# 关于prometheus设计的一些思考

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/71402695

其实设计一套完整的监控系统是挺复杂的，既要考虑的通用性，也要考虑的对各种指标和监控对象的特殊性，业内也并没有规范可言，还有考虑的数据的存储和持久化。prometheus的设计初衷就就是一个通用监控系统，它并没有设计集群，类似HDFS一套东西去存储数据，它的设计初衷就没有想过存储大量的持久的监控数据，如果你想做报表的话，它可能不适合，它更适合作为一个通用的实时的监控系统。下面针对它的设计理念的一些思考：

它是什么

它首先是一个监控系统，采集和存储监控指标，支持指标的查询和指标告警，当然还有自己的一个dashboard。

它的侧重点

它的重点是[**操作系统**](http://lib.csdn.net/base/operatingsystem)和云环境的监控   
它不不适合日志和时间采集存储，请求的跟踪，持久的存储，没有用户和鉴权，不能够自动水平伸缩。

为什么不做多租户

这个问题，想必大家都很关心，这个通用的监控系统，为啥没有多租户的设计呢！因为做多租户就要TSL、鉴权之类的东西，这样会导致监控的exporter将会很复杂，并且多租户的隔离完全可以在系统之外用户自己去维护。

Label 对比 hierarchy

这是设计理念的对比，label的优势不言而喻，在kubernetes的资源对象的关联都是通过label去完成，它相比简单层级结构要更加的灵活、高效、多维度。

promsql 对比 sql

prometheus为啥不设计一套兼容标准sql的CURD呢？应为要考虑到标准sql的解析等，这样会降低性能，并且标准sql写起来很麻烦，如果你用过promsql就会看出它的优势，能够更好的做指标计算，prometheus认为监控数据只读就可以了，所以promsql只支持读。

pull 对比 push

这个是从[**架构**](http://lib.csdn.net/base/architecture)上面思考，pull是自己去拉，push是推送。这个设计对比和之前rabbitmq和kafka的设计形成对照。到底应该推还是拉。prometheus选择拉，拉的优势是能够自动的上游监控、水平监控、更加灵活、更容易实现高可用、更少的配置、更容易扩展。展开来说就是拉的方式，降低耦合，在推送的系统中很容易出现因为向监控系统推送数据失败而导致系统瘫痪的问题，通过pull，独立于被监控系统之外，这样数据的采集存储能够更加可控，被采集端无须感知监控系统的存在。

无集群

我并不否认集群的好处，能够更大规模的采集和存储数据。但这这样设计会将整个系统复杂化，尤其告警变的复杂，可能导致数据的一致性性问题，prometheus专注实时监控。

relabel

relabel是为了系统兼容而设计的，在环境中监控的对象不同，针对不同的监控对象target，要能够改变监控查询的方式。数据存储使用float64，这样在保证精度的情况下，或得较好压缩。

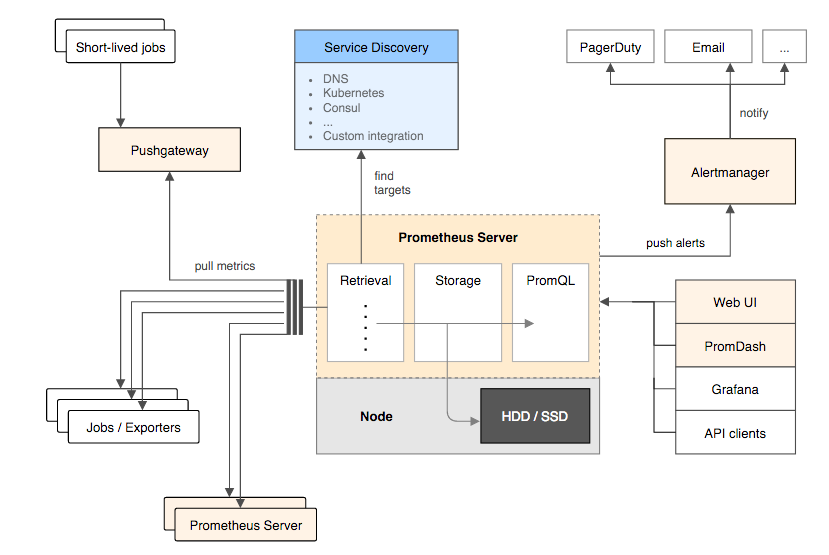
exporter

这个一个设计模式的问题，一个exporter是功能大而全，还是简单独立好呢？prometheus坚持的是，第一单一职责划分架构简单；第二是避免SPOF（Single Point Of Failure）；第三是操作瓶颈，一个大而全的采集器可能要处理更大的数据；第四，单一功能的exporter可以支持不同的语言编写，这样系统变的更加通用，只要支持prometheus风格的metric都可以被监控。

# Prometheus 实战于源码分析之部署

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70866151

Prometheus是一个通用的监控平台，它可以kubernetes结合，监控容器和主机的性能，由于它是kubernetes之前就已经有了，并不是一个专门设计用来监控kubernetes的，这一点和heapster最大差别。prometheus目标是通用，所以它不和某一个平台绑定。它的设计思想也很有意思，它是通过pull，这种设计好处是最少的降低和被监控对象的耦合，当prometheus挂掉后不会影响到被监控对象，一个经典的图片还是要和大家先分享一下   
  
简单介绍一下基本组件，后面代码再详细分析   
pushgateway是一个网关，刚才说到prometheus设计理念是pull，如果你非要push当然可以就可以通过这个gateway；exporters是各种监控agent，譬如监控主机的叫做node exporter，还有很多其他的；discovery是一个服务发现，可以通过这些服务发现动态增加监控对象；stoage是一个内存+磁盘的存储系统，负责保存监控数据，alertmanager是告警统统，当设定告警规则后如果匹配产生告警会push到里面；其它还是之前promsql查询以及webui展现。   
我们先来安装使用一下：   
可以通过源码编译也可以通过下载二进制包，还可以通过[**Docker**](http://lib.csdn.net/base/docker)启动，如果是源码编译很简单，clone下代码make build一下就行，会产生二进制文件prometheus，如果是最新代码请选择go1.8.1以上版本。

make build

>> fetching promu

>> building binaries

> prometheus

> promtool

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

如果是使用容器就更简单，直接拉取镜像运行：

docker run -d --name=prometheus --publish=9090:9090 -v /etc/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml -v /var/prometheus/storage:/prometheus prom/prometheus

* 1
* 1

这里用到一个配置文件prometheus.yml 简单说一下

global:

scrape\_interval: 15s

external\_labels:

monitor: 'codelab-monitor'

scrape\_configs:

- job\_name: 'node'

scrape\_interval: 5s

static\_configs:

- targets: ['x.x.x.x:9100']

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

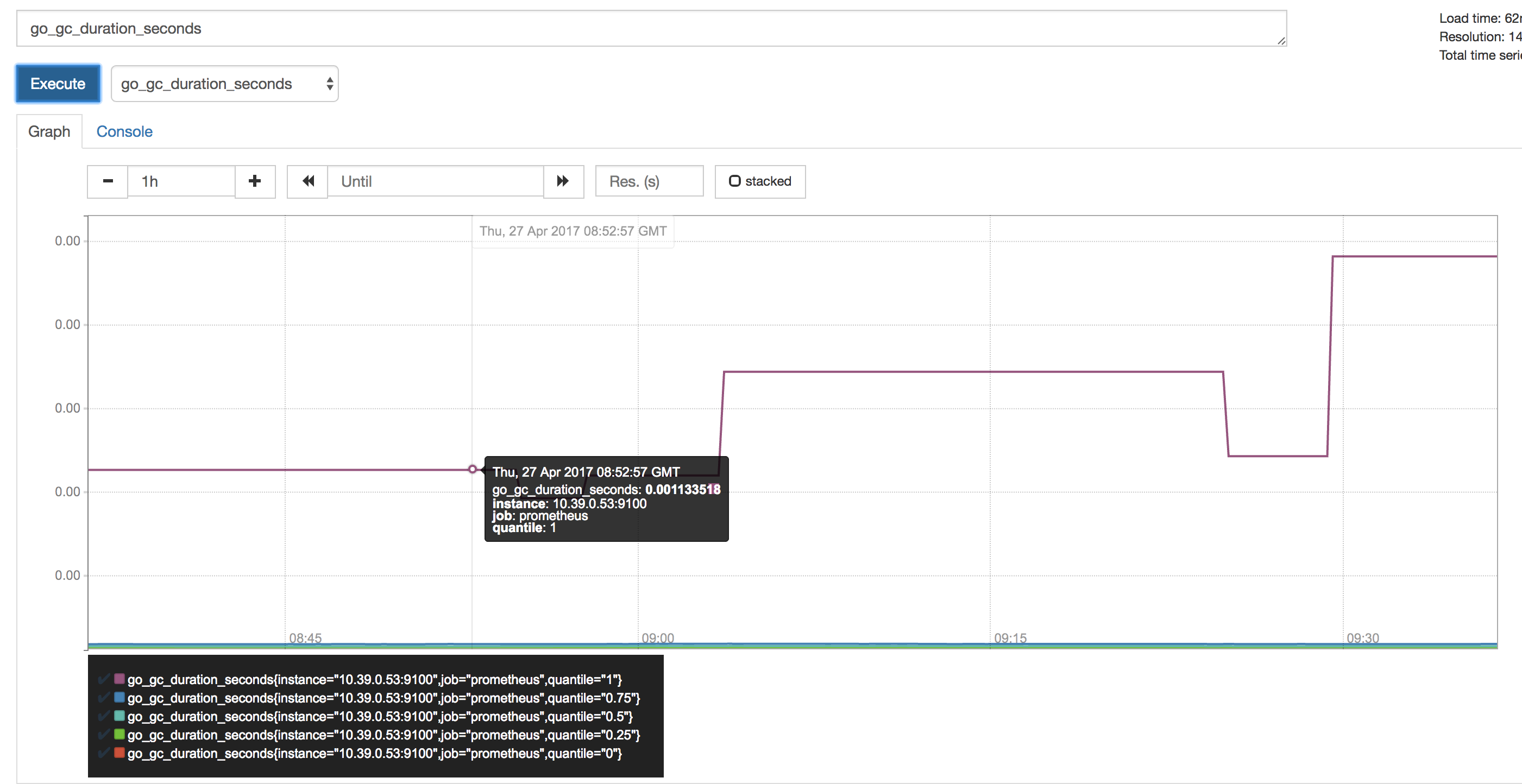
- targets: ['localhost:9090']

