

监控的目的

1. 长期趋势分析：通过对监控样本数据的持续收集和统计，对监控指标进行长期趋势分析。例如，通过对磁盘空间增长率的判断，我们可以提前预测在未来什么时间节点上需要对资源进行扩容。
2. 对照分析：两个版本的系统运行资源使用情况的差异如何？在不同容量情况下系统的并发和负载变化如何？通过监控能够方便的对系统进行跟踪和比较。
3. 故障分析与定位：当问题发生后，需要对问题进行调查和处理。通过对不同监控监控以及历史数据的分析，能够找到并解决根源问题。
4. 数据可视化：通过可视化仪表盘能够直接获取系统的运行状态、资源使用情况、以及服务运行状态等直观的信息。

如何获得获得kubernetes的监控数据？

1. 基础设施层(Node)：为整个集群和应用提供运行时资源，需要通过各节点的kubelet获取节点的基本状态，同时通过在节点上部署Node Exporter获取节点的资源使用情况。

- 为了能够采集集群中各个节点的资源使用情况，我们需要在各节点中部署一个Node Exporter实例。
- 对于Node Exporter而言每个节点只需要运行一个唯一的实例，此时需要使用Kubernetes的Daemonset。Daemonset的管理方式类似于操作系统中的守护进程。Daemonset会确保在集群中所有节点上运行一个唯一的Pod实例。
- 通过Daemonset的形式将Node Exporter部署到了集群中的各个节点中。需要通过Prometheus的服务发现模式，找到当前集群中部署的Node Exporter实例即可。这里我们为Node Exporter添加了注解：

```
1 prometheus.io/scrape: 'true'
```

由于Kubernetes中Pod可能会包含多个容器，还需要用户通过注解指定用户提供监控指标的采集端口：

```
1 prometheus.io/port: '9100'
```

而有些情况下，Pod中的容器可能并没有使用默认的/metrics作为监控采集路径，因此还需要支持用户指定采集路径：

```
1 prometheus.io/path: 'metrics'
```

安装完成之后, 可以采集到的数据如下

- node_boot_time：系统启动时间
- node_cpu：系统CPU使用量

- nodedisk*：磁盘IO
- nodefilesystem*：文件系统用量
- node_load1：系统负载
- nodememeory*：内存使用量
- nodenetwork*：网络带宽
- node_time：当前系统时间
- go_*：node exporter中go相关指标
- process_*：node exporter自身进程相关运行指标

遗留问题:

Node exporter 安装完成之后, 所有的slave节点的node exporter都可以被自动的安装上, 但是master并没有, 当前并不清楚为什么master节点没有node exporter.

2. 容器基础设施(Container): 为应用提供运行时环境, Kubelet内置了对cAdvisor的支持, 用户可以直接通过Kubelet组件获取给节点上容器相关监控指标。

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

介绍Prometheus

- Prometheus是一个开源的完整监控解决方案，其对传统监控系统的测试和告警模型进行了彻底的颠覆，形成了基于中央化的规则计算、统一分析和告警的新模型。
- Prometheus基于Pull模型的架构方式，可以在任何地方（本地电脑，开发环境，测试环境）搭建我们的监控系统。对于一些复杂的情况，还可以使用Prometheus服务发现(Service Discovery)的能力动态管理监控目标。
- 基于Prometheus丰富的Client库，用户可以轻松的在应用程序中添加对Prometheus的支持，从而让用户可以获取服务和应用内部真正的运行状态。

