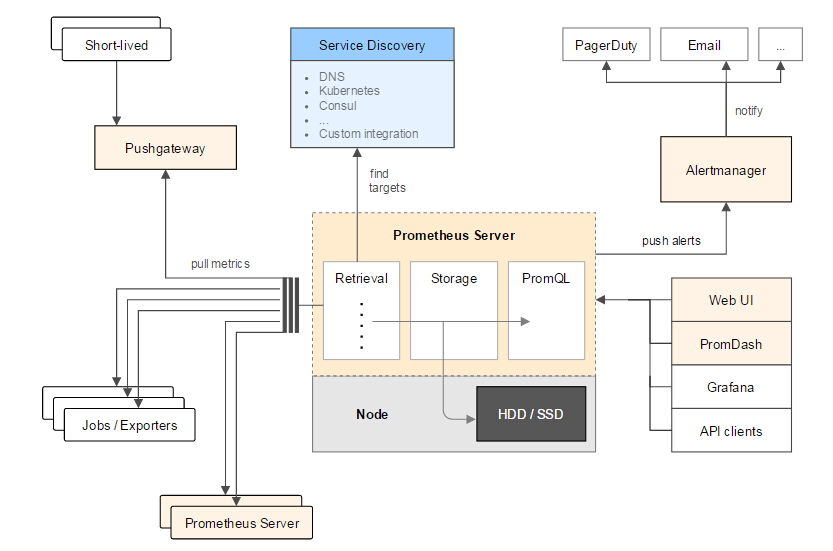
# 基于Prometheus的分布式在线服务监控实践

- 知乎专栏 https://zhuanlan.zhihu.com/p/24811652



基于Prometheus的分布式在线服务监控实践

[范禹](https://www.zhihu.com/people/fanyu83)[范禹](https://www.zhihu.com/people/fanyu83)

4 个月前

本文可以看做是对[《SRE》](http://link.zhihu.com/?target=https%3A//book.douban.com/subject/26875239/)一书第10章《基于时间序列数据进行有效报警》的实践总结。

Prometheus是一款开源的业务监控和时序数据库，可以看作是Google内部监控系统Borgmon的一个（非官方）实现。

本文会介绍我近期使用Prometheus构建的一套完整的，可用于中小规模（小于500节点）的半自动化（少量人工操作）监控系统方案。

主动监控

监控是运维系统的基础，我们衡量一个公司/部门的运维水平，看他们的监控系统就可以了。

监控手段一般可以分为三种：

* 主动监控：业务上线前，按照运维制定的标准，预先埋点。具体的实现方式又有多种，可能通过日志、向本地Agent上报、提供REST API等。
* 被动监控：通常是对主动监控的补充，从外围进行黑盒监控，通过主动探测服务的功能可用性来进行监控。比如定期ping业务端口。
* 旁路监控：主动监控和被动监控，通常还是都在内部进行的监控，内部运行平稳也不能保证用户的体验都是正常的（比如用户网络出问题），所以仍然需要通过舆情监控、第三方监控工具等的数据来间接的监控真实的服务质量。

主动监控是最理想的方案，后两种主要用作补充，本文只关注主动监控。

监控实际是一个端到端的体系（基础设施-服务器-业务-用户体验），本文只关注业务级别的主动监控。

**Prometheus**

为什么选择Prometheus而不是其它TSDB实现（如InfluxDB）？主要是因为Prometheus的核心功能，查询语言PromQL，它更像一种可编程计算器，而不是其那么像SQL，也意味着PromQL可以近乎无限之组合出各种查询结果。

比如，我们有一个http服务，监控项http\_requests\_total用于统计请求次数。某一组监控数据可能是这个样子：

http\_requests\_total{instance="1.1.1.1:80",job="cluster1",location="/a"} 100

http\_requests\_total{instance="1.1.1.1:80", job="cluster1", location="/b"} 110

http\_requests\_total{instance="1.1.1.2:80", job="cluster2", location="/b"} 100

http\_requests\_total{instance="1.1.1.3:80", job="cluster3", location="/c"} 110

这里有3个标签，分别对应抓取的实例，所属的job（一般我用集群名），访问路径（你可以理解为nginx的location），Prometheus多维数据模型意味着我们可以在任意一个或多个维度进行计算：

* 如果你想统计单机qps，sum(rate(http\_requests\_total[1m])) by (instance)
* 如果想用统计每个集群每个不同location的path的qps，sum(rate(http\_requests\_total[1m])) by (job, path)，PromQL会依据标签job-path的值聚合出结果。

除了PromQL，丰富的数据类型可以提供更有意义的监控项：

* Counter（计数器）：标识单调递增的数据，比如接口访问次数。
* Gauge（刻度）：当前瞬时的一个状态，可能增加，也可能减小，比如CPU使用率，平均延时等等。
* Historgram（直方图）：用于统计数据的分布，比如95 percentile latency。

大部分监控项都可以使用Counter来实现，少部分使用Gauge和Histogram，其中Histogram在服务端计算是相当费CPU的，所以也没要导出太多Histogram数据。

