Prometheus中文文档

Prometheus官网：https://prometheus.io/
整理自网络，侵删歉！

目　录

介绍

总览

什么是prometheus？

Prometheus是一个开源监控系统，它前身是SoundCloud的警告工具包。从2012年开始，许多公司和组织开始使用Prometheus。该项目的开发人员和用户社区非常活跃，越来越多的开发人员和用户参与到该项目中。目前它是一个独立的开源项目，且不依赖与任何公司。 为了强调这点和明确该项目治理结构，Prometheus在2016年继Kurberntes之后，加入了Cloud Native Computing Foundation。

特征：

Prometheus的主要特征有：

1. 多维度数据模型
2. 灵活的查询语言
3. 不依赖分布式存储，单个服务器节点是自主的
4. 以HTTP方式，通过pull模型拉去时间序列数据
5. 也通过中间网关支持push模型
6. 通过服务发现或者静态配置，来发现目标服务对象
7. 支持多种多样的图表和界面展示，grafana也支持它

组件

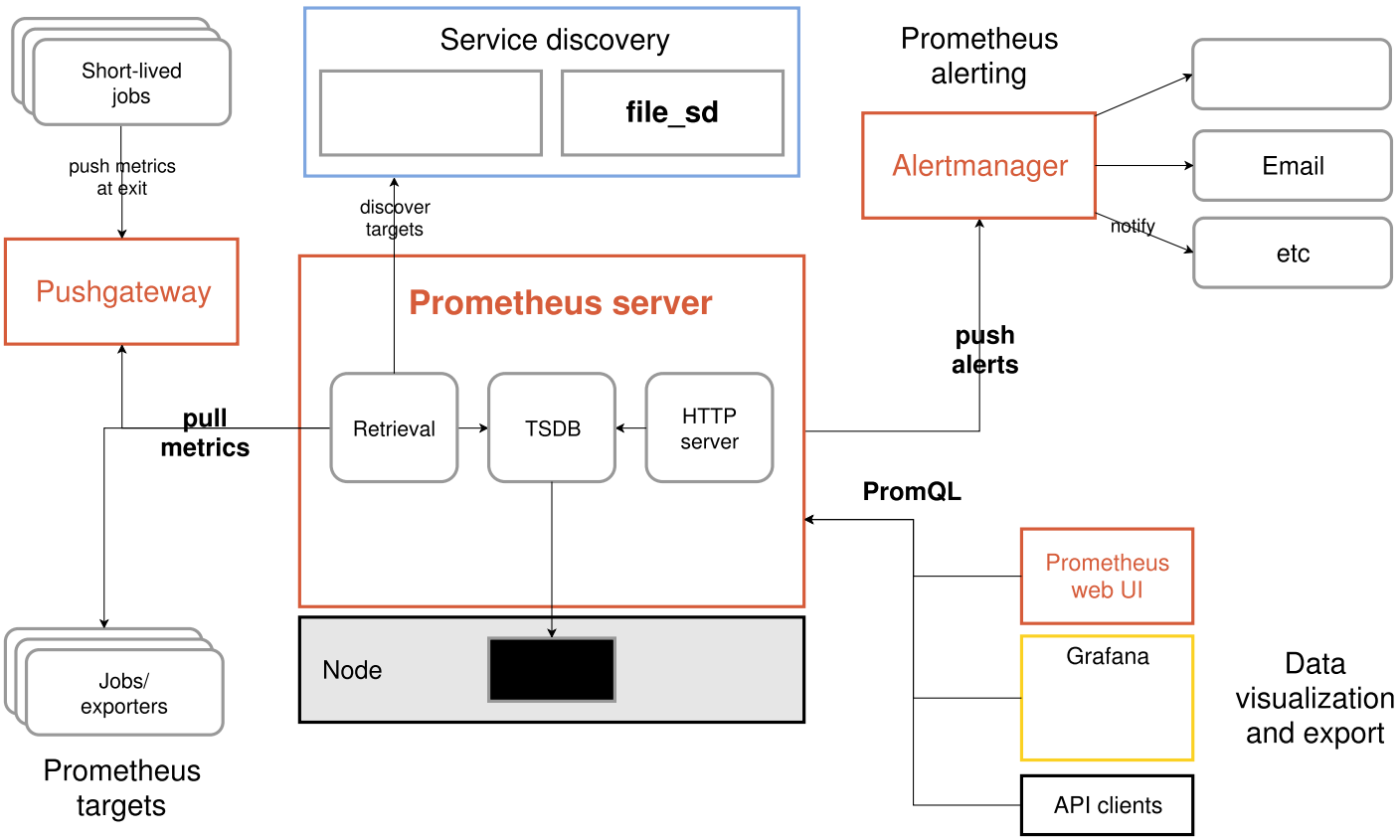
Prometheus生态包括了很多组件，它们中的一些是可选的：

1. 主服务Prometheus Server负责抓取和存储时间序列数据
2. 客户库负责检测应用程序代码
3. 支持短生命周期的PUSH网关
4. 基于Rails/SQL仪表盘构建器的GUI
5. 多种导出工具，可以支持Prometheus存储数据转化为HAProxy、StatsD、Graphite等工具所需要的数据存储格式
6. 警告管理器
7. 命令行查询工具
8. 其他各种支撑工具

多数Prometheus组件是Go语言写的，这使得这些组件很容易编译和部署。

架构

下面这张图说明了Prometheus的整体架构，以及生态中的一些组件作用:



Prometheus服务，可以直接通过目标拉取数据，或者间接地通过中间网关拉取数据。它在本地存储抓取的所有数据，并通过一定规则进行清理和整理数据，并把得到的结果存储到新的时间序列中，PromQL和其他API可视化地展示收集的数据

适用场景

Prometheus在记录纯数字时间序列方面表现非常好。它既适用于面向服务器等硬件指标的监控，也适用于高动态的面向服务架构的监控。对于现在流行的微服务，Prometheus的多维度数据收集和数据筛选查询语言也是非常的强大。

Prometheus是为服务的可靠性而设计的，当服务出现故障时，它可以使你快速定位和诊断问题。它的搭建过程对硬件和服务没有很强的依赖关系。

不适用场景

Prometheus，它的价值在于可靠性，甚至在很恶劣的环境下，你都可以随时访问它和查看系统服务各种指标的统计信息。 如果你对统计数据需要100%的精确，它并不适用，例如：它不适用于实时计费系统

安装

安装

使用预编译二进制文件

我们为Prometheus大多数的官方组件，提供了预编译二进制文件。可用版本下载列表

源码安装

如果要从源码安装Prometheus的官方组件，可以查看各个项目源码目录下的 `Makefile`

> 注意点：在web上的文档指向最新的稳定版(不包括预发布版)。下一个版本指向master分支还没有发布的版本

Docker安装

所有Prometheus服务的Docker镜像在官方组织prom下，都是可用的
在Docker上运行Prometheus服务，只需要简单地执行 `docker run -p 9090:9090 prom/prometheus` 命令行即可。这条命令会启动Prometheus服务，使用的是默认配置文件，并对外界暴露9090端口
Prometheus镜像使用docker中的volumn卷存储实际度量指标。在生产环境上使用容器卷模式, 可以在Prometheus更新和升级时轻松管理Prometheus数据， 这种使用docker volumn卷方式存储数据，是被docker官方强烈推荐的.
通过几个选项，可以达到使用自己的配置的目的。下面有两个例子。

卷&绑定挂载

在运行Prometheus服务的主机上，做一个本地到docker容器的配置文件关系映射。

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

或者为这个配置文件使用一个独立的volumn

```
docker run -p 9090:9090 -v /prometheus-data
prom/prometheus --config.file=/prometheus-data/prometheus.yml
```

自定义镜像

为了避免在主机上与docker映射配置文件，我们可以直接将配置文件拷贝到docker镜像中。如果Prometheus配置是静态的，并且在所有服务器上的配置相同，这种把配置文件直接拷贝到镜像中的方式是非常好的。
例如：利用Dockerfile创建一个Prometheus配置目录， Dockerfile应该这样写：

```
FROM prom/prometheus
ADD prometheus.yml /etc/prometheus/
```

然后编译和运行它：

```
docker build -t my-prometheus .
docker run -p 9090:9090 my-prometheus
```

一个更高级的选项是可以通过一些工具动态地渲染配置，甚至后台定期地更新配置

使用配置管理系统

如果你喜欢使用配置管理系统，你可能对下面地第三方库感兴趣：

Ansible：

- griggheo/ansible-prometheus
- William-Yeh/ansible-prometheus

Chef:

- rayrod2030/chef-prometheus

SaltStack:

- bechtoldt/saltstack-prometheus-formula

启动

入门教程

本篇是一篇 `hello, world` 风格的入门指南，使用一个简单的例子，向大家演示怎么样安装、配置和使用Prometheus。你可以下载和本地运行Prometheus服务，通过配置文件收集Prometheus服务自己产生的数据，并在这些收集数据的基础上，进行查询、制定规则和图表化显示所关心的数据

下载和运行Prometheus

最新稳定版下载地址, 选择合适的平台，然后提取并运行它

> tar xvfz prometheus-*.tar.gz

> cd prometheus-*

在运行Prometheus服务之前，我们需要指定一个该服务运行所需要的配置文件

配置Prometheus服务监控本身

Prometheus通过Http方式拉取目标机上的度量指标。Prometheus服务也暴露自己运行所产生的数据，它能够抓取和监控自己的健康状况。
实际上，Prometheus服务收集自己运行所产生的时间序列数据，是没有什么意义的。但是它是一个非常好的入门级教程。保存一下的Prometheus配置到文件中，并自定义命名该文件名，如：prometheus.yml

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here its 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'

    # Override the global default and scrape targets from this job every 5 seco
nds.
    scrape_interval: 5s
```

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

一个完整的配置选项，可以查看文件文档

启动Prometheus服务

cd到Prometheus服务目录，并指定刚刚自定义好的配置文件，并启动Prometheus服务, 如下所示：

> start Prometheus.

> By default, Prometheus stores its database in ./data (flag -storage.local.path).

> ./prometheus -config.file={$dir}/prometheus.yml # $dir = absolutely/relative path

Prometheus服务启动成功后，然后再打开浏览器在页面上数据http://localhost:9090. 服务运行几秒后，会开始收集自身的时间序列数据
你也可以通过在浏览器输入http://localhost:9090/metrics, 直接查看Prometheus服务收集到的自身数据
Prometheus服务执行的操作系统线程数量由GOMAXPROCS环境变量控制。从Go 1.5开始，默认值是可用的CPUs数量
盲目地设置 `GOMAXPROCS` 到一个比较高德值，有可能会适得其反。见Go FAQs
注意：Prometheus服务默认需要3GB的内存代销。如果你的机器内存比较小， 你可以调整Prometheus服务使用更少的内存。详细见内存使用文档

使用表达式浏览器

我们试着查看一些Prometheus服务自身产生的数据。为了使用Prometheus内置表达式浏览器，可以在浏览器中数据http://localhost:9090/graph, 选择"Console"视图，同一层级还有"Graph"tab。
如果你可以从http://localhost:9090/metrics查看到收集的度量指标数据，那么其中有一个指标数据名称为 `prometheus_target_interval_length_seconds` (两次抓取数据之间的时间差)可以被提取出来，可以在表达式控制框中输入：

> prometheus_target_interval_length_seconds

它应该会返回带有 `prometheus_target_interval_length_seconds` 度量指标的许多时间序列数据，只是带有不能标签, 这些标签有不同的延迟百分比和目标群组之间的间隔。
如果我们仅仅对p99延迟感兴趣，我们使用下面的查询表达式收集该信息

> prometheus_target_interval_length_seconds{quantile="0.99"}

为了统计时间序列数据记录的总数量，你可以写：

> count(prometheus_target_interval_length_seconds)

更多的表达式语言，详见表达式语言文档

使用图形界面

使用http://localhost:9090/graph链接，查看图表"Graph"。
例如：输入下面的表达式，绘制在Prometheus服务中每秒存储的速率.

> rate(prometheus_local_storage_chunk_ops_total[1m])

启动一些样本目标机

我们更感兴趣的是Prometheus服务抓取其他目标机的数据采样，并非自己的时间序列数据。Go客户库有一个例子，它会产生一些自己造的RPC延迟。启动三个带有不同的延时版本。
首先需要确保你有Go的环境
下载Go的Prometheus客户端，并运行下面三个服务：

```
 # Fetch the client library code and compile example.
git clone https://github.com/prometheus/client_golang.git
cd client_golang/examples/random
go get -d
go build

# Start 3 example targets in separate terminals:
./random -listen-address=:8080
./random -listen-address=:8081
./random -listen-address=:8082
```

你现在应该浏览器输入http://localhost:8080/metrics, http://localhost:8081/metrics, and http://localhost:8082/metrics, 会看到这些服务所产生的度量指标数据.

配置Prometheus服务，监听样本目标实例

现在我们将配置Prometheus服务，收集这三个例子的度量指标数据。我们把这三个服务实例命名为一个任务称为 `example-random` ，并把8080端口服务和8081端口服务作为生产目标group，8082端口成为canary group。 为了在Prometheus服务中建模这个，我们需要添加两个群组到这个任务中，增加一些标签到不同的目标群组中。在这个例子中，我们会增加 `group="production"` 标签到带个目标组中，另外一个则是 `group="canary"`
为了达到这个目的，在 `prometheus.yml` 配置文件中，增加下面任务定义到 `scrape_config` 区域中, 并重启Prometheus服务：

```
scrape_configs:
  - job_name:       'example-random'

    # Override the global default and scrape targets from this job every 5 seco
nds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

去表达式浏览器中验证Prometheus服务是否能统计到这两个群组的目标机度量数据，如：

`rpc_durations_seconds` 度量指标

为聚集到抓取的数据，设置规则并写入到新的时间序列中

当计算ad-hoc时，如果在累计到上千个时间序列数据的查询，可能会变慢。为了使这种多时间序列数据点查询更有效率，我们允许通过使用配置的记录规则，把预先记录表达式实时收集的数据存入到新的持久时间序列中。该例子中，如果我们对每秒RPCs数量(`rpc_durations_seconds_count`)的5分钟窗口流入的统计数量感兴趣的话。我们可以下面的表达式：

> avg(rate(rpc_durations_seconds_count)[5m]) by (job, service)

试着使用图形化这个表达式

为了存储这个表达式所统计到的数据，我们可以使用新的度量指标，如 `job_service:rpc_durations_seconds_count:avg_rate5m` ，创建一个配置规则文件，并把该文件保存为 `prometheus.rules` ：

> job_service:rpc_durations_seconds_count:avg_rate5m = avg(rate(rpc_durations_seconds_count[5m])) by (job, service)

为了使Prometheus服务使用这个新的规则，在 `prometheus.yml` 配置文件的global配置区域添加一个 `rule_files` 语句。这个配置应该向下面这样写：

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.
```

```
  # Attach these extra labels to all timeseries collected by this Prometheus in
stance.
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  - 'prometheus.rules'

scrape_configs:
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seco
nds.
    scrape_interval: 5s

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

  - job_name:       'example-random'

    # Override the global default and scrape targets from this job every 5 seco
nds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

指定这个新的配置文件，并重启Prometheus服务。验证新的时间序列度量指标 `job_service:rpc_durations_seconds_count:avg_rate5m` 是否能够在Console控制框中查找出时间序列数据

对比

监控系统产品比较

Prometheus vs. Graphite

范围

Graphite专注于查询语言和图表特征的时间序列数据库。其他都需要依赖外部组件实现。
Prometheus是一个基于时间序列数据的完整监控系统和趋势系统，包括内置和主动抓取、存储、查询、图表展示和报警功能。它懂得监控系统和趋势系统应该是什么样的（哪些目标机应该存在，哪些时间序列模型存在问题等等），并积极地试着找出故障

数据模型

Graphite和Prometheus一样，存储时间序列数值样本。但是Prometheus的元数据模型更加丰富：Graphite的度量指标名称是由"."分隔符，隐式地编码多维度。而Prometheus度量指标是可以自定义名称的，并以key-value键值对的标签形式，成为度量指标的标签属性列表。并在此基础上，使用Prometheus查询语言可以轻松地进行过滤，分组和匹配操作。
进一步地，当Graphite与StatsD结合使用时，Graphite就只是对一个聚合数据的存储系统了，而不是把目标实例作为一个维度，并深入分析目标实例出现的各种问题。
例如：我们用Graphite/StatsD统计HTTP请求api-server服务的数量，前置条件：返回码是 `500` ，请求方法是 `POST` ，访问URL为 `/tracks` ，key如下所示：

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

但是在Prometheus中同样的数据存储可能像下面一样(假设有三个api-server)：

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

由上可以看到，三个api-server各自的度量指标数据，Prometheus把api-server也作为了一个维度，便于分析api-server服务出现的各种问题

存储

Graphite存储以Whisper方式把时间序列数据存储到本地磁盘，这种数据存储格式是RRD格式数据库，它期望样

本能够定期到达。 任何时间序列在一个单独的文件中存储，一段时间后新采集的样本会覆盖老数据
Prometheus也为每一个时间序列创建了一个本地文件，但是它允许时间序列以任意时间到达。新采集的样本被简单地追加到文件尾部，老数据可以任意长的时间保留。Prometheus对于短生命周期、且经常变化的时间序列集也可以表现得很好

总结

Prometheus提供了一个丰富的数据模型和查询语言，而且更加容易地运行和集成到你的环境中。如果你想要一个可以长期保留历史数据的集群解决方案，Graphite可能是一个更好的选择。

Prometheus vs. InfluxDB

InfluxDB是一个开源的时间序列数据库，它的商业版本具有可扩展和集群化的特性。在Prometheus刚刚开始开发时，InfluxDB项目已经发布了近一年时间。但是这两款产品还是有很大的不同之处，这两个系统也有一些略有不同的应用小场景。

范围

公平起见，我们必须把InfluxDB和Kapacitor结合起来，与Prometheus和Prometheus的报警管理工具比较。
Graphite与Prometheus的范围差异，同样适用于InfluxDB本身。此外InfluxDB提供了连续查询，和Prometheus的记录规则一样。
Kapacitor的作用范围相当于，Prometheus的记录规则、告警规则和警告通知功能的结合。Prometheus提供了一个更加丰富地用于图表化和警告的查询语言，Prometheus告警器还提供了分组、重复数据删除和静默功能(silencing functionality)。

数据模型/存储

和Prometheus一样，InfluxDB数据模型采用的标签也是键值对形式，被称为tags。而且InfluxDB有第二级标签，被称为fields，它被更多地限制使用。InfluxDB支持高达纳秒级的时间戳，以及float64、int64、bool和string的数据类型。相反地，Prometheus仅仅支持float64的数据类型，strings和毫秒只能小范围地支持
InfluxDB使用变种的日志结构合并树结构来存储预写日志，并按时间分片。这比Prometheus的文件追加更适合事件记录
Logs and Metrics and Graphs, Oh My描述了事件日志和度量指标记录的不同

框架

Prometheus服务独立运行，没有集群架构，它仅仅依赖于本地存储。Prometheus有四个核心的功能：抓取、规则处理和警告。InfluxDB的开源版本也是类似的。
InfluxDB的商业版本具有存储和查询的分布式版本, 存储和查询由集群中的节点同时处理。
这意味着商业版本的InfluxDB更加容易的水平扩展，同时也表示你必须从一开始就要管理分布式存储系统的复杂性。而Prometheus运行非常简单，而且在某些时候，你需要在可扩展性边界（如产品，服务，数据中心或者类似方面）明确分片服务器。单Prometheus服务也可以为您提供更好的可靠性和故障隔离。

Kapacitor对规则、警告和通知当前还没有内置/冗余选项。相反地，通过运行Prometheus的冗余副本和使用警告管理器的高可用模式提供了冗余选项。Kapacitor通过用户手动水平切分能够被缩放，这点类似于Prometheus本身

总结

在两个系统之间有许多相似点。1. 利用标签（tags/labels）有效地支持多维度量指标。2. 使用相同的压缩算法。3.都可扩展集成。4.允许使用第三方进行监控系统的扩展，如：统计分析工具、自动化操作
InfluxDB更好之处：

- 使用事件日志
- 商业版本提供的集群方案，对于长期的时间序列存储是非常不错的
- 复制的数据最终一致性

Prometheus更好之处：

- 主要做度量指标监控
- 更强大的查询语言，警告和通知功能
- 图表和警告的高可靠性和稳定性

InfluxDB是有一家商业公司按照开放核心模式运营，提供高级功能，如：集群是闭源的，托管和支持。
Prometheus是一个完全开放和独立的项目，有许多公司和个人维护，其中也提供一些商业服务和支持。

Prometheus vs. OpenTSDB

OpenTSDB是一个基于hadoop和Hbase的分布式时间序列数据库

范围

和Graphite vs. Prometheus的范围一样

数据模型

OpenTSDB的数据模型几乎和Prometheus一样：时间序列由任意的tags键值对集合表示。所有的度量指标存放在一起，并限制度量指标的总数量大小。Prometheus和OpenTSDB有一些细微的差别，例如：Prometheus允许任意的标签字符，而OpenTSDB的tags命名有一定的限制.OpenTSDB缺乏灵活的查询语言支持，通过它提供的API只能简单地进行聚合和数学计算

存储

OpenTSDB的存储由Hadoop和HBase实现的。这意味着水平扩展OpenTSDB是非常容易的，但是你必须接受集群的总体复杂性
Prometheus初始运行非常简单，但是一旦超过单个节点的容量，就需要进行水平切分服务操作

总结

Prometheus提供了一个非常灵活且丰富的查询语言，能够支持更多的度量指标数量，组成整个监控系统的一部分。如果你对hadoop非常熟悉，并且对时间序列数据有长期的存储要求，OpenTSDB是一个不错的选择

Prometheus vs. Nagios

Nagios是一款产生于90s年代的监控系统

范围

Nagios是基于脚本运行结果的警告系统，又称"运行结果检查"。有警告通知，但是没有分组、路由和重复数据删除功能。
Nagios有大量的插件。例如：perfData插件抓取数据后写入到时间序列数据库（Graphite）或者使用NRPE在远程计算机上运行检查

数据模型

Nagios是基于主机的，每一台主机有一个或者多个服务。其中一个是check运行检查，但是没有标签和查询语言的概念
Nagios除了检查脚本运行状态，没有任何存储功能。有第三方插件可以存储数据，并可视化数据

架构

Nagios是单实例服务，所有的检查配置项统一由一个文件配置。

总结

Nagios对于小型监控或者黑盒测试时非常有效的。如果你想要做白盒监控，或者动态地，基于云环境的数据监控，Prometheus是一个不错的选择

Prometheus vs. Sensu

广义上说，Sensu是一个更加现代的Nagios。

范围

主要不同点在于Sensu客户端注册自己，并确定从本地还是其他地方获取配置检查。Senus对perfData的数量没有限制。 还有一个客户端socket允许把任意检查结果推送到Senus

数据模型

和Nagios一样

存储

Sensu在Redis中存储数据，存储被称作stashes。主要是静默存储，同时它也存储在Senus上注册的所有客户端

架构

Sensu有很多组件。它使用Rabbit消息队列进行数据传输，使用Redis存储当前状态，独立的服务处理数据RabbitMQ和Redis都可以是集群的，运行多个服务器副本可是实现副本和冗余

总结

如果已经有了Nagios服务，你希望扩展它，同时希望使用Senus的注册特性，那么Senus是一个不错的选择如果你想要使用白盒、或者有一个动态的云环境，那么Prometheus是一个很好的选择。

常见问题

常见问题

一般问题

1. Prometheus是什么？

> Prometheus是一款高活跃生态系统的开源系统监控和警告工具包。详见概览

2. Prometheus与其他的监控系统比较

> 详见比较

3. Prometheus有什么依赖？

> Prometheus服务独立运行，没有其他依赖

4. Prometheus有高可用的保证吗？

> 是的，在多台服务器上运行相同的Prometheus服务，相同的报警会由警告管理器删除
> 警告管理器当前不能保证高可用，但高可用是目标

5. 我被告知Prometheus"不能水平扩展"

> 事实上，有许多方式可以扩展Prometheus。 阅读Robust Percetion的博客关于Prometheus的扩展

5. Prometheus是什么语言写的？

> 大多数Prometheus组件是由Go语言写的。还有一些是由Java，Python和Ruby写的

6. Prometheus的特性、存储格式和APIs有多稳定？

> Prometheus从v1.0.0版本开始就非常稳定了，我们现在有一些版本功能规划,详见路线图

7. 为什么是使用的是pull而不是push？

基于Http方式的拉模型提供了一下优点：

- 当开发变化时，你可以在笔记本上运行你的监控
- 如果目标实例挂掉，你可以很容易地知道
- 你可以手动指定一个目标，并通过浏览器检查该目标实例的监控状况

总体来说，我们相信拉模式比推模式要好一地啊你，但是当考虑一个监控系统时，它不是主要的考虑点

Push vs. Pull监控在Brian Brazil的博客中被详细的描述

如果你必须要用Push模式，我们提供Pushgateway

8. 怎么样把日志推送到Prometheus系统中？

> 简单地回答：千万别这样做，你可以使用ELK栈去实现
> 比较详细的回答：Prometheus是一款收集和处理度量指标的系统，并非事件日志系统。Raintank的博客有关日志、度量指标和图表在日志和度量指标之间，进行了详尽地阐述。

