**Prometheus部署文档**

**一、Prometheus简介**

**(1)官网：**

<https://prometheus.io/>

**(2)背景：**

Prometheus是由前Google工程师从2012年开始在Soundcloud以开源软件的形式进行研发的系统监控和告警的软件。自此以后，许多公司和组织都采用了Prometheus作为监控告警工具。

Prometheus的开发者和用户社区非常活跃，它现在是一个独立的开源项目，可以独立于任何公司进行维护。为了证明这一点，Prometheus于2016年5月加入CNCF基金会，成为继Kubernetes之后的第二个CNCF托管项目。

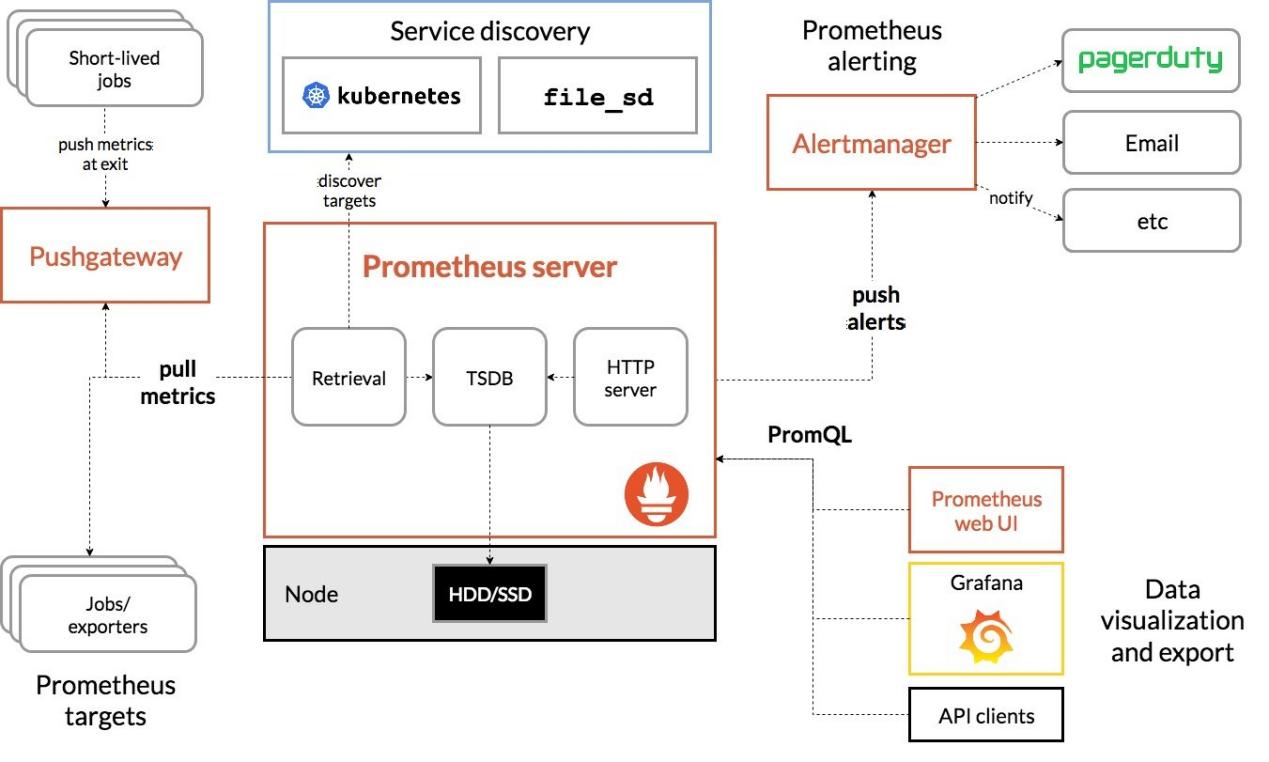
**(3)优势：**

* 由指标名称和键/值对标签标识的时间序列数据组成的多维数据模型
* 强大的查询语言引擎PromQL
* 不依赖分布式存储，单个服务节点具有自治能力
* 时间序列数据是服务通过HTTP协议主动拉去获得的
* 也可以通过中间网关来推送时间序列数据
* 可以通过静态配置文件或服务发现来获取监控指标
* 支持多种类型的图标和仪表盘

**(4)Prometheus的生态组件**

* Prometheus Server：收集和存储时间序列数据
* Client Library：客户端库，目的在于为那些期望原生提供Instrumentation功能的应用程序提供便捷的开发途径
* Push Gateway：接收那些通常由短期作业生成的指标数据的网关，并支持由Prometheus Server进行指标拉取操作
* Exporters：用于暴露现有应用程序或服务（不支持Instrumentation）的指标给Prometheus Server
* Alertmanager：从Prometheus Server接收到“告警通知”后，通过去重、分组、路由等预处理功能后以高效想用户完成告警信息发送
* Data Visualization：Prometheus Web UI（Prometheus Server内建），及Granfana等
* Service Discovery：动态发现待监控的Target，从而完成监控配置的重要组件，在容器化环境中尤为重要，该组件目前由Prometheus Server内建支持

