目　录

全书组织

这里假定你已经对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，在本书中我们不会事无巨细的讲述所有事。

本书转自：https://github.com/yunlzheng/prometheus-book

第1章，是Prometheus基础的综述，通过一个简单案例（使用Prometheus采集主机的监控数据）来了解Prometheus是什么，能做什么，以及它的架构组成。通过阅读本章希望读者能对Promentheus有一个基本的理解和认识。

第2章，读者将会了解到Prometheus的数据模型，以及时间序列模型。同时会学习到如何利用Prometheus的数据查询语言PrmQL(Prometheus Query Language)对监控数据进行查询、聚合、计算等。

第3章，我们的重点将放在监控告警部分，作为监控系统的重要能力之一，我们希望能够及时的了解系统的变化。这一章中读者将学习如何在Prometheus中自定义告警规则，同时了解如何使用AlertManager对告警进行处理。

第4章，介绍Prometheus中一些常用的Exporter的使用场景以及使用方法。之后还会带领读者通过Java和Golang实现自定义的Exporter，同时了解如何在现有应用系统上添加对Prometheus支持，从而实现应用层面的监控对接。

从第1章到第4章的部分都是本书的基础性章节，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，”You can’t fix what you can’t see”。可视化是监控的核心目标之一，这部分将会基于Grafana这一可视化工具实现监控数据可视化，并且了解Grafana作为一个通用的可视化工具是如何与Prometheus进行配合的。

第6章，读者将会了解到如何通过Prometheus的服务发现能力，自动的发现那些需要监控的资源和服务。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更加频繁，通过服务发现自动地去发现监控目标，能够充分简化Prometheus的运维和管理难度。

第7章，在单个节点的情况下Prometheus能够轻松完成对数以百万的监控指标的处理，但是当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？这一章节中重点讨论Prometheus高可用方面的能力。

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。

天降奇兵

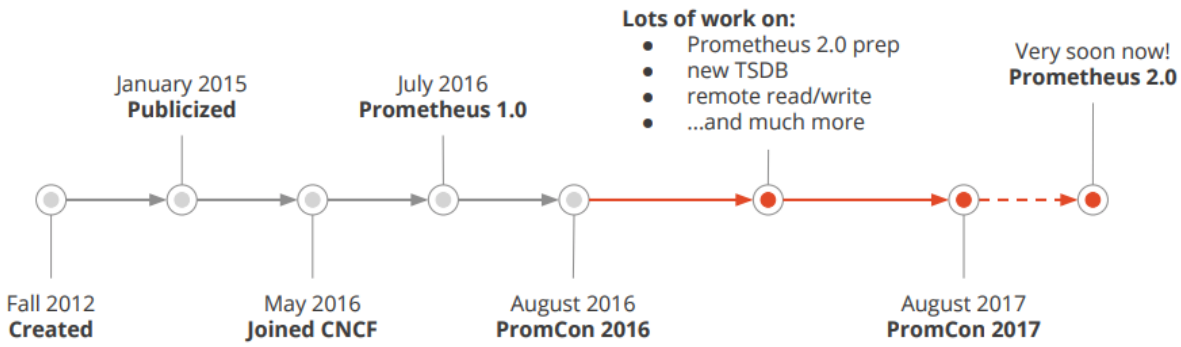
本章作为全书的开篇，我们会带读者了解Prometheus的前世今生，Prometheus是如何从众多的监控平台中脱颖而出成为下一代监控系统的首选。同时通过一个简单的例子带领读者快速了解Prometheus是如何工作的，从而了解Prometheus中的一些概念以及架构模式。

本章内容：

- Prometheus的前世今生
- 使用Prometheus监控主机
- Promthues的核心组件和概念

Prometheus简介

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月继Kubernetes之后成为第二个正式加入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。



Prometheus作为新一代的云原生监控系统，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。

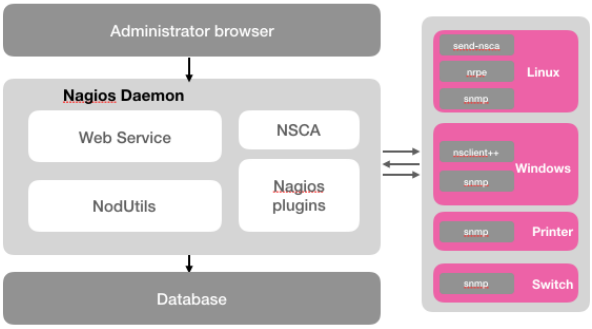
监控的目标

在《SRE: Google运维解密》一书中指出，监控系统需要能够有效的支持白盒监控和黑盒监控。通过白盒能够了解其内部的实际运行状态，通过对监控指标的观察能够预判可能出现的问题，从而对潜在的不确定因素进行优化。而黑盒监控，常见的如HTTP探针，TCP探针等，可以在系统或者服务在发生故障时能够快速通知相关的人员进行处理。通过建立完善的监控体系，从而达到以下目的：

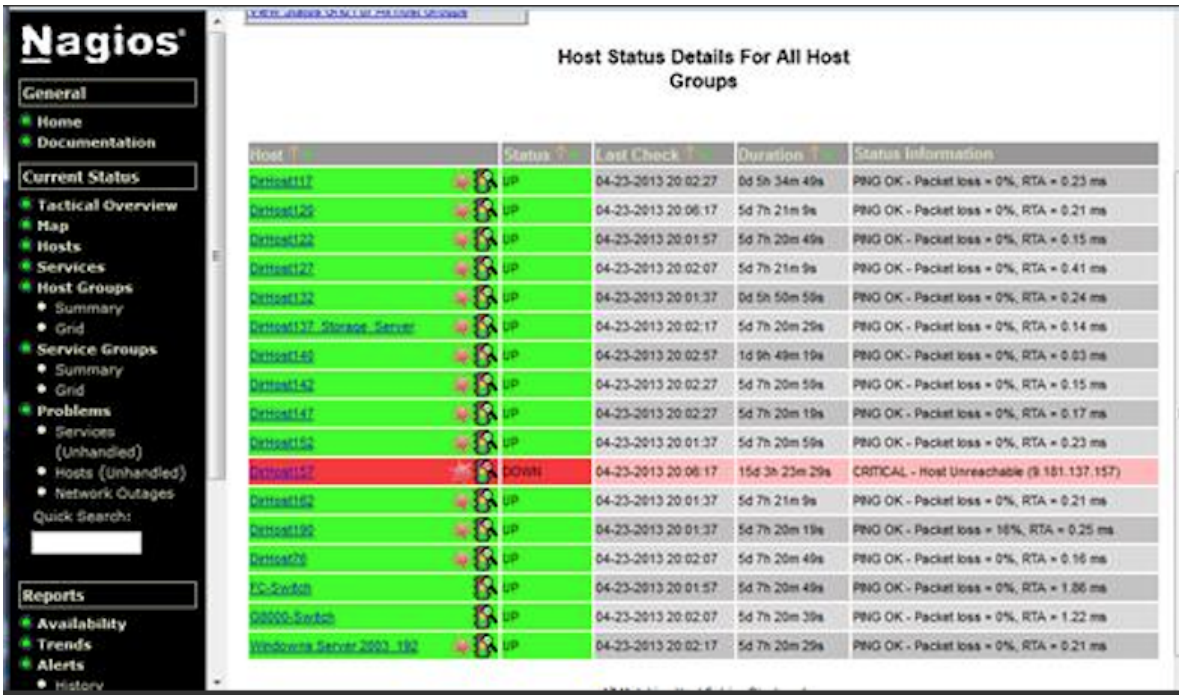
- 长期趋势分析：通过对监控样本数据的持续收集和统计，对监控指标进行长期趋势分析。例如，通过对磁盘空间增长率的判断，我们可以提前预测在未来什么时间节点上需要对资源进行扩容。
- 对照分析：两个版本的系统运行资源使用情况的差异如何？在不同容量情况下系统的并发和负载变化如何？通过监控能够方便的对系统进行跟踪和比较。
- 告警：当系统出现或者即将出现故障时，监控系统需要迅速反应并通知管理员，从而能够对问题进行快速的处理或者提前预防问题的发生，避免出现对业务的影响。
- 故障分析与定位：当问题发生后，需要对问题进行调查和处理。通过对不同监控监控以及历史数据的分析，能够找到并解决根源问题。
- 数据可视化：通过可视化仪表盘能够直接获取系统的运行状态、资源使用情况、以及服务运行状态等直观的信息。

与常见监控系统比较

对于常用的监控系统，如Nagios、Zabbix的用户而言，往往并不能很好的解决上述问题。这里以Nagios为例，如下图所示是Nagios监控系统的基本架构：



Nagios的主要功能是监控服务和主机。Nagios软件需要安装在一台独立的服务器上运行，该服务器称为监控中心。每一台被监控的硬件主机或者服务都需要运行一个与监控中心服务器进行通信的Nagios软件后台程序，可以理解为Agent或者插件。



首先对于Nagios而言，大部分的监控能力都是围绕系统的一些边缘性的问题，主要针对系统服务和资源的状态以及应用程序的可用性。 例如：Nagios通过check_disk插件可以用于检查磁盘空间，check_load用于检查CPU负载等。这些插件会返回4种Nagios可识别的状态，0(OK)表示正常，1(WARNING)表示警告，2(CRITTCAL)表示错误，3(UNKNOWN)表示未知错误，并通过Web UI显示出来。

对于Nagios这类系统而言，其核心是采用了测试和告警(check&alert)的监控系统模型。 对于基于这类模型的监控系统而言往往存在以下问题：

- 与业务脱离的监控：监控系统获取到的监控指标与业务本身也是一种分离的关系。好比客户可能关注的是服务的可用性、服务的SLA等级，而监控系统却只能根据系统负载去产生告警；
- 运维管理难度大：Nagios这一类监控系统本身运维管理难度就比较大，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；
- 可扩展性低： 监控系统自身难以扩展，以适应监控规模的变化；
- 问题定位难度大：当问题产生之后（比如主机负载异常增加）对于用户而言，他们看到的依然是一个黑盒，他们无法了解主机上服务真正的运行情况，因此当故障发生后，这些告警信息并不能有效的支持用户对于故障根源问题的分析和定位。

Prometheus的优势

Prometheus是一个开源的完整监控解决方案，其对传统监控系统的测试和告警模型进行了彻底的颠覆，形成了基于中央化的规则计算、统一分析和告警的新模型。 相比于传统监控系统Prometheus具有以下优点：

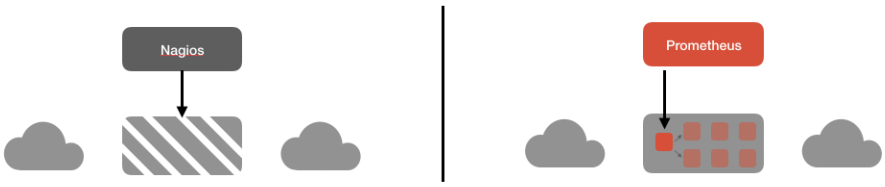
易于管理

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，因此不会有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，还可以使用Prometheus服务发现(Service Discovery)的能力动态管理监控目标。

监控服务的内部运行状态

Pometheus鼓励用户监控服务的内部状态，基于Prometheus丰富的Client库，用户可以轻松的在应用程序中添加对Prometheus的支持，从而让用户可以获取服务和应用内部真正的运行状态。



强大的数据模型

所有采集的监控数据均以指标(metric)的形式保存在内置的时间序列数据库当中(TSDB)。所有的样本除了基本的指标名称以外，还包含一组用于描述该样本特征的标签。

如下所示：

```
http_request_status{code='200',content_path='/api/path', environment='produment'} => [value1@timestamp1,value2@timestamp2...]

http_request_status{code='200',content_path='/api/path2', environment='produment'} => [value1@timestamp1,value2@timestamp2...]
```

每一条时间序列由指标名称(Metrics Name)以及一组标签(Labels)唯一标识。每条时间序列按照时间的先后顺序存储一系列的样本值。

表示维度的标签可能来源于你的监控对象的状态，比如code=404或者content_path=/api/path。也可能来源于的你的环境定义，比如environment=produment。基于这些Labels我们可以方便地对监控数据进行聚合，过滤，裁剪。

强大的查询语言PromQL

Prometheus内置了一个强大的数据查询语言PromQL。 通过PromQL可以实现对监控数据的查询、聚合。同时PromQL也被应用于数据可视化(如Grafana)以及告警当中。

通过PromQL可以轻松回答类似于以下问题：

- 在过去一段时间中95%应用延迟时间的分布范围？
- 预测在4小时后，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？(过滤)

高效

对于监控系统而言，大量的监控任务必然导致有大量的数据产生。而Prometheus可以高效地处理这些数据，对于单一Prometheus Server实例而言它可以处理：

- 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、每个团队运行独立的Prometheus Sevrer。Prometheus对于联邦集群的支持，可以让多个Prometheus实例产生一个逻辑集群，当单实例Prometheus Server处理的任务量过大时，通过使用功能分区(sharding)+联邦集群(federation)可以对其进行扩展。

易于集成

使用Prometheus可以快速搭建监控服务，并且可以非常方便地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， muini， Nagios等。

Prometheus社区还提供了大量第三方实现的监控数据采集支持：JMX， CloudWatch， EC2， MySQL， PostgresSQL， Haskell， Bash， SNMP， Consul， Haproxy， Mesos， Bind， CouchDB， Django， 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可以方便地直接对数据进行查询，并且支持直接以图形化的形式展示数据。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。

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。

接下来，在本书当中，将带领读者感受Prometheus是如何对监控系统的重新定义。

初识Prometheus

Prometheus是一个开放性的监控解决方案，用户可以非常方便的安装和使用Prometheus并且能够非常方便的对其进行扩展。为了能够更加直观的了解Prometheus Server，接下来我们将在本地部署并运行一个Prometheus Server实例，通过Node Exporter采集当前主机的系统资源使用情况。 并通过Grafana创建一个简单的可视化仪表盘。

安装Prometheus Server

Prometheus基于Golang编写，编译后的软件包，不依赖于任何的第三方依赖。用户只需要下载对应平台的二进制包，解压并且添加基本的配置即可正常启动Prometheus Server。

从二进制包安装

对于非Docker用户，可以从https://prometheus.io/download/找到最新版本的Prometheus Sevrer软件包：

```
export VERSION=2.4.3
curl -LO  https://github.com/prometheus/prometheus/releases/download/v$VERSION/prometheus-$VERSION.darwin-amd64.tar.gz
```

解压，并将Prometheus相关的命令，添加到系统环境变量路径即可：

```
tar -xzf prometheus-${VERSION}.darwin-amd64.tar.gz
cd prometheus-${VERSION}.darwin-amd64
```

解压后当前目录会包含默认的Prometheus配置文件promethes.yml:

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['localhost:9090']
```

Promtheus作为一个时间序列数据库，其采集的数据会以文件的形式存储在本地中，默认的存储路径为 `data/` ，因此我们需要先手动创建该目录：

```
mkdir -p data
```

用户也可以通过参数 `--storage.tsdb.path="data/"` 修改本地数据存储的路径。

启动prometheus服务，其会默认加载当前路径下的prometheus.yaml文件：

```
./prometheus
```

正常的情况下，你可以看到以下输出内容：

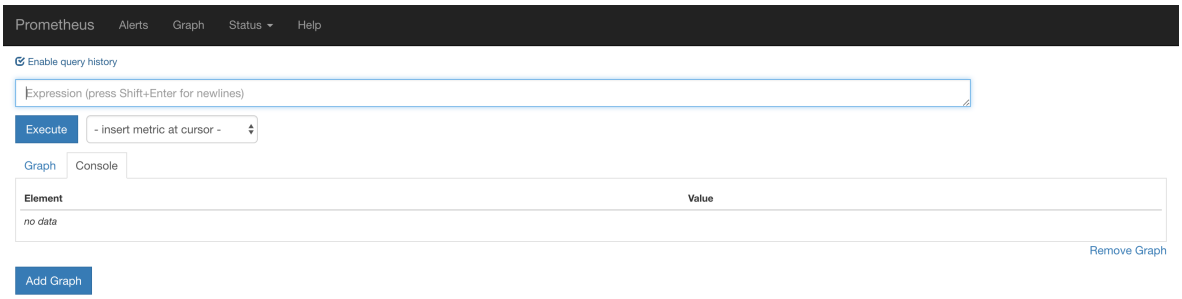
```
level=info ts=2018-10-23T14:55:14.499484Z caller=main.go:554 msg="Starting TSDB ..."
level=info ts=2018-10-23T14:55:14.499531Z caller=web.go:397 component=web msg="Start listening for connections" address=0.0.0.0:9090
level=info ts=2018-10-23T14:55:14.507999Z caller=main.go:564 msg="TSDB started"
level=info ts=2018-10-23T14:55:14.508068Z caller=main.go:624 msg="Loading configuration file" filename=prometheus.yml
level=info ts=2018-10-23T14:55:14.509509Z caller=main.go:650 msg="Completed loading of configuration file" filename=prometheus.yml
level=info ts=2018-10-23T14:55:14.509537Z caller=main.go:523 msg="Server is ready to receive web requests."
```

使用容器安装

对于Docker用户，直接使用Prometheus的镜像即可启动Prometheus Server：

```
docker run -p 9090:9090 -v /etc/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

启动完成后，可以通过http://localhost:9090访问Prometheus的UI界面：



使用Node Exporter采集主机数据

安装Node Exporter

在Prometheus的架构设计中，Prometheus Server并不直接服务监控特定的目标，其主要任务负责数据的收集，存储并且对外提供数据查询支持。因此为了能够能够监控到某些东西，如主机的CPU使用率，我们需要使用到Exporter。Prometheus周期性的从Exporter暴露的HTTP服务地址（通常是/metrics）拉取监控样本数据。

从上面的描述中可以看出Exporter可以是一个相对开放的概念，其可以是一个独立运行的程序独立于监控目标以外，也可以是直接内置在监控目标中。只要能够向Prometheus提供标准格式的监控样本数据即可。

这里为了能够采集到主机的运行指标如CPU, 内存，磁盘等信息。我们可以使用Node Exporter。

Node Exporter同样采用Golang编写，并且不存在任何的第三方依赖，只需要下载，解压即可运行。可以从https://prometheus.io/download/获取最新的node exporter版本的二进制包。

```
curl -OL https://github.com/prometheus/node_exporter/releases/download/v0.15.2/node_exporter-0.15.2.darwin-amd64.tar.gz
tar -xzf node_exporter-0.15.2.darwin-amd64.tar.gz
```

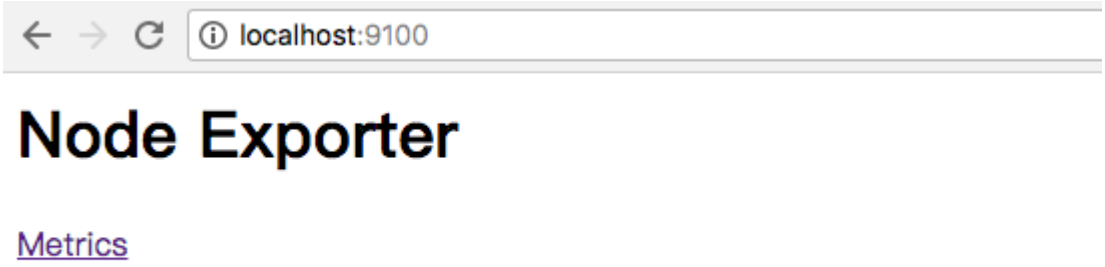
运行node exporter:

```
cd node_exporter-0.15.2.darwin-amd64
cp node_exporter-0.15.2.darwin-amd64/node_exporter /usr/local/bin/
node_exporter
```

启动成功后，可以看到以下输出：

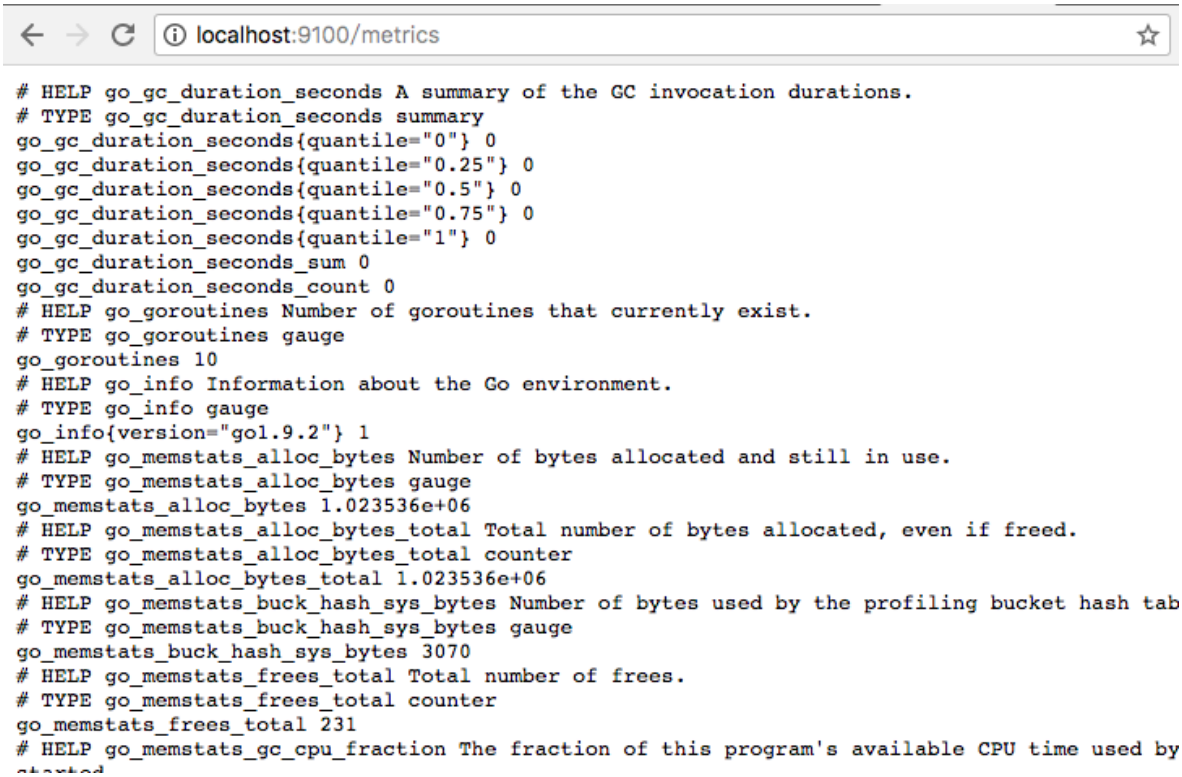
```
INFO[0000] Listening on :9100                            source="node_exporter.go:76"
```

访问http://localhost:9100/可以看到以下页面：



初始Node Exporter监控指标

访问http://localhost:9100/metrics，可以看到当前node exporter获取到的当前主机的所有监控数据，如下所示：



每一个监控指标之前都会有一段类似于如下形式的信息：

```
# HELP node_cpu Seconds the cpus spent in each mode.
# TYPE node_cpu counter
node_cpu{cpu="cpu0",mode="idle"} 362812.7890625
# HELP node_load1 1m load average.
# TYPE node_load1 gauge
node_load1 3.0703125
```

其中HELP用于解释当前指标的含义，TYPE则说明当前指标的数据类型。在上面的例子中node_cpu的注释表明当前指标是cpu0上idle进程占用CPU的总时间，CPU占用时间是一个只增不减的度量指标，从类型中也可以看出node_cpu的数据类型是计数器(counter)，与该指标的实际含义一致。又例如node_load1该指标反映了当前主机在最近一分钟以内的负载情况，系统的负载情况会随系统资源的使用而变化，因此node_load1反映的是当前状态，数据可能增加也可能减少，从注释中可以看出当前指标类型为仪表盘(gauge)，与指标反映的实际含义一致。

除了这些以外，在当前页面中根据物理主机系统的不同，你还可能看到如下监控指标：

- node_boot_time：系统启动时间
- node_cpu：系统CPU使用量
- node_disk_*：磁盘IO
- node_filesystem_*：文件系统用量
- node_load1：系统负载
- node_memeory_*：内存使用量
- node_network_*：网络带宽
- node_time：当前系统时间
- go_*：node exporter中go相关指标
- process_*：node exporter自身进程相关运行指标

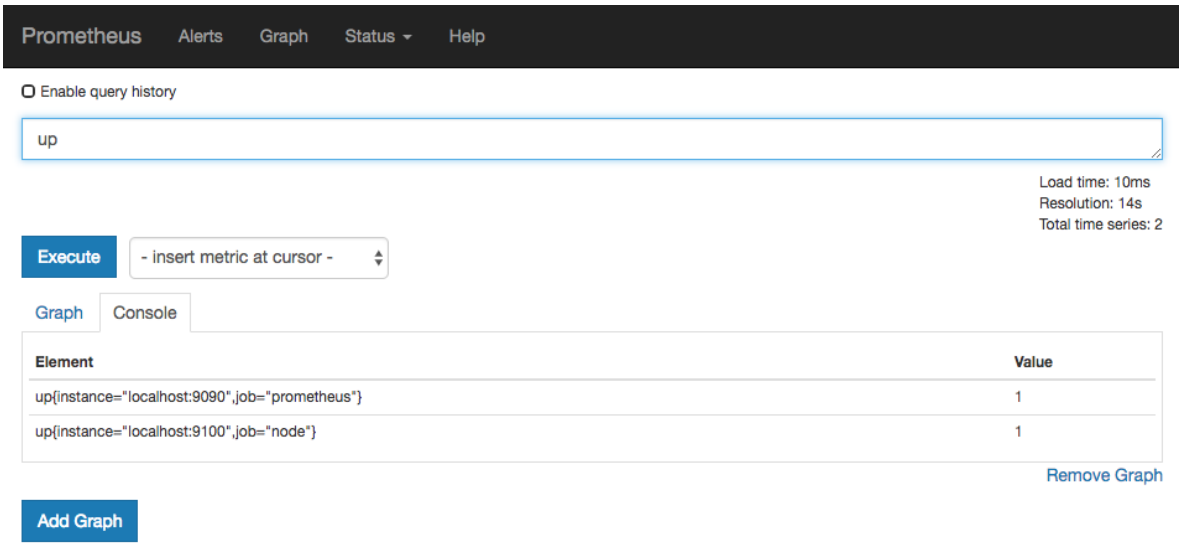
从Node Exporter收集监控数据

为了能够让Prometheus Server能够从当前node exporter获取到监控数据，这里需要修改Prometheus配置文件。编辑prometheus.yml并在scrape_configs节点下添加以下内容:

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  # 采集node exporter监控数据
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

重新启动Prometheus Server

访问http://localhost:9090，进入到Prometheus Server。如果输入“up”并且点击执行按钮以后，可以看到如下结果：



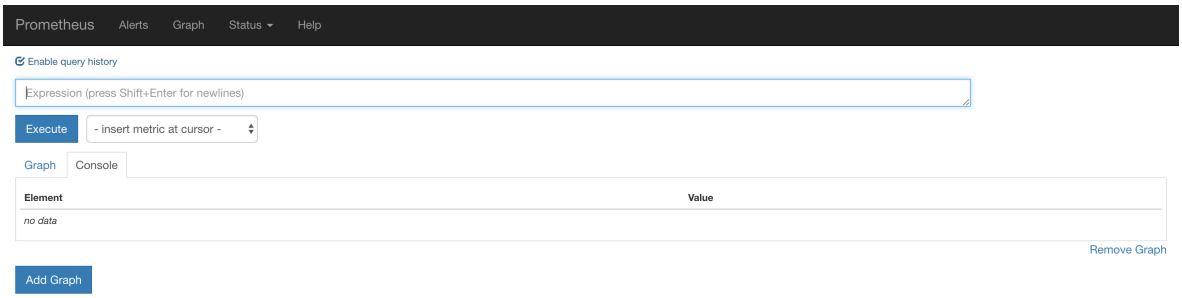
如果Prometheus能够正常从node exporter获取数据，则会看到以下结果：

```
up{instance="localhost:9090", job="prometheus"}    1
up{instance="localhost:9100", job="node"}    1
```

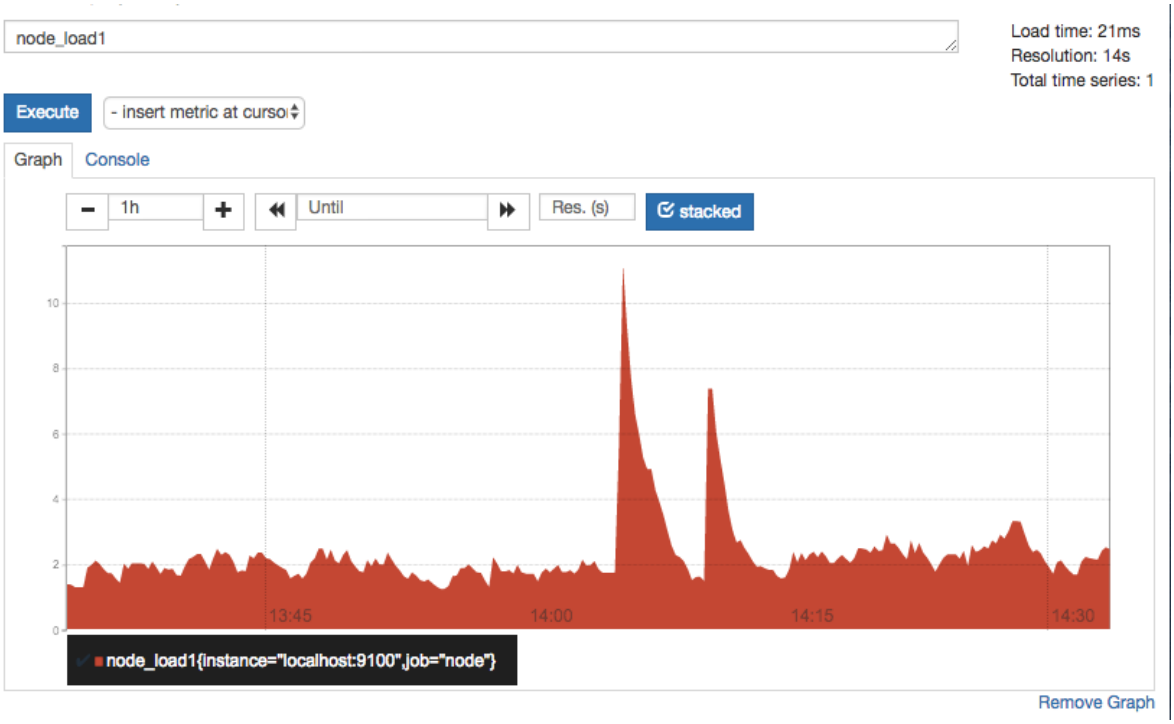
其中“1”表示正常，反之“0”则为异常。

使用PromQL查询监控数据

Prometheus UI是Prometheus内置的一个可视化管理界面，通过Prometheus UI用户能够轻松的了解Prometheus当前的配置，监控任务运行状态等。 通过 `Graph` 面板，用户还能直接使用 `PromQL` 实时查询监控数据：

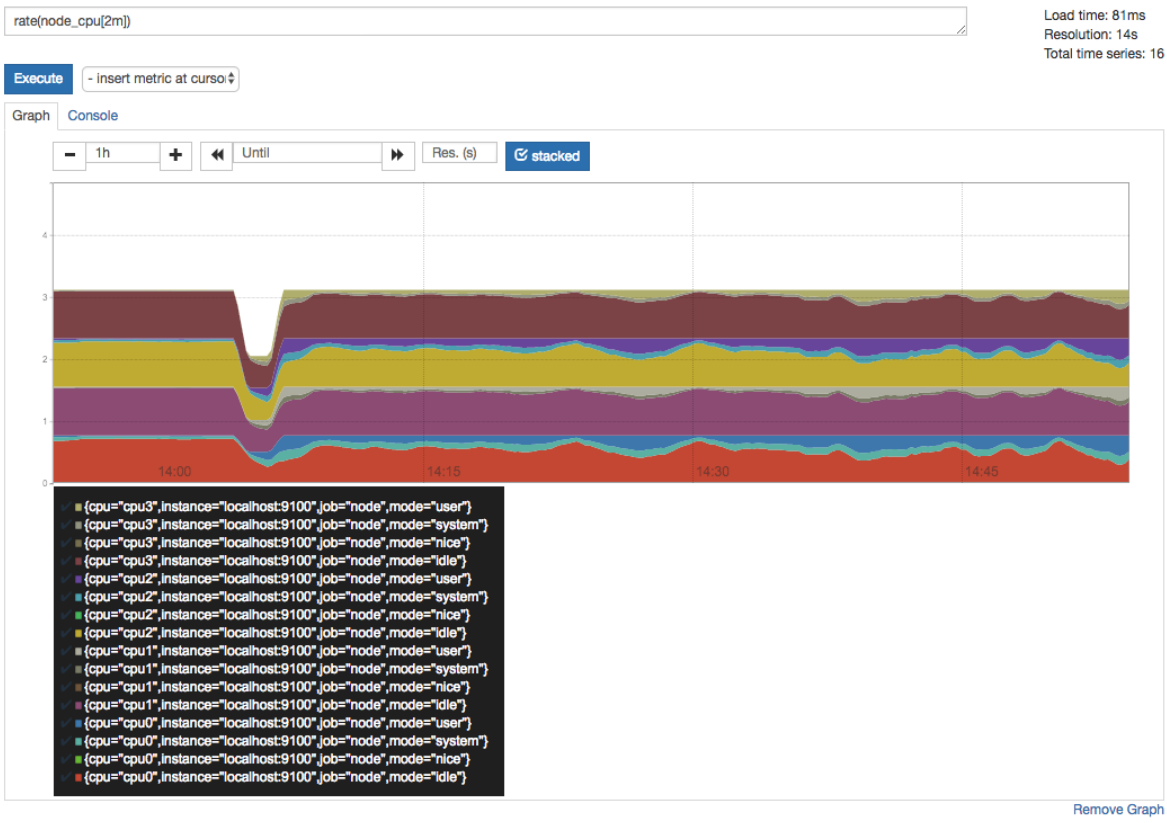


切换到 `Graph` 面板，用户可以使用PromQL表达式查询特定监控指标的监控数据。如下所示，查询主机负载变化情况，可以使用关键字 `node_load1` 可以查询出Prometheus采集到的主机负载的样本数据，这些样本数据按照时间先后顺序展示，形成了主机负载随时间变化的趋势图表：



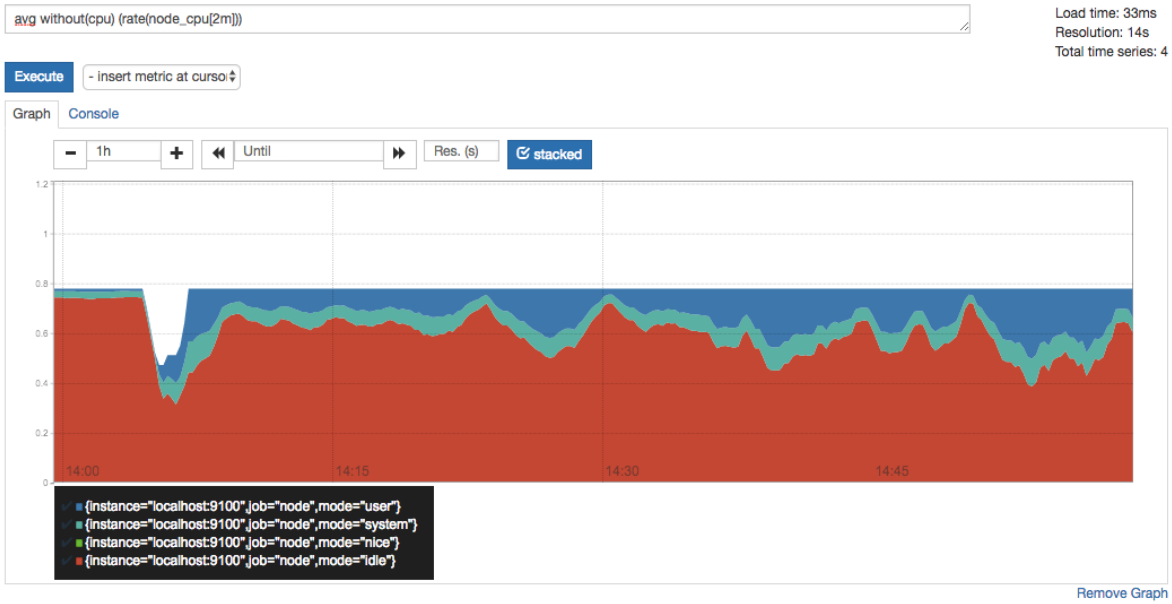
PromQL是Prometheus自定义的一套强大的数据查询语言，除了使用监控指标作为查询关键字以为，还内置了大量的函数，帮助用户进一步对时序数据进行处理。例如使用 `rate()` 函数，可以计算在单位时间内样本数据的变化情况即增长率，因此通过该函数我们可以近似的通过CPU使用时间计算CPU的利用率：

```
rate(node_cpu[2m])
```



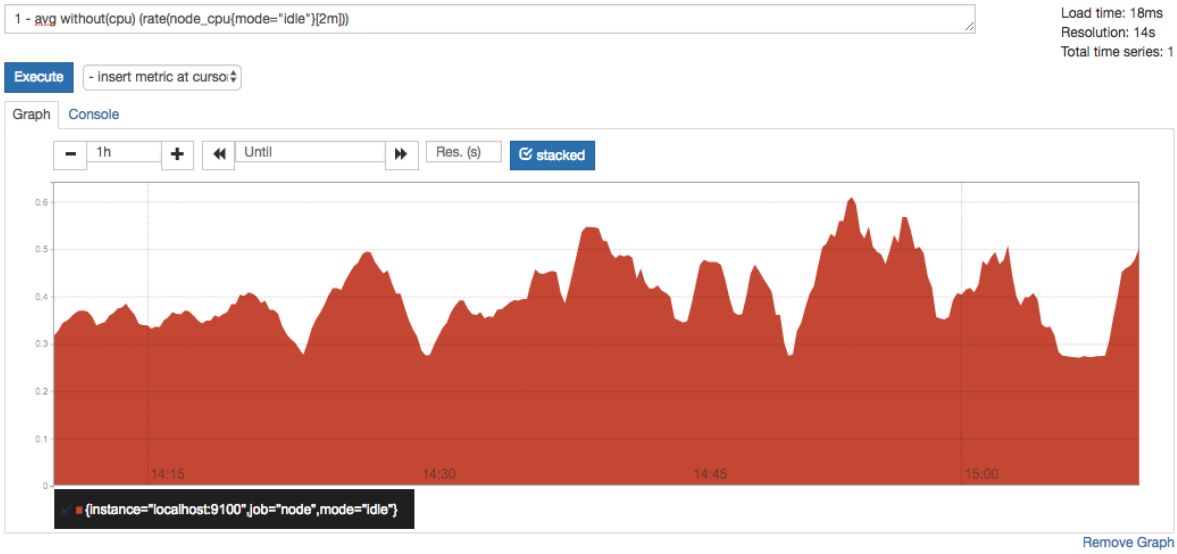
这时如果要忽略是哪一个CPU的，只需要使用without表达式，将标签CPU去除后聚合数据即可：

```
avg without(cpu) (rate(node_cpu[2m]))
```



那如果需要计算系统CPU的总体使用率，通过排除系统闲置的CPU使用率即可获得：

```
1 - avg without(cpu) (rate(node_cpu{mode="idle"}[2m]))
```



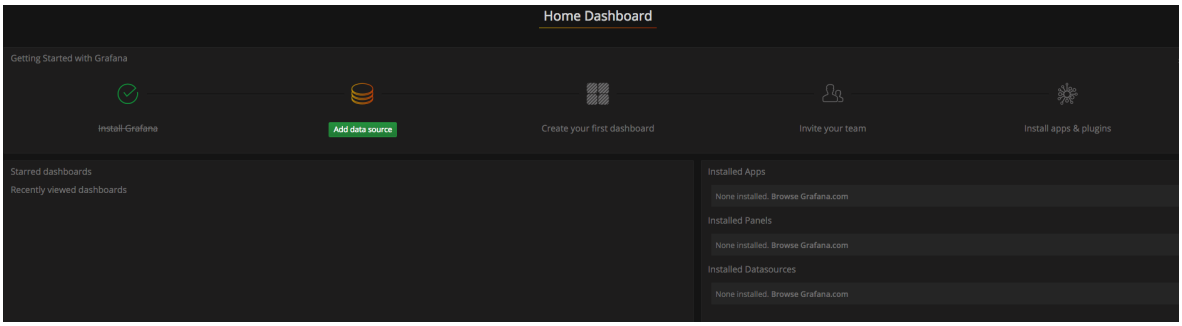
通过PromQL我们可以非常方便的对数据进行查询，过滤，以及聚合，计算等操作。通过这些丰富的表达书语句，监控指标不再是一个单独存在的个体，而是一个个能够表达出正式业务含义的语言。

监控数据可视化

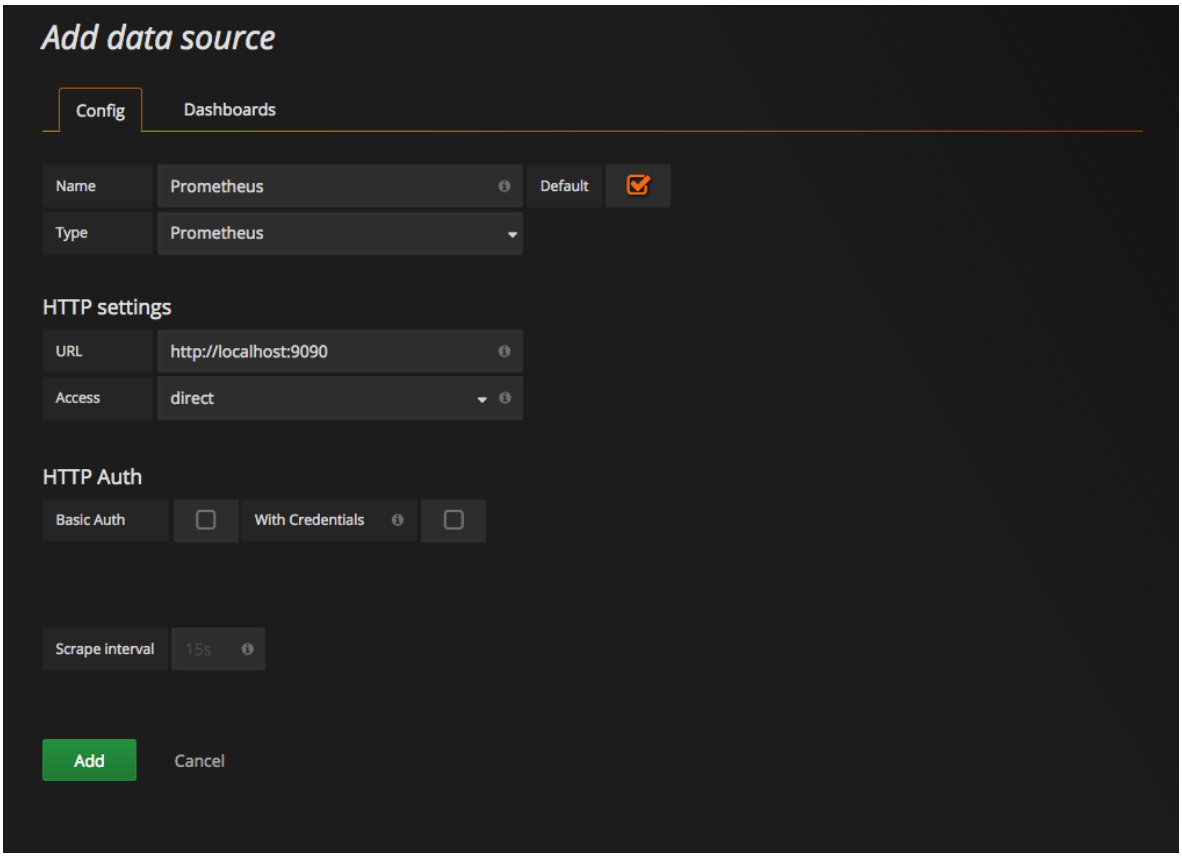
Prometheus UI提供了快速验证PromQL以及临时可视化支持的能力，而在大多数场景下引入监控系统通常还需要构建可以长期使用的监控数据可视化面板（Dashboard）。这时用户可以考虑使用第三方的可视化工具如Grafana，Grafana是一个开源的可视化平台，并且提供了对Prometheus的完整支持。

```
docker run -d -p 3000:3000 grafana/grafana
```

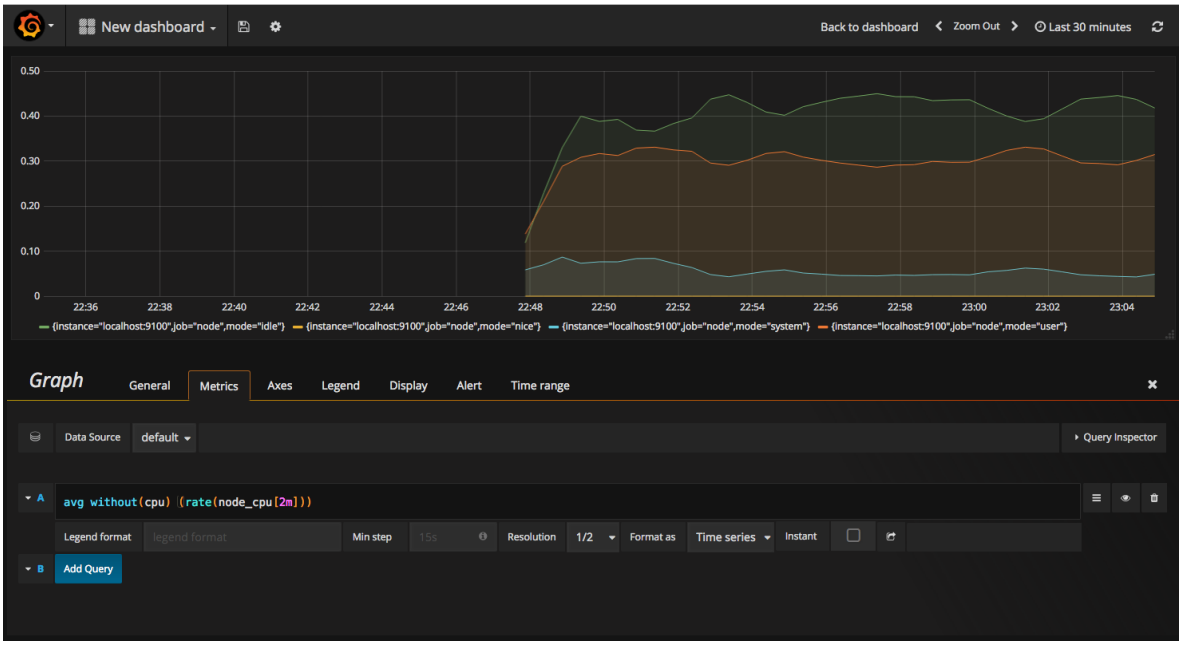
访问http://localhost:3000就可以进入到Grafana的界面中，默认情况下使用账户admin/admin进行登录。在Grafana首页中显示默认的使用向导，包括：安装、添加数据源、创建Dashboard、邀请成员、以及安装应用和插件等主要流程:



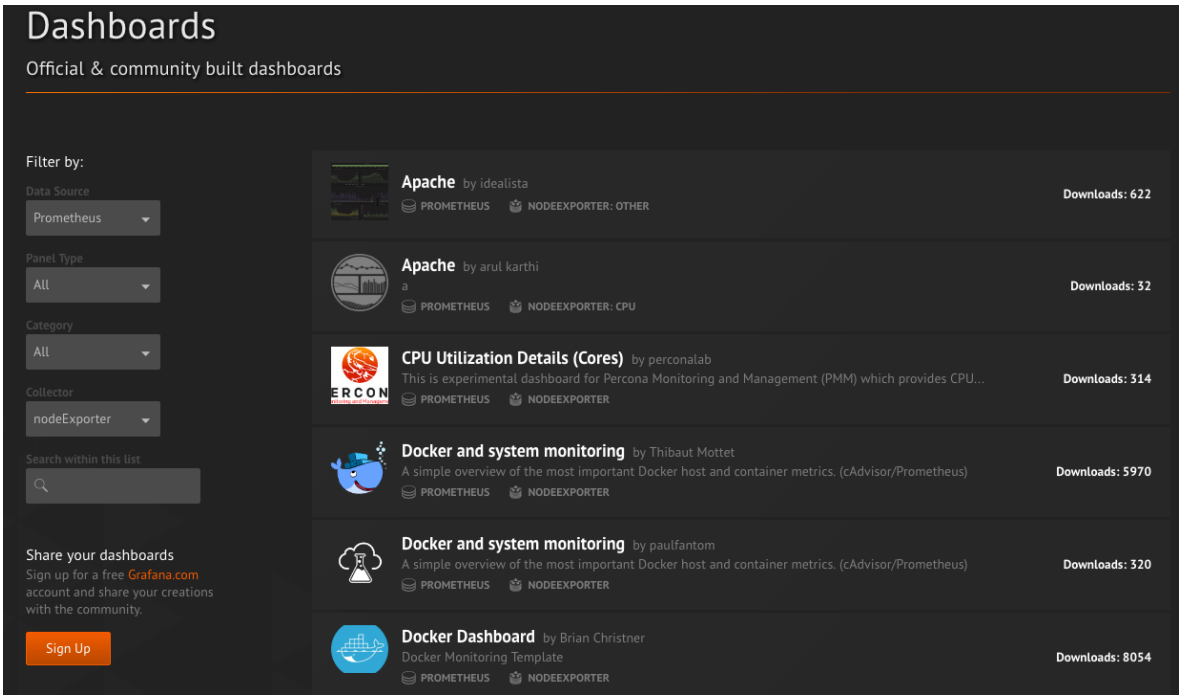
这里将添加Prometheus作为默认的数据源，如下图所示，指定数据源类型为Prometheus并且设置Prometheus的访问地址即可，在配置正确的情况下点击“Add”按钮，会提示连接成功的信息：



在完成数据源的添加之后就可以在Grafana中创建我们可视化Dashboard了。Grafana提供了对PromQL的完整支持，如下所示，通过Grafana添加Dashboard并且为该Dashboard添加一个类型为“Graph”的面板。 并在该面板的“Metrics”选项下通过PromQL查询需要可视化的数据：



点击界面中的保存选项，就创建了我们的第一个可视化Dashboard了。 当然作为开源软件，Grafana社区鼓励用户分享Dashboard通过https://grafana.com/dashboards网站，可以找到大量可直接使用的Dashboard：



Grafana中所有的Dashboard通过JSON进行共享，下载并且导入这些JSON文件，就可以直接使用这些已经定义好的Dashboard：


Host Stats
Zoom Out
Last 30 minutes
Machine: mari:9100
CPU
100.00%
75.00%
50.00%
25.00%
0%
20:30
20:35
20:40
20:45
20:50
20:55
guest
iowait
irq
nice
softirq
steal
system
user
Memory
19 GiB
14 GiB
9 GiB
5 GiB
0 B
Used
Buffers
Cached
Free
Disk I/O Utilisation
sda
sdb
sdc
sr0
Filesystem Fullness
/
/home
/opt
/srv
/tmp
Filesystem Fill Up Time

任务和实例

在上一小节中，通过在prometheus.yml配置文件中，添加如下配置。我们让Prometheus可以从node exporter暴露的服务中获取监控指标数据。

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

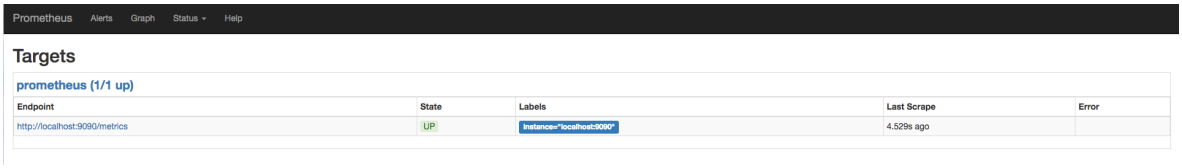
当我们需要采集不同的监控指标(例如：主机、MySQL、Nginx)时，我们只需要运行相应的监控采集程序，并且让Prometheus Server知道这些Exporter实例的访问地址。在Prometheus中，每一个暴露监控样本数据的HTTP服务称为一个实例。例如在当前主机上运行的node exporter可以被称为一个实例(Instance)。

而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则通过一个一个任务(Job)进行管理。

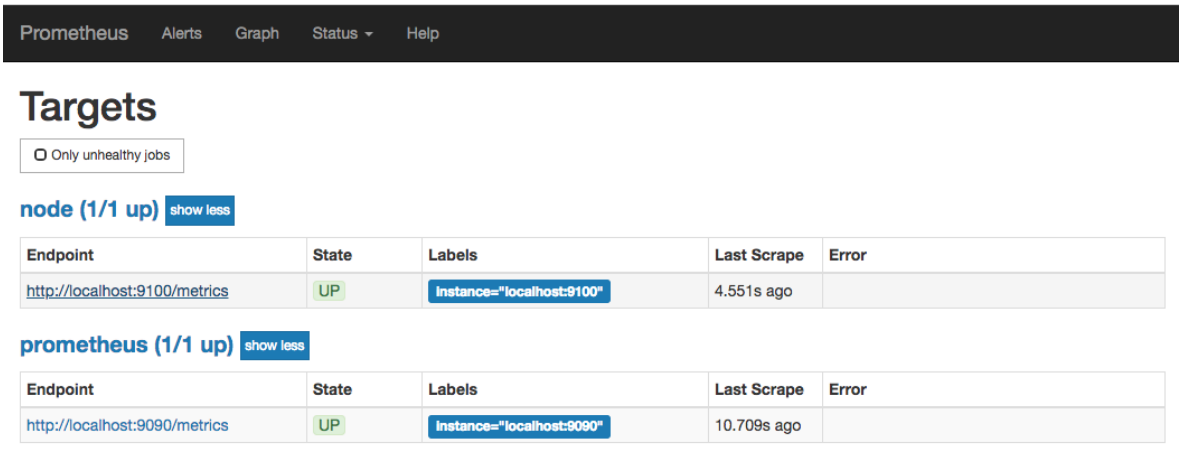
```
* job: node
    * instance 2: 1.2.3.4:9100
    * instance 4: 5.6.7.8:9100
```

当前在每一个Job中主要使用了静态配置(static_configs)的方式定义监控目标。除了静态配置每一个Job的采集Instance地址以外，Prometheus还支持与DNS、Consul、E2C、Kubernetes等进行集成实现自动发现Instance实例，并从这些Instance上获取监控数据。

除了通过使用“up”表达式查询当前所有Instance的状态以外，还可以通过Prometheus UI中的Targets页面查看当前所有的监控采集任务，以及各个任务下所有实例的状态:



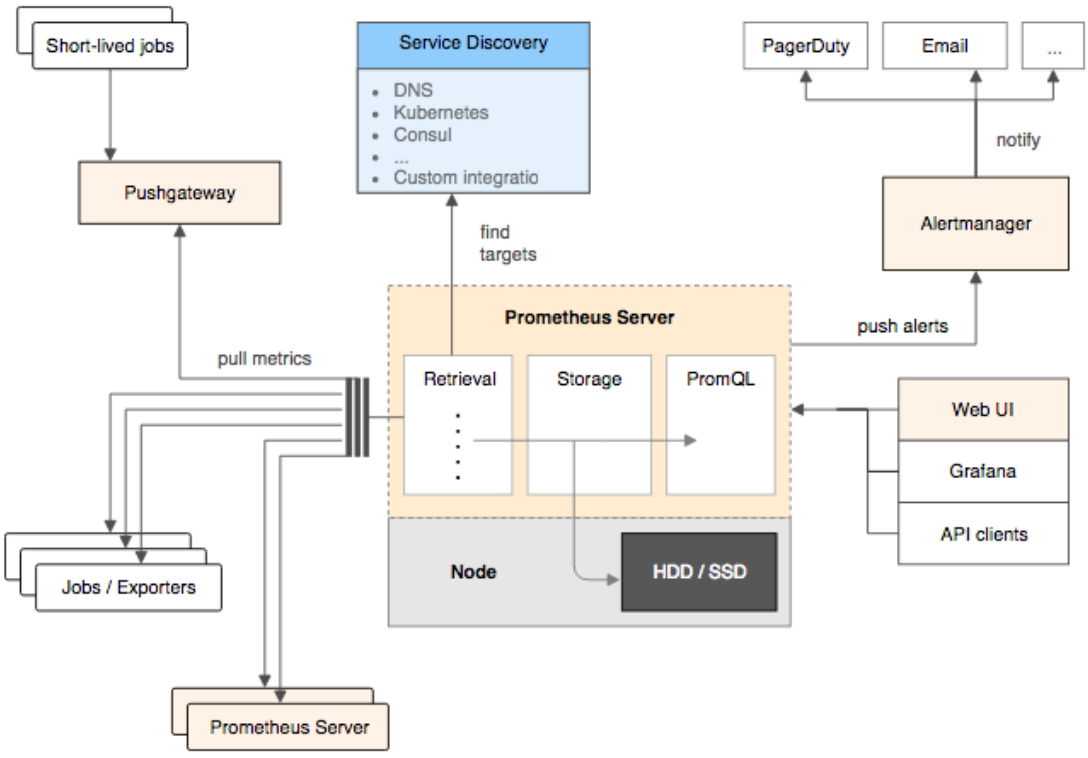
我们也可以访问http://192.168.33.10:9090/targets直接从Prometheus的UI中查看当前所有的任务以及每个任务对应的实例信息。



Prometheus核心组件

上一小节，通过部署Node Exporter我们成功的获取到了当前主机的资源使用情况。接下来我们将从Prometheus的架构角度详细介绍Prometheus生态中的各个组件。

下图展示Prometheus的基本架构：



Prometheus Server

Prometheus Server是Prometheus组件中的核心部分，负责实现对监控数据的获取，存储以及查询。
Prometheus Server可以通过静态配置管理监控目标，也可以配合使用Service Discovery的方式动态管理监控目标，并从这些监控目标中获取数据。其次Prometheus Server需要对采集到的监控数据进行存储，Prometheus Server本身就是一个时序数据库，将采集到的监控数据按照时间序列的方式存储在本地磁盘当中。最后Prometheus Server对外提供了自定义的PromQL语言，实现对数据的查询以及分析。

Prometheus Server内置的Express Browser UI，通过这个UI可以直接通过PromQL实现数据的查询以及可视化。

Prometheus Server的联邦集群能力可以使其从其他的Prometheus Server实例中获取数据，因此在大规模监控的情况下，可以通过联邦集群以及功能分区的方式对Prometheus Server进行扩展。

Exporters

Exporter将监控数据采集的端点通过HTTP服务的形式暴露给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中支持基于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体系中的告警处理中心。

PushGateway

由于Prometheus数据采集基于Pull模型进行设计，因此在网络环境的配置上必须要让Prometheus Server能够直接与Exporter进行通信。 当这种网络需求无法直接满足时，就可以利用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。

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，提供一定的参考。同时我们介绍了Prometheus的生态以及核心能力，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，相信读者通过本章能够对Prometheus有一个直观的认识。

探索PromQL

本章将带领读者探秘Prometheus的自定义查询语言PromQL。通过PromQL用户可以非常方便地对监控样本数据进行统计分析，PromQL支持常见的运算操作符，同时PromQL中还提供了大量的内置函数可以实现对数据的高级处理。当然在学习PromQL之前，用户还需要了解Prometheus的样本数据模型。PromQL作为Prometheus的核心能力除了实现数据的对外查询和展现，同时告警监控也是依赖PromQL实现的。

本章的主要内容：

- Prometheus的数据模型
- Prometheus中监控指标的类型
- 深入PromQL
- 4个黄金指标和USE方法

理解时间序列

在1.2节当中，通过Node Exporter暴露的HTTP服务，Prometheus可以采集到当前主机所有监控指标的样本数据。例如：

```
# HELP node_cpu Seconds the cpus spent in each mode.
# TYPE node_cpu counter
node_cpu{cpu="cpu0",mode="idle"} 362812.7890625
# HELP node_load1 1m load average.
# TYPE node_load1 gauge
node_load1 3.0703125
```

其中非#开头的每一行表示当前Node Exporter采集到的一个监控样本：node_cpu和node_load1表明了当前指标的名称、大括号中的标签则反映了当前样本的一些特征和维度、浮点数则是该监控样本的具体值。

样本

Prometheus会将所有采集到的样本数据以时间序列（time-series）的方式保存在内存数据库中，并且定时保存到硬盘上。time-series是按照时间戳和值的序列顺序存放的，我们称之为向量(vector). 每条time-series通过指标名称(metrics name)和一组标签集(labelset)命名。如下所示，可以将time-series理解为一个以时间为Y轴的数字矩阵：

```
  ^
  |   . . . . . . . . . . . . . . . . . .   . .   node_cpu{cpu="cpu0",mode="idle"}
  |     . . . . . . . . . . . . . . . . . .   .   node_cpu{cpu="cpu0",mode="system"}
  |     . . . . . . . . . .   . . . . . . . .   node_load1{}
  |     . . . . . . . . . . . . . . . .   . .
  v
    <------------------ 时间 ---------------->
```

在time-series中的每一个点称为一个样本（sample），样本由以下三部分组成：

- 指标(metric)：metric name和描述当前样本特征的labelsets;
- 时间戳(timestamp)：一个精确到毫秒的时间戳;
- 样本值(value)： 一个float64的浮点型数据表示当前样本的值。

```
<-------------- metric ---------------------><-timestamp -><-value->
http_request_total{status="200", method="GET"}@1434417560938 => 94355
http_request_total{status="200", method="GET"}@1434417561287 => 94334

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

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

指标(Metric)

在形式上，所有的指标(Metric)都通过如下格式标示：

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

指标的名称(metric name)可以反映被监控样本的含义（比如，`http_request_total` - 表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并必须符合正则表达式 `[a-zA-Z_:][a-zA-Z0-9_:]*` 。

标签(label)反映了当前样本的特征维度，通过这些维度Prometheus可以对样本数据进行过滤，聚合等。标签的名称只能由ASCII字符、数字以及下划线组成并满足正则表达式 `[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。在Prometheus的底层实现中指标名称实际上是以 `__name__=<metric name>` 的形式保存在数据库中的，因此以下两种方式均表示的同一条time-series：

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

等同于：

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

在Prometheus源码中也可以找到指标(Metric)对应的数据结构，如下所示：