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

很简单就是定义一个全局配置和一个特定配置，至于下面的targets是什么呢？它就是上面说的exporter，这边的x.x.x.x使用的是一个主机的node exporter，localhost是prometheus系统自身监控。

docker run -d --name=ne -p 9100:9100 prom/node-exporter

* 1
* 1

负责采集主机性能数据当然你可以直接通过他的9100端口访问其采集的性能数据   
那么prometheus也是一样，配置这些target，prometheus会按照你设定的间隔时间去pull数据保存起来，看一下自带ui的效果，是一个[**Go**](http://lib.csdn.net/base/go) gc持续时间的图：   
  
这一篇基本介绍清楚了prometheus的基本操作和原理。

# Prometheus 实战于源码分析之storage

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70980587

## 远端存储

prometheus不仅支持本地存储还支持远端存储，先从远端存储说起，他是通过一个发送队列queue完成数据发送的。先看一下队列的定义：

func NewQueueManager(cfg QueueManagerConfig) \*QueueManager {

if cfg.QueueCapacity == 0 {

cfg.QueueCapacity = defaultQueueCapacity

}

if cfg.MaxShards == 0 {

cfg.MaxShards = defaultMaxShards

}

if cfg.MaxSamplesPerSend == 0 {

cfg.MaxSamplesPerSend = defaultMaxSamplesPerSend

}

if cfg.BatchSendDeadline == 0 {

cfg.BatchSendDeadline = defaultBatchSendDeadline

}

t := &QueueManager{

cfg: cfg,

queueName: cfg.Client.Name(),

logLimiter: rate.NewLimiter(logRateLimit, logBurst),

numShards: 1,

reshardChan: make(chan int),

quit: make(chan struct{}),

samplesIn: newEWMARate(ewmaWeight, shardUpdateDuration),

samplesOut: newEWMARate(ewmaWeight, shardUpdateDuration),

samplesOutDuration: newEWMARate(ewmaWeight, shardUpdateDuration),

}

t.shards = t.newShards(t.numShards)

numShards.WithLabelValues(t.queueName).Set(float64(t.numShards))

queueCapacity.WithLabelValues(t.queueName).Set(float64(t.cfg.QueueCapacity))

return t

}

这个队列的最大分片是1000，每个分片没秒1000个sample，那么一秒就可以发送1000\*1000个sample。每一种存储，无论是本地存储还有远端存储，写数据都实现Append方法，remote的也一样，在romte的Append就调用了queue的Append方法。

func (t \*QueueManager) Append(s \*model.Sample) error {

var snew model.Sample

snew = \*s

snew.Metric = s.Metric.Clone()

for ln, lv := range t.cfg.ExternalLabels {

if \_, ok := s.Metric[ln]; !ok {

snew.Metric[ln] = lv

}

}

snew.Metric = model.Metric(

relabel.Process(model.LabelSet(snew.Metric), t.cfg.RelabelConfigs...))

if snew.Metric == nil {

return nil

}

t.shardsMtx.Lock()

enqueued := t.shards.enqueue(&snew)

t.shardsMtx.Unlock()

if enqueued {

queueLength.WithLabelValues(t.queueName).Inc()

} else {

droppedSamplesTotal.WithLabelValues(t.queueName).Inc()

if t.logLimiter.Allow() {

log.Warn("Remote storage queue full, discarding sample. Multiple subsequent messages of this kind may be suppressed.")

}

}

return nil

}

通过enqueued := t.shards.enqueue(&snew)发到队列里面，

func (s \*shards) enqueue(sample \*model.Sample) bool {

s.qm.samplesIn.incr(1)

fp := sample.Metric.FastFingerprint()

shard := uint64(fp) % uint64(len(s.queues))

select {

case s.queues[shard] <- sample:

return true

default:

return false

}

}

这个里面是简单的求余数分组的方法，如果这里使用一致hash会不会更好点呢！把数据发动到分片的队列中。QueueManager启动的时候就启动了队列发送任务

func (s \*shards) start() {

for i := 0; i < len(s.queues); i++ {

go s.runShard(i)

}

}

继续看runShard

func (s \*shards) runShard(i int) {

defer s.wg.Done()

queue := s.queues[i]

// Send batches of at most MaxSamplesPerSend samples to the remote storage.

// If we have fewer samples than that, flush them out after a deadline

// anyways.

pendingSamples := model.Samples{}

for {

select {

case sample, ok := <-queue:

if !ok {

if len(pendingSamples) > 0 {

log.Debugf("Flushing %d samples to remote storage...", len(pendingSamples))

s.sendSamples(pendingSamples)

log.Debugf("Done flushing.")

}

return

}

queueLength.WithLabelValues(s.qm.queueName).Dec()

pendingSamples = append(pendingSamples, sample)

for len(pendingSamples) >= s.qm.cfg.MaxSamplesPerSend {

s.sendSamples(pendingSamples[:s.qm.cfg.MaxSamplesPerSend])

pendingSamples = pendingSamples[s.qm.cfg.MaxSamplesPerSend:]

}

case <-time.After(s.qm.cfg.BatchSendDeadline):

if len(pendingSamples) > 0 {

s.sendSamples(pendingSamples)

pendingSamples = pendingSamples[:0]

}

}

}

}

具体发送样本的方法还要看里面的sendSamples

func (s \*shards) sendSamples(samples model.Samples) {

// Samples are sent to the remote storage on a best-effort basis. If a

// sample isn't sent correctly the first time, it's simply dropped on the

// floor.

begin := time.Now()

err := s.qm.cfg.Client.Store(samples)

duration := time.Since(begin)

if err != nil {

log.Warnf("error sending %d samples to remote storage: %s", len(samples), err)

failedSamplesTotal.WithLabelValues(s.qm.queueName).Add(float64(len(samples)))

} else {

sentSamplesTotal.WithLabelValues(s.qm.queueName).Add(float64(len(samples)))

}

sentBatchDuration.WithLabelValues(s.qm.queueName).Observe(duration.Seconds())

s.qm.samplesOut.incr(int64(len(samples)))

s.qm.samplesOutDuration.incr(int64(duration))

}

最终通过Store方法发送数据

func (c \*Client) Store(samples model.Samples) error {

req := &WriteRequest{

Timeseries: make([]\*TimeSeries, 0, len(samples)),

}

for \_, s := range samples {

ts := &TimeSeries{

Labels: make([]\*LabelPair, 0, len(s.Metric)),

}

for k, v := range s.Metric {

ts.Labels = append(ts.Labels,

&LabelPair{

Name: string(k),

Value: string(v),

})

}

ts.Samples = []\*Sample{

{

Value: float64(s.Value),

TimestampMs: int64(s.Timestamp),

},

}

req.Timeseries = append(req.Timeseries, ts)

}

data, err := proto.Marshal(req)

if err != nil {

return err

}

buf := bytes.Buffer{}

if \_, err := snappy.NewWriter(&buf).Write(data); err != nil {

return err

}

httpReq, err := http.NewRequest("POST", c.url.String(), &buf)

if err != nil {

return err

}

httpReq.Header.Add("Content-Encoding", "snappy")

ctx, \_ := context.WithTimeout(context.Background(), c.timeout)

httpResp, err := ctxhttp.Do(ctx, c.client, httpReq)

if err != nil {

return err

}

defer httpResp.Body.Close()

if httpResp.StatusCode/100 != 2 {

return fmt.Errorf("server returned HTTP status %s", httpResp.Status)

}

return nil

}

## 本地存储

Store里面就是通过POST方式发送数据。说完了远端存储，再解释一下本地存储，这个设计的挺复杂，它是先放到内存中，并会批量将内存数据导入到磁盘中保存，具体看内存存储管理

type MemorySeriesStorage struct {

// archiveHighWatermark and numChunksToPersist have to be aligned for atomic operations.

archiveHighWatermark model.Time // No archived series has samples after this time.

numChunksToPersist int64 // The number of chunks waiting for persistence.

maxChunksToPersist int // If numChunksToPersist reaches this threshold, ingestion will be throttled.

rushed bool // Whether the storage is in rushed mode.

rushedMtx sync.Mutex // Protects entering and exiting rushed mode.

throttled chan struct{} // This chan is sent to whenever NeedsThrottling() returns true (for logging).

fpLocker \*fingerprintLocker

fpToSeries \*seriesMap

options \*MemorySeriesStorageOptions

loopStopping, loopStopped chan struct{}

logThrottlingStopped chan struct{}

maxMemoryChunks int

dropAfter time.Duration

checkpointInterval time.Duration

checkpointDirtySeriesLimit int

persistence \*persistence

mapper \*fpMapper

evictList \*list.List

evictRequests chan chunk.EvictRequest

evictStopping, evictStopped chan struct{}

quarantineRequests chan quarantineRequest

quarantineStopping, quarantineStopped chan struct{}

persistErrors prometheus.Counter

queuedChunksToPersist prometheus.Counter

numSeries prometheus.Gauge

numHeadChunks prometheus.Gauge

dirtySeries prometheus.Gauge

seriesOps \*prometheus.CounterVec

ingestedSamplesCount prometheus.Counter

discardedSamplesCount \*prometheus.CounterVec

nonExistentSeriesMatchesCount prometheus.Counter

maintainSeriesDuration \*prometheus.SummaryVec

persistenceUrgencyScore prometheus.Gauge

rushedMode prometheus.Gauge

}

他是一个内存存储管理器。和remote一样，他也是实现了Append方法去保存sample。

func (s \*MemorySeriesStorage) Append(sample \*model.Sample) error {

for ln, lv := range sample.Metric {

if len(lv) == 0 {

delete(sample.Metric, ln)

}

}

rawFP := sample.Metric.FastFingerprint()

s.fpLocker.Lock(rawFP)

fp := s.mapper.mapFP(rawFP, sample.Metric)

defer func() {

s.fpLocker.Unlock(fp)

}() // Func wrapper because fp might change below.

if fp != rawFP {

// Switch locks.

s.fpLocker.Unlock(rawFP)

s.fpLocker.Lock(fp)

}

series, err := s.getOrCreateSeries(fp, sample.Metric)

if err != nil {

return err // getOrCreateSeries took care of quarantining already.