**(5)Prometueu的架构**



Prometheus Server直接从监控目标中或者间接通过推送网关来拉去监控数据，它在本地存储所有抓取到的样本数据，并对此数据执行一系列规则，以汇总和记录现有数据的新时间序列或生成告警。可以通过grafana或者其他工具来实现监控数据的可视化。

**(6)数据模型**

Prometheus所有采集的监控数据均以指标(metric)的形式保存在内置中的时间序列数据库(TSDB)当中：属于同一指标名称，同一标签集合的、有时间戳标记的数据流。除了存储的时间序列，Prometheus还可以根据查询请求产生临时的、衍生的时间序列作为返回结果。

**指标名称和标签：**

每一条时间序列由指标名称(Metrics Name)以及一组标签(键值对)唯一标识。其中指标的名称(metrics name)可以反映被监控样本的含义(例如，http\_requests\_total表示当前系统接收到的HTTP请求总量)。

通过使用标签，Prometheus开启了很强大的多维数据模型：对于相同的指标名称，通过不同的标签列表的集合，会形成特定的度量维度示例（例如：所有包含度量名称为/api/tracks的http请求，打上methos=POST的标签，就会形成具体的http请求）。该查询语言在这些指标和标签列表的基础上进行过滤和集合。改变任何度量指标上的任何标签值（包括添加和删除指标），都会创建新的时间序列。

**样本：在时间序列中的每一个点称为一个样本（sample），样本由一下三部分组成**

* 指标(metric)：指标名称和描述当前样本特征的labelsets(标签集)，通常意味着CPU速率、内存使用率或分区空闲比例等
* 时间戳(timestamp)：一个精确到毫秒的时间戳
* 样本值(value)：一个folat64的浮点型数据表示当前样本的值

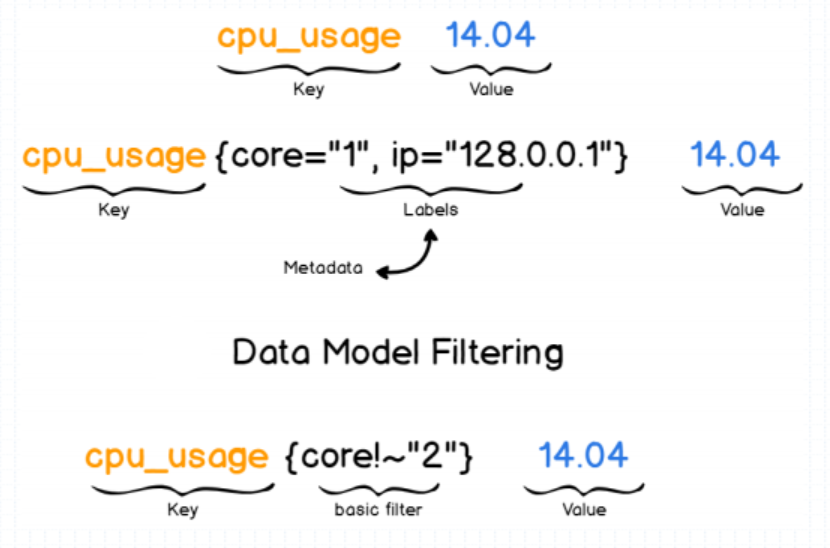
**表示方式：**

通过如下表达方式表示指定指标名称和指定标签集合时间序列

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

例如：指标名为api\_http\_requests\_total，标签为method=”POST”和handler=”/messages”的时间序列可以表示为：

