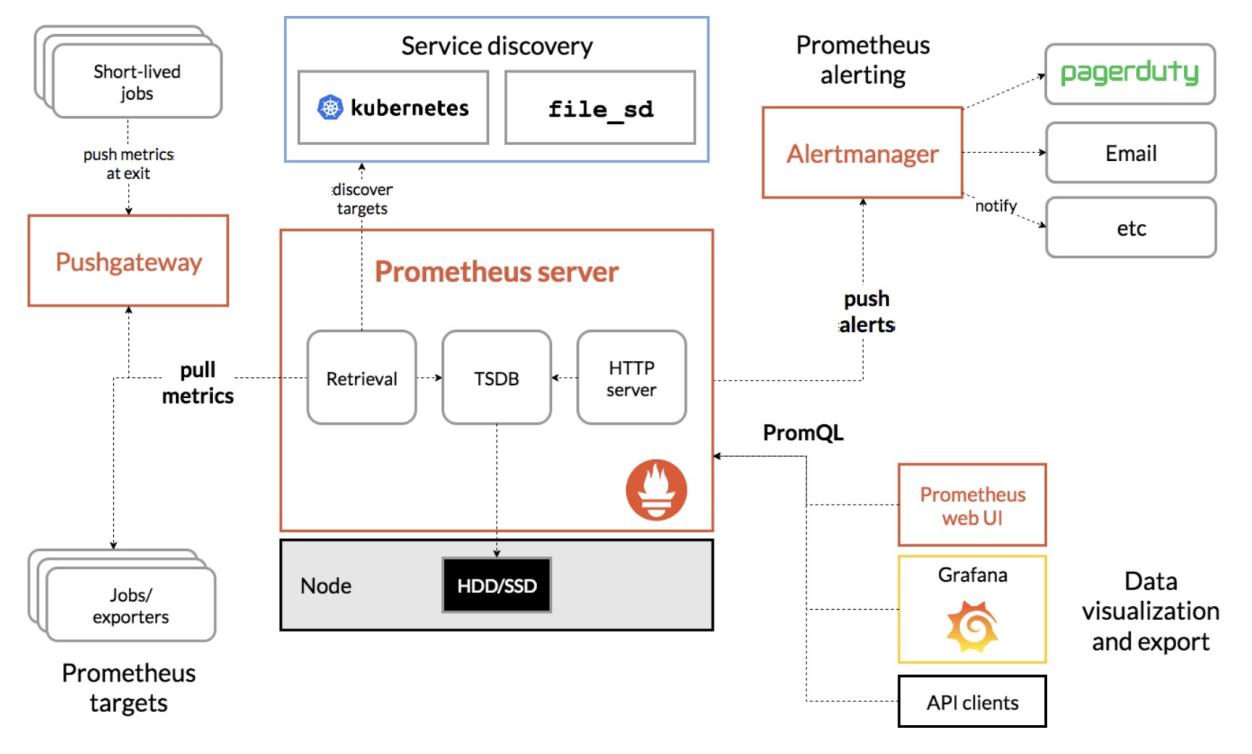
## **Prometheus在Kubernetes下的服务发现机制**

Prometheus作为容器监控领域的事实标准，随着以Kubernetes为核心的云原生热潮的兴起，已经得到了广泛的应用部署。灵活的服务发现机制是Prometheus和Kubernetes两者得以连接的基础，本文将对这部分内容进行介绍，从而让读者了解Prometheus如何对Kubernetes集群本身以及对运行其上的各种应用进行有效地监控。

### **1. Prometheus概述**

在正式进入主题之前，对Prometheus进行全面的了解是必要的。如下图所示，Prometheus Server是Prometheus生态中的核心组件。它会通过静态配置或者动态的服务发现，找到一系列的抓取对象，我们称之为target，Prometheus Server会定期从这些target中拉取时序数据，后文将对这部分内容进行详细叙述。对于不断抓取到的时序数据，Prometheus Server会进行聚合并默认存储到本地的时序数据库TSDB中。通过Prometehus Server的HTTP接口以及内置的查询语言PromQL可以对TSDB中保存的时序数据进行查询并通过Grafana等工具进行展示。同时，我们可以定义一系列的告警规则，Prometheus Server会定期查询TSDB以判断是否触发告警，若是则将报警消息推送至Alert Manager。Alert Manager会对告警消息进行聚合，去重等一系列复杂操作，在必要的时候将告警消息通过Email等方式通知用户。

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/prometheus/promarch.jpeg)

### **2. 静态配置**

当需要抓取的对象固定且数目较少时，将其直接写入Prometehus的配置文件是最好的选择。事实上，Prometheus Server作为一个应用程序，它同样对自己的运行数据进行了收集并以标准的方式对外暴露。因此，我们完全可以用Prometheus监控Prometheus。此时，Prometheus的配置文件如下，已知Prometheus Server运行在本地的9090端口：

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# A scrape configuration containing exactly one endpoint to scrape:# Here it's Prometheus itself.scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

# scheme: http

# metrics\_path: /metric

static\_configs:

- targets: ['127.0.0.1:9090']

上述配置文件的scrape\_configs中包含一个名为prometheus的job。在Prometheus中一个job代表多个配置类似的target的集合，例如它们都可以通过http或者https协议进行访问，访问路径都为/metrics等等。在上述prometheus这个job中，仅仅包含一个地址为127.0.0.1:9090的target。简单地说，Prometheus Server会每隔scrape\_interval就对target的URLhttp://127.0.0.1:9090进行访问获取时序数据。

### **3. Prometheus抓取机制**

在Prometheus内部，target的发现和抓取分别是由Scrape Discovery Manager和Scrape Manager独立实现的。两者通过如下所示的channel进行连接：

chan map[string][]\*targetgroup.Group

可以看到，上述管道传输的是map，其中的每个key和value对应的就是一个job名以及它所包含的一系列target group。事实上，在Prometheus内部一个target是用一系列的labels来描述的，每个label就是一个键值对，而一个target group就是共享某些labels的target集合。在下文中我们会了解到，一个Pod往往就和一个target group相对应。

每当从上述channel获取最新的target列表之后，Scrape Manager会进行重载。为每个job创建一个Scrape Pool，再为job中的每个target创建一个Scrape Loop。Scrape Pool会根据job的配置，创建一个http client供其中的各个Scrape Loop使用，而Scrape Loop则利用该http client具体完成对目标target的时序数据抓取工作。整体结构如下所示：

* Scrape Manager
  + Scrape Pool 1
    - Scrape Loop for target 1
    - Scrape Loop for target 2
    - [...]
    - Scrape Loop for target n
  + Scrape Pool 2
    - [...]
  + [...]
  + Scrape Pool n
    - [...]

在Scrape Discovery Manager端同样会根据job进行划分。每个job中可以包含多种的服务发现，基于File，基于Kubernetes，基于Consul等等。对于各种服务发现方式，我们统一用provider对象进行封装，类似地，provider通过channel：

chan<- []\*targetgroup.Group

定期向Scrape Discovery Manager推送最新的一系列target group。Scrape Discovery Manager则会将各个provider推送的target group进行聚合并向Scrape Manager推送。

事实上，对于静态配置的target，我们可以将它看作一种特殊的服务发现机制并且同样用provider进行封装，不同的是它只会用channel推送一次用户在配置文件中写好的target group，之后就会将channel关闭。

### **4. Kubernetes下的服务发现**

在Kubernetes环境下，要做到对系统本身，特别是运行其上的各种应用的完整监控，静态配置的方式显然是无法满足需求的。因此，基于Kubernetes本身的服务发现能力，Prometheus实现了对于Kubernetes集群，包括系统组件，Pod，Service，Ingress等各个维度的动态监控。本文将以针对Pod的服务发现作为例子，对于其他维度，其实现机制是类似的。

若要实现对于集群中的Pod的动态监控，则Prometheus的配置文件中需要加入以下job：

- job\_name: "kubernetes-pods"

kubernetes\_sd\_configs:

- role: pod

# api\_server: <host>

# basic\_auth: XXX

# tls\_config: XXX

relabel\_configs:

- source\_labels: [\_\_meta\_kubernetes\_pod\_annotation\_prometheus\_io\_scrape]

action: keep

regex: true

- source\_labels: [\_\_meta\_kubernetes\_namespace]

action: replace

target\_label: kubernetes\_namespace

- source\_labels: [\_\_meta\_kubernetes\_pod\_name]

action: replace

target\_label: kubernetes\_pod\_name

其中的kubernetes\_sd\_configs表示基于Kubernetes进行服务发现，而其中的role字段为pod，表示针对Kubernetes中的pod资源对象，进行具体的服务发现操作。role字段其余的合法值包括node, service, endpoints, ingress。

之后Prometheus会创建Kubernetes Client对role中指定的资源对象进行监听。一般Prometheus部署在Kubernetes集群中的话，Prometheus可以直接利用指定的Service Account对Kubernetes API进行访问。若Prometheus在Kubernetes集群之外，则kubernetes\_sd\_configs还需指定监控集群的API Server的URL以及相关的认证信息，从而能够创建对应集群的Client。

和Kubernetes内置的各种Controller类似，当目标资源对象为Pod时，Prometheus会对Kubernetes中的Pod进行List & Watch。每当集群中新增一个Pod，Prometheus就会对其进行处理，根据其配置创建一个target group。之后，Prometheus会遍历Pod中的每个Container：若Container没有暴露端口，则将一个Container作为一个target并将该target的\_\_address\_\_直接设置为Pod的IP地址；若Container暴露了一个或多个端口，则将每个端口设置为一个target且target的\_\_address设置为Pod IP加对应端口的组合。如上文所述，一个target group中的targets会共享某些labels，当target group对应的是一个pod时，Prometheus会将Pod的基本信息作为共享labels。例如：\_\_meta\_kubernetes\_namespace对应Pod所在的namespace，\_\_meta\_kubernetes\_pod\_name对应pod name，以及pod的ip，labels和annotations都会作为共享labels写入。

最终，这些自发现的target groups都将通过管道传递给Scrape Manager，由它来创建对于各个target的抓取实例并实现数据的抓取。

### **5. Targets的过滤机制**

从上文可知，若配置了对于Kubernetes中Pod资源对象的服务发现，Prometheus会默认将每个Pod IP + Pod Port的组合作为一个taget。显然，并不是每个Pod都会暴露符合Prometheus标准的监控数据，即使暴露了，也不太可能在它的每个容器的每个端口进行暴露。因此，通过上述服务发现机制获取到的大部分targets都将是无效的。已知targets本质上是一系列的labels，因此我们可以根据labels是否符合某些规则实现对targets的过滤，在Prometheus中，这一机制叫做relabel。

Prometheus配置文件中的relabel\_configs就包含了对于relabel规则的描述。我们可以同时定义多条relabels规则，它们会在Scrape Manager创建target实例并抓取之前，依次对targets实现过滤和修改。例如，上一节中的配置文件包含了三条relabel规则。第一条规则表示，若targets的labels中存在\_\_meta\_kubernetes\_pod\_annotation\_prometheus\_io\_scrape且该label的值为true，则对应的target保留。这条规则就要求我们在定义Pod的配置时，若期望该Pod的监控数据被Prometheus抓取，则需要为它添加prometheus.io/scrape:true这样一个annotation，其余Pod生成的target都将被丢弃。

另外，我们还可以利用relabel实现对targets中labels的修改。一般来说，以\_\_作为前缀的labels都只在内部使用，在relabels之后会统一删除。若想要保留其中的某些labels，我们可以如第二，三条规则所示，将名为\_\_meta\_kubernetes\_namespace和\_\_meta\_kubernetes\_pod\_name的labels重命名为kubernetes\_namespace和kubernetes\_pod\_name。这样一来，这两个label就能得到保留并出现在target最终的labels中，而对应的target抓取的所有指标都将额外添加kubernetes\_namespace和kubernetes\_pod\_name这两个labels。