}

if sample.Timestamp == series.lastTime {

// Don't report "no-op appends", i.e. where timestamp and sample

// value are the same as for the last append, as they are a

// common occurrence when using client-side timestamps

// (e.g. Pushgateway or federation).

if sample.Timestamp == series.lastTime &&

series.lastSampleValueSet &&

sample.Value.Equal(series.lastSampleValue) {

return nil

}

s.discardedSamplesCount.WithLabelValues(duplicateSample).Inc()

return ErrDuplicateSampleForTimestamp // Caused by the caller.

}

if sample.Timestamp < series.lastTime {

s.discardedSamplesCount.WithLabelValues(outOfOrderTimestamp).Inc()

return ErrOutOfOrderSample // Caused by the caller.

}

completedChunksCount, err := series.add(model.SamplePair{

Value: sample.Value,

Timestamp: sample.Timestamp,

})

if err != nil {

s.quarantineSeries(fp, sample.Metric, err)

return err

}

s.ingestedSamplesCount.Inc()

s.incNumChunksToPersist(completedChunksCount)

return nil

}

这个里面先通过getOrCreateSeries获取series，series你可以理解为，相同类型的监控数据放到一起，这样便于压缩查找，通过series.add保存。但这只是保存到内存中，怎么持久化呢？   
在MemorySeriesStorage启动的时候

p, err = newPersistence(

s.options.PersistenceStoragePath,

s.options.Dirty, s.options.PedanticChecks,

syncStrategy,

s.options.MinShrinkRatio,

)

if err != nil {

return err

}

s.persistence = p

// Persistence must start running before loadSeriesMapAndHeads() is called.

go s.persistence.run()

...

go s.loop()

这个persistence负责把内存中的数据写到磁盘中，loop中

for {

select {

case <-s.loopStopping:

break loop

case fp := <-memoryFingerprints:

if s.maintainMemorySeries(fp, model.Now().Add(-s.dropAfter)) {

dirty := atomic.AddInt64(&dirtySeriesCount, 1)

s.dirtySeries.Set(float64(dirty))

// Check if we have enough "dirty" series so that we need an early checkpoint.

// However, if we are already behind persisting chunks, creating a checkpoint

// would be counterproductive, as it would slow down chunk persisting even more,

// while in a situation like that, where we are clearly lacking speed of disk

// maintenance, the best we can do for crash recovery is to persist chunks as

// quickly as possible. So only checkpoint if the urgency score is < 1.

if dirty >= int64(s.checkpointDirtySeriesLimit) &&

s.calculatePersistenceUrgencyScore() < 1 {

checkpointTimer.Reset(0)

}

}

case fp := <-archivedFingerprints:

s.maintainArchivedSeries(fp, model.Now().Add(-s.dropAfter))

}

}

maintainMemorySeries保存series，

func (s \*MemorySeriesStorage) maintainMemorySeries(

fp model.Fingerprint, beforeTime model.Time,

) (becameDirty bool) {

defer func(begin time.Time) {

s.maintainSeriesDuration.WithLabelValues(maintainInMemory).Observe(

time.Since(begin).Seconds(),

)

}(time.Now())

s.fpLocker.Lock(fp)

defer s.fpLocker.Unlock(fp)

series, ok := s.fpToSeries.get(fp)

if !ok {

// Series is actually not in memory, perhaps archived or dropped in the meantime.

return false

}

defer s.seriesOps.WithLabelValues(memoryMaintenance).Inc()

closed, err := series.maybeCloseHeadChunk()

if err != nil {

s.quarantineSeries(fp, series.metric, err)

s.persistErrors.Inc()

}

if closed {

s.incNumChunksToPersist(1)

s.numHeadChunks.Dec()

}

seriesWasDirty := series.dirty

if s.writeMemorySeries(fp, series, beforeTime) {

// Series is gone now, we are done.

return false

}

iOldestNotEvicted := -1

for i, cd := range series.chunkDescs {

if !cd.IsEvicted() {

iOldestNotEvicted = i

break

}

}

// Archive if all chunks are evicted. Also make sure the last sample has

// an age of at least headChunkTimeout (which is very likely anyway).

if iOldestNotEvicted == -1 && model.Now().Sub(series.lastTime) > headChunkTimeout {

s.fpToSeries.del(fp)

s.numSeries.Dec()

s.persistence.archiveMetric(fp, series.metric, series.firstTime(), series.lastTime)

s.seriesOps.WithLabelValues(archive).Inc()

oldWatermark := atomic.LoadInt64((\*int64)(&s.archiveHighWatermark))

if oldWatermark < int64(series.lastTime) {

if !atomic.CompareAndSwapInt64(

(\*int64)(&s.archiveHighWatermark),

oldWatermark, int64(series.lastTime),

) {

panic("s.archiveHighWatermark modified outside of maintainMemorySeries")

}

}

return

}

// If we are here, the series is not archived, so check for Chunk.Desc

// eviction next.

series.evictChunkDescs(iOldestNotEvicted)

return series.dirty && !seriesWasDirty

}

writeMemorySeries把数据写到磁盘，里面再调用persistChunks

func (p \*persistence) persistChunks(fp model.Fingerprint, chunks []chunk.Chunk) (index int, err error) {

f, err := p.openChunkFileForWriting(fp)

if err != nil {

return -1, err

}

defer p.closeChunkFile(f)

if err := p.writeChunks(f, chunks); err != nil {

return -1, err

}

// Determine index within the file.

offset, err := f.Seek(0, os.SEEK\_CUR)

if err != nil {

return -1, err

}

index, err = chunkIndexForOffset(offset)

if err != nil {

return -1, err

}

return index - len(chunks), err

}

那这些series怎么查询呢？它有个index列表，通过著名的leveldb保存index，这样就可以通过index去查询了。他是一个keyvalue[**数据库**](http://lib.csdn.net/base/mysql)，接口定义storage/local/index/interface.[**Go**](http://lib.csdn.net/base/go)

type KeyValueStore interface {

Put(key, value encoding.BinaryMarshaler) error

// Get unmarshals the result into value. It returns false if no entry

// could be found for key. If value is nil, Get behaves like Has.

Get(key encoding.BinaryMarshaler, value encoding.BinaryUnmarshaler) (bool, error)

Has(key encoding.BinaryMarshaler) (bool, error)

// Delete returns (false, nil) if key does not exist.

Delete(key encoding.BinaryMarshaler) (bool, error)

NewBatch() Batch

Commit(b Batch) error

// ForEach iterates through the complete KeyValueStore and calls the

// supplied function for each mapping.

ForEach(func(kv KeyValueAccessor) error) error

Close() error

}

它的实现在storage/local/index/leveldb.go里面，代码比较多，我就不粘出来了。

# 2@Prometheus 实战于源码分析之API与联邦

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70891379

在进行源码讲解关于prometheus还有一些配置和使用，需要解释一下。首先是API的使用，prometheus提供了一套HTTP的接口

curl http://localhost:9090/api/v1/query?query=go\_goroutines|python -m json.tool

{

"data": {

"result": [

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "localhost:9090",

"job": "prometheus"

},

"value": [

1493347106.901,

"119"

]

},

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "10.39.0.45:9100",

"job": "node"

},

"value": [

1493347106.901,

"13"

]

},

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "10.39.0.53:9100",

"job": "node"

},

"value": [

1493347106.901,

"11"

]

}

],

"resultType": "vector"

},

"status": "success"

}

上面演示一个查询go\_goroutines这一个监控指标的数据。让然也可以基于开始时间和截止时间查询，但更强大的功能应该是支持OR查询

[root@slave3 ~]# curl -g 'http://localhost:9090/api/v1/series?match[]=up&match[]=process\_start\_time\_seconds{job="prometheus"}'|python -m json.tool

{

"data": [

{

"\_\_name\_\_": "up",

"instance": "10.39.0.53:9100",

"job": "node"

},

{

"\_\_name\_\_": "up",

"instance": "localhost:9090",

"job": "prometheus"

},

{

"\_\_name\_\_": "up",

"instance": "10.39.0.45:9100",

"job": "node"

},

{

"\_\_name\_\_": "process\_start\_time\_seconds",

"instance": "localhost:9090",

"job": "prometheus"

}

],

"status": "success"

}

查询一个系列的数据，当然还可以通过DELETE去删除系列。还记得上一篇说的设置job和targets了吗？也可以通过API查询

curl http://localhost:9090/api/v1/label/job/values

{"status":"success","data":["node","prometheus"]}

* 1
* 2
* 1
* 2

当然有哪些监控对象也可以查询

curl http://localhost:9090/api/v1/targets|python -m json.tool

{

"data": {

"activeTargets": [

{

"discoveredLabels": {

"\_\_address\_\_": "10.39.0.53:9100",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "node"

},

"health": "up",

"labels": {

"instance": "10.39.0.53:9100",

"job": "node"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:40.871586825Z",

"scrapeUrl": "http://10.39.0.53:9100/metrics"

},

{

"discoveredLabels": {

"\_\_address\_\_": "10.39.0.45:9100",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "node"

},

"health": "up",

"labels": {

"instance": "10.39.0.45:9100",

"job": "node"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:45.144032466Z",

"scrapeUrl": "http://10.39.0.45:9100/metrics"

},

{

"discoveredLabels": {

"\_\_address\_\_": "localhost:9090",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "prometheus"