api\_http\_requests\_total{method="POST", handler="/messages"}



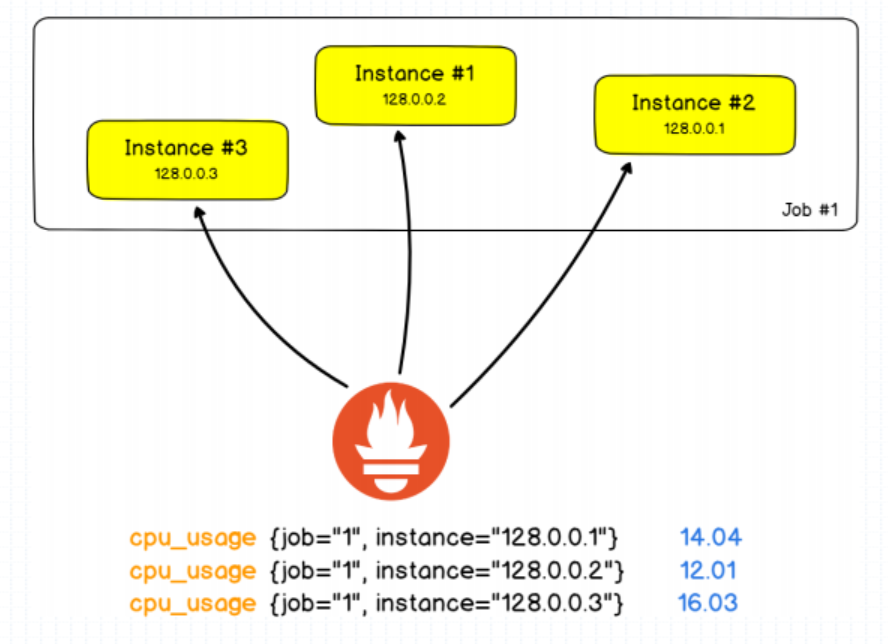
**(7)指标类型**

参考：https://prometheus.fuckcloudnative.io/di-er-zhang-gai-nian/metric\_types

* Counter：计数器，用于保存单调递增型的数据，例如站点访问次数等；不能为负值，也不支持减少，但可以重置回0。例如：可以使用counter类型的指标来表示服务的请求数、已完成的任务数、错误发生的次数等。
* Gauge：仪表盘，代表一种数据可以任意变化的指标，即可增可减。通常用于想温度或者内存使用率这种指标数据，也可以表示能随时增加或减少的“总数“，例如：当前并发请求的数量
* Histogram：直方图，在一段时间范围内对数据进行采样（通常是请求持续时间或响应时间等），并将其计入可配置的存储桶（bucket）中，后续可通过指定区间筛选样本，也可以统计样本总数，最后一般将数据展示为直方图
* Summary：摘要，与Histogram类型相似，用于表示一段时间内的数据采样结果（通常是请求持续时间或响应大小等），但它直接存储了分位数（通过客户端计算，然后展示出来），而不是通过区间来计算

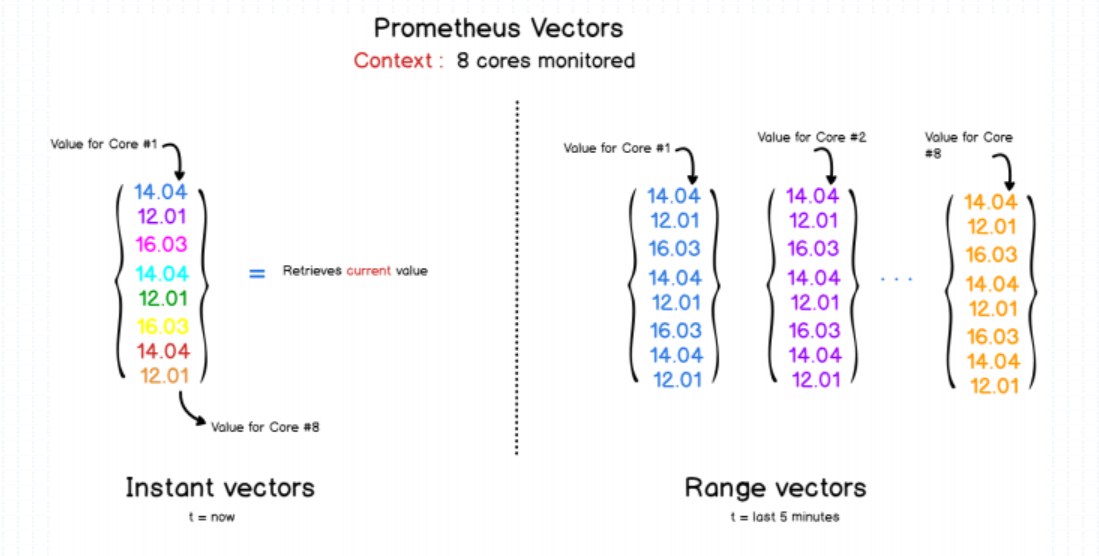
**(8)作业(Job)和实例(Instance)**

* 实例(Instance)：能够接收Prometheus Server数据Scrape操作(指标采集)的每个网络端点(endpoint)即为一个Instance(实例)
* 作业(job)：具有类似功能的Instance的集合称为一个Job，例如一个MySQL主从复制集群中的所有MySQL进程



**(9)PromQL**

* Prometheus提供了内置的数据查询语言PromQL（Prometheus Query Language），支持用户进行实时的数据查询及聚合操作
* PromQL支持处理两种向量，并内置提供了一组用于数据处理的函数
  + 即时向量：最近一次时间戳上跟踪的数据指标
  + 时间范围向量：指定时间范围内的所有时间戳上的数据指标



**(10)Instrumentation(程序仪表)**

* 任何能够支持Scrape指标数据的应用程序都首先要具有一个测量系统
* 在Prometheus的语境中，Instrumentation是指附加到应用程序中的那些用于暴露程序指标数据的客户端库
  + 程序员借助于这些客户端库编写代码生成可暴露的指标数据

**(11)Exporters**

* 对于那些未内建Instrumentation，且也不方便于自行添加该类组件以暴露指标数据的应用程序来说，常用的办法是对于待监控的目标应用程序外部运行一个独立指标暴露程序，这类型的程序即统称为Exporter
* Exporter负责从目标应用程序上采集和聚合原始格式的数据，并转换或聚合为Prometheus格式的指标向外暴露
* Prometheus站点提供了大量的Exporter，例如：node\_exporter、mysqld\_exporter等

**(12)Alerts**

* 抓取到异常值后，Prometheus支持通过告警(Alert)机制向用户发送反馈或警示，以触发用户能够及时采取应对措施
* Prometheus Server仅负责生成告警指示，具体的告警行为由另一个独立的应用程序AlertManager负责
  + 告警指示由Prometheus Server 基于用户提供的“告警规则“周期性计算生成
  + Alertmanager接收到Prometheus Server发来的告警指示后，基于用户定义的告警路由(route)向告警接收人(receivers)发送告警信息