```
type Metric LabelSet

type LabelSet map[LabelName]LabelValue

type LabelName string

type LabelValue string
```

Metrics类型

在上一小节中我们带领读者了解了Prometheus的底层数据模型，在Prometheus的存储实现上所有的监控样本都是以time-series的形式保存在Prometheus内存的TSDB（时序数据库）中，而time-series所对应的监控指标(metric)也是通过labelset进行唯一命名的。

从存储上来讲所有的监控指标metric都是相同的，但是在不同的场景下这些metric又有一些细微的差异。 例如，在Node Exporter返回的样本中指标node_load1反应的是当前系统的负载状态，随着时间的变化这个指标返回的样本数据是在不断变化的。而指标node_cpu所获取到的样本数据却不同，它是一个持续增大的值，因为其反应的是CPU的累积使用时间，从理论上讲只要系统不关机，这个值是会无限变大的。

为了能够帮助用户理解和区分这些不同监控指标之间的差异，Prometheus定义了4种不同的指标类型(metric type)：Counter（计数器）、Gauge（仪表盘）、Histogram（直方图）、Summary（摘要）。

在Exporter返回的样本数据中，其注释中也包含了该样本的类型。例如：

```
# HELP node_cpu Seconds the cpus spent in each mode.
# TYPE node_cpu counter
node_cpu{cpu="cpu0",mode="idle"} 362812.7890625
```

Counter：只增不减的计数器

Counter类型的指标其工作方式和计数器一样，只增不减（除非系统发生重置）。常见的监控指标，如http_requests_total，node_cpu都是Counter类型的监控指标。 一般在定义Counter类型指标的名称时推荐使用_total作为后缀。

Counter是一个简单但有强大的工具，例如我们可以在应用程序中记录某些事件发生的次数，通过以时序的形式存储这些数据，我们可以轻松的了解该事件产生速率的变化。
PromQL内置的聚合操作和函数可以让用户对这些数据进行进一步的分析：

例如，通过rate()函数获取HTTP请求量的增长率：

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

查询当前系统中，访问量前10的HTTP地址：

```
topk(10, http_requests_total)
```

Gauge：可增可减的仪表盘

与Counter不同，Gauge类型的指标侧重于反应系统的当前状态。因此这类指标的样本数据可增可减。常见指标如：node_memory_MemFree（主机当前空闲的内容大小）、node_memory_MemAvailable（可用内存大小）都是Gauge类型的监控指标。

通过Gauge指标，用户可以直接查看系统的当前状态：

```
node_memory_MemFree
```

对于Gauge类型的监控指标，通过PromQL内置函数delta()可以获取样本在一段时间返回内的变化情况。例如，计算CPU温度在两个小时内的差异：

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

还可以使用deriv()计算样本的线性回归模型，甚至是直接使用predict_linear()对数据的变化趋势进行预测。例如，预测系统磁盘空间在4个小时之后的剩余情况：

```
predict_linear(node_filesystem_free{job="node"}[1h], 4 * 3600)
```

使用Histogram和Summary分析数据分布情况

除了Counter和Gauge类型的监控指标以外，Prometheus还定义了Histogram和Summary的指标类型。Histogram和Summary主用用于统计和分析样本的分布情况。

在大多数情况下人们都倾向于使用某些量化指标的平均值，例如CPU的平均使用率、页面的平均响应时间。这种方式的问题很明显，以系统API调用的平均响应时间为例：如果大多数API请求都维持在100ms的响应时间范围内，而个别请求的响应时间需要5s，那么就会导致某些WEB页面的响应时间落到中位数的情况，而这种现象被称为长尾问题。

为了区分是平均的慢还是长尾的慢，最简单的方式就是按照请求延迟的范围进行分组。例如，统计延迟在0~~10ms之间的请求数有多少而10~~20ms之间的请求数又有多少。通过这种方式可以快速分析系统慢的原因。Histogram和Summary都是为了能够解决这样问题的存在，通过Histogram和Summary类型的监控指标，我们可以快速了解监控样本的分布情况。

例如，指标prometheus_tsdb_wal_fsync_duration_seconds的指标类型为Summary。 它记录了Prometheus Server中wal_fsync处理的处理时间，通过访问Prometheus Server的/metrics地址，可以获取到以下监控样本数据：

```
# HELP prometheus_tsdb_wal_fsync_duration_seconds Duration of WAL fsync.
# TYPE prometheus_tsdb_wal_fsync_duration_seconds summary
prometheus_tsdb_wal_fsync_duration_seconds{quantile="0.5"} 0.012352463
prometheus_tsdb_wal_fsync_duration_seconds{quantile="0.9"} 0.014458005
prometheus_tsdb_wal_fsync_duration_seconds{quantile="0.99"} 0.017316173
prometheus_tsdb_wal_fsync_duration_seconds_sum 2.888716127000002
prometheus_tsdb_wal_fsync_duration_seconds_count 216
```

从上面的样本中可以得知当前Prometheus Server进行wal_fsync操作的总次数为216次，耗时2.888716127000002s。其中中位数（quantile=0.5）的耗时为0.012352463，9分位数（quantile=0.9）的耗时为0.014458005s。

在Prometheus Server自身返回的样本数据中，我们还能找到类型为Histogram的监控指标prometheus_tsdb_compaction_chunk_range_bucket。

```
# HELP prometheus_tsdb_compaction_chunk_range Final time range of chunks on their first compaction
# TYPE prometheus_tsdb_compaction_chunk_range histogram
prometheus_tsdb_compaction_chunk_range_bucket{le="100"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="400"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="1600"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="6400"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="25600"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="102400"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="409600"} 0
prometheus_tsdb_compaction_chunk_range_bucket{le="1.6384e+06"} 260
prometheus_tsdb_compaction_chunk_range_bucket{le="6.5536e+06"} 780
prometheus_tsdb_compaction_chunk_range_bucket{le="2.62144e+07"} 780
prometheus_tsdb_compaction_chunk_range_bucket{le="+Inf"} 780
prometheus_tsdb_compaction_chunk_range_sum 1.1540798e+09
prometheus_tsdb_compaction_chunk_range_count 780
```

与Summary类型的指标相似之处在于Histogram类型的样本同样会反应当前指标的记录的总数(以*count作为后缀)以及其值的总量*（*以*sum作为后缀）。不同在于Histogram指标直接反应了在不同区间内样本的个数，区间通过标签len进行定义。

同时对于Histogram的指标，我们还可以通过histogram_quantile()函数计算出其值的分位数。不同在于Histogram通过histogram_quantile函数是在服务器端计算的分位数。 而Sumamry的分位数则是直接在客户端计算完成。因此对于分位数的

计算而言，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。反之对于客户端而言Histogram消耗的资源更少。在选择这两种方式时用户应该按照自己的实际场景进行选择。

初识PromQL

Prometheus通过指标名称（metrics name）以及对应的一组标签（labelset）唯一定义一条时间序列。指标名称反映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。

PromQL是Prometheus内置的数据查询语言，其提供对时间序列数据丰富的查询，聚合以及逻辑运算能力的支持。并且被广泛应用在Prometheus的日常应用当中，包括对数据查询、可视化、告警处理当中。可以这么说，PromQL是Prometheus所有应用场景的基础，理解和掌握PromQL是Prometheus入门的第一课。

查询时间序列

当Prometheus通过Exporter采集到相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。

当我们直接使用监控指标名称查询时，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{}
```

该表达式会返回指标名称为http_requests_total的所有时间序列：

```
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)
```

PromQL还支持用户根据时间序列的标签匹配模式来对时间序列进行过滤，目前主要支持两种匹配模式：完全匹配和正则匹配。

PromQL支持使用 `=` 和 `!=` 两种完全匹配模式：

- 通过使用 `label=value` 可以选择那些标签满足表达式定义的时间序列；
- 反之使用 `label!=value` 则可以根据标签匹配排除时间序列；

例如，如果我们只需要查询所有http_requests_total时间序列中满足标签instance为localhost:9090的时间序列，则可以使用如下表达式：

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

反之使用 `instance!="localhost:9090"` 则可以排除这些时间序列：

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

除了使用完全匹配的方式对时间序列进行过滤以外，PromQL还可以支持使用正则表达式作为匹配条件，多个表达式之间使用 `|` 进行分离：

- 使用 `label=~regx` 表示选择那些标签符合正则表达式定义的时间序列；
- 反之使用 `label!~regx` 进行排除；

例如，如果想查询多个环节下的时间序列序列可以使用如下表达式：

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

范围查询

直接通过类似于PromQL表达式 `http_requests_total` 查询时间序列时，返回值中只会包含该时间序列中的最新的一个样本值，这样的返回结果我们称之为**瞬时向量**。而相应的这样的表达式称之为**瞬时向量表达式**。

而如果我们想过去一段时间范围内的样本数据时，我们则需要使用**区间向量表达式**。区间向量表达式和瞬时向量表达式之间的差异在于在区间向量表达式中我们需要定义时间选择的范围，时间范围通过时间范围选择器 `[]` 进行定义。例如，通过以下表达式可以选择最近5分钟内的所有样本数据：

```
http_requests_total{}[5m]
```

该表达式将会返回查询到的时间序列中最近5分钟的所有样本数据：

```
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
]
```

通过区间向量表达式查询到的结果我们称为**区间向量**。

除了使用m表示分钟以外，PromQL的时间范围选择器支持其它时间单位：

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

时间位移操作

在瞬时向量表达式或者区间向量表达式中，都是以当前时间为基准：

```
http_request_total{} # 瞬时向量表达式，选择当前最新的数据
http_request_total{}[5m] # 区间向量表达式，选择以当前时间为基准，5分钟内的数据
```

而如果我们想查询，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**。

可以使用offset时间位移操作：

```
http_request_total{} offset 5m
http_request_total{}[1d] offset 1d
```

使用聚合操作

一般来说，如果描述样本特征的标签(label)在并非唯一的情况下，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)
```

标量和字符串

除了使用瞬时向量表达式和区间向量表达式以外，PromQL还直接支持用户使用标量(Scalar)和字符串(String)。

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。用户可以通过内置函数scalar()将单个瞬时向量转换为标量。

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`
```

合法的PromQL表达式

所有的PromQL表达式都必须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 `<metric name>{label=value}` 的形式以外，我们还可以使用内置的 `__name__` 标签来指定监控指标名称：

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

PromQL操作符

使用PromQL除了能够方便的按照查询和过滤时间序列以外，PromQL还支持丰富的操作符，用户可以使用这些操作符对进一步的对事件序列进行二次加工。这些操作符包括：数学运算符，逻辑运算符，布尔运算符等等。

数学运算

例如，我们可以通过指标node_memory_free_bytes_total获取当前主机可用的内存空间大小，其样本单位为Bytes。这是如果客户端要求使用MB作为单位响应数据，那只需要将查询到的时间序列的样本值进行单位换算即可：

```
node_memory_free_bytes_total / (1024 * 1024)
```

node_memory_free_bytes_total表达式会查询出所有满足表达式条件的时间序列，在上一小节中我们称该表达式为瞬时向量表达式，而返回的结果成为瞬时向量。

当瞬时向量与标量之间进行数学运算时，数学运算符会依次作用域瞬时向量中的每一个样本值，从而得到一组新的时间序列。

而如果是瞬时向量与瞬时向量之间进行数学运算时，过程会相对复杂一点。 例如，如果我们想根据node_disk_bytes_written和node_disk_bytes_read获取主机磁盘IO的总量，可以使用如下表达式：

```
node_disk_bytes_written + node_disk_bytes_read
```

那这个表达式是如何工作的呢？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。同时新的时间序列将不会包含指标名称。 该表达式返回结果的示例如下所示：

```
{device="sda",instance="localhost:9100",job="node_exporter"}=>1634967552@1518146427.807 + 864551424@1518146427.807
{device="sdb",instance="localhost:9100",job="node_exporter"}=>0@1518146427.807 + 1744384@1518146427.807
```

PromQL支持的所有数学运算符如下所示：

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

使用布尔运算过滤时间序列

在PromQL通过标签匹配模式，用户可以根据时间序列的特征维度对其进行查询。而布尔运算则支持用户根据时间序列中样本的值，对时间序列进行过滤。

例如，通过数学运算符我们可以很方便的计算出，当前所有主机节点的内存使用率：

```
(node_memory_bytes_total - node_memory_free_bytes_total) / node_memory_bytes_total
```

而系统管理员在排查问题的时候可能只想知道当前内存使用率超过95%的主机呢？通过使用布尔运算符可以方便的获取到该结果：

```
(node_memory_bytes_total - node_memory_free_bytes_total) / node_memory_bytes_total > 0.95
```

瞬时向量与标量进行布尔运算时，PromQL依次比较向量中的所有时间序列样本的值，如果比较结果为true则保留，反之丢弃。

瞬时向量与瞬时向量直接进行布尔运算时，同样遵循默认的匹配模式：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相应的操作，如果没找到匹配元素，则直接丢弃。

目前，Prometheus支持以下布尔运算符如下：

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

使用bool修饰符改变布尔运算符的行为

布尔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）。这时可以使用bool修饰符改变布尔运算的默认行为。 例如：

```
http_requests_total > bool 1000
```

使用bool修改符后，布尔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
```

同时需要注意的是，如果是在两个标量之间使用布尔运算，则必须使用bool修饰符

```
2 == bool 2 # 结果为1
```

使用集合运算符

使用瞬时向量表达式能够获取到一个包含多个时间序列的集合，我们称为瞬时向量。 通过集合运算，可以在两个瞬时向量与瞬时向量之间进行相应的集合操作。目前，Prometheus支持以下集合运算符：

- `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。

操作符优先级

对于复杂类型的表达式，需要了解运算操作的运行优先级

例如，查询主机的CPU使用率，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`

匹配模式详解

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ignoreing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。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。

PromQL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` (标准方差)
- `count` (计数)
- `count_values` (对value进行计数)
- `bottomk` (后n条时序)
- `topk` (前n条时序)
- `quantile` (分位数)

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 ≤ φ ≤ 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)
```

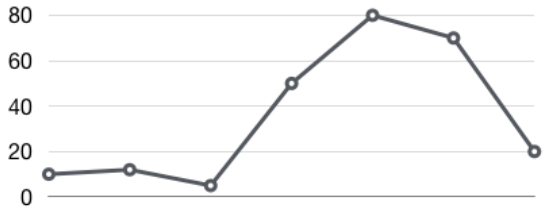
PromQL内置函数

在上一小节中，我们已经看到了类似于irate()这样的函数，可以帮助我们计算监控指标的增长率。除了irate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丰富的处理。本小节将带来读者了解一些常用的内置函数以及相关的使用场景和用法。

计算Counter指标增长率

我们知道Counter类型的监控指标其特点是只增不减，在没有发生重置（如服务器重启，应用重启）的情况下其样本值应该是不断增大的。为了能够更直观的表示样本数据的变化剧烈情况，需要计算样本的增长速率。

如下图所示，样本增长率反映出了样本变化的剧烈程度：



increase(v range-vector)函数是PromQL中提供的众多内置函数之一。其中参数v是一个区间向量，increase函数获取区间向量中的第一个后最后一个样本并返回其增长量。因此，可以通过以下表达式Counter类型指标的增长率：

```
increase(node_cpu[2m]) / 120
```

这里通过node_cpu[2m]获取时间序列最近两分钟的所有样本，increase计算出最近两分钟的增长量，最后除以时间120秒得到node_cpu样本在最近两分钟的平均增长率。并且这个值也近似于主机节点最近两分钟内的平均CPU使用率。

除了使用increase函数以外，PromQL中还直接内置了rate(v range-vector)函数，rate函数可以直接计算区间向量v在时间窗口内平均增长速率。因此，通过以下表达式可以得到与increase函数相同的结果：

```
rate(node_cpu[2m])
```

需要注意的是使用rate或者increase函数去计算样本的平均增长速率，容易陷入“长尾问题”当中，其无法反应在时间窗口内样本数据的突发变化。 例如，对于主机而言在2分钟的时间窗口内，可能在某一个由于访问量或者其它问题导致CPU占用100%的情况，但是通过计算在时间窗口内的平均增长率却无法反应出该问题。

为了解决该问题，PromQL提供了另外一个灵敏度更高的函数irate(v range-vector)。irate同样用于计算区间向量的计算率，但是其反应出的是瞬时增长率。irate函数是通过区间向量中最后两个样本数据来计算区间向量的增长速率。这种方式可以避免在时间窗口范围内的“长尾问题”，并且体现出更好的灵敏度，通过irate函数绘制的图标能够更好的反应样本数据的瞬时变化状态。

```
irate(node_cpu[2m])
```

irate函数相比于rate函数提供了更高的灵敏度，不过当需要分析长期趋势或者在告警规则中，irate的这种灵敏度反而容易造成干扰。因此在长期趋势分析或者告警中更推荐使用rate函数。

预测Gauge指标变化趋势

在一般情况下，系统管理员为了确保业务的持续可用运行，会针对服务器的资源设置相应的告警阈值。例如，当磁盘空间只剩512MB时向相关人员发送告警通知。 这种基于阈值的告警模式对于当资源用量是平滑增长的情况下是能够有效的工作的。 但是如果资源不是平滑变化的呢？ 比如有些某些业务增长，存储空间的增长速率提升了高几倍。这时，如果基于原有阈值去触发告警，当系统管理员接收到告警以后可能还没来得及处理问题，系统就已经不可用了。 因此阈值通常来说不是固定的，需要定期进行调整才能保证该告警阈值能够发挥去作用。 那么还有没有更好的方法吗？

PromQL中内置的predict_linear(v range-vector, t scalar) 函数可以帮助系统管理员更好的处理此类情况，predict_linear函数可以预测时间序列v在t秒后的值。它基于简单线性回归的方式，对时间窗口内的样本数据进行统计，从而可以对时间序列的变化趋势做出预测。例如，基于2小时的样本数据，来预测主机可用磁盘空间的是否在4个小时候被占满，可以使用如下表达式：

```
predict_linear(node_filesystem_free{job="node"}[2h], 4 * 3600) < 0
```

统计Histogram指标的分位数

在本章的第2小节中，我们介绍了Prometheus的四种监控指标类型，其中Histogram和Summary都可以用于统计和分析数据的分布情况。区别在于Summary是直接在客户端计算了数据分布的分位数情况。而Histogram的分位数计算需要通过histogram_quantile(φ float, b instant-vector)函数进行计算。其中φ（0<φ<1）表示需要计算的分位数，如果需要计算中位数φ取值为0.5，以此类推即可。

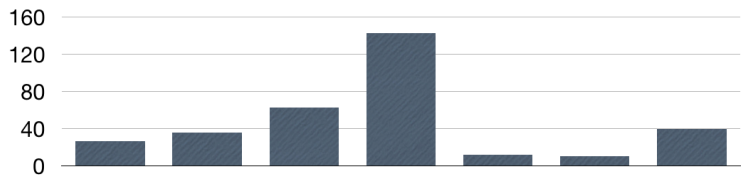
以指标http_request_duration_seconds_bucket为例：

```
# HELP http_request_duration_seconds request duration histogram
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.5"} 0
http_request_duration_seconds_bucket{le="1"} 1
http_request_duration_seconds_bucket{le="2"} 2
http_request_duration_seconds_bucket{le="3"} 3
http_request_duration_seconds_bucket{le="5"} 3
http_request_duration_seconds_bucket{le="+Inf"} 3
http_request_duration_seconds_sum 6
http_request_duration_seconds_count 3
```

当计算9分位数时，使用如下表达式：

```
histogram_quantile(0.5, http_request_duration_seconds_bucket)
```

通过对Histogram类型的监控指标，用户可以轻松获取样本数据的分布情况。同时分位数的计算，也可以非常方便的用于评判当前监控指标的服务水平。



需要注意的是通过histogram_quantile计算的分位数，并非为精确值，而是通过http_request_duration_seconds_bucket和http_request_duration_seconds_sum近似计算的结果。

动态标签替换

一般来说来说，使用PromQL查询到时间序列后，可视化工具会根据时间序列的标签来渲染图表。例如通过up指标可以获取到当前所有运行的Exporter实例以及其状态：

```
up{instance="localhost:8080",job="cadvisor"}    1
up{instance="localhost:9090",job="prometheus"}    1
up{instance="localhost:9100",job="node"}    1
```

这是可视化工具渲染图标时可能根据，instance和job的值进行渲染，为了能够让客户端的图标更具有可读性，可以通过label_replace标签为时间序列添加额外的标签。label_replace的具体参数如下：

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

该函数会依次对v中的每一条时间序列进行处理，通过regex匹配src_label的值，并将匹配部分relacement写入到dst_label标签中。如下所示：

```
label_replace(up, "host", "$1", "instance",  "(.*):.*")
```

函数处理后，时间序列将包含一个host标签，host标签的值为Exporter实例的IP地址：

```
up{host="localhost",instance="localhost:8080",job="cadvisor"}    1
up{host="localhost",instance="localhost:9090",job="prometheus"}    1
up{host="localhost",instance="localhost:9100",job="node"} 1
```

除了label_replace以外，Prometheus还提供了label_join函数，该函数可以将时间序列中v多个标签src_label的值，通过separator作为连接符写入到一个新的标签dst_label中:

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

label_replace和label_join函数提供了对时间序列标签的自定义能力，从而能够更好的于客户端或者可视化工具配合。

其它内置函数

除了上文介绍的这些内置函数以外，PromQL还提供了大量的其它内置函数。这些内置函数包括一些常用的数学计算、日期等等。这里就不一一细讲，感兴趣的读者可以通过阅读Prometheus的官方文档，了解这些函数的使用方式。

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以下几种不同的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 Unavailiable 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=<rfc3339 | unix_timestamp>：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>
}
```

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=<rfc3339 | unix_timestamp>: 起始时间。
- end=<rfc3339 | unix_timestamp>: 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表达式的结果。

```
$ 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" ]
        ]
      }
    ]
  }
}
```

最佳实践：4个黄金指标和USE方法

前面部分介绍了Prometheus的数据存储模型以及4种指标类型，同时Prometheus提供的强大的PromQL可以实现对数据的个性化处理。Promthues基于指标提供了一个通用的监控解决方案。这里先思考一个基本的问题，在实现监控时，我们到底应该监控哪些对象以及哪些指标？

监控所有

在之前**Prometheus简介**部分介绍监控的基本目标，首先是及时发现问题其次是要能够快速对问题进行定位。对于传统监控解决方案而言，用户看到的依然是一个黑盒，用户无法真正了解系统的真正的运行状态。因此Prometheus鼓励用户监控所有的东西。下面列举一些常用的监控维度。

级别	监控什么	Exporter
网络	网络协议：http、dns、tcp、icmp； 网络硬件：路由器，交换机等	BlackBox Exporter;SNMP Exporter
主机	资源用量	node exporter
容器	资源用量	cAdvisor
应用(包括Library)	延迟，错误，QPS，内部状态等	代码中集成Prmometheus Client
中间件状态	资源用量，以及服务状态	代码中集成Prmometheus Client
编排工具	集群资源用量，调度等	Kubernetes Components

监控模式

除了上述介绍的不同监控级别以外。实际上根据不同的系统类型和目标，这里还有一些通用的套路和模式可以使用。

4个黄金指标

Four Golden Signals是Google针对大量分布式监控的经验总结，4个黄金指标可以在服务级别帮助衡量终端用户体验、服务中断、业务影响等层面的问题。主要关注与以下四种类型的指标：延迟，通讯量，错误以及饱和度：

- 延迟：服务请求所需时间。

记录用户所有请求所需的时间，重点是要区分成功请求的延迟时间和失败请求的延迟时间。 例如在数据库或者其他关键祸端服务异常触发HTTP 500的情况下，用户也可能会很快得到请求失败的响应内容，如果不加区分计算这些请求的延迟，可能导致计算结果与实际结果产生巨大的差异。除此以外，在微服务中通常提倡“快速失败”，开发人员需要特别注意这些延迟较大的错误，因为这些缓慢的错误会明显影响系统的性能，因此追踪这些错误的延迟也是非常重要的。

- 通讯量：监控当前系统的流量，用于衡量服务的容量需求。

流量对于不同类型的系统而言可能代表不同的含义。例如，在HTTP REST API中, 流量通常是每秒HTTP请求数；

- 错误：监控当前系统所有发生的错误请求，衡量当前系统错误发生的速率。

对于失败而言有些是显式的(比如, HTTP 500错误)，而有些是隐式(比如，HTTP响应200，但实际业务流程依然是失败的)。

对于一些显式的错误如HTTP 500可以通过在负载均衡器(如Nginx)上进行捕获，而对于一些系统内部的异常，则可能需要直接从服务中添加钩子统计并进行获取。

- 饱和度：衡量当前服务的饱和度。

主要强调最能影响服务状态的受限制的资源。 例如，如果系统主要受内存影响，那就主要关注系统的内存状态，如果系统主要受限与磁盘I/O，那就主要观测磁盘I/O的状态。因为通常情况下，当这些资源达到饱和后，服务的性能会明显下降。同时还可以利用饱和度对系统做出预测，比如，“磁盘是否可能在4个小时候就满了”。

RED方法

RED方法是Weave Cloud在基于Google的“4个黄金指标”的原则下结合Prometheus以及Kubernetes容器实践，细化和总结的方法论，特别适合于云原生应用以及微服务架构应用的监控和度量。主要关注以下三种关键指标：

- (请求)速率：服务每秒接收的请求数。
- (请求)错误：每秒失败的请求数。
- (请求)耗时：每个请求的耗时。

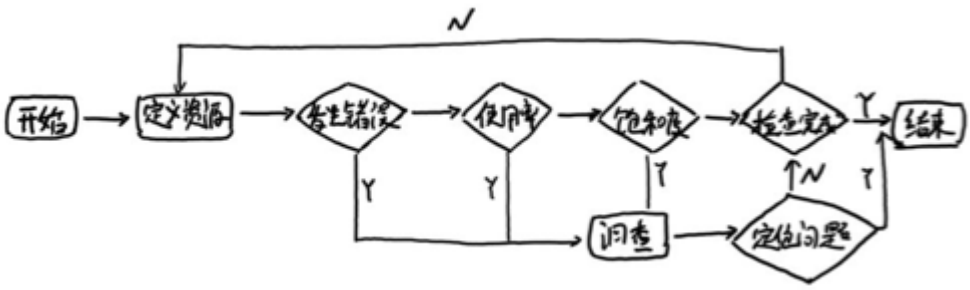
在“4大黄金信号”的原则下，RED方法可以有效的帮助用户衡量云原生以及微服务应用下的用户体验问题。

USE方法

USE方法全称”Utilization Saturation and Errors Method”，主要用于分析系统性能问题，可以指导用户快速识别资源瓶颈以及错误的方法。正如USE方法的名字所表示的含义，USE方法主要关注与资源的：使用率(Utilization)、饱和度(Saturation)以及错误(Errors)。

- 使用率：关注系统资源的使用情况。 这里的资源主要包括但不限于：CPU，内存，网络，磁盘等等。100%的使用率通常是系统性能瓶颈的标志。
- 饱和度：例如CPU的平均运行排队长度，这里主要是针对资源的饱和度(注意，不同于4大黄金信号)。任何资源在某种程度上的饱和都可能导致系统性能的下降。
- 错误：错误计数。例如：“网卡在数据包传输过程中检测到的以太网网络冲突了14次”。

通过对资源以上指标持续观察，通过以下流程可以知道用户识别资源瓶颈：



小结

PromQL是Prometheus的标准查询语句，通过强大的数据统计能力，使得将监控指标与实际业务进行关联成为可能。同时通过内置的预测函数，能够帮助用户将传统的面向结果转变为面向预测的方式。从而更有效的为业务和系统的正常运行保驾护航。

Prometheus告警处理

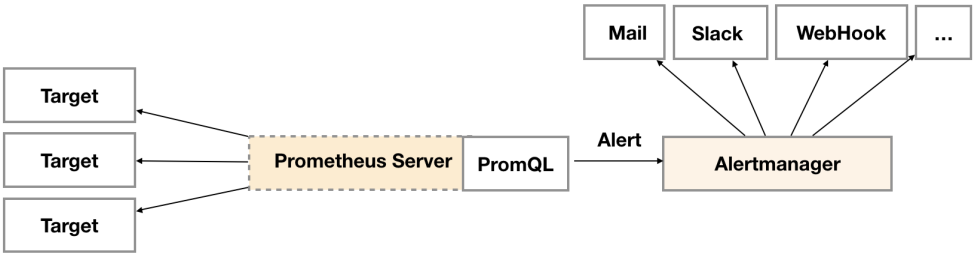
本章我们将带领读者探索Prometheus的告警处理机制，在前面的部分中已经介绍了告警能力在Prometheus的架构中被划分为两个部分，在Prometheus Server中定义告警规则以及产生告警，Alertmanager组件则用于处理这些由Prometheus产生的告警。Alertmanager即Prometheus体系中告警的统一处理中心。Alertmanager提供了多种内置第三方告警通知方式，同时还提供了对Webhook通知的支持，通过Webhook用户可以完成对告警更多个性化的扩展。

本章主要内容：

- 在Prometheus中自定义告警规则
- 理解Alertmanager特性
- 基于标签的动态告警处理
- 将告警通知发送到第三方服务
- 如何使用Webhook扩展Alertmanager
- 以及一些其他的性能优化模式

Prometheus告警简介

告警能力在Prometheus的架构中被划分成两个独立的部分。如下所示，通过在Prometheus中定义AlertRule（告警规则），Prometheus会周期性的对告警规则进行计算，如果满足告警触发条件就会向Alertmanager发送告警信息。



在Prometheus中一条告警规则主要由以下几部分组成：

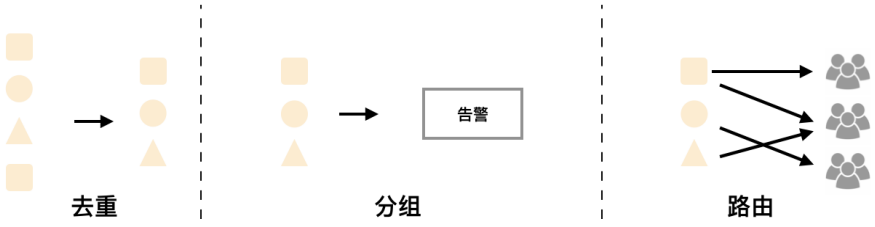
- 告警名称：用户需要为告警规则命名，当然对于命名而言，需要能够直接表达出该告警的主要内容
- 告警规则：告警规则实际上主要由PromQL进行定义，其实际意义是当表达式（PromQL）查询结果持续多长时间（During）后出发告警

在Prometheus中，还可以通过Group（告警组）对一组相关的告警进行统一定义。当然这些定义都是通过YAML文件来统一管理的。

Alertmanager作为一个独立的组件，负责接收并处理来自Prometheus Server(也可以是其它的客户端程序)的告警信息。Alertmanager可以对这些告警信息进行进一步的处理，比如当接收到大量重复告警时能够消除重复的告警信息，同时对告警信息进行分组并且路由到正确的通知方，Prometheus内置了对邮件，Slack等多种通知方式的支持，同时还支持与Webhook的集成，以支持更多定制化的场景。例如，目前Alertmanager还不支持钉钉，那用户完全可以通过Webhook与钉钉机器人进行集成，从而通过钉钉接收告警信息。同时AlertManager还提供了静默和告警抑制机制来对告警通知行为进行优化。

Alertmanager特性

Alertmanager除了提供基本的告警通知能力以外，还主要提供了如：分组、抑制以及静默等告警特性：



分组

分组机制可以将详细的告警信息合并成一个通知。在某些情况下，比如由于系统宕机导致大量的告警被同时触发，在这种情况下分组机制可以将这些被触发的告警合并为一个告警通知，避免一次性接受大量的告警通知，而无法对问题进行快速定位。

例如，当集群中有数百个正在运行的服务实例，并且为每一个实例设置了告警规则。假如此时发生了网络故障，可能导致大量的服务实例无法连接到数据库，结果就会有数百个告警被发送到Alertmanager。

而作为用户，可能只希望能够在一个通知中中就能查看哪些服务实例收到影响。这时可以按照服务所在集群或者告警名称对告警进行分组，而将这些告警内聚在一起成为一个通知。

告警分组，告警时间，以及告警的接受方式可以通过Alertmanager的配置文件进行配置。

抑制

抑制是指当某一告警发出后，可以停止重复发送由此告警引发的其它告警的机制。

例如，当集群不可访问时触发了一次告警，通过配置Alertmanager可以忽略与该集群有关的其它所有告警。这样可以避免接收到大量与实际问题无关的告警通知。

抑制机制同样通过Alertmanager的配置文件进行设置。

静默

静默提供了一个简单的机制可以快速根据标签对告警进行静默处理。如果接收到的告警符合静默的配置，Alertmanager则不会发送告警通知。

静默设置需要在Alertmanager的Werb页面上进行设置。

自定义Prometheus告警规则

Prometheus中的告警规则允许你基于PromQL表达式定义告警触发条件，Prometheus后端对这些触发规则进行周期性计算，当满足触发条件后则会触发告警通知。默认情况下，用户可以通过Prometheus的Web界面查看这些告警规则以及告警的触发状态。当Promthues与Alertmanager关联之后，可以将告警发送到外部服务如Alertmanager中并通过Alertmanager可以对这些告警进行进一步的处理。

定义告警规则

一条典型的告警规则如下所示：

```
groups:
- name: example
  rules:
  - alert: HighErrorRate
    expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
    for: 10m
    labels:
      severity: page
    annotations:
      summary: High request latency
      description: description info
```

在告警规则文件中，我们可以将一组相关的规则设置定义在一个group下。在每一个group中我们可以定义多个告警规则(rule)。一条告警规则主要由以下几部分组成：

- alert：告警规则的名称。
- expr：基于PromQL表达式告警触发条件，用于计算是否有时间序列满足该条件。
- for：评估等待时间，可选参数。用于表示只有当触发条件持续一段时间后才发送告警。在等待期间新产生告警的状态为pending。
- labels：自定义标签，允许用户指定要附加到告警上的一组附加标签。
- annotations：用于指定一组附加信息，比如用于描述告警详细信息的文字等，annotations的内容在告警产生时会一同作为参数发送到Alertmanager。

为了能够让Prometheus能够启用定义的告警规则，我们需要在Prometheus全局配置文件中通过**rule_files**指定一组告警规则文件的访问路径，Prometheus启动后会自动扫描这些路径下规则文件中定义的内容，并且根据这些规则计算是否向外部发送通知：

```
rule_files:
  [ - <filepath_glob> ... ]
```

默认情况下Prometheus会每分钟对这些告警规则进行计算，如果用户想定义自己的告警计算周期，则可以通过 `evaluation_interval` 来覆盖默认的计算周期：

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

模板化

一般来说，在告警规则文件的annotations中使用 `summary` 描述告警的概要信息， `description` 用于描述告警的详细信息。同时Alertmanager的UI也会根据这两个标签值，显示告警信息。为了让告警信息具有更好的可读性，Prometheus支

持模板化label和annotations的中标签的值。

通过 `$labels.<labelname>` 变量可以访问当前告警实例中指定标签的值。$value则可以获取当前PromQL表达式计算的样本值。

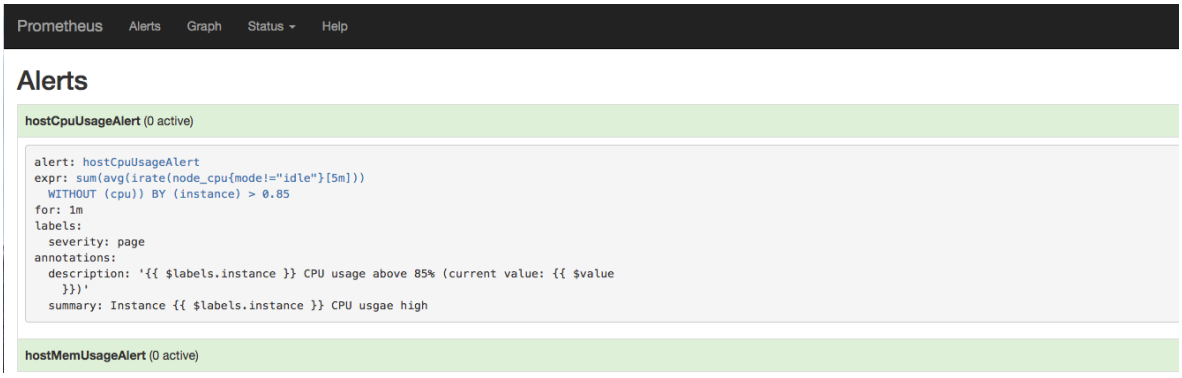
```
# To insert a firing element's label values:
{{ $labels.<labelname> }}
# To insert the numeric expression value of the firing element:
{{ $value }}
```

例如，可以通过模板化优化summary以及description的内容的可读性：

```
groups:
- name: example
  rules:

  # Alert for any instance that is unreachable for >5 minutes.
  - alert: InstanceDown
    expr: 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 has a median request latency >1s.
  - alert: APIHighRequestLatency
    expr: api_http_request_latencies_second{quantile="0.5"} > 1
    for: 10m
    annotations:
      summary: "High request latency on {{ $labels.instance }}"
      description: "{{ $labels.instance }} has a median request latency above 1s (current value: {{ $value }}
s)"
```

查看告警状态

如下所示，用户可以通过Prometheus WEB界面中的Alerts菜单查看当前Prometheus下的所有告警规则，以及其当前所处的活动状态。



同时对于已经pending或者firing的告警，Prometheus也会将它们存储到时间序列ALERTS{}中。

可以通过表达式，查询告警实例：

```
ALERTS{alertname="<alert name>", alertstate="pending|firing", <additional alert labels>}
```

样本值为1表示当前告警处于活动状态（pending或者firing），当告警从活动状态转换为非活动状态时，样本值则为0。

实例：定义主机监控告警

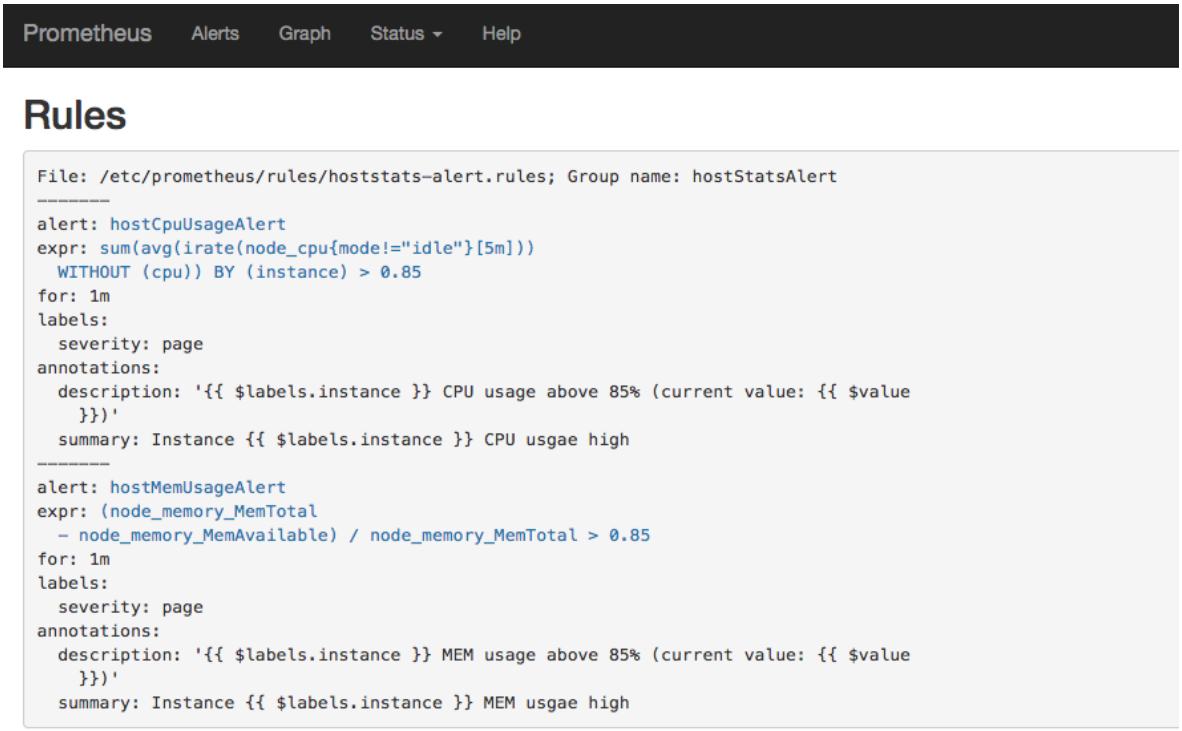
修改Prometheus配置文件prometheus.yml,添加以下配置：

```
rule_files:
  - /etc/prometheus/rules/*.rules
```

在目录/etc/prometheus/rules/下创建告警文件hoststats-alert.rules内容如下：

```
groups:
- name: hostStatsAlert
  rules:
  - alert: hostCpuUsageAlert
    expr: sum(avg without (cpu)(irate(node_cpu{mode!='idle'}[5m]))) by (instance) > 0.85
    for: 1m
    labels:
      severity: page
    annotations:
      summary: "Instance {{ $labels.instance }} CPU usgae high"
      description: "{{ $labels.instance }} CPU usage above 85% (current value: {{ $value }})"
  - alert: hostMemUsageAlert
    expr: (node_memory_MemTotal - node_memory_MemAvailable)/node_memory_MemTotal > 0.85
    for: 1m
    labels:
      severity: page
    annotations:
      summary: "Instance {{ $labels.instance }} MEM usgae high"
      description: "{{ $labels.instance }} MEM usage above 85% (current value: {{ $value }})"
```

重启Prometheus后访问Prometheus UIhttp://127.0.0.1:9090/rules可以查看当前以加载的规则文件。

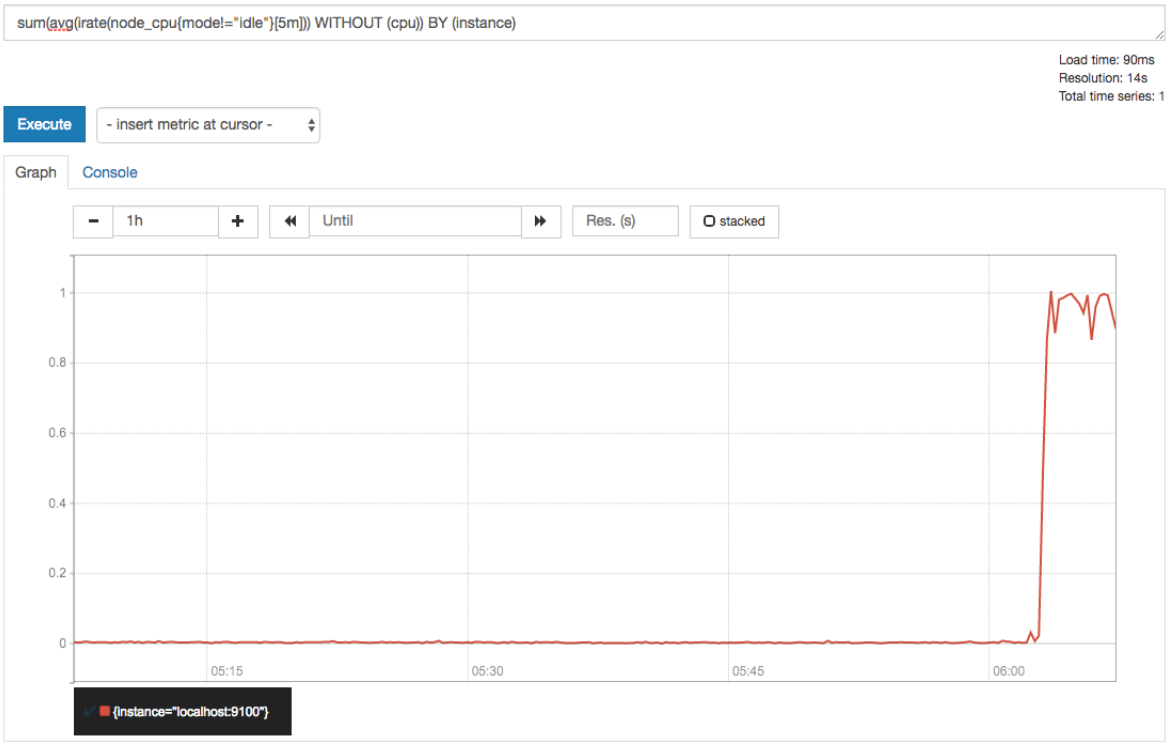


切换到Alerts标签http://127.0.0.1:9090/alerts可以查看当前告警的活动状态。

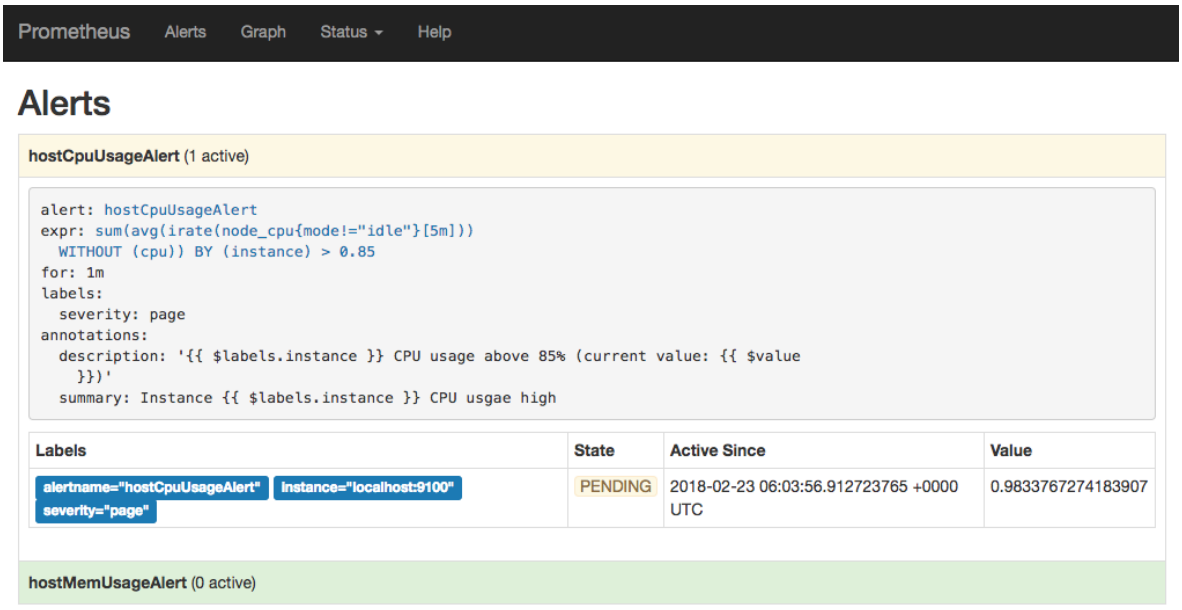
此时，我们可以手动拉高系统的CPU使用率，验证Prometheus的告警流程，在主机上运行以下命令：

```
cat /dev/zero>/dev/null
```

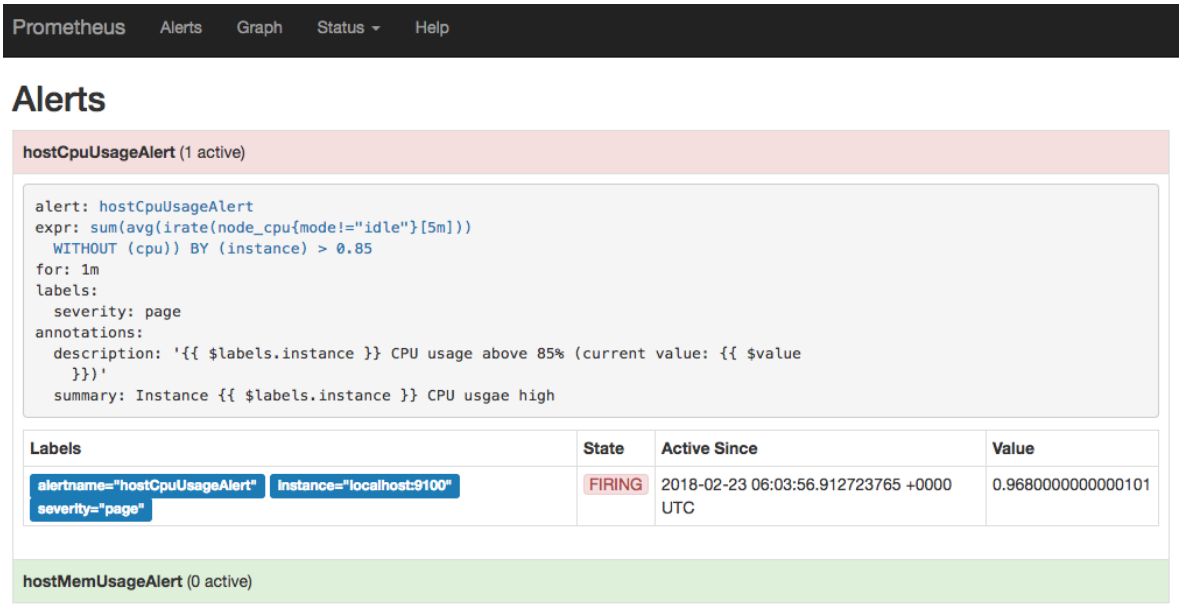
运行命令后查看CPU使用率情况，如下图所示：



Prometheus首次检测到满足触发条件后，hostCpuUsageAlert显示由一条告警处于活动状态。由于告警规则中设置了1m的等待时间，当前告警状态为PENDING，如下图所示：



如果1分钟后告警条件持续满足，则会实际触发告警并且告警状态为FIRING，如下图所示：



接下来

在这一小节中介绍了如何配置和使用Prometheus提供的告警能力，并且尝试实现了对主机CPU以及内存的告警规则设置。目前为止，我们只能通过Prometheus UI查看当前告警的活动状态。接下来，接下来我们将尝试利用Prometheus体系中的另一个组件Alertmanager对这些触发的告警进行处理，实现告警通知。

部署AlertManager

Alertmanager和Prometheus Server一样均采用Golang实现，并且没有第三方依赖。一般来说我们可以通过以下几种方式来部署Alertmanager：二进制包、容器以及源码方式安装。

使用二进制包部署AlertManager

获取并安装软件包

Alertmanager最新版本的下载地址可以从Prometheus官方网站https://prometheus.io/download/获取。

```
export VERSION=0.15.2
curl -LO https://github.com/prometheus/alertmanager/releases/download/v$VERSION/alertmanager-$VERSION.darwin-amd64.tar.gz
tar xvf alertmanager-$VERSION.darwin-amd64.tar.gz
```

创建alertmanager配置文件

Alertmanager解压后会包含一个默认的alertmanager.yml配置文件，内容如下所示：

```
global:
  resolve_timeout: 5m

route:
  group_by: ['alertname']
  group_wait: 10s
  group_interval: 10s
  repeat_interval: 1h
  receiver: 'web.hook'
receivers:
- name: 'web.hook'
  webhook_configs:
  - url: 'http://127.0.0.1:5001/'
inhibit_rules:
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['alertname', 'dev', 'instance']
```

Alertmanager的配置主要包含两个部分：路由(route)以及接收器(receivers)。所有的告警信息都会从配置中的顶级路由(route)进入路由树，根据路由规则将告警信息发送给相应的接收器。

在Alertmanager中可以定义一组接收器，比如可以按照角色(比如系统运维，数据库管理员)来划分多个接收器。接收器可以关联邮件，Slack以及其它方式接收告警信息。

当前配置文件中定义了一个默认的接收者default-receiver由于这里没有设置接收方式，目前只相当于一个占位符。关于接收器的详细介绍会在后续章节介绍。

在配置文件中使用route定义了顶级的路由，路由是一个基于标签匹配规则的树状结构。所有的告警信息从顶级路由开始，根据标签匹配规则进入到不同的子路由，并且根据子路由设置的接收器发送告警。目前配置文件中只设置了一个顶级路由route并且定义的接收器为default-receiver。因此，所有的告警都会发送给default-receiver。关于路由的详细内容会在后续进行详细介绍。

启动Alertmanager

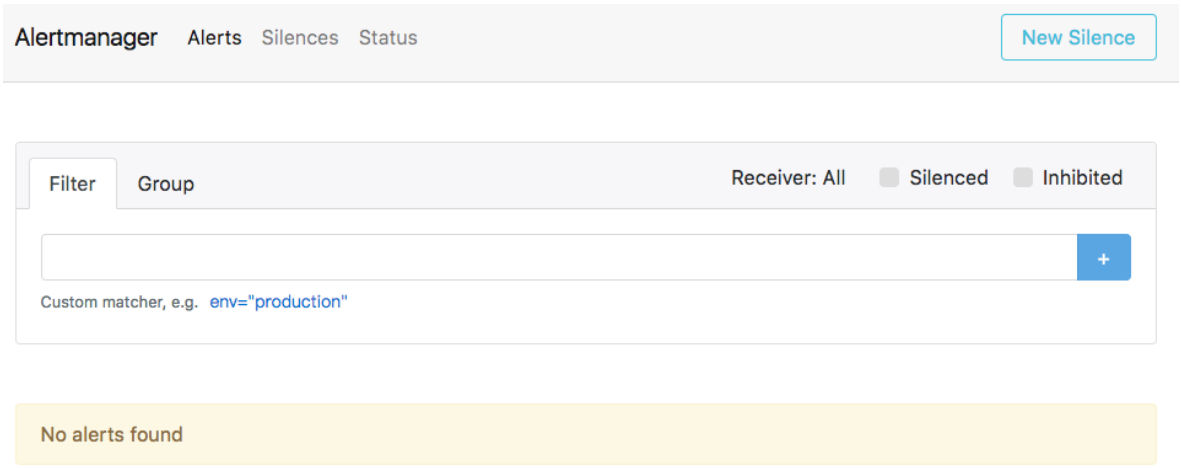
Alermanager会将数据保存到本地中，默认的存储路径为 `data/` 。因此，在启动Alertmanager之前需要创建相应的目录：

```
./alertmanager
```

用户也在启动Alertmanager时使用参数修改相关配置。 `--config.file` 用于指定alertmanager配置文件路径， `--storage.path` 用于指定数据存储路径。

查看运行状态

Alertmanager启动后可以通过9093端口访问，http://192.168.33.10:9093



Alert菜单下可以查看Alertmanager接收到的告警内容。Silences菜单下则可以通过UI创建静默规则，这部分我们会在后续部分介绍。进入Status菜单，可以看到当前系统的运行状态以及配置信息。

关联Prometheus与Alertmanager

在Prometheus的架构中被划分成两个独立的部分。Prometheus负责产生告警，而Alertmanager负责告警产生后的后续处理。因此Alertmanager部署完成后，需要在Prometheus中设置Alertmanager相关的信息。

编辑Prometheus配置文件prometheus.yml,并添加以下内容

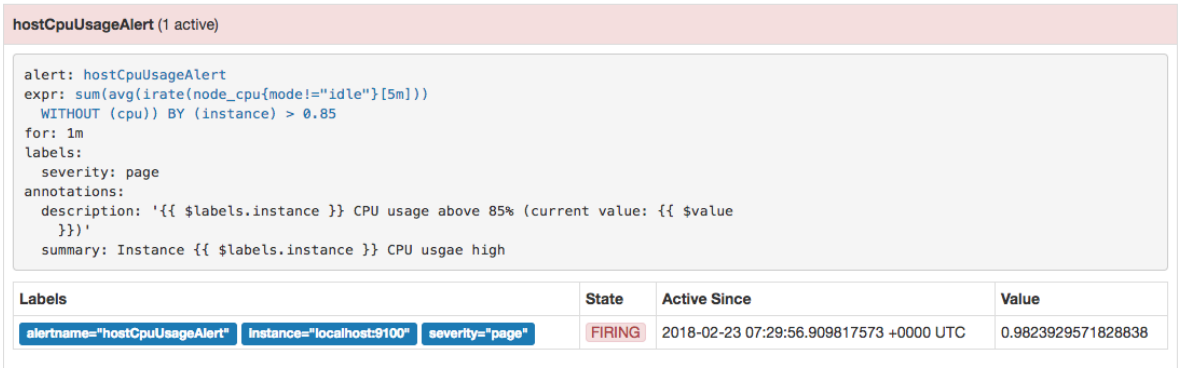
```
alerting:
  alertmanagers:
    - static_configs:
        - targets: ['localhost:9093']
```

重启Prometheus服务，成功后，可以从http://192.168.33.10:9090/config查看alerting配置是否生效。

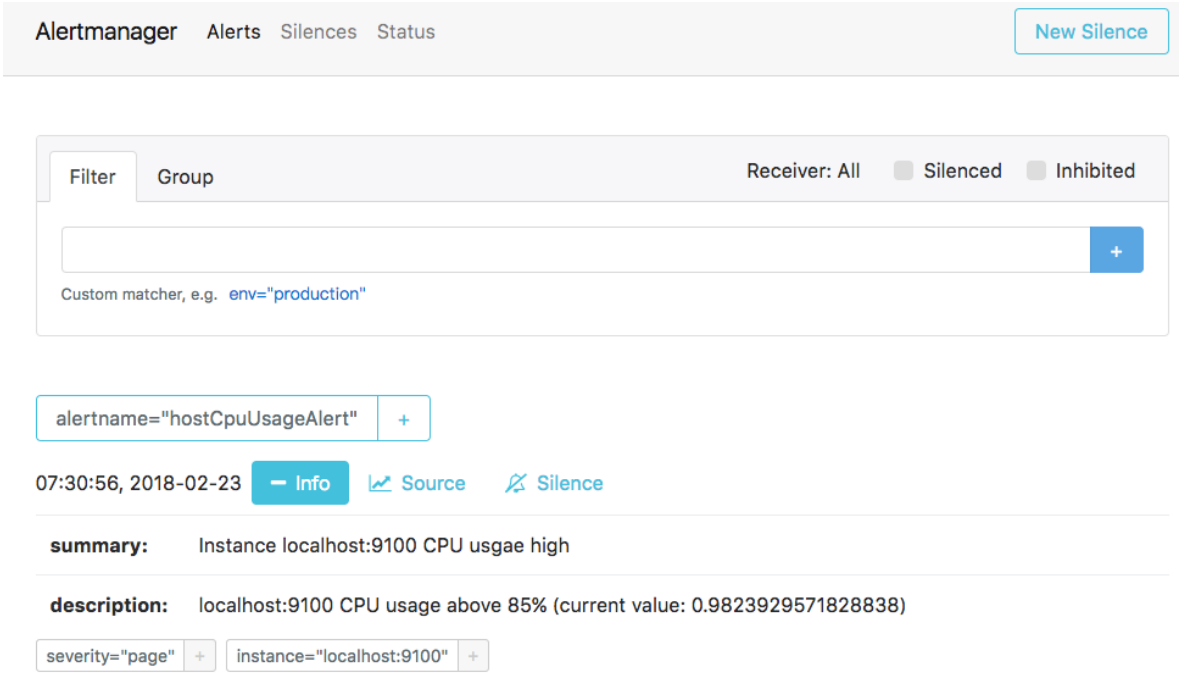
此时，再次尝试手动拉高系统CPU使用率：

```
cat /dev/zero>/dev/null
```

等待Prometheus告警进行触发状态：



查看Alertmanager UI此时可以看到Alertmanager接收到的告警信息。



接下来

目前为止，我们已经成功安装部署了Alertmanager并且与Prometheus关联，能够正常接收来自Prometheus的告警信息。接下来我们将详细介绍Alertmanager是如何处理这些接收到的告警信息的。

Alertmanager配置概述

在上面的部分中已经简单介绍过，在Alertmanager中通过路由(Route)来定义告警的处理方式。路由是一个基于标签匹配的树状匹配结构。根据接收到告警的标签匹配相应的处理方式。这里将详细介绍路由相关的内容。

Alertmanager主要负责对Prometheus产生的告警进行统一处理，因此在Alertmanager配置中一般会包含以下几个主要部分：

- 全局配置（global）：用于定义一些全局的公共参数，如全局的SMTP配置，Slack配置等内容；
- 模板（templates）：用于定义告警通知时的模板，如HTML模板，邮件模板等；
- 告警路由（route）：根据标签匹配，确定当前告警应该如何处理；
- 接收人（receivers）：接收人是一个抽象的概念，它可以是一个邮箱也可以是微信，Slack或者Webhook等，接收人一般配合告警路由使用；
- 抑制规则（inhibit_rules）：合理设置抑制规则可以减少垃圾告警的产生

其完整配置格式如下：

```
global:
  [ resolve_timeout: <duration> | default = 5m ]
  [ smtp_from: <tmpl_string> ]
  [ smtp_smarthost: <string> ]
  [ smtp_hello: <string> | default = "localhost" ]
  [ smtp_auth_username: <string> ]
  [ smtp_auth_password: <secret> ]
  [ smtp_auth_identity: <string> ]
  [ smtp_auth_secret: <secret> ]
  [ smtp_require_tls: <bool> | default = true ]
  [ slack_api_url: <secret> ]
  [ victorops_api_key: <secret> ]
  [ victorops_api_url: <string> | default = "https://alert.victorops.com/integrations/generic/20131114/alert/" ]
  [ pagerduty_url: <string> | default = "https://events.pagerduty.com/v2/enqueue" ]
  [ opsgenie_api_key: <secret> ]
  [ opsgenie_api_url: <string> | default = "https://api.opsgenie.com/" ]
  [ hipchat_api_url: <string> | default = "https://api.hipchat.com/" ]
  [ hipchat_auth_token: <secret> ]
  [ wechat_api_url: <string> | default = "https://qyapi.weixin.qq.com/cgi-bin/" ]
  [ wechat_api_secret: <secret> ]
  [ wechat_api_corp_id: <string> ]
  [ http_config: <http_config> ]

templates:
  [ - <filepath> ... ]

route: <route>

receivers:
  - <receiver> ...

inhibit_rules:
  [ - <inhibit_rule> ... ]
```

在全局配置中需要注意的是 `resolve_timeout` ，该参数定义了当Alertmanager持续多长时间未接收到告警后标记告警状态为resolved（已解决）。该参数的定义可能会影响到告警恢复通知的接收时间，读者可根据自己的实际场景进行定义，其默认值为5分钟。在接下来的部分，我们将已一些实际的例子解释Alertmanager的其它配置内容。

基于标签的告警处理路由

在Alertmanager的配置中会定义一个基于标签匹配规则的告警路由树，以确定在接收到告警后Alertmanager需要如何对其进行处理：

