填写要监控 Pod 的 Label 值, 以定位目标 Pod  [K8S metav1.LabelSelector](https://v1-1
  selector:
    [ matchExpressions: array ]
      [ example: - {key: tier, operator: In, values: [cache]} ]
    [ matchLabels: object ]
      [ example: k8s-app: redis-exporter ]
```

例子

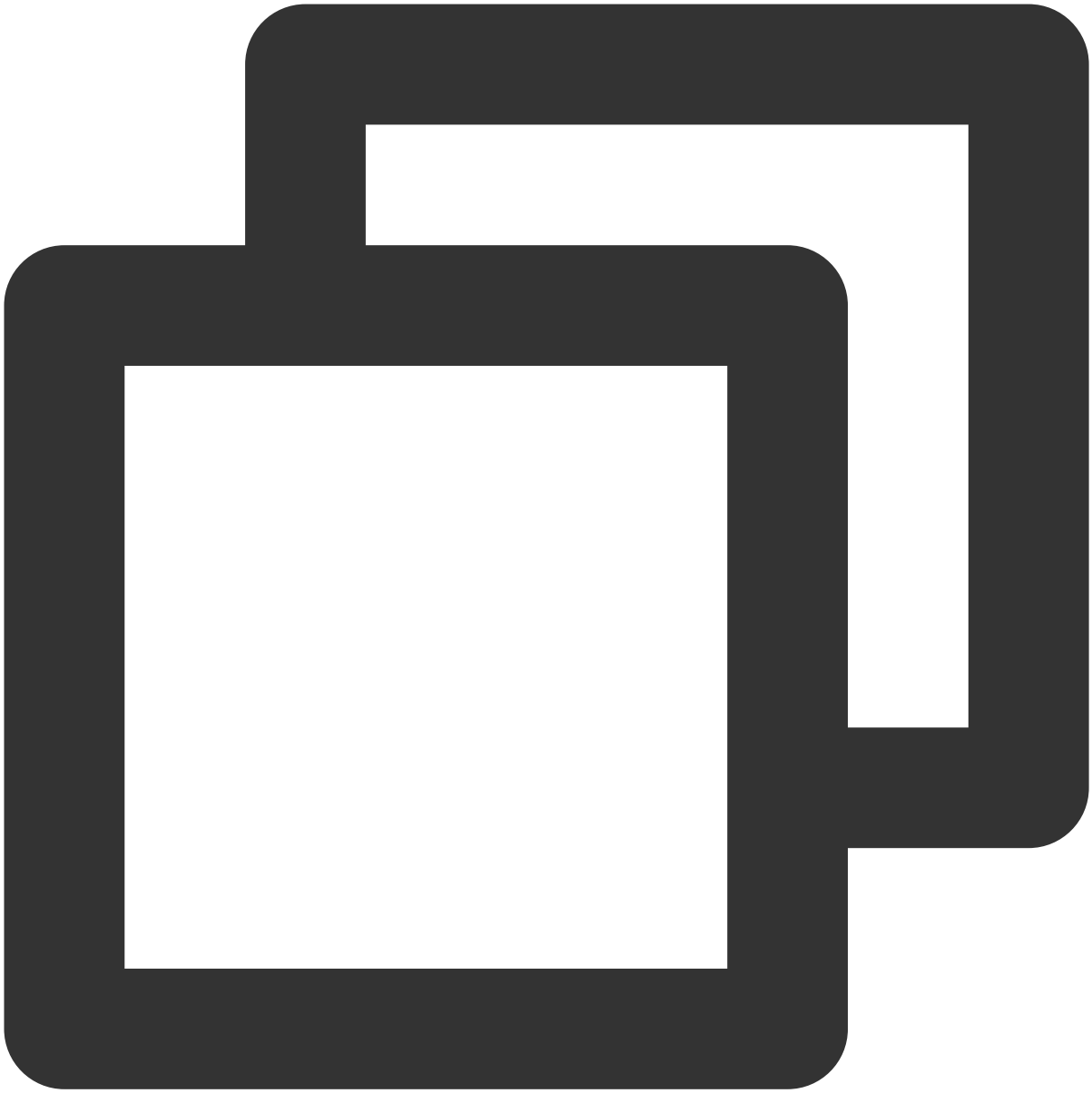


```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: go-demo     # 填写一个唯一名称
  namespace: cm-prometheus   # namespace固定，不要修改
spec:
  endpoints:
  - interval: 30s
    # 填写service yaml中Prometheus Exporter对应的Port的Name
    port: 8080-8080-tcp
    # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
```

```
    path: /metrics
    relabelings:
    # ** 必须要有一个 label 为 application，这里假设 k8s 有一个 label 为 app，
    # 我们通过 relabel 的 replace 动作把它替换成了 application
    - action: replace
      sourceLabels:  [__meta_kubernetes_pod_label_app]
      targetLabel: application
  # 选择要监控service所在的namespace
  namespaceSelector:
    matchNames:
    - golang-demo
  # 填写要监控service的Label值，以定位目标service
  selector:
    matchLabels:
      app: golang-app-demo
```

endpoint_config 配置

相应配置项说明如下：

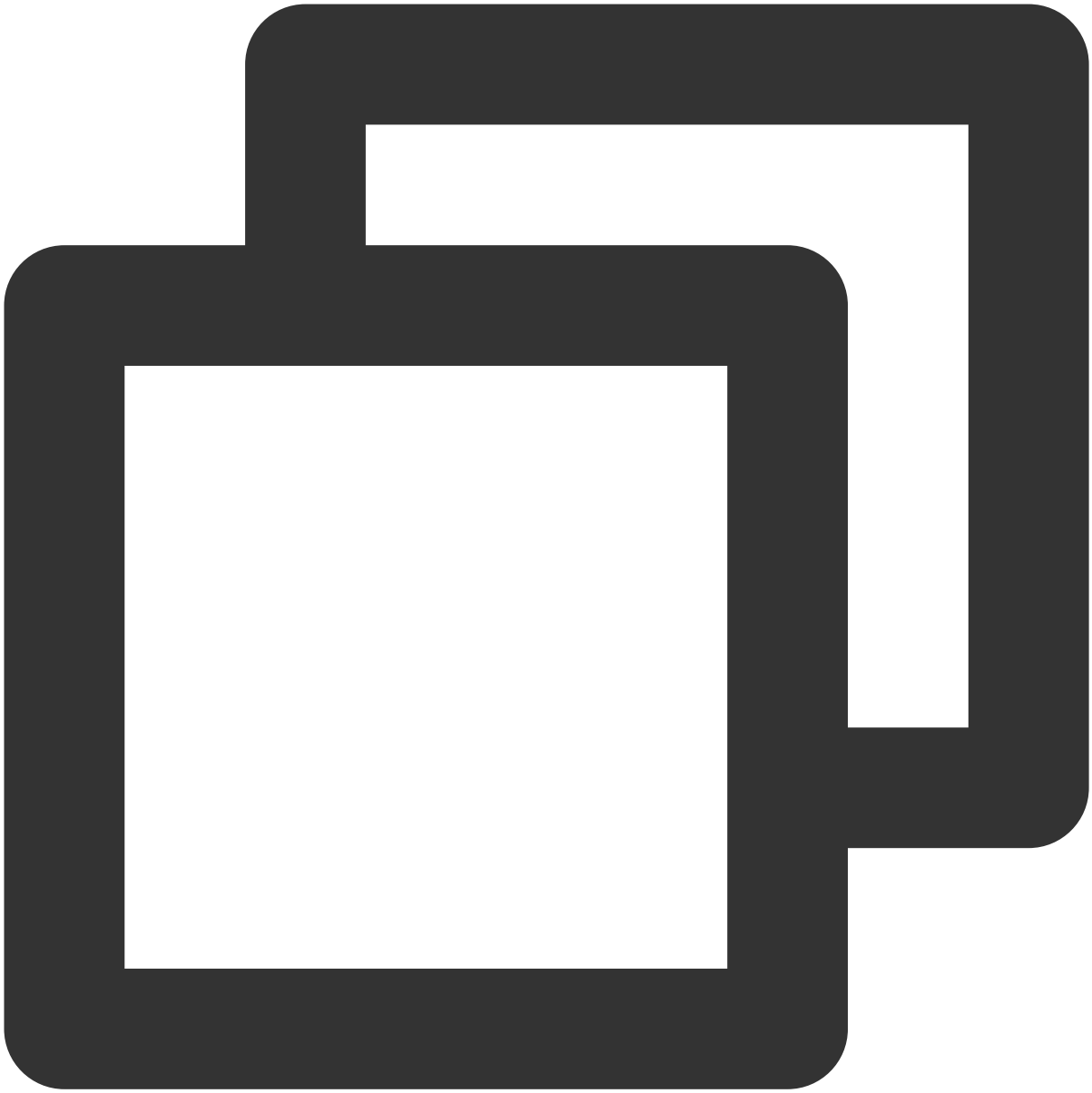


```
# 对应 port 的名称，这里需要注意不是对应的端口，默认：80，对应的取值如下：
# ServiceMonitor: 对应 Service>spec/ports/name;
# PodMonitor: 说明如下：
#   如果查看的是 Pod Yaml，取 pod.spec.containers.ports.name 中的值。
#   如果查看的是 Deployment/Daemonset/Statefulset，取 spec.template.spec.containers.po
[ port: string | default = 80]
# 抓取任务请求 URI 路径
[ path: string | default = /metrics ]
# 抓取协议：http 或者 https
[ scheme: string | default = http]
# 抓取请求对应 URL 参数
```

```
[ params: map[string][]string]
# 抓取任务间隔的时间
[ interval: string | default = 30s ]
# 抓取任务超时
[ scrapeTimeout: string | default = 30s]
# 抓取连接是否通过 TLS 安全通道，配置对应的 TLS 参数
[ tlsConfig: TLSConfig ]
# 通过对应的文件读取 bearer token 对应的值，放到抓取任务的 header 中
[ bearerTokenFile: string ]
# 通过对应的 K8S secret key 读取对应的 bearer token，注意 secret namespace 需要和 PodMon
[ bearerTokenSecret: string ]
# 解决当抓取的 label 与后端 Prometheus 添加 label 冲突时的处理。
# true: 保留抓取到的 label，忽略与后端 Prometheus 冲突的 label；
# false: 对冲突的 label，把抓取的 label 前加上 exported_<original-label>，添加后端 Promet
[ honorLabels: bool | default = false ]
# 是否使用抓取到 target 上产生的时间。
# true: 如果 target 中有时间，使用 target 上的时间；
# false: 直接忽略 target 上的时间；
[ honorTimestamps: bool | default = true ]
# basic auth 的认证信息，username/password 填写对应 K8S secret key 的值，注意 secret nam
[ basicAuth: BasicAuth ]
# 通过代理服务来抓取 target 上的指标，填写对应的代理服务地址。
[ proxyUrl: string ]
# 在抓取数据之后，把 target 上对应的 label 通过 relabel 的机制进行改写，按顺序执行多个 relabe
# relabel_config 详见下面说明。
relabelings:
[ - <relabel_config> ...]
# 数据抓取完成写入之前，通过 relabel 机制进行改写 label 对应的值，按顺序执行多个 relabel 规则。
# relabel_config 详见下面说明。
metricRelabelings:
[ - <relabel_config> ...]
```

relabel_config 配置

相应配置项说明如下：



```
# 从原始 labels 中取哪些 label 的值进行 relabel，取出来的值通过 separator 中的定义进行字符拼
# 如果是 PodMonitor/ServiceMonitor 对应的配置项为 sourceLabels 。
[ source_labels: '[' <labelname> [, ...] ']' ]
# 定义需要 relabel 的 label 值拼接的字符，默认为 ';'。
[ separator: <string> | default = ; ]

# action 为 replace/hashmod 时，通过 target_label 来指定对应 label name。
# 如果是 PodMonitor/ServiceMonitor 对应的配置项为 targetLabel 。
[ target_label: <labelname> ]
```

```
# 需要对 source labels 对应值进行正则匹配的表达式。
[ regex: <regex> | default = (.*) ]

# action 为 hashmod 时用到，根据 source label 对应值 md5 取模值。
[ modulus: <int> ]

# action 为 replace 的时候，通过 replacement 来定义当 regex 匹配之后需要替换的表达式，可以结
[ replacement: <string> | default = $1 ]

# 基于 regex 匹配到的值进行相关的操作，对应的 action 如下，默认为 replace：
# replace: 如果 regex 匹配到，通过 replacement 中定义的值替换相应的值，并通过 target_label
# keep: 如果 regex 没有匹配到，丢弃
# drop: 如果 regex 匹配到，丢弃
# hashmod: 通过 moduels 指定的值把 source label 对应的 md5 值取模，添加一个新的 label，lab
# labelmap: 如果 regex 匹配到，使用 replacement 替换对就的 label name
# labeldrop: 如果 regex 匹配到，删除对应的 label
# labelkeep: 如果 regex 没有匹配到，删除对应的 label
[ action: <relabel_action> | default = replace ]
```

自定义监控

最近更新时间：2024-01-29 15:55:14

操作场景

您可以通过 Prometheus 监控服务自定上报指标监控数据，对应用或者服务内部的一些状态进行监控，如请求处理数，下单数等，也可以对一些核心逻辑的处理耗时进行监控，如请求外部服务的耗时情况等。
本文以 Go 这个语言为例，介绍如何通过 Prometheus 监控服务进行业务自定义指标上报，可视化及告警。

支持开发语言

Prometheus 开源社区官方 SDK:
Go
Java or Scala
Python
Ruby
其它第3方开发语言 SDK:
Bash
C
C++
Common Lisp
Dart
Elixir
Erlang
Haskell
Lua for Nginx
Lua for Tarantool
.NET / C#
Node.js
Perl
PHP
R
Rust
更多信息请参考。

数据模型

Prometheus 具有多维分析的能力，数据模型有如下几部分组成。

`Metric Name(标指名称)` + `Labels(标签)` + `Timestamp(时间戳)` + `Value/Sample(监控值/样品)`

Metric Name(标指名称)：监控对象的含义（例如，http_request_total - 表示当前系统接收到的HTTP请求总量）。

标签(label)：表示当前样本的特征维度，是一个K/V结构，通过这些维度 Prometheus 可以对样本数据进行过滤，聚合等。

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。

`Metric Name(指标名称)/Labels(标签)` 只能由ASCII字符、数字、下划线以及冒号组成并必须符合正则表达式[a-zA-Z_:][a-zA-Z0-9_:]*。

更多 Data Model 说明

Metric/Label 命名最佳实践

如何监控埋点

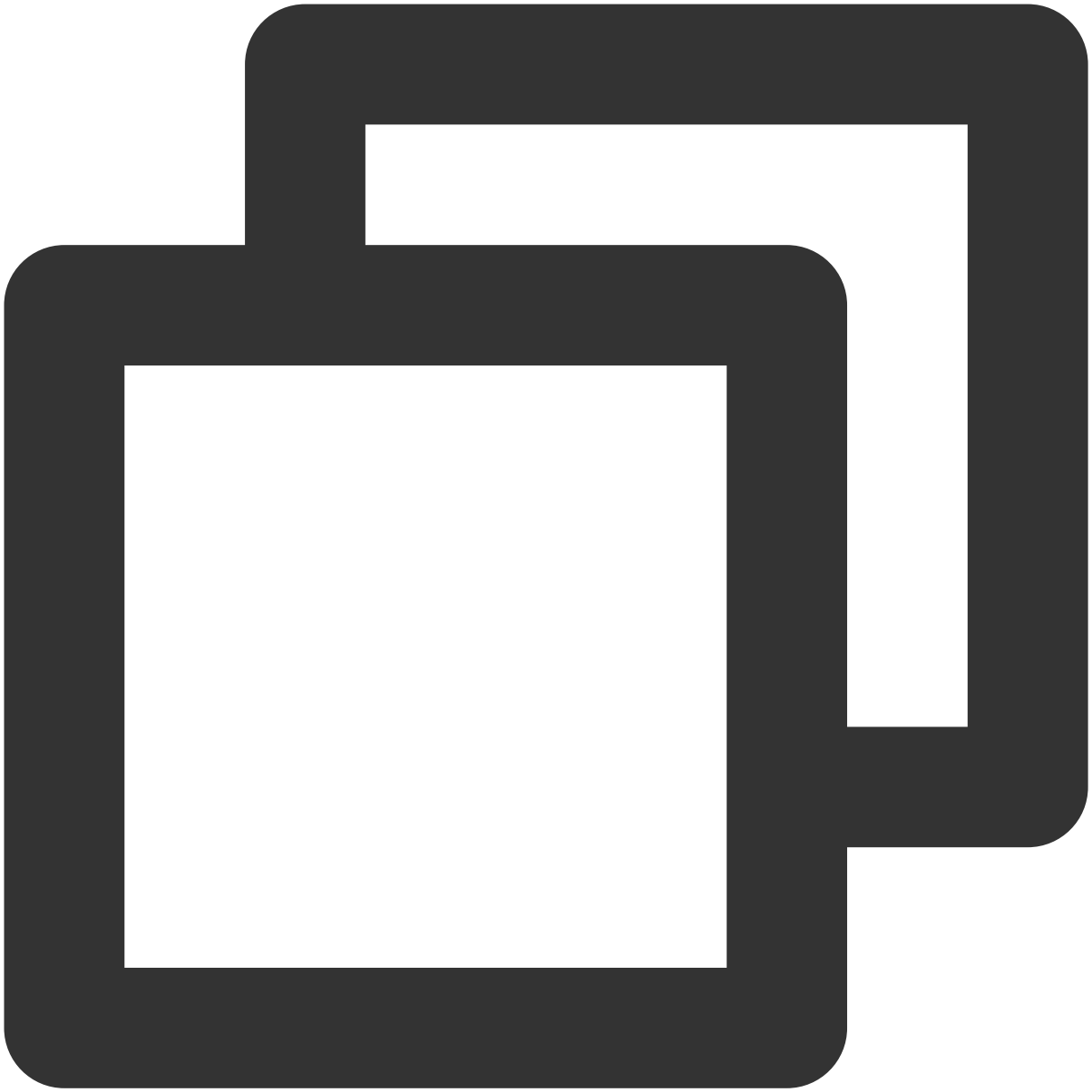
Prometheus 根据监控的不同场景提供了 `Counter` / `Gauge` / `Historgram` / `Summary` 四种指标类型，每种指标类型说明可参考下文。更多说明请参考 Prometheus 官网 METRIC TYPES。

Prometheus 社区提供了多种开发语言的 SDK，每种语言的使用方法基本上类似，主要是开发语言语法上的区别，下面主要以 Go 作为例子如何上报自定义监控指标数据。

Counter

计数类型，数据是单调递增的指标，服务重启之后会重置。可以用 Counter 来监控请求数/异常数/用户登录数/订单数等。

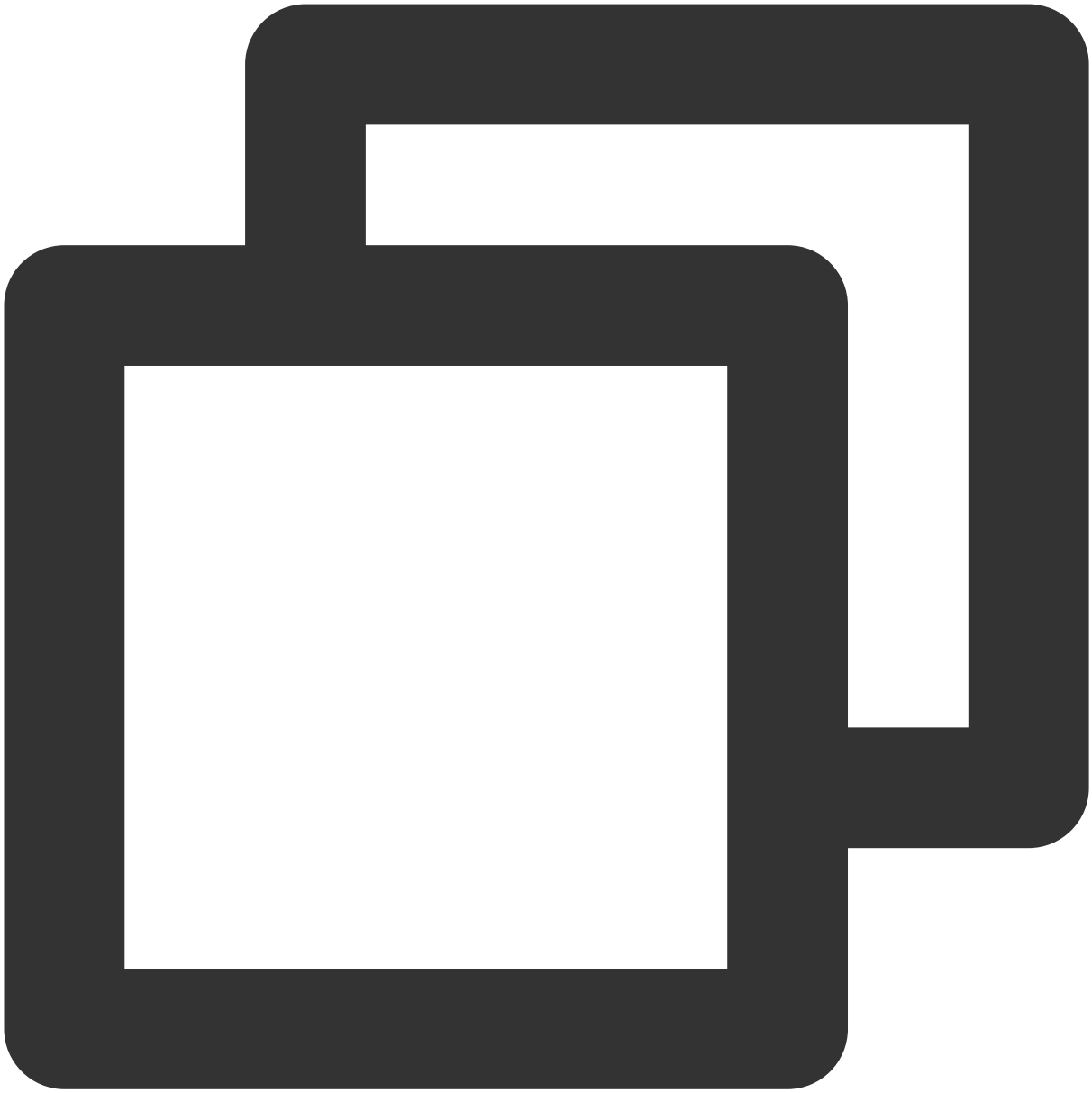
如何通过 `Counter` 来监控订单数：



```
package order

import (
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
)

// 定义需要监控 Counter 类型对象
var (
    opsProcessed = promauto.NewCounterVec(prometheus.CounterOpts{
        Name: "order_service_processed_orders_total",
```

```
        Help: "The total number of processed orders",
    }, []string{"status"}) // 处理状态
)

// 订单处理
func makeOrder() {
    opsProcessed.WithLabelValues("success").Inc() // 成功状态
    // opsProcessed.WithLabelValues("fail").Inc() // 失败状态

    // 下单的业务逻辑
}
```

例如，通过 rate() 函数获取订单的增长率：

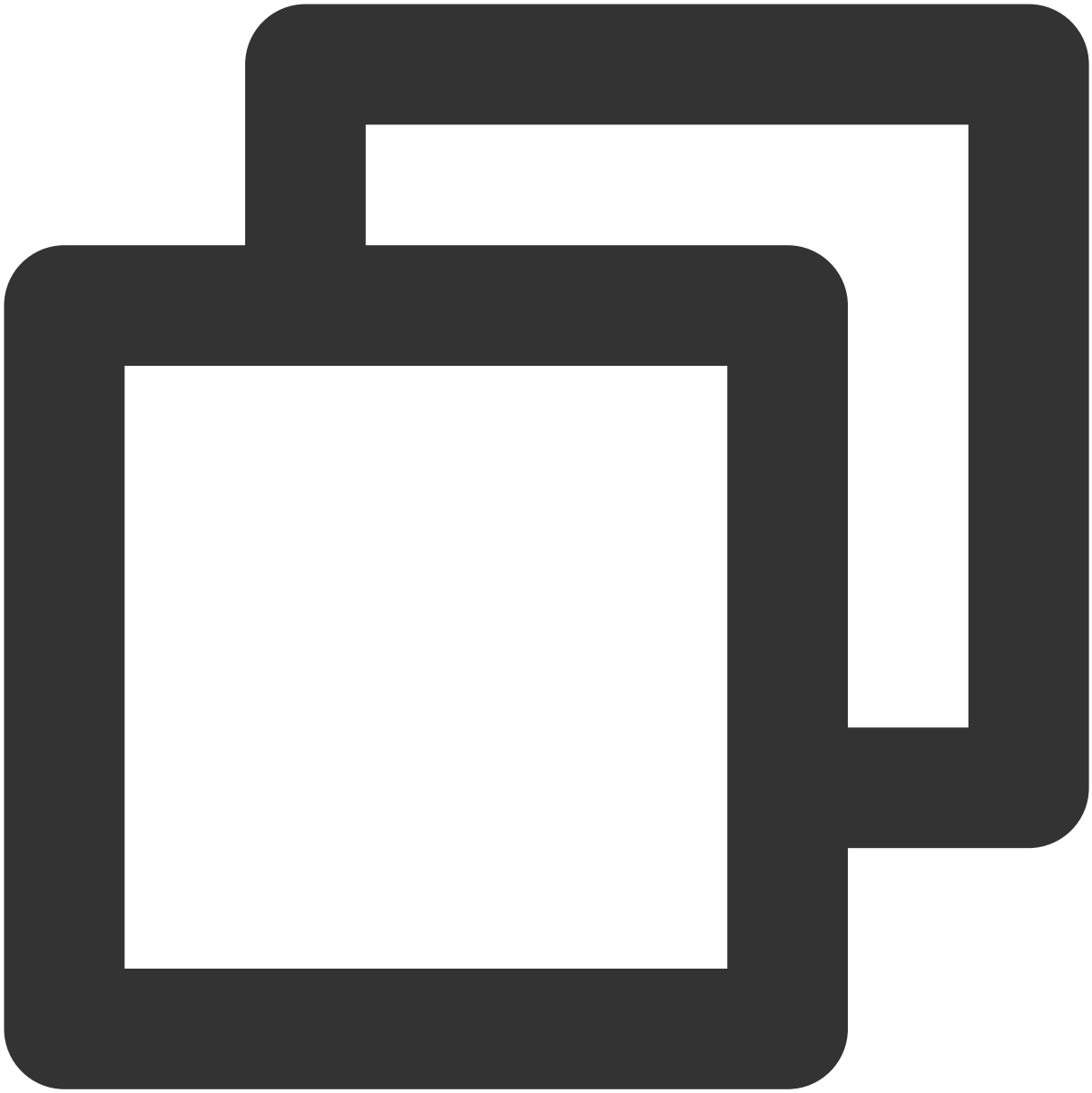


```
rate(order_service_processed_orders_total[5m])
```

Gauge

当前值，监控打点的时候可对其做加减。可以用 Gauge 来监控当前内存使用率 /CPU 使用率/当前线程数/队列个数等。

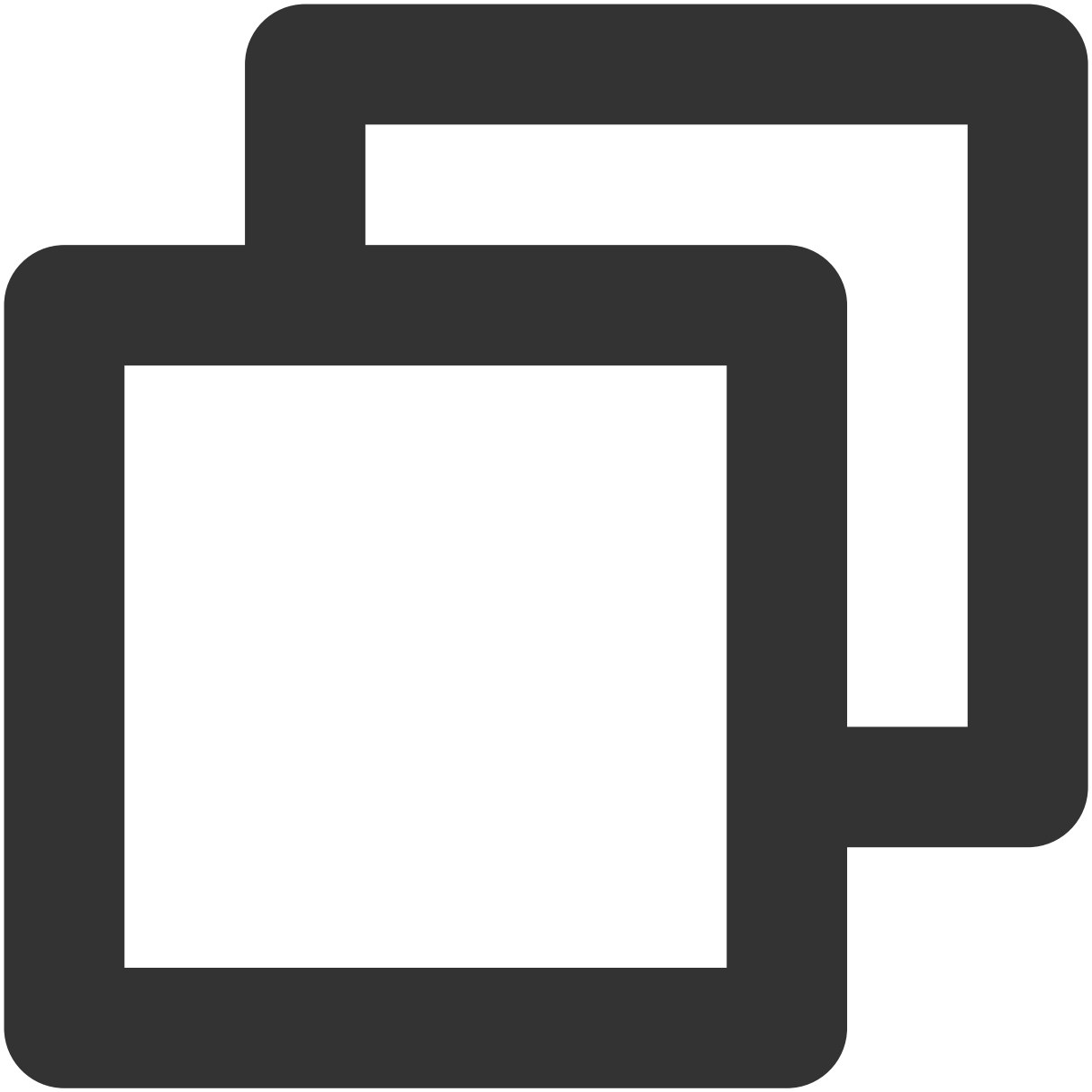
如何通过 `Gauge` 来监控订单队列大小：



```
package order

import (
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
)

// 定义需要监控 Gauge 类型对象
var (
    queueSize = promauto.NewGaugeVec(prometheus.GaugeOpts{
        Name: "order_service_order_queue_size",
```

```
        Help: "The size of order queue",
    }, []string{"type"})
)

type OrderQueue struct {
    queue chan string
}

func newOrderQueue() *OrderQueue  {
    return &OrderQueue{
        queue: make(chan string,100),
    }
}

// 产生订单消息
func (q *OrderQueue)produceOrder() {
    // 产生订单消息

    // 队列个数加1
    queueSize.WithLabelValues("make_order").Inc() // 下单队列
    // queueSize.WithLabelValues("cancel_order").Inc() // 取消订单队列
}

// 消费订单消息
func (q *OrderQueue)consumeOrder() {
    // 消费订单消息

    // 队列个数减1
    queueSize.WithLabelValues("make_order").Dec()
}
```

通过 Gauge 指标，直接查看订单每种类型队列的当前大小：



```
order_service_order_queue_size
```

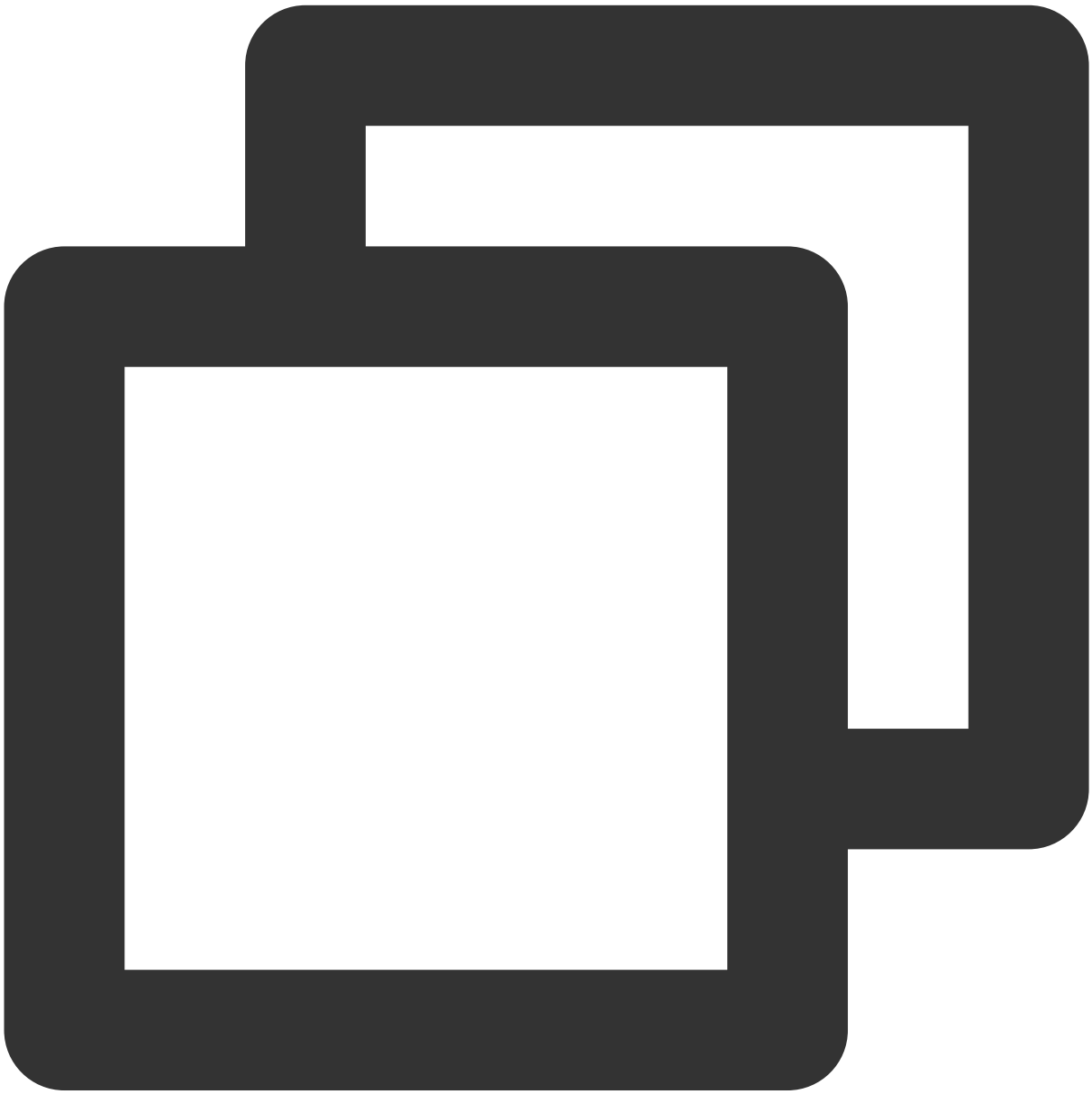
Histogram

直方图，Prometheus 会根据配置的 `Bucket` 来计算样本的分布情况，后期可以再加工，一般多用于耗时的监控，通过 Histogram 可以计算出 P99/P95/P50等耗时，同时也可以监控处理的个数，如果用上 Histogram 就不需要再用 Counter 统计个数。可以用 Histogram 来监控接口响应时间/数据库访问耗时等。
Histogram 和 Summary 的使用方式类似，可以直接参考 Summary 的使用方式。

Summary

摘要，和 Histogram 有一点类似，也是计算样本的分布情况，区别是 Summary 会在客户端计算出分布情况(P99/P95/Sum/Count)，因此也会更占客户端资源，后期不可再聚合计算处理，同样可以用 Summary 来监控接口响应时间/数据库访问耗时等。

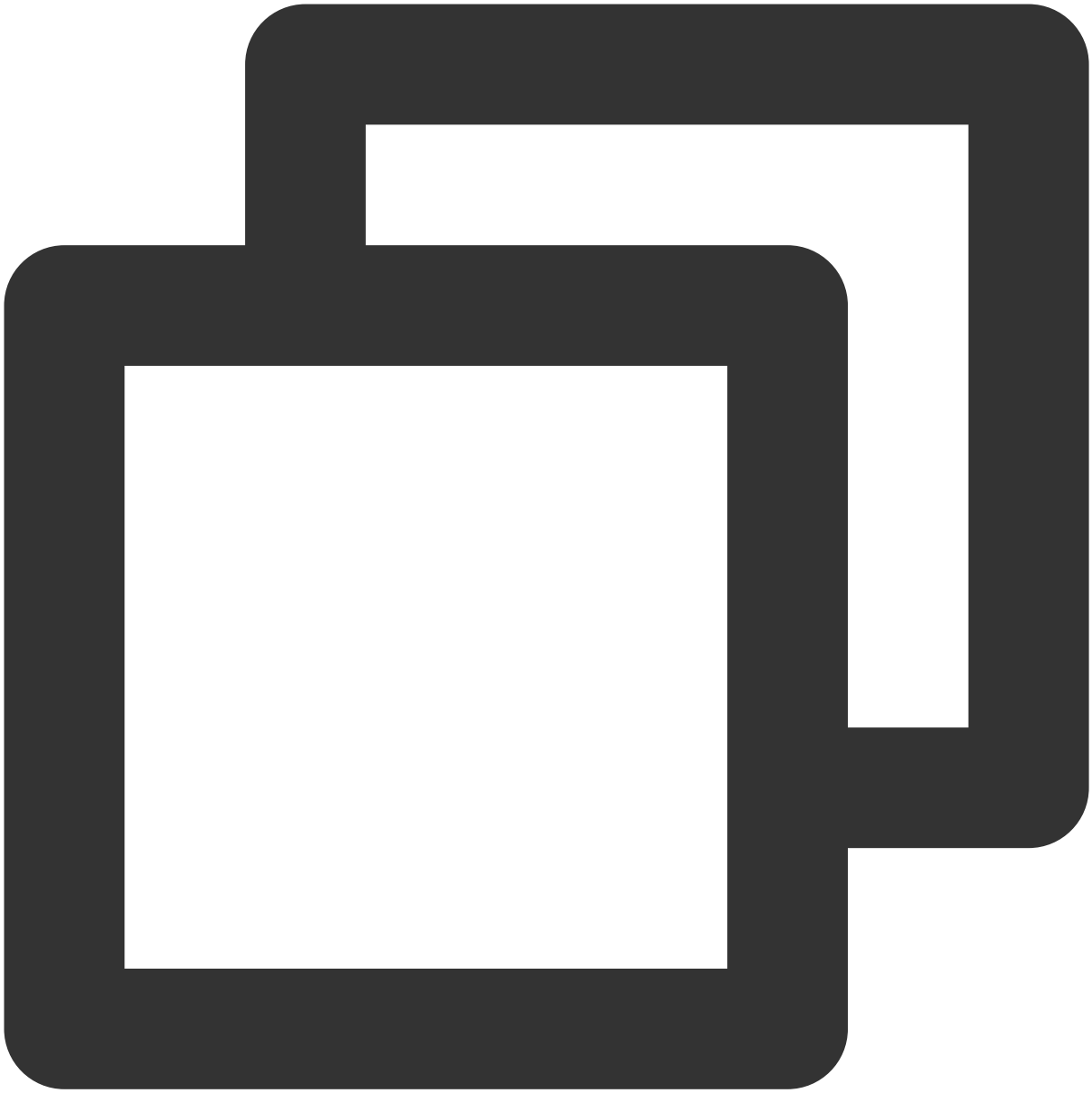
如何通过 `Summary` 来监控订单处理耗时：



```
package order

import (
    "net/http"
```

```
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

// 定义需要监控 Summary 类型对象
var (
    opsProcessCost = promauto.NewSummaryVec(prometheus.SummaryOpts{
        Name: "order_service_process_order_duration",
        Help: "The order process duration",
    }, []string{"status"})
)

func makeOrder() {
    start := time.Now().UnixNano()
    // 下单逻辑处理结束，记录处理耗时
    defer opsProcessCost.WithLabelValues("success").Observe((float64)(time.Now().Un

    // 下单的业务逻辑
    time.Sleep(time.Second) // 模拟处理耗时
}
```

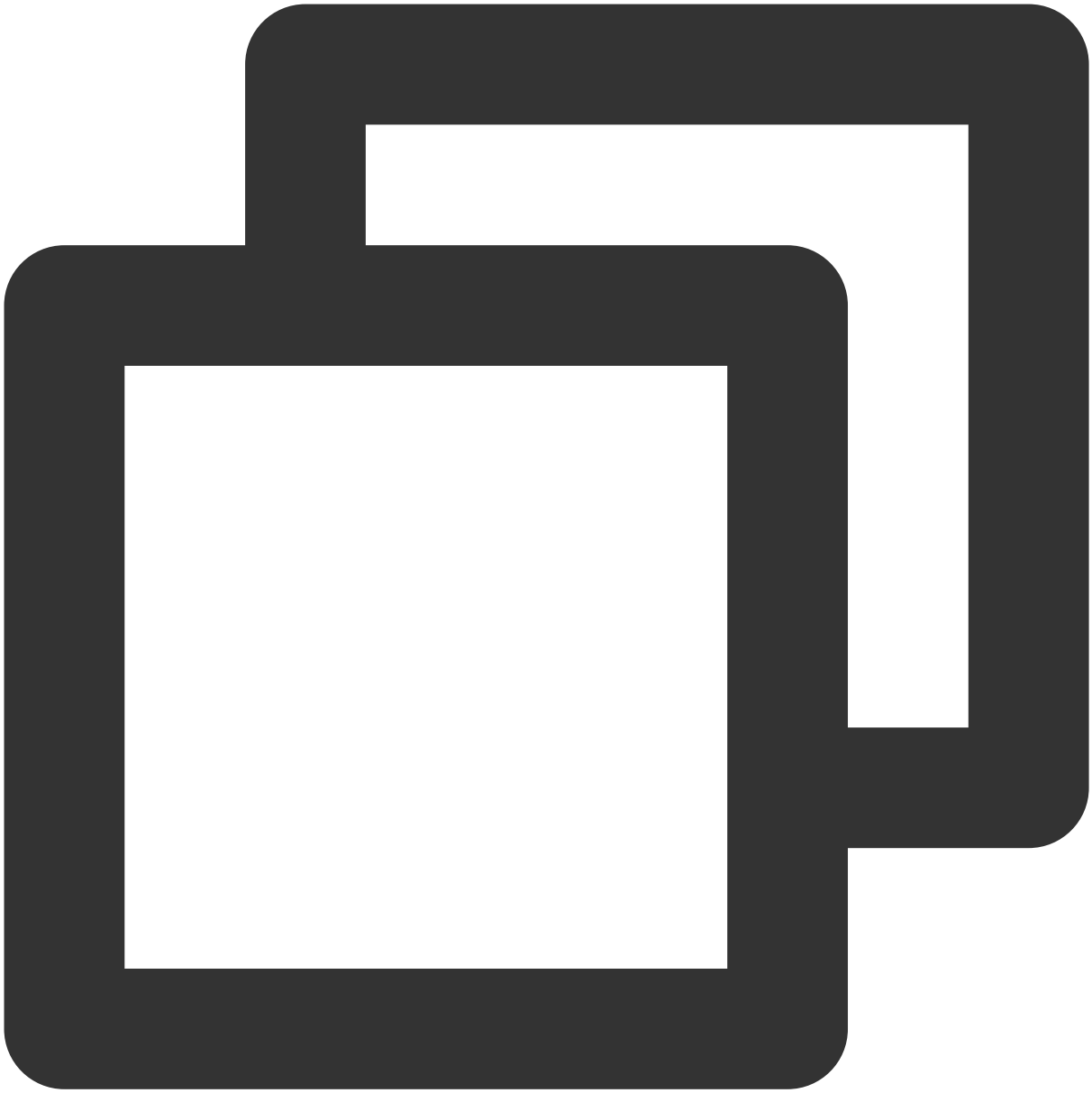
通过 Summary 指标，直接查看下单处理平均耗时：



```
order_service_processed_order_duration_sum / order_service_processed_order_duration
```

暴露 Prometheus 指标

通过 `promhttp.Handler()` 把监控埋点数据暴露到 HTTP 服务上。



```
package main

import (
    "net/http"

    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
        // 业务代码
```

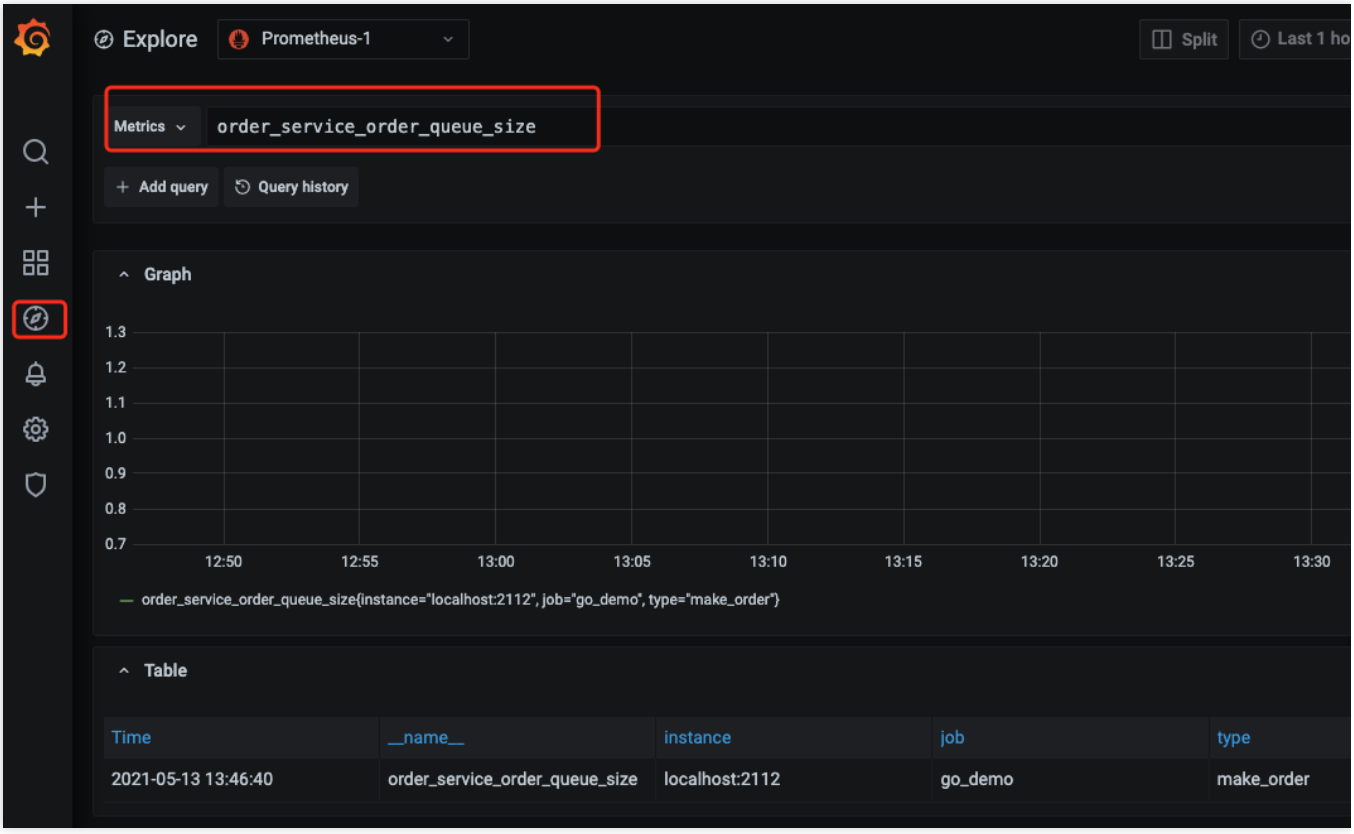
```
        // 把 Prometheus 指标暴露在 HTTP 服务上
        http.Handle("/metrics", promhttp.Handler())

        // 业务代码
}
```

采集数据

完成相关业务自定义监控埋点之后，应用发布，即可通过 Prometheus 来抓取监控指标数据。详情请参见Golang 接入。

查看监控数据和告警

打开 Prometheus 监控服务自带的 Grafana，通过 `Explore` 来查看监控指标数据，如下图，也可以 自定义 Grafana 监控大盘。



通过 `Prometheus` 和 `云监控告警` 的能力可以对自定义监控指标进行实时告警，详情请参见 告警介绍及使用。

EMR 接入
Flink 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 Flink 过程中需要对 Flink 任务运行状态进行监控，以便了解 Flink 任务是否正常运行，排查 Flink 故障等。Prometheus 监控服务对 push gateway 做了集成，支持 Flink 写入 metrics，并提供了开箱即用的 Grafana 监控大盘。

前提条件

1. 购买的腾讯云弹性 MapReduce（以下简称 EMR）产品包含 Flink 组件，并在实例上跑 Flink 任务。
2. 在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 托管版集群。

操作步骤

产品接入

获取 PushGateway 访问配置

1. 前往 **弹性 MapReduce** > **选择对应的“实例”** > **基本信息** > **实例信息**页面，获取 Pushgateway 地址和 Token。

2. 在 账号信息 页面获取 APPID。

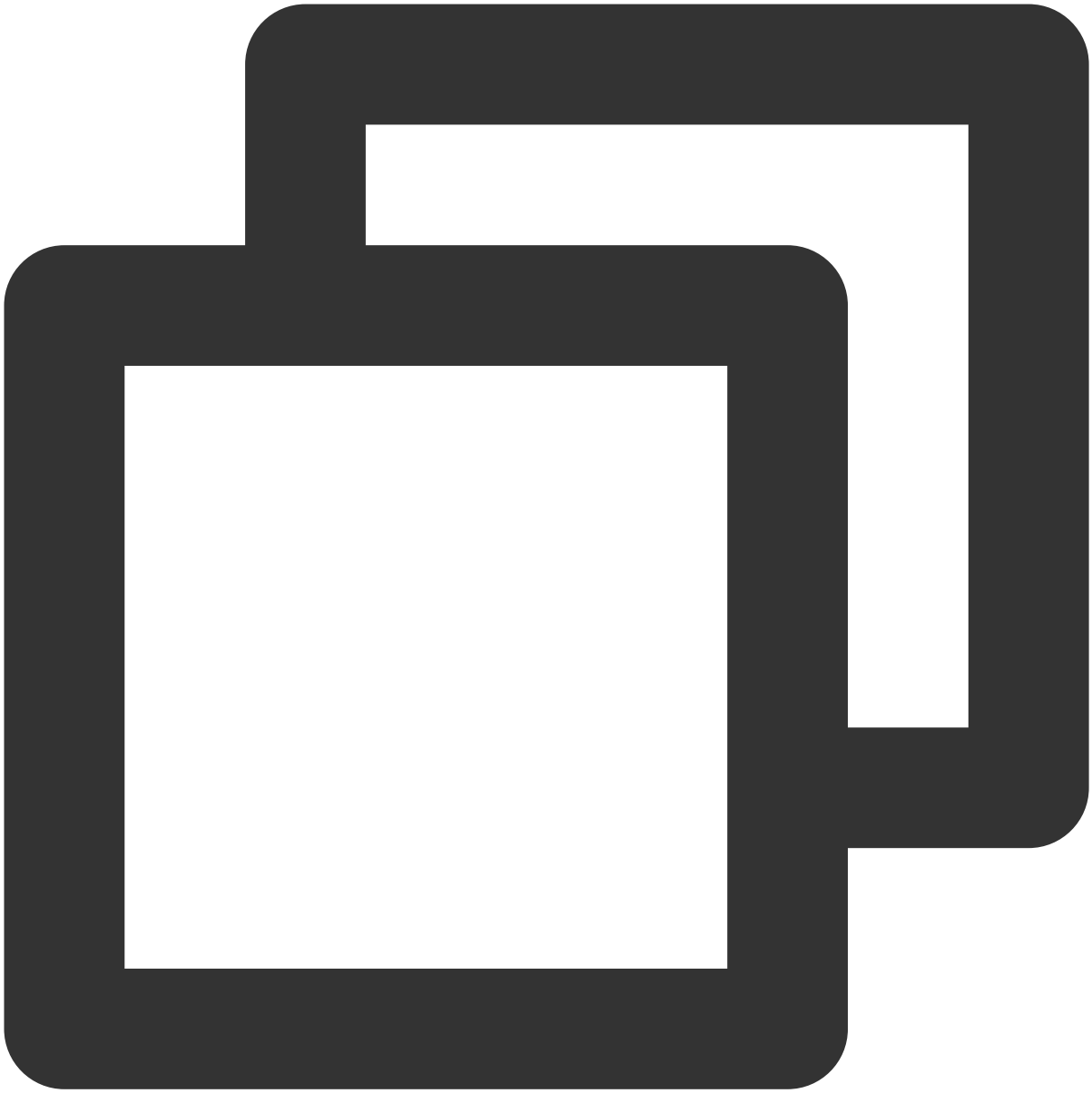
修改 Flink 配置

1. 进入 **弹性 MapReduce** > **选择对应的“实例”** > **集群服务**页面。
2. 找到 **Flink** 配置项，在右侧选择**操作** > **配置管理**，进入配置管理页面。
3. 在页面右侧单击**新增配置项**，依次添加以下配置。

配置名	默认	类型	描述	建议
metrics.reporter.promgateway.class	无	字符串	实现 metrics 导出到 push gateway 的 java 类名	-
metrics.reporter.promgateway.jobName	无	字符串	push 任务名	指定方便理解的字符串
metrics.reporter.promgateway.randomJobNameSuffix	true	布尔	是否在任务名后添加随机字符串	需设置为 true，如果不添加， Flink 任务间 metrics 会相互覆盖
metrics.reporter.promgateway.groupingKey	无	字符串	添加到每个 metrics 的全局 label，格式为 k1=v1;k2=v2	添加 EMR 实例 ID 方便区分不同实例的数据，例如 instance_id=emr-xxx

metrics.reporter.promgateway.interval	无	时间	推送 metrics 的时间间隔，例如30秒	建议设置在1分钟左右
metrics.reporter.promgateway.host	无	字符串	push gateway 的服务地址	控制台上 prometheus 实例的服务地址
metrics.reporter.promgateway.port	-1	整数	push gateway 服务端口	控制台上 prometheus 实例的服务端口
metrics.reporter.promgateway.needBasicAuth	false	布尔	push gateway 服务是否需要认证	设置为 true，prometheus 监控服务的 push gateway 需要认证
metrics.reporter.promgateway.user	无	字符串	认证的用户名	用户的 APPID
metrics.reporter.promgateway.password	无	字符串	认证的密码	控制台上 prometheus 实例的访问 Token
metrics.reporter.promgateway.deleteOnShutdown	true	布尔	Flink 任务执行完后是否删除 push gateway 上对应的 metrics	设置为 true

配置示例如下：

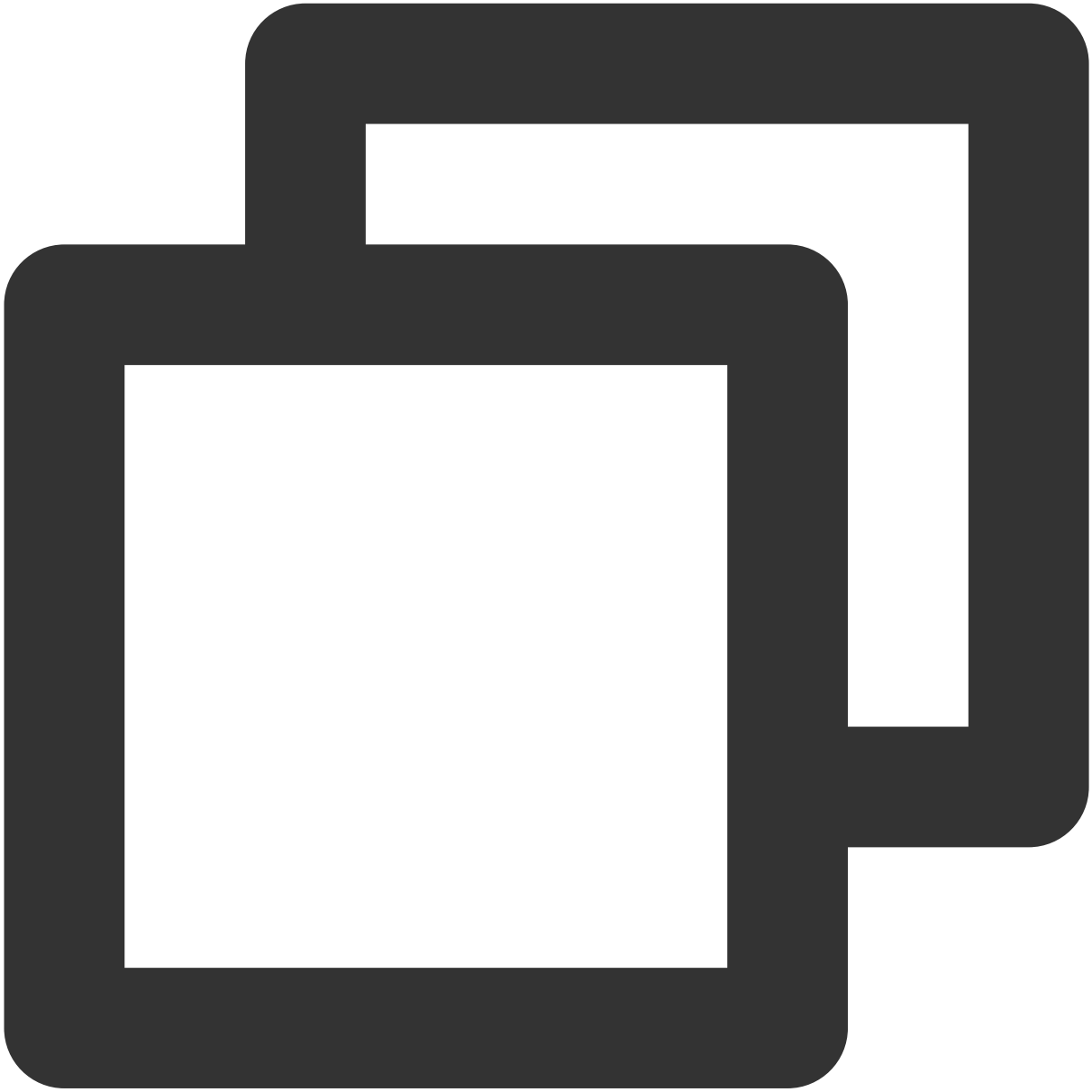


```
metrics.reporter.promgateway.class: org.apache.flink.metrics.prometheus.PrometheusP
metrics.reporter.promgateway.jobName: climatePredict
metrics.reporter.promgateway.randomJobNameSuffix:true
metrics.reporter.promgateway.interval: 60 SECONDS
metrics.reporter.promgateway.groupingKey:instance_id=emr-xxxx
metrics.reporter.promgateway.host: 172.xx.xx.xx
metrics.reporter.promgateway.port: 9090
metrics.reporter.promgateway.needBasicAuth: true
metrics.reporter.promgateway.user: appid
metrics.reporter.promgateway.password: token
```

安装 Flink PushGateway 插件

官方包中的 push gateway 插件目前还不支持配置认证信息，但是托管服务需要认证才允许写入，建议使用我们提供的 jar 包。我们也向 flink 官方提交了支持认证的 PR。

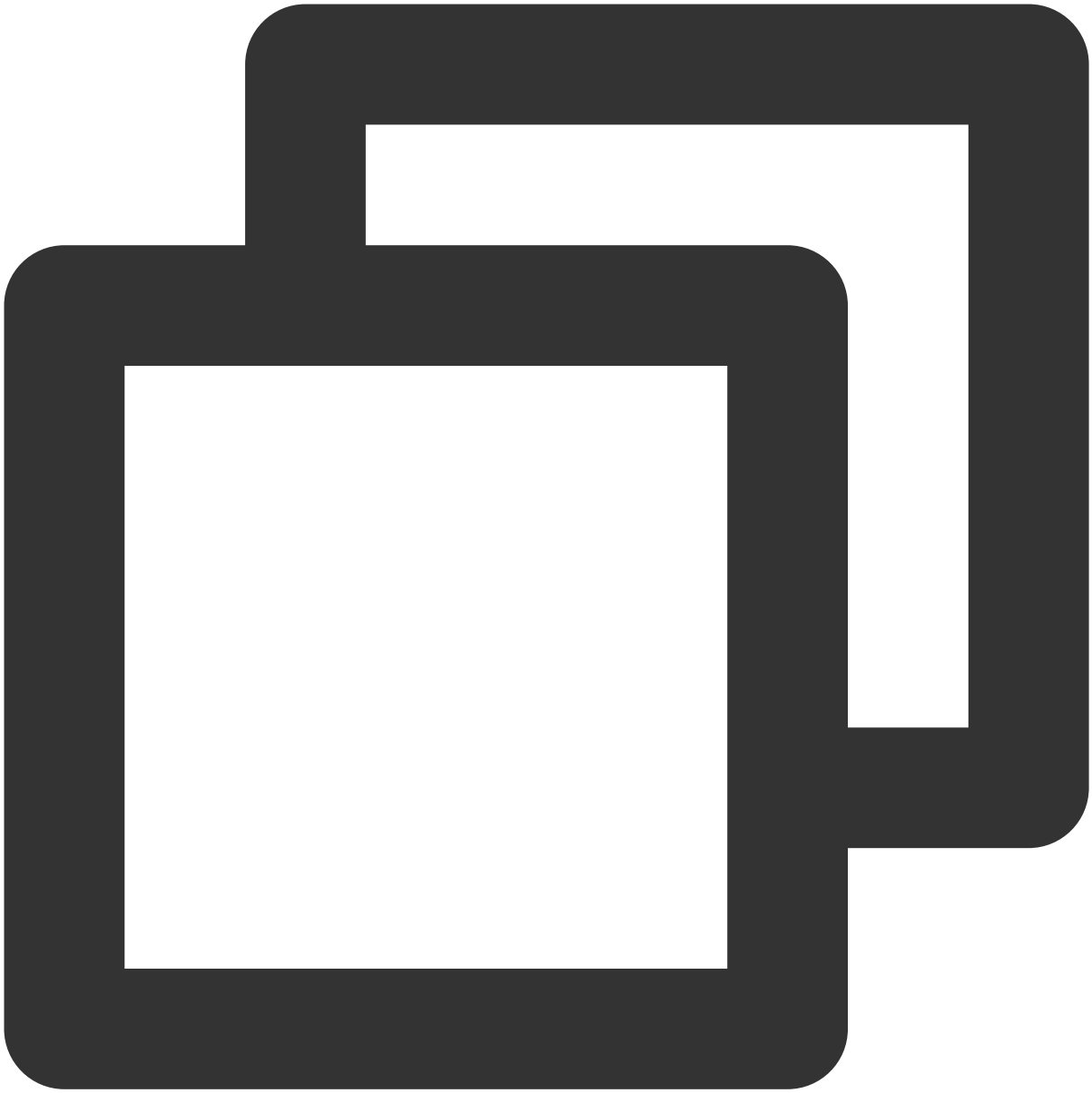
1. 为防止类冲突，如果已经使用 Flink 官方插件，需要先执行以下命令删除官方插件。



```
cd /usr/local/service/flink/lib
rm flink-metrics-prometheus*jar
```

2. . 在**弹性 MapReduce 控制台** > **选择对应的“实例”** > **集群资源** > **资源管理** > **Master**页面，查看 Master 节点。

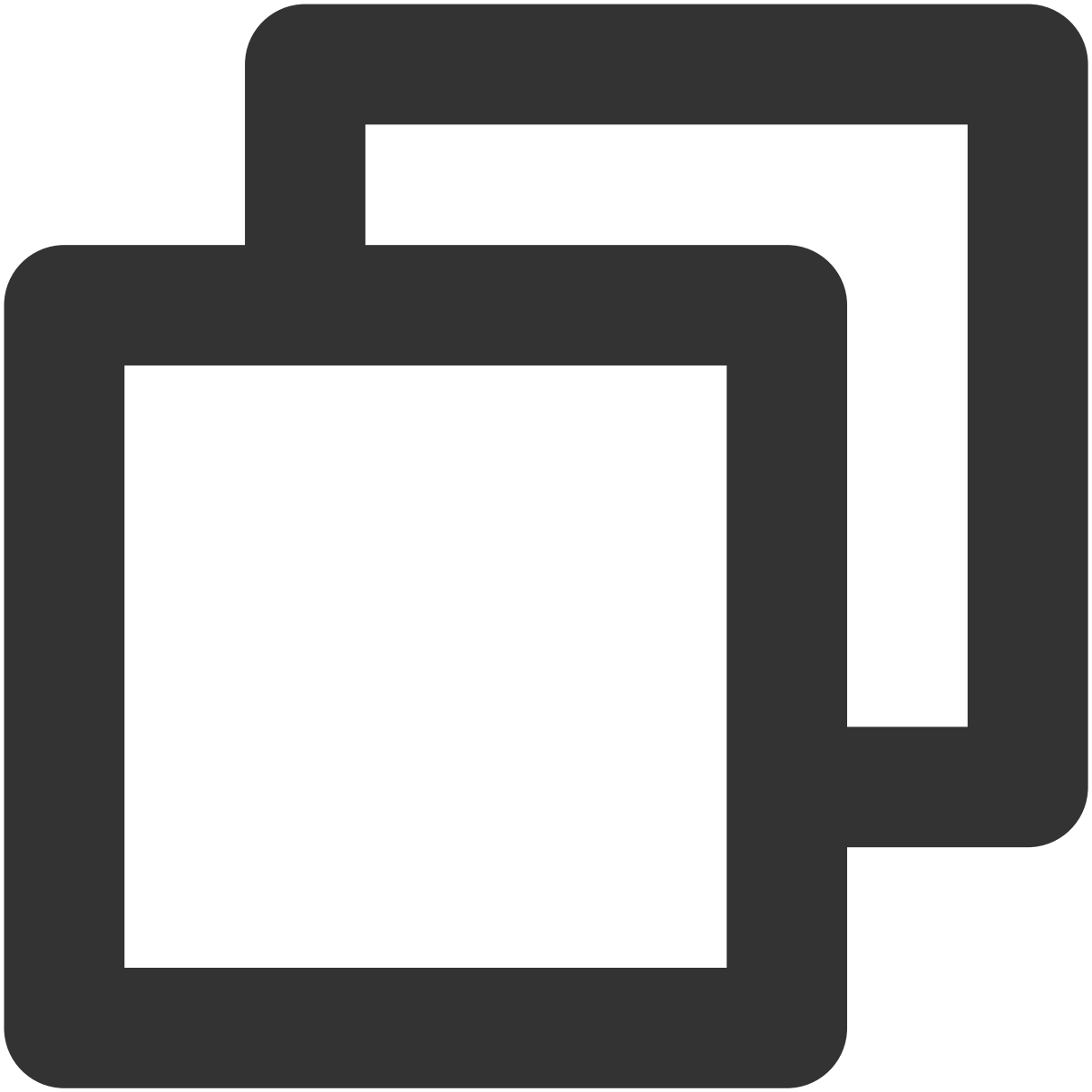
3. 单击实例 ID 跳转至 CVM 控制台，登录 CVM 执行以下命令安装插件。