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

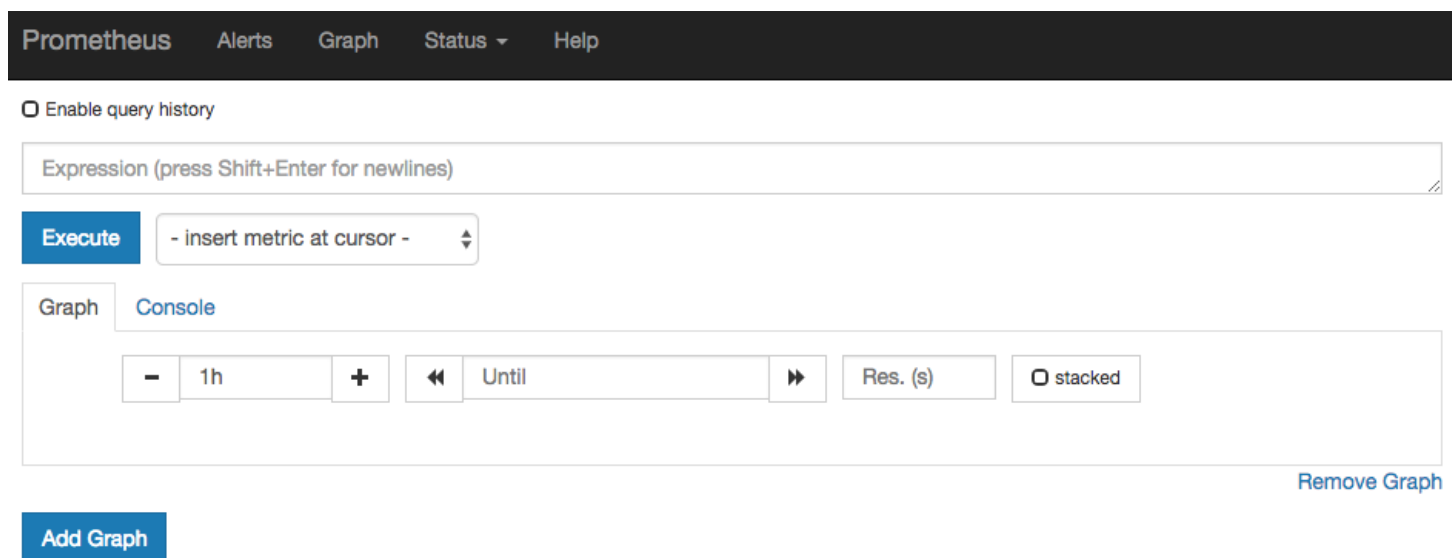
```
http_request_status{code='200',content_path='/api/path', environment='produment'} =>
[value1@timestamp1,value2@timestamp2...]
```

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

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

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

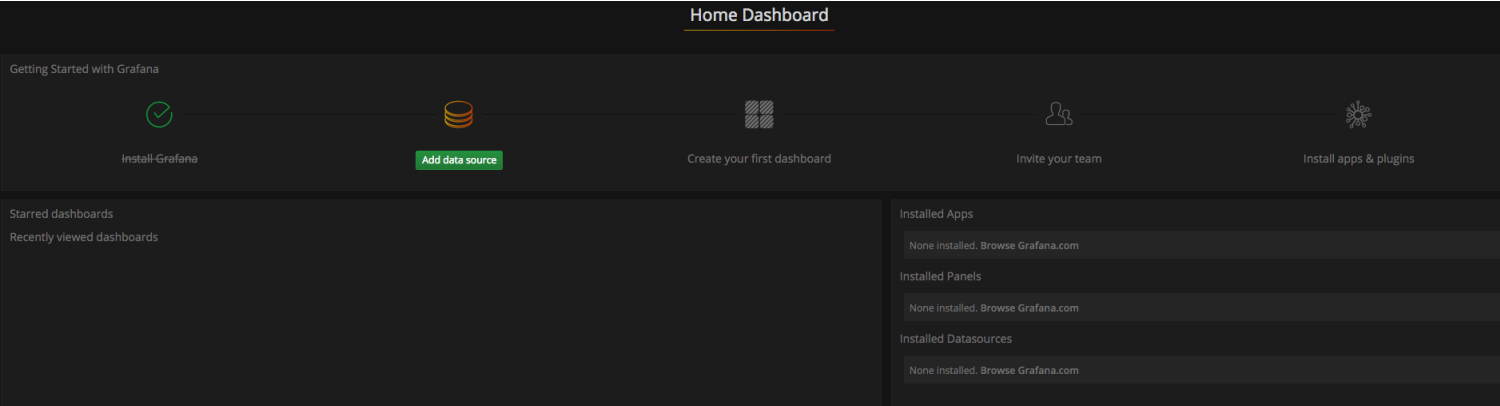
- Prometheus内置了一个强大的数据查询语言PromQL。通过PromQL可以实现对监控数据的查询、聚合。同时PromQL也被应用于数据可视化(如Grafana)以及告警当中。
- Prometheus Server中自带了一个Prometheus UI，通过这个UI可以方便地直接对数据进行查询，并且支持直接以图形化的形式展示数据。最新的Grafana可视化工具也已经提供了完整的Prometheus支持，基于Grafana可以创建更加精美的监控图标。基于Prometheus提供的API还可以实现自己的监控可视化UI。