如果你想要从应用日志中提取Prometheus度量指标中。 谷歌的mtail可能会更有帮助

9. 谁写的Prometheus？

> Prometheus项目发起人是Matt T. Proud和Julius Volz。 一开始大部分的开发是由SoundCloud赞助的
> 现在它由许多公司和个人维护和扩展

10. 当前Prometheus的许可证是用的哪个？

> Apache 2.0

11. Prometheus单词的复数是什么？

> Prometheis

12. 我能够动态地加载Prometheus的配置吗？

> 是的，通过发送SIGHUP信号量给Prometheus进行，将会重载配置文件。不同的组件会优雅地处理失败的更改

13. 我能发送告警吗？

> 是的，通过警告管理器
> 当前，下面列表的外部系统都是被支持的

- Email
- General Webhooks
- PagerDuty(http://www.pagerduty.com/)
- HipChat(https://www.hipchat.com/)
- Slack(https://slack.com/)
- Pushover(https://pushover.net/)
- Flowdock(https://www.flowdock.com/)

14. 我能创建Dashboard吗？

> 是的，但是在生产使用中，我们推荐用Grafana。PromDash和Console templates也可以

15. 我能改变timezone和UTC吗？

> 不行。为了避免任何时区的困惑和混乱，我们用了UTC这个通用单位

工具库

1. 哪些语言有工具库？

> 这里有很多客户端库，用Prometheus的度量指标度量你的服务。详见客户库
> 如果你对功能工具库非常感兴趣，详见exposition formats

2. 我能监控机器吗？

> 是的。Node Exporter暴露了很多机器度量指标，包括CPU使用率、内存使用率和磁盘利用率、文件系统的余量和网络带宽等数据

3. 我能监控网络数据吗？

> 是的。SNMP Exporter允许监控网络设备

4. 我能监控批量任务吗？

> 是的，通过Pushgateway. 详见最佳实践

5. Prometheus的第三方工具有哪些？

> 详见exporters for third-party systems

6. 我能通过JMX监控JVM应用程序吗？

> 是的。不能直接使用Java客户端进行测试的应用程序，你可以将JMX Exporter单独使用或者Java代理使用

7. 工具对性能的影响是什么？

> 客户端和语言的性能可能不同。对于Java，基准表明使用Java客户端递增计数器需要12~17ns，具体依赖于竞争。最关键的延迟关键代码之外的所有代码都是可以忽略的。

故障排除

1. 当服务崩溃恢复后，我的服务需要很多时间启动和清理垃圾日志。

> 你的服务可能遭到了不干净的关闭。Prometheus必须在SIGTERM后彻底关闭，特别地对于一些重量级服务可能需要比较长的时间去。如果服务器崩溃或者司机（如：在等待Prometheus关闭时，内核的OOM杀死你的Prometheus服务），必须执行崩溃恢复，这在正常情况下需要不到一分钟。详见崩溃恢复

2. 我在Linux上使用ZFS，单元测试TestPersistLoadDropChunks失败。尽管测试失败，我运行Prometheus服务，奇怪的事情会发生。

你在Linux上有bug的ZFS文件系统运行Prometheus服务。详见Issue #484, 在linux v.0.6.4上升级ZFS应该可以解决该问题。

实现

1. 为什么所有样品值都是float64数据类型？我想要integer数据类型。

> 我们限制了float64以简化设计,IEEE 754双精度二进制浮点格式支持高达253的值的整数精度。如果您需要高于253但低于263的整数精度，支持本地64位整数将有帮助。原则上，支持不同的样本值类型（包括某种大整数，支持甚至超过64位）可以实现，但它现在不是一个优先级。 注意，一个计数器，即使每秒增加100万次，只有在超过285年后才会出现精度问题。

2. 为什么Prometheus使用自定义的存储后端，而不是使用其他的存储方法？是不是“一个时间序列一个文件”会大大地伤害性能？

> 一开始，Prometheus是在LevelDB上存储事件序列数据，但不能达到比较好的性能，我们必须改变大量时间序列的存储方式。我们评估了当时可用的许多存储系统，但是没有得到满意的结果。所以我们实现了我们需要的部分。同时保持LevelDB的索引和大量使用文件系统功能。我们最重要的要求是对于常见查询的可接受查询速度，以及每秒数千个样本的可持续速率。后者取决于样本数据的可压缩性和样本所属的时间序列数，但是给你一个想法，这里有一些基准的结果：

- 在具有Intel Core i7 CPU，8GiB RAM和两个旋转磁盘（三星HD753LJ）的老式8核机器上，Prometheus在每个RAID-1设置中的吞吐速率为34k样本，属于170k时间序列， 600个目标。
- 在具有64GiB RAM，32个CPU内核和SSD的现代服务器上，Prometheus的每秒吞吐率为525k样本，属于1.4M时间序列，从1650个目标中剔除。

在这两种情况下，没有明显的瓶颈。在相同的流入速度下，各个阶段的处理管道或多或少都会达到他们的限度。在通常的设置中，不可能使用inode。 有一个可能的缺点：如果你想删除Prometheus的存储目录，你会注意到，一些文件系统在删除文件时非常慢。

3. 为什么Prometheus服务器组件不支持TLS或身份验证？ 我可以添加这些吗？

> 虽然TLS和身份验证是经常请求的功能，但我们有意不在Prometheus的任何服务器端组件中实现它们。 有这么多不同的选项和参数（仅限TLS的10多个选项），我们决定专注于建立最佳的监控系统，而不是在每个服务器组件中支持完全通用的TLS和身份验证解决方案。

如果您需要TLS或身份验证，我们建议将反向代理放在Prometheus前面。 参见例如使用Nginx添加对Prometheus的基本认证。

请注意，这仅适用于入站连接。 Prometheus确实支持删除TLS-和auth启用的目标，以及其他创建出站连接的

Prometheus组件具有类似的支持。

路线图

路线路

下面的一些功能使我们即将要做的事情。如果你想查看整个计划和当前工作的完整概述，请查看github上Prometheus项目的issue，如：Prometheus服务

新的存储引擎

现在一个新的，更加高效的存储引擎正在开发中。它将减少资源的使用率，更好的倒排索引，并且可以使Prometheus更好地扩展

长期存储

当前Prometheus支持本地存储样本数据，同时也有一些实验性地支持：通过一个通用机制，发送数据到远程系统。例如：TSDBs
我们计划通过Prometheus的通用机制（如：Cortex）添加来自其他TSDB的回读支持。Github issue：#10

改进陈旧性处理

当前对于一个表达式的查询结果时间超过5分钟后，Prometheus会丢弃结果中的时间序列数据，言外之意是说Prometheus当前只保存5分钟内的查询结果。目前禁止使用Pushgateway和CloudWatch导出时间序列数据，因为它可能表示超过过去5分钟的时间序列，这是不准确的查询结果。如果近期不发生时间序列数据的抓取操作，我们计划仅仅考虑时间序列的无效性。github issue：#398

服务端度量指标元数据支持

现在度量指标类型和其他元数据仅仅在客户库和展示格式中使用，并不会在Prometheus服务中持久保留或者利用。将来我们计划充分利用这些元数据。第一步是在Prometheus服务的内存中聚合这些数据，并开放一些实验性的API来提供服务

Prometheus度量指标格式作为一个标准

我们打算提交一个标准化的干净版本给IETF等组织。

回填时间序列

回填时间序列的含义是将过去大量的时间序列数据，根据一定的回溯规则，传输到其他的监控系统中。

支持生态

Prometheus有大量的客户库和导出数据器。也有大量的语言被支持，或者有一些可以从Prometheus服务中导出的时间序列系统。我们将会为这个生态做更多的创建和扩展，丰富这个生态。

学习媒介

媒体

在网络上，有一些与Prometheus相关的资源链接
对于了解Prometheus，下面这些资源的选择对你会有非常大的帮助

博客

- Prometheus有自己的博客
- SoundCloud的博客帖子宣称-比上面的博客更加详尽地描述Prometheus
- 在Robust Perception的博客上有很多与Prometheus相关的资源

教程

- Prometheus工作室的介绍和用例
- 在Ubuntu 14.04上使用Docker安装Prometheus

播客与采访

- Prometheus on FLOSS Weekly 357 - Julius Volz on the FLOSS Weekly TWiT.tv show.
- Prometheus与服务监控-Julius Volz on the Changelog podcast.

谈话记录

- Prometheus: 下一代监控系统 – Julius Volz and Björn Rabenstein at SREcon15 Europe, Dublin.
- Prometheus: 下一代监控系统-Brian Brazil at FOSDEM 2016 (slides).
- In German: [Monitoring mit Prometheus](In German: Monitoring mit Prometheus – Michael Stapelberg at Easterhegg 2016.) – Michael Stapelberg at Easterhegg 2016.
- In German: Prometheus in der Praxis – Jonas Große Sundrup at MRMCD 2016

PPT

通用

- Prometheus总览- by Brian Brazil.
- Prometheus监控系统 - Brian Brazil at Devops Ireland Meetup, Dublin.
- OMG! Prometheus Benjamin Staffin, Fitbit Site Operations, explains the case for Prometheus to his team.

Docker

- Prometheus与Docker - Brian Brazil at Docker Galway.

Python

- python版本更好的监控– Brian Brazil at Pycon Ireland.
- Monitoring your Python with Prometheus - Brian Brazil at Python Ireland Meetup, Dublin.

词汇

词汇表

Alert(警告)

警告是Prometheus服务正在激活警报规则的结果。警报将数据从Prometheus服务发送到警告管理器

(Alertmanager)警告管理器

警告管理器接收警告，并把它们聚合成组、去重复数据、应用静默和节流，然后发送通知到邮件、Pageduty或者Slack等系统中

(Bridge)网桥

网桥是一个从客户端库提取样本，然后将其暴露给非Prometheus监控系统的组件。例如：Python客户端可以将度量指标数据导出到Graphite。

(Client library)客户库

客户库是使用某种语言（Go、Java、Python、Ruby等），可以轻松直接调试代码，编写样本收集器去拉取来自其他系统的数据，并将这些度量指标数据输送给Prometheus服务。

(Collector) 收集器

收集器是表示一组度量指标导出器的一部分。它可以是单个度量指标，也可以是从另一个系统提取的多维度度量指标。

(Direct instrumentation)直接测量

直接测量是将测量在线添加到程序的代码中

(Exporter)导出器

导出器是暴露Prometheus度量指标的二进制文件，通常将非Prometheus数据格式转化为Prometheus支持的数据处理格式

(Notification)通知

通知表示一组或者多组的警告，通过警告管理器将通知发送到邮件，Pagerduty或者Slack等系统中

(PromDash) 面板

PromDash是Prometheus的Ruby-on-rails主控面板构建器。它和Grafana有高度的相似之处，但是它只能为Prometheus服务

Prometheus

Prometheus经常称作Prometheus系统的核心二进制文件。它也可以作为一个整体，被称作Prometheus监控系统

(PromQL) Prometheus查询语言

PromQL是Prometheus查询语言。它支持聚合、分片、切割、预测和连接操作

Pushgateway

Pushgateway会保留最近从批处理作业中推送的度量指标。这允许服务中断后Prometheus能够抓取它们的度量指标数据

Silence

在AlertManager中的静默可以阻止符合标签的警告通知

Target

在Prometheus服务中，一个应用程序、服务、端点的度量指标数据

概念

数据模型

数据模型

Prometheus从根本上存储的所有数据都是时间序列: 具有时间戳的数据流只属于单个度量指标和该度量指标下的多个标签维度。除了存储时间序列数据外，Prometheus也可以利用查询表达式存储5分钟的返回结果中的时间序列数据

metrics和labels(度量指标名称和标签)

每一个时间序列数据由metric度量指标名称和它的标签labels键值对集合唯一确定。
这个metric度量指标名称指定监控目标系统的测量特征（如： `http_requests_total` - 接收http请求的总计数）. metric度量指标命名ASCII字母、数字、下划线和冒号，他必须配正则表达式 `[a-zA-Z_:][a-zA-Z0-9_:]*` 。
标签开启了Prometheus的多维数据模型：对于相同的度量名称，通过不同标签列表的结合, 会形成特定的度量维度实例。(例如：所有包含度量名称为 `/api/tracks` 的http请求，打上 `method=POST` 的标签，则形成了具体的http请求)。这个查询语言在这些度量和标签列表的基础上进行过滤和聚合。改变任何度量上的任何标签值，则会形成新的时间序列图
标签label名称可以包含ASCII字母、数字和下划线。它们必须匹配正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*` 。带有 `_` 下划线的标签名称被保留内部使用。
标签labels值包含任意的Unicode码。
具体详见metrics和labels命名最佳实践。

有序的采样值

有序的采样值形成了实际的时间序列数据列表。每个采样值包括：

- 一个64位的浮点值
- 一个精确到毫秒级的时间戳
  一个样本数据集是针对一个指定的时间序列在一定时间范围的数据收集。这个时间序列是由<metric_name>{<label_name>=<label_value>, ...}

''小结：指定度量名称和度量指标下的相关标签值，则确定了所关心的目标数据，随着时间推移形成一个个点，在图表上实时绘制动态变化的线条''

Notation(符号)

表示一个度量指标和一组键值对标签，需要使用以下符号：

> [metric name]{[label name]=[label value], ...}

例如，度量指标名称是 `api_http_requests_total` ，标签为 `method="POST"` ，`handler="/messages"` 的示例如下所示：

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

这些命名和OpenTSDB使用方法是一样的

度量指标类型

metrics类型

Prometheus客户库提供了四个核心的metrics类型。这四种类型目前仅在客户库和wire协议中区分。Prometheus服务还没有充分利用这些类型。不久的将来就会发生改变。

Counter(计数器)

counter 是一个累计度量指标，它是一个只能递增的数值。计数器主要用于统计服务的请求数、任务完成数和错误出现的次数等等。计数器是一个递增的值。反例：统计goroutines的数量。计数器的使用方式在下面的各个客户端例子中：
客户端使用计数器的文档：

- Go
- Java
- Python
- Ruby

Gauge(测量器)

gauge是一个度量指标，它表示一个既可以递增, 又可以递减的值。
测量器主要测量类似于温度、当前内存使用量等，也可以统计当前服务运行随时增加或者减少的Goroutines数量
客户端使用计量器的文档：

- Go
- Java
- Python
- Ruby

Histogram(柱状图)

histogram，是柱状图，在Prometheus系统中的查询语言中，有三种作用：

1. 对每个采样点进行统计，打到各个分类值中(bucket)
2. 对每个采样点值累计和(sum)
3. 对采样点的次数累计和(count)

度量指标名称: `[basename]` 的柱状图, 上面三类的作用度量指标名称

- [basename]_bucket{le="上边界"}, 这个值为小于等于上边界的所有采样点数量

- [basename]_sum
- [basename]_count

小结：所以如果定义一个度量类型为Histogram，则Prometheus系统会自动生成三个对应的指标

*histogram的最简单的理解, DEMO

使用histogram_quantile()函数, 计算直方图或者是直方图聚合计算的分位数阈值。 一个直方图计算Apdex值也是合适的, 当在buckets上操作时，记住直方图是累计的。详见直方图和总结

客户库的直方图使用文档：

- Go
- Java
- Python
- Ruby

[Summary]总结

类似histogram柱状图，summary是采样点分位图统计，(通常的使用场景：请求持续时间和响应大小)。 它也有三种作用：

1. 对于每个采样点进行统计，并形成分位图。（如：正态分布一样，统计低于60分不及格的同学比例，统计低于80分的同学比例，统计低于95分的同学比例）
2. 统计班上所有同学的总成绩(sum)
3. 统计班上同学的考试总人数(count)

带有度量指标的 `[basename]` 的 `summary` 在抓取时间序列数据展示。

- 观察时间的φ-quantiles (0 ≤ φ ≤ 1), 显示为 `[basename]{分位数="[φ]"}`
- `[basename]_sum` ， 是指所有观察值的总和
- `[basename]_count` ，是指已观察到的事件计数值

*summary的最简单的理解, DEMO

详见histogram和summaries

有关 `summaries` 的客户端使用文档：

- Go
- Java
- Python
- Ruby

任务与实例

Jobs和Instances(任务和实例)

就Prometheus而言，pull拉取采样点的端点服务称之为instance。多个这样pull拉取采样点的instance, 则构成了一个job
例如, 一个被称作api-server的任务有四个相同的实例。

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

自动化生成的标签和时间序列

当Prometheus拉取一个目标, 会自动地把两个标签添加到度量名称的标签列表中，分别是：

- job: 目标所属的配置任务名称api-server。
- instance: 采样点所在服务: `host:port`

如果以上两个标签二者之一存在于采样点中，这个取决于 `honor_labels` 配置选项。详见文档
对于每个采样点所在服务instance，Prometheus都会存储以下的度量指标采样点：

- `up{job="[job-name]", instance="instance-id"}` ：up值=1，表示采样点所在服务健康; 否则，网络不通, 或者服务挂掉了
- `scrape_duration_seconds{job="[job-name]", instance="[instance-id]"}` ：尝试获取目前采样点的时间开销
- `scrape_samples_post_metric_relabeling{job="<job-name>", instance="<instance-id` ：表示度量指标的标签变化后，标签没有变化的度量指标数量。
- `scrape_samples_scraped{job="<job-name>", instance="<instance-id>"}` ：这个采样点目标暴露的样本点数量

备注：我查了下 `scrape_samples_post_metric_relabeling` 和 `scrape_samples_scraped` 的值好像是一样的。还是这两个值没有理解
`up` 度量指标对服务健康的监控是非常有用的。

prometheus

查询

基本概念

Prometheus查询

Prometheus提供一个函数式的表达式语言，可以使用户实时地查找和聚合时间序列数据。表达式计算结果可以在图表中展示，也可以在Prometheus表达式浏览器中以表格形式展示，或者作为数据源, 以HTTP API的方式提供给外部系统使用。

examples

这个文档仅供参考, 这里先举几个容易上手的例子。

表达式语言数据类型

在Prometheus的表达式语言中，任何表达式或者子表达式都可以归为四种类型：

- `instant vector` 瞬时向量 - 它是指在同一时刻，抓取的所有度量指标数据。这些度量指标数据的key都是相同的，也即相同的时间戳。
- `range vector` 范围向量 - 它是指在任何一个时间范围内，抓取的所有度量指标数据。
- `scalar` 标量 - 一个简单的浮点值
- `string` 字符串 - 一个当前没有被使用的简单字符串

依赖于使用场景（例如：图表 vs. 表格），根据用户所写的表达式，仅仅只有一部分类型才适用这种表达式。例如：瞬时向量类型是唯一可以直接在图表中使用的。

Literals

字符串

字符串可以用单引号、双引号或者反引号表示
PromQL遵循与Go相同的转义规则。在单引号，双引号中，反斜杠成为了转义字符，后面可以跟着a, b, f, n, r, t, v或者\。 可以使用八进制(\nnn)或者十六进制(\xnn, \unnnn和\Unnnnnnnn)提供特定字符。
在反引号内不处理转义字符。与Go不同，Prom不会丢弃反引号中的换行符。例如：

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

浮点数

标量浮点值可以直接写成形式-[.(digits)]。

> -2.43

时间序列选择器

即时向量选择器

瞬时向量选择器可以对一组时间序列数据进行筛选，并给出结果中的每个结果键值对（时间戳-样本值）：最简单的形式是，只有一个度量名称被指定。在一个瞬时向量中这个结果包含有这个度量指标名称的所有样本数据键值对。
下面这个例子选择所有时间序列度量名称为 `http_requests_total` 的样本数据：

> http_requests_total

通过在度量指标后面增加{}一组标签可以进一步地过滤这些时间序列数据。
下面这个例子选择了度量指标名称为 `http_requests_total` ，且一组标签为 `job=prometheus` ，`group=canary` ：

> http_requests_total{job="prometheus",group="canary"}

可以采用不匹配的标签值也是可以的，或者用正则表达式不匹配标签。标签匹配操作如下所示：

- `=` ：精确地匹配标签给定的值
- `!=` ：不等于给定的标签值
- `=~` ：正则表达匹配给定的标签值
- `!=` ：给定的标签值不符合正则表达式

例如：度量指标名称为 `http_requests_total` ，正则表达式匹配标签 `environment` 为 `staging, testing, development` 的值，且http请求方法不等于 `GET` 。

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

匹配空标签值的标签匹配器也可以选择没有设置任何标签的所有时间序列数据。正则表达式完全匹配。
向量选择器必须指定一个度量指标名称或者至少不能为空字符串的标签值。以下表达式是非法的:

> {job=~".*"} #Bad!

上面这个例子既没有度量指标名称，标签选择器也可以正则匹配空标签值，所以不符合向量选择器的条件
相反地，下面这些表达式是有效的，第一个一定有一个字符。第二个有一个有用的标签method

> {job=~".+"} # Good!
> {job=~".*", method="get"} # Good!

标签匹配器能够被应用到度量指标名称，使用 `__name__` 标签筛选度量指标名称。例如：表达式 `http_requests_total` 等价于 `{__name__="http_requests_total"}` 。 其他的匹配器，如：`= ( !=, =~, !~)` 都可以使用。下面的表达式选择了度量指标名称以 `job:` 开头的时间序列数据：

> {name=~"^job:.*"} #

范围向量选择器

范围向量类似瞬时向量, 不同在于，它们从当前实例选择样本范围区间。在语法上，时间长度被追加在向量选择器尾部的方括号[]中，用以指定对于每个样本范围区间中的每个元素应该抓取的时间范围样本区间。
时间长度有一个数值决定，后面可以跟下面的单位：

- `s` - seconds
- `m` - minutes
- `h` - hours
- `d` - days
- `w` - weeks
- `y` - years

在下面这个例子中, 选择过去5分钟内，度量指标名称为 `http_requests_total` ， 标签为 `job="prometheus"` 的时间序列数据:

> http_requests_total{job="prometheus"}[5m]

偏移修饰符

这个 `offset` 偏移修饰符允许在查询中改变单个瞬时向量和范围向量中的时间偏移
例如，下面的表达式返回相对于当前时间的前5分钟时的时刻, 度量指标名称为 `http_requests_total` 的时间序列数据：

> http_requests_total offset 5m

注意： `offset` 偏移修饰符必须直接跟在选择器后面，例如：

> sum(http_requests_total{method="GET"} offset 5m) // GOOD.