},

"health": "up",

"labels": {

"instance": "localhost:9090",

"job": "prometheus"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:44.079111193Z",

"scrapeUrl": "http://localhost:9090/metrics"

}

]

},

"status": "success"

}

查询这些target。alertmanagers也是通过/api/v1/alertmanagers可以查询的。对应prometheus的本地存储还有一些关键的配置需要注意：   
prometheus\_local\_storage\_memory\_series：当前的系列数量在内存中保存。   
prometheus\_local\_storage\_open\_head\_chunks：打开头块的数量。   
prometheus\_local\_storage\_chunks\_to\_persist：仍然需要将其持续到磁盘的内存块数。   
prometheus\_local\_storage\_memory\_chunks：目前在记忆中的块数。如果减去前两个，则可以得到持久化块的数量（如果查询当前没有使用，则它们是可驱动的）。   
prometheus\_local\_storage\_series\_chunks\_persisted：每个批次持续存在块数的直方图。   
prometheus\_local\_storage\_rushed\_mode如果prometheus斯处于“冲动模式”，则为1，否则为0。可用于计算prometheus处于冲动模式的时间百分比。   
prometheus\_local\_storage\_checkpoint\_last\_duration\_seconds：最后一个检查点需要多长时间   
prometheus\_local\_storage\_checkpoint\_last\_size\_bytes：最后一个检查点的大小（以字节为单位）。   
prometheus\_local\_storage\_checkpointing是1，而prometheus是检查点，否则为0。可以用来计算普罗米修斯检查点的时间百分比。   
prometheus\_local\_storage\_inconsistencies\_total：找到存储不一致的计数器。如果大于0，请重新启动服务器进行恢复。   
prometheus\_local\_storage\_persist\_errors\_total：反对持续错误。   
prometheus\_local\_storage\_memory\_dirty\_series：当前脏系列数量。   
process\_resident\_memory\_bytes广义地说，prometheus进程所占据的物理内存。   
go\_memstats\_alloc\_bytes：去堆大小（分配的对象在使用中加分配对象不再使用，但尚未被垃圾回收）。

prometheus还另一个高级应用就是集群联邦，通过定义slave，这样就可以在每个数据中心部署一个，然后通过联邦汇聚。

- scrape\_config:

- job\_name: dc\_prometheus

honor\_labels: true

metrics\_path: /federate

params:

match[]:

- '{\_\_name\_\_=~"^job:.\*"}' # Request all job-level time series

static\_configs:

- targets:

- dc1-prometheus:9090

- dc2-prometheus:9090

当然如果存储量不够还可以通过分片去采集，

global:

external\_labels:

slave: 1 # This is the 2nd slave. This prevents clashes between slaves.

scrape\_configs:

- job\_name: some\_job

# Add usual service discovery here, such as static\_configs

relabel\_configs:

- source\_labels: [\_\_address\_\_]

modulus: 4 # 4 slaves

target\_label: \_\_tmp\_hash

action: hashmod

- source\_labels: [\_\_tmp\_hash]

regex: ^1$ # This is the 2nd slave

action: keep

上面定义hash的方式去决定每个prometheus负责的targe他。

- scrape\_config:

- job\_name: slaves

honor\_labels: true

metrics\_path: /federate

params:

match[]:

- '{\_\_name\_\_=~"^slave:.\*"}' # Request all slave-level time series

static\_configs:

- targets:

- slave0:9090

- slave1:9090

- slave3:9090

- slave4:9090

下面定义了多个slave。这样数据就可以分片存储了。

# Prometheus 实战于源码分析之服务启动

- 柳清风的专栏 - 博客频道 - CSDN.NET

<http://blog.csdn.net/u010278923/article/details/70912256>

在之前的铺垫后下面进行prometheus的源码分析，首先要看的是服务启动。在cmd/prometheus/main.[**Go**](http://lib.csdn.net/base/go)中main方法，由于太长了，所以这里分段解说一下：   
先是启动本地存储

var localStorage local.Storage

switch cfg.localStorageEngine {

case "persisted":

localStorage = local.NewMemorySeriesStorage(&cfg.storage)

sampleAppender = storage.Fanout{localStorage}

case "none":

localStorage = &local.NoopStorage{}

default:

log.Errorf("Invalid local storage engine %q", cfg.localStorageEngine)

return 1

}

remoteStorage := &remote.Storage{}

sampleAppender = append(sampleAppender, remoteStorage)

reloadables = append(reloadables, remoteStorage)

由于config里面默认本地存储是persistence，也就是是存在本地磁盘中

cfg.fs.StringVar(

&cfg.localStorageEngine, "storage.local.engine", "persisted",

"Local storage engine. Supported values are: 'persisted' (full local storage with on-disk persistence) and 'none' (no local storage).",

)

如果设置是none的话就不保存在本地了，NoopStorage是空实现。prometheus不经支持本地存储还支持远端存储，这个后面再细说。

var (

notifier = notifier.New(&cfg.notifier)

targetManager = retrieval.NewTargetManager(sampleAppender)

queryEngine = promql.NewEngine(localStorage, &cfg.queryEngine)

ctx, cancelCtx = context.WithCancel(context.Background())

)

ruleManager := rules.NewManager(&rules.ManagerOptions{

SampleAppender: sampleAppender,

Notifier: notifier,

QueryEngine: queryEngine,

Context: ctx,

ExternalURL: cfg.web.ExternalURL,

})

cfg.web.Context = ctx

cfg.web.Storage = localStorage

cfg.web.QueryEngine = queryEngine

cfg.web.TargetManager = targetManager

cfg.web.RuleManager = ruleManager

cfg.web.Notifier = notifier

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

初始化上下午ctx，创建管理target的targetmanager、还有查询引擎和告警通知notifier。

reloadables = append(reloadables, targetManager, ruleManager, webHandler, notifier)

if err := reloadConfig(cfg.configFile, reloadables...); err != nil {

log.Errorf("Error loading config: %s", err)

return 1

}

* 1
* 2
* 3
* 4
* 5
* 6
* 1
* 2
* 3
* 4
* 5
* 6

初始的配置是可以通过加载配置文件修改的。下面的代码是prometheus新加的自动更新配置的功能，通过监听SIGHUP信号去重新加载配置文件

hup := make(chan os.Signal)

hupReady := make(chan bool)

signal.Notify(hup, syscall.SIGHUP)

go func() {

<-hupReady

for {

select {

case <-hup:

if err := reloadConfig(cfg.configFile, reloadables...); err != nil {

log.Errorf("Error reloading config: %s", err)

}

case rc := <-webHandler.Reload():

if err := reloadConfig(cfg.configFile, reloadables...); err != nil {

log.Errorf("Error reloading config: %s", err)

rc <- err

} else {

rc <- nil

}

}

}

}()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

下面是启动本地存储的方法：

if err := localStorage.Start(); err != nil {

log.Errorln("Error opening memory series storage:", err)

return 1

}

defer func() {

if err := localStorage.Stop(); err != nil {

log.Errorln("Error stopping storage:", err)

}

}()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

prometheus不仅能监控外部系统，还能监控自身，所以需要把自身的性能指标暴露出去

if instrumentedStorage, ok := localStorage.(prometheus.Collector); ok {

prometheus.MustRegister(instrumentedStorage)

}

prometheus.MustRegister(notifier)

prometheus.MustRegister(configSuccess)

prometheus.MustRegister(configSuccessTime)

* 1
* 2
* 3
* 4
* 5
* 6
* 1
* 2
* 3
* 4
* 5
* 6

通过MustRegister主注册，这个后面再深入讲解。

go notifier.Run()

defer notifier.Stop()

go ruleManager.Run()

defer ruleManager.Stop()

go targetManager.Run()

defer targetManager.Stop()

// Shutting down the query engine before the rule manager will cause pending queries

// to be canceled and ensures a quick shutdown of the rule manager.

defer cancelCtx()

go webHandler.Run()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

紧接着就是启动各种服务，如告警通知服务，target管理服务，规则管理服务，web等。下面就是优雅停止服务的部分

term := make(chan os.Signal)

signal.Notify(term, os.Interrupt, syscall.SIGTERM)

select {

case <-term:

log.Warn("Received SIGTERM, exiting gracefully...")

case <-webHandler.Quit():

log.Warn("Received termination request via web service, exiting gracefully...")

case err := <-webHandler.ListenError():

log.Errorln("Error starting web server, exiting gracefully:", err)

}

log.Info("See you next time!")

return 0

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

下面截取了服务的启动日志

time="2017-04-27T11:05:10Z" level=info msg="Starting prometheus (version=1.6.1, branch=master, revision=4666df502c0e239ed4aa1d80abbbfb54f61b23c3)" source="main.go:88"

time="2017-04-27T11:05:10Z" level=info msg="Build context (go=go1.8.1, user=root@7e45fa0366a7, date=20170419-14:32:22)" source="main.go:89"

time="2017-04-27T11:05:10Z" level=info msg="Loading configuration file /etc/prometheus/prometheus.yml" source="main.go:251"

time="2017-04-27T11:05:10Z" level=info msg="Loading series map and head chunks..." source="storage.go:421"

time="2017-04-27T11:05:10Z" level=info msg="0 series loaded." source="storage.go:432"

time="2017-04-27T11:05:10Z" level=info msg="Starting target manager..." source="targetmanager.go:61"

time="2017-04-27T11:05:10Z" level=info msg="Listening on :9090" source="web.go:259"

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 1
* 2
* 3
* 4
* 5
* 6
* 7

这样服务主体就已经启动了。

# Prometheus 实战于源码分析之webHandler

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70916753

上一篇介绍了服务启动基本流程，但里面有几个点没有展开，这里还是要分析一下，首先介绍这个webHandler，它是一个web http请求的服务端。先看初始

func New(o \*Options) \*Handler {

router := route.New(func(r \*http.Request) (context.Context, error) {

return o.Context, nil

})

cwd, err := os.Getwd()

if err != nil {

cwd = "<error retrieving current working directory>"