```
route: <route>
```

其中route中则主要定义了告警的路由匹配规则，以及Alertmanager需要将匹配到的告警发送给哪一个receiver，一个最简单的route定义如下所示：

```
route:
  group_by: ['alertname']
  receiver: 'web.hook'
receivers:
- name: 'web.hook'
  webhook_configs:
  - url: 'http://127.0.0.1:5001/'
```

如上所示：在Alertmanager配置文件中，我们只定义了一个路由，那就意味着所有由Prometheus产生的告警在发送到Alertmanager之后都会通过名为 `web.hook` 的receiver接收。这里的web.hook定义为一个webhook地址。当然实际场景下，告警处理可不是这么简单的一件事情，对于不同级别的告警，我们可能会有完全不同的处理方式，因此在route中，我们还可以定义更多的子Route，这些Route通过标签匹配告警的处理方式，route的完整定义如下：

```
[ receiver: <string> ]
[ group_by: '[' <labelname>, ... ']' ]
[ continue: <boolean> | default = false ]

match:
  [ <labelname>: <labelvalue>, ... ]

match_re:
  [ <labelname>: <regex>, ... ]

[ group_wait: <duration> | default = 30s ]
[ group_interval: <duration> | default = 5m ]
[ repeat_interval: <duration> | default = 4h ]

routes:
  [ - <route> ... ]
```

路由匹配

每一个告警都会从配置文件中顶级的route进入路由树，需要注意的是顶级的route必须匹配所有告警(即不能有任何的匹配设置match和match_re)，每一个路由都可以定义自己的接受人以及匹配规则。默认情况下，告警进入到顶级route后会遍历所有的子节点，直到找到最深的匹配route，并将告警发送到该route定义的receiver中。但如果route中设置**continue**的值为false，那么告警在匹配到第一个子节点之后就直接停止。如果**continue**为true，报警则会继续进行后续子节点的匹配。如果当前告警匹配不到任何的子节点，那该告警将会基于当前路由节点的接收器配置方式进行处理。

其中告警的匹配有两种方式可以选择。一种方式基于字符串验证，通过设置**match**规则判断当前告警中是否存在标签labelname并且其值等于labelvalue。第二种方式则基于正则表达式，通过设置**match_re**验证当前告警标签的值是否满足正则表达式的内容。

如果警报已经成功发送通知，如果想设置发送告警通知之前要等待时间，则可以通过**repeat_interval**参数进行设置。

告警分组

在之前的部分有讲过，Alertmanager可以对告警通知进行分组，将多条告警合合并为一个通知。这里我们可以使用**group_by**来定义分组规则。基于告警中包含的标签，如果满足**group_by**中定义标签名称，那么这些告警将会合并为一个通知发送给接收器。

有的时候为了能够一次性收集和发送更多的相关信息时，可以通过**group_wait**参数设置等待时间，如果在等待时间内当前group接收到了新的告警，这些告警将会合并为一个通知向receiver发送。

而**group_interval**配置，则用于定义相同的Group之间发送告警通知的时间间隔。

例如，当使用Prometheus监控多个集群以及部署在集群中的应用和数据库服务，并且定义以下的告警处理路由规则来对集群中的异常进行通知。

```
route:
  receiver: 'default-receiver'
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  group_by: [cluster, alertname]
  routes:
  - receiver: 'database-pager'
    group_wait: 10s
    match_re:
      service: mysql|cassandra
  - receiver: 'frontend-pager'
    group_by: [product, environment]
    match:
      team: frontend
```

默认情况下所有的告警都会发送给集群管理员default-receiver，因此在Alertmanager的配置文件的根路由中，对告警信息按照集群以及告警的名称对告警进行分组。

如果告警时来源于数据库服务如MySQL或者Cassandra，此时则需要将告警发送给相应的数据库管理员(database-pager)。这里定义了一个单独子路由，如果告警中包含service标签，并且service为MySQL或者Cassandra,则向database-pager发送告警通知，由于这里没有定义group_by等属性，这些属性的配置信息将从上级路由继承，database-pager将会接收到按cluster和alertname进行分组的告警通知。

而某些告警规则来源可能来源于开发团队的定义，这些告警中通过添加标签team来标示这些告警的创建者。在Alertmanager配置文件的告警路由下，定义单独子路由用于处理这一类的告警通知，如果匹配到告警中包含标签team，并且team的值为frontend，Alertmanager将会按照标签product和environment对告警进行分组。此时如果应用出现异常，开发团队就能清楚的知道哪一个环境(environment)中的哪一个应用程序出现了问题，可以快速对应用进行问题定位。

使用Receiver接收告警信息

前上一小节已经讲过，在Alertmanager中路由负责对告警信息进行分组匹配，并将像告警接收器发送通知。告警接收器可以通过以下形式进行配置：

```
receivers:
  - <receiver> ...
```

每一个receiver具有一个全局唯一的名称，并且对应一个或者多个通知方式：

```
name: <string>
email_configs:
  [ - <email_config>, ... ]
hipchat_configs:
  [ - <hipchat_config>, ... ]
pagerduty_configs:
  [ - <pagerduty_config>, ... ]
pushover_configs:
  [ - <pushover_config>, ... ]
slack_configs:
  [ - <slack_config>, ... ]
opsgenie_configs:
  [ - <opsgenie_config>, ... ]
webhook_configs:
  [ - <webhook_config>, ... ]
victorops_configs:
  [ - <victorops_config>, ... ]
```

目前官方内置的第三方通知集成包括：邮件、 即时通讯软件（如Slack、Hipchat）、移动应用消息推送(如Pushover)和自动化运维工具（例如：Pagerduty、Opsgenie、Victorops）。Alertmanager的通知方式中还可以支持Webhook，通过这种方式开发者可以实现更多个性化的扩展支持。

集成邮件系统

邮箱应该是目前企业最常用的告警通知方式，Alertmanager内置了对SMTP协议的支持，因此对于企业用户而言，只需要一些基本的配置即可实现通过邮件的通知。

在Alertmanager使用邮箱通知，用户只需要定义好SMTP相关的配置，并且在receiver中定义接收方的邮件地址即可。在Alertmanager中我们可以直接在配置文件的global中定义全局的SMTP配置：

```
global:
  [ smtp_from: <tmpl_string> ]
  [ smtp_smarthost: <string> ]
  [ smtp_hello: <string> | default = "localhost" ]
  [ smtp_auth_username: <string> ]
  [ smtp_auth_password: <secret> ]
  [ smtp_auth_identity: <string> ]
  [ smtp_auth_secret: <secret> ]
  [ smtp_require_tls: <bool> | default = true ]
```

完成全局SMTP之后，我们只需要为receiver配置email_configs用于定义一组接收告警的邮箱地址即可，如下所示：

```
name: <string>
email_configs:
  [ - <email_config>, ... ]
```

每个email_config中定义相应的接收人邮箱地址，邮件通知模板等信息即可，当然如果当前接收人需要单独的SMTP配置，那直接在email_config中覆盖即可：

```
[ send_resolved: <boolean> | default = false ]
to: <tmpl_string>
[ html: <tmpl_string> | default = '{{ template "email.default.html" . }}' ]
[ headers: { <string>: <tmpl_string>, ... } ]
```

如果当前收件人需要接受告警恢复的通知的话，在email_config中定义 `send_resolved` 为true即可。

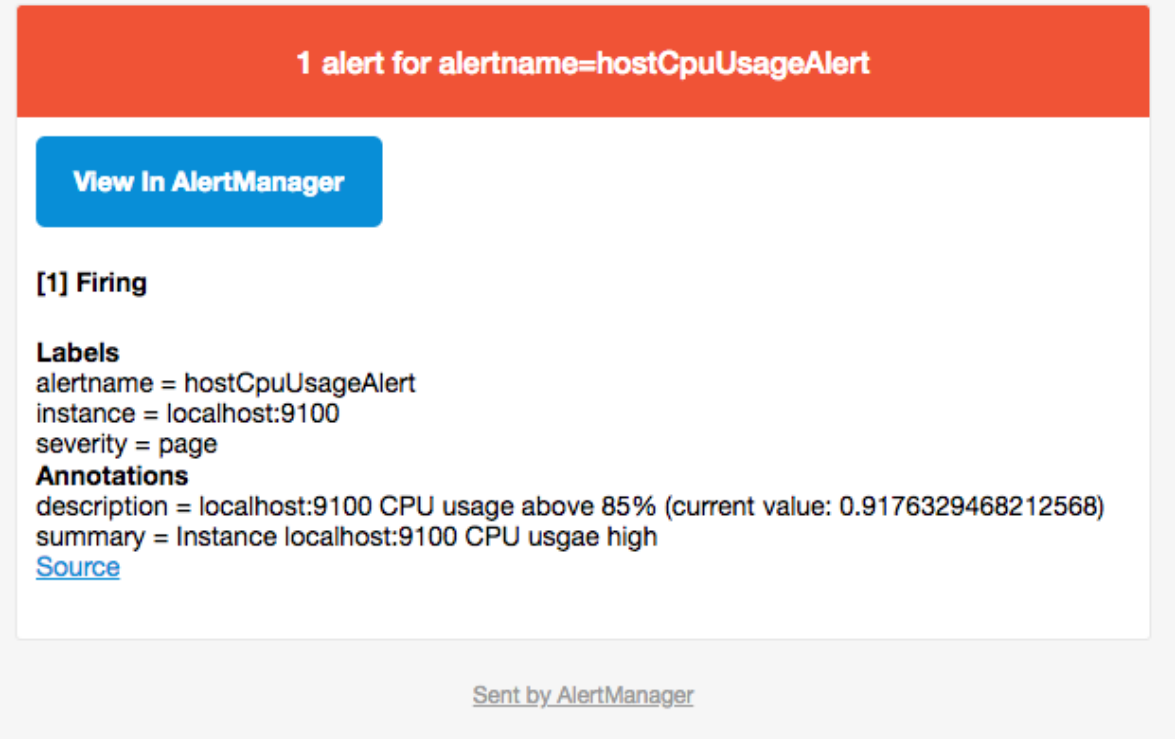
如果所有的邮件配置使用了相同的SMTP配置，则可以直接定义全局的SMTP配置。

这里，以Gmail邮箱为例，我们定义了一个全局的SMTP配置，并且通过route将所有告警信息发送到default-receiver中：

```
global:
  smtp_smarthost: smtp.gmail.com:587
  smtp_from: <smtp mail from>
  smtp_auth_username: <usernae>
  smtp_auth_identity: <username>
  smtp_auth_password: <password>

route:
  group_by: ['alertname']
  receiver: 'default-receiver'

receivers:
  - name: default-receiver
    email_configs:
      - to: <mail to address>
        send_resolved: true
```

> 需要注意的是新的Google账号安全规则需要使用”应用专有密码“作为邮箱登录密码

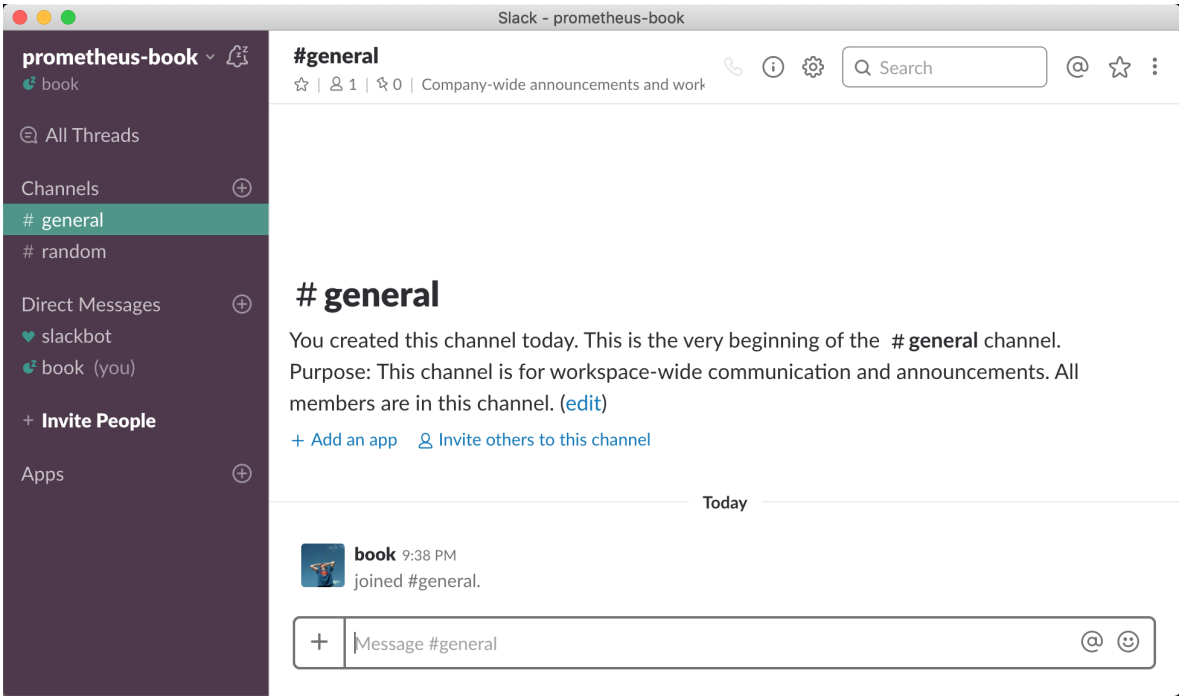
这时如果手动拉高主机CPU使用率，使得监控样本数据满足告警触发条件。在SMTP配置正确的情况下，可以接收到如下的告警内容：



集成Slack

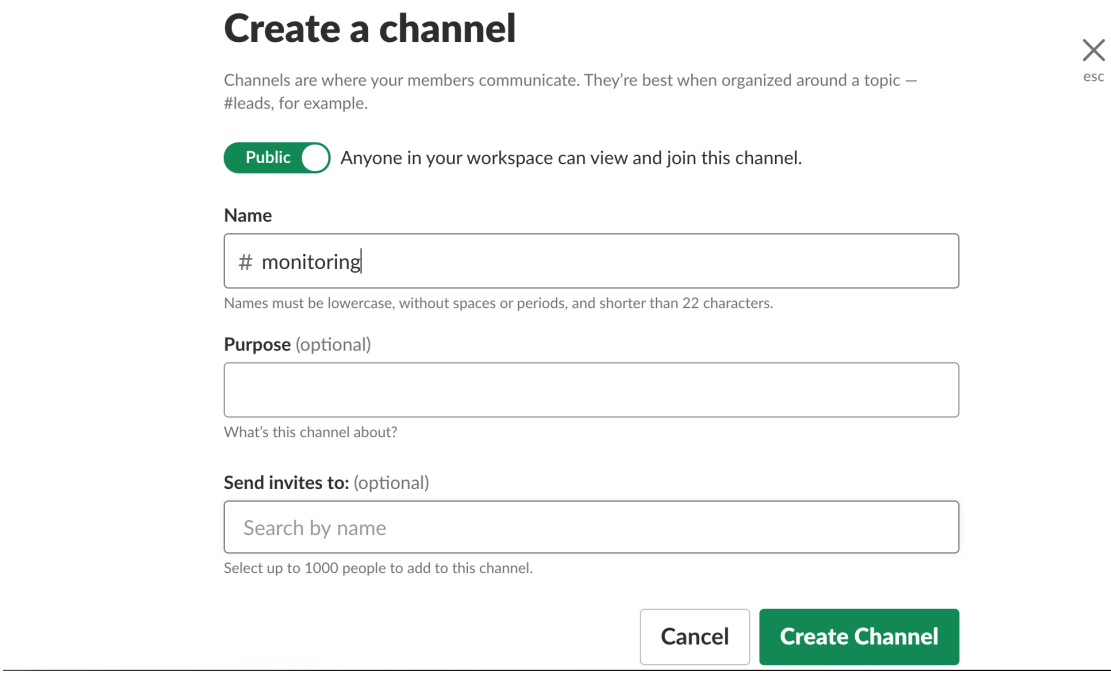
Slack是非常流行的团队沟通应用，提供群组聊天和直接消息发送功能，支持移动端，Web 和桌面平台。在国外有大量的IT团队使用Slack作为团队协作平台。同时其提供了强大的集成能力，在Slack的基础上也衍生出了大量的ChatOps相关的技术实践。这部分将介绍如何将Slack集成到Alertmanager中。

认识Slack

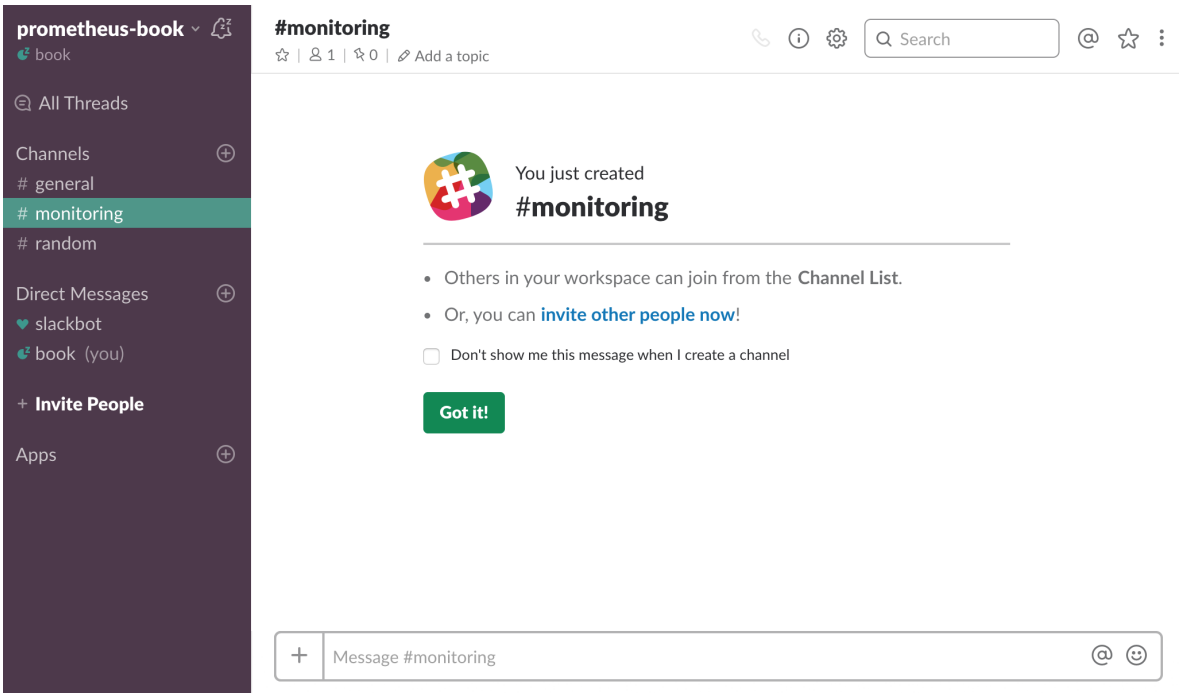


Slack作为一款即时通讯工具，协作沟通主要通过Channel（平台）来完成，用户可以在企业中根据用途添加多个Channel，并且通过Channel来集成各种第三方工具。

例如，我们可以为监控建立一个单独的Channel用于接收各种监控信息：



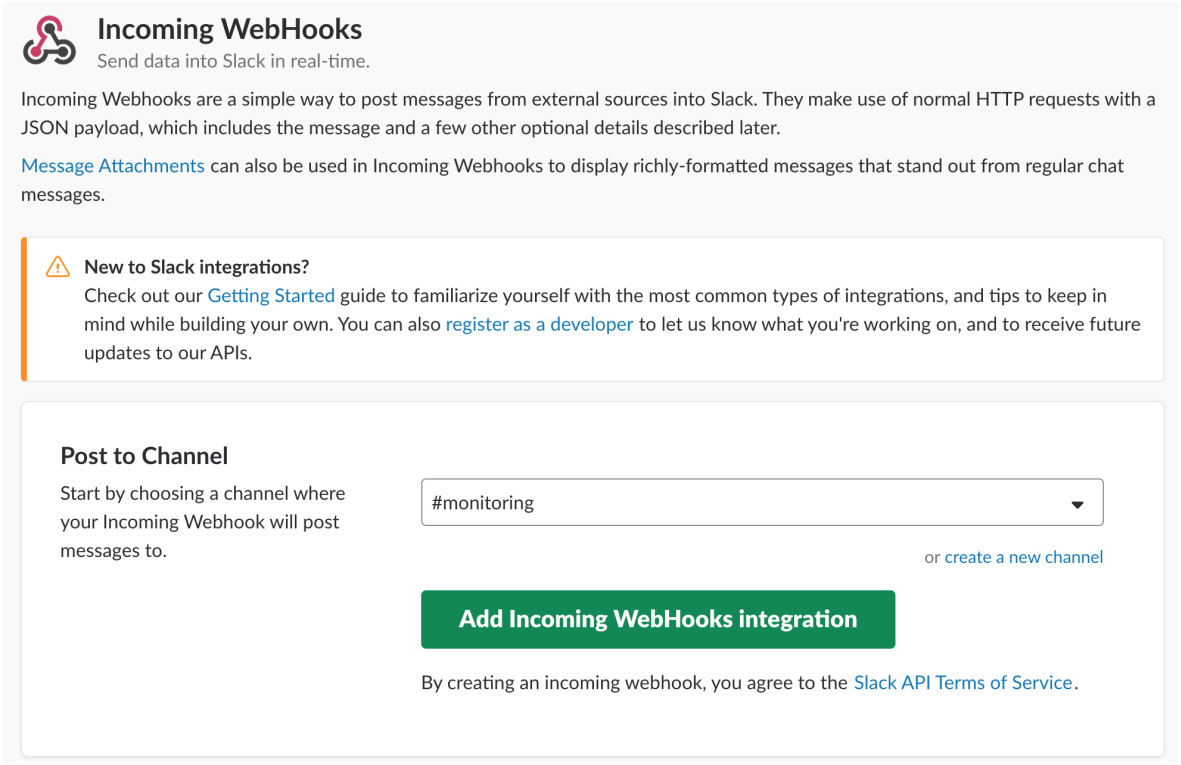
通过一个独立的Channle可以减少信息对用户工作的干扰，并且将相关信息聚合在一起：



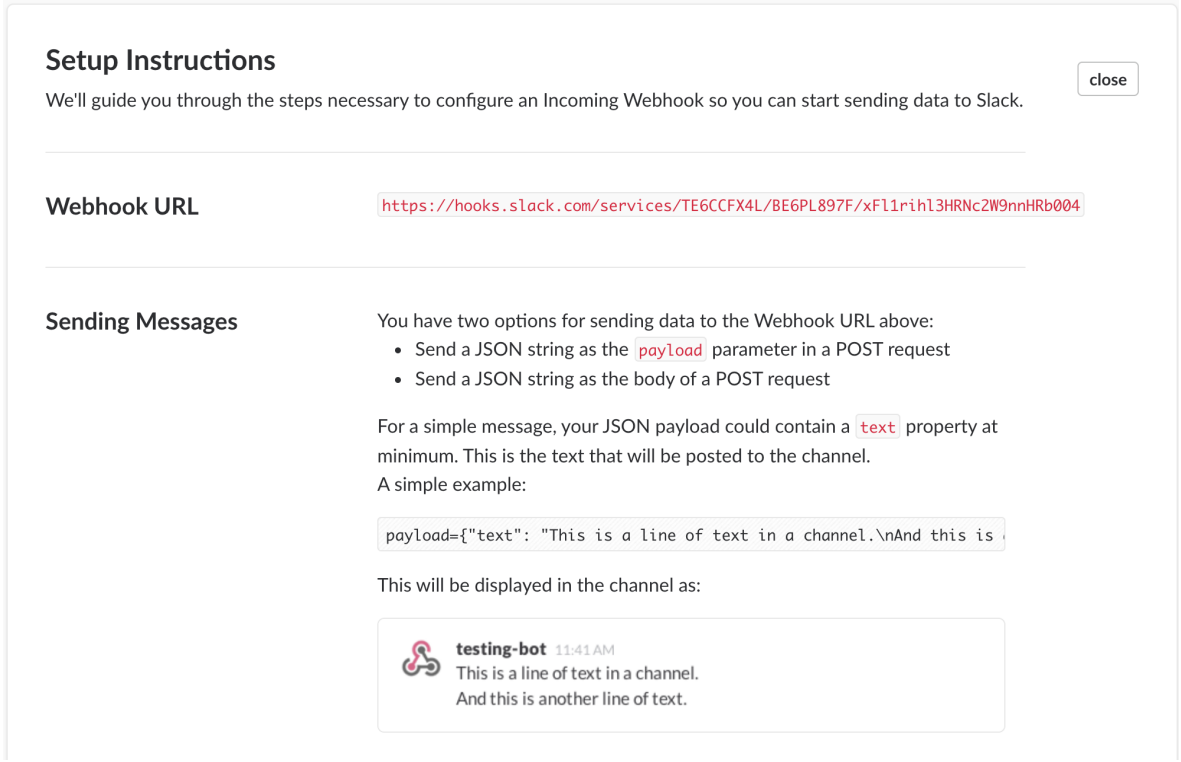
Slack的强大之处在于在Channel中添加各种第三方服务的集成，用户也可以基于Slack开发自己的聊天机器人来实现一些更高级的能力，例如自动化运维，提高开发效率等。

添加应用：Incomming Webhooks

为了能够在Monitoring中接收来自Alertmanager的消息，我们需要在Channel的设置选项中使用”Add an App”为Monitoring channel添加一个名为 `Incoming WebHooks` 的应用：



添加成功后Slack会显示 `Incoming WebHooks` 配置和使用方式：



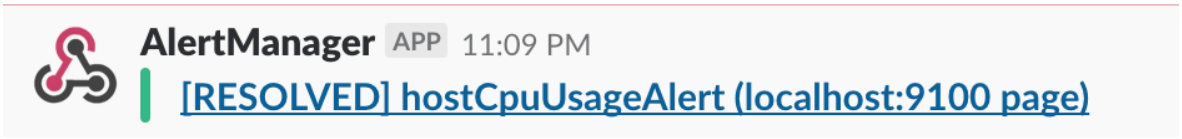
Incomming Webhook的工作方式很简单，Slack为当前Channel创建了一个用于接收消息的API地址：

```
https://hooks.slack.com/services/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
```

用户只需要使用Post方式向Channel发送需要通知的消息即可，例如，我们可以在命令行中通过curl模拟一次消息通知：

```
curl -d "payload={'text': 'This is a line of text in a channel.\nAnd this is another line of text.'}" https://hooks.slack.com/services/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
```

在网络正常的情况下，在Channel中会显示新的通知信息，如下所示：



除了发送纯文本以外，slack还支持在文本内容中添加链接，例如：

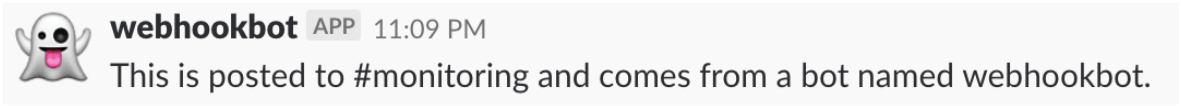
```
payload={"text": "A very important thing has occurred! <https://alert-system.com/alerts/1234|Click here> for details!"}
```

此时接收到的消息中建辉包含一个可点击的超链接地址。除了payload以外，Incomming Webhhook还支持一些其他的参数：

参数	作用	示例
username	设置当前聊天机器人的名称	webhookbot
icon_url	当前聊天机器人的头像地址	https://slack.com/img57.png
icon_emoji	使用emoji作为聊天机器人的头像	:ghost:
channel	消息发送的目标channel, 需要直接发给特定用户时使用@username即可	#monitoring 或者 @us

例如，使用以上参数发送一条更有趣的消息：

```
curl -X POST --data-urlencode "payload={'channel': '#monitoring', 'username': 'webhookbot', 'text': 'This is posted to #monitoring and comes from a bot named webhookbot.', 'icon_emoji': ':ghost:'}" https://hooks.slack.com/services/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
```



在Alertmanager中使用Slack

在了解了Slack以及Incomming Webhhook的基本使用方式后，在Alertmanager中添加Slack支持就非常简单了。

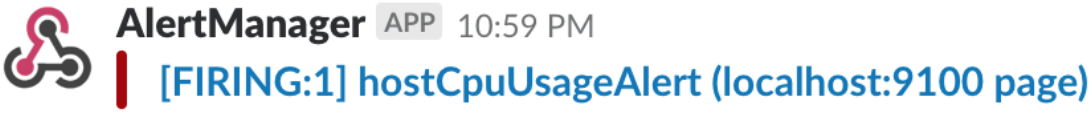
在Alertmanager的全局配置中，将Incomming Webhhook地址作为slack_api_url添加到全局配置中即可：

```
global:
  slack_api_url: https://hooks.slack.com/services/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
```

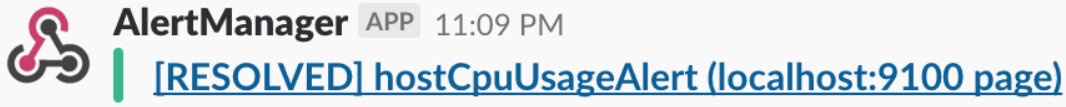
当然，也可以在每个receiver中单独定义自己的slack_configs即可：

```
receivers:
- name: slack
  slack_configs:
    - channel: '#monitoring'
      send_resolved: true
```

这里如果我们手动拉高当前主机的CPU利用率，在#Monitoring平台中，我们会接收到一条告警信息如下所示：



而当告警项恢复正常后，则可以接收到如下通知：



对于Incomming Webhhook支持的其它自定义参数，也可以在slack_config中进行定义，slack_config的主要配置如下：

```
channel: <tmpl_string>
[ send_resolved: <boolean> | default = false ]
[ api_url: <secret> | default = global.slack_api_url ]
[ icon_emoji: <tmpl_string> ]
[ icon_url: <tmpl_string> ]
[ link_names: <boolean> | default = false ]
[ username: <tmpl_string> | default = '{{ template "slack.default.username" . }}' ]
[ color: <tmpl_string> | default = '{{ if eq .Status "firing" }}danger{{ else }}good{{ end }}' ]
[ footer: <tmpl_string> | default = '{{ template "slack.default.footer" . }}' ]
[ pretext: <tmpl_string> | default = '{{ template "slack.default.pretext" . }}' ]
[ text: <tmpl_string> | default = '{{ template "slack.default.text" . }}' ]
[ title: <tmpl_string> | default = '{{ template "slack.default.title" . }}' ]
[ title_link: <tmpl_string> | default = '{{ template "slack.default.titlelink" . }}' ]
[ image_url: <tmpl_string> ]
[ thumb_url: <tmpl_string> ]
```

如果要覆盖默认的告警内容，直接使用Go Template即可。例如：

```
color: '{{ if eq .Status "firing" }}danger{{ else }}good{{ end }}'
```

集成企业微信

Alertmanager已经内置了对企业微信的支持，我们可以通过企业微信来管理报警，更进一步可以通过企业微信和微信的互通来直接将告警消息转发到个人微信上。

prometheus官网中给出了企业微信的相关配置说明

```
# Whether or not to notify about resolved alerts.
[ send_resolved: <boolean> | default = false ]

# The API key to use when talking to the WeChat API.
[ api_secret: <secret> | default = global.wechat_api_secret ]

# The WeChat API URL.
[ api_url: <string> | default = global.wechat_api_url ]

# The corp id for authentication.
[ corp_id: <string> | default = global.wechat_api_corp_id ]

# API request data as defined by the WeChat API.
[ message: <tmpl_string> | default = '{{ template "wechat.default.message" . }}' ]
[ agent_id: <string> | default = '{{ template "wechat.default.agent_id" . }}' ]
[ to_user: <string> | default = '{{ template "wechat.default.to_user" . }}' ]
[ to_party: <string> | default = '{{ template "wechat.default.to_party" . }}' ]
[ to_tag: <string> | default = '{{ template "wechat.default.to_tag" . }}' ]
```

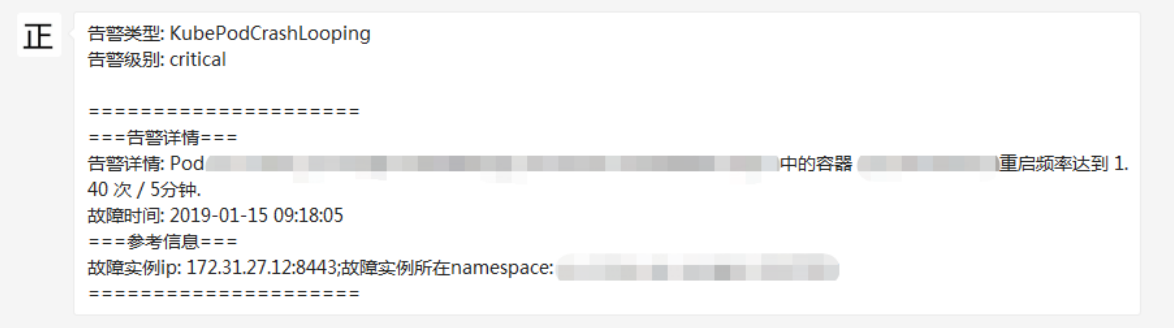
企业微信相关概念说明请参考企业微信API说明，可以在企业微信的后台中建立多个应用，每个应用对应不同的报警分组，由企业微信来做接收成员的划分。具体配置参考如下：

```
global:
  resolve_timeout: 10m
  wechat_api_url: 'https://qyapi.weixin.qq.com/cgi-bin/'
  wechat_api_secret: '应用的secret，在应用的配置页面可以看到'
  wechat_api_corp_id: '企业id，在企业的配置页面可以看到'
templates:
- '/etc/alertmanager/config/*.tmpl'
route:
  group_by: ['alertname']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  routes:
  - receiver: 'wechat'
    continue: true
inhibit_rules:
- source_match:
receivers:
- name: 'wechat'
  wechat_configs:
  - send_resolved: false
    corp_id: '企业id，在企业的配置页面可以看到'
    to_user: '@all'
    to_party: ' PartyID1 | PartyID2 '
    message: '{{ template "wechat.default.message" . }}'
    agent_id: '应用的AgentId，在应用的配置页面可以看到'
    api_secret: '应用的secret，在应用的配置页面可以看到'
```

配置模板示例如下：

```
{{ define "wechat.default.message" }}
{{- if gt (len .Alerts.Firing) 0 -}}
{{- range $index, $alert := .Alerts -}}
{{- if eq $index 0 -}}
告警类型: {{ $alert.Labels.alertname }}
告警级别: {{ $alert.Labels.severity }}

====================
{{- end }}
===告警详情===
告警详情: {{ $alert.Annotations.message }}
故障时间: {{ $alert.StartsAt.Format "2006-01-02 15:04:05" }}
===参考信息===
{{ if gt (len $alert.Labels.instance) 0 -}}故障实例ip: {{ $alert.Labels.instance }};{{- end -}}
{{- if gt (len $alert.Labels.namespace) 0 -}}故障实例所在namespace: {{ $alert.Labels.namespace }};{{- end -}}
{{- if gt (len $alert.Labels.node) 0 -}}故障物理机ip: {{ $alert.Labels.node }};{{- end -}}
{{- if gt (len $alert.Labels.pod_name) 0 -}}故障pod名称: {{ $alert.Labels.pod_name }}{{- end }}
====================
{{- end }}
{{- end }}

{{- if gt (len .Alerts.Resolved) 0 -}}
{{- range $index, $alert := .Alerts -}}
{{- if eq $index 0 -}}
告警类型: {{ $alert.Labels.alertname }}
告警级别: {{ $alert.Labels.severity }}

====================
{{- end }}
===告警详情===
告警详情: {{ $alert.Annotations.message }}
故障时间: {{ $alert.StartsAt.Format "2006-01-02 15:04:05" }}
恢复时间: {{ $alert.EndsAt.Format "2006-01-02 15:04:05" }}
===参考信息===
{{ if gt (len $alert.Labels.instance) 0 -}}故障实例ip: {{ $alert.Labels.instance }};{{- end -}}
{{- if gt (len $alert.Labels.namespace) 0 -}}故障实例所在namespace: {{ $alert.Labels.namespace }};{{- end -}}
{{- if gt (len $alert.Labels.node) 0 -}}故障物理机ip: {{ $alert.Labels.node }};{{- end -}}
{{- if gt (len $alert.Labels.pod_name) 0 -}}故障pod名称: {{ $alert.Labels.pod_name }};{{- end }}
====================
{{- end }}
{{- end }}
{{- end }}
```

这时如果某一容器频繁重启，可以接收到如下的告警内容：



集成钉钉：基于Webhook的扩展

在某些情况下除了Alertmanager已经内置的集中告警通知方式以外，对于不同的用户和组织而言还需要一些自定义的告知方式支持。通过Alertmanager提供的webhook支持可以轻松实现这一类的扩展。除了用于支持额外的通知方式，webhook还可以与其他第三方系统集成实现运维自动化，或者弹性伸缩等。

在Alertmanager中可以使用如下配置定义基于webhook的告警接收器receiver。一个receiver可以对应一组webhook配置。

```
name: <string>
webhook_configs:
  [ - <webhook_config>, ... ]
```

每一项webhook_config的具体配置格式如下：

```
# Whether or not to notify about resolved alerts.
[ send_resolved: <boolean> | default = true ]

# The endpoint to send HTTP POST requests to.
url: <string>

# The HTTP client's configuration.
[ http_config: <http_config> | default = global.http_config ]
```

send_resolved用于指定是否在告警消除时发送回执消息。url则是用于接收webhook请求的地址。http_configs则是在需要对请求进行SSL配置时使用。

当用户定义webhook用于接收告警信息后，当告警被触发时，Alertmanager会按照以下格式向这些url地址发送HTTP Post请求，请求内容如下：

```
{
  "version": "4",
  "groupKey": <string>,    // key identifying the group of alerts (e.g. to deduplicate)
  "status": "<resolved|firing>",
  "receiver": <string>,
  "groupLabels": <object>,
  "commonLabels": <object>,
  "commonAnnotations": <object>,
  "externalURL": <string>,  // backlink to the Alertmanager.
  "alerts": [
    {
      "labels": <object>,
      "annotations": <object>,
      "startsAt": "<rfc3339>",
      "endsAt": "<rfc3339>"
    }
  ]
}
```

使用Golang创建webhook服务

首先我们尝试使用Golang创建用于接收webhook告警通知的服务。首先创建model包，用于映射ALertmanager发送的告警信息，Alertmanager的一个通知中根据配置的group_by规则可能会包含多条告警信息Alert。创建告警通知对应的结构体Notification。

```
package model

import "time"

type Alert struct {
    Labels      map[string]string `json:"labels"`
    Annotations map[string]string `json:annotations`
    StartsAt    time.Time         `json:"startsAt"`
    EndsAt      time.Time         `json:"endsAt"`
}

type Notification struct {
    Version           string            `json:"version"`
    GroupKey          string            `json:"groupKey"`
    Status            string            `json:"status"`
    Receiver          string            `json:receiver`
    GroupLabels       map[string]string `json:groupLabels`
    CommonLabels      map[string]string `json:commonLabels`
    CommonAnnotations map[string]string `json:commonAnnotations`
    ExternalURL       string            `json:externalURL`
    Alerts            []Alert           `json:alerts`
}
```

这里使用gin-gonic框架创建用于接收Webhook通知的Web服务。定义路由/webhook接收来自Alertmanager的POST请求。

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
    model "github.com/yunlzheng/alertmanaer-dingtalk-webhook/model"
)

func main() {
    router := gin.Default()
    router.POST("/webhook", func(c *gin.Context) {
        var notification model.Notification

        err := c.BindJSON(&notification)

        if err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

        c.JSON(http.StatusOK, gin.H{"message": " successful receive alert notification message!"})

    })
    router.Run()
}
```

与钉钉集成

钉钉，阿里巴巴出品，专为中国企业打造的免费智能移动办公平台，提供了即时通讯以及移动办公等丰富的功能。

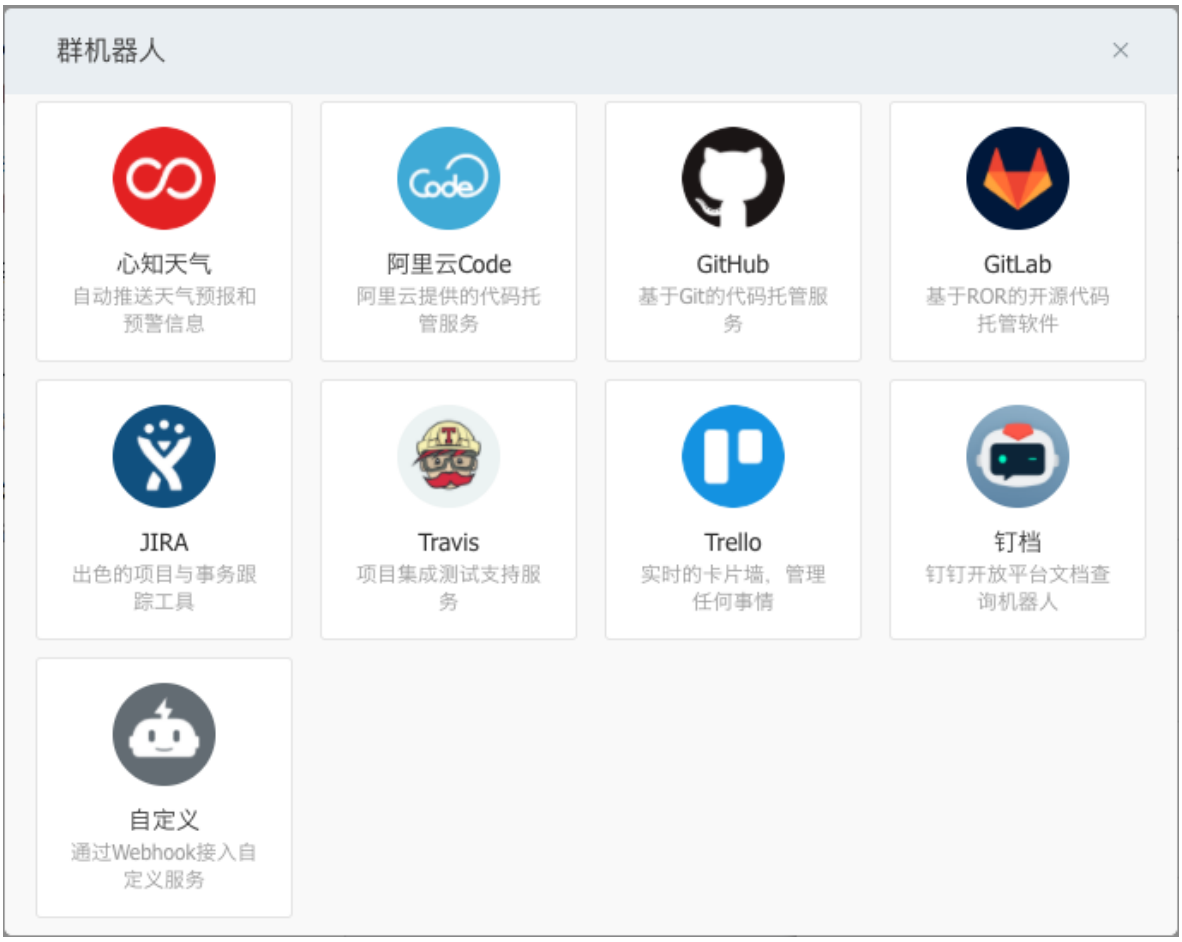
钉钉群机器人是钉钉群的高级扩展功能。群机器人可以将第三方服务的信息聚合到群聊中，实现自动化的信息同步。例如：通过聚合GitHub，GitLab等源码管理服务，实现源码更新同步；通过聚合Trello，JIRA等项目协调服务，实现项目信息同步。不

仅如此，群机器人支持Webhook协议的自定义接入，支持更多可能性。这里我们将演示如果将Alertmanager运维报警提醒通过自定义机器人聚合到钉钉群。

这里将继续扩展webhook服务，以支持将Alertmanager的告警通知转发到钉钉平台。完整的示例代码可以从github仓库https://github.com/yunlzheng/alertmanaer-dingtalk-webhook中获取。

自定义webhook群机器人

通过钉钉客户端（如：桌面或者手机）进入到群设置后选择“群机器人”。将显示如下界面：



选择“自定义机器人”，并且按照提示填写机器人名称，获取机器人webhook地址，如下所示：

webhook机器人创建成功后，用户就可以使用任何方式向该地址发起HTTP POST请求，即可实现向该群主发送消息。目前自定义机器人支持文本(text)，连接(link)，markdown三种消息类型。

例如，可以向webhook地址以POST形式发送以下

```
{
    "msgtype": "markdown",
    "markdown": {
        "title":"Prometheus告警信息",
        "text": "#### 监控指标\n" +
                "> 监控描述信息\n\n" +
                "> ###### 告警时间 \n"
    },
   "at": {
        "atMobiles": [
            "156xxxx8827",
            "189xxxx8325"
        ],
        "isAtAll": false
    }
}
```

可以使用curl验证钉钉webhook是否能够成功调用：

```
$ curl -l -H "Content-type: application/json" -X POST -d '{"msgtype": "markdown","markdown": {"title":"Prometheus告警信息","text": "#### 监控指标\n> 监控描述信息\n\n> ###### 告警时间 \n"},"at": {"isAtAll": false}}' https://oapi.dingtalk.com/robot/send?access_token=xxxx
{"errcode":0,"errmsg":"ok"}
```

调用成功后，可以在钉钉应用群消息中接收到类似于如下通知消息:

定义转换器将告警通知转化为Dingtalk消息对象

这里定义结构体DingTalkMarkdown用于映射Dingtalk的消息体。

```
package model

type At struct {
    AtMobiles []string `json:"atMobiles"`
    IsAtAll   bool     `json:"isAtAll"`
}

type DingTalkMarkdown struct {
    MsgType  string    `json:"msgtype"`
    At       *At       `json:at`
    Markdown *Markdown `json:"markdown"`
}

type Markdown struct {
    Title string `json:"title"`
    Text  string `json:"text"`
}
```

定义转换器将Alertmanager发送的告警通知转换为Dingtalk的消息体。

```
package transformer

import (
    "bytes"
    "fmt"

    "github.com/yunlzheng/alertmanaer-dingtalk-webhook/model"
)

// TransformToMarkdown transform alertmanager notification to dingtalk markdow message
func TransformToMarkdown(notification model.Notification) (markdown *model.DingTalkMarkdown, err error) {

    groupKey := notification.GroupKey
    status := notification.Status

    annotations := notification.CommonAnnotations

    var buffer bytes.Buffer

    buffer.WriteString(fmt.Sprintf("### 通知组%s(当前状态:%s) \n", groupKey, status))

    buffer.WriteString(fmt.Sprintf("#### 告警项:\n"))
```

```
    for _, alert := range notification.Alerts {
        annotations := alert.Annotations
        buffer.WriteString(fmt.Sprintf("##### %s\n > %s\n", annotations["summary"], annotations["description"
]))
        buffer.WriteString(fmt.Sprintf("\n> 开始时间：%s\n", alert.StartsAt.Format("15:04:05")))
    }

    markdown = &model.DingTalkMarkdown{
        MsgType: "markdown",
        Markdown: &model.Markdown{
            Title: fmt.Sprintf("通知组：%s(当前状态:%s)", groupKey, status),
            Text:  buffer.String(),
        },
        At: &model.At{
            IsAtAll: false,
        },
    }

    return
}
```

创建Dingtalk通知发送包

notifier包中使用golang的net/http包实现与Dingtalk群机器人的交互。Send方法包含两个参数：接收到的告警通知结构体指针，以及Dingtalk群机器人的Webhook地址。

通过包transformer.TransformToMarkdown将Alertmanager告警通知与Dingtalk消息进行映射。

```
package notifier

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/yunlzheng/alertmanaer-dingtalk-webhook/model"
    "github.com/yunlzheng/alertmanaer-dingtalk-webhook/transformer"
)

func Send(notification model.Notification, dingtalkRobot string) (err error) {

    markdown, err := transformer.TransformToMarkdown(notification)

    if err != nil {
        return
    }

    data, err := json.Marshal(markdown)
    if err != nil {
        return
    }

    req, err := http.NewRequest(
        "POST",
        dingtalkRobot,
        bytes.NewBuffer(data))

    if err != nil {
```

```
        return
    }

    req.Header.Set("Content-Type", "application/json")
    client := &http.Client{}
    resp, err := client.Do(req)

    if err != nil {
        return
    }

    defer resp.Body.Close()
    fmt.Println("response Status:", resp.Status)
    fmt.Println("response Headers:", resp.Header)

    return
}
```

扩展启动函数

首先为程序添加命令行参数支持，用于在启动时添加全局的Dingtalk群聊机器人地址。

```
package main

import (
  "flag"
  ...
  "github.com/yunlzheng/alertmanaer-dingtalk-webhook/notifier"
)

var (
    h             bool
    defaultRobot string
)

func init() {
    flag.BoolVar(&h, "h", false, "help")
    flag.StringVar(&defaultRobot, "defaultRobot", "", "global dingtalk robot webhook")
}

func main() {

    flag.Parse()

    if h {
        flag.Usage()
        return
    }

  ...

}
```

同时通过notifier包的Send方法将告警通知发送给Dingtalk群聊机器人

```
func main() {

  ...
```

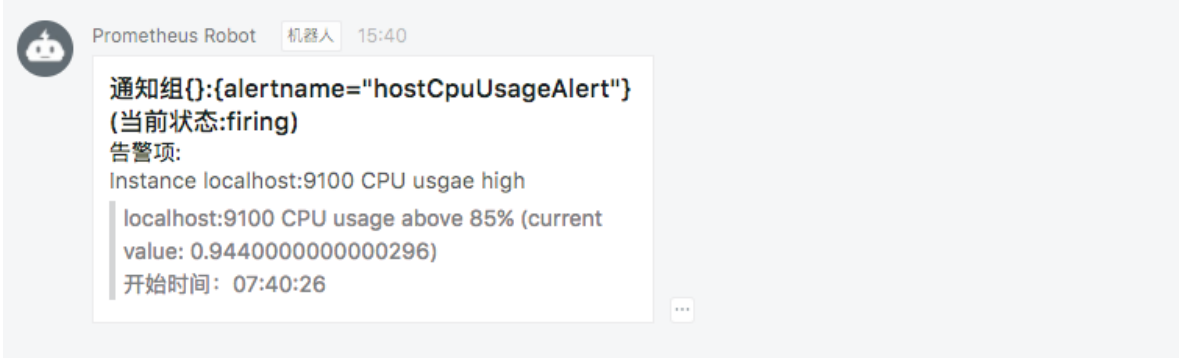
```
  err = notifier.Send(notification, defaultRobot)

  if err != nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

  }

  c.JSON(http.StatusOK, gin.H{"message": "send to dingtalk successful!"})
}
```

使用Dingtalk扩展

运行并启动dingtalk webhook服务之后，修改Alertmanager配置文件，为default-receiver添加webhook配置，如下所示：

```
receivers:
  - name: default-receiver
    email_configs:
      - to: yunl.zheng@wise2c.com
    webhook_configs:
      - url: http://localhost:8080/webhook
```

重启Alertmanager服务后，手动拉高虚拟机CPU使用率触发告警条件，此时Dingtalk即可接收到相应的告警通知信息：



告警模板详解

默认情况下Alertmanager使用了系统自带的默认通知模板，模板源码可以从https://github.com/prometheus/alertmanager/blob/master/template/default.tmpl获得。Alertmanager的通知模板基于Go的模板系统。Alertmanager也支持用户定义和使用自己的模板，一般来说有两种方式可以选择。

第一种，基于模板字符串。用户可以直接在Alertmanager的配置文件中使用模板字符串，例如:

```
receivers:
- name: 'slack-notifications'
  slack_configs:
  - channel: '#alerts'
    text: 'https://internal.myorg.net/wiki/alerts/{{ .GroupLabels.app }}/{{ .GroupLabels.alertname }}'
```

第二种方式，自定义可复用的模板文件。例如，可以创建自定义模板文件custom-template.tmpl，如下所示：

```
{{ define "slack.myorg.text" }}https://internal.myorg.net/wiki/alerts/{{ .GroupLabels.app }}/{{ .GroupLabels.
alertname }}{{ end}}
```

通过在Alertmanager的全局设置中定义templates配置来指定自定义模板的访问路径:

```
# Files from which custom notification template definitions are read.
# The last component may use a wildcard matcher, e.g. 'templates/*.tmpl'.
templates:
  [ - <filepath> ... ]
```

在设置了自定义模板的访问路径后，用户则可以直接在配置中使用该模板：

```
receivers:
- name: 'slack-notifications'
  slack_configs:
  - channel: '#alerts'
    text: '{{ template "slack.myorg.text" . }}'

templates:
- '/etc/alertmanager/templates/myorg.tmpl'
```

屏蔽告警通知

Alertmanager提供了方式可以帮助用户控制告警通知的行为，包括预先定义的抑制机制和临时定义的静默规则。

抑制机制

Alertmanager的抑制机制可以避免当某种问题告警产生之后用户接收到大量由此问题导致的一系列的其它告警通知。例如当集群不可用时，用户可能只希望接收到一条告警，告诉他这时候集群出现了问题，而不是大量的如集群中的应用异常、中间件服务异常的告警通知。

在Alertmanager配置文件中，使用inhibit_rules定义一组告警的抑制规则：

```
inhibit_rules:
  [ - <inhibit_rule> ... ]
```

每一条抑制规则的具体配置如下：

```
target_match:
  [ <labelname>: <labelvalue>, ... ]
target_match_re:
  [ <labelname>: <regex>, ... ]

source_match:
  [ <labelname>: <labelvalue>, ... ]
source_match_re:
  [ <labelname>: <regex>, ... ]

[ equal: '[' <labelname>, ... ']' ]
```

当已经发送的告警通知匹配到target_match和target_match_re规则，当有新的告警规则如果满足source_match或者定义的匹配规则，并且已发送的告警与新产生的告警中equal定义的标签完全相同，则启动抑制机制，新的告警不会发送。

例如，定义如下抑制规则：

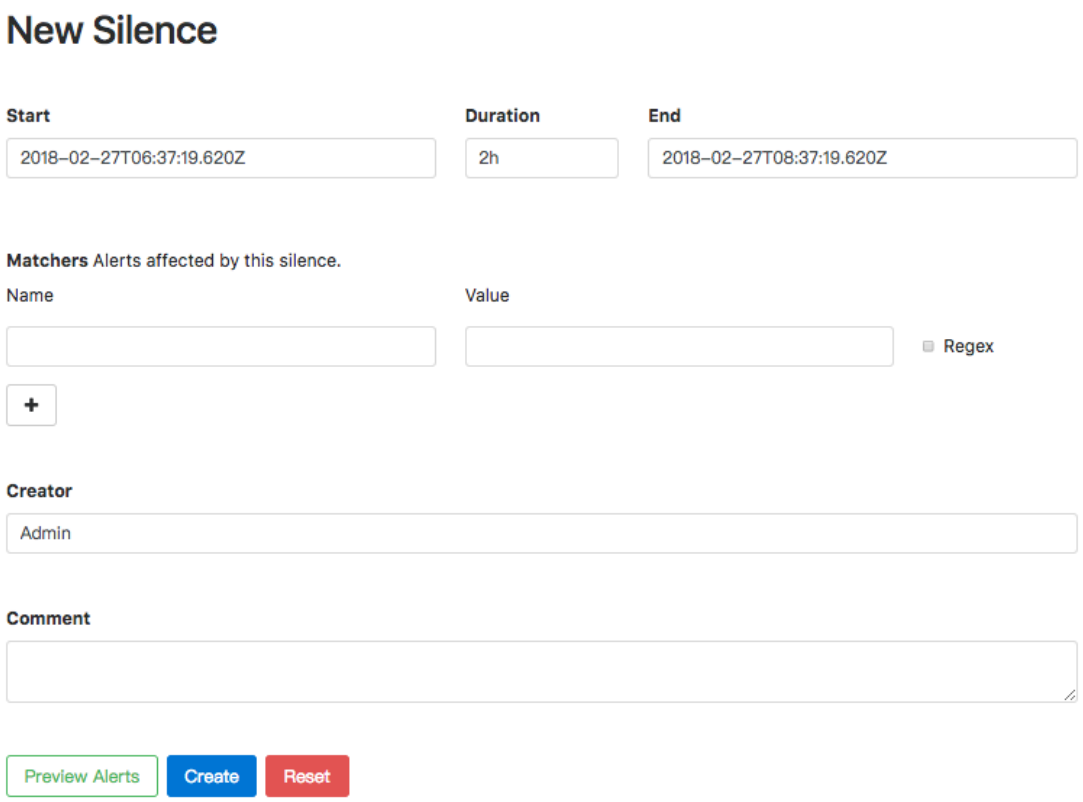
```
- source_match:
    alertname: NodeDown
    severity: critical
  target_match:
    severity: critical
  equal:
    - node
```

例如当集群中的某一个主机节点异常宕机导致告警NodeDown被触发，同时在告警规则中定义了告警级别severity=critical。由于主机异常宕机，该主机上部署的所有服务，中间件会不可用并触发报警。根据抑制规则的定义，如果有新的告警级别为severity=critical，并且告警中标签node的值与NodeDown告警的相同，则说明新的告警是由NodeDown导致的，则启动抑制机制停止向接收器发送通知。

临时静默

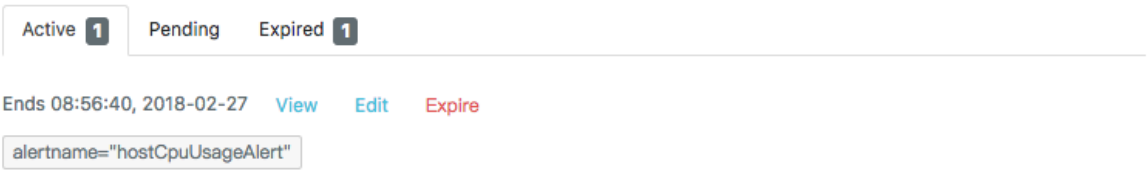
除了基于抑制机制可以控制告警通知的行为以外，用户或者管理员还可以直接通过Alertmanager的UI临时屏蔽特定的告警通知。通过定义标签的匹配规则(字符串或者正则表达式)，如果新的告警通知满足静默规则的设置，则停止向receiver发送通知。

进入Alertmanager UI，点击"New Silence"显示如下内容：



用户可以通过该UI定义新的静默规则的开始时间以及持续时间，通过Matchers部分可以设置多条匹配规则(字符串匹配或者正则匹配)。填写当前静默规则的创建者以及创建原因后，点击”Create”按钮即可。

通过”Preview Alerts”可以查看预览当前匹配规则匹配到的告警信息。静默规则创建成功后，Alertmanager会开始加载该规则并且设置状态为Pending,当规则生效后则进行到Active状态。



当静默规则生效以后，从Alertmanager的Alerts页面下用户将不会看到该规则匹配到的告警信息。

对于已经生效的规则，用户可以通过手动点击”Expire“按钮使当前规则过期。

使用Recoding Rules优化性能

通过PromQL可以实时对Prometheus中采集到的样本数据进行查询，聚合以及其它各种运算操作。而在某些PromQL较为复杂且计算量较大时，直接使用PromQL可能会导致Prometheus响应超时的情况。这时需要一种能够类似于后台批处理的机制能够在后台完成这些复杂运算的计算，对于使用者而言只需要查询这些运算结果即可。Prometheus通过Recoding Rule规则支持这种后台计算的方式，可以实现对复杂查询的性能优化，提高查询效率。

定义Recoding rules

在Prometheus配置文件中，通过rule_files定义recoding rule规则文件的访问路径。

```
rule_files:
  [ - <filepath_glob> ... ]
```

每一个规则文件通过以下格式进行定义：

```
groups:
  [ - <rule_group> ]
```

一个简单的规则文件可能是这个样子的：

```
groups:
  - name: example
    rules:
    - record: job:http_inprogress_requests:sum
      expr: sum(http_inprogress_requests) by (job)
```

rule_group的具体配置项如下所示：

```
# The name of the group. Must be unique within a file.
name: <string>

# How often rules in the group are evaluated.
[ interval: <duration> | default = global.evaluation_interval ]

rules:
  [ - <rule> ... ]
```

与告警规则一致，一个group下可以包含多条规则rule。

```
# The name of the time series to output to. Must be a valid metric name.
record: <string>

# The PromQL expression to evaluate. Every evaluation cycle this is
# evaluated at the current time, and the result recorded as a new set of
# time series with the metric name as given by 'record'.
expr: <string>

# Labels to add or overwrite before storing the result.
labels:
  [ <labelname>: <labelvalue> ]
```

根据规则中的定义，Prometheus会在后台完成expr中定义的PromQL表达式计算，并且将计算结果保存到新的时间序列record中。同时还可以通过labels为这些样本添加额外的标签。

这些规则文件的计算频率与告警规则计算频率一致，都通过global.evaluation_interval定义：

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

小结

当故障发生时，即时获取到异常结果是大多数用户使用监控系统的最主要的目的之一。通过Prometheus提供的告警以及告警处理能力，通过内置的告警通知能力，能过帮助用户快速实现告警的通知。同时其还提供了简单有效的扩展方式，让用户可以基于Prometheus的告警处理模式实现更多的定制化需求。

Exporter详解

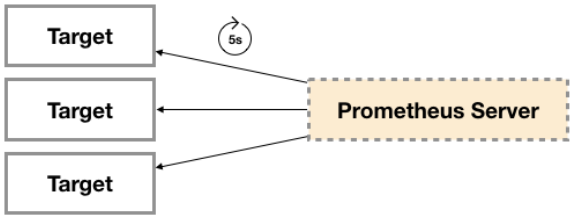
在第1章中为了采集主机的监控样本数据，我们在主机上安装了一个Node Exporter程序，该程序对外暴露了一个用于获取当前监控样本数据的HTTP访问地址。这样的一个程序称为Exporter，Exporter的实例称为一个Target。Prometheus通过轮询的方式定时从这些Target中获取监控数据样本，并且存储在数据库当中。 在这一章节当中我们将重点讨论这些用于获取特定目标监控样本数据的程序Exporter。

本章的主要内容：

- 常用Exporter的使用，例如如何监控数据库，消息中间件等
- 如何实现自定义的Exporter程序
- 如何对已有的应用程序扩展Prometheus监控支持