然而，下面这种情况是不正确的

> sum(http_requests_total{method="GET"}) offset 5m // INVALID.

offset偏移修饰符在范围向量上和瞬时向量用法一样的。下面这个返回了相对于当前时间的前一周时，过去5分钟

的度量指标名称为 `http_requests_total` 的速率：

> rate(http_requests_total[5m] offset 1w)

操作符

Prometheus支持二元和聚合操作符。详见表达式语言操作符

函数

Prometheus提供了一些函数列表操作时间序列数据。详见表达式语言函数

陷阱

插值和陈旧

当运行查询后，独立于当前时刻被选中的时间序列数据所对应的时间戳，这个时间戳主要用来进行聚合操作，包括 `sum` ， `avg` 等，大多数聚合的时间序列数据所对应的时间戳没有对齐。由于它们的独立性，我们需要在这些时间戳中选择一个时间戳，并已这个时间戳为基准，获取小于且最接近这个时间戳的时间序列数据。
如果5分钟内，没有获取到任何的时间序列数据，则这个时间戳不会存在。那么在图表中看到的数据都是在当前时刻5分钟前的数据。

```
注意：差值和陈旧处理可能会发生变化。详见[https://github.com/prometheus/prometheus/issues/398](https://github.com/prometheus/prometheus/issues/398)和[https://github.com/prometheus/prometheus/issues/581](https://github.com/prometheus/prometheus/issues/581)

#### 避免慢查询和高负载
如果一个查询需要操作非常大的数据量，图表绘制很可能会超时，或者服务器负载过高。因此，在对未知数据构建查询时，始终需要在Prometheus表达式浏览器的表格视图中构建查询，直到结果是看起来合理的（最多为数百个，而不是数千个）。只有当你已经充分过滤或者聚合数据时，才切换到图表模式。如果表达式的查询结果仍然需要很长时间才能绘制出来，则需要通过记录规则重新清洗数据。

像`api_http_requests_total`这样简单的度量指标名称选择器，可以扩展到具有不同标签的数千个时间序列中，这对于Prometheus的查询语言是非常重要的。还要记住，聚合操作即使输出的结果集非常少，但是它会在服务器上产生负载。这类似于关系型数据库查询可一个字段的总和，总是非常缓慢。
```

操作符

操作符

二元操作符

Prometheus的查询语言支持基本的逻辑运算和算术运算。对于两个瞬时向量, 匹配行为可以被改变。

算术二元运算符

在Prometheus系统中支持下面的二元算术操作符：

- `+` 加法
- `-` 减法
- `*` 乘法
- `/` 除法
- `%` 模
- `^` 幂等

二元运算操作符支持 `scalar/scalar(标量/标量)` 、 `vector/scalar(向量/标量)` 、和 `vector/vector(向量/向量)` 之间的操作。
在两个标量之间进行操作符运算，得到的结果也是标量
在向量和标量之间，这个操作符会作用于这个向量的每个样本值上。例如：如果一个时间序列瞬时向量除以2，操作结果也是一个新的瞬时向量，且度量指标名称不变, 它是原度量指标瞬时向量的每个样本值除以2.
在两个向量之间，一个二元算术操作符作用在左边瞬时向量的每个样本值，且该样本值与操作符右边能匹配上的样本值计算，向量匹配。结果写入到一个没有度量指标名称的瞬时向量。

比较二元操作符

在Prometheus系统中，比较二元操作符有：

- `==` 等于
- `!=` 不等于
- `>` 大于
- `<` 小于
- `>=` 大于等于
- `<=` 小于等于

比较二元操作符被应用于 `scalar/scalar（标量/标量）` 、 `vector/scalar(向量/标量)` ，和 `vector/vector（向量/向量）` 。比较操作符得到的结果是 `bool` 布尔类型值，返回1或者0值。

在两个标量之间的比较运算，bool结果写入到另一个结果标量中。
瞬时向量和标量之间的比较运算，这个操作符会应用到某个当前时刻的每个时间序列数据上，如果一个时间序列数据值与这个标量比较结果是 `false` ，则这个时间序列数据被丢弃掉，如果是 `true` ，则这个时间序列数据被保留在结果中。
在两个瞬时向量之间的比较运算，左边度量指标数据中的每个时间序列数据，与右边度量指标中的每个时间序列数据匹配，没有匹配上的，或者计算结果为false的，都被丢弃，不在结果中显示。否则将保留左边的度量指标和标签的样本数据写入瞬时向量。

逻辑/集合二元操作符

逻辑/集合二元操作符只能作用在即时向量， 包括：

- `and` 交集
- `or` 并集
- `unless` 补集

`vector1 and vector2` 的逻辑/集合二元操作符，规则： `vector1` 瞬时向量中的每个样本数据与 `vector2` 向量中的所有样本数据进行标签匹配，不匹配的，全部丢弃。运算结果是保留左边的度量指标名称和值。
`vector1 or vector2` 的逻辑/集合二元操作符，规则: 保留 `vector1` 向量中的每一个元素，对于 `vector2` 向量元素，则不匹配 `vector1` 向量的任何元素，则追加到结果元素中。
`vector1 unless vector2` 的逻辑/集合二元操作符，又称差积。规则：包含在 `vector1` 中的元素，但是该元素不在 `vector2` 向量所有元素列表中，则写入到结果集中。

向量匹配

向量之间的匹配是指右边向量中的每一个元素，在左边向量中也存在。这里有两种基本匹配行为特征：

- 一对一，找到这个操作符的两边向量元素的相同元素。默认情况下，操作符的格式是 `vector1 [operate] vector2` 。如果它们有相同的标签和值，则表示相匹配。 `ingoring` 关键字是指，向量匹配时，可以忽略指定标签。 `on` 关键字是指，在指定标签上进行匹配。格式如下所示：

> [vector expr] [bin-op] ignoring([label list]) [vector expr]

> [vector expr] [bin-op] on([lable list]) [vector expr]

例如样本数据：

```
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="404"} 21

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

查询例子：

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

两个向量之间的除法操作运算的向量结果是，每一个向量样本http请求方法标签值是500，且在过去5分钟的运算值。如果没有忽略 `code="500"` 的标签，这里不能匹配到向量样本数据。两个向量的请求方法是 `put` 和 `delete` 的样本数据不会出现在结果列表中

> {method="get"} 0.04 // 24 / 600

> {method="post"} 0.05 // 6 / 120

多对一和一对多的匹配，是指向量元素中的一个样本数据匹配标签到了多个样本数据标签。这里必须直接指定两个修饰符 `group_left` 或者 `group_right` ，左或者右决定了哪边的向量具有较高的子集。

> <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>

这个group带标签的修饰符标签列表包含了“一对多”中的“一”一侧的额外标签。对于 `on` 标签只能是这些列表中的一个。结果向量中的每一个时间序列数据都是唯一的。
`group` 修饰符只能被用在比较操作符和算术运算符。
查询例子：

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

在这个例子中，左向量的标签数量多于左边向量的标签数量，所以我们使用 `group_left` 。右边向量的时间序列元素匹配左边的所有相同 `method` 标签:

> {method="get", code="500"} 0.04 // 24 /600

> {method="get", code="404"} 0.05 // 30 /600

> {method="post", code="500"} 0.05 // 6 /600

> {method="post", code="404"} 0.175 // 21 /600

多对一和一对多匹配应该更多地被谨慎使用。经常使用 `ignoring(\<labels\>)` 输出想要的结果。

聚合操作符

Prometheus支持下面的内置聚合操作符。这些聚合操作符被用于聚合单个即时向量的所有时间序列列表，把聚合的结果值存入到新的向量中。

- `sum` (在维度上求和)
- `max` (在维度上求最大值)
- `min` (在维度上求最小值)
- `avg` (在维度上求平均值)
- `stddev` (求标准差)
- `stdvar` (求方差)
- `count` (统计向量元素的个数)
- `count_values` (统计相同数据值的元素数量)
- `bottomk` (样本值第k个最小值)
- `topk` (样本值第k个最大值)
- `quantile` (统计分位数)

这些操作符被用于聚合所有标签维度，或者通过 `without` 或者 `by` 子句来保留不同的维度。

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

`parameter` 只能用于 `count_values` ， `quantile` ， `topk` 和 `bottomk` 。 `without` 移除结果向量中的标签集合，其他标签被保留输出。 `by` 关键字的作用正好相反，即使它们的标签值在向量的所有元素之间。 `keep_common` 子句允许保留额外的标签（在元素之间相同，但不在by子句中的标签）
`count_values` 对每个唯一的样本值输出一个时间序列。每个时间序列都附加一个标签。这个标签的名字有聚合参数指定，同时这个标签值是唯一的样本值。每一个时间序列值是结果样本值出现的次数。
`topk` 和 `bottomk` 与其他输入样本子集聚合不同，返回的结果中包括原始标签。 `by` 和 `without` 仅仅用在输入向量的桶中
例如：
如果度量指标名称 `http_requests_total` 包含由 `group` ， `application` ， `instance` 的标签组成

的时间序列数据，我们可以通过以下方式计算去除 `instance` 标签的http请求总数：

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

如果我们对所有应用程序的http请求总数，我们可以简单地写下：

```
sum(http_requests_total)
```

统计每个编译版本的二进制文件数量，我们可以如下写：

```
count_values("version", build_version)
```

通过所有实例，获取http请求第5个最大值，我们可以简单地写下：

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

二元运算符优先级

在Prometheus系统中，二元运算符优先级从高到低：

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

函数

函数列表

一些函数有默认的参数，例如： `year(v=vector(time()) instant-vector)` 。v是参数值，instant-vector是参数类型。vector(time())是默认值。

abs()

`abs(v instant-vector)` 返回输入向量的所有样本的绝对值。

absent()

`absent(v instant-vector)` ，如果赋值给它的向量具有样本数据，则返回空向量；如果传递的瞬时向量参数没有样本数据，则返回不带度量指标名称且带有标签的样本值为1的结果
当监控度量指标时，如果获取到的样本数据是空的， 使用absent方法对告警是非常有用的

> absent(nonexistent{job="myjob"}) # => key: value = {job="myjob"}: 1

> absent(nonexistent{job="myjob", instance=~".*"}) # => {job="myjob"} 1
> so smart !

> absent(sum(nonexistent{job="myjob"})) # => key:value {}: 0

ceil()

`ceil(v instant-vector)` 是一个向上舍入为最接近的整数。

changes()

`changes(v range-vector)` 输入一个范围向量， 返回这个范围向量内每个样本数据值变化的次数。

clamp_max()

`clamp_max(v instant-vector, max scalar)` 函数，输入一个瞬时向量和最大值，样本数据值若大于max，则改为max，否则不变

clamp_min()

`clamp_min(v instant-vector)` 函数，输入一个瞬时向量和最大值，样本数据值小于min，则改为min。否则不变

count_saclar()

`count_scalar(v instant-vector)` 函数, 输入一个瞬时向量，返回key:value="scalar": 样本个数。而 `count()` 函数，输入一个瞬时向量，返回key:value=向量：样本个数，其中结果中的向量允许通过 `by` 条件分组。

day_of_month()

`day_of_month(v=vector(time()) instant-vector)` 函数，返回被给定UTC时间所在月的第几天。返回值范围：1~31。

day_of_week()

`day_of_week(v=vector(time()) instant-vector)` 函数，返回被给定UTC时间所在周的第几天。返回值范围：0~6. 0表示星期天。

days_in_month()

`days_in_month(v=vector(time()) instant-vector)` 函数，返回当月一共有多少天。返回值范围：28~31.

delta()

`delta(v range-vector)` 函数，计算一个范围向量v的第一个元素和最后一个元素之间的差值。返回值：key:value=度量指标：差值
下面这个表达式例子，返回过去两小时的CPU温度差：

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

`delta` 函数返回值类型只能是gauges。

deriv()

`deriv(v range-vector)` 函数，计算一个范围向量v中各个时间序列二阶导数，使用简单线性回归 `deriv` 二阶导数返回值类型只能是gauges。

drop_common_labels()

`drop_common_labels(instant-vector)` 函数，输入一个瞬时向量，返回值是key:value=度量指标：样本值，其中度量指标是去掉了具有相同标签。
例如：http_requests_total{code="200", host="127.0.0.1:9090", method="get"} : 4, http_requests_total{code="200", host="127.0.0.1:9090", method="post"} : 5, 返回值: http_requests_total{method="get"} : 4, http_requests_total{code="200", method="post"} : 5

exp()

`exp(v instant-vector)` 函数，输入一个瞬时向量, 返回各个样本值的e指数值，即为e^N次方。特殊情况如下所示：

> Exp(+inf) = +Inf
> Exp(NaN) = NaN

floor()

`floor(v instant-vector)` 函数，与 `ceil()` 函数相反。 4.3 为 4 。

histogram_quantile()

`histogram_quatile(φ float, b instant-vector)` 函数计算b向量的φ-直方图 (0 ≤ φ ≤ 1) 。参考中文文献[https://www.howtoing.com/how-to-query-prometheus-on-ubuntu-14-04-part-2/]

holt_winters()

`holt_winters(v range-vector, sf scalar, tf scalar)` 函数基于范围向量v，生成事件序列数据平滑值。平滑因子 `sf` 越低, 对老数据越重要。趋势因子 `tf` 越高，越多的数据趋势应该被重视。0< sf, tf <=1。 `holt_winters` 仅用于gauges

hour()

`hour(v=vector(time()) instant-vector)` 函数返回被给定UTC时间的当前第几个小时，时间范围：0~23。

idelta()

`idelta(v range-vector)` 函数，输入一个范围向量，返回key: value = 度量指标： 每最后两个样本值差值。

increase()

`increase(v range-vector)` 函数， 输入一个范围向量，返回：key:value = 度量指标：last值-first值，自动调整单调性，如：服务实例重启，则计数器重置。与 `delta()` 不同之处在于delta是求差值，而increase返回最后一个减第一个值,可为正为负。
下面的表达式例子，返回过去5分钟，连续两个时间序列数据样本值的http请求增加值。

> increase(http_requests_total{job="api-server"}[5m])

`increase` 的返回值类型只能是counters，主要作用是增加图表和数据的可读性，使用 `rate` 记录规则的使用率，以便持续跟踪数据样本值的变化。

irate

`irate(v range-vector)` 函数, 输入：范围向量，输出：key: value = 度量指标： (last值-last前一个值)/时间戳差值。它是基于最后两个数据点，自动调整单调性， 如：服务实例重启，则计数器重置。
下面表达式针对范围向量中的每个时间序列数据，返回两个最新数据点过去5分钟的HTTP请求速率。

```
irate(http_requests_total{job="api-server"}[5m])
```

`irate` 只能用于绘制快速移动的计数器。因为速率的简单更改可以重置FOR子句，利用警报和缓慢移动的计数器，完全由罕见的尖峰组成的图形很难阅读。

label_replace()

对于v中的每个时间序列，
`label_replace(v instant-vector, dst_label string, replacement string, src_label s`
将正则表达式与标签值src_label匹配。如果匹配，则返回时间序列，标签值dst_label被替换的扩展替换。$1替换为第一个匹配子组，$2替换为第二个等。如果正则表达式不匹配，则时间序列不会更改。
另一种更容易的理解是： `label_replace` 函数，输入：瞬时向量，输出：key: value = 度量指标： 值（要替换的内容：首先，针对src_label标签，对该标签值进行regex正则表达式匹配。如果不能匹配的度量指标，则不发生任何改变；否则，如果匹配，则把dst_label标签的标签纸替换为replacement
下面这个例子返回一个向量值a带有 `foo` 标签：
`label_replace(up{job="api-server", serice="a:c"}, "foo", "$1", "service", "(.*):.`

ln()

`ln(v instance-vector)` 计算瞬时向量v中所有样本数据的自然对数。特殊例子：

```
ln(+Inf) = +Inf
ln(0) = -Inf
ln(x<0) = NaN
ln(NaN) = NaN
```

log2()

`log2(v instant-vector)` 函数计算瞬时向量v中所有样本数据的二进制对数。

log10()

`log10(v instant-vector)` 函数计算瞬时向量v中所有样本数据的10进制对数。相当于ln()

minute()

`minute(v=vector(time()) instant-vector)` 函数返回给定UTC时间当前小时的第多少分钟。结果范围：0~59。

month()

`month(v=vector(time()) instant-vector)` 函数返回给定UTC时间当前属于第几个月，结果范围：0~12。

predict_linear()

`predict_linear(v range-vector, t scalar)` 预测函数，输入：范围向量和从现在起t秒后，输出：不带有度量指标，只有标签列表的结果值。

```
例如：predict_linear(http_requests_total{code="200",instance="120.77.65.193:9090
",job="prometheus",method="get"}[5m], 5)
结果：
{code="200",handler="query_range",instance="120.77.65.193:9090",job="prometheus
",method="get"}  1
{code="200",handler="prometheus",instance="120.77.65.193:9090",job="prometheus"
,method="get"}   4283.449995397104
{code="200",handler="static",instance="120.77.65.193:9090",job="prometheus",met
hod="get"}   22.99999999999999
{code="200",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    130.90381188596754
{code="200",handler="graph",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    2
{code="200",handler="label_values",instance="120.77.65.193:9090",job="prometheu
s",method="get"} 2
```

rate()

`rate(v range-vector)` 函数, 输入：范围向量，输出：key: value = 不带有度量指标，且只有标签列表：(last值-first值)/时间差s

```
rate(http_requests_total[5m])
结果：
{code="200",handler="label_values",instance="120.77.65.193:9090",job="prometheu
s",method="get"} 0
{code="200",handler="query_range",instance="120.77.65.193:9090",job="prometheus
",method="get"}  0
{code="200",handler="prometheus",instance="120.77.65.193:9090",job="prometheus"
,method="get"}   0.2
{code="200",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0.003389830508474576
{code="422",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
{code="200",handler="static",instance="120.77.65.193:9090",job="prometheus",met
hod="get"}   0
{code="200",handler="graph",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
```

```
{code="400",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
```

`rate()` 函数返回值类型只能用counters， 当用图表显示增长缓慢的样本数据时，这个函数是非常合适的。注意：当rate函数和聚合方式联合使用时，一般先使用rate函数，再使用聚合操作, 否则，当服务实例重启后，rate无法检测到counter重置。

resets()

`resets()` 函数, 输入：一个范围向量，输出：key-value=没有度量指标，且有标签列表[在这个范围向量中每个度量指标被重置的次数]。在两个连续样本数据值下降，也可以理解为counter被重置。
示例：

```
resets(http_requests_total[5m])
结果：
{code="200",handler="label_values",instance="120.77.65.193:9090",job="prometheu
s",method="get"} 0
{code="200",handler="query_range",instance="120.77.65.193:9090",job="prometheus
",method="get"}  0
{code="200",handler="prometheus",instance="120.77.65.193:9090",job="prometheus"
,method="get"}   0
{code="200",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
{code="422",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
{code="200",handler="static",instance="120.77.65.193:9090",job="prometheus",met
hod="get"}   0
{code="200",handler="graph",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
{code="400",handler="query",instance="120.77.65.193:9090",job="prometheus",meth
od="get"}    0
```

resets只能和counters一起使用。

round()

`round(v instant-vector, to_nearest 1= scalar)` 函数，与 `ceil` 和 `floor` 函数类似，输入：瞬时向量，输出：指定整数级的四舍五入值, 如果不指定，则是1以内的四舍五入。

scalar()

`scalar(v instant-vector)` 函数, 输入：瞬时向量，输出：key: value = "scalar", 样本值[如果度量指标样本数量大于1或者等于0, 则样本值为NaN, 否则，样本值本身]

sort()

`sort(v instant-vector)` 函数，输入：瞬时向量，输出：key: value = 度量指标：样本值[升序排列]

sort_desc()

`sort(v instant-vector` 函数，输入：瞬时向量，输出：key: value = 度量指标：样本值[降序排列]

sqrt()

`sqrt(v instant-vector)` 函数，输入：瞬时向量，输出：key: value = 度量指标： 样本值的平方根

time()

`time()` 函数，返回从1970-01-01到现在的秒数，注意：它不是直接返回当前时间，而是时间戳

vector()

`vector(s scalar)` 函数，返回：key: value= {}, 传入参数值

year()

`year(v=vector(time()) instant-vector)` ，返回年份。

_over_time()

下面的函数列表允许传入一个范围向量，返回一个带有聚合的瞬时向量:

- `avg_over_time(range-vector)` : 范围向量内每个度量指标的平均值。
- `min_over_time(range-vector)` : 范围向量内每个度量指标的最小值。
- `max_over_time(range-vector)` : 范围向量内每个度量指标的最大值。
- `sum_over_time(range-vector)` : 范围向量内每个度量指标的求和值。
- `count_over_time(range-vector)` : 范围向量内每个度量指标的样本数据个数。
- `quantile_over_time(scalar, range-vector)` : 范围向量内每个度量指标的样本数据值分位数，φ-quantile (0 ≤ φ ≤ 1)
- `stddev_over_time(range-vector)` : 范围向量内每个度量指标的总体标准偏差。
- `stdvar_over_time(range-vector): 范围向量内每个度量指标的总体标准方差。

举例

查询例子

简单的时间序列选择

返回度量指标名称是 `http_requests_total` 的所有时间序列样本数据：

> http_requests_total

返回度量指标名称是 `http_requests_total` ，标签分别是 `job="apiserver` ，`handler="/api/comments` 的所有时间序列样本数据：

> http_requests_total{job="apiserver", hanlder="/api/comments"}

返回度量指标名称是 `http_requests_total` ，标签分别是 `job="apiserver` ，`handler="/api/comments` ，且是5分钟内的所有时间序列样本数据：

> http_requests_total{job="apiserver", handler="/api/comments"}[5m]

注意：一个范围向量表达式结果不能直接在Graph图表中，但是可以在"console"视图中展示。
使用正则表达式，你可以通过特定模式匹配标签为job的特定任务名，获取这些任务的时间序列。在下面这个例子中, 所有任务名称以 `server` 结尾。

> http_requests_total{job=~"server$"}

返回度量指标名称是 `http_requests_total` ， 且http返回码不以4开头的所有时间序列数据：

> http_requests_total{status!~"^4..$"}

使用函数，操作符等

返回度量指标名称 `http_requests_total` ，且过去5分钟的所有时间序列数据值速率。

> rate(http_requests_total[5m])

假设度量名称是 `http_requests_total` ，且过去5分钟的所有时间序列数据的速率和，速率的维度是job

> sum(rate(http_requests_total)[5m]) by (job)

如果我们有相同维度标签，我们可以使用二元操作符计算样本数据，返回值：key: value=标签列表：计算样本

值。例如，下面这个表达式返回每一个实例剩余内存，单位是M, 如果不同，则需要使用 `ignoring(label_lists)` ，如果多对一，则采用group_left, 如果是一对多，则采用group_right。

> (instance_memory_limit_byte - instant_memory_usage_bytes) / 1024 / 1024

相同表达式，求和可以采用下面表达式：

> sum(instance_memory_limit_bytes - instance_memory_usage_bytes) by (app, proc) / 1024 / 1024

如果相同集群调度器任务，显示CPU使用率度量指标的话，如下所示：

> instance_cpu_time_ns{app="lion", pro="web", rev="34d0f99", env="prod", job="cluster-manager"}
> instance_cpu_time_ns{app="elephant", proc="worker", rev="34d0f99", env="prod", job="cluster-manager"}
> instance_cpu_time_ns{app="turtle", proc="api", rev="4d3a513", env="prod", job="cluster-manager"}
> ...

我们可以获取最高的3个CPU使用率，按照标签列表 `app` 和 `proc` 分组

> topk(3, sum(rate(instance_cpu_time_ns[5m])) by(app, proc))

假设一个服务实例只有一个时间序列数据，那么我们通过下面表达式，可以统计出每个应用的实例数量：

> count(instance_cpu_time_ns) by (app)

记录规则

定义recording rules

配置规则

Prometheus支持可以配置，然后定期执行的两种规则: recording rules(记录规则)和alerting rules警告规则。为了在Prometheus系统中包括规则，我们需要创建一个包含规则语句的文件，并通过在Prometheus配置的 `rule_fields` 字段加载这个记录规则文件。
这些规则文件可以通过像Prometheus服务发送 `SIGNUP` 信号量，实时重载记录规则。如果所有的记录规则有正确的格式和语法，则这些变化能够生效。

语法检查规则

在没有启动Prometheus服务之前，想快速知道一个规则文件是否正确，可以通过安装和运行Prometheus的 `promtool` 命令行工具检验:

> go get github.com/prometheus/prometheus/cmd/promtool
> promtool check-rules /path/to/examples.rules

当记录规则文件是有效的，则这个检查会打印出解析到规则的文本表示，并以返回值0退出程序。
如果有任何语法错误的话，则这个命令行会打印出一个错误信息到标准输出，并以返回值1退出程序。无效的输入参数，则以返回值2退出程序。

记录规则

记录规则允许你预先计算经常需要的，或者计算复杂度高的表达式，并将结果保存为一组新的时间序列数据。查询预计算结果通常比需要时进行计算表达式快得多。对于dashboard是非常有用的，因为dashboard需要实时刷新查询表达式的结果。
为了增加一个新记录规则，增加下面的记录规则到你的规则文件中：

> <new time series name>[{<label overrides>}] = <expression to record>

一些例子：

> # 计算每个job的http请求总数，保存到新的度量指标中
>
> job:http_inprogree_requests:sum = sum(http_inprogress_requests) by (job)

> # 放弃老标签，写入新标签的结果时间序列数据：
>
> new_time_series{label_to_change="new_value", label_to_drop=""} = old_time_series

记录规则的执行周期有Prometheus的配置文件中的 `evaluate_interval` 指定。规则语句的右侧表达式一旦被执行，则新的时间戳key为当前时间，value为右边表达式的样本值，新的度量指标名称和标签列表为左边名称。

HttpAPI访问

HTTP API

在Prometheus服务上 `/api/v1` 版本api是稳定版。

格式概述

这个API返回是JSON格式。每个请求成功的返回值都是以 `2xx` 开头的编码。

到达API处理的无效请求，返回一个JSON错误对象，并返回下面的错误码：

- `400 Bad Request` 。当参数错误或者丢失时。
- `422 Unprocessable Entity` 。当一个表达式不能被执行时。
- `503 Service Unavailable` 。当查询超时或者中断时。

在请求到达API之前，其他非 `2xx` 的错误码可能会被返回。

JSON返回格式如下所示：

```
{
"status": "success" | "error",
"data": <data>,

// 如果status是"error", 这个数据字段还会包括下面的数据
"errorType": "<string>",
"error": "<string>"
}
```

输入时间戳可以被RFC3339格式或者Unix时间戳提供。输出时间戳以Unix时间戳的方式呈现。

查询参数名称可以用 `[]` 中括号重复次数。

`<series_selector>` 占位符提供像 `http_requests_total` 或者 `http_requests_total{method=~"^GET|POST$"}` 的Prometheus时间序列选择器，并需要在URL中编码传输。

`<duration>` 占位符涉及到 `[0-9]-[smhdwy]` 。例如： `5m` 表示5分钟的持续时间。

表达式查询

查询语言表达式可以在瞬时向量或者范围向量中执行。

Instant queries(即时查询)

瞬时向量的http restful api查询：

> GET /api/v1/query

URL查询参数：

- `query=<string>`：Prometheus表达式查询字符串。
- `time=<rfc3339 | uninx_timestamp>`：执行时间戳，可选项。
- `timeout=<duration>`：执行超时时间设置，默认由 `-query.timeout` 标志设置

如果 `time` 缺省，则用当前服务器时间表示执行时刻。

这个查询结果的 `data` 部分有下面格式：

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

是一个查询结果数据，依赖于这个 `resultType` 格式, 不同的结果类型，则会有不同的结果数据格式。见表达式查询结果格式。

下面例子执行了在时刻是 `2015-07-01T20:10:51.781Z` 的 `up` 表达式：

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

