

目录

Introduction	1.1
作者	1.2
版本变更历史	1.3
第一章 天降奇兵	1.4
监控系统的发展	1.4.1
Prometheus来了	1.4.2
初识Prometheus	1.4.3
使用NodeExporter监控主机	1.4.4
Prometheus核心组件	1.4.5
百里挑一	1.4.6
小结	1.4.7
第二章 探索PromQL	1.5
什么是Metrics和Labels	1.5.1
Metrics类型	1.5.2
初识PromQL	1.5.3
PromQL操作符	1.5.4
PromQL内置函数	1.5.5
在HTTP API中使用PromQL	1.5.6
新的存储层	1.5.7
最佳实践	1.5.8
小结	1.5.9
第三章 Prometheus告警处理	1.6
AlertManager简介	1.6.1
部署AlertManager	1.6.2
自定义Prometheus告警规则	1.6.3
基于Label的动态告警处理	1.6.4
抑制机制 inhibit	1.6.5

告警模板	1.6.6
第三方集成	1.6.7
小结	1.6.8
第四章 可视化一切	1.7
Grafana简介	1.7.1
安装Grafana	1.7.2
使用Prometheus数据源	1.7.3
创建监控Dashboard	1.7.4
基于Grafana的告警配置	1.7.5
小结	1.7.6
第五章 扩展Prometheus	1.8
常用Exporter	1.8.1
使用Java创建自定义Metrics	1.8.2
使用Golang创建自定义Metrics	1.8.3
扩展Spring Boot应用支持应用指标采集	1.8.4
小结	1.8.5
第六章 Prometheus服务发现	1.9
为什么需要服务发现	1.9.1
基于文件的服务发现机制	1.9.2
基于Consul的服务发现机制	1.9.3
基于DNS的服务发现机制	1.9.4
服务发现与Relabel机制	1.9.5
小结	1.9.6
第七章 集群与高可用	1.10
数据管理	1.10.1
远程数据存储	1.10.2
联邦集群	1.10.3
使用Prometheus Operator管理Prometheus	1.10.4
使用Promgen管理Prometheus	1.10.5
从1.0迁移到2.0	1.10.6

总结	1.10.7
第八章 使用Prometheus Operator管理Prometheus	1.11
第九章 Kubernetes监控实战	1.12
 Kubernetes简介	1.12.1
 搭建Kubernetes本地测试环境	1.12.2
 Prometheus Vs Heapster	1.12.3
 在Kubernetes下部署Prometheus	1.12.4
 采集Kubelet状态	1.12.5
 采集集群状态	1.12.6
 采集应用资源用量	1.12.7
 采集集群节点状态指标	1.12.8
 使用Grafana创建可视化仪表盘	1.12.9
 基于Prometheus实现应用的弹性伸缩	1.12.10
 总结	1.12.11
参考资料	1.13

Prometheus: 云原生监控之道

Prometheus: 云原生监控之道

全书组织

我们假定你已经对Linux系统以及Docker技术有一定的基本认识，也可能使用过像Java, Golang这样的编程语言，所以我们不会像对一个没有任何基础的初学者那样事无巨细的讲述所有事。

第1章，是Prometheus基础的综述，通过一个简单的（使用Prometheus采集主机的监控数据）例子来了解Prometheus是什么？能做什么？以及Prometheus的基础架构。希望读者从这一章中能对Prometheus有一个基本的理解和认识。

第2章中我们先讲述Prometheus的数据模型以及，以及时间序列存储方式。并且利用Prometheus的数据查询语言(Prometheus Query Language)对监控数据进行查询，聚合以及计算等。

第3章，我们重点放在监控告警部分，作为监控系统的重要能力之一，我们需要及时了解系统环境的变化，因此这一章，我们会介绍如何在Prometheus中定义告警规则，并且结合Prometheus中的另外一个重要组件AlertManager来对告警进行处理。

第4章，"You can't fix what you can't see"我们讨论可视化，如何基于Grafana这一可视化工具自定义我们的可视化仪表盘，介绍Grafana作为一个通用的可视化工具是如何与Prometheus进行配合的。

可以看出，从第1章到第4章的部分是基础性的，对大部分的研发或者运维人员来说可以快速掌握，并且能够使用Prometheus来完成一些基本的日常任务。余下的章节我们会关注到Prometheus的高级用法部分。

第5章，我们会介绍一些常用的Prometheus Exporter的使用场景以及用法。最本章的最后部分我们会带领读者通过Java和Golang实现我们的Exporter，并且在现有应用系统上添加Prometheus支持，从而实现应用层面的监控对接。

第6章，我们会了解如何通过Prometheus的服务发现能力，实现动态监控。特别是在云平台或者容器平台中，资源的创建和销毁成本变得如此的低的情况下，通过服务发现自动的去发现监控目标，能够充分简化Prometheus的运维和管理难度。

第7章，Prometheus在单个节点的情况下能够轻松完成对N级别资源的监控，但是监控的目标资源以及数据量变得更大的时候，我们如何实现对Prometheus的扩展。这一章节我们重点讨论Prometheus的数据管理和高可用方面。

第8章，这一章节中我们的另外一位重要成员Kubernetes将会登场，这里我们会带领读者对Kubernetes有一个基本的认识，并且通过Prometheus构建我们的容器云监控系统。并且介绍如何通过Prometheus与Kubernetes结合实现应用程序的弹性伸缩。

目录结构

```
book
|- CHANGELOGS.md 版本更新历史
|- SUMMARY.md 目录
|- examples 示例代码
  |- ch[n] 各个章节的示例代码以及测试环境
|- chapter[n] 章节
  |- static/ 静态文件资源
  |- README.md 章节头
  |- SUMMARY.md 章节尾
```

本地预览

```
npm install
```

Start Local Preview

```
npm run start
```

作者介绍

郑云龙，全栈工程师，CNCF基金会Certified Kubernetes Administrator。在敏捷和DevOps领域有丰富的实践经验，曾作为敏捷和DevOps技术教练向多家大型企业提供咨询和培训。当前在一家容器创业公司负责CaaS产品研发和设计。

Prometheus实践之路

“You can't fixed what you can't see”。Prometheus被誉为下一代监控系统的首选，是继Kubernetes之后的第一个CNCF基金会顶级项目。本书是一本介绍Prometheus以及周边相关技术的实践书籍。本书主要分为三个部分。第一部分，主要通过一个实际的案例介绍了Prometheus的是什么？能做什么？以及Prometheus的基础架构，让读者对Prometheus有一个基本的认识以及概念。第二部分，本书则重点放在Prometheus的高级实践，包括社区提供的已经实现的Exporter的使用场景以及方法，以及教会读者如果通过Prometheus提供的Client Library根据自身需求创建自定义的Exporter。基于如何使用Prometheus的服务发现能力，提升平台的动态能力。在第二部分的最后我们会介绍如何对Prometheus运维管理，根据需求的不同实现Prometheus的扩容。以及如何通过一些其他的开源工具如Prometheus Operator实现对Prometheus的管理。最后，在第三部分，我们将会在Kubernetes下基于Prometheus构建我们的容器云监控平台，并且基于监控数据实现应用的Auto Scalling。

在通读这些内容后，相信读者能够对Prometheus有一个全面的认识。

版本更新历史

- 2018/1/18 初始化目录

第一章： 天降奇兵

监控系统的发展

为什么需要监控

一般来说对于一个组织而言，引入监控系统一般都是为了解决以下几个主要问题：

- 组织需要了解系统在什么时候发生了异常，当发生异常的时候可以及时进行处理，避免产生业务层面的影响，或者提前处理预防问题的发生；
- 用于对问题进行定位，以及历史数据跟踪；
- 通过历史监控数据，为技术或者业务决策提供数据支撑；
- 用于支持其它的系统或者流程，比如自动化、安全、测试等；

面临的挑战

而过去对于大部分组织而言，当使用监控系统时，通常往往会面临各种挑战：

- 过高的运维成本，比如zabbix或者nagios这类工具网络需要有专业的人员进行安装，配置和管理，而且过程并不简单；
- 同时很难对监控系统自身做扩展，以适应监控规模的变化；
- 同时监控系统本身也会限制技术和业务本身；
- 监控指标与业务分离，无法有效的支撑业务需求；

好比客户可能关注的是服务的可用性，以及服务的SLA等级，而你的监控系统却只能根据CPU使用率去产生告警。业务和监控指标之间无法有效协作。

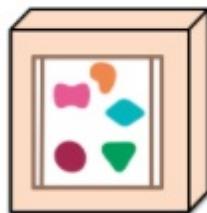
除了这些基本的挑战以外，应用架构以及基础设施架构的变化同样也带来了巨大的挑战。

从单体架构到微服务架构，通过各个小的独立的进程支撑整个业务，每一个部分都可以独立根据需求进行独立开发，独立部署，独立伸缩。应用程序无论从结构，还是实例都是不断演进变化的。

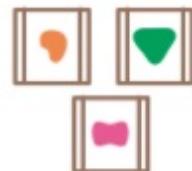
A monolithic application puts all its functionality into a single process...



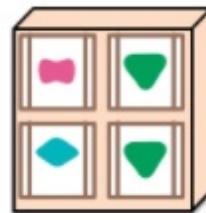
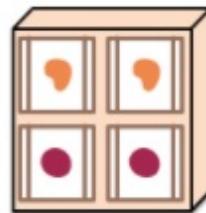
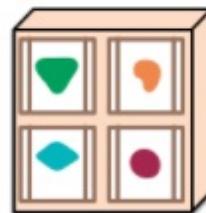
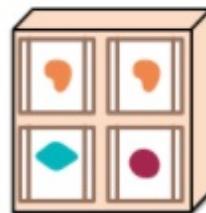
... and scales by replicating the monolith on multiple servers



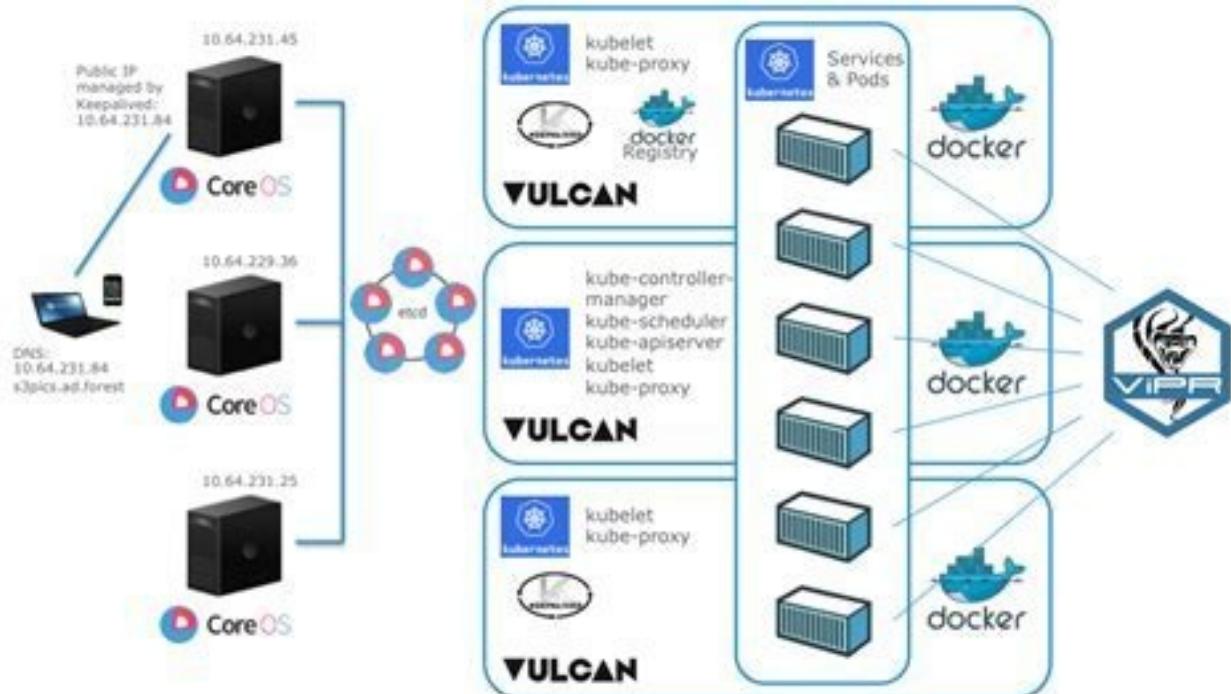
A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



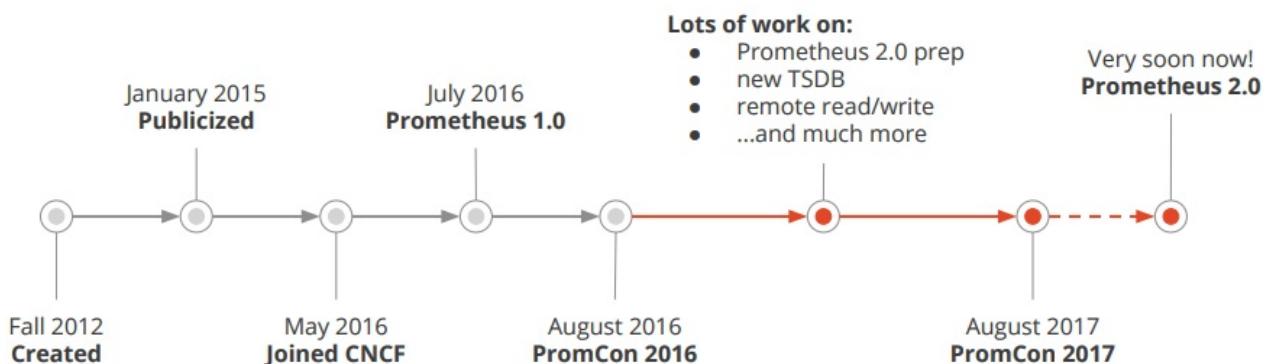
物理主机、虚拟机、容器运行应用的基础设施正在不断的发生变化。这些变化让底层资源的创建、销毁变得更加频繁。同时基础设施还可能根据当前业务的需求不断发生变化，因此基础设施也变成了一个动态的基础服务。



这一系列的变化，都在预示着一件事情，就是上一代监控系统已经无法有效的应对当前软件设计所带来的变化，我们需要的是一个全新的监控系统，来适应当前万物皆云的现状。

Prometheus来了

Prometheus受启发于Google的Brogmon监控系统（相似的Kubernetes是从Google的Brog系统演变而来），从2012年开始由前Google工程师在Soundcloud以开源软件的形式进行研发，并且于2015年早期对外发布早期版本。2016年5月正式成为继Kubernetes之后的第二个正式加入CNCF基金会的项目，同年6月正式发布1.0版本。2017年底发布了基于全新存储层的2.0版本，能更好地与容器平台，云平台配合。