最后，Prometheus采用PULL模型的实时抓取存储计算，主动去抓取监控实例数据，相比于PUSH模型对业务侵入更低，相比于基于log的离线统计则更实时，而监控实例只需提供一个文本格式的/metrics接口也更容易debug。

**服务框架的改造**

笔者所在团队使用统一的服务框架来规范项目开发并有效降低了开发难度。

这里先介绍下我们的服务框架：

* 类似于nginx的多进程架构（master/worker），但同时也支持多线程的事件循环编程模型
* 支持多种接入协议（HTTP，Thrift，PB等），但主流是HTTP
* 业务通过Module来加载进框架执行（类似nginx的module，但更简单）
* 提供纯异步的下游访问API

为了使服务框架可以导出内部监控项，主要涉及几方面的工作：

* 提供基础数据类型
  + 目前并没有官方的Prometheus Client Library，几种开源实现也都不太符合框架的需求。目前实现了支持多线程多进程的Counter和Histogram（除了初始化之外，更新操作都是无锁的），而Gauge由于多进程场景有的情况是无法聚合监控数据的（没用统一的聚合方法，并不一定都可以相加），所以没有提供具体实现
  + 基础数据要有类似注册表的功能，方便自动导出数据到/metrics接口
* 在服务框架埋点
  + 要足够灵活，将容易变化的信息通过标签来表达。比如一个web服务可能有echo，date两个location，如果要统计它们qps，不要定义echo\_requests\_total, date\_requests\_total两个不同名字的metrics，而应该定义一个名为http\_requests\_total的metrics，通过标签location（分别为echo/date）来区分，这样再增加/减少接口是不需要改代码的
  + 理想情况是业务几乎为各种通信功能自行埋点，所以内置埋点要将常用监控项都要覆盖到（QPS，Latency，Error Ratio）

数据的抓取与展现

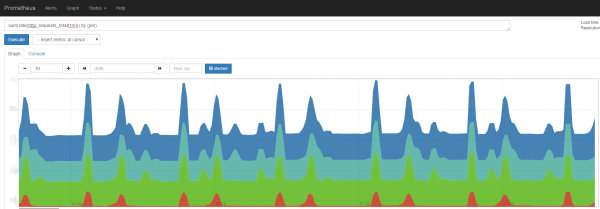
具备导出能力后，就可以通过Prometheus进行抓取了，但还有几个小坑：

* 用户定义的metrics名字，可能是不符合Prometheus规范的，而遇到一条不合法的数据，Prometheus就会停止抓取，所以导出数据时要先做一遍过滤和改写
* 要控制导出数据规模，一些只对单机监控有意义的数据可以不导出（框架有针对单机的监控页面）

在使用Prometheus时，也有几个地方要注意：

* Prometheus即是一个CPU密集型（查询）也是一个IO密集型（数据落地）的，CPU数量是多多益善，内存越大越好（来缓存抓取的数据，所以应该减少不必要的业务数据导出），尽量要使用SSD（这个很关键！），因为一旦Prometheus的内存使用量达到阈值会停止抓取数据！这个停止抓取的时间，至少是分钟级，甚至是无法恢复！所以只要有条件就要用SSD。
* Prometheus号称支持reload，但目测不是很好用，比如你修改了告警规则文件，重载之后，新旧告警规则似乎会一起计算执行....

Prometheus本身也提供图形界面，但是很简陋：

通常还是使用Grafana来展示监控数据。

因为是统一的业务框架，统一的监控指标，所以Grafana的Dashboard很容易统一配置：

* 我没有找到将默认模板打包进Grafana的方法，只能迂回的创建了一个新的Grafana Plugin，在启动之后，每个业务实例只需要启动下这个插件，然后配置一个默认的Prometheus数据源，就可以使用统一的监控Dashboard
* Dashboard分为3行
  + 第一行展示实时的QPS，平均延时，平均排队时间，Coredump数量，下游引擎失败率，下游引擎延时变化
  + 第二行展示业务的延迟（50%和95%延迟），流量，吞吐（按照不同错误码）
  + 第三行展示下游引擎的延迟（50%和95%延迟），流量，吞吐（按照不同错误码）

能够展示Prometheus强大威力的是，这里面每一个图表，都可以同时展示所有机房的监控指标，而每一个指标的计算只需要一条Query语句。比如第一行第五列，各个机房的各个下游的失败率统计并排序，只用了一条语句：

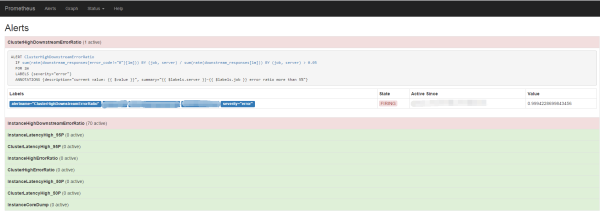
topk(5, 100\*sum(rate(downstream\_responses{error\_code!="0"}[5m])) by (job, server)/sum(rate(downstream\_responses[5m])) by (job, server))