}

h := &Handler{

router: router,

listenErrCh: make(chan error),

quitCh: make(chan struct{}),

reloadCh: make(chan chan error),

options: o,

versionInfo: o.Version,

birth: time.Now(),

cwd: cwd,

flagsMap: o.Flags,

context: o.Context,

targetManager: o.TargetManager,

ruleManager: o.RuleManager,

queryEngine: o.QueryEngine,

storage: o.Storage,

notifier: o.Notifier,

apiV1: api\_v1.NewAPI(o.QueryEngine, o.Storage, o.TargetManager, o.Notifier),

now: model.Now,

}

if o.RoutePrefix != "/" {

// If the prefix is missing for the root path, prepend it.

router.Get("/", func(w http.ResponseWriter, r \*http.Request) {

http.Redirect(w, r, o.RoutePrefix, http.StatusFound)

})

router = router.WithPrefix(o.RoutePrefix)

}

instrh := prometheus.InstrumentHandler

instrf := prometheus.InstrumentHandlerFunc

router.Get("/", func(w http.ResponseWriter, r \*http.Request) {

router.Redirect(w, r, "/graph", http.StatusFound)

})

router.Get("/alerts", instrf("alerts", h.alerts))

router.Get("/graph", instrf("graph", h.graph))

router.Get("/status", instrf("status", h.status))

router.Get("/flags", instrf("flags", h.flags))

router.Get("/config", instrf("config", h.config))

router.Get("/rules", instrf("rules", h.rules))

router.Get("/targets", instrf("targets", h.targets))

router.Get("/version", instrf("version", h.version))

router.Get("/heap", instrf("heap", dumpHeap))

router.Get(o.MetricsPath, prometheus.Handler().ServeHTTP)

router.Get("/federate", instrh("federate", httputil.CompressionHandler{

Handler: http.HandlerFunc(h.federation),

}))

h.apiV1.Register(router.WithPrefix("/api/v1"))

router.Get("/consoles/\*filepath", instrf("consoles", h.consoles))

router.Get("/static/\*filepath", instrf("static", serveStaticAsset))

if o.UserAssetsPath != "" {

router.Get("/user/\*filepath", instrf("user", route.FileServe(o.UserAssetsPath)))

}

if o.EnableQuit {

router.Post("/-/quit", h.quit)

}

router.Post("/-/reload", h.reload)

router.Get("/-/reload", func(w http.ResponseWriter, r \*http.Request) {

w.WriteHeader(http.StatusMethodNotAllowed)

fmt.Fprintf(w, "This endpoint requires a POST request.\n")

})

router.Get("/debug/\*subpath", http.DefaultServeMux.ServeHTTP)

router.Post("/debug/\*subpath", http.DefaultServeMux.ServeHTTP)

return h

}

这个代码写的还是很工整的，先看router，这个里面定义了这个handler所提供的方法，譬如，获取告警的方法router.Get(“/alerts”, instrf(“alerts”, h.alerts))，还有就是打开首页router.Redirect(w, r, “/graph”, http.StatusFound)跳转到首页等，还有一个特别的方法也需要单独接受一下就是router.Post(“/-/reload”, h.reload)，这个牛逼了，就是可以通过发送post请求就可以重新relaod配置了，这个和上一篇启动里面说的对应上了，可以通过SIGHUP和reload请求两种方式重新加载配置。看代码：

func (h \*Handler) reload(w http.ResponseWriter, r \*http.Request) {

rc := make(chan error)

h.reloadCh <- rc

if err := <-rc; err != nil {

http.Error(w, fmt.Sprintf("failed to reload config: %s", err), http.StatusInternalServerError)

}

}

管道reloadCh chan chan error里是管道的管道。   
这个webhandler还提供其他的查询服务如target

func (h \*Handler) targets(w http.ResponseWriter, r \*http.Request) {

// Bucket targets by job label

tps := map[string][]\*retrieval.Target{}

for \_, t := range h.targetManager.Targets() {

job := string(t.Labels()[model.JobLabel])

tps[job] = append(tps[job], t)

}

h.executeTemplate(w, "targets.html", struct {

TargetPools map[string][]\*retrieval.Target

}{

TargetPools: tps,

})

}

还有最后一个服务需要介绍就是prometheus提供自身监控数据的接口

mfs, err := DefaultGatherer.Gather()

通过Gather()方法收集

wg.Add(len(r.collectorsByID))

go func() {

wg.Wait()

close(metricChan)

}()

for \_, collector := range r.collectorsByID {

go func(collector Collector) {

defer wg.Done()

collector.Collect(metricChan)

}(collector)

}

可以通过<http://x.x.x.x:9090/metrics>访问监控数据

剩下的就是服务的启动，这个就是一个http server

func (h \*Handler) Run() {

log.Infof("Listening on %s", h.options.ListenAddress)

server := &http.Server{

Addr: h.options.ListenAddress,

Handler: h.router,

ErrorLog: log.NewErrorLogger(),

ReadTimeout: h.options.ReadTimeout,

}

listener, err := net.Listen("tcp", h.options.ListenAddress)

if err != nil {

h.listenErrCh <- err

} else {

limitedListener := netutil.LimitListener(listener, h.options.MaxConnections)

h.listenErrCh <- server.Serve(limitedListener)

}

}

这个里面的h.router就是上面注册的router，这样就可以做http的分发了。

# Prometheus 实战于源码分析之collector

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70927773

在prometheus里面有很多的exporter，每个exporter里面的都有一个collector，我在这里先写分析一下prometheus自身的监控系统，采集自己的监控数据。   
先看接口

type Collector interface {

Describe(chan<- \*Desc)

Collect(chan<- Metric)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 1
* 2
* 3
* 4
* 5
* 6

有很多数据类型实现了这个接口

Gauge

type Gauge interface {

Metric

Collector

// Set sets the Gauge to an arbitrary value.

Set(float64)

// Inc increments the Gauge by 1.

Inc()

// Dec decrements the Gauge by 1.

Dec()

// Add adds the given value to the Gauge. (The value can be

// negative, resulting in a decrease of the Gauge.)

Add(float64)

// Sub subtracts the given value from the Gauge. (The value can be

// negative, resulting in an increase of the Gauge.)

Sub(float64)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

Histogram

type Histogram interface {

Metric

Collector

// Observe adds a single observation to the histogram.

Observe(float64)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 1
* 2
* 3
* 4
* 5
* 6
* 7

Counter

type Counter interface {

Metric

Collector

// Set is used to set the Counter to an arbitrary value. It is only used

// if you have to transfer a value from an external counter into this

// Prometheus metric. Do not use it for regular handling of a

// Prometheus counter (as it can be used to break the contract of

// monotonically increasing values).

//

// Deprecated: Use NewConstMetric to create a counter for an external

// value. A Counter should never be set.

Set(float64)

// Inc increments the counter by 1.

Inc()

// Add adds the given value to the counter. It panics if the value is <

// 0.

Add(float64)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

Summary

type Summary interface {

Metric

Collector

// Observe adds a single observation to the summary.

Observe(float64)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 1
* 2
* 3
* 4
* 5
* 6
* 7

这是Collector接口还有一个prometheus自己的一个实现selfCollector

type selfCollector struct {

self Metric

}

// init provides the selfCollector with a reference to the metric it is supposed

// to collect. It is usually called within the factory function to create a

// metric. See example.

func (c \*selfCollector) init(self Metric) {

c.self = self

}

// Describe implements Collector.

func (c \*selfCollector) Describe(ch chan<- \*Desc) {

ch <- c.self.Desc()

}

// Collect implements Collector.

func (c \*selfCollector) Collect(ch chan<- Metric) {

ch <- c.self

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

当执行selfCollector的Collect方法就是返回本身的Metric。还记得第一篇说的注册吗？prometheus.MustRegister(configSuccess)注册这个configSuccess

configSuccess = prometheus.NewGauge(prometheus.GaugeOpts{

Namespace: "prometheus",

Name: "config\_last\_reload\_successful",

Help: "Whether the last configuration reload attempt was successful.",

})

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

在NewGauge里面，本质上就创建一个value。这个value里面有selfCollector，就是上面的selfCollector

type value struct {

valBits uint64

selfCollector

desc \*Desc

valType ValueType

labelPairs []\*dto.LabelPair

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

创建完Gauge后就可以注册MustRegister(…Collector)，具体看

func (r \*Registry) MustRegister(cs ...Collector) {

for \_, c := range cs {

if err := r.Register(c); err != nil {

panic(err)

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 1
* 2
* 3
* 4
* 5
* 6
* 7

再深入看一下Register方法

if len(newDescIDs) == 0 {

return errors.New("collector has no descriptors")

}

if existing, exists := r.collectorsByID[collectorID]; exists {

return AlreadyRegisteredError{

ExistingCollector: existing,

NewCollector: c,

}

}

// If the collectorID is new, but at least one of the descs existed

// before, we are in trouble.

if duplicateDescErr != nil {

return duplicateDescErr

}

// Only after all tests have passed, actually register.

r.collectorsByID[collectorID] = c

for hash := range newDescIDs {

r.descIDs[hash] = struct{}{}

}

for name, dimHash := range newDimHashesByName {

r.dimHashesByName[name] = dimHash

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

就是注册到collectorsByID这map里面，collectorsByID map[uint64]Collector 它的key是descID，值就是我们注册的collector。   
通过这个map去维护collector。取消注册的方法是删除

r.mtx.RLock()

if \_, exists := r.collectorsByID[collectorID]; !exists {

r.mtx.RUnlock()

return false

}

r.mtx.RUnlock()

r.mtx.Lock()

defer r.mtx.Unlock()

delete(r.collectorsByID, collectorID)

for id := range descIDs {

delete(r.descIDs, id)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

现在已经把collector的结构和注册讲完了，那么采集就变的顺理成章了，Gather()方法采集数据

wg.Add(len(r.collectorsByID))

go func() {

wg.Wait()

close(metricChan)

}()

for \_, collector := range r.collectorsByID {

go func(collector Collector) {

defer wg.Done()

collector.Collect(metricChan)