目前已经有超过650+位贡献者参与到Prometheus的研发工作上，并且超过120+项的第三方集成。

Prometheus是什么？

- 一个时序数据库

首先Prometheus是一个基于时间序列（time-series）的数据库系统。Prometheus与其他主流的监控系统一样均采用了基于时间序列的方式对数据进行存储。相比的时序数据库（例如：OpenTsdb），Prometheus中采集到的监控样本数据，通过指标名称(metric)和一组描述当前样本特征维度的键值对（Labels）进行唯一标示（例如：`api_http_requests_total{method="POST", handler="/messages"}`）。通过这种多维的数据模型，Prometheus可以提供高效的存储和查询能力。

Prometheus将数据存储在内存和本地磁盘上。当需要Prometheus进行扩展时，可以通过功能分片或者联邦集群实现。关于如果实现Prometheus扩展的部分我们会在第4章进行讨论。

- 一个数据查询引擎

Prometheus提供了灵活高效的数据查询系统，通过Promethues自定义的数据查询语言，可以对数据进行查询和聚合，实现各种报表和数据分析。

关于使用Prometheus进行数据聚合分析的部分我们会在第2章节进行讨论。

- 一个监控告警预警平台

Prometheus支持用户创建基于Prometheus的自定义查询语言，定义告警。并且结合AlertManager组件实现件监控告警，可以与第三方工具和平台进行集成。

- 一个开放的监控框架

社区同时提供了大量的第三方Exporter。可以快速实现对诸如服务器、容器、中间件的监控指标采集。Prometheus同时提供超过10种以上的客户端SDK，基于这些客户端SDK，可以轻松实现自己的Exporter或者直接在应用程序上集成Prometheus。

Prometheus的核心优势

完整的监控支持

Prometheus作为系统监控方案(不同于应用监控APM)，提供了完整的监控支持，包括但不限于：基础设施监控，中间件监控，应用监控；

提供了完整的可视化，以及告警支持，并且易于集成。

强大的数据模型

所有的监控指标数据都是基于标签的多维度数据。基于这些维度我们可以方便的对监控数据进行聚合，过滤，裁剪。并且这些数据均以时间序列的形式存储在本地磁盘中。

强大的查询语言

通过Prometheus提供的自定义查询语言PromQL，我们可以直接对基于时间序列的数据进行数学运算(加，减，乘，除)，聚合并立即得到结果。基于这些运算结果我们可以方便地绘制出需要的可视化图标。

例如，通过PromQL我们可以回答类似于以下问题：

- 在过去一段时间95%的应用延迟的分布范围？
- 在4小时候后，磁盘空间占用大致会是什么情况？
- CPU占用率前5位的服务有哪些？

易于管理

Prometheus核心部分只有一个单独的二进制文件，不存在任何的第三方依赖(数据库，缓存等等)。唯一需要的就是本地磁盘，因此不会潜在级联故障的风险。

Prometheus基于Pull模型的架构方式，可以在任何地方（本地电脑，开发环境，测试环境）搭建我们的监控系统。

对于一些复杂的情况，我们可以基于服务发现(Service Discovery)用于动态发现监控目标。

高效

对于Prometheus而言大量的监控任务，意味着有大量样本数据的产生。而Prometheus可以高效的处理这些数据，对于单实例的Prometheus Server，可以处理：

- 数以百万的监控指标
- 每秒处理数十万的数据点。

可扩展

Prometheus是如此简单，因此你可以在每个数据中心，每个团队运行度量的Prometheus Server。Prometheus对于联邦集群地支持，可以让Prometheus从其他Prometheus Server拉取监控指标样本。因此当对于单实例Prometheus Server来说监控的任务量过大时，我们可以使用功能分区(sharding)+联邦集群(federation)来进行扩展。

易于集成

使用Prometheus可以快速搭建监控服务，并且可以非常方便的在应用程序中进行集成。目前支持：Java，JMX，Python，Go，Ruby，.Net，Node.js等等语言的客户端SDK，基于这些SDK可以快速让应用程序纳入到Prometheus的监控当中，或者开发自己的监控数据收集程序。同时这些客户端收集的监控数据，不仅仅支持Prometheus，还能支持Graphite这些其他的监控工具。

同时Prometheus还支持与其他的监控系统进行集成：Graphite，Statsd，Collected，Scollector，munin，Nagios等。

同时Prometheus还存在官方的第三方实现的监控数据采集支持：JMX，CloudWatch，EC2，MySQL，PostgreSQL，Haskell，Bash，SNMP，Consul，Haproxy，Mesos，Bind，CouchDB，Diango，Memcached，RabbitMQ，Redis，RethinkDB，Rsyslog等等。

可视化

Prometheus Server中自带了一个Prometheus UI，通过这个UI我们可以方便的直接对数据进行查询，并且可以直接以图的形式展示数据。同时Prometheus还提供了一个独立的基于Ruby On Rails的Dashboard解决方案Promdash。最新的Grafana可视化工具也已经提供了完整的Prometheus支持，基于Grafana可以创建更加精美的监控图标。基于Prometheus提供的API还可以实现自己的监控可视化UI。

开放性

通常来说当我们需要监控一个应用程序时，一般需要该应用程序提供对相应监控系统协议的支持。因此应用程序会与所选择的监控系统进行绑定。为了减少这种绑定所带来的限制。对于决策者而言要么你就直接在应用中集成该监控系统的支持，要么就在外部创建单独的服务来适配不同的监控系统。

而对于Prometheus来说，使用Prometheus的client library的输出格式不止支持Prometheus的格式化数据，也可以输出支持其它监控系统的格式化数据，比如Graphite等。

因此你甚至可以在不使用Prometheus的情况下，采用Prometheus的client library来让你的应用程序支持监控数据采集。

初识Prometheus

本节将带来读者在本地基于搭建一个单实例的Prometheus Server，以此让读者能够对Prometheus有一个直观的认识。

环境准备

我们使用[Vagrant](#)创建了一个ubuntu/xenial64本地的虚拟机。我们将在该环境下安装部署Prometheus。

这里我们使用Vagrantfile来定义我们的基础环境，我们使用ubuntu/xenial64作为基础镜像，并且为虚拟机分配了一个私有的IP地址192.168.33.10，通过该IP地址我们可以直接在本地访问运行在该虚拟机中的服务。

Vagrantfile内容如下：

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.network "private_network", ip: "192.168.33.10"
end
```

启动虚拟机

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: hostonly
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: ubuntu
    default: SSH auth method: password
==> default: Machine booted and ready!
[default] GuestAdditions 5.1.30 running --- OK.
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /Workspace/books/prometheus-in-action/examples/ch1
==> default: Machine already provisioned. Run `vagrant provision` or use the `--provision` flag to force provisioning. Provisioners marked to run always will still run.
```

启动完成后，我们可以通过命令vagrant ssh登录到该虚拟机。

```
$ vagrant ssh
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-112-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

96 packages can be updated.
1 update is a security update.

Last login: Wed Jan 31 01:41:40 2018 from 10.0.2.2
ubuntu@ubuntu-xenial:~$
```

如何获取二进制包

Prometheus的源代码托管在Github的仓库<https://github.com/prometheus/prometheus>在下，在[releases](#)链接下可以找到Prometheus所有已发布版本的软件包。

安装Prometheus Server

创建本地用户

```
sudo useradd --no-create-home prometheus

sudo mkdir /etc/prometheus
sudo mkdir /var/lib/prometheus

sudo chown prometheus:prometheus /etc/prometheus
sudo chown prometheus:prometheus /var/lib/prometheus
```

获取并安装软件包

```
cd ~
curl -LO https://github.com/prometheus/prometheus/releases/download/v2.0.0/prometheus-2.0.0.linux-amd64.tar.gz
tar xvf prometheus-2.0.0.linux-amd64.tar.gz
```

```
sudo cp prometheus-2.0.0.linux-amd64/prometheus /usr/local/bin/
sudo chown prometheus:prometheus /usr/local/bin/prometheus
```

创建Prometheus配置文件

创建配置文件: /etc/prometheus/prometheus.yml

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
```

```
$ sudo chown prometheus:prometheus /etc/prometheus/prometheus.yml
```

这里我们让Prometheus Server每15秒，轮询一次Prometheus Server暴露的监控采集地址<http://localhost:9090/metrics>。

运行Prometheus

```
$ sudo -u prometheus /usr/local/bin/prometheus \
--config.file /etc/prometheus/prometheus.yml \
--storage.tsdb.path /var/lib/prometheus/
```

```
level=info ts=2018-02-05T05:58:12.438943323Z caller=main.go:215 msg="Starting Prometheus" version="(version=2.0.0, branch=HEAD, revision=0a74f98628a0463dddc90528220c94de5032d1a0)"
level=info ts=2018-02-05T05:58:12.439697472Z caller=main.go:216 build_context="(go=go1.9.2, user=root@615b82cb36b6, date=20171108-07:11:59)"
level=info ts=2018-02-05T05:58:12.4403636Z caller=main.go:217 host_details="(Linux 4.4.0-112-generic #135-Ubuntu SMP Fri Jan 19 11:48:36 UTC 2018 x86_64 ubuntu-xenial (none))"
level=info ts=2018-02-05T05:58:12.445696612Z caller=web.go:380 component=web msg="Start listening for connections" address=0.0.0.0:9090
level=info ts=2018-02-05T05:58:12.445720858Z caller=main.go:314 msg="Starting TSDB"
level=info ts=2018-02-05T05:58:12.445785952Z caller=targetmanager.go:71 component="target manager" msg="Starting target manager..."
level=info ts=2018-02-05T05:58:12.761069867Z caller=main.go:326 msg="TSDB started"
level=info ts=2018-02-05T05:58:12.762288533Z caller=main.go:394 msg="Loading configuration file" filename=/etc/prometheus/prometheus.yml
level=info ts=2018-02-05T05:58:12.764189762Z caller=main.go:371 msg="Server is ready to receive requests."
```

创建Prometheus Server的Service Unit文件

```
sudo vim /etc/systemd/system/prometheus.service
```

```
[Unit]
Description=Prometheus
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
--config.file /etc/prometheus/prometheus.yml \
--storage.tsdb.path /var/lib/prometheus/ \
--web.console.templates=/etc/prometheus/consoles \
--web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target
```

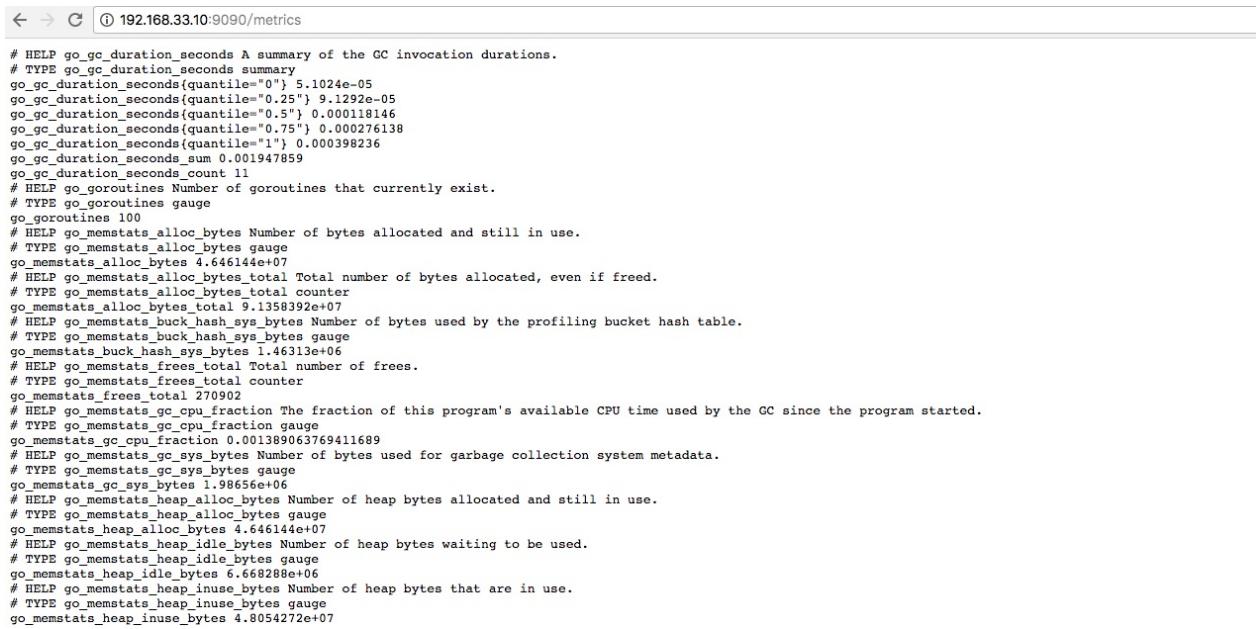
```
sudo systemctl daemon-reload
sudo systemctl status prometheus
sudo systemctl enable prometheus
sudo systemctl restart prometheus
```

使用Expression Browser

到目前为止我们已经完成了Prometheus Server的部署，在Prometheus启动完成后，访问<http://192.168.33.10:9090>即可访问Prometheus内置的UI程序Expression Browser。



访问<http://192.168.33.10:9090/metrics>，我们可以查看当前Prometheus Server自身的监控指标数据。

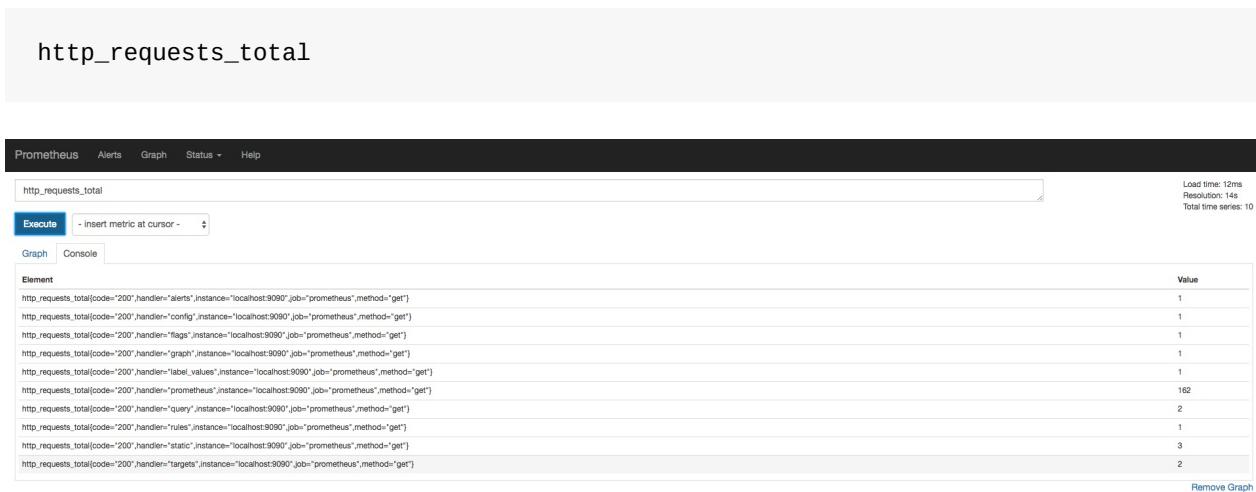