这里我们只是对Prometheus的relabel机制进行了简单的介绍，事实上可以利用多条relabel规则的组合实现对于targets的复杂的过滤和修改，在此不再赘述。另外，Scrape Manager会在对target进行relabel之前，根据配置文件额外添加job，\_\_metrics\_path\_\_和\_\_scheme\_\_这三个label，而在relabel之后，若target中没有instance这个label，则会向其添加instance这个label，值为\_\_address\_\_这个label的值。

### **6. 总结**

Prometheus基于Kubernetes等目标监控系统自身的服务发现能力提供了抓取对象自发现机制，由此带来的巨大灵活性是它能够成为云原生时代监控领域事实标准的重要原因之一。通过上文的叙述，我们可以知道，一个抓取对象，即target，在Prometheus是用一系列labels来描述的，全局的配置文件描述了某些targets的公共配置，而Kubernetes等系统的服务发现能力则为Prometheus提供了targets的大部分配置信息。最终，我们可以通过relabel机制对targets对象进行修改过滤，从而实现对于有效target的抓取。而对于target的抓取，本质上是通过target中的\_\_scheme\_\_，\_\_address\_\_和\_\_metrcis\_path\_\_这几个label，拼凑出一个URL，比如http://10.32.0.2/metrics，对该URL发起一个GET请求，获取监控数据。

### **参考文献**

* [Prometheus Source Code](https://github.com/prometheus/prometheus)
* [Prometheus Internal Architecture](https://github.com/prometheus/prometheus/blob/master/documentation/internal_architecture.md)
* [Prometheus Doc](https://prometheus.io/docs/prometheus/latest/configuration/configuration/)

## **如何利用Prometheus监控你的应用**

Prometheus作为一套完整的开源监控接近方案，因为其诸多强大的特性以及生态的开放性，俨然已经成为了监控领域的事实标准并在全球范围内得到了广泛的部署应用。那么应该如何利用Prometheus对我们的应用形成有效的监控呢？事实上，作为应用我们仅仅需要以符合Prometheus标准的方式暴露监控数据即可，后续对于监控数据的采集，处理，保存都将由Prometheus自动完成。

一般来说，Prometheus监控对象有两种：如果用户对应用的代码有定制化能力，Prometheus提供了各种语言的SDK，用户能够方便地将其集成至应用中，从而对应用的内部状态进行有效监控并将数据以符合Prometheus标准的格式对外暴露；对于MySQL，Nginx等应用，一方面定制化代码难度颇大，另一方面它们已经以某种格式对外暴露了监控数据，对于此类应用，我们需要一个中间组件，利用此类应用的接口获取原始监控数据并转化成符合Prometheus标准的格式对外暴露，此类中间组件，我们称之为Exporter，社区中已经有大量现成的Exporter可以直接使用。

本文将以一个由Golang编写的HTTP Server为例，借此说明如何利用Prometheus的Golang SDK，逐步添加各种监控指标并以标准的方式暴露应用的内部状态。后续的内容将分为基础、进阶两个部分，通常第一部分的内容就足以满足大多数需求，但是若想要获得更多的定制化能力，那么第二部分的内容可以提供很好的参考。

### **1. 基础**

Promethues提供的各种语言的SDK包其实已经非常完善了，如果希望某个Golang程序能够被Prometheus监控，我们需要做的仅仅是引入client\_golang这个包并添加几行代码而已，代码实例如下：

package main

import (

"net/http"

"github.com/prometheus/client\_golang/prometheus/promhttp"

)

func main() {

http.Handle("/metrics", promhttp.Handler())

http.ListenAndServe(":8080", nil)

}

可以看到，上述代码中，我们仅仅启动了HTTP Server并将client\_golang提供的一个默认的HTTP Handler注册到了路径/metrics上。我们可以试着运行该程序并对接口进行访问，结果如下：

$ curl http://127.0.0.1:8080/metrics

...

# HELP go\_goroutines Number of goroutines that currently exist.

# TYPE go\_goroutines gauge

go\_goroutines 7

# HELP go\_info Information about the Go environment.

# TYPE go\_info gauge

go\_info{version="go1.12.1"} 1

# HELP go\_memstats\_alloc\_bytes Number of bytes allocated and still in use.

# TYPE go\_memstats\_alloc\_bytes gauge

go\_memstats\_alloc\_bytes 418912

...

Prometheus Golang SDK提供的默认Handler会自动注册一系列用于监控Golang运行时以及应用进程相关信息的监控指标。所以，在我们未注册任何自定义指标的情况下，依然暴露了一系列的指标。指标暴露的格式也非常统一，首行以# HELP开头用于说明该指标的用途，次行以# TYPE开头用于说明指标的类型，后续几行则是指标的具体内容。对于Prometheus来说，只要提供抓取对象的地址以及访问路径（一般为/metrics），它就能对如上所示的监控数据进行抓取。而对于用户来说，则只需要自定义指标并按照程序的运行情况对相应的指标值进行修改即可。中间对于监控数据的聚合暴露，Prometheus的SDK会自动帮你处理。下面，我们将结合具体的需求试着添加各种类型的自定义监控指标。

对于一个HTTP Server来说，了解当前请求的接收速率是非常重要的。Prometheus支持一种称为Counter的数据类型，这一类型本质上就是一个只能单调递增的计数器。如果我们定义一个Counter表示累积接收的HTTP请求的数目，那么最近一段时间内该Counter的增长量其实就是接收速率。另外，Prometheus中定义了一种自定义查询语句PromQL，能够方便地对样本的监控数据进行统计分析，包括对于Counter类型的数据求速率。因此，经过修改后的程序如下：

package main

import (

"net/http"

"github.com/prometheus/client\_golang/prometheus"

"github.com/prometheus/client\_golang/prometheus/promauto"

"github.com/prometheus/client\_golang/prometheus/promhttp"

)

var (

http\_request\_total = promauto.NewCounter(

prometheus.CounterOpts{

Name: "http\_request\_total",

Help: "The total number of processed http requests",

},

)

)

func main() {

http.HandleFunc("/", func(http.ResponseWriter, \*http.Request){

http\_request\_total.Inc()

})

http.Handle("/metrics", promhttp.Handler())

http.ListenAndServe(":8080", nil)

}

我们利用promauto包提供的NewCounter方法定义了一个Counter类型的监控指标，只需要填充名字以及帮助信息，该指标就创建完成了。需要注意的是，Counter类型数据的名字要尽量以\_total作为后缀。否则当Prometheus与其他系统集成时，可能会出现指标无法识别的问题。每当有请求访问根目录时，该指标就会调用Inc()方法加一，当然，我们也可以调用Add()方法累加任意的非负数。

再次运行修改后的程序，先对根路径进行多次访问，再对/metrics路径进行访问，可以看到新定义的指标已经成功暴露了：

$ curl http://127.0.0.1:8080/metrics | grep http\_request\_total

# HELP http\_request\_total The total number of processed http requests

# TYPE http\_request\_total counterhttp\_request\_total 5

监控累积的请求处理显然还是不够的，通常我们还想知道当前正在处理的请求的数量。Prometheus中的Gauge类型数据，与Counter不同，它既能增大也能变小。将正在处理的请求数量定义为Gauge类型是合适的。因此，我们新增的代码块如下：

...

var (

...

http\_request\_in\_flight = promauto.NewGauge(

prometheus.GaugeOpts{

Name: "http\_request\_in\_flight",

Help: "Current number of http requests in flight",

},

)

)

...

http.HandleFunc("/", func(http.ResponseWriter, \*http.Request){

http\_request\_in\_flight.Inc()

defer http\_request\_in\_flight.Dec()

http\_request\_total.Inc()

})

...

Gauge和Counter类型的数据操作起来的差别并不大，唯一的区别是Gauge支持Dec()或者Sub()方法减小指标的值。

对于一个网络服务来说，能够知道它的平均时延是重要的，不过很多时候我们更想知道响应时间的分布状况。Prometheus中的Histogram类型就对此类需求提供了很好的支持。具体到需要新增的代码如下：

...

var (

...

http\_request\_duration\_seconds = promauto.NewHistogram(

prometheus.HistogramOpts{

Name: "http\_request\_duration\_seconds",

Help: "Histogram of lantencies for HTTP requests",

// Buckets: []float64{.1, .2, .4, 1, 3, 8, 20, 60, 120},

},

)

)

...

http.HandleFunc("/", func(http.ResponseWriter, \*http.Request){

now := time.Now()

http\_request\_in\_flight.Inc()

defer http\_request\_in\_flight.Dec()

http\_request\_total.Inc()

time.Sleep(time.Duration(rand.Intn(1000)) \* time.Millisecond)

http\_request\_duration\_seconds.Observe(time.Since(now).Seconds())

})

...

在访问了若干次上述HTTP Server的根路径之后，从/metrics路径得到的响应如下：

# HELP http\_request\_duration\_seconds Histogram of lantencies for HTTP requests# TYPE http\_request\_duration\_seconds histogram

http\_request\_duration\_seconds\_bucket{le="0.005"} 0

http\_request\_duration\_seconds\_bucket{le="0.01"} 0

http\_request\_duration\_seconds\_bucket{le="0.025"} 0

http\_request\_duration\_seconds\_bucket{le="0.05"} 0

http\_request\_duration\_seconds\_bucket{le="0.1"} 3

http\_request\_duration\_seconds\_bucket{le="0.25"} 3

http\_request\_duration\_seconds\_bucket{le="0.5"} 5

http\_request\_duration\_seconds\_bucket{le="1"} 8

http\_request\_duration\_seconds\_bucket{le="2.5"} 8

http\_request\_duration\_seconds\_bucket{le="5"} 8

http\_request\_duration\_seconds\_bucket{le="10"} 8

http\_request\_duration\_seconds\_bucket{le="+Inf"} 8

http\_request\_duration\_seconds\_sum 3.238809838

http\_request\_duration\_seconds\_count 8

Histogram类型暴露的监控数据要比Counter和Gauge复杂得多，最后以\_sum和\_count开头的指标分别表示总的响应时间以及对于响应时间的计数。而它们之上的若干行表示：时延在0.005秒内的响应数目，0.01秒内的响应次数，0.025秒内的响应次数...最后的+Inf表示响应时间无穷大的响应次数，它的值和\_count的值是相等的。显然，Histogram类型的监控数据很好地呈现了数据的分布状态。当然，Histogram默认的边界设置，例如0.005,0.01这类数值一般是用来衡量一个网络服务的时延的。对于具体的应用场景，我们也可以对它们进行自定义，类似于上述代码中被注释掉的那一行（最后的+Inf会自动添加）。

与Histogram类似，Prometheus中定义了一种类型Summary，从另一个角度描绘了数据的分布状况。对于响应时延，我们可能想知道它们的中位数是多少？九分位数又是多少？对于Summary类型数据的定义及使用如下：

...

var (

...

http\_request\_summary\_seconds = promauto.NewSummary(

prometheus.SummaryOpts{

Name: "http\_request\_summary\_seconds",

Help: "Summary of lantencies for HTTP requests",

// Objectives: map[float64]float64{0.5: 0.05, 0.9: 0.01, 0.99: 0.001, 0.999, 0.0001},

},

)

)

...

http.HandleFunc("/", func(http.ResponseWriter, \*http.Request){

now := time.Now()

http\_request\_in\_flight.Inc()

defer http\_request\_in\_flight.Dec()

http\_request\_total.Inc()

time.Sleep(time.Duration(rand.Intn(1000)) \* time.Millisecond)

http\_request\_duration\_seconds.Observe(time.Since(now).Seconds())

http\_request\_summary\_seconds.Observe(time.Since(now).Seconds())

})

...