}(collector)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

循环遍历执行collecto去采集，把结果放到metricChan，然后就参数解析封装了，这里涉及到了数据类型，和上面接口组合是对应的

dtoMetric := &dto.Metric{}

if err := metric.Write(dtoMetric); err != nil {

errs = append(errs, fmt.Errorf(

"error collecting metric %v: %s", desc, err,

))

continue

}

...

metricFamily.Metric = append(metricFamily.Metric, dtoMetric)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

上面的write方法在需要解释一下，如果是value类型

func (v \*value) Write(out \*dto.Metric) error {

val := math.Float64frombits(atomic.LoadUint64(&v.valBits))

return populateMetric(v.valType, val, v.labelPairs, out)

}

func populateMetric(

t ValueType,

v float64,

labelPairs []\*dto.LabelPair,

m \*dto.Metric,

) error {

m.Label = labelPairs

switch t {

case CounterValue:

m.Counter = &dto.Counter{Value: proto.Float64(v)}

case GaugeValue:

m.Gauge = &dto.Gauge{Value: proto.Float64(v)}

case UntypedValue:

m.Untyped = &dto.Untyped{Value: proto.Float64(v)}

default:

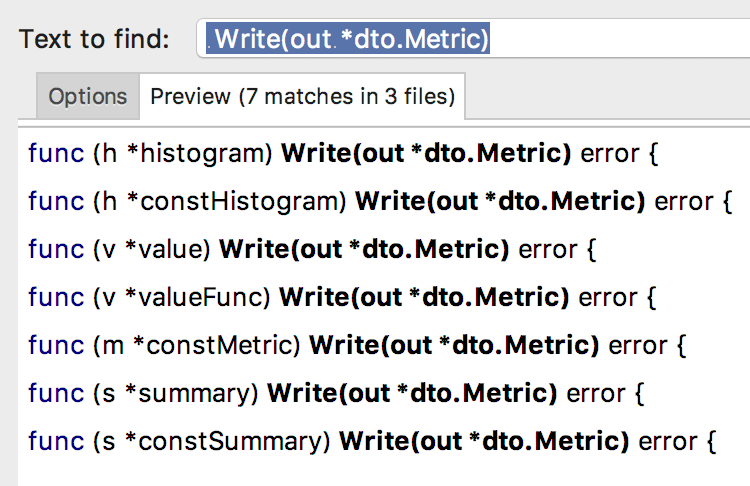
return fmt.Errorf("encountered unknown type %v", t)

}

return nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

如果是其它类型，在自己的   
  
这里还有补充一下对于指标的定义

type Metric struct {

Label []\*LabelPair `protobuf:"bytes,1,rep,name=label" json:"label,omitempty"`

Gauge \*Gauge `protobuf:"bytes,2,opt,name=gauge" json:"gauge,omitempty"`

Counter \*Counter `protobuf:"bytes,3,opt,name=counter" json:"counter,omitempty"`

Summary \*Summary `protobuf:"bytes,4,opt,name=summary" json:"summary,omitempty"`

Untyped \*Untyped `protobuf:"bytes,5,opt,name=untyped" json:"untyped,omitempty"`

Histogram \*Histogram `protobuf:"bytes,7,opt,name=histogram" json:"histogram,omitempty"`

TimestampMs \*int64 `protobuf:"varint,6,opt,name=timestamp\_ms" json:"timestamp\_ms,omitempty"`

XXX\_unrecognized []byte `json:"-"`

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

# Prometheus 实战于源码分析之discovery

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70943506

prometheus与时俱进在现在各种容器管理平台流行的当下，能够对各种容器管理平台进行数据采集和处理，并且能够自动发现监控对象，这个就是今天要说的discovery。他是一个资源发现的组件，能够根据资源变化调整监控对象。目前已经支持的discovery主要有DNS、kubernetes、marathon、zk、consul、file等，我们先从这个kubernetes的服务发现开始讲解吧   
先是创建

func New(l log.Logger, conf \*config.KubernetesSDConfig) (\*Discovery, error) {

var (

kcfg \*rest.Config

err error

)

if conf.APIServer.URL == nil {

// Use the Kubernetes provided pod service account

// as described in https://kubernetes.io/docs/admin/service-accounts-admin/

kcfg, err = rest.InClusterConfig()

if err != nil {

return nil, err

}

// Because the handling of configuration parameters changes

// we should inform the user when their currently configured values

// will be ignored due to precedence of InClusterConfig

l.Info("Using pod service account via in-cluster config")

if conf.TLSConfig.CAFile != "" {

l.Warn("Configured TLS CA file is ignored when using pod service account")

}

if conf.TLSConfig.CertFile != "" || conf.TLSConfig.KeyFile != "" {

l.Warn("Configured TLS client certificate is ignored when using pod service account")

}

if conf.BearerToken != "" {

l.Warn("Configured auth token is ignored when using pod service account")

}

if conf.BasicAuth != nil {

l.Warn("Configured basic authentication credentials are ignored when using pod service account")

}

} else {

kcfg = &rest.Config{

Host: conf.APIServer.String(),

TLSClientConfig: rest.TLSClientConfig{

CAFile: conf.TLSConfig.CAFile,

CertFile: conf.TLSConfig.CertFile,

KeyFile: conf.TLSConfig.KeyFile,

},

Insecure: conf.TLSConfig.InsecureSkipVerify,

}

token := conf.BearerToken

if conf.BearerTokenFile != "" {

bf, err := ioutil.ReadFile(conf.BearerTokenFile)

if err != nil {

return nil, err

}

token = string(bf)

}

kcfg.BearerToken = token

if conf.BasicAuth != nil {

kcfg.Username = conf.BasicAuth.Username

kcfg.Password = conf.BasicAuth.Password

}

}

kcfg.UserAgent = "prometheus/discovery"

c, err := kubernetes.NewForConfig(kcfg)

if err != nil {

return nil, err

}

return &Discovery{

client: c,

logger: l,

role: conf.Role,

}, nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66

创建的核心就是创建一个c，这个c就是一个kubernetes的client，负责调用kubernetes api的。那他是是怎么做到资源发现的呢？不用说大家应该都能想到之前ingress的设计，对，就是listwatch机制。

func (d \*Discovery) Run(ctx context.Context, ch chan<- []\*config.TargetGroup) {

rclient := d.client.Core().GetRESTClient()

switch d.role {

case "endpoints":

elw := cache.NewListWatchFromClient(rclient, "endpoints", api.NamespaceAll, nil)

slw := cache.NewListWatchFromClient(rclient, "services", api.NamespaceAll, nil)

plw := cache.NewListWatchFromClient(rclient, "pods", api.NamespaceAll, nil)

eps := NewEndpoints(

d.logger.With("kubernetes\_sd", "endpoint"),

cache.NewSharedInformer(slw, &apiv1.Service{}, resyncPeriod),

cache.NewSharedInformer(elw, &apiv1.Endpoints{}, resyncPeriod),

cache.NewSharedInformer(plw, &apiv1.Pod{}, resyncPeriod),

)

go eps.endpointsInf.Run(ctx.Done())

go eps.serviceInf.Run(ctx.Done())

go eps.podInf.Run(ctx.Done())

for !eps.serviceInf.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

for !eps.endpointsInf.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

for !eps.podInf.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

eps.Run(ctx, ch)

case "pod":

plw := cache.NewListWatchFromClient(rclient, "pods", api.NamespaceAll, nil)

pod := NewPod(

d.logger.With("kubernetes\_sd", "pod"),

cache.NewSharedInformer(plw, &apiv1.Pod{}, resyncPeriod),

)

go pod.informer.Run(ctx.Done())

for !pod.informer.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

pod.Run(ctx, ch)

case "service":

slw := cache.NewListWatchFromClient(rclient, "services", api.NamespaceAll, nil)

svc := NewService(

d.logger.With("kubernetes\_sd", "service"),

cache.NewSharedInformer(slw, &apiv1.Service{}, resyncPeriod),

)

go svc.informer.Run(ctx.Done())

for !svc.informer.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

svc.Run(ctx, ch)

case "node":

nlw := cache.NewListWatchFromClient(rclient, "nodes", api.NamespaceAll, nil)

node := NewNode(

d.logger.With("kubernetes\_sd", "node"),

cache.NewSharedInformer(nlw, &apiv1.Node{}, resyncPeriod),

)

go node.informer.Run(ctx.Done())

for !node.informer.HasSynced() {

time.Sleep(100 \* time.Millisecond)

}

node.Run(ctx, ch)

default:

d.logger.Errorf("unknown Kubernetes discovery kind %q", d.role)

}

<-ctx.Done()

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74

这个里面根据role去启动不同的资源监听，下面以node为例，就是节点事件，譬如添加节点或者删除节点。

n.informer.AddEventHandler(cache.ResourceEventHandlerFuncs{

AddFunc: func(o interface{}) {

eventCount.WithLabelValues("node", "add").Inc()

node, err := convertToNode(o)

if err != nil {

n.logger.With("err", err).Errorln("converting to Node object failed")

return

}

send(n.buildNode(node))

},

DeleteFunc: func(o interface{}) {

eventCount.WithLabelValues("node", "delete").Inc()

node, err := convertToNode(o)

if err != nil {

n.logger.With("err", err).Errorln("converting to Node object failed")

return

}

send(&config.TargetGroup{Source: nodeSource(node)})

},

UpdateFunc: func(\_, o interface{}) {

eventCount.WithLabelValues("node", "update").Inc()

node, err := convertToNode(o)

if err != nil {

n.logger.With("err", err).Errorln("converting to Node object failed")

return

}

send(n.buildNode(node))

},

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

针对不同事件的方法。当添加节点的时候，通过buildNode创建targetgroup如下：

func (n \*Node) buildNode(node \*apiv1.Node) \*config.TargetGroup {

tg := &config.TargetGroup{

Source: nodeSource(node),

}

tg.Labels = nodeLabels(node)

addr, addrMap, err := nodeAddress(node)

if err != nil {

n.logger.With("err", err).Debugf("No node address found")

return nil

}

addr = net.JoinHostPort(addr, strconv.FormatInt(int64(node.Status.DaemonEndpoints.KubeletEndpoint.Port), 10))

t := model.LabelSet{

model.AddressLabel: lv(addr),

model.InstanceLabel: lv(node.Name),

}

for ty, a := range addrMap {

ln := strutil.SanitizeLabelName(nodeAddressPrefix + string(ty))

t[model.LabelName(ln)] = lv(a[0])