**二、部署Prometheus**

|  |
| --- |
| **(1)官网下载安装包**  <https://prometheus.io/download/> 选择Liunx amd64架构    **(2)部署prometheus server**  #tar xf prometheus-2.27.1.linux-amd64.tar.gz -C /data/  #cd /data  #mv prometheus-2.27.1.linux-amd64/ prometheus  #cd prometheus  # ./prometheus //启动Prometheus    由于prometheus也会对自己进行监控所以访问10.0.1.190:9090测试：    默认指标输出路径：10.0.1.190:9090/meterics    添加Prometheus为systemctl服务管理  #vim /usr/lib/systemd/system/prometheus.service  **[Unit]**  **Description=Prometheus**  **After=network.target**  **[Service]**  **Type=simple**  **Environment="GOMAXPROCS=4"**  **#User=prometheus**  **#Group=prometheus**  **ExecReload=/bin/kill -HUP $MAINPID**  **ExecStart=/data/prometheus/prometheus \**  **--config.file=/data/prometheus/prometheus.yml \**  **--storage.tsdb.path=/data/prometheus/data \**  **--storage.tsdb.retention=30d \**  **--web.console.libraries=/data/prometheus/console\_libraries \**  **--web.console.templates=/data/prometheus/consoles \**  **--web.listen-address=0.0.0.0:9090 \**  **--web.read-timeout=5m \**  **--web.max-connections=10 \**  **--query.max-concurrency=20 \**  **--query.timeout=2m \**  **--web.enable-lifecycle**  **PrivateTmp=true**  **PrivateDevices=true**  **ProtectHome=true**  **NoNewPrivileges=true**  **LimitNOFILE=infinity**  **ReadWriteDirectories=/data/prometheus**  **ProtectSystem=full**  **SyslogIdentifier=prometheus**  **Restart=always**  **[Install]**  **WantedBy=multi-user.target**  #systemctl daemon-reload  #systemctl start prometheus  #systemctl status prometheus  **(3)部署node\_exporter**  在其他节点部署node\_exporter暴露节点指标能让prometheus收集到  在10.0.1.184上：  #cd /data  # tar -xf node\_exporter-1.1.2.linux-amd64.tar.gz  #vim /usr/lib/systemd/system/node\_exporter.service  **[Unit]**  **Description=node\_exporter**  **Documentation=https://prometheus.io/**  **After=network.target**  **[Service]**  **Type=simple**  **#User=prometheus**  **ExecStart=/data/node\_exporter/node\_exporter \**  **--collector.mountstats \**  **--collector.systemd \**  **--collector.ntp \**  **--collector.tcpstat \**  **--web.listen-address=:1880**  **ExecReload=/bin/kill -HUP $MAINPID**  **TimeoutStopSec=20s**  **Restart=always**  **[Install]**  **WantedBy=multi-user.target**  #systemctl daemon-reload  #systemctl start node\_exporter  -web.listen-address=":1880" /修改端口，默认数据获取路径/metrics    **在Prometheus Server上修改配置文件定义target主机**：  # vim prometheus.yml //添加target  scrape\_configs:  # The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.  - job\_name: 'prometheus'  # metrics\_path defaults to '/metrics'  # scheme defaults to 'http'.  static\_configs:  - targets: ['localhost:9090']  **- job\_name: 'zhongshi'**  **metrics\_path: '/metrics'**  **static\_configs:**  **- targets:**  **- '10.0.1.184:1880'**  重启Prometheus Server并查看target状态  # systemctl restart prometheus    通过PromQL查询机器在线状态：    通过标签进行筛选：    统计http总请求并且状态码为200：    统计内存使用率： |

**三、Prometheus自动发现服务**

* Prometheus服务发现的方式
  + 基于文件的服务发现
  + 基于DNS的服务发现
  + 基于API服务的发现：Kubernetes、Consul、Azure、……

|  |
| --- |
| **(1)基于文件的服务发现**  基于文件的服务发现是仅仅略优于静态配置的服务发现方式，它不依赖于任何平台或第三方服务，因为也是最为简单和通用的实现方式。   * Prometheus Server定期从文件中加载Target信息 * 文件可使用JSON和YAML格式，它含有定义的Target列表，以及可选的标签信息； * 下面第一个配置能够将Prometheus默认的静态配置转换为基于文件的服务发现时所需的配置； * 这些文件可由另一个系统生成，例如Puppet、Ansible或Saltstack等配置管理系统，也可能是由脚本基于CMDB定期查询生成   #cd /data/prometheus  #mkdir targets  #cd targets/ //创建目标主机配置目录  #vim node-linux.yaml //添加node target  **- targets:**  **- 10.0.1.184:1880**  **labels:**  **app: node\_exporter**    #vim prometheus-servers.yaml //添加prometheus target  **- targets:**  **- 10.0.1.190:9090**  **labels:**  **app: prometheus-server**    修改Prometheus主配置文件：  #cd /data/prometheus  #cp prometheus.yaml prometheus.yaml-v1  #vim prometheus.yaml //添加自动检测文件配置  **scrape\_configs:**  **- job\_name: 'prometheus'**  **file\_sd\_configs:**  **- files:**  **- targets/prometheus-\*.yaml**  **refresh\_interval: 1m**  **- job\_name: 'nodes'**  **file\_sd\_configs:**  **- files:**  **- targets/node-\*.yaml**  **refresh\_interval: 1m**    重启Prometheus：  #systemctl restart prometheus  访问Prometheus查看target状态：    测试在targets配置里添加主机是否会自动重载配置文件，在10.0.1.183上启动node\_exporter，并把node\_exporter配置成systemctl服务  #cd /data  #tar xf  #vim /usr/lib/systemd/system/node\_exporter.service  **[Unit]**  **Description=node\_exporter**  **Documentation=https://prometheus.io/**  **After=network.target**  **[Service]**  **Type=simple**  **#User=prometheus**  **ExecStart=/data/node\_exporter/node\_exporter \**  **--collector.mountstats \**  **--collector.systemd \**  **--collector.ntp \**  **--collector.tcpstat \**  **--web.listen-address=:1880**  **ExecReload=/bin/kill -HUP $MAINPID**  **TimeoutStopSec=20s**  **Restart=always**  **[Install]**  **WantedBy=multi-user.target**  #systemctl daemon-reload  #systemctl start node\_exporter  在prometheus-server上的targets目录下添加10.0.1.183主机：  #cd /data/prometheus/targets  #vim node-linux.yaml //增加10.0.1.183:1880配置    访问Prometheus查看是否有新的target加入： |