Summary的定义和使用与Histogram是类似的，最终我们得到的结果如下：

$ curl http://127.0.0.1:8080/metrics | grep http\_request\_summary# HELP http\_request\_summary\_seconds Summary of lantencies for HTTP requests# TYPE http\_request\_summary\_seconds summary

http\_request\_summary\_seconds{quantile="0.5"} 0.31810446

http\_request\_summary\_seconds{quantile="0.9"} 0.887116164

http\_request\_summary\_seconds{quantile="0.99"} 0.887116164

http\_request\_summary\_seconds\_sum 3.2388269649999994

http\_request\_summary\_seconds\_count 8

同样，\_sum和\_count分别表示请求的总时延以及请求的数目，与Histogram不同的是，Summary其余的部分分别表示，响应时间的中位数是0.31810446秒，九分位数位0.887116164等等。我们也可以根据具体的需求对Summary呈现的分位数进行自定义，如上述程序中被注释的Objectives字段。令人疑惑的是，它是一个map类型，其中的key表示的是分位数，而value表示的则是误差。例如，上述的0.31810446秒是分布在响应数据的0.45~0.55之间的，而并非完美地落在0.5。

事实上，上述的Counter，Gauge，Histogram，Summary就是Prometheus能够支持的全部监控数据类型了（其实还有一种类型Untyped，表示未知类型）。一般使用最多的是Counter和Gauge这两种基本类型，结合PromQL对基础监控数据强大的分析处理能力，我们就能获取极其丰富的监控信息。

不过，有的时候，我们可能希望从更多的特征维度去衡量一个指标。例如，对于接收到的HTTP请求的数目，我们可能希望知道具体到每个路径接收到的请求数目。假设当前能够访问/和/foo目录，显然定义两个不同的Counter，比如http\_request\_root\_total和http\_request\_foo\_total，并不是一个很好的方法。一方面扩展性比较差：如果定义更多的访问路径就需要创建更多新的监控指标，同时，我们定义的特征维度往往不止一个，可能我们想知道某个路径且返回码为XXX的请求数目是多少，这种方法就无能为力了；另一方面，PromQL也无法很好地对这些指标进行聚合分析。

Prometheus对于此类问题的方法是为指标的每个特征维度定义一个label，一个label本质上就是一组键值对。一个指标可以和多个label相关联，而一个指标和一组具体的label可以唯一确定一条时间序列。对于上述分别统计每条路径的请求数目的问题，标准的Prometheus的解决方法如下：

...var (

http\_request\_total = promauto.NewCounterVec(

prometheus.CounterOpts{

Name: "http\_request\_total",

Help: "The total number of processed http requests",

},

[]string{"path"},

)...

http.HandleFunc("/", func(http.ResponseWriter, \*http.Request){

...

http\_request\_total.WithLabelValues("root").Inc()

...

})

http.HandleFunc("/foo", func(http.ResponseWriter, \*http.Request){

...

http\_request\_total.WithLabelValues("foo").Inc()

...

})

)

此处以Counter类型的数据举例，对于其他另外三种数据类型的操作是完全相同的。此处我们在调用NewCounterVec方法定义指标时，我们定义了一个名为path的label，在/和/foo的Handler中，WithLabelValues方法分别指定了label的值为root和foo，如果该值对应的时间序列不存在，则该方法会新建一个，之后的操作和普通的Counter指标没有任何不同。而最终通过/metrics暴露的结果如下：

$ curl http://127.0.0.1:8080/metrics | grep http\_request\_total# HELP http\_request\_total The total number of processed http requests

# TYPE http\_request\_total counter

http\_request\_total{path="foo"} 9

http\_request\_total{path="root"} 5

可以看到，此时指标http\_request\_total对应两条时间序列，分别表示path为foo和root时的请求数目。那么如果我们反过来想统计，各个路径的请求总和呢？我们是否需要定义个path的值为total，用来表示总体的计数情况？显然是不必的，PromQL能够轻松地对一个指标的各个维度的数据进行聚合，通过如下语句查询Prometheus就能获得请求总和：

sum(http\_request\_total)

label在Prometheus中是一个简单而强大的工具，理论上，Prometheus没有限制一个指标能够关联的label的数目。但是，label的数目也并不是越多越好，因为每增加一个label，用户在使用PromQL的时候就需要额外考虑一个label的配置。一般来说，我们要求添加了一个label之后，对于指标的求和以及求均值都是有意义的。

### **2. 进阶**

基于上文所描述的内容，我们就能很好地在自己的应用程序里面定义各种监控指标并且保证它能被Prometheus接收处理了。但是有的时候我们可能需要更强的定制化能力，尽管使用高度封装的API确实很方便，不过它附加的一些东西可能不是我们想要的，比如默认的Handler提供的Golang运行时相关以及进程相关的一些监控指标。另外，当我们自己编写Exporter的时候，该如何利用已有的组件，将应用原生的监控指标转化为符合Prometheus标准的指标。为了解决上述问题，我们有必要对Prometheus SDK内部的实现机理了解地更为深刻一些。

在Prometheus SDK中，Register和Collector是两个核心对象。Collector里面可以包含一个或者多个Metric，它事实上是一个Golang中的interface，提供如下两个方法：

type Collector interface {

Describe(chan<- \*Desc)

Collect(chan<- Metric)

}

简单地说，Describe方法通过channel能够提供该Collector中每个Metric的描述信息，Collect方法则通过channel提供了其中每个Metric的具体数据。单单定义Collector还是不够的，我们还需要将其注册到某个Registry中，Registry会调用它的Describe方法保证新添加的Metric和之前已经存在的Metric并不冲突。而Registry则需要和具体的Handler相关联，这样当用户访问/metrics路径时，Handler中的Registry会调用已经注册的各个Collector的Collect方法，获取指标数据并返回。

在上文中，我们定义一个指标如此方便，根本原因是promauto为我们做了大量的封装，例如，对于我们使用的promauto.NewCounter方法，其具体实现如下：

http\_request\_total = promauto.NewCounterVec(

prometheus.CounterOpts{

Name: "http\_request\_total",

Help: "The total number of processed http requests",

},

[]string{"path"},

)

// client\_golang/prometheus/promauto/auto.go

func NewCounterVec(opts prometheus.CounterOpts, labelNames []string) \*prometheus.CounterVec {

c := prometheus.NewCounterVec(opts, labelNames)

prometheus.MustRegister(c)

return c

}

// client\_golang/prometheus/counter.go

func NewCounterVec(opts CounterOpts, labelNames []string) \*CounterVec {

desc := NewDesc(

BuildFQName(opts.Namespace, opts.Subsystem, opts.Name),

opts.Help,

labelNames,

opts.ConstLabels,

)

return &CounterVec{

metricVec: newMetricVec(desc, func(lvs ...string) Metric {

if len(lvs) != len(desc.variableLabels) {

panic(makeInconsistentCardinalityError(desc.fqName, desc.variableLabels, lvs))

}

result := &counter{desc: desc, labelPairs: makeLabelPairs(desc, lvs)}

result.init(result) // Init self-collection.

return result

}),

}

}

一个Counter（或者CounterVec，即包含label的Counter）其实就是一个Collector的具体实现，它的Describe方法提供的描述信息，无非就是指标的名字，帮助信息以及定义的Label的名字。promauto在对它完成定义之后，还调用prometheus.MustRegister(c)进行了注册。事实上，prometheus默认提供了一个Default Registry，prometheus.MustRegister会将Collector直接注册到Default Registry中。如果我们直接使用了promhttp.Handler()来处理/metrics路径的请求，它会直接将Default Registry和Handler相关联并且向Default Registry注册Golang Collector和Process Collector。所以，假设我们不需要这些自动注入的监控指标，只要构造自己的Handler就可以。

当然，Registry和Collector也都是能自定义的，特别在编写Exporter的时候，我们往往会将所有的指标定义在一个Collector中，根据访问应用原生监控接口的结果对所需的指标进行填充并返回结果。基于上述对于Prometheus SDK的实现机制的理解，我们可以实现一个最简单的Exporter框架如下所示：

package main

import (

"net/http"

"math/rand"

"time"

"github.com/prometheus/client\_golang/prometheus"

"github.com/prometheus/client\_golang/prometheus/promhttp"

)

type Exporter struct {

up \*prometheus.Desc

}

func NewExporter() \*Exporter {

namespace := "exporter"

up := prometheus.NewDesc(prometheus.BuildFQName(namespace, "", "up"), "If scrape target is healthy", nil, nil)

return &Exporter{

up: up,

}

}

func (e \*Exporter) Describe(ch chan<- \*prometheus.Desc) {

ch <- e.up

}

func (e \*Exporter) Scrape() (up float64) {

// Scrape raw monitoring data from target, may need to do some data format conversion here

rand.Seed(time.Now().UnixNano())

return float64(rand.Intn(2))

}

func (e \*Exporter) Collect(ch chan<- prometheus.Metric) {

up := e.Scrape()

ch <- prometheus.MustNewConstMetric(e.up, prometheus.GaugeValue, up)

}

func main() {

registry := prometheus.NewRegistry()

exporter := NewExporter()

registry.Register(exporter)

http.Handle("/metrics", promhttp.HandlerFor(registry, promhttp.HandlerOpts{}))

http.ListenAndServe(":8080", nil)

}

在这个Exporter的最简实现中，我们创建了新的Registry，手动对exporter这个Collector完成了注册并且基于这个Registry自己构建了一个Handler并且与/metrics相关联。在初始exporter的时候，我们仅仅需要调用NewDesc()方法填充需要监控的指标的描述信息。当用户访问/metrics路径时，经过完整的调用链，最后在进行Collect的时候，我们才会对应用的原生监控接口进行访问，获取监控数据。在真实的Exporter实现中，该步骤应该在Scrape()方法中完成。最后，根据返回的原生监控数据，利用MustNewConstMetric()构造出我们所需的Metric，返回给channel即可。访问该Exporter的/metrics得到的结果如下：

$ curl <http://127.0.0.1:8080/metrics>

# HELP exporter\_up If scrape target is healthy

# TYPE exporter\_up gauge

exporter\_up 1

### **3. 总结**

经过本文的分析，可以发现，利用Prometheus SDK将应用程序进行简单的二次开发，它就能被Prometheus有效地监控，从而享受整个Prometheus监控生态带来的便利。同时，Prometheus SDK也提供了多层次的抽象，通常情况下，高度封装的API就能快速地满足我们的需求。至于更多的定制化需求，Prometheus SDK也有很多底层的，更为灵活的API可供使用。

本文中的示例代码以及如何对应用进行Prometheus化二次开发和编写Exporter的详细规范，参见参考文献中的相关内容。

### **参考文献**

* [Prometheus client\_golang Source Code](https://github.com/prometheus/client_golang)
* [Prometheus Doc: Writing Exporters](https://prometheus.io/docs/instrumenting/writing_exporters/)
* [Prometheus Doc: Instrumentation](https://prometheus.io/docs/practices/instrumentation/)
* [示例代码](https://github.com/YaoZengzeng/practice/tree/master/promdemo)