```
}
```

范围查询

下面评估了一个范围时间的查询表达式：

> GET /api/v1/query_range

URL查询参数

- `query=<string>`：Prometheus表达式查询字符串。
- `start=<rfc3339 | unix_timestamp>`：开始时间戳。
- `end=<rfc3339 | unix_timestamp>`：结束时间戳。
- `step=<duration>`：查询时间步长，范围时间内每step秒执行一次。

下面查询结果格式的 `data` 部分：

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

对于 `<value>` 占位符的格式，详见范围向量结果格式。

下面例子评估的查询条件 `up` ，且30s范围的查询，步长是15s。

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

查询元数据

通过标签匹配器找到度量指标列表

下面例子返回了度量指标列表 且不返回时间序列数据值。

> GET /api/v1/series

URL查询参数：

- `match[]=<series_selector>`：选择器是series_selector。这个参数个数必须大于等于1.
- `start=<rfc3339 | unix_timestamp>`：开始时间戳。
- `end=<rfc3339 | unix_timestamp>`：结束时间戳。

返回结果的 `data` 部分，是由key-value键值对的对象列表组成的。

下面这个例子返回时间序列数据, 选择器是 `up` 或者 `process_start_time_seconds{job="prometheus"}`

```
$ curl -g 'http://localhost:9090/api/v1/series?match[]=up&match[]=process_start
_time_seconds{job="prometheus"}'
{
   "status" : "success",
   "data" : [
      {
         "__name__" : "up",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      },
      {
         "__name__" : "up",
         "job" : "node",
         "instance" : "localhost:9091"
```

```
      },
      {
         "__name__" : "process_start_time_seconds",
         "job" : "prometheus",
         "instance" : "localhost:9090"
      }
   ]
}
```

查询标签值

下面这个例子，返回了带有指定标签的标签值列表

> GET /api/v1/label/<label_name>/values

这个返回JSON结果的 `data` 部分是带有label_name=job的值列表：

```
$ curl http://localhost:9090/api/v1/label/job/values
{
   "status" : "success",
   "data" : [
      "node",
      "prometheus"
   ]
}
```

删除时间序列

下面的例子，是从Prometheus服务中删除匹配的度量指标和标签列表：

> DELETE /api/v1/series

URL查询参数

- `match[]=<series_selector>`：删除符合series_selector匹配器的时间序列数据。参数个数必须大于等于1.

返回JSON数据中的 `data` 部分有以下的格式

```
{
   "numDeleted": <number of deleted series>
}
```

下面的例子删除符合度量指标名称是 `up` 或者时间序列为 `process_start_time_seconds{job="prometheus"}` ：

```
$ curl -XDELETE -g 'http://localhost:9090/api/v1/series?match[]=up&match[]=proc
ess_start_time_seconds{job="prometheus"}'
{
   "status" : "success",
   "data" : {
      "numDeleted" : 3
   }
}
```

表达式查询结果格式

表达式查询结果，在 `data` 部分的 `result` 部分中，返回下面的数据。 `\<sample_value\>` 占位符有数值样本值。JSON不支持特殊浮点值，例如： `NaN` ， `Inf` 和 `-Inf` 。因此样本值返回结果是字符串，不是原生的数值。

范围向量

范围向量返回的result类型是一个 `matrix` 矩阵。下面返回的结果是 `result` 部分的数据格式：

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

瞬时向量

瞬时向量的 `result` 类型是 `vector` 。下面是 `result` 部分的数据格式

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

Scalars标量

标量查询返回 `result` 类型是 `scalar` 。下面是 `result` 部分的数据格式：

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

字符串

字符串的 `result` 类型是 `string` 。下面是 `result` 部分的数据格式：

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

Targets目标

这个API是实验性的，暂不翻译。

Alertmanagers

这个API也是实验性的，暂不翻译

启动

启动

这是个类似"hello,world"的试验，教大家怎样快速安装、配置和简单地搭建一个DEMO。你会下载和本地化运行Prometheus服务，并写一个配置文件，监控Prometheus服务本身和一个简单的应用，然后配合使用query、rules和图表展示采样点数据

下载和运行Prometheus

最新下载页, 然后提取和运行它，so easy：

```
tar zxvf prometheus-*.tar.gz
cd prometheus-*
```

在开始启动Prometheus之前，我们要配置它

配置Prometheus监控自身

Prometheus从目标机上通过http方式拉取采样点数据, 它也可以拉取自身服务数据并监控自身的健康状况
当然Prometheus服务拉取自身服务采样数据，并没有多大的用处，但是它是一个好的DEMO。保存下面的Prometheus配置，并命名为： `prometheus.yml` ：

```
global:
  scrape_interval:     15s # 默认情况下, 每15s拉取一次目标采样点数据。

  # 我们可以附加一些指定标签到采样点度量标签列表中, 用于和第三方系统进行通信, 包括：federat
ion, remote storage, Alertmanager
  external_labels:
    monitor: 'codelab-monitor'

# 下面就是拉取自身服务采样点数据配置
scrape_configs:
  # job名称会增加到拉取到的所有采样点上, 同时还有一个instance目标服务的host：port标签也会
增加到采样点上
  - job_name: 'prometheus'

    # 覆盖global的采样点, 拉取时间间隔5s
    scrape_interval: 5s

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

对于一个完整的配置选项，请见配置文档

启动Prometheus

指定启动Prometheus的配置文件，然后运行

```
./prometheus --config.file=prometheus.yml
```

这样Prometheus服务应该起来了。你可以在浏览器上输入： `http://localhost:9090` ，就可以看到Prometheus的监控界面
你也可以通过输入 `http://localhost:9090/metrics` ，直接拉取到所有最新的采样点数据集

使用expression browser(暂翻译：浏览器上输入表达式)

为了使用Prometheus内置浏览器表达式，导航到 `http://localhost:9090/graph` ，并选择带有"Graph"的"Console".
在拉取到的度量采样点数据中， 有一个metric叫 `prometheus_target_interval_length_seconds` ，两次拉取实际的时间间隔，在表达式的console中输入:

```
prometheus_target_interval_length_seconds
```

这个应该会返回很多不同的倒排时间序列数据，这些度量名称都是 `prometheus_target_interval_length_seconds` ，但是带有不同的标签列表值，这些标签列表值指定了不同的延迟百分比和目标组间隔
如果我们仅仅对99%的延迟感兴趣，则我们可以使用下面的查询去清洗信息：

```
prometheus_target_interval_length_seconds{quantile="0.99"}
```

为了统计返回时间序列数据个数，你可以写：

```
count(prometheus_target_interval_length_seconds)
```

有关更多的表达式语言，请见表达式语言文档

使用graph interface

见图表表达式，导航到 `http://localhost:9090/graph` ， 然后使用"Graph" tab
例如，进入下面表达式，绘图最近1分钟产生chunks的速率：

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

启动其他一些采样目标

Go客户端包括了一个例子，三个服务只见的RPC调用延迟

首先你必须有Go的开发环境，然后才能跑下面的DEMO, 下载Prometheus的Go客户端，运行三个服务:

```
git clone https://github.com/prometheus/client_golang.git
cd client_golang/examples/random
go get -d
go build

## 启动三个服务
./random -listen-address=:8080
./random -listen-address=:8081
./random -listen-address=:8082
```

现在你在浏览器输入: `http://localhost:8080/metrics` ， `http://localhost:8081/metrics` ， `http://localhost:8082/metrics` ，能看到所有采集到的采样点数据

配置Prometheus去监控这三个目标服务

现在我们将会配置Prometheus，拉取三个目标服务的采样点。我们把这三个目标服务组成一个job, 叫 `example-radom` . 然而，想象成，前两个服务是生产环境服务，后者是测试环境服务。我们可以通过group标签分组，在这个例子中，我们通过 `group="production"` 标签和 `group="test"` 来区分生产和测试

```
scrape_configs:
  - job_name:       'example-random'

    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'test'
```

进入浏览器，输入 `rpc_duration_seconds` ，验证Prometheus所拉取到的采样点中每个点都有group标签，且这个标签只有两个值 `production` ， `test`

聚集到的采样点数据配置规则

上面的例子没有什么问题， 但是当采样点海量时，计算成了瓶颈。查询、聚合成千上万的采样点变得越来越慢。

为了提高性能，Prometheus允许你通过配置文件设置规则，对表达式预先记录为全新的持续时间序列。让我们继续看RPCs的延迟速率(`rpc_durations_seconds_count`), 如果存在很多实例，我们只需要对特定的 `job` 和 `service` 进行时间窗口为5分钟的速率计算，我们可以写成这样：

```
avg(rate(rpc_durations_seconds_count[5m])) by (job, service)
```

为了记录这个计算结果，我们命名一个新的度量：`job_service:rpc_durations_seconds_count:avg_rate5m` ，创建一个记录规则文件，并保存为 `prometheus.rules.yml` ：

```
groups:
- name: example
  rules:
  - record: job_service:rpc_durations_seconds_count:avg_rate5m
    expr: avg(rate(rpc_durations_seconds_count[5m])) by (job, service)
```

然后再在Prometheus配置文件中，添加 `rule_files` 语句到 `global` 配置区域， 最后配置文件应该看起来是这样的：

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.

  # Attach these extra labels to all timeseries collected by this Prometheus in
stance.
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  - 'prometheus.rules.yml'

scrape_configs:
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seco
nds.
    scrape_interval: 5s

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

  - job_name:       'example-random'

    # Override the global default and scrape targets from this job every 5 seco
```

```
nds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'test'
```

然后重启Prometheus服务，并指定最新的配置文件，查询并验证

`job_service:rpc_durations_seconds_count:avg_rate5m` 度量指标

可视化

表达式浏览器

Expression Browing 表达式浏览器

这个表达式浏览器在Prometheus服务中在 `/graph` 是可用的，允许用户通过输入表达式，在table或者graph中获取时间序列数据

表达式浏览器主要用于即席查询和调试。对于Graph，可以使用Grafana或者console templates

Grafana

Grafana支持Prometheus可视化

Grafana支持Prometheus查询。从Grafana 2.5.0 (2015-10-28)开始Prometheus可以作为它的数据源。

下面的例子：Prometheus查询在Grafana Dashboard界面的图表展示

Grafana安装

如果要Grafana的完整安装教程，详见Grafana官方文档

在Linux安装Grafana，如下所示：

```
# Download and unpack Grafana from binary tar (adjust version as appropriate).
curl -L -O https://grafanarel.s3.amazonaws.com/builds/grafana-2.5.0.linux-x64.tar.gz
tar zxf grafana-2.5.0.linux-x64.tar.gz

# Start Grafana.
cd grafana-2.5.0/
./bin/grafana-server web
```

使用方法

默认情况下，Grafana服务端口http://localhost:3000。默认登录用户名和密码“admin/admin”。

创建一个Prometheus数据源

为了创建一个Prometheus数据源Data source：

1. 点击Grafana的logo，打开工具栏。
2. 在工具栏中，点击"Data Source"菜单。
3. 点击"Add New"。
4. 数据源Type选择“Prometheus”。
5. 设置Prometheus服务访问地址（例如：`http://localhost:9090`）。
6. 调整其他想要的设置（例如：关闭代理访问）。
7. 点击“Add”按钮，保存这个新数据源。

下面显示了一个Prometheus数据源配置例子：

创建一个Prometheus Graph图表

下面是添加一个新的Grafana的标准方法：

1. 点击图表Graph的title，它在图表上方中间。然后点击“Edit”。
2. 在“Metrics”tab下面，选择你的Prometheus数据源（下面右边）。
3. 在“Query”字段中输入你想查询的Prometheus表达式，同时使用“Metrics”字段通过自动补全查找度量指标。
4. 为了格式化时间序列的图例名称，使用“Legend format”图例格式输入。例如，为了仅仅显示这个标签为 `method` 和 `status` 的查询结果，你可以使用图例格式 `{{method{} - {{status}}` 。
5. 调节其他的Graph设置，知道你有一个工作图表。

下面显示了一个Prometheus图表配置：

从Grafana.net导入预构建的dashboard

Grafana.net维护一个共享dashboard的收集，它们能够被下载，并在Grafana服务中使用。使用Grafana.net的“Filter”选项去浏览来自Prometheus数据源的dashboards

你当前必须手动编辑下载下来的JSON文件和更改 `datasource` ：选择Prometheus服务作为Grafana的数据源，使用“Dashboard”->"Home"->"Import"选项去导入编辑好的dashboard文件到你的Grafana中。

控制模板

console template（控制模板）

控制模板允许使用Go语言模板创建任意的console。这些由Prometheus服务提供
console模板是最强有力的方法，它可以在源码控制中创建容易管理的模板。我们可以首先尝试Grafana，减少学习成本。

Getting started

Prometheus提供了一系列的控制模板来帮助您。这些可以在Prometheus服务上的 `console/index.html.example` 中找到，如果Prometheus服务正在删除带有标签 `job="node"` 的Node Exporter, 则会显示NodeExporter控制台
这个例子控制台包括5部分：

1. 在顶部的导航栏
2. 左边的一个菜单
3. 底部的时间控制
4. 在中心的主内容，通常是图表
5. 右边的表格

这个导航栏是链接到其他系统，例如Prometheus其他方面的文档，以及其他任何使你明白的。该菜单用于在同一个Prometheus服务中导航，它可以快速在另一个tar中打开一个控制台。这些都是在 `console_libraries/menu.lib` 中配置。
时间控制台允许持久性和图表范围的改变。控制台URLs能够被分享，并且在其他的控制台中显示相同的图表。
主要内容通常是图表。这里有一个可配置的JavaScript图表库，它可以处理来自Prometheus服务的请求，并通过Rickshaw来渲染
最后，在右边的表格可以用笔图表更紧凑的形式显示统计信息。

例子控制台

这是一个最基本的控制台。它显示任务的数量，其中CPU平均使用率、以及右侧表中的平均内存使用率。主要内容具有每秒查询数据。

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

{{template "prom_right_table_head"}}
<tr>
  <th>MyJob</th>
  <th>{{ template "prom_query_drilldown" (args "sum(up{job='myjob'})") }}
```

```
      / {{ template "prom_query_drilldown" (args "count(up{job='myjob'})") }}
  </th>
</tr>
<tr>
  <td>CPU</td>
  <td>{{ template "prom_query_drilldown" (args
      "avg by(job)(rate(process_cpu_seconds_total{job='myjob'}[5m]))"
      "s/s" "humanizeNoSmallPrefix") }}
  </td>
</tr>
<tr>
  <td>Memory</td>
  <td>{{ template "prom_query_drilldown" (args
       "avg by(job)(process_resident_memory_bytes{job='myjob'})"
       "B" "humanize1024") }}
  </td>
</tr>
{{template "prom_right_table_tail"}}


{{template "prom_content_head" .}}
<h1>MyJob</h1>

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

{{template "prom_content_tail" .}}

{{template "tail"}}
```

模板部分不翻译了，建议大家用Grafana，不喜欢后台服务渲染模板，还是让前端的童鞋去做数据呈现工作吧

Instrumenting

客户库

工具

客户端库

在你能够监控你的服务器之前，你需要通过Prometheus客户端库把监控的代码放在被监控的服务代码中。下面实现了Prometheus的度量指标类型metric types。
选择你需要的客户端语言，在你的服务实例上通过HTTP端口提供内部度量指标。

- Go
- Java or Scala
- Python
- Ruby

非正式的第三方客户端库

- Bash
- C++
- Common Lisp
- Elixir
- Erlang
- Haskell
- Lua for Nginx
- Lua for Tarantool
- .Net/C#
- Node.js
- PHP
- Rust

当Prometheus获取实例的HTTP端点时，客户库发送所有跟踪的度量指标数据到服务器上。
如果没有可用的客户端语言版本，或者你想要避免依赖，你也可以实现一个支持的导入格式到度量指标数据中。
在实现一个新的Prometheus客户端库时，请遵循客户端指南。注意，这个文档在仍然在更新中。同时也请关注开发邮件列表。我们非常乐意地给出合适的意见或者建议。

写客户库

编写客户端库

这篇文档包括Prometheus客户端API应该提供的基础功能，目的是在客户端库之间保持一致性，轻松上手并避免提供导致用户出错的功能。
已经有10种客户端语言支持Prometheus客户端了，因此我们知道怎么写好一个客户端。这个指南旨在帮助写Prometheus客户端其他语言的作者写一个好的库。

Conventions约定

MUST/MUST NOT/SHOULD/SHOULD NOT/MAY在https://www.ietf.org/rfc/rfc2119.txt
另一个ENCOURAGE的含义是，一个特性对于一个库是非常好的，但是如果关闭这个特性的话，不会影响库的使用。
记住下面的几点：

- 记住每个特性的好处。
- 常用用例应该很简单
- 做事情正确方式是简单的方法
- 更复杂的例子应该是可能的

常用用例（有序）：

- 没有标签的Counters在库或者应用程序之间传播
- Summaries/Histograms的时序功能/代码块
- Gauges跟踪事情的当前状态
- 批量任务监控

总体结构

客户端 `必须` 在内部写入回调。客户通常 `应该` 遵循下面描述的结构。
这个关键类是 `Collector` 。这个有一个典型的方法 `collect` ，返回0~N个度量指标和这些指标的样本数据。 `Collector` 用 `CollectorRegistry` 进行注册。通过传递 `CollectorRegistry` 给称之为 `bridge` 的class/method/function来暴露数据。该 `bridge` 返回Prometheus支持的数据格式数据。每次这个 `CollectorRegistry` 被收集时，都必须回调 `Collector` 的collect方法。
和用户交互最多的接口是 `Counter` ， `Gauge` ， `Summary` 和 `Histogram Collectors` 。这些表示单个度量指标，写的代码覆盖绝大多数的用例。
更高级的用例（例如来自其他监控/检测系统的代理）需要编写一个自定义 `Collector` 收集器。有人也可能像写一个带有 `CollectorRegistry` 的"bridge"，以不同的监控/测量系统理解的格式生产数据, 允许用户只需

要考虑一个测量系统。

`CollectorRegistry` 应该提供 `register()/unregister()` 方法，以及一个 `Collector` 应该注册多个 `CollectorRegistrys`

客户库必须是线程安全的。

对于非面向对象的客户端，如：C语言，客户库编写在实践中应该遵循这种结构的理念。

命名

客户库应该遵循 `function/method/class` 在这个文档中提及的命名规则，记住他们正在使用的语言命名规范。例如： `set_to_current_time()` 对于python而言是非常好的方法名称， `SetToCurrentTime` 对于Go语言是更好的， `setToCurrentTime()` 对于Java是更好的。由于技术原因（例如：不允许功能重载），名称不能，文档/帮助文档应该将用户指向其他名称。

库禁止提供与此处给出的相同或者相似 `functions/methods/classes` ，但具有不同的语义。

Metrics

`Counter` 、 `Gauge` 、 `Summary` 和 `Histogram` 度量指标类型是最主要的接口。

`Counter` 和 `Gauge` 必须是客户库的一部分。 `Summary` 和 `Histogram` 至少被提供一个。

这些主要用作文件静态变量，也就是说，全局变量与他们正在调试的代码在同一个文件中定义。客户端库应该启用此功能。常见的用例是整体测试一段代码，而不是在对象的一个实例上下文中的一段代码。用户不必担心在他们的代码中管理他们的指标，客户端库应该为他们做到这一点（如果不这样做，用户将会围绕库写一个 `wrapper` ，使其更容易，少即是多）。

必须有一个默认的 `CollectorRegistry` ， 四种度量指标类型必须在不需要用户任何干预下，就能完成默认被注册，同时也提供一种别的注册方法，用于批处理作业和单元测试。自定义的 `Collectors` 也应该遵循这点。

究竟应该如何创建度量指标因语言而异。对于某些语言（Go，Java），builder是最好的，对于其他（Python）函数参数足够丰富，可以在一个调用中执行。

例如，一个简单的Java客户端，我们可以这样写：

```
class YourClass {
  static final Counter requests = Counter.build()
      .name("requests_total")
      .help("Requests.").register();
}
```

上面的例子，使用默认的 `CollectorRegistry` 进行注册。如果只是调用build()方法， 度量指标将不会被注册（对于单元测试来说很方便），你还可以将 `CollectorRegistry` 传递给register()(方便批作业处理)。

Counter

`Counter` [https://prometheus.io/docs/concepts/metric_types/#counter]是一个单调递增的计数器。它不允许counter值下降，但是它可以被重置为0（例如：客户端服务重启）。
一个counter必须有以下方法：

- `inc()`：增量为1.
- `inc(double v)`：增加给定值v。必须检查v>=0。

Counter在给定代码段抛出/引发异常的方式，也可以只选择某些类型的一场，这是Python中的count_exceptions。
Counters必须从0开始。

Gauge

Gauge表示一个可以上下波动的值。
gauge必须有以下的方法：

- `inc()`：每次增加1
- `inc(double v)`：每次增加给定值v
- `dec()`：每次减少1
- `dec(double v)`：每次减少给定值v
- `set(double v)`：设置gauge值成v

Gauges值必须从0开始，你可以提供一个从不等于0的值开始。
gauge应该有以下方法：

- `set_to_current_time()`：将gauge设置为当前的unix时间（以秒为单位）。

gauge被建议有：

- 一种在某些代码/方法中跟踪正在进行的请求方法。这是python种的 `track_inprogress` 。

执行一段代码，设置gauge类型数据样本值为这段代码执行的时间，这对于批量任务是非常有用的。在Java中是 `startTimer/setDuration` ，在python中是 `time()` decorator/上下文管理器。这应该符合在 `Summary` 和 `Histogram` 中的pattern(通过 `set()` 而不是 `observe()`)。

Summary

summary通过时间滑动窗口抽样观察（通常是要求持续时间），并提供对其分布、频率和总和的即时观察。
summary不允许用户设置"quantile"作为一个标签，因为这个名称已在内部使用，用来指定分位数。summary鼓励提供“quantile”导出，虽然这些不能被汇总，而且需要大量时间。summary必须允许没有quantiles，因为只有 `_count/_sum` 是飞铲更拥有的，这必须是默认值。
summary必须有以下方法：

- `observe(double v)`：观察被给定值

summary应该有以下方法：

- 统计用户执行代码的时间，以秒为单位。在python中，这是 `time()` decorateor/context管理器。在Java中这是 `startTimer/observeDuration` 。 不能提供秒意外的单位（如果用户想要别的，自己手动做）。这应该遵循Gauge/Histogram相同的模式。

Summary `_count/_sum` 必须从0开始。

Histogram

Histogram允许时间的可聚合分布，如：请求延迟。每个bucket中都会有一个count值, 表示累加的样本数量值
一个histogram直方图不允许使用 `le` 作为一个标签，它已经内部用于在分bucket时的步长大小。
直方图必须提供一个方法来手动选择buckets。应该提供一linear(start, width, count)和exponential(start，factor, count）方式设置buckets的方法。参数count值必须是有界的
直方图应该具有与其他客户端库相同的默认值，创建度量指标后bucket不能再更改。
一个直方图必须有下面的方法：

- `observe(double v)`：观察给定值

直方图应该有以下的方法：
统计代码执行时间的一些方法，以秒为单位。在Python中是 `time()` decorator/context管理器。在Java中是 `startTimer/observeDuration` 。不提供秒以外的单位（如果用户需要别的，可以手动做）。这应该遵循与Gauge/Summary相同的模式。
直方图 `_count/_sum` 和buckets必须从0开始。
进一步的度量指标考量
提供额外的功能，超出以上记录的指标，对于给定的语言是有意义的
如果有一个常见用例，例如：次优度量指标/标签布局或者在客户端进行计算，可以使其更简单。

标签

标签Labels是Prometheus系统最强大的特性之一，但是很容易被滥用。因此，客户端库必须非常小心地如何向用户提供labels。
客户库在任何情况下禁止用户对于"Gauge/counter/summary/histogram"或者由库提供的其他Collector的度量指标，提供不相同的标签列表。
如果你的客户库在收集样本数据时间内对其进行了度量指标的验证，那么它也可以为自定义Collector进行验证。
虽然标签功能很强大，但大多数度量指标没有标签。因此，API允许有标签，但不是强制的。
客户库必须允许在Gauge/Counter/Summary/Histogram创建时间可选地指定标签名称列表。客户端库应该支持任意大小的标签列表。客户端库必须验证标签名称是否符合要求。

提供访问度量指标名称列表最常用的方法, 是通过 `labels()` 方法，该方法可以获取标签值列表，或者获取Map键值对(标签名称：标签值)列表，并返回“child”，然后在Child上调用常用的 `.inc()/.desc()/.observe()` 等方法。

`label()` 返回Child应该由用户缓存，以避免再次查找，这在延迟至关重要的代码中很重要。

带有标签的度量指标应该支持一个具有与 `labels()` 相同签名的 `remove()` 方法，它将从不再导出它的度量标准中删除一个Child，另一个clear()方法可以从度量指标中删除所有的 `Child` 。

应该有一种使用默认初始化给定Child的方法，通常只需要调用labels()。没有标签的度量指标必须被初始化，已避免缺少度量指标的问题。

度量指标名称

度量指标名称补习遵循规范。与标签名称一样，必须满足使用 `Counter/Gauge/Summary/Histogram` 和库中提供的任意其他 `Collector` 的使用。

许多客户库提供三个部分的名称： `namespace_subsystem_name` ，其中只有该 `name` 是强制性的。

不鼓励使用动态/自动生成的度量指标名称或者其子部分，除非自定义"Collector"是从其他工具/监控系统代理的。你可以使用标签名称替代动态或者自动生成的度量指标名称。

度量指标描述和帮助

`Gauge/Counter/Summary/Histogram` 要求必须提供度量指标的desc和help。

带有自定义Collector的客户库，在度量指标上必须有desc/help

建议将度量指标名称的desc/help作为强制性参数，但不需要检查其长度，提供Collectors的库应该要有一个比较好的desc，帮助理解其含不需要检查其长度，提供Collectors的库应该要有一个比较好的desc，帮助理解其含义.