```
cd /usr/local/service/flink/lib
wget https://rig-1258344699.cos.ap-guangzhou.myqcloud.com/flink/flink-metrics-prome
```

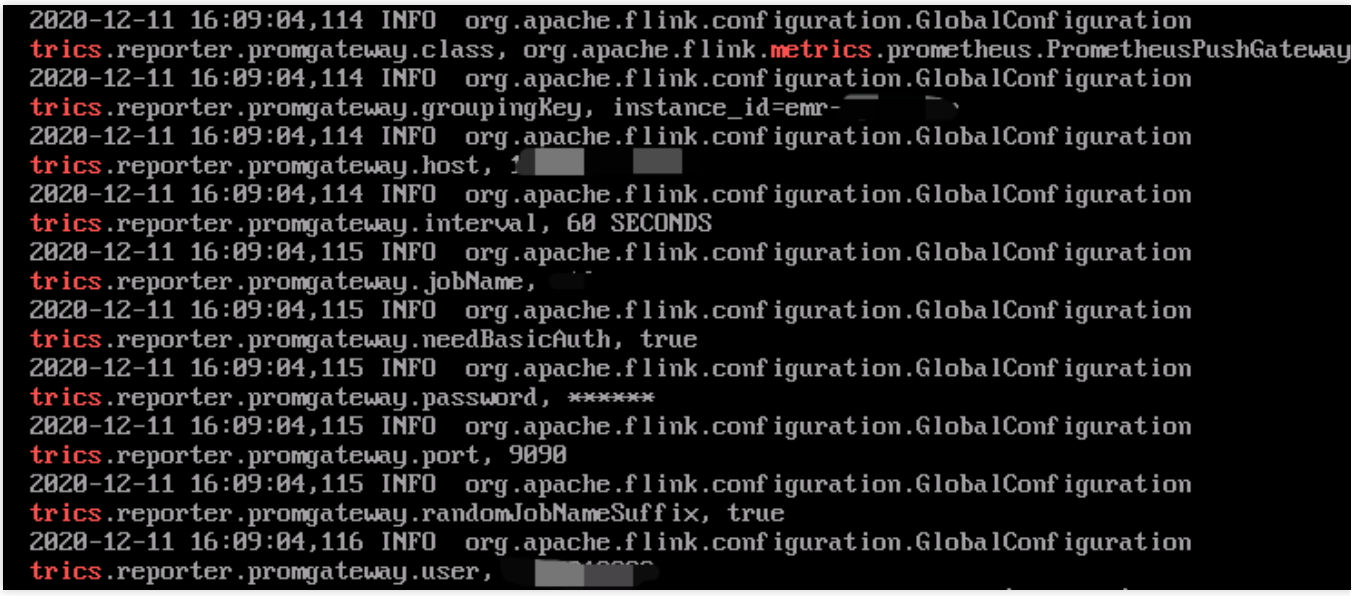
验证

1. 在 Master 节点上执行 `flink run` 命令提交新任务，查看任务日志。



```
grep metrics /usr/local/service/flink/log/flink-hadoop-client-*.log
```

2. 日志中包含下图内容，表示配置加载成功：



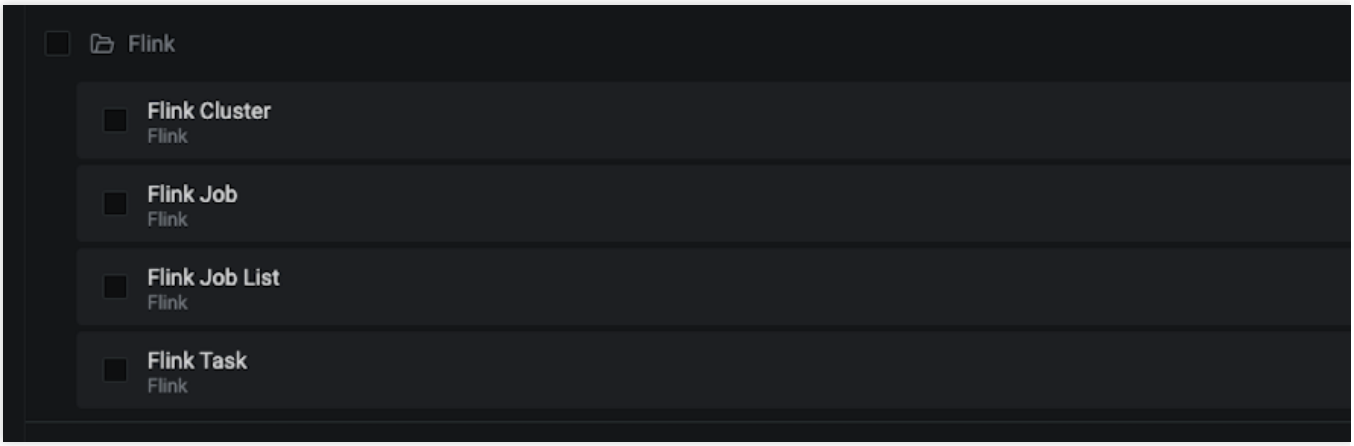
注意：

集群中已经提交的任务，由于使用的是旧配置文件，因此不会上报 metrics。

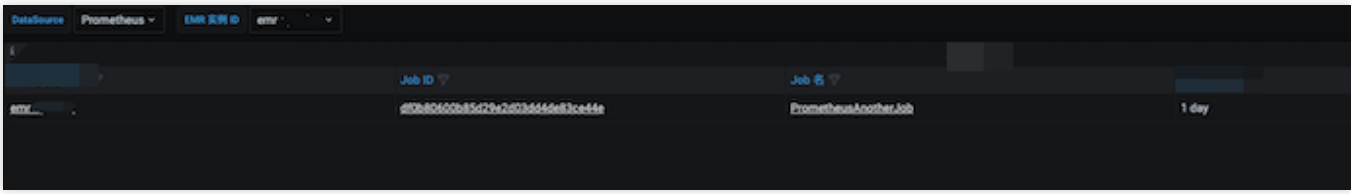
查看监控

1. 在对应 Prometheus 实例 >**集成中心**中找到 `Flink` 监控，安装对应的 Grafana Dashboard 即可开启 Flink 监控大盘。
2. 进入 Grafana，单击

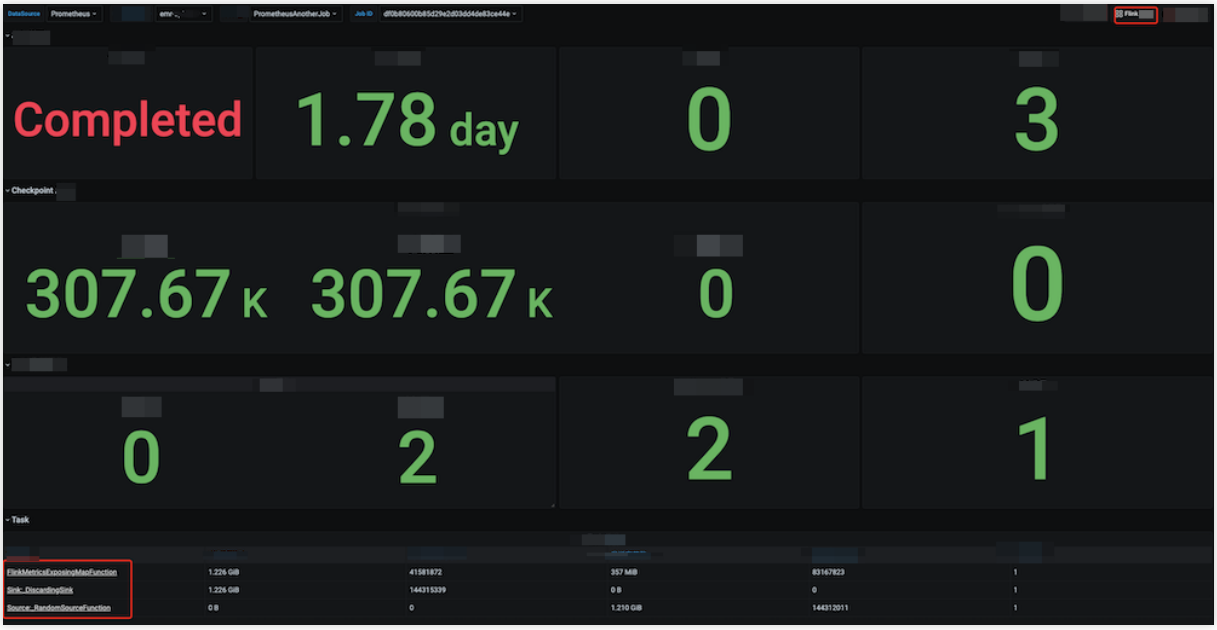
展开 Flink 监控面板。



3. 单击 **Flink Job List**查看监控。

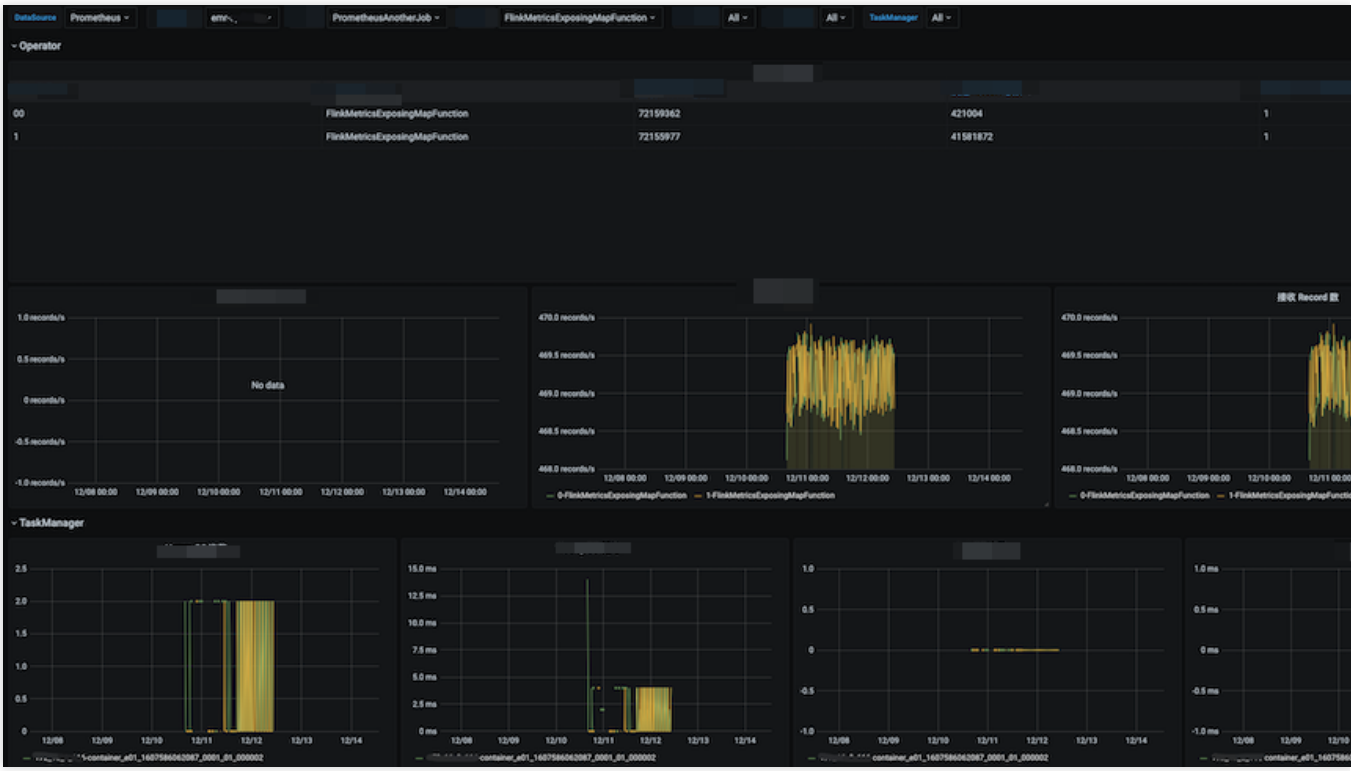


4. 单击表格中的 **Job 名**或 **Job ID 列值**，查看 Job 监控详情。



5. 单击右上角的 **Flink 集群**，查看 Flink 集群监控。

6. 单击表格中的 **Task 名列值**，查看 Task 监控详情。



告警接入

1. 登录 Prometheus 监控控制台，选择对应 Prometheus 实例进入管理页面。

2. 单击告警策略，可以添加相应的告警策略，详情请参见 新建告警策略。

Java 应用接入
Spring Boot 接入

最近更新时间：2024-01-29 17:01:27

操作场景

在使用 Spring Boot 作为开发框架时，需要监控应用的状态，例如 JVM/Spring MVC 等。 Prometheus 监控服务基于 Spring Actuator 机制采集 JVM 等数据，结合配套提供的 Grafana Dashboard 可以方便的监控 Spring Boot 应用的状态。
本文档以在容器服务上部署 Spring Boot 应用为例，介绍如何通过 Prometheus监控服务监控其状态。

前提条件

创建 腾讯云容器服务—托管版集群：在腾讯云容器服务中创建 Kubernetes 集群。
使用私有镜像仓库管理应用镜像。
应用基于 Spring Boot 框架进行开发。

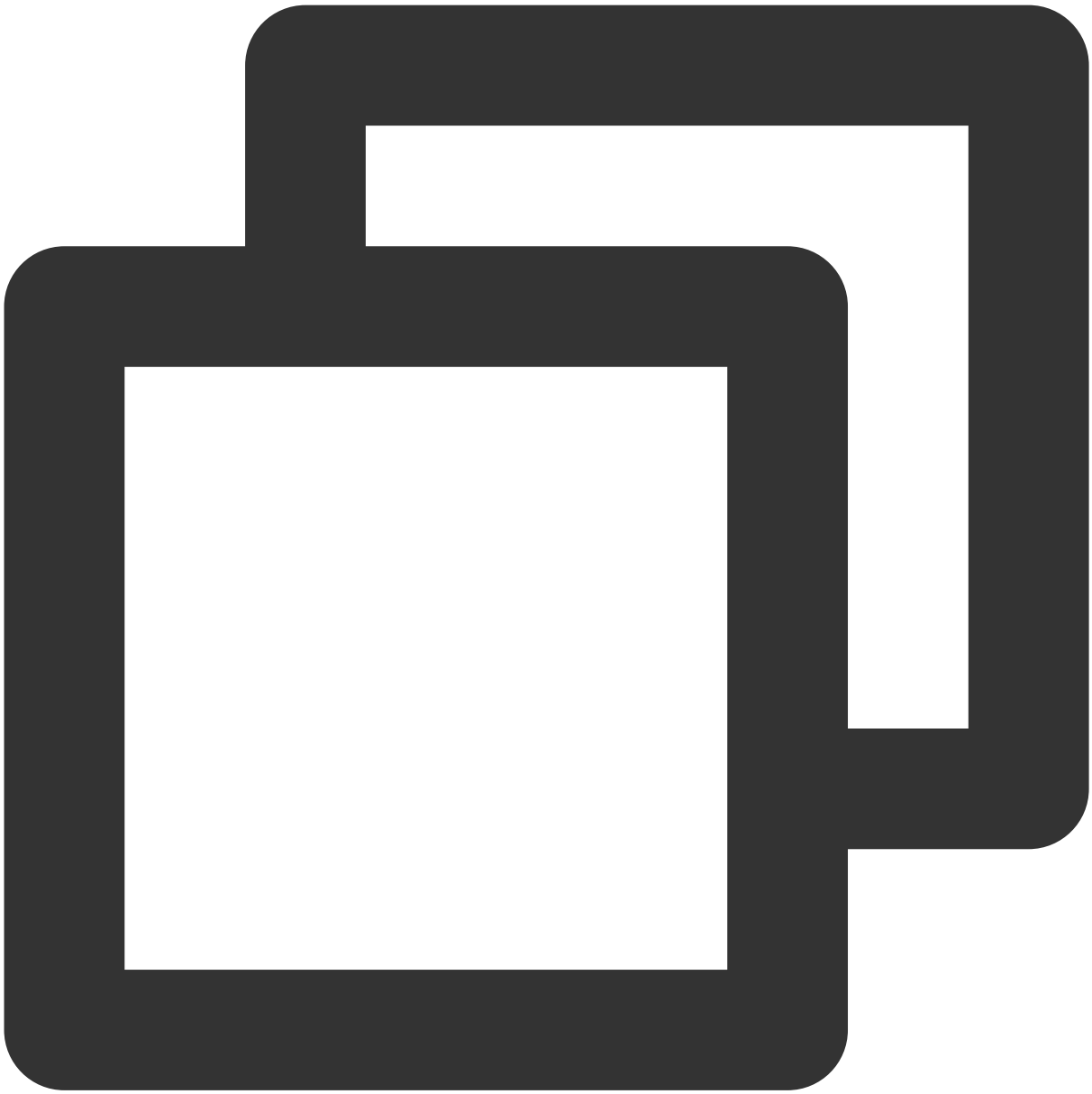
操作步骤

注意：
Spring Boot 已提供 actuator 组件来对应用进行监控，简化了开发的使用成本，所以这里直接使用 actuator 为 Spring Boot 应用进行监控埋点，基于 Spring Boot 2.0 及以上的版本，低版本会有配置上的差别需要注意。
若您使用spring boot 1.5 接入，接入时和2.0会有一定区别，需要注意如下几点：
1. 访问 `prometheus metrics` 的地址和2.0不一样，1.5默认的是 `/prometheus` ，
即 `http://localhost:8080/prometheus` 。
2. 若报401错误则表示没有权限(Whitelabel Error Page)，1.5默认对 `management` 接口加了安全控制，需要修改 `management.security.enabled=false` 。
3. 若项目中用 `bootstrap.yml` 来配置参数，在 `bootstrap.yml` 中需改 `management` 不启作用，需要在 `application.yml` 中修改，原因： spring boot 启动加载顺序有关。
4. `metric common tag` 不能通过 `yml` 来添加，只有通过代码加一个 `bean` 的方式添加。

修改应用的依赖及配置

步骤1：修改 pom 依赖

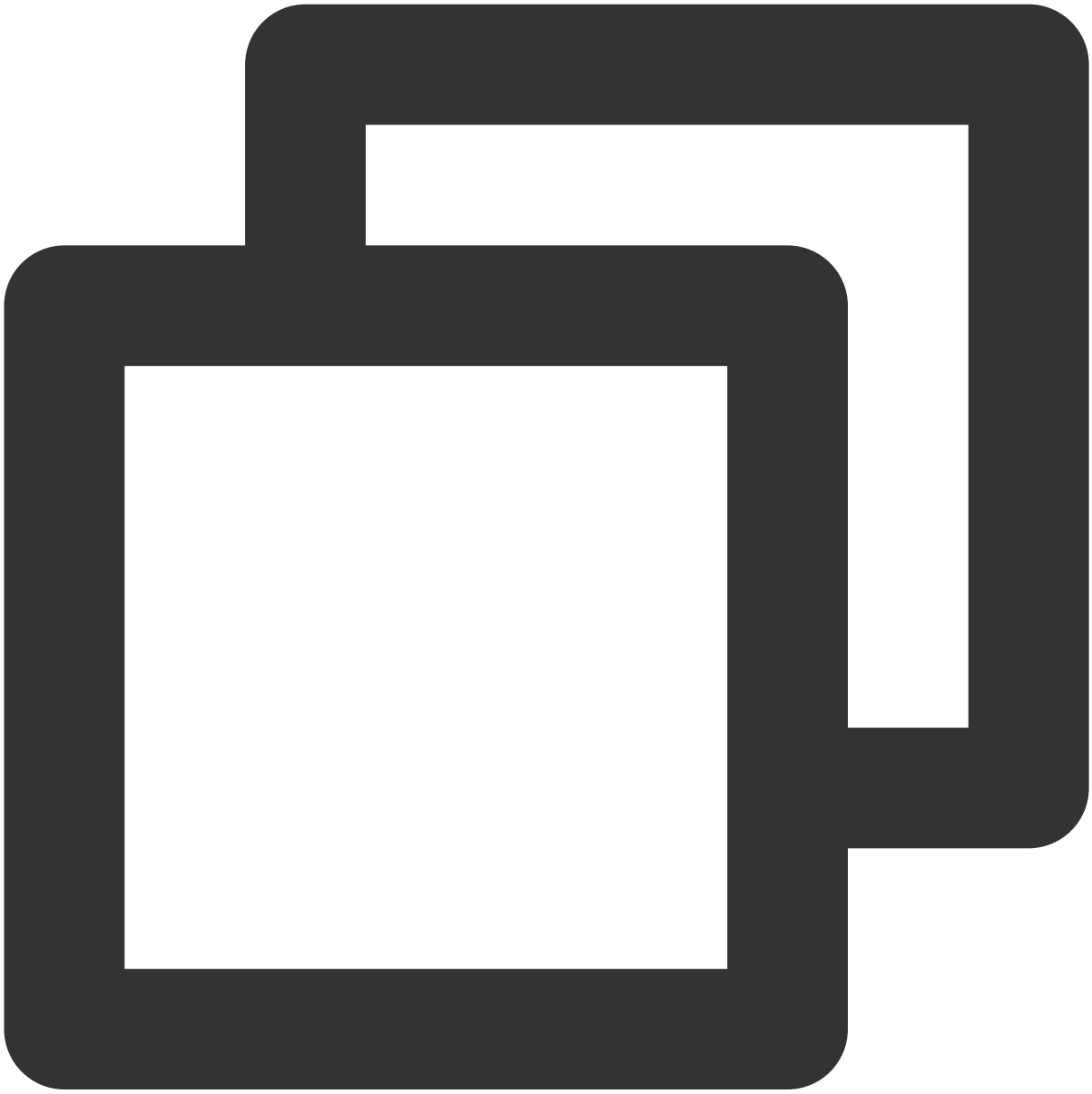
项目中已经引用 `spring-boot-starter-web` 的基础上，在 `pom.xml` 文件中添加 `actuator/prometheus` Maven 依赖项。



```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

步骤2：修改配置

编辑 `resources` 目录下的 `application.yml` 文件，修改 `actuator` 相关的配置来暴露 Prometheus 协议的指标数据。



```
management:
  endpoints:
    web:
      exposure:
        include: prometheus  # 打开 Prometheus 的 Web 访问 Path
```

```
metrics:
  # 下面选项建议打开，以监控 http 请求的 P99/P95 等，具体的时间分布可以根据实际情况设置
  distribution:
    sla:
      http:
        server:
          requests: 1ms,5ms,10ms,50ms,100ms,200ms,500ms,1s,5s
  # 在 Prometheus 中添加特别的 Labels
  tags:
    # 必须加上对应的应用名，因为需要以应用的维度来查看对应的监控
    application: spring-boot-mvc-demo
```

步骤3：本地验证

在项目当前目录下，运行 `mvn spring-boot:run` 之后，可以通过 `http://localhost:8080/actuator/prometheus` 访问到 Prometheus 协议的指标数据，说明相关的依赖配置已经正确。

说明：

例子中配置默认配置，对应的端口和路径以实际项目为准。

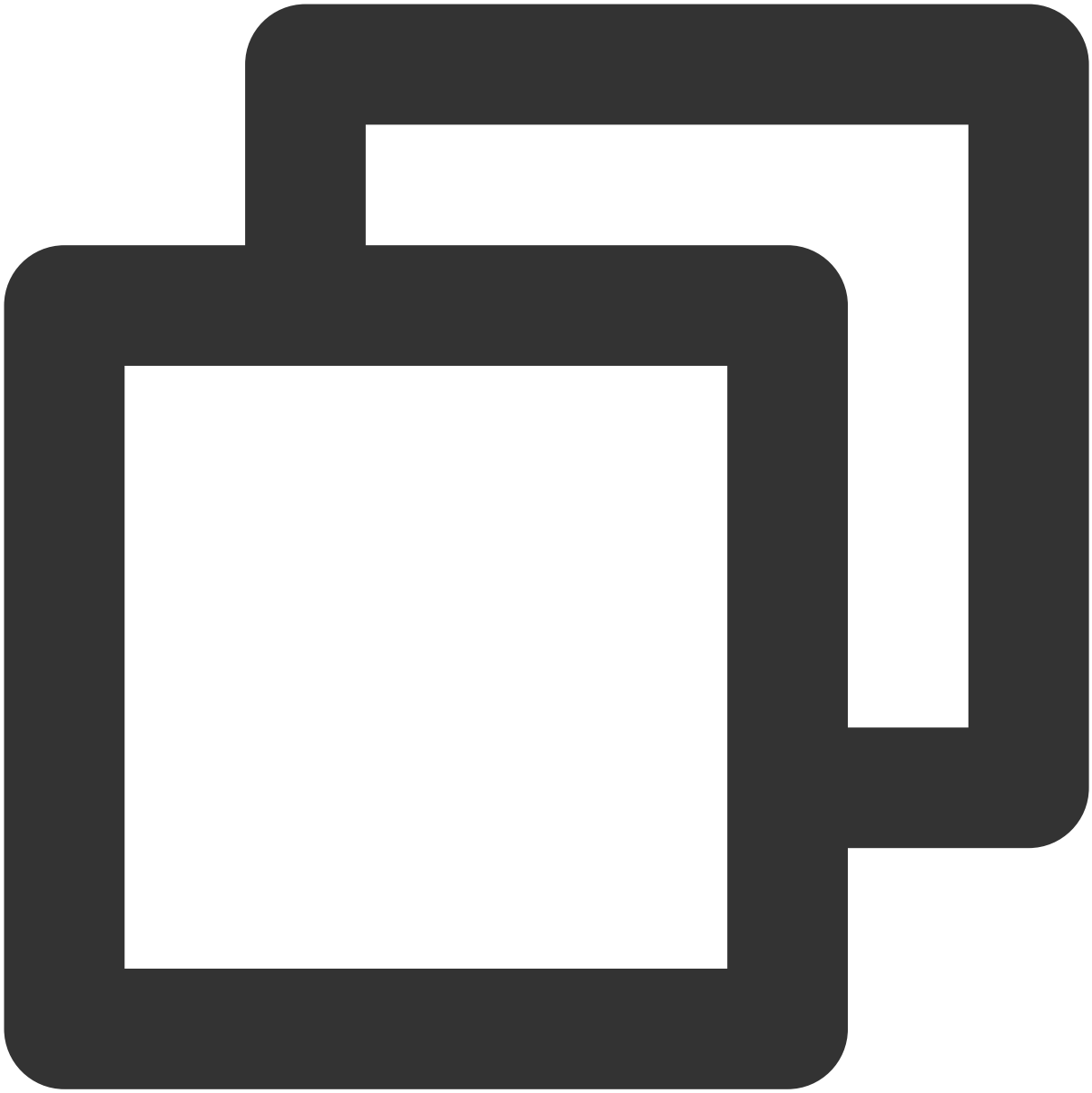
将应用发布到腾讯云容器服务上

步骤1：本地配置 Docker 镜像环境

如果本地之前未配置过 Docker 镜像环境，可以参考 镜像仓库基本教程 进行配置，如果已经配置可以直接执行下一步。

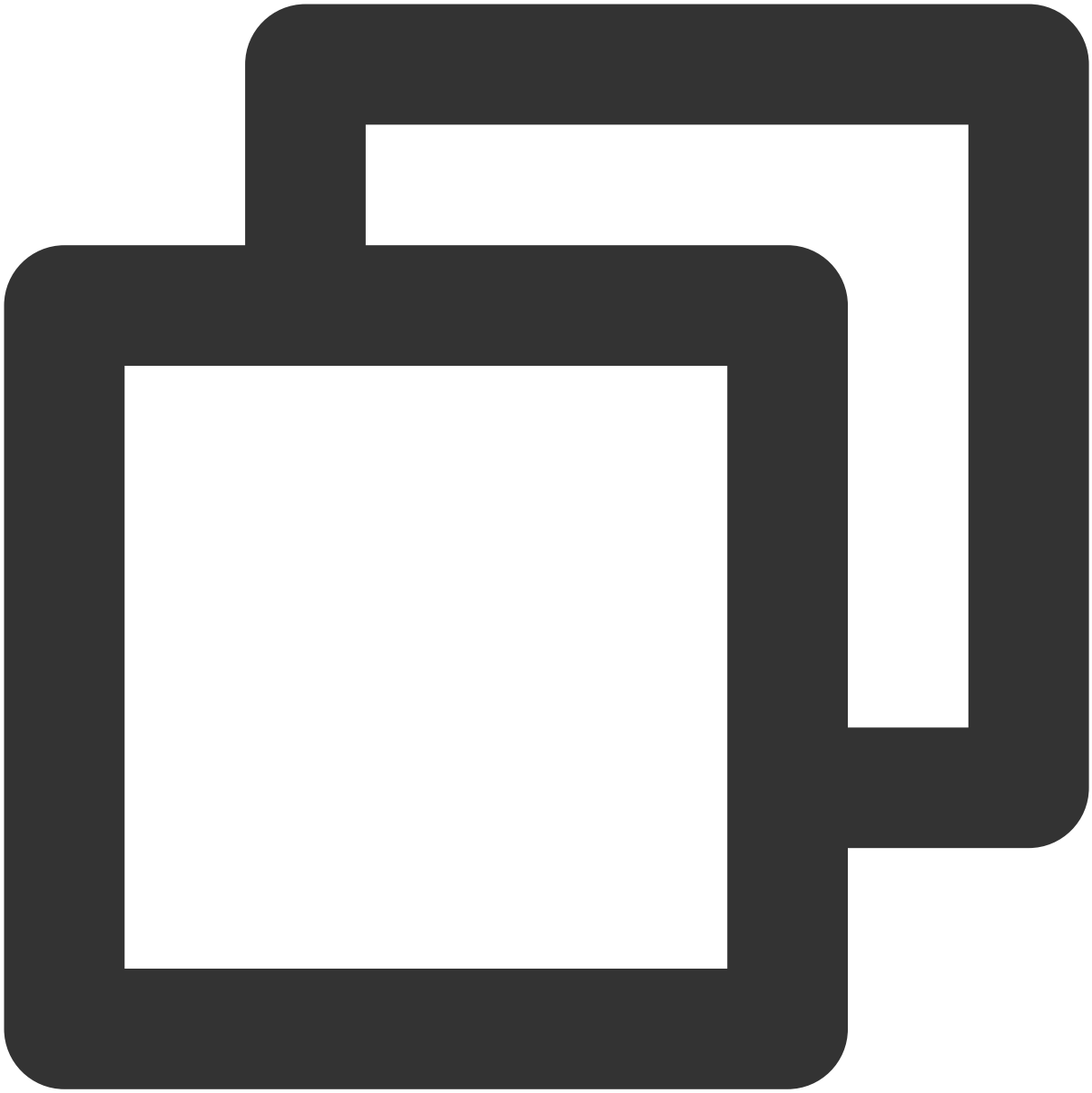
步骤2：打包及上传镜像

1. 在项目根目录下添加 `Dockerfile` ，您可以参考如下示例进行添加，在实际项目中需要修改 `Dockerfile` 。



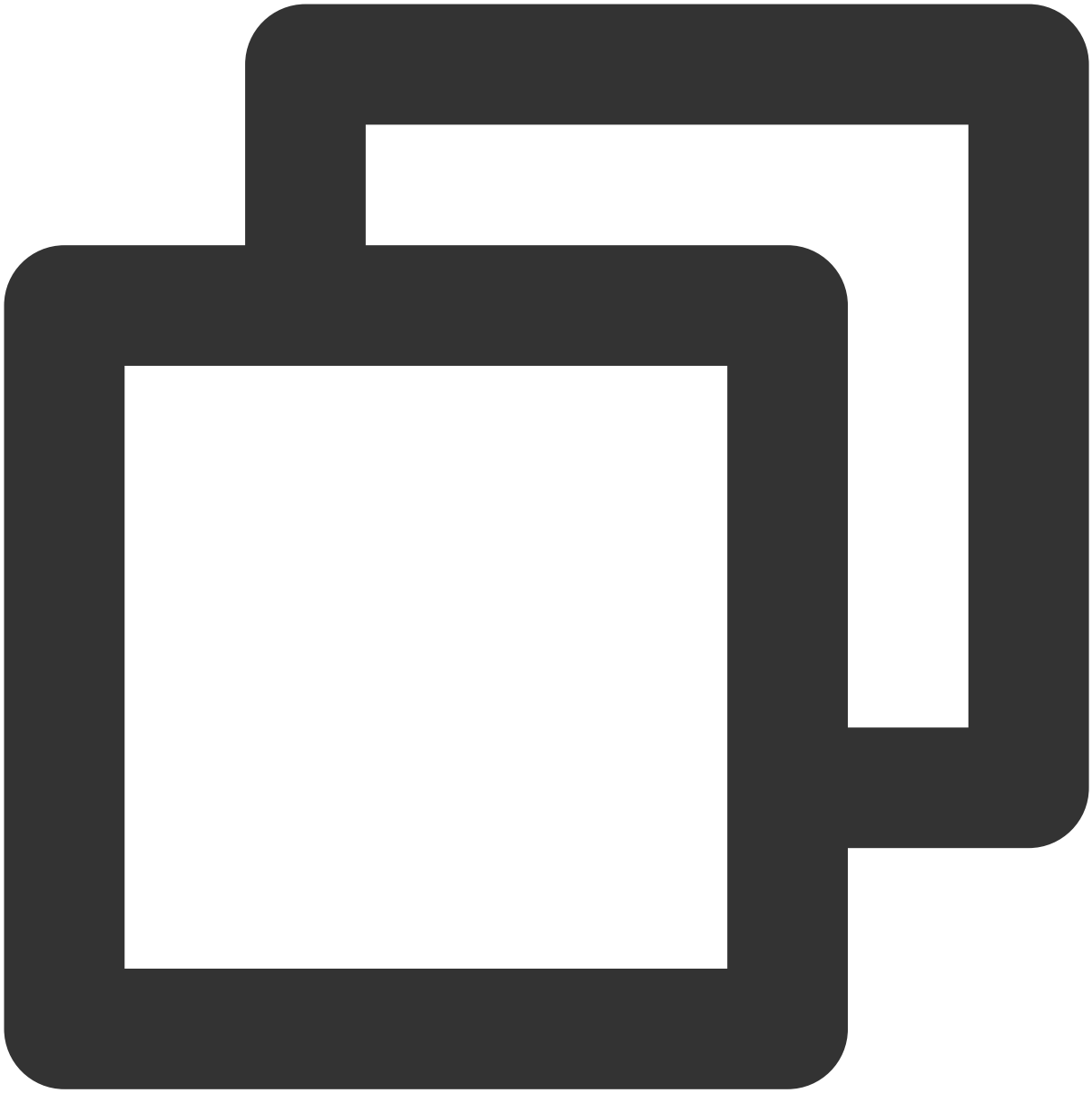
```
FROM openjdk:8-jdk
WORKDIR /spring-boot-demo
ADD target/spring-boot-demo-*.jar /spring-boot-demo/spring-boot-demo.jar
CMD ["java","-jar","spring-boot-demo.jar"]
```

2. 打包镜像，在项目根目录下运行如下命令，在实际项目中需要替换对应的 `namespace` 、 `ImageName` 、 `镜像版本号` 。



```
mvn clean package
docker build . -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

例如：

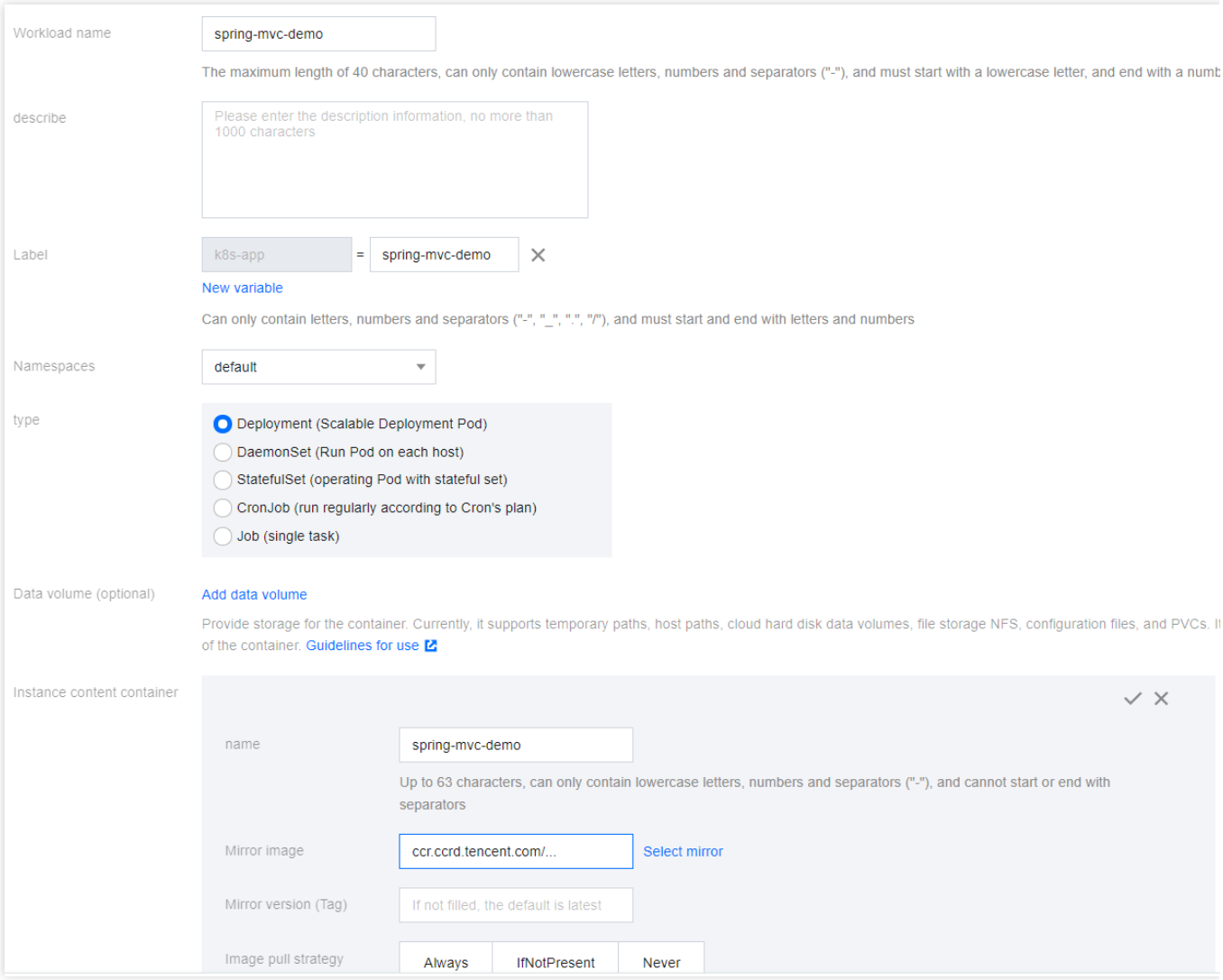


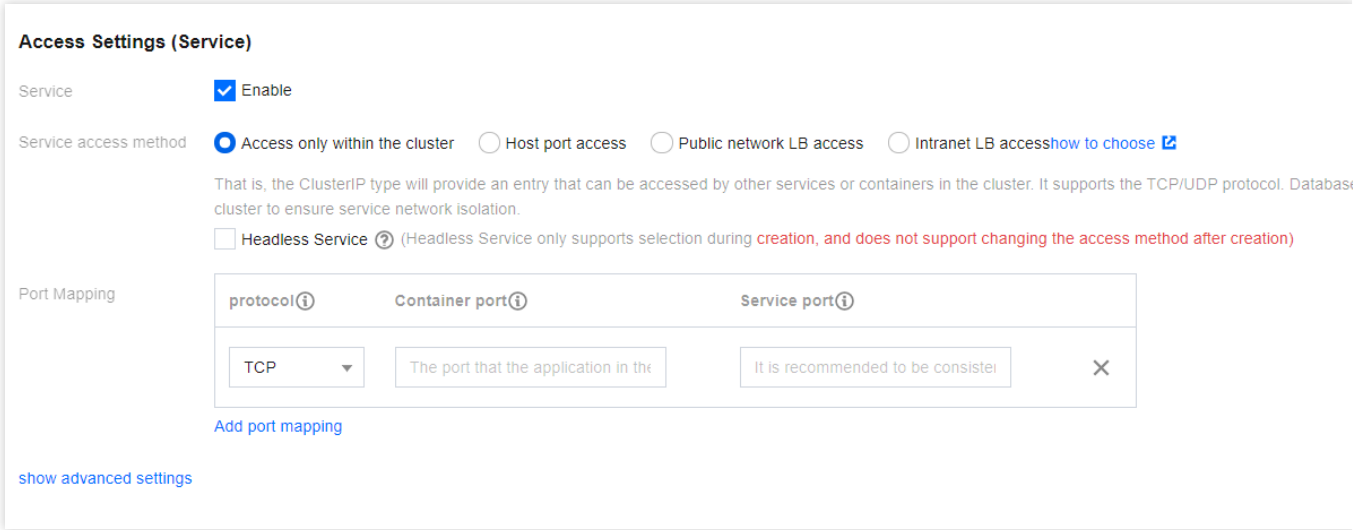
```
mvn clean package
docker build . -t ccr.ccs.tencentyun.com/prom_spring_demo/spring-boot-demo:latest
docker push ccr.ccs.tencentyun.com/prom_spring_demo/spring-boot-demo:latest
```

步骤3：应用部署

1. 登录 容器服务控制台，选择需要部署的容器集群。

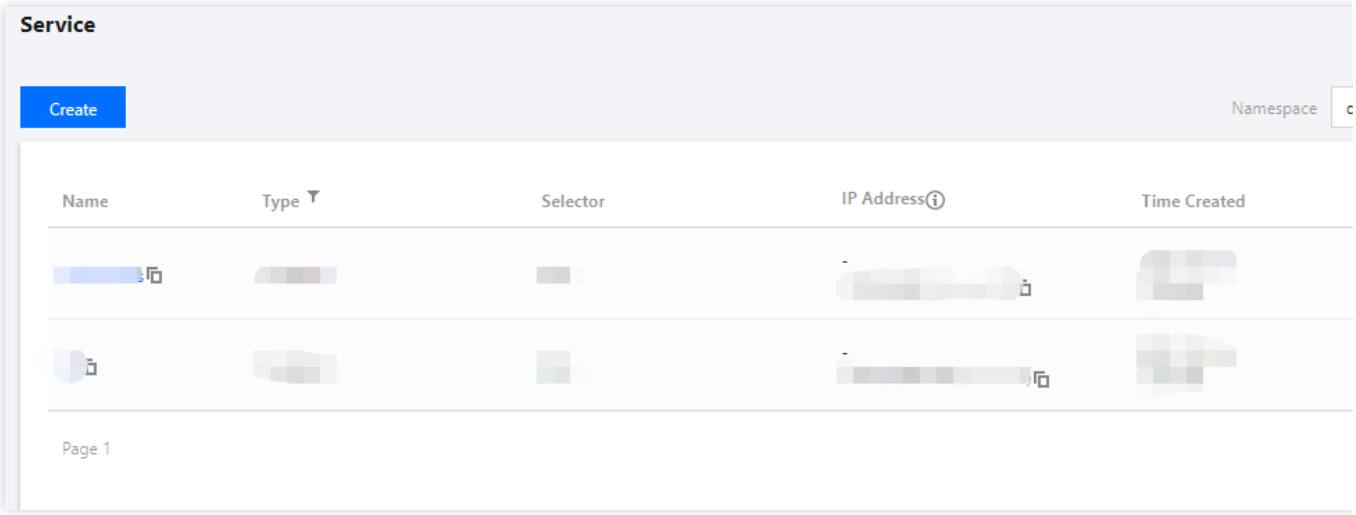
2. 单击**工作负载** > **Deployment**，进入 Deployment 管理页面，选择对应的命名空间来进行部署服务，这里选择通过控制台的方式创建，同时打开 Service 访问方式，您也可以选择通过命令行的方式创建。



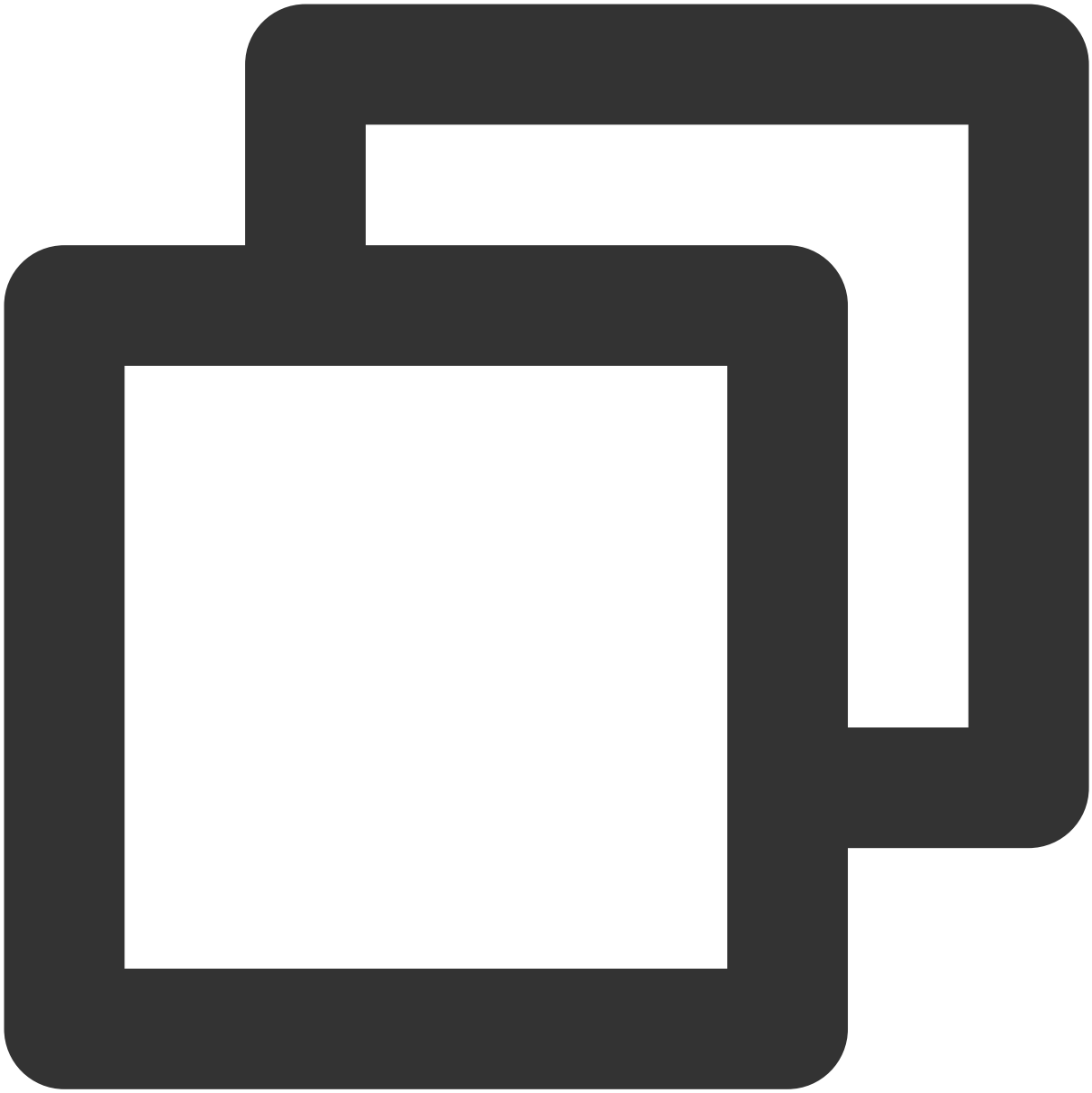


3. 为对应的 Service 添加 K8S Labels，如果使用命令方式新建，可以将 Labels 直接加上。这里介绍在容器控制台调整配置，选择需要调整的容器集群。

单击**服务与路由** > **Service**，进入 Service 管理页面，选择对应的命名空间来调整 Service Yaml 配置，如下图：



配置示例如下：



```
apiVersion: v1
kind: Service
metadata:
  labels: # 可以根据实际情况添加对应的 labels
  k8sapp: spring-mvc-demo
  name: spring-mvc-demo
  namespace: spring-demo
spec:
  ports:
  - name: 8080-8080-tcp  # ServiceMonitor 抓取任务中 port 对应的值
    port: 8080
```

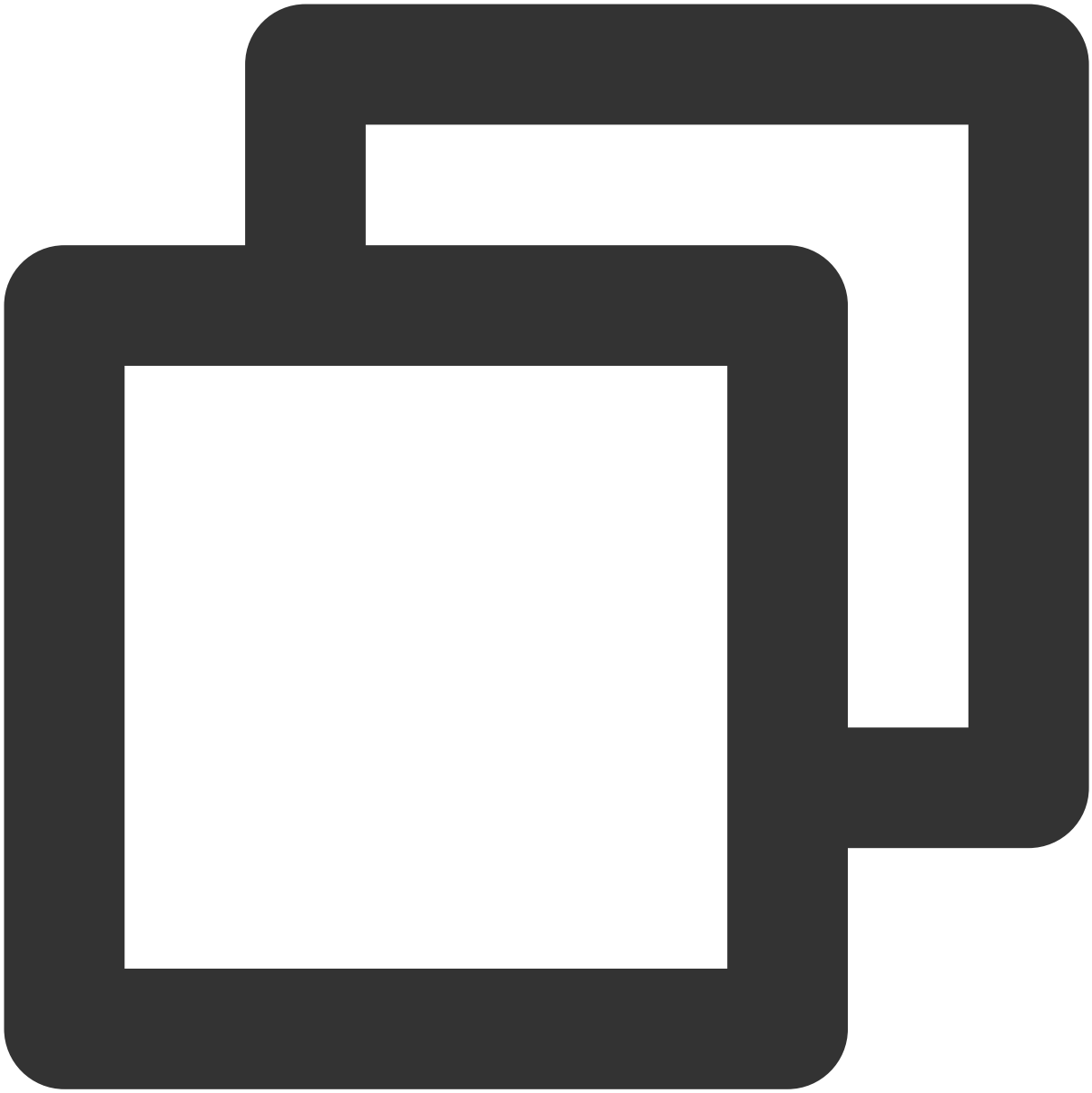
```
    protocol: TCP
    targetPort: 8080
  selector:
    k8s-app: spring-mvc-demo
    qcloud-app: spring-mvc-demo
  sessionAffinity: None
  type: ClusterIP
```

步骤4：添加采取任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击集成容器服务列表中的**集群 ID**，进入到容器服务集成管理页面。
3. 通过服务发现添加 Service Monitor，目前支持基于 Labels 发现对应的目标实例地址，所以可以对一些服务添加特定的 K8S Labels，配置之后在 Labels 下的服务都将被 Prometheus 服务自动识别出来，不需要再为每个服务一一添加采取任务。以该例子介绍，配置信息如下：

说明：

这里需要注意的是 `port` 的取值为 `service yaml` 配置文件里的 `spec/ports/name` 对应的值。



```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: spring-mvc-demo # 填写一个唯一名称
  namespace: cm-prometheus # namespace固定，不要修改
spec:
  endpoints:
  - interval: 30s
    port: 8080-8080-tcp # 填写service yaml中Prometheus Exporter对应的Port的Name
    path: /actuator/prometheus  # 填写Prometheus Exporter对应的Path的值，不填默认/met
  namespaceSelector:  # 选择要监控service所在的namespace
```

```
    matchNames:
    - spring-demo
  selector: # 填写要监控service的Label值，以定位目标service
    matchLabels:
      k8sapp: spring-mvc-demo
```

步骤5：查看监控

打开 Prometheus 实例对应的 Grafana 地址，在 `Dashboards/Manage/Application` 下查看应用相关的监控大屏。

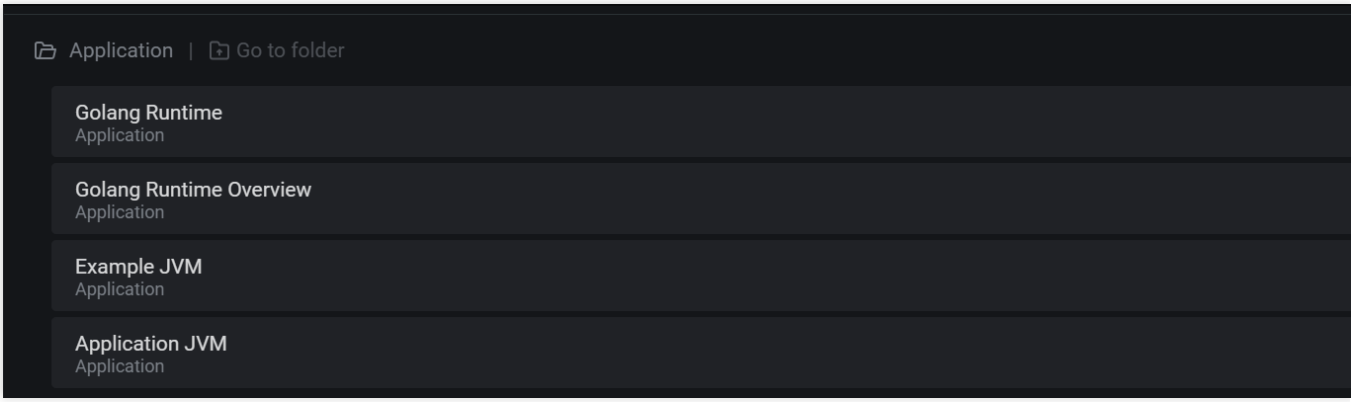
Spring MVC 应用：监控 MVC 的状态，例如请求耗时/请求量/成功率/异常分布等。

Spring MVC 接口：接口级监控，可以对应多个接口，方便定位是哪个接口出问题。

Tomcat：Tomcat 内部状态的监控大屏，例如线程使用情况等。

应用 JVM：从应用角度出发，查看该应用下所有实例是否有问题，当发现某个实例有问题时可以下钻到对应的实例监控。

实例 JVM：单实例 JVM 详细的监控数据。



JVM 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 Java 作为开发语言的时候，需要监控 JVM 的性能。 Prometheus 监控服务通过采集应用暴露出来的 JVM 监控数据，并提供了开箱即用的 Grafana 监控大盘。
本文以如何在容器服务上部署普通 Java 应用为例，介绍如何通过 Prometheus 监控服务监控其状态。
说明：
若已使用 Spring Boot 作为开发框架，请参见 Spring Boot 接入。

前提条件

创建腾讯云容器服务 托管版集群。
使用私有镜像仓库管理应用镜像。

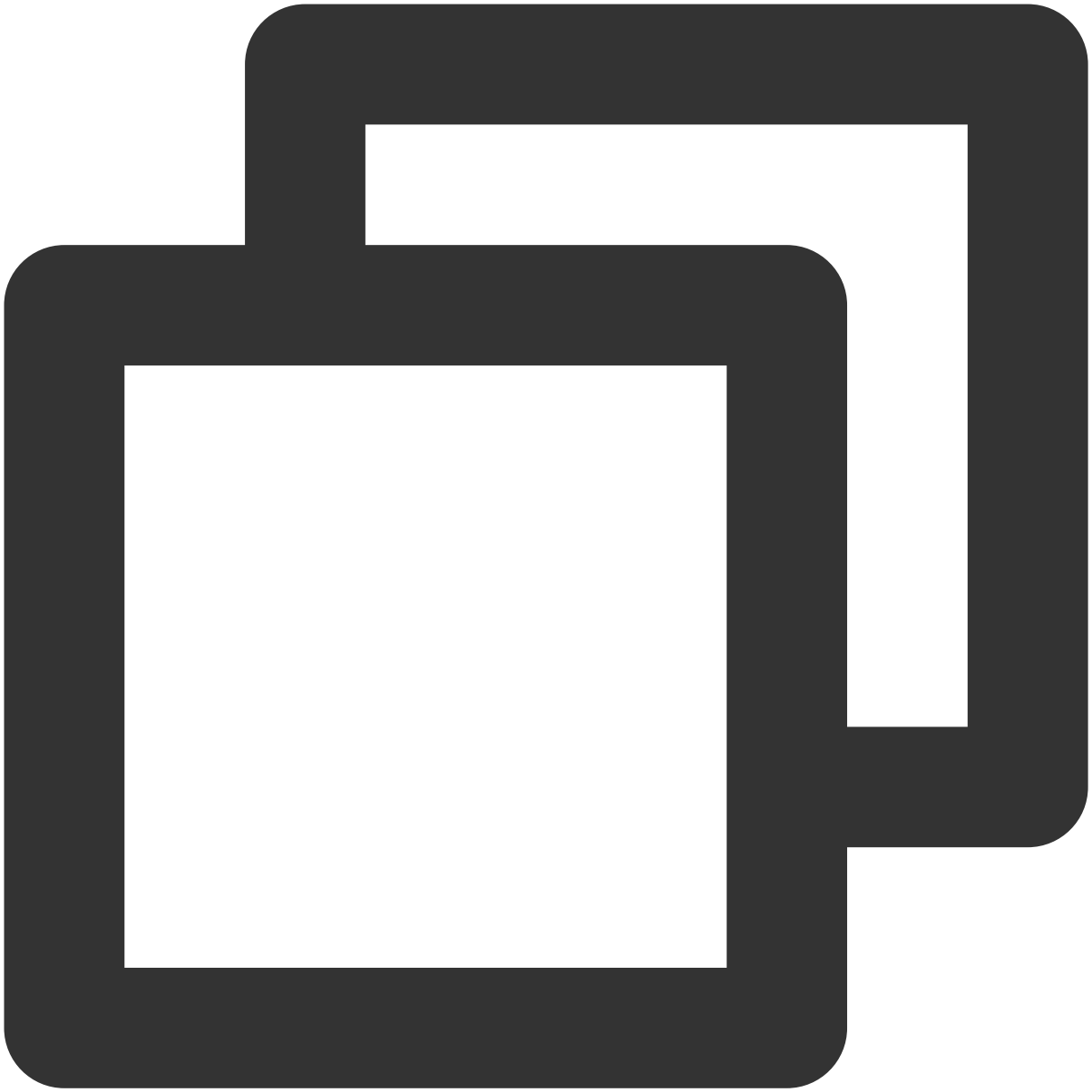
操作步骤

说明：
Java 作为主流的开发语言其生态较为完善，其中 micrometer 作为指标打点 SDK 已经被广泛运行，本文以 micrometer 为例介绍如何监控 JVM。

修改应用的依赖及配置

步骤1：修改 pom 依赖

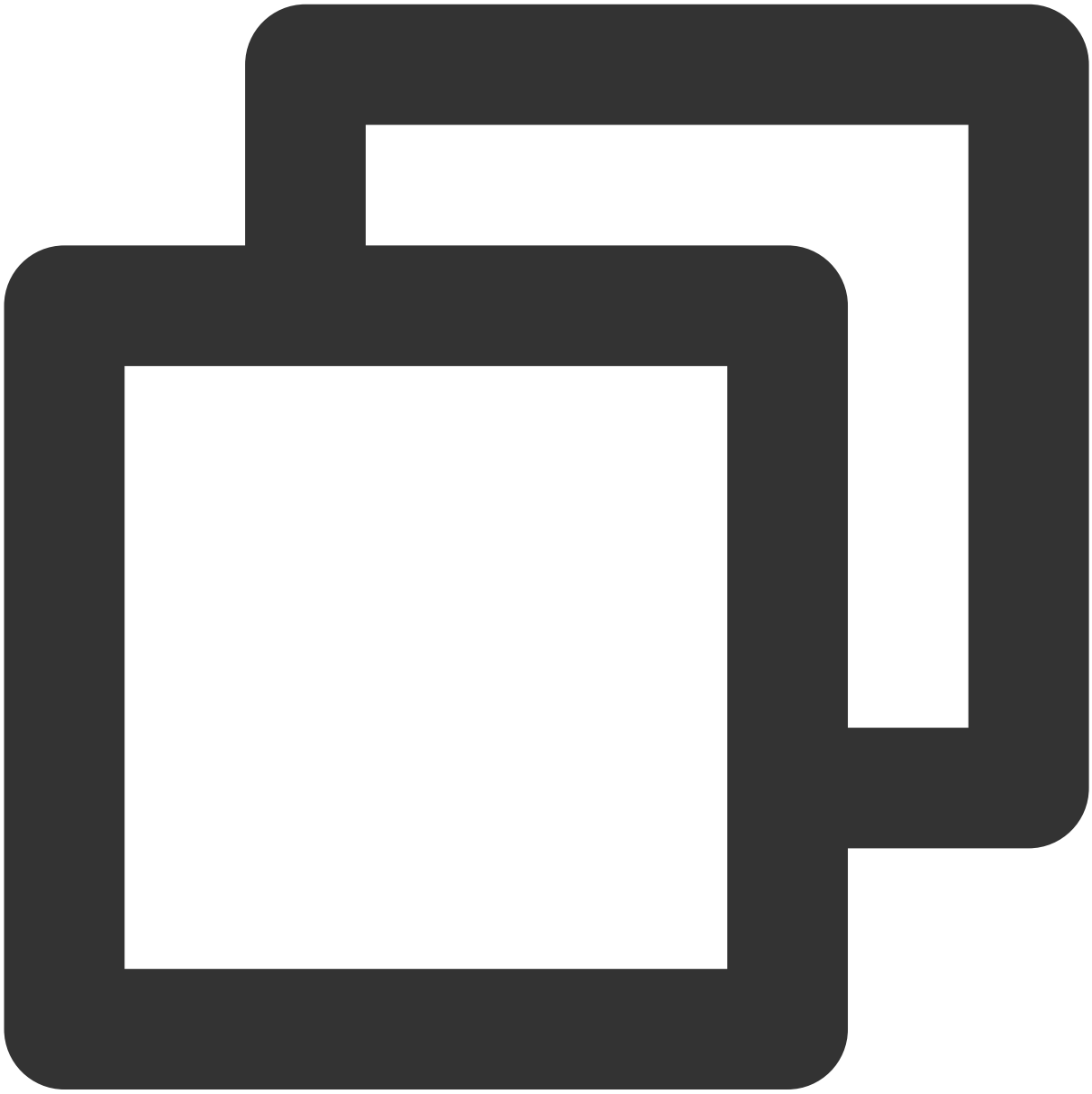
在 `pom.xml` 文件中添加相关的 Maven 依赖项，试情况调整相应的版本，示例如下：



```
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient</artifactId>
    <version>0.9.0</version>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <version>1.1.7</version>
</dependency>
```

步骤2：修改代码

在项目启动时，添加相应的监控配置，同时 micrometer 也提供了部分常用的监控数据采集，具体在 `io.micrometer.core.instrument.binder` 包下，可以按实际情况添加。示例如下：



```
public class Application {
    // 作为全局变量，可以在自定义监控中使用
    public static final PrometheusMeterRegistry registry = new PrometheusMeterRegis
    static {
        // 添加 Prometheus 全局 Label，建议加一上对应的应用名
```

```
        registry.config().commonTags("application", "java-demo");
    }

    public static void main(String[] args) throws Exception {
        // 添加 JVM 监控
        new ClassLoaderMetrics().bindTo(registry);
        new JvmMemoryMetrics().bindTo(registry);
        new JvmGcMetrics().bindTo(registry);
        new ProcessorMetrics().bindTo(registry);
        new JvmThreadMetrics().bindTo(registry);
        new UptimeMetrics().bindTo(registry);
        new FileDescriptorMetrics().bindTo(registry);
        System.gc(); // Test GC
        try {
            // 暴露 Prometheus HTTP 服务，如果已经有，可以使用已有的 HTTP Server
            HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
            server.createContext("/metrics", httpExchange -> {
                String response = registry.scrape();
                httpExchange.sendResponseHeaders(200, response.getBytes().length);
                try (OutputStream os = httpExchange.getResponseBody()) {
                    os.write(response.getBytes());
                }
            });

            new Thread(server::start).start();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

说明：

由于 JVM GC Pause 监控是通过 GarbageCollector Notification 机制实现，因此只有发生 GC 之后才有监控数据。上述示例为了测试更直观，主动调用了 `System.gc()` 。