**四、relabel**

**(1)target重新打标**

* 对target重新打标是在数据抓取之前动态重写target标签的强大工具，在每个数据抓取配置中，可以定义多个relabel步骤，它们将按照定义的顺序一次执行
* 对于发现的每个target，Prometheus默认会执行如下操作
  + Job的标签设定为其所属的job\_name的值
  + \_\_address\_\_标签的值为该target的套接字地址”<host>:<port>”
  + Instance标签的值为\_\_address\_\_的值
  + \_\_scheme\_\_标签的值为抓取该target上指标时使用的协议(http或http)
  + \_\_metrics\_path\_\_标签的值为抓取该target上的指标时使用的URL路径，默认为/metrics
  + \_\_param\_<name>标签的值为传递的URL参数中第一个名称为<name>的参数的值
* 重新标记期间，还可以使用该target上以”\_\_meta\_“开头的元标签
  + 各服务发现机制为其target添加的元标签会有所不同
* 重新标记完成后，该target上以”\_\_”开头的所有标签都会被移除
  + 若在relabel的过程中需要临时存储标签值，则要使用\_\_tmp标签名称为前缀进行保存，以避免同Prometheus的内建标签冲突

**Relabel\_configs配置文件使用:**

<relabel\_action>字段用于定义重新标记的行为，其可用取值如下：

* 替换标签值
  + replace：首先将source\_labels中指定的各标签的值进行串连，而后将regex字段中的正则表达式对源标签值进行匹配判定，若匹配，则将target\_label字段中指定的标签的值替换为replacement字段中保存的值
    - replacement可按需引用保存regex中的某个”分组模式“匹配到的值；默认保存整个regex匹配到的内容
    - 进行值替换时，replacement字段中指定标签的值也支持以分组格式进行引用
  + hashmod：将target\_label的值设置为一个hash值，该hash则由modules字段指定的hash模块对source\_labels上各标签的串连值进行hash计算生成
* 删除指标
  + keep：regex**不能**匹配到target上的source\_labels上的各标签的串连值时，则删除该target
  + drop：regex**能**够匹配到target上的source\_labels上的各标签的串连值时，则删除该target
* 创建或删除标签
  + labelmap：将regex对所有的标签名进行匹配判定，而后将匹配到的标签的值赋给replacement字段指定的标签名之上；通常用于取出匹配的标签名的一部分生成新标签
  + labeldrop：将regex对所有的标签名进行匹配判定，能够匹配到的标签将从该target的标签集中删除
  + labelkeep：将regex对所有的标签名进行匹配判定，不能够匹配到的标签将从该target的标签集中删除

注意：要确保在labeldrop和labelkeep操作后，余下的标签集依然能唯一表示该指标

|  |
| --- |
| * Relabel示例之replace   下面的示例将三个源标签的值按顺序串联后，由指定的正则表达式进行模式匹配，而后由replacement引用模式匹配的结果，并加以改造后，将其赋值给endpoint标签  #vim prometheus.yaml  scrape\_configs:  # The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.  - job\_name: 'prometheus'  file\_sd\_configs:  - files:  - targets/prometheus-\*.yaml  refresh\_interval: 1m  - job\_name: 'nodes'  file\_sd\_configs:  - files:  - targets/node-\*.yaml  refresh\_interval: 1m  **relabel\_configs:**  **- source\_labels: //源标签**  **- \_\_scheme\_\_**  **- \_\_address\_\_**  **- \_\_metrics\_path\_\_**  **regex: '(http|https)(.\*)' //正则匹配**  **target\_label: "endpoint" //定义新的标签名称**  **replacement: "${1}://${2}" //$1对应(http|https)匹配的结果,$2对应(.\*)匹配的结果**  **action: replace**  重启Prometheus查看target标签状态：    有新的endpoint标签生成但是http://多了一个; 需要添加配置separator: “”  #vim prometheus.yaml  relabel\_configs:  - source\_labels:  - \_\_scheme\_\_  - \_\_address\_\_  - \_\_metrics\_path\_\_  regex: '(http|https)(.\*)'  target\_label: "endpoint"  replacement: "${1}://${2}"  **separator: ""**  action: replace    重启Prometheus查看状态：     * Relabel示例之labelmap   下面的示例将regex指定的模式对target上的所有标签进行匹配判定，对于匹配到的标签名，它将以该标签名中匹配的部分为前缀，指定的”\_name”为后缀生成新的标签名，而新标签的值则与源标签的值相同  #vim prometheus.yaml  relabel\_configs:  - source\_labels:  - \_\_scheme\_\_  - \_\_address\_\_  - \_\_metrics\_path\_\_  regex: '(http|https)(.\*)'  target\_label: "endpoint"  replacement: "${1}://${2}"  separator: ""  action: replace  **- regex: "(app)"**  **replacement: ${1}\_name**  **action: labelmap**    # systemctl restart prometheus |