导出

客户端必须实现一个文档导出格式。

客户端可以实现多种导出格式。而且是可读性非常好的格式。

如果有疑问，请去文本格式。它不具有依赖性（protobuf），往往易于生成，是可读取的，并且protobuf的性能优势对于大多数用例来说并不重要。

如果可以在没有显著的资源成本情况下实现，可以重现可用的度量指标顺序（特别是对于人类可读格式）。

标准化和运行时收集器

客户端库应该提供标准导出，如下所述：

这些应该作为自定义Collectors实现，默认情况下在默认的CollectorRegistry上注册。应该有一种方法来禁用这些，因为有一些非常适用于他们的使用方式。

处理度量指标

这些导出应该有前缀process_。如果一种语言或者运行时没有公开其中一个变量，它不会被导出它。所有内存值

以字节为单位，以时间戳/秒为单位。

度量指标名称	含义	单位
process_cpu_seconds_total	用户和系统CPU花费的时间	秒
process_open_fds	打开的文件描述符数量	文件描述符
process_max_fds	打开描述符最大值	文件描述符
process_virtual_memory_bytes	虚拟内存大小	字节
process_resident_memory_bytes	驻留内存大小	字节
process_heap_bytes	进程head堆大小	字节
process_start_time_seconds	unix时间	秒

运行时度量指标

另外，客户端库也被提供给他们的语言运行时（如：垃圾回收统计信息）的指标方面，提供了一些合适的前缀，比如： go_, hostspot_等。

单元测试

客户端库应该包含核心工具库和展示的单元测试。

客户端库被鼓励提供方便用户单元测试其使用的工具代码。例如，python中的CollectorRegistry.get_sample_value。

包和依赖

理想情况下，客户端库可以包含在任何应用程序中以添加一些工具，而无需担心它会破坏应用程序。

因此，当向客户端添加依赖关系时，建议谨慎。例如：如果用户添加使用添加版本1.4的Protobuf的Prometheus客户端库，但是应用程序在其他地方使用1.2，会发生什么？

建议在可能出现的情况下，将核心工具和给定格式的度量指标/展示分开。例如：Java简单客户端模块没有依赖关系，而simpleclient_servlet具有Http比特位。

性能考虑

由于客户端库必须是线程安全的，因此需要进行某种形式的并发控制，并且必须考虑多核机器和应用程序的性能。

在我们的经验中，性能最差的是互斥体。

处理器原子指令往往处于中间，并且通常可以接受。

避免不同CPU突然使用RAM的方法效果最好，例如：Java简单客户端中的DoubleAdder。有内存成本。

如上所述，labels()的结果应该是可缓存的。趋向于使用标签返回度量的并发映射往往相对较慢。没有标签的特殊套管指标，已避免labels()，像查找可以帮助很多。

度量指标应该避免阻塞，当它们递增/递减/设置等时，因为整个应用程序在持续获取时不会被组织。

主要工具操作的基准（包括labels）得到了鼓励。

应该牢记资源消耗，特别是RAM。考虑通过stream传输结果来减少内存占用，并且可能对并发获取的数量有限制。

推送度量指标

推送度量指标

偶尔你需要监控不能被获取的实例。它们可能被防火墙保护，或者它们生命周期太短而不能通过拉模式获取数据。Prometheus的Pushgateway允许你将这些实例的时间序列数据推送到Prometheus的代理任务中。结合Prometheus简单的文本导出格式，这使得即使没有客户库，也能使用shell脚本获取数据。

- shell实现用例，查看Readme
- Java, 详见PushGateway类
- Go，详见Push和PushAdd
- Python, 详见Pushgateway
- Ruby, 详见Pushgateway

Java批量任务例子

这个例子主要说明, 如何执行一个批处理任务，并且在没有执行成功时报警
如果使用Maven，添加下面的代码到 `pom.xml` 文件中：

```
        <dependency>
            <groupId>io.prometheus</groupId>
            <artifactId>simpleclient</artifactId>
            <version>0.0.10</version>
        </dependency>
        <dependency>
            <groupId>io.prometheus</groupId>
            <artifactId>simpleclient_pushgateway</artifactId>
            <version>0.0.10</version>
        </dependency>
```

执行批量作业的代码：

```
import io.prometheus.client.CollectorRegistry;
import io.prometheus.client.Gauge;
import io.prometheus.client.exporter.PushGateway;

void executeBatchJob() throws Exception {
 CollectorRegistry registry = new CollectorRegistry();
 Gauge duration = Gauge.build()
     .name("my_batch_job_duration_seconds")
     .help("Duration of my batch job in seconds.")
     .register(registry);
 Gauge.Timer durationTimer = duration.startTimer();
```

```
 try {
   // Your code here.

   // This is only added to the registry after success,
   // so that a previous success in the Pushgateway is not overwritten on failu
re.
   Gauge lastSuccess = Gauge.build()
       .name("my_batch_job_last_success_unixtime")
       .help("Last time my batch job succeeded, in unixtime.")
       .register(registry);
   lastSuccess.setToCurrentTime();
 } finally {
   durationTimer.setDuration();
   PushGateway pg = new PushGateway("127.0.0.1:9091");
   pg.pushAdd(registry, "my_batch_job");
 }
}
```

警报一个Pushgateway，如果需要的话，修改host和port

如果任务最近没有运行，请创建一个警报到Alertmanager。将以下内容添加到Pushgateway的Prometheus服务的记录规则中：

```
ALERT MyBatchJobNotCompleted
  IF min(time() - my_batch_job_last_success_unixtime{job="my_batch_job"}) > 60
* 60
  FOR 5m
  WITH { severity="page" }
  SUMMARY "MyBatchJob has not completed successfully in over an hour"
```

导出与集成

导出和集成

有很多库和服务, 支持从第三方系统的度量指标导入到Prometheus的度量指标。这对于（例如：HAProxy或者Linux系统统计信息）不能直接使用Prometheus度量指标是非常有用的。

第三方导出器

有一些exporter主要有Prometheus Github组织维护的,详见地址，这些项目被标记为official， 其他都是由外部贡献和维护的。
我们鼓励更多exporters的出现，但无法对所有人进行推广，通常这些exporters被托管在Prometheus Github组织之外。
导出器默认端口wiki页面已经成为了导出器的目录列表，包括的一些不在这里列出的导出器，主要原因是有些导出器功能类似，或者还在开发中。
JMX exporter能够从大多数JVM应用中导出数据，例如：kafka和cassandra。

数据库Databases

- Aerospike exporter
- ClickHouse exporter
- Consul exporter官方
- CouchDB exporter
- ElasticSearch exporter
- Memcached exporter官方
- MongoDB exporter
- MySQL server exporter官方
- PgBouncer exporter
- PostgreSQL exporter
- ProxySQL exporter
- Redis exporter
- RethinkDB exporter
- SQL query result set metrics exporter
- Tarantool metric library

硬件相关Hardware related

- apcupsd exporter

- IoT Edison exporter
- knxd exporter
- Node/System metrics exporter官方
- Ubiquiti UniFi exporter

消息系统

- NATS exporter
- NSQ exporter
- RabbitMQ exporter
- RabbitMQ Management Plugin exporter
- Mirth Connect exporter

存储Storage

- Ceph exporter
- ScaleIO exporter
- Gluster exporter

HTTP

- Apache exporter
- HAProxy exporter官方
- Nginx metric library
- Nginx VTS exporter
- Passenger exporter
- Varnish exporter
- WebDriver exporter

APIs

- AWS ECS exporter
- Cloudfare exporter
- DigitalOcean exporter
- Docker Cloud exporter
- Docker Hub exporter
- Github exporter
- Mozilla Observatory exporter
- OpenWeatherMap exporter
- Rancher exporter

- Speedtest.net.exporter

Logging

- Google's mtail log data extractor
- Grok exporter

其他的监控系统

- Akamai Colud monitor exporter
- AWS CloudWatch exporter官方
- Cloud Foundry Firehose exporter
- Collectd exporter官方
- Graphite exporter官方
- Heka dashboard exporter
- Heka exporter
- InfluxDB exporter官方
- JMX exporter官方
- Munin exporter
- New Relic exporter
- Pingdom exporter
- scollector exporter
- SNMP exporter官方
- StatsD exporter

其他杂项

- BIG-IP exporter
- BIND exporter
- BlackBox exporter官方
- BOSH exporter
- Dovecot exporter
- Jenkins exporter
- Kemp LoadBalancer exporter
- Meteor JS web framework exporter
- Minecraft exporter module
- PowerDNS exporter
- Process exporter
- rTorrent exporter

- Script exporter
- SMTP/Maildir MDA blackbox prober
- Transmission exporter
- Unbound exporter
- Xen exporter

当实现一个新的Prometheus导出器时，请遵循[writing exporter指南]（https://prometheus.io/docs/instrumenting/writing_exporters ）,也请参考邮件列表。我们乐意给出建议告诉你怎样写一个尽可能有用地，一致性地导出器

可直接使用的软件

一些第三方软件本身就已经暴露了Prometheus度量指标, 因此不需要exporter：

- cAdvisor
- Doorman
- Etcd
- Kubernetes-Mesos
- Kubernetes
- RobustIRC
- Quobyte
- SkyDNS
- Weave Flux

其他第三方实用工具

本节列出了帮助您以特定语言编写代码的库和其他实用程序。它们不是Prometheus客户端库，而是使用客户库。对于所有独立维护的软件，我们无法对其进行优化和改进。

- Coljure: prometheus-clj
- Go: go-metrics instrumentation library
- Go: gokit
- Java/JVM: Hystrix metrics publisher
- Python-Django: django-prometheus

写导出器

编写导出器

当直接检测你的代码质量时，通过Prometheus客户库检测你的代码质量是一个常用手段。当从另一个监控或者检测系统获取度量指标时，事情往往不是一成不变的。
当你准备写度量指标的导出器或者自定义收集器时，这篇文章值得你借鉴和学习。涉及的理论对于做一些检测的事情是非常有帮助的。
如果你正在写一个导出器，且有任何不清楚的，可以随时联系我们, 详见IRC和邮件

可维护性和Purity

当你写一个exporter时，你需要做出的最大决定是你会投入多大的精力，来获得比较完美的度量指标测量
一方面，如果系统的度量指标很少有变化，则写导出器是不需要很长的时间和精力（例如：HAProxy exporter）；另一方面，当系统版本每次更新时，若有大量的度量指标发生变化，则你需要投入大量的时间和精力写一个导出器，mysql导出器就是一个例子
这个node exporter是混合态的，不同的模块的处理方式也不尽相同。对于mdadm，我们必须手动解析文件，并获取我们想要的度量指标，所以我们可以在需要的时候获取度量指标。对于meminfo，不同的内核版本命名可能会有变化，所以我们需要创建有效的度量指标，以适应不同内核版本的内存使用情况

配置

当导出器和应用程序一起工作时，你的导出器目标是尽量不要让用户使用自定义配置，而是尽可能地使用默认配置就ok了。你也可以给你的导出器提供过滤度量指标的支持，因为有些度量指标是非常多且琐碎，也有些需要花费很大的代价才能获取度量指标，这些都可以进行过滤（例如：HAProxy exporter允许过滤每个服务器的统计信息）。类似地，默认配置可以直接禁用获取代价比较的度量指标。
当导出器和监控系统协同工作时，框架和协议相关工作可能会很复杂。
使用监控系统时，框架和协议不是那么简单的。
最好的情况是其他监控或者检测系统与Prometheus有非常相似的数据模型，则系统可以自动导出度量指标到Prometheus监控系统中，如：Cloudwatch, SNMP和Collectd。但是大多数情况下，我们都需要用户自己指定其他监控或者检测系统需要导出的度量指标，并把这些度量指标样本数据导入到Prometheus系统中。
但是更多常见的情况是，来自其他监控或者检测系统的度量指标往往都是非标准的，它们往往依赖于用户使用它的方法和底层应用程序是什么样的。这种情况下，用户必须告诉我们怎么转换这些度量指标。这个JMX导出器是最糟糕的，graphite和statsd导出器也要求在配置中抽取标签集。
提供一些导出器的使用方式是值得建议的，例如：生成这个导出器的输出和实例配置的选择。当为这些导出器写一些配置文档时，这篇文档应该时刻牢记在心。
YAML是Prometheus的标准配置格式

度量指标

命名

遵循度量指标最佳实践

通常度量指标名称应该允许对Prometheus监控系统比较熟悉的人， 对一个度量名称的字面含义比较容易理解它的真正含义。一个命名为 `http_requests_total` 的度量指标名称不是很有用，因为它是针对所有进入系统的请求计数。如果改为 `requests_total` ，这度量指标名称更差劲，它不能表示是什么样的请求，是tcp，udp还是http？

为了把度量指标推送到检测系统中，一个给定的度量指标应该在一个指定的文件中存在。根据导出器和收集器，一个度量指标名称应该应用在一个子系统中。

度量指标名称永远不要程序化的生成，除非在编写自定义收集器或者导出器时。

应用程序的度量指标名称应该以exporter名称为前缀，如 `haproxy_up` 。

度量指标名称必须使用基本单位（例如：秒，字节），并将其转换为图形软件更易读的内容。无论你最终使用什么样的计量单位，指标名称中的单位都必须与正在使用的单位相匹配。类似地展示比率，而不是百分比（尽管两个组件的计数器比率更好）。

度量指标名称不应该包括他们导出的labels（如： `by_type` ）如果标签聚在一起，就没有意义。

这里有一个特例，当你通过多个度量指标用不同标签导出相同数据时，通常是区分他们最好的方式。一个例子是，当对单个标签用所有标签导出数据时，这个计算是非常复杂的，这种情况下使用这个特例是非常合适的。

Prometheus度量指标和标签名称写在 `snake_case` 中。把 `camelCase` 转换成 `snake_case` 度量指标是可取的，尽管自动化地做这些事情并不总是产生好的结果，例如： `myTCPExample` 或 `isNaN` ，所以有时最好将他们保留。

暴露的度量指标不应包含冒号，这些用于聚合时可供用户使用。度量指标名称只有符合正则表达式[a-zA-Z0-9:]是有效的，任何其他字符串都需要被改成""下划线

`_sum` ， `_count` ， `_bucket` ，和 `_total` 用于Summaries，Histogram和Counters的后缀。如果你正好要使用这些统计则可以使用这些后缀，否则避免使用这些后缀。

如果你正在使用COUNTER类型，则应该使用 `_total` 作为度量指标的后缀。

这个 `process_` 和 `scrape_` 的前缀被保留。如果它们遵循匹配的语义，可以将这些前缀添加到度量指标上。例如：Prometheus的持续获取时间为 `scrape_duration_seconds` ，这是很好的做法。

`jmx_scrape_duration_seconds` 表示JMX收集器花费多久时间获取度量指标数据。对于对于进程信息统计，你可以通过PID进程号获取，Go和Python都会为你提供处理此选项的收集器（请参阅HAProxy exporter示例）

当你统计请求的成功数量和失败数量时，最佳方法是暴露两个度量指标，一个是总的请求数，另一个度量指标是失败的请求度量指标。这使得计算失败比率变得很容易。不要使用带有failed/success的标签。类似于缓存中的 `hit/miss` ，有一个总的度量指标和另一个hits的度量指标是更好的。

考虑使用监控的人可能会对指标名称执行代码或者网络搜索。如果这些名称是非常完善的，并且不太可能在用于

这些名称的人的领域之外（例如：SNMP和网络工程师）使用，那么按原样离开它们可能是一个好主意。该逻辑不适用于（例如：作为非数据库管理员的MySQL），可以预期会围绕这些指标。具有原始名称的帮助文档可以提供与使用以前的名称大致相同的好处。

labels

阅读一个有关labels标签的建议,详见advice

避免把 `type` 做为标签名称，它太通用化，而且无意义。你也应该试着尽可能地避免与目标标签冲突，例如： `region` ， `zone` ， `cluster` ， `availability` ， `az` ， `datacenter` ， `dc` ， `owner` ， `customer` ， `stage` ， `environment` 和 `env` ，尽管这是应用程序调用的东西，但是最好不要通过重命名造成混乱。

不要仅仅因为它们共享了同一个前缀，就把一些样本数据全部集成到一个度量指标下。除非你确定某些指标是有意义的，否则请使用多个度量指标。

标签 `le` 对于Histograms有特殊的含义。同样对于Summaries来说， `quantile` 也是如此。避免使用这些标签。

对于Read/write和send/receive, 不要把它们当做一个标签使用，而是应该作为独立的度量指标使用。因为你关心的是某个时刻它们中的一个，度量指标命名并使用它们是容易的。

实践经验是，当求和或者求平均时，一个度量指标应该是有意义的。 还有另一种情况导出器的出现，数据基本是用表格的方式呈现，否则将要求用户对度量标准名称进行正则表达式可用。 考虑您主板上的电压传感器，而对它们进行数学计算是无意义的，将它们置于一个度量标准中而不是每个传感器有一个度量是有意义的。 度量中的所有值（几乎）总是具有相同的单位（如果风扇速度与电压混合，则无法自动分离）。

不要做这些：

```
my_metric{label=a} 1
my_metric{label=b} 6
**my_metric{label=total} 7**
```

或者

```
my_metric{label=a} 1
my_metric{label=b} 6
**my_metric{} 7**
```

前者打破了用户Prometheus聚合操作 `sum()` 函数作用, 后者也一样。某些客户端库（如：Go）会积极地尝试阻止你在自定义收集器中执行后者。依赖Prometheus的 `sum` 聚合操作，可以轻易地达到这一点。

如果你的监控使用了一个total，请删除它。如果你必须因为某些原因（例如：这个total并不是统计计数）保留它，请使用其他的度量指标名称。

Target labels, not static scraped labels(目标标签，非静态获取标签)

如果你发现自己想要对所有度量指标使用相同的标签，请立即停止。
通常有两种情况出现。
第一个是有些标签关联到度量指标上市非常有用的，例如：软件的版本号。请使用文档地址中所述的方法。
另一种情况是真正的实例标签。这些标签是区域，集群名称等，它们来自目标实例的硬件信息而不是应用程序本身。这类标签不应该在分类中使用， 这个Prometheus服务加载服务配置，对于相同的应用可以给出不同的度量指标名称，但是可以有相同的标签名称
因此，这些标签通过你使用的任何服务发现，都属于Prometheus的获取配置。还可以在这里应用机器角色的概念，因为至少有一些人获取它，可能是非常有用的信息。

Types类型

你应该尝试将你的度量指标类型与Prometheus类型相匹配。这通常意味着主要类型是Counter和Gauge。summaries的 `_count` 和 `_sum` 也是比较常见的，有时你会看到quantiles。Histogram是罕见的，如果你碰到了，记住那展示格式用来暴露累计值。
通常，衡量度量指标的类型是不明显的（特别是如果你自动处理一组度量指标），那么在这种情况下使用UNTYPED无类型度量指标。一般来说，UNTYPED类型是一个比较安全的默认值。
Counter递增，如果你在其他监控或者检测系统中发现了一个counter类型，但是它能够减少(例如：Dropwizard指标），这不是一个Counter类型而是一个Gauge类型, UNTYPED无类型可能是那里使用最好的类型，因为如果它被用作counter，则 `GAUGE` 将会被误导。

帮助文档

当你转换度量指标时，用户能够跟踪原始内容记忆导致该转换的规则是有用的。以收件人/导出者的身份，应用程序的任何规则ID和原始度量的名称/详细信息记录到帮助文档，会极大地帮助用户。
Prometheus不喜欢一个度量指标名称有不同帮助文档。如果你正在使用来自多个其他系统的一个度量指标，请选择其中一个来放置帮助文档。
例如：SNMP导出器使用OID，JMX导出器放入一个样例mBean名称中。HAProxy exporter有手写字符串。node exporter有大量的例子可以使用。

放弃无用的统计数据

某些检测系统提供了一些度量指标，包含1m/5m/15m速率、程序运行到现在的平均速率（例如：在dropwizard指标中统计为平均值）、最小值、最大值和标准偏差。
这些检测系统的统计度量指标数据都应该被丢弃，因为Prometheus提供了同样的一套统计数据，数据类型为Summary，而且统计的样本数据更加准确，所以不需要再在写导出器时统计这类数据。
Quantiles涉及的相关issue，你可以选择丢弃或者将其放在Summary中。

.字符串(Dotted strings)

许多监控系统没有标签，而是做成像

`my.class.path.mymetric.labelvalue1.labelvalue2.labelvalue3` 这样。
graphite和statsd的导出器采用了一种使用配置文件来达到带有标签的目的。其他exporters也应该这样做。它目前仅在Go中实现，并将受益于将其分解为单独的库。

Collectors

当在导出器中实现收集器的功能模块时，你永远不要"使用通用且直接的检测方案，然后在每次抓取后更新度量指标样本数据"。
我们宁愿在抓取度量指标数据时，创建新的度量指标。在Go中，在Update()方法中调用MustNewConstMetric完成此操作。对于Python，请参与https://github.com/prometheus/client_python#custom-collectors, 对于Java，在Collector方法中生成列表<MetricFamilySamples>, 请参与StardardExports.java作为示例。
我们在抓取数据时采用创建度量指标的方案，主要有两个原因： 1. 如果在同一时刻发生两次抓取，这样在度量指标数据时会发生资源竞争问题；2. 如果一个度量指标的标签值消失后，更新度量指标数据，那么空标签数据还是会被导出。
通过直接的检测来判断你写的导出器是否是ok的。例如：总字节数在所有的数据抓取传输或者调用时由导出器执行，例如：blackbox exporters和snmp exporter，他们关联了多个目标实例，所以这些只能在一个vanilla `/metrics` 调用上，而不是在特定目标上。

关于获取度量指标本身

有时候你想要导出关于一次抓取本身相关的度量指标数据时，例如：这次抓取花费了多长时间，处理了多少记录。
这些数据的数据类型应该为gauges，数据指标名称以导出器名称为前缀，例如：
`jmx_scrape_duration_seconds` 。通常 `_exporter` 被排除（如果exporter仅仅作为collector使用，绝对排除它）。

Machine & Process metrics（硬件，进程度量指标）

许多系统（如：elasticsearch）提供了一些硬件度量指标，如：cpu，内存和文件系统信息。Prometheus生态中的node导出器已经提供了这些度量指标的检测方案，所以不需要在写导出器时实现这些硬件度量指标的检测或者获取方案。
在Java世界中，许多检测框架提供了进程级别和JVM级别的统计测量方案，例如：CPU占用率，GC次数等。Java客户端和JMX导出器已经通过形如DefaultExports.java的机制，可以获取这些度量指标，所以也不再需要在写导出器时再实现一套这样的方案。
与其他语言类似。

Deployment部署

每个导出器在实例所在的服务器上部署并监控，这意味着如果你想要haproxy，则需要在实例所在的服务器上运行 `haproxy_exporter` 导出器进程。对于每个有mesos从节点的服务器上，你需要在这个服务器上运行一个

mesos导出器进程（如果在这台服务器上还有mesos主节点，则还需要再启动一个mesos导出器进程服务）
这背后的理论是，对于在所有实例上的导出器进程，我们需要获取其上的度量指标样本数据的话，这意味着在Prometheus监控系统上需要提供服务发现机制，而不是由导出器提供。Prometheus监控系统有目标实例的相关信息的优点，主要在于它允许用户用blackbox导出器探测你的服务。
有两个例外：
第一个是被监控的实例所在的服务器位置是完全不关心的。例如：SNMP、blackbox和IPMI。IPMI和SNMP作为设备是有效地黑盒，它是不可能运行代码的（如果你能够在其上运行node导出器代替，那会更好），blackbox作为你监控像DNS名称这样的，完全不需要运行代码）。但是Prometheus仍然应该需要做服务发现机制，通过传输被抓取的目标实例。
请注意，目前只有使用python和Java客户端库编写这种类型的exporter（在Go中编写的blackbox exporter手工执行文本格式，请不要这样做）。
第二个是，你从一个系统拉取一个随机服务实例的一些统计时，这个服务器的位置是你完全不关心的。考虑一种情况，你想要通过Mysql从节点运行一些商业查询，然后导出数据。这时候使用一个导出器和Mysql从节点通信是最佳方案。
当你监控一个带有master选取的系统时，也不能用Prometheus服务发现机制。这种情况下，我们应该监控每个实例服务，并通过Prometheus和目标系统的主节点通信。这里并不总是准确的，改变目标实例也可能会在Prometheus监控系统下造成数据错乱。