步骤3：本地验证

本地启动之后，可以通过 `http://localhost:8080/metrics` 访问到 Prometheus 协议的指标数据。

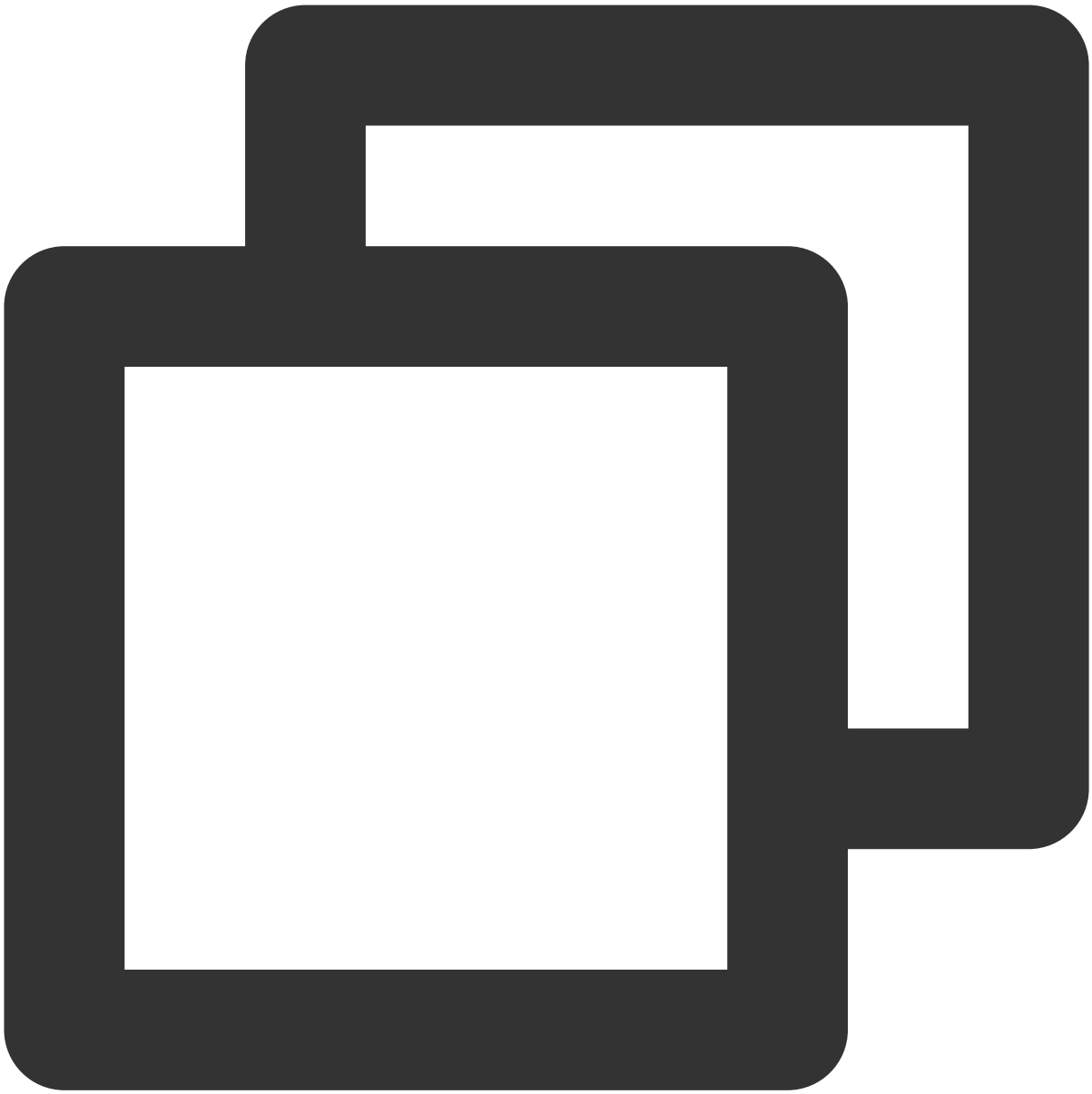
将应用发布到腾讯云容器服务上

步骤1：本地配置 Docker 镜像环境

如果本地之前未配置过 Docker 镜像环境，可以参见容器镜像服务 快速入门 文档进行配置。若已配置请执行下一步。

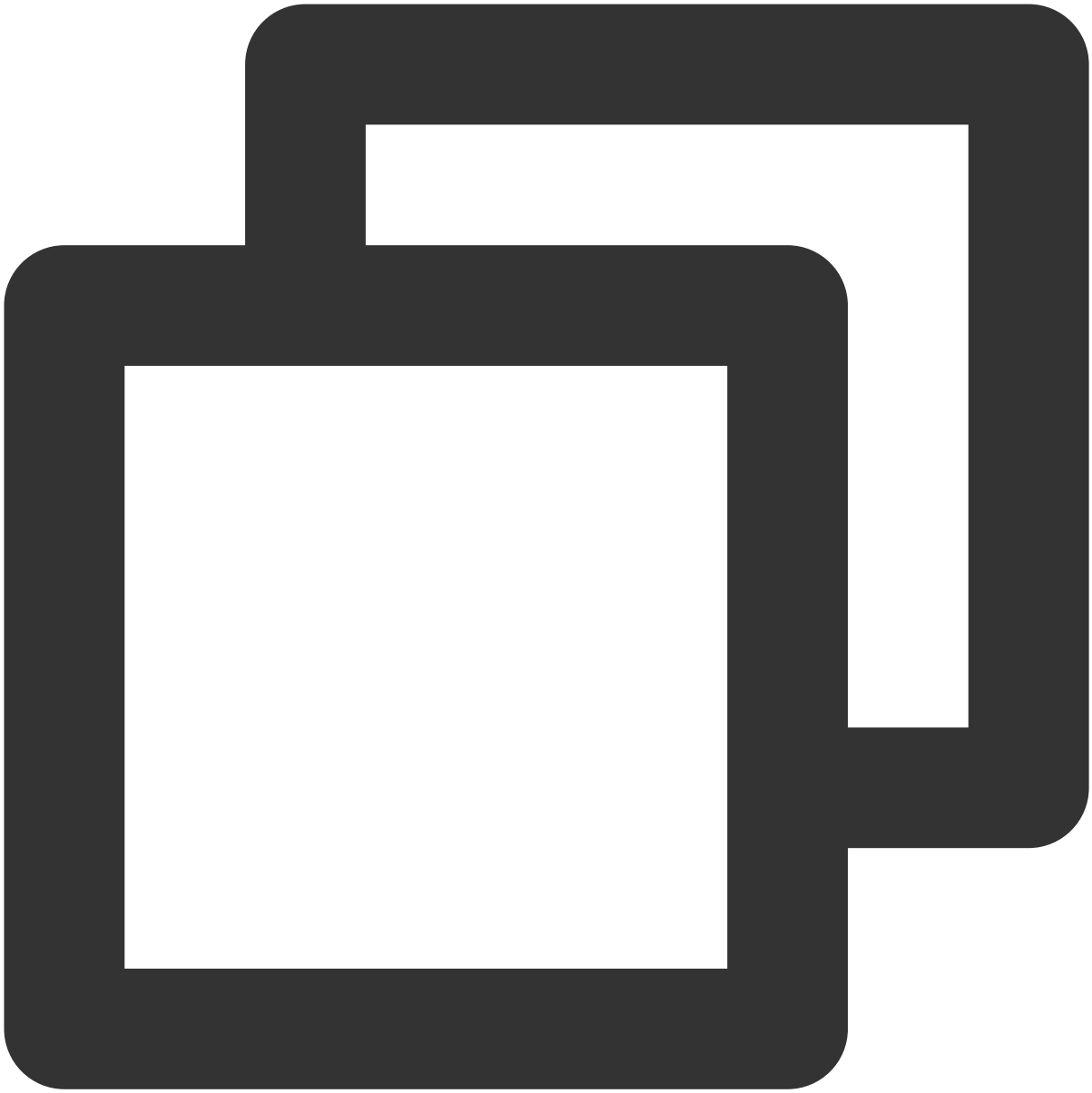
步骤2：打包及上传镜像

1. 在项目根目录下添加 `Dockerfile`，请根据实际项目进行修改。示例如下：



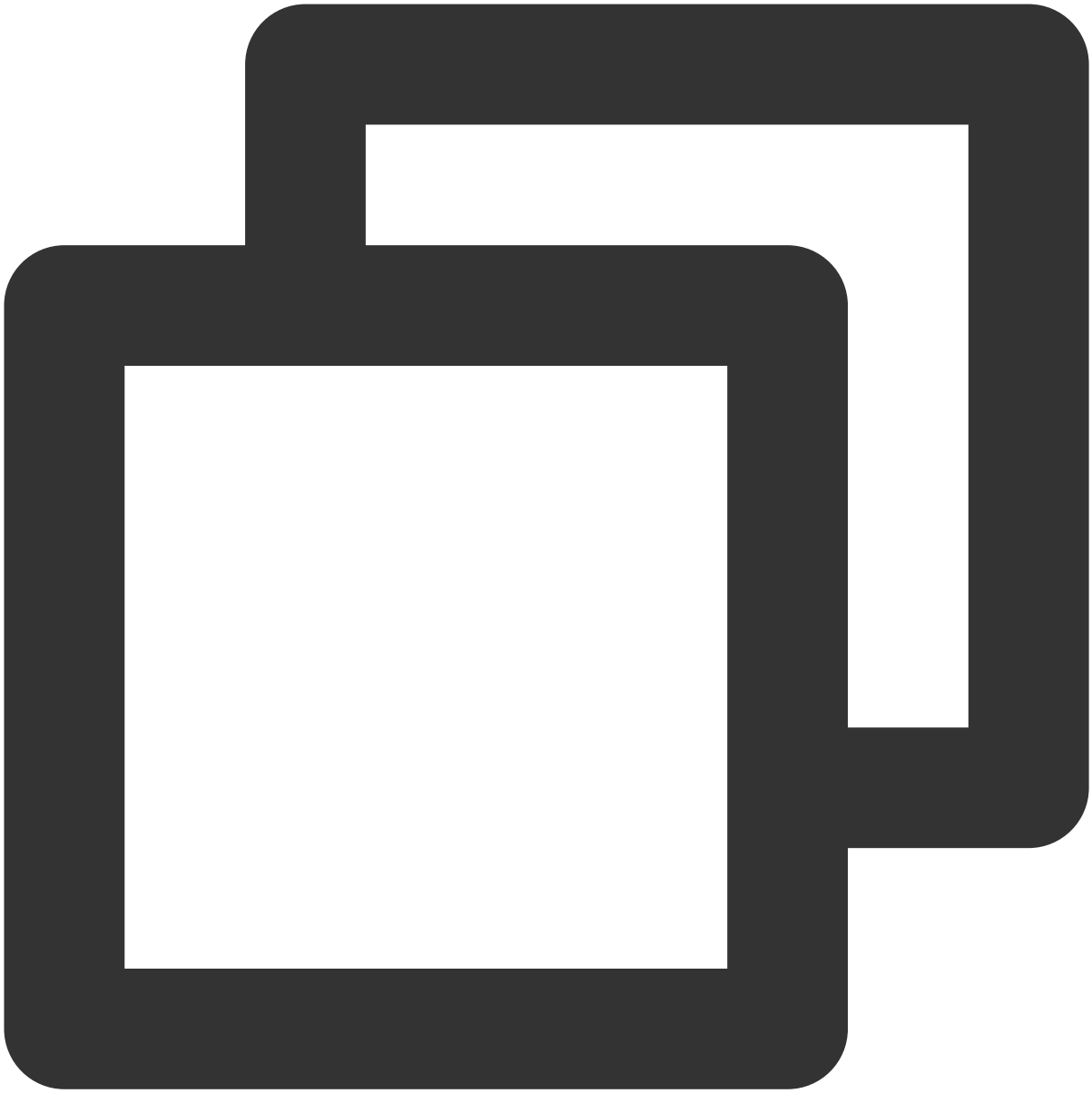
```
FROM openjdk:8-jdk
WORKDIR /java-demo
ADD target/java-demo-*.jar /java-demo/java-demo.jar
CMD ["java","-jar","java-demo.jar"]
```

2. 打包镜像，在项目根目录下运行如下命令，需要替换对应的 `namespace` / `ImageName` / `镜像版本号` 。



```
mvn clean package
 docker build . -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
 docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

示例如下：



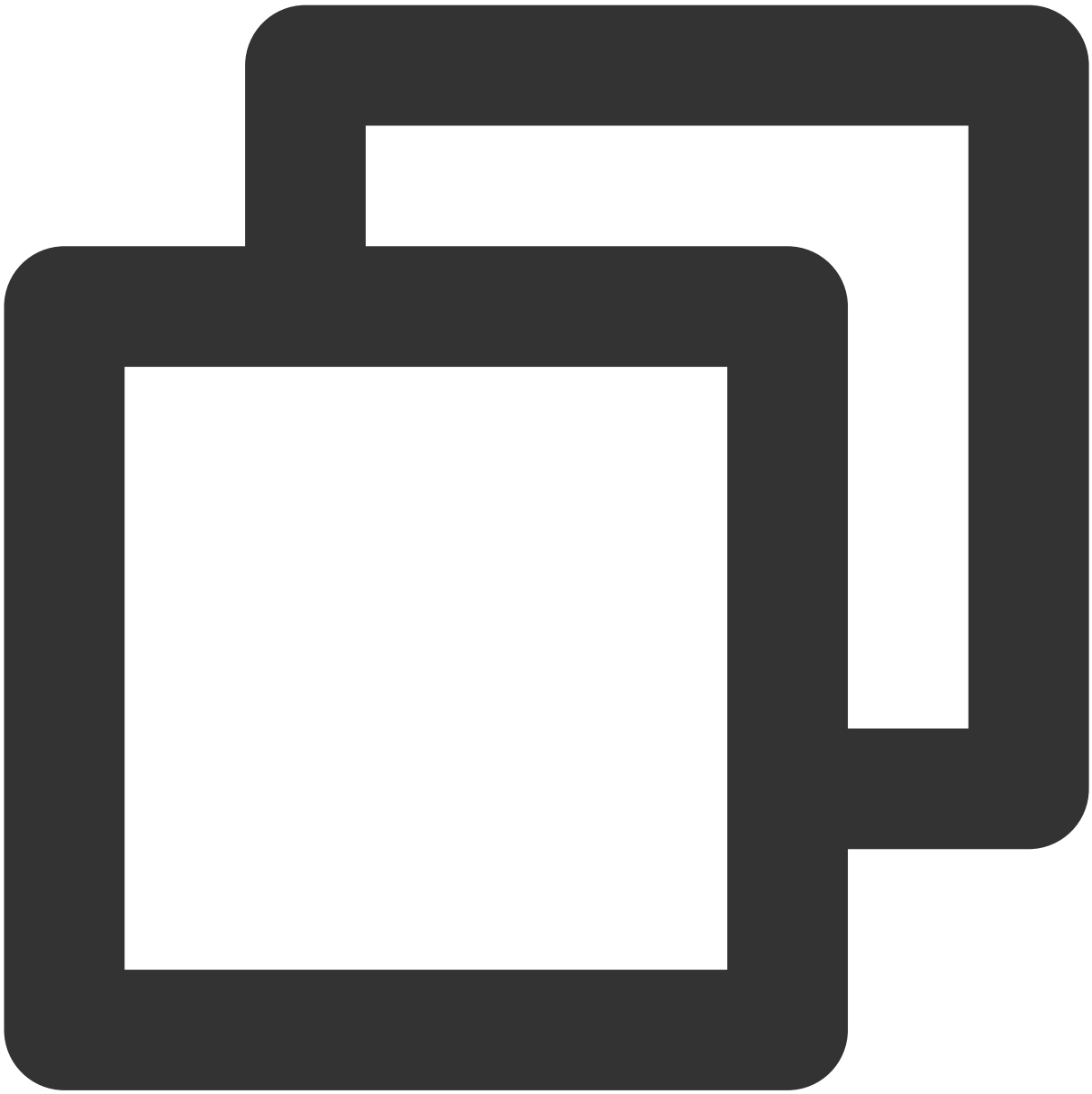
```
mvn clean package
 docker build . -t ccr.ccs.tencentyun.com/prom_spring_demo/java-demo:latest
 docker push ccr.ccs.tencentyun.com/prom_spring_demo/-demo:latest
```

步骤3：应用部署

1. 登录 容器服务控制台，选择需要部署的容器集群。
2. 通过**工作负载** > **Deployment**进入 `Deployment` 管理页面，选择对应的 `命名空间` 来进行部署服务，通过 YAML 来创建对应的 `Deployment` ，YAML 配置如下。

说明：

如需通过控制台创建，请参见Spring Boot 接入。

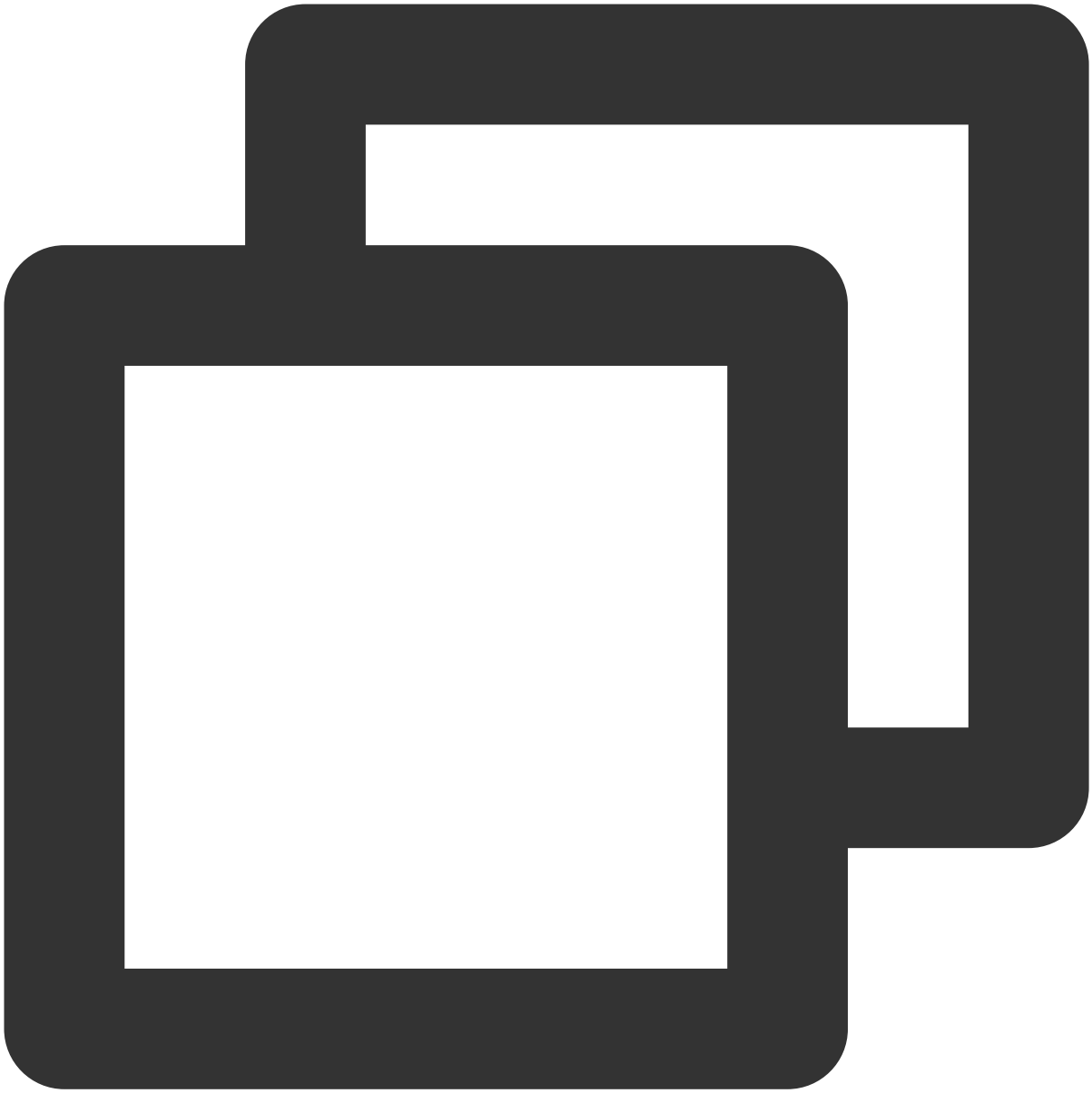


```
apiVersion: apps/v1
kind: Deployment
metadata:
    labels:
      k8s-app: java-demo
    name: java-demo
    namespace: spring-demo
spec:
```

```
    replicas: 1
    selector:
      matchLabels:
        k8s-app: java-demo
    template:
      metadata:
        labels:
          k8s-app: java-demo
    spec:
      containers:
      - image: ccr.ccs.tencentyun.com/prom_spring_demo/java-demo
        imagePullPolicy: Always
        name: java-demo
        ports:
        - containerPort: 8080
          name: metric-port
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      terminationGracePeriodSeconds: 30
```

步骤4：添加采取任务

1. 登录 Prometheus 控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: java-demo
  namespace: cm-prometheus
spec:
  namespaceSelector:
    matchNames:
    - java-demo
  podMetricsEndpoints:
  - interval: 30s
```

```
      path: /metrics
      port: metric-port
    selector:
      matchLabels:
        k8s-app: java-demo
```

步骤5：查看监控

1. 在对应 Prometheus 实例 >**集成中心**中找到 `JVM` 监控，安装对应的 Grafana Dashboard 即可开启 JVM 监控大盘。
2. 打开 Prometheus 实例对应的 Grafana 地址，在 `Dashboards/Manage/Application` 下查看应用相关的监控大屏。

应用 JVM：从应用角度出发，查看该应用下所有实例是否有问题，当发现某个实例有问题时可以下钻到对应的实例监控。

实例 JVM：单实例 JVM 详细的监控数据。


Uptime
1.1 week
Start time
2020-10-13 09:53:48
Heap used
11.58%
Non-Heap u
5.64
GC
GC 次数
0.040 ops
0.030 ops
0.020 ops
0.010 ops
0 ops
18:50
19:00
19:10
19:20
19:30
19:40
end of major GC (Allocation Failure)
end of minor GC (Allocation Failure)
GC 耗时
4 ms
3 ms
2 ms
1 ms
0 ns
avg end of major GC (Allocation Failure)
avg end of minor GC (Allocation Failure)
max end of major GC (Allocation Failure)
max end of minor GC (Allocation Failure)
Allocated/Promoted
500 kB/s
400 kB/s
300 kB/s
200 kB/s
100 kB/s
0 B/s
allocated
promoted
JVM Memory
JVM Heap
286 MiB
191 MiB
95 MiB
0 B
used Max: 31.4 MiB Current: 28.7 MiB
committed Max: 41.4 MiB Current: 41.4 MiB
max Max: 247.5 MiB Current: 247.5 MiB
JVM Non-Heap
1.4 GiB
954 MiB
477 MiB
0 B
used Max: 71 MiB Current: 71 MiB
committed Max: 75 MiB Current: 75 MiB
max Max: 1.234 GiB Current: 1.234 GiB
JVM Total
1.9 GiB
1.4 GiB
954 MiB
477 MiB
0 B
used Max: 103 MiB Current: 100 MiB
max Max: 1.476 GiB Current: 1.476 GiB
JVM Misc
CPU Usage
100.0%
80.0%
60.0%
40.0%
20.0%
0%
Load
1.5
1.0
0.5
Threads
25
20
15
10
5
Thread Sta
15
10
5
0

Golang 应用接入

最近更新时间：2024-01-29 15:55:14

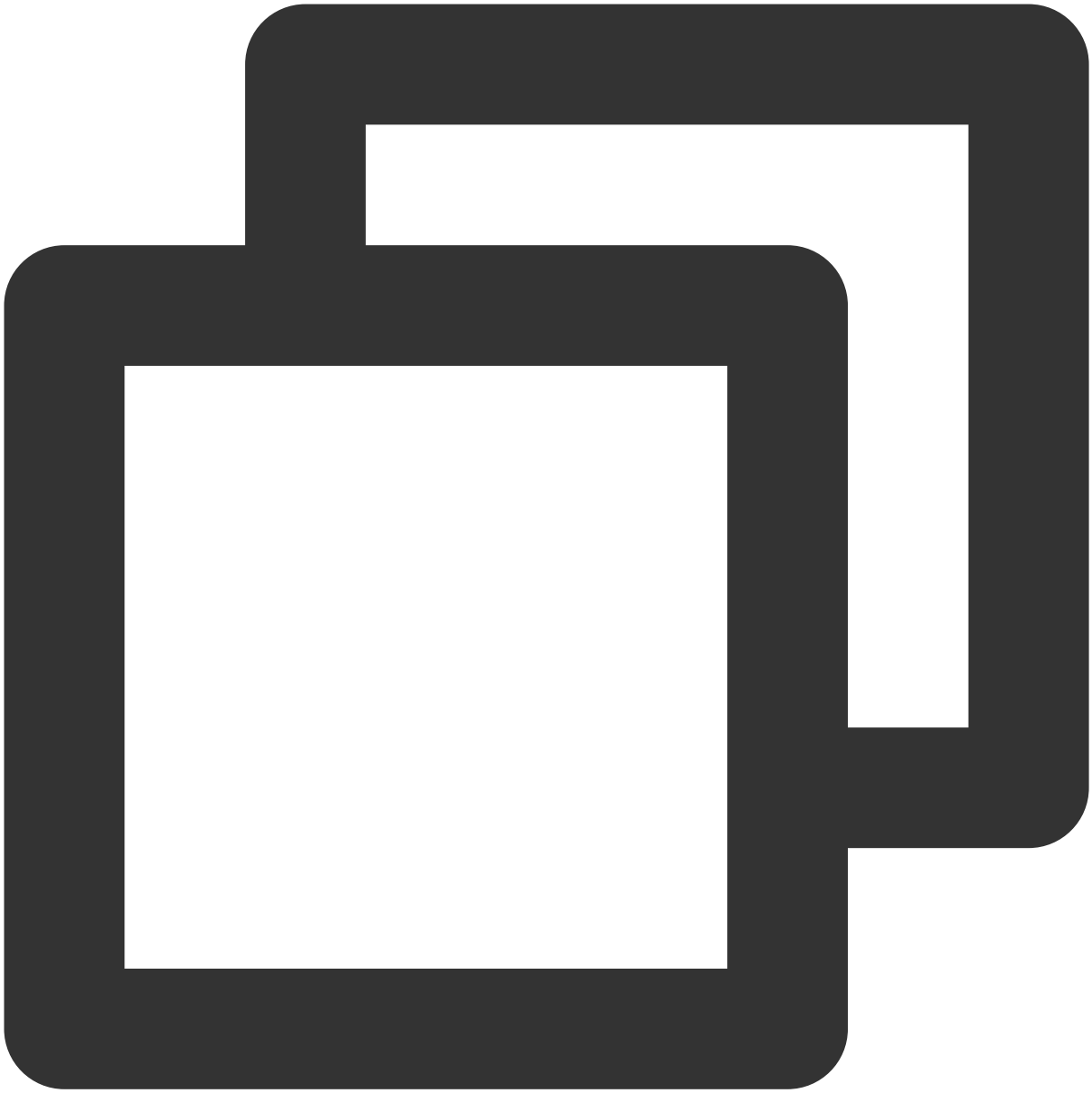
Prometheus 提供了 官方版 Golang 库 用于采集并暴露监控数据，本文为您介绍如何使用官方版 Golang 库来暴露 Golang runtime 相关的数据，以及其它一些基本简单的示例，并使用 Prometheus 监控服务来采集指标展示数据等。

说明：

Golang Client API 相关的文档详见 GoDoc。

安装

通过 `go get` 命令来安装相关依赖，示例如下：

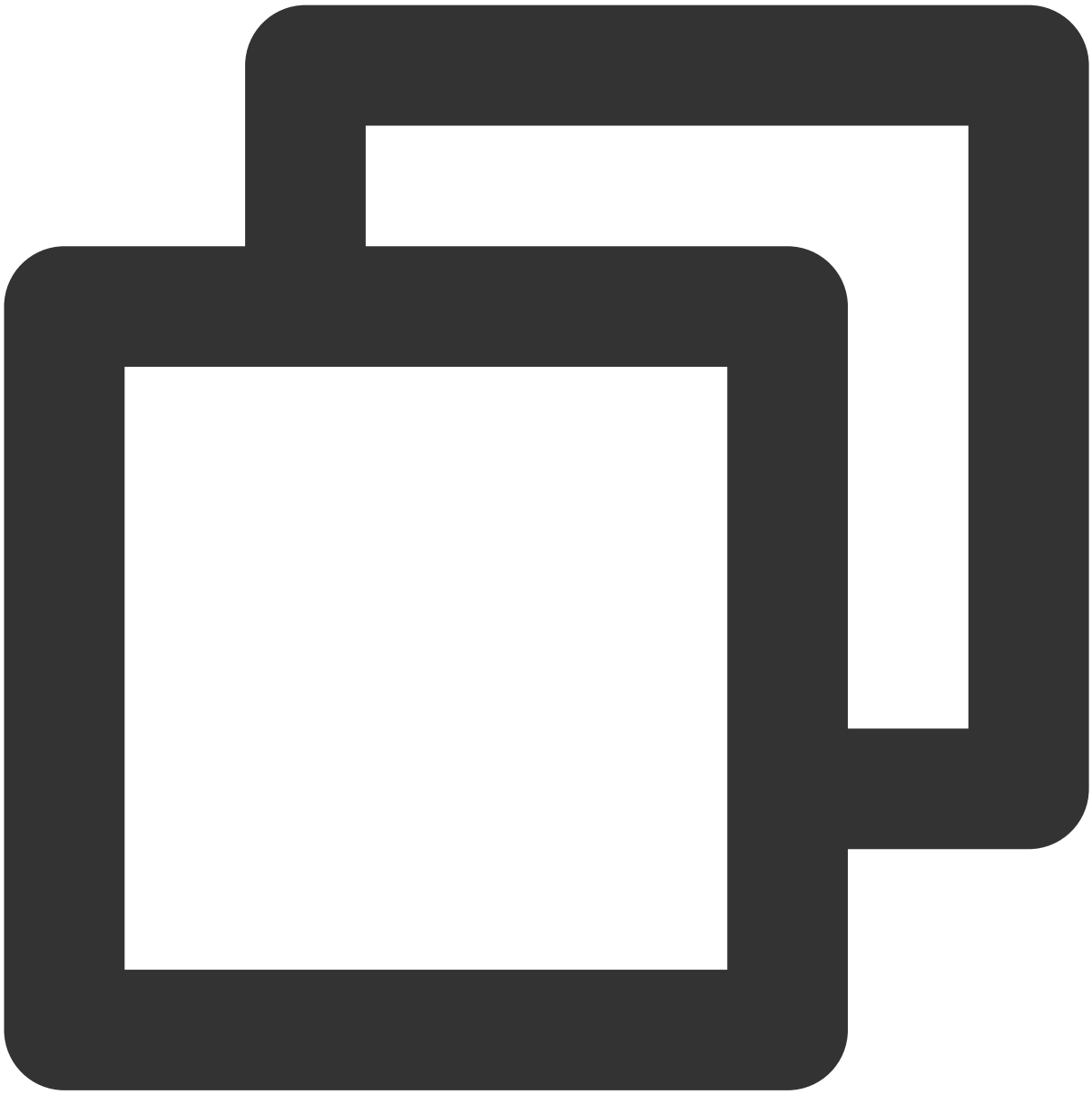


```
go get github.com/prometheus/client_golang/prometheus
go get github.com/prometheus/client_golang/prometheus/promauto
go get github.com/prometheus/client_golang/prometheus/promhttp
```

开始（运行时指标）

1. 准备一个 HTTP 服务，路径通常使用 `/metrics` 。可以直接使用 `prometheus/promhttp` 里提供的 `Handler` 函数。

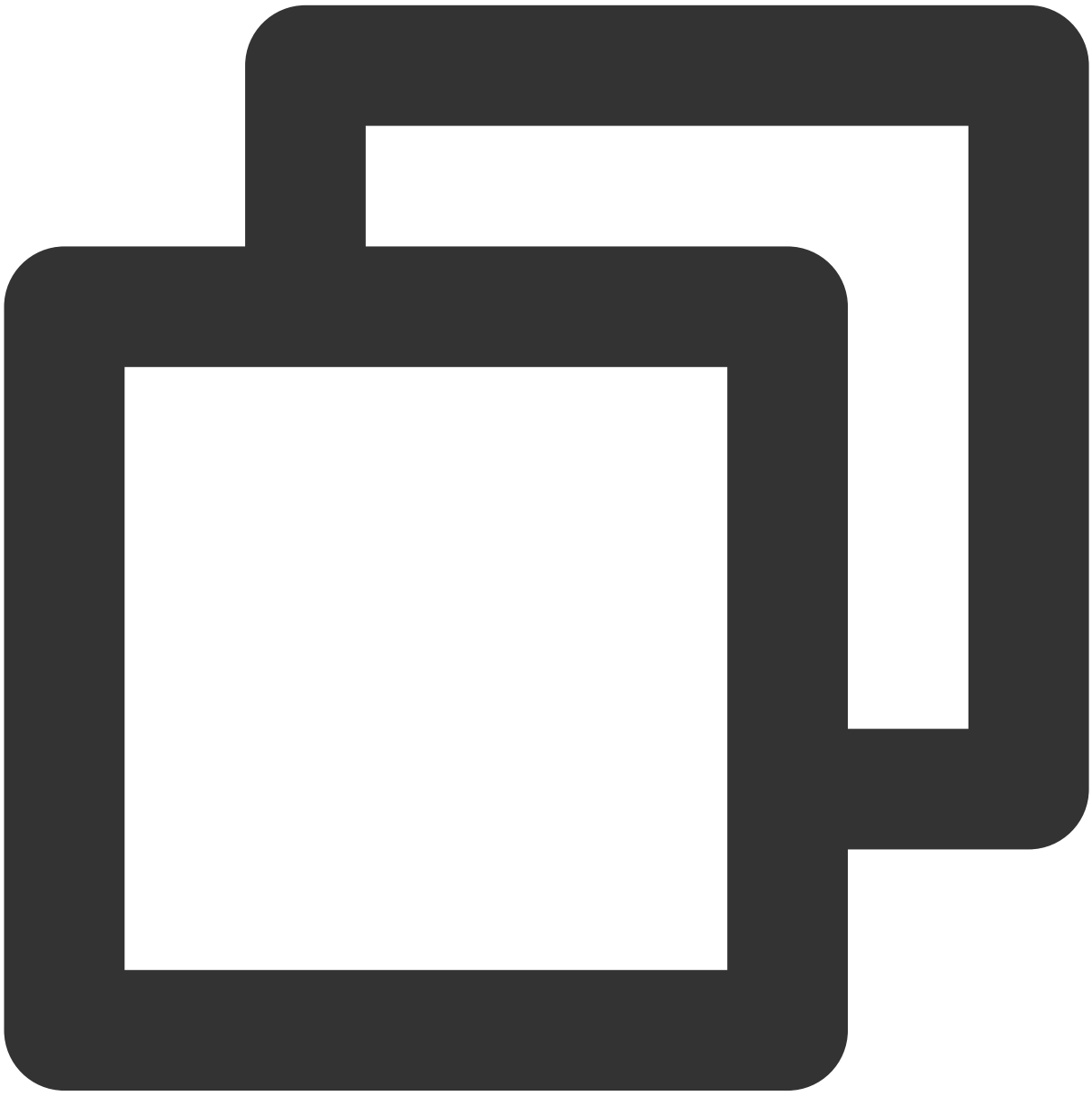
如下是一个简单的示例应用，通过 `http://localhost:2112/metrics` 暴露 Golang 应用的一些默认指标数据（包括运行时指标、进程相关指标以及构建相关的指标）：



```
package main

import (
        "net/http"

        "github.com/prometheus/client_golang/prometheus/promhttp"
)
```

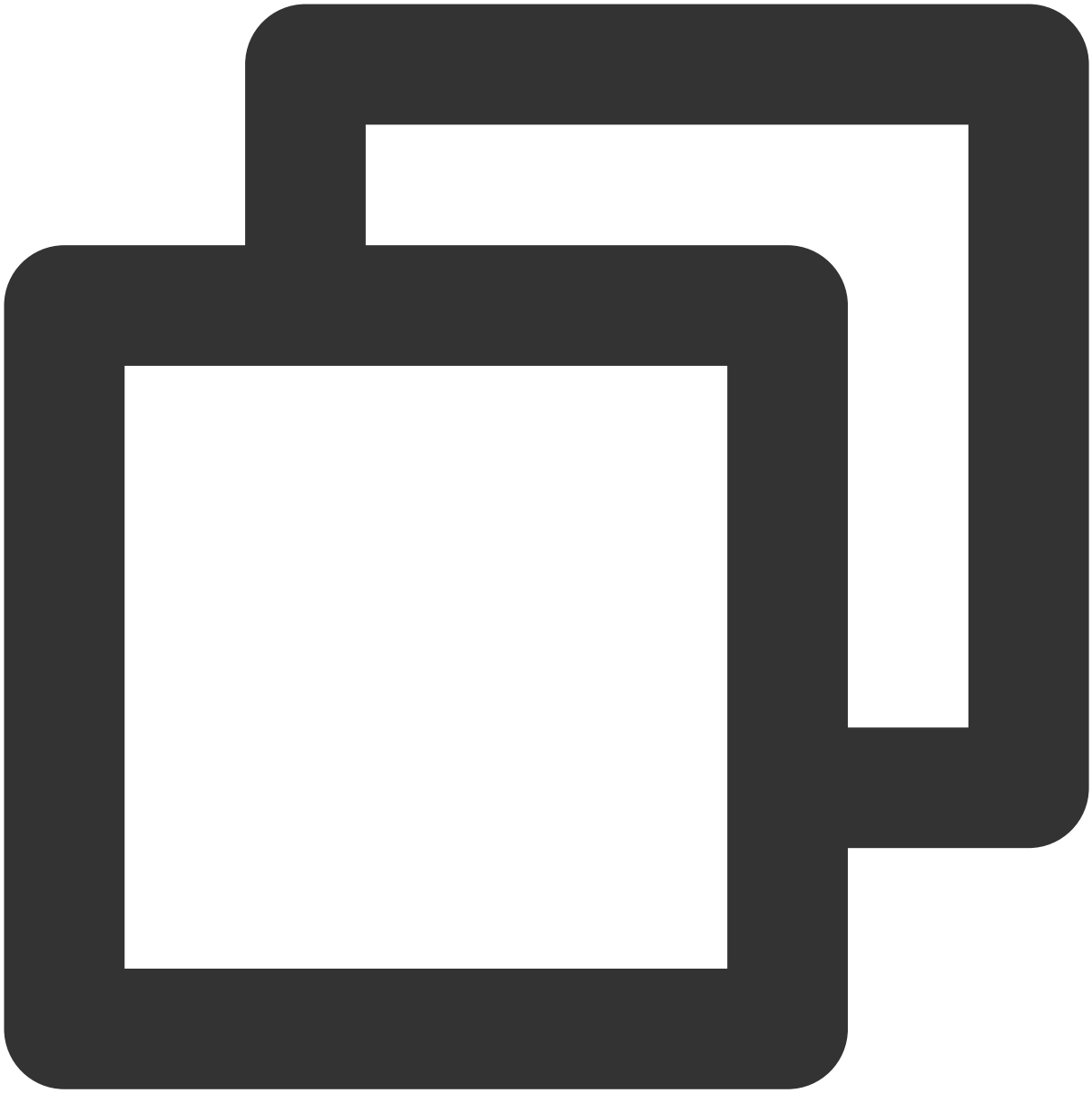
```
func main() {
        http.Handle("/metrics", promhttp.Handler())
        http.ListenAndServe(":2112", nil)
}
```

2. 执行以下命令启动应用：



```
go run main.go
```

3. 执行以下命令，访问基础内置指标数据：

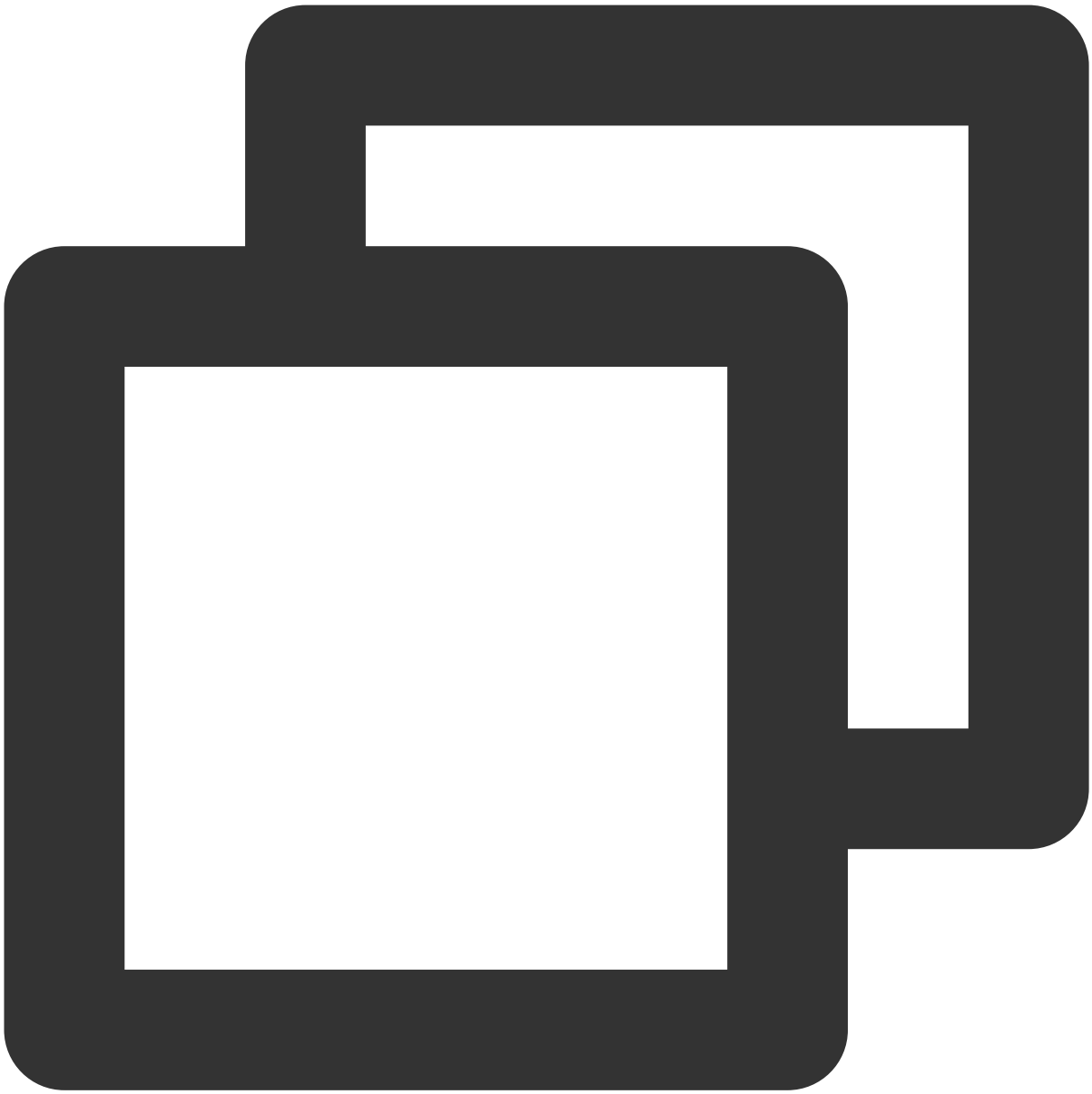


```
curl http://localhost:2112/metrics
```

应用层面指标

1. 上述示例仅仅暴露了一些基础的内置指标。应用层面的指标还需要额外添加（后续我们将提供一些 SDK 方便接入）。如下示例暴露了一个名为 `myapp_processed_ops_total` 的 计数类型 指标，用于对目前已经完成的操

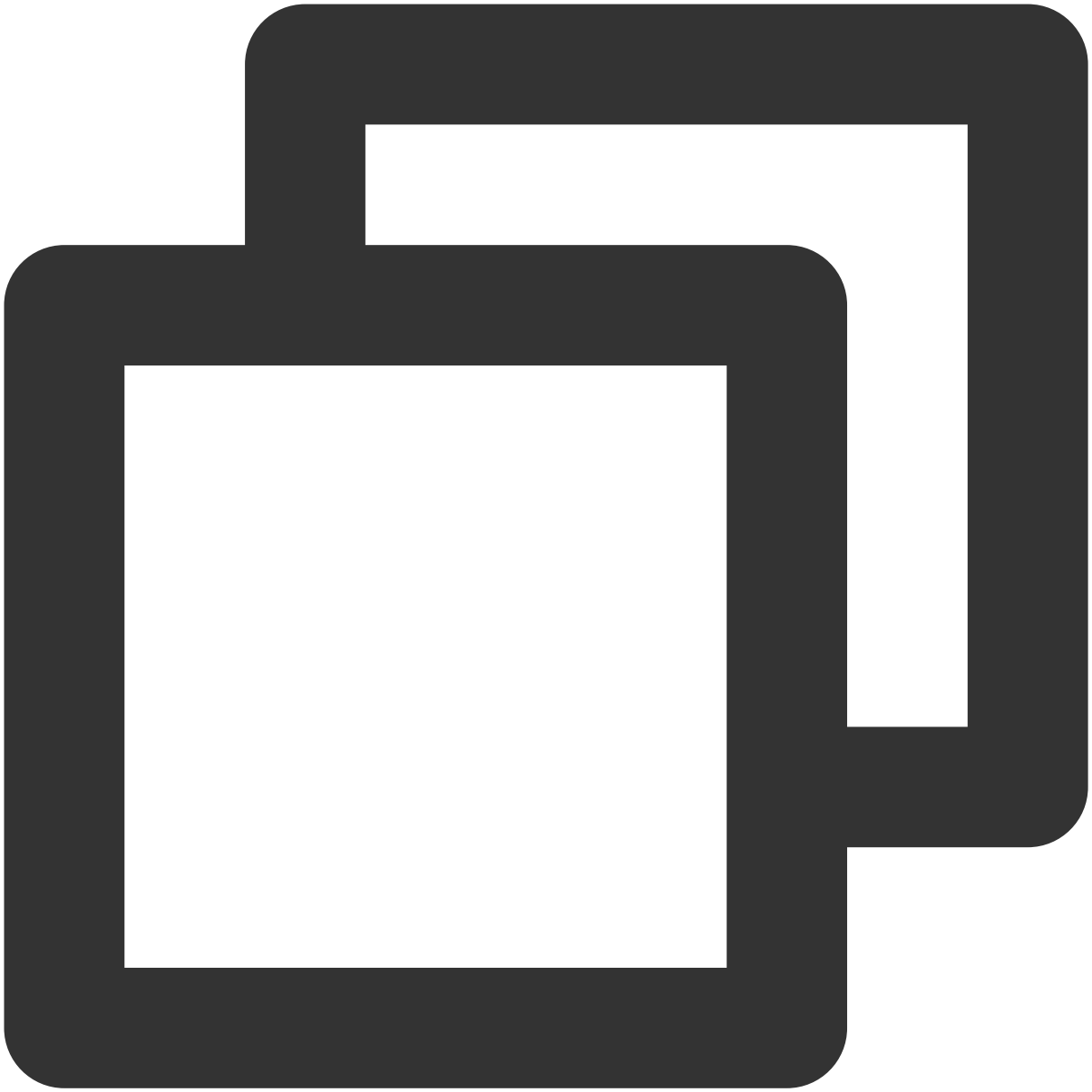
作进行计数。如下每两秒操作一次，同时计数器加1：



```
package main

import (
        "net/http"
        "time"

        "github.com/prometheus/client_golang/prometheus"
        "github.com/prometheus/client_golang/prometheus/promauto"
        "github.com/prometheus/client_golang/prometheus/promhttp"
```

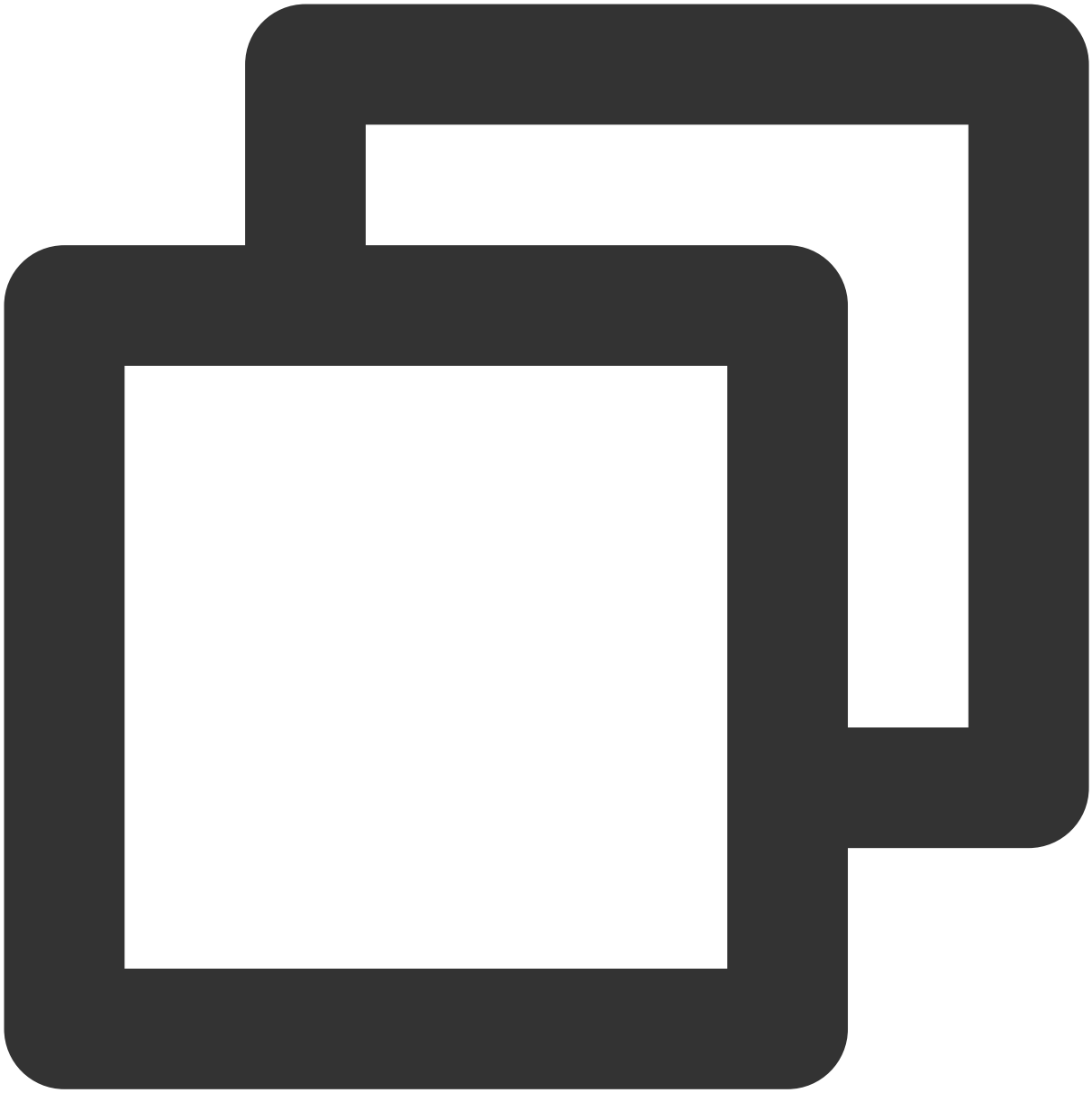
```
)

func recordMetrics() {
        go func() {
                for {
                        opsProcessed.Inc()
                        time.Sleep(2 * time.Second)
                }
        }()
}

var (
        opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
                Name: "myapp_processed_ops_total",
                Help: "The total number of processed events",
        })
)

func main() {
        recordMetrics()

        http.Handle("/metrics", promhttp.Handler())
        http.ListenAndServe(":2112", nil)
}
```

2. 执行以下命令启动应用：



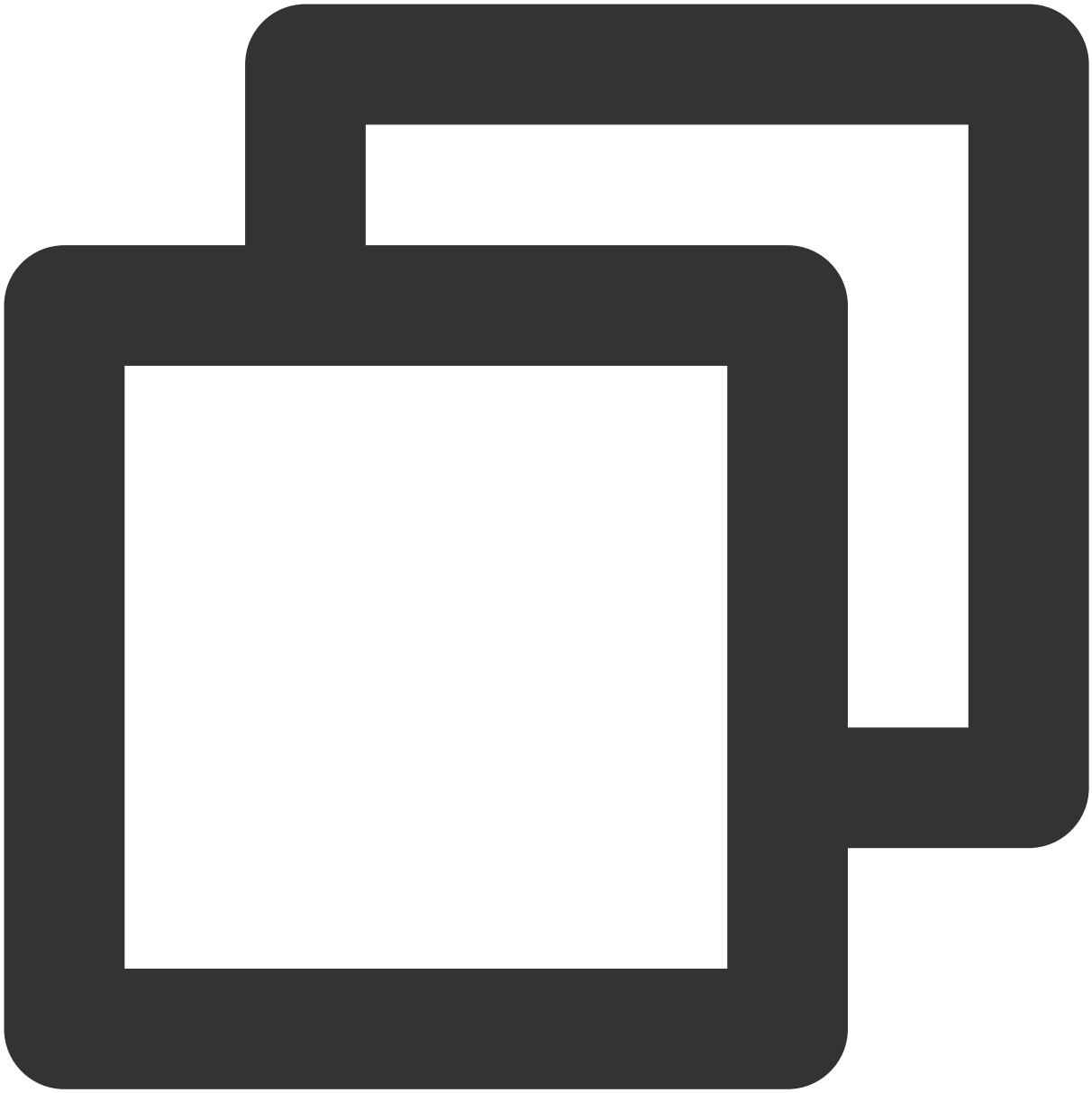
```
go run main.go
```

3. 执行以下命令，访问暴露的指标：



```
curl http://localhost:2112/metrics
```

从输出结果我们可以看到 `myapp_processed_ops_total` 计数器相关的信息，包括帮助文档、类型信息、指标名和当前值，如下所示：



```
# HELP myapp_processed_ops_total The total number of processed events
# TYPE myapp_processed_ops_total counter
myapp_processed_ops_total 666
```

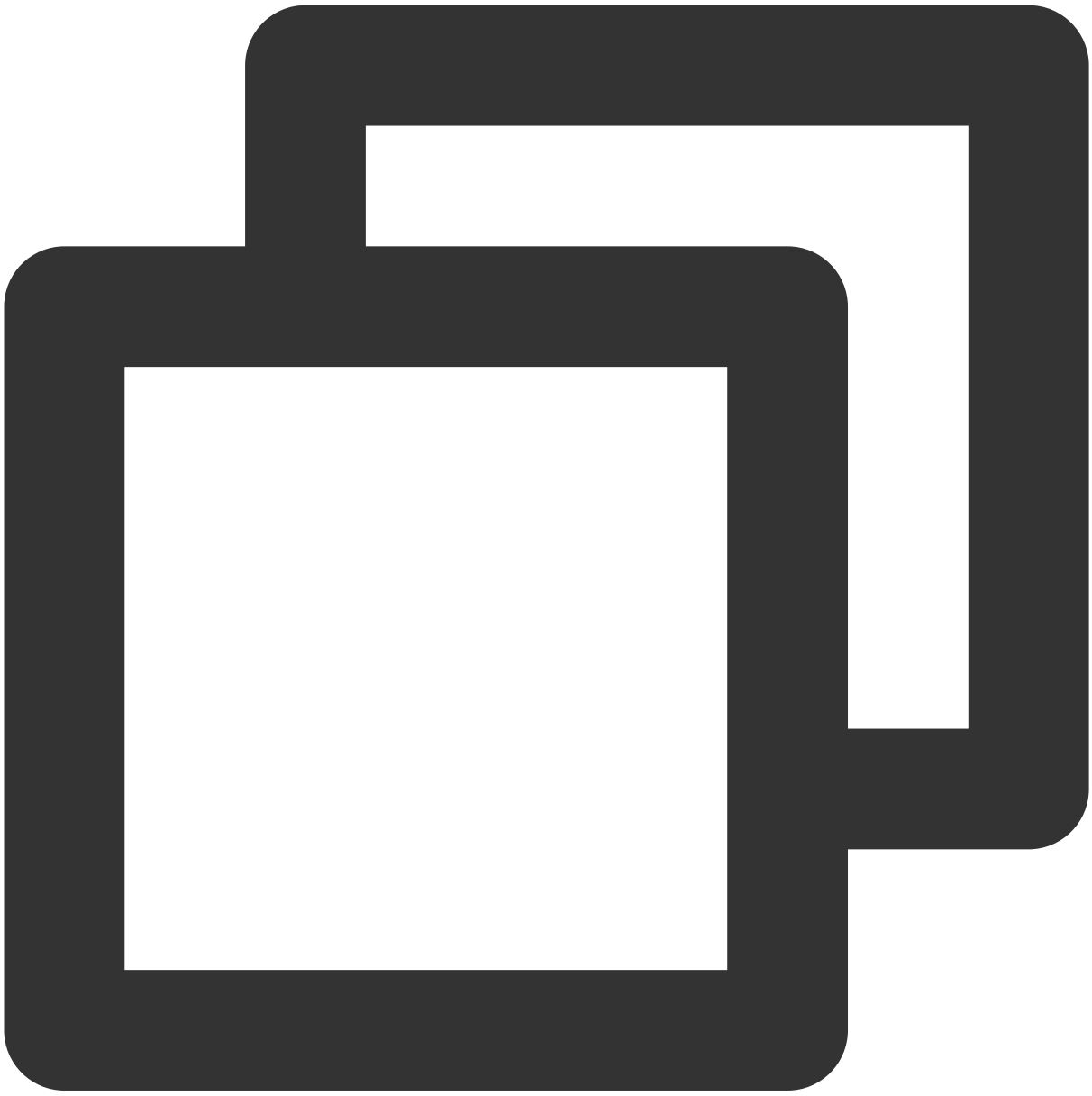
使用 Prometheus 监控服务

上述我们提供了两个示例展示如何使用 Prometheus Golang 库来暴露应用的指标数据，但暴露的监控指标数据为文本类型，需要搭建维护额外的 Prometheus 服务来抓取指标，可能还需要额外的 Grafana 来对数据进行可视化展示。

通过使用 Prometheus 监控服务可以直接省去如上步骤，只需简单的单击操作即可使用。详情请参见快速使用指南。

打包部署应用

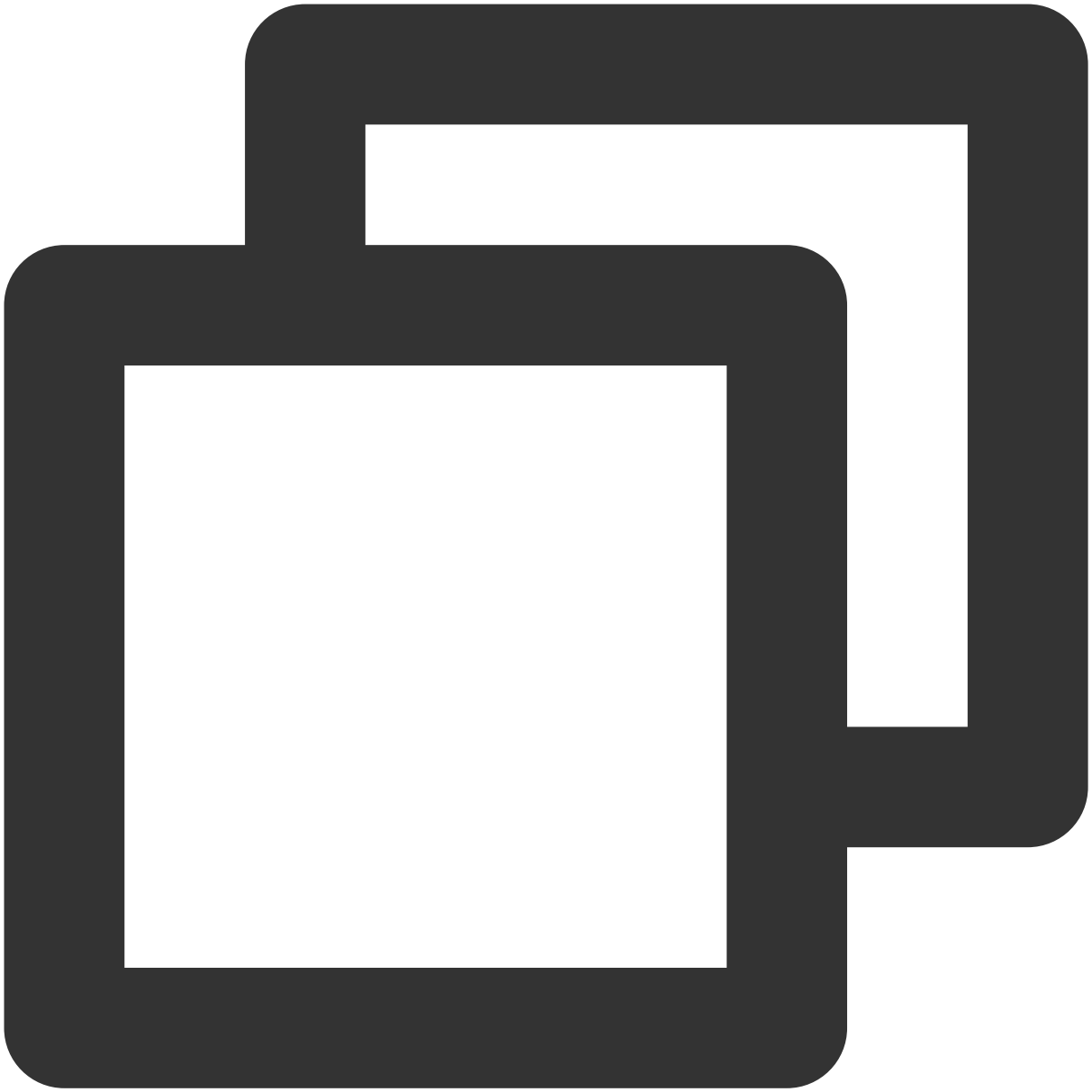
1. Golang 应用一般可以使用如下形式的 Dockerfile（按需修改）：



```
FROM golang:alpine AS builder
RUN apk add --no-cache ca-certificates \\
     make \\
     git
COPY . /go-build
RUN cd /go-build && \\
```

```
        export GO111MODULE=on && \\
        export GOPROXY=https://goproxy.io && \\
        go build -o 'golang-exe' path/to/main/

FROM alpine
RUN apk add --no-cache tzdata
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs
COPY --from=builder /go-build/golang-exe /usr/bin/golang-exe
ENV TZ Asia/Shanghai
CMD ["golang-exe"]
```