}

tg.Targets = append(tg.Targets, t)

return tg

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

这个里面先是创建一个targetgroup，他是target的组，每个组都有自己标签tg.Labels，每个组都有自己的成员tg.Targets。再把这个tg返回到discovery，这样discovery就可以感知资源变化，那么谁来调用呢？   
其实很容易理解，这个discovery是干嘛的，就是动态更新target嘛，prometheus谁来维护呢？答案就是TargetManager

type TargetManager struct {

appender storage.SampleAppender

scrapeConfigs []\*config.ScrapeConfig

mtx sync.RWMutex

ctx context.Context

cancel func()

wg sync.WaitGroup

// Set of unqiue targets by scrape configuration.

targetSets map[string]\*targetSet

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

这个里面有一个targetSets的属性，它的key是job的命名，value是targetSet，

func (ts \*TargetSet) Run(ctx context.Context) {

Loop:

for {

// Throttle syncing to once per five seconds.

select {

case <-ctx.Done():

break Loop

case p := <-ts.providerCh:

ts.updateProviders(ctx, p)

case <-time.After(5 \* time.Second):

}

select {

case <-ctx.Done():

break Loop

case <-ts.syncCh:

ts.sync()

case p := <-ts.providerCh:

ts.updateProviders(ctx, p)

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

这个TargetSet启动死循环调用updateProviders，而这个updateProviders

for name, prov := range providers {

wg.Add(1)

updates := make(chan []\*config.TargetGroup)

go prov.Run(ctx, updates)

go func(name string, prov TargetProvider) {

select {

case <-ctx.Done():

case initial, ok := <-updates:

// Handle the case that a target provider exits and closes the channel

// before the context is done.

if !ok {

break

}

// First set of all targets the provider knows.

for \_, tgroup := range initial {

ts.setTargetGroup(name, tgroup)

}

case <-time.After(5 \* time.Second):

// Initial set didn't arrive. Act as if it was empty

// and wait for updates later on.

}

wg.Done()

// Start listening for further updates.

for {

select {

case <-ctx.Done():

return

case tgs, ok := <-updates:

// Handle the case that a target provider exits and closes the channel

// before the context is done.

if !ok {

return

}

for \_, tg := range tgs {

ts.update(name, tg)

}

}

}

}(name, prov)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

变量所有的providers，这里的每个provider都是一个资源发现的提供者，通过provider的Run方法去加载更新，这个Run就是discovery的Run。回到了上面说的方法了，这样target就能够更新了。   
上面演示了kubernetes的资源发现，下面还补充一个marathon的，其实原理是一样的discovery/marathon/marathon.[**Go**](http://lib.csdn.net/base/go)，

func (d \*Discovery) updateServices(ctx context.Context, ch chan<- []\*config.TargetGroup) (err error) {

t0 := time.Now()

defer func() {

refreshDuration.Observe(time.Since(t0).Seconds())

if err != nil {

refreshFailuresCount.Inc()

}

}()

targetMap, err := d.fetchTargetGroups()

if err != nil {

return err

}

all := make([]\*config.TargetGroup, 0, len(targetMap))

for \_, tg := range targetMap {

all = append(all, tg)

}

select {

case <-ctx.Done():

return ctx.Err()

case ch <- all:

}

// Remove services which did disappear.

for source := range d.lastRefresh {

\_, ok := targetMap[source]

if !ok {

select {

case <-ctx.Done():

return ctx.Err()

case ch <- []\*config.TargetGroup{{Source: source}}:

log.Debugf("Removing group for %s", source)

}

}

}

d.lastRefresh = targetMap

return nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

这个里面定时执行，通过fetchTargetGroups获取TargetGroups，

func (d \*Discovery) fetchTargetGroups() (map[string]\*config.TargetGroup, error) {

url := RandomAppsURL(d.servers)

apps, err := d.appsClient(d.client, url, d.token)

if err != nil {

return nil, err

}

groups := AppsToTargetGroups(apps)

return groups, nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

这个方法就是调用marathon的/v2/apps接口去获取APP、task（[**Container**](http://lib.csdn.net/base/docker)）信息。这个就是discovery的基本工作原理。

# Prometheus 实战于源码分析之scrape

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70946004

[上一篇](http://blog.csdn.net/u010278923/article/details/70943506)介绍了tagert管理和发现discovery，那个这些targets是怎样采集的呢？下面就介绍scrape，它是数据采集器。

type scrapePool struct {

appender storage.SampleAppender

ctx context.Context

mtx sync.RWMutex

config \*config.ScrapeConfig

client \*http.Client

// Targets and loops must always be synchronized to have the same

// set of hashes.

targets map[uint64]\*Target

loops map[uint64]loop

// Constructor for new scrape loops. This is settable for testing convenience.

newLoop func(context.Context, scraper, storage.SampleAppender, model.LabelSet, \*config.ScrapeConfig) loop

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

先定义一个scrapePool采集的池，这个里面有targets，所以需要采集的对象。还记得上一篇的TargetManager里面启动discovery里面的targetset

go func(ts \*targetSet) {

ts.ts.Run(ctx)

ts.sp.stop()

tm.wg.Done()

}(ts)

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

具体Run方法discovery/discovery.[**Go**](http://lib.csdn.net/base/go)

func (ts \*TargetSet) Run(ctx context.Context) {

Loop:

for {

// Throttle syncing to once per five seconds.

select {

case <-ctx.Done():

break Loop

case p := <-ts.providerCh:

ts.updateProviders(ctx, p)

case <-time.After(5 \* time.Second):

}

select {

case <-ctx.Done():

break Loop

case <-ts.syncCh:

ts.sync()

case p := <-ts.providerCh:

ts.updateProviders(ctx, p)

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

这个里面sync方法

func (ts \*TargetSet) sync() {

ts.mtx.RLock()

var all []\*config.TargetGroup

for \_, tg := range ts.tgroups {

all = append(all, tg)

}

ts.mtx.RUnlock()

ts.syncer.Sync(all)

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

遍历TargetGroup收集所有的target放到all变量里面

func (sp \*scrapePool) Sync(tgs []\*config.TargetGroup) {

start := time.Now()

var all []\*Target

for \_, tg := range tgs {

targets, err := targetsFromGroup(tg, sp.config)

if err != nil {

log.With("err", err).Error("creating targets failed")

continue

}

all = append(all, targets...)

}

sp.sync(all)

targetSyncIntervalLength.WithLabelValues(sp.config.JobName).Observe(

time.Since(start).Seconds(),

)

targetScrapePoolSyncsCounter.WithLabelValues(sp.config.JobName).Inc()

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

具体看sync方法

func (sp \*scrapePool) sync(targets []\*Target) {

sp.mtx.Lock()

defer sp.mtx.Unlock()

var (

uniqueTargets = map[uint64]struct{}{}

interval = time.Duration(sp.config.ScrapeInterval)

timeout = time.Duration(sp.config.ScrapeTimeout)

)

for \_, t := range targets {

hash := t.hash()

uniqueTargets[hash] = struct{}{}

if \_, ok := sp.targets[hash]; !ok {

s := &targetScraper{Target: t, client: sp.client}

l := sp.newLoop(sp.ctx, s, sp.appender, t.Labels(), sp.config)

sp.targets[hash] = t

sp.loops[hash] = l

go l.run(interval, timeout, nil)

}

}

var wg sync.WaitGroup

// Stop and remove old targets and scraper loops.

for hash := range sp.targets {

if \_, ok := uniqueTargets[hash]; !ok {

wg.Add(1)

go func(l loop) {

l.stop()

wg.Done()

}(sp.loops[hash])

delete(sp.loops, hash)

delete(sp.targets, hash)

}

}

wg.Wait()

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42

针对每个target启动go l.run(interval, timeout, nil)采集

func (sl \*scrapeLoop) run(interval, timeout time.Duration, errc chan<- error) {

defer close(sl.done)

select {

case <-time.After(sl.scraper.offset(interval)):

// Continue after a scraping offset.

case <-sl.ctx.Done():

return

}

var last time.Time

ticker := time.NewTicker(interval)

defer ticker.Stop()

for {

select {

case <-sl.ctx.Done():

return

default:

}

if !sl.appender.NeedsThrottling() {

var (

start = time.Now()

scrapeCtx, \_ = context.WithTimeout(sl.ctx, timeout)

numPostRelabelSamples = 0

)

// Only record after the first scrape.

if !last.IsZero() {

targetIntervalLength.WithLabelValues(interval.String()).Observe(

time.Since(last).Seconds(),

)

}

samples, err := sl.scraper.scrape(scrapeCtx, start)

if err == nil {

numPostRelabelSamples, err = sl.append(samples)

}

if err != nil && errc != nil {

errc <- err

}

sl.report(start, time.Since(start), len(samples), numPostRelabelSamples, err)

last = start

} else {

targetSkippedScrapes.Inc()

}

select {

case <-sl.ctx.Done():

return

case <-ticker.C:

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56

通过sl.scraper.scrape采集，这个方法就是通过GET请求去获取

func (s \*targetScraper) scrape(ctx context.Context, ts time.Time) (model.Samples, error) {

req, err := http.NewRequest("GET", s.URL().String(), nil)

if err != nil {

return nil, err

}

req.Header.Add("Accept", acceptHeader)

req.Header.Set("User-Agent", userAgentHeader)

resp, err := ctxhttp.Do(ctx, s.client, req)

if err != nil {

return nil, err

}

defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {

return nil, fmt.Errorf("server returned HTTP status %s", resp.Status)

}

var (

allSamples = make(model.Samples, 0, 200)

decSamples = make(model.Vector, 0, 50)