Exporter是什么

广义上讲所有可以向Prometheus提供监控样本数据的程序都可以被称为一个Exporter。而Exporter的一个实例称为target，如下所示，Prometheus通过轮询的方式定期从这些target中获取样本数据:



Exporter的来源

从Exporter的来源上来讲，主要分为两类：

- 社区提供的

Prometheus社区提供了丰富的Exporter实现，涵盖了从基础设施，中间件以及网络等各个方面的监控功能。这些Exporter可以实现大部分通用的监控需求。下表列举一些社区中常用的Exporter：

范围	常用Exporter
数据库	MySQL Exporter, Redis Exporter, MongoDB Exporter, MSSQL Exporter等
硬件	Apcupsd Exporter，IoT Edison Exporter， IPMI Exporter, Node Exporter等
消息队列	Beanstalkd Exporter, Kafka Exporter, NSQ Exporter, RabbitMQ Exporter等
存储	Ceph Exporter, Gluster Exporter, HDFS Exporter, ScaleIO Exporter等
HTTP服务	Apache Exporter, HAProxy Exporter, Nginx Exporter等
API服务	AWS ECS Exporter， Docker Cloud Exporter, Docker Hub Exporter, GitHub Exporter等
日志	Fluentd Exporter, Grok Exporter等
监控系统	Collectd Exporter, Graphite Exporter, InfluxDB Exporter, Nagios Exporter, SNMP Exporter等

范围	常用Exporter
其它	Blockbox Exporter, JIRA Exporter, Jenkins Exporter， Confluence Exporter等

- 用户自定义的

除了直接使用社区提供的Exporter程序以外，用户还可以基于Prometheus提供的Client Library创建自己的Exporter程序，目前Promthues社区官方提供了对以下编程语言的支持：Go、Java/Scala、Python、Ruby。同时还有第三方实现的如：Bash、C++、Common Lisp、Erlang,、Haskeel、Lua、Node.js、PHP、Rust等。

Exporter的运行方式

从Exporter的运行方式上来讲，又可以分为：

- 独立使用的

以我们已经使用过的Node Exporter为例，由于操作系统本身并不直接支持Prometheus，同时用户也无法通过直接从操作系统层面上提供对Prometheus的支持。因此，用户只能通过独立运行一个程序的方式，通过操作系统提供的相关接口，将系统的运行状态数据转换为可供Prometheus读取的监控数据。 除了Node Exporter以外，比如MySQL Exporter、Redis Exporter等都是通过这种方式实现的。 这些Exporter程序扮演了一个中间代理人的角色。

- 集成到应用中的

为了能够更好的监控系统的内部运行状态，有些开源项目如Kubernetes，ETCD等直接在代码中使用了Prometheus的Client Library，提供了对Prometheus的直接支持。这种方式打破的监控的界限，让应用程序可以直接将内部的运行状态暴露给Prometheus，适合于一些需要更多自定义监控指标需求的项目。

Exporter规范

所有的Exporter程序都需要按照Prometheus的规范，返回监控的样本数据。以Node Exporter为例，当访问/metrics地址时会返回以下内容：

```
# HELP node_cpu Seconds the cpus spent in each mode.
# TYPE node_cpu counter
node_cpu{cpu="cpu0",mode="idle"} 362812.7890625
# HELP node_load1 1m load average.
# TYPE node_load1 gauge
node_load1 3.0703125
```

这是一种基于文本的格式规范，在Prometheus 2.0之前的版本还支持Protocol buffer规范。相比于Protocol buffer文本具有更好的可读性，以及跨平台性。Prometheus 2.0的版本也已经不再支持Protocol buffer，这里就不对Protocol buffer规范做详细的阐述。

Exporter返回的样本数据，主要由三个部分组成：样本的一般注释信息（HELP），样本的类型注释信息（TYPE）和样本。Prometheus会对Exporter响应的内容逐行解析：

如果当前行以# HELP开始，Prometheus将会按照以下规则对内容进行解析，得到当前的指标名称以及相应的说明信息：

```
# HELP <metrics_name> <doc_string>
```

如果当前行以# TYPE开始，Prometheus会按照以下规则对内容进行解析，得到当前的指标名称以及指标类型：

```
# TYPE <metrics_name> <metrics_type>
```

TYPE注释行必须出现在指标的第一个样本之前。如果没有明确的指标类型需要返回为untyped。 除了# 开头的所有行都会被视为是监控样本数据。 每一行样本需要满足以下格式规范:

```
metric_name [
  "{" label_name "=" `"` label_value `"` { "," label_name "=" `"` label_value `"` } [ "," ] "}"
] value [ timestamp ]
```

其中metric_name和label_name必须遵循PromQL的格式规范要求。value是一个float格式的数据，timestamp的类型为int64（从1970-01-01 00:00:00以来的毫秒数），timestamp为可选默认为当前时间。具有相同metric_name的样本必须按照一个组的形式排列，并且每一行必须是唯一的指标名称和标签键值对组合。

需要特别注意的是对于histogram和summary类型的样本。需要按照以下约定返回样本数据：

- 类型为summary或者histogram的指标x，该指标所有样本的值的总和需要使用一个单独的x_sum指标表示。
- 类型为summary或者histogram的指标x，该指标所有样本的总数需要使用一个单独的x_count指标表示。
- 对于类型为summary的指标x，其不同分位数quantile所代表的样本，需要使用单独的x{quantile="y"}表示。
- 对于类型histogram的指标x为了表示其样本的分布情况，每一个分布需要使用x_bucket{le="y"}表示，其中y为当前分布的上位数。同时必须包含一个样本x_bucket{le="+Inf"}，并且其样本值必须和x_count相同。
- 对于histogram和summary的样本，必须按照分位数quantile和分布le的值的递增顺序排序。

以下是类型为histogram和summary的样本输出示例：

```
# A histogram, which has a pretty complex representation in the text format:
# HELP http_request_duration_seconds A histogram of the request duration.
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.05"} 24054
http_request_duration_seconds_bucket{le="0.1"} 33444
http_request_duration_seconds_bucket{le="0.2"} 100392
http_request_duration_seconds_bucket{le="+Inf"} 144320
http_request_duration_seconds_sum 53423
http_request_duration_seconds_count 144320

# Finally a summary, which has a complex representation, too:
# HELP rpc_duration_seconds A summary of the RPC duration in seconds.
# TYPE rpc_duration_seconds summary
rpc_duration_seconds{quantile="0.01"} 3102
rpc_duration_seconds{quantile="0.05"} 3272
rpc_duration_seconds{quantile="0.5"} 4773
rpc_duration_seconds_sum 1.7560473e+07
rpc_duration_seconds_count 2693
```

对于某些Prometheus还没有提供支持的编程语言，用户只需要按照以上规范返回响应的文本数据即可。

指定样本格式的版本

在Exporter响应的HTTP头信息中，可以通过Content-Type指定特定的规范版本，例如：

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Length: 2906
Content-Type: text/plain; version=0.0.4
Date: Sat, 17 Mar 2018 08:47:06 GMT
```

其中version用于指定Text-based的格式版本，当没有指定版本的时候，默认使用最新格式规范的版本。同时HTTP响应头还需要指定压缩格式为gzip。

常用Exporter

在第1章中，我们已经初步了解了Node Exporter的使用场景和方法。本小节，将会介绍更多常用的Exporter用法。包括如何监控容器运行状态，如何监控和评估MySQL服务的运行状态以及如何通过Prometheus实现基于网络探测的黑盒监控。

容器监控：cAdvisor

Docker是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的Linux/Windows/Mac机器上。容器镜像正成为一个新的标准化软件交付方式。

例如，可以通过以下命令快速在本地启动一个Nginx服务：

```
docker run -itd nginx
```

为了能够获取到Docker容器的运行状态，用户可以通过Docker的stats命令获取到当前主机上运行容器的统计信息，可以查看容器的CPU利用率、内存使用量、网络IO总量以及磁盘IO总量等信息。

```
$ docker stats
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK
 I/O           PIDS
9a1648bec3b2        0.30%               196KiB / 3.855GiB     0.00%               828B / 0B           827kB /
0B            1
```

除了使用命令以外，用户还可以通过Docker提供的HTTP API查看容器详细的监控统计信息。

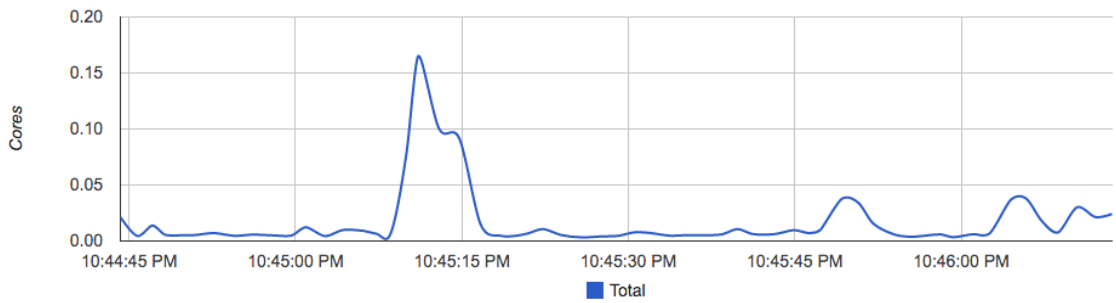
使用CAdvisor

CAdvisor是Google开源的一款用于展示和分析容器运行状态的可视化工具。通过在主机上运行CAdvisor用户可以轻松的获取到当前主机上容器的运行统计信息，并以图表的形式向用户展示。

在本地运行CAdvisor也非常简单，直接运行一下命令即可：

```
docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

通过访问http://localhost:8080可以查看，当前主机上容器的运行状态，如下所示：



CAdvisor是一个简单易用的工具，相比于使用Docker命令行工具，用户不用再登录到服务器中即可以可视化图表的形式查看主机上所有容器的运行状态。

而在多主机的情况下，在所有节点上运行一个CAdvisor再通过各自的UI查看监控信息显然不太方便，同时CAdvisor默认只保存2分钟的监控数据。好消息是CAdvisor已经内置了对Prometheus的支持。访问http://localhost:8080/metrics即可获取到标准的Prometheus监控样本输出：

```
# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version, OS version, docker version, cadvisor version & cadvisor revision.
# TYPE cadvisor_version_info gauge
cadvisor_version_info{cadvisorRevision="1e567c2",cadvisorVersion="v0.28.3",dockerVersion="17.09.1-ce",kernelVersion="4.9.49-moby",osVersion="Alpine Linux v3.4"} 1
# HELP container_cpu_load_average_10s Value of container cpu load average over the last 10 seconds.
# TYPE container_cpu_load_average_10s gauge
container_cpu_load_average_10s{container_label_maintainer="",id="/",image="",name=""} 0
container_cpu_load_average_10s{container_label_maintainer="",id="/docker",image="",name=""} 0
container_cpu_load_average_10s{container_label_maintainer="",id="/docker/15535a1e09b3a307b46d90400423d5b262ec84dc55b91ca9e7dd886f4f764ab3",image="busybox",name="lucid_shaw"} 0
container_cpu_load_average_10s{container_label_maintainer="",id="/docker/46750749b97bae47921d49dccdf9011b503e954312b8cffdec6268c249afa2dd",image="google/cadvisor:latest",name="cadvisor"} 0
container_cpu_load_average_10s{container_label_maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>",id="/docker/f51fd4d4f410965d3a0fd7e9f3250218911c1505e12960fb6dd7b889e75fc114",image="nginx",name="confident_brattain"} 0
```

下面表格中列举了一些CAdvisor中获取到的典型监控指标：

指标名称	类型	含义
container_cpu_load_average_10s	gauge	过去10秒容器CPU的平均负载
container_cpu_usage_seconds_total	counter	容器在每个CPU内核上的累积(单位：秒)
container_cpu_system_seconds_total	counter	System CPU累积占用时间（秒）
container_cpu_user_seconds_total	counter	User CPU累积占用时间（单位
container_fs_usage_bytes	gauge	容器中文件系统的使用量(单位
container_fs_limit_bytes	gauge	容器可以使用的文件系统总量(字节)
container_fs_reads_bytes_total	counter	容器累积读取数据的总量(单位
container_fs_writes_bytes_total	counter	容器累积写入数据的总量(单位
container_memory_max_usage_bytes	gauge	容器的最大内存使用量（单位
container_memory_usage_bytes	gauge	容器当前的内存使用量（单位

指标名称	类型	含义
container_spec_memory_limit_bytes	gauge	容器的内存使用量限制
machine_memory_bytes	gauge	当前主机的内存总量
container_network_receive_bytes_total	counter	容器网络累积接收数据总量（字节）
container_network_transmit_bytes_total	counter	容器网络累积传输数据总量（字节）

与Prometheus集成

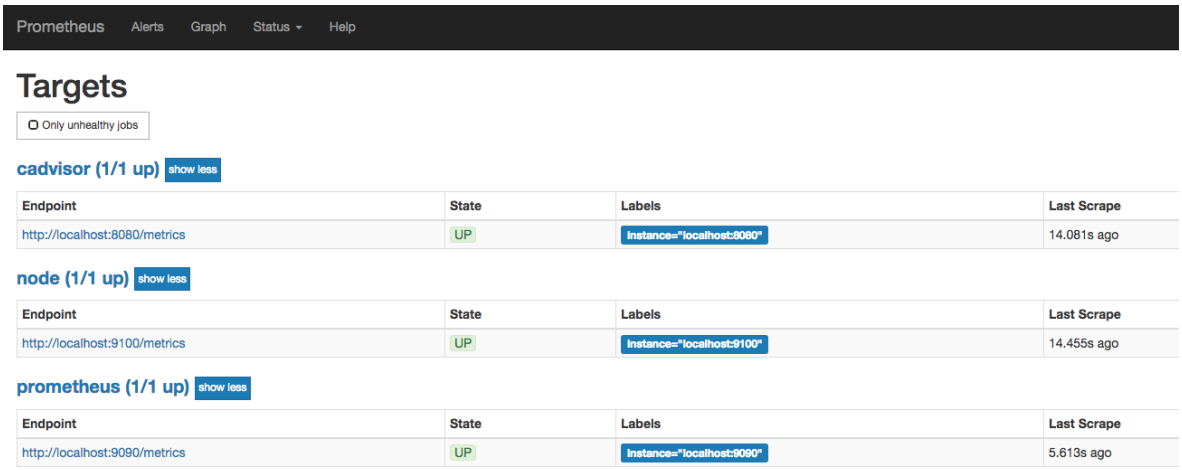
修改/etc/prometheus/prometheus.yml，将cAdvisor添加监控数据采集任务目标当中：

```
- job_name: cadvisor
  static_configs:
  - targets:
    - localhost:8080
```

启动Prometheus服务：

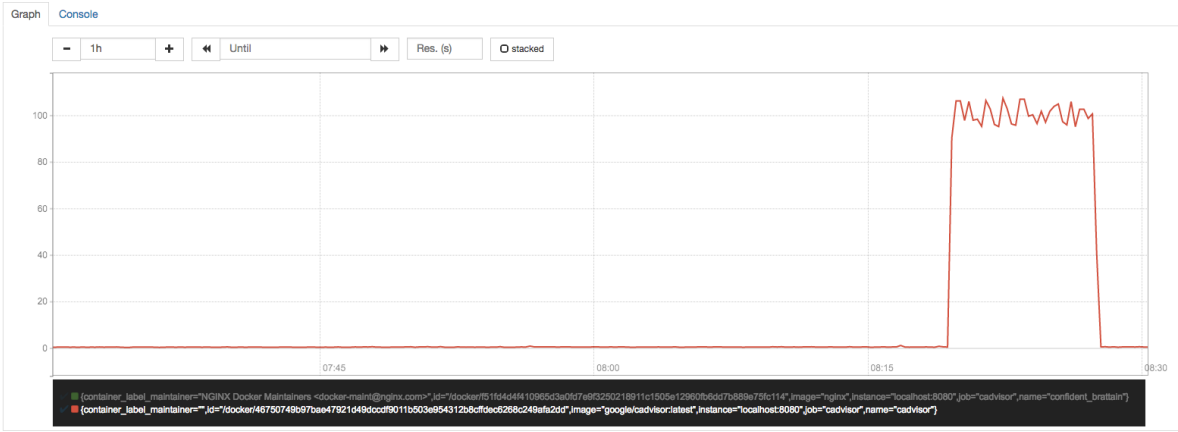
```
prometheus --config.file=/etc/prometheus/prometheus.yml --storage.tsdb.path=/data/prometheus
```

启动完成后，可以在Prometheus UI中查看到当前所有的Target状态：



当能够正常采集到cAdvisor的样本数据后，可以通过以下表达式计算容器的CPU使用率：

```
sum(irate(container_cpu_usage_seconds_total{image!=""}[1m])) without (cpu)
```

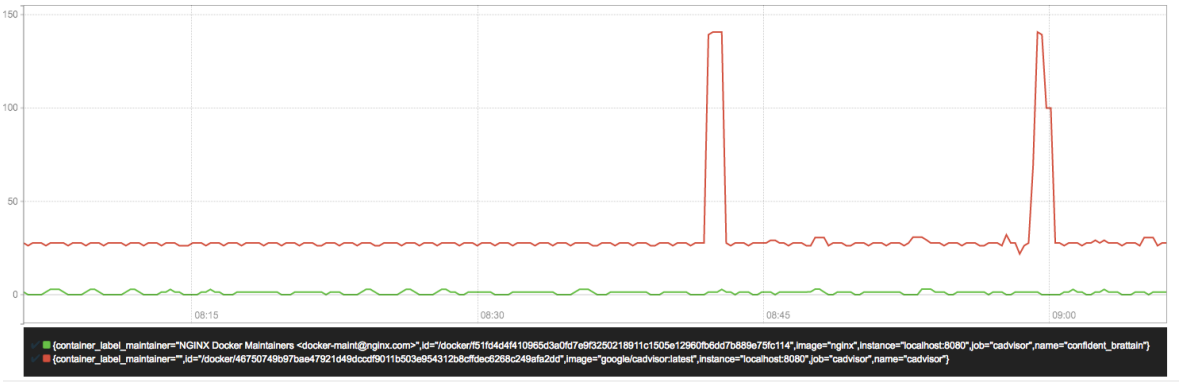


查询容器内存使用量（单位：字节）：

```
container_memory_usage_bytes{image!=""}
```

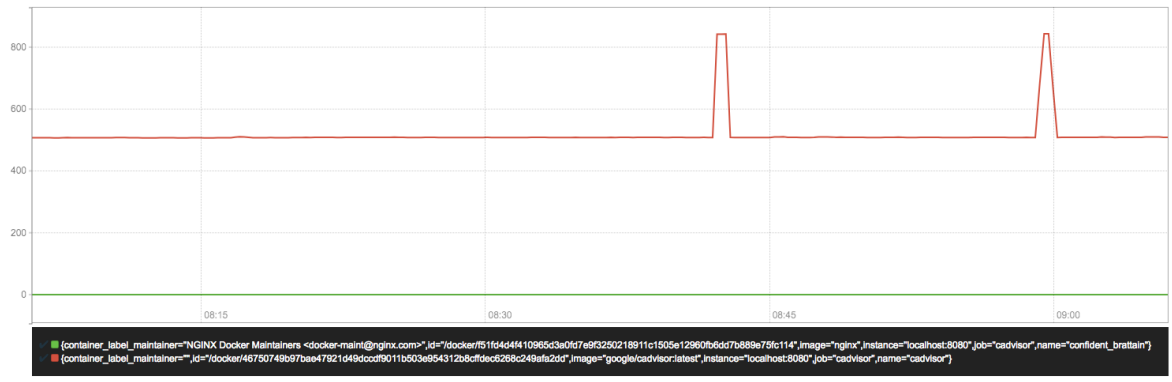
查询容器网络接收量速率（单位：字节/秒）：

```
sum(rate(container_network_receive_bytes_total{image!=""}[1m])) without (interface)
```



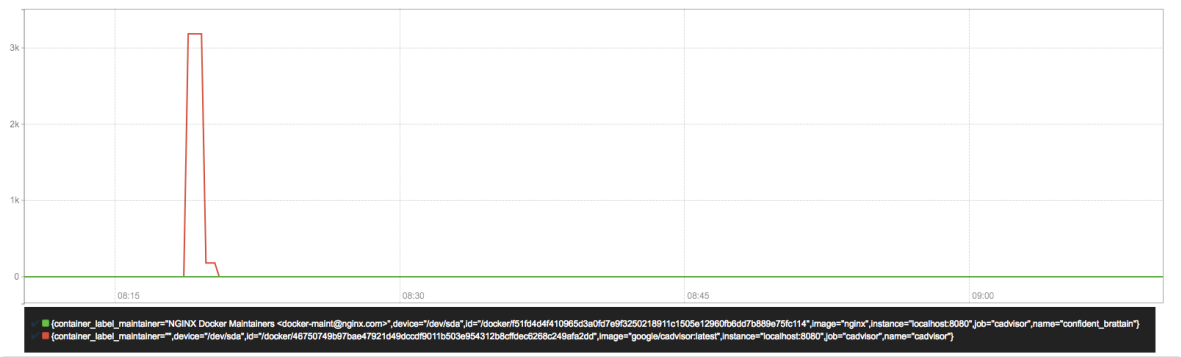
查询容器网络传输量速率（单位：字节/秒）：

```
sum(rate(container_network_transmit_bytes_total{image!=""}[1m])) without (interface)
```



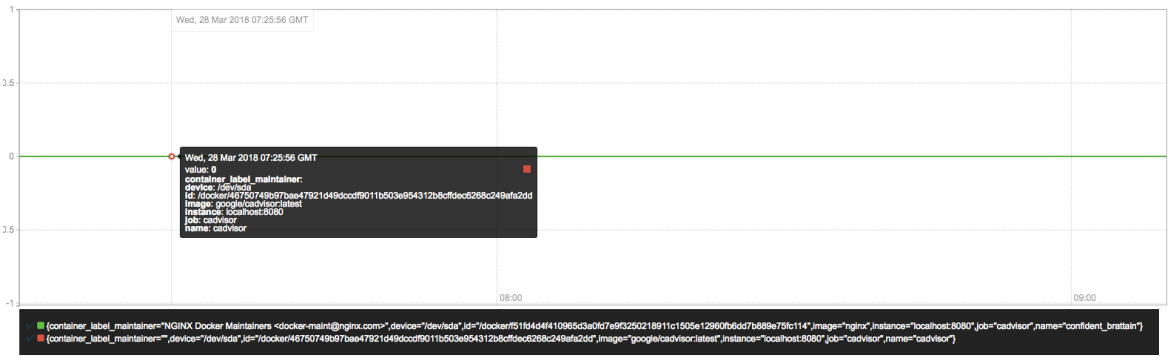
查询容器文件系统读取速率（单位：字节/秒）：

```
sum(rate(container_fs_reads_bytes_total{image!=""}[1m])) without (device)
```



查询容器文件系统写入速率（单位：字节/秒）：

```
sum(rate(container_fs_writes_bytes_total{image!=""}[1m])) without (device)
```



监控MySQL运行状态：MySQLD Exporter

监控MySQL运行状态：MySQLD Exporter

MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle旗下的产品。 MySQL是最流行的关系型数据库管理系统之一。数据库的稳定运行是保证业务可用性的关键因素之一。这一小节当中将介绍如何使用Prometheus提供的MySQLD Exporter实现对MySQL数据库性能以及资源利用率的监控和度量。

部署MySQLD Exporter

为了简化测试环境复杂度，这里使用Docker Compose定义并启动MySQL以及MySQLD Exporter：

```
version: '3'
services:
  mysql:
    image: mysql:5.7
    ports:
      - "3306:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=database
  mysqlexporter:
    image: prom/mysqld-exporter
    ports:
      - "9104:9104"
    environment:
      - DATA_SOURCE_NAME=root:password@(mysql:3306)/database
```

这里通过环境变量DATA_SOURCE_NAME方式定义监控目标。使用Docker Compose启动测试用的MySQL实例以及MySQLD Exporter:

```
$ docker-compose up -d
```

启动完成后，可以通过以下命令登录到MySQL容器当中，并执行MySQL相关的指令：

```
$ docker exec -it <mysql_container_id> mysql -uroot -ppassword
mysql>
```

可以通过http://localhost:9104访问MySQLD Exporter暴露的服务：

可以通过/metrics查看mysql_up指标判断当前MySQLD Exporter是否正常连接到了MySQL实例，当指标值为1时表示能够正常获取监控数据：

```
# HELP mysql_up Whether the MySQL server is up.
# TYPE mysql_up gauge
mysql_up 1
```

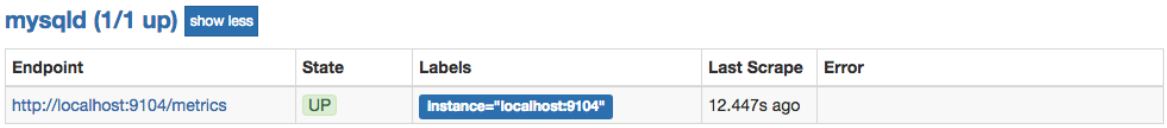
修改Prometheus配置文件/etc/prometheus/prometheus.yml，增加对MySQLD Exporter实例的采集任务配置:

```
- job_name: mysqld
  static_configs:
  - targets:
    - localhost:9104
```

启动Prometheus:

```
prometheus --config.file=/etc/prometheus/prometheus.yml --storage.tsdb.path=/data/prometheus
```

通过Prometheus的状态页，可以查看当前Target的状态：



为了确保数据库的稳定运行，通常会关注一下四个与性能和资源利用率相关的指标：查询吞吐量、连接情况、缓冲池使用情况以及查询执行性能等。

监控数据库吞吐量

对于数据库而言，最重要的工作就是实现对数据的增、删、改、查。为了衡量数据库服务器当前的吞吐量变化情况。在MySQL内部通过一个名为Questions的计数器，当客户端发送一个查询语句后，其值就会+1。可以通过以下MySQL指令查询Questions等服务器状态变量的值：

```
mysql> SHOW GLOBAL STATUS LIKE "Questions";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Questions     | 1326  |
+---------------+-------+
1 row in set (0.00 sec)
```

MySQLD Exporter中返回的样本数据中通过mysql_global_status_questions反映当前Questions计数器的大小：

```
# HELP mysql_global_status_questions Generic metric from SHOW GLOBAL STATUS.
# TYPE mysql_global_status_questions untyped
mysql_global_status_questions 1016
```

通过以下PromQL可以查看当前MySQL实例查询速率的变化情况，查询数量的突变往往暗示着可能发生了某些严重的问题，因此用于用户应该关注并且设置响应的告警规则，以及时获取该指标的变化情况：

```
rate(mysql_global_status_questions[2m])
```

一般还可以从监控读操作和写操作的执行情况进行判断。通过MySQL全局状态中的Com_select可以查询到当前服务器执行查询语句的总次数：相应的，也可以通过Com_insert、Com_update以及Com_delete的总量衡量当前服务器写操作的总次数，例如，可以通过以下指令查询当前MySQL实例insert语句的执行次数总量：

```
mysql> SHOW GLOBAL STATUS LIKE "Com_insert";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Com_insert    | 0     |
```

```
+---------------+-------+
1 row in set (0.00 sec)
```

从MySQLD Exporter的/metrics返回的监控样本中，可以通过global_status_commands_total获取当前实例各类指令执行的次数：

```
# HELP mysql_global_status_commands_total Total number of executed MySQL commands.
# TYPE mysql_global_status_commands_total counter
mysql_global_status_commands_total{command="admin_commands"} 0
mysql_global_status_commands_total{command="alter_db"} 0
mysql_global_status_commands_total{command="alter_db_upgrade"} 0
mysql_global_status_commands_total{command="select"} 10
mysql_global_status_commands_total{command="insert"} 2
mysql_global_status_commands_total{command="update"} 2
mysql_global_status_commands_total{command="delete"} 1
```

用户可以通过以下PromQL查看当前MySQL实例写操作速率的变化情况：

```
sum(rate(mysql_global_status_commands_total{command=~"insert|update|delete"}[2m])) without (command)
```

连接情况

在MySQL中通过全局设置max_connections限制了当前服务器允许的最大客户端连接数量。一旦可用连接数被用尽，新的客户端连接都会被直接拒绝。 因此当监控MySQL运行状态时，需要时刻关注MySQL服务器的连接情况。用户可以通过以下指令查看当前MySQL服务的max_connections配置：

```
mysql> SHOW VARIABLES LIKE 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
1 row in set (0.01 sec)
```

MySQL默认的最大链接数为151。临时调整最大连接数，可以通过以下指令进行设置：

```
SET GLOBAL max_connections = 200;
```

如果想永久化设置，则需要通过修改MySQL配置文件my.cnf，添加以下内容：

```
max_connections = 200
```

通过Global Status中的Threads_connected、Aborted_connects、Connection_errors_max_connections以及Threads_running可以查看当前MySQL实例的连接情况。

例如，通过以下指令可以直接当前MySQL实例的连接数：

```
mysql> SHOW GLOBAL STATUS LIKE "Threads_connected";
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Threads_connected | 1     |
```

```
+-------------------+-------+
1 row in set (0.00 sec)
```

当所有可用连接都被占用时，如果一个客户端尝试连接至MySQL，会出现“Too many connections(连接数过多)”错误，同时Connection_errors_max_connections的值也会增加。为了防止出现此类情况，你应该监控可用连接的数量，并确保其值保持在max_connections限制以内。同时如果Aborted_connects的数量不断增加时，说明客户端尝试连接到MySQL都失败了。此时可以通过Connection_errors_max_connections以及Connection_errors_internal分析连接失败的问题原因。

下面列举了与MySQL连接相关的监控指标：

- mysql_global_variables_max_connections： 允许的最大连接数；
- mysql_global_status_threads_connected： 当前开放的连接；
- mysql_global_status_threads_running：当前开放的连接；
- mysql_global_status_aborted_connects：当前开放的连接；
- mysql_global_status_connection_errors_total{error=”max_connections”}：由于超出最大连接数导致的错误；
- mysql_global_status_connection_errors_total{error=”internal”}：由于系统内部导致的错误；

通过PromQL查询当前剩余的可用连接数：

```
mysql_global_variables_max_connections - mysql_global_status_threads_connected
```

使用PromQL查询当前MySQL实例连接拒绝数：

```
mysql_global_status_aborted_connects
```

监控缓冲池使用情况

MySQL默认的存储引擎InnoDB使用了一片称为缓冲池的内存区域，用于缓存数据表以及索引的数据。 当缓冲池的资源使用超出限制后，可能会导致数据库性能的下降，同时很多查询命令会直接在磁盘中执行，导致磁盘I/O不断攀升。 因此，应该关注MySQL缓冲池的资源使用情况，并且在合理的时间扩大缓冲池的大小可以优化数据库的性能。

Innodb_buffer_pool_pages_total反映了当前缓冲池中的内存页的总页数。可以通过以下指令查看：

```
mysql> SHOW GLOBAL STATUS LIKE "Innodb_buffer_pool_pages_total";
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_buffer_pool_pages_total | 8191  |
+-------------------------------+-------+
1 row in set (0.02 sec)
```

MySQLD Exporter通过以下指标返回缓冲池中各类内存页的数量：

```
# HELP mysql_global_status_buffer_pool_pages Innodb buffer pool pages by state.
# TYPE mysql_global_status_buffer_pool_pages gauge
mysql_global_status_buffer_pool_pages{state="data"} 516
mysql_global_status_buffer_pool_pages{state="dirty"} 0
mysql_global_status_buffer_pool_pages{state="free"} 7675
mysql_global_status_buffer_pool_pages{state="misc"} 0
```

Innodb_buffer_pool_read_requests记录了正常从缓冲池读取数据的请求数量。可以通过以下指令查看：

```
mysql> SHOW GLOBAL STATUS LIKE "Innodb_buffer_pool_read_requests";
+---------------------------------+--------+
| Variable_name                   | Value  |
+---------------------------------+--------+
| Innodb_buffer_pool_read_requests | 797023 |
+---------------------------------+--------+
1 row in set (0.00 sec)
```

MySQLD Exporter通过以下指标返回缓冲池中Innodb_buffer_pool_read_requests的值：

```
# HELP mysql_global_status_innodb_buffer_pool_read_requests Generic metric from SHOW GLOBAL STATUS.
# TYPE mysql_global_status_innodb_buffer_pool_read_requests untyped
mysql_global_status_innodb_buffer_pool_read_requests 736711
```

当缓冲池无法满足时，MySQL只能从磁盘中读取数据。Innodb_buffer_pool_reads即记录了从磁盘读取数据的请求数量。通常来说从内存中读取数据的速度要比从磁盘中读取快很多，因此，如果Innodb_buffer_pool_reads的值开始增加，可能意味着数据库的性能有问题。 可以通过以下只能查看Innodb_buffer_pool_reads的数量

```
mysql> SHOW GLOBAL STATUS LIKE "Innodb_buffer_pool_reads";
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Innodb_buffer_pool_reads | 443   |
+-------------------------+-------+
1 row in set (0.00 sec)
```

在MySQLD Exporter中可以通过以下指标查看Innodb_buffer_pool_reads的数量。

```
# HELP mysql_global_status_innodb_buffer_pool_reads Generic metric from SHOW GLOBAL STATUS.
# TYPE mysql_global_status_innodb_buffer_pool_reads untyped
mysql_global_status_innodb_buffer_pool_reads 443
```

通过以上监控指标，以及实际监控的场景，我们可以利用PromQL快速建立多个监控项。

通过以下PromQL可以得到各个MySQL实例的缓冲池利用率。一般来说还需要结合Innodb_buffer_pool_reads的增长率情况来结合判断缓冲池大小是否合理：

```
(sum(mysql_global_status_buffer_pool_pages) by (instance) - sum(mysql_global_status_buffer_pool_pages{state="free"}) by (instance)) / sum(mysql_global_status_buffer_pool_pages) by (instance)
```

也可以通过以下PromQL计算2分钟内磁盘读取请求次数的增长率的变化情况：

```
rate(mysql_global_status_innodb_buffer_pool_reads[2m])
```

查询性能

MySQL还提供了一个Slow_queries的计数器，当查询的执行时间超过long_query_time的值后，计数器就会+1，其默认值为10秒，可以通过以下指令在MySQL中查询当前long_query_time的设置：

```
mysql> SHOW VARIABLES LIKE 'long_query_time';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
```

```
| long_query_time | 10.000000 |
+-----------------+-----------+
1 row in set (0.00 sec)
```

通过以下指令可以查看当前MySQL实例中Slow_queries的数量：

```
mysql> SHOW GLOBAL STATUS LIKE "Slow_queries";
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Slow_queries  | 0     |
+---------------+-------+
1 row in set (0.00 sec)
```

MySQLD Exporter返回的样本数据中，通过以下指标展示当前的Slow_queries的值：

```
# HELP mysql_global_status_slow_queries Generic metric from SHOW GLOBAL STATUS.
# TYPE mysql_global_status_slow_queries untyped
mysql_global_status_slow_queries 0
```

通过监控Slow_queries的增长率，可以反映出当前MySQL服务器的性能状态，可以通过以下PromQL查询Slow_queries的增长情况：

```
rate(mysql_global_status_slow_queries[2m])
```

在MySQL中还可以通过安装response time插件，从而支持记录查询时间区间的统计信息。启动该功能后MySQLD Exporter也会自动获取到相关数据，从而可以细化MySQL查询响应时间的分布情况。 感兴趣的读者可以自行尝试。

网络探测：Blackbox Exporter

在本章的前几个小节中我们主要介绍了Prometheus下如何进行白盒监控，我们监控主机的资源用量、容器的运行状态、数据库中间件的运行数据。 这些都是支持业务和服务的基础设施，通过白盒能够了解其内部的实际运行状态，通过对监控指标的观察能够预判可能出现的问题，从而对潜在的不确定因素进行优化。而从完整的监控逻辑的角度，除了大量的应用白盒监控以外，还应该添加适当的黑盒监控。黑盒监控即以用户的身份测试服务的外部可见性，常见的黑盒监控包括HTTP探针、TCP探针等用于检测站点或者服务的可访问性，以及访问效率等。

黑盒监控相较于白盒监控最大的不同在于黑盒监控是以故障为导向当故障发生时，黑盒监控能快速发现故障，而白盒监控则侧重于主动发现或者预测潜在的问题。一个完善的监控目标是要能够从白盒的角度发现潜在问题，能够在黑盒的角度快速发现已经发生的问题。

使用Blackbox Exporter

Blackbox Exporter是Prometheus社区提供的官方黑盒监控解决方案，其允许用户通过：HTTP、HTTPS、DNS、TCP以及ICMP的方式对网络进行探测。用户可以直接使用go get命令获取Blackbox Exporter源码并生成本地可执行文件：

```
go get prometheus/blackbox_exporter
```

运行Blackbox Exporter时，需要用户提供探针的配置信息，这些配置信息可能是一些自定义的HTTP头信息，也可能是探测时需要的一些TSL配置，也可能是探针本身的验证行为。在Blackbox Exporter每一个探针配置称为一个module，并且以YAML配置文件的形式提供给Blackbox Exporter。 每一个module主要包含以下配置内容，包括探针类型（prober）、验证访问超时时间（timeout）、以及当前探针的具体配置项：

```
# 探针类型：http、 tcp、 dns、 icmp.
prober: <prober_string>

# 超时时间
[ timeout: <duration> ]

# 探针的详细配置，最多只能配置其中的一个
[ http: <http_probe> ]
[ tcp: <tcp_probe> ]
[ dns: <dns_probe> ]
[ icmp: <icmp_probe> ]
```

下面是一个简化的探针配置文件blockbox.yml，包含两个HTTP探针配置项：

```
modules:
  http_2xx:
    prober: http
    http:
      method: GET
```

```
  http_post_2xx:
    prober: http
    http:
      method: POST
```

通过运行以下命令，并指定使用的探针配置文件启动Blockbox Exporter实例：

```
blackbox_exporter --config.file=/etc/prometheus/blackbox.yml
```

启动成功后，就可以通过访问http://127.0.0.1:9115/probe?module=http_2xx&target=baidu.com对baidu.com进行探测。这里通过在URL中提供module参数指定了当前使用的探针，target参数指定探测目标，探针的探测结果通过Metrics的形式返回：

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.011633673
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.117332275
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
# TYPE probe_http_content_length gauge
probe_http_content_length 81
# HELP probe_http_duration_seconds Duration of http request by phase, summed over all redirects
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0.055551141
probe_http_duration_seconds{phase="processing"} 0.049736019
probe_http_duration_seconds{phase="resolve"} 0.011633673
probe_http_duration_seconds{phase="tls"} 0
probe_http_duration_seconds{phase="transfer"} 3.8919e-05
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 0
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 0
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 200
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 1.1
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

从返回的样本中，用户可以获取站点的DNS解析耗时、站点响应时间、HTTP响应状态码等等和站点访问质量相关的监控指标，从而帮助管理员主动的发现故障和问题。

与Prometheus集成

接下来，只需要在Prometheus下配置对Blockbox Exporter实例的采集任务即可。最直观的配置方式：

```
- job_name: baidu_http2xx_probe
  params:
    module:
    - http_2xx
    target:
    - baidu.com
  metrics_path: /probe
  static_configs:
  - targets:
    - 127.0.0.1:9115
- job_name: prometheus_http2xx_probe
  params:
    module:
    - http_2xx
    target:
    - prometheus.io
  metrics_path: /probe
  static_configs:
  - targets:
    - 127.0.0.1:9115
```

这里分别配置了名为baidu_http2x_probe和prometheus_http2xx_probe的采集任务，并且通过params指定使用的探针（module）以及探测目标（target）。

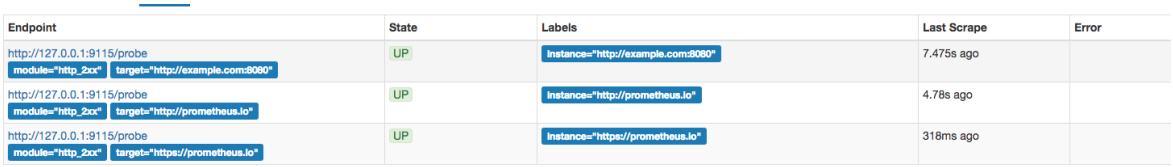
那问题就来了，假如我们有N个目标站点且都需要M种探测方式，那么Prometheus中将包含N * M个采集任务，从配置管理的角度来说显然是不可接受的。 在第7章的“服务发现与Relabel”小节，我们介绍了Prometheus的Relabeling能力，这里我们也可以采用Relabling的方式对这些配置进行简化，配置方式如下：

```
scrape_configs:
  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
        - http://prometheus.io    # Target to probe with http.
        - https://prometheus.io   # Target to probe with https.
        - http://example.com:8080 # Target to probe with http on port 8080.
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 127.0.0.1:9115
```

这里针对每一个探针服务（如http_2xx）定义一个采集任务，并且直接将任务的采集目标定义为我们需要探测的站点。在采集样本数据之前通过relabel_configs对采集任务进行动态设置。

- 第1步，根据Target实例的地址，写入`__param_target`标签中。`__param_<name>`形式的标签表示，在采集任务时会在请求目标地址中添加`<name>`参数，等同于params的设置；
- 第2步，获取*param*target的值，并覆写到instance标签中；
- 第3步，覆写Target实例的`__address__`标签值为BlockBox Exporter实例的访问地址。

通过以上3个relabel步骤，即可大大简化Prometheus任务配置的复杂度：



Endpoint	State	Labels	Last Scrape	Error
http://127.0.0.1:9115/probe module="http_2xx" target="http://example.com:8080"	UP	instance="http://example.com:8080"	7.475s ago	
http://127.0.0.1:9115/probe module="http_2xx" target="http://prometheus.io"	UP	instance="http://prometheus.io"	4.78s ago	
http://127.0.0.1:9115/probe module="http_2xx" target="https://prometheus.io"	UP	instance="https://prometheus.io"	318ms ago	

接下来，我们将详细介绍Blackbox中常用的HTTP探针使用方式

HTTP探针

HTTP探针是进行黑盒监控时最常用的探针之一，通过HTTP探针能够网站或者HTTP服务建立有效的监控，包括其本身的可用性，以及用户体验相关的如响应时间等等。除了能够在服务出现异常的时候及时报警，还能帮助系统管理员分析和优化网站体验。

在上一小节讲过，Blockbox Exporter中所有的探针均是以Module的信息进行配置。如下所示，配置了一个最简单的HTTP探针：

```
modules:
  http_2xx_example:
    prober: http
    http:
```

通过prober配置项指定探针类型。配置项http用于自定义探针的探测方式，这里有没对http配置项添加任何配置，表示完全使用HTTP探针的默认配置，该探针将使用HTTP GET的方式对目标服务进行探测，并且验证返回状态码是否为2XX，是则表示验证成功，否则失败。

自定义HTTP请求

HTTP服务通常会以不同的形式对外展现，有些可能就是一些简单的网页，而有些则可能是一些基于REST的API服务。 对于不同类型的HTTP的探测需要管理员能够对HTTP探针的行为进行更多的自定义设置，包括：HTTP请求方法、HTTP头信息、请求参数等。对于某些启用了安全认证的服务还需要能够对HTTP探测设置相应的Auth支持。对于HTTPS类型的服务还需要能够对证书进行自定义设置。

如下所示，这里通过method定义了探测时使用的请求方法，对于一些需要请求参数的服务，还可以通过headers定义相关的请求头信息，使用body定义请求内容：

```
http_post_2xx:
    prober: http
    timeout: 5s
    http:
      method: POST
      headers:
        Content-Type: application/json
      body: '{}'
```

如果HTTP服务启用了安全认证，Blockbox Exporter内置了对basic_auth的支持，可以直接设置相关的认证信息即可：

```
http_basic_auth_example:
    prober: http
    timeout: 5s
    http:
      method: POST
      headers:
        Host: "login.example.com"
      basic_auth:
```

```
      username: "username"
      password: "mysecret"
```

对于使用了Bear Token的服务也可以通过bearer_token配置项直接指定令牌字符串，或者通过bearer_token_file指定令牌文件。

对于一些启用了HTTPS的服务，但是需要自定义证书的服务，可以通过tls_config指定相关的证书信息：

```
http_custom_ca_example:
  prober: http
  http:
    method: GET
    tls_config:
      ca_file: "/certs/my_cert.crt"
```

自定义探针行为

在默认情况下HTTP探针只会对HTTP返回状态码进行校验，如果状态码为2XX（200 <= StatusCode < 300）则表示探测成功，并且探针返回的指标probe_success值为1。

如果用户需要指定HTTP返回状态码，或者对HTTP版本有特殊要求，如下所示，可以使用valid_http_versions和valid_status_codes进行定义：

```
 http_2xx_example:
  prober: http
  timeout: 5s
  http:
    valid_http_versions: ["HTTP/1.1", "HTTP/2"]
    valid_status_codes: []
```

默认情况下，Blockbox返回的样本数据中也会包含指标probe_http_ssl，用于表明当前探针是否使用了SSL：

```
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 0
```

而如果用户对于HTTP服务是否启用SSL有强制的标准。则可以使用fail_if_ssl和fail_if_not_ssl进行配置。fail_if_ssl为true时，表示如果站点启用了SSL则探针失败，反之成功。fail_if_not_ssl刚好相反。

```
 http_2xx_example:
  prober: http
  timeout: 5s
  http:
    valid_status_codes: []
    method: GET
    no_follow_redirects: false
    fail_if_ssl: false
    fail_if_not_ssl: false
```

除了基于HTTP状态码，HTTP协议版本以及是否启用SSL作为控制探针探测行为成功与否的标准以外，还可以匹配HTTP服务的响应内容。使用fail_if_matches_regexp和fail_if_not_matches_regexp用户可以定义一组正则表达式，用于验证HTTP返回内容是否符合或者不符合正则表达式的内容。

```
 http_2xx_example:
  prober: http
```

```
    timeout: 5s
    http:
      method: GET
      fail_if_matches_regexp:
        - "Could not connect to database"
      fail_if_not_matches_regexp:
        - "Download the latest version here"
```

最后需要提醒的时，默认情况下HTTP探针会走IPV6的协议。 在大多数情况下，可以使用preferred_ip_protocol=ip4强制通过IPV4的方式进行探测。在Bloackbox响应的监控样本中，也会通过指标probe_ip_protocol，表明当前的协议使用情况：

```
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
```

除了支持对HTTP协议进行网络探测以外，Blackbox还支持对TCP、DNS、ICMP等其他网络协议，感兴趣的读者可以从Blackbox的Github项目中获取更多使用信息  https://github.com/prometheus/blackbox_exporter。

使用Java自定义Exporter

本小节将带领读者了解Promrtheus提供的client_java的基本用法，并且在最后在Spring Boot应用程序中使用client_java，直接在应用程序层面提供对Prometheus的支持。

使用Client Java构建Exporter程序

client_java是Prometheus针对JVM类开发语言的client library库，我们可以直接基于client_java用户可以快速实现独立运行的Exporter程序，也可以在我们的项目源码中集成client_java以支持Prometheus。

自定义Collector

在client_java的simpleclient模块中提供了自定义监控指标的核心接口。

如果使用Gradle作为项目构建工具，可以通过向build.gradle添加simpleclient依赖：

```
compile 'io.prometheus:simpleclient:0.3.0'
```

当无法直接修改监控目标时，可以通过自定义Collector的方式，实现对监控样本收集，该收集器需要实现collect()方法并返回一组监控样本，如下所示：

```
public class YourCustomCollector extends Collector {
    public List<MetricFamilySamples> collect() {
        List<MetricFamilySamples> mfs = new ArrayList<MetricFamilySamples>();

        String metricName = "my_guage_1";

        // Your code to get metrics

        MetricFamilySamples.Sample sample = new MetricFamilySamples.Sample(metricName, Arrays.asList("l1"), A
rrays.asList("v1"), 4);
        MetricFamilySamples.Sample sample2 = new MetricFamilySamples.Sample(metricName, Arrays.asList("l1",
"l2"), Arrays.asList("v1", "v2"), 3);

        MetricFamilySamples samples = new MetricFamilySamples(metricName, Type.GAUGE, "help", Arrays.asList(s
ample, sample2));

        mfs.add(samples);
        return mfs;
    }
}
```

这里定义了一个名为my_guage的监控指标，该监控指标的所有样本数据均转换为一个MetricFamilySamples.Sample实例，该实例中包含了该样本的指标名称、标签名数组、标签值数组以及样本数据的值。

监控指标my_guage的所有样本值，需要持久化到一个MetricFamilySamples实例中，MetricFamilySamples指定了当前监控指标的名称、类型、注释信息等。需要注意的是MetricFamilySamples中所有样本的名称必须保持一致，否则生成的数据将无法符合Prometheus的规范。

直接使用MetricFamilySamples.Sample和MetricFamilySamples的方式适用于当某监控指标的样本之间的标签可能不一致的情况，例如，当监控容器时，不同容器实例可能包含一些自定义的标签，如果需要将这些标签反应到样本上，那么每个样本的标签则不可能保持一致。而如果所有样本的是一致的情况下，我们还可以使用client_java针对不同指标类型的实现GaugeMetricFamily，CounterMetricFamily，SummaryMetricFamily等，例如：

```
class YourCustomCollector2 extends Collector {
  List<MetricFamilySamples> collect() {
    List<MetricFamilySamples> mfs = new ArrayList<MetricFamilySamples>();

    // With no labels.
    mfs.add(new GaugeMetricFamily("my_gauge_2", "help", 42));
```

```
    // With labels
    GaugeMetricFamily labeledGauge = new GaugeMetricFamily("my_other_gauge", "help", Arrays.asList("labelnam
e"));
    labeledGauge.addMetric(Arrays.asList("foo"), 4);
    labeledGauge.addMetric(Arrays.asList("bar"), 5);
    mfs.add(labeledGauge);

    return mfs;
  }
}
```

使用HTTP Server暴露样本数据

client_java下的simpleclient_httpserver模块实现了一个简单的HTTP服务器，当向该服务器发送获取样本数据的请求后，它会自动调用所有Collector的collect()方法，并将所有样本数据转换为Prometheus要求的数据输出格式规范。如果用户使用了Gradle构建项目，可以添加以下依赖：

```
compile 'io.prometheus:simpleclient_httpserver:0.3.0'
```

添加依赖之后，就可以在Exporter程序的main方法中启动一个HTTPServer实例：

```
public class CustomExporter {
    public static void main(String[] args) throws IOException {
        HTTPServer server = new HTTPServer(1234);
    }
}
```

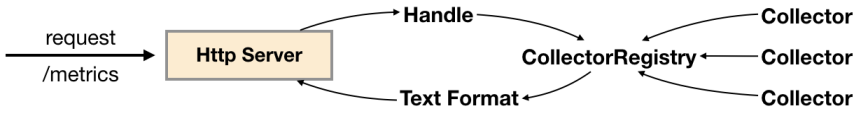
而在启动之前，别忘记调用Collector的register()方法。否则HTTPServer是找不到任何的Collector实例的：

```
new YourCustomCollector().register();
new YourCustomCollector2().register();
```

运行CustomExporter并访问http://127.0.0.1:1234/metrics，即可获取到以下数据：

```
$ curl http://127.0.0.1:1234/metrics
# HELP my_gauge help
# TYPE my_gauge gauge
my_gauge 42.0
# HELP my_other_gauge help
# TYPE my_other_gauge gauge
my_other_gauge{labelname="foo",} 4.0
my_other_gauge{labelname="bar",} 5.0
# HELP my_guage help
# TYPE my_guage gauge
my_guage{l1="v1",} 4.0
my_guage{l1="v1",l2="v2",} 3.0
```

当然HTTPServer中并不存在什么黑魔法，其内部实现如下所示：



当调用Collector实例register()方法时，会将该实例保存到CollectorRegistry当中，CollectorRegistry负责维护当前系统中所有的Collector实例。 HTTPServer在接收到HTTP请求之后，会从CollectorRegistry中拿到所有的Collector实例，并调用其collect()方法获取所有样本，最后格式化为Prometheus的标准输出。

除了直接使用HTTPServer以外暴露样本数据以外，client_java中还提供了对Spring Boot、Spring Web以及Servlet的支持。

使用内置的Collector

通过client_java中定义的标准接口，用户可以快速实现自己的监控数据收集器，并通过HTTPServer将样本数据输出给Prometheus。除了提供接口规范以外，client_java还提供了多个内置的Collector模块，以simpleclient_hotspot为例，该模块中内置了对JVM虚拟机运行状态（GC，内存池，JMX，类加载，线程池等）数据的Collector实现，用户可以通过在Gradle中添加以下依赖，导入simpleclient_hotspot：

```
compile 'io.prometheus:simpleclient_hotspot:0.3.0'
```

通过调用io.prometheus.client.hotspot.DefaultExport的initialize方法注册该模块中所有的Collector实例：

```
DefaultExports.initialize();
```

重新运行CustomExporter，并获取样本数据：

```
$ curl http://127.0.0.1:1234/metrics
# HELP jvm_buffer_pool_used_bytes Used bytes of a given JVM buffer pool.
# TYPE jvm_buffer_pool_used_bytes gauge
jvm_buffer_pool_used_bytes{pool="direct",} 8192.0
jvm_buffer_pool_used_bytes{pool="mapped",} 0.0
```

除了之前自定义的监控指标以外，在响应内容中还会得到当前JVM的运行状态数据。在client_java项目中除了使用内置了对JVM监控的Collector以外，还实现了对Hibernate，Guava Cache，Jetty，Log4j、Logback等监控数据收集的支持。用户只需要添加相应的依赖，就可以直接进行使用。

在业务代码中进行监控埋点

在client_java中除了使用Collector直接采集样本数据以外，还直接提供了对Prometheus中4种监控类型的实现分别是：Counter、Gauge、Summary和Histogram。 基于这些实现，开发人员可以非常方便的在应用程序的业务流程中进行监控埋点。

简单类型Gauge和Counter

以Gauge为例，当我们需要监控某个业务当前正在处理的请求数量，可以使用以下方式实现：

```
public class YourClass {

    static final Gauge inprogressRequests = Gauge.build()
            .name("inprogress_requests").help("Inprogress requests.").register();
```

```
    void processRequest() {
        inprogressRequests.inc();
        // Your code here.
        inprogressRequests.dec();
    }

}
```

Gauge继承自Collector，registoer()方法会将该Gauge实例注册到CollectorRegistry中。这里创建了一个名为inprogress_requests的监控指标，其注释信息为"Inprogress requests"。

Gauge对象主要包含两个方法inc()和dec()，分别用于计数器+1和-1。

如果监控指标中还需要定义标签，则可以使用Gauge构造器的labelNames()方法，声明监控指标的标签，同时在样本计数时，通过指标的labels()方法指定标签的值，如下所示：

```
public class YourClass {

    static final Gauge inprogressRequests = Gauge.build()
            .name("inprogress_requests")
            .labelNames("method")
            .help("Inprogress requests.").register();

    void processRequest() {
        inprogressRequests.labels("get").inc();
        // Your code here.
        inprogressRequests.labels("get").dec();
    }

}
```

Counter与Gauge的使用方法一致，唯一的区别在于Counter实例只包含一个inc()方法，用于计数器+1。

复杂类型Summary和Histogram

Summary和Histogram用于统计和分析样本的分布情况。如下所示，通过Summary可以将HTTP请求的字节数以及请求处理时间作为统计样本，直接统计其样本的分布情况。

```
class YourClass {
  static final Summary receivedBytes = Summary.build()
     .name("requests_size_bytes").help("Request size in bytes.").register();
  static final Summary requestLatency = Summary.build()
     .name("requests_latency_seconds").help("Request latency in seconds.").register();

  void processRequest(Request req) {
    Summary.Timer requestTimer = requestLatency.startTimer();
    try {
      // Your code here.
    } finally {
      receivedBytes.observe(req.size());
      requestTimer.observeDuration();
    }
  }
}
```

除了使用Timer进行计时以外，Summary实例也提供了timer()方法，可以对线程或者Lamda表达式运行时间进行统计：

```
class YourClass {
  static final Summary requestLatency = Summary.build()
    .name("requests_latency_seconds").help("Request latency in seconds.").register();

  void processRequest(Request req) {
    requestLatency.timer(new Runnable() {
      public abstract void run() {
        // Your code here.
      }
    });

    // Or the Java 8 lambda equivalent
    requestLatency.timer(() -> {
      // Your code here.
    });
  }
}
```

Summary和Histogram的用法基本保持一致，区别在于Summary可以指定在客户端统计的分位数，如下所示：

```
static final Summary requestLatency = Summary.build()
    .quantile(0.5, 0.05)   // 其中0.05为误差
    .quantile(0.9, 0.01)   // 其中0.01为误差
    .name("requests_latency_seconds").help("Request latency in seconds.").register();
```

对于Histogram而言，默认的分布桶为[.005, .01, .025, .05, .075, .1, .25, .5, .75, 1, 2.5, 5, 7.5, 10]，如果需要指定自定义的桶分布，可以使用buckets()方法指定，如下所示：

```
static final Histogram requestLatency = Histogram.build()
           .name("requests_latency_seconds").help("Request latency in seconds.")
           .buckets(0.1, 0.2, 0.4, 0.8)
           .register();
```

与PushGateway集成

对于一些短周期或者临时采集的样本数据，client_java还提供了对PushGateway的支持：

添加依赖：

```
compile 'io.prometheus:simpleclient_pushgateway:0.3.0'
```

如下所示，PushGateway的实现类可以从所有注册到defaultRegistry的Collector实例中获取样本数据并直接推送 到外部部署的PushGateway服务中。

```
public class PushGatewayIntegration {

    public void push() throws IOException {
        CollectorRegistry registry = CollectorRegistry.defaultRegistry;
        PushGateway pg = new PushGateway("127.0.0.1:9091");
        pg.pushAdd(registry, "my_batch_job");
    }

}
```

在应用中内置Prometheus支持

本小节将以Spring Boot为例，介绍如何在应用代码中集成client_java。

添加Prometheus Java Client相关的依赖：

```
dependencies {
    compile 'io.prometheus:simpleclient:0.0.24'
    compile "io.prometheus:simpleclient_spring_boot:0.0.24"
    compile "io.prometheus:simpleclient_hotspot:0.0.24"
}
```

通过注解@EnablePrometheusEndpoint启用Prometheus Endpoint，这里同时使用了simpleclient_hotspot中提供的DefaultExporter。该Exporter会在metrics endpoint中统计当前应用JVM的相关信息：

```
@SpringBootApplication
@EnablePrometheusEndpoint
public class SpringApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }

    @Override
    public void run(String... strings) throws Exception {
        DefaultExports.initialize();
    }
}
```

默认情况下Prometheus暴露的metrics endpoint为 /prometheus，可以通过endpoint配置进行修改：

```
endpoints:
  prometheus:
    id: metrics
  metrics:
    id: springmetrics
    sensitive: false
    enabled: true
```

启动应用程序访问http://localhost:8080/metrics可以看到以下输出内容：

```
# HELP jvm_gc_collection_seconds Time spent in a given JVM garbage collector in seconds.
# TYPE jvm_gc_collection_seconds summary
jvm_gc_collection_seconds_count{gc="PS Scavenge",} 11.0
jvm_gc_collection_seconds_sum{gc="PS Scavenge",} 0.18
jvm_gc_collection_seconds_count{gc="PS MarkSweep",} 2.0
jvm_gc_collection_seconds_sum{gc="PS MarkSweep",} 0.121
# HELP jvm_classes_loaded The number of classes that are currently loaded in the JVM
# TYPE jvm_classes_loaded gauge
jvm_classes_loaded 8376.0
# HELP jvm_classes_loaded_total The total number of classes that have been loaded since the JVM has started e
xecution
# TYPE jvm_classes_loaded_total counter
...
```

添加拦截器，为监控埋点做准备

除了获取应用JVM相关的状态以外，我们还可能需要添加一些自定义的监控Metrics实现对系统性能，以及业务状态进行采集，以提供日后优化的相关支撑数据。首先我们使用拦截器处理对应用的所有请求。

继承WebMvcConfigurerAdapter类并复写addInterceptors方法，对所有请求/**添加拦截器

```
@SpringBootApplication
@EnablePrometheusEndpoint
public class SpringApplication extends WebMvcConfigurerAdapter implements CommandLineRunner {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new PrometheusMetricsInterceptor()).addPathPatterns("/**");
    }
}
```

PrometheusMetricsInterceptor继承自HandlerInterceptorAdapter，通过复写父方法preHandle和afterCompletion可以拦截一个HTTP请求生命周期的不同阶段：

```
public class PrometheusMetricsInterceptor extends HandlerInterceptorAdapter {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        return super.preHandle(request, response, handler);
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exc
eption ex) throws Exception {
        super.afterCompletion(request, response, handler, ex);
    }
}
```

自定义监控指标

一旦PrometheusMetricsInterceptor能够成功拦截和处理请求之后，我们就可以使用client java自定义多种监控指标。

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

使用Counter.build()创建Counter类型的监控指标，并且通过name()方法定义监控指标的名称，通过labelNames()定义该指标包含的标签。最后通过register()将该指标注册到Collector的defaultRegistry中中。

```
public class PrometheusMetricsInterceptor extends HandlerInterceptorAdapter {

    static final Counter requestCounter = Counter.build()
            .name("io_namespace_http_requests_total").labelNames("path", "method", "code")
            .help("Total requests.").register();

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exc
eption ex) throws Exception {
        String requestURI = request.getRequestURI();
        String method = request.getMethod();
        int status = response.getStatus();

        requestCounter.labels(requestURI, method, String.valueOf(status)).inc();
        super.afterCompletion(request, response, handler, ex);
```

```
    }
}
```

在afterCompletion方法中，可以获取到当前请求的请求路径、请求方法以及状态码。 这里通过labels指定了当前样本各个标签对应的值，最后通过.inc()计数器+1：

```
requestCounter.labels(requestURI, method, String.valueOf(status)).inc();
```

通过指标io_namespace_http_requests_total我们可以实现：

- 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))
```

使用Gauge可以反映应用的**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。这里我们使用Gauge记录当前应用正在处理的Http请求数量。

```
public class PrometheusMetricsInterceptor extends HandlerInterceptorAdapter {

    ...省略的代码
    static final Gauge inprogressRequests = Gauge.build()
            .name("io_namespace_http_inprogress_requests").labelNames("path", "method")
            .help("Inprogress requests.").register();

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        ...省略的代码
        // 计数器+1
        inprogressRequests.labels(requestURI, method).inc();
        return super.preHandle(request, response, handler);
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exc
eption ex) throws Exception {
        ...省略的代码
        // 计数器-1
        inprogressRequests.labels(requestURI, method).dec();

        super.afterCompletion(request, response, handler, ex);
```

```
    }
}
```

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。

```
public class PrometheusMetricsInterceptor extends HandlerInterceptorAdapter {

    static final Histogram requestLatencyHistogram = Histogram.build().labelNames("path", "method", "code")
            .name("io_namespace_http_requests_latency_seconds_histogram").help("Request latency in seconds.")
            .register();

    private Histogram.Timer histogramRequestTimer;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        ...省略的代码
        histogramRequestTimer = requestLatencyHistogram.labels(requestURI, method, String.valueOf(status)).st
artTimer();
        ...省略的代码
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exc
eption ex) throws Exception {
        ...省略的代码
        histogramRequestTimer.observeDuration();
        ...省略的代码
    }
}
```

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",} 13.10767080300000
1
```

- 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：rate(basename_sum[5m]) / rate(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<= 分位数Quantiles <= 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Sumamry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。

```
public class PrometheusMetricsInterceptor extends HandlerInterceptorAdapter {

    static final Summary requestLatency = Summary.build()
            .name("io_namespace_http_requests_latency_seconds_summary")
            .quantile(0.5, 0.05)
            .quantile(0.9, 0.01)
            .labelNames("path", "method", "code")
            .help("Request latency in seconds.").register();


    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        ...省略的代码
        requestTimer = requestLatency.labels(requestURI, method, String.valueOf(status)).startTimer();
        ...省略的代码
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exc
eption ex) throws Exception {
        ...省略的代码
        requestTimer.observeDuration();
        ...省略的代码
    }
}
```

使用Summary指标，会自动创建多个时间序列：

- 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请求响应时间的中位数是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请求响应时间的9分位数是8.003261666s
io_namespace_http_requests_latency_seconds_summary{path="/",method="GET",code="200",quantile="0.9",} 8.003261666
```

使用Collector暴露其它指标

除了在拦截器中使用Prometheus提供的Counter,Summary,Gauage等构造监控指标以外，我们还可以通过自定义的Collector实现对相关业务指标的暴露。例如，我们可以通过自定义Collector直接从应用程序的数据库中统计监控指标.