## **Prometheus存储模型分析**

Prometheus是时下最为流行的开源监控解决方案，我们可以很轻松地以Prometheus为核心快速构建一套包含监控指标的抓取，存储，查询以及告警的完整监控系统。单个的Prometheus实例就能实现每秒上百万的采样，同时支持对于采集数据的快速查询，而对于Kubernetes这类抓取对象变更频繁的环境，Prometheus也是最好的选择。显然，这些优秀特性的实现都离不开一个设计优良的时序数据库的支撑。本文就将对Prometheus内置的时序数据库tsdb的设计与实现进行剖析，从架构设计以及代码层面理解它何以支持Prometheus强大的读写表现。

### **1. 时序数据概述**

Prometheus读写的是时序数据，与一般的数据对象相比，时序数据有其特殊性，tsdb对此进行了大量针对性的设计与优化。因此理解时序数据是理解Prometheus存储模型的第一步。通常，它由如下所示的标识和采样数据两部组成：

标识 -> {(t0, v0), (t1, v1), (t2, v2), (t3, v3)...}

标识用于区分各个不同的监控指标，在Prometheus中通常用指标名+一系列的label唯一地标识一个时间序列。如下为Prometheus抓取的一条时间序列，其中http\_request\_total为指标名，表示HTTP请求的总数，它有path和method两个label，用于表示各种请求的路径和方法。

http\_request\_total{path="/", method="GET"} -> {(t0, v1), (t1, v1)...}

事实上指标名最后也是作为一个特殊的label被存储的，它的key为\_\_name\_\_，如下所示。最终Prometheus存储在数据库中的时间序列标识就是一堆label。我们将这堆label称为series。

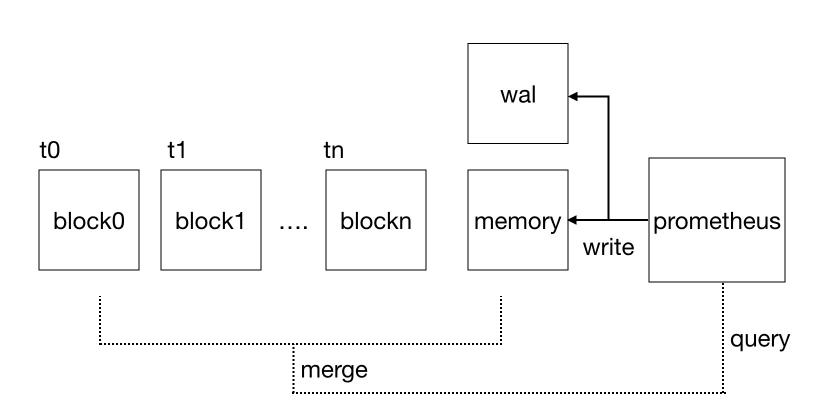
{\_\_name\_\_="http\_request\_total", path="/", method="GET"}

采样数据则由诸多的采样点（Prometheus中称为sample）构成，t0, t1, t2...表示样本采集的时间，v0, v1, v2...则表示指标在采集时刻的值。采样时间一般是单调递增的并且相邻sample的时间间隔往往相同，Prometheus中默认为15s。而且一般相邻sample的指标值v并不会相差太多。基于采样数据的上述特性，对它进行高效地压缩存储是完全可能的。Prometheus对于采样数据压缩算法的实现，参考了Facebook的时序数据库[Gorilla](http://www.vldb.org/pvldb/vol8/p1816-teller.pdf)中的做法，通过该算法，16字节的sample平均只需要1.37个字节的存储空间。

### **2. 架构设计**

监控数据是一种时效性非常强的数据类型，它被查询的热度会随着时间的流逝而不断降低，而且对于监控指标的访问通常会指定一个时间段，例如，最近十五分钟，最近一小时，最近一天等等。一般来说，最近一个小时采集到的数据被访问地是最为频繁的，过去一天的数据也经常会被访问用来了解某个指标整体的波动情况，而一个月乃至一年之前的数据被访问的意义就不是很大了。

基于监控数据的上述特性，tsdb的设计就非常容易理解了，其整体架构如下：

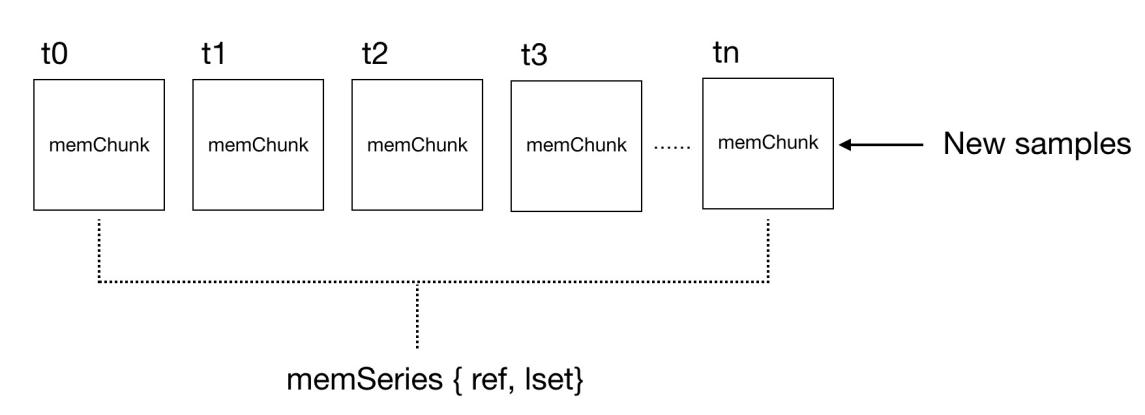
[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/prometheusstorage/arch.jpg)

对于最新采集到的数据，Prometheus会直接将它们存放在内存中，从而加快数据的读写。但是内存的空间是有限的，而且随着时间的推移，内存中较老的那部分数据被访问的概率也逐渐降低。因此，默认情况下，每隔两小时Prometheus就会将部分“老”数据持久化到磁盘，每一次持久化的数据都独立存放在磁盘的一个Block中。例如上图中的block0就存放了[t0, t1]时间段内Prometheus采集的所有监控数据。这样做的好处很明显，如果我们想要访问某个指标在[t0, t2]范围内的数据，那么只需要加载block0和block1中的数据并进行查找即可，这样一来大大缩小了查找的范围，从而提高了查询的速度。

虽然最近采集的数据存放在内存中能够提高读写效率，但是由于内存的易失性，一旦Prometheus崩溃（如果系统内存不足，Prometheus被OOM的概率并不算低）那么这部分数据就彻底丢失了。因此**Prometheus在将采集到的数据真正写入内存之前，会首先存入WAL（Write Ahead Log）中**。因为WAL是存放在磁盘中的，相当于对内存中的监控数据做了一个完全的备份，即使Prometheus崩溃这部分的数据也不至于丢失。当Prometheus重启之后，它首先会将WAL的内容加载到内存中，从而完美恢复到崩溃之前的状态，接着再开始新数据的抓取。

### **3. 内存存储结构**

在Prometheus的内存中使用如下所示的memSeries结构存储时间序列，一条时间序列对应一个memSeries结构：

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/prometheusstorage/memseries.jpg)

可以看到，一个memSeries主要由三部分组成：

1. lset：用以识别这个series的label集合
2. ref：每接收到一个新的时间序列（即它的label集合与已有的时间序列都不同）Prometheus就会用一个唯一的整数标识它，如果有ref，我们就能轻易找到相应的series
3. memChunks：每一个memChunk是一个时间段内该时间序列所有sample的集合。如果我们想要读取[tx, ty]（t1 < tx < t2, t2 < ty < t3 ）时间范围内该时间序列的数据，只需要对[t1, t3]范围内的两个memChunk的sample数据进行裁剪即可，从而提高了查询的效率。每当采集到新的sample，Prometheus就会用Gorilla中类似的算法将它压缩至最新的memChunk中

但是ref仅仅是供Prometheus内部使用的，如果用户要查询某个具体的时间序列，通常会利用一堆的label用以唯一指定一个时间序列。那么如何通过一堆label最快地找到对应的series呢？哈希表显然是最佳的方案。基于label计算一个哈希值，维护一张哈希值与memSeries的映射表，如果产生哈希碰撞的话，则直接用label进行匹配。因此，Prometheus有必要在内存中维护如下所示的两张哈希表，从而无论利用ref还是label都能很快找到对应的memSeries：

{

series map[uint64]\*memSeries // ref到memSeries的映射

hashes map[uint64][]\*memSeries // labels的哈希值到memSeries的映射

}

然而我们知道Golang中的map并不是并发安全的，而Prometheus中又有大量对于memSeries的增删操作，如果在读写上述结构时简单地用一把大锁锁住，显然无法满足性能要求。所以Prometheus用了如下数据结构将锁的控制精细化:

const stripSize = 1 << 14

// 为表达直观，已将Prometheus原生数据结构简化

type stripeSeries struct {

series [stripeSize]map[uint64]\*memSeries

hashes [stripeSize]map[uint64][]\*memSeries

locks [stripeSize]sync.RWMutex

}

Prometheus将一整个大的哈希表进行了切片，切割成了16k个小的哈希表。如果想要利用ref找到对应的series，首先要将ref对16K取模，假设得到的值为x，找到对应的小哈希表series[x]。至于对小哈希表的操作，只需要锁住模对应的locks[x]，从而大大减小了读写memSeries时对锁的抢占造成的损耗，提高了并发性能。对于基于label哈希值的读写，操作类似。

然而上述数据结构仅仅只能支持对于时间序列的精确查询，必须严格指定每一个label的值从而能够唯一地确定一条时间序列。但很多时候，模糊查询才是更为常用的。例如，我们想知道访问路径为/的各类HTTP请求的数目（请求的方法可以为GET，POST等等），此时提交给Prometheus的查询条件如下：

http\_request\_total{path="/"}

如果路径/曾经接收了GET,POST以及DELETE三种方法的HTTP请求，那么此次查询应该返回如下三条时间序列：

http\_request\_total{path="/", method="GET"} ....

http\_request\_total{path="/", method="POST"} ....

http\_request\_total{path="/", method="DELETE"} ....

Prometheus甚至支持在指定label时使用正则表达式：

http\_request\_total{method="GET|POST"}

上述查询将返回所有包含label名为method，且值为GET或者POST的指标名为http\_request\_total的时间序列。

针对如此复杂的查询需求，暴力地遍历所有series进行匹配是行不通的。一个指标往往会包含诸多的label，每个label又可以有很多的取值。因此Prometheus中会存在大量的series，为了能快速匹配到符合要求的series，Prometheus引入了**倒排索引**，结构如下：

struct MemPostings struct {

mtx sync.Mutex

m map[string]map[string][]uint64

ordered bool

}

当Prometheus抓取到一个新的series，假设它的ref为x，包含如下的label pair：

{\_\_name\_\_="http\_request\_total", path="/", method="GET"}

在初始化相应的memSeries并且更新了哈希表之后，还需要对倒排索引进行刷新：

MemPostings.m["\_\_name\_\_"]["http\_request\_total"]{..., x ,...}

MemPostings.m["path"]["/"]{..., x ,...}

MemPostings.m["method"]["GET"]{..., x, ...}

可以看到，倒排索引能够将所有包含某个label pair的series都聚合在一起。如果要得到匹配多个label pair的series，只要将每个label pair包含的series做交集即可。对于查询请求

http\_request\_total{path="/"}

的匹配过程如下：

MemPostings.["\_\_name\_\_"]["http\_request\_total"]{3, 4, 2, 1}

MemPostings.["path"]["/"]{5, 4, 1, 3}

{3, 4, 2, 1} x {5, 4, 1, 3} -> {1, 3, 4}

但是如果每个label pair包含的series足够多，那么对多个label pair的series做交集也将是非常耗时的操作。那么能不能进一步优化呢？事实上，只要保持每个label pair里包含的series有序就可以了，这样就能将复杂度从指数级瞬间下降到线性级。