)

sdec := expfmt.SampleDecoder{

Dec: expfmt.NewDecoder(resp.Body, expfmt.ResponseFormat(resp.Header)),

Opts: &expfmt.DecodeOptions{

Timestamp: model.TimeFromUnixNano(ts.UnixNano()),

},

}

for {

if err = sdec.Decode(&decSamples); err != nil {

break

}

allSamples = append(allSamples, decSamples...)

decSamples = decSamples[:0]

}

if err == io.EOF {

// Set err to nil since it is used in the scrape health recording.

err = nil

}

return allSamples, err

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

整个数据采集的流程就是这样。

# Prometheus 实战于源码分析之alert

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70946469

prometheus不仅可以提供数据采集功能，而且还可以做告警服务，通过匹配的性能参数，发出告警，先看看alert配置。

rule\_files:

[ - <filepath\_glob> ... ]

# A list of scrape configurations.

scrape\_configs:

[ - <scrape\_config> ... ]

# Alerting specifies settings related to the Alertmanager.

alerting:

alert\_relabel\_configs:

[ - <relabel\_config> ... ]

alertmanagers:

[ - <alertmanager\_config> ... ]

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

其中的rule\_file就是定义告警的规则的文件，譬如下面这个例子：

# Alert for any instance that is unreachable for >5 minutes.

ALERT InstanceDown

IF up == 0

FOR 5m

LABELS { severity = "page" }

ANNOTATIONS {

summary = "Instance {{ $labels.instance }} down",

description = "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes.",

}

# Alert for any instance that have a median request latency >1s.

ALERT APIHighRequestLatency

IF api\_http\_request\_latencies\_second{quantile="0.5"} > 1

FOR 1m

ANNOTATIONS {

summary = "High request latency on {{ $labels.instance }}",

description = "{{ $labels.instance }} has a median request latency above 1s (current value: {{ $value }}s)",

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

下面就介绍一下告警的产生和通知。先看rules的加载

func (m \*Manager) loadGroups(interval time.Duration, filenames ...string) (map[string]\*Group, error) {

rules := []Rule{}

for \_, fn := range filenames {

content, err := ioutil.ReadFile(fn)

if err != nil {

return nil, err

}

stmts, err := promql.ParseStmts(string(content))

if err != nil {

return nil, fmt.Errorf("error parsing %s: %s", fn, err)

}

for \_, stmt := range stmts {

var rule Rule

switch r := stmt.(type) {

case \*promql.AlertStmt:

rule = NewAlertingRule(r.Name, r.Expr, r.Duration, r.Labels, r.Annotations)

case \*promql.RecordStmt:

rule = NewRecordingRule(r.Name, r.Expr, r.Labels)

default:

panic("retrieval.Manager.LoadRuleFiles: unknown statement type")

}

rules = append(rules, rule)

}

}

// Currently there is no group syntax implemented. Thus all rules

// are read into a single default group.

g := NewGroup("default", interval, rules, m.opts)

groups := map[string]\*Group{g.name: g}

return groups, nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35

通过rule读取配置文件加载rules。这些告警是可以分组的，这默认是default组。   
然后看一下alert是怎么定义的

type Alert struct {

State AlertState

Labels model.LabelSet

Annotations model.LabelSet

Value model.SampleValue

ActiveAt, ResolvedAt model.Time

}

const (

StateInactive AlertState = iota

// 少于阈值持续时间

StatePending

// 大于阈值持续时间

StateFiring

)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

上面定义alert以及它的三种状态。

type AlertingRule struct {

name string

vector promql.Expr

holdDuration time.Duration

labels model.LabelSet

annotations model.LabelSet

mtx sync.Mutex

active map[model.Fingerprint]\*Alert

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

active记录了这个规则触发的告警。   
现在开始走流程rules/manager.[**Go**](http://lib.csdn.net/base/go),run方法

func (g \*Group) run() {

defer close(g.terminated)

// Wait an initial amount to have consistently slotted intervals.

select {

case <-time.After(g.offset()):

case <-g.done:

return

}

iter := func() {

iterationsScheduled.Inc()

if g.opts.SampleAppender.NeedsThrottling() {

iterationsSkipped.Inc()

return

}

start := time.Now()

g.Eval()

iterationDuration.Observe(time.Since(start).Seconds())

}

iter()

tick := time.NewTicker(g.interval)

defer tick.Stop()

for {

select {

case <-g.done:

return

default:

select {

case <-g.done:

return

case <-tick.C:

iter()

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40

通过g.Eval()，遍历rules去匹配

func (g \*Group) Eval() {

var (

now = model.Now()

wg sync.WaitGroup

)

for \_, rule := range g.rules {

rtyp := string(typeForRule(rule))

wg.Add(1)

// BUG(julius): Look at fixing thundering herd.

go func(rule Rule) {

defer wg.Done()

defer func(t time.Time) {

evalDuration.WithLabelValues(rtyp).Observe(time.Since(t).Seconds())

}(time.Now())

evalTotal.WithLabelValues(rtyp).Inc()

vector, err := rule.Eval(g.opts.Context, now, g.opts.QueryEngine, g.opts.ExternalURL.Path)

if err != nil {

// Canceled queries are intentional termination of queries. This normally

// happens on shutdown and thus we skip logging of any errors here.

if \_, ok := err.(promql.ErrQueryCanceled); !ok {

log.Warnf("Error while evaluating rule %q: %s", rule, err)

}

evalFailures.WithLabelValues(rtyp).Inc()

return

}

if ar, ok := rule.(\*AlertingRule); ok {

g.sendAlerts(ar, now)

}

var (

numOutOfOrder = 0

numDuplicates = 0

)

for \_, s := range vector {

if err := g.opts.SampleAppender.Append(s); err != nil {

switch err {

case local.ErrOutOfOrderSample:

numOutOfOrder++

log.With("sample", s).With("error", err).Debug("Rule evaluation result discarded")

case local.ErrDuplicateSampleForTimestamp:

numDuplicates++

log.With("sample", s).With("error", err).Debug("Rule evaluation result discarded")

default:

log.With("sample", s).With("error", err).Warn("Rule evaluation result discarded")

}

}

}

if numOutOfOrder > 0 {

log.With("numDropped", numOutOfOrder).Warn("Error on ingesting out-of-order result from rule evaluation")

}

if numDuplicates > 0 {

log.With("numDropped", numDuplicates).Warn("Error on ingesting results from rule evaluation with different value but same timestamp")

}

}(rule)

}

wg.Wait()

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62

通过rules/alerting.go里面的Eval犯法返回告警

func (r \*AlertingRule) Eval(ctx context.Context, ts model.Time, engine \*promql.Engine, externalURLPath string) (model.Vector, error) {

query, err := engine.NewInstantQuery(r.vector.String(), ts)

if err != nil {

return nil, err

}

res, err := query.Exec(ctx).Vector()

if err != nil {

return nil, err

}

r.mtx.Lock()

defer r.mtx.Unlock()

// Create pending alerts for any new vector elements in the alert expression

// or update the expression value for existing elements.

resultFPs := map[model.Fingerprint]struct{}{}

for \_, smpl := range res {

// Provide the alert information to the template.

l := make(map[string]string, len(smpl.Metric))

for k, v := range smpl.Metric {

l[string(k)] = string(v)

}

tmplData := struct {

Labels map[string]string

Value float64

}{

Labels: l,

Value: float64(smpl.Value),

}

// Inject some convenience variables that are easier to remember for users

// who are not used to Go's templating system.

defs := "{{$labels := .Labels}}{{$value := .Value}}"

expand := func(text model.LabelValue) model.LabelValue {

tmpl := template.NewTemplateExpander(

ctx,

defs+string(text),

"\_\_alert\_"+r.Name(),

tmplData,

ts,

engine,

externalURLPath,

)

result, err := tmpl.Expand()

if err != nil {

result = fmt.Sprintf("<error expanding template: %s>", err)

log.Warnf("Error expanding alert template %v with data '%v': %s", r.Name(), tmplData, err)

}

return model.LabelValue(result)

}

delete(smpl.Metric, model.MetricNameLabel)

labels := make(model.LabelSet, len(smpl.Metric)+len(r.labels)+1)

for ln, lv := range smpl.Metric {

labels[ln] = lv

}

for ln, lv := range r.labels {

labels[ln] = expand(lv)

}

labels[model.AlertNameLabel] = model.LabelValue(r.Name())

annotations := make(model.LabelSet, len(r.annotations))

for an, av := range r.annotations {

annotations[an] = expand(av)

}

fp := smpl.Metric.Fingerprint()

resultFPs[fp] = struct{}{}

// Check whether we already have alerting state for the identifying label set.

// Update the last value and annotations if so, create a new alert entry otherwise.

if alert, ok := r.active[fp]; ok && alert.State != StateInactive {

alert.Value = smpl.Value

alert.Annotations = annotations

continue

}

r.active[fp] = &Alert{

Labels: labels,

Annotations: annotations,

ActiveAt: ts,

State: StatePending,

Value: smpl.Value,

}

}

var vec model.Vector

// Check if any pending alerts should be removed or fire now. Write out alert timeseries.

for fp, a := range r.active {

if \_, ok := resultFPs[fp]; !ok {

if a.State != StateInactive {

vec = append(vec, r.sample(a, ts, false))

}

// If the alert was previously firing, keep it around for a given

// retention time so it is reported as resolved to the AlertManager.

if a.State == StatePending || (a.ResolvedAt != 0 && ts.Sub(a.ResolvedAt) > resolvedRetention) {

delete(r.active, fp)

}

if a.State != StateInactive {

a.State = StateInactive

a.ResolvedAt = ts

}

continue

}

if a.State == StatePending && ts.Sub(a.ActiveAt) >= r.holdDuration {

vec = append(vec, r.sample(a, ts, false))

a.State = StateFiring

}

vec = append(vec, r.sample(a, ts, true))

}

return vec, nil

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116

这个方法有点长，主要是做状态判断。res是执行查新语句返回的符合条件的监控资源，后面转化成告警。