```
@SpringBootApplication
@EnablePrometheusEndpoint
public class SpringApplication extends WebMvcConfigurerAdapter implements CommandLineRunner {

    @Autowired
    private CustomExporter customExporter;

    ...省略的代码

    @Override
    public void run(String... args) throws Exception {
        ...省略的代码
        customExporter.register();
    }
}
```

CustomExporter集成自io.prometheus.client.Collector，在调用Collector的register()方法后，当访问/metrics时，则会自动从Collector的collection()方法中获取采集到的监控指标。

由于这里CustomExporter存在于Spring的IOC容器当中，这里可以直接访问业务代码，返回需要的业务相关的指标。

```
import io.prometheus.client.Collector;
import io.prometheus.client.GaugeMetricFamily;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

@Component
public class CustomExporter extends Collector {
    @Override
    public List<MetricFamilySamples> collect() {
        List<MetricFamilySamples> mfs = new ArrayList<>();
```

```
        # 创建metrics指标
        GaugeMetricFamily labeledGauge =
                new GaugeMetricFamily("io_namespace_custom_metrics", "custom metrics", Collections.singletonL
ist("labelname"));

        # 设置指标的label以及value
        labeledGauge.addMetric(Collections.singletonList("labelvalue"), 1);

        mfs.add(labeledGauge);
        return mfs;
    }
}
```

这里也可以使用CounterMetricFamily，SummaryMetricFamily声明其它的指标类型。

小结

Prometheus负责数据的统一收集并且提供统一的查询接口PromQL，而所有监控数据的产生则是由Exporter来进行实现，对于任何能够提供Promethues标准的监控样本的程序都可以称为Exporter。Exporter可以是一个单独的为了采集特定数据而构建的应用程序，也可以直接内置于特定的系统当中。

数据与可视化

“You can’t fix what you can’t see”。可视化是监控的核心目标之一，在本章中我们将介绍Prometheus下的可视化技术。例如，Prometheus自身提供的Console Template能力以及Grafana这一可视化工具实现监控数据可视化。Prometheus UI提供了基本的数据可视化能力，可以帮助用户直接使用PromQL查询数据，并将数据通过可视化图表的方式进行展示，而实际的应用场景中往往不同的人对于可视化的需求不一样，关注的指标也不一样，因此我们需要能够有能力，构建出不同的可视化报表页面。 本章学习的内容就主要解决以上问题。

本章的主要内容：

- 使用Console Template创建可视化页面
- 使用Grafana创建更精美的数据仪表盘

使用Console Template

使用Console Template

在第1章以及第2章的内容中，读者已经对Prometheus已经有了一个相对完成的认识，并且我们已经学习了如何通过PromQL对时间序列数据进行查询和分析，并且通过Prometheus中的Graph面板查询数据形成图表。但是缺点也很明显，这些查询结果都是临时的，无法持久化的，更别说我们想实时关注某些特定监控指标的变化趋势。

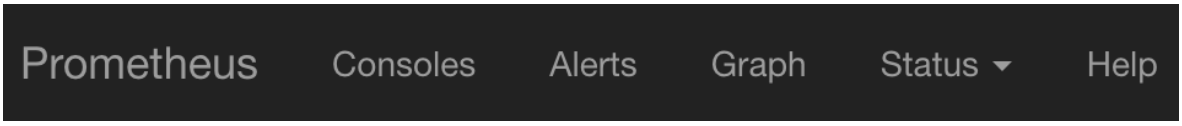
为了简化这些问题Prometheus内置了一个简单的解决方案 `Console Template` ,它允许用户通过Go模板语言创建任意的控制台界面，并且通过Prometheus Server对外提供访问路径。

快速开始

首先我们先从一个小例子开始，创建我们的第一个Console Template页面。与Console Template有关的两个启动参数为 `--web.console.libraries` 和 `--web.console.templates` ,其分别指定页面组件以及页面的存储路径。默认情况下其分别指向Prometheus当前安装路径的 `console_libraries` 和 `consoles` 目录。

Prometheus在 `console_libraries` 目录中已经内置了一些基本的界面组件，用户可以直接使用。

在 `consoles` 目录下创建index.html文件后，刷新Prometheus界面可以看到在顶部菜单中多了一个Consoles菜单项，如下所示。该选项默认指向 `consoles/index.html` 文件：



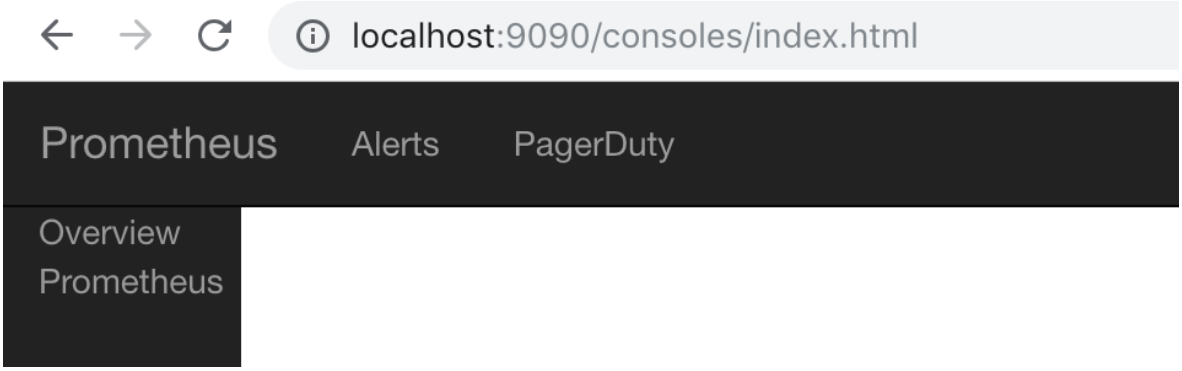
当然，这个时候点击该菜单，我们会看到一个空白页。因为目前index.html文件中还未填充任何内容：

定义页面菜单

首先，我们先直接使用console_libraries中定义的 `head` 组件，并加入到index.html文件中：

```
{{template "head" .}}
```

此时，如果我们刷新浏览器可以看到以下内容：



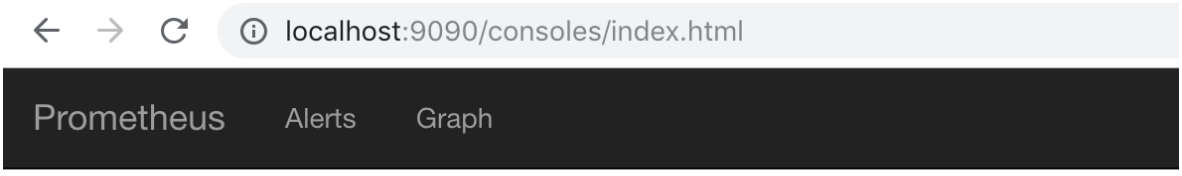
`head` 组件的定义，读者可以通过关键字 `define "head"` 在console_libraries目录中查找。默认其应该是定义在 `prom.lib` 文件中：

```
{{ define "head" }}
<html>
<head>
{{ template "prom_console_head" }}
</head>
<body>
{{ template "navbar" . }}
{{ end }}
```

如果需要定制化菜单的内容，那一样的读者只需要找到 `navbar` 组件的定义即可。当然用户也可以创建自己的组件。 例如，如果我们希望Console Template页面的菜单与Prometheus UI一致，只需要修改navbar组件的定义即可，找到 `menu.lib` 并修改navbar组件:

```
{{ define "navbar" }}
<nav class="navbar navbar-inverse navbar-static-top">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{ pathPrefix }}/">Prometheus</a>
    </div>

    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav">
        <li><a href="{{ pathPrefix }}/alerts">Alerts</a></li>
        <li><a href="{{ pathPrefix }}/graph">Graph</a></li>
      </ul>
    </div>
  </div>
</nav>
{{ end }}
```

如果不需要侧边菜单栏，直接在head组件中移除 `{{ template "menu" . }}` 部分即可，修改后刷新页面，如下所示：



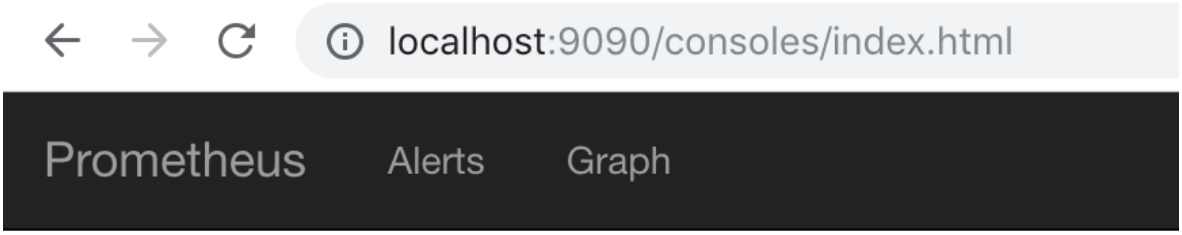
无论是 `.lib` 文件还是 `.html` 文件均使用了Go Template的语言，感兴趣的读者可以自行在Go语言官网了解更多内容 `https://golang.org/pkg/text/template/`

定义图表

在Console Template中我们可以在页面中使用内置的 `PromConsole.Graph()` 函数，该函数通过 `head` 加载相应的js源码，在该函数中，通过指定特定的DOM节点以及相应的PromQL表达式，即可在特定区域图形化显示相应的图表内容，如下所示：

```
<h1>Prometheus HTTP Request Rate</h1>

<h3>Queries</h3>
<div id="queryGraph"></div>
<script>
new PromConsole.Graph({
  node: document.querySelector("#queryGraph"),
  expr: "sum(rate(prometheus_http_request_duration_seconds_count{job='prometheus'}[5m]))",
  name: "Queries",
  yAxisFormatter: PromConsole.NumberFormatter.humanizeNoSmallPrefix,
  yHoverFormatter: PromConsole.NumberFormatter.humanizeNoSmallPrefix,
  yUnits: "/s",
  yTitle: "Queries"
})
</script>
```

这里创建了一个id为queryGraph的div节点，通过在页面中使用PromConsole.Graph函数，我们可以绘制出表达式 `sum(rate(prometheus_http_request_duration_seconds_count{job='prometheus'}[5m]))` 的可视化图表如下所示：



除了最基本的node以及expr参数以外，该函数还支持的完整参数如下：

参数名称	作用
expr	Required. Expression to graph. Can be a list.
node	Required. DOM node to render into.
duration	Optional. Duration of the graph. Defaults to 1 hour.
endTime	Optional. Unixtime the graph ends at. Defaults to now.
width	Optional. Width of the graph, excluding titles. Defaults to auto-detection.

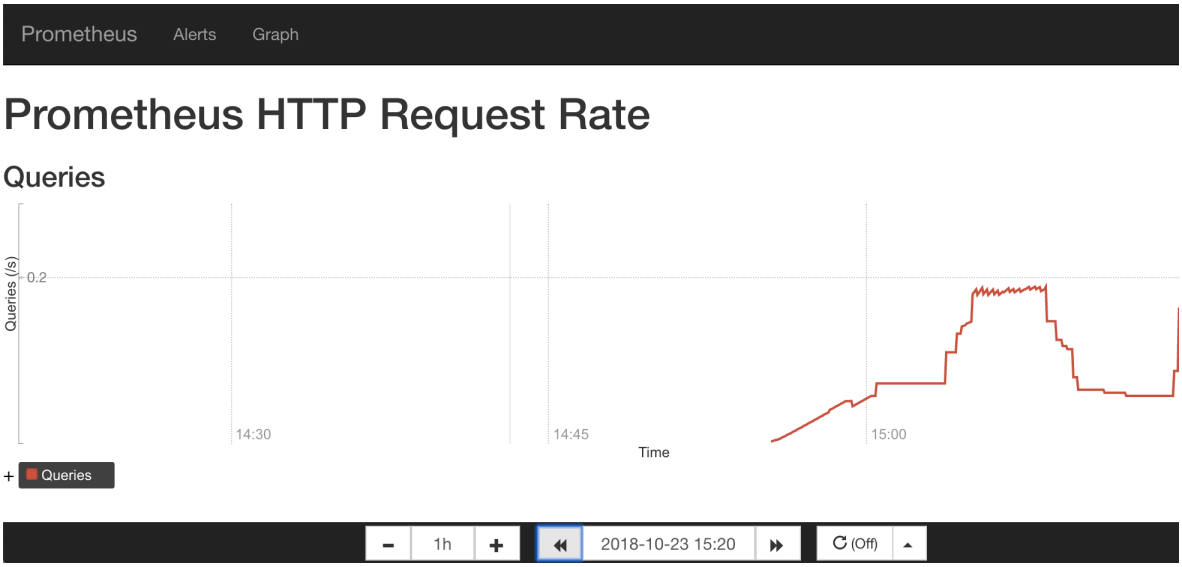
参数名称	作用
height	Optional. Height of the graph, excluding titles and legends. Defaults to 200 pixels.
min	Optional. Minimum x-axis value. Defaults to lowest data value.
max	Optional. Maximum y-axis value. Defaults to highest data value.
renderer	Optional. Type of graph. Options are line and area (stacked graph). Defaults to line.
name	Optional. Title of plots in legend and hover detail. If passed a string, [[label]] will be substituted with the label value. If passed a function, it will be passed a map of labels and should return the name as a string. Can be a list.
xTitle	Optional. Title of the x-axis. Defaults to Time.
yUnits	Optional. Units of the y-axis. Defaults to empty.
yTitle	Optional. Title of the y-axis. Defaults to empty.
yAxisFormatter	Optional. Number formatter for the y-axis. Defaults to PromConsole.NumberFormatter.humanize.
yHoverFormatter	Optional. Number formatter for the hover detail. Defaults to PromConsole.NumberFormatter.humanizeExact.
colorScheme	Optional. Color scheme to be used by the plots. Can be either a list of hex color codes or one of the color scheme names supported by Rickshaw. Defaults to ‘colorwheel’.

需要注意的是，如果参数 `expr` 和 `name` 均是list类型，其必须是一一对应的。

除了直接使用 `PromConsole.Graph` 函数显示可视化图表以外，在Console Template中还可以使用模板组件 `prom_query_drilldown` 定义一个连接直接跳转到Graph页面，并显示相应表达式的查询结果， 如下所示：

```
<h3>Links</h3>
{{ template "prom_query_drilldown" (args "prometheus_http_response_size_bytes_bucket") }}
```

除了以上部分以外，我们也可以和原生Prometheus UI一样定义一个时间轴控制器，方便用户按需查询数据：



加入这个时间轴控制器的方式也很简单，直接引用以下模板即可：

```
{{ template "prom_graph_timecontrol" . }}
```

Grafana的基本概念

Console Teamplet虽然能满足一定的可视化需求，但是也仅仅是对Prometheus的基本能力的补充。同时使用也会有许多问题，首先用户需要学习和了解Go Template模板语言，其它其支持的可视化图表类型也非常有限，最后其管理也有一定的成本。在第1章的“初识Prometheus”中我们已经尝试通过Grafana快速搭建过一个主机监控的Dashboard，在本章中将会带来读者学习如何使用Grafana创建更加精美的可视化报表。

Grafana基本概念

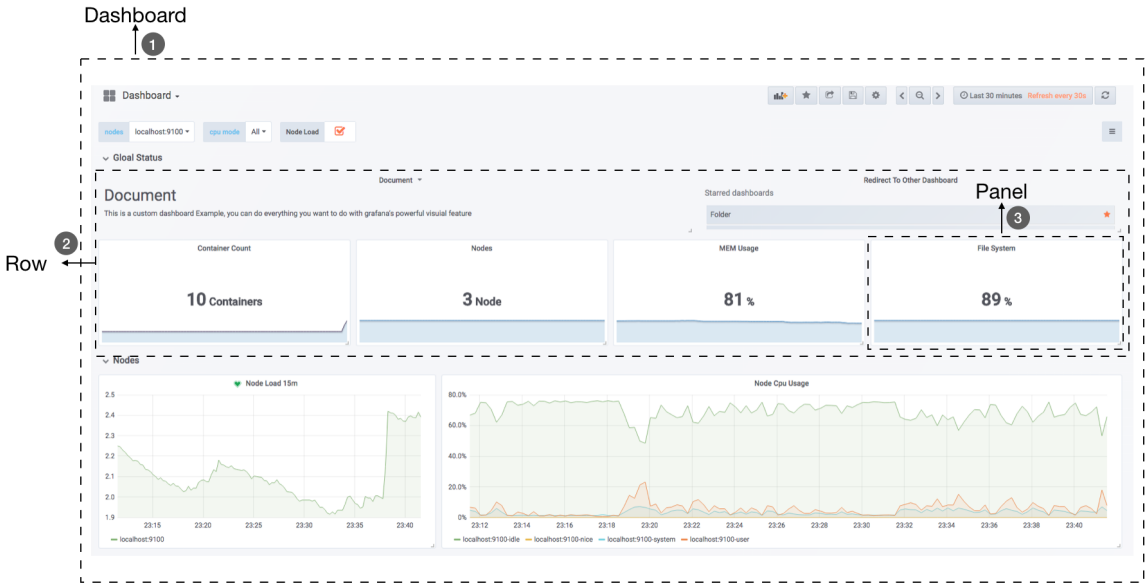
首先Grafana是一个通用的可视化工具。‘通用’意味着Grafana不仅仅适用于展示Prometheus下的监控数据，也同样适用于一些其他的数据可视化需求。在开始使用Grafana之前，我们首先需要明确一些Grafana下的基本概念，以帮助用户能够快速理解Grafana。

数据源（Data Source）

对于Grafana而言，Prometheus这类为其提供数据的对象均称为数据源（Data Source）。目前，Grafana官方提供了对：Graphite, InfluxDB, OpenTSDB, Prometheus, Elasticsearch, CloudWatch的支持。对于Grafana管理员而言，只需要将这些对象以数据源的形式添加到Grafana中，Grafana便可以轻松的实现对这些数据的可视化工作。

仪表盘（Dashboard）

通过数据源定义好可视化的数据来源之后，对于用户而言最重要的事情就是实现数据的可视化。在Grafana中，我们通过Dashboard来组织和管理我们的数据可视化图表：



如上所示，在一个Dashboard中一个最基本的可视化单元为一个**Panel（面板）**，Panel通过如趋势图，热力图的形式展示可视化数据。 并且在Dashboard中每一个Panel是一个完全独立的部分，通过Panel的**Query Editor（查询编辑器）**我们可以为每一个Panel自己查询的数据源以及数据查询方式，例如，如果以Prometheus作为数据源，那在Query Editor中，我们实际上使用的是PromQL，而Panel则会负责从特定的Prometheus中查询出相应的数据，并且将其可视化。由于每个Panel是完全独立的，因此在一个Dashboard中，往往可能会包含来自多个Data Source的数据。

Grafana通过插件的形式提供了多种Panel的实现，常用的如：Graph Panel，Heatmap Panel，SingleStat Panel以及Table Panel等。用户还可通过插件安装更多类型的Panel面板。

除了Panel以外，在Dashboard页面中，我们还可以定义一个**Row（行）**，来组织和管理一组相关的Panel。

除了Panel, Row这些对象以外，Grafana还允许用户为Dashboard定义**Templating variables（模板参数）**，从而实现可以与用户动态交互的Dashboard页面。同时Grafana通过JSON数据结构管理了整个Dasboard的定义，因此这些Dashboard也是非常方便进行共享的。Grafana还专门为Dashboard提供了一个共享服务：https://grafana.com/dashboards，通过该服务用户可以轻松实现Dashboard的共享，同时我们也能快速的从中找到我们希望的Dashboard实现，并导入到自己的Grafana中。

组织和用户

作为一个通用可视化工具，Grafana除了提供灵活的可视化定制能力以外，还提供了面向企业的组织级管理能力。在Grafana中Dashboard是属于一个**Organization（组织）**，通过Organization，可以在更大规模上使用Grafana，例如对于一个企业而言，我们可以创建多个Organization，其中**User（用户）**可以属于一个或多个不同的Organization。 并且在不同的Organization下，可以为User赋予不同的权限。 从而可以有效的根据企业的组织架构定义整个管理模型。

Grafana与数据可视化

在第1章的“初始Prometheus”部分，我们已经带领读者大致了解了Grafana的基本使用方式。对于Grafana而言，Prometheus就是一个用于存储监控样本数据的数据源（Data Source）通过使用PromQL查询特定Prometheus实例中的数据并且在Panel中实现可视化。

接下来，我们将带领读者了解如何通过Panel创建精美的可视化图表。

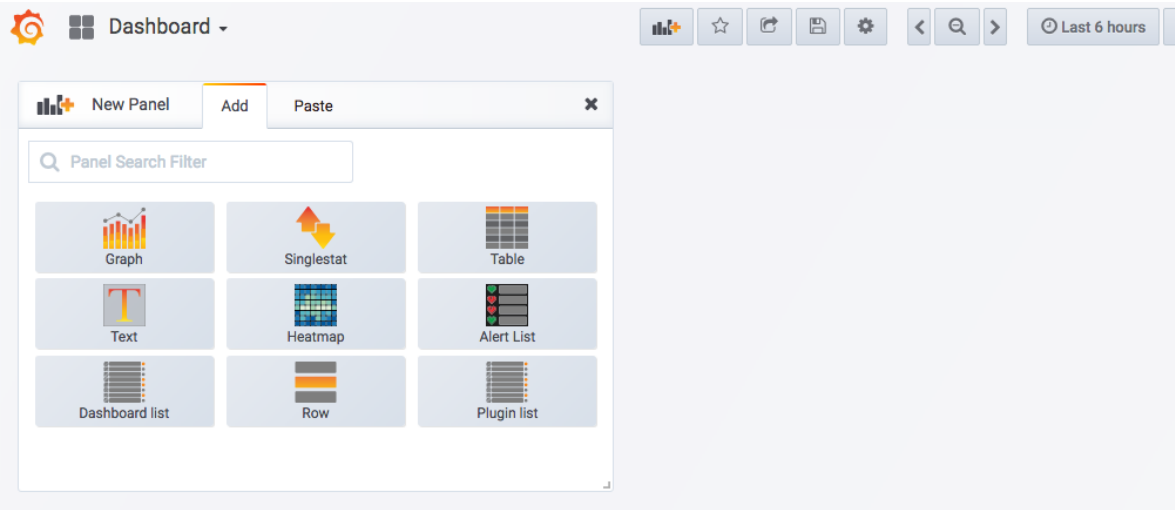
认识面板（Panel）

Panel是Grafana中最基本的可视化单元。每一种类型的面板都提供了相应的查询编辑器(Query Editor)，让用户可以从不同的数据源（如Prometheus）中查询出相应的监控数据，并且以可视化的方式展现。

Grafana中所有的面板均以插件的形式进行使用，当前内置了5种类型的面板，分别是：Graph，Singlestat，Heatmap，Dashlist，Table以及Text。

其中像Graph这样的面板允许用户可视化任意多个监控指标以及多条时间序列。而Siglestat则必须要求查询结果为单个样本。Dashlist和Text相对比较特殊，它们与特定的数据源无关。

通过Grafana UI用户可以在一个Dashboard下添加Panel，点击Dashboard右上角的“Add Panel”按钮，如下所示，将会显示当前系统中所有可使用的Panel类型：

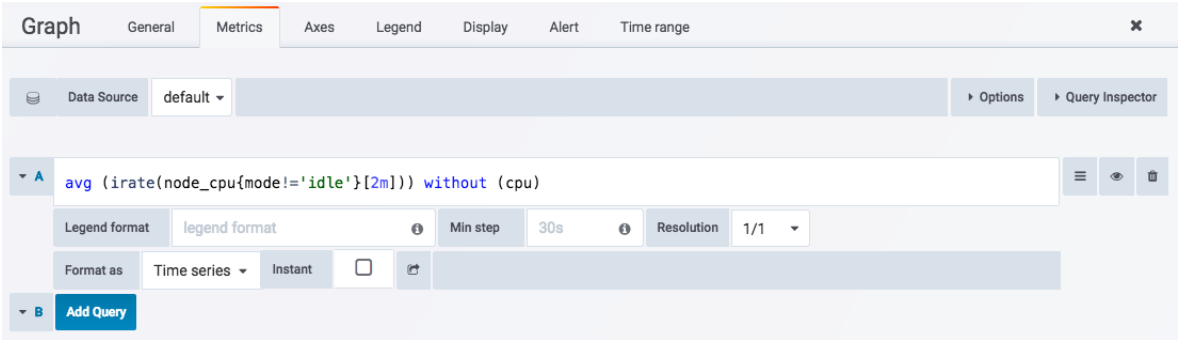


选择想要创建的面板类型即可。这里以Graph面板为例，创建Panel之后，并切换到编辑模式，就可以进入Panel的配置页面。对于一个Panel而言，一般来说会包含2个主要的配置选项：General（通用设置）、Metrics（度量指标）。其余的配置则根据Panel类型的不同而不同。

在通用设置中，除了一些Panel的基本信息以外，最主要的能力就是定义动态Panel的能力，这部分内容会在本章的“模板化Dashboard”小结中详细介绍。

对于使用Prometheus作为数据源的用户，最主要的需要了解的就是Metrics设置的使用。在Metric选项中可以定义了Grafana从哪些数据源中查询样本数据。**Data Source**中指定当前查询的数据源，Grafana会加载当前组织中添加的所有数据源。其中还会包含两个特殊的数据源：**Mixed**和**Grafana**。 Mixed用于需要从多个数据源中查询和渲染数据的场景，Grafana则用于需要查询Grafana自身状态时使用。

当选中数据源时，Panel会根据当前数据源类型加载不同的Query Editor界面。这里我们主要介绍Prometheus Query Editor，如下所示，当选中的数据源类型为Prometheus时，会显示如下界面：



Grafana提供了对PromQL的完整支持，在Query Editor中，可以添加任意个Query，并且使用PromQL表达式从Prometheus中查询相应的样本数据。

```
avg (irate(node_cpu{mode!='idle'}[2m])) without (cpu)
```

每个PromQL表达式都可能返回多条时间序列。**Legend format**用于控制如何格式化每条时间序列的图例信息。Grafana支持通过模板的方式，根据时间序列的标签动态生成图例名称，例如：使用{{instance}}表示使用当前时间序列中的instance标签的值作为图例名称：

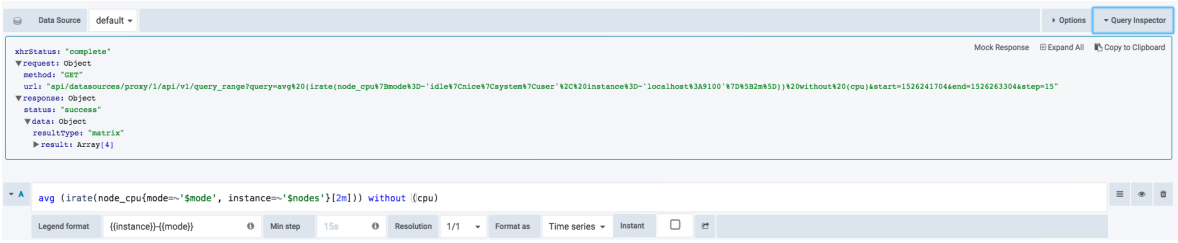
```
{{instance}}-{{mode}}
```

当查询到的样本数据量非常大时可以导致Grafana渲染图标时出现一些性能问题，通过**Min Step**可以控制Prometheus查询数据时的最小步长（Step），从而减少从Prometheus返回的数据量。

Resolution选项，则可以控制Grafana自身渲染的数据量。例如，如果**Resolution**的值为**1/10**，Grafana会将Prometeus返回的10个样本数据合并成一个点。因此**Resolution**越小可视化的精确性越高，反之，可视化的精度越低。

Format as选项定义如何格式化Prometheus返回的样本数据。这里提供了3个选项：Table,Time Series和Heatmap，分别用于Tabel面板，Graph面板和Heatmap面板的数据可视化。

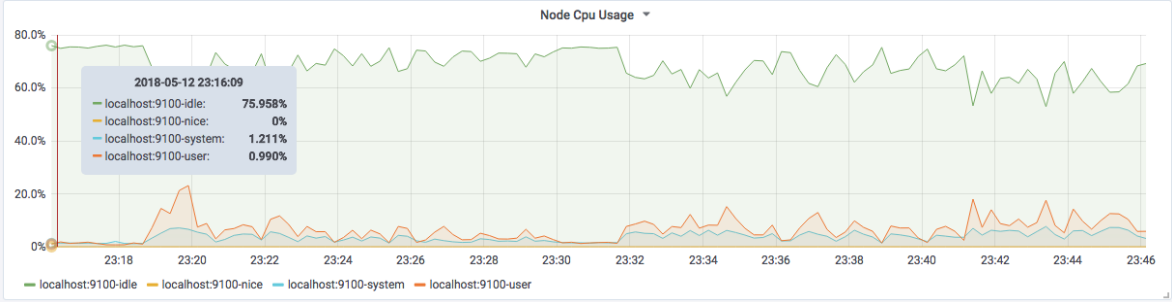
除此以外，Query Editor还提供了调试相关的功能，点击**Query Inspector**可以展开相关的调试面板：



在面板中，可以查看当前Prometheus返回的样本数据，用户也可以提供Mock数据渲染图像。

变化趋势：Graph面板

Graph面板是最常用的一种可视化面板，其通过折线图或者柱状图的形式显示监控样本随时间而变化的趋势。Graph面板天生适用于Prometheus中Gauge和Counter类型监控指标的监控数据可视化。例如，当需要查看主机CPU、内存使用率的随时间变化的情况时，可以使用Graph面板。同时，Graph还可以非常方便的支持多个数据之间的对比。

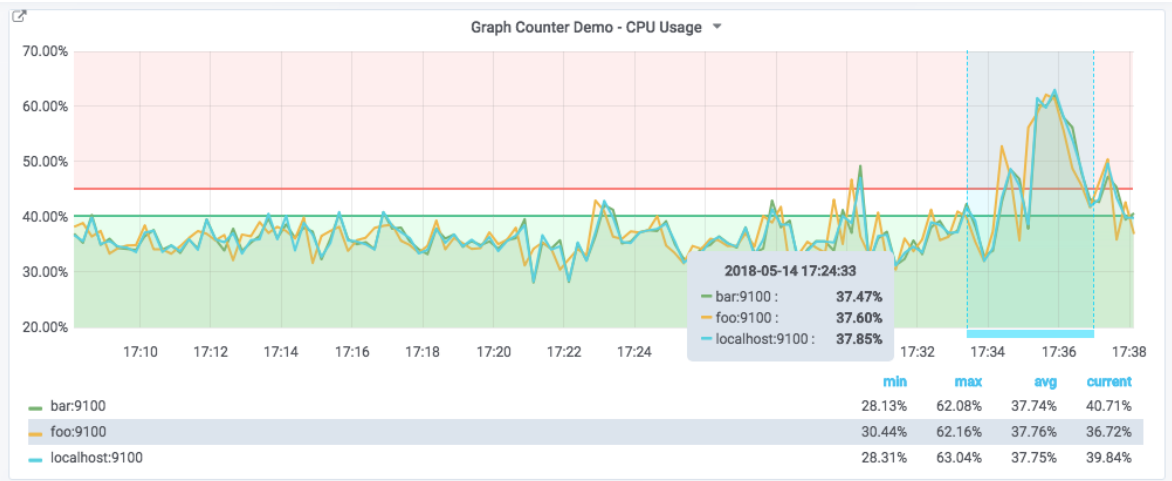


Graph面板与Prometheus

Graph面板通过折线图或者柱状图的形式，能够展示监控样本数据在一段时间内的变化趋势，因此其天生适合Prometheus中的Counter和Gauge类型的监控指标的可视化，对于Histogram类型的指标也可以支持，不过可视化效果不如Heatmap Panel来的直观。

使用Graph面板可视化Counter/Gauge

以主机为例，CPU使用率的变化趋势天然适用于使用Grapn面板来进行展示：



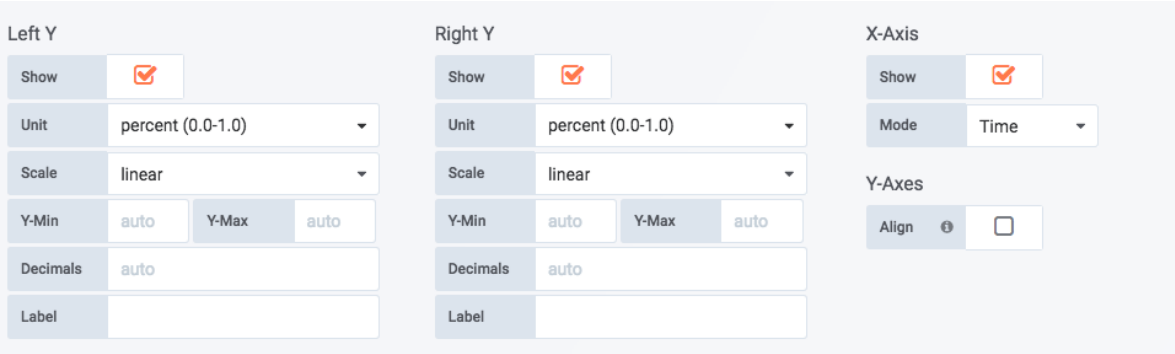
在**Metrics选项**中，我们使用以下PromQL定义如何从Prometheus中读取数据：

```
1 - (avg(irate(node_cpu{mode='idle'}[5m])) without (cpu))
```

如下所示：

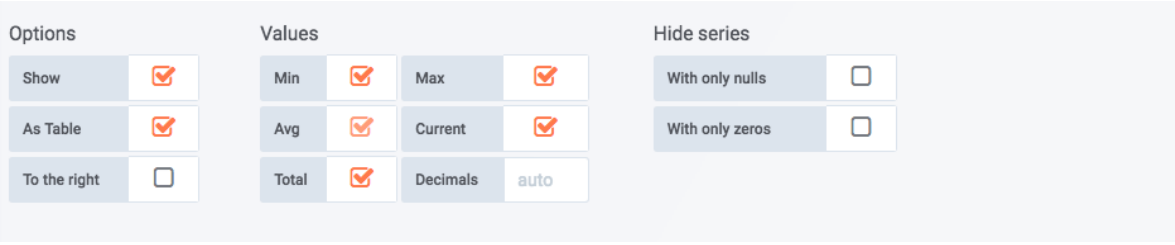
根据当前Prometheus的数据采集情况，该PromQL会返回多条时间序列（在示例中会返回3条）。Graph面板会从时间序列中获取样本数据，并绘制到图表中。 为了让折线图有更好的可读性，我们可以通过定义**Legend format**为 `{{ instance }}` 控制每条线的图例名称：

由于当前使用的PromQL的数据范围为0~1表示CPU的使用率，为了能够更有效的表达出度量单位的概念，我们需要对Graph图表的坐标轴显示进行优化。如下所示，在**Axes选项**中可以控制图标的X轴和Y轴相关的行为：



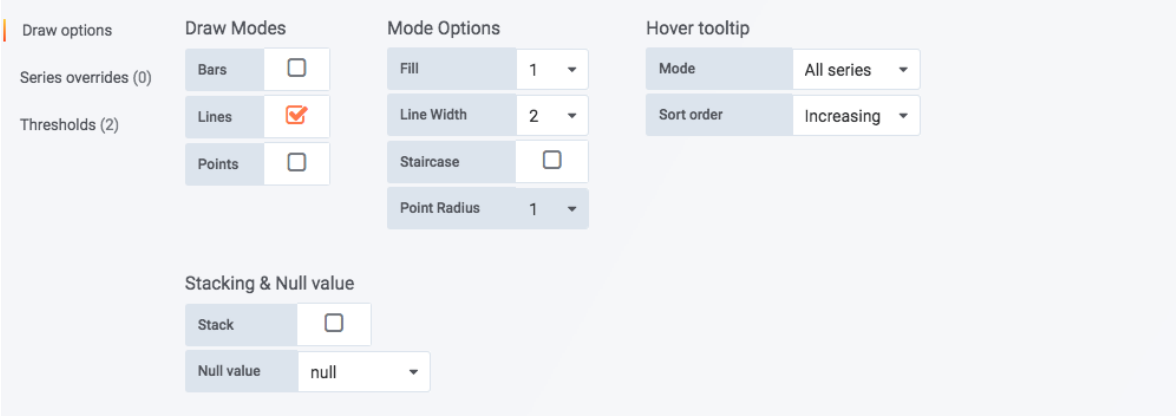
默认情况下，Y轴会直接显示当前样本的值，通过**Left Y**的**Unit**可以让Graph面板自动格式化样本值。当前表达式返回的当前主机CPU使用率的小数表示，因此，这里选择单位为**percent(0.0.-1.0)**。除了百分比以外，Graph面板支持如日期、货币、重量、面积等各种类型单位的自动换算，用户根据自己当前样本的值含义选择即可。

除了在Metrics设置图例的显示名称以外，在Graph面板的**Legend选项**可以进一步控制图例的显示方式，如下所示：



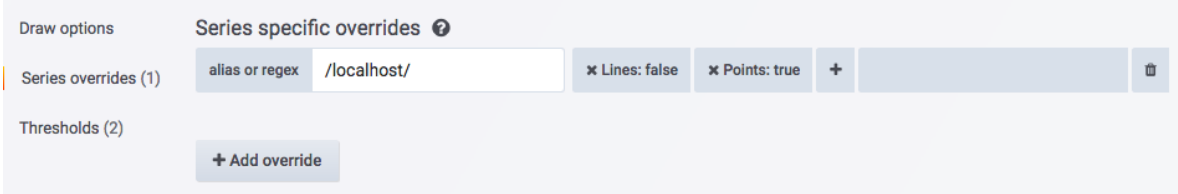
Options中可以设置图例的显示方式以及展示位置，**Values**中可以设置是否显示当前时间序列的最小值，平均值等。**Decimals**用于配置这些值显示时保留的小数位，如下所示：

除了以上设置以外，我们可能还需要对图表进行一些更高级的定制化，以便能够更直观的从可视化图表中获取信息。在Graph面板中**Display选项**可以帮助我们实现更多的可视化定制的能力，其中包含三个部分：Draw options、Series overrides和Thresholds。

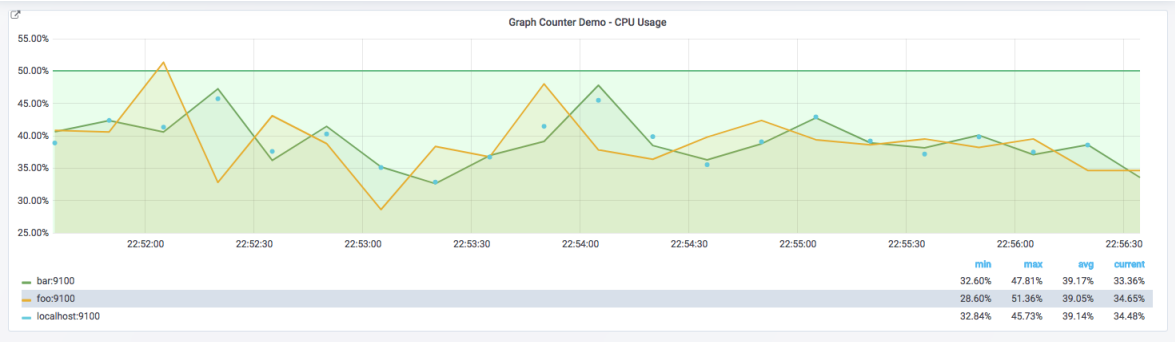


Draw Options用于设置当前图标的展示形式、样式以及交互提示行为。其中，Draw Modes用于控制图形展示形式：Bar（柱状）、Lines（线条）、Points（点），用户可以根据自己的需求同时启用多种模式。Mode Options则设置各个展示模式下的相关样式。Hover tooltip用于控制当鼠标移动到图形时，显示提示框中的内容。

如果希望当前图表中的时间序列以不同的形式展示，则可以通过**Series overrides**控制，顾名思义，可以为指定的时间序列指定自定义的Draw Options配置，从而让其以不同的样式展示。例如：



这里定义了一条自定义规则，其匹配图例名称满足**/localhost/**的时间序列，并定义其以点的形式显示在图表中，修改后的图标显示效果如下：



Display选项中的最后一个是**Thresholds**，Threshold主要用于一些自定义一些样本的阈值，例如，定义一个Threshold规则，如果CPU超过50%的区域显示为warning状态，可以添加如下配置：

Graph面板则会在图表中显示一条阈值，并且将所有高于该阈值的区域显示为warining状态，通过可视化的方式直观的在图表中显示一些可能出现异常的区域。

需要注意的是，如果用户为该图表自定义了Alert（告警）配置，Thresholds将会被警用，并且根据Alert中定义的Threshold在图形中显示阈值内容。关于Alert的使用会在后续部分，详细介绍。

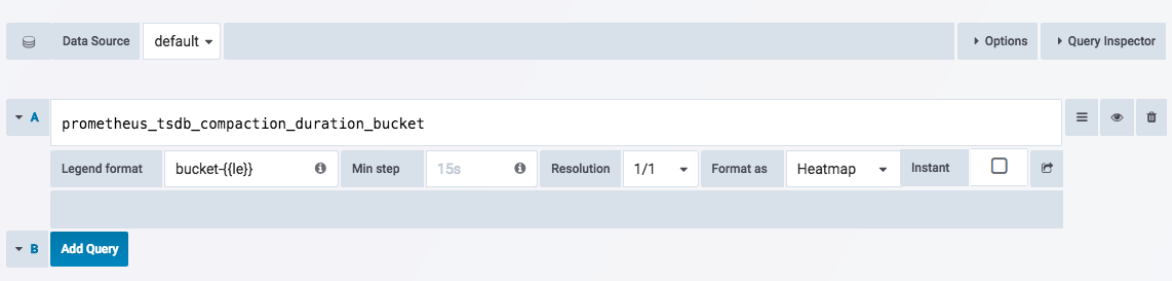
使用Graph面板可视化Histogram

以Prometheus自身的监控指标prometheus_tsdb_compaction_duration为例，该监控指标记录了Prometheus进行数据压缩任务的运行耗时的分布统计情况。如下所示，是Prometheus返回的样本数据：

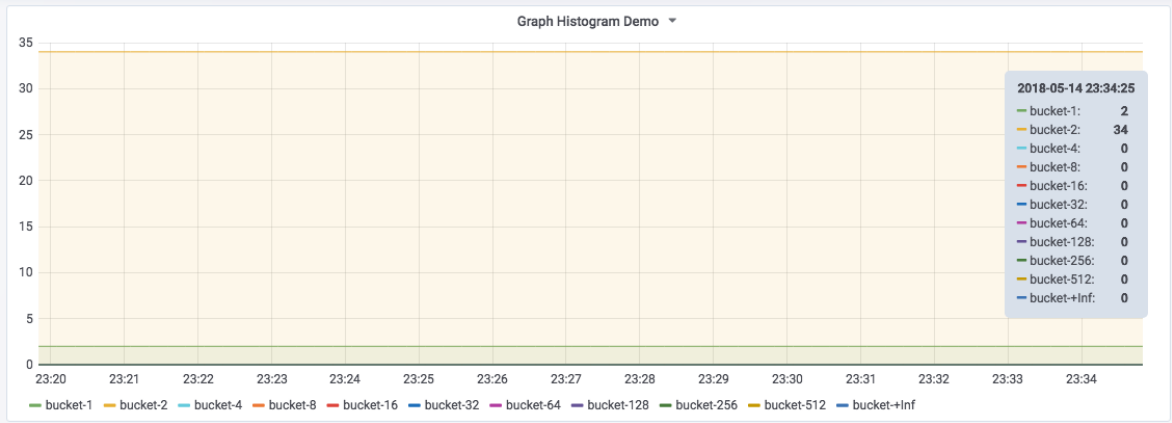
```
# HELP prometheus_tsdb_compaction_duration Duration of compaction runs.
# TYPE prometheus_tsdb_compaction_duration histogram
prometheus_tsdb_compaction_duration_bucket{le="1"} 2
prometheus_tsdb_compaction_duration_bucket{le="2"} 36
prometheus_tsdb_compaction_duration_bucket{le="4"} 36
prometheus_tsdb_compaction_duration_bucket{le="8"} 36
prometheus_tsdb_compaction_duration_bucket{le="16"} 36
prometheus_tsdb_compaction_duration_bucket{le="32"} 36
prometheus_tsdb_compaction_duration_bucket{le="64"} 36
prometheus_tsdb_compaction_duration_bucket{le="128"} 36
prometheus_tsdb_compaction_duration_bucket{le="256"} 36
prometheus_tsdb_compaction_duration_bucket{le="512"} 36
prometheus_tsdb_compaction_duration_bucket{le="+Inf"} 36
prometheus_tsdb_compaction_duration_sum 51.31017077500001
prometheus_tsdb_compaction_duration_count 36
```

在第2章的“Metric类型”小节中，我们已经介绍过Histogram的指标，Histogram用于统计样本数据的分布情况，其中标签le定义了分布桶Bucket的边界，如上所示，表示当前Prometheus共进行了36次数据压缩，总耗时为51.31017077500001ms。其中任务耗时在0~~1ms区间内的为2次，在0~~2ms区间范围内为36次，以此类推。

如下所示，如果需要在Graph中显示Histogram类型的监控指标，需要在Query Editor中定义查询结果的**Format as**为Heatmap。通过该设置Grafana会自动计算Histogram中的Bucket边界范围以及该范围内的值：



Graph面板重新计算了Bucket边界，如下所示，在0~~1ms范围内的任务次数为2，在1~~2ms范围内的运行任务次数为34。通过图形的面积，可以反映出各个Bucket下的大致数据分布情况：



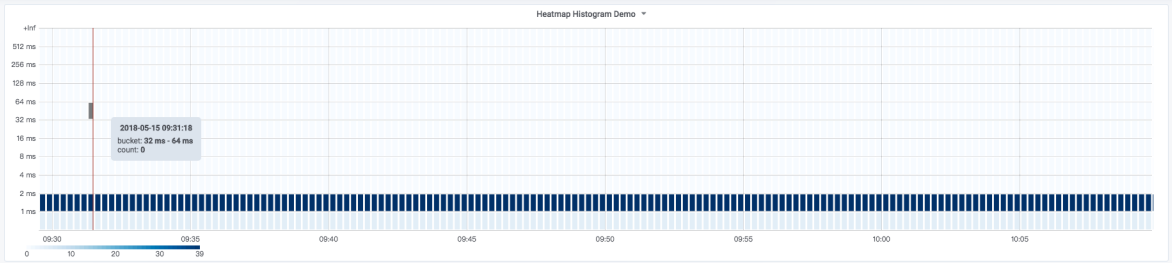
不过通过Graph面板展示Histogram也并不太直观，其并不能直接反映出Bucket的大小以及分布情况，因此在Grafana V5版本以后更推荐使用Heatmap面板的方式展示Histogram样本数据。关于Heatmap面板的使用将会在接下来的部分介绍。

分布统计：Heatmap面板

Heatmap是是Grafana v4.3版本以后新添加的可视化面板，通过热图可以直观的查看样本的分布情况。在Grafana v5.1版本中Heatmap完善了对Prometheus的支持。这部分，将介绍如何使用Heatmap Panel实现对Prometheus监控指标的可视化。

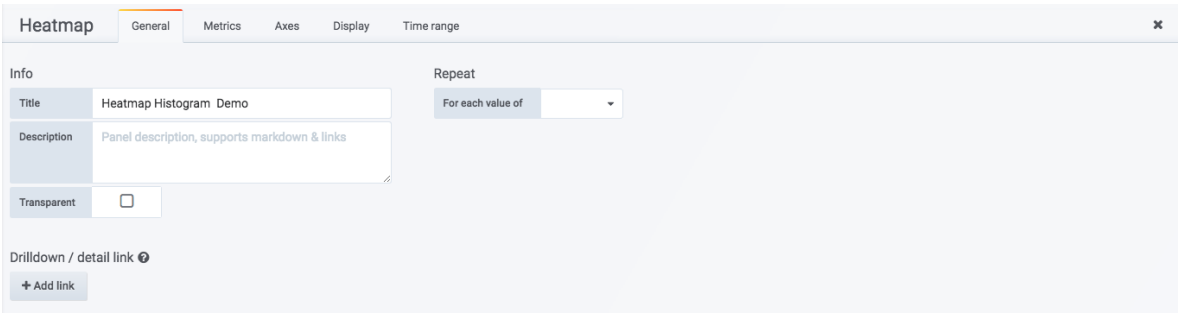
使用Heatmap可视化Histogram样本分布情况

在上一小节中，我们尝试了使用Graph面板来可视化Histogram类型的监控指标prometheus_tsdb_compaction_duration_bucket。虽然能展示各个Bucket区间内的样本分布，但是无论是以线图还是柱状图的形式展示，都不够直观。对于Histogram类型的监控指标来说，更好的选择是采用Heatmap Panel，如下所示，Heatmap Panel可以自动对Histogram类型的监控指标分布情况进行计划，获取到每个区间范围内的样本个数，并且以颜色的深浅来表示当前区间内样本个数的大小。而图形的高度，则反映出当前时间点，样本分布的离散程度。



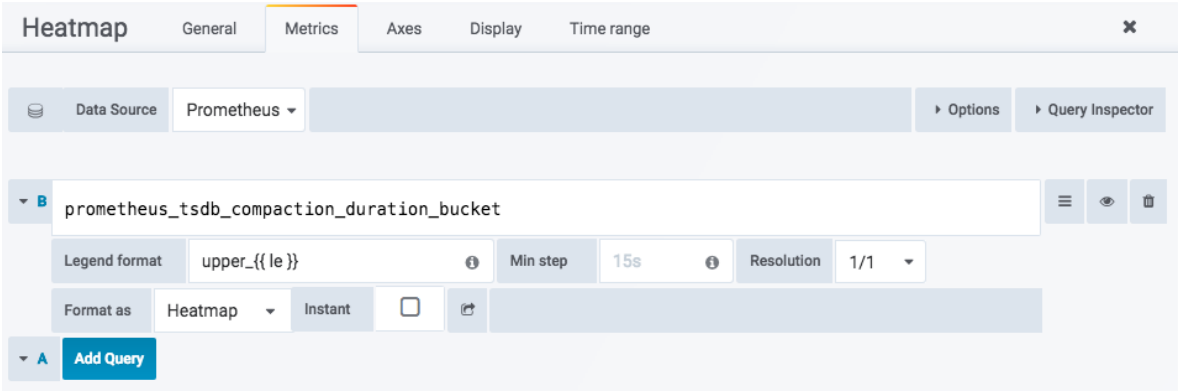
在Grafana中使用Heatmap Panel也非常简单，在Dashboard页面右上角菜单中点击“add panel”按钮，并选择Heatmap Panel即可。

如下所示，Heapmap Panel的编辑页面中，主要包含5类配置选项，分别是：General、Metrics、Axes、Display、Time range。

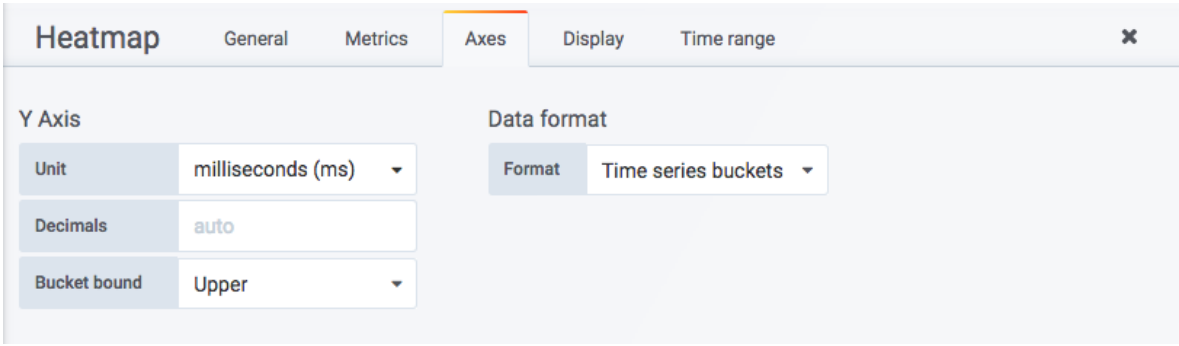


其中大部分的配置选项与Graph面板基本保持一致，这里就不重复介绍了。

当使用Heatmap可视化Histogram类型的监控指标时，需要设置**Format as**选项为**Heatmap**。当使用Heatmap格式化数据后，Grafana会自动根据样本的中的le标签，计算各个Bucket桶内的分布，并且按照Bucket对数据进行重新排序。**Legend format**模板则将会控制Y轴中的显示内容。如下所示：



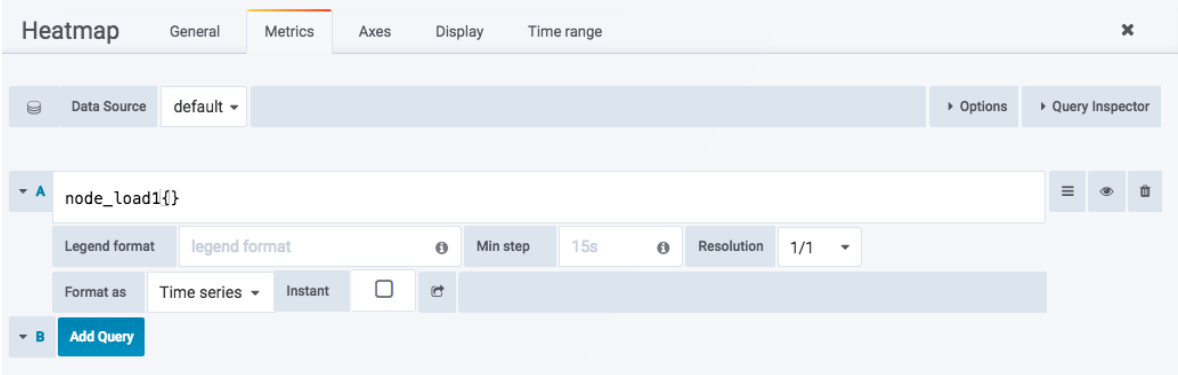
默认情况下，Heatmap Panel会自行对PromQL查询出的数据进行分布情况统计，而在Prometheus中Histogram类型的监控指标其实是已经自带了分布的Bucket信息的，因此为了直接使用这些Bucket信息，我们需要在**Axes选项**中定义数据的Date format需要定义为**Time series buckets**。该选项表示Heatmap Panel不需要自身对数据的分布情况进行计算，直接使用时间序列中返回的Bucket即可。如下所示：



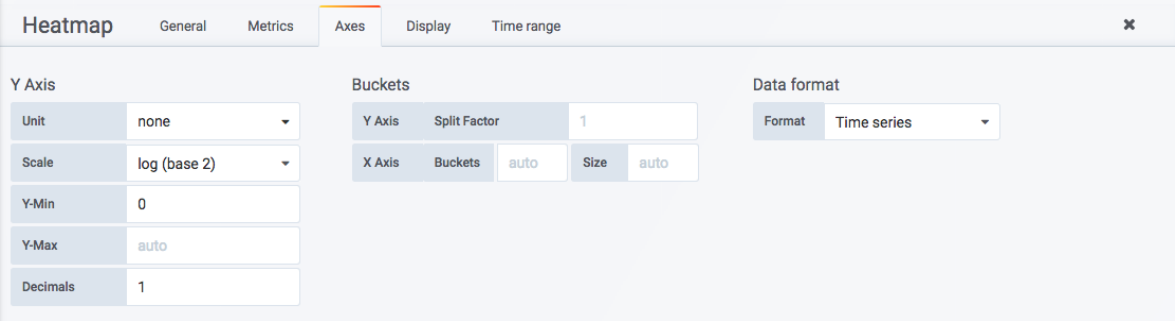
通过以上设置，即可实现对Histogram类型监控指标的可视化。

使用Heatmap可视化其它类型样本分布情况

对于非Histogram类型，由于其监控样本中并不包含Bucket相关信息，因此在**Metrics选项中**需要定义**Format as**为**Time series**，如下所示：

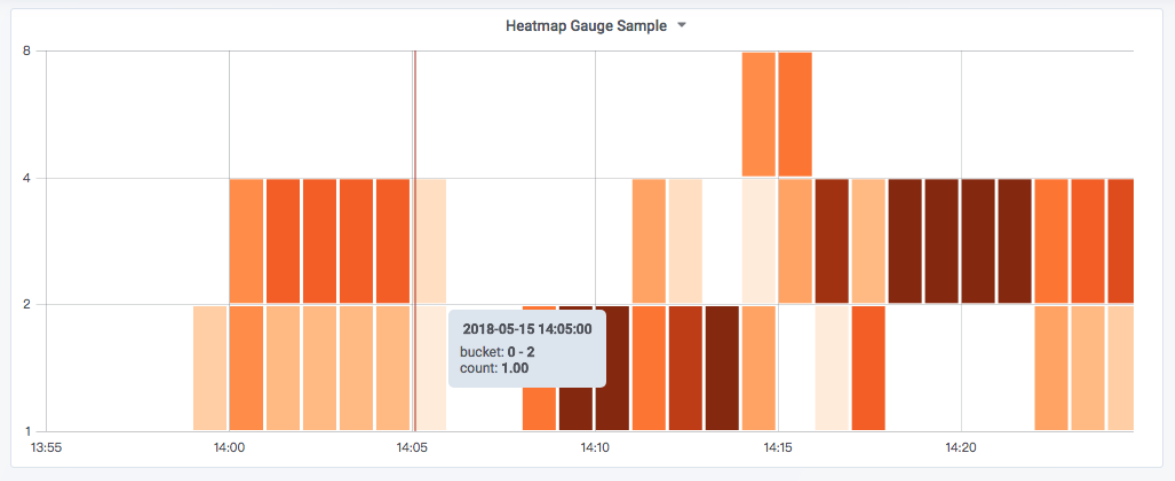


并且通过**Axes选项**中选择**Data format**方式为**Time series**。设置该选项后Heatmap Panel会要求用户提供Bucket分布范围的设置，如下所示：



在Y轴（Y Axis）中需要通过Scale定义Bucket桶的分布范围，默认的Bucket范围支持包括：liner（线性分布）、log(base 10)（10的对数）、log(base 32)（32的对数）、log(base 1024)（1024的对数）等。

例如，上图中设置的Scale为log(base 2)，那么在Bucket范围将2的对数的形式进行分布，即[1,2,4,8,....]，如下所示：



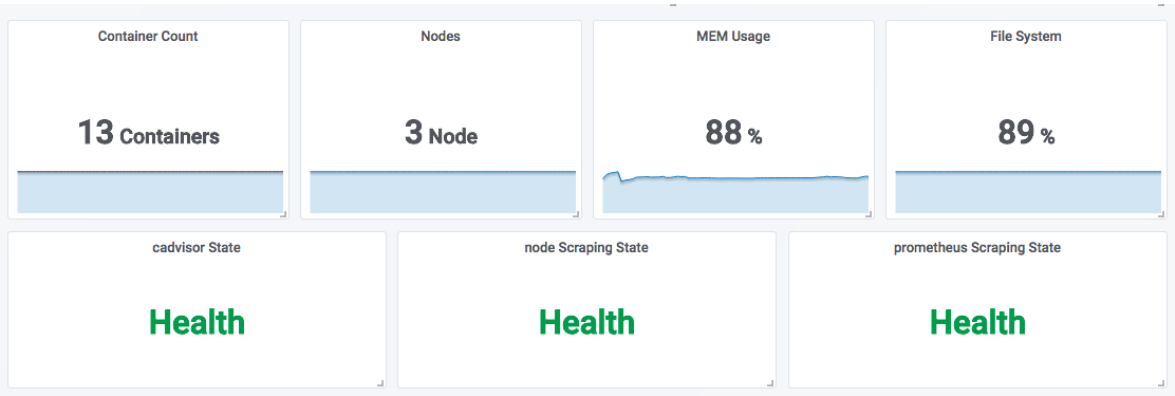
通过以上设置，Heatmap会自动根据用户定义的Bucket范围对Prometheus中查询到的样本数据进行分布统计。

当前状态：SingleStat面板

Singlem Panel侧重于展示系统的当前状态而非变化趋势。如下所示，在以下场景中特别适用于使用SingleStat：

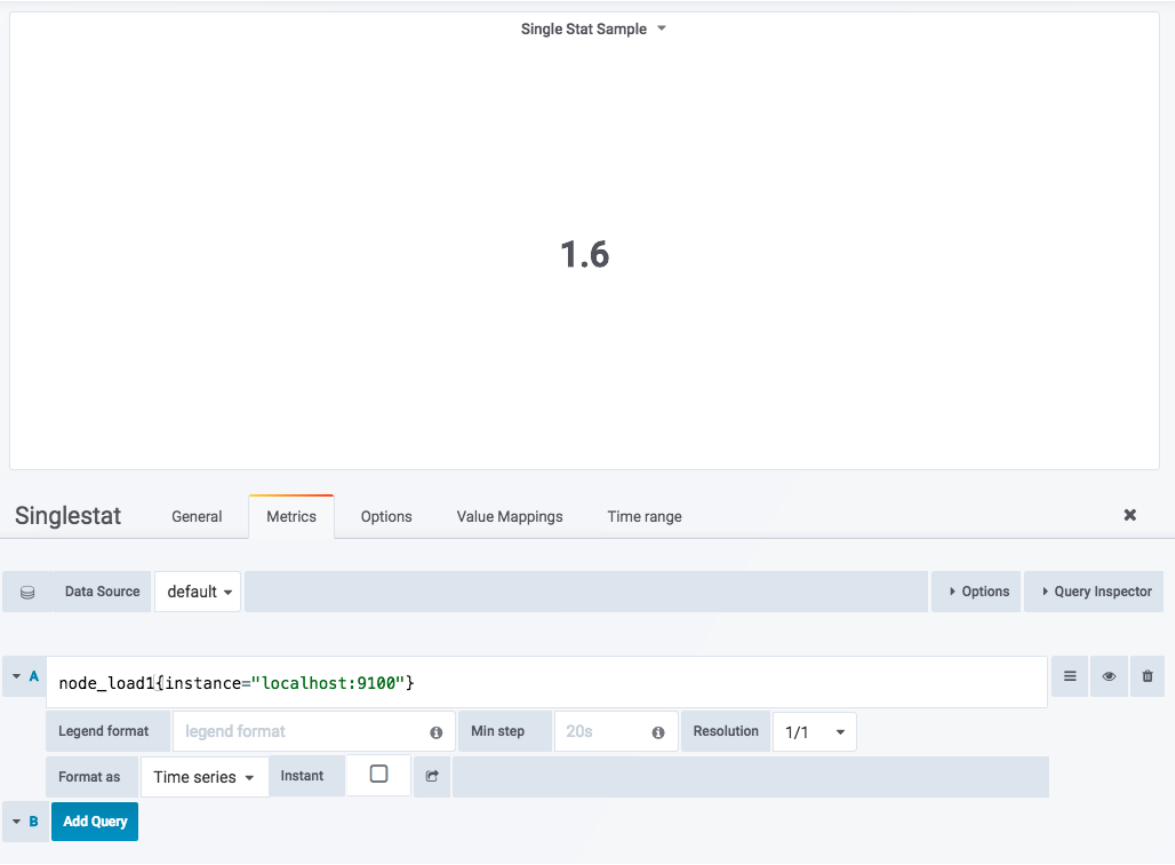
- 当前系统中所有服务的运行状态；
- 当前基础设施资源的使用量；
- 当前系统中某些事件发生的次数或者资源数量等。

如下所示，是使用SingleStat进行数据可视化的显示效果：



使用SingleStat Panel

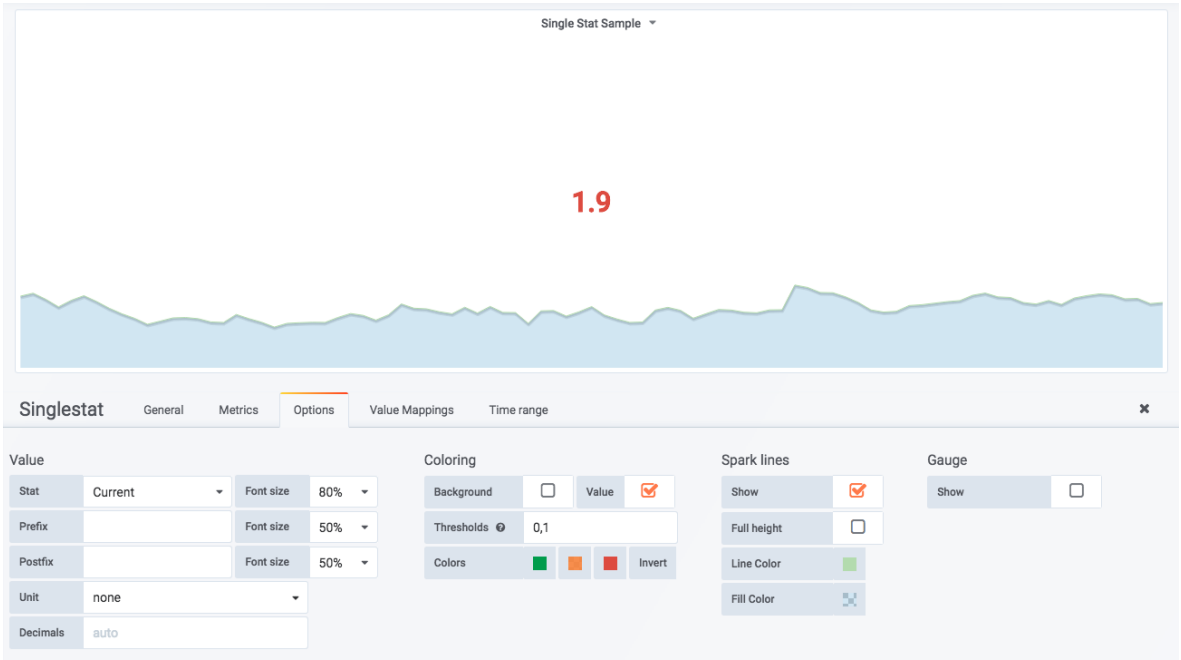
从Dashboardc创建Singlestat Panel，并进入编辑页面， 如下所示：



对于SingleStat Panel而言，其只能处理一条时间序列，否则页面中会提示“Multiple Series Error”错误信息。这里使用如下PromQL查询当前主机负载：