**(2)对抓起到的metric重新打标**

* 对metric重新打标是在数据抓取之后动态重写metric标签的工具，在每个数据抓取配置中，可以定义多个metric relabel的步骤，它们将按照定义的顺序依次执行
  + 删除不必要的指标
  + 从指标中删除敏感或不需要的标签
  + 添加、编辑或修改指标的标签值或标签格式
* 对metric重新打标的配置格式与target重新打标的格式相同，但前者要定义在专用的metric\_relabel\_configs字段中
* 注意：更改或添加标签会创建新的时间序列
  + 应该明确地使用各个标签，并尽可能保持不变，以避免创建出一个动态的数据环境
  + 标签是时间序列的唯一约束，删除标签并导致时间序列重复时，可能会导致系统出现问题

|  |
| --- |
| * Metric relabel删除指标 * 在source\_labels字段上，通过指标上元标签“\_\_name\_\_”引用指标名称，而后由regex进行匹配判定，可使用drop action删除匹配的指标或使用keep action仅保留匹配的指标 * 下面的示例用于相应的job上，在发现各target之上，删除以“go\_info”为名称前缀的指标 |

**六、查询持久化**

* 在Prometheus的表达式浏览器进行的查询会生成新的数据序列，但其结果仅会临时保存于Prometheus Server上
* 在样本数据量较大、工作较为繁忙的Prometheus Server上，对于那些查询频率较高且运算较为复杂的查询来说，实时查询可能会存在一定成都的响应延迟
* 记录规则(Recording rule)能够预先运行频繁用到或计算消耗较大的表达式，并将其结果保存为一组新的时间序列
  + 记录规则是定义在prometheus配置文件中的查询，由Server加载后，它能够用于以类似批处理任务的方式在后台周期性的执行并记录查询结果
  + 客户端只需要查询由记录规则生成的结果序列上的样本数据即可，速度远快于实时查询
  + 常用于跨多个时间序列生成的聚合数据，或者计算消耗较大的查询等场景中
  + 多见于同可视化工具结合使用的需求中，也可用于生成可产生告警信息的时间序列
* 告警规则(Alert rule)是另一种定义在Prometheus配置文件中的PromQL表达式，它通常是一个基于查询语句的表达式，该表达式负责出发告警
  + 告警规则中用的查询语句较为复杂时，可将其保存为记录规则，而后通过查询该记录规则生成的时间序列来参与比较，从而避免实时查询导致的较长时间延迟

|  |
| --- |
| **Recording rule使用示例：**  #cd /data/prometheus  #mkdir rules //创建一个规则目录  #vim rules/recording\_rules.yaml  groups:  - name: custom\_rules  interval: 5s  rules:  - record: instance:node\_cpu:avg\_rate5m  expr: 100 - avg(irate(node\_cpu\_seconds\_total{job="nodes",mode="idle"}[5m])) by (instance) \* 100  - record: instance:node\_memory\_MemFree:present  expr: (node\_memory\_MemFree\_bytes / node\_memory\_MemTotal\_bytes) \* 100  - record: instance:root:node\_filesystem\_free\_persent  expr: node\_filesystem\_free\_bytes{mountpoint="/data"} / node\_filesystem\_size\_bytes{mountpoint="/data"} \* 100    #vim prometheus.yml //引入配置文件  rule\_files:  #- alerting\_rules/\*.yaml  **- rules/\*.yaml**  # - "second\_rules.yml"  #systemctl reload prometheus  在prometheus web ui可以看到由对应的规则生成：    通过指标名称进行查询： |