```

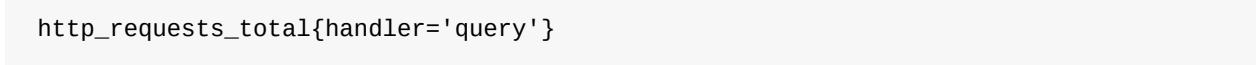
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 5.1024e-05
go_gc_duration_seconds{quantile="0.25"} 9.1292e-05
go_gc_duration_seconds{quantile="0.5"} 0.00018146
go_gc_duration_seconds{quantile="0.75"} 0.000276138
go_gc_duration_seconds{quantile="1"} 0.000398236
go_gc_duration_seconds_sum 0.001947859
go_gc_duration_seconds_count 11
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 100
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 4.646144e+07
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 9.1358392e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.46313e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 270902
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0.001389063769411689
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 1.98656e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 4.646144e+07
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 6.668288e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 4.8054272e+07

```

回到Expression Browser, <http://192.168.33.10:9090/graph>, 并切换到Console标签。我们可以通过，输入表达式http_requests_total来查看Prometheus Server的Http请求量的情况。在UI中输入表达式。

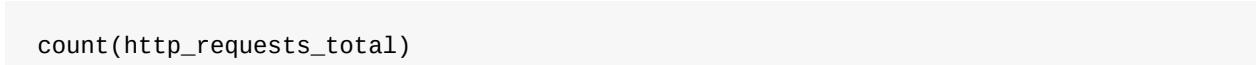


我们还可以通过返回数据样本中标签对数据进行过滤，例如我们只关心handler=query的请求次数。则通过在表达式中对样本特征进行限定来获取响应的数据：



```
http_requests_total{handler='query'}
```

如果只需要计算当前表达式返回的时间序列的条数则可以使用count函数进行计算。

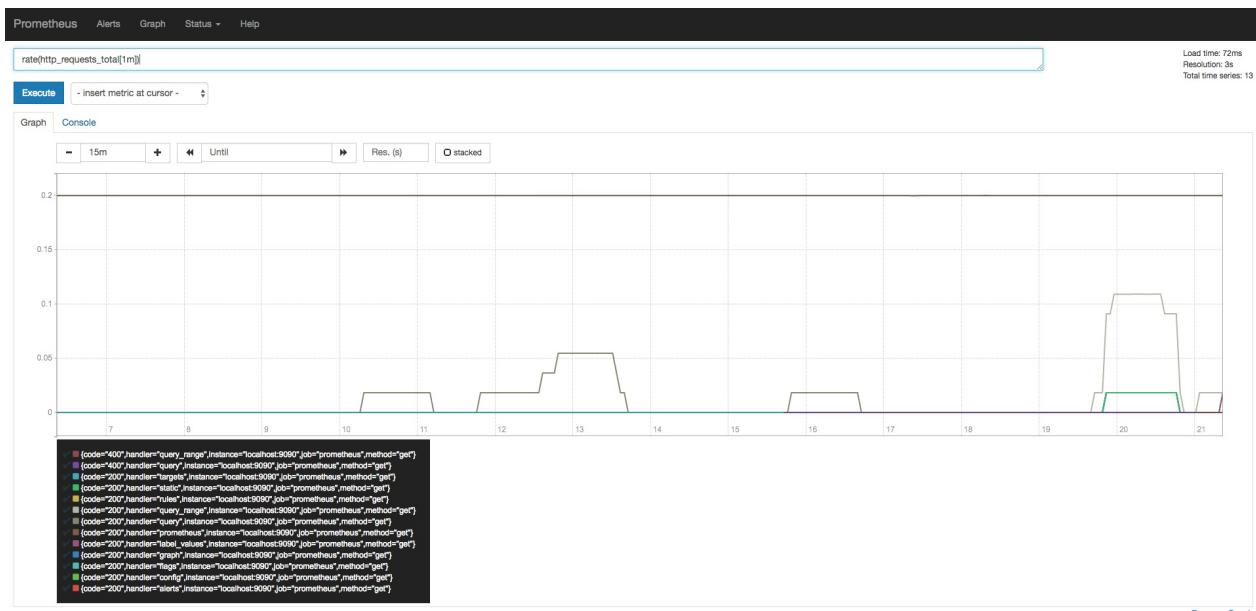


```
count(http_requests_total)
```

使用Prometheus的Expression Browser UI，我们还可以直接通过表达式计算实时产生统计图表。回到Expression Browser，<http://192.168.33.10:9090/graph>，并切换到Graph标签。

输入以下表达式，可以统计当前Prometheus Server每秒接收的请求次数速率。

```
rate(http_requests_total[1m])
```



这里使用的表达式即Prometheus提供的PromQL查询语言，通过PromQL我们可以方便的按需对数据进行查询，过滤，分片，聚合等操作，同时PromQL中还提供了大量的内置函数，可以实现复杂的数据统计分析需求。

任务和实例

在Prometheus中每一个可以用于采集数据的端点被称为一个实例（Instance）。实例通过URL端点的形式将自身采集到的监控数据样本对外暴露。而Prometheus则直接从这些URL端点中获取监控数据。一组用于相同采集目的的实例，或者同一个采集进程的多个副本，称可以为一个任务（Job）。

```
* job: api-server
  * instance 1: 1.2.3.4:5670
  * instance 2: 1.2.3.4:5671
  * instance 3: 5.6.7.8:5670
  * instance 4: 5.6.7.8:5671
```

回到prometheus.yml配置中，我们可以看到scrape_configs节点，定义了一个scrape_config的数组，每一个scrape_config配置项即对应了一个Job。当使用static_configs定义监控目标时，targets即对应一个任务中的多个实例。

```
scrape_configs:
- job_name: 'prometheus'
  scrape_interval: 5s
  static_configs:
    - targets: ['localhost:9090']
```



The screenshot shows the Prometheus UI's 'Targets' page. At the top, there are navigation links: Prometheus, Alerts, Graph, Status, and Help. Below the header, it says 'Targets' and shows a table for 'prometheus (1/1 up)'. The table has columns: Endpoint, State, Labels, Last Scrape, and Error. One row is listed: 'http://localhost:9090/metrics' with 'UP' state, 'Instance="localhost:9090"' in the Labels column, '4.529s ago' in the Last Scrape column, and no error.

http://p2n2em8ut.bkt.clouddn.com/prometheus_ui_targets.png

我们也可以访问<http://192.168.33.10:9090/targets>直接从Prometheus的UI中查看当前所有的任务以及每个任务对应的实例信息。

因此当我们需要采集不同的监控指标(例如：主机、MySQL、Nginx)时，我们只需要运行相应的监控采集程序，并且让Prometheus Server知道这些Exporter实例的访问地址。这些用于向Prometheus暴露监控URL端点的程序我们可以称为一个Exporter。

使用NodeExporter监控主机

上一小节，我们已经部署了一个Prometheus Server的实例，并且通过修改Prometheus的配置文件，使Prometheus Server可以采集自身的监控指标。并且我们可以通过Prometheus内置的UI，直接对数据进行查询，过滤，聚合。还可以直接以图表的形式对数据进行展示。

除此之外也了解到，为了满足特定监控目的的需求，需要运行单独Exporter程序，从而使Prometheus Server可以从该Exporter暴露的监控端点获取监控数据。

接下来，我们将尝试通过部署Node Exporter实现对主机监控指标（CPU，内存，磁盘）的采集。

安装Node Exporter

创建用户

```
sudo useradd --no-create-home node_exporter
```

获取并安装软件包

```
cd ~
curl -LO https://github.com/prometheus/node_exporter/releases/download/v0.15.1/node_exporter-0.15.1.linux-amd64.tar.gz

tar xvf node_exporter-0.15.1.linux-amd64.tar.gz

sudo cp node_exporter-0.15.1.linux-amd64/node_exporter /usr/local/bin
sudo chown node_exporter:node_exporter /usr/local/bin/node_exporter
rm -rf node_exporter-0.15.1.linux-amd64.tar.gz node_exporter-0.15.1.linux-amd64
```

创建Node Exporter的Service Unit文件

```
sudo vim /etc/systemd/system/node_exporter.service
```

```
[Unit]
Description=Node Exporter
Wants=network-online.target
After=network-online.target

[Service]
User=node_exporter
Group=node_exporter
Type=simple
ExecStart=/usr/local/bin/node_exporter

[Install]
WantedBy=multi-user.target
```

启动Node Exporter

```
service node_exporter start
```

NodeExporter启动后，访问<http://192.168.33.10:9100/metrics>，我们可以获取到当前NodeExporter所在主机的当前资源使用情况的监控数据。



```
# HELP gc_gc_duration_seconds A summary of the GC invocation durations.
# TYPE gc_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 2.5478e-05
go_gc_duration_seconds{quantile="0.25"} 0.000196e-05
go_gc_duration_seconds{quantile="0.5"} 9.4018e-05
go_gc_duration_seconds{quantile="0.75"} 0.000291357
go_gc_duration_seconds{quantile="1"} 0.007849046
go_gc_duration_seconds{sum} 0.319006564
go_gc_duration_seconds{count} 486
# HELP goroutines Number of goroutines that currently exist.
# TYPE goroutines gauge
go_goroutines 11
# HELP gc_info Information about the Go environment.
# TYPE gc_info gauge
go_info_github_commit{"git_describe": "v1.9.2"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.152776e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 7.88331792e+08
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.512452e+06
# HELP go_gc_frees_total Total number of frees.
# TYPE go_gc_frees_total counter
```

http://p2n2em8ut.bkt.clouddn.com/node_exporter_metrics.png

配置主机监控采集任务

配置Prometheus采集主机信息

编辑配置文件/etc/prometheus/prometheus.yml，并添加以下内容：

```
- job_name: 'node_exporter'  
  scrape_interval: 5s  
  static_configs:  
    - targets: ['localhost:9100']
```

这里我们添加了一个新的Job名字为node_exporter。并且定义了一个实例为localhost:9100。

完整的Prometheus配置文件/etc/prometheus/prometheus.yml如下：

```
global:  
  scrape_interval: 15s  
  
scrape_configs:  
  - job_name: 'prometheus'  
    scrape_interval: 5s  
    static_configs:  
      - targets: ['localhost:9090']  
  - job_name: 'node_exporter'  
    scrape_interval: 5s  
    static_configs:  
      - targets: ['localhost:9100']
```

重新启动Prometheus Server

```
sudo service prometheus restart
```

验证结果

访问<http://192.168.33.10:9090/targets>查看所有的采集目标实例，这时我们可以看到新的采集任务：node_exporter以及相应的实例。

Targets				
node_exporter (1/1 up)		State	Labels	Last Scrape
Endpoint	http://localhost:9100/metrics	UP	Instance="localhost:9100"	4.018s ago
prometheus (1/1 up)				
Endpoint		State	Labels	Last Scrape
http://localhost:9090/metrics		UP	Instance="localhost:9090"	4.517s ago

http://p2n2em8ut.bkt.clouddn.com/node_exporter_targets.png

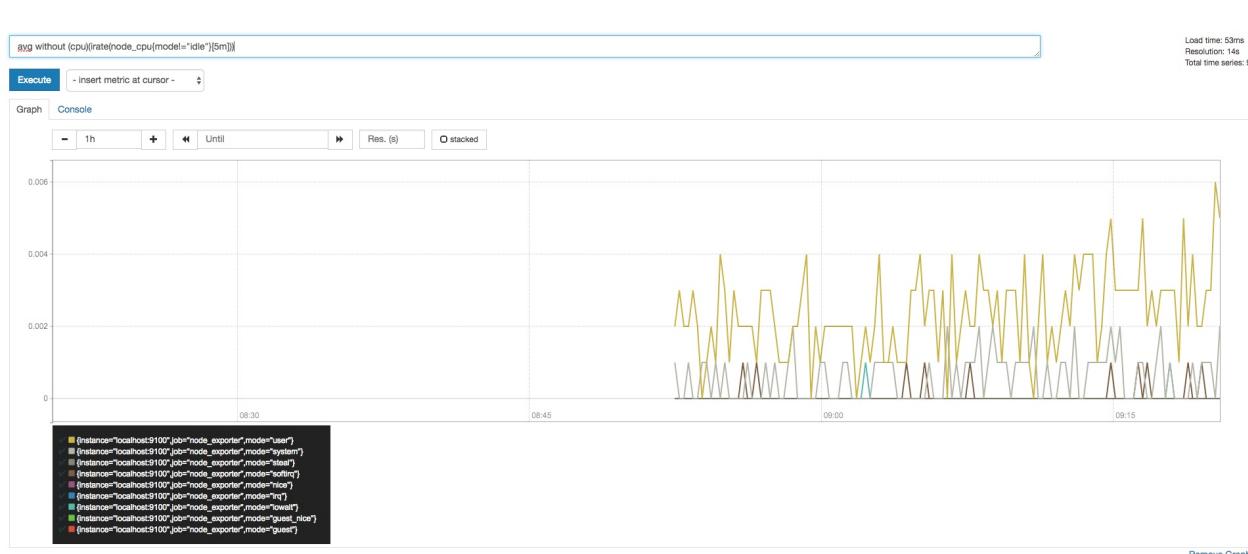
使用NodeExporter监控主机

这是我们可以通过PromQL语言在， Prometheus UI上直接查询主机相关资源的使用情况。

例如：

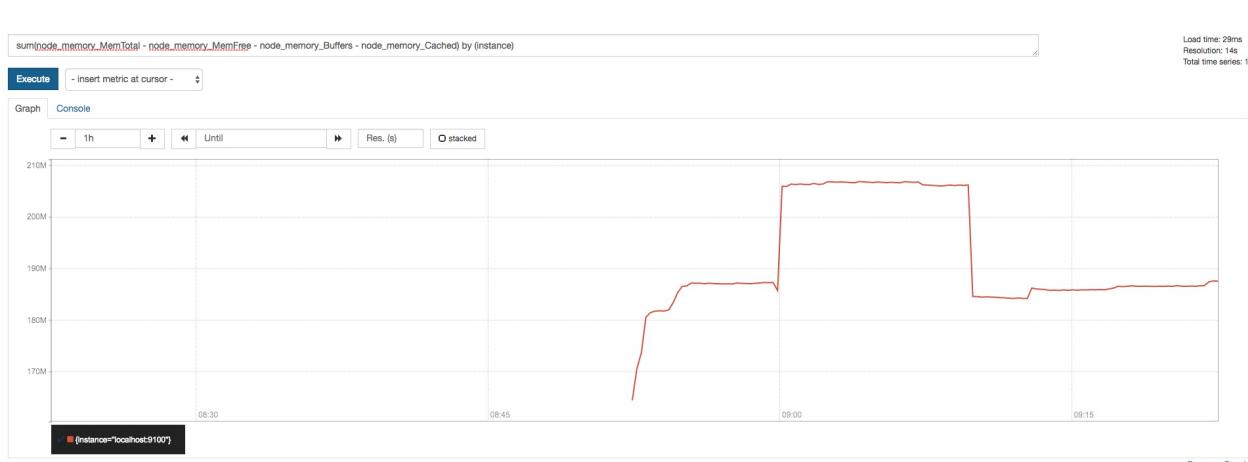
按CPU模式查询主机的CPU使用率：