MemPostings.["\_\_name\_\_"]["http\_request\_total"]{1, 2, 3, 4}

MemPostings.["path"]["/"]{1, 3, 4, 5}

{1, 2, 3, 4} x {1, 3, 4, 5} -> {1, 3, 4}

Prometheus内存中的存储结构大致如上，Gorilla的压缩算法提高了samples的存储效率，而哈希表以及倒排索引的使用，则对Prometheus复杂的时序数据查询提供了高效的支持。

### **WAL**

Prometheus启动时，往往需要在参数中指定存储目录，该目录包含WAL以及用于持久化的Block，结构如下：

# ls

01DJQ428PCD7Z06M6GKHES65P2 01DJQAXZY8MPVWMD2M4YWQFD9T 01DJQAY7F9WT8EHT0M8540F0AJ lock wal

形如01DJQ428PCD7Z06M6GKHES65P2都是Block目录，用于存放持久化之后的时序数据，这部分内容后文会有详细的叙述，本节重点关注WAL目录，它的内部结构如下：

[wal]# ls -lht

total 596M

-rw-r--r-- 1 65534 65534 86M Aug 20 19:32 00000012

drwxr-xr-x 2 65534 65534 4.0K Aug 20 19:00 checkpoint.000006

-rw-r--r-- 1 65534 65534 128M Aug 20 19:00 00000011

-rw-r--r-- 1 65534 65534 128M Aug 20 18:37 00000010

-rw-r--r-- 1 65534 65534 128M Aug 20 17:47 00000009

-rw-r--r-- 1 65534 65534 128M Aug 20 17:00 00000008

-rw-r--r-- 1 65534 65534 128M Aug 20 16:38 00000007

WAL目录中包含了多个连续编号的且大小为128M的文件，Prometheus称这样的文件为Segment，其中存放的就是对内存中series以及sample数据的备份。另外还包含一个以checkpoint为前缀的子目录，由于内存中的时序数据经常会做持久化处理，WAL中的数据也将因此出现冗余。所以每次在对内存数据进行持久化之后，Prometheus都会对部分编号靠后的Segment进行清理。但是我们并没有办法做到恰好将已经持久化的数据从Segment中剔除，也就是说被删除的Segment中部分的数据依然可能是有用的。所以在清理Segment时，我们会将肯定无效的数据删除，剩下的数据就存放在checkpoint中。而在Prometheus重启时，应该首先加载checkpoint中的内容，再按序加载各个Segment的内容。

那么series和sample在Segment中是如何组织的呢？在将时序数据备份到WAL的过程中，由于涉及到磁盘文件Segment的写入，批量操作显然是最经济的。对于批量写入，Prometheus提供了一个名为Appender的接口如下：

type Appender interface {

Add(l labels.Labels, t int64, v float64) (uint64, error)

AddFast(ref uint64, t int64, v float64) error

Commit() error

Rollback() error

}

每当需要写入数据时，就要创建一个Appender，Appender是一个临时结构，仅供一次批量操作使用。一个Appender类似于其他数据库中事务的概念，通过Add()或者AddFast()添加的时序数据会临时在Appender中进行缓存，只有在最后调用Commit()之后，这批数据才正式提交给Prometheus，同时写入WAL。而如果最后调用的Rollback()，则这批数据的samples会被全部丢弃，但是通过Add()方法新增的series结构则依然会被保留。

series和sample在Appender中是分开存储的，它们在Appender中的结构如下：

// headAppender是Appender的一种实现

type headAppender struct {

...

series []RefSeries

samples []RefSample

}

type RefSeries struct {

Ref uint64

Labels labels.Labels

}

type RefSample struct {

Ref uint64

T int64

V float64

series \*memSeries

}

当调用Appender的Commit()方法提交这些时序数据时，series和samples这两个切片会分别编码，形成两条Record，如下所示：

|RecordType|RecordContent|

RecordType可以取“RecordSample”或者“RecordSeries”，表示这条Record的类型

RecordContent则根据RecordType可以series或者samples编码后的内容

最后，series和samples以Record的形式被批量写入Segment文件中，默认当Segment超过128M时，会创建新的Segment文件。若Prometheus因为各种原因崩溃了，WAL里的各个Segment以及checkpoint里的内容就是在崩溃时刻Prometheus内存的映像。Prometheus在重启时只要加载WAL中的内容就能完全"恢复现场"。

### **Block**

虽然将时序数据存储在内存中能够最大化读写效率，但是时序数据的写入是稳定而持续的，随着时间的流逝，数据量会线性增长，而且相对较老的数据被访问的概率也将逐渐下降。因此，定期将内存中的数据持久化到磁盘是合理的。每一个Block存储了对应时间窗口内的所有数据，包括所有的series，samples以及相关的索引结构。Block目录的详细内容如下：

[01DJNTVX7GZ2M1EKB4TM76APV8]# ls

chunks index meta.json tombstones

[01DJNTVX7GZ2M1EKB4TM76APV8]# ls chunks/

000001

meta.json包含了当前Block的元数据信息，其内容如下：

# cat meta.json

{

"ulid": "01DJNTVX7GZ2M1EKB4TM76APV8",

"minTime": 1566237600000,

"maxTime": 1566244800000,

"stats": {

"numSamples": 30432619,

"numSeries": 65064,

"numChunks": 255203

},

"compaction": {

"level": 1,

"sources": [

"01DJNTVX7GZ2M1EKB4TM76APV8"

]

},

"version": 1

}

各字段的含义如下：

ulid：用于识别这个Block的编号，它与Block的目录名一致

minTime和maxTime：表示这个Block存储的数据的时间窗口

stats：表示这个Block包含的sample, series以及chunks数目

compaction：这个Block的压缩信息，因为随着时间的流逝，多个Block也会压缩合并形成更大的Block。level字段表示了压缩的次数，刚从内存持久化的Block的level为1，每被联合压缩一次，子Block的level就会在父Block的基础上加一，而sources字段则包含了构成当前这个Block的所有祖先Block的ulid。事实上，对于level >= 2的Block，还会有一个parent字段，包含了它的父Block的ulid以及时间窗口信息。

chunks是一个子目录，包含了若干个从000001开始编号的文件，一般每个文件大小的上限为512M。文件中存储的就是在时间窗口[minTime，maxTime]以内的所有samples，本质上就是对于内存中符合要求的memChunk的持久化。

tombstones用于存储对于series的删除记录。如果删除了某个时间序列，Prometheus并不会立即对它进行清理，而是会在tombstones做一次记录，等到下一次Block压缩合并的时候统一清理。

index文件存储了索引相关的内容，虽然持久化后的数据被读取的概率是比较低的，但是依然存在被读取的可能。这样一来，如何尽快地从Block中读取时序数据就显得尤为重要了，而快速读取索引并且基于索引查找时序数据则是加快整体查询效率的关键。为了达到这一目标，存储索引信息的index文件在设计上就显得比较复杂了。

┌────────────────────────────┬─────────────────────┐

│ magic(0xBAAAD700) <4b> │ version(1) <1 byte> │

├────────────────────────────┴─────────────────────┤

│ ┌──────────────────────────────────────────────┐ │

│ │ Symbol Table │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Series │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Label Index 1 │ │

│ ├──────────────────────────────────────────────┤ │

│ │ ... │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Label Index N │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Postings 1 │ │

│ ├──────────────────────────────────────────────┤ │

│ │ ... │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Postings N │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Label Index Table │ │

│ ├──────────────────────────────────────────────┤ │

│ │ Postings Table │ │

│ ├──────────────────────────────────────────────┤ │

│ │ TOC │ │

│ └──────────────────────────────────────────────┘ │

└──────────────────────────────────────────────────┘

除了文件开头的Magic Number以及版本信息，index文件可以分为7个部分，各部分的内容及作用如下：

TOC：虽然位于文件的末尾，但是TOC包含了整个index文件的全局信息，它存储的内容是其余六部分的位置信息，即它们的起始位置在index文件中的偏移量。

Symbol Table：一个symbol既可以是一个label的key，也可以是它的value，事实上Symbol Table存储的就是在[minTime, maxTime]范围内的samples所属的series的所有label的key和value集合，并且为每个symbol进行了编号。之所以要这样做，是因为后面在存储series以及Label Index等信息的时候，就不需要完整存储所有的label了，只需将label的key和value用对应的字符串在Symbol Table中的编号表示即可，从而大大减小了index文件的体积。

Series：存储的自然是series的相关信息，首先存储series的各个label，正如上文所述，存储的是对应key和value在Symbol Table中的编号。紧接着存储series相关的chunks信息，包含每个chunk的时间窗口，以及该chunk在chunks子目录下具体的位置信息。

Label Index：存储了各个label的key和它所有可能的value的关联关系。例如，对于一个有着四个不同的value的key，它在这部分存储的条目如下所示：

┌────┬───┬───┬──────────────┬──────────────┬──────────────┬──────────────┬───────┐

│ 24 │ 1 │ 4 │ ref(value\_0) | ref(value\_1) | ref(value\_2) | ref(value\_3) | CRC32 |

└────┴───┴───┴──────────────┴──────────────┴──────────────┴──────────────┴───────┘

各段含义如下：

24 --> 存储的内容包含24个字节

1 --> 本条目仅仅包含一个key

4 --> 与keys相关的有4个value

ref -> 各个value在Symbol Table中的编号

事实上这样一个条目可以存储多个key和它们的value的映射关系，但一般key的个数都为1。这部分的内容乍一看非常让人疑惑，key的信息呢？为什么只记录key的数目，而没有保存具体的key，哪怕是它在Symbol Table中的编号？其实，我们应该将这部分的内容和Label Index Table中的内容联合起来看。

Label Index Table：存储了所有label的key，以及它们在Label Index中对应的位置信息。那么为什么要将这两部分的内容分开存储呢？Prometheus在读取Block中的数据时会加载index文件，但是只会首先加载Label Index Table获取所有label的key，只有在需要对key相关的value进行匹配时，才会加载Label Index相应的部分以及对应的Symbol。通过Label Index Table和Label Index的分离，使得我们能够只对必要数据进行加载，从而加快了index文件的加载速度。

Postings： 这部分存储的显然是倒排索引的信息，每一个条目存储的都是包含某个label pair的所有series的ID。但是与Label Index相似，条目中并没有指定具体的key和value。

Postings Offset Table：这部分直接对每个label的key和value以及相关索引在Postings中的位置进行存储。同样，它会首先被加载到内存中，如果需要知道包含某个label的所有series，再通过相关索引的偏移位置从Postings中依次获取。

可以看到，index文件结构颇为复杂，但其实设计得相当巧妙，环环相扣。不仅高效地对索引信息进行了存储，而且也最大限度保证了对它进行加载的速度。

### **总结**

Prometheus精巧的设计使它能够对海量时序数据进行高效地读写。但是，通篇分析下来，Prometheus并没有所谓"黑科技"式的创新，有的不过是“逢山开路，遇水搭桥”式的优化。而这本身也是Prometheus的设计哲学，"Do one thing and do it well"。事实上，Prometheus默认仅支持15天的本地存储，超过这个期限的Block都会被清除。当然，社区对此已经提供了解决方案，[Thanos](https://github.com/thanos-io/thanos)和[Cortex](https://github.com/cortexproject/cortex)都基于Prometheus进行了扩展，提供了持久化存储，高可用等特性，从而能够真正做到"Prometheus As A Service"。