调度

当Prometheus服务抓取度量指标数据时，度量指标才能够从实例中被抓取。导出器不应该设定自己的时间去抓取目标数据，而是由Prometheus监控系统统一管理和下放。即所有的抓取是同步的。如果你需要时间戳，你可以通过pushgateway代替
如果一个度量指标的数据获取需要花费大量时间和代价，那么暂时先缓存这些数据也是可以接受的。它应该在 `HELP` 中说明
在Prometheus中默认的抓取度量指标数据的超时设置为10s。如果你的导出器超过了这个时间长度，你需要在你的导出器用户文档中明确指出。

推送

一些应用程序和监控系统仅仅推送几个度量指标，如：statsd, grphite和collected.
有两点需要考虑：
第一点， 你的度量指标什么时间过期？Collected和Graphite导出器定时导出度量指标数据，但是当他们停止工作后，我们也想要停止提供这些度量指标。Collected包含我们经常使用的过期时间，而Graphite则不会，在导出器中它是一个标志。
Statsd有很大不同，它处理的是事件，而不是度量指标。最好的数据模型是在服务所在的目标实例上运行一个Statsd导出器，当应用程序重启后，这个导出器也重启，并清空度量指标数据。
第二点，这些应用程序和监控系统允许用户发送delta或者原生计数器。你应该尽可能地依赖原生计数器，这是

Prometheus的通用模型。

对于服务级别的度量指标（例如：服务级别的批量任务），你应该用导出器把这些度量指标数据推送到Push gateway中，当推送完成后退出。对于实例级别的度量指标，目前还没有明确的模式。选项不是滥用node导出器的textfile收集器，而是依赖当前内存状态。

抓取失败

当前有两种模式，当导出器同实例服务通信没有响应或者其他问题时，称之为抓取失败。

第一种，返回响应吗：5xx错误。

第二种，有一个度量指标名称为 `myexporter_up` (例如： `haproxy_up`), 判断一个导出器是否工作时通过这个度量指标变量的值0/1状态。0：表示不工作；1：表示工作。

第二种模式是比较好的，当抓取失败时，我们能够通过一些有用的度量指标数据判断导出器服务的当前工作状态，例如：haproxy导出器提供了进程统计信息，第一种对于用户来说，更加容易处理，但是 `up` 是非常通用的处理方式（因为你不能辨别出事导出器服务有问题，还是应用程序有问题）

登录页面

如果用户访问 `http://yourexporter/` 时返回一个带有导出器名称的简单页面， 并且通过 `/metrics` 链接到Prometheus系统中国，这对用户是非常友好的。

端口

在一台服务器上用户可以有多个导出器和Prometheus组件，因此有一个唯一的端口变得非常容易 https://github.com/prometheus/prometheus/wiki/Default-port-allocations ， 这个是我们官方的端口规划。

宣布

一旦你对外准备宣布你写的导出器时，请发送一封邮件，并且发送一个PR到这个可用导出器的列表

导出格式

提供的数据传输格式 exposition formats

Prometheus实现了两种不同的数据传输格式，客户端可以使用检测的度量指标给Prometheus服务，第一种：是基于文本的简单格式, 第二种：更有效和更强大的协议缓冲格式。Prometheus服务和客户端通过HTTP通信获取要使用的是哪种协议。一个服务比较倾向于接收第二种更有效和更强大的协议缓冲格式。但是如果客户端不支持前者，数据传输格式会回退到第二种基于文本的简单格式
大多数用户应该使用已经存在的客户库，且这些客户库已经实现了两种数据传输格式。

Format v0.0.4

v0.0.4是当前度量指标的数据传输格式版本
到这个版本，Prometheus有两种备用的数据传输格式：基于协议缓冲的格式和文本格式。客户端必须支持这两种备用格式中的一种。
另外，客户端可选择性地提供不被Prometheus服务理解的其他数据传输文本格式。他们仅仅是为了易于理解和方便调试。强烈建议客户端库至少一种可读的格式。有一种情况: 在HTTP头部的Content-Type内容，如果客户库不能理解, 则易于理解的数据传输协议应该是候选项，v0.0.4的数据传输协议容易被理解，所以它是一个很好的数据传输格式候选者（并且也被Prometheus理解）。

格式变体比较

	协议缓冲格式	文本格式
开始时间	2014.4月	2014.4月
传输协议	HTTP	HTTP
编码	- 32-bit varint-encoded record length-delimited	utf-9, \n行结尾
	- Protocol Buffer messages of type	
	- io.prometheus.client.MetricFamily	
HTTP Content-Type	application/vnd.google.protobuf; proto=io.prometheus.client.MetricFamily; encoding=delimited	text/plain; version=0.0.4 (A missing version value will lead to a fall-back to the most recent text format version.)

Optional HTTP Content-Encoding	gzip	gzip
优点	- 跨平台	-可读性好
	- Size	- 易于组合，特别适用于简单情况（无需嵌套）
	- 编解码代价小	- 逐行阅读（处理类型提示和文本字符串外）
	- 严格的范式	
	- 支持链接和理论流（仅服务端行为需要更改）	
限制	- 不具有可读性	- 信息不全
		- 类型和文档格式不是语法的组成部分，意味着很少到不存在的度量契约验证
		- 解析代价大
支持度量指标原子性	- Counter	- Counter
	- Gauge	- Gauge
	- Histogram	- Histogram
	- Summary	- Summary
	- Untyped	- Untyped
兼容性	- 低于v.0.0.3无效	无
	- v0.0.4有效	

Protocol buffer format details 协议缓冲区格式详细信息

在重复数据传输协议中，协议缓冲区的可重复排序是优选的，但不是必需的，即如果计算成本过高，则不排序同一个数据传输协议中 `MetricFamily` 的必须有一个唯一的名称。且 `MetricFamily` 中的每个度量指标都必须有一组唯一的 `LabelPair` 字段。否则，这种嵌入行为是未定义的。

Text format details 文本格式详解

协议是面向行的。换行字符(\n)分隔行。最后一行必须以换行字符结束。空行被忽略。
在一行内，令牌可以被任何数量的空格和/或制表符分开（如果它们不想和以前的token合并，则必须至少分开一个）。头尾部的空白被忽略。
具有"#"作为第一个费空格字符的行是注释。它们被忽略，除非“#”之后的一个令牌是HELP或TYPE。这些行被视为如下：如果令牌是HELP，则至少需要一个令牌，这是度量指标名称。所有剩余的令牌都被认为是该度量指标名称的docstring。HELP行可以包含任何UTF-8字符序列（度量名称后）。但反斜杠和换行字符必须分别转移为

\和\n。相同的度量名称只能有一个HELP行。
如果令牌是TYPE，则预期只有两个令牌。第一个是度量名称，第二个是计数器，规格，直方图，摘要或无类型，定义该名称的度量标准的类型。相同的度量名称只能有一个TYPE行。用于度量名称的TYPE行必须出现在为该度量名称报告的第一个样本之前。如果度量名称没有TYPE行，则该类型将设置为无类型。剩余行用以下语法描述样本，每行一个(EBNF)：

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

`metric_name` 和 `label_name` 具有普通的Prometheus表达式语言限制。 `label_value` 可以是UTF-8字符的任何序列，但反斜杠，双引号和换行字符必须分别转义为\，\“和\ n，value为浮点数和时间戳 一个int64（从时代以来的毫秒，即1970-01-01 00:00:00 UTC，不包括闰秒），由Go strconv软件包（见函数ParseInt和ParseFloat）表示，特别是Nan，+ Inf和 -Inf是有效值。
给定度量的所有行必须作为一个不间断的组提供，可选的HELP和TYPE行首先（不是特定的顺序）。 除此之外，重复展示的可重复排序是优选的，但不是必需的，即不计算计算成本是否可以排序。
每行必须具有度量名称和标签的唯一组合。 否则，嵌入行为是未定义的。
`histogram` 和 `summary` 类型很难以文本格式表示。 适用以下约定：

- 名为x的摘要或直方图的样本总和作为名为x_sum的单独样本给出。
- 名为x的摘要或直方图的样本计数作为名为x_count的单独样本给出。
- 名为x的摘要的每个分位数作为具有相同名称x和标签{quantile =“y”}的单独样本行给出。
- 名为x的直方图的每个桶计数作为单独的样本行（名称为x_bucket）和一个标签{le =“y”}（其中y是存储桶的上限）给出。
- 直方图必须带有{le =“+ Inf”}的存储桶。 其值必须与x_count的值相同。
- 直方图的桶和总结的分位数必须以其标签值的增加数字顺序（分别为le或分位数标签）出现。
  另见下面的例子。

```
# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total counter
http_requests_total{method="post",code="200"} 1027 1395066363000
http_requests_total{method="post",code="400"}    3 1395066363000

# Escaping in label values:
msdos_file_access_time_seconds{path="C:\\DIR\\FILE.TXT",error="Cannot find file
:\n\"FILE.TXT\""} 1.458255915e9

# Minimalistic line:
metric_without_timestamp_and_labels 12.47
```

```
# A weird metric from before the epoch:
something_weird{problem="division by zero"} +Inf -3982045

# 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="0.5"} 129389
http_request_duration_seconds_bucket{le="1"} 133988
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{quantile="0.9"} 9001
rpc_duration_seconds{quantile="0.99"} 76656
rpc_duration_seconds_sum 1.7560473e+07
rpc_duration_seconds_count 2693
```

可选文本表示 Optional Text Representation

以下三种可选文本格式仅供使用，且不被Prometheus理解。因此，他们的定义可能会有些随意。客户端库可能支持或可能不支持这些格式。工具不应该依赖这些格式。

1. HTML：此格式由HTTP Content-Type头部请求，其值为text/html。在浏览器中查看的指标是一个“pretty”渲染。虽然生成客户端在技术上完全免费组装HTML，但客户端库之间的一致性应该是针对的。
2. 协议缓冲区文本格式：与协议缓冲区格式相同，但以文本形式。它由以文本格式（也称为“调试字符串”）连接的协议消息组成，由另外的新行字符分隔（即在协议消息之间有空行）。请求格式为协议缓冲区格式，但HTTP Content-Type头文件中的编码设置为文本。
3. 协议缓冲区紧凑文本格式：与（2）相同，但使用紧凑文本格式而不是普通文本格式。紧凑文本格式将整个协议消息放在一行上。协议消息仍然以新的行字符分隔，但是不需要“空行”来进行分离。（每行只需一个协议消息）。格式被请求为协议缓冲区格式，但HTTP Content-Type头中的编码设置为compact-text。

历史版本

有关历史格式版本的详细信息，请参阅旧版客户端数据展示格式文档。

操作

配置

配置configuration

Prometheus可以通过命令行参数和配置文件来配置它的服务参数。命令行主要用于配置系统参数（例如：存储位置，保留在磁盘和内存中的数据量大小等），配置文件主要用于配置与抓取任务和任务下的实例相关的所有内容, 并且加载指定的抓取规则file。

可以通过运行 `prometheus -h` 命令, 查看Prometheus服务所有可用的命令行参数，

Prometheus服务可以reload它的配置。如果这个配置错误，则更改后的配置不生效。配置reolad是通过给Prometheus服务发送信号量 `SIGHUP` 或者通过http发送一个post请求到 `/-/reload` 。这也会重载所有配置的规则文件(rule files)。

配置文件 Configuration file

使用 `-config.file` 命令行参数来指定Prometheus启动所需要的配置文件。

这个配置文件是YAML格式， 通过下面描述的范式定义, 括号表示参数是可选的。对于非列表参数，这个值被设置了默认值。

通用占位符由下面定义：

- `\<boolean\>` : 一个布尔值，包括 `true` 或者 `false` .
- `\<duration\>` : 持续时间，与正则表达式 `[0-9]+(ms|smhdwy)` 匹配
- `\<labelname\>` : 一个与正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*` 匹配的字符串
- `\<labelvalue\>` : 一个为unicode字符串
- `\<filename\>` : 当前工作目录下的有效路径
- `\<host\>` : 一个包含主机名或者IP地址，并且可以带上一个非必需的端口号的有效字符串
- `\<path\>` : 一个有效的URL路径
- `\<scheme\>` : 一个可以是 `http` 或者 `https` 的字符串
- `\<string\>` : 一个正则表达式字符串

其他的占位符被分开指定：

一个有效的配置文件示例。

全局配置指定的参数，在其他上下文配置中是生效的。这也默认这些全局参数在其他配置区域有效。

```
global:
    # 抓取目标实例的频率时间值, 默认10s
    [ scrape_interval: <duration> | default = 10s ]

    # 一次抓取请求超时时间值, 默认10s
    [ scrape_timeout: <duration> | default = 10s ]
```

```
    # 执行配置文件规则的频率时间值, 默认1m
    [ evaluation_interval: <duration> | default=1m ]

    # 当和外部系统通信时(federation, remote storage, Alertmanager), 这些标签会增加到
度量指标数据中
    external_labels:
        [ <labelname>: <labelvalue> ... ]

# 规则文件指定规则文件路径列表。规则和警报是从所有匹配的文件中读取的
rule_files:
    [ - <filepath_glob> ...]

# 抓取配置的列表
scrape_configs:
    [ - <scrape_config> ... ]

# 警报设置
alerting:
    alert_relabel_configs:
    [  - <relabel_config> ... ]
    alertmanagers:
    [ - <alertmanager_config> ... ]

# 设置涉及到未来的实验特征
remote_write:
    [url: <string> ]
    [ remote_timeout: <duration> | default = 30s ]
    tls_config:
    [ <tls_config> ]
    [proxy_url: <string> ]
    basic_auth:
    [user_name: <string> ]
    [password: <string>  ]
    write_relabel_configs:
    [ - <relabel_config> ... ]
```

<scrape_config>

`<scrape_config>` 区域指定了目标列表和目标下的配置参数, 这些配置参数描述了如何抓取度量指标数据。

通常，一个scrape_config只指定一个job，但是可以改变，一个scrape_config可以指定多个job，每个job下有多个targets

通过 `static_configs` 参数静态指定要监控的目标列表，或者使用一些服务发现机制发现目标。

另外， `relabel_configs` 允许在获取度量指标数据之前，对任何目标和它的标签进行进一步地修改。

```
# 默认下任务名称赋值给要抓取的度量指标
job_name: <job_name>
```

```
# 从这个任务中抓取目标的频率时间值
[ scrape_interval: <duration> | default= <global_config.scrape_interval>]

# 当抓取这个任务的所有目标时，超时时间值
[ scrape_timeout: <duration> | default = <global_config.scrape_timeout> ]

# 从目标列表中抓取度量指标的http资源路径，默认为/metrics
[ metrics_path: <path> | default = /metrics ]

# honor_labels controls how Prometheus handles conflicts between would labels t
hat are already present in scraped data and labels that Prometheus would attach
 server-side ("job" and "instance" labels, manually configured target  labels,
and labels generated by service discovery implementations).
# If honor_labels is set to "true", label conflicts are resolved by keeping lab
el
# values from the scraped data and ignoring the conflicting server-side labe# l
s. If honor_labels is set to "false", label conflicts are resolved by ren# amin
 conflicting labels in the scraped data to "exported_<original-label>" (for exa
mple "exported_instance", "exported_job") and then attaching server-side labels
. This is useful for use cases such as federation, where all label#s specified
in the target should be preserved. Note that any globally configured "external_
labels" are unaffected by this
# setting. In communication with external systems, they are always applied
# only when a time series does not have a given label yet and are ignored other
wise.
[ honor_labels: <boolean> | default = false ]

# 配置请求的协议范式，默认为http请求
[ scheme: <scheme> | default = http ]

# 可选的http url参数
params:
    [ <string>:[<string>, ...]]

# 在`Authorization`头部设置每次抓取请求的用户名和密码
basic_auth:
[username: <string>]
[password: <string>]

# Sets the `Authorization` header on every scrape request with
# the configured bearer token. It is mutually exclusive with `bearer_token_file
`.
[ bearer_token: <string> ]

# Sets the `Authorization` header on every scrape request with the bearer token
 read from the configured file. It is mutually exclusive with `bearer_token`.
[ bearer_token_file: /path/to/bearer/token/file ]

# 配置抓取请求的TLS设置
```

```
tls_config:
  [ <tls_config> ]

# 可选的代理URL
[ proxy_url: <string> ]

# 微软的Azure服务发现配置列表
azure_sd_configs:
  [ - <azure_sd_config> ... ]

# Consul服务发现配置列表
consul_sd_configs:
  [ - <consul_sd_config> ... ]

# DNS服务发现配置列表
dns_sd_configs:
  [ - <dns_sd_config> ... ]

# 亚马逊EC2服务发现的配置列表
ec2_sd_configs:
  [ - <ec2_sd_config> ... ]

# 文件服务发现配置列表
file_sd_configs:
  [ - <file_sd_config> ... ]

# google GCE服务发现配置列表
gce_sd_configs:
  [ - <gce_sd_config> ... ]

# Kubernetes服务发现配置列表
kubernetes_sd_configs:
  [ - <kubernetes_sd_config> ... ]

# Marathon服务发现配置列表
marathon_sd_configs:
  [ - <marathon_sd_config> ... ]

# AirBnB的Nerve服务发现配置列表
nerve_sd_configs:
  [ - <nerve_sd_config> ... ]

# Zookeeper服务发现配置列表
serverset_sd_configs:
  [ - <serverset_sd_config> ... ]

# Triton服务发现配置列表
triton_sd_configs:
  [ - <triton_sd_config> ... ]
```

```
# 静态配置目标列表
static_configs:
  [ - <static_config> ... ]

# 抓取之前的标签重构配置列表
relabel_configs:
  [ - <relabel_config> ... ]

# List of metric relabel configurations.
metric_relabel_configs:
  [ - <relabel_config> ... ]

# Per-scrape limit on number of scraped samples that will be accepted.
# If more than this number of samples are present after metric relabelling
# the entire scrape will be treated as failed. 0 means no limit.
[ sample_limit: <int> | default = 0 ]
```

记住：在所有获取配置中 `<job_name>` 必须是唯一的。

<tls_config>

`<tls_config>` 允许配置TLS连接。

```
# CA证书
[ ca_file: <filename> ]

# 证书和key文件
[ cert_file: <filename> ]
[ key_file: <filename> ]

# ServerName extension to indicate the name of the server.
# http://tools.ietf.org/html/rfc4366#section-3.1
[ server_name: <string> ]

# Disable validation of the server certificate.
[ insecure_skip_verify: <boolean> ]
```

<azure_sd_config>

Azure SD正处于测试阶段：在未来的版本中，仍然可能对配置进行实质性修改

Azure SD配置允许从Azure虚拟机中检索和获取目标。

下面的测试标签在relabeling期间在目标上仍然是可用的：

- `__meta_azure_machine_id`：机器ID
- `__meta_azure_machine_location`：机器运行的位置
- `__meta_azure_machine_name`：机器名称

- `__meta_azure_machine_private_ip`：机器的内网IP
- `__meta_azure_machine_resource_group`：机器的资源组
- `__meta_azure_tag_<tagname>`：机器的每个tag值

对于Azure发现，看看下面的配置选项：

```
# The information to access the Azure API.
# The subscription ID.
subscription_id: <string>
# The tenant ID.
tenant_id: <string>
# The client ID.
client_id: <string>
# The client secret.
client_secret: <string>

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 300s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]
```

<consul_sd_config>

Consul服务发现配置允许从Consul's Catalog API中检索和获取目标。

下面的meta标签在relabeling期间在目标上仍然是可用的：

- `__meta_consul_address`：目标地址
- `__meta_consul_dc`：目标的数据中心名称
- `__meta_consul_node`：目标的节点名称
- `__meta_consul_service_address`：目标的服务地址
- `__meta_consul_service_id`：目标的服务ID
- `__meta_consul_service_port`：目标的服务端口
- `__meta_consul_service`：这个目标属于哪个服务名称
- `__meta_consul_tags`：由标签分隔符链接的目标的标签列表

```
# 下面配置是访问Consul API所需要的信息
server: <host>
[ token: <string> ]
[ datacenter: <string> ]
[ scheme: <string> ]
[ username: <string> ]
[ password: <string> ]
```

```
# 指定对于某个目标的服务列表被检测， 如果省略，所有服务被抓取
services:
  [ - <string> ]

# The string by which Consul tags are joined into the tag label.
[ tag_separator: <string> | default = , ]
```

注意：用于获取目标的IP和PORT，被组装到 `<__meta_consul_address>:<__meta_consul_service_port>` 。然而，在一些Consul创建过程中，这个相关地址在 `__meta_consul_service_address` 。在这些例子中，你能使用relabel特性去替换指定的 `__address__` 标签。

<dns_sd_config>

一个基于DNS的服务发现配置允许指定一系列的DNS域名称，这些DNS域名被周期性地查询，用来发现目标列表。这些DNS服务是从 `/etc/resolv.conf` 获取的。
这些服务发现方法仅仅支持基本的DNS A，AAAA和SRV记录查询，但不支持在RFC6763中指定更高级的DNS-SD方案。
在重构标签阶段，这个标签 `__meta_dns_name` 在每一个目标上都是可用的，并且会设置生产发现的目标到记录名称中。

```
# 将被查询的DNS域名列表
names:
  [ - <domain_name> ]

# 要执行DNS查询类型，默认为SRV， 其他方式：A、AAAA和SRV
[ type: <query_type> | default = 'SRV' ]

# 如果查询类型不是SRV，这端口被使用
[ port: <number>]

# 刷新周期，默认30s
[ refresh_interval: <duration> | default = 30s ]
```

`<domain_name>` 必须是一个有效的DNS域名。 `<query_type>` 必须是 `SRV, A, AAAA` 三种之一。

<ec2_sd_config>

EC2 SD配置允许从AWS EC2实例中检索目标。默认情况下用内网IP地址，但是在relabeling期间可以改变成公网ID地址。
下面meta标签在relabeling期间在目标上是可用的：

- `__meta_ec2_availability_zone` ：正在运行的实例的可用域。

- `__meta_ec2_instance_id`：EC2的实例ID
- `__meta_ec2_instance_state`：EC2的实例状态
- `__meta_ec2_instance_type`：EC2的实例类型
- `__meta_ec2_private_ip`：如果存在，表示内网IP的地址
- `__meta_ec2_public_dns_name`：如果可用，表示实例的公网DNS名称
- `__meta_ec2_public_ip`：如果可用，表示实例的公网IP地址
- `__meta_ec2_subnet_id`：如果可用，表示子网IDs的列表。
- `__meta_ec2_tag_<tagkey>`：这个实例的tag值
- `__meta_ec2_vpc_id`：如果可用，表示正在运行的实例的VPC的ID

对于EC2 discovery，看看下面的配置选项：

```
# 访问EC2 API的信息

# AWS域
region: <string>

# AWS API keys. 如果空白, 环境变量`AWS_ACCESS_KEY_ID`和`AWS_SECRET_ACCESS_KEY`可以
被使用
[ access_key: <string> ]
[ secret_key: <string> ]
# Named AWS profile used to connect to the API.
[ profile: <string> ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]
```

<file_sd_config>

基于文件的服务发现提供了一些通用方法去配置静态目标，以及作为插件自定义服务发现机制的接口。

它读取包含零个或者多个 `<static_config>s` 的一些文件。通过磁盘监视器检测对所有定义文件的更改，并立即应用。文件可能以YAML或JSON格式提供。只应用于形成良好目标群体的变化。

这个JSON文件必须包含静态配置的列表，使用这个格式：

```
[
  {
    "targets": [ "<host>", ... ],
    "labels": {
      "<labelname>": "<labelvalue>", ...
```

```
    }
  },
  ...
]
```

文件内容也可以通过周期性刷新时间重新加载。

在标签重构阶段，每个目标有一个meta标签 `__meta_filepath` 。它的值被设置成从目标中提取的文件路径。

```
# Patterns for files from which target groups are extracted.
files:
  [ - <filename_pattern> ... ]

# Refresh interval to re-read the files.
[ refresh_interval: <duration> | default = 5m ]
```

`filename_pattern` 可以是以 `.json, .yml, .yaml` 结尾。最后路径段可以包含单个 `*` ，它匹配任何字符顺序，例如: `my/path/tg_*.json` 。

在 `v0.20` ， `names` ：用 `files:` 代替。

<gce_sd_config>

GCE SD在测试中：在将来版本中，配置可能会有实质性变化。

从GCP GCE实例中，GCE SD配置允许检索和获取目标。这个内网IP地址被默认使用，但是在relabeling期间，这个公网IP地址可能会发生变化。

在relabeling期间，下面的meta标签在目标上是可用的：

- `__meta_gce_instance_name` ：实例名称
- `__meta_gce_metadata_<name>` ：实例每一个metadata项
- `__meta_gce_network` ：实例的网络
- `__meta_gce_private_ip` ：实例的内网IP
- `__meta_gce_project` ：正在运行的GCP项目
- `__meta_gce_public_ip` ：如果存在，表示GCP的公网IP地址
- `__meta_gce_subnetwork` ：实例的子网
- `__meta_gce_tags` ：实例的tag列表
- `__meta_gce_zone` ：正在运行的实例的GCE区域