```
avg without (cpu)(irate(node_cpu{mode!="idle"}[5m]))
```



按主机查询主机内存使用量：

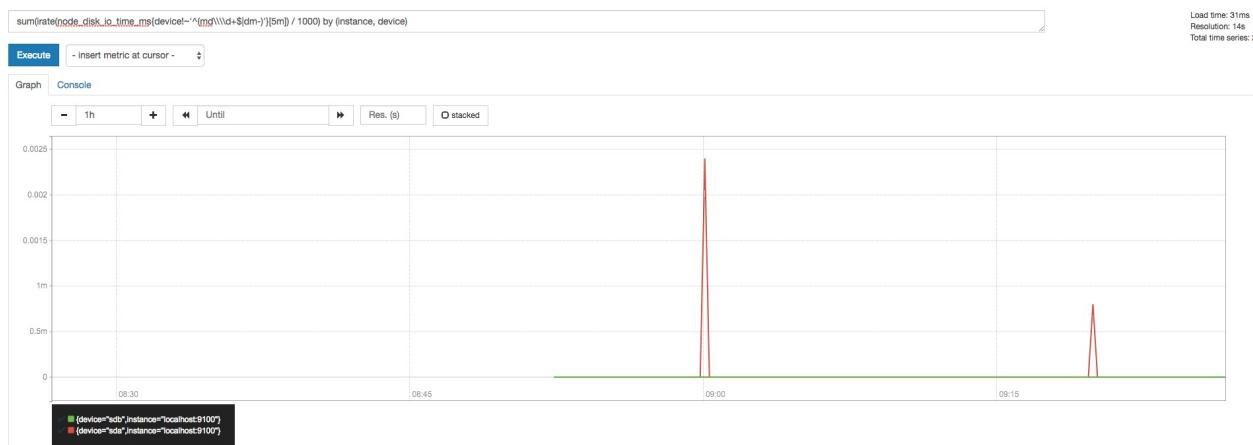
```
sum(node_memory_MemTotal - node_memory_MemFree - node_memory_Buffers - node_memory_Cached) by (instance)
```



按主机查询各个磁盘的IO状态：

```
sum(irate(node_disk_io_time_ms{device!~'^(\md\\\\\\d+$|dm-)'})[5m]) / 1000 by (instance, device)
```

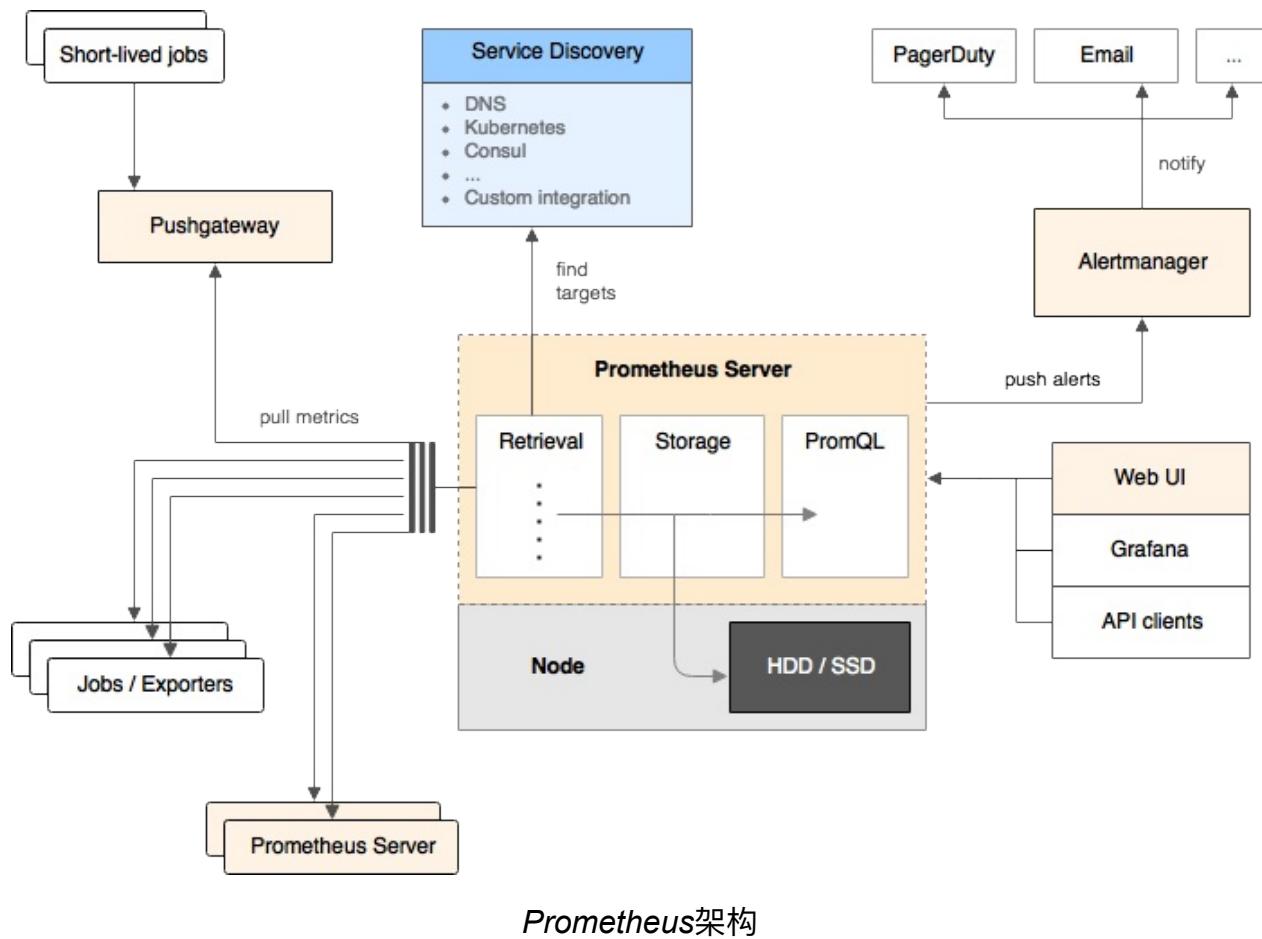
使用NodeExporter监控主机



Prometheus的核心组件

上一部分，我们通过部署Node Exporter成功获取到当前主机的资源使用情况。接下来我们将从Prometheus架构部分详细介绍组成Prometheus生态的各个组件。

下图展示Prometheus的基本架构：



Prometheus Server

在之前章节中我们通过部署Prometheus Server大致对Prometheus Server有了初步的认识。Prometheus Server是Prometheus组件中的核心部分，Prometheus Server的第一个能力是监控目标的检索和管理能力，Prometheus Server可以通过静态配置管理监控目标，也可以使用外部的Service Discovery服务动态管理监控目标。其次Prometheus

Server需要对采集到的监控数据进行存储，因此Prometheus Server本身也是一个时序数据库，在本地将采集到的监控数据按照时序的方式存在在本地磁盘当中。最后Prometheus Server对外提供了自定义的PromQL语言，支持用户按需查询监控数据。

Prometheus Server内置了一个Express Browser的UI，可以在这个UI上直接通过PromQL查询数据，并且可以通过图表的形式展示。

Prometheus Server还可以从其他的Prometheus Server实例中获取数据，因此在大规模监控的情况下，我们可以通过功能分区以及联邦集群的方式对Prometheus Server进行扩展。

Exporters

Exporter将监控数据采集的EndPoint暴露给Prometheus Server，Prometheus Server通过访问该Exporter提供的Endpoint端点，即可获取到需要采集的监控数据。

一般来说我们可以将Exporter分为2类：

- 直接采集：这一类Exporter直接内置了对Prometheus监控的支持，比如cAdvisor, Kubernetes, Etcd, gokit等，都直接内置了用于向Prometheus暴露监控数据的端点。
- 间接采集：间接采集，原有监控目标并不直接支持Prometheus，因此我们需要通过Prometheus提供的Client Library编写该监控目标的监控采集程序。例如：Mysql Exporter, JMX Exporter, Consul Exporter等。

AlertManager

在Prometheus Server中我们可以基于PromQL创建告警规则，如果满足PromQL定义的规则，则会产生一条告警，而告警的后续处理流程则由AlertManager进行管理。在AlertManager中我们可以与邮件、Slack等等内置的通知方式进行集成，也可以通过Webhook自定义告警处理方式。AlertManager即Prometheus提醒的告警处理中心。

PushGateway

由于Prometheus数据采集基于Pull模型进行设计，因此对于网络环境的配置上，必须要让Prometheus Server能够直接与Exporter进行通讯。当这种网络需求无法直接满足时，我们则可以利用PushGateway来进行中转。可以通过PushGateway将内部网络的监控数据

主动Push到Gateway当中。而Prometheus Server则可以采用同样Pull的方式从PushGateway中获取到监控数据。

百里挑一

Prometheus Vs Graphite

范围

Graphite专注于时序数据库本身，对外提供查询和图形可视化的功能。而其他监控相关的问题都需要由外部组件来解决。

Prometheus则是一个完整的监控系统，包括内置的数据采集，存储，查询，图形可视化以及基于时间序列数据的告警能力。

数据模型

Graphite和Prometheus一样，对基于时间序列的数据进行存储。

Graphite中的监控指标通过一组基于“.”的关键字维度组成：

```
stats.api-server.tracks.post.500 -> 93
```

而Prometheus中，每一个监控指标拥有对个基于key-value形式的标签组成。通过这些标签，我们可以更容易的对数据进行过滤，分组，以及查询。

```
api_server_http_requests_total{method="POST", handler="/tracks", status="500", instance=<sample1>} -> 34
api_server_http_requests_total{method="POST", handler="/tracks", status="500", instance=<sample2>} -> 28
api_server_http_requests_total{method="POST", handler="/tracks", status="500", instance=<sample3>} -> 31
```

存储

Graphite使用Whisper的格式将时间序列数据存储在本地磁盘中，这是一种RRD风格的数据库，期望采集到的样本数据能定期到达。每一条时间序列存储在单独的文件当中，并且新的样本数据会在一段时间后覆盖旧的样本数据。

Prometheus同样将时间序列数据分别存储在独立的本地磁盘中，但是运行样本已不同的周期进行采集。因为新的样本数据只是简单的追加到时间序列上，因此老的数据可能会保留较长的时间。Prometheus也适用于那些生命周期较短，变化频繁的时间序列。

总结

Prometheus提供了更灵活的数据模型以及查询语言，同时更容易运行以及集成到现有的环境中。但是如果你想要一个可以长期保存历史数据的集群解决方案，那么Graphite可能是一个更好的选择。

Prometheus Vs InfluxDB

InfluxDB是一个开源的时间序列数据库，同时具有支持扩展以及集群的商业版本。

Prometheus和InfluxDB之间存在着一些显著的差异，并且由各自适用的使用场景。

但是当将Kapacitor和InfluxDB一起考虑时，它们的组合与Prometheus与AlertManager解决了相同的问题。

范围

数据模型/存储

架构设计

总结

Prometheus Vs OpenTSDB

范围

数据模型

存储

总结

Prometheus Vs Nagios

范围

架构

总结

Prometheus vs. Sensu

范围

数据模型

架构

总结

本章小结

在这一章中，我们初步了解了Prometheus以及相比于其他相似方案的优缺点，可以为读者在选择监控解决方案时，提供一定的查考。同时我们介绍了Prometheus的生态以及核心能力，同时在本地使用Prometheus和NodeExporter搭建了一个主机监控的环境，并且对数据进行了聚合以及可视化，相信读者通过本章节，能够对Prometheus有一个直观的认识。

第二章：探索PromQL

什么是Metrics和Labels

理解时序数据模型

在之前的部分我们讲Prometheus除了是监控系统以外，本身也是一个时序（time series）数据库。从本质上将Prometheus将所有的数据，按照时间序列进行存储。每一条时间序列都通过唯一定义的指标名称即Metric名称以及一组key-value的键值对组成，这组键值对被称为Labels。

对于所有采集到的样本数据由一下几部分组成：

- 指标名称Metrics
- 用于描述样本的维度的键值对Labels
- 样本的采集时间
- 当前样本的值。