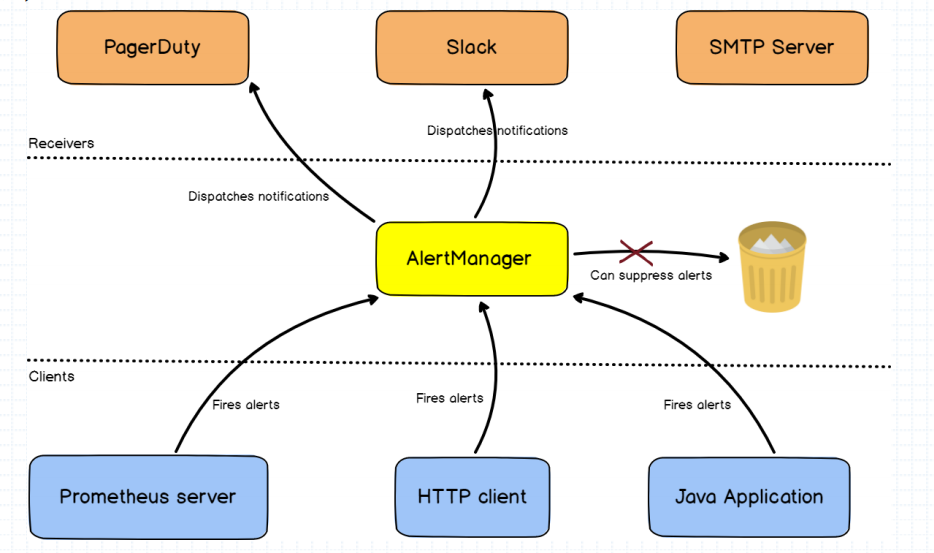
**六、部署Grafana**

|  |
| --- |
| 通过国内镜像站找到Granfana的安装包，例如在华为镜像站上：<https://mirrors.huaweiclound.com/grafana>选择对应的版本进行下载  #cd /data/soft  #wget https://mirrors.huaweicloud.com/grafana/7.5.7/grafana-7.5.7-1.x86\_64.rpm //下载rpm包  #yum -y install grafana-7.5.7-1.x86\_64.rpm  #cd /etc/grafana  #ls //其中grafana.ini 是配置文件  #vim grafana.ini //修改存储路径  data = /data/grafana  #mkdir /data/grafana  #chown -R grafana.grafana /data/grafana/  #systemctl start grafana-server  #systemctl status grafana-status    访问测试，第一次登录需要修改密码：  10.0.1.190:3000  默认账号：admin  默认密码：admin      添加数据源:          导入dashboard模板：  在官网上找到对应的模板idi进行导入 |

1. **Alerts**

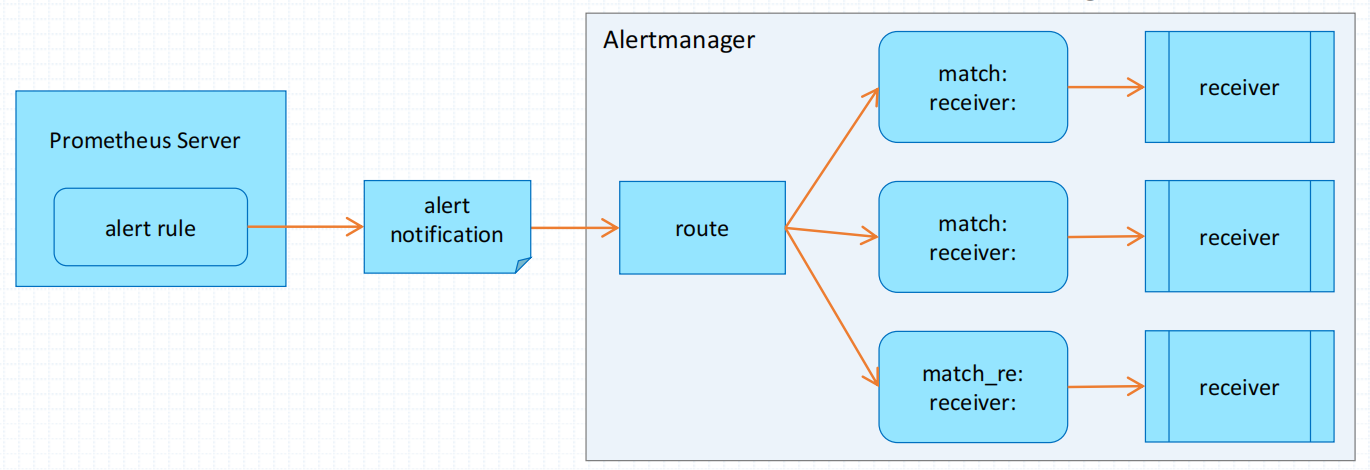
**(1)告警功能概述**

* Prometheus对指标的收集、存储同告警能力分属于Prometheus Server和AlertManager两个独立的组件，前者仅负责基于“告警规则”生成告警通知，具体的告警操作则由后者完成
  + AlertManager负责处理由客户端发来的告警通知
* 客户端通常是prometheus server，但它也支持接收来自其他工具的告警
* AlertManager对告警通知进行分组、去重后，根据路由规则将其路由到不同的receiver，如Email、短信或PagerDuty等



1. **告警逻辑**

* 首先要配置Prometheus成为AlertManger的告警客户端，反过来AlertManager也是应用程序，它自身同样应该纳入Prometheus的监控目标
* 配置逻辑
  + 再AlertManager上定义receiver，他们通常是能够基于某个媒介接受告警信息的特定用户
    - Email、Wechat、Pagerdury、Slack和Webhook等是较为常见的发送告警信息的 媒介
    - 在AlertManager上定义路由规则(route)，以便将接收到的告警通知按需分别进行处理
    - 在Prometheus上定义告警规则生成告警通知，发送给AlertManager