对于GCE discovery，看看下面的配置选项：

```
# The information to access the GCE API.
```

```
# The GCP Project
project: <string>

# The zone of the scrape targets. If you need multiple zones use multiple
# gce_sd_configs.
zone: <string>

# Filter can be used optionally to filter the instance list by other criteria
[ filter: <string> ]

# Refresh interval to re-read the instance list
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# The tag separator is used to separate the tags on concatenation
[ tag_separator: <string> | default = , ]
```

Google Cloud SDK默认客户端通过查找一下位置发现凭据，优先选择找到的第一个位置：

1. 由GOOGLE_APPLICATION_CREENTIALS环境变量指定的JSON文件
2. 一个JSON文件在大家都熟悉的路径下：$HOME/.config/glooud/application_default_credentials.json
3. 从GCE元数据服务器获取

如果Prometheus运行在GCE上，关联这个正在运行的实例的服务账号，应该至少可以从计算资源上有读取数据的权限。如果运行在GCE外面，需要确保创建一个合适的服务账号，并把证书文件放置在指定的某个地方。

<kubernets_sd_config>

Kubernets SD在测试中，在将来的版本中，配置可能会有实质性的变化
从Kubernetes's REST API上，Kubernets SD配置允许检索和获取目标，并且始终保持与集群状态同步。
下面 `role` 类型中的任何一个都能在发现目标上配置：

节点node

这个 `node` 角色发现带有地址的每一个集群节点一个目标，都指向Kublelet的HTTP端口。这个目标地址默认为Kubernetes节点对象的第一个现有地址，地址类型为
`NodeInernalIP, NodeExternalIP, NodeLegacyHostIP和NodeHostName` 。
可用的meta标签：

- `__meta_kubernetes_node_name` ：节点对象的名称
- `__meta_kubernetes_node_label_<labelname>` ：节点对象的每个标签
- `__meta_kubernetes_node_annotation_<annotationname>` ：节点对象的每个注释

_meta_kubernetes_node_address<address_type>: 如果存在，每一个节点对象类型的第一个地址

另外，对于节点的 `instance` 标签，将会被设置成从API服务中获取的节点名称。

服务service

对于每个服务每个服务端口， `service` 角色发现一个目标。对于一个服务的黑盒监控是通常有用的。这个地址被设置成这个服务的Kubernetes DNS域名, 以及各自的服务端口。
可用的meta标签：

- `__meta_kubernetes_namespace` : 服务对象的命名空间
- `__meta_kubernetes_service_name` : 服务对象的名称
- `__meta_kubernetes_service_label_<labelname>` : 服务对象的标签。
- `__meta_kubernetes_service_annotation_<annotationname>` : 服务对象的注释
- `__meta_kubernetes_service_port_name` : 目标服务端口的名称
- `__meta_kubernetes_service_port_number` : 目标服务端口的数量
- `__meta_kubernetes_service_port_portocol` : 目标服务端口的协议

pod

`pod` 角色发现所有的pods，并暴露它们的容器作为目标。对于每一个容器的声明端口，单个目标被生成。 如果一个容器没有指定端口，每个容器的无端口目标都是通过relabeling手动添加端口而创建的。
可用的meta标签：

- `__meta_kubernetes_namespace` : pod对象的命名空间
- `__meta_kubernetes_pod_name` : pod对象的名称
- `__meta_kubernetes_pod_ip` : pod对象的IP地址
- `__meta_kubernetes_pod_label_<labelname>` : pod对象的标签
- `__meta_kubernetes_pod_annotation_<annotationname>` : pod对象的注释
- `__meta_kubernetes_pod_container_name` : 目标地址的容器名称
- `__meta_kubernetes_pod_container_port_name` : 容器端口名称
- `__meta_kubernetes_pod_container_port_number` : 容器端口的数量
- `__meta_kubernetes_pod_container_port_protocol` : 容器端口的协议
- `__meta_kubernetes_pod_ready` : 设置pod ready状态为true或者false
- `__meta_kubernetes_pod_node_name` : pod调度的node名称
- `__meta_kubernetes_pod_host_ip` : 节点对象的主机IP

endpoints端点

`endpoints` 角色发现来自于一个服务的列表端点目标。对于每一个终端地址，一个目标被一个port发现。如果这个终端被写入到pod中，这个节点的所有其他容器端口，未绑定到端点的端口，也会被目标发现。

可用的meta标签：

- `__meta_kubernetes_namespace`：端点对象的命名空间
- `__meta_kubernetes_endpoints_name`：端点对象的名称
- 对于直接从端点列表中获取的所有目标，下面的标签将会被附加上。
  - `__meta_kubernetes_endpoint_ready`：endpoint ready状态设置为true或者false。
  - `__meta_kubernetes_endpoint_port_name`：端点的端口名称
  - `__meta_kubernetes_endpoint_port_protocol`：端点的端口协议
- 如果端点属于一个服务，这个角色的所有标签：服务发现被附加上。
- 对于在pod中的所有目标，这个角色的所有表掐你：pod发现被附加上

对于Kuberntes发现，看看下面的配置选项：

```
# The information to access the Kubernetes API.

# The API server addresses. If left empty, Prometheus is assumed to run inside
# of the cluster and will discover API servers automatically and use the pod's
# CA certificate and bearer token file at /var/run/secrets/kubernetes.io/servic
eaccount/.
[ api_server: <host> ]

# The Kubernetes role of entities that should be discovered.
role: <role>

# Optional authentication information used to authenticate to the API server.
# Note that `basic_auth`, `bearer_token` and `bearer_token_file` options are
# mutually exclusive.

# Optional HTTP basic authentication information.
basic_auth:
  [ username: <string> ]
  [ password: <string> ]

# Optional bearer token authentication information.
[ bearer_token: <string> ]

# Optional bearer token file authentication information.
[ bearer_token_file: <filename> ]

# TLS configuration.
tls_config:
  [ <tls_config> ]
```

`<role>` 必须是 `endpoints`， `service`， `pod` 或者 `node` 。

关于Prometheus的一个详细配置例子，见[路径]

（https://github.com/prometheus/prometheus/blob/master/documentation/examples/prometheus-kubernetes.yml）

你可能希望查看第三方的Prometheus操作符，它可以自动执行Kubernetes上的Prometheus设置。

<marathon_sd_config>

Marathon SD正在测试中：在将来的版本中配置可能会有实质性的变化

Marathon SD配置使用MarathonREST API允许检索和获取目标。Prometheus将会定期地检查当前运行的任务REST端点，以及对每个app创建一个目标组，这个app至少有一个健康的任务。

在relabeling期间，下面的meta标签在目标机上是可用的：

- `__meta_marathon_app`：app的名称
- `__meta_marathon_image`：正在使用的Docker镜像名称
- `__meta_marathon_task`：Mesos任务ID
- `__meta_marathon_app_label_<labelname>`：附加在app上的Marathon标签

对于Marathon发现，详见下面的配置选项：

```
# List of URLs to be used to contact Marathon servers.
# You need to provide at least one server URL, but should provide URLs for
# all masters you have running.
servers:
  - <string>

# Polling interval
[ refresh_interval: <duration> | default = 30s ]
```

默认情况下，在Markdown的每个列出的app会被Prometheus抓取。如果不是所有提供Prometheus度量指标，你能使用一个Marathon标签和Prometheus relabeling去控制实际过程中被获取的实例。默认情况下所有的app也会以Prometheus系统中的一个任务的形式显示出来，这可以通过使用relabeling改变这些。

<nerve_sd_config>

从存储在Zookeeper中的AirBnB's Nerve上，Nerve SD配置允许检索和获取目标。

在relabeling期间，下面的meta标签在目标上是可用的：

- `__meta_nerve_path`：在Zookeeper集群中的端节点全路径
- `__meta_nerve_endpoint_host`：端点的IP
- `__meta_nerve_endpoint_port`：端点的端口
- `__meta_nerve_endpoint_name`：端点的名称

```
# The Zookeeper servers.
```

```
servers:
  - <host>
# Paths can point to a single service, or the root of a tree of services.
paths:
  - <string>
[ timeout: <duration> | default = 10s ]
```

<serverset_sd_config>

Serverset SD配置允许检索和获取从存储在Zookeeper中的Serversetsd的目标。Servesets由Finagle和Aurora经常使用。

在relabeling期间，下面的meta标签在目标上是可用的：

- `__meta_serverset_path`：在zookeeper里的serverset成员的全路径
- `__meta_serverset_endpoint_host`：默认端点的host
- `__meta_serverset_endpoint_port`：默认端点的端口
- `__meta_serverset_endpoint_host_<endpoint>`：给定端点的host
- `__meta_serverset_endpoint_port_<endpoint>`：给定端点的port
- `__meta_serverset_shard`：成员的分片数
- `__meta_serverset_status`：成员的状态

```
# The Zookeeper servers.
servers:
  - <host>
# Paths can point to a single serverset, or the root of a tree of serversets.
paths:
  - <string>
[ timeout: <duration> | default = 10s ]
```

Serverset数据必须是JSON格式，Thrift格式当前不被支持

<triton_sd_config>

** Triton SD正在测试中：在将来的版本中配置可能会有实质性的变化**

Triton SD配置允许从容器监控发现端点的目标中检索和获取。

在relabeling期间，下面的meta标签在目标上是可用的：

- `__meta_triton_machine_id`：目标容器的UUID
- `__meta_triton_machine_alias`：目标容器的别名
- `__meta_triton_machine_image`：目标容器的镜像类型
- `__meta_triton_machine_server_id`：目标容器的服务UUID

```
# The information to access the Triton discovery API.

# The account to use for discovering new target containers.
account: <string>

# The DNS suffix which should be applied to target containers.
dns_suffix: <string>

# The Triton discovery endpoint (e.g. 'cmon.us-east-3b.triton.zone'). This is
# often the same value as dns_suffix.
endpoint: <string>

# The port to use for discovery and metric scraping.
[ port: <int> | default = 9163 ]

# The interval which should should be used for refreshing target containers.
[ refresh_interval: <duration> | default = 60s ]

# The Triton discovery API version.
[ version: <int> | default = 1 ]

# TLS configuration.
tls_config:
  [ <tls_config> ]
```

<static_config>

一个 `static_config` 允许指定目标列表，以及附带的通用标签。在获取配置中指定静态目标是规范的方法

```
# The targets specified by the static config.
targets:
  [ - '<host>' ]

# Labels assigned to all metrics scraped from the targets.
labels:
  [ <labelname>: <labelvalue> ... ]
```

<relabel_config>

Relabeling是一个非常强大的工具，在获取度量指标之前，它可以动态地重写标签集合。 每个获取配置过程中，多个relabeling步骤能够被配置。它们按照出现在配置文件中的顺序，应用到每个目标的标签集中。

最初，除了配置的每个目标标签之外，目标的作业标签设置为相应获取配置的 `job_name` 值，这个 `__address__` 标签设置为目标地址:。在relabeling之后，这个 `instance` 标签默认设置为 `__address__` 标签值。这个 `__scheme__` 和 `__metrics_path__` 标签设置为各自目标的范式和度量指标路径。 `__param_<name>` 标签设置为成为 `<name>` 的第一个传入的URL参数。

另外以 `__meta__` 为前缀的标签在relabeling阶段是可用的。他们由服务发现机制设置。
在relabeling完成之后，由 `__` 开头的标签将会从标签集合从移除。
如果一个relabeling步骤仅仅需要临时地存储标签值（作为后续relabeling步骤的输入），使用以 `__tmp` 为前缀的标签名称。这个前缀需要确保Prometheus本身从没有使用。

```
# The source labels select values from existing labels. Their content is concat
enated
# using the configured separator and matched against the configured regular exp
ression
# 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.
[ 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.
[ replacement: <string> | default = $1 ]

# Action to perform based on regex matching.
[ action: <relabel_action> | default = replace ]
```

`<regex>` 是任何有效的正则表达式，它提供
`replace, keep, drop, labelmap, labeldrop, labelkeep` 动作，正则表达式处于两端。要取消指定正则表达式，请使用。...。
`<relabel_action>` 决定要采取的relabeling动作。

- `replace`：匹配与 `source_labels` 相反的regex。然后，设置 `target_label` 替换 `source_labels`，返回结果包括(${1}, ${2}, ...)。 如果正则表达会不匹配，则不进行任何替换。
- `keep`：放弃与 `source_labels` 标签不匹配的目标
- `drop`：放弃与 `source_labels` 标签匹配的目标
- `hashmod`：将 `target_label` 设置为 `source_labels` 的散列模数
- `labelmap`：匹配所有的标签名称，然后将匹配到的标签值复制为由匹配组引用(${1}, ${2},...) 替换的标签

名称替换为其值

- `labeldrop`：匹配所有的标签名称。然后删除匹配到的标签集合。
- `labelkeep`：匹配所有的标签名称。然后保留匹配到的标签集合。

必须注意 `labeldrop` 和 `labelkeep`，以确保除去标签后，度量指标仍然会被唯一标识。

<alert_relabel_configs>

在警告被发送到Alertmanager之前，警告relabeling应用到alerts。它有相同配置格式和目标relabeling动作。警告relabeling被应用到外部标签。

一个用途是确保HA对Prometheus服务与不同的外部标签发送相同的警告。

<alertmanager_config>

Alertmanager实例的动态发现是处于alpha状态。在将来的版本中配置会发生较大地更改。通过 `-alertmanager.url` 标志使用静态配置

`alertmanager_config` 区域指定了Prometheus服务发送警告的Alertmanager实例。它也提供参数配置与这些Alertmanagers的通信。

Alertmanagers可以通过 `static_configs` 参数静态配置，或者使用服务发现机制动态发现目标。

另外，从发现的实体和使用的API路径， `relabel_configs` 允许从发现的实体列表和提供可使用的API路径中选择路径。这个api path是通过 `__alerts_path__` 标签暴露出来的。

```
# Per-target Alertmanager timeout when pushing alerts.
[ timeout: <duration> | default = 10s ]

# Prefix for the HTTP path alerts are pushed to.
[ path_prefix: <path> | default = / ]

# Configures the protocol scheme used for requests.
[ scheme: <scheme> | default = http ]

# Sets the `Authorization` header on every request with the
# configured username and password.
basic_auth:
  [ username: <string> ]
  [ password: <string> ]

# Sets the `Authorization` header on every request with
# the configured bearer token. It is mutually exclusive with `bearer_token_file
`.
[ bearer_token: <string> ]

# Sets the `Authorization` header on every request with the bearer token
# read from the configured file. It is mutually exclusive with `bearer_token`.
[ bearer_token_file: /path/to/bearer/token/file ]
```

```
# Configures the scrape request's TLS settings.
tls_config:
  [ <tls_config> ]

# Optional proxy URL.
[ proxy_url: <string> ]

# List of Azure service discovery configurations.
azure_sd_configs:
  [ - <azure_sd_config> ... ]

# List of Consul service discovery configurations.
consul_sd_configs:
  [ - <consul_sd_config> ... ]

# List of DNS service discovery configurations.
dns_sd_configs:
  [ - <dns_sd_config> ... ]

# List of EC2 service discovery configurations.
ec2_sd_configs:
  [ - <ec2_sd_config> ... ]

# List of file service discovery configurations.
file_sd_configs:
  [ - <file_sd_config> ... ]

# List of GCE service discovery configurations.
gce_sd_configs:
  [ - <gce_sd_config> ... ]

# List of Kubernetes service discovery configurations.
kubernetes_sd_configs:
  [ - <kubernetes_sd_config> ... ]

# List of Marathon service discovery configurations.
marathon_sd_configs:
  [ - <marathon_sd_config> ... ]

# List of AirBnB's Nerve service discovery configurations.
nerve_sd_configs:
  [ - <nerve_sd_config> ... ]

# List of Zookeeper Serverset service discovery configurations.
serverset_sd_configs:
  [ - <serverset_sd_config> ... ]

# List of Triton service discovery configurations.
triton_sd_configs:
```

```
  [ - <triton_sd_config> ... ]

# List of labeled statically configured Alertmanagers.
static_configs:
  [ - <static_config> ... ]

# List of Alertmanager relabel configurations.
relabel_configs:
  [ - <relabel_config> ... ]
```

<remote_write>

远程写是实验性的：在将来的版本中配置可能会实质性地变化

`url` 是发送样本的端点URL。 `remote_timeout` 指定发送请求到URL的超时时间。目前没有重试机制

`basic_auth` ， `tls_config` 和 `proxy_url` 和在 `scrape_config` 区域里有相同的含义。

`write_relabel_configs` 是relabeling应用到样本数据的。写relabeling是应用到外部标签之后的。这可能有样本发送数量的限制。

这里有一个小Demo，告诉你怎样使用这个功能

存储

STORAGE 存储

Prometheus有一个复杂的本地存储子系统。对于索引，它使用levelDB。对于批量的样本数据，它由自己的自定义存储层，并以固定大小（1024个字节有效负载）的块组织样本数据。然后将这些块存储在每个时间序列的一个文件中的磁盘上。

Memory usage 内存使用量

Prometheus将所有当前使用的块保留在内存中。此外，它将最新使用的块保留在内存中，最大内存可以通过 `storage.local.memory-chunks` 标志配置。如果你有较多的可用内存，你可能希望将其增加到默认值1048576字节以上（反之亦然，如果遇到RAM问题，可以尝试减少内存值）。请注意，服务器的实际RAM使用率将高于将 `storage.local.memory-chunks` *1024字节所期望的RAM使用率。管理存储层中的样本数据是不可避免的开销。此外，服务器正在做更多的事情，而不仅仅存储样本数据。实际开销取决于你的使用模式。在极端情况下，Prometheus必须保持更多的内存块，而不是配置，因为所有这些块都在同一时间使用。你必须试一下。Prometheus导出导出的度量指标 `prometheus_local_storage_memory_chunks` 和 `process_resident_memory_bytes` 将派上用场。作为经验法则，你应该至少拥有内存块所需三倍以上。设计到大量时间序列的PromQL查询大量使用LevelDB支持的索引。如果需要运行这种查询，则可能需要调整索引缓存大小。以下标志是相关的：

- `-storage.local.index-cache-size.label-name-to-label-values`：正则表达式匹配
- `-storage.local.index-cache-size.label-pair-to-fingerprints`：如果大量的时间序列共享相同的标签，增加内存大小
- `-storage.local.index-cache-size.fingerprint-to-metric` and `-storage.local.index-cache-size.fingerprint-to-timerange`：如果你有大量的目标时间序列，例如：一段时间还没有被接收的样本数据时间序列，但是数据又还没有失效。这时也需要增加内存.

你必须尝试使用flag，才能找出有用的。如果一个查询触及到100000多个时间序列数据，几百M内存使用可能是合理的。如果你有足够的内存可用，对于LevelDB使用更多的内存不会有任何伤害。

磁盘使用量 disk usage

Prometheus存储时间序列在磁盘上，目录由flag `storage.local.path` 指定。默认path是 `./data`（关联到工作目录），这是很好的快速尝试，但很可能不是你想要的实际操作。这个flag `stroage.local.retention` 允许你配置这个保留的样本数据。根据你的需求和你的可用磁盘空间做出合适的调整。

Chunking encoding

Prometheus当前提供三种不同类型的块编码（chunk encodings）。对于新创建块的编码由flag `-storage.local.chunk-encoding-version` 决定。 有效值分别是0，1和2.
对于Prometheus的第一块存储存，类型值为0实现了delta编码。类型值为1是当前默认编码, 这是有更好的压缩算法的双delta编码，比类型值为0的delta编码要好。这两种编码在整个块中都具有固定的每个样本字节宽度，这允许快速随机访问。然而类型值为0 的delta编码是最快的编码，与类型值为1的编码相比，编码成本的差异很小。由于具有更好的压缩算法的编码1，除了兼容Prometheus更老的版本，一般建议使用编码1。
类型2是可变宽度的编码，例如：在块中的每个样本能够使用一个不同数量的bit位数。时间戳也是双delta编码。但是算法稍微有点不同。一些不同编码范式对于样本值都是可用的。根据样本值类型来决定使用哪种编码范式，样本值类型有：constant，int型，递增，随机等
编码2的主要部分的灵感来源于Facebook工程师发表的一篇论文：Gorilla: A Fast, Scalable, In-Memory Time Series Database
编码2必须顺序地访问块，并且编解码的代价比较高。总体来看，对比编码1，编码2造成了更高的CPU使用量和增加了查询延时，但是它提供一个改进的压缩比。准确值非常依赖于数据集和查询类型。下面的结果来自典型的生产环境的服务器中：

块编码类型	每个样本数据占用的比特位数	CPU核数	规则评估时间
1	3.3	1.6	2.9s
2	1.3	2.4	4.9s

每次启动Prometheus服务时，你可以改变块的编码类型，因此在实验中测试不同编码类型是我们非常鼓励的。但是考虑到，仅仅是新创建的块会使用新选择块编码，因此你将需要一段时间才能看到效果。

设置大量时间序列数据

Prometheus能够处理百万级别的时间序列数据。然而，你必须调整存储设置到处理多余100000活跃的时间序列。基本上，对于每个时间序列要存储到内存中，你想要允许这几个确定数量的块。对于 `storage.local.memory-chunks` flag标志的默认值是1048567。高达300000个时间序列时，平均来看，每个时间序列仍然有三个可用的块。对于更多的时间序列，应该增加 `storage.local.memory-chunks` 值。三倍于时间序列的数量是一个非常好的近似值。但请注意内存使用的含义（见上文）。
如果你比配置的内存块有更多的时间序列数据，Prometheus不可避免地遇到一种情况，它必须保持比配置更多的内存块。如果使用块数量超过配置限制的10%， Prometheus将会减少获取的样本数据量（通过skip scrape和rule evaluation）直到减少到超过配置的5%。减少获取样本数量是非常糟糕的情况，这是你我都不愿意看到的。
同样重要地，特别是如果写入磁盘，会增长 `storage.local.max-chunks-to-persist` flag值。根据经验，保持它是 `storage.local.memory-chunks` 值的50%是比较好的。
`storage.local.max-chunks-to-persist` 控制了多少块等待写入到你的存储设备，它既可以是

spinning磁盘，也可以是SSD。如果等待块过多，这Prometheus将会减少获取样本数量，知道等待写入的样本数量下降到配置值的95%以下。在发生这种情况之前，Prometheus试着加速写入块。详见文档
每个时间序列可以保留更多的内存块，你就可以批量编写更多的写操作。对spinning磁盘是相当重要的。注意每个活跃的时间序列将会有个不完整的头块，目前还不能被持久化。它是在内存的块，不是磁盘块数据。如果你有1M的活跃时间序列数据，你需要3M `storage.local.memory-chunks` 块，为每个时间序列提供可用的3块内存。仅仅有2M可持久化，因此设置 `storage.local.max-to-persist` 值大于2M，可以很容易地让内存超过3M块。尽管存储 `storage.local.memory-chunks` 的设置，这再次导致可怕的减少样本数量（Prometheus服务将尽快再此出现之前加速消费）。
等待持久性块的高价值的另一个缺点是检查点较大。
如果将大量时间序列与非常快速和/或较大的scrapes相结合，则预先分配的时间序列互斥所可能不会很奏效。如果你在Prometheus服务正在编写检查点或者处理代价大的查询时，看到获取较慢，请尝试增加 `storage.local.num-fingerprint-mutexes` flag值。有时需要数万甚至更多。

持续压力和“冲动模式” Persist pressure and “rushed mode”

本质上，Prometheus服务将尽可能快递将完成的块持久化到磁盘上。这样的策略可能会导致许多小的写入操作，会占用更多的I/O带宽并保持服务器的繁忙。spinning磁盘在这里更加敏感，但是即使是SSD也不会喜欢这样。Prometheus试图尽可能的批量编写写操作，如果允许使用更多的内存，这样做法更好。因此，将上述flag设置为导致充分利用可用内存的值对于高性能非常重要。
Prometheus还将在每次写入后同步时间序列文件（使用 `storage.local.series-sync-strategy = adaptive` ，这是默认值），并将磁盘带宽用于更频繁的检查点（根据“脏的时间序列”的计数，见下文），都试图在崩溃的情况下最小化数据丢失。
但是，如果等待写入的块数量增长太多，怎么办？Prometheus计算一个持久块的紧急度分数，这取决于等待与 `storage.local.max-chunks-to-persist` 值相关的持久性的快数量，以及内存中的快数量超过存储空间的数量。 `local.memory-chunks` 值（如果有的话，只有等待持久性的块的最小数量，以便更快的帮助）。分数在0~1.其中1是指对应于最高的紧急程度。根据得分，Prometheus将更频繁地写入磁盘。如果得分超过0.8的门槛，Prometheus将进入“冲动模式”（你可以在日志中国看到）。在冲动模式下，采用以下策略来加速持久化块：

- 时间序列文件不再在写操作之后同步（更好地利用操作系统的页面缓存，在服务器崩溃的情况下，丢失数据的风险会增加），这个行为通过 `storage.local.series-sync-strategy` flag。
- 检查点仅仅通过 `storage.local.checkpoint-interval` flag启动配置时创建（对于持久化块，以崩溃的情况下更多丢失数据的代码和运行随后崩溃恢复的时间增加，来释放更多的磁盘带宽）
- 对于持久化块的写操作不再被限制，并且尽可能快地执行。

一段得分下降到0.7以下，Prometheus将退出冲动模式。

设置更长的保留时间 setting for very long retention time

如果你有通过 `storage.local.retention` flag(超过一个月), 设置一个更长的留存时间，你可能想要增加 `storage.local.series-file-shrink-ratio` flag值。
每当Prometheus需要从一系列文件的开头切断一些块时，它将简单地重写整个文件。（某些文件系统支持“头截断”，Prometheus目前由于几个原因目前不使用）。为了不重写一个非常大的系列文件来摆脱很少的块，重写只会发生在至少10%的块中 系列文件被删除。 该值可以通过上述的 `storage.local.series-file-shrink-ratio` flag来更改。 如果您有很多磁盘空间，但希望最小化重写（以浪费磁盘空间为代价），请将标志值增加到更高的值，例如。 30%所需的块删除为0.3。

有用的度量指标

在Prometheus暴露自己的度量指标之外，以下内容对于调整上述flag特别有用：

- prometheus_local_storage_memory_series: 时间序列持有的内存当前块数量
- prometheus_local_storage_memory_chunks: 在内存中持久块的当前数量
- `prometheus_local_storage_chunks_to_persist` : 当前仍然需要持久化到磁盘的的内存块数量
- `prometheus_local_storage_persistence_urgency_score` : 上述讨论的紧急程度分数
- 如果Prometheus处于冲动模式下， `prometheus_local_storage_rushed_mode` 值等于1; 否则等于0.