```
<--指标名称--><----- 标签 -----><-- 时间戳 --><- 值 ->
http_request_total{status="200", method="GET"}@1434417560938 => 94355
http_request_total{status="200", method="GET"}@1434417561287 => 94334
http_request_total{status="200", method="GET"}@1434417562344 => 94383

http_request_total{status="404", method="GET"}@1434417560938 => 38473
http_request_total{status="404", method="GET"}@1434417561287 => 38544
http_request_total{status="404", method="GET"}@1434417562344 => 38663

http_request_total{status="200", method="POST"}@1434417560938 => 4748
http_request_total{status="200", method="POST"}@1434417561287 => 4785
http_request_total{status="200", method="POST"}@1434417562344 => 4833
```

如果将Prometheus存储的数据理解为一个二维的平面，如下所示，我们则可以看到每一个唯一的指标名称+键值对采集到的样本数据，组成了以时间为X轴的一条间序数据。

指标名称和标签

指标的名称可以反映被监控系统的特征（比如，`httprequest_total` - 标示当前系统接收到的Http请求总量）。指标名称只能由ASCII字符，数字，下划线以及冒号组成。每一个指标名称都必须符合正则表达式`^/[a-zA-Z][a-zA-Z0-9:_]*$`。

而标签则反映出当前样本数据的多个特征维度，通过这些维度Prometheus可以对样本数据进行过滤，聚合等复杂操作。Prometheus提供了强大的自定义查询预言PromQL对这些数据进行查询。标签的名称只能由ASCII字符，数字，以及下划线组成。每一个标签名必须满足正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*`。其中以`_`作为前缀的标签，是系统保留的关键字，只能在系统内部使用。标签的值则可以包含任何Unicode编码的字符。

样本

在时序数据库中存储的每一个样本都有两个部分组成：

- 一个float64的浮点型数据表示当前样本的值
 - 一个精确到毫秒的时间戳

表示方法

通过给定的指标名称以及一组标签，可以唯一定义一条时间序列：

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

例如，如果一条时间序列的指标名称为api_http_request_total并且标签为method="POST", handler="/message"可以表示为如下形式：

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

指标类型

之前我们讲过Prometheus对于所有的数据样本，使用了metrics name和labels唯一标示一条时间序列。而这些监控的样本数据，在不同的场景下又具有不同的意义。比如http_request_total采集到的监控样本数据反应的是当前系统的所有Http总量，因此当我们观察数据变化时，会发现对于一条http_request_total所反应的时序数据是一条持续增长的样本。而又比如container_cpu_usage一条时序数据则反映出的则是一条变化的曲线。

因为为了更好的定义这些不同场景下监控样本数据所代表的含义，Prometheus提供了四种指标类型(Metrics Type)。用以帮助开发人员更好的定义和使用这些监控指标。

这四种监控指标类型分别为：Counter, Gauge, Histogram, Summary。

Counter：只增不减的计数器

计数器可以用于记录只会增加不会减少的指标类型,比如记录应用请求的总量(http_requests_total), cpu使用时间(process_cpu_seconds_total)等。

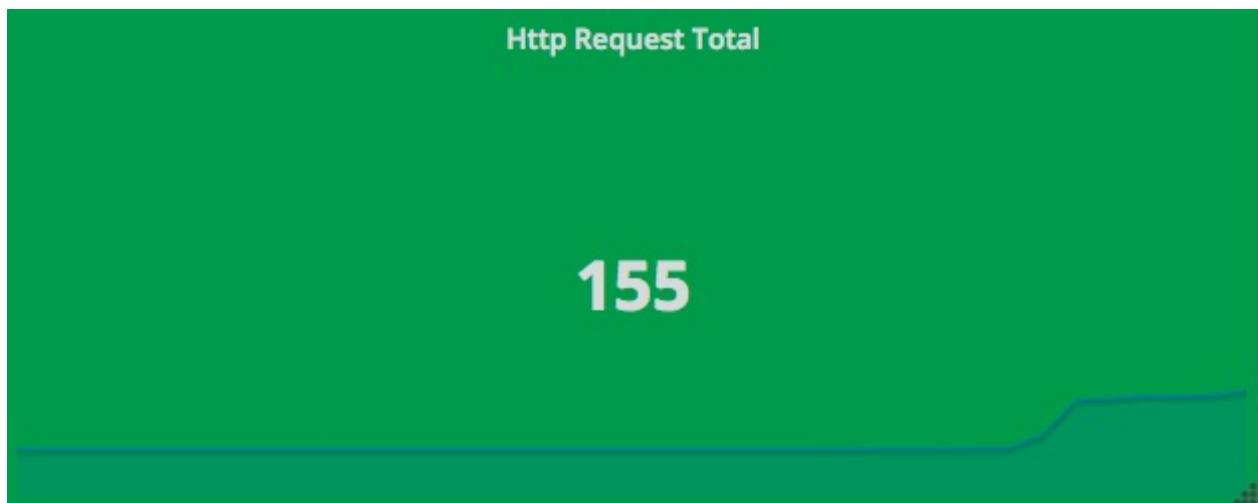
对于Counter类型的指标，只包含一个inc()方法，用于计数器+1

一般而言，Counter类型的metrics指标在命名中我们使用_total结束。

通过指标io_namespace_http_requests_total我们可以：

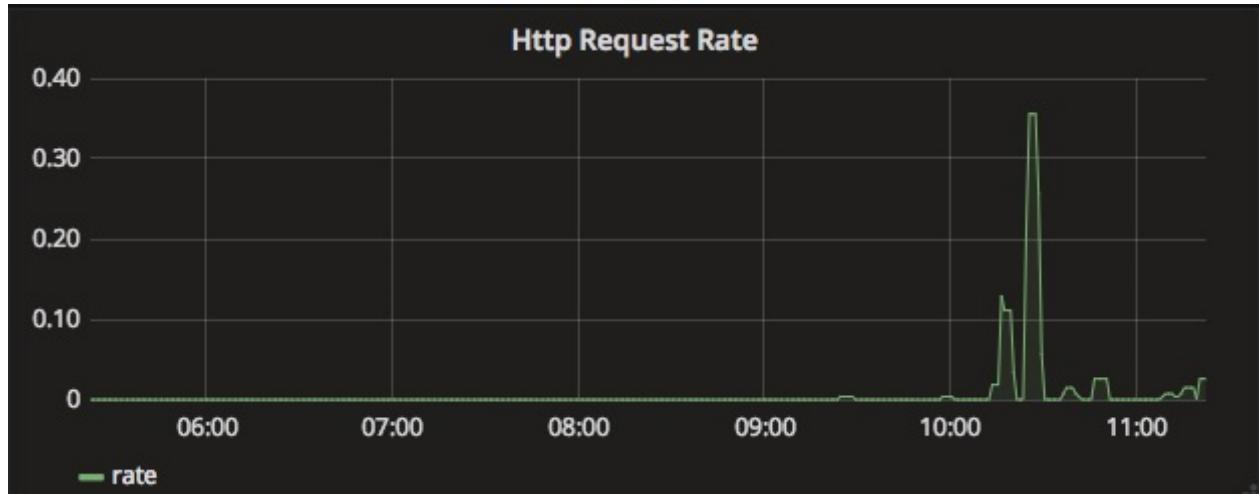
- 查询应用的请求总量

```
# PromQL
sum(io_namespace_http_requests_total)
```



- 查询每秒Http请求量

```
# PromQL
sum(rate(io_wise2c_gateway_requests_total[5m]))
```



- 查询当前应用请求量Top N的URI

```
# PromQL
topk(10, sum(io_namespace_http_requests_total) by (path))
```

Gauge: 可增可减的仪表盘

对于这类可增可减的指标，可以用于反应用的当前状态,例如在监控主机时，主机当前空闲的内容大小(node_memory_MemFree)，可用内存大小(node_memory_MemAvailable)。或者容器当前的cpu使用率,内存使用率。

对于Gauge指标的对象则包含两个主要的方法inc()以及dec(),用户添加或者减少计数。在这里我们使用Gauge记录当前正在处理的Http请求数量。

通过指标io_namespace_http_inprogress_requests我们可以直接查询应用当前正在处理中的Http请求数量:

```
# PromQL
io_namespace_http_inprogress_requests{}
```

Histogram: 自带分区统计的分布统计图

主要用于在指定分布范围内(Buckets)记录大小(如http request bytes)或者事件发生的次数。

以请求响应时间requests_latency_seconds为例，假如我们需要记录http请求响应时间符合在分布范围{.005, .01, .025, .05, .075, .1, .25, .5, .75, 1, 2.5, 5, 7.5, 10}中的次数时。

使用Histogram构造器可以创建Histogram监控指标。默认的buckets范围为{.005, .01, .025, .05, .075, .1, .25, .5, .75, 1, 2.5, 5, 7.5, 10}。如何需要覆盖默认的buckets，可以使用.buckets(double... buckets)覆盖。

Histogram会自动创建3个指标，分别为：

- 事件发生总次数： basename_count

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

- 所有事件产生值的大小的总和: basename_sum

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

- 事件产生的值分布在bucket中的次数： basename_bucket{le="上包含"}

```

# 在总共2次请求当中。http请求响应时间 <=0.005 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.005",} 0.0
# 在总共2次请求当中。http请求响应时间 <=0.01 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.01",} 0.0
# 在总共2次请求当中。http请求响应时间 <=0.025 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.025",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.05",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.075",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.1",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.25",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.5",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="0.75",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="1.0",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="2.5",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="5.0",} 0.0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="7.5",} 2.0
# 在总共2次请求当中。http请求响应时间 <=10 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="10.0",} 2.0
# 在总共2次请求当中。http请求响应时间 10 秒 的请求次数为0
io_namespace_http_requests_latency_seconds_histogram_bucket{path="/",method="GET",
code="200",le="+Inf",} 2.0

```

Summary：客户端定义的分布统计图

Summary和Histogram非常类型相似，都可以统计事件发生的次数或者发小，以及其分布情况。

Summary和Histogram都提供了对于事件的计数_count以及值的汇总_sum。因此使用_count,和_sum时间序列可以计算出相同的内容，例如http每秒的平均响应时间：
 $\text{rate}(\text{basename_sum}[5m]) / \text{rate}(\text{basename_count}[5m])$ 。

同时Summary和Histogram都可以计算和统计样本的分布情况，比如中位数，9分位数等等。其中 $0.0 \leq \text{分位数Quantiles} \leq 1.0$ 。

不同在于Histogram可以通过histogram_quantile函数在服务器端计算分位数。而Summary的分位数则是直接在客户端进行定义。因此对于分位数的计算。Summary在通过PromQL进行查询时有更好的性能表现，而Histogram则会消耗更多的资源。相对的对于客户端而言Histogram消耗的资源更少。

Summary指标，会对应多个时间序列：

- 事件发生总的次数

```
# 含义：当前http请求发生总次数为12次
io_namespace_http_requests_latency_seconds_summary_count{path="/",method="GET",code="200",} 12.0
```

- 事件产生的值的总和

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

- 事件产生的值的分布情况

```
# 含义：这12次http请求数量的中位数是3.052404983s
io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.5",} 3.052404983
# 含义：这12次http请求数量的9分位数是8.003261666s
io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.9",} 8.003261666
```

初识PromQL

Prometheus通过指标名称Metrics Name以及对应的一组键值对Labels唯一定义一条时间序列。指标名称反应了监控样本的基本标识，而Label则在这个基本特征上为采集到的数据提供了多种特征维度。用户可以基于这些特征维度过滤，聚合，统计从而产生新的计算后的一条时间序列。

查询基础

当Prometheus通过Exporter采集到响应的监控指标样本数据后，我们就可以通过PromQL对监控样本数据进行查询。

基本查询

当我们直接使用监控指标的名称为查询时，可以查询该指标下的所有时间序列。如：

```
http_requests_total
```

等同于

```
http_requests_total{}
```

该表达式会返回所有指标名称为http_requests_total的时间序列。

```
http_requests_total{code="200", handler="alerts", instance="localhost:9090", job="prometheus", method="get"} -> 1
http_requests_total{code="200", handler="graph", instance="localhost:9090", job="prometheus", method="get"} -> 3
http_requests_total{code="200", handler="graph", instance="other:9090", job="prometheus", method="get"} -> 3
```

SQL:

```
SELECT * FROM http_requests_total;
```

精确查询

在查询数据时，我们还可以通过标签选择器对时间序列进行精确匹配查询。

- 使用“=”表示选择标签完全匹配的时间序列，
- “!=”表示排除这些匹配的时间序列。

如下所示，我们只查询所有http_requests_total时间序列中，标签instance为localhost:9090的时间序列。这里在标签选择器中我们使用“=”表示精确匹配

```
http_requests_total{instance="localhost:9090"}
```

相反的我们可以使用“!=”表示排除：

```
http_requests_total{instance!="localhost:9090"}
```

返回结果：

```
http_requests_total{code="200", handler="graph", instance="other:9090", job="prometheus", method="get"} -> 3
```

SQL:

```
SELECT * FROM http_requests_total WHERE instance="localhost:9090"
```

模糊查询

除了精确查询以外，PromQL还可以通过正则表达式的方式，实现模糊查询。

- 当需要使用正则表达式进行模糊查询时，需要使用“=~”。
- 相反的，“!~”表示排除所有的匹配的时间序列。

例如

```
http_requests_total{environment=~"staging|testing|development", method!="GET"}
```

SQL:

```
SELECT * FROM http_requests_total WHERE environment LIKE '%testing%'
```

使用内置函数

一般来说，如果描述样本特征的标签(label)在不是唯一的情况下，通过PromQL查询数据，会返回多条满足这些特征维度的时间序列。而PromQL提供的聚合操作可以用来对这些多条时间序列进行处理，形成一条新的时间序列。

```
# 查询系统所有http请求的总量
sum(http_request_total)

# 按照mode计算主机cpu的平均使用时间
avg(node_cpu) by (mode)

# 按照主机查询各个主机的cpu使用率
sum(sum(irate(node_cpu{mode != 'idle'}[5m])) / sum(irate(node_cpu[5m]))) by (instance)
```

PromQL的返回值

通过上面的几个简单例子我们可以看出，通过指标名称(metric name)以及指标的维度labels，通过Prometheus提供的PromQL查询语言，我们可以根据样本特征对时序数据进行过滤。同时多条时间序列之间的数据还可以进行聚合以及数学操作，从而形成一条新的时间序列。

在PromQL中如果表达式返回的是一组时序数据，并且每条时间序列只包含给定时间戳（瞬时）的单个样本数据 这些返回数据的类型在Prometheus中我们称为瞬时向量(Instant vector)。

瞬时向量(Instant vector)

例如，使用如下表达式，会可以过滤并查询到一组时间序列以及给定时间戳（瞬时，一般为最后一次采集数据的时戳）的单个样本数据。