### **参考文献**

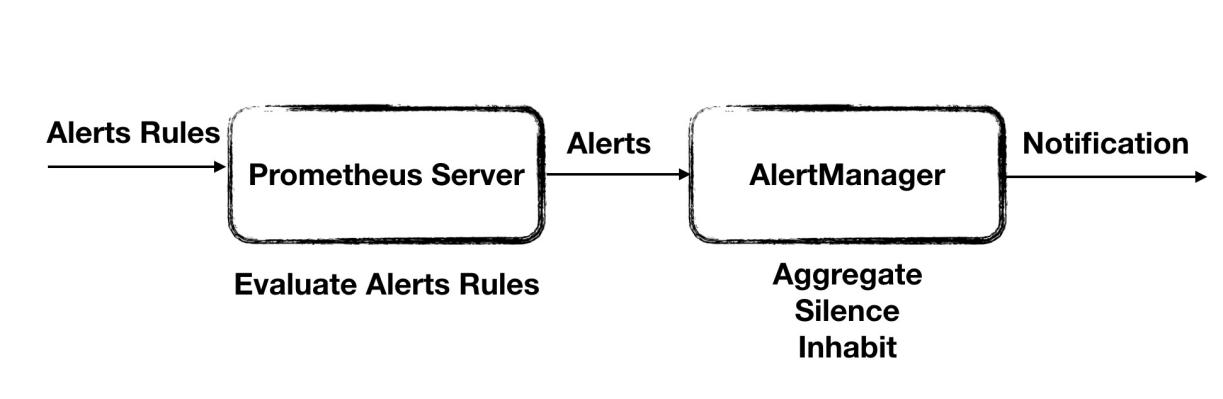
* [Prometheus Source Code](https://github.com/YaoZengzeng/prometheus/tree/2.11comments)
* [Write a Time Series Database from Scratch](https://fabxc.org/tsdb/)
* [Index Disk Format](https://github.com/prometheus/prometheus/blob/master/tsdb/docs/format/index.md)

## **Prometheus告警模型分析**

Prometheus作为时下最为流行的开源监控系统，其庞大的生态体系：包括针对各种传统应用的Exporter，完整的二次开发工具链，与Kubernetes等主流平台的高度亲和以及由此带来的强大的自发现能力，使得我们通过简单的配置就能获取大量的监控指标且包含的维度及其丰富。一方面，如此多样的指标极大地提高了集群的可观测性，配合Grafana等Dashboard就能让我们实时了解集群各个维度的状态；另一方面，基于监控数据进行实时地告警也是在可观测性得到满足之后必然要实现的需求。当然，Prometheus社区已经很好地解决了这个问题，本文也将对Prometheus的告警模型进行详细的叙述。

### **1. 概述**

如果对Prometheus项目有所了解的话，可以发现，Prometheus一个非常重要的原则就是尽量让设计保持简洁并且用简洁的设计满足绝大多数场景的需求。同时让项目保持良好的扩展性，针对极端场景，可以拼接Prometheus生态的一些外围组件来对已有能力进行增强，从而满足要求。对于告警也是类似的，基于Prometheus的告警系统的整体架构下图所示：

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/alertmanager/arch.jpg)

告警系统整体被解耦为两部分：

1. **Prometheus Server**会读取一系列的告警规则并基于采集的监控数据定期对这些规则进行评估，一旦满足触发条件就会生成相应的告警实例发送至AlertManager
2. **AlertManager**是一个独立于Prometheus Server运行的HTTP Server，它负责接受来自Client端的告警实例并对这些实例进行聚合（aggregation），静默（silence），抑制（inhibit）等高级操作并且支持Email，Slack等多种通知平台对告警进行通知。对于AlertManager来说，它并不在乎告警实例是否是由Prometheus Server发出的，因此我们只要构造出符合要求的告警实例并发送至Alertmanager，它就能无差别地进行处理。

### **2. Alert Rules**

通常来说，Prometheus的告警规则都会以文件的形式保存在磁盘中，我们需要在配置文件中指定这些规则文件的位置供Prometheus Server启动时读取：

rule\_files:

- /etc/prometheus/rules/\*.yaml

一般一个规则文件的内容如下：

groups:

- name: example

rules:

- alert: HighRequestLoad

expr: rate(http\_request\_total{pod="p1"}[5m]) > 1000

for: 1m

labels:

severity: warning

annotations:

info: High Request Load

在一个规则文件中可以指定若干个group，每个group内可以指定多条告警规则。一般来说，一个group中的告警规则之间会存在某种逻辑上的联系，但即使它们毫无关联，对后续的流程也不会有任何影响。而一条告警规则中包含的字段及其含义如下：

1. **alert:** 告警名称
2. **expr:** 告警的触发条件，本质上是一条promQL查询表达式，Prometheus Server会定期（一般为15s）对该表达式进行查询，若能够得到相应的时间序列，则告警被触发
3. **for:** 告警持续触发的时间，因为数据可能存在毛刺，Prometheus并不会因为在expr第一次满足的时候就生成告警实例发送到AlertManager。比如上面的例子意为名为"p1"的Pod，每秒接受的HTTP请求的数目超过1000时触发告警且持续时间为一分钟，若告警规则每15s评估一次，则表示只有在连续四次评估该Pod的负载都超过1000QPS的情况下，才会真正生成告警实例。
4. **labels:** 用于附加到告警实例中的标签，Prometheus会将此处的标签和评估expr得到的时间序列的标签进行合并作为告警实例的标签。告警实例中的标签构成了该实例的唯一标识。事实上，告警名称最后也会包含在告警实例的label中，且key为"alertname"。
5. **annotations:** 用于附加到告警实例中的额外信息，Prometheus会将此处的annotations作为告警实例的annotations，一般annotations用于指定告警详情等较为次要的信息

需要注意的是，一条告警规则并不只会生成一类告警实例，例如对于上面的例子，可能有如下多条时间序列满足告警的触发条件，即n1和n2这两个namespace下名为p1的pod的QPS都持续超过了1000：

http\_request\_total{namespace="n1", pod="p1"}

http\_request\_total{namespace="n2", pod="p1"}

最终生成的两类告警实例为：

# 此处只显示实例的label

{alertname="HighRequestLoad", severity="warning", namespace="n1", pod="p1"}

{alertname="HighRequestLoad", severity="warning", namespace="n2", pod="p1"}

因此，例如在K8S场景下，由于Pod具有易失性，我们完全可以利用强大的promQL语句，定义一条Deployment层面的告警，只要其中任何的Pod满足触发条件，都会产生对应的告警实例。

### **3. 在Kubernetes下操作Alert Rules**

初一看，Prometheus这种将所有告警规则一股脑写入文件中的方式貌似很简单，事实上，这的确简化了Prometheus本身的设计实现难度。但是，真正在生产环境中，尤其是当把Prometheus Server以Pod的形式部署在Kubernetes集群中时，对告警规则的增删改差操作将变得异常繁琐。特别地，在Kubernetes环境中，显然我们只能将若干告警规则文件包含在ConfigMap中并挂载到Prometheus所在Pod的指定目录中，如果要进行增删改操作，最直观的方法就是整体加载该ConfigMap并在修改后重新写入。

所幸，对此社区早已准备了一套完整的解决方案。我们知道，在Kubernetes体系下，管理复杂有状态应用最常用的方式就是为其编写一个专门的Operator。[Prometheus Operator](https://github.com/coreos/prometheus-operator)作为社区最早实现的Operator之一，大大简化了Prometheus的配置部署流程。Prometheus Operator将Prometheus相关的概念都抽象为了CRD。与本文相关的主要是Prometheus和PrometheusRule这两个CRD。

apiVersion: monitoring.coreos.com/v1

kind: Prometheus

metadata:

name: prometheus

spec:

ruleSelector:

matchLabels:

role: alert-rules

---

apiVersion: monitoring.coreos.com/v1

kind: PrometheusRule

metadata:

labels:

role: alert-rules

name:

spec:

groups:

- name: example

rules:

- alert: HighRequestLoad

expr: rate(http\_request\_total{pod="p1"}[5m]) > 1000

for: 1m

labels:

severity: none

annotations:

info: High Request Load

上面展示的就是近乎最简的Prometheus和PrometheusRule资源对象。当上述yaml文件被提交至Kubernetes APIServer之后，Prometheus Operator会马上同步到并根据Prometheus的配置生成一个StatefulSet用于运行Prometheus Server实例，同时将Prometheus中的配置写入Server的配置文件中。对于PrometheusRule，我们可以发现它的内容与上面的告警规则文件是基本一致的。Prometheus Operator会依据PrometheusRule的内容生成相应的ConfigMap并将其以Volume的形式挂载到Prometheus Server所在Pod的对应目录中。最终一个PrometheusRule资源对象对应一个挂载目录中的告警规则文件。

那么Operator是如何将Prometheus和PrometheusRule关联在一起的呢？类似于Service通过Selector字段指定关联的Pod。Prometheus也通过ruleSelector字段指定了一组label，Operator会将任何包含这些label的PrometheusRule都整合到一个ConfigMap(若超出单个ConfigMap的限制，则生成多个)并挂载到Prometheus对应的StatefulSet的各个Pod实例中。因此，在Prometheus Operator的帮助下，对于Prometheus告警规则进行增删改查的难度已经退化到对Kubernetes资源对象的CRUD操作，整个过程中最为繁琐的部分已经完全被Operator自动化了。事实上，Prometheus Server的高级配置乃至AlertManager的部署都可以通过Prometheus Operator提供的CRD轻松实现，因为与本文关联不大，所以不再赘述了。

最后，虽然Operator能保证对于PrometheusRule的增删改查能及时反映到相应的ConfigMap中，而Kubernetes本身则保证了ConfigMap的修改也最终能同步到相应Pod的挂载文件中，但是Prometheus Server并不会监听告警规则文件的变更。因此，我们需要以Sidecar的形式将[ConfigMap Reloader](https://github.com/jimmidyson/configmap-reload)部署在Prometheus Server所在的Pod内。由它来监听告警规则所在ConfigMap的变更，一旦监听到变化，它就会调用Prometheus Server提供的Reload接口，触发Prometheus对于配置的重新加载。

### **4. 告警实例结构**

AlertManager本质上是一个HTTP Server用于接受并处理来自Client的告警实例。Client一般都为Prometheus Server，但是任何程序只要能构造出符合标准的告警实例，都能通过POST方法将它们提交至AlertManger进行处理。因此，在生产环境中，对于无法利用Prometheus时序数据生成的告警，例如对于Kubernetes中的Event，我们也可以通过适当的构造，将其发送至AlertManager进行统一处理。告警实例的结构如下：

[

{

"labels": {

"alertname": "<requiredAlertName>",

"<labelname>": "<labelvalue>",

...

},

"annotations": {

"<labelname>": "<labelvalue>",

},

"startsAt": "<rfc3339>",

"endsAt": "<rfc3339>",

"generatorURL": "<generator\_url>"

},

...