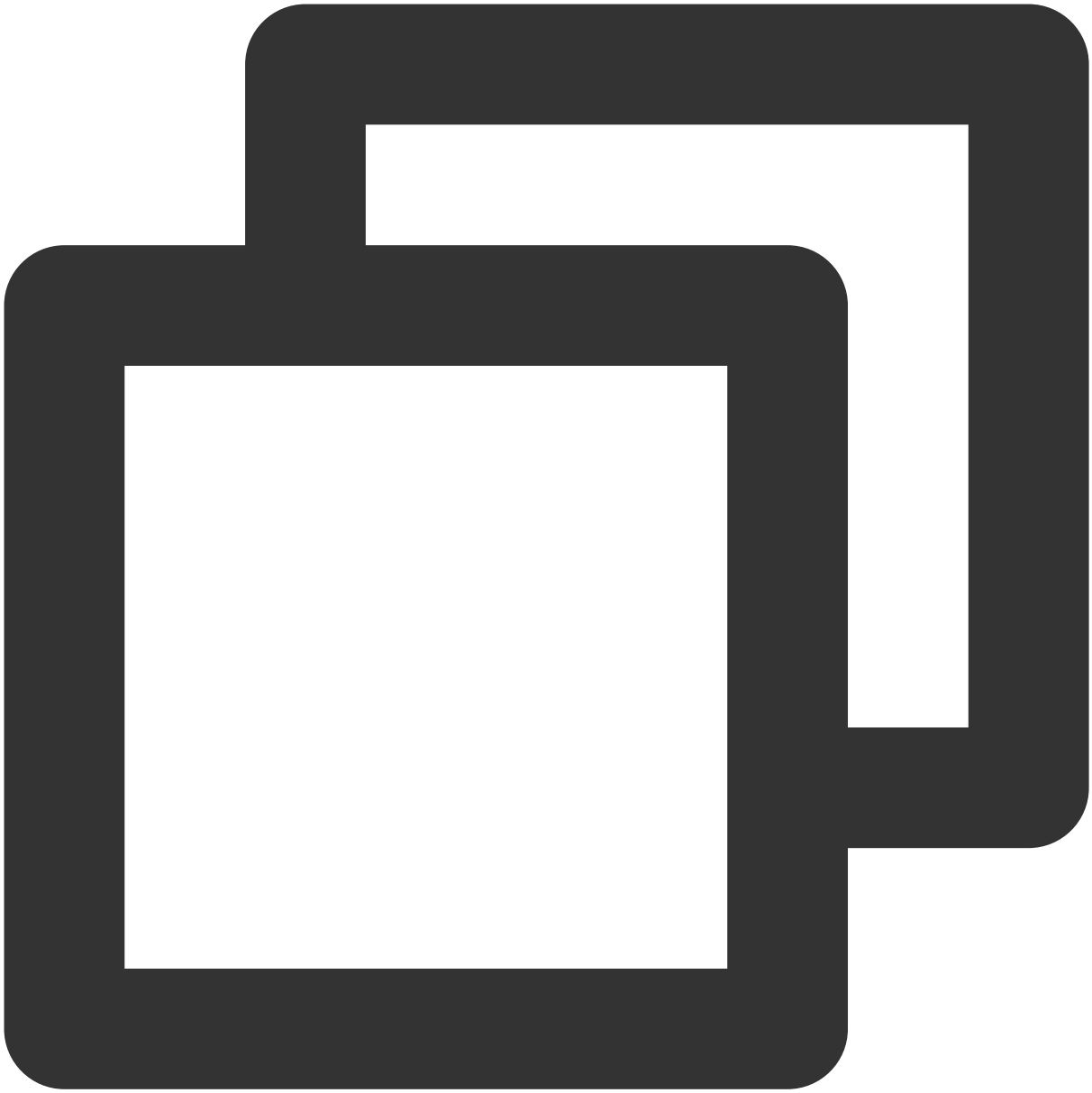
2. 镜像可以使用 腾讯云的镜像仓库，或者使用其它公有或者自有镜像仓库。
3. 需要根据应用类型定义一个 Kubernetes 的资源，这里我们使用 Deployment，示例如下：



```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: golang-app-demo
 labels:
 app: golang-app-demo
spec:
 replicas: 3
 selector:
 matchLabels:
   app: golang-app-demo
```

```
    template:
    metadata:
      labels:
        app: golang-app-demo
    spec:
      containers:
      - name: golang-exe-demo:v1
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

4. 同时需要 Kubernetes Service 做服务发现和负载均衡。



```
apiVersion: v1
kind: Service
metadata:
 name: golang-app-demo
spec:
 selector:
 app: golang-app-demo
 ports:
 - protocol: TCP
   port: 80
   targetPort: 80
```

注意：

必须添加一个 Label 来标明目前的应用，Label 名不一定为 app，但是必须有类似含义的 Label 存在，其它名字的 Label 我们可以在后面添加数据采集任务的时候做 relabel 来达成目的。

5. 可以通过 容器服务控制台 或者直接使用 kubectl 将这些资源定义提交给 Kubernetes，然后等待创建成功。

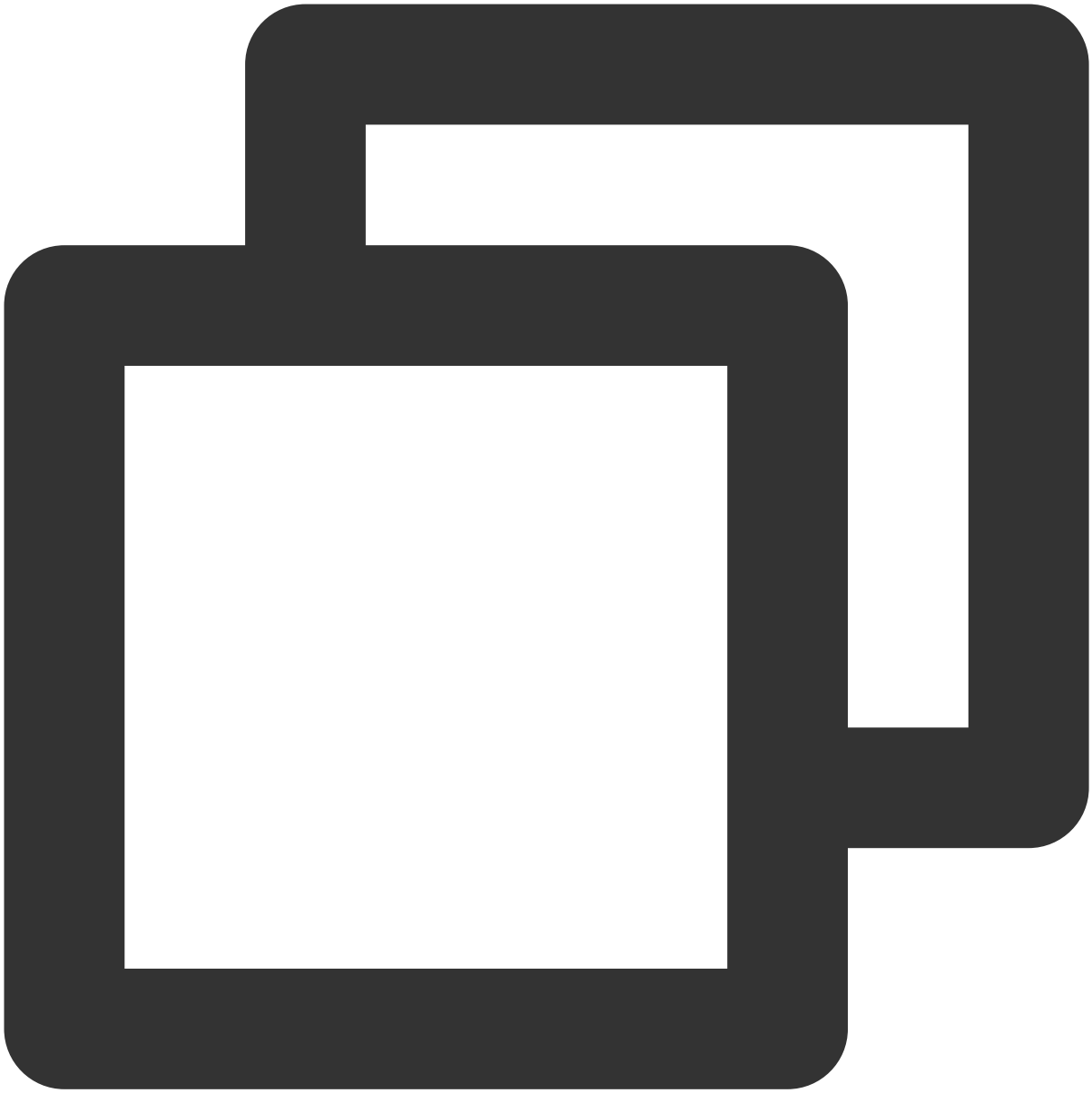
添加数据采集任务

当服务运行起来之后，需要进行如下操作让腾讯云 Prometheus 监控服务发现并采集监控指标：

1. 登录 云监控 Prometheus 控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID**，进入到容器服务集成管理页面。
3. 通过服务发现添加 `Service Monitor`，目前支持基于 `Labels` 发现对应的目标实例地址，因此可以对一些服务添加特定的 `K8S Labels`，可以使 `Labels` 下的服务都会被 Prometheus 服务自动识别出来，不需要再为每个服务一一添加采取任务，以上面的例子配置信息如下：

说明：

`port` 的取值为 `service yaml` 配置文件里的 `spec/ports/name` 对应的值。



```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: go-demo     # 填写一个唯一名称
  namespace: cm-prometheus   # namespace固定，不要修改
spec:
  endpoints:
  - interval: 30s
    # 填写service yaml中Prometheus Exporter对应的Port的Name
    port: 2112
    # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
```

```
    path: /metrics
    relabelings:
    # ** 必须要有一个 label 为 application，这里假设 k8s 有一个 label 为 app，
    # 我们通过 relabel 的 replace 动作把它替换成了 application
    - action: replace
      sourceLabels:  [__meta_kubernetes_pod_label_app]
      targetLabel: application
  # 选择要监控service所在的namespace
  namespaceSelector:
    matchNames:
    - golang-demo
    # 填写要监控service的Label值，以定位目标service
  selector:
    matchLabels:
      app: golang-app-demo
```

注意：

示例中名称为 application 的 Label 必须配置，否则无法使用我们提供一些其它的开箱即用的集成功能。更多高阶用法请参见 ServiceMonitor 或 PodMonitor。

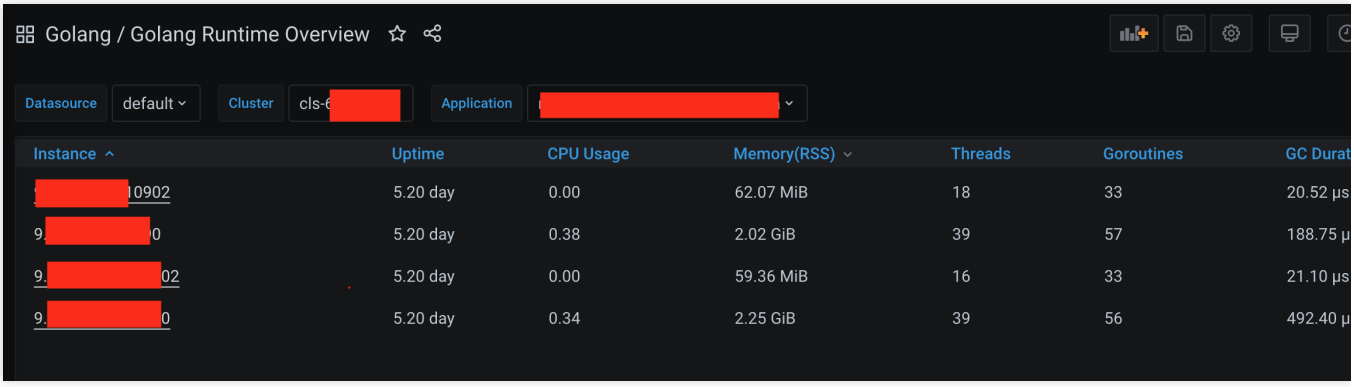
查看监控

1. 在 Prometheus 实例 列表，找到对应的 Prometheus 实例，单击实例ID 右侧

图标，打开您的专属 Grafana，输入您的账号密码，即可进行 Grafana 可视化大屏操作区。

2. 进入 Grafana，单击

图表，展开监控面板，单击对应的监控图表名称即可查看监控数据。



总结

本文通过两个示例展示了如何将 Golang 相关的指标暴露给 Prometheus 监控服务，以及如何使用内置的可视化的图表查看监控数据。文档只使用了计数类型 Counter 的指标，对于其它场景可能还需要 Gauge、Histgram 以及 Summary 类型的指标，指标类型。

对于其它应用场景，我们会集成更多框架提供更多开箱即用的指标监控、可视化面板以及告警模板。

Exporters 接入
ElasticSearch Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 ElasticSearch 过程中需要对 ElasticSearch 运行状态进行监控，例如集群及索引状态等，Prometheus 监控服务提供了基于 Exporter 的方式来监控 ElasticSearch 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文介绍如何部署 Exporter 以及实现 ElasticSearch Exporter 告警接入等操作。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 托管版集群，并为集群创建 命名空间。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见Agent 管理。

操作步骤

Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 ElasticSearch 连接串 > 部署 Kafka Exporter > 验证 步骤完成 Exporter 部署。

使用 Secret 管理 ElasticSearch 连接串

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML创建资源**，创建 YAML 配置，配置说明如下：

使用 Kubernetes 的 Secret 来管理密码并对密码进行加密处理，在启动 ElasticSearch Exporter 的时候直接使用 Secret Key，需要调整对应的 URI，YAML 配置示例如下：

操作场景

在使用 ElasticSearch 过程中需要对 ElasticSearch 运行状态进行监控，例如集群及索引状态等， Prometheus 监控服务提供了基于 Exporter 的方式来监控 ElasticSearch 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文介绍如何部署 Exporter 以及实现 ElasticSearch Exporter 告警接入等操作。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 托管版集群，并为集群创建 命名空间。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见Agent 管理。

操作步骤

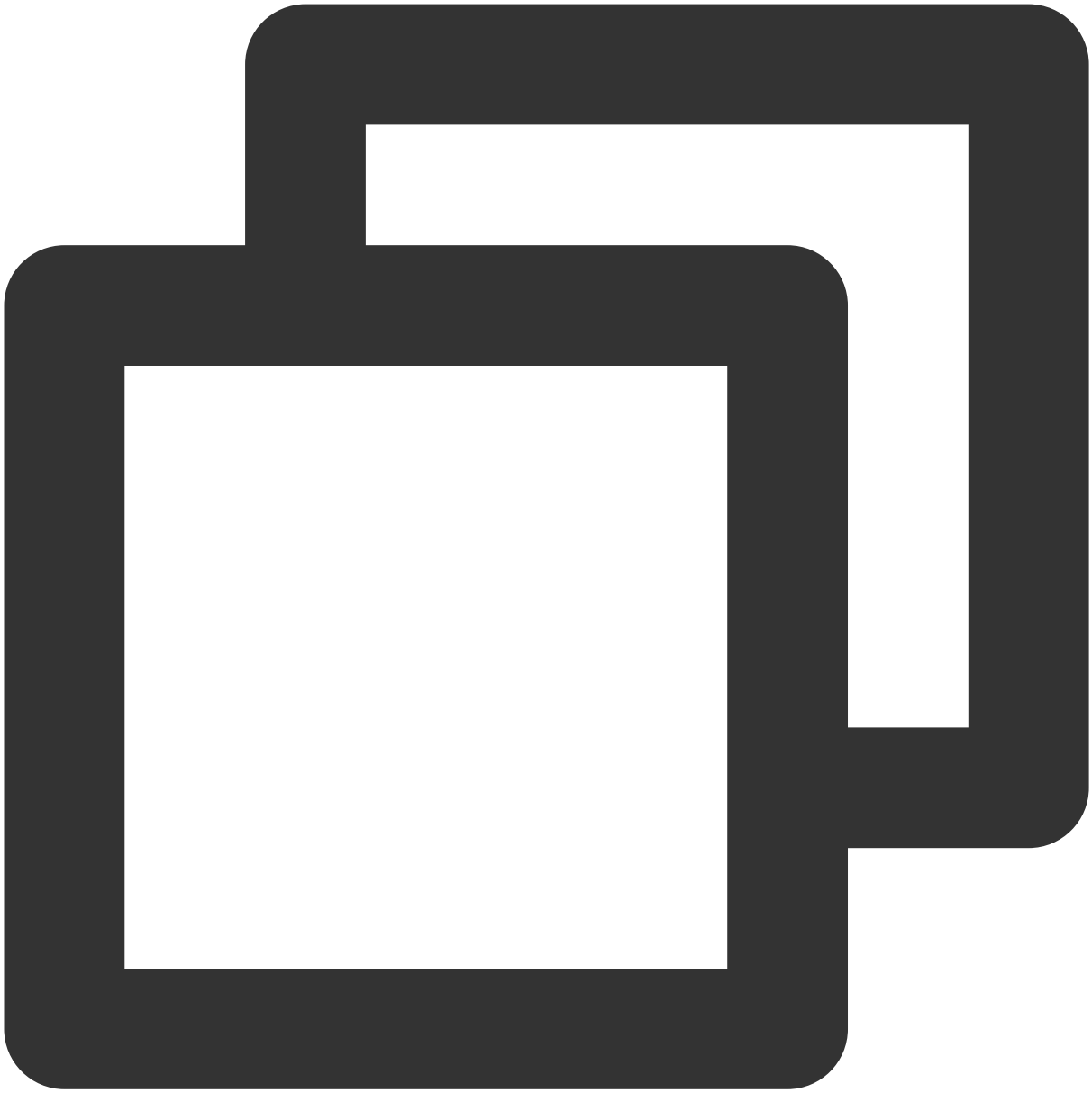
Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 ElasticSearch 连接串 > 部署 Kafka Exporter > 验证 步骤完成 Exporter 部署。

使用 Secret 管理 ElasticSearch 连接串

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML创建资源**，创建 YAML 配置，配置说明如下：

使用 Kubernetes 的 Secret 来管理密码并对密码进行加密处理，在启动 ElasticSearch Exporter 的时候直接使用 Secret Key，需要调整对应的 URI，YAML 配置示例如下：



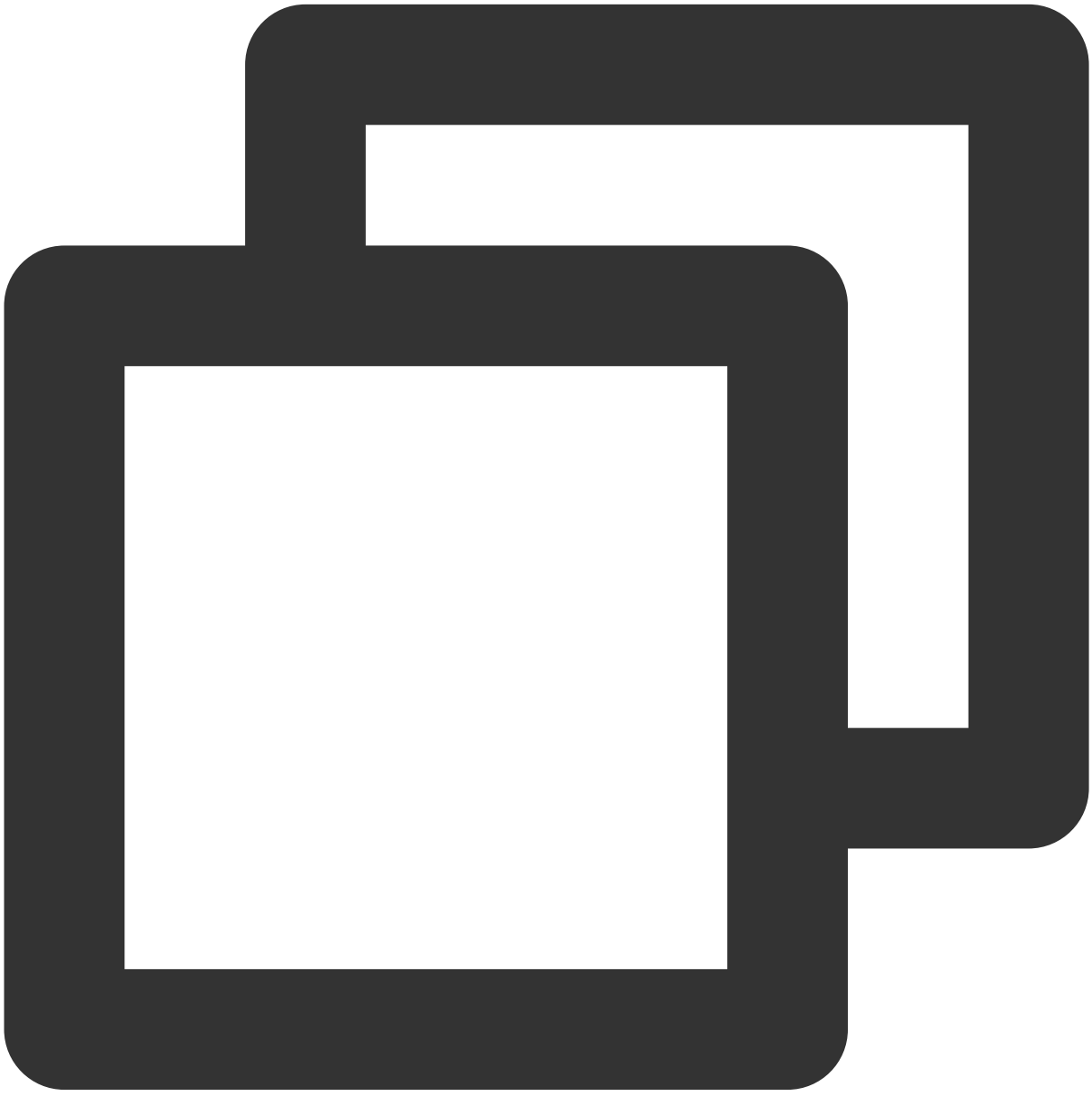
```
apiVersion: v1
kind: Secret
metadata:
  name: es-secret-test
  namespace: es-demo
type: Opaque
stringData:
  esURI: you-guess  #对应 ElasticSearch 的 URI
```

说明：

ElasticSearch 连接串的格式为 `<proto>://<user>:<password>@<host>:<port>`，例如 `http://admin:pass@localhost:9200`。

部署 ElasticSearch Exporter

在 Deployment 管理页面，单击**新建**，选择对应的**命名空间**来进行部署服务。可以通过控制台的方式创建，如下以 YAML 的方式部署 Exporter，YAML 配置示例如下：



```
apiVersion: apps/v1
kind: Deployment
metadata:
```

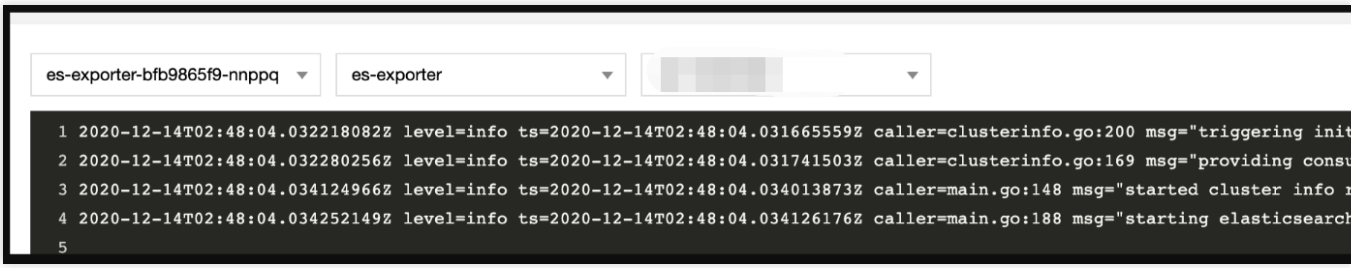
```
    labels:
      k8s-app: es-exporter
    name: es-exporter
    namespace: es-demo
  spec:
    replicas: 1
    selector:
      matchLabels:
        k8s-app: es-exporter
    template:
      metadata:
        labels:
          k8s-app: es-exporter
      spec:
        containers:
        - env:
            - name: ES_URI
              valueFrom:
                secretKeyRef:
                  name: es-secret-test
                  key: esURI
            - name: ES_ALL
              value: "true"
          image: bitnami/elasticsearch-exporter:latest
          imagePullPolicy: IfNotPresent
          name: es-exporter
          ports:
          - containerPort: 9114
            name: metric-port
          securityContext:
            privileged: false
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
        dnsPolicy: ClusterFirst
        imagePullSecrets:
        - name: qcloudregistrykey
        restartPolicy: Always
        schedulerName: default-scheduler
        securityContext: {}
        terminationGracePeriodSeconds: 30
```

说明：

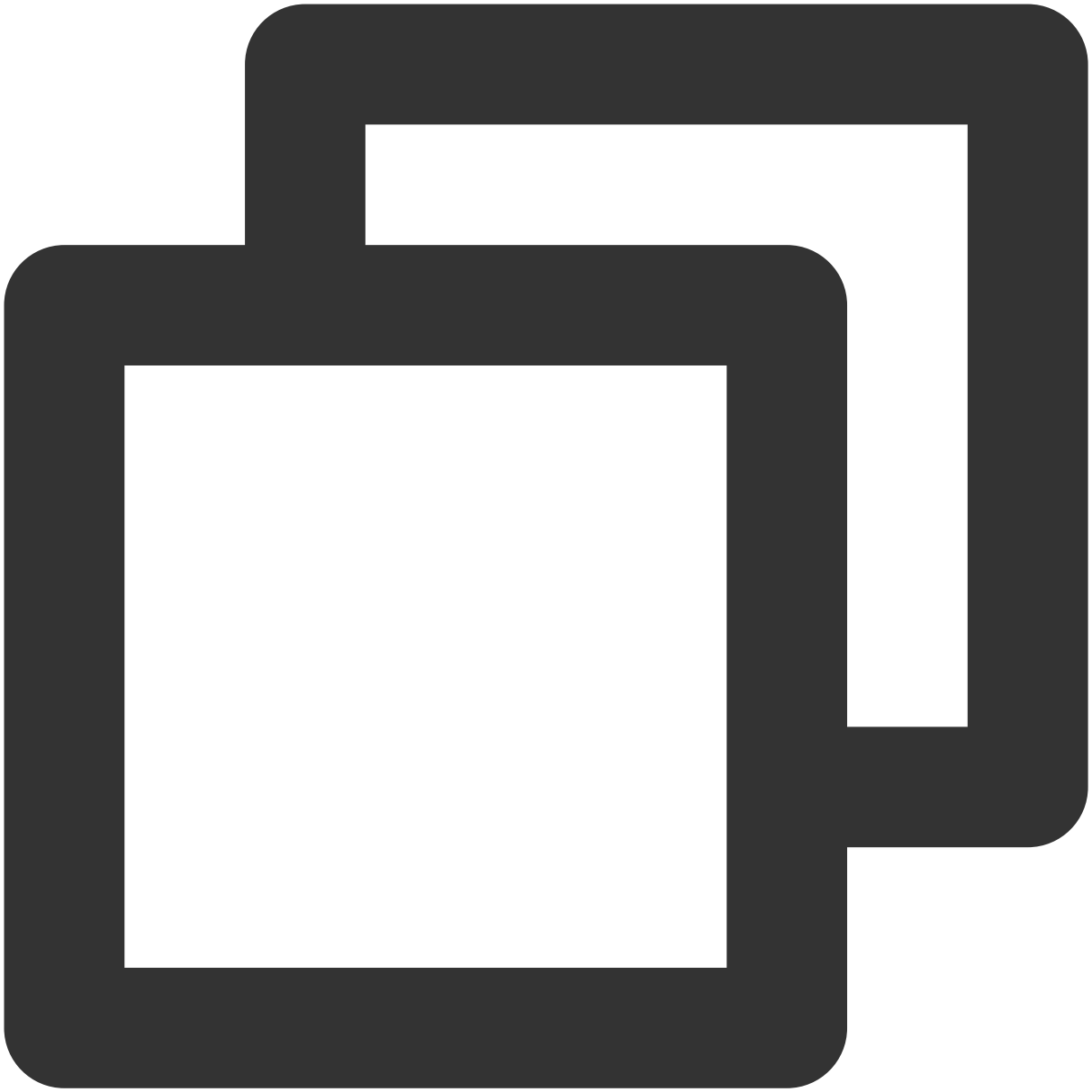
上述示例通过 `ES_ALL` 采集了所有 ElasticSearch 的监控项，可以通过对应的参数进行调整，Exporter 更多详细的参数请参见 elasticsearch_exporter。

验证

1. 在 Deployment 页面单击上述步骤创建的 Deployment，进入 Deployment 管理页面。
2. 单击**日志**页签，可以查看到 Exporter 成功启动并暴露对应的访问地址，如下图所示：



3. 单击**Pod管理**页签进入 Pod 页面。
4. 在右侧的操作项下单击**远程登录**登录 Pod，在命令行窗口中执行以下 curl 命令对应 Exporter 暴露的地址，可以正常得到对应的 ElasticSearch 指标。如发现未能得到对应的数据，请检查**连接串**是否正确，具体如：



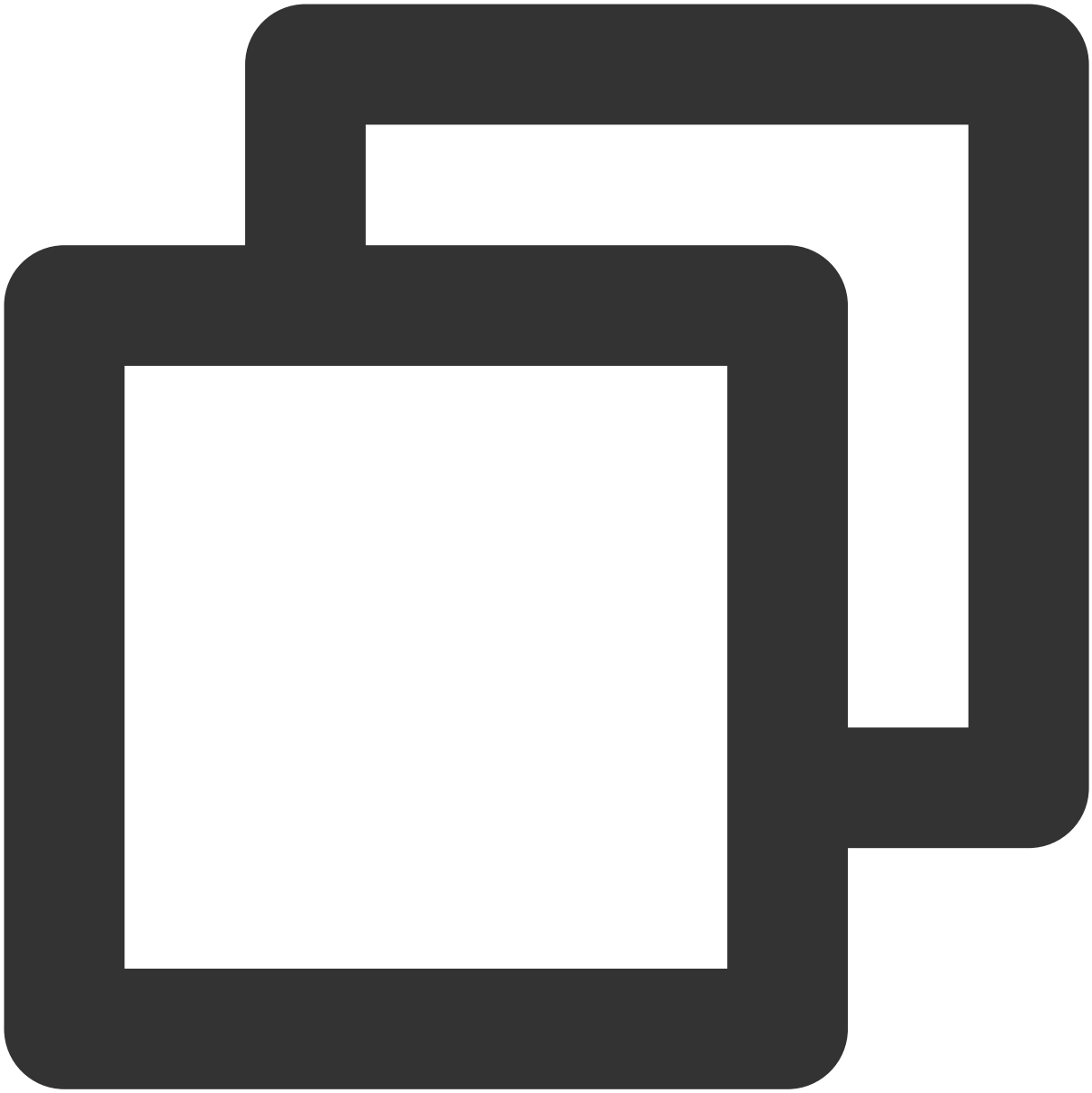
```
curl localhost:9114/metrics
```

执行结果如下图所示：

```
# HELP elasticsearch_breakers_estimated_size_bytes Estimated size
# TYPE elasticsearch_breakers_estimated_size_bytes gauge
elasticsearch_breakers_estimated_size_bytes{breaker="accounting",c
2.0102643e+07
elasticsearch_breakers_estimated_size_bytes{breaker="accounting",c
1.9926654e+07
elasticsearch_breakers_estimated_size_bytes{breaker="accounting",c
1.9685163e+07
elasticsearch_breakers_estimated_size_bytes{breaker="fielddata",cl
elasticsearch_breakers_estimated_size_bytes{breaker="fielddata",cl
elasticsearch_breakers_estimated_size_bytes{breaker="fielddata",cl
elasticsearch_breakers_estimated_size_bytes{breaker="in_flight_req
0
elasticsearch_breakers_estimated_size_bytes{breaker="in_flight_req
1167
elasticsearch_breakers_estimated_size_bytes{breaker="in_flight_req
1167
elasticsearch_breakers_estimated_size_bytes{breaker="parent",clust
2.0102643e+07
```

添加采取任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID**进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：

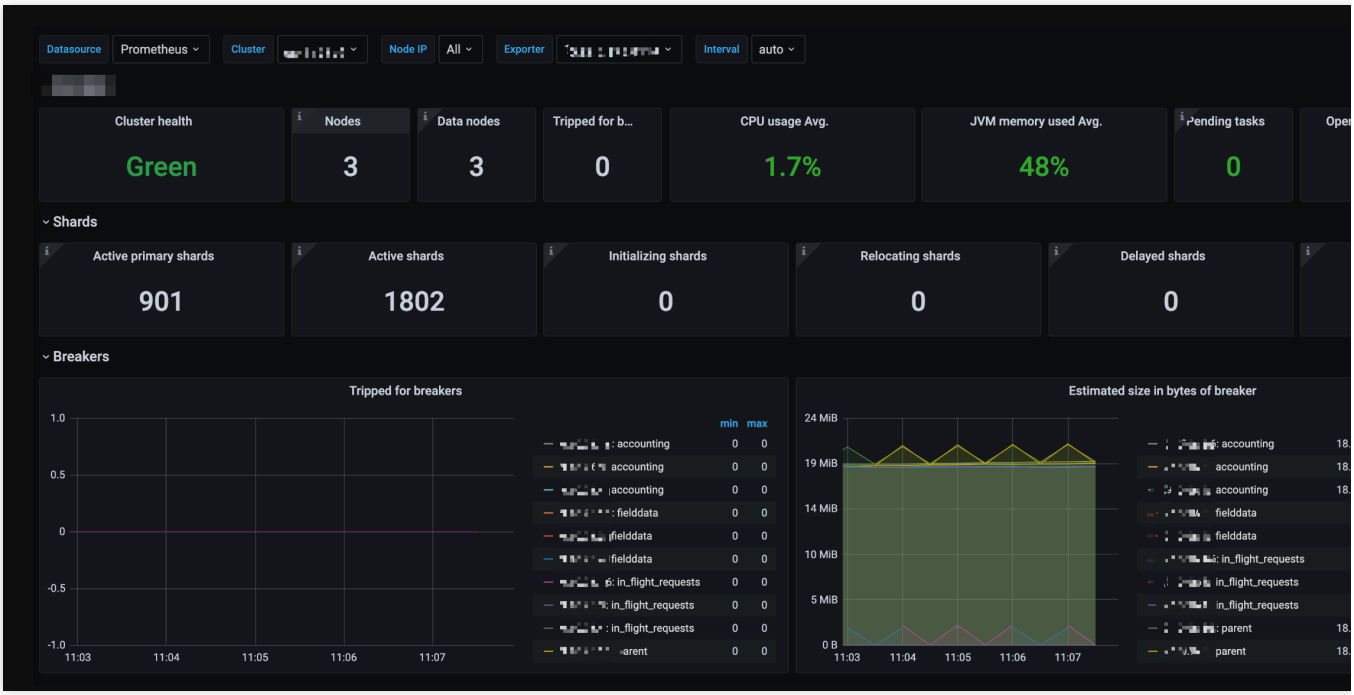


```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: es-exporter
  namespace: cm-prometheus
spec:
  namespaceSelector:
    matchNames:
      - es-demo
  podMetricsEndpoints:
  - interval: 30s
```

```
    path: /metrics
    port: metric-port
    selector:
    matchLabels:
    k8s-app: es-exporter
```

查看监控

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击**集成中心**，进入集成中心页面。找到 ElasticSearch 监控，安装对应的 Grafana Dashboard 即可开启 ElasticSearch 监控大盘，查看实例相关的监控数据，如下图所示：



告警以及接入

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击告警策略，可以添加相应的告警策略，详情请参见 新建告警策略。

Kafka Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 Kafka 过程中需要对 Kafka 运行状态进行监控，例如集群状态、消息消费情况是否有积压等， Prometheus 监控服务提供基于 Exporter 的方式来监控 Kafka 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文介绍如何部署 Exporter 以及实现 Kafka Exporter 告警接入等操作。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

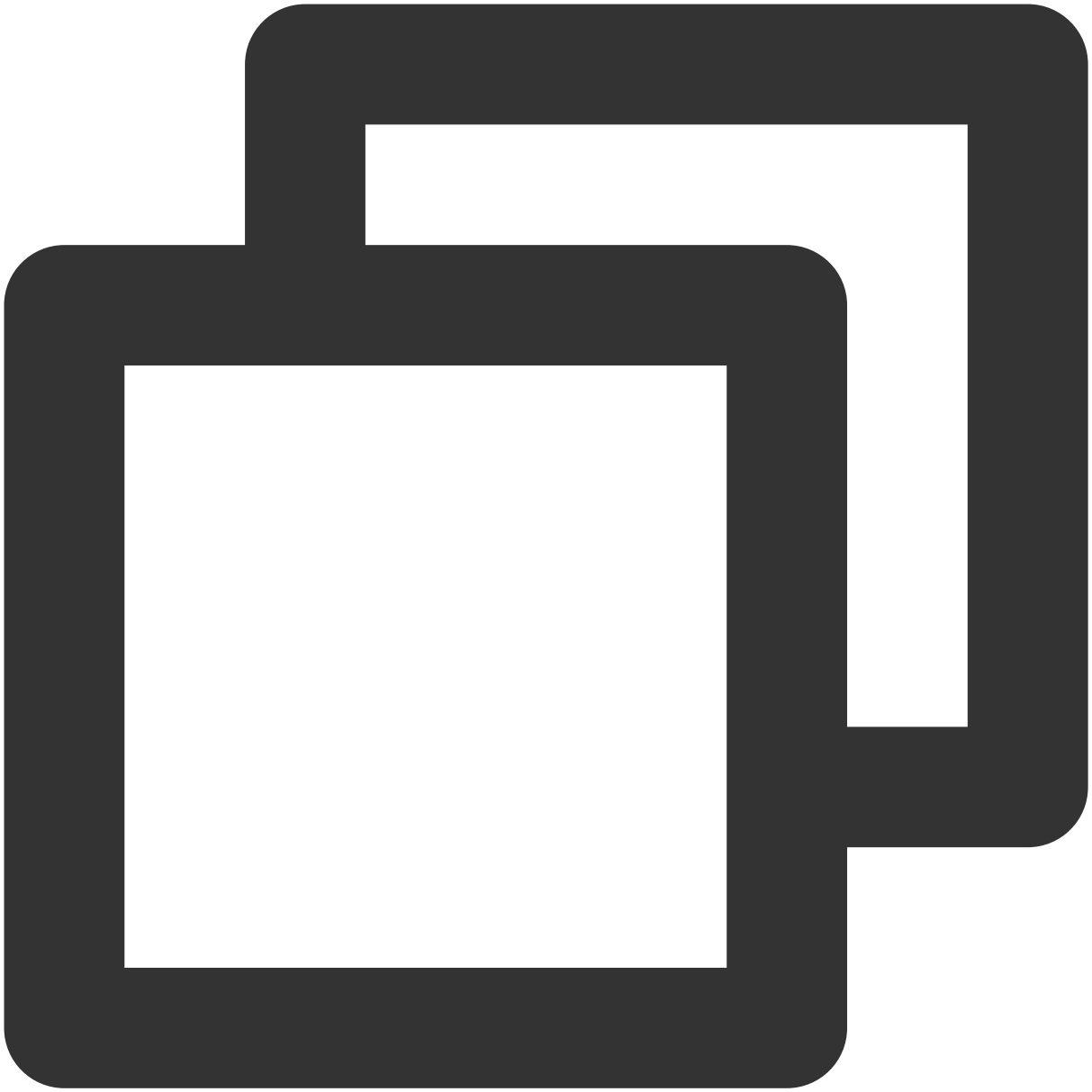
在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 托管版集群，并为集群创建 命名空间。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见Agent 管理。

操作步骤

Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
4. 在 Deployment 管理页面，单击**新建**，选择对应的**命名空间**来进行部署服务。可以通过控制台的方式创建，如下以 YAML 的方式部署 Exporter，YAML 配置示例如下：



```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: kafka-exporter # 根据业务需要调整成对应的名称，建议加上 Kafka 实例的信息
  name: kafak-exporter # 根据业务需要调整成对应的名称，建议加上 Kafka 实例的信息
  namespace: kafka-demo
spec:
  replicas: 1
  selector:
    matchLabels:
```

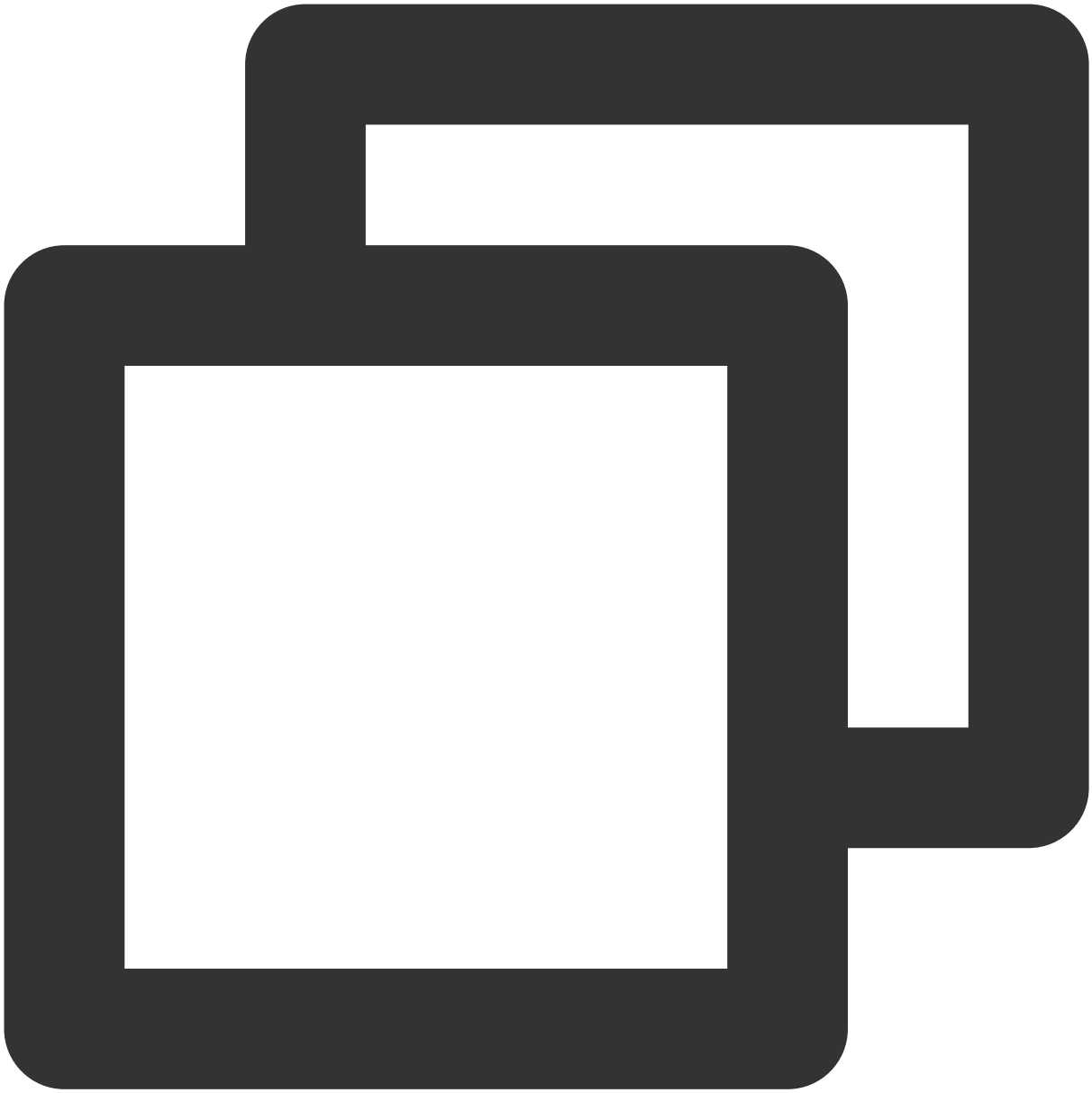
```
      k8s-app: kafka-exporter # 根据业务需要调整成对应的名称，建议加上 Kafka 实例的信息
  template:
    metadata:
      labels:
        k8s-app: kafka-exporter # 根据业务需要调整成对应的名称，建议加上 Kafka 实例的信息
    spec:
      containers:
      - args:
        - --kafka.server=x.x.x.x:9092 # 对应 Kafka 实例的地址信息
        image: danielqsj/kafka-exporter:latest
        imagePullPolicy: IfNotPresent
        name: kafka-exporter
        ports:
        - containerPort: 9121
          name: metric-port  # 这个名称在配置抓取任务的时候需要
        securityContext:
          privileged: false
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

说明：

Exporter 详细参数请参见 kafka_exporter。

添加采取任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: kafka-exporter   # 填写一个唯一名称
  namespace: cm-prometheus   # namespace固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
```

```
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: instance
      replacement: 'ckafka-xxxxxx' # 调整成对应的 Kafka 实例 ID
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: ip
      replacement: '1.x.x.x' # 调整成对应的 Kafka 实例 IP
      namespaceSelector:
    matchNames:
    - kafka-demo
      selector:  # 填写要监控pod的Label值，以定位目标pod
    matchLabels:
      k8s-app: kafka-exporter
```

说明：

由于 `Exporter` 和 `Kafka` 部署在不同的服务器上，因此建议通过 Prometheus Relabel 机制将 Kafka 实例的信息放到监控指标中，以便定位问题。

查看监控

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击**集成中心**，进入集成中心页面。找到 kafka 监控，安装对应的 Grafana Dashboard 即可开启 kafaka 监控大盘，查看实例相关的监控数据，如下图所示：

告警以及接入

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击告警策略，可以添加相应的告警策略，详情请参见新建告警策略。

MongoDB Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 MongoDB 过程中需要对 MongoDB 运行状态进行监控，以便了解 MongoDB 服务是否运行正常，排查 MongoDB 故障问题原因， Prometheus 监控服务提供了基于 Exporter 的方式来监控 MongoDB 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文介绍如何部署 Exporter 以及实现 MongoDB Exporter 告警接入等操作。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 Kubernetes 集群。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见 Agent 管理。

操作步骤

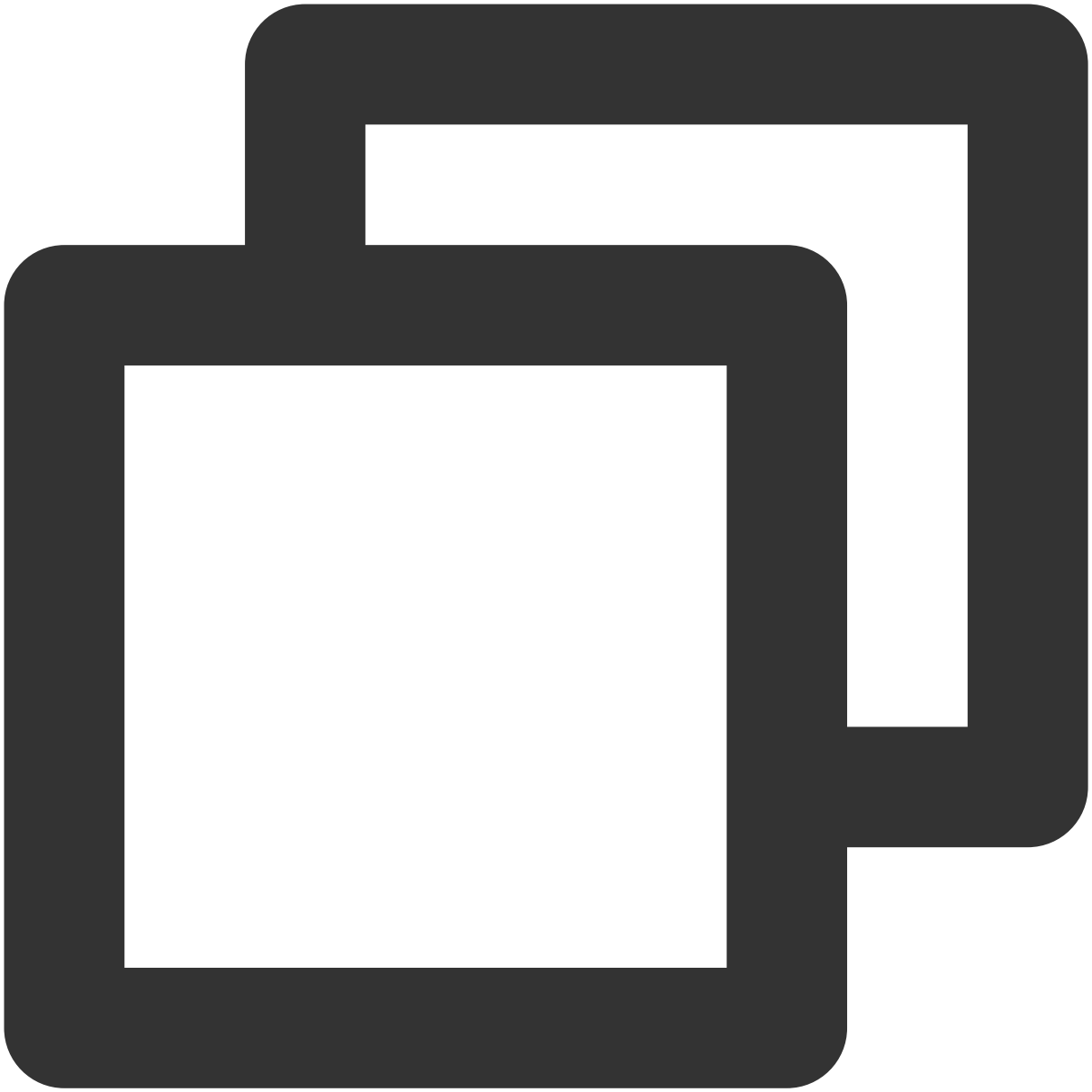
Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 MongoDB 连接串 > 部署 MongoDB Exporter > 验证 步骤完成 Exporter 部署。

使用 Secret 管理 MongoDB 连接串

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML创建资源**，创建 YAML 配置，配置说明如下：

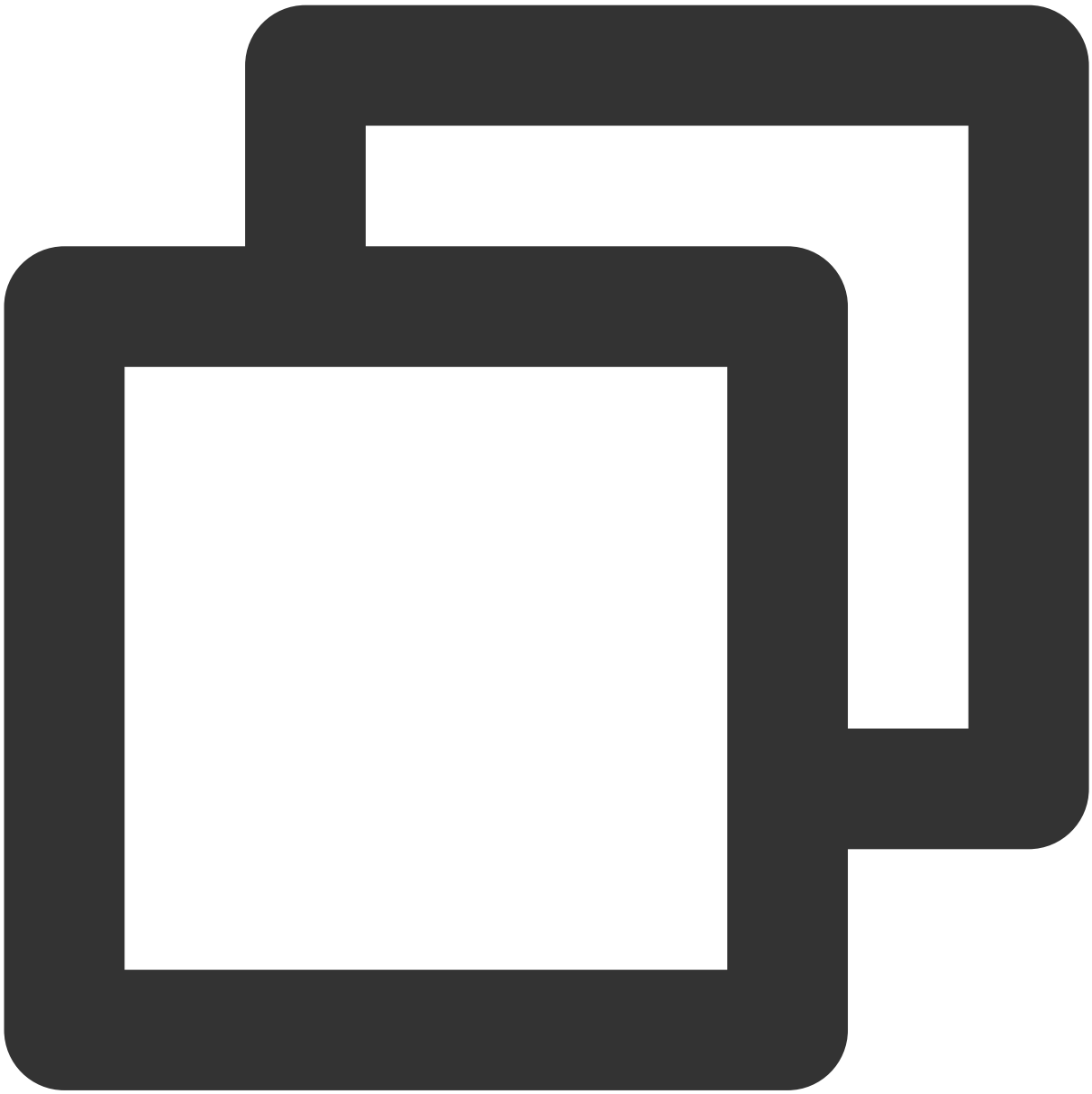
使用 Kubernetes 的 Secret 来管理密码并对密码进行加密处理，在启动 MongoDB Exporter 的时候直接使用 Secret Key，需要调整对应的 URI，YAML 配置示例如下：



```
apiVersion: v1
kind: Secret
metadata:
 name: mongodb-secret-test
 namespace: mongodb-test
type: Opaque
stringData:
 datasource: "mongodb://{user}:{passwd}@{host1}:{port1},{host2}:{port2},{host3}:
```

部署 MongoDB Exporter

在 Deployment 管理页面，单击**新建**，选择对应的**命名空间**来进行部署服务。可以通过控制台的方式创建，如下以 YAML 的方式部署 Exporter，YAML 配置示例如下：



```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: mongodb-exporter # 根据业务需要调整成对应的名称，建议加上 MongoDB 实例的信息
  name: mongodb-exporter # 根据业务需要调整成对应的名称，建议加上 MongoDB 实例的信息
  namespace: mongodb-test
```

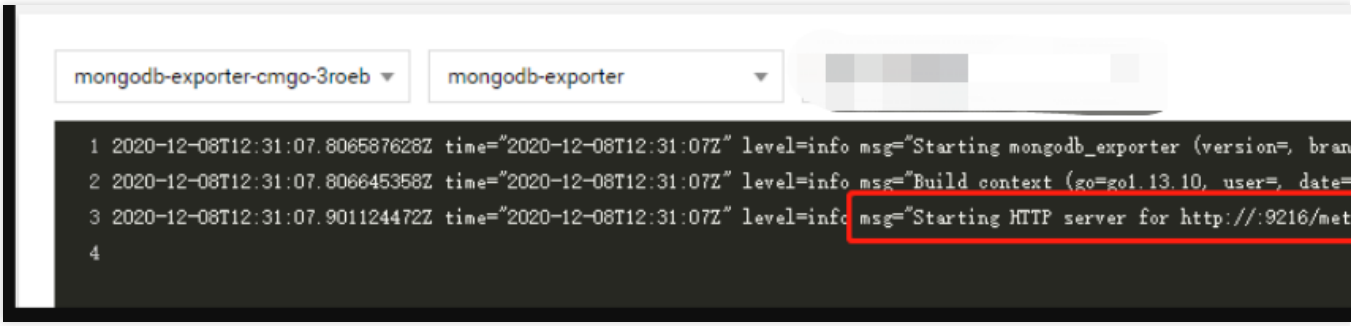
```
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: mongodb-exporter # 根据业务需要调整成对应的名称，建议加上 MongoDB 实例的信息
  template:
    metadata:
      labels:
        k8s-app: mongodb-exporter # 根据业务需要调整成对应的名称，建议加上 MongoDB 实例的信
    spec:
      containers:
        - args:
            - --collect.database       # 启用采集 Database metrics
            - --collect.collection     # 启用采集 Collection metrics
            - --collect.topmetrics     # 启用采集 table top metrics
            - --collect.indexusage     # 启用采集 per index usage stats
            - --collect.connpoolstats  # 启动采集 MongoDB connpoolstats
          env:
            - name: MONGODB_URI
              valueFrom:
                secretKeyRef:
                  name: mongodb-secret-test
                  key: datasource
          image: ssheehy/mongodb-exporter
          imagePullPolicy: IfNotPresent
          name: mongodb-exporter
          ports:
            - containerPort: 9216
              name: metric-port  # 这个名称在配置抓取任务的时候需要
          securityContext:
            privileged: false
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
        - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: { }
      terminationGracePeriodSeconds: 30
```

说明：

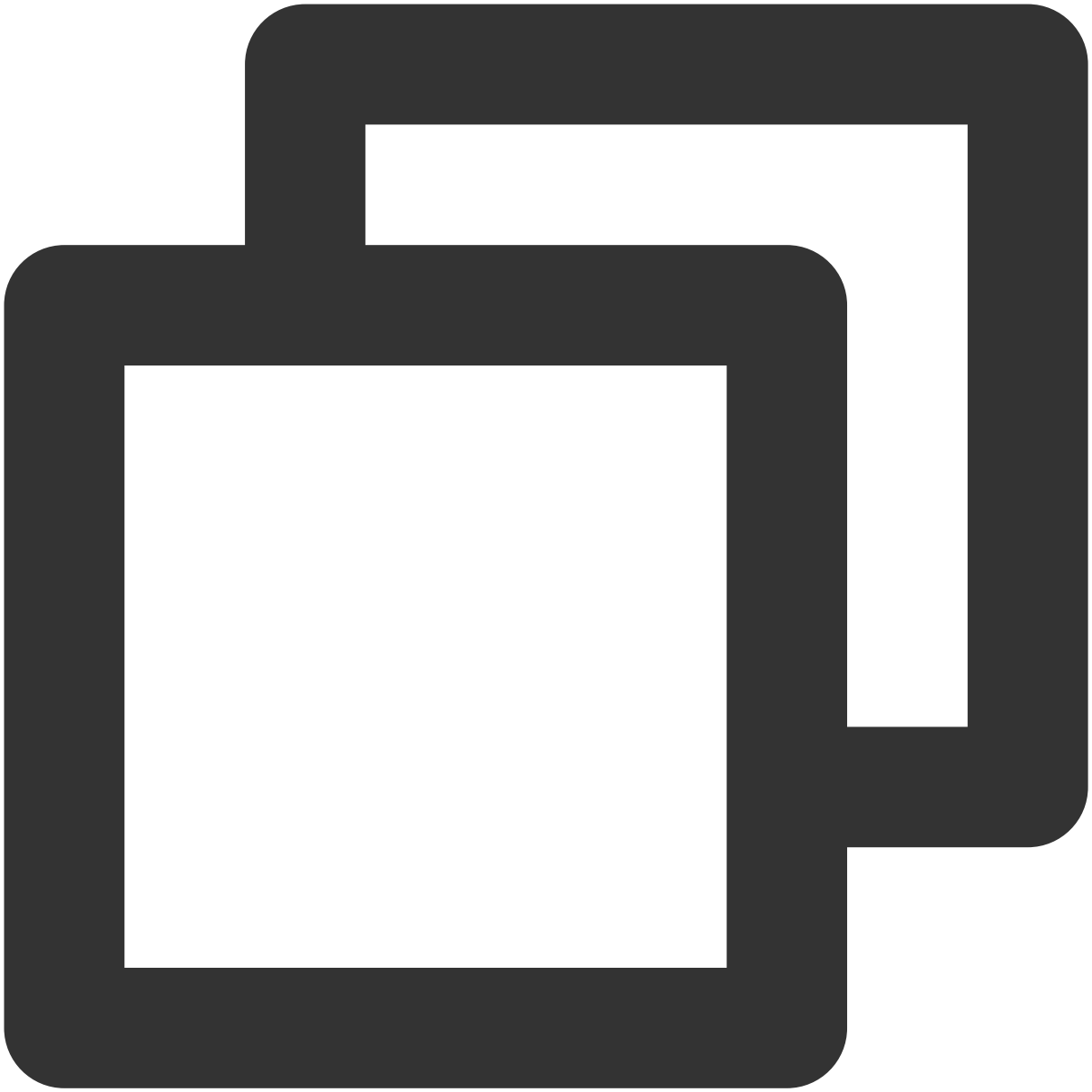
Exporter 详细参数请参见 mongodb_exporter。

验证

1. 在 Deployment 页面单击上述步骤创建的 Deployment，进入 Deployment 管理页面。
2. 单击**日志**页签，可以查看到 Exporter 成功启动并暴露对应的访问地址，如下图所示：

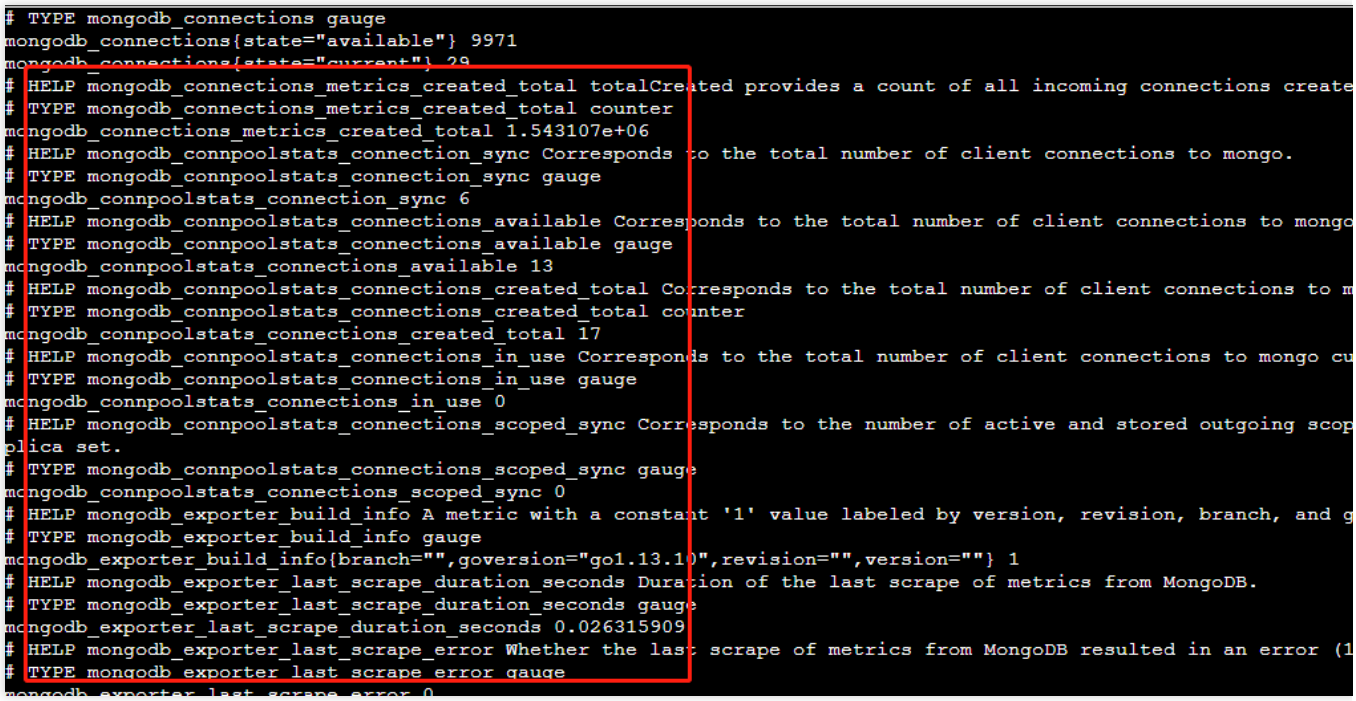


3. 单击 **Pod 管理**页签，进入 Pod 页面。
4. 在右侧的操作项下单击**远程登录**登录 Pod，在命令行中执行以下 wget 命令对应 Exporter 暴露的地址，可以正常得到对应的 MongoDB 指标，若发现未能得到对应的数据，请检查一下连接 URI 是否正确，具体如下：



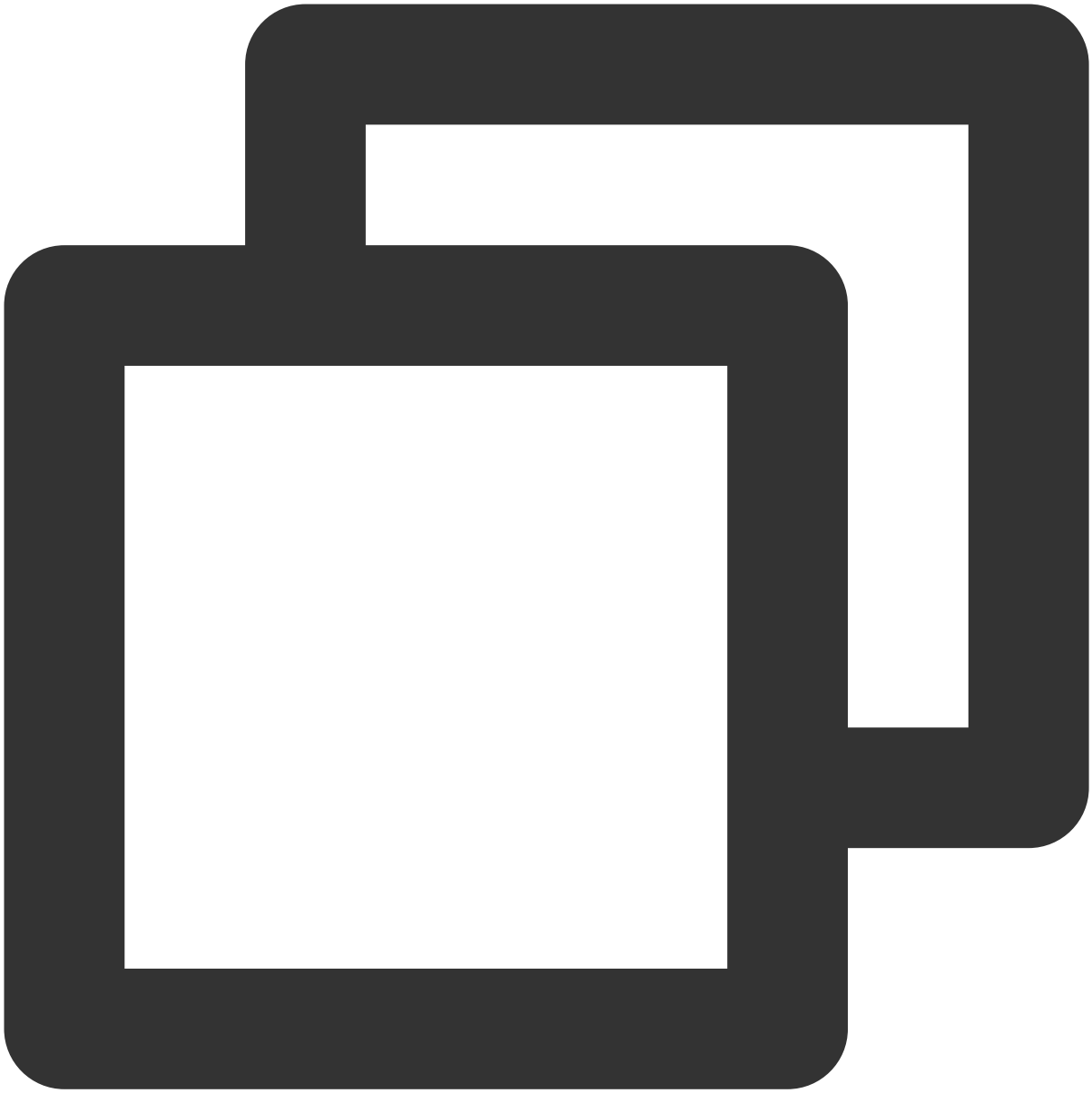
```
wget 127.0.0.1:9216/metrics
cat metrics
```

命令执行结果如下图所示：



添加采集任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: mongodb-exporter # 填写一个唯一名称
  namespace: cm-prometheus   # namespace固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port    # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics  # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
```

```
      - action: replace
        sourceLabels:
        - instance
        regex: (.*)
        targetLabel: instance
        replacement: 'cmgo-xxxxxxxx' # 调整成对应的 MongoDB 实例 ID
    namespaceSelector:  # 选择要监控pod所在的namespace
      matchNames:
      - mongodb-test
    selector: # 填写要监控pod的Label值，以定位目标pod
      matchLabels:
        k8s-app: mongodb-exporter
```