```
node_load1{instance="localhost:9100"}
```

默认情况下，当前面板中会显示当前时间序列中所有样本的平均值，而实际情况下，我们需要显示的是当前主机当前的负载情况，因此需要通过SingleStat Panel的**Options**选项控制当前面板的显示模式：



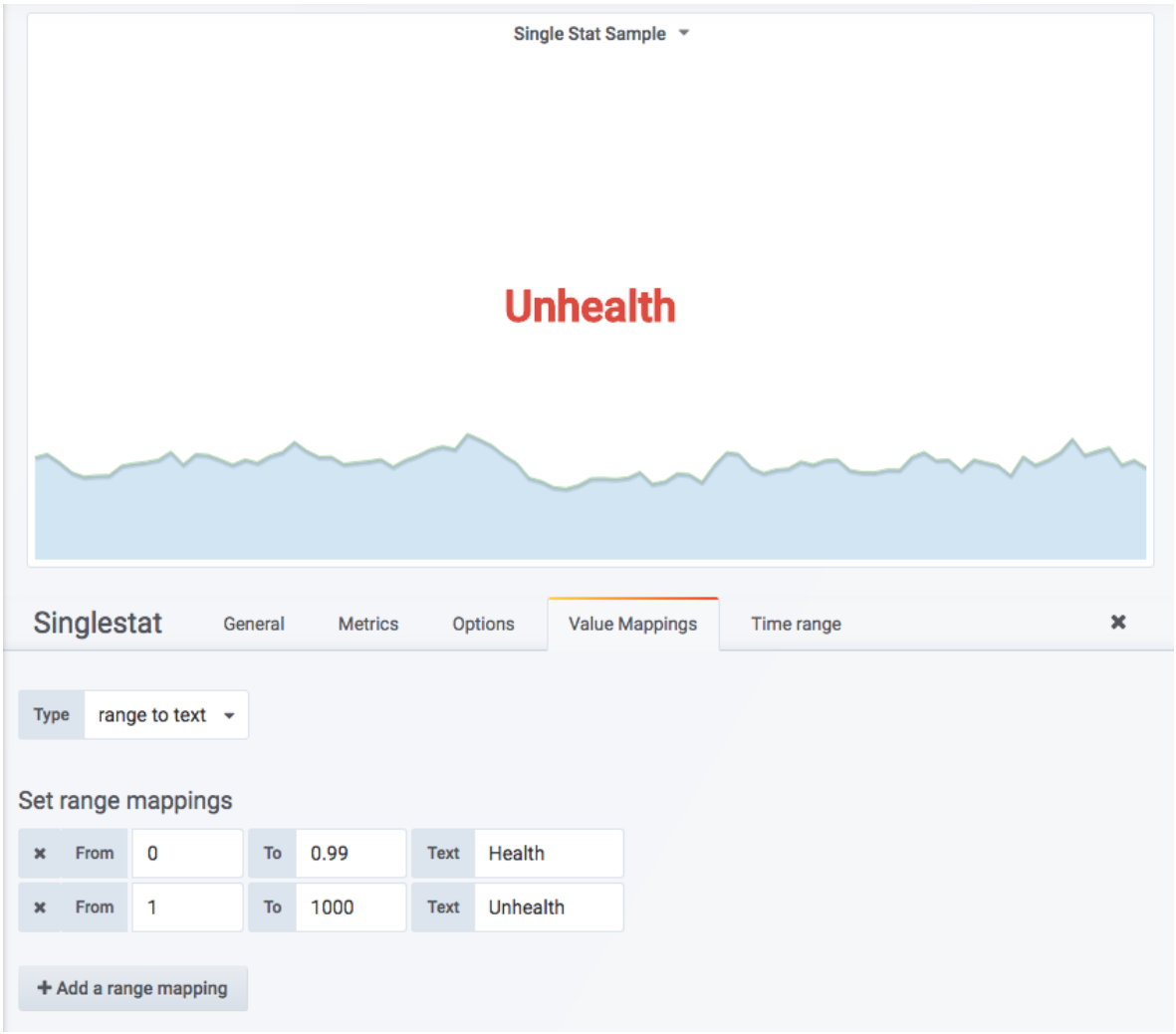
如上所示，通过Value配置项组可以控制当前面板中显示的值，以及字体大小等。对于主机负载而言，我们希望能够显示当前的最新值，因此修改Stat为**Current**即可。

如果希望面板能够根据不同的值显示不同的颜色的话，则可以定义**Thresholds**与**Colors**的映射关系，例如，定义Thresholds的分割区间值为“0,1”，则当Value的值落到不同的范围内时，将显示不同的颜色。

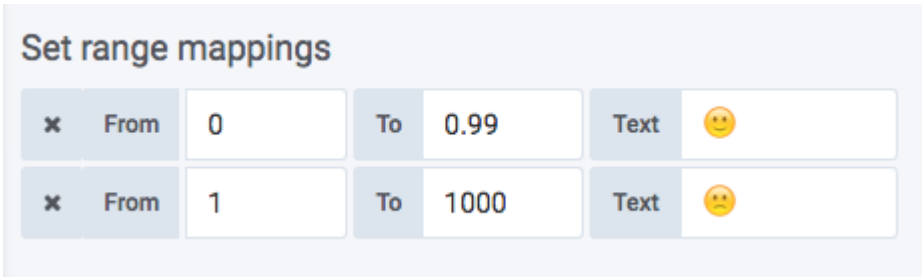
如果希望能够显示当前时间序列的样本值变化情况，则可以启用Spark lines配置。启用之后，Singlestat面板中除了会显示当前的最新样本值以外，也会同时将时间序列中的数据已趋势图的形式进行展示。

除了通过数字大小反应当前状态以外，在某些场景下我们可能更关心的是这些数字表示的意义。例如，在Promthues监控服务的健康状态时，在样本数据中会通过0表示不健康，1表示健康。 但是如果直接将0或1显示在面板中，那么可视化效果将缺乏一定的可读性。

为了提升数字的可读性，可以在Singlestat Panel中可以通过**Value Mappings**定义值的映射关系。Siglesta支持值映射（value to text）和区间映射（range to text）两种方式。 如下所示：



当面板中Value的值在0~0.99范围内则显示为Health，否则显示为Unhealth。这种模式特别适合于展示服务的健康状态。 当然你也可以将Value映射为任意的字符，甚至是直接使用Emoji(http://www.iemoji.com/)表情：

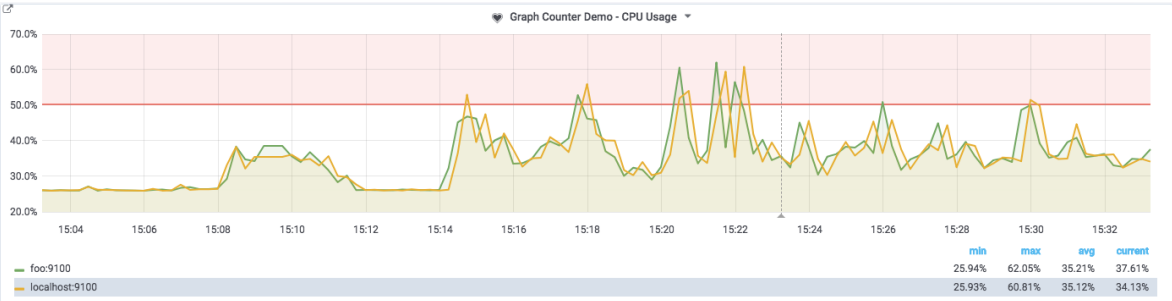


模板化Dashboard

在前面的小节中介绍了Grafana中4中常用的可视化面板的使用，通过在面板中使用PromQL表达式，Grafana能够方便的将Prometheus返回的数据进行可视化展示。例如，在展示主机CPU使用率时，我们使用了如下表达式：

```
1 - (avg(irate(node_cpu{mode='idle'}[5m])) without (cpu))
```

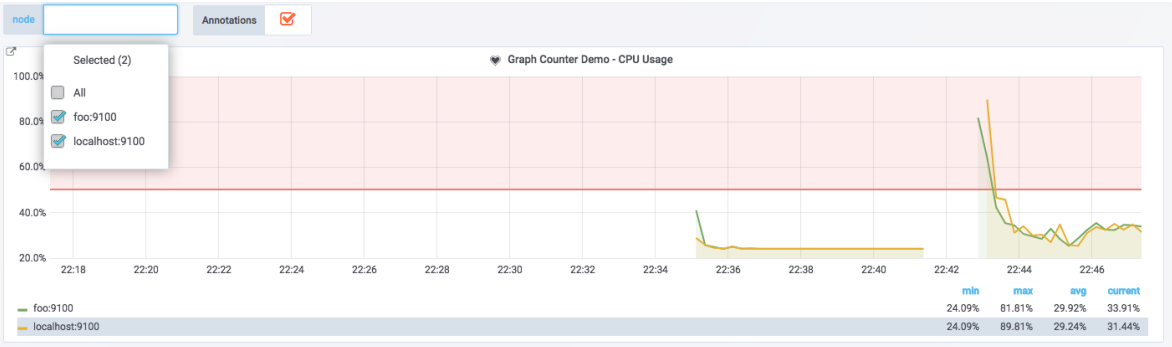
该表达式会返回当前Promthues中存储的所有时间序列，每一台主机都会有一条单独的曲线用于体现其CPU使用率的变化情况：



而当用户只想关注其中某些主机时，基于当前我们已经学习到的知识只有两种方式，要么每次手动修改Panel中的PromQL表达式，要么直接为这些主机创建单独的Panel。但是无论如何，这些硬编码方式都会直接导致Dashboard配置的频繁修改。在这一小节中我们将学习使用Dashboard变量的方式解决以上问题。

变量

在Grafana中用户可以为Dashboard定义一组变量（Variables），变量一般包含一个到多个可选值。如下所示，Grafana通过将变量渲染为一个下拉框选项，从而使用户可以动态的改变变量的值：



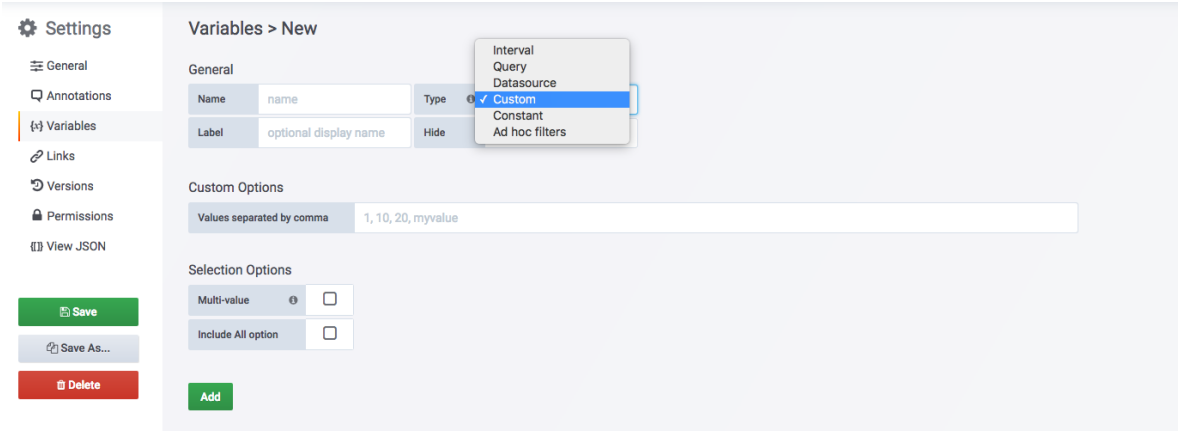
例如，这里定义了一个名为node的变量，用户可以通过在PromQL表达式或者Panel的标题中通过以下形式使用该变量：

```
1 - (avg(irate(node_cpu{mode='idle', instance=~"$node"}[5m])) without (cpu))
```

变量的值可以支持单选或者多选，当对接Prometheus时，Grafana会自动将$node的值格式化为如“**host1|host2|host3**”的形式。配合使用PromQL的标签正则匹配“**=~**”，通过动态改变PromQL从而实现基于标签快速对时间序列进行过滤。

变量定义

通过Dashboard页面的Settings选项，可以进入Dashboard的配置页面并且选择Variables子菜单：



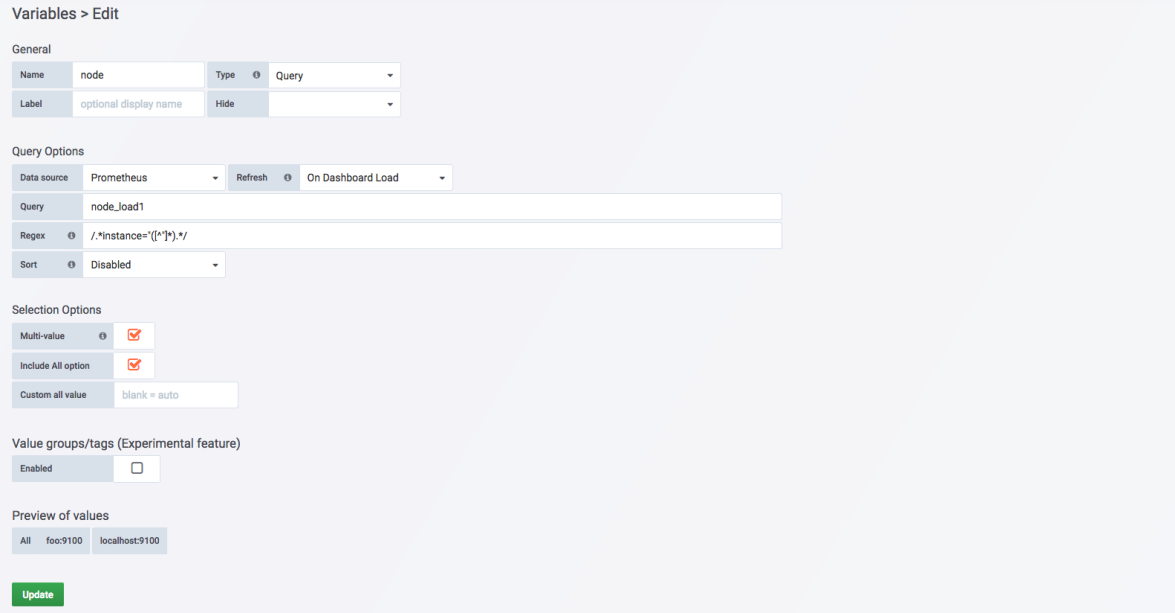
用户需要指定变量的名称，后续用户就可以通过$variable_name的形式引用该变量。Grafana目前支持6种不同的变量类型，而能和Prometheus一起工作的主要包含以下5种类型：

类型	工作方式
Query	允许用户通过Datasource查询表达式的返回值动态生成变量的可选值
Interval	该变量代表时间跨度，通过Interval类型的变量， 可以动态改变PromQL区间向量表达式中的时间范围。 如rate(node_cpu[2m])
Datasource	允许用户动态切换当前Dashboard的数据源， 特别适用于同一个Dashboard展示多个数据源数据的情况
Custom	用户直接通过手动的方式，定义变量的可选值
Constant	常量，在导入Dashboard时，会要求用户设置该常量的值

Label属性用于指定界面中变量的显示名称，Hide属性则用于指定在渲染界面时是否隐藏该变量的下拉框。

使用变量过滤时间序列

当Prometheus同时采集了多个主机节点的监控样本数据时，用户希望能够手动选择并查看其中特定主机的监控数据。这时我们需要使用Query类型的变量。



如上所示，这里我们为Dashboard创建了一个名为node的变量，并且指定其类型为Query。Query类型的变量，允许用户指定数据源以及查询表达式，并通过正则匹配（Regex）的方式对查询结果进行处理，从而动态生成变量的可选值。在这里指定了数据源为Prometheus，通过使用node_load1我们得到了两条时间序列：

```
node_load1{instance="foo:9100", job="node"}
node_load1{instance="localhost:9100", job="node"}
```

通过指定正则匹配表达式为 `/.*instance="([^"]*).*/` 从而匹配出标签instance的值作为node变量的所有可选项，即：

```
foo:9100
localhost:9100
```

Selection Options选项中可以指定该变量的下拉框是否支持多选，以及是否包含全选（All）选项。

保存变量后，用户可以在Panel的General或者Metrics中通过$node的方式使用该变量，如下所示：

这里需要注意的是，如果允许用户多选在PromQL表达式中应该使用标签的正则匹配模式，因为Grafana会自动将多个选项格式化为如“foo:9100|localhost:9100”的形式。

使用Query类型的变量能够根据允许用户能够根据时间序列的特征维度对数据进行过滤。在定义Query类型变量时，除了使用PromQL查询时间序列以过滤标签的方式以外，Grafana还提供了几个有用的函数：

函数	作用
label_values(label)	返回Promthues所有监控指标中， 标签名为label的所有可选值
label_values(metric, label)	返回Promthues所有监控指标metric中， 标签名为label的所有可选值
metrics(metric)	返回所有指标名称满足metric定义正则表达式的指标名称
query_result(query)	返回prometheus查询语句的查询结果

例如，当需要监控Prometheus所有采集任务的状态时，可以使用如下方式，获取当前所有采集任务的名称：

```
label_values(up, job)
```

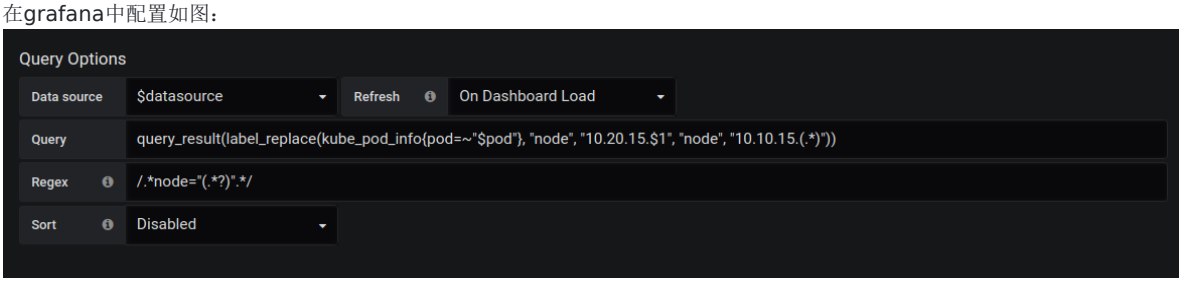
例如，有时候我们想要动态修改变量查询结果。比如某一个节点绑定了多个ip，一个用于内网访问，一个用于外网访问，此时prometheus采集到的指标是内网的ip，但我们需要的是外网ip。这里我们想要能在Grafana中动态改变标签值，进行ip段的替换，而避免从prometheus或exporter中修改采集指标。

这时需要使用grafana的query_result函数

```
# 将10.10.15.xxx段的ip地址替换为10.20.15.xxx段 注：替换端口同理
query_result(label_replace(kube_pod_info{pod=~"$pod"}, "node", "10.20.15.$1", "node", "10.10.15.(.*)"))
```

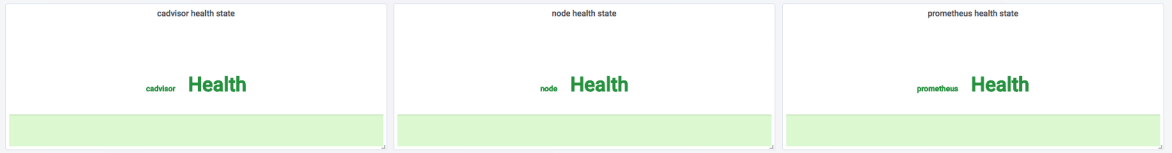
```
# 通过正则从返回结果中匹配出所需要的ip地址
regex: /.*node="(.*?)".*/
```

在grafana中配置如图：

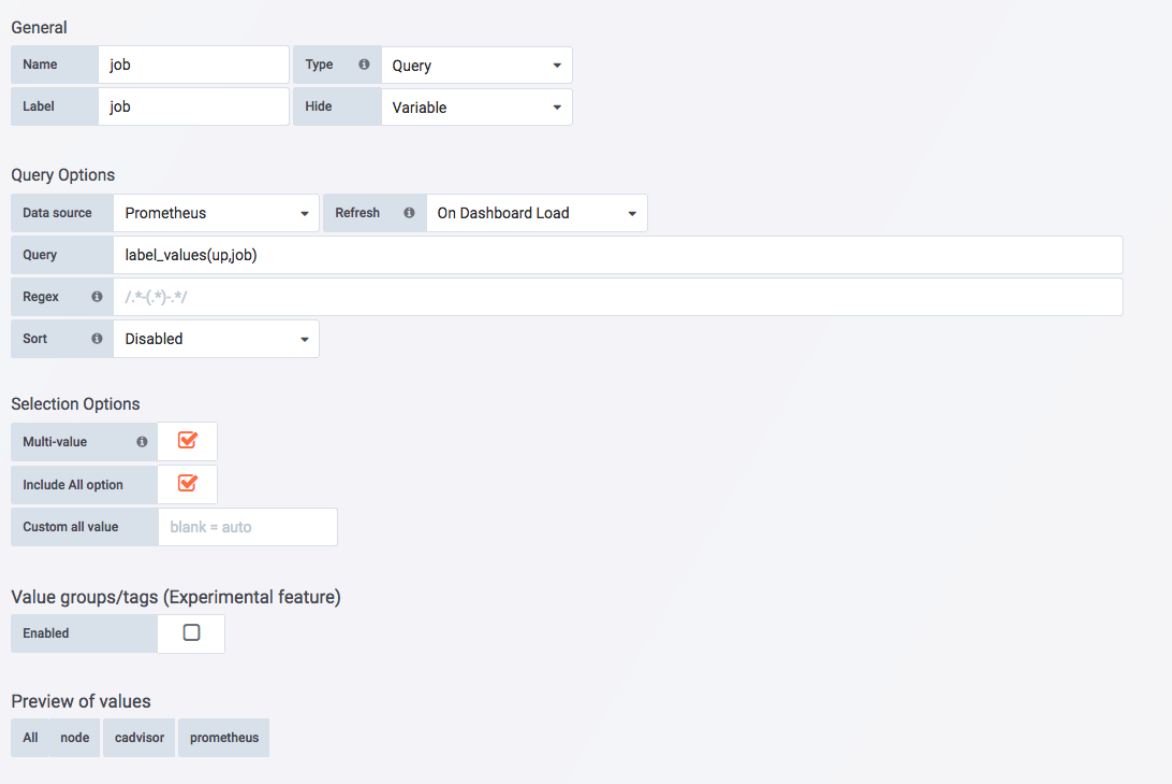


使用变量动态创建Panel和Row

当在一个Panel中展示多条时间序列数据时，通过使用变量可以轻松实现对时间序列的过滤，提高用户交互性。除此以外，我们还可以使用变量自动生成Panel或者Row。 如下所示，当需要可视化当前系统中所有采集任务的监控任务运行状态时，由于Prometheus的采集任务配置可能随时发生变更，通过硬编码的形式实现，会导致Dashboard配置的频繁变更：



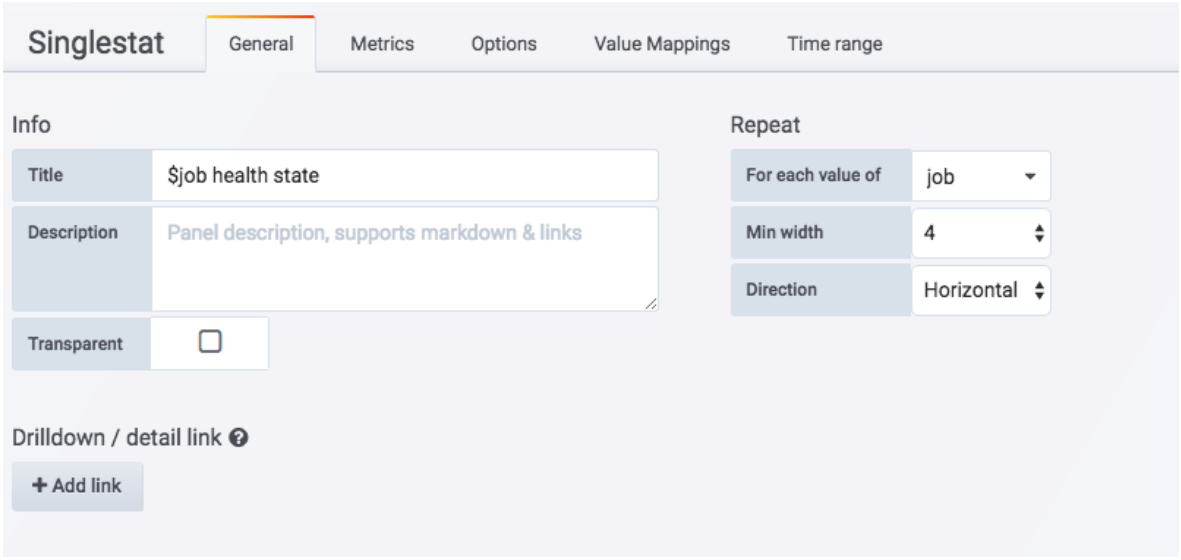
如下所示，这里为Dashboard定义了一遍名为job的变量：



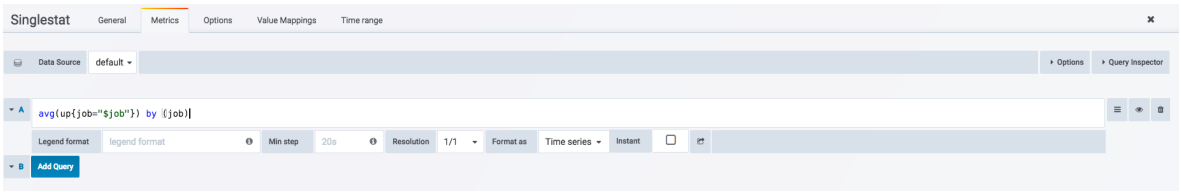
通过使用label_values函数，获取到当前Promthues监控指标up中所有可选的job标签的值：

```
label_values(up, job)
```

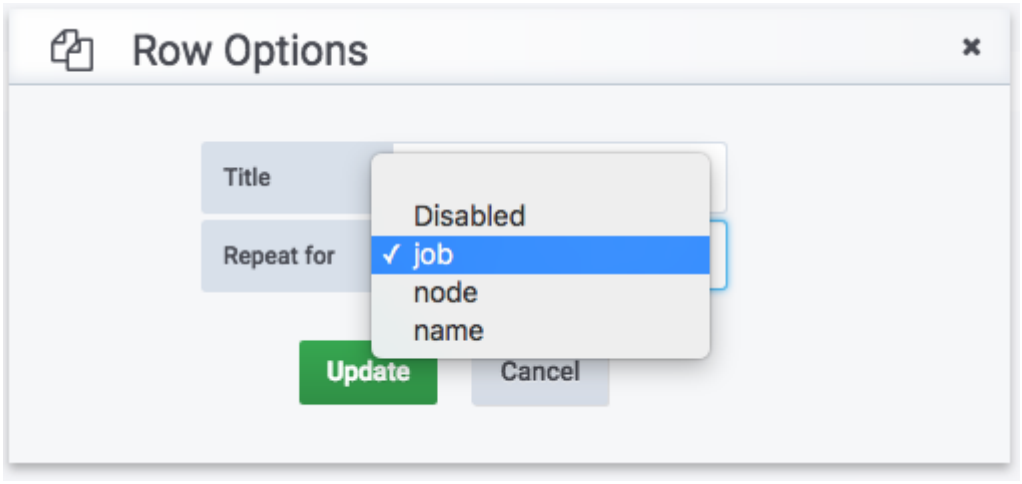
如果变量启用了Multi-value或者Include All Option选项的变量，那么在Panel的General选项的Repeat中可以选择自动迭代的变量，这里使用了Singlestat展示所有监控采集任务的状态：



Repeat选项设置完成后，Grafana会根据当前用户的选择，自动创建一个到多个Panel实例。 为了能够使Singlestat Panel能够展示正确的数据，如下所示，在Prometheus中，我们依然使用了$job变量，不过此时的$job反应的是当前迭代的值：



而如果还希望能够自动生成Row，只需要在Row的设置中，选择需要Repeat的变量即可：



小结

“You can’t fix what you can’t see”。可视化是监控的核心目标之一，在本章中我们学习了如何通过Prometheus内置的Console Template实现基本的可视化能力，以及通过更专业的开源工具Grafana实现Prometheus的数据可视化。

集群与高可用

Prometheus内置了一个基于本地存储的时间序列数据库。在Prometheus设计上，使用本地存储可以降低Prometheus部署和管理的复杂度同时减少高可用（HA）带来的复杂性。 在默认情况下，用户只需要部署多套Prometheus，采集相同的Targets即可实现基本的HA。同时由于Promethus高效的数据处理能力，单个Prometheus Server基本上能够应对大部分用户监控规模的需求。

当然本地存储也带来了一些不好的地方，首先就是数据持久化的问题，特别是在像Kubernetes这样的动态集群环境下，如果Promthues的实例被重新调度，那所有历史监控数据都会丢失。 其次本地存储也意味着Prometheus不适合保存大量历史数据(一般Prometheus推荐只保留几周或者几个月的数据)。最后本地存储也导致Prometheus无法进行弹性扩展。为了适应这方面的需求，Prometheus提供了remote_write和remote_read的特性，支持将数据存储到远端和从远端读取数据。通过将监控与数据分离，Prometheus能够更好地进行弹性扩展。

除了本地存储方面的问题，由于Prometheus基于Pull模型，当有大量的Target需要采样本时，单一Prometheus实例在数据抓取时可能会出现一些性能问题，联邦集群的特性可以让Prometheus将样本采集任务划分到不同的Prometheus实例中，并且通过一个统一的中心节点进行聚合，从而可以使Prometheuse可以根据规模进行扩展。

除了讨论Prometheus自身的高可用，Alertmanager作为Promthues体系中的告警处理中心，本章的最后部分会讨论如何实现Alertmanager的高可用部署。

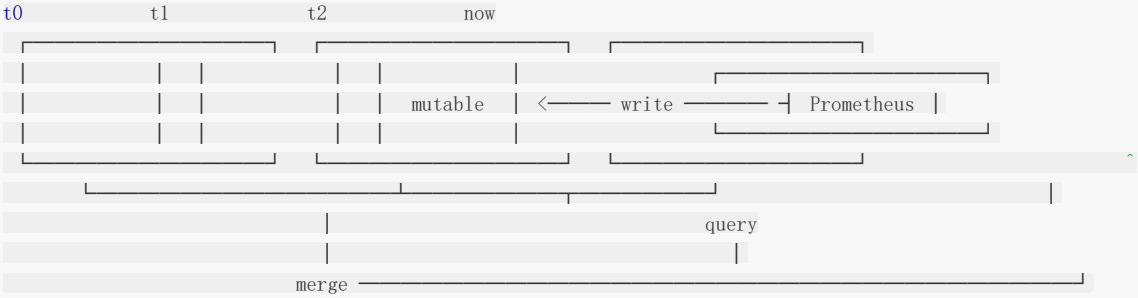
本章的主要内容：

- Prometheus本地存储机制
- Prometheus的远程存储机制
- Prometheus联邦集群
- Prometheus高可用部署架构
- Alertmanager高可用部署架构

本地存储

本地存储

Prometheus 2.x 采用自定义的存储格式将样本数据保存在本地磁盘当中。如下所示，按照两个小时为一个时间窗口，将两小时内产生的数据存储在一个块(Block)中，每一个块中包含该时间窗口内的所有样本数据(chunks)，元数据文件(meta.json)以及索引文件(index)。



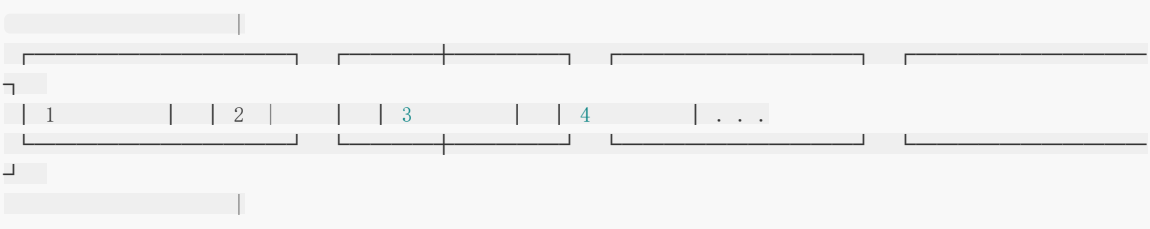
当前时间窗口内正在收集的样本数据，Prometheus则会直接将数据保存在内存当中。为了确保此期间如果Prometheus发生崩溃或者重启时能够恢复数据，Prometheus启动时会从写入日志(WAL)进行重播，从而恢复数据。此期间如果通过API删除时间序列，删除记录也会保存在单独的逻辑文件当中(tombstone)。

在文件系统中这些块保存在单独的目录当中，Prometheus保存块数据的目录结构如下所示：

```
./data
   |- 01BKGV7JBM69T2G1BGBGM6KB12 # 块
      |- meta.json  # 元数据
      |- wal        # 写入日志
        |- 000002
        |- 000001
   |- 01BKGTZQ1SYQJTR4PB43C8PD98  # 块
      |- meta.json  #元数据
      |- index   # 索引文件
      |- chunks  # 样本数据
        |- 000001
      |- tombstones # 逻辑数据
   |- 01BKGTZQ1HHWHV8FBJXW1Y3W0K
      |- meta.json
      |- wal
        |-000001
```

通过时间窗口的形式保存所有的样本数据，可以明显提高Prometheus的查询效率，当查询一段时间范围内的所有样本数据时，只需要简单的从落在该范围内的块中查询数据即可。

同时该存储方式可以简化历史数据的删除逻辑。只要一个块的时间范围落在了配置的保留范围之外，直接丢弃该块即可。



```
                    |
            retention boundary
```

本地存储配置

用户可以通过命令行启动参数的方式修改本地存储的配置。

启动参数	默认值	含义
–storage.tsdb.path	data/	Base path for metrics storage
–storage.tsdb.retention	15d	How long to retain samples in the storage
–storage.tsdb.min-block-duration	2h	The timestamp range of head blocks after which they get persisted
–storage.tsdb.max-block-duration	36h	The maximum timestamp range of compacted blocks,It’s the minimum duration of any persisted block.
–storage.tsdb.no-lockfile	false	Do not create lockfile in data directory

在一般情况下，Prometheus中存储的每一个样本大概占用1-2字节大小。如果需要对Prometheus Server的本地磁盘空间做容量规划时，可以通过以下公式计算：

```
needed_disk_space = retention_time_seconds * ingested_samples_per_second * bytes_per_sample
```

从上面公式中可以看出在保留时间(retention_time_seconds)和样本大小(bytes_per_sample)不变的情况下，如果想减少本地磁盘的容量需求，只能通过减少每秒获取样本数(ingested_samples_per_second)的方式。因此有两种手段，一是减少时间序列的数量，二是增加采集样本的时间间隔。考虑到Prometheus会对时间序列进行压缩效率，减少时间序列的数量效果更明显。

从失败中恢复

如果本地存储由于某些原因出现了错误，最直接的方式就是停止Prometheus并且删除data目录中的所有记录。当然也可以尝试删除那些发生错误的块目录，不过相应的用户会丢失该块中保存的大概两个小时的监控记录。

远程存储

Prometheus的本地存储设计可以减少其自身运维和管理的复杂度，同时能够满足大部分用户监控规模的需求。但是本地存储也意味着Prometheus无法持久化数据，无法存储大量历史数据，同时也无法灵活扩展和迁移。

为了保持Prometheus的简单性，Prometheus并没有尝试在自身中解决以上问题，而是通过定义两个标准接口(remote_write/remote_read)，让用户可以基于这两个接口对接将数据保存到任意第三方的存储服务中，这种方式在Promthues中称为Remote Storage。

Remote Write

用户可以在Prometheus配置文件中指定Remote Write(远程写)的URL地址，一旦设置了该配置项，Prometheus将采集到的样本数据通过HTTP的形式发送给适配器(Adaptor)。而用户则可以在适配器中对接外部任意的服务。外部服务可以是真正的存储系统，公有云的存储服务，也可以是消息队列等任意形式。

Remote Read

如下图所示，Promthues的Remote Read(远程读)也通过了一个适配器实现。在远程读的流程当中，当用户发起查询请求后，Promthues将向remote_read中配置的URL发起查询请求(matchers,ranges)，Adaptor根据请求条件从第三方存储服务中获取响应的数据。同时将数据转换为Promthues的原始样本数据返回给Prometheus Server。

当获取到样本数据后，Promthues在本地使用PromQL对样本数据进行二次处理。

> 注意：启用远程读设置后，只在数据查询时有效，对于规则文件的处理，以及Metadata API的处理都只基于Prometheus本地存储完成。

配置文件

Prometheus配置文件中添加remote_write和remote_read配置，其中url用于指定远程读/写的HTTP服务地址。如果该URL启动了认证则可以通过basic_auth进行安全认证配置。对于https的支持需要设定tls_concig。proxy_url主要用于Prometheus无法直接访问适配器服务的情况下。

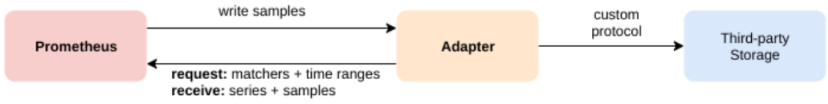
remote_write和remote_write具体配置如下所示：

```
remote_write:
    url: <string>
    [ remote_timeout: <duration> | default = 30s ]
    write_relabel_configs:
    [ - <relabel_config> ... ]
    basic_auth:
    [ username: <string> ]
    [ password: <string> ]
```

```
    [ bearer_token: <string> ]
    [ bearer_token_file: /path/to/bearer/token/file ]
    tls_config:
    [ <tls_config> ]
    [ proxy_url: <string> ]

remote_read:
    url: <string>
    required_matchers:
    [ <labelname>: <labelvalue> ... ]
    [ remote_timeout: <duration> | default = 30s ]
    [ read_recent: <boolean> | default = false ]
    basic_auth:
    [ username: <string> ]
    [ password: <string> ]
    [ bearer_token: <string> ]
    [ bearer_token_file: /path/to/bearer/token/file ]
    [ <tls_config> ]
    [ proxy_url: <string> ]
```

自定义Remote Storage Adaptor

实现自定义Remote Storage需要用户分别创建用于支持remote_read和remote_write的HTTP服务。



当前Prometheus中Remote Storage相关的协议主要通过以下proto文件进行定义：

```
syntax = "proto3";
package prometheus;

option go_package = "prompb";

import "types.proto";

message WriteRequest {
  repeated prometheus.TimeSeries timeseries = 1;
}

message ReadRequest {
  repeated Query queries = 1;
}

message ReadResponse {
  // In same order as the request's queries.
  repeated QueryResult results = 1;
}

message Query {
  int64 start_timestamp_ms = 1;
  int64 end_timestamp_ms = 2;
  repeated prometheus.LabelMatcher matchers = 3;
}
```

```
message QueryResult {
  // Samples within a time series must be ordered by time.
  repeated prometheus.TimeSeries timeseries = 1;
}
```

以下代码展示了一个简单的remote_write服务，创建用于接收remote_write的HTTP服务，将请求内容转换成WriteRequest后，用户就可以按照自己的需求进行后续的逻辑处理。

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"

    "github.com/gogo/protobuf/proto"
    "github.com/golang/snappy"
    "github.com/prometheus/common/model"

    "github.com/prometheus/prometheus/prompb"
)

func main() {
    http.HandleFunc("/receive", func(w http.ResponseWriter, r *http.Request) {
        compressed, err := ioutil.ReadAll(r.Body)
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }

        reqBuf, err := snappy.Decode(nil, compressed)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        var req prompb.WriteRequest
        if err := proto.Unmarshal(reqBuf, &req); err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        for _, ts := range req.Timeseries {
            m := make(model.Metric, len(ts.Labels))
            for _, l := range ts.Labels {
                m[model.LabelName(l.Name)] = model.LabelValue(l.Value)
            }
            fmt.Println(m)

            for _, s := range ts.Samples {
                fmt.Printf("  %f %d\n", s.Value, s.Timestamp)
            }
        }
    })

    http.ListenAndServe(":1234", nil)
}
```

使用Influxdb作为Remote Storage

目前Prometheus社区也提供了部分对于第三方数据库的Remote Storage支持：

存储服务	支持模式
AppOptics	write
Chronix	write
Cortex:	read/write
CrateDB	read/write
Gnocchi	write
Graphite	write
InfluxDB	read/write
OpenTSDB	write
PostgreSQL/TimescaleDB:	read/write
SignalFx	write

这里将介绍如何使用Influxdb作为Prometheus的Remote Storage，从而确保当Prometheus发生宕机或者重启之后能够从Influxdb中恢复和获取历史数据。

这里使用docker-compose定义并启动Influxdb数据库服务，docker-compose.yml定义如下：

```
version: '2'
services:
  influxdb:
    image: influxdb:1.3.5
    command: -config /etc/influxdb/influxdb.conf
    ports:
      - "8086:8086"
    environment:
      - INFLUXDB_DB=prometheus
      - INFLUXDB_ADMIN_ENABLED=true
      - INFLUXDB_ADMIN_USER=admin
      - INFLUXDB_ADMIN_PASSWORD=admin
      - INFLUXDB_USER=prom
      - INFLUXDB_USER_PASSWORD=prom
```

启动influxdb服务

```
$ docker-compose up -d
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORT
S                   NAMES
795d0ead87a1        influxdb:1.3.5      "/entrypoint.sh -c..."   3 hours ago         Up 3 hours          0.0.
0.0:8086->8086/tcp   localhost_influxdb_1
```

获取并启动Prometheus提供的Remote Storage Adapter：

```
go get github.com/prometheus/prometheus/documentation/examples/remote_storage/remote_storage_adapter
```

获取remote_storage_adapter源码后，go会自动把相关的源码编译成可执行文件，并且保存在$GOPATH/bin/目录下。

启动remote_storage_adapter并且设置Influxdb相关的认证信息：

```
INFLUXDB_PW=prom $GOPATH/bin/remote_storage_adapter -influxdb-url=http://localhost:8086 -influxdb.username=pr
om -influxdb.database=prometheus -influxdb.retention-policy=autogen
```

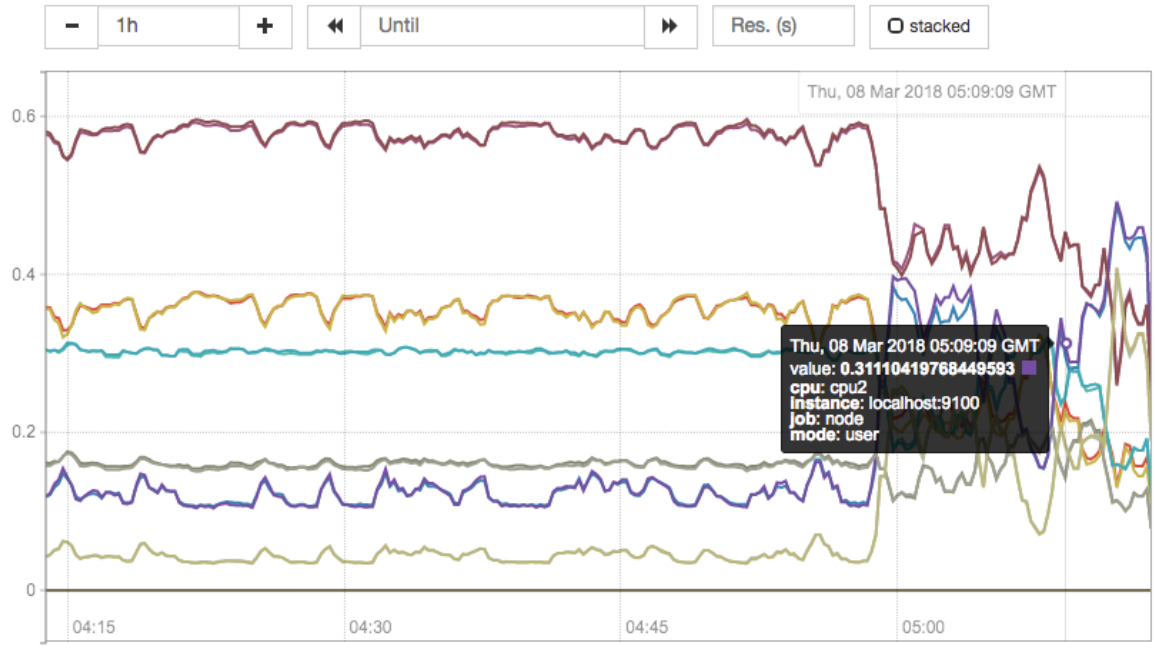
修改prometheus.yml添加Remote Storage相关的配置内容：

```
remote_write:
  - url: "http://localhost:9201/write"

remote_read:
  - url: "http://localhost:9201/read"
```

重新启动Prometheus能够获取数据后，登录到influxdb容器，并验证数据写入。如下所示，当数据能够正常写入Influxdb后可以看到Prometheus相关的指标。

```
docker exec -it 795d0ead87a1 influx
Connected to http://localhost:8086 version 1.3.5
InfluxDB shell version: 1.3.5
> auth
username: prom
password:

> use prometheus
> SHOW MEASUREMENTS
name: measurements
name
----
go_gc_duration_seconds
go_gc_duration_seconds_count
go_gc_duration_seconds_sum
go_goroutines
go_info
go_memstats_alloc_bytes
go_memstats_alloc_bytes_total
go_memstats_buck_hash_sys_bytes
go_memstats_frees_total
go_memstats_gc_cpu_fraction
go_memstats_gc_sys_bytes
go_memstats_heap_alloc_bytes
go_memstats_heap_idle_bytes
```

当数据写入成功后，停止Prometheus服务。同时删除Prometheus的data目录，模拟Promthues数据丢失的情况后重启Prometheus。打开Prometheus UI如果配置正常，Prometheus可以正常查询到本地存储以删除的历史数据记录。

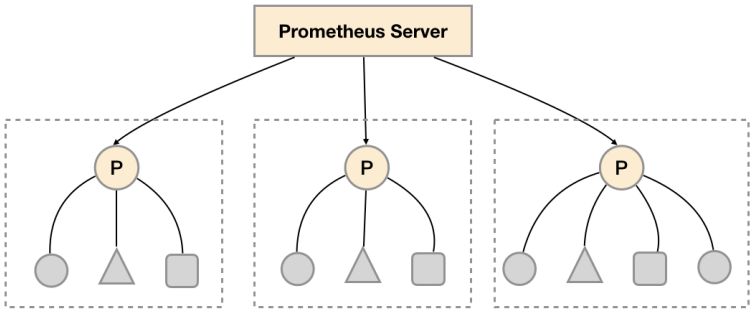

1h
Until
Res. (s)
stacked
Thu, 08 Mar 2018 05:09:09 GMT
0.6
0.4
0.2
0
Thu, 08 Mar 2018 05:09:09 GMT
value: 0.31110419768449593
cpu: cpu2
instance: localhost:9100
job: node
mode: user
04:15
04:30
04:45
05:00

联邦集群

通过Remote Storage可以分离监控样本采集和数据存储，解决Prometheus的持久化问题。这一部分会重点讨论如何利用联邦集群特性对Promthues进行扩展，以适应不同监控规模的变化。

使用联邦集群

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。



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。这一特性在Promthues中称为联邦集群。

联邦集群的核心在于每一个Prometheus Server都包含一个用于获取当前实例中监控样本的接口/federate。对于中心Prometheus Server而言，无论是从其他的Prometheus实例还是Exporter实例中获取数据实际上并没有任何差异。

```
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'
```

为了有效的减少不必要的时间序列，通过params参数可以用于指定只获取某些时间序列的样本数据，例如

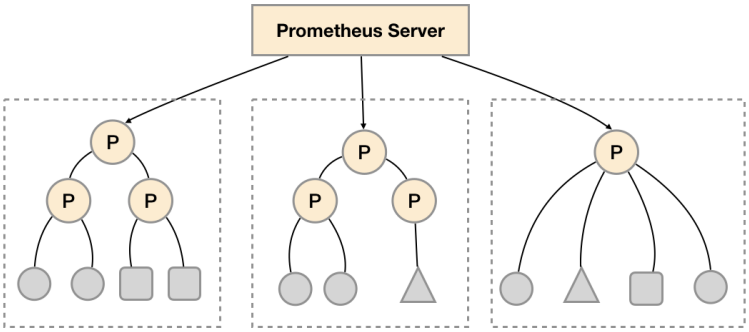
```
"http://192.168.77.11:9090/federate?match[]={job%3D"prometheus"}&match[]={__name__%3D~"job%3A.*"}&match[]={__name__%3D~"node.*"}"
```

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。

horbor_labels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

联邦集群的特性可以帮助用户根据不同的监控规模对Promthues部署架构进行调整。例如如下所示，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)，例如可以将不同的监控任务分离到不同的Prometheus实例当中，再有中心Prometheus实例进行聚合。



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。

Prometheus高可用

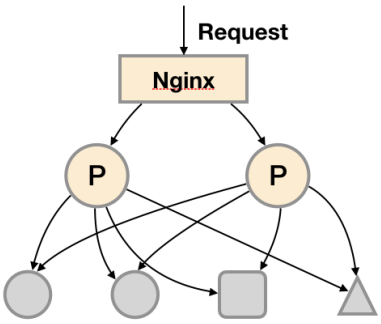
Prometheus的本地存储给Prometheus带来了简单高效的使用体验，可以让Promthues在单节点的情况下满足大部分用户的监控需求。但是本地存储也同时限制了Prometheus的可扩展性，带来了数据持久化等一系列的问题。通过Prometheus的Remote Storage特性可以解决这一系列问题，包括Promthues的动态扩展，以及历史数据的存储。

而除了数据持久化问题以外，影响Promthues性能表现的另外一个重要因素就是数据采集任务量，以及单台Promthues能够处理的时间序列数。因此当监控规模大到Promthues单台无法有效处理的情况下，可以选择利用Promthues的联邦集群的特性，将Promthues的监控任务划分到不同的实例当中。

这一部分将重点讨论Prometheus的高可用架构，并且根据不同的使用场景介绍了一种常见的高可用方案。

基本HA：服务可用性

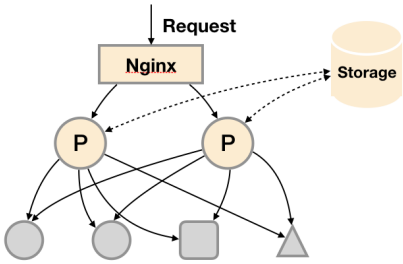
由于Promthues的Pull机制的设计，为了确保Promthues服务的可用性，用户只需要部署多套Prometheus Server实例，并且采集相同的Exporter目标即可。



基本的HA模式只能确保Promthues服务的可用性问题，但是不解决Prometheus Server之间的数据一致性问题以及持久化问题(数据丢失后无法恢复)，也无法进行动态的扩展。因此这种部署方式适合监控规模不大，Promthues Server也不会频繁发生迁移的情况，并且只需要保存短周期监控数据的场景。

基本HA + 远程存储

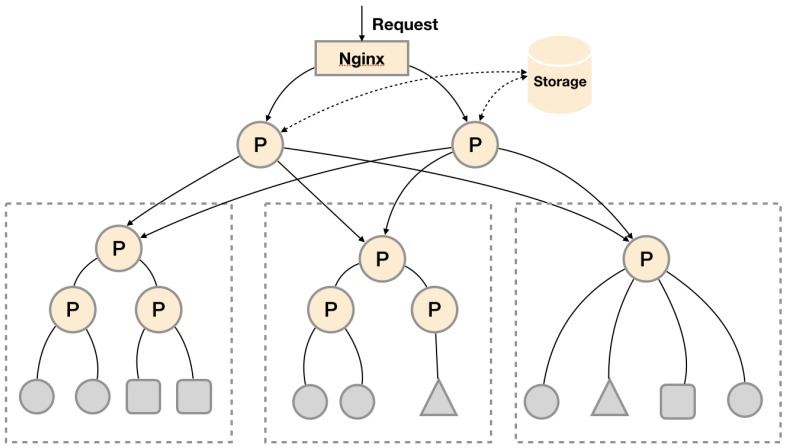
在基本HA模式的基础上通过添加Remote Storage存储支持，将监控数据保存在第三方存储服务上。



在解决了Promthues服务可用性的基础上，同时确保了数据的持久化，当Promthues Server发生宕机或者数据丢失的情况下，可以快速的恢复。 同时Promthues Server可能很好的进行迁移。因此，该方案适用于用户监控规模不大，但是希望能够将监控数据持久化，同时能够确保Promthues Server的可迁移性的场景。

基本HA + 远程存储 + 联邦集群

当单台Promthues Server无法处理大量的采集任务时，用户可以考虑基于Prometheus联邦集群的方式将监控采集任务划分到不同的Promthues实例当中即在任务级别功能分区。



这种部署方式一般适用于两种场景：

场景一：单数据中心 + 大量的采集任务

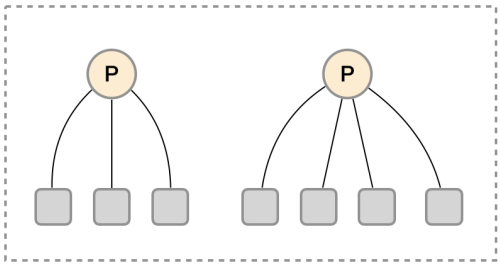
这种场景下Promthues的性能瓶颈主要在于大量的采集任务，因此用户需要利用Prometheus联邦集群的特性，将不同类型的采集任务划分到不同的Promthues子服务中，从而实现功能分区。例如一个Promthues Server负责采集基础设施相关的监控指标，另外一个Prometheus Server负责采集应用监控指标。再有上层Prometheus Server实现对数据的汇聚。

场景二：多数据中心

这种模式也适合与多数据中心的情况，当Promthues Server无法直接与数据中心中的Exporter进行通讯时，在每一个数据中部署一个单独的Promthues Server负责当前数据中心的采集任务是一个不错的方式。这样可以避免用户进行大量的网络配置，只需要确保主Promthues Server实例能够与当前数据中心的Prometheus Server通讯即可。 中心Promthues Server负责实现对多数据中心数据的聚合。

按照实例进行功能分区

这时在考虑另外一种极端情况，即单个采集任务的Target数也变得非常巨大。这时简单通过联邦集群进行功能分区，Prometheus Server也无法有效处理时。这种情况只能考虑继续在实例级别进行功能划分。