1. **AlertManager**

* 除了基本的告警通知能力外，AlertManager还支持对告警进行去重、分组、抑制、静默和路由等功能
  + 分组(Grouping)：将相似告警合并为单个告警通知的机制，在系统因大面积故障而出发告警潮时，分组机制能够避免用户被大量的告警噪声淹没，进而导致关键信息的隐射
  + 抑制(Inhibition)：系统中某个组件或服务故障而触发告警通知后，那些依赖于该组件或服务的其它组件或服务可能也会因此而出发告警，抑制便是避免类似的级联告警的一种特性，从而让用户能将精力集中于真正的故障所在
  + 静默(Slient)：是指在一个特定的时间窗口内，即便接受到告警通知，AlertManager也不会真正向用户发送告警信息的行为；通常，在系统例行维护期间，需要激活告警系统的静默特性
  + 路由(route)：用于配置AlertManager如何处理传入的特定类型的告警通知，其基本逻辑是根据路由匹配规则的匹配结果来确定处理当前告警通知的路径和行为

|  |
| --- |
| (1)安装Alertmanager  官网下载Alertmanager安装包  #cd /data/soft  #tar -xf alertmanager-0.22.0.linux-amd64.tar.gz -C /data/  #mv alertmanager-0.22.0.linux-amd64 /data/alertmanager  #vim /usr/lib/systemd/system/alertmanager.service  **[Unit]**  **Description=alertmanager**  **Documentation=https://prometheus.io/**  **After=network.target**  **[Service]**  **Type=simple**  **#User=prometheus**  **ExecStart=/data/alertmanager/alertmanager --config.file=/data/alertmanager/alertmanager.yml --storage.path=/data/alertmanager/data**  **Restart=on-failure**  **[Install]**  **WantedBy=multi-user.target**  #systemctl daemon-reload  #systemctl start alertmanager  #systemctl status alertmanager    **(2)Prometheus结合Alertmanager进行邮件报警**  配置动态发现：  #cd /data/prometheus //  #vim targets/alertmanager-servers.yaml  - targets:  - 10.0.1.190:9093  labels:  app: alertmanger-server  #vim prometheus.yml  scrape\_configs:  - job\_name: 'alertmanager'  file\_sd\_configs:  - files:  - targets/alertmanager-\*.yaml  refresh\_interval: 1m  #systemctl reload prometheus    定义告警规则：  #cd /data/prometheus  #mkdir alerting\_rules  #vim alerting\_rles/instances-alert.yaml //报警规则如果有up的值=0持续20s则报警  groups:  - name: AllInstances  rules:  - alert: InstancDown  expr: up == 0  for: 20s  annotations:  title: 'Instance down'  description: 'Instance has been down for more than 1 minute.'  labels:  severity: 'critical'    #vim prometheus.yml //引入配置文件  alerting:  **alertmanagers:**  **- file\_sd\_configs:**  **- files:**  **- targets/alertmanager-\*.yaml**  **# - alertmanager:9093**  rule\_files:  **- alerting\_rules/\*.yaml**  - rules/\*.yaml  #systemctl reload prometheus  可以看到对应的规则      可以看到Prometheus已经发现alert节点，接下来设置alertmanager报警媒介：  #cd /data/alertmanager  #vim alertmanager.yml  global:  resolve\_timeout: 5m  # smtp配置  smtp\_from: "gitlab@sunnsoft.com"  smtp\_smarthost: "smtp.exmail.qq.com:465"  smtp\_auth\_username: "gitlab@sunnsoft.com"  smtp\_auth\_password: "Sunnsoft520\_git"  smtp\_require\_tls: false  route:  receiver: 'ops'  group\_by: ['alertname']  group\_wait: 30s  group\_interval: 5m  repeat\_interval: 10s  receivers:  - name: ops  email\_configs:  - to: 'zhangzs520@dingtalk.com'  #systemctl reload alertmanager  找一台装有node\_exporter的主机进行测试  #systemctl stop node\_exporter  然后在Prometheus上可以看到报警状态由inactive -> pending -> firing，当状态时firing时则调用alert进行报警  状态：      可以看到邮件通知：    Alertmanager配置说明：   * Global：全局配置，主要配置告警方式，如邮件、webhook等 * Route：Prometheus的告警先是到达alertmanager的根路由(route)，alertmanager的根路由不能包含任何匹配项，因为根路由是所有告警的入口。另外，根路由需要配置一个接收器(receiver)，用来处理那些没有匹配到任何子路由的告警(如果没有配置子路由，则全部由根路由发送告警)。 * Group\_by：用于分组聚合，对告警通知按标签(label)进行分组，将具有相同标签或者相同告警名称(alertname)的告警通知聚合在一个组，然后作为一个通知发送。如果想要完全禁用聚合，可以设置为group\_by * Group\_wait：当一个新的告警组被创建时，需要等待‘group\_wait’后才发送初始通知。这样可以确保在发送等待前能收集更多具有相同标签的告警，最后合并为一个通知发送 * Group\_interal：当第一次告警通知发出后，等待’group\_interval’时间后，开始发送该组触发的的新告警，可以理解为，group相当于一个通道(channel)。 * Repeat\_interal：告警通知成功发送后，若问题一直未修复，则再次重复发送的间隔。   检测配置文件语法正确命令：  # ./amtool check-config alertmanager.yml |