说明：

由于 `Exporter` 和 `MongoDB` 部署在不同的服务器上，因此建议通过 Prometheus Relabel 机制将 MongoDB 实例的信息放到监控指标中，以便定位问题。

查看监控

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击**集成中心**，进入集成中心页面。找到 `MongoDB` 监控，安装对应的 Grafana Dashboard 即可开启 `MongoDB` 监控大盘，查看实例相关的监控数据，如下图所示：

MongoDB 概览：以实例的纬度查看实例状态，例如文档个数、连接使用率、读写耗时等，可单击实例跳转到该实例详情。

MongoDB 详情：可以查看某个实例的详细状态，例如元数据概览、核心指标、命令操作、请求流量、读写 Top 等。

说明：

每个图表可以单击左侧的!进行查看说明。

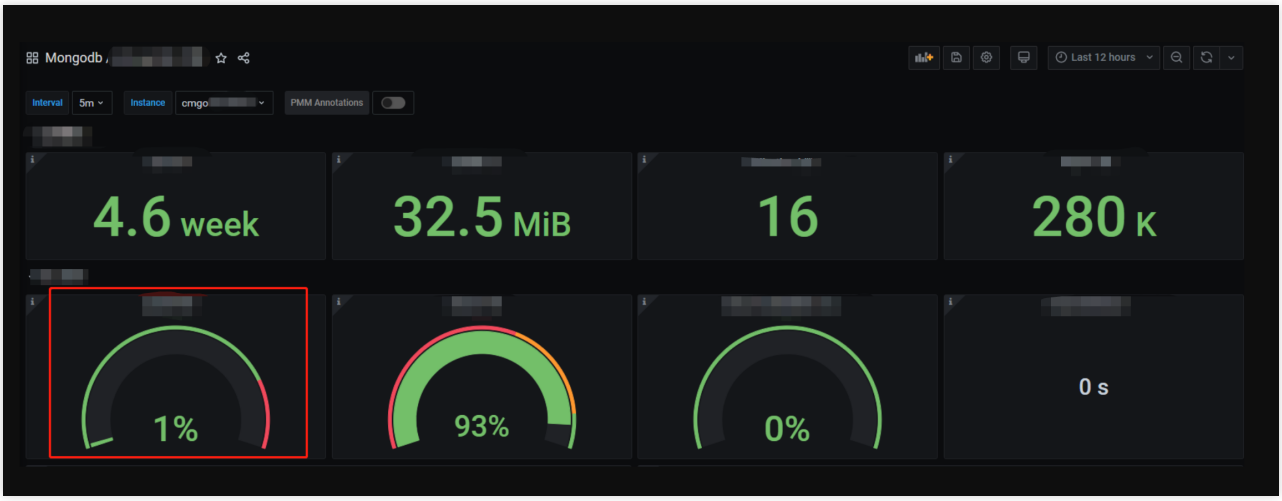
告警以及接入

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击告警策略，可以添加相应的告警策略，详情请参见 新建告警策略。

常见问题

客户端报错：client checkout connect timeout，该如何处理？

可能是连接池使用率达到100%，导致创建连接失败。可以通过 Grafana 大盘 **MongoDB 详情/核心指标/连接使用率** 指标排查。



写入不断超时，该如何处理？

需检查 Cache 使用率是否过高、Transactions 可用个数是否为0，可以通过 Grafana 大盘 **MongoDB 详情/核心指标/ WiredTiger Transactions 可用个数| WiredTiger Cache 使用率| GetLastError 写耗时| GetLastError 写超时**指标排查。

PostgreSQL Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用 PostgreSQL 过程中都需要对 PostgreSQL 运行状态进行监控，以便了解 PostgreSQL 服务是否运行正常，排查 PostgreSQL 故障问题原因， Prometheus 监控服务提供了基于 Exporter 的方式来监控 PostgreSQL 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文介绍如何部署 Exporter 以及实现 PostgreSQL Exporter 告警接入等操作。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 Kubernetes 集群。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见 Agent 管理。

操作步骤

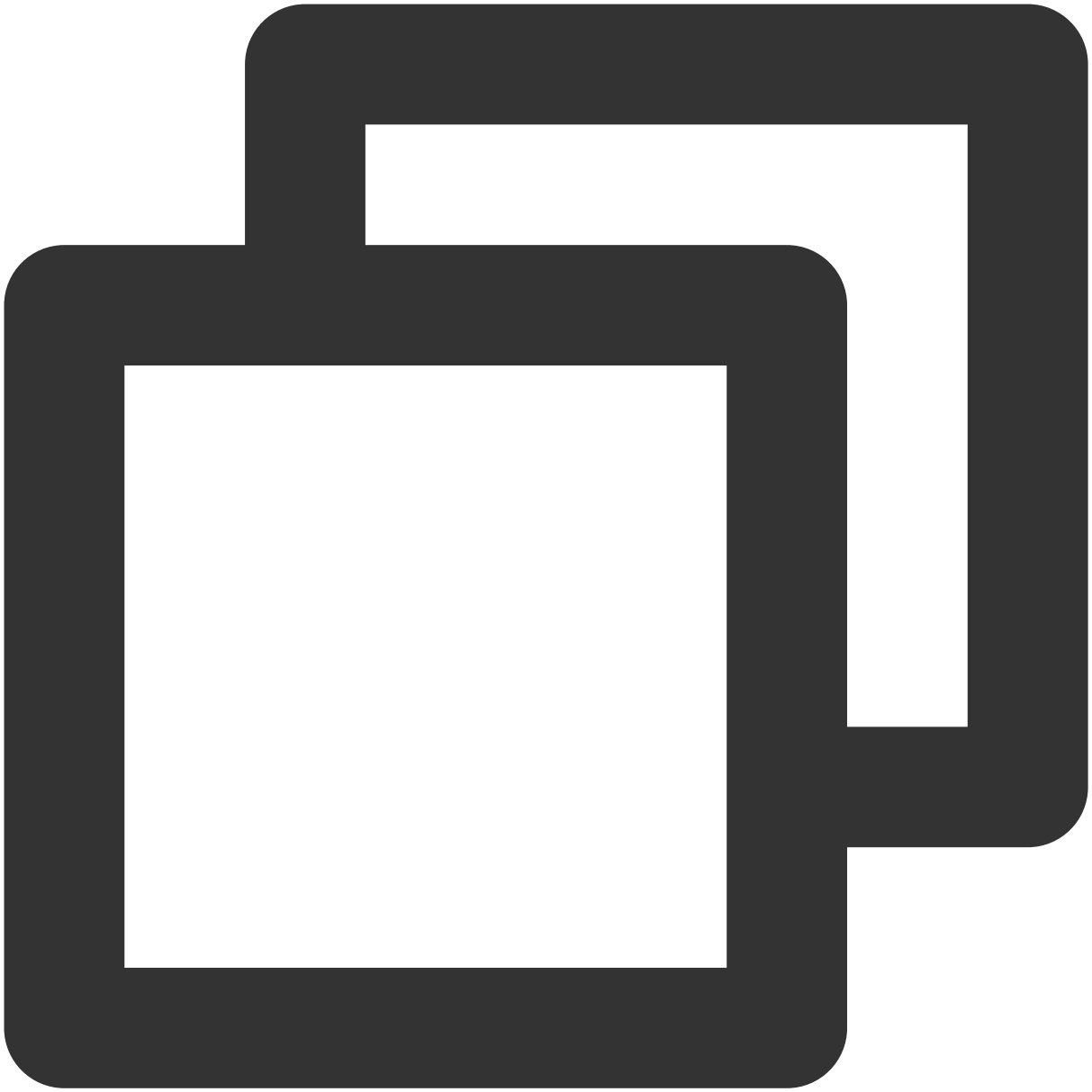
Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 PostgreSQL 密码 > 部署 PostgreSQL Exporter > 获取指标 步骤完成 Exporter 部署。

使用 Secret 管理 PostgreSQL 密码

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML 创建资源**，创建 YAML 配置，配置说明如下：

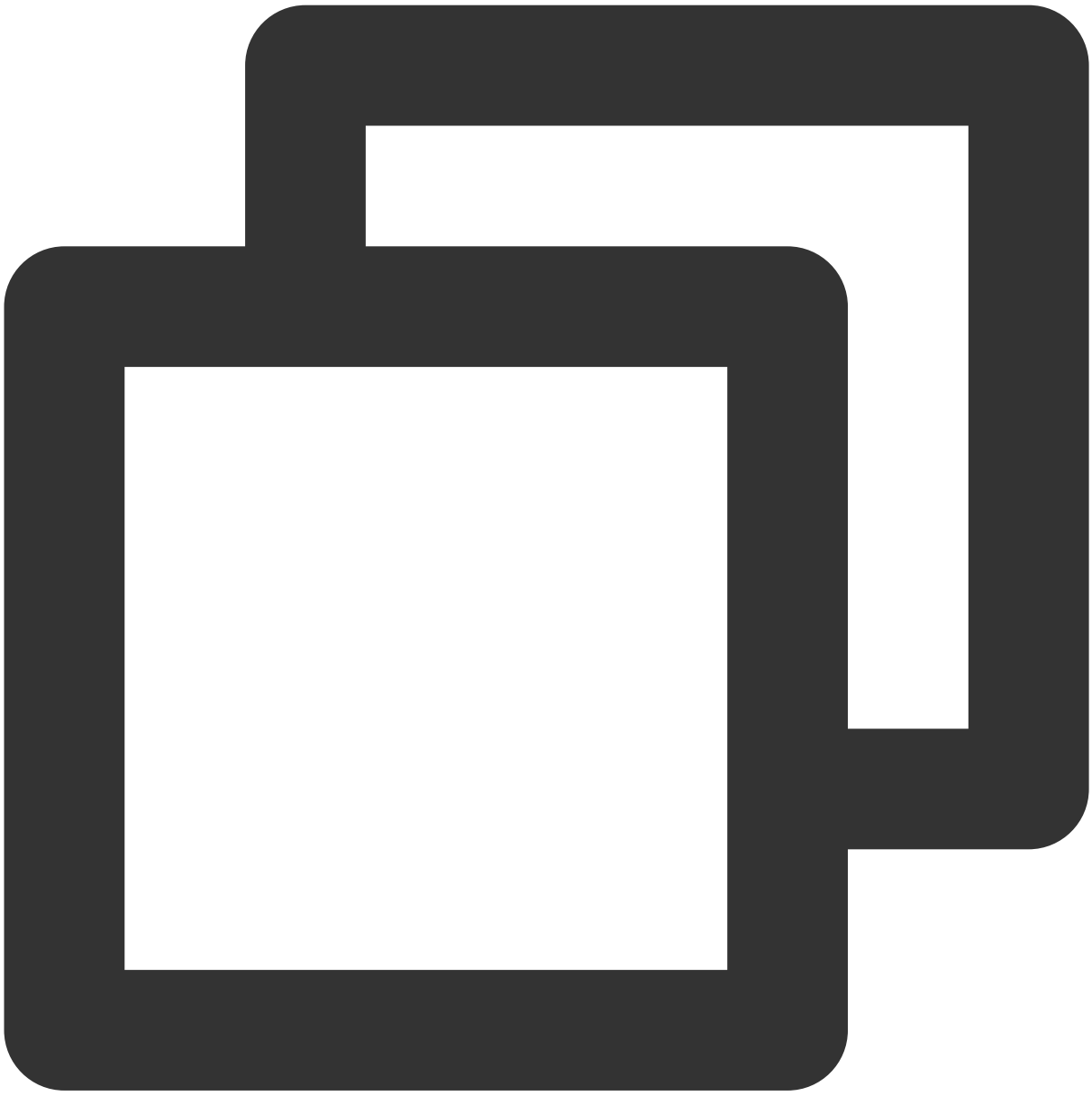
使用 Kubernetes 的 Secret 来管理密码并对密码进行加密处理，在启动 PostgreSQL Exporter 的时候直接使用 Secret Key，需要调整对应的 `password` ，YAML 配置示例如下：



```
apiVersion: v1
kind: Secret
metadata:
 name: postgres-test
type: Opaque
stringData:
 username: postgres
 password: you-guess #对应 PostgreSQL 密码
```

部署 PostgreSQL Exporter

在 Deployment 管理页面，单击**新建**，选择对应的**命名空间**来进行部署服务。可以通过控制台的方式创建，如下以 YAML 的方式部署 Exporter，YAML 配置示例如下（ 请直接复制下面的内容，根据实际业务调整相应的参数 ）：



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-test
  namespace: postgres-test
  labels:
    app: postgres
```

```
    app.kubernetes.io/name: postgresql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
      app.kubernetes.io/name: postgresql
  template:
    metadata:
      labels:
        app: postgres
        app.kubernetes.io/name: postgresql
    spec:
      containers:
      - name: postgres-exporter
        image: wrouesnel/postgres_exporter:latest
        args:
          - "--web.listen-address=:9187"
          - "--log.level=debug"
        env:
          - name: DATA_SOURCE_USER
            valueFrom:
              secretKeyRef:
                name: postgres-test
                key: username
          - name: DATA_SOURCE_PASS
            valueFrom:
              secretKeyRef:
                name: postgres-test
                key: password
          - name: DATA_SOURCE_URI
            value: "x.x.x.x:5432/postgres?sslmode=disable"
        ports:
        - name: http-metrics
          containerPort: 9187
```

说明：

上述示例将 Secret 中的用户名密码传给了环境变量 `DATA_SOURCE_USER` 和 `DATA_SOURCE_PASS`，使用户无法查看到明文的用户名密码。您还可以用 `DATA_SOURCE_USER_FILE` / `DATA_SOURCE_PASS_FILE` 从文件读取用户名密码。或使用 `DATA_SOURCE_NAME` 将用户名密码也放在连接串里，例如 `postgresql://login:password@hostname:port/dbname` 。

参数说明

`DATA_SOURCE_URI` / `DATA_SOURCE_NAME` 连接串 query 部分（`?` 之后）支持的参数如下（最新的以 GoDoc 为准）：

参数	参数说明
sslmode	是否使用 SSL，支持的值如下：
- disable	不使用 SSL
- require	总是使用（跳过验证）
- verify-ca	总是使用（检查服务端提供的证书是不是由一个可信的 CA 签发）
- verify-full	总是使用（检查服务端提供的证书是不是由一个可信的 CA 签发，并且检查 hostname 是不是被证书所匹配）
fallback_application_name	一个备选的 application_name
connect_timeout	最大连接等待时间，单位秒。0 值等于无限大
sslcert	证书文件路径。文件数据格式必须是 PEM
SSHKey	私钥文件路径。文件数据格式必须是 PEM
sslrootcert	root 证书文件路径。文件数据格式必须是 PEM

另外 Exporter 支持其他参数，如下说明（详情请参见 README）：

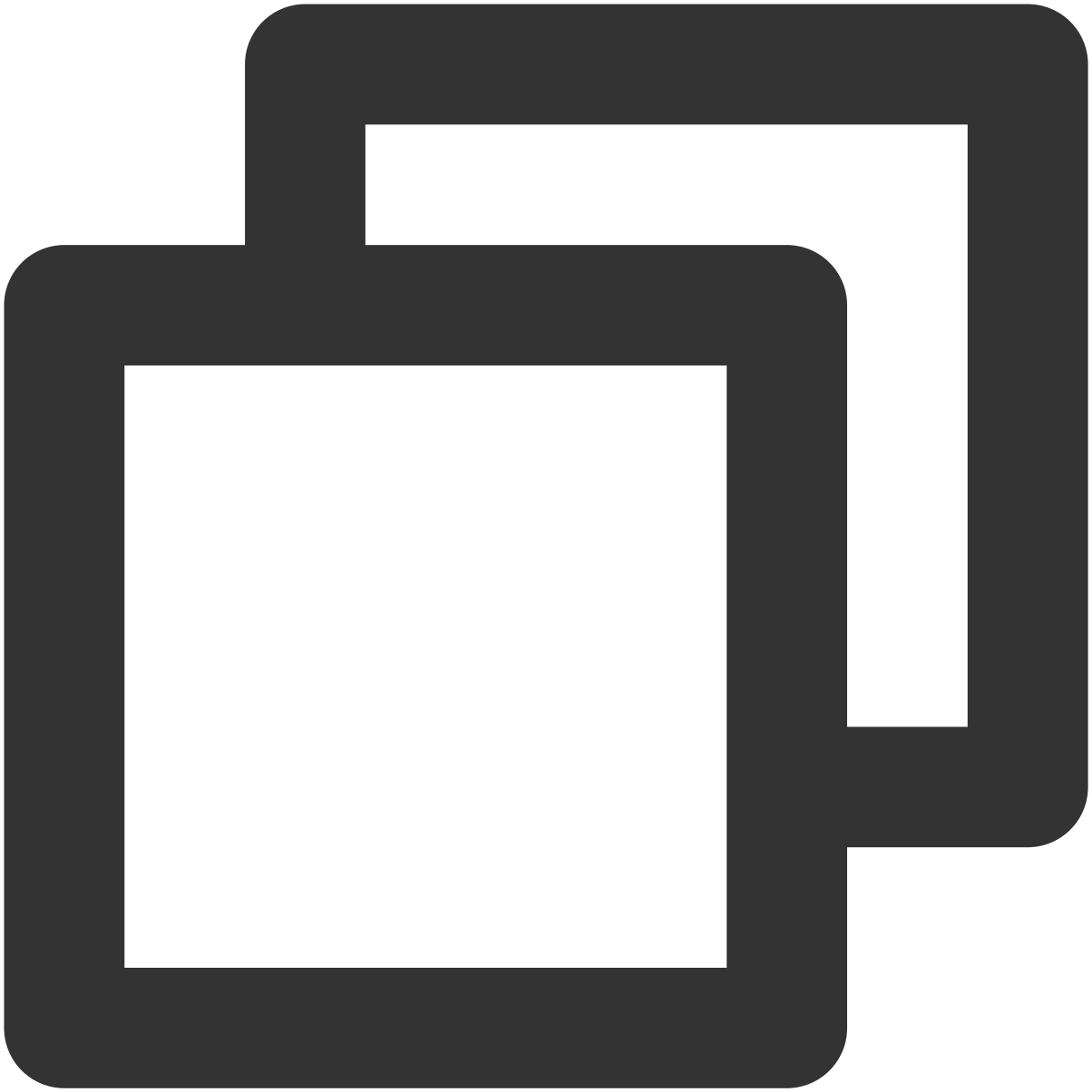
参数	参数说明	环境变量
--web.listen-address	监听地址，默认 :9487	PG_EXPORTER_WEB_LISTEN_ADDRESS
--web.telemetry-path	暴露指标的路径，默认 /metrics	PG_EXPORTER_WEB_TELEMETRY_PATH
--extend.query-path	指定一个包含自定义查询语句的 YAML 文件，参考 queries.yaml。	PG_EXPORTER_EXTEND_QUERY_PATH
--disable-default-metrics	只使用通过 queries.yaml 提供的指标	PG_EXPORTER_DISABLE_DEFAULT_METRICS
--disable-settings-metrics	不抓取 pg_settings 相关的指标	PG_EXPORTER_DISABLE_SETTINGS_METRICS
--auto-	是否自动发现 Postgres 实例上的	PG_EXPORTER_AUTO_DISCOVER_DATABASES

discover-databases	数据库	
--dumpmaps	打印内部的指标信息，除了 debug 不要使用，方便排查自定义 queries 相关的问题	-
--constantLabels	自定义标签，通过 key=value 的形式提供，多个标签对使用 , 分隔	PG_EXPORTER_CONSTANT_LABELS
--exclude-databases	需要排除的数据库，仅在 --auto-discover-databases 开启的情况下有效	PG_EXPORTER_EXCLUDE_DATABASES
--log.level	日志级别 debug/info/warn/error/fatal	PG_EXPORTER_LOG_LEVEL

获取指标

通过 `curl http://exporter:9187/metrics` 无法获取 `Postgres` 实例运行时间。我们可以通过自定义一个 `queries.yaml` 来获取该指标：

1. 创建一个包含 `queries.yaml` 的 ConfigMap。
2. 将 ConfigMap 作为 Volume 挂载到 Exporter 某个目录下面。
3. 通过 `--extend.query-path` 来使用 ConfigMap，将上述的 Secret 以及 Deployment 进行汇总，汇总后的 YAML 如下所示：

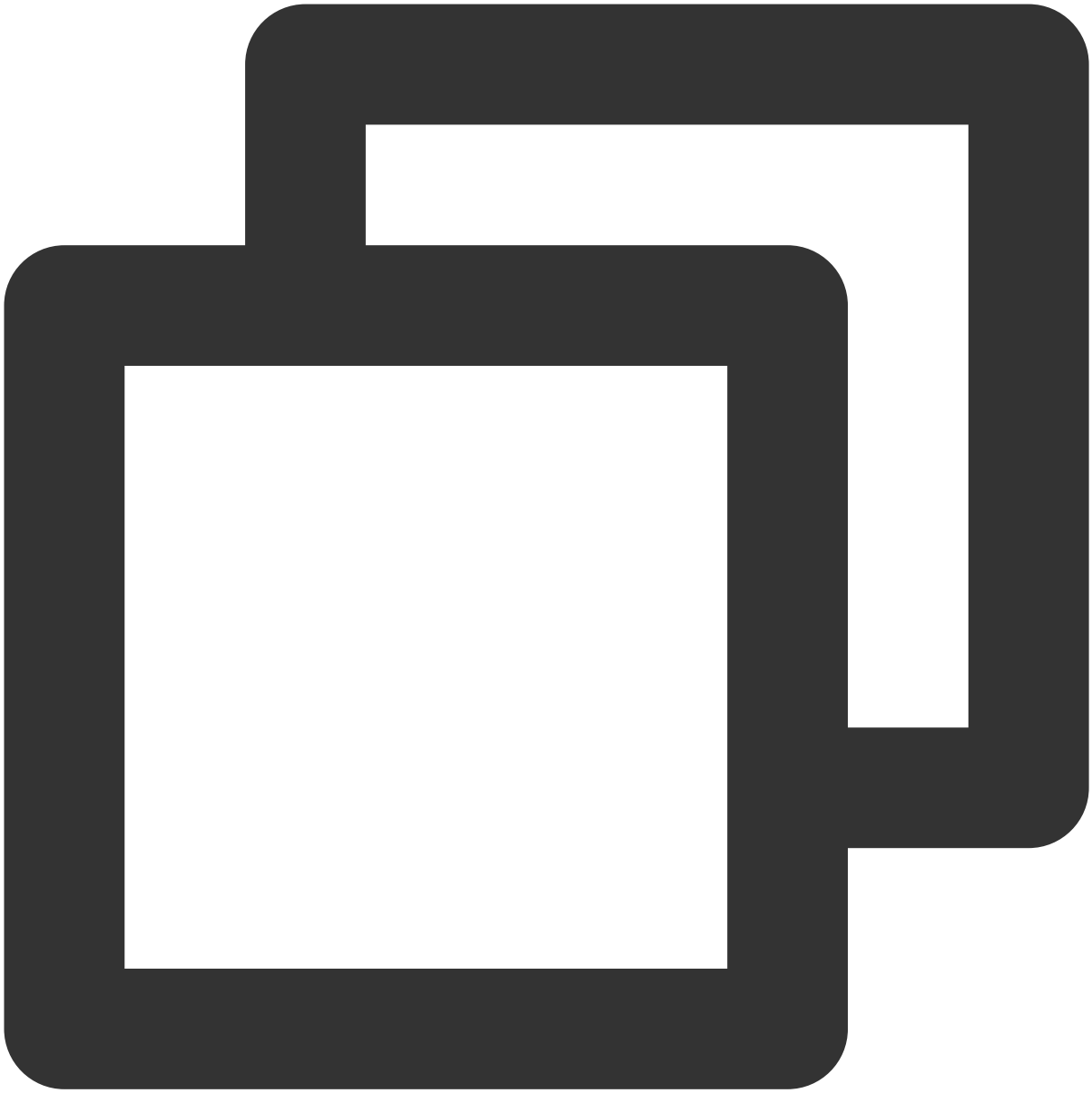


```
# 注意：以下 document 创建一个名为 postgres-test 的 Namespace，仅作参考
apiVersion: v1
kind: Namespace
metadata:
  name: postgres-test

# 以下 document 创建一个包含用户名密码的 Secret
---
apiVersion: v1
kind: Secret
metadata:
```

```
  name: postgres-test-secret
  namespace: postgres-test
type: Opaque
stringData:
  username: postgres
  password: you-guess

# 以下 document 创建一个包含自定义指标的 queries.yaml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-test-configmap
  namespace: postgres-test
data:
  queries.yaml: |
    pg_postmaster:
      query: "SELECT pg_postmaster_start_time as start_time_seconds from pg_postmas
      master: true
      metrics:
        - start_time_seconds:
            usage: "GAUGE"
            description: "Time at which postmaster started"

# 以下 document 挂载了 Secret 和 ConfigMap ，定义了部署 Exporter 相关的镜像等参数
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-test
  namespace: postgres-test
  labels:
    app: postgres
    app.kubernetes.io/name: postgresql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
      app.kubernetes.io/name: postgresql
  template:
    metadata:
      labels:
        app: postgres
        app.kubernetes.io/name: postgresql
    spec:
      containers:
```

```
        - name: postgres-exporter
          image: wrouesnel/postgres_exporter:latest
          args:
            - "--web.listen-address=:9187"
            - "--extend.query-path=/etc/config/queries.yaml"
            - "--log.level=debug"
          env:
            - name: DATA_SOURCE_USER
              valueFrom:
                secretKeyRef:
                  name: postgres-test-secret
                  key: username
            - name: DATA_SOURCE_PASS
              valueFrom:
                secretKeyRef:
                  name: postgres-test-secret
                  key: password
            - name: DATA_SOURCE_URI
              value: "x.x.x.x:5432/postgres?sslmode=disable"
          ports:
            - name: http-metrics
              containerPort: 9187
          volumeMounts:
            - name: config-volume
              mountPath: /etc/config
      volumes:
        - name: config-volume
          configMap:
            name: postgres-test-configmap
```

4. 执行 `curl http://exporter:9187/metrics`，即可通过自定义的 `queries.yaml` 查询到 Postgres 实例启动时间指标。示例如下：



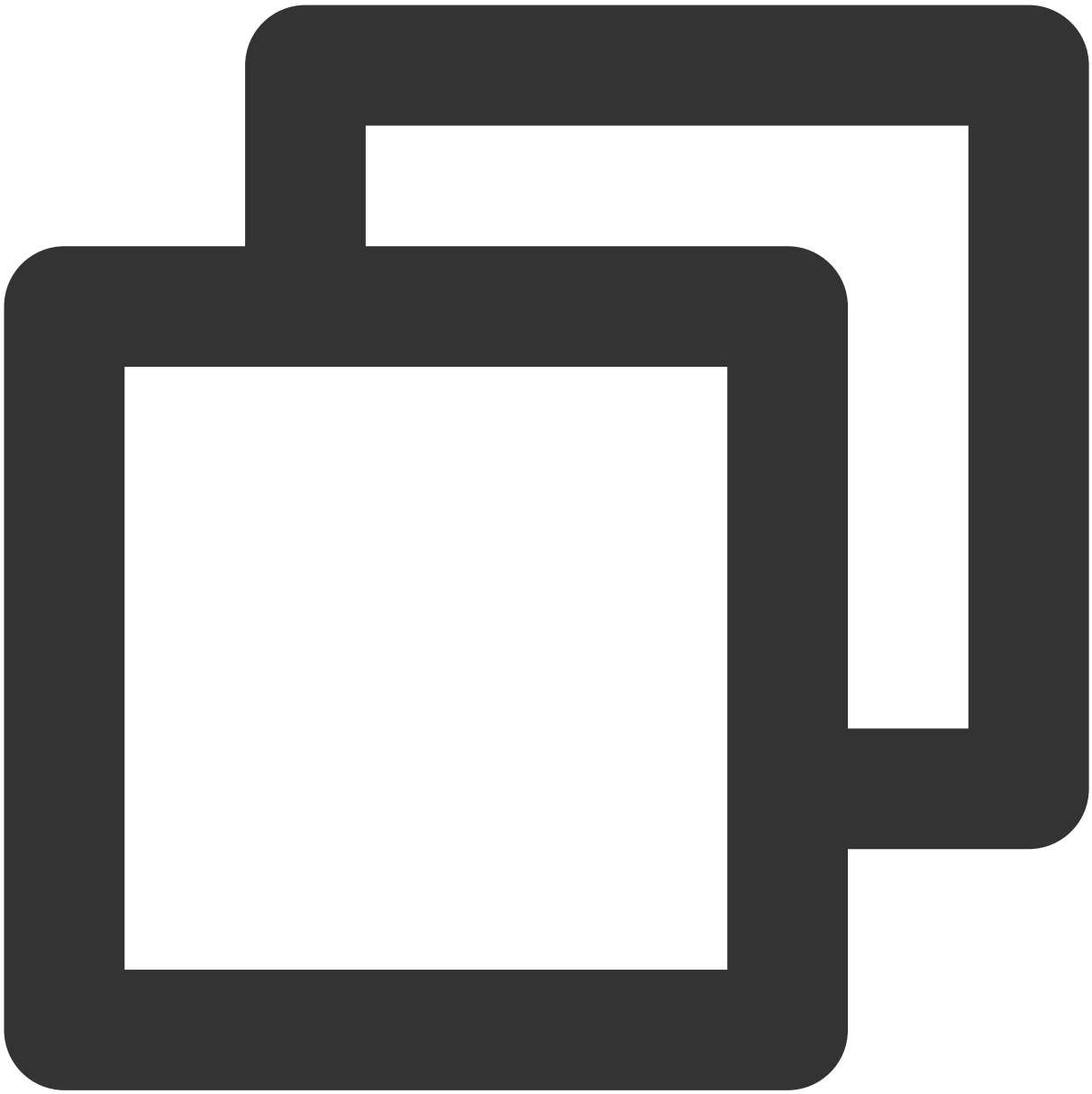
```
# HELP pg_postmaster_start_time_seconds Time at which postmaster started
# TYPE pg_postmaster_start_time_seconds gauge
pg_postmaster_start_time_seconds{server="x.x.x.x:5432"} 1.605061592e+09
```

添加采取任务

当 Exporter 运行起来之后，需要进行以下操作配置 Prometheus 监控服务发现并采集监控指标：

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。

3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: postgres-exporter
  namespace: cm-prometheus
spec:
  namespaceSelector:
    matchNames:
    - postgres-test
```

```
    podMetricsEndpoints:
    - interval: 30s
      path: /metrics
      port: http-metrics # 前面 Exporter 那个 Container 的端口名
      relabelings:
      - action: labeldrop
        regex: __meta_kubernetes_pod_label_(pod_|statefulset_|deployment_|controlle
      - action: replace
        regex: (.*)
        replacement: postgres-xxxxxx
        sourceLabels:
        - instance
        targetLabel: instance
    selector:
      matchLabels:
        app: postgres
```

说明：

更多高阶用法请参见 ServiceMonitor 和 PodMonitor。

Grafana 大屏可视化

说明：

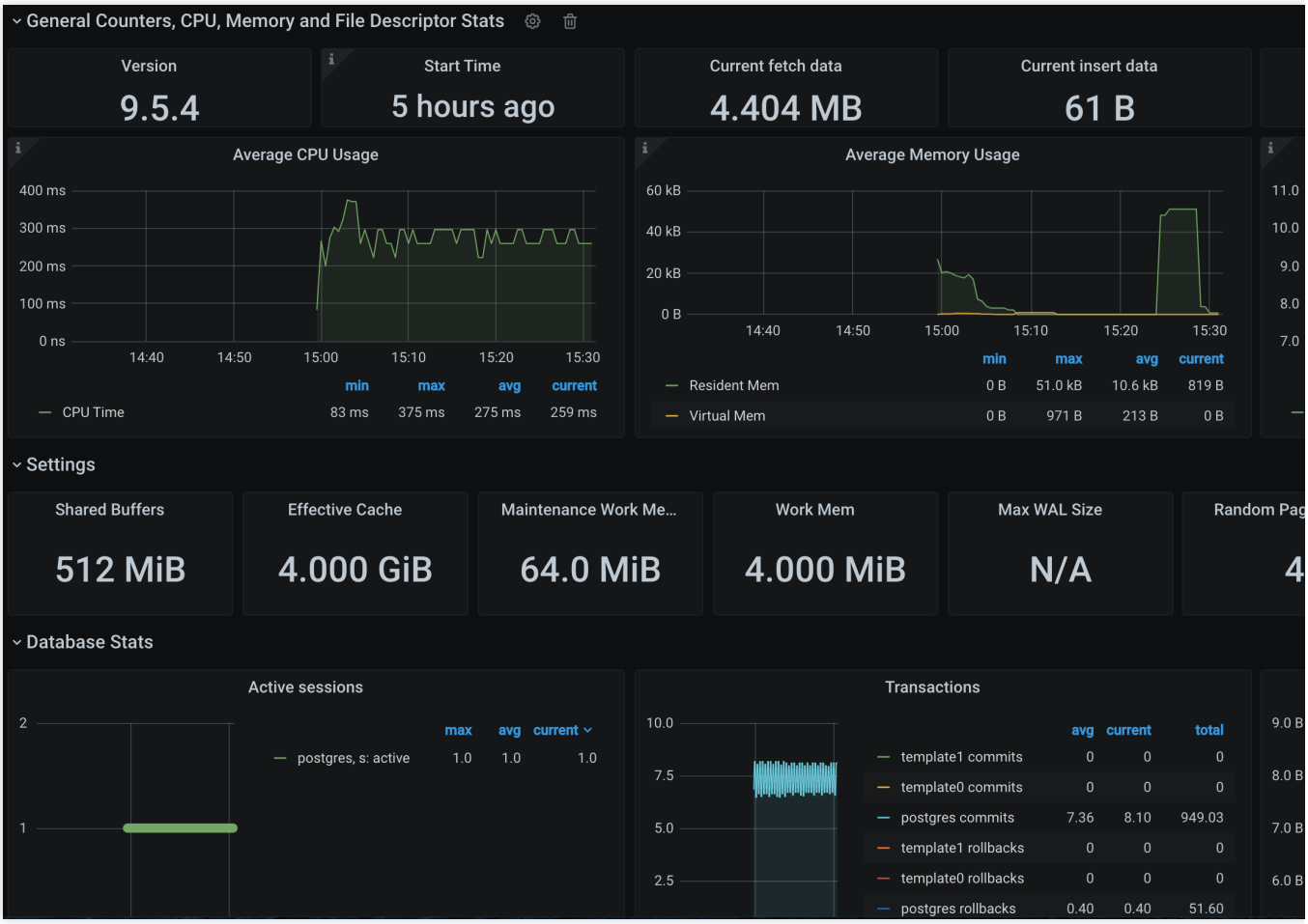
需要使用上述 获取指标 配置来获取 Postgres 实例的启动时间。

1. 在 Prometheus 实例 列表，找到对应的 Prometheus 实例，单击 实例ID 右侧

图标，打开您的专属 Grafana，输入您的账号密码，即可进行 Grafana 可视化大屏操作区。

2. 进入 Grafana，单击

图表，展开监控面板，单击对应的监控图表名称即可查看监控数据。



告警以及接入

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击告警策略，可以添加相应的告警策略，详情请参见 新建告警策略。

说明：

后续 Prometheus 监控服务将提供更多 PostgreSQL 相关的告警模板。

Nginx Exporter 接入

最近更新时间：2024-01-29 15:55:14

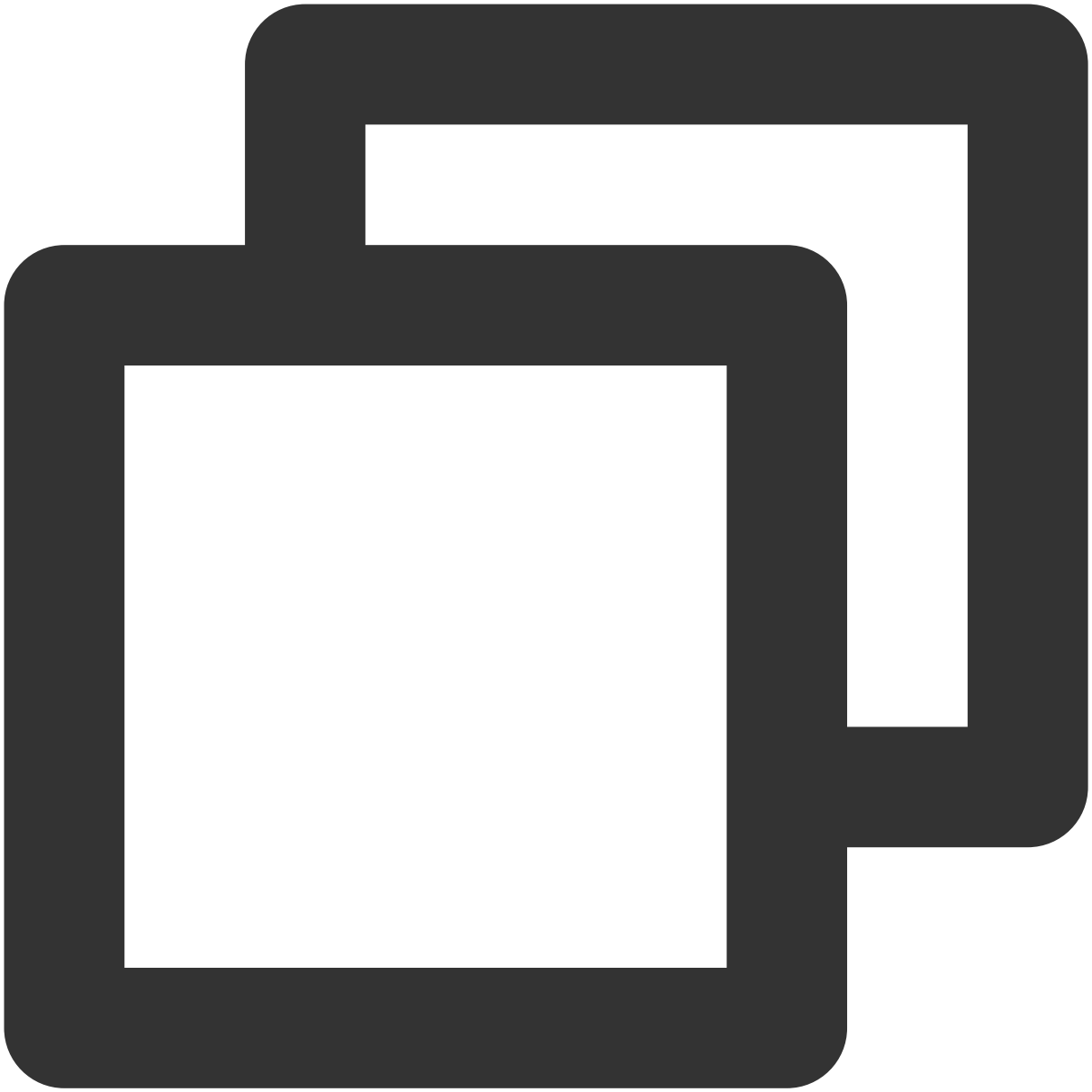
操作场景

Nginx 通过 stub_status 页面暴露了部分监控指标。Nginx Prometheus Exporter 会采集单个 Nginx 实例指标，并将其转化为 Prometheus 可用的监控数据， 最终通过 HTTP 协议暴露给 Prometheus 服务进行采集。我们可以通过 Exporter 上报重点关注的监控指标，用于异常报警和大盘展示。

操作步骤

使用 Docker 容器运行 Exporter

方式1：使用 nginx-prometheus-exporter 通过 Docker 容器快速部署 Exporter。执行 Docker 命令如下：



```
$ docker run -p 9113:9113 nginx/nginx-prometheus-exporter:0.8.0 -nginx.scrape-uri h
```

方式2：使用 nginx-prometheus-exporter 镜像将服务部署在腾讯云 容器服务 TKE 中，通过托管 Prometheus 的监控自发现 CRD PodMonitor 或者 ServiceMonitor 来采集监控数据。

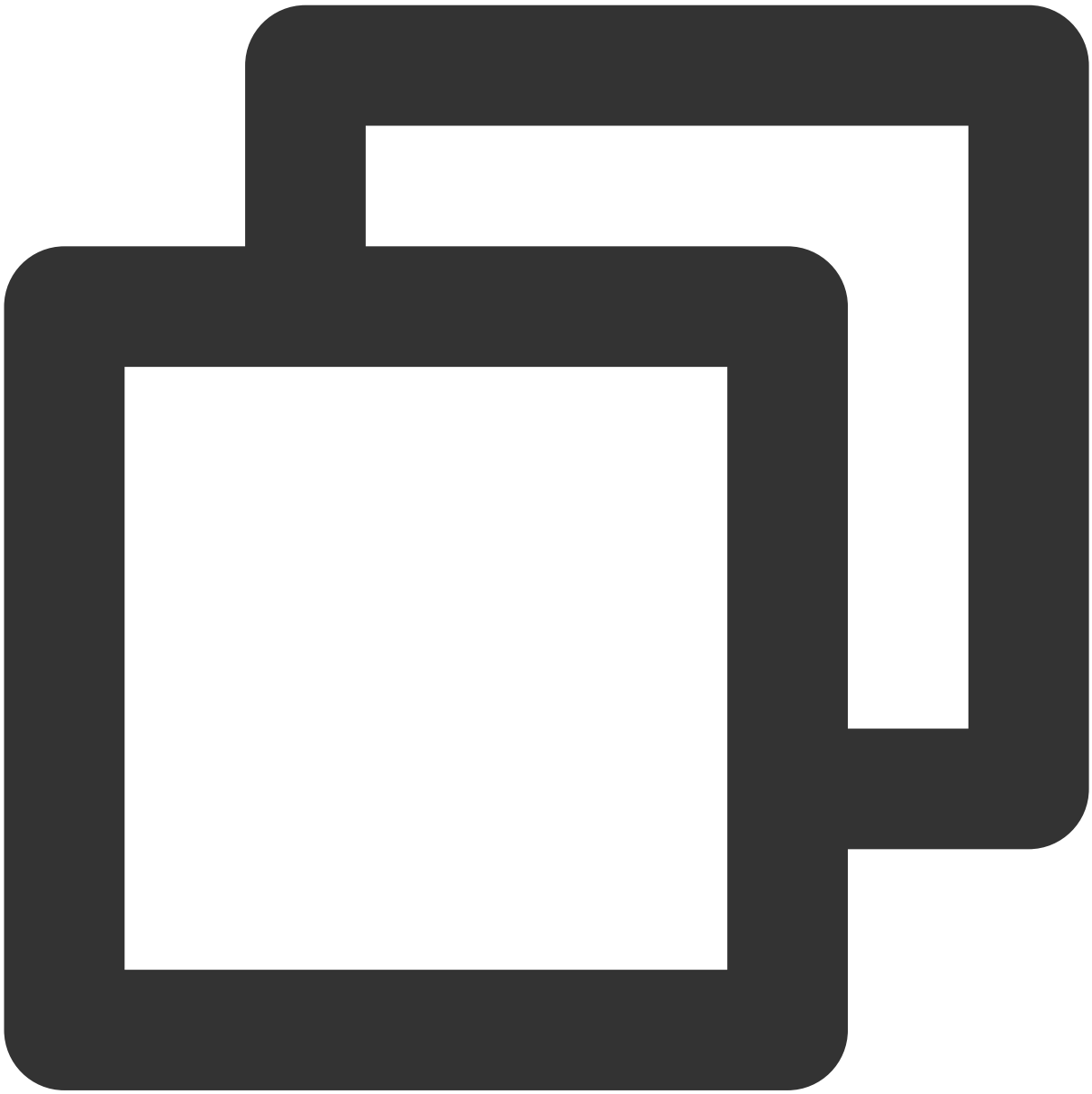
使用二进制程序运行 Exporter

下载安装

1. 根据实际运行环境在社区中下载相应的 Nginx Prometheus Exporter 。
2. 安装 Nginx Prometheus Exporter。

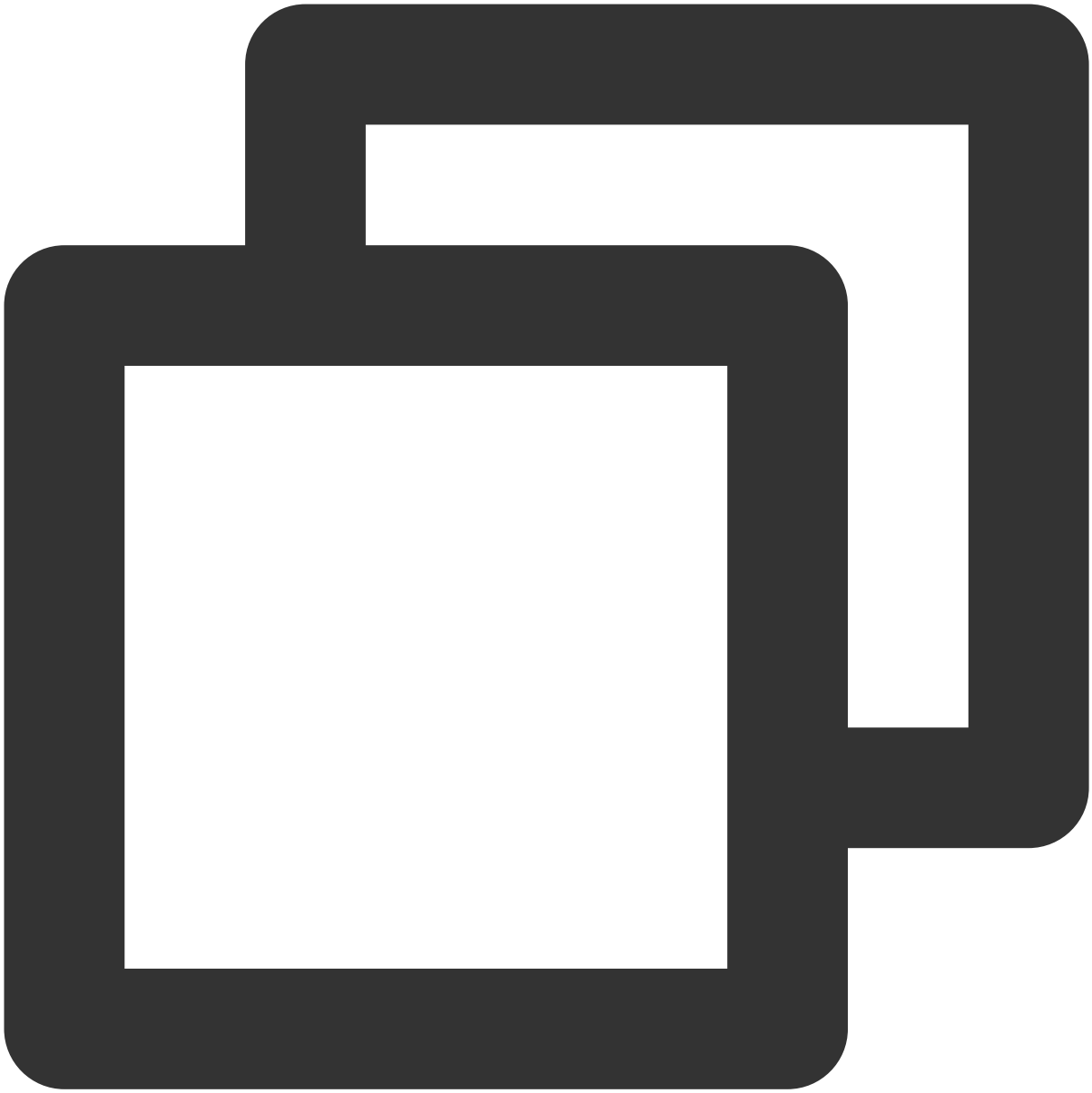
开启 NGINX stub_status 功能

1. 开源 Nginx 提供一个简单页面用于展示状态数据，该页面由 tub_status 模块提供。执行以下命令检查 Nginx 是否已经开启了该模块：



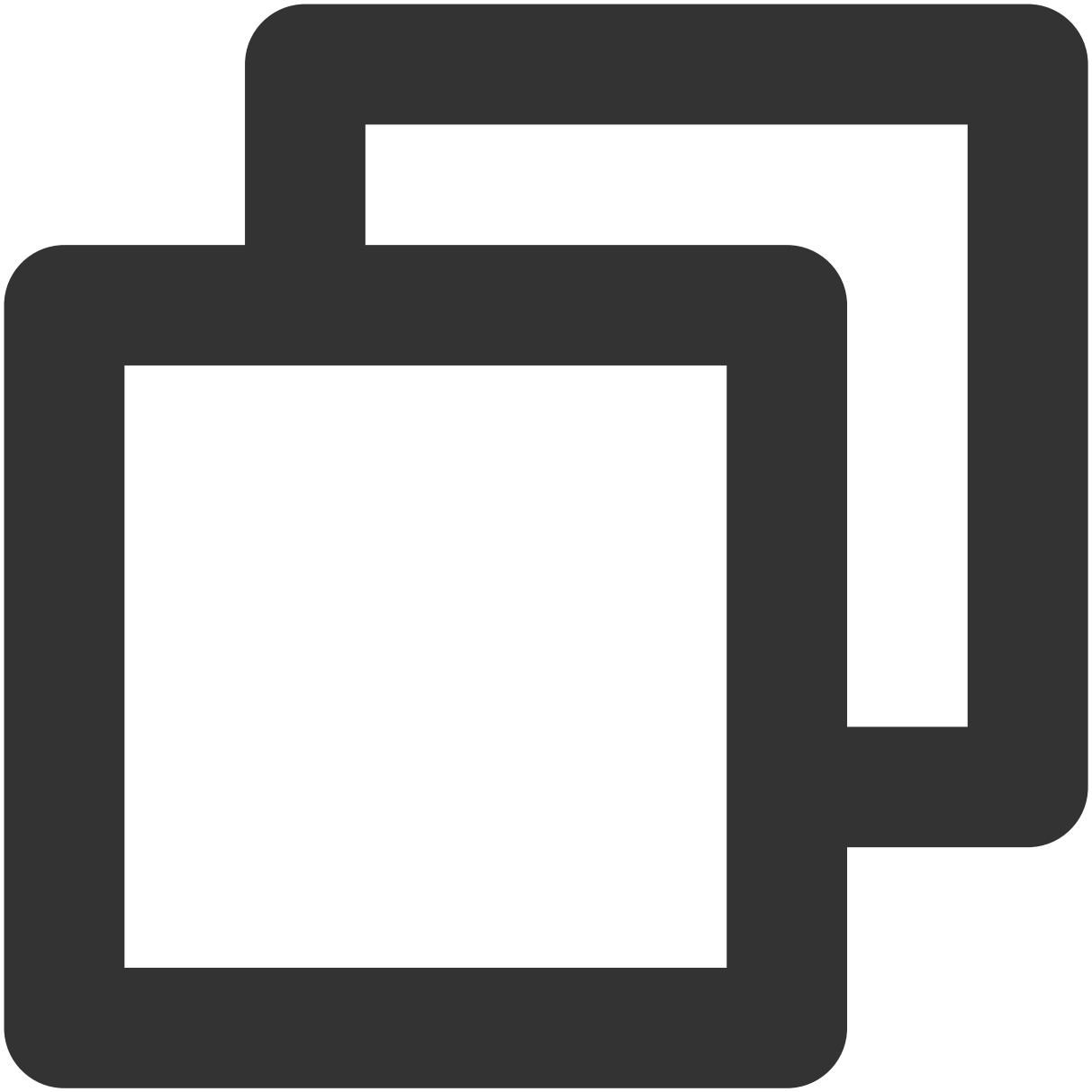
```
nginx -V 2>&1 | grep -o with-http_stub_status_module
```

如果在终端中输出 `with-http_stub_status_module` ，则说明 Nginx 已启用 tub_status 模块。
如果未输出任何结果，则可以使用 `--with-http_stub_status_module` 参数从源码重新配置编译一个 Nginx。示例如下：



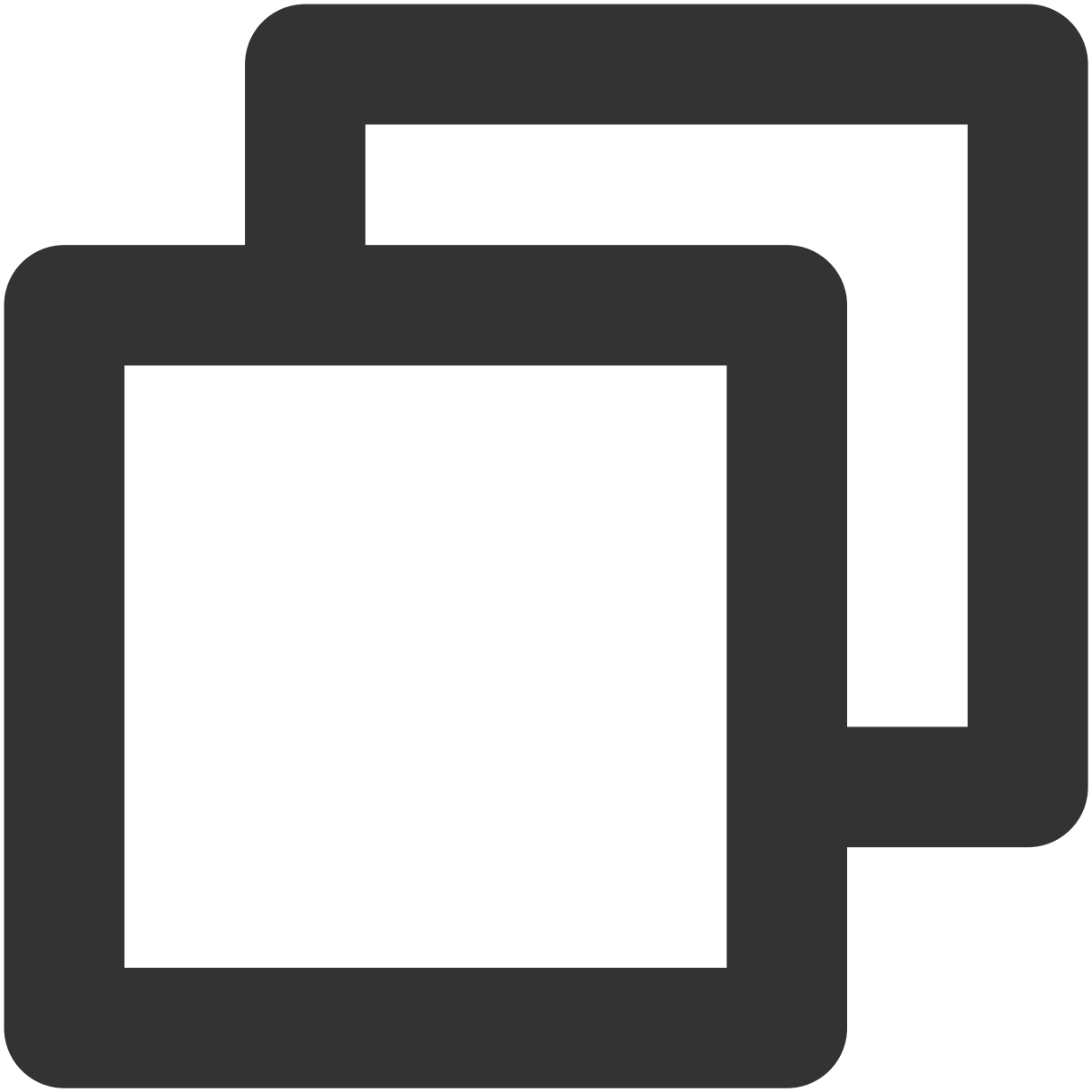
```
./configure \\
… \\--with-http_stub_status_module
make
sudo make install
```

2. 确认 stub_status 模块启用之后，修改 Nginx 的配置文件指定 status 页面的 URL。示例如下：



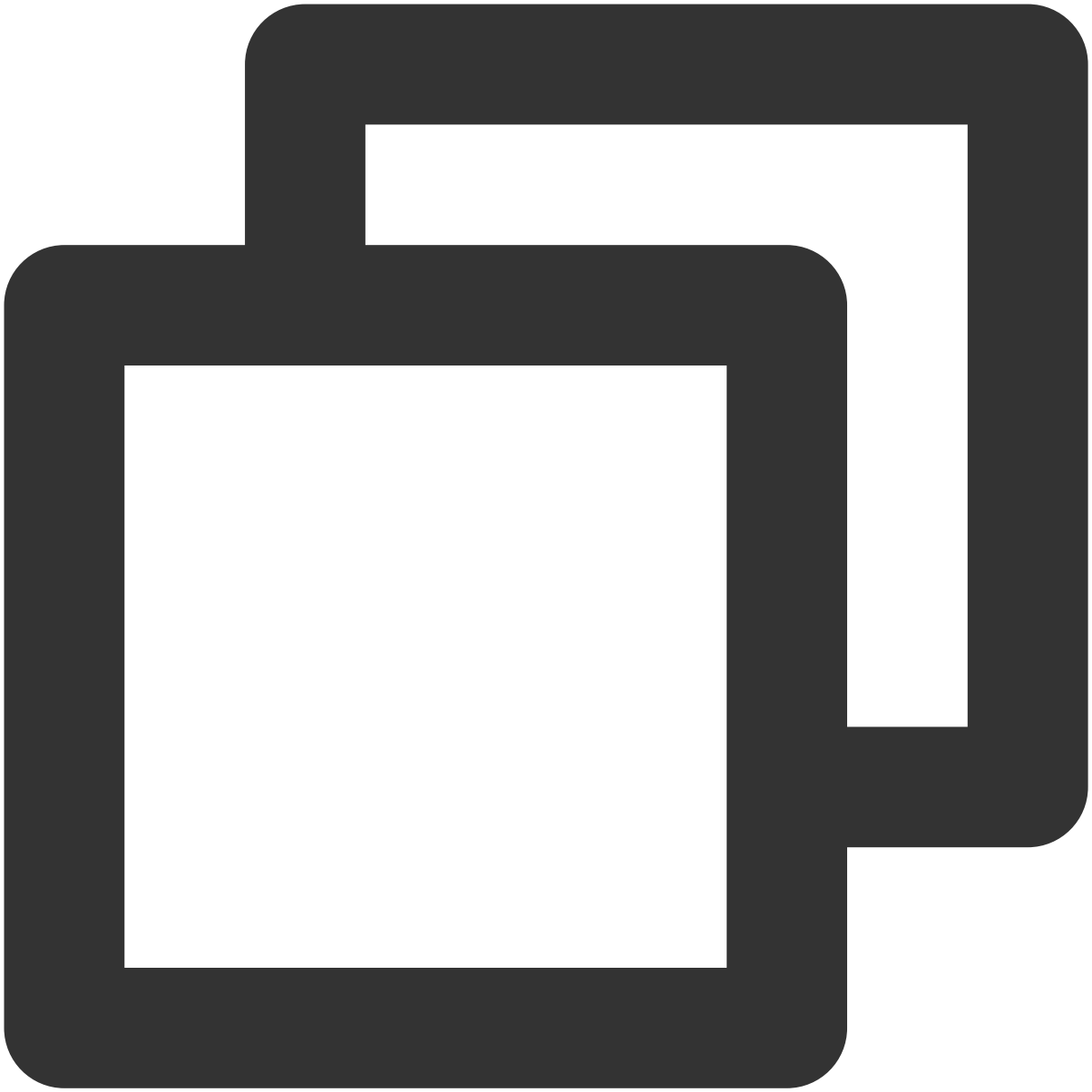
```
server {
 location /nginx_status {
      stub_status;

      access_log off;
      allow 127.0.0.1;
      deny all;
 }
}
```

3. 检查并重新加载 nginx 的配置使其生效。



```
nginx -t
nginx -s reload
```

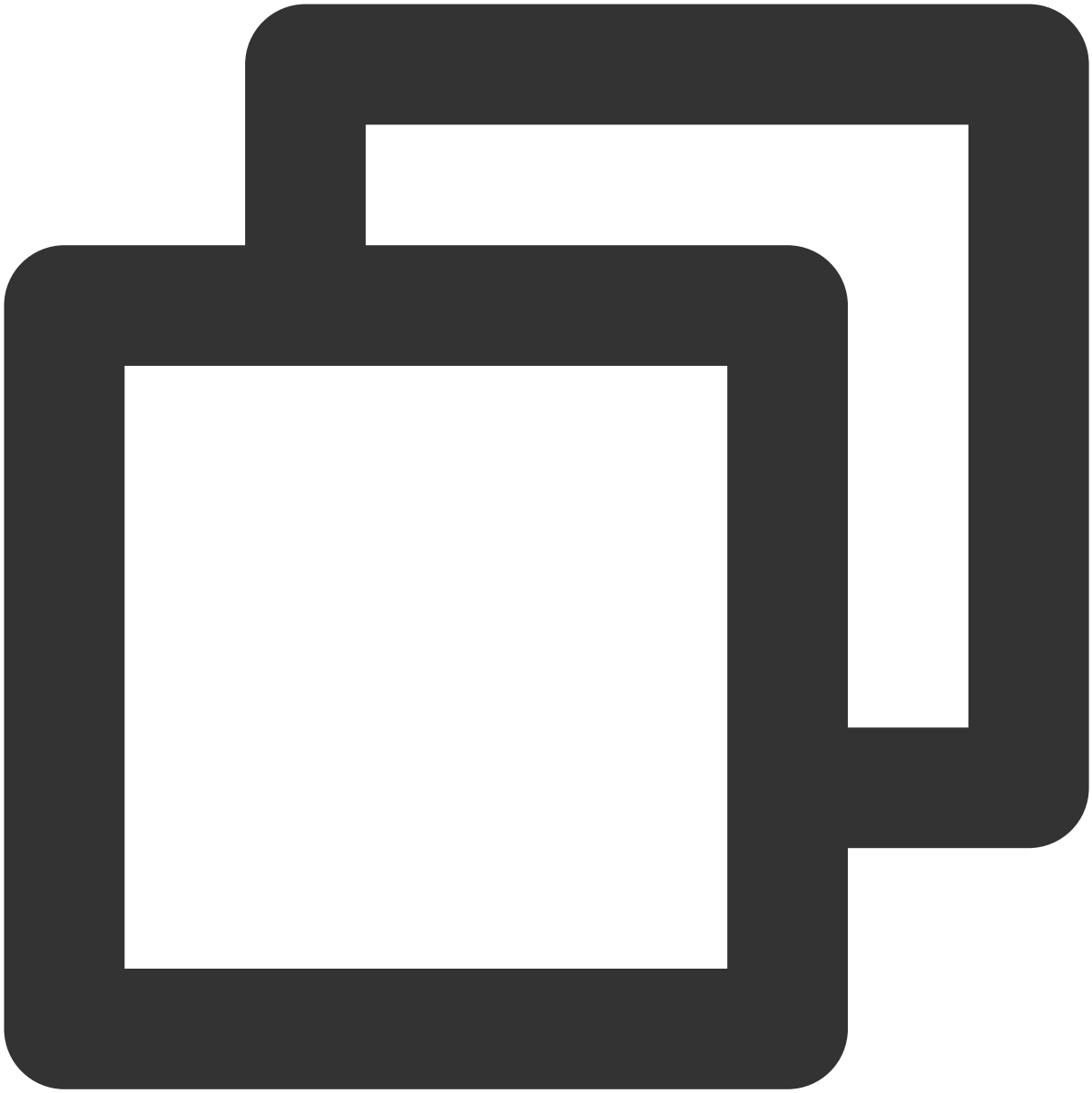
4. 完成上述步之后，可以通过配置的 URL 查看 Nginx 的指标：



```
Active connections: 45
server accepts handled requests
1056958 1156958 4491319
Reading: 0 Writing: 25 Waiting : 7
```

运行 NGINX Prometheus Exporter

执行以下命令启动 NGINX Prometheus Exporter：



```
$ nginx-prometheus-exporter -nginx.scrape-uri http://<nginx>:8080/nginx_status
```

