最前沿開源監控 Prometheus專題講座

第十一讲：Prometheus 之 exporter模块源代码示例

本节课内容

- 编写一个exporter的流程
- 用go开发一个exporter的源代码 及讲解
- 对于编写exporter的个人建议

（一）编写一个exporter的流程

官网有介绍 编写exporter的详细内容

我们这里给大家做一个 简单的归纳

首先 不同于pushgateway，exporter是一个独立运行 的采集程序

其中的功能需要有这三个部分

1） 自身是HTTP 服务器，可以响应 从外发过来的 HTTP GET 请求
2） 自身需要运行在后台，并可以定期触发 抓取本地的监控数据
3） 返回给prometheus_server 的内容 是需要符合 prometheus 规定的metrics 类型（Key-Value）
4) key-value -> prometheus(TS) values(float int) return string *

nil

接下来 给大家一份我之前用 go编写一份 exporter源代码

(二) 用go开发一个exporter的源代码 及讲解

```
package main

import (
  "fmt"
  _ "net/http/pprof"
    "sync"
//  "github.com/hpcloud/tail"
"github.com/prometheus/client_golang/prometheus"
"github.com/prometheus/common/log"
"github.com/prometheus/common/version"
//  //"net"
  "net/http"
//  "net/url"
//   "errors"
  "flag"
  "os"
  "os/exec"
  "io/ioutil"
//  "regexp"
//  "strconv"
//  "strings"
    "time"
```

```
)


const (
  namespace = "my_exporter"
  )



type Exporter struct {

cmd               string
mutex              sync.RWMutex
up,freeRam,TotalRam   prometheus.Gauge
Debug                 bool
upstreamResponseTimes  *prometheus.HistogramVec


}


func (e *Exporter) scrape() {

log.Infoln("--------")
e.up.Set(1)
    }


func (e *Exporter) Describe(ch chan<- *prometheus.Desc) {
  e.upstreamResponseTimes.Describe(ch)
}
```

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

e.mutex.Lock() // To protect metrics from concurrent collects
  defer e.mutex.Unlock()

  e.scrape()
  e.upstreamResponseTimes.Collect(ch)
  e.up.Collect(ch)
}



func NewExporter(cmd string, timeout time.Duration) (*Exporter,
error) {
  return &Exporter{
    cmd: cmd,
    up: prometheus.NewGauge(prometheus.GaugeOpts{
      Namespace: namespace,
      Name:      "up",
      Help:      "Was the last scrape of nginx exporter successful.",
    }),
    upstreamResponseTimes: prometheus.NewHistogramVec(
      prometheus.HistogramOpts{
        Namespace: namespace,
        Name:      "http_upstream_request_times",
        Help:      "Response times from backends/upstreams",
      },
      []string{"method", "endpoint", "response_code",
"client_type"},
    ),
```

```
  }, nil
}



func run() {

  cmd := exec.Command("/bin/sh", "-c", `free -m `)

  stdout , err := cmd.StdoutPipe()

  if err != nil {
    panic(err.Error())
  }


  if err != nil {
    fmt.Println("Stdoutpipe error", err.Error())
  }

    stderr , err := cmd.StderrPipe()

  if err != nil {
    fmt.Println("Stderrpipe error", err.Error())
  }

  if err := cmd.Start(); err != nil {
    fmt.Println("error start", err )

  }

  bytesErr, err := ioutil.ReadAll(stderr)
```

```
    if err != nil {
        fmt.Println("ReadAll stderr: ", err.Error())
        return
    }

    if len(bytesErr) != 0 {
        fmt.Printf("stderr is not nil: %s", bytesErr)
        return
    }

    bytes, err := ioutil.ReadAll(stdout)
    if err != nil {
        fmt.Println("ReadAll stdout: ", err.Error())
        return
    }

    if err := cmd.Wait(); err != nil {
        fmt.Println("Wait: ", err.Error())
        return
    }

    fmt.Printf("stdout: %s", string(bytes))



}

func main() {
  var (
     listenAddress = flag.String("web.listen-address", ":9101",
"Address to listen on for web interface and telemetry.")
```

```
    metricsPath   = flag.String("web.telemetry-path", "/metrics",
"Path under which to expose metrics.")
    cmd           = flag.String("free", "", "command to run")
    nginxTimeout  = flag.Duration("nginx.timeout",
5*time.Second, "Timeout for trying to get stats from HA.")
   showVersion   = flag.Bool("version", false, "Print version
information.")
//    debug         = flag.Bool("debug", false, "Print debug
information.")
  )
  flag.Parse()

  if *showVersion {
   fmt.Println(os.Stdout, version.Print("shannon_exporter"))
   os.Exit(0)
  }

  log.Infoln("Starting shannon_exporter", version.Info())
  log.Infoln("Build context", version.BuildContext())

  run()



  exporter, _ := NewExporter(*cmd, *nginxTimeout)



  prometheus.MustRegister(exporter)
  prometheus.MustRegister(version.NewCollector("nginx_exporte
r"))
```

```
  http.Handle(*metricsPath, prometheus.Handler())
  http.HandleFunc("/", func(w http.ResponseWriter, r
*http.Request) {
    w.Write([]byte(`<html>
      <head><title>Haproxy Exporter</title></head>
      <body>
      <h1>Haproxy Exporter</h1>
      <p><a href='` + *metricsPath + `'>Metrics</a></p>
      </body>
      </html>`))
  })

  log.Fatal(http.ListenAndServe(*listenAddress, nil))


  }
```

// 前面的 scrape describe Collect 是struct类型的成员函数，这几个函数并没有直接在这个go里被调用，而是 MustRegister注册进去了它们。

// http.Handle 里的prometheus.Handler 将上一部2个Mustregister 的注册 关联进入http.handle. 也就进一步注册进入了httpserver-listerner

// http.lister是组赛程序，只有启动后 被curl的时候 才会执行。并且由于之前经过2步的注册，跟exporter的成员函数建立了关联（其实是指针关联） 所以，每次被curl的时候 所有上面的成员函数都会被调用

// 例如上面的up.set(1) 就是只有被curl的时候才会被调用，成员函数是作为 生成metrics的入口

注： 本Go exporter 源代码 是半年前写的了 其中一些地方 可能需要进一步修改才可以正常使用 （本人go开发能力也比较菜 - _ -）

（三） 个人建议

通过上面的对一个exporter源代码的讲解
我们也看得出来
编写一个 exporter 远远比 写一个**pushgateway**脚本要复杂的多的多

个人建议：除非是工作中真的有需要（例如：社区的exporters都不能满足需求，且对于监控客户端的规范化比较严格） 且对自己的编程能力有自信
那么可以自行开发new exporter

exporter-prometheus 5s GET => exporter Handler-> handler-lister 开发的过程中 要特别注意 各种文件句柄等等资源的 完整回收
因为一个非常被频繁调用的后台程序中 一旦循环中出现资源泄漏 那么后果是很严重的
（之前其实就遇到过一次 资源没有正确回收 导致每一次promethue_server访问过来 都造成僵尸进程 最终服务器被拖垮的窘境 …囧 当然了 如果是 开发高手的话 相信我的担心也就多余啦 ~ ~）

所以 如果企业中 运维开发的实力不够过硬，我还是尽量建议大家 使用现成的社区exporters, 配合自行开发pushgateway脚本 😄