Crash恢复 Carsh Recovery

Prometheus在完成后尽快将块保存到磁盘。常规检查点中不完整的块保存到磁盘。您可以使用 `storage.local.checkpoint-interval` flag配置检查点间隔。如果太多的时间序列处于“脏”状态，那么Prometheus更频繁地创建检查点，即它们当前的不完整的头部块不是包含在最近检查点中的。此限制可通过 `storage.local.checkpoint-dirty-series-limit` flag进行配置。
然而，如果您的服务器崩溃，您可能仍然丢失数据，并且您的存储空间可能处于不一致的状态。因此，Prometheus在意外关机后执行崩溃恢复，类似于文件系统的fsck运行。将记录关于崩溃恢复的详细信息，因此如果需要，您可以将其用于取证。无法恢复的数据被移动到名为孤立的目录（位于storage.local.path下）。如果不再需要，请记住删除该数据。
崩溃恢复通常需要不到一分钟。如果需要更长时间，请咨询日志，以了解出现的问题。

Data corrution 数据损坏

如果您怀疑数据库中的损坏引起的问题，则可以通过使用 `storage.local.dirty` flag启动服务器来强制执行崩溃恢复。
如果没有帮助，或者如果您只想删除现有的数据库，可以通过删除存储目录的内容轻松地启动：

1. stop prometheus.
2. `rm -r <storage path>/*`
3. start prometheus

federation

FEDERATION(联合)

Federation允许一个Prometheus服务从另一个Prometheus服务中获取选中的时间序列数据。

Use cases 用例

对于federation，有几种不同的用例。它常常用于实现Prometheus监控的扩展，或者从另一个Prometheus服务中拉取相关度量指标数据。

Hierarchical federation分层扩展

分层扩展允许Prometheus扩展到数十个数据中心和上百万的节点数量。在这个用例中，federation拓扑结构类似一颗树，较高层级的Prometheus服务从大量较低层级的Prometheus服务中检索和聚集时间序列数据。
例如：环境中可能包含许多以数据中心为基础的Prometheus服务，可以从较高层级收集数据，还有一组全局的Prometheus服务，仅仅从本地服务器收集和存储聚合的数据。这提供了一个聚合的全局视图和本地视图。

跨服务的federation

在跨服务的federation中，一个Prometheus服务配置成从另一个Prometheus服务中获取选中的数据，这个Prometheus服务能够对单个服务中的两个数据集启用警告和查询。
例如，一个运行多个服务的集群调度器可能暴露了集群资源使用信息（例如：CPU和内存使用量）。另一方面，运行在集群上的一个服务仅仅暴露应用程序指定的服务度量指标。经常地，独立的Prometheus服务抓取这两个度量指标CPU和内存。使用federation，这个Prometheus服务包含服务级别的度量指标，这个指标可以从集群Prometheus服务中获取有关其指定的服务集群资源使用量，以便在该服务中使用两组度量指标。

配置federation

任何Prometheus服务，这个 `/federation` 允许从服务中选中的时间序列检索当前值。至少一个 `match[]` URL参数必须指定要选择的暴露时间序列。每一个 `match[]` 参数需要指定一个即时向量选择器，例如： `up` 或者 `{job="api-server"}` 。如果提供了多个 `match` 参数，将会选取所有匹配的时间序列数据的并集。
为了一个Prometheus服务从另一个Prometheus服务中federate度量指标,从一个源服务的 `/federate` 端点，配置你的目标Prometheus服务。当然，也可以使用 `honor_labels` 获取选项和输入想要的 `match[]` 的参数。例如，下面 `scrape_config` federates任何带有 `job="prometheus` 标签的所有时间序列，或者一个以 `job` 开头的度量指标名称：从这个Prometheus服务上端口服务 `prometheus-{1,2,3}:9090` 上获取其他Prometheus服务度量指标数据。

```
- job_name: 'federate'
```

```
  scrape_interval: 15s

  honor_labels: true
  metrics_path: '/federate'

  params:
    'match[]':
      - '{job="prometheus"}'
      - '{__name__=~"job:.*"}'

  static_configs:
    - targets:
      - 'source-prometheus-1:9090'
      - 'source-prometheus-2:9090'
      - 'source-prometheus-3:9090'
```

警告

警告概览

警告概览 alerting overview

Pormetheus的警告由独立的两部分组成。Prometheus服务中的警告规则发送警告到Alertmanager。然后这个Alertmanager管理这些警告。包括silencing, inhibition, aggregation，以及通过一些方法发送通知，例如：email，PagerDuty和HipChat。
建立警告和通知的主要步骤：

- 创建和配置Alertmanager
- 启动Prometheus服务时，通过 `-alertmanager.url` 标志配置Alermanager地址，以便Prometheus服务能和Alertmanager建立连接。
- 在Prometheus服务中创建警告规则

警告器

Alertmanager 警报管理器

Alertmanager处理由客户端应用程序发送的警告，例如：Prometheus服务发送的警告。它关心deduplication, grouping和routing到正确的接收器，如：email，PageDuty或者OpsGenie。它还负责保护和抑制警报。
下面描述有关实现Alertmanager的核心概念。参考配置文件，它会教你怎样使用Alertmanager。

Grouping

Grouping分组将性质类似的警告分组成一个通知类。当许多系统同时出现故障时，这种情况尤其有用，而数百到数千个警报可能同时触发。
例如: 当出现网络分区时，十个到数百个服务实例正在集群中运行。这时多半服务实例暂时无法访问数据库。如果服务实例不能和数据库通信，则对于已经配置好警报规则的Prometheus服务将会对每个服务实例发送一个警报。这样数百个警报会发送到Alertmanager。
如果一个用户仅仅想看到一个页面，这个页面上的数据是精确地表示哪个服务实例受影响了。Alertmanager通过它们的集群和警报名称来分组标签, 这样它可以发送一个单独受影响的通知。
警报分组，分组通知的时间，和通知的接受者是在配置文件中由一个路由树配置的。

Inhibition 抑制

如果某些其他警报已经触发了，则对于某些警报，Inhibition是一个抑制通知的概念。
例如：一个警报已经触发，它正在通知整个集群是不可达的时，Alertmanager则可以配置成关心这个集群的其他警报无效。这可以防止与实际问题无关的数百或数千个触发警报的通知。
通过Alertmanager的配置文件配置Inhibition。

Sliences 静默

静默是一个非常简单的方法，可以在给定时间内简单地忽略所有警报。slience基于matchers配置，类似路由树。来到的警告将会被检查，判断它们是否和活跃的slience相等或者正则表达式匹配。如果匹配成功，则不会将这些警报发送给接收者。
Silences在Alertmanager的web接口中配置。

Client behavior 客户行为

对于客户行为，Alertmanager有特别要求。这些仅仅适用于Prometheus服务不用于发送警报的高级用例。

配置

configuration 配置

Alertmanager通过命令行和一个配置文件配置。命令行配置不可变的系统参数，而配置文件定义inhibiton规则，通知路由和通知接收者。

可视化编辑器可以帮助构建路由树。

如果想要查看所有命令，请使用命令 `alertmanager -h` 。

`Alertmanager` 能够在运行时动态加载配置文件。如果新的配置有错误，则配置中的变化不会生效，同时错误日志被输出到终端。通过发送 `SIGHUP` 信号量给这个进程，或者通过HTTP POST请求 `/-/reload` ，Alertmanager配置动态加载到内存。

配置文件

使用 `-config.file` 指定要加载的配置文件

> ./alertmanager -config.file=simple.yml

这个配置文件使用 `yaml` 格式编写的，括号表示参数是可选的，对于非列表参数，该值将设置为指定的默认值。

- `<duration>` : 与正则表达式匹配的持续时间 `[0-9]+(ms|[smhdwy])`
- `<labeltime>` : 与正则表达式匹配的字符串 `[a-zA-Z_][a-zA-Z0-9_]*`
- `<filepath>` : 当前工作目录下的有效路径
- `<boolean>` : 布尔值： `false` 或者 `true` 。
- `<string>` : 常规字符串
- `<tmpl_string>` : 一个在使用前被模板扩展的字符串

其他占位符被分开指定， 一个有效的示例，点击这里

全局配置指定的参数在所有其他上下文配置中是有效的。它们也作为其他区域的默认值。

```
global:
  # ResolveTimeout is the time after which an alert is declared resolved
  # if it has not been updated.
  [ resolve_timeout: <duration> | default = 5m ]

  # The default SMTP From header field.
  [ smtp_from: <tmpl_string> ]
  # The default SMTP smarthost used for sending emails.
  [ smtp_smarthost: <string> ]
```

```
  # SMTP authentication information.
  [ smtp_auth_username: <string> ]
  [ smtp_auth_password: <string> ]
  [ smtp_auth_secret: <string> ]
  # The default SMTP TLS requirement.
  [ smtp_require_tls: <bool> | default = true ]

  # The API URL to use for Slack notifications.
  [ slack_api_url: <string> ]

  [ pagerduty_url: <string> | default = "https://events.pagerduty.com/generic/2
010-04-15/create_event.json" ]
  [ opsgenie_api_host: <string> | default = "https://api.opsgenie.com/" ]
  [ hipchat_url: <string> | default = "https://api.hipchat.com/" ]
  [ hipchat_auth_token: <string> ]

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

# The root node of the routing tree.
route: <route>

# A list of notification receivers.
receivers:
  - <receiver> ...

# A list of inhibition rules.
inhibit_rules:
  [ - <inhibit_rule> ... ]
```

一个路由块在路由树和它的孩子中定义了一个节点。如果不设置，它的可选配置参数从父节点中继承其值。

每个警报在已配置路由树的顶部节点，这个节点必须匹配所有警报。然后遍历所有的子节点。如果 `continue` 设置成 `false`，当匹配到第一个孩子时，它会停止下来；如果 `continue` 设置成 `true`，则警报将继续匹配后续的兄弟姐妹节点。如果一个警报不匹配一个节点的任何孩子，这个警报将会基于当前节点的配置参数来处理警报。

```
[ receiver: <string> ]
[ group_by: '[' <labelname>, ... ']' ]

# Whether an alert should continue matching subsequent sibling nodes.
[ continue: <boolean> | default = false ]

# A set of equality matchers an alert has to fulfill to match the node.
match:
  [ <labelname>: <labelvalue>, ... ]
```

```
# A set of regex-matchers an alert has to fulfill to match the node.
match_re:
  [ <labelname>: <regex>, ... ]

# How long to initially wait to send a notification for a group
# of alerts. Allows to wait for an inhibiting alert to arrive or collect
# more initial alerts for the same group. (Usually ~0s to few minutes.)
[ group_wait: <duration> ]

# How long to wait before sending notification about new alerts that are
# in are added to a group of alerts for which an initial notification
# has already been sent. (Usually ~5min or more.)
[ group_interval: <duration> ]

# How long to wait before sending a notification again if it has already
# been sent successfully for an alert. (Usually ~3h or more).
[ repeat_interval: <duration> ]

# Zero or more child routes.
routes:
  [ - <route> ... ]
```

示例

```
# The root route with all parameters, which are inherited by the child
# routes if they are not overwritten.
route:
  receiver: 'default-receiver'
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  group_by: [cluster, alertname]
  # All alerts that do not match the following child routes
  # will remain at the root node and be dispatched to 'default-receiver'.
  routes:
  # All alerts with service=mysql or service=cassandra
  # are dispatched to the database pager.
  - receiver: 'database-pager'
    group_wait: 10s
    match_re:
      service: mysql|cassandra
  # All alerts with the team=frontend label match this sub-route.
  # They are grouped by product and environment rather than cluster
  # and alertname.
  - receiver: 'frontend-pager'
    group_by: [product, environment]
    match:
      team: frontend
```

<inhibit_rule>

一个inhibition规则是在与另一组匹配器匹配的警报存在的条件下，使匹配一组匹配器的警报失效的规则。两个警报必须具有一组相同的标签。

```
# Matchers that have to be fulfilled in the alerts to be muted.
target_match:
  [ <labelname>: <labelvalue>, ... ]
target_match_re:
  [ <labelname>: <regex>, ... ]

# Matchers for which one or more alerts have to exist for the
# inhibition to take effect.
source_match:
  [ <labelname>: <labelvalue>, ... ]
source_match_re:
  [ <labelname>: <regex>, ... ]

# Labels that must have an equal value in the source and target
# alert for the inhibition to take effect.
[ equal: '[' <labelname>, ... ']' ]
```

接收者是一个或者多个通知集成的命名配置

Alertmanager在v0.0.4中可用的其他接收器尚未实现。我们乐意接受任何贡献，并将其添加到新的实现中

```
# The unique name of the receiver.
name: <string>

# Configurations for several notification integrations.
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>, ... ]
```

<email_config>

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

# The email address to send notifications to.
to: <tmpl_string>
# The sender address.
[ from: <tmpl_string> | default = global.smtp_from ]
# The SMTP host through which emails are sent.
[ smarthost: <string> | default = global.smtp_smarthost ]
# SMTP authentication information.
[ auth_username: <string> ]
[ auth_password: <string> ]
[ auth_secret: <string> ]
[ auth_identity: <string> ]

[ require_tls: <bool> | default = global.smtp_require_tls ]

# The HTML body of the email notification.
[ html: <tmpl_string> | default = '{{ template "email.default.html" . }}' ]

# Further headers email header key/value pairs. Overrides any headers
# previously set by the notification implementation.
[ headers: { <string>: <tmpl_string>, ... } ]
```

<hipchat_config>

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

# The HipChat Room ID.
room_id: <tmpl_string>
# The auth token.
[ auth_token: <string> | default = global.hipchat_auth_token ]
# The URL to send API requests to.
[ url: <string> | default = global.hipchat_url ]

# See https://www.hipchat.com/docs/apiv2/method/send_room_notification
# A label to be shown in addition to the sender's name.
[ from:  <tmpl_string> | default = '{{ template "hipchat.default.from" . }}' ]
# The message body.
[ message:  <tmpl_string> | default = '{{ template "hipchat.default.message" .
}}' ]
# Whether this message should trigger a user notification.
[ notify:  <boolean> | default = false ]
# Determines how the message is treated by the alertmanager and rendered inside
 HipChat. Valid values are 'text' and 'html'.
[ message_format:  <string> | default = 'text' ]
# Background color for message.
[ color:  <tmpl_string> | default = '{{ if eq .Status "firing" }}red{{ else }}g
```

```
reen{{ end }}' ]
```

<pagerduty_config>

通过PagerDuty ApI发送通知：

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

# The PagerDuty service key.
service_key: <tmpl_string>
# The URL to send API requests to
[ url: <string> | default = global.pagerduty_url ]

# The client identification of the Alertmanager.
[ client:  <tmpl_string> | default = '{{ template "pagerduty.default.client" .
}}' ]
# A backlink to the sender of the notification.
[ client_url:  <tmpl_string> | default = '{{ template "pagerduty.default.client
URL" . }}' ]

# A description of the incident.
[ description: <tmpl_string> | default = '{{ template "pagerduty.default.descri
ption" .}}' ]

# A set of arbitrary key/value pairs that provide further detail
# about the incident.
[ details: { <string>: <tmpl_string>, ... } | default = {
  firing:       '{{ template "pagerduty.default.instances" .Alerts.Firing }}'
  resolved:     '{{ template "pagerduty.default.instances" .Alerts.Resolved }}'
  num_firing:   '{{ .Alerts.Firing | len }}'
  num_resolved: '{{ .Alerts.Resolved | len }}'
} ]
```

<pushover_config>

通过PUSHover API发送通知：

```
# The recipient user’s user key.
user_key: <string>

# Your registered application’s API token, see https://pushover.net/apps
token: <string>

# Notification title.
[ title: <tmpl_string> | default = '{{ template "pushover.default.title" . }}'
]
```

```
# Notification message.
[ message: <tmpl_string> | default = '{{ template "pushover.default.message" .
}}' ]

# A supplementary URL shown alongside the message.
[ url: <tmpl_string> | default = '{{ template "pushover.default.url" . }}' ]

# Priority, see https://pushover.net/api#priority
[ priority: <tmpl_string> | default = '{{ if eq .Status "firing" }}2{{ else }}0
{{ end }}' ]

# How often the Pushover servers will send the same notification to the user.
# Must be at least 30 seconds.
[ retry: <duration> | default = 1m ]

# How long your notification will continue to be retried for, unless the user
# acknowledges the notification.
[ expire: <duration> | default = 1h ]
```

<slack_config>

通过Slack webhooks发送通知：

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

# The Slack webhook URL.
[ api_url: <string> | default = global.slack_api_url ]

# The channel or user to send notifications to.
channel: <tmpl_string>

# API request data as defined by the Slack webhook API.
[ color: <tmpl_string> | default = '{{ if eq .Status "firing" }}danger{{ else }
}}good{{ end }}' ]
[ username: <tmpl_string> | default = '{{ template "slack.default.username" . }
}'
[ title: <tmpl_string> | default = '{{ template "slack.default.title" . }}' ]
[ title_link: <tmpl_string> | default = '{{ template "slack.default.titlelink"
. }}' ]
[ icon_emoji: <tmpl_string> ]
[ icon_url: <tmpl_string> ]
[ pretext: <tmpl_string> | default = '{{ template "slack.default.pretext" . }}'
 ]
[ text: <tmpl_string> | default = '{{ template "slack.default.text" . }}' ]
[ fallback: <tmpl_string> | default = '{{ template "slack.default.fallback" . }
}' ]
```

<opsgenie_config>

通过OpsGenie API发送通知:

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

# The API key to use when talking to the OpsGenie API.
api_key: <string>

# The host to send OpsGenie API requests to.
[ api_host: <string> | default = global.opsgenie_api_host ]

# A description of the incident.
[ description: <tmpl_string> | default = '{{ template "opsgenie.default.descrip
tion" . }}' ]
# A backlink to the sender of the notification.
[ source: <tmpl_string> | default = '{{ template "opsgenie.default.source" . }}
' ]

# A set of arbitrary key/value pairs that provide further detail
# about the incident.
[ details: { <string>: <tmpl_string>, ... } ]

# Comma separated list of team responsible for notifications.
[ teams: <tmpl_string> ]
# Comma separated list of tags attached to the notifications.
[ tags: <tmpl_string> ]
```

<webhook_config>

webhook接收者允许配置一个通用的接收者

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

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

Alertmanager通过HTTP POST请求发送json格式的数据到配置端点：

```
{
  "version": "3",
  "groupKey": <number>     // key identifying the group of alerts (e.g. to dedu
plicate)
  "status": "<resolved|firing>",
  "receiver": <string>,
```

```
  "groupLabels": <object>,
  "commonLabels": <object>,
  "commonAnnotations": <object>,
  "externalURL": <string>,  // backling to the Alertmanager.
  "alerts": [
    {
      "labels": <object>,
      "annotations": <object>,
      "startsAt": "<rfc3339>",
      "endsAt": "<rfc3339>"
    },
    ...
  ]
}
```

警告规则

Alerting rules 警报规则

警报规则允许你基于Prometheus表达式语言的表达式定义报警报条件，并在触发警报时发送通知给外部的接收者。每当警报表达式在给定时间点产生一个或者多个向量元素，这个警报统计活跃的这些元素标签集。
警报规则在Prometheus系统中用同样的record rules方式进行配置

定义警报规则

警报规则的定义遵循下面的风格：

```
ALERT <alert name>
  IF <expression>
    [ FOR <duration> ]
      [ LABELS <label set> ]
        [ ANNOTATIONS <label set> ]
```

`FOR` 选项语句会使Prometheus服务等待指定的时间, 在第一次遇到新的表达式输出向量元素（如：具有高HTTP错误率的实例）之间，并将该警报统计为该元素的触发。如果该元素的活跃的，且尚未触发，表示正在挂起状态。
`LABELS` 选项语句允许指定额外的标签列表，把它们附加在警告上。任何已存在的冲突标签会被重写。这个标签值能够被模板化。
`ANNOTATIONS` 选项语句指定了另一组标签，它们不被当做警告实例的身份标识。它们经常用于存储额外的信息，例如：警告描述，后者runbook链接。这个注释值能够被模板化。

Templating 模板

```
# Alert for any instance that is unreachable for >5 minutes.
ALERT InstanceDown
  IF up == 0
    FOR 5m
      LABELS { severity = "page" }
        ANNOTATIONS {
                summary = "Instance {{ $labels.instance }} down",
                    description = "{{ $labels.instance }} of job {{ $labels.job
 }} has been down for more than 5 minutes.",
                     }

# Alert for any instance that have a median request latency >1s.
ALERT APIHighRequestLatency
  IF api_http_request_latencies_second{quantile="0.5"} > 1
```

```
    FOR 1m
      ANNOTATIONS {
              summary = "High request latency on {{ $labels.instance }}",
                  description = "{{ $labels.instance }} has a median request la
tency above 1s (current value: {{ $value }}s)",
                    }
```

运行时检查警告

为了能够手动检查哪个警告是活跃的（挂起或者触发），导航到你的Prometheus服务实例的"Alerts"tab页面。这个会显示精确的标签集合，它们每一个定义的警告都是当前活跃的。

对于挂起和触发警告，Prometheus也存储形如 `ALERTS{alertname="<alert name>", alertstat=s"pending|firing", <additional alert la` . 只要警告是在指定的活跃（挂起或者触发）状态上，这个样本值设置为1。当一个警告从活跃状态变成不活跃状态时，这个样本值被设置为0。一旦不活跃，这个时间序列将不会再更新。

发送警告通知

Prometheus的警告规则擅长确定当前哪个实例有问题。但它们并不是一个完整的通知解决方案。在简单的警报定义上，需要另一个层来添加总结，通知速率限制，silencing，警报依赖。在Prometheus的生态系统中，Alertmanager发挥了这一作用。因此，Prometheus可能被配置为定期向Alertmanager实例发送有关警报信息，该实例然后负责调用正确的通知，可以通过 `-alertmanager.url` 命令行标志配置Alertmanager实例。

客户端

发送警报

免责声明：Prometheus自动处理发送由其配置的警报规则生成的警报。强烈建议你根据时间序列数据配置Prometheus中的警报规则，而不是直接使用客户端。

Alertmanager用http的API `/api/v1/alerts` 监听警报。只要Alertmanager仍然活跃（经常使用30s到3min时间），客户端期望持续地重发警报。客户端通过下面的POST请求，能够推送警报列表到指定端点：

```
[
  {
    "labels": {
      "<labelname>": "<labelvalue>",
      ...
    },
    "annotations": {
      "<labelname>": "<labelvalue>",
    },
    "startsAt": "<rfc3339>",
    "endsAt": "<rfc3339>"
    "generatorURL": "<generator_url>"
  },
  ...
]
```

这个标签用于识别一个警告的唯一实例和执行去重数据操作。这个注释总是设置给最近经常被接收的警告实例。timestamps是可选的。如果 `startsAt` 省略，这个当前时间被赋值给Alertmanager。如果一个警报的结束时间是已知的，则只有 `endsAt` 被设置。如果这个警报是最后被接收的，它将会设置一个可配置的超时时间。`generatorURL` 字段是唯一的后端链接，用于标识客户端中此警报的引发实体。

Alertmanager还支持 `/api/alerts` 上的传统端点。与Prometheus的v0.16.2级更低版本兼容。