]

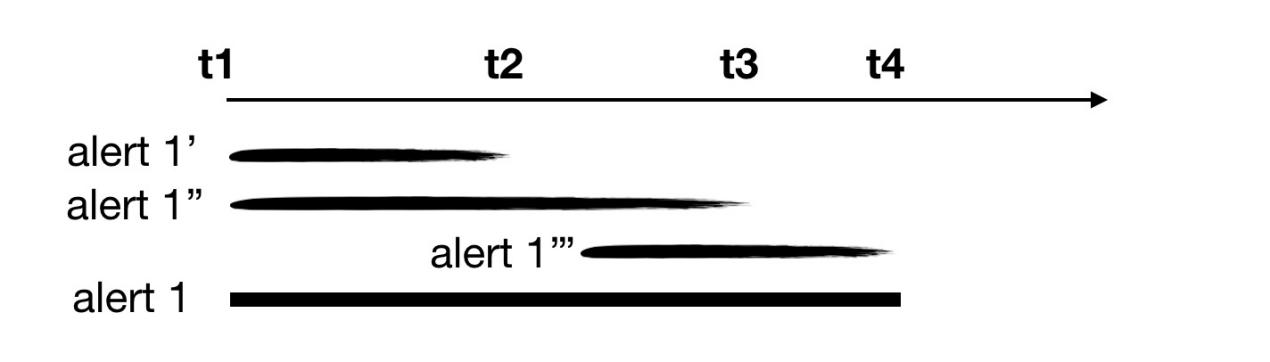
labels和annotations字段在前文已经有所提及：labels用于唯一标识一个告警，AlertManger会对labels完全相同的告警实例进行压缩聚合操作。annotations是一些类似于告警详情等的附加信息。这里我们重点关注startsAt和endsAt这两个字段，这两个字段分别表示告警的起始时间和终止时间，不过两个字段都是可选的。当AlertManager收到告警实例之后，会分以下几类情况对这两个字段进行处理：

1. 两者都存在：不做处理
2. 两者都为指定：startsAt指定为当前时间，endsAt为当前时间加上告警持续时间，默认为5分钟
3. 只指定startsAt：endsAt指定为当前时间加上默认的告警持续时间
4. 只指定endsAt：将startsAt设置为endsAt

AlertManager一般以当前时间和告警实例的endsAt字段进行比较用以判断告警的状态：

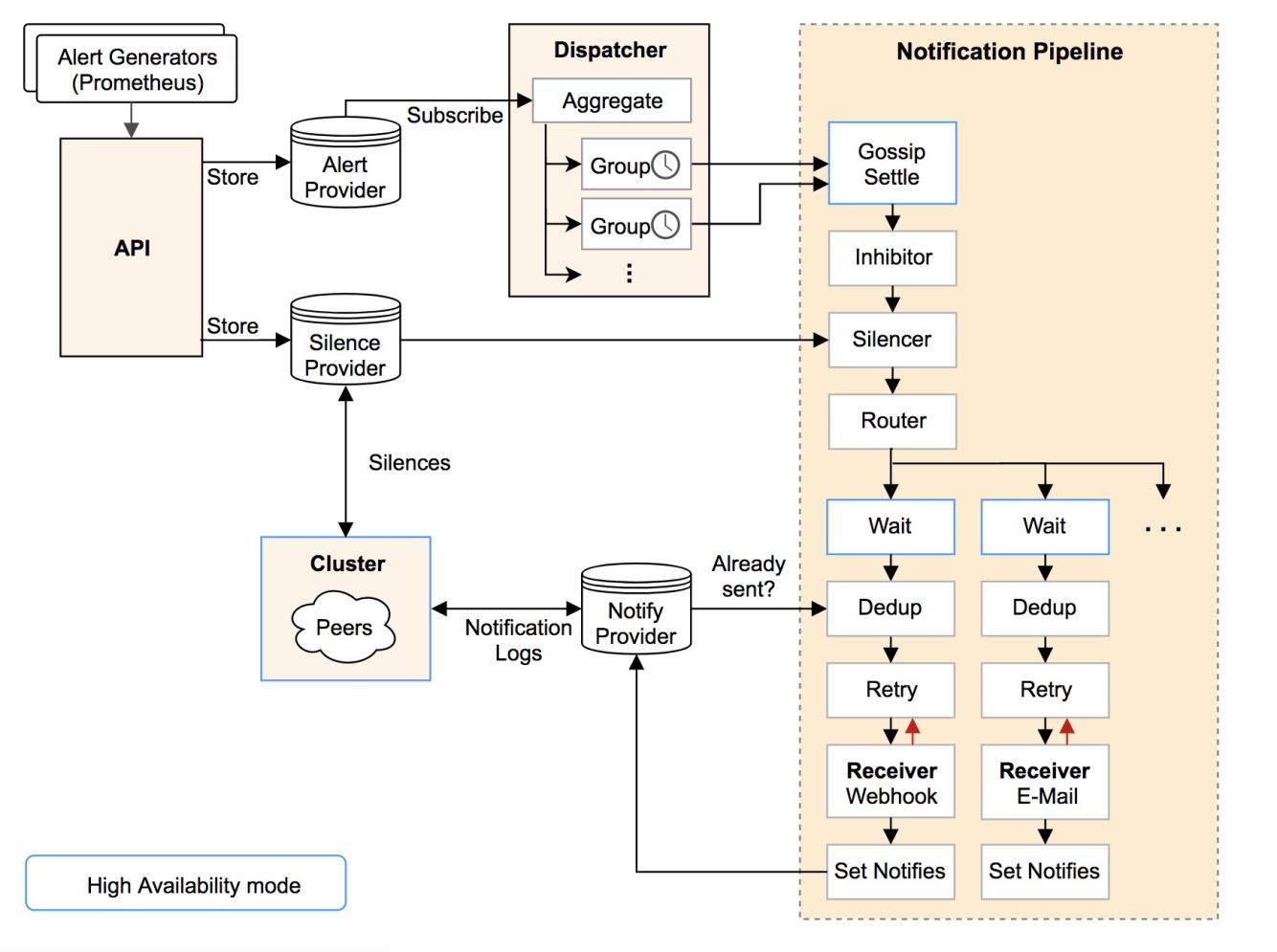
1. 若当前时间位于endsAt之前，则表示告警仍然处于触发状态（firing）
2. 若当前时间位于endsAt之后，则表示告警已经消除（resolved）

另外，当Prometheus Server中配置的告警规则被持续满足时，默认会每隔一分钟发送一个告警实例。显然，这些实例除了startsAt和endsAt字段以外都完全相同（其实Prometheus Server会将所有实例的startsAt设置为告警第一次被触发的时间）。最终，这些实例都会以如下图所示的方式进行压缩去重：

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/alertmanager/alerts.jpg)

三条最终labels相同的告警最终被压缩聚合为一条告警。当我们进行查询时，只会得到一条起始时间为t1，结束时间为t4的告警实例。

### **5. AlertManager架构概述**

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/alertmanager/alertmanager.jpg)

AlertManager本质上来说是一个强大的告警分发过滤器。所有告警统一存放在Alert Provider中，Dispatcher则会对其中的告警进行订阅。每当AlertManager接受到新的告警实例就会先在Alert Provider进行存储，之后立刻转发到Dispatcher中。Dispatcher则定义了一系列的路由规则将告警发送到预定的接收者。而告警在真正发送到接收者之前，还需要经过一系列的处理，即图中的Notification Pipeline，例如对相关告警在时间和空间维度进行聚合，对用户指定的告警进行静默，检测当前告警是否被已经发送的告警抑制，甚至在高可用模式下检测该告警是否已由集群中的其他节点发送。而这一切的操作的最终目的，都为了让能让接收者准确接受到其最关心的告警信息，同时避免告警的冗余重复。

### **6. Alert Provider**

所有进入AlertManager的告警实例都会首先存储在Alert Provider中。Alert Provider本质上是一个内存中的哈希表，用于存放所有的告警实例。因为labels唯一标识了一个告警，因此哈希表的key就是告警实例的label取哈希值，value则为告警实例的具体内容。若新接受到的告警实例在哈希表中已经存在且两者的[startsAt, endsAt]有重合，则会先将两者进行合并再刷新哈希表。同时，Alert Provider提供了订阅接口，每当接收到新的告警实例，它都会在刷新哈希表之后依次发送给各个订阅者。

值得注意的是，Alert Provider是存在GC机制的。默认每隔30分钟就会对已经消除的告警（即endsAt早于当前时间）进行清除。显然，AlertManager从实现上来看并不支持告警的持久化存储。已经消除的告警会定时清除，由于存储在内存中，若程序重启则将丢失所有告警数据。但是如果研读过AlertManager的代码，对于Alert Provider的实现是做过良好的封装的。我们完全可以实现一套底层存储基于MySQL，ElasticSearch或者Kafka的Alert Provider，从而实现告警信息的持久化（虽然AlertManager并不提供显式的插件机制，只能通过hack代码实现）。

### **7. 告警的路由与分组**

将所有告警统一发送给所有人显然是不合适的。因此AlertManager允许我们按照如下规则定义一系列的接收者并制定路由策略将告警实例分发到对应的目标接收者：

global:

// 所有告警统一从此处进入路由表

route:

// 根路由

receiver: ops-mails

group\_by: ['cluster', 'alertname']

group\_wait: 30s

group\_interval: 5m

repeat\_interval: 5h

routes:

// 子路由1

- match\_re:

service: ^(foo1|foo2|baz)$

receiver: team-X-webhook

// 子路由2

- match:

service: database

receiver: team-DB-pager

// 接收者

receivers:

- name: 'ops-mails'

email\_configs:

- to: 'ops1@example.org, ops2@example.com'

- name: 'team-X-webhook'

webhook\_configs:

- url: 'http://127.0.0.1:8080/webhooks'

- name: 'team-DB-pager'

pagerduty\_configs:

- routing\_key: <team-DB-key>

上述AlertManager的配置文件中定义了一张路由表以及三个接收者。AlertManager已经内置了Email，Slack，微信等多种通知方式，如果用户想要将告警发送给内置类型以外的其他信息平台，可以将这些告警通过webhook接口统一发送到webhook server，再由其转发实现。AlertManager的路由表整体上是一个树状结构，所有告警实例进入路由表之后会进行深度优先遍历，直到最终无法匹配并发送至父节点的Receiver。

需要注意的是路由表的根节点默认匹配所有告警实例，示例中根节点的receiver是ops-mails，表示告警默认都发送给运维团队。路由表的匹配规则是根据labels的匹配实现的。例如，对于子路由1，若告警包含key为service的label，且label的value为foo1, foo2或者baz，则匹配成功，告警将发送至team X。若告警包含service=database的label，则将其发送至数据库团队。

有的时候，作为告警的接收者，我们希望相关的告警能统一通过一封邮件进行发送，一方面能减少同类告警的重复，另一方面也有利于我们对告警进行归档。AlertManager通过Group机制对这一点做了很好的支持。每个路由节点都能配置以下四个字段对属于本节点的告警进行分组（若当前节点未显式声明，则继承父节点的配置）：

1. **group\_by：**指定一系列的label键值作为分组的依据，示例中利用cluster和alertname作为分组依据，则同一集群中，所有名称相同的告警都将统一通知。若不想对任何告警进行分组，则可以将该字段指定为'...'
2. **group\_wait：**当相应的Group从创建到第一次发送通知的等待时间，默认为30s，该字段的目的为进行适当的等待从而在一次通知中发送尽量多的告警。在每次通知之后会将已经消除的告警从Group中移除。
3. **group\_interval：**Group在第一次通知之后会周期性地尝试发送Group中告警信息，因为Group中可能有新的告警实例加入，本字段为该周期的时间间隔
4. **repeat\_interval：**在Group没有发生更新的情况下重新发送通知的时间间隔

综上，AlertManager的Dispatcher会将新订阅得到的告警实例根据label进行路由并加入或者创建一个新的Group。而新建的Group经过指定时间间隔会将组中的告警实例统一发送并周期性地检测组内是否有新的告警加入（或者有告警消除，但需要显式配置），若是则再次发送通知。另外每隔repeat\_interval，即使Group未发生变更也将再次发送通知。