上报指标

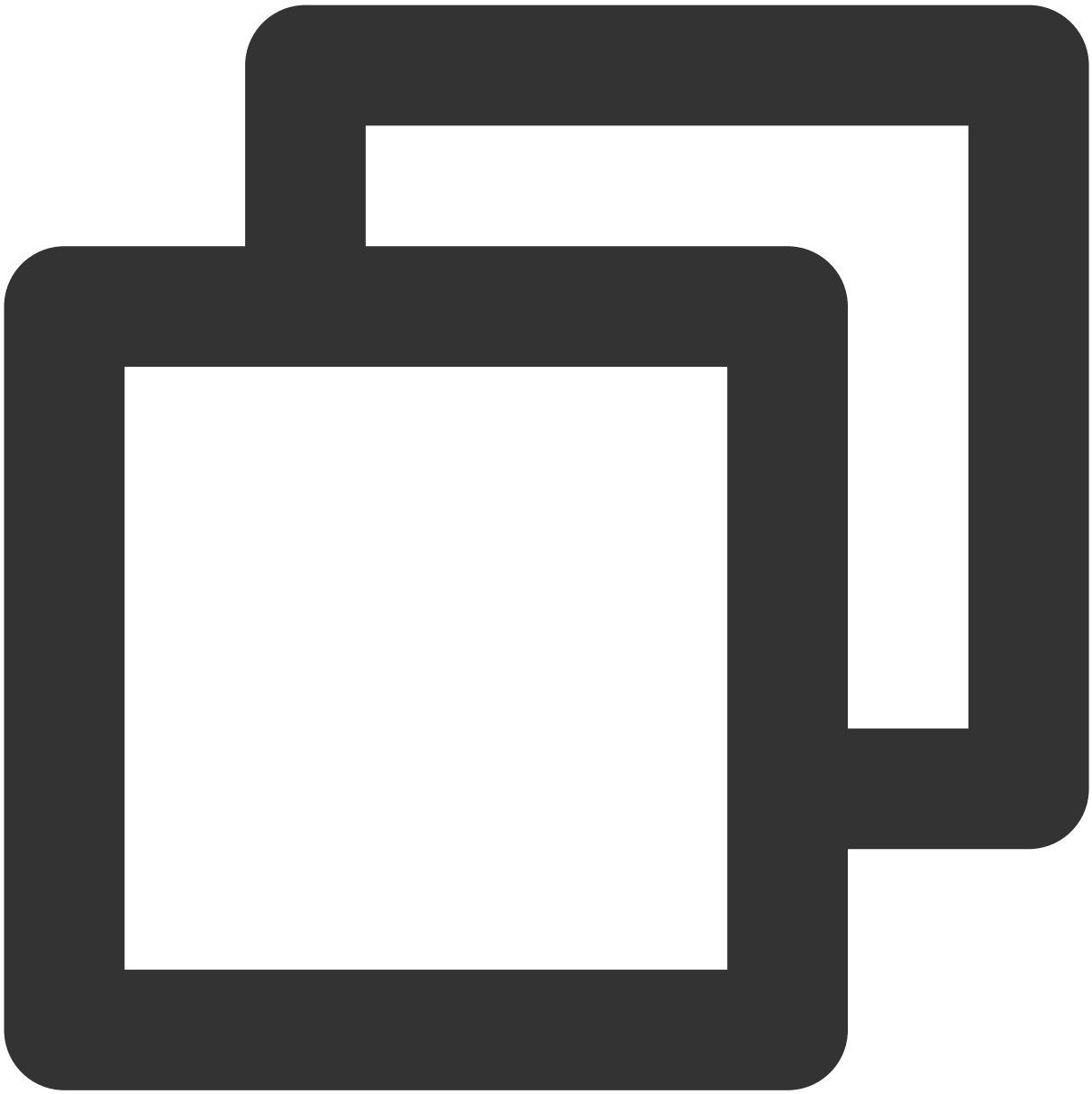
`nginxexporter_build_info` -- exporter 编译信息。

所有的 stub_status 指标。

`nginx_up` -- 展示上次抓取的状态：1表示抓取成功， 0表示抓取失败。

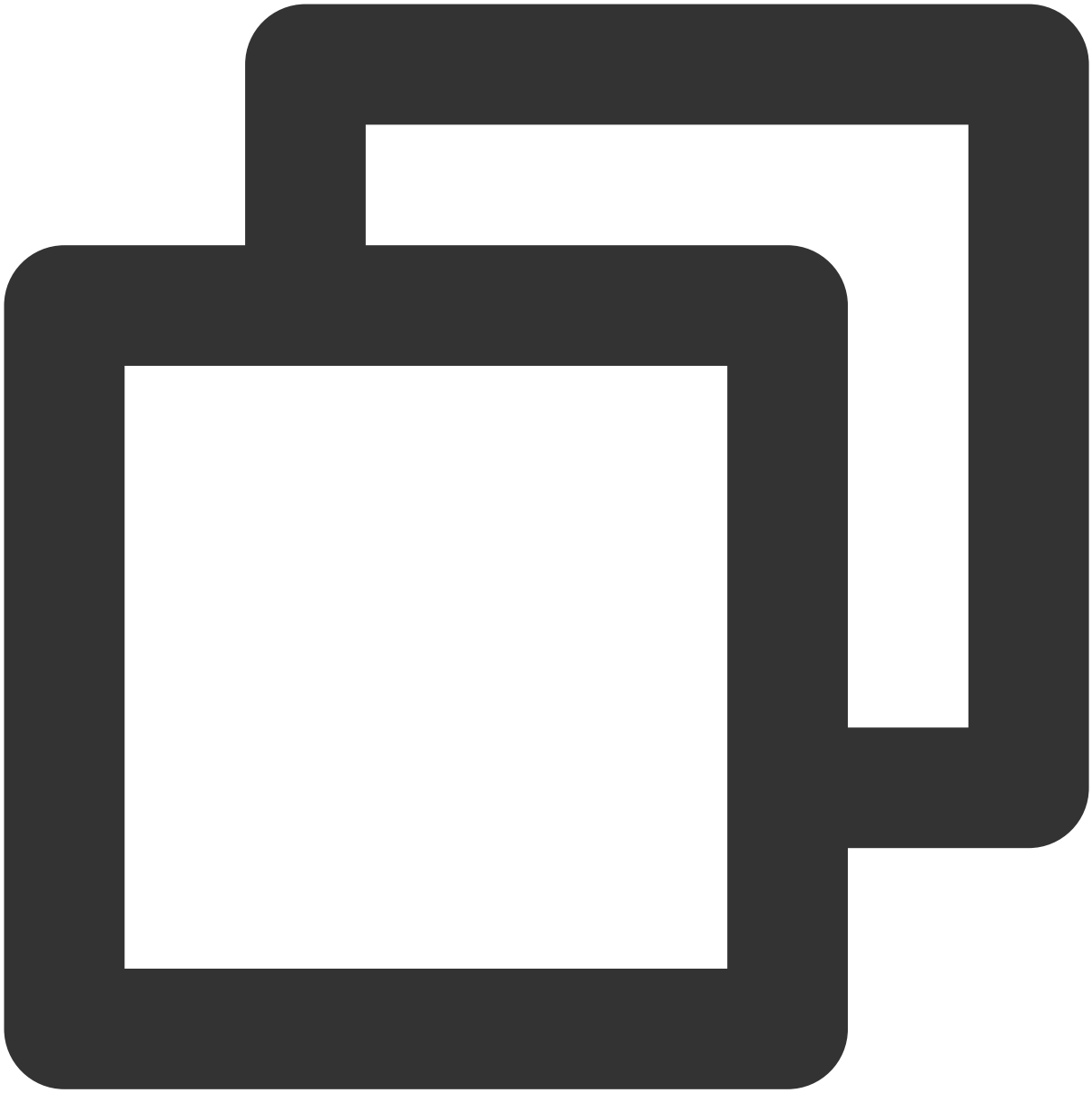
配置 Prometheus 的抓取 Job

1. Nginx Prometheus Exporter 正常运行后，执行以下命令，将 Job 添加到 Prometheus 的抓取任务中。



```
...
  - job_name: 'nginx_exporter'
    static_configs:
      - targets: ['your_exporter:port']
```

2. 通常情况下，Exporter 和 Nginx 并非共同运行，所以数据上报的 `instance` 并不能真实描述是哪个实例，为了方便数据的检索和观察，我们可以修改 `instance` 标签，使用真实的 IP 进行替换以便更加直观。示例如下：



```
...
  - job_name: 'nginx_exporter'
    static_configs:
    - targets: ['192.168.10.10:8080']
    relabel_configs:
     - source_labels: [__address__]
       regex: '.*'
       target_label: instance
       replacement: '10.0.0.1:80'
```

启用数据库监控大盘

Prometheus 监控服务在 Grafana 中提供预先配置的 Nginx Exporter Dashboard，您可以根据以下操作步骤查看 Nginx 监控数据。

1. 登录 Prometheus 监控服务控制台。
2. 单击对应实例 ID 右侧的

，即可查看数据。

Redis Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

在使用数据库 Redis 过程中需要对 Redis 运行状态进行监控，以便了解 Redis 服务是否运行正常，排查 Redis 故障等。Prometheus 监控服务提供基于 Exporter 的方式来监控 Redis 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文为您介绍如何使用 Prometheus 监控 Redis。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 进行统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络 VPC 下，创建腾讯云容器服务 Kubernetes 集群，并为集群创建 命名空间。

在 **Prometheus 监控服务控制台** > **选择“对应的 Prometheus 实例”** > **集成容器服务**中找到对应容器集群完成集成操作，详情请参见Agent 管理。

操作步骤

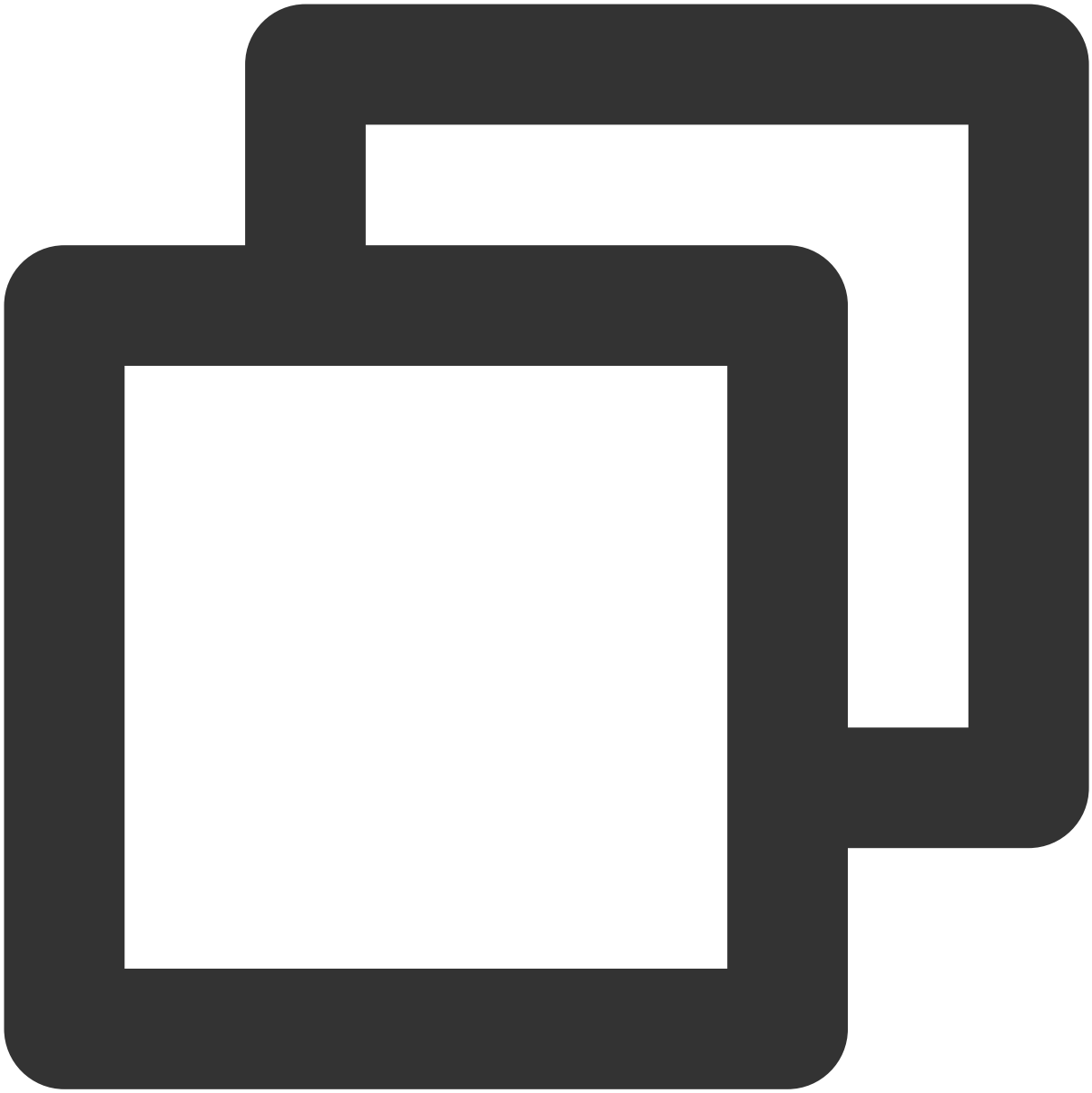
Exporter 部署

1. 登录 容器服务 控制台。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 Redis 密码 > 部署 Redis Exporter > 验证 步骤完成 Exporter 部署。

使用 Secret 管理 Redis 密码

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML 创建资源**，创建 YAML 配置，配置说明如下：

使用 Kubernetes 的 Secret 来管理密码并对密码进行加密处理，在启动 Redis Exporter 的时候直接使用 Secret Key，需要调整对应的 `password`，YAML 配置示例如下：



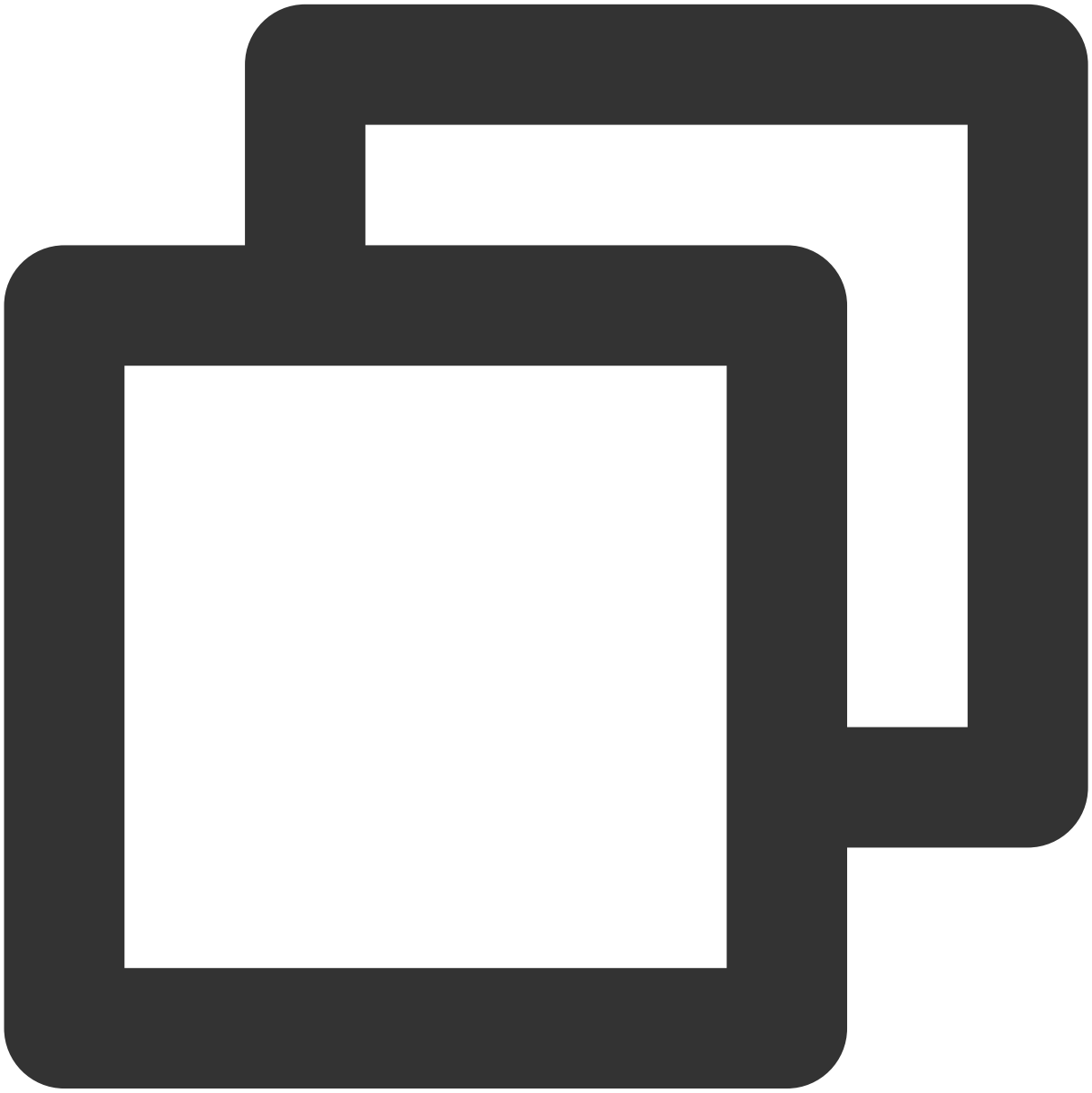
```
apiVersion: v1
kind: Secret
metadata:
    name: redis-secret-test
    namespace: redis-test
type: Opaque
stringData:
    password: you-guess  #对应 Redis 密码
```

部署 Redis Exporter

在 Deployment 管理页面，单击**新建**，选择对应的**命名空间**来进行部署服务。可以通过控制台的方式创建，如下以 YAML 的方式部署 Exporter，YAML 配置示例如下：

说明：

更多 Exporter 详细参数介绍请参见 redis_exporter。

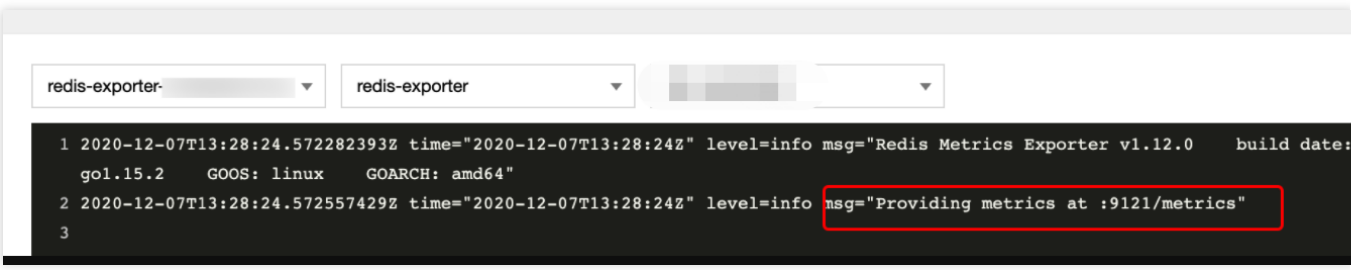


```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
```

```
    k8s-app: redis-exporter # 根据业务需要调整成对应的名称，建议加上 Redis 实例的信息
  name: redis-exporter # 根据业务需要调整成对应的名称，建议加上 Redis 实例的信息
  namespace: redis-test
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: redis-exporter # 根据业务需要调整成对应的名称，建议加上 Redis 实例的信息
  template:
    metadata:
      labels:
        k8s-app: redis-exporter # 根据业务需要调整成对应的名称，建议加上 Redis 实例的信息
    spec:
      containers:
      - env:
        - name: REDIS_ADDR
          value: ip:port # 对应 Redis 的 ip:port
        - name: REDIS_PASSWORD
          valueFrom:
            secretKeyRef:
              name: redis-secret-test
              key: password
        image: ccr.ccs.tencentyun.com/redis-operator/redis-exporter:1.12.0
        imagePullPolicy: IfNotPresent
        name: redis-exporter
        ports:
        - containerPort: 9121
          name: metric-port  # 这个名称在配置抓取任务的时候需要
        securityContext:
          privileged: false
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

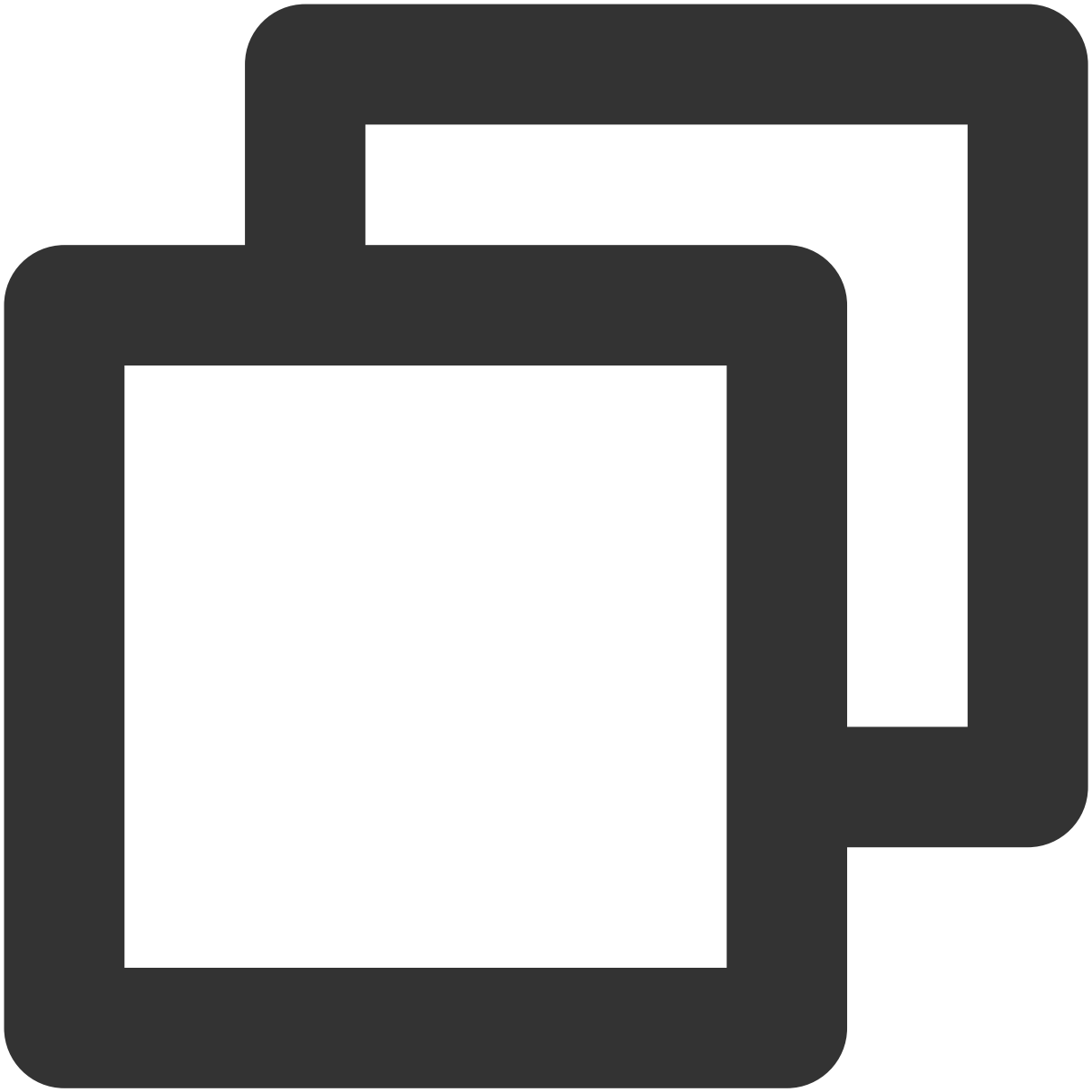
验证

1. 在 Deployment 页面单击上述步骤创建的 Deployment，进入 Deployment 管理页面。
2. 单击**日志**页签，可以查看到 Exporter 成功启动并暴露对应的访问地址，如下图所示：



3. 单击 **Pod 管理**页签，进入 Pod 页面。

4. 在右侧的操作项下单击**远程登录**登录 Pod，在命令行窗口中执行以下 curl 命令对应 Exporter 暴露的地址，可以正常得到对应的 Redis 指标。如发现未能得到对应的数据，请检查一下 `REDIS_ADDR` 和 `REDIS_PASSWORD` 是否正确。示例如下：



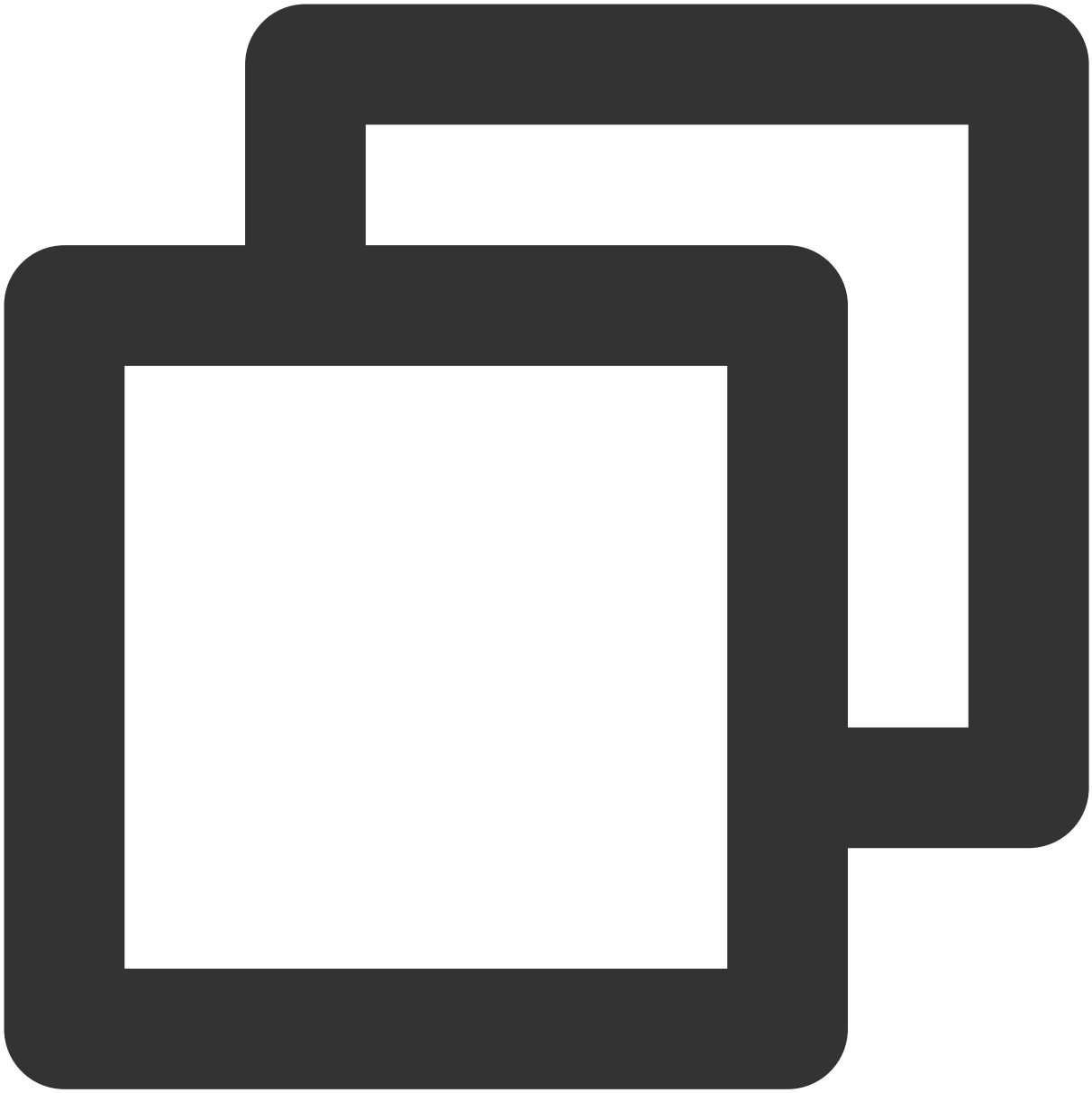
```
curl localhost:9121/metrics
```

命令执行结果如下图所示：

```
# TYPE redis_keyspace_hits_total counter
redis_keyspace_hits_total 29916
# HELP redis_keyspace_misses_total keyspace_misses_total metric
# TYPE redis_keyspace_misses_total counter
redis_keyspace_misses_total 29
# HELP redis_last_slow_execution_duration_seconds The amount of time needed for last slow execu
# TYPE redis_last_slow_execution_duration_seconds gauge
redis_last_slow_execution_duration_seconds 0.011276
# HELP redis_latency_spike_duration_seconds Length of the last latency spike in seconds
# TYPE redis_latency_spike_duration_seconds gauge
redis_latency_spike_duration_seconds{event_name="command"} 0.011
redis_latency_spike_duration_seconds{event_name="fast-command"} 0.022
# HELP redis_latency_spike_last When the latency spike last occurred
# TYPE redis_latency_spike_last gauge
redis_latency_spike_last{event_name="command"} 1.604752448e+09
redis_latency_spike_last{event_name="fast-command"} 1.604738646e+09
# HELP redis_latest_fork_seconds latest_fork_seconds metric
# TYPE redis_latest_fork_seconds gauge
redis_latest_fork_seconds 0
# HELP redis_lazyfree_pending_objects lazyfree_pending_objects metric
# TYPE redis_lazyfree_pending_objects gauge
redis_lazyfree_pending_objects 0
# HELP redis_loading_dump_file loading_dump_file metric
# TYPE redis_loading_dump_file gauge
redis_loading_dump_file 0
# HELP redis_master_repl_offset master_repl_offset metric
# TYPE redis_master_repl_offset gauge
redis_master_repl_offset 2.37644710082e+11
# HELP redis_mem_fragmentation_ratio mem_fragmentation_ratio metric
# TYPE redis_mem_fragmentation_ratio gauge
redis_mem_fragmentation_ratio 1.43
# HELP redis_memory_max_bytes memory_max_bytes metric
# TYPE redis_memory_max_bytes gauge
redis_memory_max_bytes 1.2884901888e+10
# HELP redis_memory_used_bytes memory_used_bytes metric
# TYPE redis_memory_used_bytes gauge
redis_memory_used_bytes 1.1479248e+07
```

添加采取任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: redis-exporter # 填写一个唯一名称
  namespace: cm-prometheus  # namespace固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port  # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics  # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
```

```
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: instance
      replacement: 'crs-xxxxxx' # 调整成对应的 Redis 实例 ID
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: ip
      replacement: '1.x.x.x' # 调整成对应的 Redis 实例 IP
  namespaceSelector:    # 选择要监控pod所在的namespace
    matchNames:
    - redis-test
  selector:     # 填写要监控pod的Label值，以定位目标pod
    matchLabels:
      k8s-app: redis-exporter
```

说明：

由于 `Exporter` 和 `Redis` 部署在不同的服务器上，因此建议通过 Prometheus Relabel 机制将 Redis 实例的信息放到监控指标中，以方便定位问题。

查看监控

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击**集成中心**，进入集成中心页面。找到 Redis 监控，安装对应的 Grafana Dashboard 即可开启 Redis 监控大盘，查看实例相关的监控数据，如下图所示：

告警以及接入

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 单击告警策略，可以添加相应的告警策略，详情请参见 新建告警策略。

MySQL Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

MySQL Exporter 是社区专门为采集 MySQL/MariaDB 数据库监控指标而设计开发，通过 Exporter 上报核心的数据库指标，用于异常报警和监控大盘展示，云监控 Prometheus 提供了与 MySQL Exporter 集成及开箱即用的 Grafana 监控大盘。

目前，Exporter 支持高于5.6版本的 MySQL 和高于10.1版本的 MariaDB。在 MySQL/MariaDB 低于5.6版本时，部分监控指标可能无法被采集。

说明：

为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 来统一管理。

前提条件

在 Prometheus 实例对应地域及私有网络（VPC）下，创建腾讯云容器服务 Kubernetes 集群，并为集群创建 命名空间。

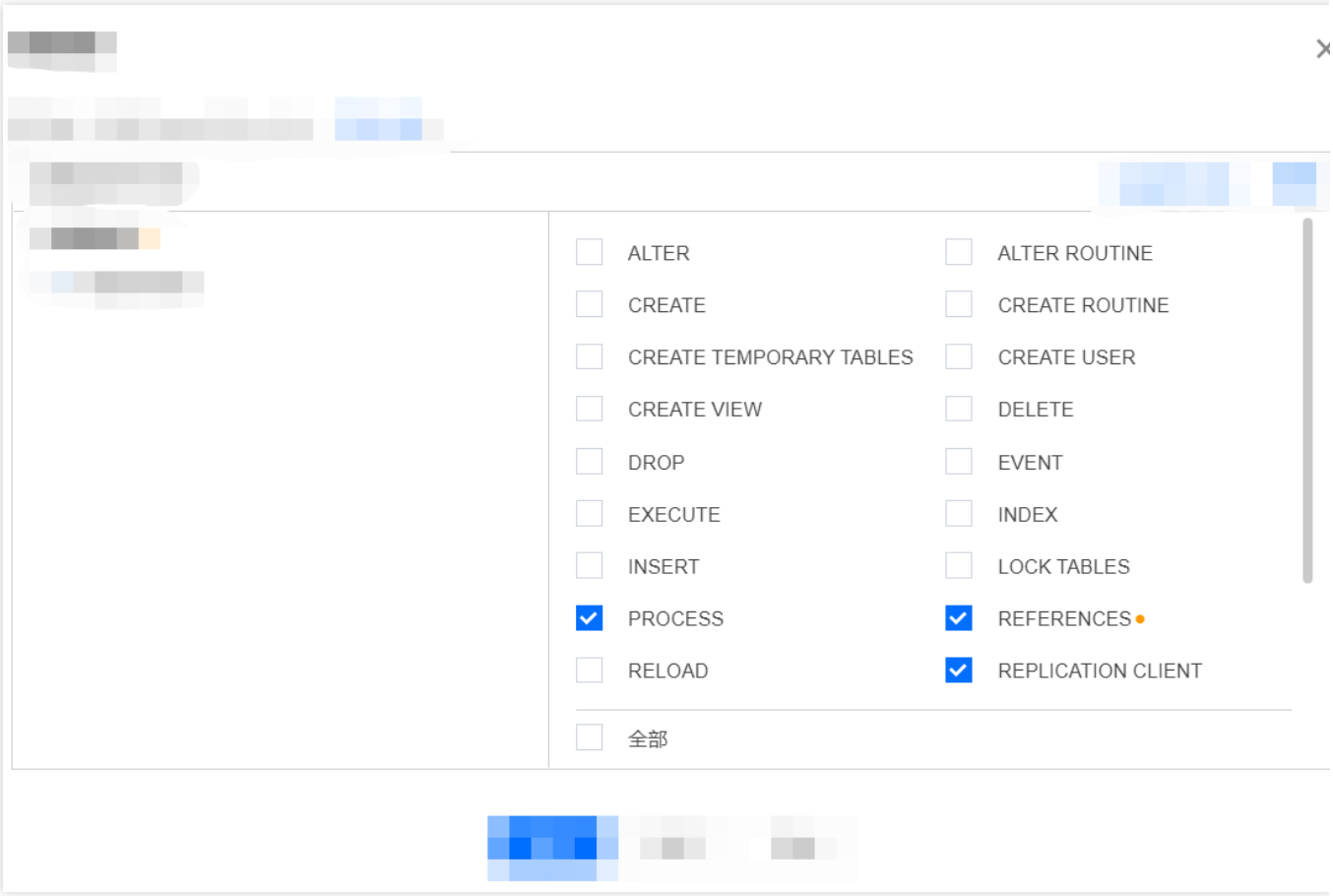
在**云监控 Prometheus 控制台 >选择“对应的 Prometheus 实例” >集成容器服务**中找到对应容器集群完成集成操作，详情请参见 Agent 管理。

操作步骤

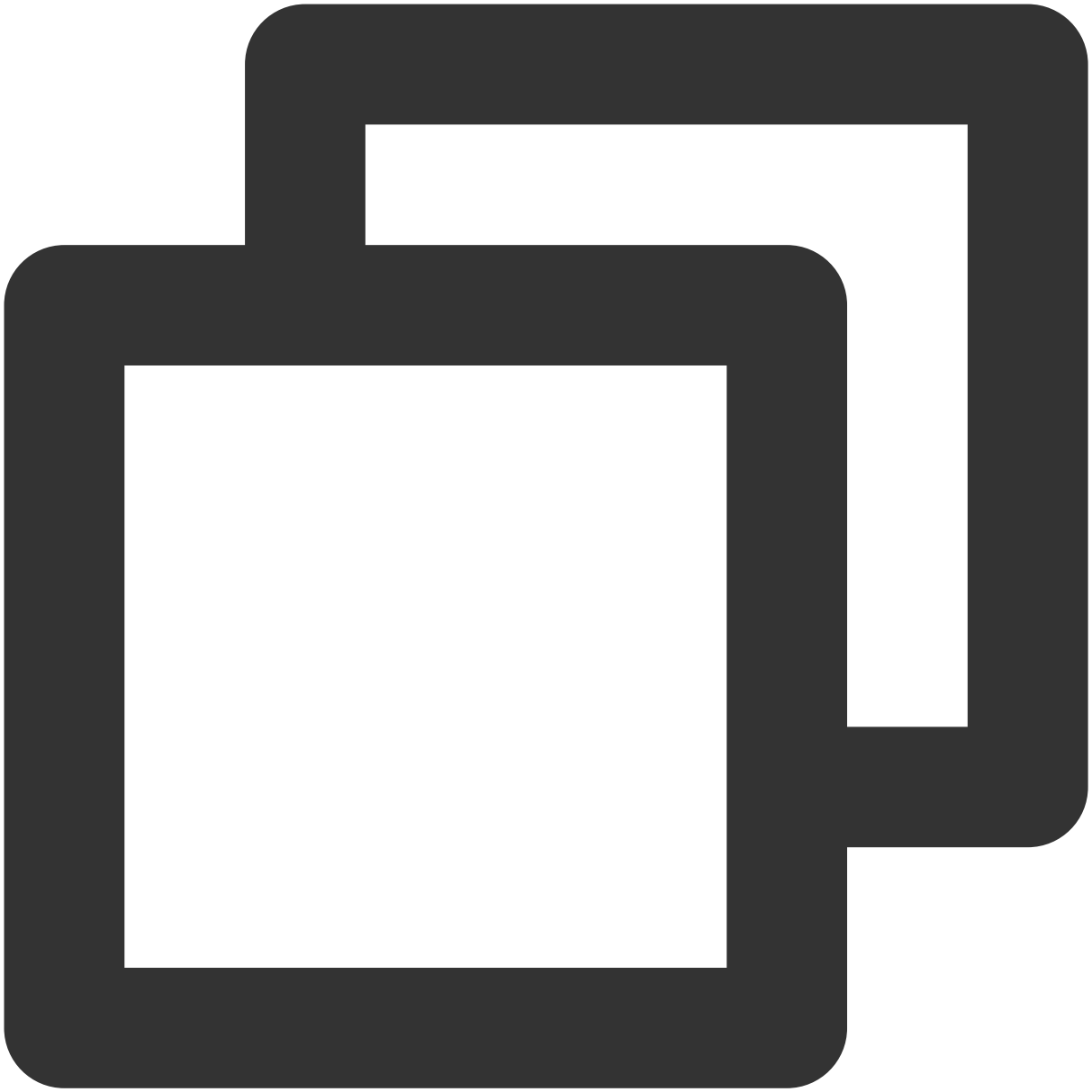
数据库授权

因为 MySQL Exporter 是通过查询数据库中状态数据来对其进行监控，所以需要为对应的数据库实例进行授权。账号和密码需根据实际情况而定，授权步骤如下：

1. 登录 云数据库 MySQL 控制台。
2. 在实例列表页面单击需要授权的数据库名称，进入数据库详情页。
3. 选择**数据库管理 > 账号管理**，进入账号管理页面，根据业务实际需要创建监控建立的账号。
4. 单击账号右侧操作项下的**修改权限**，修改对应权限。示例如下图所示：



您可以通过执行以下命令进行授权：



```
CREATE USER 'exporter'@'ip' IDENTIFIED BY 'XXXXXXXX' WITH MAX_USER_CONNECTIONS 3;
GRANT PROCESS, REPLICATION CLIENT, SELECT ON *.* TO 'exporter'@'ip';
```

说明：

建议为该用户设置最大连接数限制，以避免因监控数据抓取对数据库带来影响。但并非所有的数据库版本中都可以生效，例如 MariaDB 10.1 版本不支持最大连接数设置，则无法生效。详情请参见 MariaDB 说明。

Exporter 部署

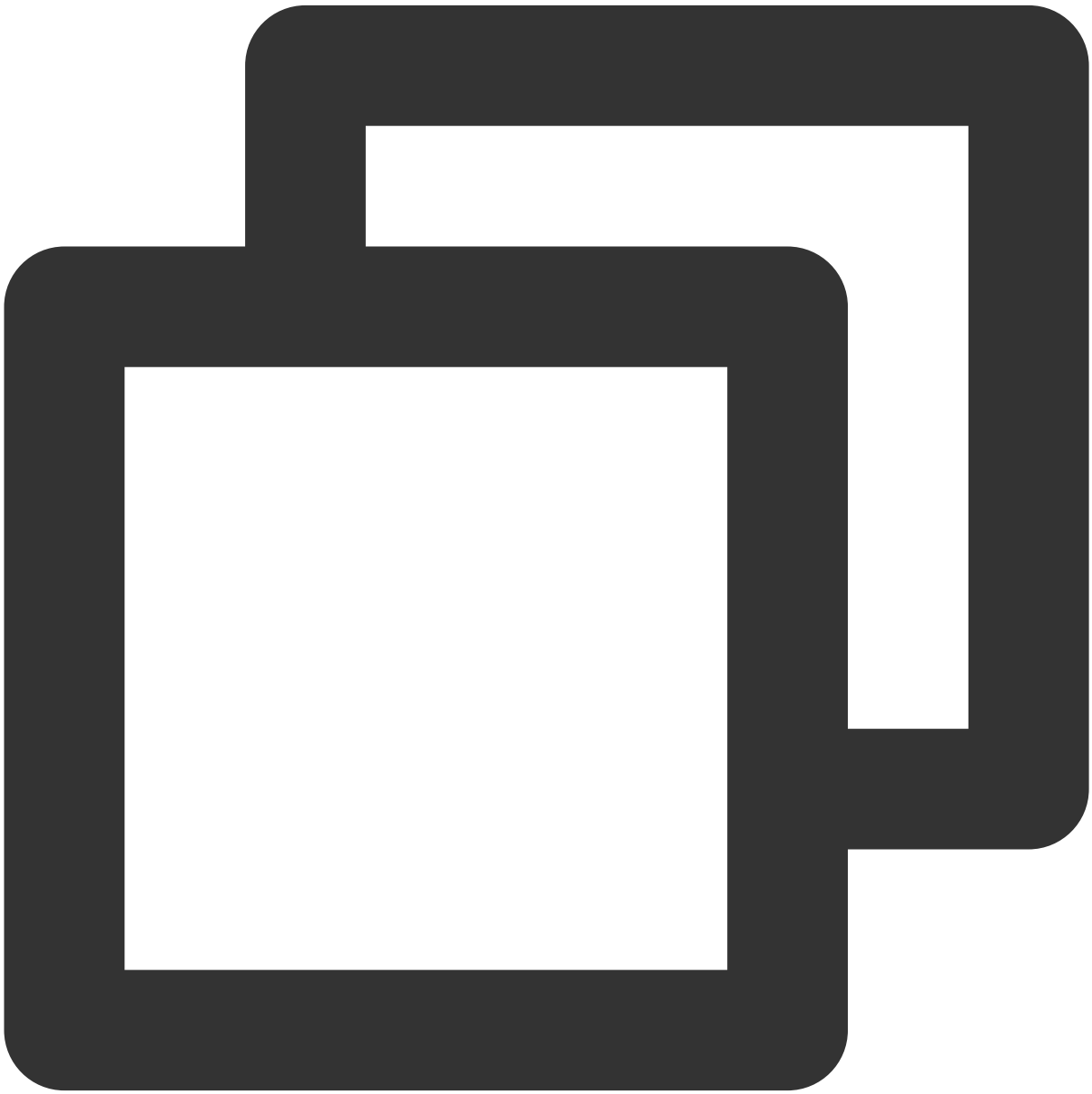
1. 登录 容器服务 控制台。

2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。
3. 执行以下 使用 Secret 管理 MySQL 连接串 > 部署 MySQL Exporter > 验证 步骤完成 Exporter 部署。

使用 Secret 管理 MySQL 连接串

1. 在左侧菜单中选择**工作负载** > **Deployment**，进入 Deployment 页面。
2. 在页面右上角单击 **YAML 创建资源**，创建 YAML 配置，配置说明如下：

使用 Kubernetes 的 Secret 来管理连接串，并对连接串进行加密处理，在启动 MySQL Exporter 的时候直接使用 Secret Key，需要调整对应的**连接串**，YAML 配置示例如下：

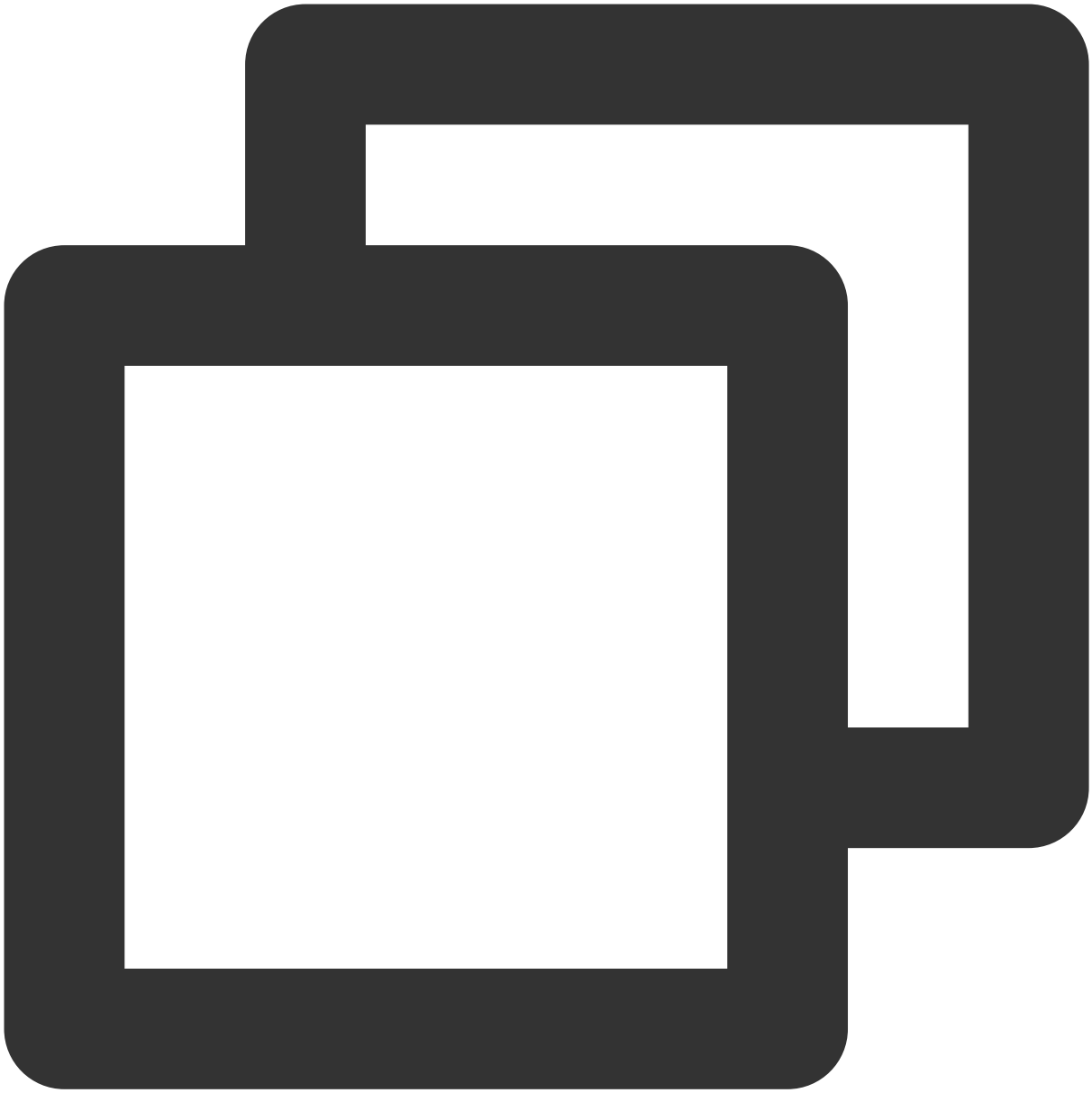


```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: mysql-secret-test
  namespace: mysql-demo
type: Opaque
stringData:
  datasource: "user:password@tcp(ip:port)/"   #对应 MySQL 连接串信息
```

部署 MySQL Exporter

在 Deployment 管理页面，选择对应的命名空间来进行部署服务，可以通过控制台的方式创建。如下以 YAML 的方式部署 Exporter， 配置示例如下：

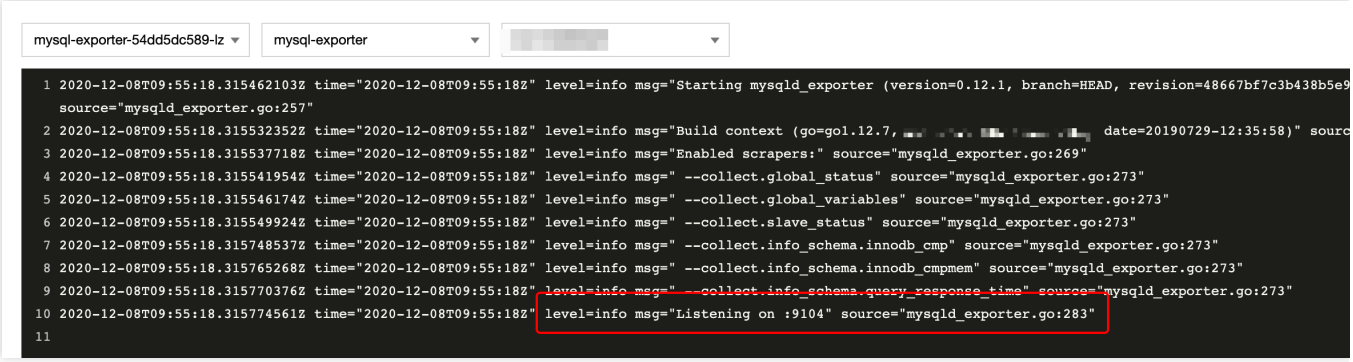


```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: mysql-exporter   # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  name: mysql-exporter   # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  namespace: mysql-demo
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      k8s-app: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  template:
    metadata:
      labels:
        k8s-app: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
    spec:
      containers:
      - env:
        - name: DATA_SOURCE_NAME
          valueFrom:
            secretKeyRef:
              name: mysql-secret-test
              key: datasource
        image: ccr.ccs.tencentyun.com/k8s-comm/mysqld-exporter:0.12.1
        imagePullPolicy: IfNotPresent
        name: mysql-exporter
        ports:
        - containerPort: 9104
          name: metric-port
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

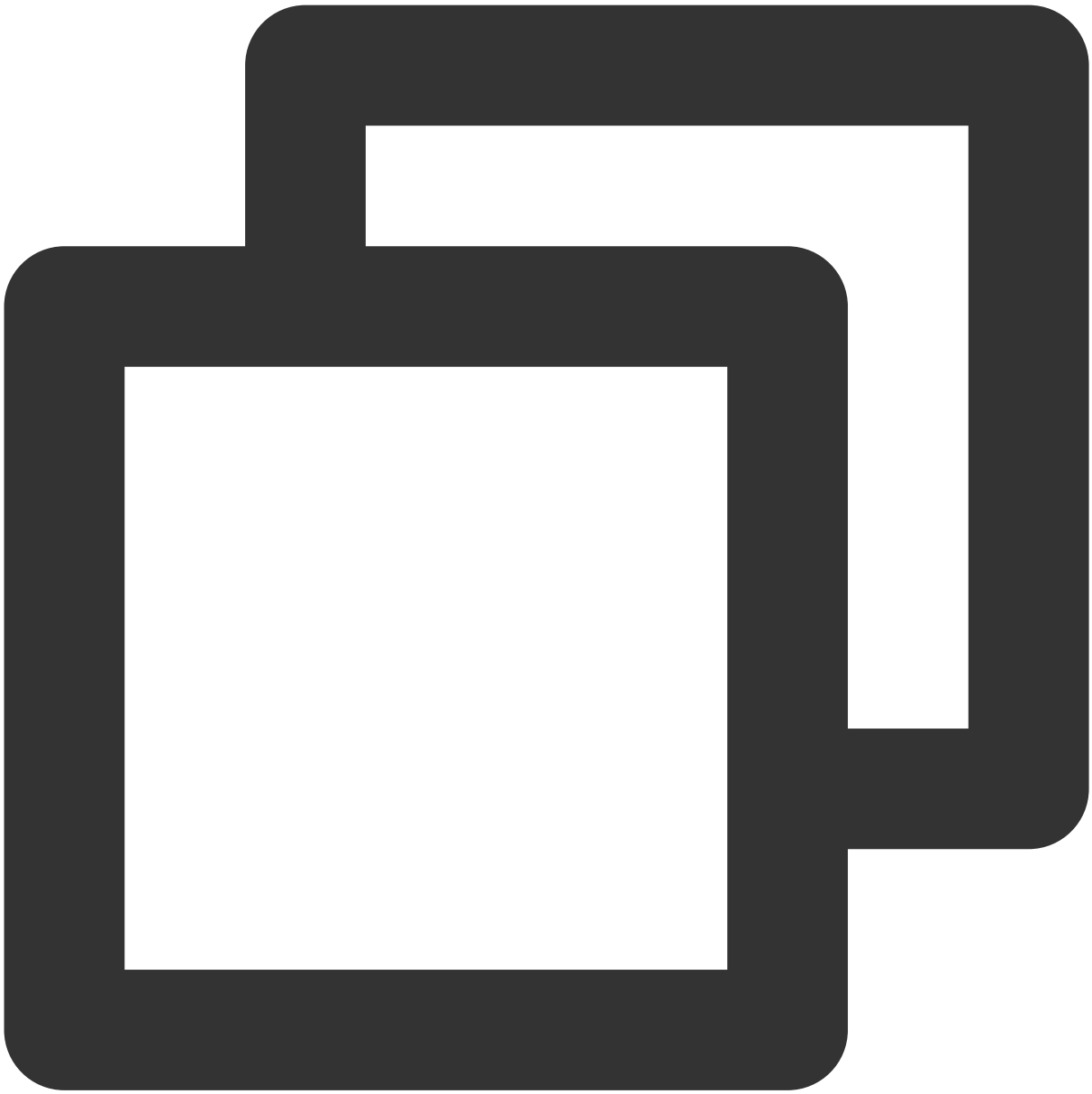
验证

1. 在 Deployment 页面单击上述步骤创建的 Deployment，进入 Deployment 管理页面。
2. 单击**日志**页签，可以查看到 Exporter 成功启动并暴露对应的访问地址，如下图所示：



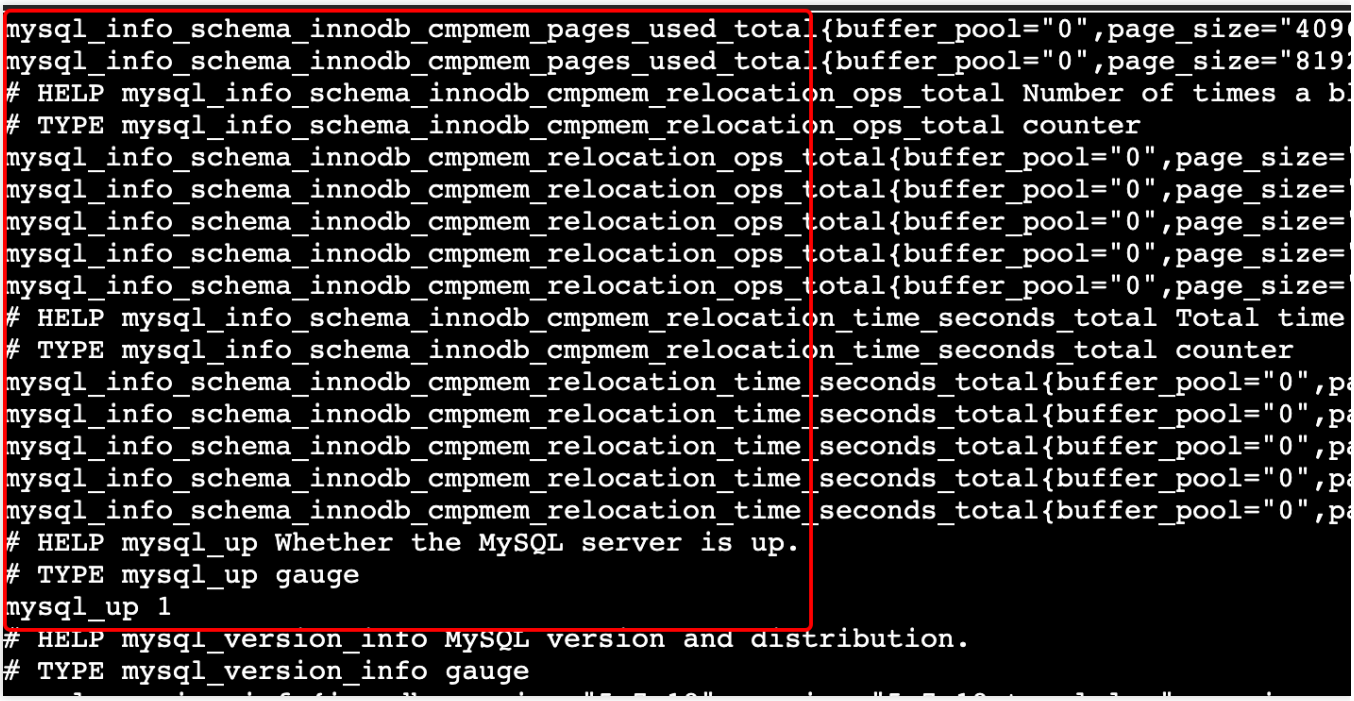
3. 单击 **Pod 管理**页签进入 Pod 页面。

4. 在右侧的操作项下单击**远程登录**登录 Pod，在命令行窗口中执行以下 curl 命令对应 Exporter 暴露的地址，可以正常得到对应的 MySQL 指标。如发现未能得到对应的数据，请检查**连接串**是否正确，具体如下：



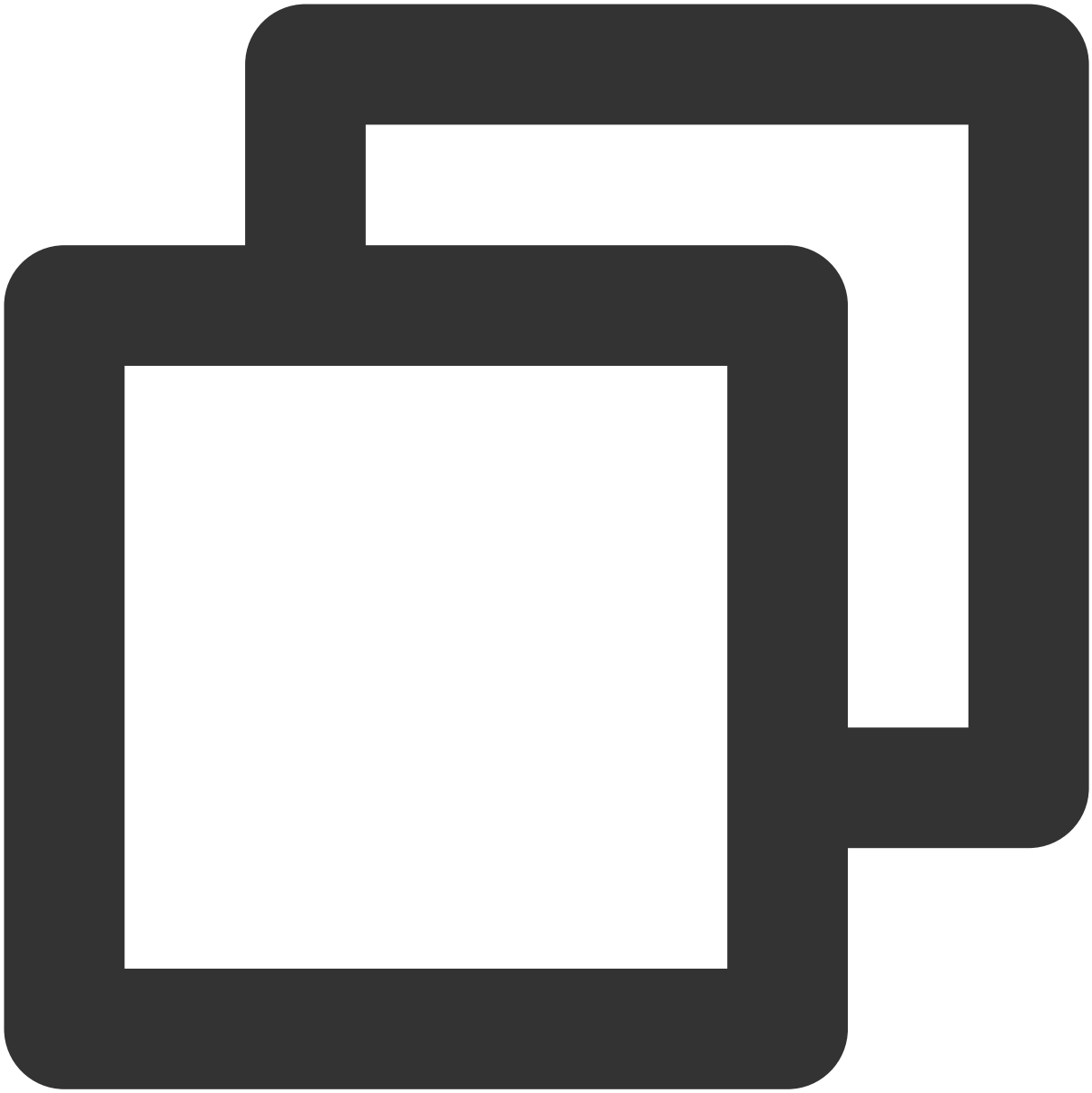
```
curl localhost:9104/metrics
```

执行结果如下图所示：



添加采取任务

1. 登录 云监控 Prometheus 控制台，选择对应 Prometheus 实例进入管理页面。
2. 通过集成容器服务列表单击**集群 ID** 进入到容器服务集成管理页面。
3. 通过服务发现添加 `Pod Monitor` 来定义 Prometheus 抓取任务，YAML 配置示例如下：



```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: mysql-exporter   # 填写一个唯一名称
  namespace: cm-prometheus   # namespace固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port     # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics   # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
```