介绍Grafana

Grafana是一个开源的可视化平台，并且提供了对Prometheus的完整支持。

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



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

Add data source

Config

Dashboards

Name	Prometheus	Default	<input checked="" type="checkbox"/>
Type	Prometheus		

HTTP settings

URL	http://localhost:9090
Access	direct

HTTP Auth

Basic Auth	<input type="checkbox"/>	With Credentials	<input type="checkbox"/>
------------	--------------------------	------------------	--------------------------

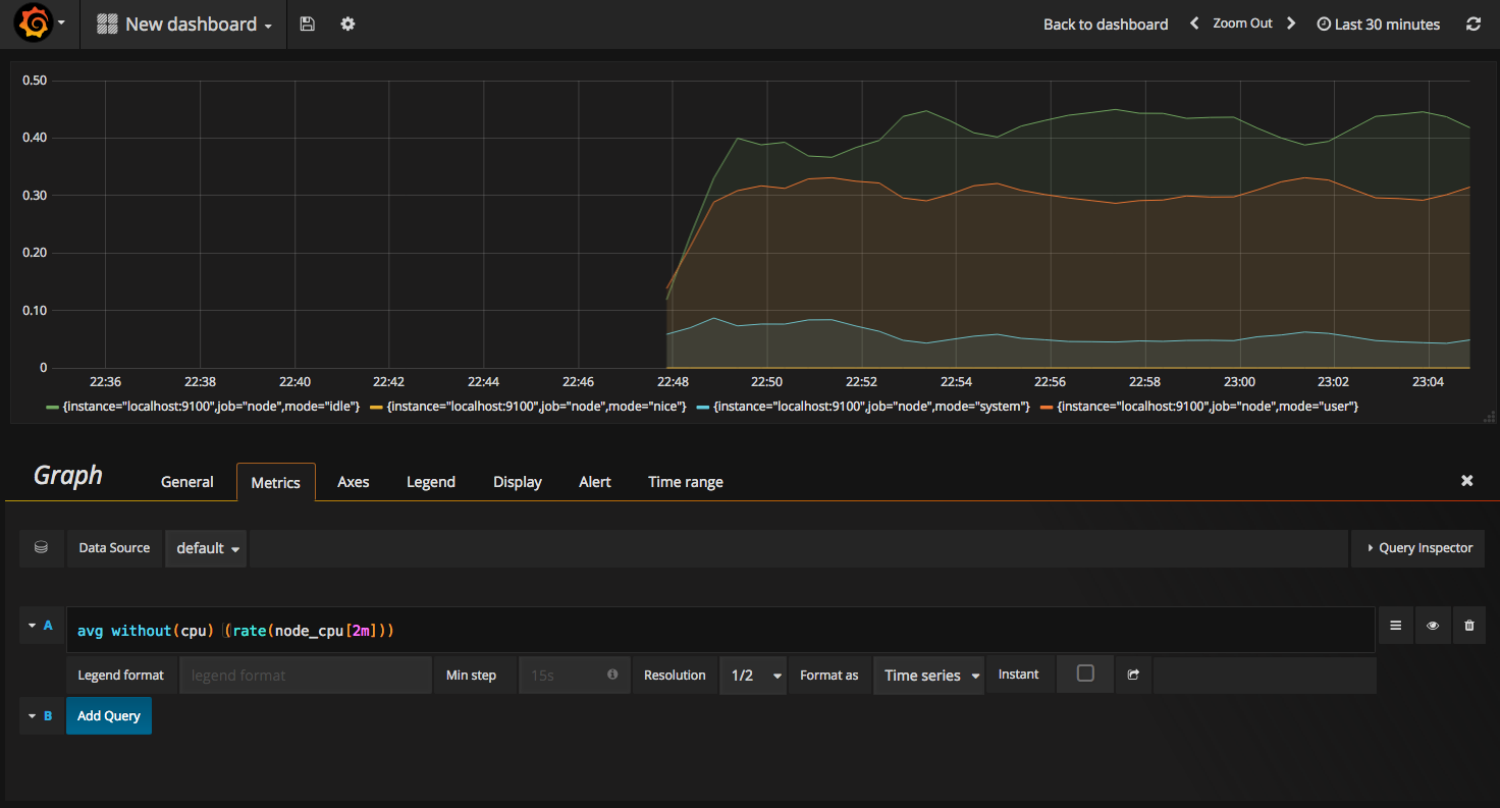
Scrape interval

15s

Add

Cancel

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



点击界面中的保存选项，就创建了我们的第一个可视化Dashboard了。当然作为开源软件，Grafana社区鼓励用户分享Dashboard通过<https://grafana.com/dashboards>网站，可以找到大量可直接使用的Dashboard: 列如 3158739 1860

Dashboards

Official & community built dashboards

Filter by:


Data Source
Prometheus


Panel Type
All


Category
All


Collector
nodeExporter


Search within this list


**Apache** by idealista
PROMETHEUS NODEEXPORTER: OTHER
Downloads: 622

**Apache** by arul karthi
PROMETHEUS NODEEXPORTER: CPU
Downloads: 32

**CPU Utilization Details (Cores)** by perconalab
This is experimental dashboard for Percona Monitoring and Management (PMM) which provides CPU...
PROMETHEUS NODEEXPORTER
Downloads: 314

**Docker and system monitoring** by Thibaut Mottet
A simple overview of the most important Docker host and container metrics. (cAdvisor/Prometheus)
PROMETHEUS NODEEXPORTER
Downloads: 5970

**Docker and system monitoring** by paulfantom
A simple overview of the most important Docker host and container metrics. (cAdvisor/Prometheus)
PROMETHEUS NODEEXPORTER
Downloads: 320

**Docker Dashboard** by Brian Christner
Docker Monitoring Template
PROMETHEUS NODEEXPORTER