```
http_request_total{code="200"}
```

会返回一组时间序列

```
http_requests_total{code="200", handler="alerts", instance="localhost:9090", job="prometheus", method="get"}=(20889@1518096812.326)
http_requests_total{code="200", handler="graph", instance="localhost:9090", job="prometheus", method="get"}=(21287@1518096812.326)
```

并且这组时间序列的样本数据共享相同的时间蹉。

这一类表达式，我们称为**瞬时向量选择器**，瞬时向量选择器返回的数据类型为**瞬时向量**。

瞬时向量选择器，至少包含一个指标名称(例如http_request_total)，或者一个不会匹配到空字符串的标签过滤器(例如{code="200"})。

因此以下两种方式，均为合法的表达式：

```
http_request_total # 合法  
http_request_total{} # 合法  
{method="get"} # 合法
```

而如下表达式，则不合法：

```
{job=~".*"} # 不合法
```

同时，除了使用metrics{label=value}的形式，使用metrics指定监控指标以外，我们还可以使用内置的 __name__ 指定监控指标名称：

```
{__name__=~"http_request_total"} # 合法  
{__name__=~"node_disk_bytes_read|node_disk_bytes_written"} # 合法
```

区间向量(Range vector)

除了瞬时向量以外，PromQL表达式还可以查询区间向量，区间向量和瞬时向量非常相似，区别在于区间向量返回是从当前时刻开始选择的一个范围的样本数据。返回区间向量类型的表达式，我们称为**区间向量选择器**。

例如：

```
http_request_total{code="200"}[5m]
```

该表达式，表示查询时间序列名称为http_request_total并且满足code="200"的时序数据中，最近5分钟内的样本数据。如下：

```
http_requests_total{code="200", handler="alerts", instance="localhost:9090", job="prometheus", method="get"}=[  
    1@1518096812.326  
    1@1518096817.326  
    1@1518096822.326  
    1@1518096827.326  
    1@1518096832.326  
    1@1518096837.325  
]  
http_requests_total{code="200", handler="graph", instance="localhost:9090", job="prometheus", method="get"}=[  
    4 @1518096812.326  
    4@1518096817.326  
    4@1518096822.326  
    4@1518096827.326  
    4@1518096832.326  
    4@1518096837.325  
]
```

除了使用m表示分钟以外，PromQL还可以使用其他的时间单位：

- s - 秒
- m - 分钟
- h - 小时
- d - 天
- w - 周
- y - 年

标量(Scalar): 一个浮点型的数字值

标量只有一个数字，没有时序

例如：

```
10
```

需要注意的是，当使用表达式count(http_requests_total)，返回的数据类型，依然
是瞬时向量。

字符串(String): 一个简单的字符串值

直接使用字符串，作为PromQL表达式，则会直接返回字符串。

```
"this is a string"  
'these are unescaped: \n \\ \t'  
'these are not unescaped: \n ' \" \t`
```

时间位移

在瞬时选择器，或者区间选择器中，都是以当前时间为基准

```
http_request_total{code="200"} # 瞬时选择器，选择当前最新的数据  
http_request_total{code="200"}[5m] # 区间选择器，选择以当前时间为基准，5分钟内的数据
```

那如果我们想查询，5分钟前的瞬时样本数据，或昨天一天的区间内的样本数据呢？这个时候我们就可以使用位移操作，位移操作的关键字为**offset**。

因此我们可以使用时间位移操作：

```
http_request_total{code="200"} offset 5m  
http_request_total{code="200"}[1d] offset 1d
```

接下来

接下来我们会详细探索Prometheus提供的这一强大工具PromQL。以及它给我们带来的强大的数据统计功能。

PromQL操作符

PromQL支持基本的逻辑和数学运算。当在两个瞬时向量之间进行计算时，运算的匹配规则可以进行修改。

二进制运算

数学运算符

算数运算支持用于：标量和标量、瞬时向量和标量、瞬时向量和瞬时之间的运算。

目前PromQL支持一下算数运算符：

- `+` (加法)
- `-` (减法)
- `*` (乘法)
- `/` (除法)
- `%` (求余)
- `^` (幂运算)

标量与标量

标量和标量之间进行数学运算，产生一个新的标量。

```
2 * 2 # 产生标量4
```

标量与瞬时向量

标量和瞬时向量之间进行数学运算，数学运算符将被作用域瞬时向量中的每一个样本值。并且产生一个新的瞬时向量。

```
http_requests_total{}
```

```
http_requests_total{code="200", handler="alerts", instance="localhost:9090", job="prometheus", method="get"} => 1@1518145642.308
http_requests_total{code="200", handler="federate", instance="localhost:9090", job="prometheus", method="get"} => 1@1518145642.308
```

例如，如果表达式`http_requests_total{}`查询出一组时间序列的瞬时样本数据，那`_5`操作会将每一条时间序列数据中的瞬时样本数据`_5`，并且产生一组新的时间序列。

```
http_requests_total{} * 5
```

返回结果：

```
http_requests_total{code="200", handler="alerts", instance="localhost:9090", job="prometheus", method="get"} => 5@1518145642.308
http_requests_total{code="200", handler="federate", instance="localhost:9090", job="prometheus", method="get"} => 5@1518145642.308
```

瞬时向量与瞬时向量

两个瞬时向量之间进行数学运算，数学运算将将会作用于左边数据中的每一个样本数据，与该样本在右边数据中**匹配到**的样本数据之间。

例如，

```
node_disk_bytes_written + node_disk_bytes_written
```

表达式`node_disk_bytes_written`返回当前主机中各个磁盘的写入数据总量的瞬时向量。

```
node_disk_bytes_written{device="sda", instance="localhost:9100", job="node_exporter"}=>1634967552@1518146427.807
node_disk_bytes_written{device="sdb", instance="localhost:9100", job="node_exporter"}=>0@1518146427.807
```

表达式`node_disk_bytes_read{}`会返回，当前主机中各个磁盘的读取数据总量的瞬时向量。

```
node_disk_bytes_read{device="sda", instance="localhost:9100", job="node_exporter"}=>864551424@1518146427.807
node_disk_bytes_read{device="sdb", instance="localhost:9100", job="node_exporter"}=>1744384@1518146427.807
```

匹配规则，会比较两个表达式返回的瞬时向量中的所有标签，标签的键值对完全相等，则表示匹配成功。并将运算符作用域两个匹配的样本数据中。并且返回一组新的瞬时向量，同时结果中会丢弃指标名称。对于没有匹配的样本数据，则不会出现在运算结果中。

```
{device="sda",instance="localhost:9100",job="node_exporter"}=>1634967552@151814642  
7.807 + 864551424@1518146427.807  
{device="sdb",instance="localhost:9100",job="node_exporter"}=>0@1518146427.807 + 1  
744384@1518146427.807
```

即结果为

```
{device="sda",instance="localhost:9100",job="node_exporter"}=>2499568128@151814642  
7.807  
{device="sdb",instance="localhost:9100",job="node_exporter"}=>1744384@1518146427.8  
07
```

比较运算符

比较运算符运算支持：标量和标量、瞬时向量和标量、瞬时向量和瞬时向量之间的运算。

目前，Prometheus支持一下，比较运算符：

- `==` (相等)
- `!=` (不相等)
- `>` (大于)
- `<` (小于)
- `>=` (大于等于)
- `<=` (小于等于)

瞬时向量与标量

瞬时向量和标量之间进行比较运算时，PromQL的默认行为会依次将瞬时向量中的所有样本与标量之间进行比较运算。如果比较结果为true则保留样本，如果比较结果为false则丢弃该样本，从而产生一条新的瞬时时间序列。

例如，当需要找到当前系统请求量大于100次的处理模块，可以使用表达式：

```
http_requests_total > 100
```

该表达式会过滤监控指标`http_requests_total`的所有时间序列，返回瞬时样本值满足条件比较条件(`> 1000`)的所有时间序列。

```
http_requests_total{code="200", handler="prometheus", instance="localhost:9090", job="prometheus", method="get"} 36733
http_requests_total{code="200", handler="prometheus", instance="localhost:9100", job="node_exporter", method="get"} 37131
http_requests_total{code="200", handler="query", instance="localhost:9090", job="prometheus", method="get"} 126
```

使用bool改变比较运算的默认行为

默认情况下比较运算符的默认行为是对时序数据进行过滤。而在其它的情况下我们可能需要的是真正的布尔结果。例如，只需要知道当前模块的HTTP请求量是否 ≥ 1000 ，如果大于等于1000则返回1 (true) 否则返回0 (false)。这时我们可以使用bool改变比较运算的默认行为。

例如：

```
http_requests_total > bool 100
```

使用bool修改比较运算的默认行为之后，比较运算不会对时间序列进行过滤，而是直接依次瞬时向量中的各个样本数据与标量的比较结果0或者1。从而形成一条新的时间序列。

```
http_requests_total{code="200", handler="query", instance="localhost:9090", job="prometheus", method="get"} 1
http_requests_total{code="200", handler="query_range", instance="localhost:9090", job="prometheus", method="get"} 0
```

标量和标量

标量和标量之间进行比较运算，根据比较的结果产生一个新的标量0 (false) 或者1 (true) 用于返回比较的结果。需要注意的是，标量和标量之间进行比较时，必须使用bool进行修饰，例如：

```
2 == bool 2 # 结果为1
```

瞬时向量和瞬时向量

瞬时向量和瞬时向量之间，进行比较运算时，跟根据默认的匹配规则，依次比较匹配到的样本数据。默认情况下，如果匹配到的数据比较结果为true则保留，反之则丢弃。从而形成一条新的时间序列。同样，我们可以通过bool修饰符来改变比较运算的默认行为。

例如，使用表达式获取当前正常任务状态：

```
up == 1
```

或者我们只想知道当前任务的状态是否为正常：

```
up == bool 1
```

逻辑/集合运算符

逻辑运算符只支持，瞬时向量和瞬时向量之间。

目前，Prometheus支持一下，比较逻辑运算符有：

- `and` (并且)
- `or` (或者)
- `unless` (排除)

vector1 and vector2 会产生一个由vector1的元素组成的新的向量。该向量包含vector1中完全匹配vector2中的元素组成。

vector1 or vector2 会产生一个新的向量，该向量包含vector1中所有的样本数据，以及vector2中没有与vector1匹配到的样本数据。

vector1 unless vector2 会产生一个新的向量，新向量中的元素由vector1中没有与vector2匹配的元素组成。

向量匹配模式

前面部分已经讲过，默认情况下向量与向量之间进行运算操作时会基于默认的匹配规则：依次找到与左边向量元素匹配（标签完全一致）的右边向量元素进行运算，如果没找到匹配元素，则直接丢弃。

接下来将介绍在PromQL中有两种典型的匹配模式：一对一（one-to-one），多对一（many-to-one）或一对多（one-to-many）。

一对一匹配

一对一匹配模式会从操作符两边表达式获取的瞬时向量依次比较并找到唯一匹配(标签完全一致)的样本值。默认情况下，使用表达式：

```
vector1 <operator> vector2
```

在操作符两边表达式标签不一致的情况下，可以使用on(label list)或者ignoring(label list)来修改便签的匹配行为。使用ignoring可以在匹配时忽略某些便签。而on则用于将匹配行为限定在某些便签之内。

```
<vector expr> <bin-op> ignoring(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) <vector expr>
```

例如当存在样本：

```
method_code:http_errors:rate5m{method="get", code="500"} 24
method_code:http_errors:rate5m{method="get", code="404"} 30
method_code:http_errors:rate5m{method="put", code="501"} 3
method_code:http_errors:rate5m{method="post", code="500"} 6
method_code:http_errors:rate5m{method="post", code="404"} 21

method:http_requests:rate5m{method="get"} 600
method:http_requests:rate5m{method="del"} 34
method:http_requests:rate5m{method="post"} 120
```

使用PromQL表达式：

```
method_code:http_errors:rate5m{code="500"} / ignoring(code) method:http_requests:rate5m
```

该表达式会返回在过去5分钟内，HTTP请求状态码为500的在所有请求中的比例。如果没有使用ignoring(code)，操作符两边表达式返回的瞬时向量中将找不到任何一个标签完全相同的匹配项。

因此结果如下：

```
{method="get"} 0.04          // 24 / 600
{method="post"} 0.05          // 6 / 120
```

同时由于method为put和del的样本找不到匹配项，因此不会出现在结果当中。

多对一和一对多

多对一和一对多两种匹配模式指的是“一”侧的每一个向量元素可以与“多”侧的多个元素匹配的情况。在这种情况下，必须使用group修饰符：group_left或者group_right来确定哪一个向量具有更高的基数（充当“多”的角色）。

```
<vector expr> <bin-op> ignoring(<label list>) group_left(<label list>) <vector expr>
<vector expr> <bin-op> ignoring(<label list>) group_right(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) group_left(<label list>) <vector expr>
<vector expr> <bin-op> on(<label list>) group_right(<label list>) <vector expr>
```

多对一和一对多两种模式一定是出现在操作符两侧表达式返回的向量标签不一致的情况下。因此需要使用ignoring和on修饰符来排除或者限定匹配的标签列表。

例如，使用表达式：

```
method_code:http_errors:rate5m / ignoring(code) group_left method:http_requests:rate5m
```

该表达式中，左向量 `method_code:http_errors:rate5m` 包含两个标签`method`和`code`。而右向量 `method:http_requests:rate5m` 中只包含一个标签`method`，因此匹配时需要使用`ignoring`限定匹配的标签为`code`。在限定匹配标签后，右向量中的元素可能匹配到多个左向量中的元素。因此该表达式的匹配模式为多对一，需要使用group修饰符`group_left`指定左向量具有更好的基数。

最终的运算结果如下：

```
{method="get", code="500"} 0.04          // 24 / 600
{method="get", code="404"} 0.05          // 30 / 600
{method="post", code="500"} 0.05          // 6 / 120
{method="post", code="404"} 0.175         // 21 / 120
```

提醒：group修饰符只能在比较和数学运算符中使用。在逻辑运算and,unless和or才注意操作中默认与右向量中的所有元素进行匹配。

聚合操作

Prometheus还提供了下列内置的聚合操作符，这些操作符作用域瞬时向量。可以将瞬时表达式返回的样本数据进行聚合，形成一个新的时间序列。

- `sum` (求和)

- `min` (最小值)
- `max` (最大值)
- `avg` (平均值)
- `stddev` (标准差)
- `stdvar` (标准差异)
- `count` (计数)
- `count_values`
- `bottomk`
- `topk`
- `quantile`

使用聚合操作的语法如下：

```
<aggr-op>([parameter,] <vector expression>) [without|by (<label list>)]
```

其中只有 `count_values` , `quantile` , `topk` , `bottomk` 支持参数(parameter)。

`without`用于从计算结果中移除列举的标签，而保留其它标签。`by`则正好相反，结果向量中只保留列出的标签，其余标签则移除。通过`without`和`by`可以按照样本的问题对数据进行聚合。

例如：