```
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: instance
      replacement: 'crs-xxxxxx' # 调整成对应的 MySQL 实例 ID
    - action: replace
      sourceLabels:
      - instance
      regex: (.*)
      targetLabel: ip
      replacement: '1.x.x.x' # 调整成对应的 MySQL 实例 IP
  namespaceSelector:    # 选择要监控pod所在的namespace
    matchNames:
    - mysql-demo
  selector:  # 填写要监控pod的Label值，以定位目标pod
    matchLabels:
      k8s-app: mysql-exporter
```

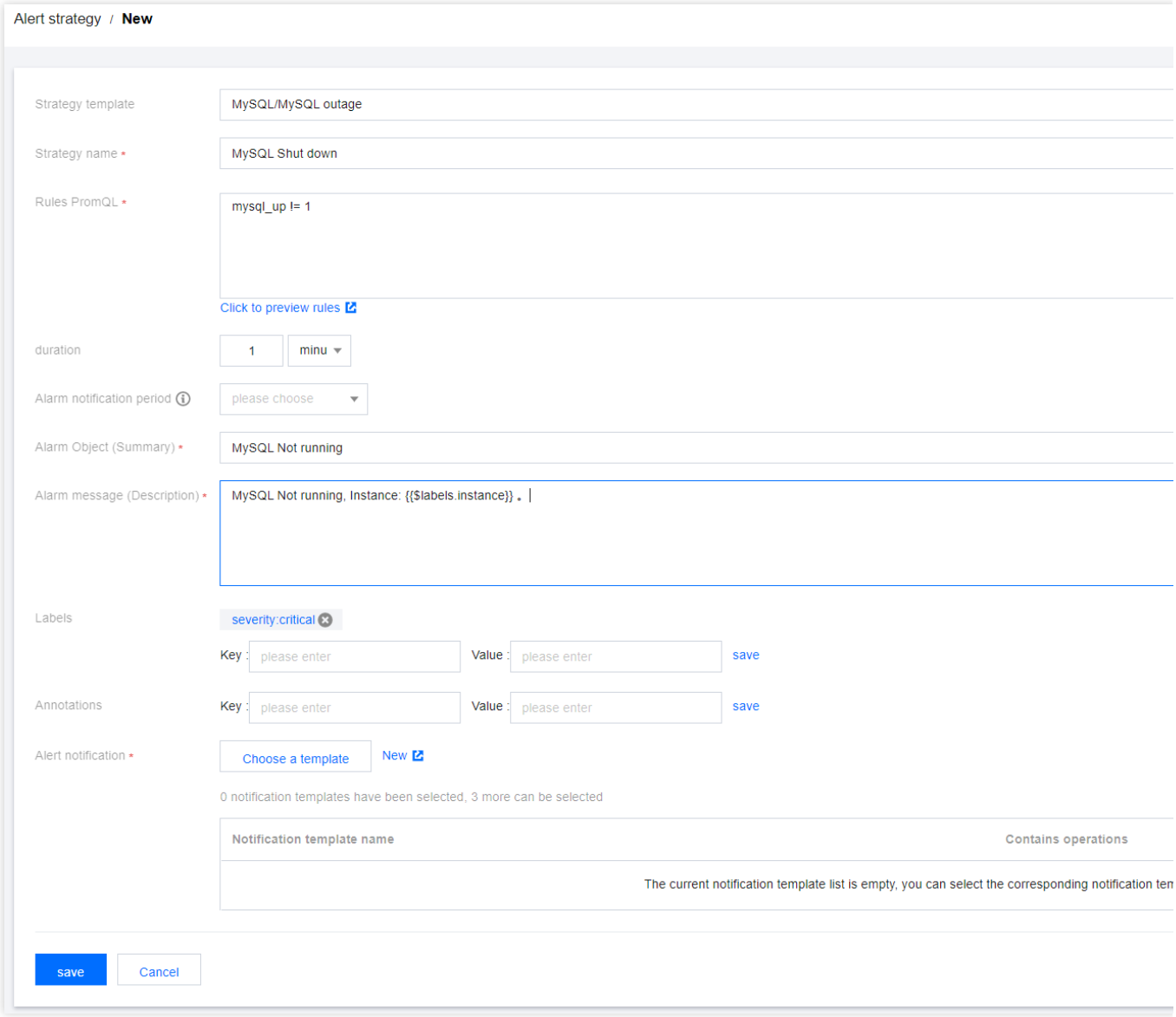
查看监控

1. 登录 云监控 Prometheus 控制台，选择对应 Prometheus 实例进入管理页面。

2. 单击**集成中心**，进入集成中心页面。找到 MySQL 监控，安装对应的 Grafana Dashboard 即可开启 MySQL 监控大盘，查看实例相关的监控数据，如下图所示：

告警以及接入

腾讯云 Prometheus 托管服务内置了部分 MySQL 数据库的报警策略模板，可根据业务实际的情况调整对应的阈值来添加告警策略。详情请参见 新建告警策略。



MySQL Exporter 采集参数说明

MySQL Exporter 使用各种 `Collector` 来控制采集数据的启停，具体参数如下：

名称	MySQL 版本	描述
collect.auto_increment.columns	5.1	在 information_schema 中采集 auto_incr
collect.binlog_size	5.1	采集所有注册的 binlog 文件大小。

collect.engine_innodb_status	5.1	从 SHOW ENGINE INNODB STATUS 中
collect.engine_tokudb_status	5.6	从 SHOW ENGINE TOKUDB STATUS 中
collect.global_status	5.1	从 SHOW GLOBAL STATUS（默认开启
collect.global_variables	5.1	从 SHOW GLOBAL VARIABLES（默认开
collect.info_schema.clientstats	5.5	如果设置了 userstat=1，设置成 true 来开
collect.info_schema.innodb_metrics	5.6	从 information_schema.innodb_metrics 中
collect.info_schema.innodb_tablespaces	5.7	从 information_schema.innodb_sys_tabl
collect.info_schema.innodb_cmp	5.5	从 information_schema.innodb_cmp 中采集 据。
collect.info_schema.innodb_cmpmem	5.5	从 information_schema.innodb_cmpmem compression 的监控数据。
collect.info_schema.processlist	5.1	从 information_schema.processlist 中采集
collect.info_schema.processlist.min_time	5.1	线程可以被统计所维持的状态的最小时间
collect.info_schema.query_response_time	5.5	如果 query_response_time_stats 被设置 分布。
collect.info_schema.replica_host	5.6	从 information_schema.replica_host_sta
collect.info_schema.tables	5.1	从 information_schema.tables 中采集状态
collect.info_schema.tables.databases	5.1	设置需要采集表状态的数据库, 或者设置
collect.info_schema.tablestats	5.1	如果设置了 userstat=1，设置成 true 来采
collect.info_schema.schemastats	5.1	如果设置了 userstat=1，设置成 true 来采
collect.info_schema.userstats	5.1	如果设置了 userstat=1，设置成 true 来采
collect.perf_schema.eventsstatements	5.6	从 performance_schema.events_statem 采集监控数据。
collect.perf_schema.eventsstatements.digest_text_limit	5.6	设置正常文本语句的最大长度。 (默认：
collect.perf_schema.eventsstatements.limit	5.6	事件语句的限制数量。（默认：250）
collect.perf_schema.eventsstatements.timelimit	5.6	限制事件语句 'last_seen' 可以保持多久,
collect.perf_schema.eventsstatementssum	5.7	从 performance_schema.events_statem

		summed 中采集监控数据。
collect.perf_schema.eventswaits	5.5	从 performance_schema.events_waits_sun 中采集监控数据。
collect.perf_schema.file_events	5.6	从 performance_schema.file_summary_ 据。
collect.perf_schema.file_instances	5.5	从 performance_schema.file_summary_ 据。
collect.perf_schema.indexiowaits	5.6	从 performance_schema.table_io_waits_ 中采集监控数据。
collect.perf_schema.tableiowaits	5.6	从 performance_schema.table_io_waits_ 控数据。
collect.perf_schema.tablelocks	5.6	从 performance_schema.table_lock_wai 监控数据。
collect.perf_schema.replication_group_members	5.7	从 performance_schema.replication_gro 据。
collect.perf_schema.replication_group_member_stats	5.7	从 from performance_schema.replicatior 集监控数据。
collect.perf_schema.replication_applier_status_by_worker	5.7	从 performance_schema.replication_app 监控数据。
collect.slave_status	5.1	从 SHOW SLAVE STATUS（默认开启）
collect.slave_hosts	5.1	从 SHOW SLAVE HOSTS 中采集监控数
collect.heartbeat	5.1	从 heartbeat 中采集监控数据。
collect.heartbeat.database	5.1	数据库心跳检测的数据源。（默认：hea
collect.heartbeat.table	5.1	表心跳检测的数据源。（默认：heartbea
collect.heartbeat.utc	5.1	对当前的数据库服务器使用 UTC 时间戳 with `--utc`)。（默认：false）

全局配置参数

名称	描述
config.my-cnf	用来读取数据库认证信息的配置文件 `.my.cnf` 位置。 (默

	认： `~/.my.cnf`)
log.level	日志级别。（默认：info）
exporter.lock_wait_timeout	为链接设置 lock_wait_timeout（单位：秒）以避免对元数据的锁时间太长。（默认：2）
exporter.log_slow_filter	添加 log_slow_filter 以避免抓取的慢查询被记录。 提示：不支持 Oracle MySQL。
web.listen-address	web 端口监听地址。
web.telemetry-path	metrics 接口路径。
version	打印版本信息。

heartbeat 心跳检测

如果开启 `collect.heartbeat` ， mysqld_exporter 会通过心跳检测机制抓取复制延迟数据。

Consul Exporter 接入

最近更新时间：2024-01-29 15:55:14

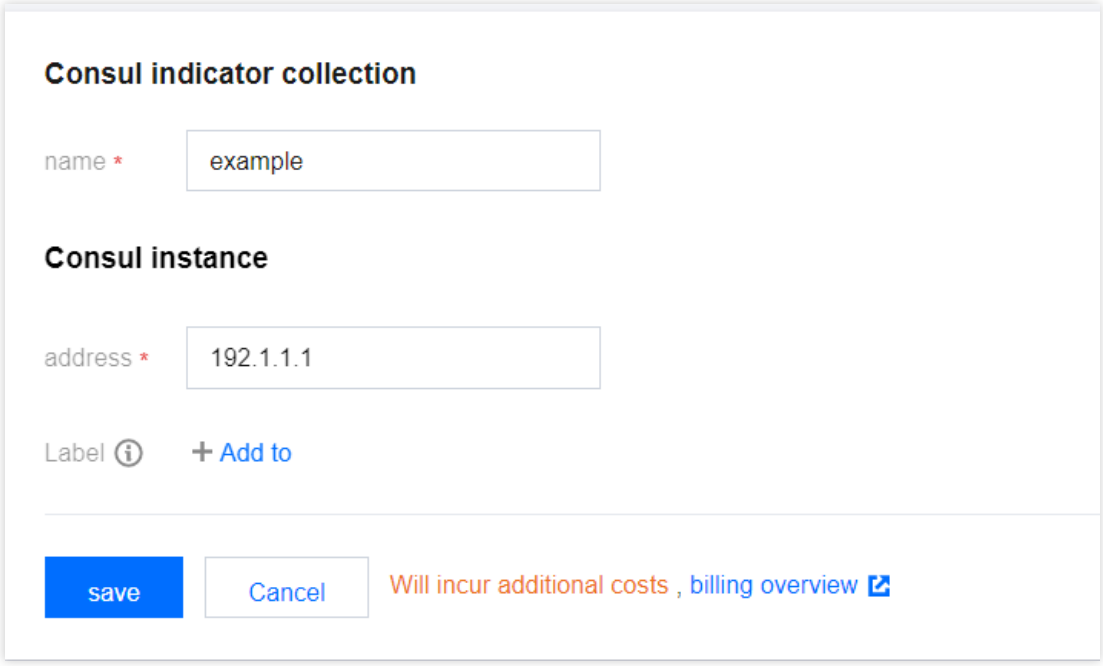
操作场景

在使用 Consul 过程中需要对 Consul 运行状态进行监控，以便了解 Consul 服务是否运行正常，排查 Consul 故障等。 Prometheus 监控服务提供基于 Exporter 的方式来监控 Consul 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文为您介绍如何使用 Prometheus 监控服务 Consul。

操作步骤

1. 登录 Prometheus 控制台。
2. 在实例列表中，选择对应的 Prometheus 实例。
3. 进入实例详情页，单击**集成中心**。
4. 在集成中心选择 `Consul` 单击安装进行集成。

配置说明



名称	描述

名称	每个集成需要一个唯一名称
地址	要采集的 Consul 实例的地址和端口
标签	增加具有业务含义的标签，会自动添加到 Prometheus 的 Label 中

查看监控

可以通过监控大盘清晰看到如下监控状态：

1. Consul 集群节点状态。
2. Consul 上注册服务的状态。

Memcached Exporter 接入

最近更新时间：2024-01-29 15:55:14

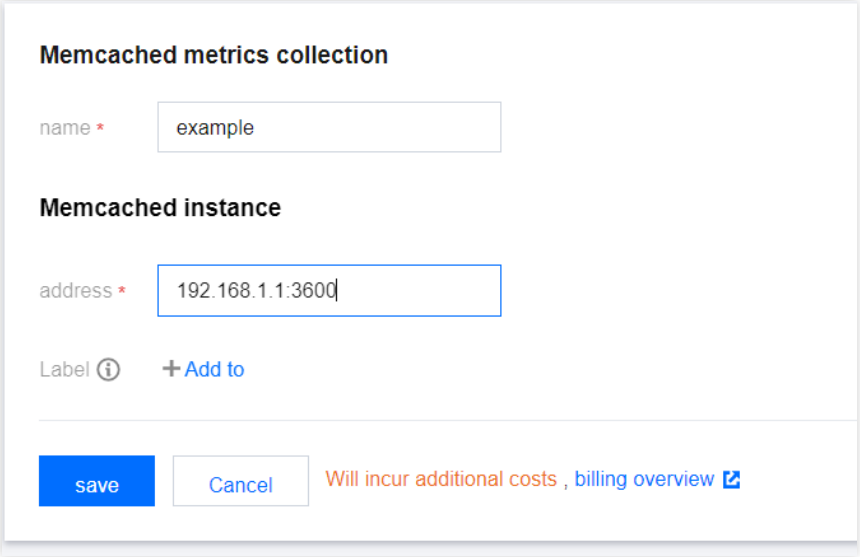
操作场景

在使用 Memcached 过程中需要对 Memcached 运行状态进行监控，以便了解 Memcached 服务是否运行正常，排查 Memcached 故障等。 Prometheus 监控服务提供基于 Exporter 的方式来监控 Memcached 运行状态，并提供了开箱即用的 Grafana 监控大盘。本文为您介绍如何使用 Prometheus 监控服务 Memcached。

操作步骤

1. 登录 Prometheus 控制台。
2. 在实例列表中，选择对应的 Prometheus 实例。
3. 进入实例详情页，单击**集成中心**。
4. 在集成中心选择 `Memcached` 单击安装进行集成。

配置说明



名称	描述
名称	每个集成需要一个唯一名称
地址	要采集的 Memcached 实例的地址和端口

标签	增加具有业务含义的标签，会自动添加到 Prometheus 的 Label 中

查看监控

可以通过监控大盘清晰看到如下监控状态：

1. 内存使用率，同时显示已使用的内存和内存总量。
2. 当前的 Get 命令的命中率，同时显示服务运行期间 Get 命令命中和未命中的比例。
3. Memcached 移除旧数据和回收过期数据的速率，同时显示服务运行期间移除和回收的数据总数。
4. Memcached 存储的数据总量。
5. 从网络中读取和写入的字节数。
6. 当前打开的连接数。
7. 服务运行期间 Get 命令和 Set 命令的比例。
8. 当前各命令的产生速率。


(Read / Written bytes)
98 KiB
49 KiB
0 B
-49 KiB
-98 KiB
19:10
19:20
19:30
19:40
19:50
20:00
read 84.9 KiB 46.0 KiB 42.4 KiB 5.3065 MiB
written 72.7 KiB 25.7 KiB 14.1 KiB 2.9559 MiB
(Connections)
30
20
10
0
max avg current
count 23.00 5.84 4.00
(Get & Set ratio)
100.00%
75.00%
50.00%
25.00%
0%
Get
Set
Commands
4 K
3 K
2 K
1 K
0
curre
cas
decr
delete
flush
incr
touch

其他 Exporter 接入

最近更新时间：2024-01-29 15:55:14

操作场景

Prometheus 监控服务目前已经提供了常用基础组件的集成方式，及开箱即用的监控大屏，由于兼容原生 Prometheus，所以您也可以安装社区别的 Exporter。

操作方式

如果您所使用的基础组件还没有提供相应的集成方式，可以参考如下方式进行集成，及自定义监控大屏来满足相应的监控需求。

1. 开源社区 Exporter 列表。
2. 参见 MySQL 的集成方式。

CVM node_expoter

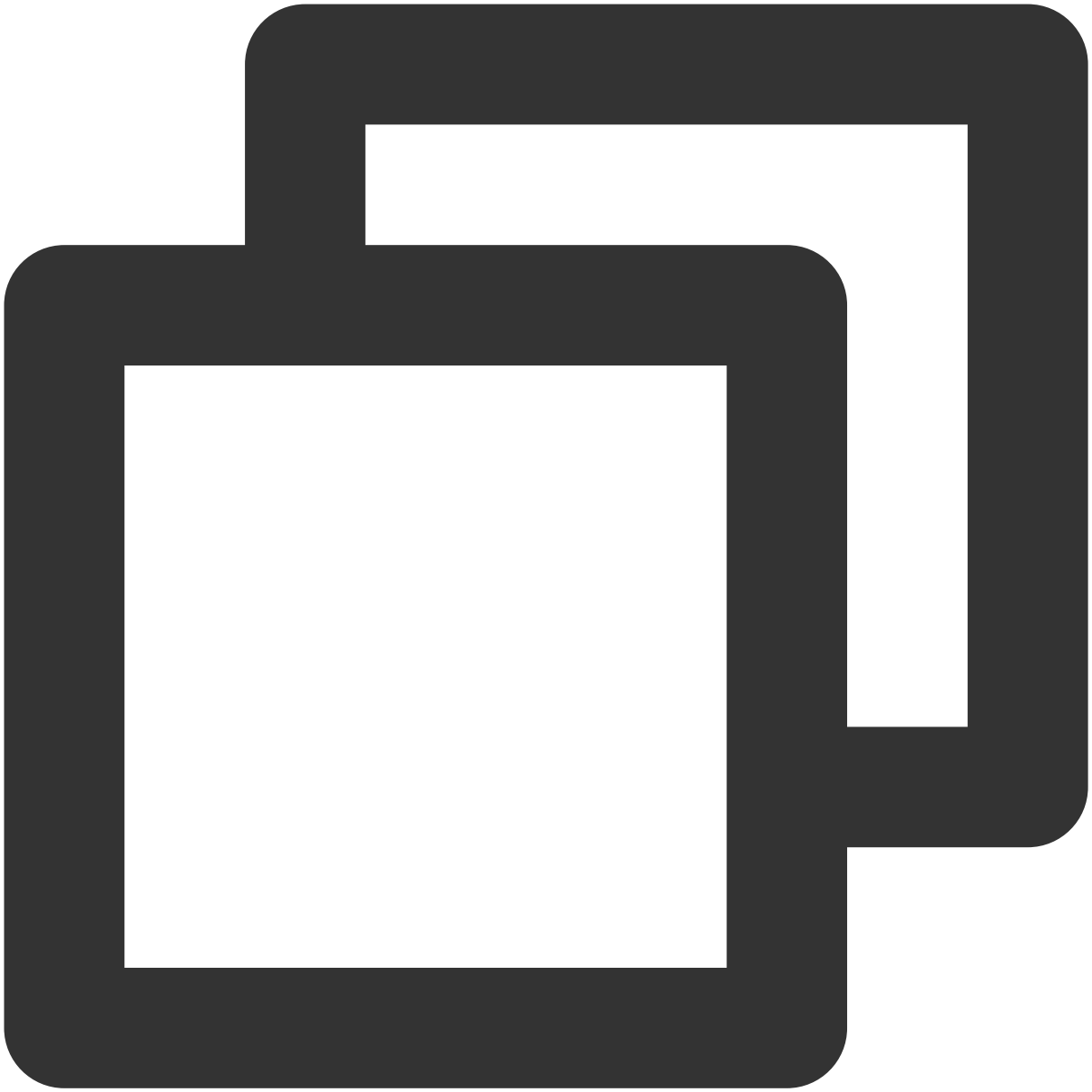
最近更新时间：2024-01-29 15:55:14

本文将为您介绍如何通过安装 node_exporter，暴露云服务器基础指标至 Prometheus 监控服务。

操作步骤

步骤1：下载安装 node_exporter

在需要上报的云服务器上，下载并安装 node_exporter（采集基础指标数据的 exporter），您可以单击进入 Prometheus 开源官网下载地址 node_exporter，也可以直接执行下列命令，下载解压：



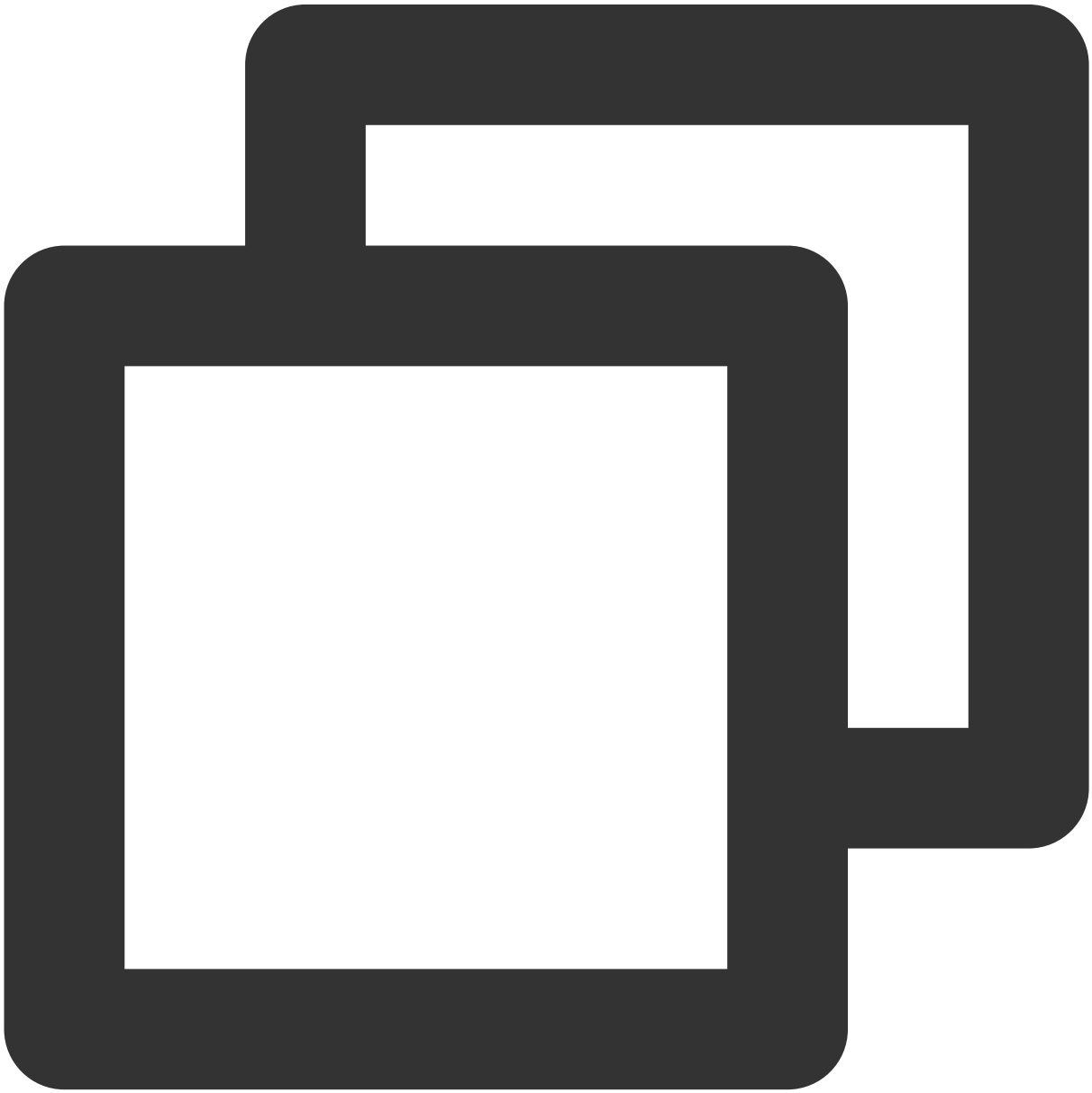
```
wget https://github.com/prometheus/node_exporter/releases/download/v1.3.1/node_expo
```

文件目录如下:

```
[root@VM-0-7-centos node_exporter-1.2.2.linux-amd64]# ll
total 18080
-rw-r--r-- 1 3434 3434    11357 Aug  6  2021 LICENSE
-rwxr-xr-x 1 3434 3434 18494215 Aug  6  2021 node_exporter
-rw-r--r-- 1 3434 3434      463 Aug  6  2021 NOTICE
[root@VM-0-7-centos node_exporter-1.2.2.linux-amd64]# ./node_exporter
```

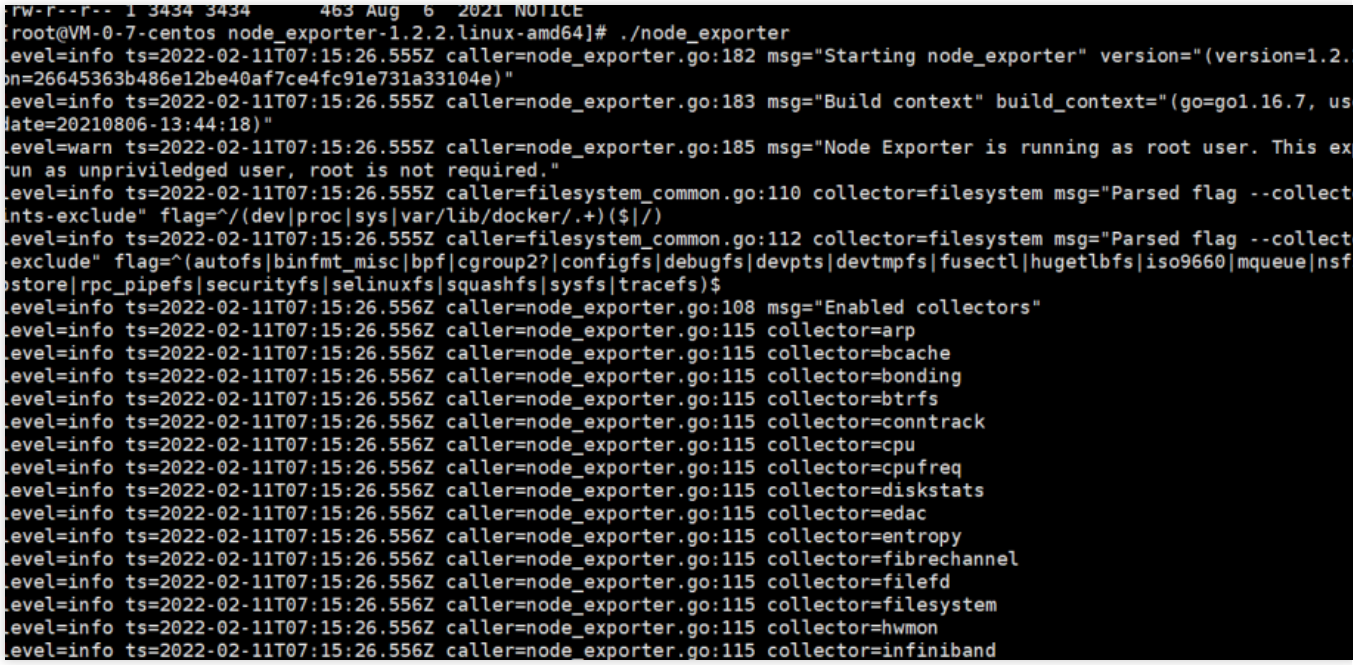
步骤2：运行 node_exporter ， 采集基础监控数据

1. 进入相关文件夹，执行 node_exporter。

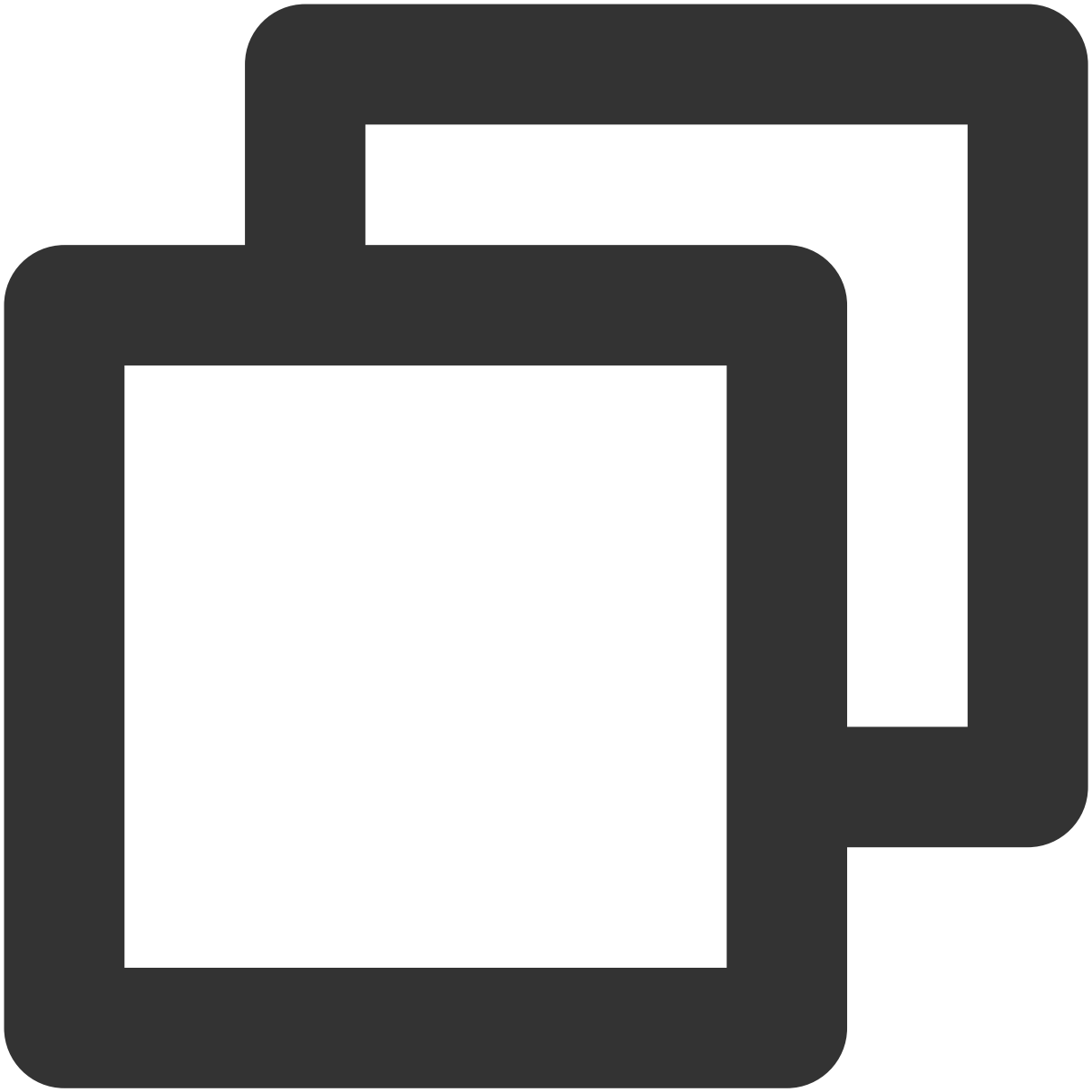


```
cd node_exporter-1.3.1.linux-amd64
./node_exporter
```

如下图所示即为成功采集到了基础监控数据。

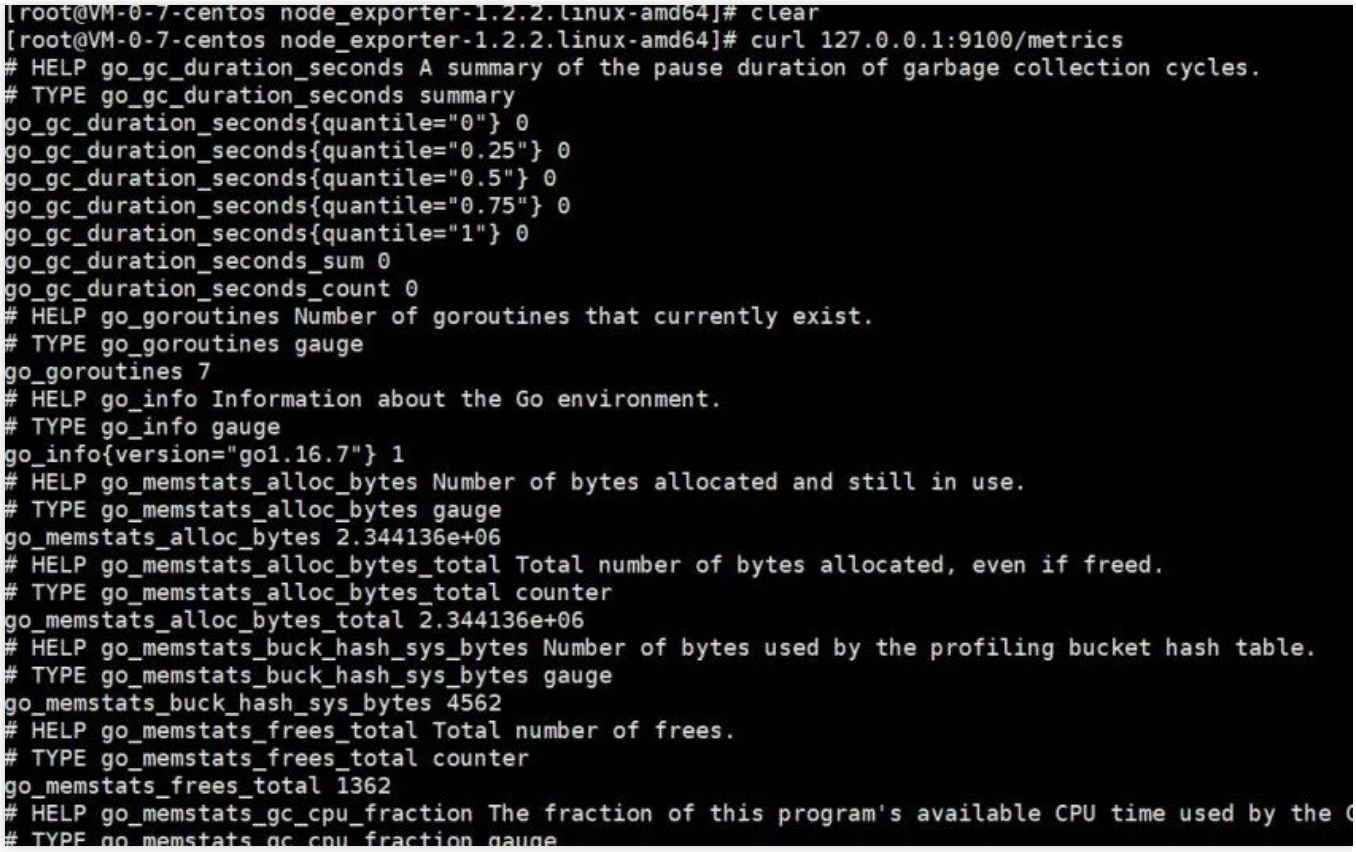


2. 可通过下列命令，将该基础监控数据暴露在9100端口：



```
curl 127.0.0.1:9100/metrics
```

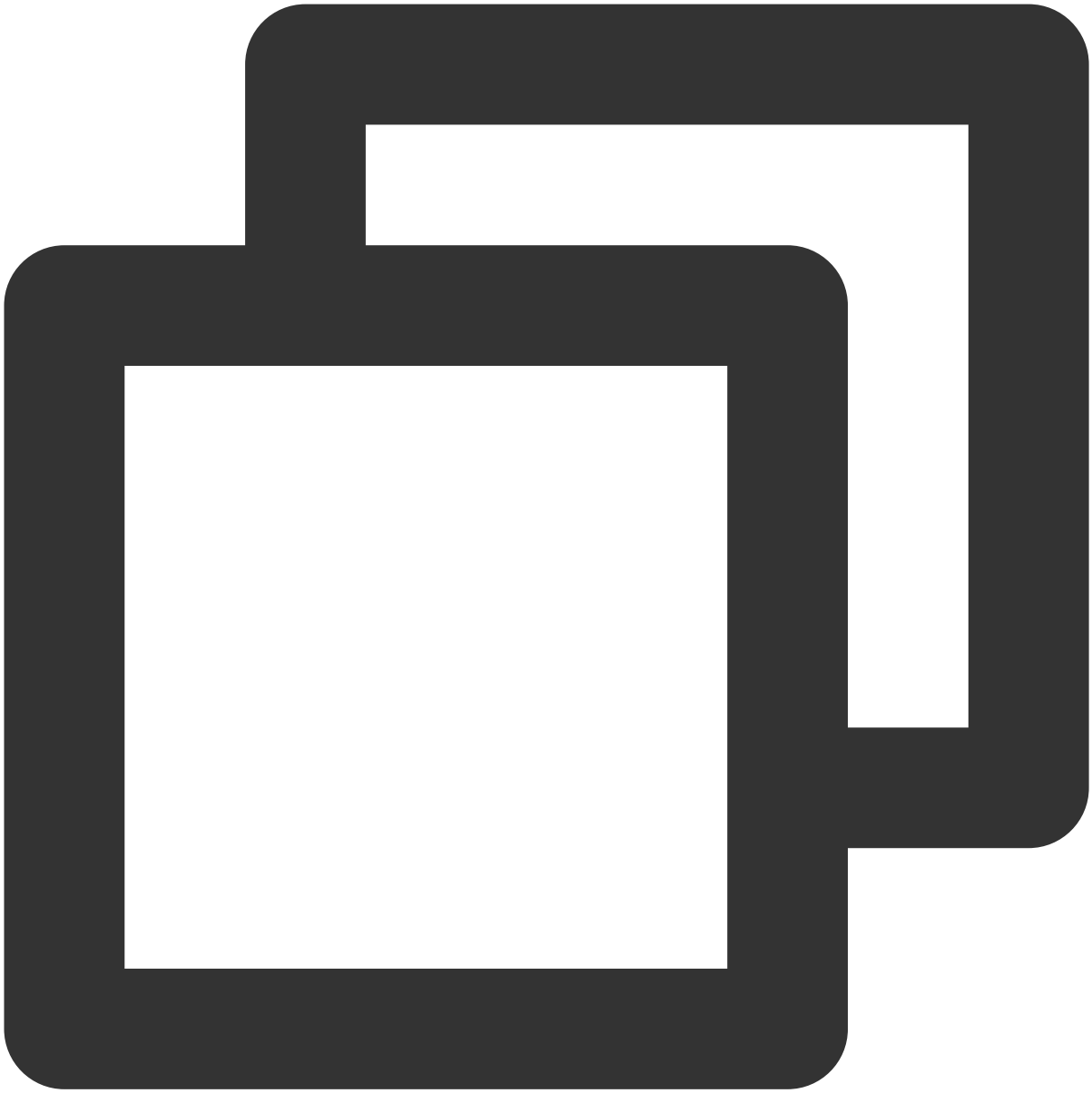
如下图为执行命令后，可看到暴露出来的指标监控数据。



步骤3：采集配置

登录 Prometheus 监控服务控制台，进入**集成中心** > **选择云服务器**，在任务配置中根据页面提示进行配置。

抓取任务参考配置如下：

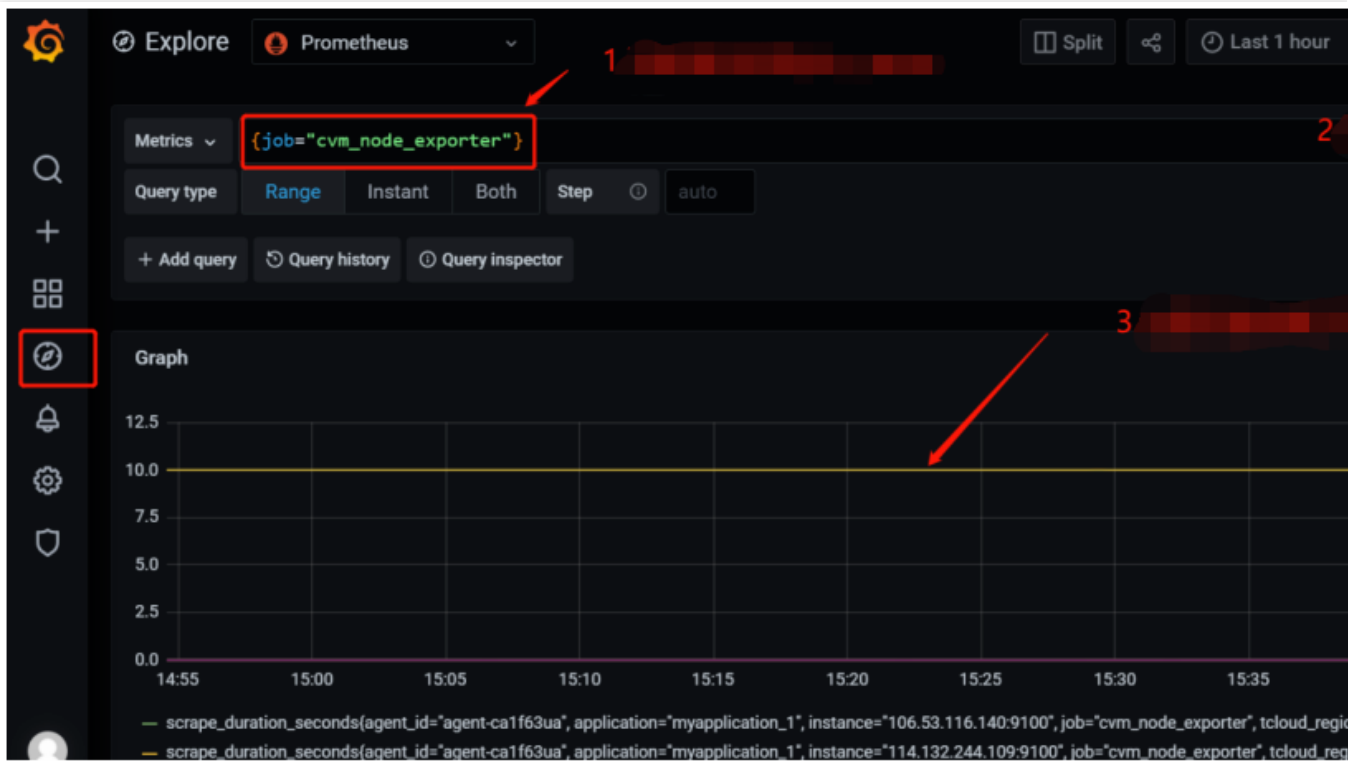


```
job_name: example-job-name
metrics_path: /metrics
cvm_sd_configs:
- region: ap-guangzhou
  ports:
  - 9100
  filters:
  - name: tag:示例标签键
    values:
    - 示例标签值
relabel_configs:
```

```
- source_labels: [__meta_cvm_instance_state]
  regex: RUNNING
  action: keep
- regex: __meta_cvm_tag_(.*)
  replacement: $1
  action: labelmap
- source_labels: [__meta_cvm_region]
  target_label: region
  action: replace
```

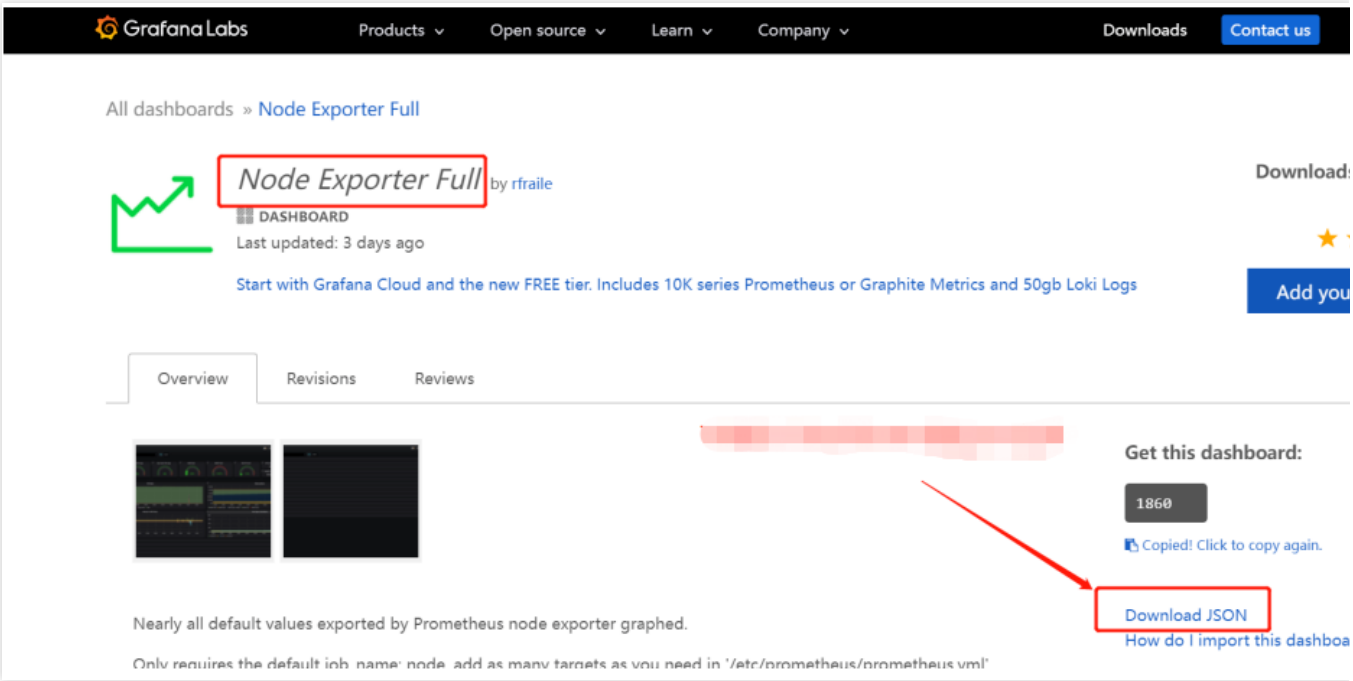
步骤4：查看数据是否上报成功：

登录 Prometheus 监控服务控制台，单击 Grafana 图标，进入 Grafana。
如下图所示，到 Explore 搜索下 `{job="cvm_node_exporter"}` 看是否有数据，若有数据，则表示上报成功。

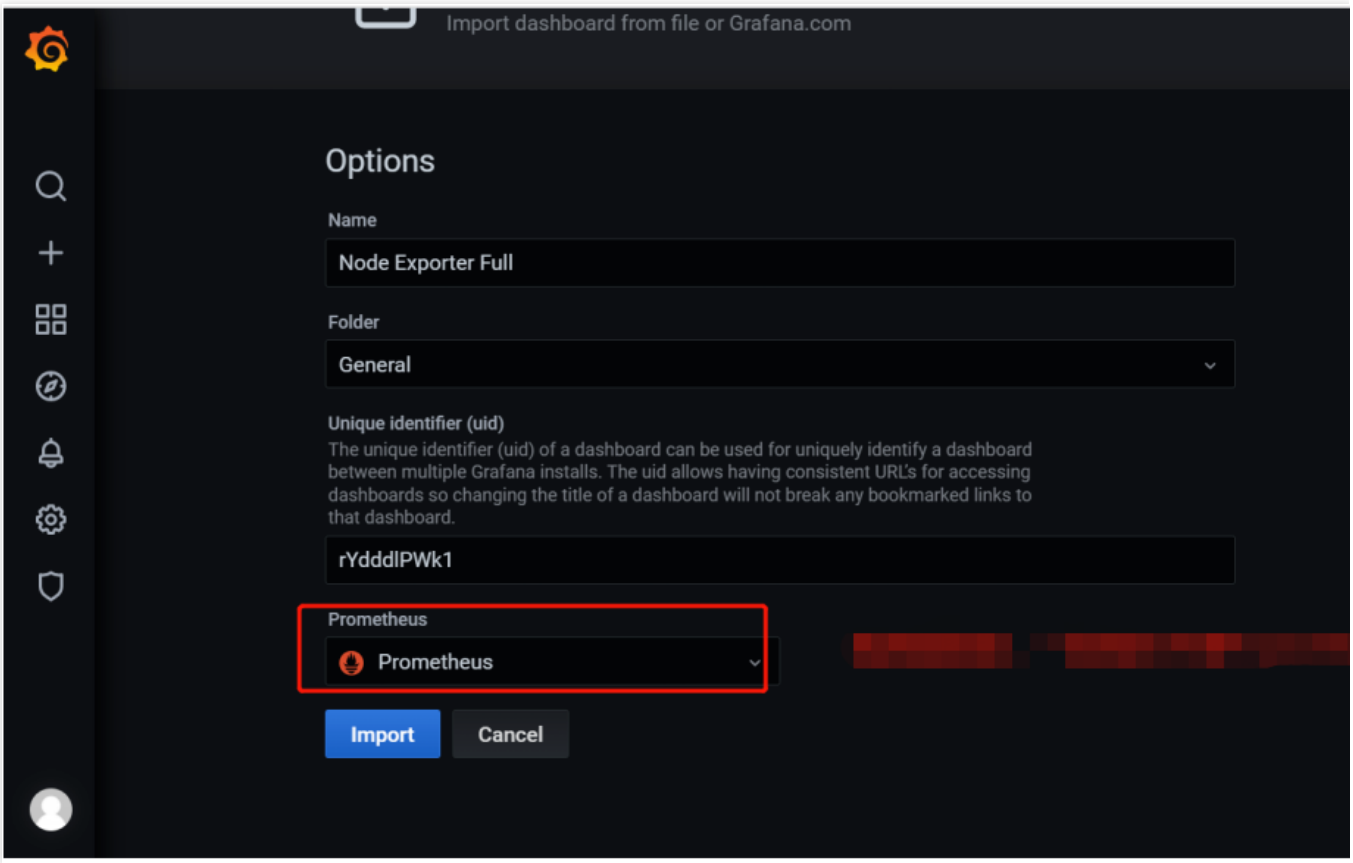


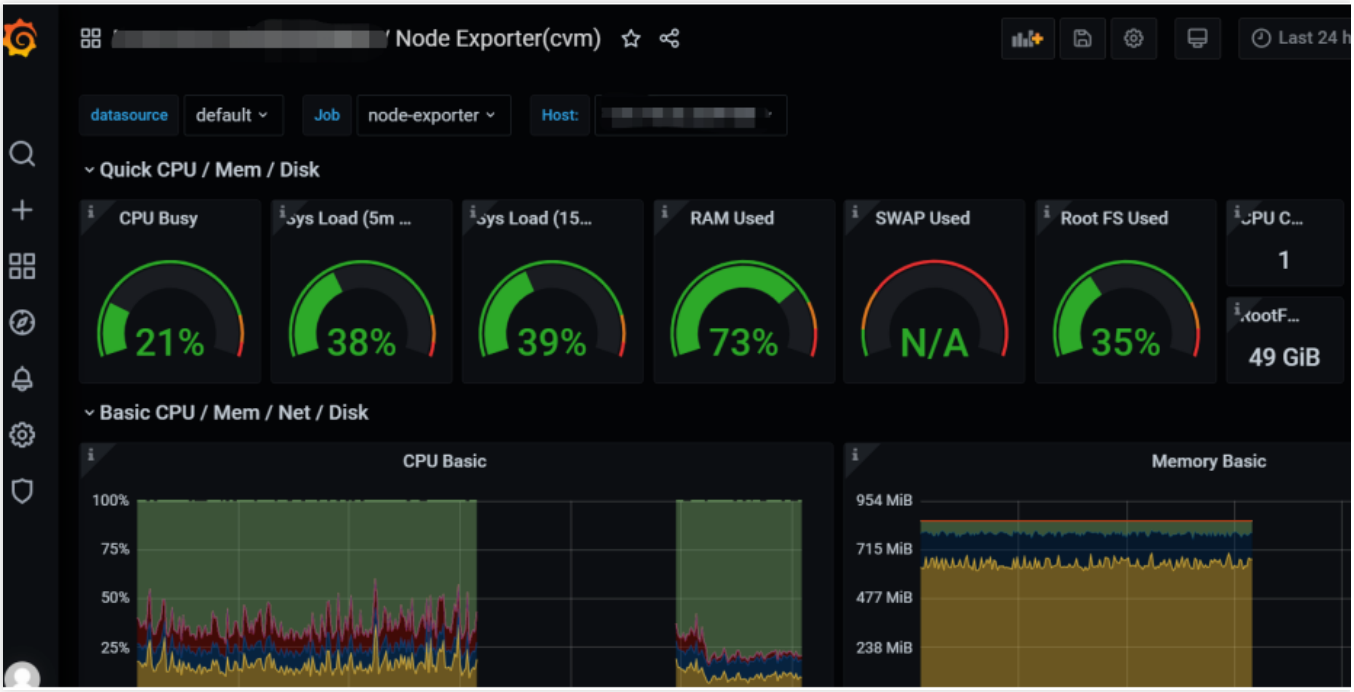
步骤5：配置 Dashboard

每个产品都会有一些现成的 json 文件，可以直接导入 Dashboard 。
1. **下载 Dashboard 文件**：登录 Dashboard 界面，单击搜索 node_exporter，选择最新的 Dashboard 并下载。



2. **导入 Dashboard 的 json 文件**：登录 Prometheus 监控服务控制台，进入**基本信息** > **Grafana 地址**，单击进入 Grafana，在 **Grafana 控制台** > **Create** > **Import** > 在 **Upload JSON file** 中上传 Dashboard 文件。




Node Exporter(cvm)
Last 24 h
datasource
default
Job
node-exporter
Host:
Quick CPU / Mem / Disk
CPU Busy
21%
38%
39%
RAM Used
73%
SWAP Used
N/A
Root FS Used
35%
1
49 GiB
Basic CPU / Mem / Net / Disk
CPU Basic
100%
75%
50%
25%
Memory Basic
954 MiB
715 MiB
477 MiB
238 MiB

健康巡检

最近更新时间：2024-01-29 15:55:14

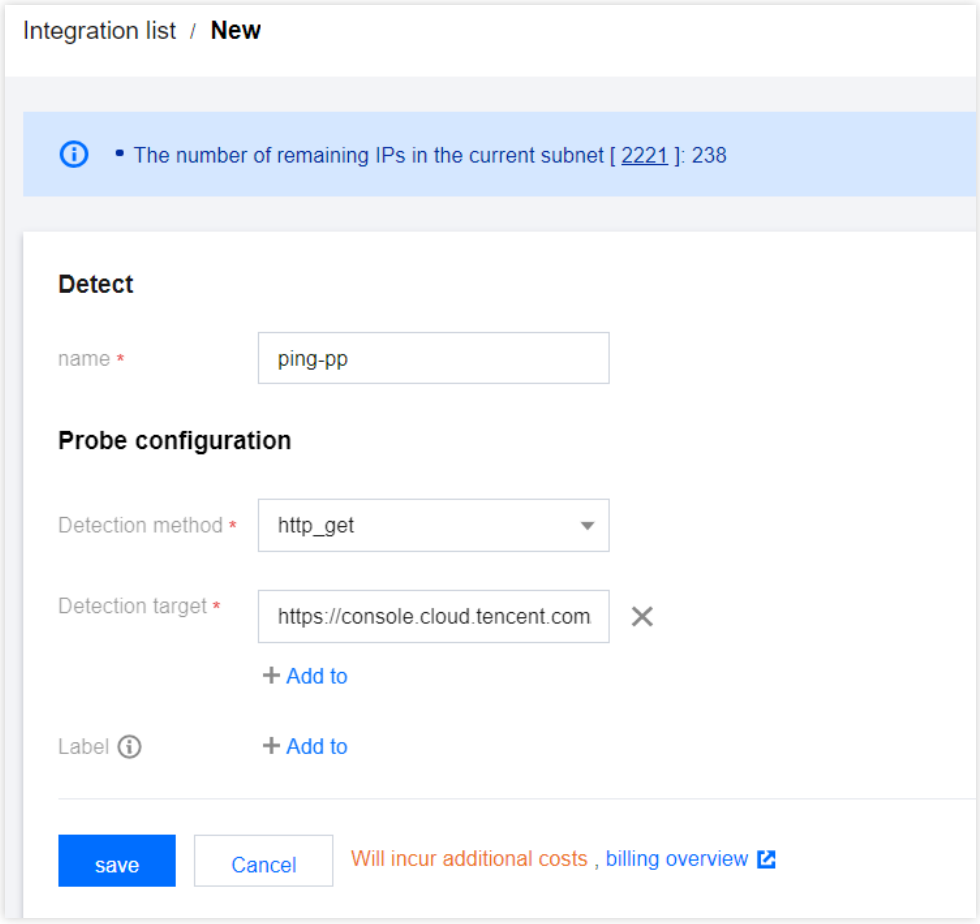
操作场景

通过定期探测对应服务的连通性，来检测其服务的健康情况，帮助您实时的掌握服务的健康状况，及时发现异常，来提升服务的 SLA。

操作步骤

1. 登录 Prometheus 控制台。
2. 在实例列表中，选择对应的 Prometheus 实例。
3. 进入实例详情页，单击**集成中心**。
4. 在集成中心选择 `健康巡检` 进行对应服务的探测配置。

探测说明



参数	说明
名称	每个探测任务需要一个唯一名称，对应 Grafana 监控大屏中的探测分组
探测方式	目前支持如下几种探测方式： http_get http_post tcp ssh ping
探测目标	被探测服务对应地址
标签	增加具有业务含义的标签，会自动添加到 Prometheus 的 Label 中

查看监控

可以通过监控大盘清晰看到如下状态：

1. 服务访问的延时，是否健康；
2. 服务访问各阶段处理的延时；
3. 如果是 HTTPS 可以监控证书的过期时间；

4. 以及各种探测类型的状态；

TKE 集群内安装组件说明

最近更新时间：2024-07-23 17:53:53

概述

本文介绍 Prometheus 监控服务在 集成容器服务 过程中在用户 TKE 集群内安装的各个组件的功能，使用权限和占用资源。

proxy-agent

组件介绍

由于 TKE 集群有独立的网络环境，proxy-agent 部署在集群内为集群外的采集组件提供访问代理。外部采集组件一方面通过 proxy-agent 服务发现集群内的资源，另一方面通过 proxy-agent 抓取指标并写到 Prometheus 实例的时序存储中。

部署在集群内的资源对象

Namespace	kubernetes 对象名称	类型	资源量	说明
<Prometheus 实例 ID>	proxy-agent	Deployment	0.25C256Mi*2	采集代理
<Prometheus 实例 ID>	<Prometheus 实例 ID>	ServiceAccount	-	权限载体
-	<Prometheus 实例 ID>	ClusterRole	-	采集权限相关
-	<Prometheus 实例 ID>-crb	ClusterRoleBinding	-	采集权限相关

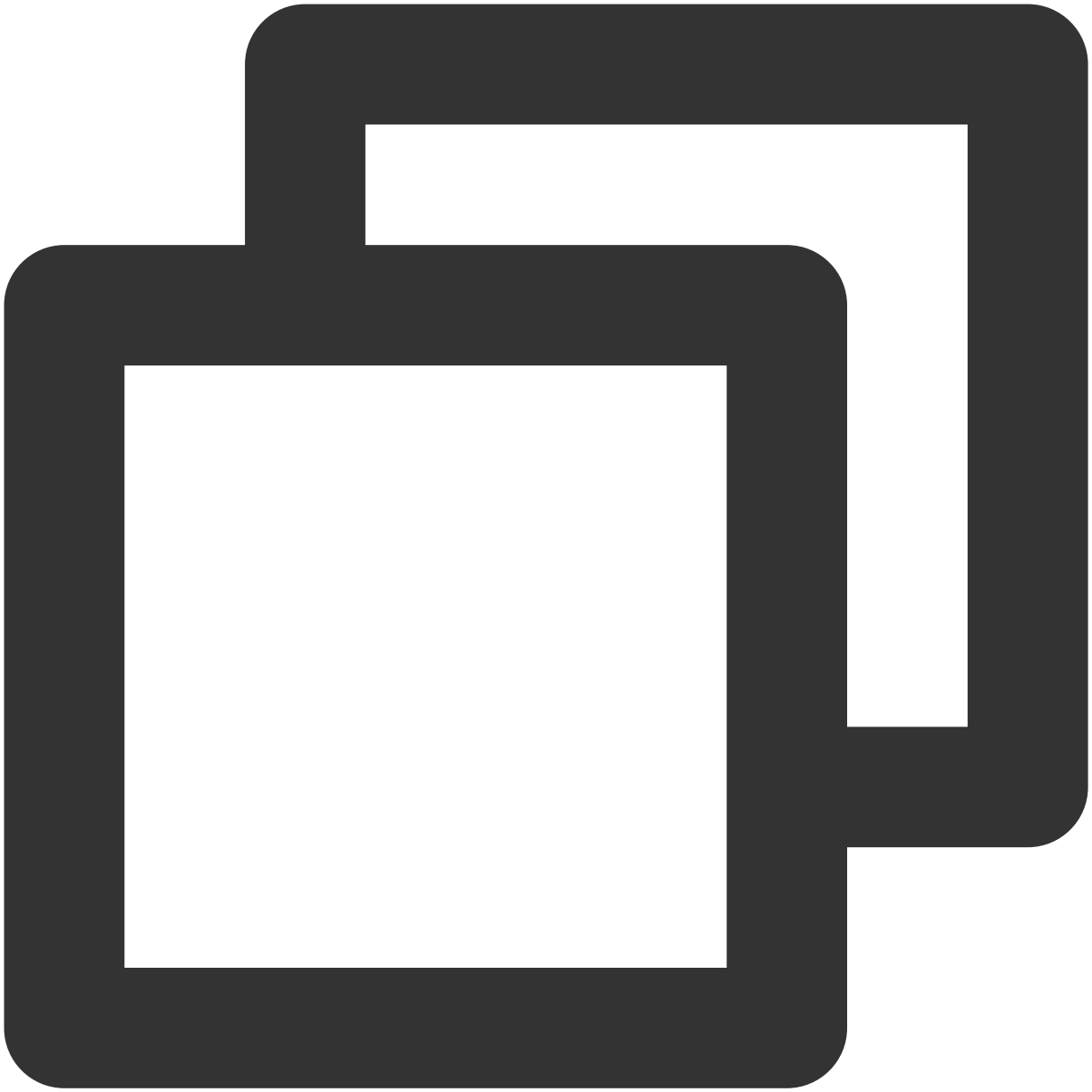
组件权限说明

权限场景

功能	涉及对象	涉及操作权限
采集配置管理	scrapeconfigs,servicemonitors,podmonitors,probes,configmaps,secrets,namespaces	get/list/watch

服务发现	services,endpoints,nodes,pods,ingresses	get/list/watch
部分系统组件指标抓取	nodes/metrics,nodes/proxy,pods/proxy	get/list/watch
带RBAC鉴权的指标抓取	/metrics,/metrics/cadvisor	get

权限定义



```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prom-instance
rules:
  - apiGroups:
      - monitoring.coreos.com
    resources:
      - scrapeconfigs
      - servicemonitors
      - podmonitors
```

```
      - probes
      - prometheuses
      - prometheusrules
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - namespaces
      - configmaps
      - secrets
      - nodes
      - services
      - endpoints
      - pods
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - networking.k8s.io
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
  - apiGroups: [ "" ]
    resources:
      - nodes/metrics
      - nodes/proxy
      - pods/proxy
    verbs:
      - get
      - list
      - watch
  - nonResourceURLs: [ "/metrics", "/metrics/cadvisor" ]
    verbs:
      - get
```

tke-kube-state-metrics

组件介绍

tke-kube-state-metrics 使用开源组件 kube-state-metrics，监听集群的 API server，生成集群内各种对象的状态指标。

部署在集群内的资源对象

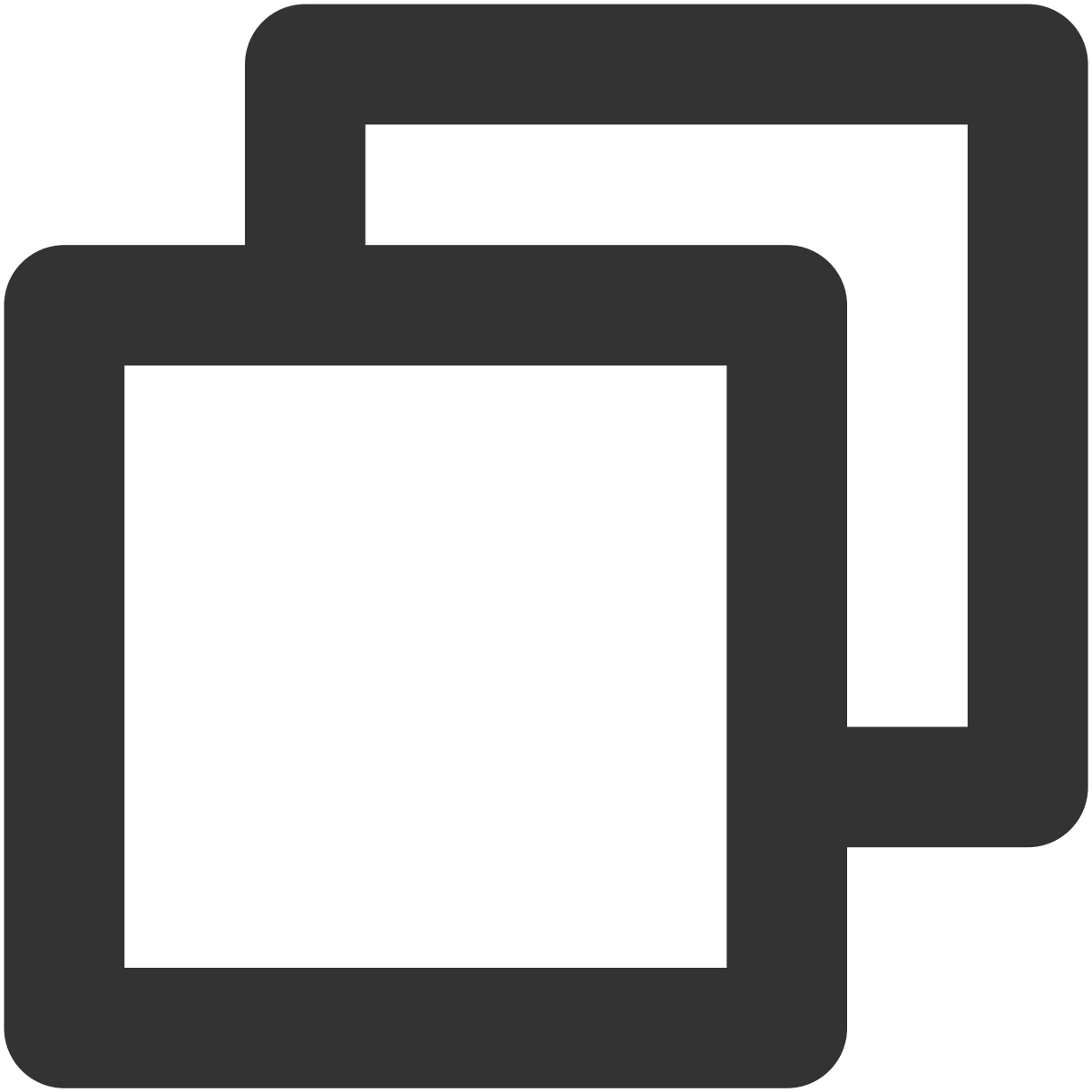
Namespace	kubernetes 对象名称	类型	资源量	说明
kube-system	tke-kube-state-metrics	Statefulset	0.5C512Mi	采集程序
kube-system	tke-kube-state-metrics	ServiceAccount	-	权限载体
-	tke-kube-state-metrics	ClusterRole	-	采集权限相关
-	tke-kube-state-metrics	ClusterRoleBinding	-	采集权限相关
kube-system	tke-kube-state-metrics	Service	-	采集程序对应服务，供服务发现使用
kube-system	tke-kube-state-metrics	ServiceMonitor	-	采集配置
kube-system	tke-kube-state-metrics	Role	-	分片采集权限相关
kube-system	tke-kube-state-metrics	RoleBinding	-	分片采集权限相关

组件权限说明

权限场景

功能	涉及对象	涉及操作权限
监听集群内各种资源的状态	绝大部分 Kubernetes 资源	list/watch
获取采集 Pod 所在分片序号	statefulsets,pods	get

权限定义



```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: tke-kube-state-metrics
rules:
  - apiGroups:
      - ""
    resources:
      - configmaps
      - secrets
      - nodes
```

```
      - pods
      - services
      - serviceaccounts
      - resourcequotas
      - replicationcontrollers
      - limitranges
      - persistentvolumeclaims
      - persistentvolumes
      - namespaces
      - endpoints
    verbs:
      - list
      - watch
  - apiGroups:
      - apps
    resources:
      - statefulsets
      - daemonsets
      - deployments
      - replicasets
    verbs:
      - list
      - watch
  - apiGroups:
      - batch
    resources:
      - cronjobs
      - jobs
    verbs:
      - list
      - watch
  - apiGroups:
      - autoscaling
    resources:
      - horizontalpodautoscalers
    verbs:
      - list
      - watch
  - apiGroups:
      - authentication.k8s.io
    resources:
      - tokenreviews
    verbs:
      - create
  - apiGroups:
      - authorization.k8s.io
    resources:
```

```
      - subjectaccessreviews
    verbs:
      - create
  - apiGroups:
      - policy
    resources:
      - poddisruptionbudgets
    verbs:
      - list
      - watch
  - apiGroups:
      - certificates.k8s.io
    resources:
      - certificatesigningrequests
    verbs:
      - list
      - watch
  - apiGroups:
      - storage.k8s.io
    resources:
      - storageclasses
      - volumeattachments
    verbs:
      - list
      - watch
  - apiGroups:
      - admissionregistration.k8s.io
    resources:
      - mutatingwebhookconfigurations
      - validatingwebhookconfigurations
    verbs:
      - list
      - watch
  - apiGroups:
      - networking.k8s.io
    resources:
      - networkpolicies
      - ingresses
    verbs:
      - list
      - watch
  - apiGroups:
      - coordination.k8s.io
    resources:
      - leases
    verbs:
      - list
```