Downloads: 8054

Share your dashboards

Sign up for a free [Grafana.com](https://grafana.com) account and share your creations with the community.

Sign Up

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



grafana 示例

<http://localhost:3000>

配置监控系统示例

创建一个 Prometheus/install/monitoring_deployment.yml

如何监控应用数据？

在vanguard中集成prometheus的步骤:

1. init Collectors

例如初始化Metrics类型为Counter的请求总数和类型为Gauge的QPS

```
RequestCount = prometheus.NewCounterVec(prometheus.CounterOpts{
    Namespace: "zdns",
    Subsystem: "vanguard",
    Name: "request_count_total",
    Help: "Counter of DNS requests made per view.",
}, []string{"module", "view"})
```

```
QPS = prometheus.NewGaugeVec(prometheus.GaugeOpts{
    Namespace: "zdns",
    Subsystem: "vanguard",
    Name: "qps",
```

Help: "requests per second, view.",

```
}, []string{"module", "view"})
```

在prometheus服务的web页面中我们可以看到这些collectors名字是以 zdns_vanguard_开头的

module和view的值会在record collectors时被填充

并以request_count_total或者qps结尾

2. register Collectors to prometheus

调用prometheus.MustRegister函数注册RequestCount和QPS

3. 为了计算qps， 初始化一个map， 一个视图对应一个计数器对象

4. record collectors

当有个vanguard处理完一次dns请求， metrics模块就会对其进行记录

例如当视图v1处理完一次请求后

1) 调用函数RequestCount.WithLabelValues("server", "v1").Inc()

zdns_vanguard_server_v1_request_count_total的值就会 + 1

2) 将v1对应的计数器 + 1

5. QPS

例如视图v1的计数器的值为325

每秒调用一次函数QPS.WithLabelValues("server", "v1").Set(float64(325))

zdns_vanguard_server_v1_qps的值为325

并重置v1的计数器

配置vanguard时加入annotations

```
1 spec:
2   replicas: 2
3   selector:
4     matchLabels:
5       app: user-dns
6   template:
7     metadata:
8       labels:
9         app: user-dns
10    annotations:
11      prometheus.io/scrape: 'true'
12      prometheus.io/port: '9200'
13      prometheus.io/path: '/metrics'
```