```
sum(http_requests_total) without (instance)
```

等价于

```
sum(http_requests_total) by (code,handler,job,method)
```

如果只需要计算整个应用的HTTP请求总量，可以直接使用表达式：

```
sum(http_requests_total)
```

`count_values`用于时间序列中每一个样本值出现的次数。`count_values`会为每一个唯一的样本值输出一个时间序列，并且每一个时间序列包含一个额外的标签。

例如：

```
count_values("count", http_requests_total)
```

topk和bottomk则用于对样本值进行排序，返回当前样本值前n位，或者后n位的时间序列。

获取HTTP请求数前5位的时序样本数据，可以使用表达式：

```
topk(5, http_requests_total)
```

quantile用于计算当前样本数据值的分布情况quantile(φ , express)其中 $0 \leq \varphi \leq 1$ 。

例如，当 φ 为0.5时，即表示找到当前样本数据中的中位数：

```
quantile(0.5, http_requests_total)
```

操作符优先级

最后对于复杂类型的表达式，我们需要了解运算操作的优先级。

例如，查询主机的CPU使用率，我们可以使用表达式：

```
100 * (1 - avg (irate(node_cpu{mode='idle'}[5m])) by(job) )
```

其中irate是PromQL中的内置函数，用于计算区间向量中时间序列每秒的即时增长率。关于内置函数的部分，会在下一节详细介绍。

在PromQL操作符中优先级由高到低依次为：

1. `^`
2. `*, /, %`
3. `+, -`
4. `==, !=, <=, <, >=, >`
5. `and, unless`
6. `or`

内置函数

在上一小节中，我们已经看到了类似于`rate()`这样的函数，可以帮助我们计算监控指标的实时增长率。除了`rate`以外，Prometheus还提供了其它大量的内置函数，可以对时序数据进行更多的处理。

abs()

绝对值

```
abs(v instant-vector)
```

absent()

```
absent(v instant-vector)
```

判断当前序列是否存在。

ceil()

```
ceil(v instant-vector)
```

将向量中的所有元素样本值向上取整。

changes()

```
changes(v range-vector)
```

`changes`将返回每一个区间向量中样本变化的次数，作为一个新的瞬时向量。

clamp_max()

```
clamp_max(v instant-vector, max scalar) 最大值上限。
```

限制瞬时向量中样本值的最大范围，使其具有最大值的上限。即如果样本值大于最大值，则使用最大值替换该样本值。

clamp_min()

```
clamp_min(v instant-vector, min scalar) 最小值上限,
```

限制瞬时向量中样本值的最小范围。即如果样本值中小于最小值，则使用定义的最小值替换该样本值。

day_of_month()

```
day_of_month(v=vector(time()) instant-vector) 返回给定时间戳，在当前月中的日。返回值在1到31之间。
```

day_of_week()

```
day_of_week(v=vector(time()) instant-vector) 返回给定时间戳，在一周中的第几天。返回值范围为0到6，0表示星期天。
```

days_in_month()

```
days_in_month(v=vector(time()) instant-vector) 给定时间戳所在月份的天数。返回值范围在28到31之间。
```

delta()

```
delta(v range-vector)
```

计算区间向量v中每一个时间序列元素的第一个值和最后一个值之间的差值，并且以该增量作为瞬时向量的样本值。

例如，使用表达式，可以查询当前CPU与两个小时之间的差异。

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

delta只适用于仪表盘。

deriv()

```
deriv(v range-vector)
```

使用简单线性回归计算区间向量v中的时间序列每秒的导数。

deriv只适用于仪表盘。

exp()

```
exp(v instant-vector)
```

计算瞬时向量v中所有元素的指数。

特殊情况：

- $\text{Exp}(+\text{Inf}) = +\text{Inf}$
- $\text{Exp}(\text{NaN}) = \text{NaN}$

floor()

```
floor(v instant-vector)
```

将v中所有元素的样本值向下取整。

histogram_quantile()

```
histogram_quantile(φ float, b instant-vector)
```

holt_winters()

```
holt_winters(v range-vector, sf scalar, tf scalar)
```

hour()

```
hour(v=vector(time()) instant-vector)
```

iDelta()

```
idelta(v range-vector)
```

increase()

```
increase()
```

irate()

```
irate()
```

label_join()

```
label_join(v instant-vector, dst_label string, separator string, src_label_1 string,  
src_label_2 string, ...)
```

label_replace()

```
label_replace(v instant-vector, dst_label string, replacement string, src_label  
string, regex string)
```

ln()

```
ln(v instant-vector)
```

log2()

```
log2()
```

log10()

```
log10()
```

minute()

```
minute()
```

month()

```
month()
```

predict_linear()

```
predict_linear()
```

rate()

```
rate()
```

resets()

```
resets()
```

round()

```
round(v instant-vector, to_nearest=1 scalar)
```

scalar()

```
scalar(v instant-vector)
```

sort()

```
sort(v instant-vector)
```

sort_desc()

```
sort_desc(v instant-vector)
```

sqrt()

```
sqrt(v instant-vector)
```

time()

```
time()
```

timestamp()

```
timestamp(v instant-vector)
```

vector()

```
vector()
```

year()

```
year(v=vector(time()) instant-vector)
```

_over_time()

- `avg_over_time(range-vector)`
- `min_over_time(range-vector)`
- `max_over_time(range-vector)`
- `sum_over_time(range-vector)`
- `count_over_time(range-vector)`
- `quantile_over_time(scalar, range-vector)`
- `stddev_over_time(range-vector)`
- `stdvar_over_time(range-vector)`

在HTTP API中使用PromQL

Prometheus当前稳定的HTTP API可以通过/api/v1访问。

API响应格式

Prometheus API使用了JSON格式的响应内容。当API调用成功后将会返回2xx的HTTP状态码。

反之，当API调用失败时可能返回一下几种不同的HTTP状态码：

- 404 Bad Request：当参数错误或者缺失时。
- 422 Unprocessable Entity 当表达式无法执行时。
- 503 Service Unavailable 当请求超时或者被中断时。

所有的API请求均使用以下的JSON格式：

```
{  
  "status": "success" | "error",  
  "data": <data>,  
  
  // Only set if status is "error". The data field may still hold  
  // additional data.  
  "errorType": "<string>",  
  "error": "<string>"  
}
```

在HTTP API中使用PromQL

通过HTTP API我们可以分别通过/api/v1/query和/api/v1/query_range查询PromQL表达式当前或者一定时间范围内的计算结果。

瞬时数据查询

通过使用QUERY API我们可以查询PromQL在特定时间点下的计算结果。

```
GET /api/v1/query
```

URL请求参数：

- `query=`: PromQL表达式。
- `time=`: 用于指定用于计算PromQL的时间戳。可选参数， 默认情况下使用当前系统时间。
- `timeout=`: 超时设置。可选参数， 默认情况下使用`-query,timeout`的全局设置。

例如使用以下表达式查询表达式`up`在时间点`2015-07-01T20:10:51.781Z`的计算结果：

```
$ curl 'http://localhost:9090/api/v1/query?query=up&time=2015-07-01T20:10:51.781Z'
{
  "status" : "success",
  "data" : {
    "resultType" : "vector",
    "result" : [
      {
        "metric" : {
          "__name__" : "up",
          "job" : "prometheus",
          "instance" : "localhost:9090"
        },
        "value": [ 1435781451.781, "1" ]
      },
      {
        "metric" : {
          "__name__" : "up",
          "job" : "node",
          "instance" : "localhost:9100"
        },
        "value" : [ 1435781451.781, "0" ]
      }
    ]
  }
}
```

响应数据类型

当API调用成功后， Prometheus会返回JSON格式的响应内容， 格式如上小节所示。并且在`data`节点中返回查询结果。`data`节点格式如下：

```
{
  "resultType": "matrix" | "vector" | "scalar" | "string",
  "result": <value>
}
```

在前面我们已经了解了，PromQL表达式可能返回多种数据类型，在响应内容中使用resultType表示当前返回的数据类型，包括：

- 瞬时向量：vector

当返回数据类型resultType为vector时，result响应格式如下：

```
[  
  {  
    "metric": { "<label_name>": "<label_value>", ... },  
    "value": [ <unix_time>, "<sample_value>" ]  
  },  
  ...  
]
```

其中metrics表示当前时间序列的特征维度，value只包含一个唯一的样本。

- 区间向量：matrix

当返回数据类型resultType为matrix时，result响应格式如下：

```
[  
  {  
    "metric": { "<label_name>": "<label_value>", ... },  
    "values": [ [ <unix_time>, "<sample_value>" ], ... ]  
  },  
  ...  
]
```

其中metrics表示当前时间序列的特征维度，values包含当前事件序列的一组样本。

- 标量：scalar

当返回数据类型resultType为scalar时，result响应格式如下：

```
[ <unix_time>, "<scalar_value>" ]
```

由于标量不存在时间序列一说，因此result表示为当前系统时间一个标量的值。

- 字符串：string

当返回数据类型resultType为string时，result响应格式如下：

```
[ <unix_time>, "<string_value>" ]
```

字符串类型的响应内容格式和标量相同。

区间数据查询

使用QUERY_RANGE API我们则可以直接查询PromQL表达式在一段时间返回内的计算结果。

```
GET /api/v1/query_range
```

URL请求参数：

- `query`=: PromQL表达式。
- `start`=: 起始时间。
- `end`=: 结束时间。
- `step`=: 查询步长。
- `timeout`=: 超时设置。可选参数， 默认情况下使用`-query,timeout`的全局设置。

当使用QUERY_RANGE API查询PromQL表达式时，返回结果一定是一个区间向量：

```
{
  "resultType": "matrix",
  "result": <value>
}
```

需要注意的是，在QUERY_RANGE API中PromQL只能使用瞬时向量选择器类型的表达式。

例如使用以下表达式查询表达式`up`在30秒范围内以15秒为间隔计算PromQL表达式的結果。

```
$ curl 'http://localhost:9090/api/v1/query_range?query=up&start=2015-07-01T20:10:30.781Z&end=2015-07-01T20:11:00.781Z&step=15s'
{
  "status" : "success",
  "data" : [
    {
      "resultType" : "matrix",
      "result" : [
        {
          "metric" : {
            "__name__" : "up",
            "job" : "prometheus",
            "instance" : "localhost:9090"
          },
          "values" : [
            [ 1435781430.781, "1" ],
            [ 1435781445.781, "1" ],
            [ 1435781460.781, "1" ]
          ]
        },
        {
          "metric" : {
            "__name__" : "up",
            "job" : "node",
            "instance" : "localhost:9091"
          },
          "values" : [
            [ 1435781430.781, "0" ],
            [ 1435781445.781, "0" ],
            [ 1435781460.781, "1" ]
          ]
        }
      ]
    }
  ]
}
```

新的存储层

最佳实践

Custom Docker Runtime Metrics Exporter With Spring Boot

前言

一般来说在Prometheus社区我们能够找到大量已经实现的Exporter，例如监控主机时我们会使用Node Exporter，监控Mysql数据库时我们会用到Mysql Exporter。监控容器的运行指标时，则使用cAdvisor。

而在一些情况下我们可能需要实现自己的Exporter。例如我们要获取一些业务相关的监控指标，又或者是现成的Exporter无法直接提供我们需要的监控指标时，我们需要创建自定义的Exporter或者对现有Exporter进行补充。

本文中我们将介绍如何在Spring Boot基础上使用Prometheus Java Client实现自定义Exporter用于采集Docker Runtime Metrics数据。

指标类型

Prometheus中定义了四种指标类型：Counter、Gauge、Histogram以及Summary。首先我们先了解一下这四种基本类型的定义以及使用场景

- Counter计数器，只增不减
- Gauge用于反映当前状态，可增可减
- Summary用于统计监控数据的大小或者事件发生的次数
- Histogram主要用于统计监控数据的大小或者事件发生的次数的分布情况

Docker Runtime Metrics API

在使用Docker的过程中如果我们需要查看某些容器的运行时指标数据时，我们会使用docker stats命令来查看该容器的实时状态。