### **8. Alert Notification Pipeline**

通常来说，一个Group中会包含多条告警实例，但是并不是其中的所有告警都是用户想要看到的。而且已知Group都会周期性地尝试发送其包含的告警，如果没有新的告警实例加入，在一定时间内，显然没有再重复发送告警通知的必要，另外如果对AlertManager进行高可用部署的话，多个AlertManager之间也需要做好协同，避免重复告警。如上文中AlertManager的整体架构图所示，当Group尝试发送告警通知时，总是先要经过一条Notification Pipeline的过滤，最终满足条件的告警实例才能通过邮件的方式发出。一般过滤分为抑制（Inhibit），静默（silence）以及去重（dedup）三个步骤。下面我们将逐个进行分析。

#### **8.1 告警抑制**

所谓的**告警抑制**其实是指，当某些告警已经触发时，则不再发送其他受它抑制的告警。一个典型的使用场景为：如果产生了一条集群不可用的告警，那么任何与该集群相关的告警都应当不再通知给用户，因为这些告警都是由集群不可用引起的，发送它们只会增加用户找到问题根因的难度。告警的抑制规则会配置在AlertManager全局的配置文件中，如下所示：

inhibit\_rules:

- source\_match:

alertname: ClusterUnavailable

severity: critical

target\_match:

severity: critical

equal:

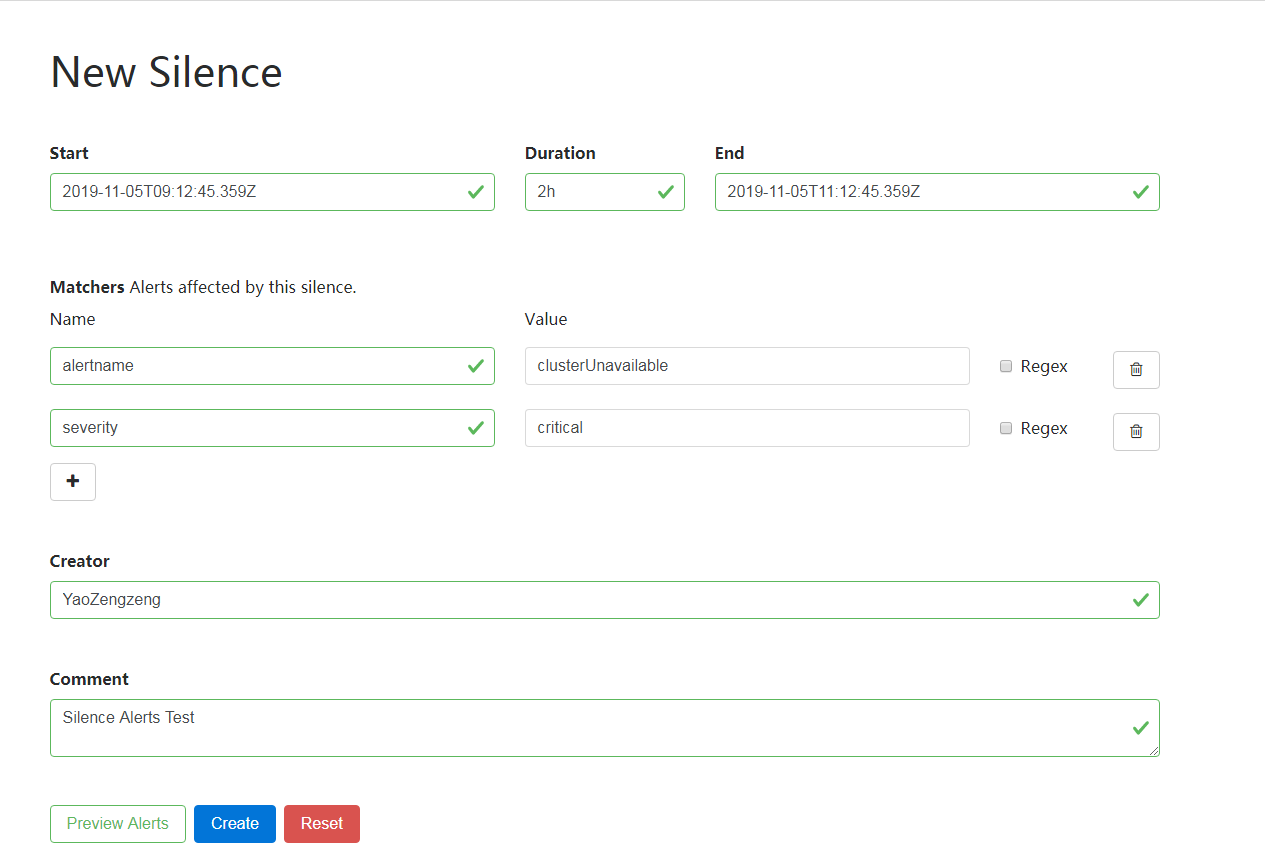
- cluster

该配置的含义为，若出现了包含label为{alertname="ClusterUnavailable", severity="critical"}的告警实例A，AlertManager就会对其进行记录。当后续出现告警实例包含label为{severity="critical"}且"cluster"这个label对应的value和A的“cluster”对应的value相同，则该告警实例被抑制，不再发送。

当Group每次尝试发送告警实例时，AlertManager都会先用抑制规则筛选掉满足条件的实例，剩余的实例才能进入Notification Pipeline的下一个步骤，即告警静默。

#### **8.2 告警静默**

**告警静默**指的是用户可以选择在一段时间内不接收某些告警。与Inhibit rule不同的是，静默规则可以由用户动态配置，AlertManager甚至提供了如下所示的图形UI：

[](https://github.com/YaoZengzeng/KubernetesResearch/blob/master/pic/alertmanager/silence.png)

与告警自身的定义方式类似，静默规则也是在一个时间段内起作用，需用明确指定开始时间与结束时间。而静默规则同样通过指定一组label来匹配作用的告警实例。例如在上图的例子中，任何包含label为{alertname="clusterUnavailable", severity="critical"}的告警实例都将不再出现在通知中。

显然，静默规则又将滤去一部分告警实例，如果此时Group中仍有剩余的实例，则将进入Notification的下一步骤，告警去重。

#### **8.3 告警去重**

每当一个Group第一次成功发送告警通知之后，AlertManager就会为其创建一个Notification Log（简称nflog），其结构如下：

e := &pb.MeshEntry{

Entry: &pb.Entry{

Receiver; r,

GroupKey: []byte(gkey),

Timestamp: now,

FiringAlerts: firingAlerts,

ResolvedAlerts: resolvedAlerts,

},

ExpiredAt: now.Add(l.retention)

}

可以看到，每个Notification Log中包含：

1. 该Group的Key（即该Group用于筛选Alerts的labels的哈希值）
2. 该Group对应的Receiver
3. 该Notification Log创建的时间
4. 该Group中正在触发的各个告警实例的哈希值
5. 该Group中各个已经消除的告警实例的哈希值
6. 该Notification Log过期的时间，默认为120小时

当Group再次周期性地尝试推送通知并经过抑制和静默的两层筛选之后，若仍然有告警实例存在，则会进入告警去重阶段。首先找到该Group对应的Notification Log并只在以下任一条件满足的时候发送通知：

1. Group剩余的告警实例中，处于触发状态的告警实例不是Notification Log中的FiringAlerts的子集，即有新的告警实例被触发
2. Notification Log中FiringAlerts的数目不为零，但是当前Group中处于触发状态的告警实例数为0，即Group中的告警全部被消除了
3. Group中已消除的告警不是Notification Log中ResolvedAlerts的子集，说明有新的告警被消除，且通知配置中设置对于告警消除进行通知。例如，Email默认不在个别告警实例消除时通知而Webhook则默认会进行通知。

综上，通过Notification Pipeline通过对告警的抑制，静默以及去重确保了用户能够专注于真正重要的告警而不会被过多无关的或者重复的告警信息所困扰。

### **9. 高可用**

当真正部署到生产环境中，如果只部署单个实例的AlertManager显然是无法满足可用性的。因此AlertManager原生支持多实例的部署方式并用Gossip协议来同步实例间的状态。因为AlertManager并非是无状态的，它有如下两个关键信息需要同步：

1. 告警静默规则：当存在多个AlertManager实例时，用户依然只会向其中一个实例发起请求，对静默规则进行增删。但是对于静默规则的应用显然应当是全局的，因此各个实例应当广播各自的静默规则，直到全局一致。
2. Notification Log：既然要保证高可用，即确保告警实例不丢失，而AlertManager实例又是将告警保存在各自的内存中的，因此Prometheus显然不应该在多个AlertManager实例之间做负载均衡而是应该将告警发往所有的AlertManager实例。但是对于同一个Alert Group的通知则只能由一个AlertManager发送，因此我们也应该把Notification Log在全集群范围内进行同步。

当以集群模式运行AlertManager时，AlertManager的命令行参数配置如下：

--cluster.listen-address=0.0.0.0:9094

--cluster.peer=192.168.1.1:9094

--cluster.peer=192.168.1.2:9094

当AlertManager启动时，它会首先从cluster.peer参数指定的地址和端口进行Push/Pull：即首先将本节点的状态信息（全部的Silence以及Notification Log）发送到对端，再从对端拉取状态信息并与本节点的状态信息合并：例如，对于从对端拉取到的静默规则，如果有本节点不存在的规则则直接添加，若是规则在本节点已存在但是更新时间更晚，则用对端规则覆盖已有的规则。对于Notification Log的做法类似。最终，集群中的所有AlertManager都会有同样的静默规则以及Notification Log。

如果此时用户在某个AlertManager请求增加新的静默规则呢？根据Gossip协议，该实例应该从集群中选取几个实例，将新增的静默规则发送给它们。而当这些实例收到广播信息时，一方面它会合并这一新的静默规则同时再对其进行广播。最后，整个集群都会接收到这一新添加的静默规则，实现了最终一致性。

不过，Notification Log的同步并没有静默规则这么容易。我们可以假设如下场景：由于高可用的要求，Prometheus会向每个AlertManager发送告警实例。如果该告警实例不属于任何之前已有的Alert Group，则会新建一个Group并最终创建一个相应的Notification Log。而Notification Log是在通知完成之后创建的，所以在这种情况下，针对同一个告警发送了多次通知。

为了避免这种情况的发生，社区给出的解决方案是错开各个AlertManager发送通知的时间。如上文的整体架构图所示，Notification Pipeline在进行去重之前其实还有一个Wait阶段。该阶段会将对于告警的通知处理暂停一段时间，不同的AlertManager实例等待的时间会因为该实例在整个集群中的位置有所不同。根据实例名进行排序，排名每靠后一位，默认多等待15秒。

假设集群中有两个AlertManager实例，排名靠前的实例为A0，排名靠后的实例为A1，此时对于上述问题的处理如下：

1. 假设两个AlertManager同时收到告警实例并同时到达Notification Pipeline的Wait阶段。在该阶段A0无需等待而A1需要等待15秒。
2. A0直接发送通知，生成相应的Notification Log并广播
3. A1等待15秒之后进入去重阶段，但是由于已经同步到A0广播的Notification Log，通知不再发送

可以看到，Gossip协议事实上是一个弱一致性的协议，上述的机制能在绝大多数情况下保证AlertManager集群的高可用并且避免实例间同步的不及时对用户造成的困扰。但是仍然有待在严苛生产环境下的进一步验证，所幸的是，告警数据的强一致性并不是那么敏感。

### **10. 总结**

本文对基于Prometheus的告警系统进行了较为详尽的分析：包括从告警规则在Prometheus Server的配置，Prometheus Server对告警规则的评估并触发告警实例发送至AlertManager，AlertManager的整体架构以及AlertManager对于告警实例的处理。可以看到，虽然执行链路基本完备，但是与Prometheus的监控模型已经成为事实标准相比，整个Prometheus告警模型的通用性和实用性仍然是存疑的，以笔者经验来看，如果要真正应用到生产环境中还需要做大量的适配与增强。