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
    relabel_configs:
    - source_labels: [__address__]
      modulus:       4
      target_label:  __tmp_hash
      action:        hashmod
    - source_labels: [__tmp_hash]
      regex:         ^1$
      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
```

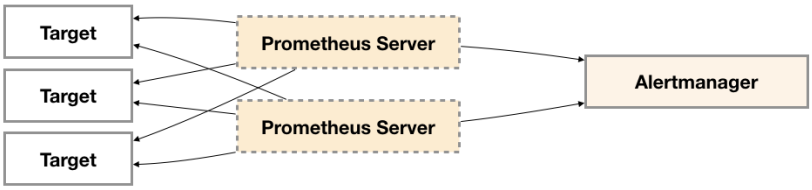
高可用方案选择

上面的部分，根据不同的场景演示了3种不同的高可用部署方案。当然对于Promthues部署方案需要用户根据监控规模以及自身的需求进行动态调整，下表展示了Promthues和高可用有关3个选项各自解决的问题，用户可以根据自己的需求灵活选择。

选项\需求	服务可用性	数据持久化	水平扩展
主备HA	v	x	x
远程存储	x	v	x
联邦集群	x	x	v

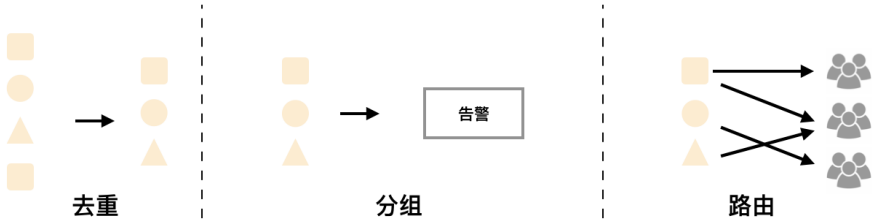
Alertmanager高可用

在上一小节中我们主要讨论了Prometheus Server自身的高可用问题。而接下来，重点将放在告警处理也就是Alertmanager部分。如下所示。



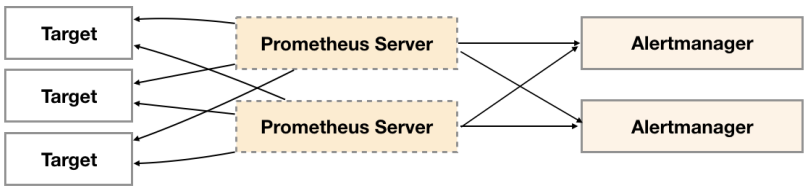
为了提升Promthues的服务可用性，通常用户会部署两个或者两个以上的Promthus Server，它们具有完全相同的配置包括Job配置，以及告警配置等。当某一个Prometheus Server发生故障后可以确保Promthues持续可用。

同时基于Alertmanager的告警分组机制即使不同的Prometheus Sever分别发送相同的告警给Alertmanager，Alertmanager也可以自动将这些告警合并为一个通知向receiver发送。

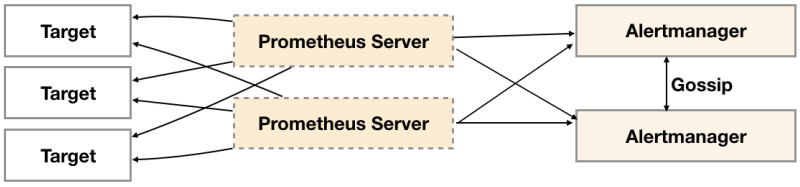


但不幸的是，虽然Alertmanager能够同时处理多个相同的Prometheus Server所产生的告警。但是由于单个Alertmanager的存在，当前的部署结构存在明显的单点故障风险，当Alertmanager单点失效后，告警的后续所有业务全部失效。

如下所示，最直接的方式，就是尝试部署多套Alertmanager。但是由于Alertmanager之间不存在并不了解彼此的存在，因此则会出现告警通知被不同的Alertmanager重复发送多次的问题。

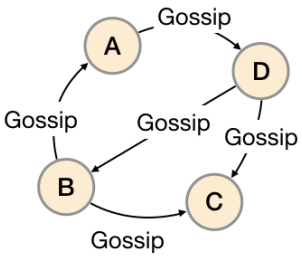


为了解决这一问题，如下所示。Alertmanager引入了Gossip机制。Gossip机制为多个Alertmanager之间提供了信息传递的机制。确保及时在多个Alertmanager分别接收到相同告警信息的情况下，也只有一个告警通知被发送给Receiver。



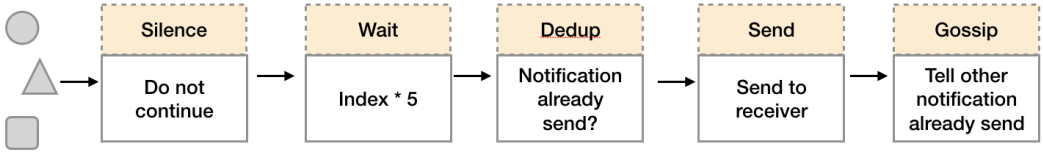
Gossip协议

Gossip是分布式系统中被广泛使用的协议，用于实现分布式节点之间的信息交换和状态同步。Gossip协议同步状态类似于流言或者病毒的传播，如下所示：



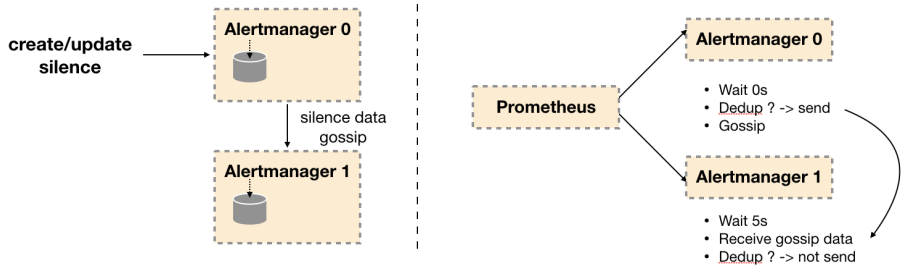
一般来说Gossip有两种实现方式分别为Push-based和Pull-based。在Push-based当集群中某一节点A完成一个工作后，随机的从其它节点B并向其发送相应的消息，节点B接收到消息后在重复完成相同的工作，直到传播到集群中的所有节点。而Pull-based的实现中节点A会随机的向节点B发起询问是否有新的状态需要同步，如果有则返回。

在简单了解了Gossip协议之后，我们来看Alertmanager是如何基于Gossip协议实现集群高可用的。如下所示，当Alertmanager接收到来自Prometheus的告警消息后，会按照以下流程对告警进行处理：



1. 在第一个阶段Silence中，Alertmanager会判断当前通知是否匹配到任何的静默规则，如果没有则进入下一个阶段，否则则中断流水线不发送通知。
2. 在第二个阶段Wait中，Alertmanager会根据当前Alertmanager在集群中所在的顺序(index)等待index * 5s的时间。
3. 当前Alertmanager等待阶段结束后，Dedup阶段则会判断当前Alertmanager数据库中该通知是否已经发送，如果已经发送则中断流水线，不发送告警，否则则进入下一阶段Send对外发送告警通知。
4. 告警发送完成后该Alertmanager进入最后一个阶段Gossip，Gossip会通知其他Alertmanager实例当前告警已经发送。其他实例接收到Gossip消息后，则会在自己的数据库中保存该通知已发送的记录。

因此如下所示，Gossip机制的关键在于两点：



- Silence设置同步：Alertmanager启动阶段基于Pull-based从集群其它节点同步Silence状态，当有新的Silence产生时使用Push-based方式在集群中传播Gossip信息。
- 通知发送状态同步：告警通知发送完成后，基于Push-based同步告警发送状态。Wait阶段可以确保集群状态一致。

Alertmanager基于Gossip实现的集群机制虽然不能保证所有实例上的数据时刻保持一致，但是实现了CAP理论中的AP系统，即可用性和分区容错性。同时对于Prometheus Server而言保持了配置了简单性，Promthues Server之间不需要任何的状态同步。

搭建本地集群环境

为了能够让Alertmanager节点之间进行通讯，需要在Alertmanager启动时设置相应的参数。其中主要的参数包括：

- –cluster.listen-address string: 当前实例集群服务监听地址
- –cluster.peer value: 初始化时关联的其它实例的集群服务地址

例如：

定义Alertmanager实例a1，其中Alertmanager的服务运行在9093端口，集群服务地址运行在8001端口。

```
alertmanager  --web.listen-address=":9093" --cluster.listen-address="127.0.0.1:8001" --config.file=/etc/prometheus/alertmanager.yml  --storage.path=/data/alertmanager/
```

定义Alertmanager实例a2，其中主服务运行在9094端口，集群服务运行在8002端口。为了将a1，a2组成集群。 a2启动时需要定义–cluster.peer参数并且指向a1实例的集群服务地址:8001。

```
alertmanager  --web.listen-address=":9094" --cluster.listen-address="127.0.0.1:8002" --cluster.peer=127.0.0.1:8001 --config.file=/etc/prometheus/alertmanager.yml  --storage.path=/data/alertmanager2/
```

为了能够在本地模拟集群环境，这里使用了一个轻量级的多线程管理工具goreman。使用以下命令可以在本地安装goreman命令行工具。

```
go get github.com/mattn/goreman
```

创建Alertmanager集群

创建Alertmanager配置文件/etc/prometheus/alertmanager-ha.yml, 为了验证Alertmanager的集群行为，这里在本地启动一个webhook服务用于打印Alertmanager发送的告警通知信息。

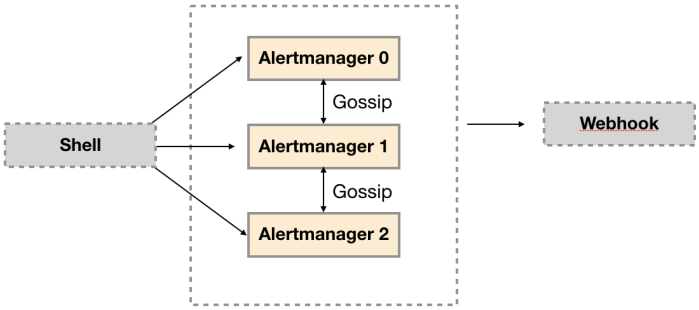
```
route:
  receiver: 'default-receiver'
receivers:
```

```
  - name: default-receiver
    webhook_configs:
    - url: 'http://127.0.0.1:5001/'
```

本地webhook服务可以直接从Github获取。

```
# 获取alertmanager提供的webhook示例，如果该目录下定义了main函数，go get会自动将其编译成可执行文件
go get github.com/prometheus/alertmanager/examples/webhook
# 设置环境变量指向GOPATH的bin目录
export PATH=$GOPATH/bin:$PATH
# 启动服务
webhook
```

示例结构如下所示：



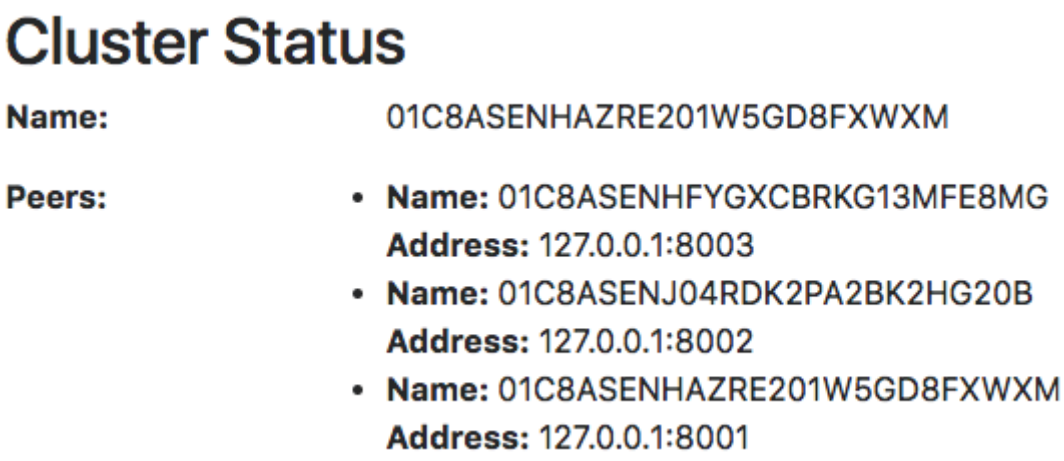
创建alertmanager.procfile文件，并且定义了三个Alertmanager节点（a1，a2，a3）以及用于接收告警通知的webhook服务:

```
a1: alertmanager  --web.listen-address=":9093" --cluster.listen-address="127.0.0.1:8001" --config.file=/etc/prometheus/alertmanager-ha.yml  --storage.path=/data/alertmanager/ --log.level=debug
a2: alertmanager  --web.listen-address=":9094" --cluster.listen-address="127.0.0.1:8002" --cluster.peer=127.0.0.1:8001 --config.file=/etc/prometheus/alertmanager-ha.yml  --storage.path=/data/alertmanager2/ --log.level=debug
a3: alertmanager  --web.listen-address=":9095" --cluster.listen-address="127.0.0.1:8003" --cluster.peer=127.0.0.1:8001 --config.file=/etc/prometheus/alertmanager-ha.yml  --storage.path=/data/alertmanager2/ --log.level=debug

webhook: webhook
```

在Procfile文件所在目录，执行goreman start命令，启动所有进程:

```
$ goreman -f alertmanager.procfile start
10:27:57      a1 | level=debug ts=2018-03-12T02:27:57.399166371Z caller=cluster.go:125 component=cluster msg="joined cluster" peers=0
10:27:57      a3 | level=info ts=2018-03-12T02:27:57.40004678Z caller=main.go:346 msg=Listening address=:9095
10:27:57      a1 | level=info ts=2018-03-12T02:27:57.400212246Z caller=main.go:271 msg="Loading configuration file" file=/etc/prometheus/alertmanager.yml
10:27:57      a1 | level=info ts=2018-03-12T02:27:57.405638714Z caller=main.go:346 msg=Listening address=:9093
```

启动完成后访问任意Alertmanager节点http://localhost:9093/#/status,可以查看当前Alertmanager集群的状态。



当集群中的Alertmanager节点不在一台主机时，通常需要使用–cluster.advertise-address参数指定当前节点所在网络地址。

> 注意：由于goreman不保证进程之间的启动顺序，如果集群状态未达到预期，可以使用 `goreman -f alertmanager.procfile run restart a2` 重启a2，a3服务。

当Alertmanager集群启动完成后，可以使用send-alerts.sh脚本对集群进行简单测试，这里利用curl分别向3个Alertmanager实例发送告警信息。

```
alerts1='[
  {
    "labels": {
       "alertname": "DiskRunningFull",
       "dev": "sda1",
       "instance": "example1"
    },
    "annotations": {
       "info": "The disk sda1 is running full",
       "summary": "please check the instance example1"
    }
  },
  {
    "labels": {
       "alertname": "DiskRunningFull",
       "dev": "sdb2",
       "instance": "example2"
    },
    "annotations": {
       "info": "The disk sdb2 is running full",
       "summary": "please check the instance example2"
    }
  },
  {
    "labels": {
       "alertname": "DiskRunningFull",
       "dev": "sda1",
       "instance": "example3",
       "severity": "critical"
    }
  },
  {
    "labels": {
       "alertname": "DiskRunningFull",
       "dev": "sda1",
       "instance": "example3",
```

```
      "severity": "warning"
    }
  }
]'

curl -XPOST -d"$alerts1" http://localhost:9093/api/v1/alerts
curl -XPOST -d"$alerts1" http://localhost:9094/api/v1/alerts
curl -XPOST -d"$alerts1" http://localhost:9095/api/v1/alerts
```

运行send-alerts.sh后，查看alertmanager日志，可以看到以下输出，3个Alertmanager实例分别接收到模拟的告警信息：

```
10:43:36      a1 | level=debug ts=2018-03-12T02:43:36.853370185Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=DiskRunningFull[6543bc1][active]
10:43:36      a2 | level=debug ts=2018-03-12T02:43:36.871180749Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=DiskRunningFull[8320f0a][active]
10:43:36      a3 | level=debug ts=2018-03-12T02:43:36.894923811Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=DiskRunningFull[8320f0a][active]
```

查看webhook日志只接收到一个告警通知：

```
10:44:06 webhook | 2018/03/12 10:44:06 {
10:44:06 webhook |  >  "receiver": "default-receiver",
10:44:06 webhook |  >  "status": "firing",
10:44:06 webhook |  >  "alerts": [
10:44:06 webhook |  >    {
10:44:06 webhook |  >      "status": "firing",
10:44:06 webhook |  >      "labels": {
10:44:06 webhook |  >        "alertname": "DiskRunningFull",
```

多实例Prometheus与Alertmanager集群

由于Gossip机制的实现，在Promthues和Alertmanager实例之间不要使用任何的负载均衡，需要确保Promthues将告警发送到所有的Alertmanager实例中：

```
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - 127.0.0.1:9093
      - 127.0.0.1:9094
      - 127.0.0.1:9095
```

创建Promthues集群配置文件/etc/prometheus/prometheus-ha.yml，完整内容如下：

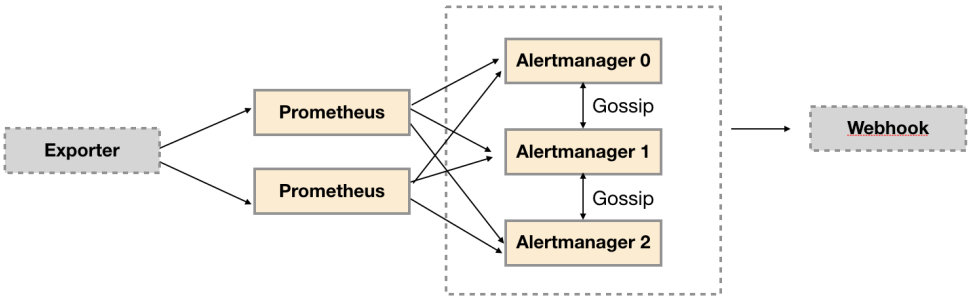
```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 15s
rule_files:
  - /etc/prometheus/rules/*.rules
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - 127.0.0.1:9093
      - 127.0.0.1:9094
```

```
      - 127.0.0.1:9095
scrape_configs:
- job_name: prometheus
  static_configs:
  - targets:
    - localhost:9090
- job_name: 'node'
  static_configs:
  - targets: ['localhost:9100']
```

同时定义告警规则文件/etc/prometheus/rules/hoststats-alert.rules，如下所示：

```
groups:
- name: hostStatsAlert
  rules:
  - alert: hostCpuUsageAlert
    expr: sum(avg without (cpu)(irate(node_cpu{mode!='idle'}[5m]))) by (instance) * 100 > 50
    for: 1m
    labels:
      severity: page
    annotations:
      summary: "Instance {{ $labels.instance }} CPU usgae high"
      description: "{{ $labels.instance }} CPU usage above 50% (current value: {{ $value }})"
  - alert: hostMemUsageAlert
    expr: (node_memory_MemTotal - node_memory_MemAvailable)/node_memory_MemTotal * 100 > 85
    for: 1m
    labels:
      severity: page
    annotations:
      summary: "Instance {{ $labels.instance }} MEM usgae high"
      description: "{{ $labels.instance }} MEM usage above 85% (current value: {{ $value }})"
```

本示例部署结构如下所示：



创建prometheus.procfile文件，创建两个Promthues节点，分别监听9090和9091端口：

```
p1: prometheus --config.file=/etc/prometheus/prometheus-ha.yml --storage.tsdb.path=/data/prometheus/ --web.li
sten-address="127.0.0.1:9090"
p2: prometheus --config.file=/etc/prometheus/prometheus-ha.yml --storage.tsdb.path=/data/prometheus2/ --web.l
isten-address="127.0.0.1:9091"

node_exporter: node_exporter -web.listen-address="0.0.0.0:9100"
```

使用goreman启动多节点Promthues：

```
goreman -f prometheus.procfile -p 8556 start
```

Promthues启动完成后，手动拉高系统CPU使用率：

```
cat /dev/zero>/dev/null
```

> 注意，对于多核主机，如果CPU达不到预期，运行多个命令。

当CPU利用率达到告警规则触发条件，两个Prometheus实例告警分别被触发。查看Alertmanager输出日志：

```
11:14:41      a3 | level=debug ts=2018-03-12T03:14:41.945493505Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=hostCpuUsageAlert[7d698ac][active]
11:14:41      a1 | level=debug ts=2018-03-12T03:14:41.945534548Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=hostCpuUsageAlert[7d698ac][active]
11:14:41      a2 | level=debug ts=2018-03-12T03:14:41.945687812Z caller=dispatch.go:188 component=dispatcher
 msg="Received alert" alert=hostCpuUsageAlert[7d698ac][active]
```

3个Alertmanager实例分别接收到来自不同Prometheus实例的告警信息。而Webhook服务只接收到来自Alertmanager集群的一条告警通知：

```
11:15:11 webhook | 2018/03/12 11:15:11 {
11:15:11 webhook |  >  "receiver": "default-receiver",
11:15:11 webhook |  >  "status": "firing",
11:15:11 webhook |  >  "alerts": [
11:15:11 webhook |  >    {
11:15:11 webhook |  >      "status": "firing",
11:15:11 webhook |  >      "labels": {
11:15:11 webhook |  >        "alertname": "hostCpuUsageAlert",
```

小结

Prometheus的简单性贯穿于整个Prometheus的使用过程中，无论是单机部署还是集群化部署，简单性一致是Prometheus设计的基本原则。这本章中，我们系统学习了如果实现Prometheus下各个中间的高可用部署方式，同时给出了集中常用的高可用方案，读者可以根据自己的实际需求来选择如何部署自己的Promethues集群。

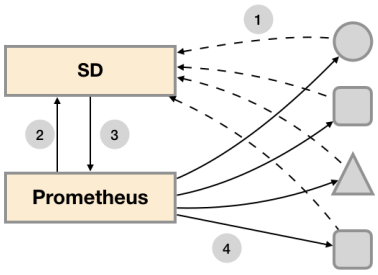
Prometheus服务发现

Prometheus与服务发现

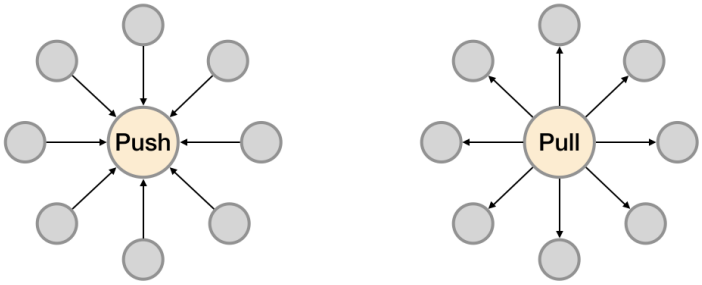
在基于云(IaaS或者CaaS)的基础设施环境中用户可以像使用水、电一样按需使用各种资源（计算、网络、存储）。按需使用就意味着资源的动态性，这些资源可以随着需求规模的变化而变化。例如在AWS中就提供了专门的AutoScall服务，可以根据用户定义的规则动态地创建或者销毁EC2实例，从而使用户部署在AWS上的应用可以自动的适应访问规模的变化。

这种按需的资源使用方式对于监控系统而言就意味着没有了一个固定的监控目标，所有的监控对象(基础设施、应用、服务)都在动态的变化。对于Nagias这类基于Push模式传统监控软件就意味着必须在每一个节点上安装相应的Agent程序，并且通过配置指向中心的Nagias服务，受监控的资源与中心监控服务器之间是一个强耦合的关系，要么直接将Agent构建到基础设施镜像当中，要么使用一些自动化配置管理工具(如Ansible、Chef)动态的配置这些节点。当然实际场景下除了基础设施的监控需求以外，我们还需要监控在云上部署的应用，中间件等等各种各样的服务。要搭建起这样一套中心化的监控系统实施成本和难度是显而易见的。

而对于Prometheus这一类基于Pull模式的监控系统，显然也无法继续使用的static_configs的方式静态的定义监控目标。而对于Prometheus而言其解决方案就是引入一个中间的代理人（服务注册中心），这个代理人掌握着当前所有监控目标的访问信息，Prometheus只需要向这个代理人询问有哪些监控目标即可， 这种模式被称为服务发现。



在不同的场景下，会有不同的东西扮演代理人（服务发现与注册中心）这一角色。比如在AWS公有云平台或者OpenStack的私有云平台中，由于这些平台自身掌握着所有资源的信息，此时这些云平台自身就扮演了代理人的角色。Prometheus通过使用平台提供的API就可以找到所有需要监控的云主机。在Kubernetes这类容器管理平台中，Kubernetes掌握并管理着所有的容器以及服务信息，那此时Prometheus只需要与Kubernetes打交道就可以找到所有需要监控的容器以及服务对象。Prometheus还可以直接与一些开源的服务发现工具进行集成，例如在微服务架构的应用程序中，经常会使用到例如Consul这样的服务发现注册软件，Promethues也可以与其集成从而动态的发现需要监控的应用服务实例。除了与这些平台级的公有云、私有云、容器云以及专门的服务发现注册中心集成以外，Prometheus还支持基于DNS以及文件的方式动态发现监控目标，从而大大的减少了在云原生，微服务以及云模式下监控实施难度。



如上所示，展示了Push系统和Pull系统的核心差异。相较于Push模式，Pull模式的优点可以简单总结为以下几点：

- 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；
- 你可以更容易的查看监控目标实例的健康状态，并且可以快速定位故障；
- 更利于构建DevOps文化的团队；
- 松耦合的架构模式更适合于云原生的部署环境。

基于文件的服务发现

在Prometheus支持的众多服务发现的实现方式中，基于文件的服务发现是最通用的方式。这种方式不需要依赖于任何的平台或者第三方服务。对于Prometheus而言也不可能支持所有的平台或者环境。通过基于文件的服务发现方式下，Prometheus会定时从文件中读取最新的Target信息，因此，你可以通过任意的方式将监控Target的信息写入即可。

用户可以通过JSON或者YAML格式的文件，定义所有的监控目标。例如，在下面的JSON文件中分别定义了3个采集任务，以及每个任务对应的Target列表：

```
[
  {
    "targets": [ "localhost:8080"],
    "labels": {
      "env": "localhost",
      "job": "cadvisor"
    }
  },
  {
    "targets": [ "localhost:9104" ],
    "labels": {
      "env": "prod",
      "job": "mysqld"
    }
  },
  {
    "targets": [ "localhost:9100"],
    "labels": {
      "env": "prod",
      "job": "node"
    }
  }
]
```

同时还可以通过为这些实例添加一些额外的标签信息，例如使用env标签标示当前节点所在的环境，这样从这些实例中采集到的样本信息将包含这些标签信息，从而可以通过该标签按照环境对数据进行统计。

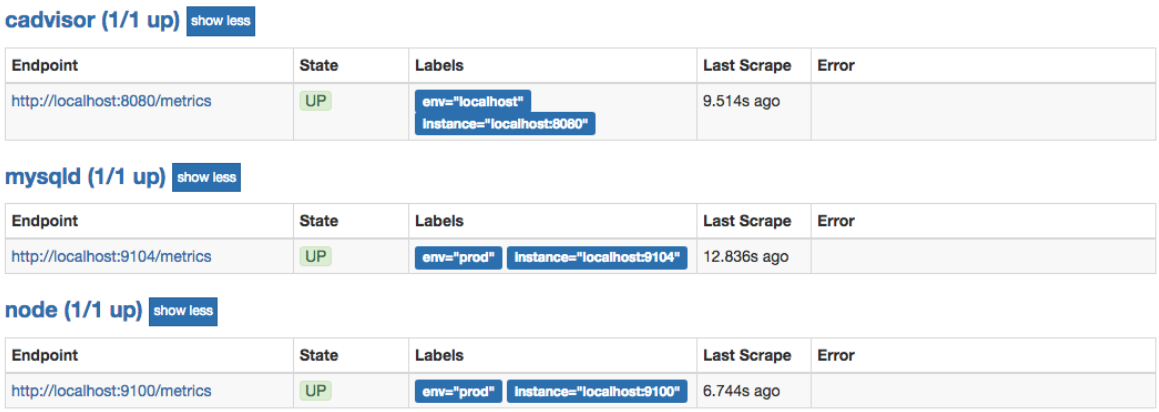
创建Prometheus配置文件/etc/prometheus/prometheus-file-sd.yml，并添加以下内容：

```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 15s
scrape_configs:
- job_name: 'file_ds'
  file_sd_configs:
  - files:
    - targets.json
```

这里定义了一个基于file_sd_configs的监控采集任务，其中模式的任务名称为file_ds。在JSON文件中可以使用job标签覆盖默认的job名称，此时启动Prometheus服务：

```
prometheus --config.file=/etc/prometheus/prometheus-file-sd.yml --storage.tsdb.path=/data/prometheus
```

在Prometheus UI的Targets下就可以看到当前从targets.json文件中动态获取到的Target实例信息以及监控任务的采集状态，同时在Labels列下会包含用户添加的自定义标签：



cadvisor (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://localhost:8080/metrics	UP	env="localhost" instance="localhost:8080"	9.514s ago	

mysqld (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9104/metrics	UP	env="prod" instance="localhost:9104"	12.836s ago	

node (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9100/metrics	UP	env="prod" instance="localhost:9100"	6.744s ago	

Prometheus默认每5m重新读取一次文件内容，当需要修改时，可以通过refresh_interval进行设置，例如：

```
- job_name: 'file_ds'
  file_sd_configs:
  - refresh_interval: 1m
    files:
    - targets.json
```

通过这种方式，Prometheus会自动的周期性读取文件中的内容。当文件中定义的内容发生变化时，不需要对Prometheus进行任何的重启操作。

这种通用的方式可以衍生了很多不同的玩法，比如与自动化配置管理工具(Ansible)结合、与Cron Job结合等等。 对于一些Prometheus还不支持的云环境，比如国内的阿里云、腾讯云等也可以使用这种方式通过一些自定义程序与平台进行交互自动生成监控Target文件，从而实现对这些云环境中基础设施的自动化监控支持。

基于Consul的服务发现

Consul是由HashiCorp开发的一个支持多数据中心的分布式服务发现和键值对存储服务的开源软件，被大量应用于基于微服务的软件架构当中。

Consul初体验

用户可以通过Consul官网https://www.consul.io/downloads.html下载对应操作系统版本的软件包。Consul与Prometheus同样使用Go语言进行开发，因此安装和部署的方式也极为简单，解压并将命令行工具放到系统PATH路径下即可。

在本地可以使用开发者模式在本地快速启动一个单节点的Consul环境：

```
$ consul agent -dev
==> Starting Consul agent...
==> Consul agent running!
           Version: 'v1.0.7'
           Node ID: 'd7b590ba-e2f8-3a82-e8a8-2a911bdf67d5'
         Node name: 'localhost'
        Datacenter: 'dc1' (Segment: '<all>')
            Server: true (Bootstrap: false)
       Client Addr: [127.0.0.1] (HTTP: 8500, HTTPS: -1, DNS: 8600)
      Cluster Addr: 127.0.0.1 (LAN: 8301, WAN: 8302)
           Encrypt: Gossip: false, TLS-Outgoing: false, TLS-Incoming: false
```

在启动成功后，在一个新的terminal窗口中运行consul members可以查看当前集群中的所有节点：

```
$ consul members
Node       Address         Status  Type    Build  Protocol  DC   Segment
localhost  127.0.0.1:8301  alive   server  1.0.7  2         dc1  <all>
```

用户还可以通过HTTP API的方式查看当前集群中的节点信息：

```
$ curl localhost:8500/v1/catalog/nodes
[
    {
        "ID": "d7b590ba-e2f8-3a82-e8a8-2a911bdf67d5",
        "Node": "localhost",
        "Address": "127.0.0.1",
        "Datacenter": "dc1",
        "TaggedAddresses": {
            "lan": "127.0.0.1",
            "wan": "127.0.0.1"
        },
        "Meta": {
            "consul-network-segment": ""
        },
        "CreateIndex": 5,
        "ModifyIndex": 6
    }
]
```

Consul还提供了内置的DNS服务，可以通过Consul的DNS服务的方式访问其中的节点：

```
$ dig @127.0.0.1 -p 8600 localhost.node.consul

; <<>> DiG 9.9.7-P3 <<>> @127.0.0.1 -p 8600 localhost.node.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50684
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;localhost.node.consul.         IN    A

;; ANSWER SECTION:
localhost.node.consul.    0    IN    A    127.0.0.1

;; Query time: 5 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Sun Apr 15 22:10:56 CST 2018
;; MSG SIZE  rcvd: 66
```

在Consul当中服务可以通过服务定义文件或者是HTTP API的方式进行注册。这里使用服务定义文件的方式将本地运行的node_exporter通过服务的方式注册到Consul当中。

创建配置目录：

```
sudo mkdir /etc/consul.d
```

```
echo '{"service": {"name": "node_exporter", "tags": ["exporter"], "port": 9100}}' \
    | sudo tee /etc/consul.d/node_exporter.json
```

重新启动Consul服务，并且声明服务定义文件所在目录：

```
$ consul agent -dev -config-dir=/etc/consul.d
==> Starting Consul agent...
    2018/04/15 22:23:47 [DEBUG] agent: Service "node_exporter" in sync
```

一旦服务注册成功之后，用户就可以通过DNS或HTTP API的方式查询服务信息。默认情况下，所有的服务都可以使用NAME.service.consul域名的方式进行访问。

例如，可以使用node_exporter.service.consul域名查询node_exporter服务的信息：

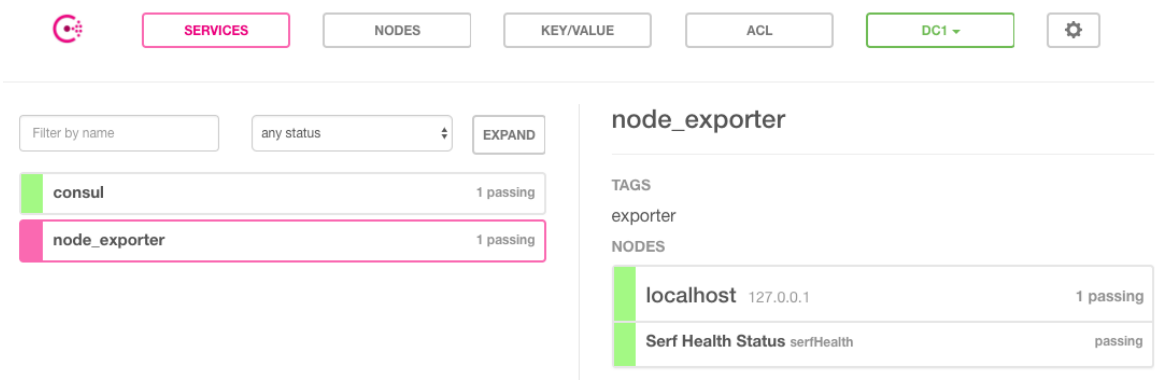
```
$ dig @127.0.0.1 -p 8600 node_exporter.service.consul

;; QUESTION SECTION:
;node_exporter.service.consul.    IN    A

;; ANSWER SECTION:
node_exporter.service.consul. 0    IN    A    127.0.0.1
```

如上所示DNS记录会返回当前可用的node_exporter服务实例的IP地址信息。

除了使用DNS的方式以外，Consul还支持用户使用HTTP API的形式获取服务列表：

```
$ curl http://localhost:8500/v1/catalog/service/node_exporter
[
    {
        "ID": "e561b376-2c1b-653d-61a0-1d844bce06a7",
        "Node": "localhost",
        "Address": "127.0.0.1",
        "Datacenter": "dc1",
        "TaggedAddresses": {
            "lan": "127.0.0.1",
            "wan": "127.0.0.1"
        },
        "NodeMeta": {
            "consul-network-segment": ""
        },
        "ServiceID": "node_exporter",
        "ServiceName": "node_exporter",
        "ServiceTags": [
            "exporter"
        ],
        "ServiceAddress": "",
        "ServiceMeta": {},
        "ServicePort": 9100,
        "ServiceEnableTagOverride": false,
        "CreateIndex": 6,
        "ModifyIndex": 6
    }
]
```

Consul也提供了一个Web UI可以查看Consul中所有服务以及节点的状态：



当然Consul还提供了更多的API用于支持对服务的生命周期管理（添加、删除、修改等）这里就不做过多的介绍，感兴趣的同学可以通过Consul官方文档了解更多的详细信息。

与Prometheus集成

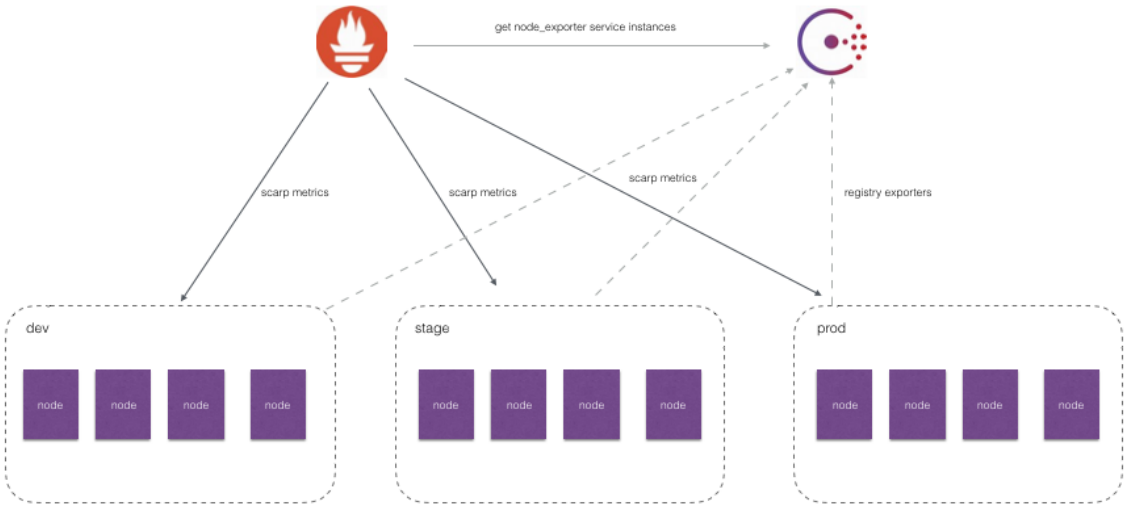
Consul作为一个通用的服务发现和注册中心，记录并且管理了环境中所有服务的信息。Prometheus通过与Consul的交互可以获取到相应Exporter实例的访问信息。在Prometheus的配置文件当可以通过以下方式与Consul进行集成：

```
- job_name: node_exporter
    metrics_path: /metrics
    scheme: http
    consul_sd_configs:
      - server: localhost:8500
        services:
          - node_exporter
```

在consul_sd_configs定义当中通过server定义了Consul服务的访问地址，services则定义了当前需要发现哪些类型服务实例的信息，这里限定了只获取node_exporter的服务实例信息。

服务发现与Relabel

在本章的前几个小节中笔者已经分别介绍了Prometheus的几种服务发现机制。通过服务发现的方式，管理员可以在不重启Prometheus服务的情况下动态的发现需要监控的Target实例信息。



如上图所示，对于线上环境我们可能会划分为:dev, stage, prod不同的集群。每一个集群运行多个主机节点，每个服务器节点上运行一个Node Exporter实例。Node Exporter实例会自动注册到Consul中，而Prometheus则根据Consul返回的Node Exporter实例信息动态的维护Target列表，从而向这些Target轮询监控数据。

然而，如果我们可能还需要：

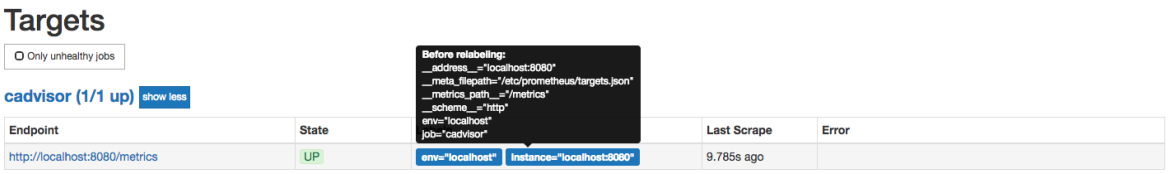
- 按照不同的环境dev, stage, prod聚合监控数据？
- 对于研发团队而言，我可能只关心dev环境的监控数据，如何处理？
- 如果为每一个团队单独搭建一个Prometheus Server。那么如何让不同团队的Prometheus Server采集不同的环境监控数据？

面对以上这些场景下的需求时，我们实际上是希望Prometheus Server能够按照某些规则（比如标签）从服务发现注册中心返回的Target实例中有选择性的采集某些Exporter实例的监控数据。

接下来，我们将学习如何通过Prometheus强大的Relabel机制来实现以上这些具体的目标。

Prometheus的Relabeling机制

在Prometheus所有的Target实例中，都包含一些默认的Metadata标签信息。可以通过Prometheus UI的Targets页面中查看这些实例的Metadata标签的内容：



默认情况下，当Prometheus加载Target实例完成后，这些Target时候都会包含一些默认的标签：

- `__address__`：当前Target实例的访问地址`<host>:<port>`

- `__scheme__`：采集目标服务访问地址的HTTP Scheme，HTTP或者HTTPS
- `__metrics_path__`：采集目标服务访问地址的访问路径
- `__param_<name>`：采集任务目标服务的中包含的请求参数

上面这些标签将会告诉Prometheus如何从该Target实例中获取监控数据。除了这些默认的标签以外，我们还可以为Target添加自定义的标签，例如，在“基于文件的服务发现”小节中的示例中，我们通过JSON配置文件，为Target实例添加了自定义标签env，如下所示该标签最终也会保存到从该实例采集的样本数据中：

```
node_cpu{cpu="cpu0",env="prod",instance="localhost:9100",job="node",mode="idle"}
```

一般来说，Target以 `__` 作为前置的标签是在系统内部使用的，因此这些标签不会被写入到样本数据中。不过这里有一些例外，例如，我们会发现所有通过Prometheus采集的样本数据中都会包含一个名为instance的标签，该标签的内容对应到Target实例的 `__address__` 。 这里实际上是发生了一次标签的重写处理。

这种发生在采集样本数据之前，对Target实例的标签进行重写的机制在Prometheus被称为Relabeling。

Prometheus允许用户在采集任务设置中通过relabel_configs来添加自定义的Relabeling过程。

使用replace/labelmap重写标签

Relabeling最基本的应用场景就是基于Target实例中包含的metadata标签，动态的添加或者覆盖标签。例如，通过Consul动态发现的服务实例还会包含以下Metadata标签信息：

- *__meta*consul_address：consul地址
- *__meta*consul_dc：consul中服务所在的数据中心
- *__meta*consulmetadata：服务的metadata
- *__meta*consul_node：服务所在consul节点的信息
- *__meta*consul_service_address：服务访问地址
- *__meta*consul_service_id：服务ID
- *__meta*consul_service_port：服务端口
- *__meta*consul_service：服务名称
- *__meta*consul_tags：服务包含的标签信息

在默认情况下，从Node Exporter实例采集上来的样本数据如下所示：

```
node_cpu{cpu="cpu0",instance="localhost:9100",job="node",mode="idle"} 93970.8203125
```

我们希望能有一个额外的标签dc可以表示该样本所属的数据中心：

```
node_cpu{cpu="cpu0",instance="localhost:9100",job="node",mode="idle", dc="dc1"} 93970.8203125
```

在每一个采集任务的配置中可以添加多个relabel_config配置，一个最简单的relabel配置如下：

```
scrape_configs:
  - job_name: node_exporter
```

```
    consul_sd_configs:
      - server: localhost:8500
        services:
          - node_exporter
    relabel_configs:
    - source_labels:  ["__meta_consul_dc"]
      target_label: "dc"
```

该采集任务通过Consul动态发现Node Exporter实例信息作为监控采集目标。在上一小节中，我们知道通过Consul动态发现的监控Target都会包含一些额外的Metadata标签，比如标签_*meta*consul_dc表明了当前实例所在的Consul数据中心，因此我们希望从这些实例中采集到的监控样本中也可以包含这样一个标签，例如：

```
node_cpu{cpu="cpu0",dc="dc1",instance="172.21.0.6:9100",job="consul_sd",mode="guest"}
```

这样可以方便的根据dc标签的值，根据不同的数据中心聚合分析各自的数据。

在这个例子中，通过从Target实例中获取_*meta*consul_dc的值，并且重写所有从该实例获取的样本中。

完整的relabel_config配置如下所示：

```
# The source labels select values from existing labels. Their content is concatenated
# using the configured separator and matched against the configured regular expression
# for the replace, keep, and drop actions.
[ source_labels: '[' <labelname> [, ...] ']' ]

# Separator placed between concatenated source label values.
[ separator: <string> | default = ; ]

# Label to which the resulting value is written in a replace action.
# It is mandatory for replace actions. Regex capture groups are available.
[ target_label: <labelname> ]

# Regular expression against which the extracted value is matched.
[ regex: <regex> | default = (.*) ]

# Modulus to take of the hash of the source label values.
[ modulus: <uint64> ]

# Replacement value against which a regex replace is performed if the
# regular expression matches. Regex capture groups are available.
[ replacement: <string> | default = $1 ]

# Action to perform based on regex matching.
[ action: <relabel_action> | default = replace ]
```

其中action定义了当前relabel_config对Metadata标签的处理方式，默认的action行为为replace。 replace行为会根据regex的配置匹配source_labels标签的值（多个source_label的值会按照separator进行拼接），并且将匹配到的值写入到target_label当中，如果有多个匹配组，则可以使用${1}, ${2}确定写入的内容。如果没匹配到任何内容则不对target_label进行重新。

repalce操作允许用户根据Target的Metadata标签重写或者写入新的标签键值对，在多环境的场景下，可以帮助用户添加与环境相关的特征维度，从而可以更好的对数据进行聚合。

除了使用replace以外，还可以定义action的配置为labelmap。与replace不同的是，labelmap会根据regex的定义去匹配Target实例所有标签的名称，并且以匹配到的内容为新的标签名称，其值作为新标签的值。

例如，在监控Kubernetes下所有的主机节点时，为将这些节点上定义的标签写入到样本中时，可以使用如下relabel_config配置：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
  - role: node
  relabel_configs:
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
```

而使用labelkeep或者labeldrop则可以对Target标签进行过滤，仅保留符合过滤条件的标签，例如：

```
relabel_configs:
  - regex: label_should_drop_(.+)
    action: labeldrop
```

该配置会使用regex匹配当前Target实例的所有标签，并将符合regex规则的标签从Target实例中移除。labelkeep正好相反，会移除那些不匹配regex定义的所有标签。

使用keep/drop过滤Target实例

在上一部分中我们介绍了Prometheus的Relabeling机制，并且使用了replace/labelmap/labelkeep/labeldrop对标签进行管理。而本节开头还提到过第二个问题，使用中心化的服务发现注册中心时，所有环境的Exporter实例都会注册到该服务发现注册中心中。而不同职能（开发、测试、运维）的人员可能只关心其中一部分的监控数据，他们可能各自部署的自己的Prometheus Server用于监控自己关心的指标数据，如果让这些Prometheus Server采集所有环境中的所有Exporter数据显然会存在大量的资源浪费。如何让这些不同的Prometheus Server采集各自关心的内容？答案还是Relabeling，relabel_config的action除了默认的replace以外，还支持keep/drop行为。例如，如果我们只希望采集数据中心dc1中的Node Exporter实例的样本数据，那么可以使用如下配置：

```
scrape_configs:
  - job_name: node_exporter
    consul_sd_configs:
      - server: localhost:8500
        services:
          - node_exporter
    relabel_configs:
    - source_labels:  ["__meta_consul_dc"]
      regex: "dc1"
      action: keep
```

当action设置为keep时，Prometheus会丢弃source_labels的值中没有匹配到regex正则表达式内容的Target实例，而当action设置为drop时，则会丢弃那些source_labels的值匹配到regex正则表达式内容的Target实例。可以简单理解为keep用于选择，而drop用于排除。

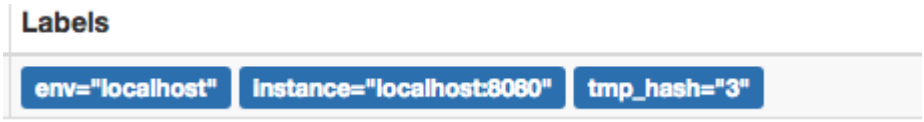
使用hashmod计算source_labels的Hash值

当relabel_config设置为hashmod时，Prometheus会根据modulus的值作为系数，计算source_labels值的hash值。例如：

```
scrape_configs
- job_name: 'file_ds'
  relabel_configs:
    - source_labels: [__address__]
      modulus:       4
      target_label:  tmp_hash
      action:        hashmod
  file_sd_configs:
```

```
  - files:
    - targets.json
```

根据当前Target实例 `__address__` 的值以4作为系数，这样每个Target实例都会包含一个新的标签tmp_hash，并且该值的范围在1~4之间，查看Target实例的标签信息，可以看到如下的结果，每一个Target实例都包含了一个新的tmp_hash值：



在第6章的“Prometheus高可用”小节中，正是利用了Hashmod的能力在Target实例级别实现对采集任务的功能分区的:

```
scrape_configs:
  - job_name: some_job
    relabel_configs:
    - source_labels: [__address__]
      modulus:       4
      target_label:  __tmp_hash
      action:        hashmod
    - source_labels: [__tmp_hash]
      regex:         ^1$
      action:        keep
```

这里需要注意的是，如果relabel的操作只是为了产生一个临时变量，以作为下一个relabel操作的输入，那么我们可以使用 `__tmp` 作为标签名的前缀，通过该前缀定义的标签就不会写入到Target或者采集到的样本的标签中。

小结

相比于直接使用静态配置，在云环境以及容器环境下我们更多的监控对象都是动态的。通过服务发现，使得Prometheus相比于其他传统监控解决方案更适用于云以及容器环境下的监控需求。

监控Kubernetes

Kubenetes是一款由Google开发的开源的容器编排工具，在Google已经使用超过15年。作为容器领域事实的标准，Kubernetes可以极大的简化应用的管理和部署复杂度。本章中，我们将介绍Kubernetes的一些基本概念，并且从0开始利用Prometheus构建一个完整的Kubernetes集群监控系统。同时我们还将学习如何通过Prometheus Operator简化在Kubernetes下部署和管理Promethues的过程。

本章的主要内容：

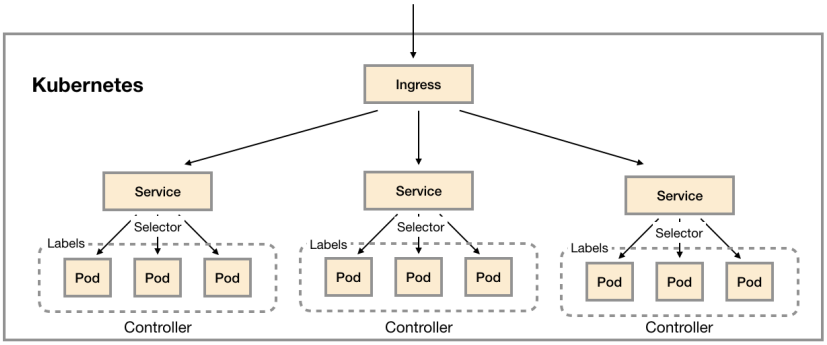
- 理解Kubernetes的工作机制
- Prometheus在Kubernetes下的服务发现机制
- 监控Kubernetes集群状态
- 监控集群基础设施基础设施
- 监控集群应用容器资源使用情况
- 监控用户部署的应用程序
- 对Service和Ingress进行网络探测
- 通过Operator高效管理和部署在Kubernetes集群中的Prometheus

初识Kubernetes

Kubenetes是一款由Google开发的开源的容器编排工具（GitHub源码），在Google已经使用超过15年（Kubernetest前身是Google的内部工具Borg）。Kubernetes将一系列的主机看做是一个受管理的海量资源，这些海量资源组成了一个能够方便进行扩展的操作系统。而在Kubernetes中运行着的容器则可以视为是这个操作系统中运行的“进程”，通过Kubernetes这一中央协调器，解决了基于容器应用程序的调度、伸缩、访问负载均衡以及整个系统的管理和监控的问题。

Kubernetes应用管理模型

下图展示了Kubernetes的应用管理模型：



Pod是Kubernetes中的最小调度资源。Pod中会包含一组容器，它们一起工作，并且对外提供一个（或者一组）功能。对于这组容器而言它们共享相同的网络和存储资源，因此它们之间可以直接通过本地网络（127.0.0.1）进行访问。当Pod被创建时，调度器（kube-schedule）会从集群中找到满足条件的节点运行它。

如果部署应用程序时，需要启动多个实例（副本），则需要使用到控制器（Controller）。用户可以在Controller定义Pod的调度规则、运行的副本数量以及升级策略等等信息，当某些Pod发生故障之后，Controller会尝试自动修复，直到Pod的运行状态满足Controller中定义的预期状态为止。Kubernetes中提供了多种Controller的实现，包括：Deployment（无状态应用）、StatefulSet（有状态应用）、Daemonset（守护模式）等，以支持不同类型应用的部署和调度模式。

通过Controller和Pod我们定义了应用程序是如何运行的，接下来需要解决如何使用这些部署在Kubernetes集群中的应用。Kubernetes将这一问题划分为两个问题域，第一，集群内的应用如何通信。第二，外部的用户如何访问部署在集群内的应用？

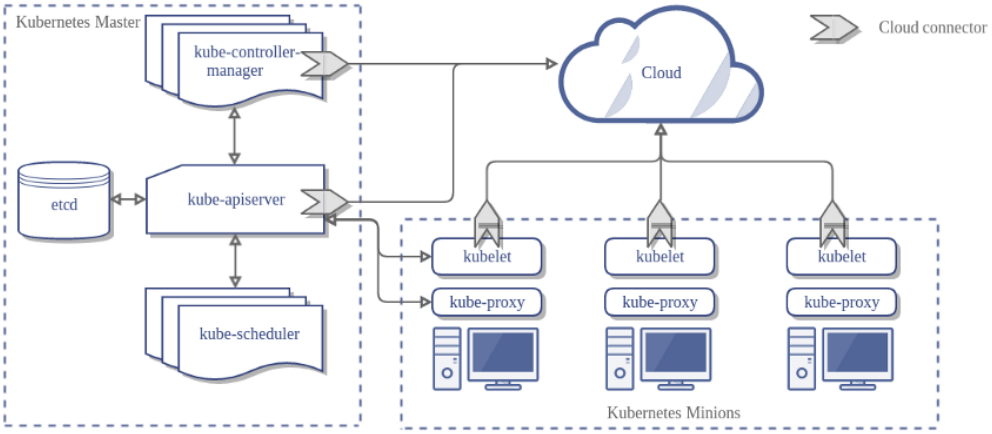
对于第一个问题，在Kubernetes中通过定义Service（服务）来解决。Service在Kubernetes集群内扮演了服务发现和负载均衡的作用。在Kubernetes下部署的Pod实例都会包含一组描述自身信息的Lable，而创建Service，可以声明一个Selector（标签选择器）。Service通过Selector，找到匹配标签规则的Pod实例，并将对Service的请求转发到代理的Pod中。Service创建完成后，集群内的应用就可以通过使用Service的名称作为DNS域名进行相互访问。

而对于第二个问题，Kubernetes中定义了单独的资源Ingress（入口）。Ingress是一个工作在7层的负载均衡器，其负责代理外部进入集群内的请求，并将流量转发到对应的服务中。

最后，对于同一个Kubernetes集群其可能被多个组织使用，为了隔离这些不同组织创建的应用程序，Kubernetes定义了Namespace（命名空间）对资源进行隔离。

Kubernetes架构模型

为了能够更好的理解Kubernetes下的监控体系，我们需要了解Kubernetes的基本架构，如下所示，是Kubernetes的架构示意图：



Kubernetes的核心组件主要由两部分组成：Master组件和Node组件，其中Matser组件提供了集群层面的管理功能，它们负责响应用户请求并且对集群资源进行统一的调度和管理。Node组件会运行在集群的所有节点上，它们负责管理和维护节点中运行的Pod，为Kubernetes集群提供运行时环境。

Master组件主要包括：

- kube-apiserver：负责对外暴露Kubernetes API；
- etcd：用于存储Kubernetes集群的所有数据；
- kube-scheduler: 负责为新创建的Pod选择可供其运行的节点；
- kube-controller-manager： 包含Node Controller，Deployment Controller，Endpoint Controller等等，通过与apiserver交互使相应的资源达到预期状态。

Node组件主要包括：

- kubelet：负责维护和管理节点上Pod的运行状态；
- kube-proxy：负责维护主机上的网络规则以及转发。
- Container Runtime：如Docker,rkt,runc等提供容器运行时环境。

Kubernetes监控策略

Kubernetes作为开源的容器编排工具，为用户提供了一个可以统一调度，统一管理的云操作系统。其解决如用户应用程序如何运行的问题。而一旦在生产环境中大量基于Kubernetes部署和管理应用程序后，作为系统管理员，还需要充分了解应用程序以及Kubernetes集群服务运行质量如何，通过对应用以及集群运行状态数据的收集和分析，持续优化和改进，从而提供一个安全可靠的生产运行环境。 这一小节中我们将讨论当使用Kubernetes时的监控策略该如何设计。

从物理结构上讲Kubernetes主要用于整合和管理底层的基础设施资源，对外提供应用容器的自动化部署和管理能力，这些基础设施可能是物理机、虚拟机、云主机等等。因此，基础资源的使用直接影响当前集群的容量和应用的状态。在这部分，我们需要关注集群中各个节点的主机负载，CPU使用率、内存使用率、存储空间以及网络吞吐等监控指标。

从自身架构上讲，kube-apiserver是Kubernetes提供所有服务的入口，无论是外部的客户端还是集群内部的组件都直接与kube-apiserver进行通讯。因此，kube-apiserver的并发和吞吐量直接决定了集群性能的好坏。其次，对于外部用户而言，Kubernetes是否能够快速的完成pod的调度以及启动，是影响其使用体验的关键因素。而这个过程主要由kube-scheduler负责完成调度工作，而kubelet完成pod的创建和启动工作。因此在Kubernetes集群本身我们需要评价其自身的服务质量，主要关注在Kubernetes的API响应时间，以及Pod的启动时间等指标上。

Kubernetes的最终目标还是需要为业务服务，因此我们还需要能够监控应用容器的资源使用情况。对于内置了对Prometheus支持的应用程序，也要支持从这些应用程序中采集内部的监控指标。最后，结合黑盒监控模式，对集群中部署的服务进行探测，从而当应用发生故障后，能够快速处理和恢复。

综上所述，我们需要综合使用白盒监控和黑盒监控模式，建立从基础设施，Kubernetes核心组件，应用容器等全面的监控体系。

在白盒监控层面我们需要关注：

- 基础设施层（Node）：为整个集群和应用提供运行时资源，需要通过各节点的kubelet获取节点的基本状态，同时通过在节点上部署Node Exporter获取节点的资源使用情况；
- 容器基础设施（Container）：为应用提供运行时环境，Kubelet内置了对cAdvisor的支持，用户可以直接通过Kubelet组件获取给节点上容器相关监控指标；
- 用户应用（Pod）：Pod中会包含一组容器，它们一起工作，并且对外提供一个（或者一组）功能。如果用户部署的应用程序内置了对Prometheus的支持，那么我们还应该采集这些Pod暴露的监控指标；
- Kubernetes组件：获取并监控Kubernetes核心组件的运行状态，确保平台自身的稳定运行。

而在黑盒监控层面，则主要需要关注以下：

- 内部服务负载均衡（Service）：在集群内，通过Service在集群暴露应用功能，集群内应用和应用之间访问时提供内部的负载均衡。通过Blackbox Exporter探测Service的可用性，确保当Service不可用时能够快速得到告警通知；
- 外部访问入口（Ingress）：通过Ingress提供集群外的访问入口，从而可以使外部客户端能够访问到部署在Kubernetes集群内的服务。因此也需要通过Blackbox Exporter对Ingress的可用性进行探测，确保外部用户能够正常访问集群内的功能；

搭建本地Kubernetes集群

为了能够更直观的了解和使用Kubernetes，我们将在本地通过工具Minikube(https://github.com/kubernetes/minikube)搭建一个本地的Kubernetes测试环境。Minikube会在本地通过虚拟机运行一个单节点的Kubernetes集群，可以方便用户或者开发人员在本地进行与Kubernetes相关的开发和测试工作。

安装MiniKube的方式很简单，对于Mac用户可以直接使用Brew进行安装:

```
brew cask install minikube
```

其它操作系统用户，可以查看Minikube项目的官方说明文档进行安装即可。安装完成后，在本机通过命令行启动Kubernetes集群:

```
$ minikube start
Starting local Kubernetes v1.7.5 cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

MiniKube会自动配置本机的kubelet命令行工具，用于与对集群资源进行管理。同时Kubernetes也提供了一个Dashboard管理界面，在MiniKube下可以通过以下命令打开：

```
$ minikube dashboard
Opening kubernetes dashboard in default browser...
```

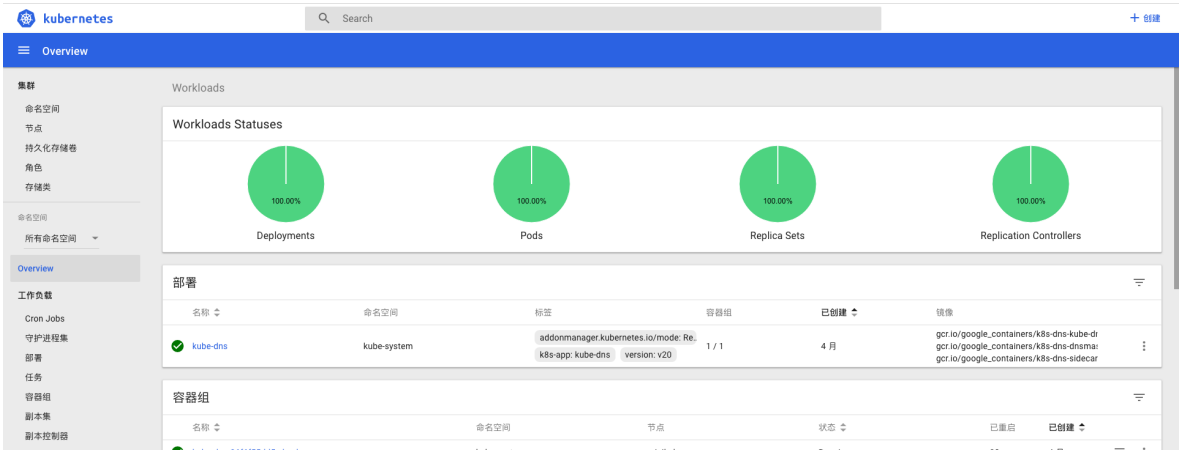
Kubernetes中的Dashboard本身也是通过Deployment进行部署的，因此可以通过MiniKube找到当前集群虚拟机的IP地址：

```
$ minikube ip
192.168.99.100
```

通过kubectl命令行工具，找到Dashboard对应的Service对外暴露的端口，如下所示，kubernetes-dashboard是一个NodePort类型的Service，并对外暴露了30000端口：

```
$ kubectl get service --namespace=kube-system
NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kube-dns               ClusterIP   10.96.0.10       <none>        53/UDP,53/TCP    131d
kubernetes-dashboard   NodePort    10.105.168.160   <none>        80:30000/TCP     131d
```

在Dashbord中，用户可以可视化的管理当前集群中运行的所有资源，以及监视其资源运行状态。



Kubernetes环境准备完成后，就可以开始尝试在Kubernetes下尝试部署一个应用程序。Kubernetes中管理的所有资源都可以通过YAML文件进行描述。如下所示，创建了一个名为nginx-deploymeht.yml文件：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

在该YAML文件中，我们定义了需要创建的资源类型为Deployment，在metadata中声明了该Deployment的名称以及标签。spec中则定义了该Deployment的具体设置，通过replicas定义了该Deployment创建后将会自动创建3个Pod实例。运行的Pod以及进行则通过template进行定义。

在命令行中使用，如下命令：

```
$ kubectl create -f nginx-deploymeht.yml
deployment "nginx-deployment" created
```

在未指定命名空间的情况下，kubectl默认关联default命名空间。由于这里没有指定Namespace，该Deployment将会在默认的命令空间default中创建。 通过kubectl get命令查看当前Deployment的部署进度：

```
# 查看Deployment的运行状态
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           1m