```
$ docker stats redis1 redis2
CONTAINER          CPU %           MEM USAGE / LIMIT      MEM %
NET I/O            BLOCK I/O
redis1              0.07%           796 KB / 64 MB       1.21%
788 B / 648 B      3.568 MB / 512 KB
redis2              0.07%           2.746 MB / 64 MB     4.29%
1.266 KB / 648 B   12.4 MB / 0 B
```

该命令会返回当前容器的CPU使用率，内存用量，内存使用率，以及网络IO等数据。基于这些指标我们可以获取并判断容器的一些基本状态

为什么不采用cAdvisor

为什么需要服务发现

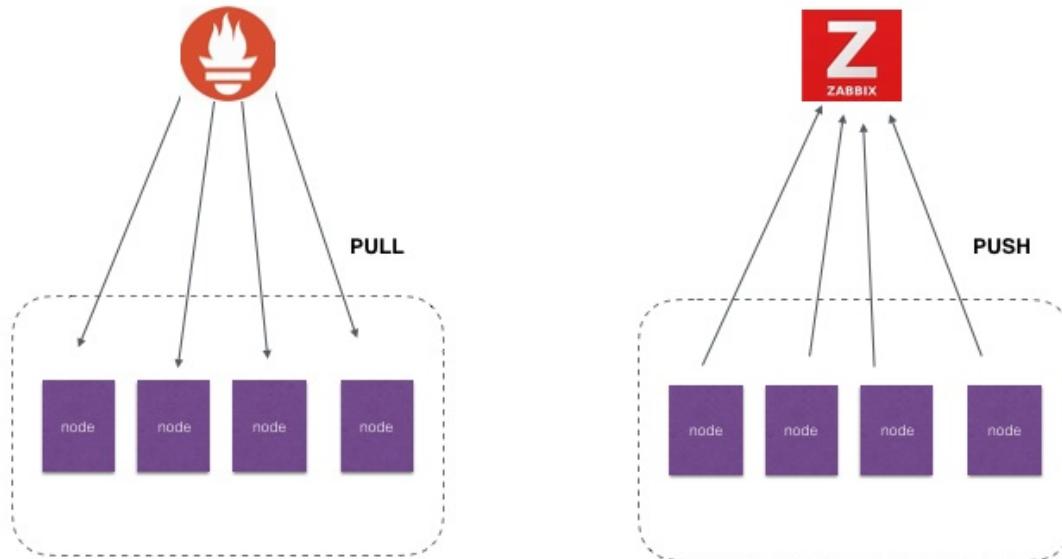
在前面章节中，我们大量使用了static_config来定义我们的监控目标。在本地或者开发环境中，static_config是最直接的定义监控目标的方式。

Targets

cadvisor (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://cadvisor:8080/metrics	UP	Instance="cadvisor:8080"	3.524s ago	
docker (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://docker_exporter:9095/metrics	UP	Instance="docker_exporter:9095"	3.18s ago	
node (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://node_exporter:9100/metrics	UP	Instance="node_exporter:9100"	4.422s ago	
prometheus (1/1 up)				
Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	Instance="localhost:9090"	2.309s ago	

现在我们已经清楚，Prometheus通过Job定义一个采集任务，一个Job可以对应多个Target或者称为Instance。

对于Zabbix以及Nagios这类Push系统而言，通常由采集的Agent来决定和哪一个监控服务进行通讯。而对于Prometheus这类基于Pull的监控平台而言，则由Server侧决定采集的目标有哪些。



当然相比于Push System而言，Pull System：

- 只要Exporter在运行，你可以在任何地方（比如在本地），搭建你的监控系统

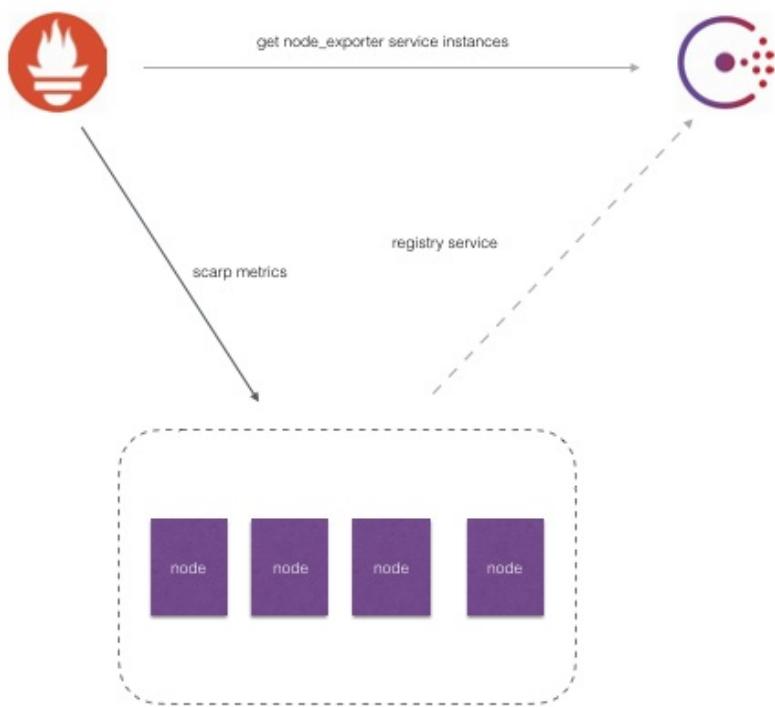
- 你可以更容易的去定位Instance实例的健康状态以及故障定位
- 更利于构建DevOps文化的团队
- 更适合于云原生的部署环境

现在越来越多的企业将自己的基础设施，应用托管到云(公有或者私有)当中。云环境可以更好的根据当前系统容量需求去扩容或者缩容我们的基础设施，或者应用实例。在这种场景下Push System几乎无法有效的适应这种场景，因为所有的监控对象都是耦合了监控系统的信息。

而对于Pull System而言，Server侧和Agent是一种解耦的关系，因此更适合于云下的监控场景。当然对于Prometheus这类Pull System而言，需要解决的一个问题就是如何去发现和管理这些具有动态属性的监控目标。

自动发现监控对象

为了解决以上问题，Prometheus提供了服务发现的机制来自动发现这些自动创建或者销毁的监控目标。



Prometheus提供了多种服务发现机制。包括：file, DNS, Consul, Kubernetes, OpenStack, EC2等等。而无论对于哪一种服务发现机制而言，工作原理都类似：Prometheus通过与服务发现注册中心进行通讯，发现注册到服务发现注册中心的服务实例。再对获取到的这些实例进行筛选(relabel机制)，从而维护一个动态的Target列表。

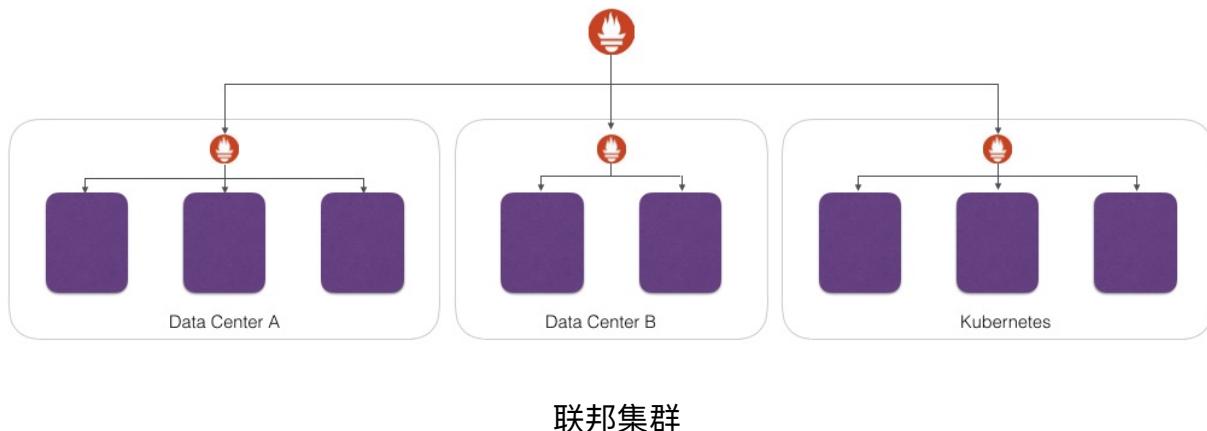
因此相比于Zabbix这一类老牌监控解决方案。通过Prometheus提供的服务发现机制以及Pull的设计原则。监控对象(基础设施，应用，服务等)与监控服务器直接解耦，更适合于在当前云原生的趋势当中。因此Prometheus也被称为下一代监控系统的首选，而其中下一代即代表这云原生。

联邦集群

单个Prometheus Server可以轻松的处理数以百万的时间序列。当然根据规模的不同变化，Prometheus同样可以轻松的进行扩展。本小节将会介绍利用Prometheus的联邦集群特性，对Prometheus进行扩展。

使用联邦集群

Prometheus支持使用联邦集群的方式，对Prometheus进行扩展。对于大部分监控规模而言，我们只需要在每一个数据中心(例如：EC2可用区，Kubernetes集群)安装一个Prometheus Server实例，就可以在各个数据中心处理上千规模的集群。同时将Prometheus Server部署到不同的数据中心可以避免网络配置的复杂性。



联邦集群

如上图所示，在每个数据中心部署单独的Prometheus Server用于采集当前数据中心监控数据。并由一个中心的Prometheus Server负责聚合多个数据中心的监控数据。

每一个Prometheus Server实例包含一个/federate接口，用于获取一组指定的时间序列的监控数据。因此在中心Prometheus Server中只需要配置一个采集任务用于从其他Prometheus Server中获取监控数据。

```

scrape_configs:
  - job_name: 'federate'
    scrape_interval: 15s
    honor_labels: true
    metrics_path: '/federate'
    params:
      'match[]':
        - '{job="prometheus"}'
        - '{__name__=~"job.*"}'
        - '{__name__=~"node.*"}'
    static_configs:
      - targets:
          - '192.168.77.11:9090'
          - '192.168.77.12:9090'

```

通过params可以用于控制Prometheus Server向Target实例请求监控数据的URL当中添加请求参数。例如：

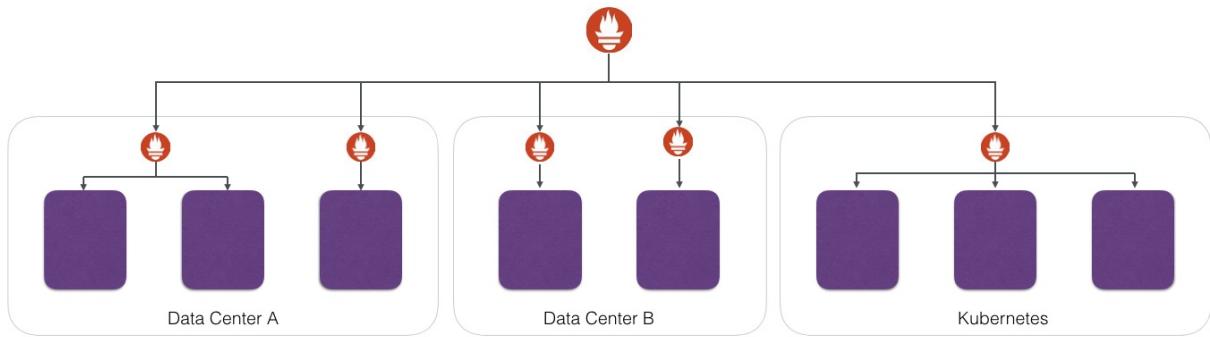
```
"http://192.168.77.11:9090/federate?match[]={job%3D"prometheus"}&match[]={__name__%3D~"job%3A.*"}&match[]={__name__%3D~"node.*"}"
```

通过URL中的match[]参数指定我们可以指定需要获取的时间序列。match[]参数必须是一个瞬时向量选择器，例如up或者{job="api-server"}。配置多个match[]参数，用于获取多组时间序列的监控数据。

honorlabels配置true可以确保当采集到的监控指标冲突时，能够自动忽略冲突的监控数据。如果为false时，prometheus会自动将冲突的标签替换为"exported"的形式。

功能分区

而当你的监控大道单个Prometheus Server无法处理的情况下，我们可以在各个数据中心中部署多个Prometheus Server实例。每一个Prometheus Server实例只负责采集当前数据中心中的一部分任务(Job)，例如可以将应用监控和主机监控分离到不同的Prometheus实例当中。



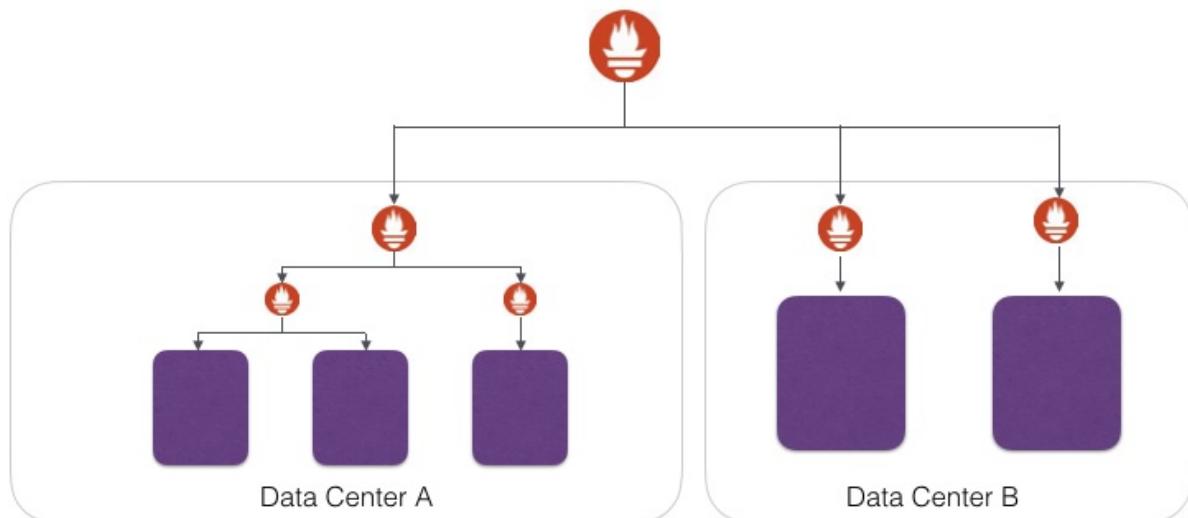
功能分区

假如监控采集任务的规模继续增大，通过功能分区的方式可以进一步细化采集任务。对于中心Prometheus Server只需要从这些实例中聚合数据即可。

功能分区，即通过联邦集群的特性在任务级别对Prometheus采集任务进行划分，以支持规模的扩展。

水平扩展

另外一种极端的情况，假如当单个采集任务的量也变得非常的大，这时候单纯通过功能分区Prometheus Server也无法有效处理。在这种情况下，我们只能考虑在任务(Job)的实例级别进行水平扩展。将采集任务的目标实例划分到不同的Prometheus Server当中。



水平扩展

如上图所示，将统一任务的不同实例的监控数据采集任务划分到不同的Prometheus实例。通过relabel设置，我们可以确保当前Prometheus Server只收集当前采集任务的一部分实例的监控指标。

```
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
```

并且通过当前数据中心的一个中心Prometheus Server将监控数据进行聚合到任务级别。

```
- 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
```

水平扩展，即通过联邦集群的特性在任务的实例级别对Prometheus采集任务进行划分，以支持规模的扩展。

参考资料

- <https://www.robustperception.io/how-much-ram-does-my-prometheus-need-for-ingestion/>
- https://www.youtube.com/watch?list=PL0z-W_CUquUICq-Q0hy53TolAhaED9vm&v=likpVWB5Lvo

参考资料

- <https://github.com/kubernetes/kube-state-metrics>

参考资料

Install & Configuration

- <https://www.digitalocean.com/community/tutorials/how-to-install-prometheus-on-ubuntu-16-04>

Storage

- <https://coreos.com/blog/prometheus-2.0-storage-layer-optimization>

Kubernetes

- <https://docs.bitnami.com/kubernetes/how-to/configure-autoscaling-custom-metrics/>

Others

- <https://news.ycombinator.com/item?id=12455045>
- <https://github.com/digitalocean/vulcan>
- <https://github.com/coreos/prometheus-operator/blob/master/Documentation/high-availability.md>
- <https://github.com/katosys/kato/issues/43>
- <https://www.robustperception.io/tag/tuning/>
- <https://www.robustperception.io/how-much-ram-does-my-prometheus-need-for-ingestion/>
- <https://jaxenter.com/prometheus-product-devops-mindset-130860.html>
- <https://www.slideshare.net/brianbrazil/so-you-want-to-write-an-exporter>

PromSQL

- <https://www.youtube.com/watch?v=lrfTpNZq3Kw>