```
      - watch
  - apiGroups:
      - rbac.authorization.k8s.io
    resources:
      - clusterrolebindings
      - clusterroles
      - rolebindings
      - roles
    verbs:
      - list
      - watch
---
kind: Role
metadata:
  name: tke-kube-state-metrics
  namespace: kube-system
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
  - apiGroups:
      - apps
    resourceNames:
      - tke-kube-state-metrics
    resources:
      - statefulsets
    verbs:
      - get
```

tke-node-exporter

组件介绍

tke-node-exporter 使用开源项目 node_exporter，部署在集群内的每个 Node 上，用来采集硬件和类Unix操作系统指标。

部署在集群内的资源

Namespace	kubernetes 对象名称	类型	资源量	说明

kube-system	tke-node-exporter	DaemonSet	0.1C180Mi*node 数量	采集程序
kube-system	tke-node-exporter	Service	-	采集程序对应服务，供服务发现使用
kube-system	tke-node-exporter	ServiceMonitor	-	采集配置

组件权限说明

该组件不使用任何集群权限。

云监控

最近更新时间：2024-01-29 15:55:14

操作场景

云监控模块集成腾讯云产品基础监控数据，通过 Prometheus 监控进行统一采集、存储和可视化。

操作步骤

1. 登录 Prometheus 控制台。
2. 在实例列表中，选择对应的 Prometheus 实例。
3. 进入实例详情页，单击 **集成中心**。
4. 在集成中心选择 **云监控**。定义集成名称、进行 Exporter 配置和选择对应的云产品。

说明：

时间偏移量：选取时间范围，结束时间=当前时间-时间偏移量。

集成的监控数据包含云产品的标签数据，若标签中包含中文或特殊字符将会被过滤掉。

该模块数据采集频率为：1分钟。

通过 Remote Read 读取云托管 Prometheus 实例数据

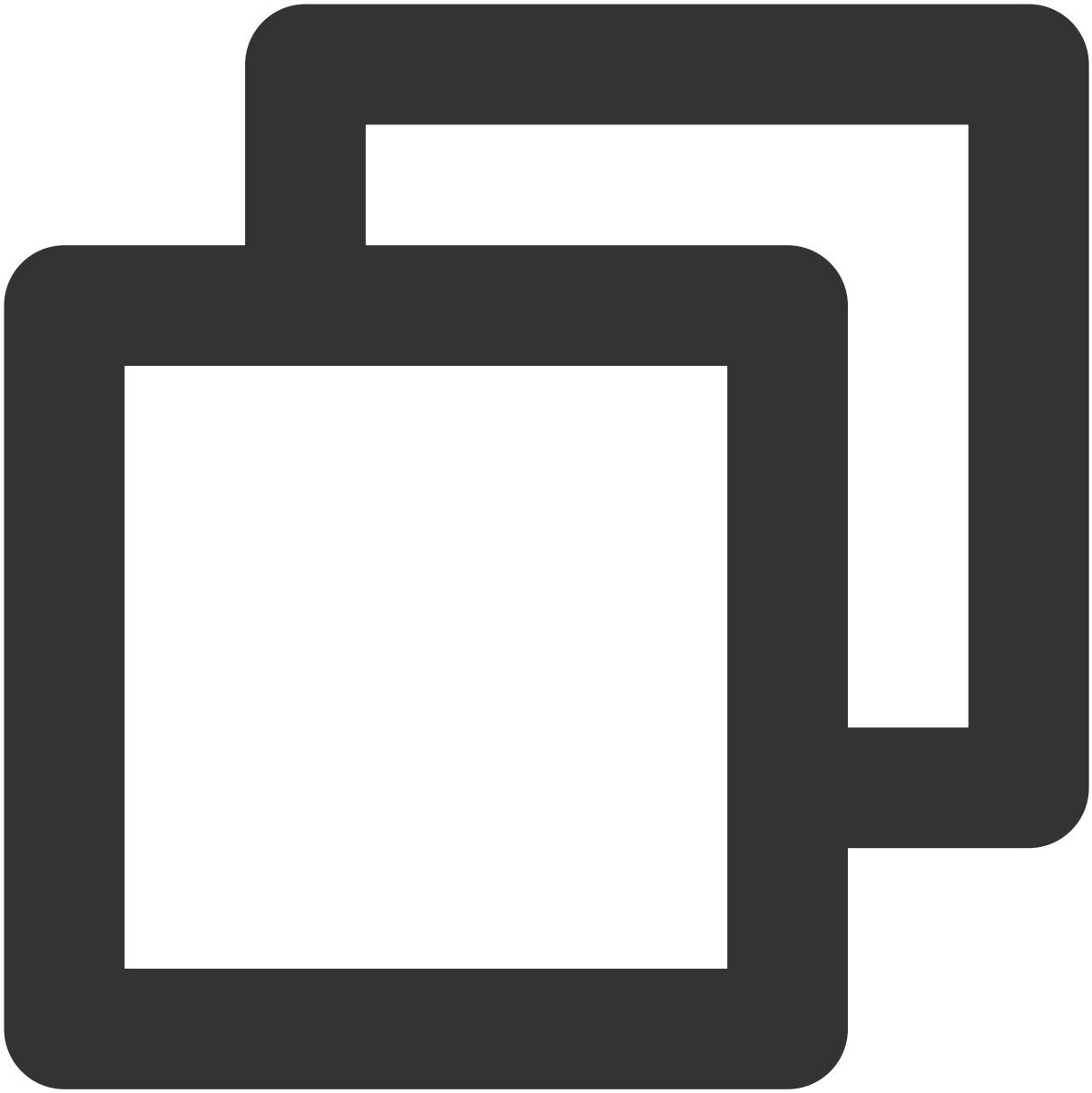
最近更新时间：2024-01-29 15:55:14

操作场景

Prometheus 提供了 Remote read 接口，该接口支持将一系列 Prometheus 协议的数据源组织为单一数据源查询。本文介绍如何使用自建 Prometheus 通过 Remote read 读取云托管 Prometheus 实例的数据。

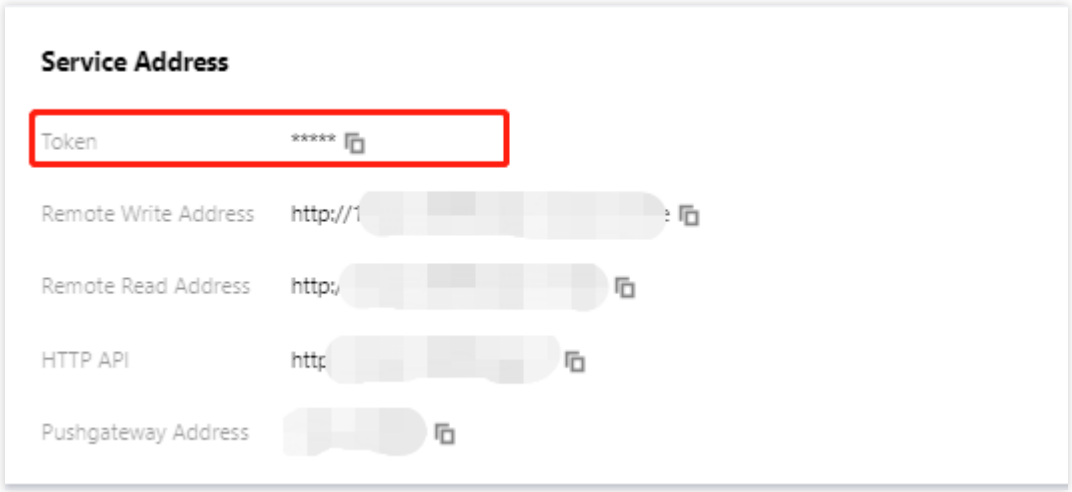
Remote Read 配置

推荐配置 `prometheus.yml` 如下：



```
remote_read:
  - url: 'http://prom_ip:prom_port/api/v1/read'
    read_recent: true
    basic_auth:
      username: app_id
      password: token
```

推荐使用 Basic Auth 方式访问云托管 Prometheus 实例，username 为账号 AppID ， password 为 Prometheus 控制台 > **基本信息** > **服务地址**中获取的 Token。



注意事项

配置 Remote read 的 Prometheus 需谨慎配置 `global:external_labels` ：

external_labels 会被附加在 Remote read 的查询条件中，不正确的 label 可能导致查询不到需要的数据。

`filter_external_labels: false` 配置项可以避免将 external_labels 加入查询条件（v2.34 版本以上支持）。

避免出现相同的 series：

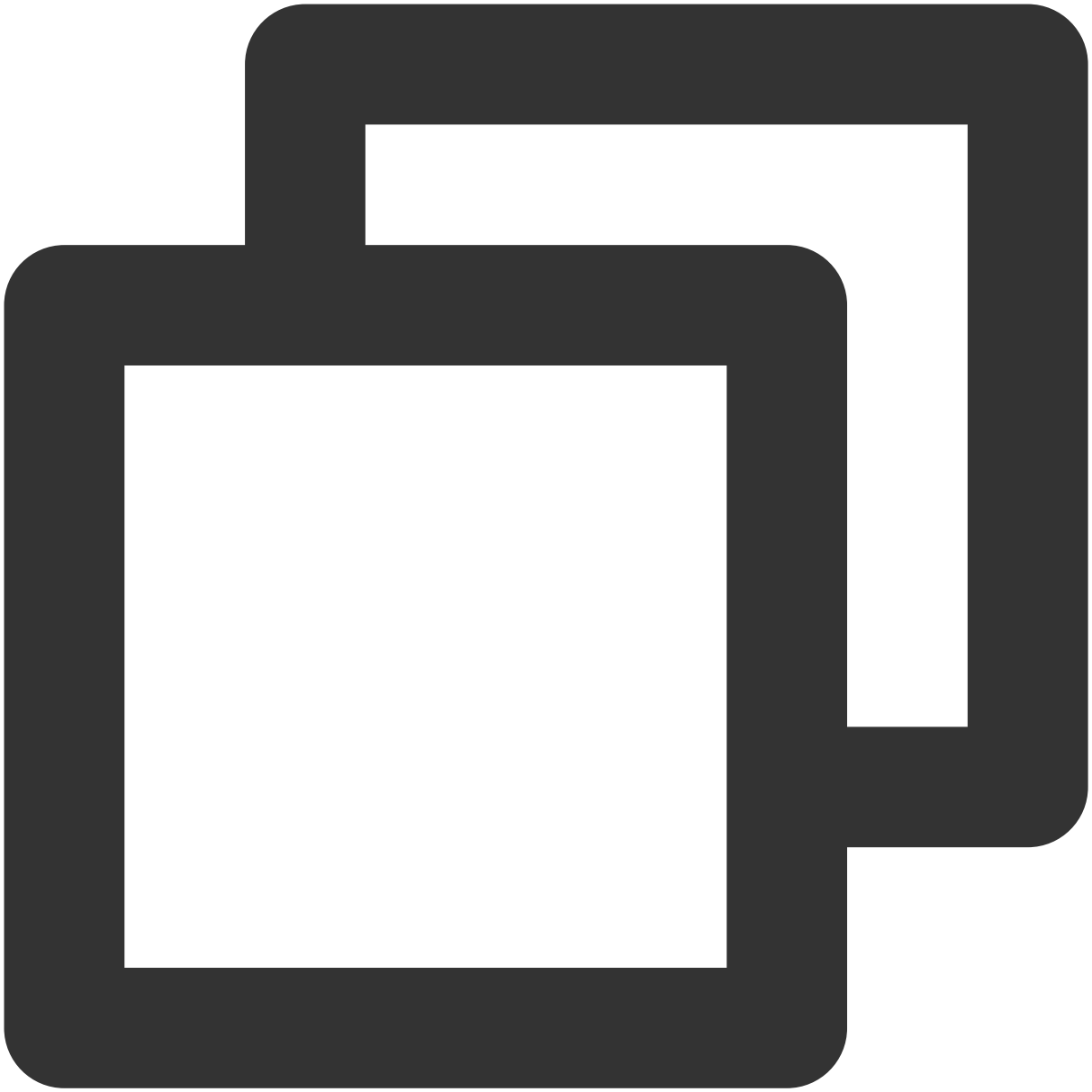
对于完全相同的两个 series，Prometheus 会在查询合并时在每个时间点在随机一个 series 取值组成新的 series 作为查询结果，这会导致查询结果不准确。

在 Prometheus 的设计理念中不存在多副本冗余存储的情况，所以不会对这种场景提供支持。

Remote read 完整配置项

说明：

`[]` 中的配置项为可选项（本文展示 Prometheus:v2.40 版本配置，低版本可能缺少部分配置项，详见 prometheus 官方文档）



```
# remote read 目标 prometheus 实例的 api 地址
url: <string>

# 标识一个唯一的 remote read 配置名称
[ name: <string> ]

# 查询 promql 中必须包含以下 label 过滤条件才会进行 remote read 查询
required_matchers:
  [ <labelname>: <labelvalue> ... ]

# remote read 查询超时时间
```

```
[ remote_timeout: <duration> | default = 1m ]

# 自定义 remote read 请求中附带的 headers，无法覆盖 prometheus 原本添加的 headers
headers:
  [ <string>: <string> ... ]

# 在本地有完整数据存储的时间范围是否进行 remote read 查询
[ read_recent: <boolean> | default = false ]

# 为每个 remote read 请求添加 Authorization header，password password_file 二选一
basic_auth:
  [ username: <string> ]
  [ password: <secret> ]
  [ password_file: <string> ]

# 自定义 Authorization header 配置
authorization:
  # 认证类型
  [ type: <string> | default: Bearer ]
  # 认证密钥，credentials credentials_file 二选一
  [ credentials: <secret> ]
  # 密钥从文件中获取
  [ credentials_file: <filename> ]

# OAuth2.0认证，不能与 basic_auth authorization 同时使用
oauth2:
  [ <oauth2> ]

# TLS 配置
tls_config:
  [ <tls_config> ]

# 代理 URL
[ proxy_url: <string> ]

# 查询请求是否接受3XX 跳转
[ follow_redirects: <boolean> | default = true ]

# 是否启用 HTTP2
[ enable_http2: <bool> | default: true ]

# 是否在 remote read 时附加 external_labels
[ filter_external_labels: <boolean> | default = true ]
```

Agent 自助接入

最近更新时间：2024-08-15 17:08:56

使用场景

采集自建 IDC 上的服务，需要部署 Agent 并管理采集配置，上报监控数据到云上 Prometheus 监控服务。对于云上服务推荐使用 集成中心，集成中心会托管 Agent，提供多种中间件和抓取任务的自动化集成。

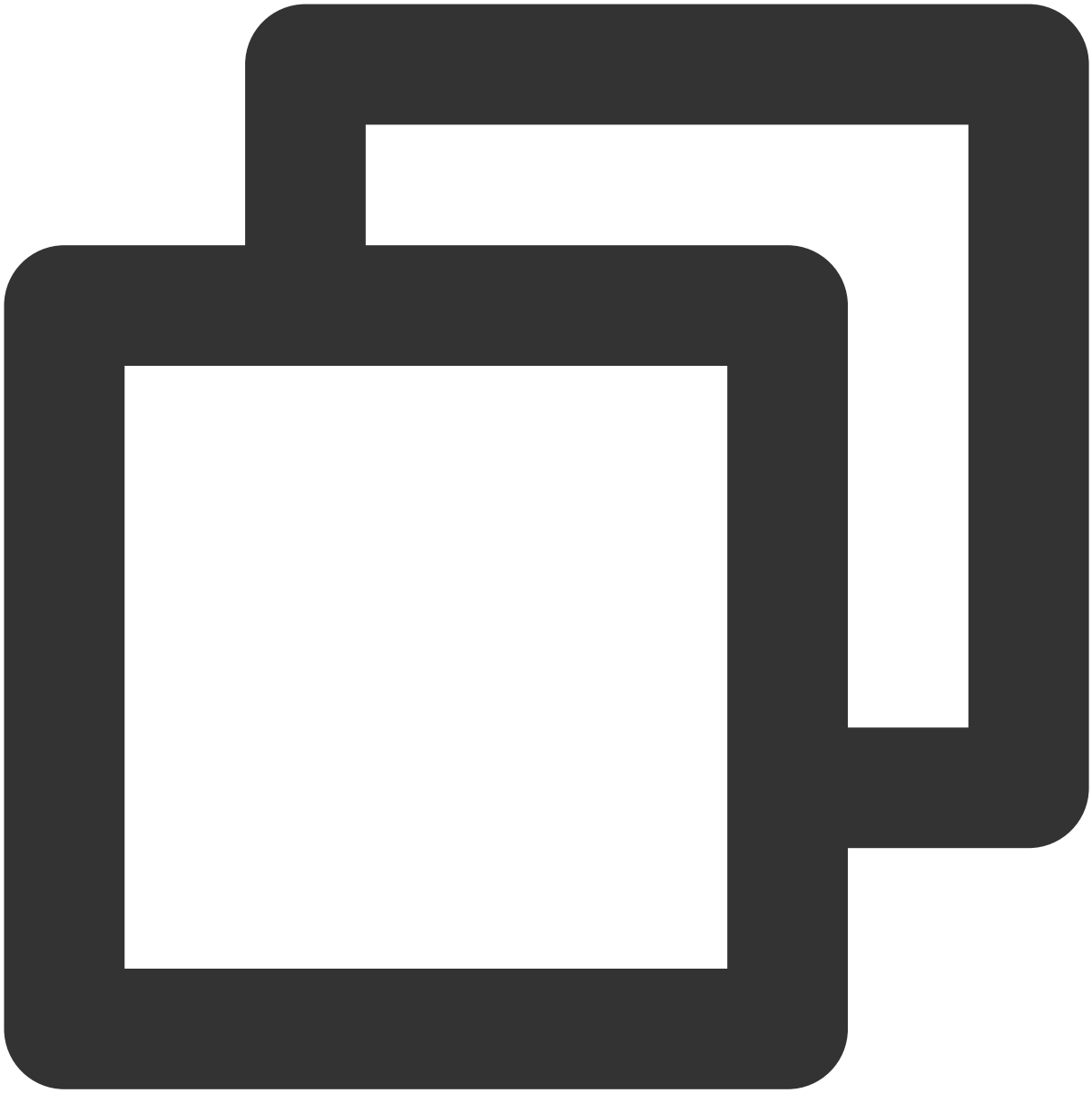
获取 Prometheus 实例访问配置

1. 前往 Prometheus 监控控制台，选择对应的实例 ID/名称，在**基本信息 > 服务地址**页面，获取 Remote Write 地址和 Token。

2. 在 账号信息 页面获取 APPID。

确认所在网络环境和云上实例联通

根据获取的 RemoteWrite 地址，执行如下命令，如果网络联通返回信息会包含 `401 Unauthorized` 。



```
curl -v -X POST  ${RemoteWriteURL}
```

安装并启动 vmagent

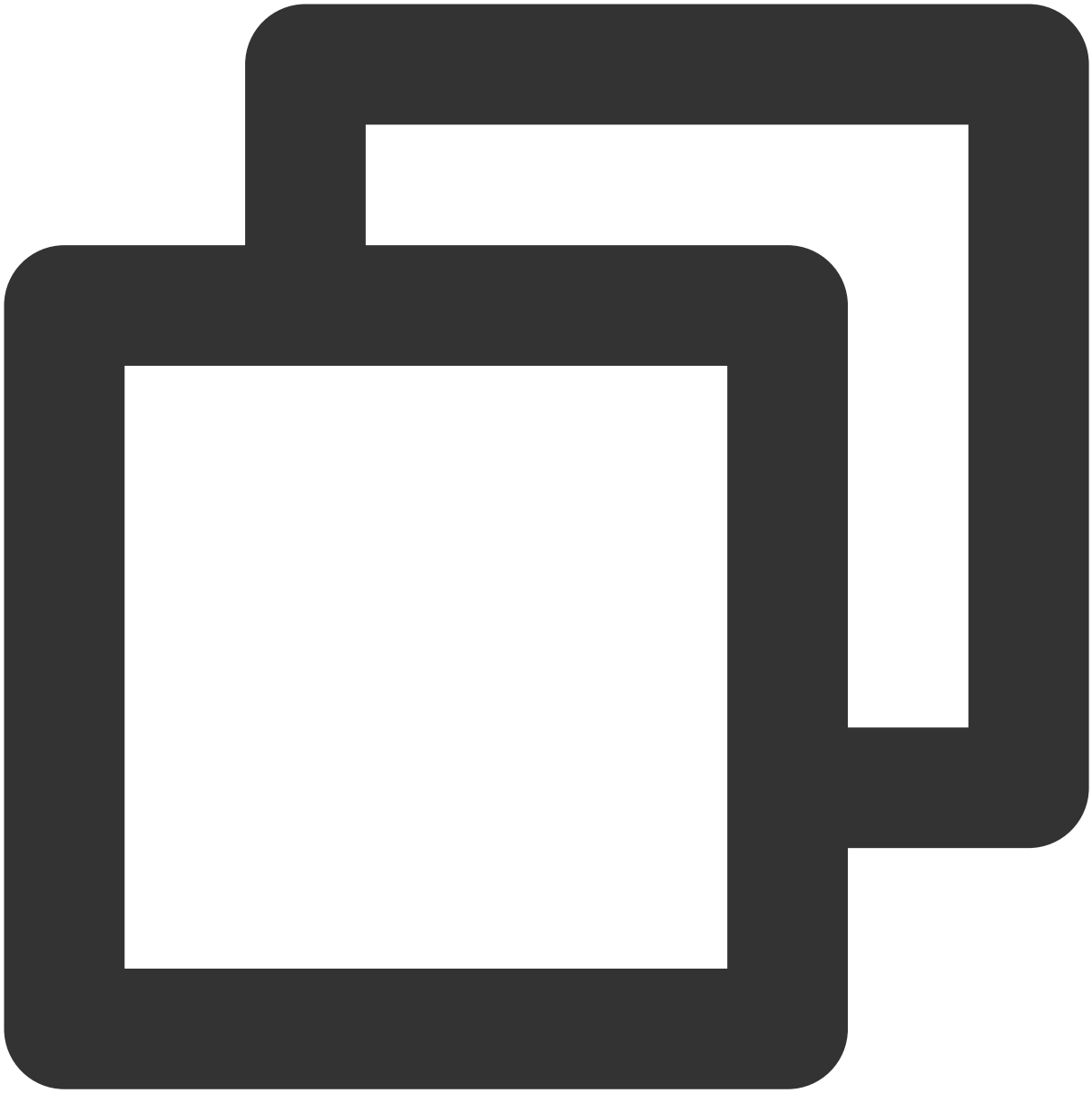
vmagent 占用较少的资源且兼容 Prometheus 采集配置和 Remote Write 协议而得到广泛使用。本文只介绍 vmagent 常用启动选项，通过 Systemd 或 Docker 来管理 vmagent，更多详细信息可以参考 官方文档。

常用启动选项

-promscrape.noStaleMarkers：如果采集目标消失，默认会生成所有关联指标的 stale marker 并写到远程存储。设置该选项禁止该行为，能减少内存的占用。
-loggerTimezone：日志中时间的时区，例如 `Asia/Shanghai, Europe/Berlin` 或者 `Local` (默认是 "UTC")。
-remoteWrite.tmpDataPath：采集后待写出数据临时存储的文件路径。
-remoteWrite.url：数据写向远程存储的 URL。
-remoteWrite.basicAuth.username：远程存储 -remoteWrite.url 对应的 basic auth 用户名。
-remoteWrite.basicAuth.password：远程存储 -remoteWrite.url 对应的 basic auth 密码。
-promscrape.config：采集配置的路径，可以是文件路径或者 HTTP URL，更多详情 参考文档。
-promscrape.configCheckInterval：检查 -promscrape.config 配置变化的时间间隔，关于配置更新可以 参考文档。

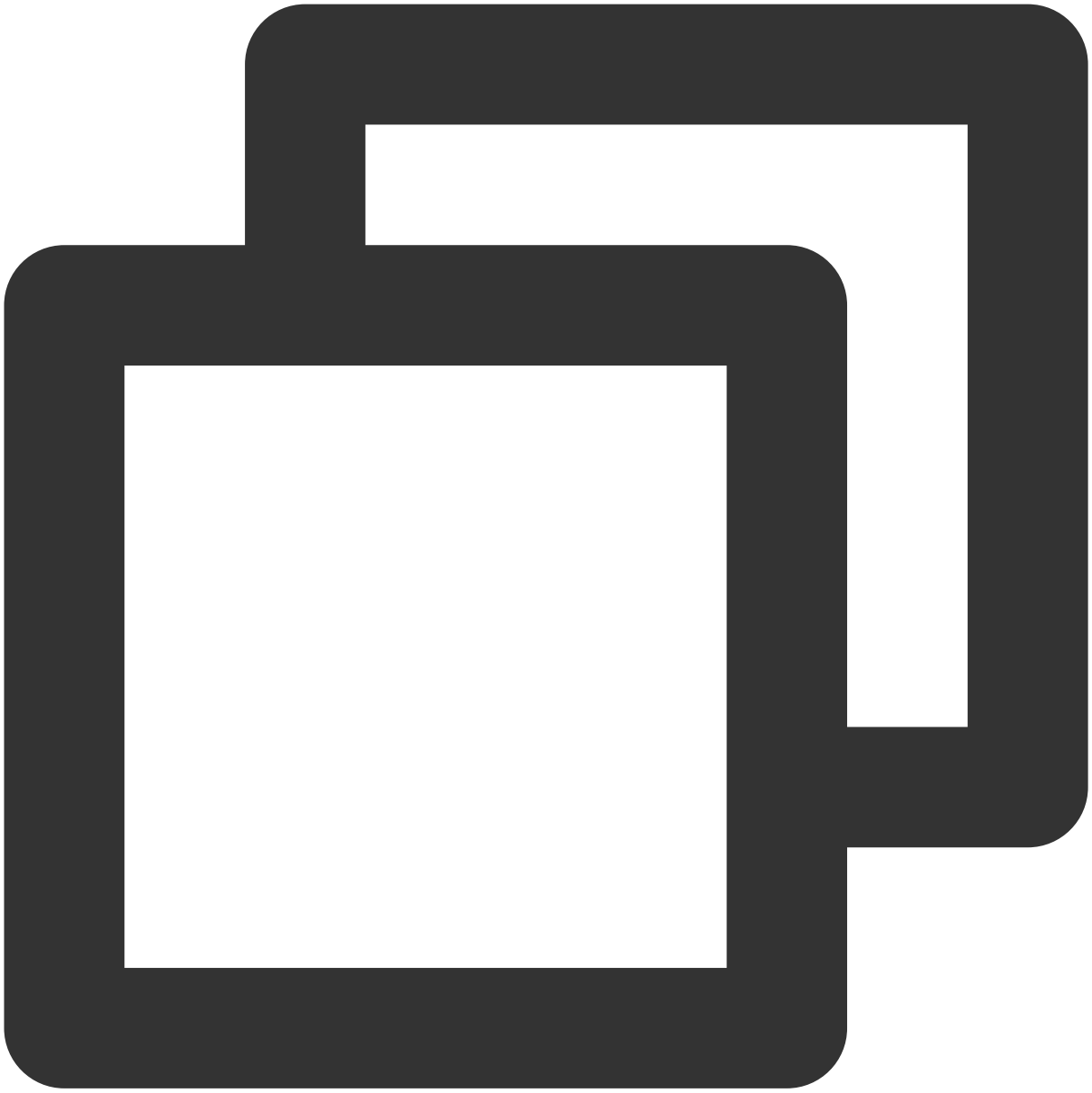
通过 Docker 管理

1. 在 vmagent 发布页面 选择镜像版本，推荐使用 latest。
2. 替换脚本中的 Prometheus 实例信息，启动 vmagent。



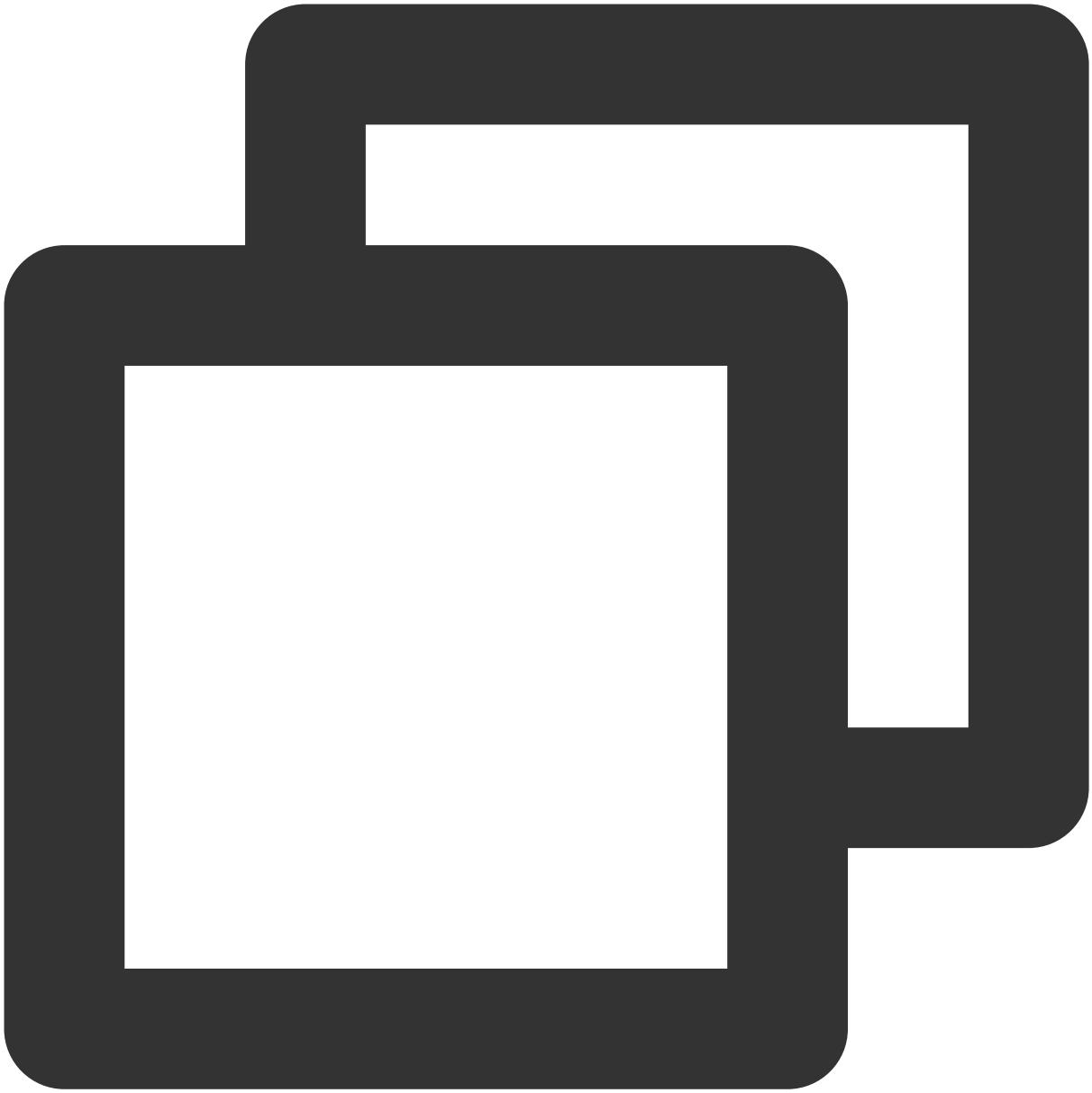
```
mkdir /etc/prometheus
touch /etc/prometheus/scrape-config.yaml
docker run -d --name vmagent --restart always --net host -v /etc/prometheus:/etc/pr
-promscrape.noStaleMarkers \\
-loggerTimezone=Local \\
-remoteWrite.url="${RemoteWriteURL}" \\
-remoteWrite.basicAuth.username="${APPID}" \\
-remoteWrite.basicAuth.password='${Token}' \\
-remoteWrite.tmpDataPath=/var/lib/vmagent  \\
-promscrape.config=/etc/prometheus/scrape-config.yaml \\
-promscrape.configCheckInterval=5s
```

3. 查看 vmagent 日志



```
docker ps
docker logs vmagent
```

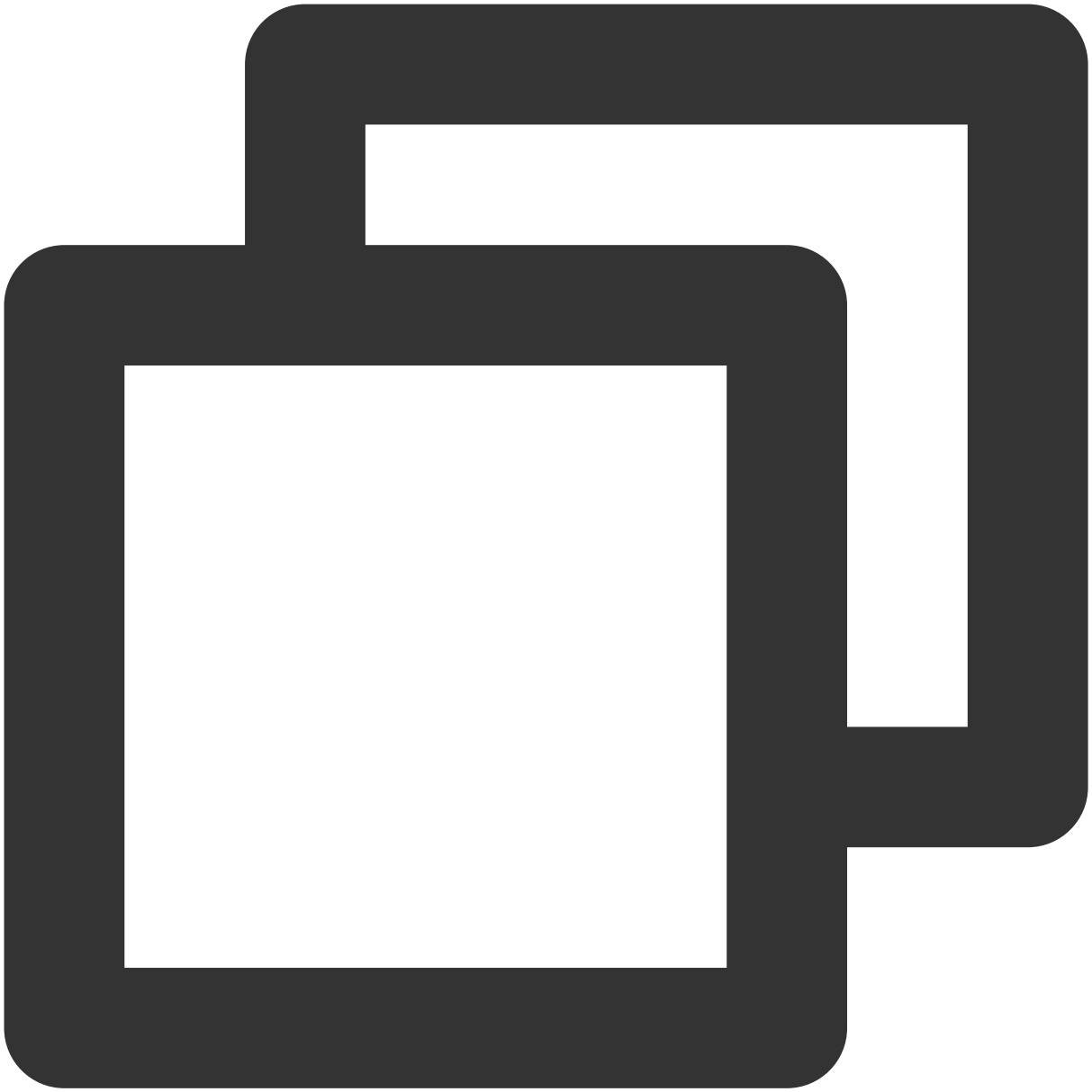
如果正常启动，执行如下命令会返回 `OK` 。



```
curl localhost:8429/health
```

通过 Systemd 管理

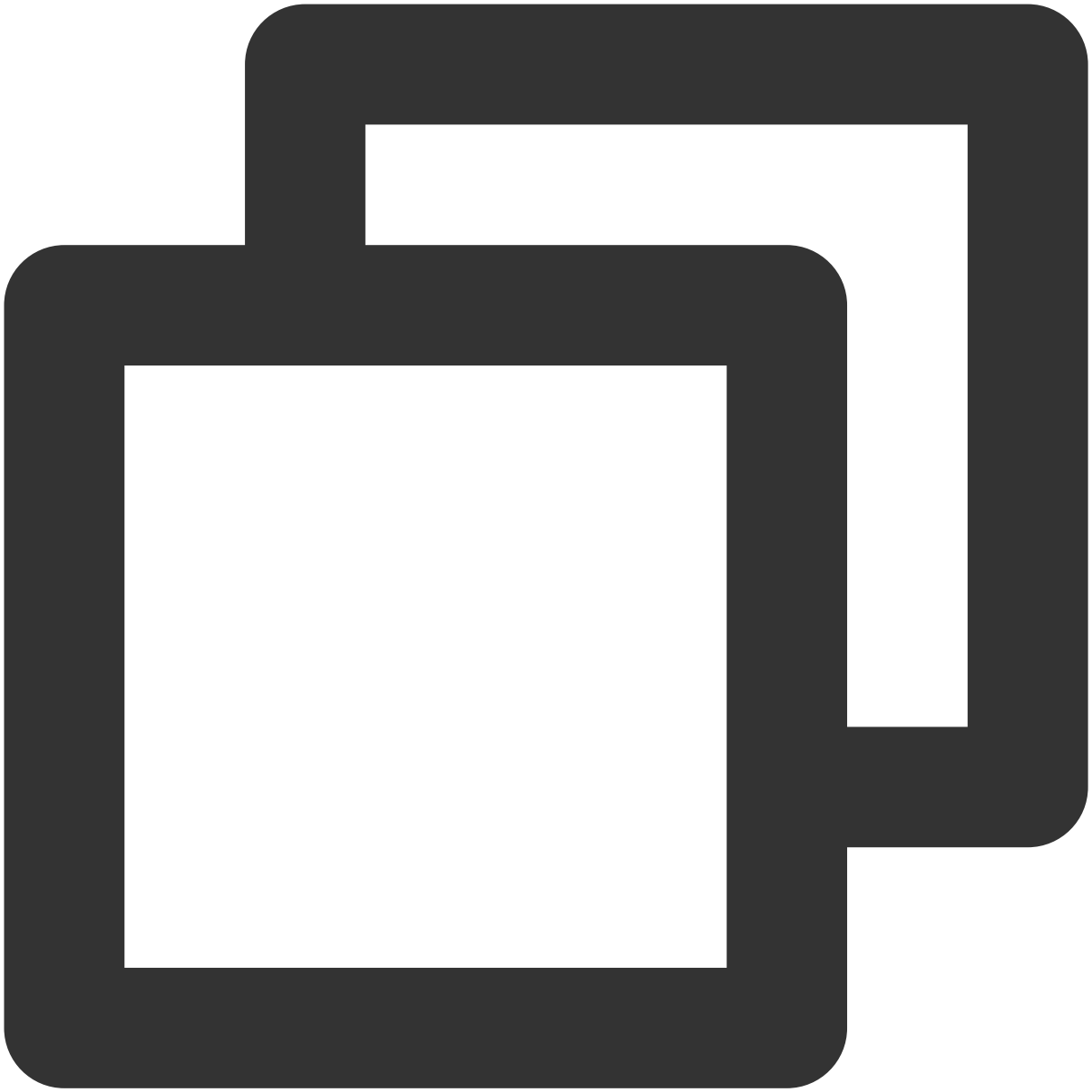
1. 在 vmagent 发布页面，根据操作系统和 CPU 架构，下载对应 vmutils-* 压缩包并解压。
2. 替换脚本中的 Prometheus 实例访问信息，启动 vmagent。



```
mkdir /etc/prometheus
touch /etc/prometheus/scrape-config.yaml
cat >/usr/lib/systemd/system/vmagent.service <<EOF
[Unit]
Description=VictoriaMetrics Agent
After=network.target

[Service]
LimitNOFILE=10240
ExecStart=/usr/bin/vmagent \\
-promscrape.noStaleMarkers \\
```

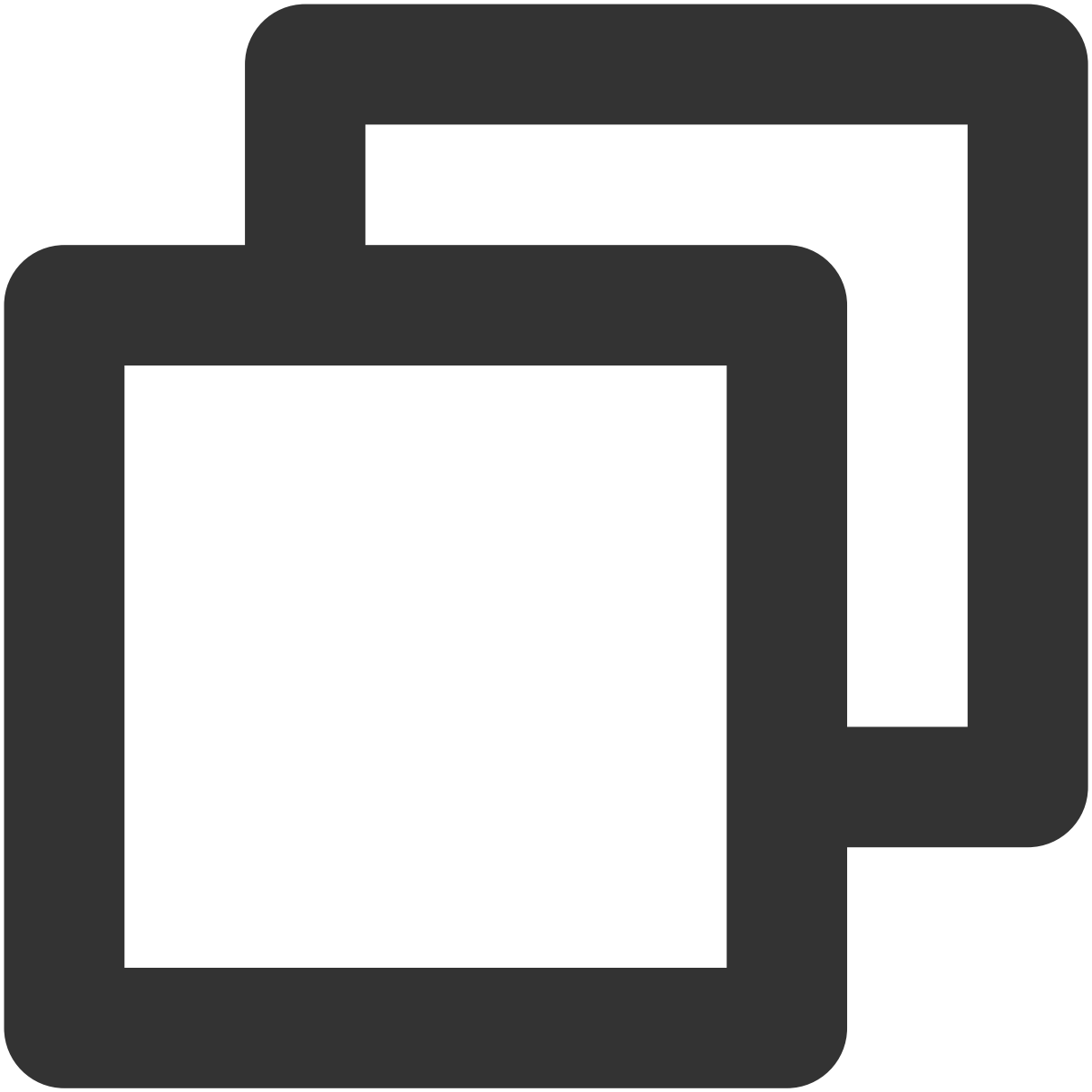
```
-loggerTimezone=Local   \\
-remoteWrite.url="${RemoteWriteURL}" \\
-remoteWrite.basicAuth.username="${APPID}" \\
-remoteWrite.basicAuth.password="${Token}" \\
-remoteWrite.tmpDataPath=/var/lib/vmagent \\
-promscrape.config=/etc/prometheus/scrape-config.yaml \\
-promscrape.configCheckInterval=5s
Restart=always
RestartSec=10s

[Install]
WantedBy=multi-user.target
EOF
systemctl daemon-reload
systemctl enable vmagent
systemctl start vmagent
sleep 3
systemctl status vmagent
```

3. 查看日志



```
journalctl -u vmagent
```

如果正常启动，执行如下命令会返回 `OK` 。

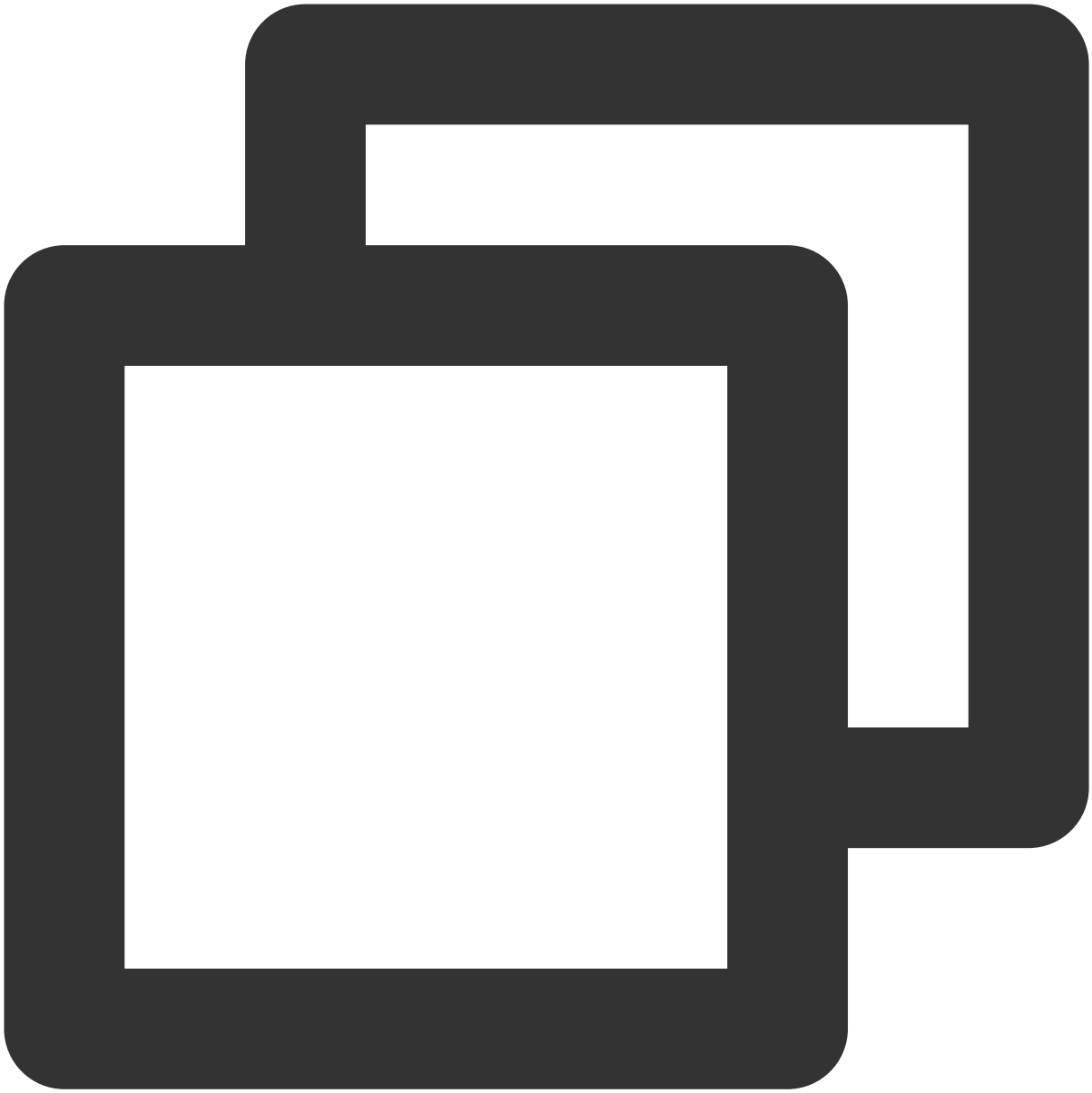


```
curl localhost:8429/health
```

管理配置

修改配置文件

编辑采集配置文件 `/etc/prometheus/scrape-config.yaml` 添加/更新/删除采集任务，关于 Prometheus 采集任务配置可以参考 官方文档。

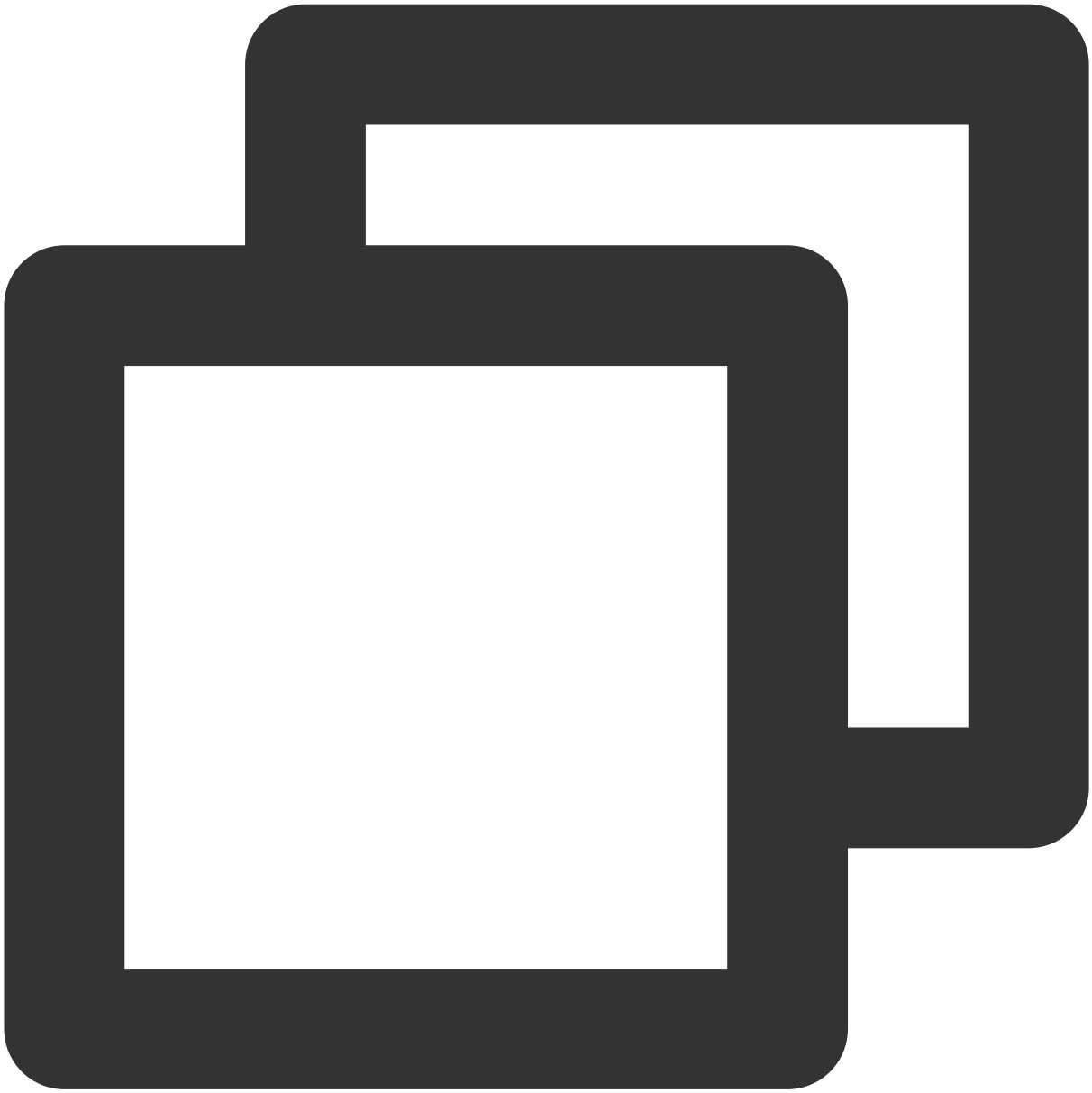


```
global:
   scrape_interval: 30s
scrape_configs:
   - job_name: agent-monitor
     static_configs:
        - targets:
             - localhost:8429
```

修改配置后，要过选项 `-promscrape.configCheckInterval` 设置时间才会生效。

查看监控目标信息

执行如下命令，查看采集目标，可以查看配置是否生效并符合预期。



```
curl localhost:8429/api/v1/targets
```

安全组开放说明

最近更新时间：2024-08-15 17:57:53

概述

本文介绍 Prometheus 监控服务 在 集成容器服务 过程中，托管集群和用户集群安全组需要开放的端口说明。同时介绍绑定出现安全组相关问题时的解决方式。

托管集群

托管集群安全组由 Prometheus 监控服务创建，一般情况下不需要修改。

安全组

规则	协议端口	策略
入站规则	TCP:9093,9090,10901,10902,9990,3000,8080,8008	允许
入站规则	TCP:8100-8200	允许
出站规则	ALL	允许

端口说明

端口	作用	备注
TCP:8008	proxy-server 监听 proxy-agent 连接端口	-
TCP:8080	集群内部接口调用端口	-
TCP:3000	grafana 代理端口	-
TCP:9990	cm-notify 同步端口	即将下线
TCP:10901,10902	thanos sidecar 监听地址	-
TCP:9090	配置 reload 端口，采集数据查询接口	-
TCP:9093	告警用端口	-

TCP:8100-8200	proxy-server 监听采集用端口	由于采集端口范围为100个，所以关联集群最多不能超过100个

查看方式

登录 Prometheus 监控，选择实例的 ID/名称 > **实例诊断**，采集诊断选择**集成中心**，在**采集架构图**中就可以看到托管集群安全组，点击安全组可通过超链接跳转至安全组界面，即可查看托管集群安全组。

用户集群

用户集群安全组在用户创建节点时指定，若不指定则为默认安全组。

安全组

规则	集群类型	协议端口	策略	说明
出站规则	-	TCP:8008	允许	保证 proxy-agent 与 proxy-server 能够建立连接

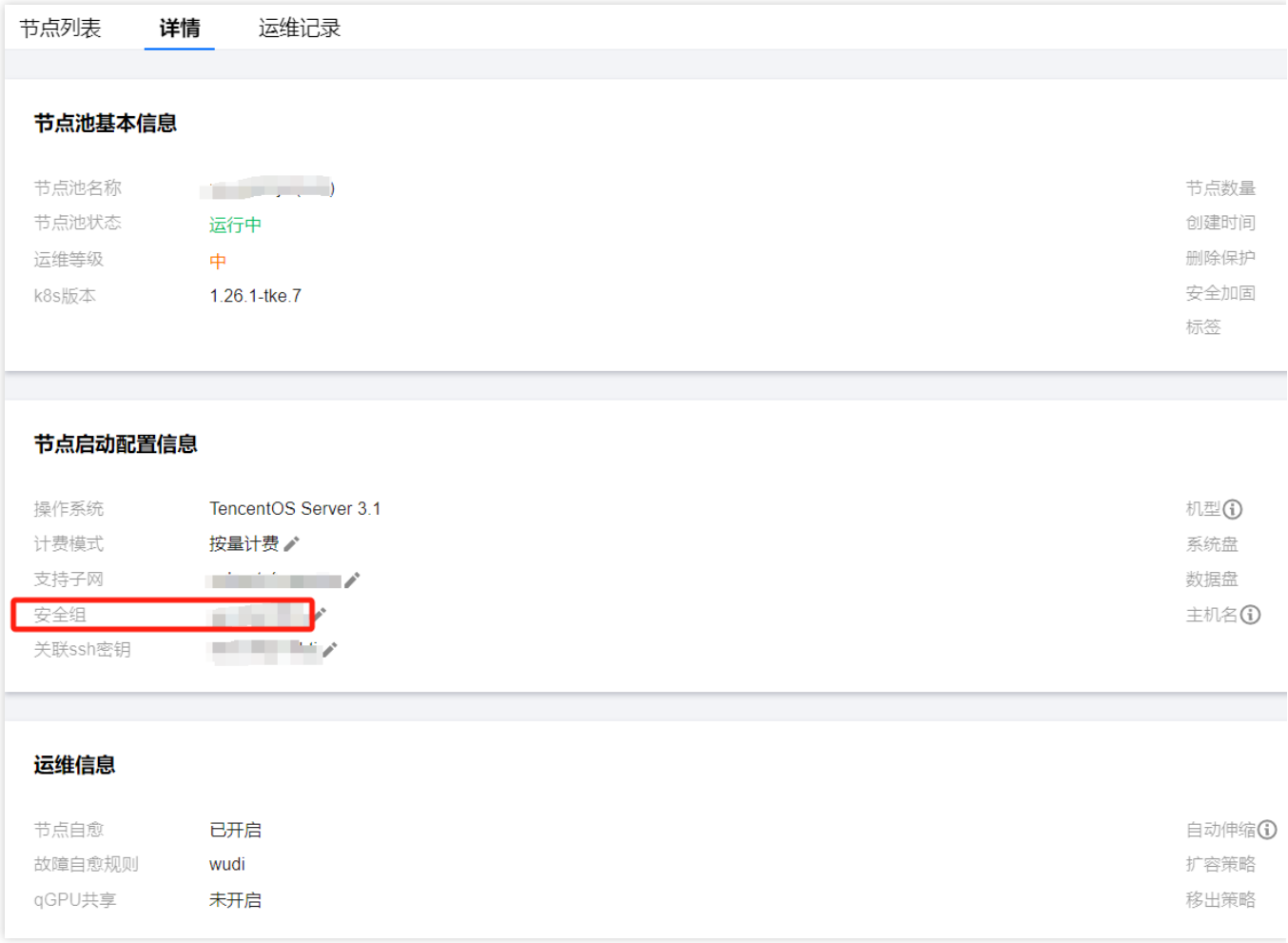
入站规则	标准集群	-		标准集群无需开放端口
入站规则	独立集群	TCP:9092,8180,443,10249,9100,60002,10252,10257,10259,10251等	允许	独立集群需额外开放 master 节点相关端口，保证 proxy-agent 能够拉取 master 节点相关监控数据

查看方式

登录 Prometheus 监控，选择实例的 ID/名称 > **数据采集**，点击集群 ID/名称即可跳转到该集群容器服务界面。

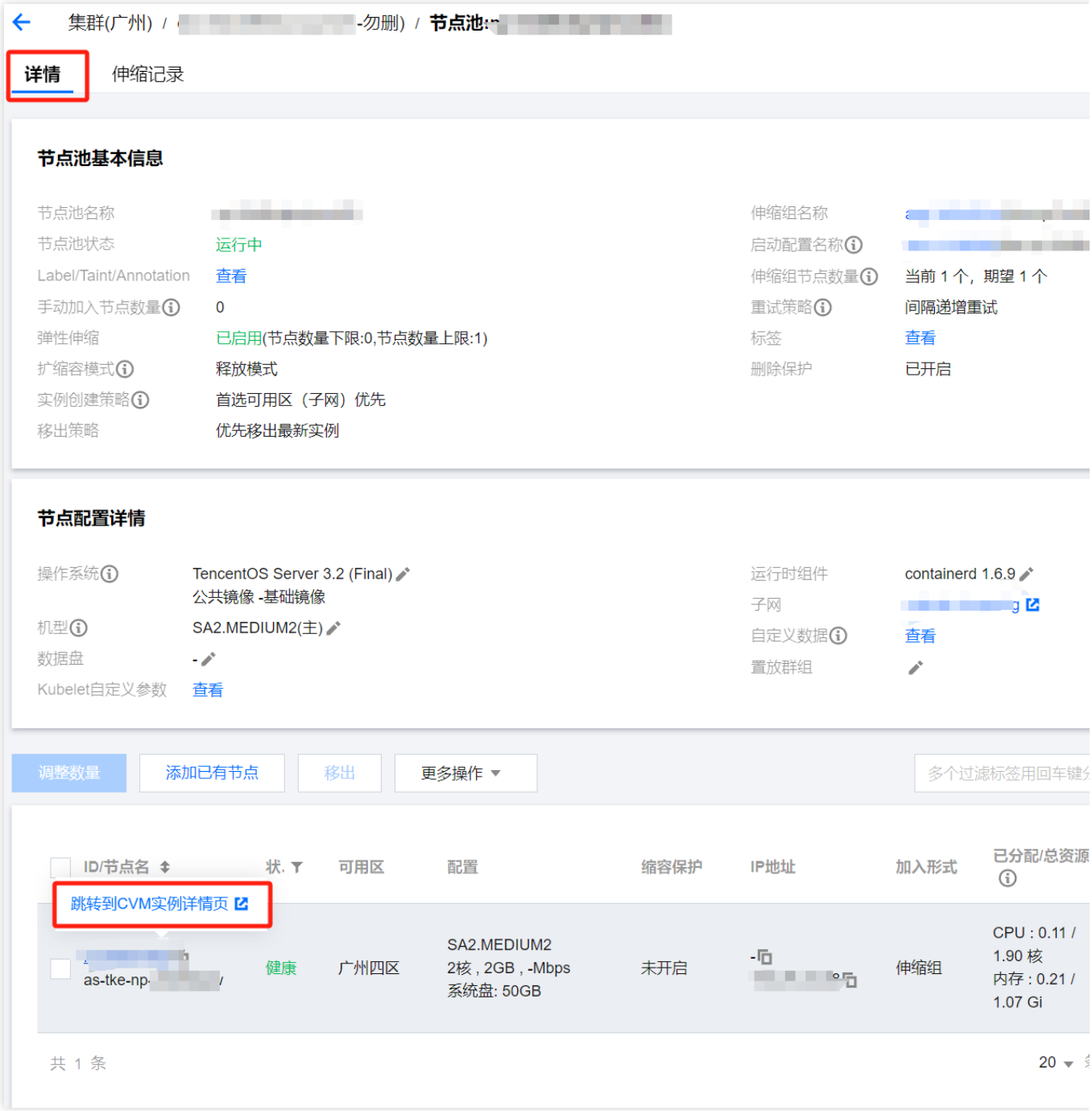
原生节点

点击**节点管理** > **Worker 节点** > **节点池**，点击节点池 id，在**详情**页即可看到安全组，在**腾讯云安全组**按安全组 id 进行搜索，查看具体规则即可。



普通节点

点击**节点管理 > Worker 节点 > 节点池**，点击节点池 id，在**详情**页将鼠标放在节点 ID 上点击**跳转到 CVM 实例详情页**：



跳转至实例详情页面后，点击**安全组**，即可查看具体安全组信息：

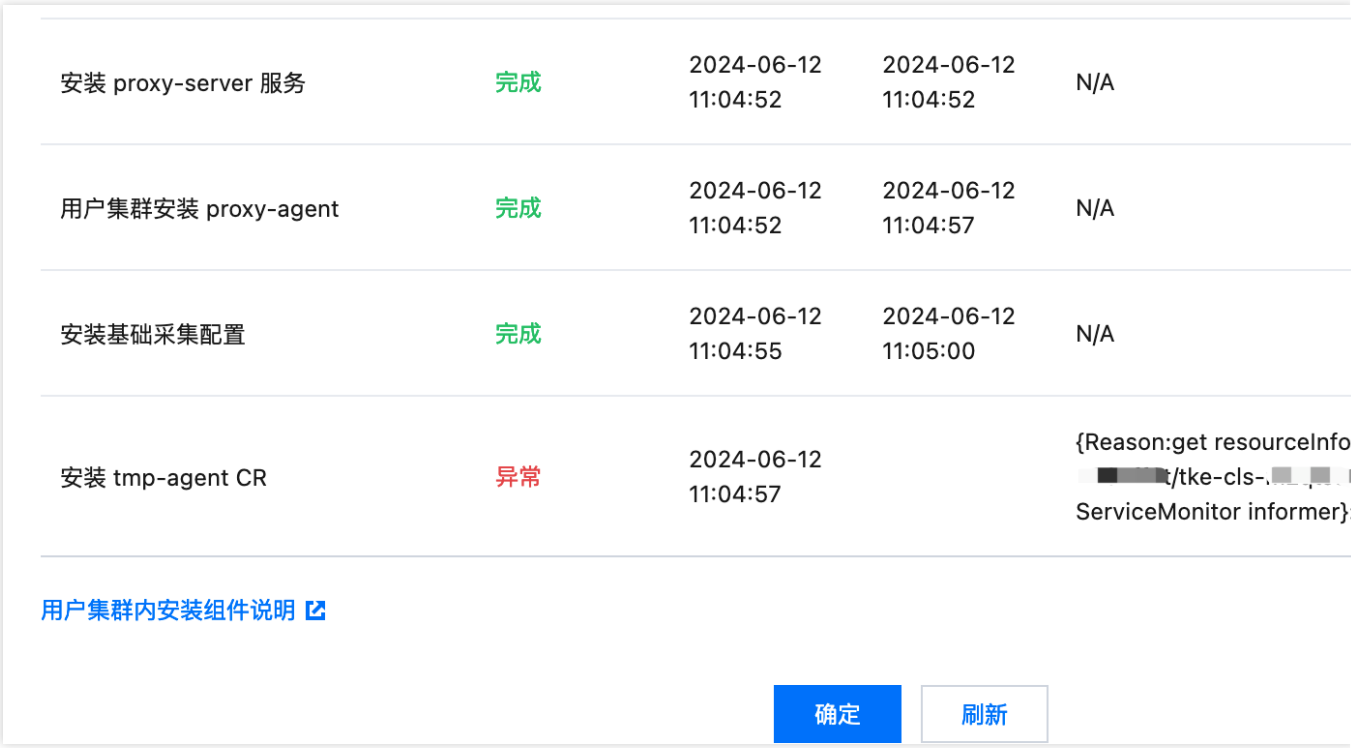
超级节点

点击**节点管理** > **Worker 节点** > **节点池**，点击节点池 id，在**节点池基本信息**中即可查看安全组：

相关问题

问题描述

绑定状态异常，“安装 tmp-agent CR”步骤显示“context deadline exceeded”：



问题处理

vpc 是否相同，或者打通

1. 点击用户集群链接，打开关联集群，查看集群节点网络（即 vpcid）：

2. 在 Prometheus 实例的**基本信息**页面，点击**所属网络**，打开后即可查看集群网络：

3. 对比 vpcid，若不一致，查看 vpc 是否通过云联网打通。若云联网没有打通，则需要关联云联网，打通两边的 vpc，或者在关联集群时勾选**创建公网 CLB**。若云联网已打通，仍不能关联成功，需查看云联网是否达到带宽上限，若已达到则需要调大云联网带宽上限。

关联云联网：

勾选创建公网 CLB：

安全组是否放通

1. 查看用户集群安全组，查看方式见 用户集群安全组查看方式，查看规则是否与要求一致；

2. 若用户集群为独立集群，查看 Master&Etcd 安全组信息，点击**节点管理 > Master&Etcd > 节点池**，点击节点池 id，并将鼠标放在节点 ID 上，点击**跳转到 CVM 实例详情页**，然后在 CVM 的**安全组**页面即可查看具体安全组信息：

检查安全组规则是否符合要求。