# 查看运行的Pod实例
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-6d8f46cfb7-5f9qm   1/1       Running   0          1m
nginx-deployment-6d8f46cfb7-9ppb8   1/1       Running   0          1m
nginx-deployment-6d8f46cfb7-nfmsw   1/1       Running   0          1m
```

为了能够让用户或者其它服务能够访问到Nginx实例，这里通过一个名为nginx-service.yml的文件定义Service资源：

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: NodePort
```

默认情况下，Service资源只能通过集群网络进行访问(type=ClusterIP)。这里为了能够直接访问该Service，需要将容器端口映射到主机上，因此定义该Service类型为NodePort。

创建并查看Service资源：

```
$ kubectl create -f nginx-service.yml
service "nginx-service" created

$ kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        131d
nginx-service   NodePort    10.104.103.112   <none>        80:32022/TCP   10s
```

通过nginx-server映射到虚拟机的32022端口，就可以直接访问到Nginx实例的80端口：

部署完成后，如果需要对Nginx实例进行扩展，可以使用：

```
$ kubectl scale deployments/nginx-deployment --replicas=4
deployment "nginx-deployment" scaled
```

通过kubectl命令还可以对镜像进行滚动升级：

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated

$ kubectl get pods
NAME                                READY     STATUS              RESTARTS   AGE
nginx-deployment-58b94fcb9-8fjm6    0/1       ContainerCreating   0          52s
nginx-deployment-58b94fcb9-qzlwx    0/1       ContainerCreating   0          51s
nginx-deployment-6d8f46cfb7-5f9qm   1/1       Running             0          45m
nginx-deployment-6d8f46cfb7-7xs6z   0/1       Terminating         0          2m
nginx-deployment-6d8f46cfb7-9ppb8   1/1       Running             0          45m
nginx-deployment-6d8f46cfb7-nfmsw   1/1       Running             0          45m
```

如果升级后服务出现异常，那么可以通过以下命令对应用进行回滚：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment"
```

Kubernetes依托于Google丰富的大规模应用管理经验。通过将集群环境抽象为一个统一调度和管理的云”操作系统，视容器为这个操作中独自运行的“进程”，进程间的隔离通过命名空间（Namespace）完成，实现了对应用生命周期管理从自动化到自主化的跨越。

部署Prometheus

在上一小节总我们介绍了与Kubernetes的应用管理模型，并且利用MiniKube在本地搭建了一个单节点的Kubernetes。这一部分我们将带领读者通过Kubernetes部署Prometheus实例。

使用ConfigMaps管理应用配置

当使用Deployment管理和部署应用程序时，用户可以方便了对应用进行扩容或者缩容，从而产生多个Pod实例。为了能够统一管理这些Pod的配置信息，在Kubernetes中可以使用ConfigMaps资源定义和管理这些配置，并且通过环境变量或者文件系统挂载的方式让容器使用这些配置。

这里将使用ConfigMaps管理Prometheus的配置文件，创建prometheus-config.yml文件，并写入以下内容：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval:     15s
      evaluation_interval: 15s
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
        - targets: ['localhost:9090']
```

使用kubectl命令行工具，在命名空间default创建ConfigMap资源：

```
kubectl create -f prometheus-config.yml
configmap "prometheus-config" created
```

使用Deployment部署Prometheus

当ConfigMap资源创建成功后，我们就可以通过Volume挂载的方式，将Prometheus的配置文件挂载到容器中。 这里我们通过Deployment部署Prometheus Server实例，创建prometheus-deployment.yml文件，并写入以下内容:

```
apiVersion: v1
kind: "Service"
metadata:
  name: prometheus
  labels:
    name: prometheus
spec:
  ports:
  - name: prometheus
    protocol: TCP
    port: 9090
    targetPort: 9090
  selector:
    app: prometheus
  type: NodePort
---
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
  labels:
    name: prometheus
  name: prometheus
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
      - name: prometheus
        image: prom/prometheus:v2.2.1
        command:
        - "/bin/prometheus"
        args:
        - "--config.file=/etc/prometheus/prometheus.yml"
        ports:
        - containerPort: 9090
          protocol: TCP
        volumeMounts:
        - mountPath: "/etc/prometheus"
          name: prometheus-config
      volumes:
      - name: prometheus-config
        configMap:
          name: prometheus-config
```

该文件中分别定义了Service和Deployment，Service类型为NodePort，这样我们可以通过虚拟机IP和端口访问到Prometheus实例。为了能够让Prometheus实例使用ConfigMap中管理的配置文件，这里通过volumes声明了一个磁盘卷。并且通过volumeMounts将该磁盘卷挂载到了Prometheus实例的/etc/prometheus目录下。

使用以下命令创建资源，并查看资源的创建情况：

```
$ kubectl create -f prometheus-deployment.yml
service "prometheus" created
deployment "prometheus" created

$ kubectl get pods
NAME                                 READY     STATUS              RESTARTS   AGE
prometheus-55f655696d-wjqcl          1/1       Running             0          5s

$ kubectl get svc
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes       ClusterIP   10.96.0.1        <none>        443/TCP          131d
prometheus       NodePort    10.101.255.236   <none>        9090:32584/TCP   42s
```

至此，我们可以通过MiniKube虚拟机的IP地址和端口32584访问到Prometheus的服务。

Kubernetes下的服务发现

目前为止，我们已经能够在Kubernetes下部署一个简单的Prometheus实例，不过当前来说它并不能发挥其监控系统的作用，除了Prometheus，暂时没有任何的监控采集目标。在第7章中，我们介绍了Prometheus的服务发现能力，它能够与通过与“中间代理人“的交互，从而动态的获取需要监控的目标实例。而在Kubernetes下Prometheus就是需要与Kubernetes的API进行交互，从而能够动态的发现Kubernetes中部署的所有可监控的目标资源。

Kubernetes的访问授权

为了能够让Prometheus能够访问收到认证保护的Kubernetes API，我们首先需要做的是，对Prometheus进行访问授权。在Kubernetes中主要使用基于角色的访问控制模型(Role-Based Access Control)，用于管理Kubernetes下资源访问权限。首先我们需要在Kubernetes下定义角色（ClusterRole），并且为该角色赋予相应的访问权限。同时创建Prometheus所使用的账号（ServiceAccount），最后则是将该账号与角色进行绑定（ClusterRoleBinding）。这些所有的操作在Kubernetes同样被视为是一系列的资源，可以通过YAML文件进行描述并创建，这里创建prometheus-rbac-setup.yml文件，并写入以下内容：

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: default
```

其中需要注意的是ClusterRole是全局的，不需要指定命名空间。而ServiceAccount是属于特定命名空间的资源。通过kubectl命令创建RBAC对应的各个资源：

```
$ kubectl create -f prometheus-rbac-setup.yml
clusterrole "prometheus" created
serviceaccount "prometheus" created
clusterrolebinding "prometheus" created
```

在完成角色权限以及用户的绑定之后，就可以指定Prometheus使用特定的ServiceAccount创建Pod实例。修改prometheus-deployment.yml文件，并添加serviceAccountName和serviceAccount定义：

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      serviceAccount: prometheus
```

通过kubectl apply对Deployment进行变更升级：

```
$ kubectl apply -f prometheus-deployment.yml
service "prometheus" configured
deployment "prometheus" configured

$ kubectl get pods
NAME                                READY     STATUS        RESTARTS   AGE
prometheus-55f655696d-wjqcl         0/1       Terminating   0          38m
prometheus-69f9ddb588-czn2c         1/1       Running       0          6s
```

指定ServiceAccount创建的Pod实例中，会自动将用于访问Kubernetes API的CA证书以及当前账户对应的访问令牌文件挂载到Pod实例的/var/run/secrets/kubernetes.io/serviceaccount/目录下，可以通过以下命令进行查看：

```
kubectl exec -it prometheus-69f9ddb588-czn2c ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt     namespace  token
```

服务发现

在Kubernetes下，Promethues通过与Kubernetes API集成目前主要支持5种服务发现模式，分别是：Node、Service、Pod、Endpoints、Ingress。

通过kubectl命令行，可以方便的获取到当前集群中的所有节点信息：

```
$ kubectl get nodes -o wide
NAME       STATUS    ROLES     AGE       VERSION   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAIN
ER-RUNTIME
minikube   Ready     <none>    164d      v1.8.0    <none>        Buildroot 2017.02   4.9.13           docker:
//Unknown
```

为了能够让Prometheus能够获取到当前集群中所有节点的信息，在Promtheus的配置文件中，我们添加如下Job配置：

```
- job_name: 'kubernetes-nodes'
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: node
```

通过指定kubernetes_sd_config的模式为node，Prometheus会自动从Kubernetes中发现到所有的node节点并作为当前Job监控的Target实例。如下所示，这里需要指定用于访问Kubernetes API的ca以及token文件路径。

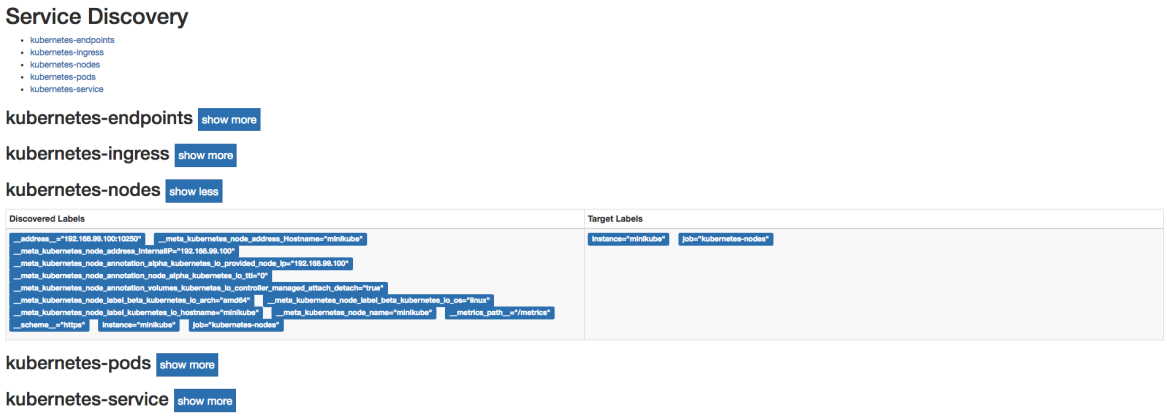
对于Ingress，Service，Endpoints, Pod的使用方式也是类似的，下面给出了一个完整Prometheus配置的示例：

```
apiVersion: v1
data:
  prometheus.yml: |-
    global:
      scrape_interval:     15s
      evaluation_interval: 15s
    scrape_configs:

    - job_name: 'kubernetes-nodes'
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: node

    - job_name: 'kubernetes-service'
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: service

    - job_name: 'kubernetes-endpoints'
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: endpoints

    - job_name: 'kubernetes-ingress'
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: ingress

    - job_name: 'kubernetes-pods'
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: pod

kind: ConfigMap
metadata:
  name: prometheus-config
```

更新Prometheus配置文件，并重建Prometheus实例：

```
$ kubectl apply -f prometheus-config.yml
configmap "prometheus-config" configured

$ kubectl get pods
prometheus-69f9ddb588-rbrs2        1/1       Running   0          4m

$ kubectl delete pods prometheus-69f9ddb588-rbrs2
pod "prometheus-69f9ddb588-rbrs2" deleted

$ kubectl get pods
prometheus-69f9ddb588-rbrs2        0/1       Terminating   0          4m
prometheus-69f9ddb588-wtlsn        1/1       Running       0          14s
```

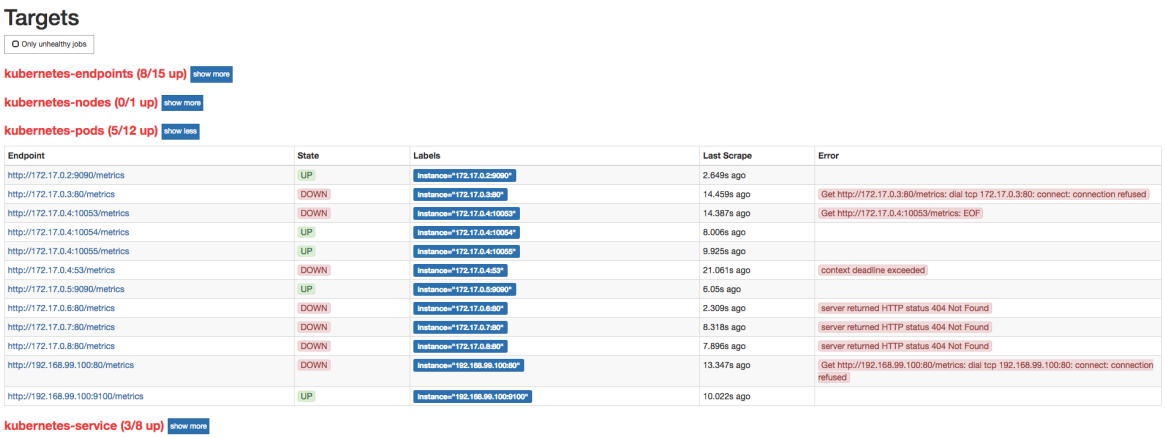
Prometheus使用新的配置文件重建之后，打开Prometheus UI，通过Service Discovery页面可以查看到当前Prometheus通过Kubernetes发现的所有资源对象了：



同时Prometheus会自动将该资源的所有信息，并通过标签的形式体现在Target对象上。如下所示，是Promthues获取到的Node节点的标签信息：

```
__address__="192.168.99.100:10250"
__meta_kubernetes_node_address_Hostname="minikube"
__meta_kubernetes_node_address_InternalIP="192.168.99.100"
__meta_kubernetes_node_annotation_alpha_kubernetes_io_provided_node_ip="192.168.99.100"
__meta_kubernetes_node_annotation_node_alpha_kubernetes_io_ttl="0"
__meta_kubernetes_node_annotation_volumes_kubernetes_io_controller_managed_attach_detach="true"
__meta_kubernetes_node_label_beta_kubernetes_io_arch="amd64"
__meta_kubernetes_node_label_beta_kubernetes_io_os="linux"
__meta_kubernetes_node_label_kubernetes_io_hostname="minikube"
__meta_kubernetes_node_name="minikube"
__metrics_path__="/metrics"
__scheme__="https"
instance="minikube"
job="kubernetes-nodes"
```

目前为止，我们已经能够通过Prometheus自动发现Kubernetes集群中的各类资源以及其基本信息。不过，如果现在查看Promtheus的Target状态页面，结果可能会让人不太满意：



Targets

Only unhealthy jobs

kubernetes-endpoints (8/15 up) show more

kubernetes-nodes (0/1 up) show more

kubernetes-pods (5/12 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://172.17.0.2:9090/metrics	UP	instance="172.17.0.2:9090"	2.649s ago	
http://172.17.0.3:80/metrics	DOWN	instance="172.17.0.3:80"	14.459s ago	Get http://172.17.0.3:80/metrics: dial tcp 172.17.0.3:80: connect: connection refused
http://172.17.0.4:10053/metrics	DOWN	instance="172.17.0.4:10053"	14.387s ago	Get http://172.17.0.4:10053/metrics: EOF
http://172.17.0.4:10054/metrics	UP	instance="172.17.0.4:10054"	8.006s ago	
http://172.17.0.4:10055/metrics	UP	instance="172.17.0.4:10055"	9.925s ago	
http://172.17.0.4:53/metrics	DOWN	instance="172.17.0.4:53"	21.061s ago	context deadline exceeded
http://172.17.0.5:9090/metrics	UP	instance="172.17.0.5:9090"	6.05s ago	
http://172.17.0.6:80/metrics	DOWN	instance="172.17.0.6:80"	2.309s ago	server returned HTTP status 404 Not Found
http://172.17.0.7:80/metrics	DOWN	instance="172.17.0.7:80"	8.318s ago	server returned HTTP status 404 Not Found
http://172.17.0.8:80/metrics	DOWN	instance="172.17.0.8:80"	7.896s ago	server returned HTTP status 404 Not Found
http://192.168.99.100:80/metrics	DOWN	instance="192.168.99.100:80"	13.347s ago	Get http://192.168.99.100:80/metrics: dial tcp 192.168.99.100:80: connect: connection refused
http://192.168.99.100:9100/metrics	UP	instance="192.168.99.100:9100"	10.022s ago	

kubernetes-service (3/8 up) show more

虽然Prometheus能够自动发现所有的资源对象，并且将其作为Target对象进行数据采集。 但并不是所有的资源对象都是支持Promethues的，并且不同类型资源对象的采集方式可能是不同的。因此，在实际的操作中，我们需要有明确的监控目标，并且针对不同类型的监控目标设置不同的数据采集方式。

接下来，我们将利用Promtheus的服务发现能力，实现对Kubernetes集群的全面监控。

监控Kubernetes集群

上一小节中，我们介绍了Promtheus在Kubernetes下的服务发现能力，并且通过kubernetes_sd_config实现了对Kubernetes下各类资源的自动发现。在本小节中，我们将带领读者利用Promethues提供的服务发现能力，实现对Kubernetes集群以及其中部署的各类资源的自动化监控。

下表中，梳理了监控Kubernetes集群监控的各个维度以及策略：

目标	服务发现模式
从集群各节点kubelet组件中获取节点kubelet的基本运行状态的监控指标	node
从集群各节点kubelet内置的cAdvisor中获取， 节点中运行的容器的监控指标	node
从部署到各个节点的Node Exporter中采集主机资源相关的运行资源	node
对于内置了Promthues支持的应用， 需要从Pod实例中采集其自定义监控指标	pod
获取API Server组件的访问地址， 并从中获取Kubernetes集群相关的运行监控指标	endpoints
获取集群中Service的访问地址，并通过Blackbox Exporter获取网络探测指标	service
获取集群中Ingress的访问信息，并通过Blackbox Exporter获取网络探测指标	ingress

从Kubelet获取节点运行状态

Kubelet组件运行在Kubernetes集群的各个节点中，其负责维护和管理节点上Pod的运行状态。kubelet组件的正常运行直接关系到该节点是否能够正常的被Kubernetes集群正常使用。

基于Node模式，Prometheus会自动发现Kubernetes中所有Node节点的信息并作为监控的目标Target。 而这些Target的访问地址实际上就是Kubelet的访问地址，并且Kubelet实际上直接内置了对Promtheus的支持。

修改prometheus.yml配置文件，并添加以下采集任务配置：

```
  - job_name: 'kubernetes-kubelet'
    scheme: https
    tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
    - role: node
    relabel_configs:
```

```
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
```

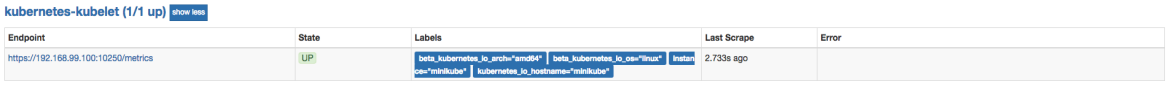
这里使用Node模式自动发现集群中所有Kubelet作为监控的数据采集目标，同时通过labelmap步骤，将Node节点上的标签，作为样本的标签保存到时间序列当中。

重新加载promethues配置文件，并重建Promthues的Pod实例后，查看kubernetes-kubelet任务采集状态，我们会看到以下错误提示信息：

```
Get https://192.168.99.100:10250/metrics: x509: cannot validate certificate for 192.168.99.100 because it doesn't contain any IP SANs
```

这是由于当前使用的ca证书中，并不包含192.168.99.100的地址信息。为了解决该问题，第一种方法是直接跳过ca证书校验过程，通过在tls_config中设置
insecure_skip_verify为true即可。 这样Promthues在采集样本数据时，将会自动跳过ca证书的校验过程，从而从kubelet采集到监控数据：

```
  - job_name: 'kubernetes-kubelet'
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      insecure_skip_verify: true
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
    - role: node
    relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
```



第二种方式，不直接通过kubelet的metrics服务采集监控数据，而通过Kubernetes的api-server提供的代理API访问各个节点中kubelet的metrics服务，如下所示：

```
  - job_name: 'kubernetes-kubelet'
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
    - role: node
    relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
    - target_label: __address__
      replacement: kubernetes.default.svc:443
    - source_labels: [__meta_kubernetes_node_name]
      regex: (.+)
      target_label: __metrics_path__
      replacement: /api/v1/nodes/${1}/proxy/metrics
```

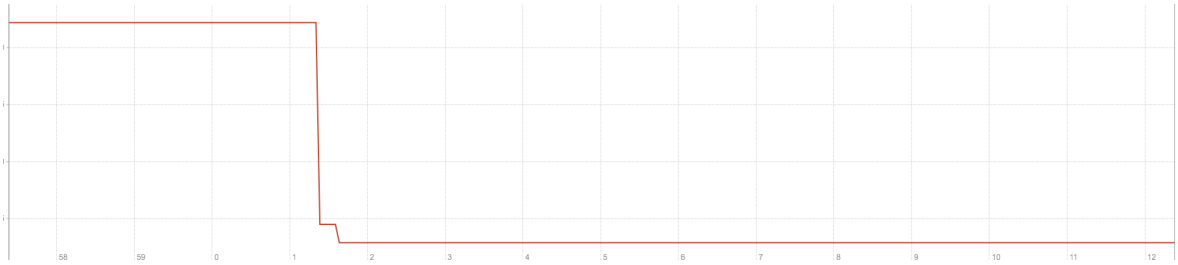
通过relabeling，将从Kubernetes获取到的默认地址 `__address__` 替换为kubernetes.default.svc:443。同时将 `__metrics_path__` 替换为api-server的代理地址/api/v1/nodes/${1}/proxy/metrics。

通过获取各个节点中kubelet的监控指标，用户可以评估集群中各节点的性能表现。例如,通过指标kubelet_pod_start_latency_microseconds可以获得当前节点中Pod启动时间相关的统计数据。

```
kubelet_pod_start_latency_microseconds{quantile="0.99"}
```

Pod平均启动时间大致为42s左右（包含镜像下载时间）：

```
kubelet_pod_start_latency_microseconds_sum / kubelet_pod_start_latency_microseconds_count
```



除此以外，监控指标kubelet_docker_*还可以体现出kubelet与当前节点的docker服务的调用情况，从而可以反映出docker本身是否会影响kubelet的性能表现等问题。

从Kubelet获取节点容器资源使用情况

各节点的kubelet组件中除了包含自身的监控指标信息以外，kubelet组件还内置了对cAdvisor的支持。cAdvisor能够获取当前节点上运行的所有容器的资源使用情况，通过访问kubelet的/metrics/cadvisor地址可以获取到cadvisor的监控指标，因此和获取kubelet监控指标类似，这里同样通过node模式自动发现所有的kubelet信息，并通过适当的relabel过程，修改监控采集任务的配置。 与采集kubelet自身监控指标相似，这里也有两种方式采集cadvisor中的监控指标：

方式一：直接访问kubelet的/metrics/cadvisor地址，需要跳过ca证书认证：

```
    - job_name: 'kubernetes-cadvisor'
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        insecure_skip_verify: true
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: node
      relabel_configs:
      - source_labels: [__meta_kubernetes_node_name]
```

```
      regex: (.+)
      target_label: __metrics_path__
      replacement: metrics/cadvisor
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
```

方式二：通过api-server提供的代理地址访问kubelet的/metrics/cadvisor地址：

```
  - job_name: 'kubernetes-cadvisor'
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
    - role: node
    relabel_configs:
    - target_label: __address__
      replacement: kubernetes.default.svc:443
    - source_labels: [__meta_kubernetes_node_name]
      regex: (.+)
      target_label: __metrics_path__
      replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
```

使用NodeExporter监控集群资源使用情况

为了能够采集集群中各个节点的资源使用情况，我们需要在各节点中部署一个Node Exporter实例。在本章的“部署Prometheus”小节，我们使用了Kubernetes内置的控制器之一Deployment。Deployment能够确保Prometheus的Pod能够按照预期的状态在集群中运行，而Pod实例可能随机运行在任意节点上。而与Prometheus的部署不同的是，对于Node Exporter而言每个节点只需要运行一个唯一的实例，此时，就需要使用Kubernetes的另外一种控制器Daemonset。顾名思义，Daemonset的管理方式类似于操作系统中的守护进程。Daemonset会确保在集群中所有（也可以指定）节点上运行一个唯一的Pod实例。

创建node-exporter-daemonset.yml文件，并写入以下内容：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-exporter
spec:
  template:
    metadata:
      annotations:
        prometheus.io/scrape: 'true'
        prometheus.io/port: '9100'
        prometheus.io/path: 'metrics'
      labels:
        app: node-exporter
      name: node-exporter
    spec:
```

```
      containers:
      - image: prom/node-exporter
        imagePullPolicy: IfNotPresent
        name: node-exporter
        ports:
        - containerPort: 9100
          hostPort: 9100
          name: scrape
      hostNetwork: true
      hostPID: true
```

由于Node Exporter需要能够访问宿主机，因此这里指定了hostNetwork和hostPID，让Pod实例能够以主机网络以及系统进程的形式运行。同时YAML文件中也创建了NodeExporter相应的Service。这样通过Service就可以访问到对应的NodeExporter实例。

```
$ kubectl create -f node-exporter-daemonset.yml
service "node-exporter" created
daemonset "node-exporter" created
```

查看Daemonset以及Pod的运行状态

```
$ kubectl get daemonsets
NAME            DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
node-exporter   1         1         1         1            1           <none>          15s

$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
...
node-exporter-9h56z                 1/1       Running   0          51s
```

由于Node Exporter是以主机网络的形式运行，因此直接访问MiniKube的虚拟机IP加上Pod的端口即可访问当前节点上运行的Node Exporter实例:

```
$ minikube ip
192.168.99.100

$ curl http://192.168.99.100:9100/metrics
...
process_start_time_seconds 1.5251401593e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.1984896e+08
```

目前为止，通过Daemonset的形式将Node Exporter部署到了集群中的各个节点中。接下来，我们只需要通过Prometheus的pod服务发现模式，找到当前集群中部署的Node Exporter实例即可。 需要注意的是，由于Kubernetes中并非所有的Pod都提供了对Prometheus的支持，有些可能只是一些简单的用户应用，为了区分哪些Pod实例是可以供Prometheus进行采集的，这里我们为Node Exporter添加了注解：

```
prometheus.io/scrape: 'true'
```

由于Kubernetes中Pod可能会包含多个容器，还需要用户通过注解指定用户提供监控指标的采集端口：

```
prometheus.io/port: '9100'
```

而有些情况下，Pod中的容器可能并没有使用默认的/metrics作为监控采集路径，因此还需要支持用户指定采集路径：

```
prometheus.io/path: 'metrics'
```

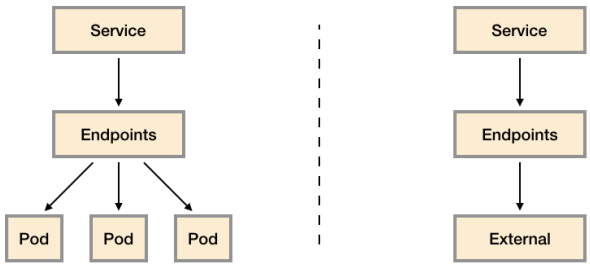
为Prometheus创建监控采集任务kubernetes-pods，如下所示：

```
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
    - role: pod
    relabel_configs:
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
      action: keep
      regex: true
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
      action: replace
      target_label: __metrics_path__
      regex: (.+)
    - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
      action: replace
      regex: ([^:]+)(?::\d+)?;(\d+)
      replacement: $1:$2
      target_label: __address__
    - action: labelmap
      regex: __meta_kubernetes_pod_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      action: replace
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_pod_name]
      action: replace
      target_label: kubernetes_pod_name
```

通过以上relabel过程实现对Pod实例的过滤，以及采集任务地址替换，从而实现对特定Pod实例监控指标的采集。需要说明的是kubernetes-pods并不是只针对Node Exporter而言，对于用户任意部署的Pod实例，只要其提供了对Prometheus的支持，用户都可以通过为Pod添加注解的形式为其添加监控指标采集的支持。

从kube-apiserver获取集群运行监控指标

在开始正式内容之前，我们需要先了解一下Kubernetes中Service是如何实现负载均衡的，如下图所示，一般来说Service有两个主要的使用场景：



- 代理对集群内部应用Pod实例的请求：当创建Service时如果指定了标签选择器，Kubernetes会监听集群中所有的Pod变化情况，通过Endpoints自动维护满足标签选择器的Pod实例的访问信息；

- 代理对集群外部服务的请求：当创建Service时如果不指定任何的标签选择器，此时需要用户手动创建Service对应的Endpoint资源。例如，一般来说，为了确保数据的安全，我们通常讲数据库服务部署到集群外。 这是为了避免集群内的应用硬编码数据库的访问信息，这是就可以通过在集群内创建Service，并指向外部的数据库服务实例。

kube-apiserver扮演了整个Kubernetes集群管理的入口的角色，负责对外暴露Kubernetes API。kube-apiserver组件一般是独立部署在集群外的，为了能够让部署在集群内的应用（kubernetes插件或者用户应用）能够与kube-apiserver交互，Kubernetes会默认在命名空间下创建一个名为kubernetes的服务，如下所示：

```
$ kubectl get svc kubernetes -o wide
NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE         SELECTOR
kubernetes            ClusterIP   10.96.0.1       <none>        443/TCP          166d        <none>
```

而该kubernetes服务代理的后端实际地址通过endpoints进行维护，如下所示：

```
$ kubectl get endpoints kubernetes
NAME         ENDPOINTS         AGE
kubernetes   10.0.2.15:8443    166d
```

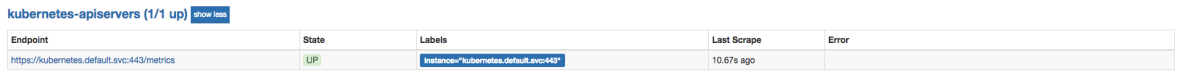
通过这种方式集群内的应用或者系统主机就可以通过集群内部的DNS域名kubernetes.default.svc访问到部署外部的kube-apiserver实例。

因此，如果我们想要监控kube-apiserver相关的指标，只需要通过endpoints资源找到kubernetes对应的所有后端地址即可。

如下所示，创建监控任务kubernetes-apiservers，这里指定了服务发现模式为endpoints。Promtheus会查找当前集群中所有的endpoints配置，并通过relabel进行判断是否为apiserver对应的访问地址：

```
    - job_name: 'kubernetes-apiservers'
      kubernetes_sd_configs:
      - role: endpoints
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name, __meta_kubernetes_endpoi
nt_port_name]
        action: keep
        regex: default;kubernetes;https
      - target_label: __address__
        replacement: kubernetes.default.svc:443
```

在relabel_configs配置中第一步用于判断当前endpoints是否为kube-apiserver对用的地址。第二步，替换监控采集地址到kubernetes.default.svc:443即可。重新加载配置文件，重建Promthues实例，得到以下结果。



对Ingress和Service进行网络探测

为了能够对Ingress和Service进行探测，我们需要在集群部署Blackbox Exporter实例。 如下所示，创建blackbox-exporter.yaml用于描述部署相关的内容：

```
apiVersion: v1
```

```
kind: Service
metadata:
  labels:
    app: blackbox-exporter
  name: blackbox-exporter
spec:
  ports:
  - name: blackbox
    port: 9115
    protocol: TCP
  selector:
    app: blackbox-exporter
  type: ClusterIP
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: blackbox-exporter
  name: blackbox-exporter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: blackbox-exporter
  template:
    metadata:
      labels:
        app: blackbox-exporter
    spec:
      containers:
      - image: prom/blackbox-exporter
        imagePullPolicy: IfNotPresent
        name: blackbox-exporter
```

通过kubectl命令部署Blackbox Exporter实例，这里将部署一个Blackbox Exporter的Pod实例，同时通过服务blackbox-exporter在集群内暴露访问地址blackbox-exporter.default.svc.cluster.local，对于集群内的任意服务都可以通过该内部DNS域名访问Blackbox Exporter实例：

```
$ kubectl get pods
NAME                                    READY     STATUS        RESTARTS    AGE
blackbox-exporter-f77fc78b6-72b15       1/1       Running       0           4s

$ kubectl get svc
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
blackbox-exporter           ClusterIP   10.109.144.192   <none>        9115/TCP         3m
```

为了能够让Prometheus能够自动的对Service进行探测，我们需要通过服务发现自动找到所有的Service信息。 如下所示，在Prometheus的配置文件中添加名为kubernetes-services的监控采集任务：

```
    - job_name: 'kubernetes-services'
      metrics_path: /probe
      params:
        module: [http_2xx]
      kubernetes_sd_configs:
      - role: service
      relabel_configs:
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_probe]
        action: keep
```

```
      regex: true
    - source_labels: [__address__]
      target_label: __param_target
    - target_label: __address__
      replacement: blackbox-exporter.default.svc.cluster.local:9115
    - source_labels: [__param_target]
      target_label: instance
    - action: labelmap
      regex: __meta_kubernetes_service_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_service_name]
      target_label: kubernetes_name
```

在该任务配置中，通过指定kubernetes_sd_config的role为service指定服务发现模式：

```
kubernetes_sd_configs:
  - role: service
```

为了区分集群中需要进行探测的Service实例，我们通过标签‘prometheus.io/probe: true’进行判断，从而过滤出需要探测的所有Service实例：

```
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_probe]
      action: keep
      regex: true
```

并且将通过服务发现获取到的Service实例地址 `__address__` 转换为获取监控数据的请求参数。同时将 `__address__` 执行Blackbox Exporter实例的访问地址，并且重写了标签instance的内容：

```
    - source_labels: [__address__]
      target_label: __param_target
    - target_label: __address__
      replacement: blackbox-exporter.default.svc.cluster.local:9115
    - source_labels: [__param_target]
      target_label: instance
```

最后，为监控样本添加了额外的标签信息：

```
    - action: labelmap
      regex: __meta_kubernetes_service_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_service_name]
      target_label: kubernetes_name
```

对于Ingress而言，也是一个相对类似的过程，这里给出对Ingress探测的Promthues任务配置作为参考：

```
  - job_name: 'kubernetes-ingresses'
    metrics_path: /probe
    params:
      module: [http_2xx]
    kubernetes_sd_configs:
    - role: ingress
    relabel_configs:
    - source_labels: [__meta_kubernetes_ingress_annotation_prometheus_io_probe]
      action: keep
```

```
      regex: true
    - source_labels: [__meta_kubernetes_ingress_scheme,__address__,__meta_kubernetes_ingress_path]
      regex: (.+);(.+);(.+)
      replacement: ${1}://${2}${3}
      target_label: __param_target
    - target_label: __address__
      replacement: blackbox-exporter.default.svc.cluster.local:9115
    - source_labels: [__param_target]
      target_label: instance
    - action: labelmap
      regex: __meta_kubernetes_ingress_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_ingress_name]
      target_label: kubernetes_name
```

基于Prometheus的弹性伸缩

弹性伸缩（AutoScaling）是指应用可以根据当前的资源使用情况自动水平扩容或者缩容的能力。

小结

Kubernetes与Promethues有着十分相似的历程，均是源自Google内部多年的运维经验。并且相继从CNCF基金会正式毕业。它们分别代表了云原生模式下容器编排以及监控的事实标准。

Prometheus Operator

本章，我们将介绍如何使用Prometheus Operator简化在Kubernetes下部署和管理Prmetheus的复杂度。

本章的主要内容：

- 为什么需要使用Prometheus Operator
- Prometheus Operator的主要概念
- 如何利用Prometheus Operator自动化运维Prometheus
- 如何使用Prometheus Operator自动化管理监控配置

什么是Prometheus Operator

在第8章章中，为了在Kubernetes能够方便的管理和部署Prometheus，我们使用ConfigMap了管理Prometheus配置文件。每次对Prometheus配置文件进行升级时，，我们需要手动移除已经运行的Pod实例，从而让Kubernetes可以使用最新的配置文件创建Prometheus。 而如果当应用实例的数量更多时，通过手动的方式部署和升级Prometheus过程繁琐并且效率低下。

从本质上来讲Prometheus属于是典型的有状态应用，而其有包含了一些自身特有的运维管理和配置管理方式。而这些都无法通过Kubernetes原生提供的应用管理概念实现自动化。为了简化这类应用程序的管理复杂度，CoreOS率先引入了Operator的概念，并且首先推出了针对在Kubernetes下运行和管理Etcd的Etcd Operator。并随后推出了Prometheus Operator。

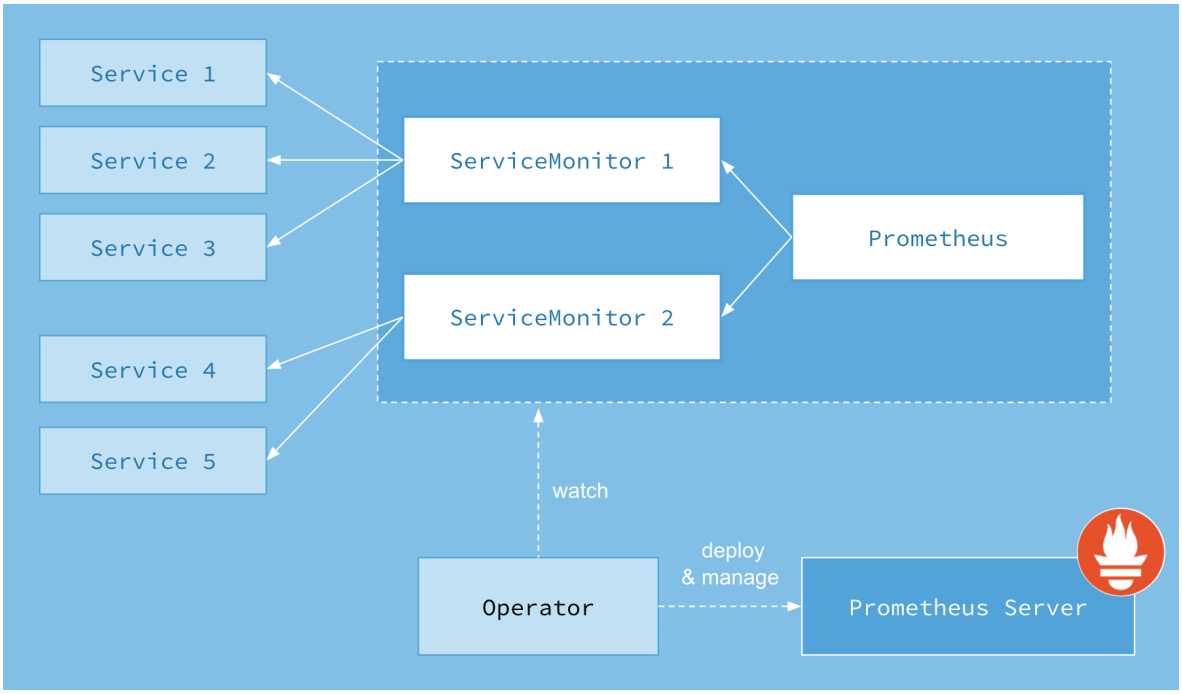
Prometheus Operator的工作原理

从概念上来讲Operator就是针对管理特定应用程序的，在Kubernetes基本的Resource和Controller的概念上，以扩展Kubernetes api的形式。帮助用户创建，配置和管理复杂的有状态应用程序。从而实现特定应用程序的常见操作以及运维自动化。

在Kubernetes中我们使用Deployment、DamenSet，StatefulSet来管理应用Workload，使用Service，Ingress来管理应用的访问方式，使用ConfigMap和Secret来管理应用配置。我们在集群中对这些资源的创建，更新，删除的动作都会被转换为事件(Event)，Kubernetes的Controller Manager负责监听这些事件并触发相应的任务来满足用户的期望。这种方式我们成为声明式，用户只需要关心应用程序的最终状态，其它的都通过Kubernetes来帮助我们完成，通过这种方式可以大大简化应用的配置管理复杂度。

而除了这些原生的Resource资源以外，Kubernetes还允许用户添加自己的自定义资源(Custom Resource)。并且通过实现自定义Controller来实现对Kubernetes的扩展。

如下所示，是Prometheus Operator的架构示意图：



Prometheus的本职就是一组用户自定义的CRD资源以及Controller的实现，Prometheus Operator负责监听这些自定义资源的变化，并且根据这些资源的定义自动化的完成如Prometheus Server自身以及配置的自动化管理工作。

Prometheus Operator能做什么

要了解Prometheus Operator能做什么，其实就是要了解Prometheus Operator为我们提供了哪些自定义的Kubernetes资源，列出了Prometheus Operator目前提供的4类资源：

- Prometheus：声明式创建和管理Prometheus Server实例；
- ServiceMonitor：负责声明式的管理监控配置；
- PrometheusRule：负责声明式的管理告警配置；
- Alertmanager：声明式的创建和管理Alertmanager实例。

简言之，Prometheus Operator能够帮助用户自动化的创建以及管理Prometheus Server以及其相应的配置。

在Kubernetes集群中部署Prometheus Operator

在Kubernetes中安装Prometheus Operator非常简单，用户可以从以下地址中过去Prometheus Operator的源码：

```
git clone https://github.com/coreos/prometheus-operator.git
```

这里，我们为Promethues Operator创建一个单独的命名空间monitoring：

```
kubectl create namespace monitoring
```

由于需要对Prometheus Operator进行RBAC授权，而默认的bundle.yaml中使用了default命名空间，因此，在安装Prometheus Operator之前需要先替换一下bundle.yaml文件中所有namespace定义，由default修改为monitoring。 通过运行一下命令安装Prometheus Operator的Deployment实例：

```
$ kubectl -n monitoring apply -f bundle.yaml
clusterrolebinding.rbac.authorization.k8s.io/prometheus-operator created
clusterrole.rbac.authorization.k8s.io/prometheus-operator created
deployment.apps/prometheus-operator created
serviceaccount/prometheus-operator created
service/prometheus-operator created
```

Prometheus Operator通过Deployment的形式进行部署，为了能够让Prometheus Operator能够监听和管理Kubernetes资源同时也创建了单独的ServiceAccount以及相关的授权动作。

查看Prometheus Operator部署状态，以确保已正常运行：

```
$ kubectl -n monitoring get pods
NAME                                   READY     STATUS    RESTARTS   AGE
prometheus-operator-6db8dbb7dd-2hz55   1/1       Running   0          19s
```

使用Operator管理Prometheus

创建Prometheus实例

当集群中已经安装Prometheus Operator之后，对于部署Prometheus Server实例就变成了声明一个Prometheus资源，如下所示，我们在Monitoring命名空间下创建一个Prometheus实例：

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst
  namespace: monitoring
spec:
  resources:
    requests:
      memory: 400Mi
```

将以上内容保存到prometheus-inst.yaml文件，并通过kubectl进行创建：

```
$ kubectl create -f prometheus-inst.yaml
prometheus.monitoring.coreos.com/inst-1 created
```

此时，查看monitoring命名空间下的statefulsets资源，可以看到Prometheus Operator自动通过Statefulset创建的Prometheus实例：

```
$ kubectl -n monitoring get statefulsets
NAME              DESIRED   CURRENT   AGE
prometheus-inst   1         1         1m
```

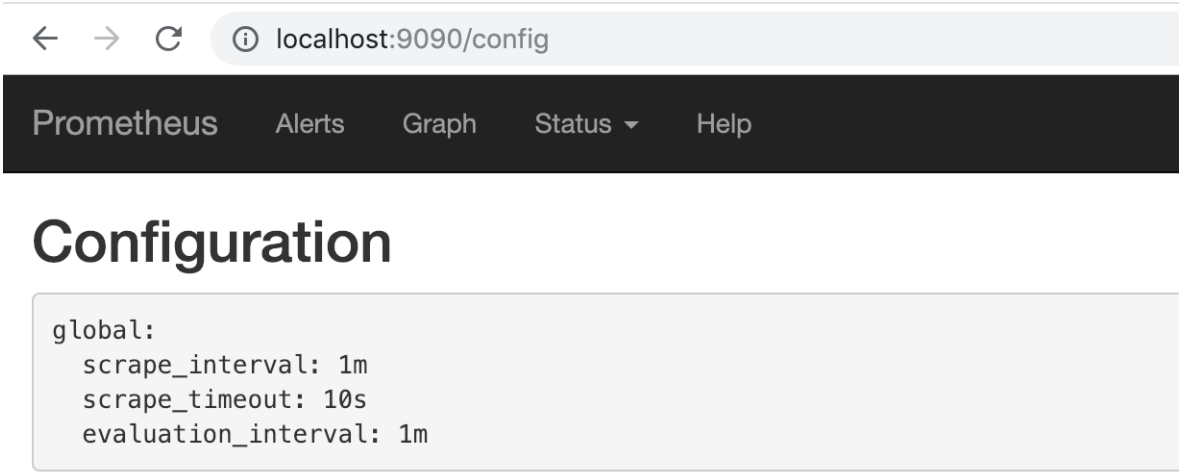
查看Pod实例：

```
$ kubectl -n monitoring get pods
NAME                                   READY     STATUS    RESTARTS   AGE
prometheus-inst-0                      3/3       Running   1          1m
prometheus-operator-6db8dbb7dd-2hz55   1/1       Running   0          45m
```

通过port-forward访问Prometheus实例:

```
$ kubectl -n monitoring port-forward statefulsets/prometheus-inst 9090:9090
```

通过http://localhost:9090可以在本地直接打开Prometheus Operator创建的Prometheus实例。查看配置信息，可以看到目前Operator创建了只包含基本配置的Prometheus实例：



使用ServiceMonitor管理监控配置

修改监控配置项也是Prometheus下常用的运维操作之一，为了能够自动化的管理Prometheus的配置，Prometheus Operator使用了自定义资源类型ServiceMonitor来描述监控对象的信息。

这里我们首先在集群中部署一个示例应用，将以下内容保存到example-app.yaml，并使用kubectl命令行工具创建：

```
kind: Service
apiVersion: v1
metadata:
  name: example-app
  labels:
    app: example-app
spec:
  selector:
    app: example-app
  ports:
  - name: web
    port: 8080
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: example-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: example-app
    spec:
      containers:
      - name: example-app
        image: fabxc/instrumented_app
        ports:
        - name: web
          containerPort: 8080
```

示例应用会通过Deployment创建3个Pod实例，并且通过Service暴露应用访问信息。

```
$ kubectl get pods
NAME                           READY     STATUS    RESTARTS   AGE
```

```
example-app-94c8bc8-l27vx   2/2       Running   0          1m
example-app-94c8bc8-lcsrm   2/2       Running   0          1m
example-app-94c8bc8-n6wp5   2/2       Running   0          1m
```

在本地同样通过port-forward访问任意Pod实例

```
$ kubectl port-forward deployments/example-app 8080:8080
```

访问本地的http://localhost:8080/metrics实例应用程序会返回以下样本数据：

```
# TYPE codelab_api_http_requests_in_progress gauge
codelab_api_http_requests_in_progress 3
# HELP codelab_api_request_duration_seconds A histogram of the API HTTP request durations in seconds.
# TYPE codelab_api_request_duration_seconds histogram
codelab_api_request_duration_seconds_bucket{method="GET",path="/api/bar",status="200",le="0.0001"} 0
```

为了能够让Prometheus能够采集部署在Kubernetes下应用的监控数据，在原生的Prometheus配置方式中，我们在Prometheus配置文件中定义单独的Job，同时使用kubernetes_sd定义整个服务发现过程。而在Prometheus Operator中，则可以直接声明一个ServiceMonitor对象，如下所示：

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: example-app
  namespace: monitoring
  labels:
    team: frontend
spec:
  namespaceSelector:
    matchNames:
    - default
  selector:
    matchLabels:
      app: example-app
  endpoints:
  - port: web
```

通过定义selector中的标签定义选择监控目标的Pod对象，同时在endpoints中指定port名称为web的端口。默认情况下ServiceMonitor和监控对象必须是在相同Namespace下的。在本示例中由于Prometheus是部署在Monitoring命名空间下，因此为了能够关联default命名空间下的example对象，需要使用namespaceSelector定义让其可以跨命名空间关联ServiceMonitor资源。保存以上内容到example-app-service-monitor.yaml文件中，并通过kubectl创建：

```
$ kubectl create -f example-app-service-monitor.yaml
servicemonitor.monitoring.coreos.com/example-app created
```

如果希望ServiceMonitor可以关联任意命名空间下的标签，则通过以下方式定义：

```
spec:
  namespaceSelector:
    any: true
```

如果监控的Target对象启用了BasicAuth认证，那在定义ServiceMonitor对象时，可以使用endpoints配置中定义basicAuth如下所示：

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: example-app
  namespace: monitoring
  labels:
    team: frontend
spec:
  namespaceSelector:
    matchNames:
    - default
  selector:
    matchLabels:
      app: example-app
  endpoints:
  - basicAuth:
      password:
        name: basic-auth
        key: password
      username:
        name: basic-auth
        key: user
    port: web
```

其中basicAuth中关联了名为basic-auth的Secret对象，用户需要手动将认证信息保存到Secret中:

```
apiVersion: v1
kind: Secret
metadata:
  name: basic-auth
data:
  password: dG9vcg== # base64编码后的密码
  user: YWRtaW4= # base64编码后的用户名
type: Opaque
```

关联Promethues与ServiceMonitor

Prometheus与ServiceMonitor之间的关联关系使用serviceMonitorSelector定义，在Prometheus中通过标签选择当前需要监控的ServiceMonitor对象。修改prometheus-inst.yaml中Prometheus的定义如下所示：
为了能够让Prometheus关联到ServiceMonitor，需要在Pormtheus定义中使用serviceMonitorSelector，我们可以通过标签选择当前Prometheus需要监控的ServiceMonitor对象。修改prometheus-inst.yaml中Prometheus的定义如下所示：

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst
  namespace: monitoring
spec:
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  resources:
    requests:
      memory: 400Mi
```

将对Prometheus的变更应用到集群中：

```
$ kubectl -n monitoring apply -f prometheus-inst.yaml
```

此时，如果查看Prometheus配置信息，我们会惊喜的发现Prometheus中配置文件自动包含了一条名为monitoring/example-app/0的Job配置：

```
global:
  scrape_interval: 30s
  scrape_timeout: 10s
  evaluation_interval: 30s
  external_labels:
    prometheus: monitoring/inst
    prometheus_replica: prometheus-inst-0
alerting:
  alert_relabel_configs:
  - separator: ;
    regex: prometheus_replica
    replacement: $1
    action: labeldrop
rule_files:
- /etc/prometheus/rules/prometheus-inst-rulefiles-0/*.yaml
scrape_configs:
- job_name: monitoring/example-app/0
  scrape_interval: 30s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: http
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - default
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_label_app]
    separator: ;
    regex: example-app
    replacement: $1
    action: keep
  - source_labels: [__meta_kubernetes_endpoint_port_name]
    separator: ;
    regex: web
    replacement: $1
    action: keep
  - source_labels: [__meta_kubernetes_endpoint_address_target_kind, __meta_kubernetes_endpoint_address_target
_name]
    separator: ;
    regex: Node;(.*)
    target_label: node
    replacement: ${1}
    action: replace
  - source_labels: [__meta_kubernetes_endpoint_address_target_kind, __meta_kubernetes_endpoint_address_target
_name]
    separator: ;
    regex: Pod;(.*)
    target_label: pod
    replacement: ${1}
    action: replace
  - source_labels: [__meta_kubernetes_namespace]
    separator: ;
    regex: (.*)
    target_label: namespace
```

```
    replacement: $1
    action: replace
  - source_labels: [__meta_kubernetes_service_name]
    separator: ;
    regex: (.*)
    target_label: service
    replacement: $1
    action: replace
  - source_labels: [__meta_kubernetes_pod_name]
    separator: ;
    regex: (.*)
    target_label: pod
    replacement: $1
    action: replace
  - source_labels: [__meta_kubernetes_service_name]
    separator: ;
    regex: (.*)
    target_label: job
    replacement: ${1}
    action: replace
  - separator: ;
    regex: (.*)
    target_label: endpoint
    replacement: web
    action: replace
```

不过，如果细心的读者可能会发现，虽然Job配置有了，但是Prometheus的Target中并没包含任何的监控对象。查看Prometheus的Pod实例日志，可以看到如下信息：

```
level=error ts=2018-12-15T12:52:48.452108433Z caller=main.go:240 component=k8s_client_runtime err="github.com/prometheus/prometheus/discovery/kubernetes/kubernetes.go:300: Failed to list *v1.Endpoints: endpoints is forbidden: User \"system:serviceaccount:monitoring:default\" cannot list endpoints in the namespace \"default\""
```

自定义ServiceAccount

由于默认创建的Prometheus实例使用的是monitoring命名空间下的default账号，该账号并没有权限能够获取default命名空间下的任何资源信息。

为了修复这个问题，我们需要在Monitoring命名空间下为创建一个名为Prometheus的ServiceAccount，并且为该账号赋予相应的集群访问权限。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: monitoring
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - services
  - endpoints
```

```
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
  - configmaps
  verbs: ["get"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: monitoring
```

将以上内容保存到prometheus-rbac.yaml文件中，并且通过kubectl创建相应资源：

```
$ kubectl -n monitoring create -f prometheus-rbac.yaml
serviceaccount/prometheus created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
```

在完成ServiceAccount创建后，修改prometheus-inst.yaml，并添加ServiceAccount如下所示：

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst
  namespace: monitoring
spec:
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  resources:
    requests:
      memory: 400Mi
```

保存Prometheus变更到集群中：

```
$ kubectl -n monitoring apply -f prometheus-inst.yaml
prometheus.monitoring.coreos.com/inst configured
```

等待Prometheus Operator完成相关配置变更后，此时查看Prometheus，我们就能看到当前Prometheus已经能够正常的采集实例应用的相关监控数据了。

使用Operator管理监控配置

对于Prometheus而言，在原生的管理方式上，我们需要手动创建Prometheus的告警文件，并且通过在Prometheus配置中声明式的加载。而在Prometheus Operator模式中，告警规则也编程一个通过Kubernetes API 声明式创建的一个资源，如下所示：

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: example
    role: alert-rules
  name: prometheus-example-rules
spec:
  groups:
  - name: ./example.rules
    rules:
    - alert: ExampleAlert
      expr: vector(1)
```

将以上内容保存为example-rule.yaml文件，并且通过kubectl命令创建相应的资源：

```
$ kubectl -n monitoring create -f example-rule.yaml
prometheusrule "prometheus-example-rules" created
```

告警规则创建成功后，通过在Prometheus中使用ruleSelector通过选择需要关联的PrometheusRule即可：

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst
  namespace: monitoring
spec:
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  ruleSelector:
    matchLabels:
      role: alert-rules
      prometheus: example
  resources:
    requests:
      memory: 400Mi
```

Prometheus重新加载配置后，从UI中我们可以查看到通过PrometheusRule自动创建的告警规则配置：

如果查看Alerts页面，我们会看到告警已经处于触发状态。

使用Operator管理Alertmanager实例

到目前为止，我们已经通过Prometheus Operator的自定义资源类型管理了Promtheus的实例，监控配置以及告警规则等资源。通过Prometheus Operator将原本手动管理的工作全部变成声明式的管理模式，大大简化了Kubernetes下的Prometheus运维管理的复杂度。 接下来，我们将继续使用Promtheus Operator定义和管理Alertmanager相关的内容。

为了通过Prometheus Operator管理Alertmanager实例，用户可以通过自定义资源Alertmanager进行定义，如下所示，通过replicas可以控制Alertmanager的实例数：

```
apiVersion: monitoring.coreos.com/v1
kind: Alertmanager
metadata:
  name: inst
  namespace: monitoring
spec:
  replicas: 3
```

当replicas大于1时，Prometheus Operator会自动通过集群的方式创建Alertmanager。将以上内容保存为文件alertmanager-inst.yaml，并通过以下命令创建：

```
$ kubectl -n monitoring create -f alertmanager-inst.yaml
alertmanager.monitoring.coreos.com/inst created
```

查看Pod的情况如下所示，我们会发现Alertmanager的Pod实例一直处于ContainerCreating的状态中:

```
$ kubectl -n monitoring get pods
NAME                                  READY     STATUS              RESTARTS   AGE
alertmanager-inst-0                   0/2       ContainerCreating   0          32s
```

通过kubectl describe命令查看该Alertmanager的Pod实例状态，可以看到类似于以下内容的告警信息：

```
MountVolume.SetUp failed for volume "config-volume" : secrets "alertmanager-inst" not found
```

这是由于Prometheus Operator通过Statefulset的方式创建的Alertmanager实例，在默认情况下，会通过 `alertmanager-{ALERTMANAGER_NAME}` 的命名规则去查找Secret配置并以文件挂载的方式，将Secret的内容作为配置文件挂载到Alertmanager实例当中。因此，这里还需要为Alertmanager创建相应的配置内容，如下所示，是Alertmanager的配置文件：

```
global:
  resolve_timeout: 5m
route:
  group_by: ['job']
  group_wait: 30s
```

```
  group_interval: 5m
  repeat_interval: 12h
  receiver: 'webhook'
receivers:
- name: 'webhook'
  webhook_configs:
  - url: 'http://alertmanagerwh:30500/'
```

将以上内容保存为文件alertmanager.yaml，并且通过以下命令创建名为alrtmanager-inst的Secret资源：

```
$ kubectl -n monitoring create secret generic alertmanager-inst --from-file=alertmanager.yaml
secret/alertmanager-inst created
```

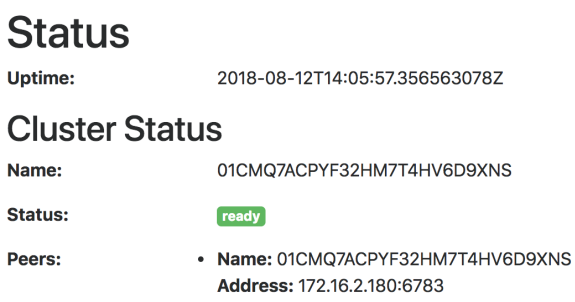
在Secret创建成功后，查看当前Alertmanager Pod实例状态。如下所示：

```
$ kubectl -n monitoring get pods
NAME                                  READY     STATUS    RESTARTS   AGE
alertmanager-inst-0                   2/2       Running   0          5m
alertmanager-inst-1                   2/2       Running   0          52s
alertmanager-inst-2                   2/2       Running   0          37s
```

使用port-forward将Alertmanager映射到本地：

```
$ kubectl -n monitoring port-forward statefulsets/alertmanager-inst 9093:9093
```

访问http://localhost:9093/#/status，并查看当前集群状态：



接下来，我们只需要修改我们的Prometheus资源定义，通过alerting指定使用的Alertmanager资源即可：

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst
  namespace: monitoring
spec:
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  ruleSelector:
    matchLabels:
      role: alert-rules
      prometheus: example
  alerting:
```

```
    alertmanagers:
    - name: alertmanager-example
      namespace: monitoring
      port: web
  resources:
    requests:
      memory: 400Mi
```

等待Prometheus重新加载后，我们可以看到Prometheus Operator在配置文件中添加了以下配置：

```
  alertmanagers:
  - kubernetes_sd_configs:
    - role: endpoints
      namespaces:
        names:
        - monitoring
    scheme: http
    path_prefix: /
    timeout: 10s
    relabel_configs:
    - source_labels: [__meta_kubernetes_service_name]
      separator: ;
      regex: alertmanager-example
      replacement: $1
      action: keep
    - source_labels: [__meta_kubernetes_endpoint_port_name]
      separator: ;
      regex: web
      replacement: $1
      action: keep
```

通过服务发现规则将Prometheus与Alertmanager进行了自动关联。

在Prometheus Operator中使用自定义配置

在Prometheus Operator我们通过声明式的创建如Prometheus, ServiceMonitor这些自定义的资源类型来自动化部署和管理Prometheus的相关组件以及配置。而在一些特殊的情况下，对于用户而言，可能还是希望能够手动管理Prometheus配置文件，而非通过Prometheus Operator自动完成。 为什么？ 实际上Prometheus Operator对于Job的配置只适用于在Kubernetes中部署和管理的应用程序。如果你希望使用Prometheus监控一些其他的资源，例如AWS或者其他平台中的基础设施或者应用，这些并不在Prometheus Operator的能力范围之内。

为了能够在通过Prometheus Operator创建的Prometheus实例中使用自定义配置文件，我们只能创建一个不包含任何与配置文件内容相关的Prometheus实例

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: inst-cc
  namespace: monitoring
spec:
  serviceAccountName: prometheus
  resources:
    requests:
      memory: 400Mi
```

将以上内容保存到prometheus-inst-cc.yaml文件中，并且通过kubectl创建:

```
$ kubectl -n monitoring create -f prometheus-inst-cc.yaml
prometheus.monitoring.coreos.com/inst-cc created
```

如果查看新建Prometheus的Pod实例YAML定义，我们可以看到Pod中会包含一个volume配置：

```
volumes:
  - name: config
    secret:
      defaultMode: 420
      secretName: prometheus-inst-cc
```

Prometheus的配置文件实际上是保存在名为 `prometheus-<name-of-prometheus-object>` 的Secret中，当用户创建的Prometheus中关联ServiceMonitor这类会影响配置文件内容的定义时，Promethues Operator会自动管理。而如果Prometheus定义中不包含任何与配置相关的定义，那么Secret的管理权限就落到了用户自己手中。

通过修改prometheus-inst-cc的内容，从而可以让用户可以使用自定义的Prometheus配置文件，作为示例，我们创建一个prometheus.yaml文件并添加以下内容：

```
global:
  scrape_interval: 10s
  scrape_timeout: 10s
  evaluation_interval: 10s
```

生成文件内容的base64编码后的内容：

```
$ cat prometheus.yaml | base64
Z2xvYmFsOgogIHNjcmFwZV9pbnRlcnZhbDogMTBzCiAgc2NyYXBlX3RpbWVvdXQ6IDEwcwogIGV2YWx1YXRpb25faW50ZXJ2YWw6IDEwcw==
```

修改名为prometheus-inst-cc的Secret内容，如下所示：

```
$ kubectl -n monitoring edit secret prometheus-inst-cc
# 省略其它内容
data:
  prometheus.yaml: "Z2xvYmFsOgogIHNjcmFwZV9pbnRlcnZhbDogMTBzCiAgc2NyYXBlX3RpbWVvdXQ6IDEwcwogIGV2YWx1YXRpb25faW50ZXJ2YWw6IDEwcw=="
```

通过port-forward在本地访问新建的Prometheus实例，观察配置文件变化即可：

```
kubectl -n monitoring port-forward statefulsets/prometheus-inst-cc 9091:9090
```

小结

在本章中，我们介绍了在Kubernetes下如何使用Operator来有状态的运维和管理Prometheus以及Alertmanager等组件。