注意这里的Range Vector Selector - [5m]，意味着我们是基于过去5分钟的数据来计算rate，这个值取的越小，得到的监控结果波动越大，越大则越平滑，选择多大的值，取决于你想要什么结果。建议图表使用5m，而告警规则计算采用1m。如果业务不是很重要，可以适当增大这个值。

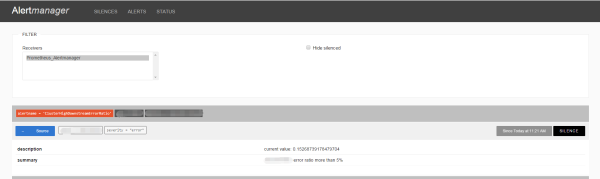
这一套监控模板基本覆盖了业务对可用性监控的需求，同时业务也可以自己定义监控指标并进行监控。

AlertManager

Prometheus周期性进行抓取数据，完成抓取后会检查是否有告警规则并进行计算，满足告警规则就会触发告警，发送到alertmanager。基于这个流程，当你在监控图表看到异常时，告警已经先行触发了。

默认情况我们配置了不到10条告警规则，要注意的是周期的选择，过长的话会产生较大延迟，太短的话一个小的流量波动都会导致大量报警出现。

Prometheus的设计是指产生报警，但报警的汇总、分发、屏蔽则在AlertManager服务完成。



AlertManager目前还是非常简单的，但它可以将告警继续分发到其他接收者：

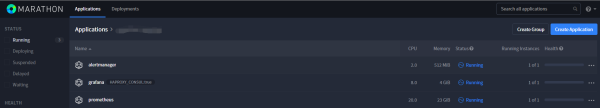
* 可以通过webhook机制，发送告警到一个中间服务转换格式再发送到内部告警接口
* 如果使用第三方告警管理平台，如PageDuty、OneAlert，可以直接用内置的pageduty支持或webhook发送告警过去
* 如果是一穷二白的团队，建议配置email + slack，实现告警归档和手机Push

更复杂告警分级管理，AlertManager还是有很长的路要走，这个话题也值得今后单独讲下。

Prometheus + Grafana + Mesos

Prometheus + Grafana的方案，加上统一的服务框架，可以满足大部分中小团队的监控需求。我们将这几个组件打包一起部署在Mesos之上，统一的安装包进一步降低监控系统部署的难度，用户进需要配置一些简单的参数即可。但还需要注意几点：

* 目前并没有将Prometheus和Grafana容器化部署，因为这两者本身就没有什么特殊依赖；安装包存储在minio中。
* 由于Prometheus系统的特殊性，我们通常将其指定在一台固定的机器上执行，且将数据落地到一个固定的目录，这样重启Prometheus的影响会非常低
* Grafana是展示给用户的，需要尽可能的保持固定入口，所以我们通过HAPROXY\_CONSUL给其配置了代理



结论

Prometheus是相当强大并快速成长的一个监控系统实现，虽然在稳定性、性能、文档上仍有很大提升空间，但对于中小团队是一个很棒的选择，通过定制服务框架，设计完善的埋点，统一的Prometheus/Grafana配置模板，再加上Mesos平台，可以半自动化的部署实时业务监控系统。

# 2@Prometheus VS InfluxDB – Tiantian Gao ( gtt116 )

https://www.gaott.info/prometheus-vs-influxdb/

**目录**[[隐藏](https://www.gaott.info/prometheus-vs-influxdb/)]

* [1 前言](https://www.gaott.info/prometheus-vs-influxdb/#i)
* [2 监控系统](https://www.gaott.info/prometheus-vs-influxdb/#i-2)
  + [2.1 OpenTSDB](https://www.gaott.info/prometheus-vs-influxdb/#OpenTSDB)
  + [2.2 InfluxDB](https://www.gaott.info/prometheus-vs-influxdb/#InfluxDB)
  + [2.3 Prometheus](https://www.gaott.info/prometheus-vs-influxdb/#Prometheus)
* [3 Push vs Pull](https://www.gaott.info/prometheus-vs-influxdb/#Push_vs_Pull)
  + [3.1 发起者不同](https://www.gaott.info/prometheus-vs-influxdb/#i-3)
  + [3.2 逻辑架构不同](https://www.gaott.info/prometheus-vs-influxdb/#i-4)
* [4 查询语法](https://www.gaott.info/prometheus-vs-influxdb/#i-5)
  + [4.1 基本查询](https://www.gaott.info/prometheus-vs-influxdb/#i-6)
* [5 基本的算数计算](https://www.gaott.info/prometheus-vs-influxdb/#i-7)
* [6 计算速度](https://www.gaott.info/prometheus-vs-influxdb/#i-8)
* [7 维度之间的计算](https://www.gaott.info/prometheus-vs-influxdb/#i-9)
* [8 总结](https://www.gaott.info/prometheus-vs-influxdb/#i-10)

前言

除了传统的监控系统如 Nagios，Zabbix，Sensu 以外，基于时间序列数据库的监控系统随着微服务的兴起越来越受欢迎，比如 Prometheus，比如 InfluxDB。gtt 也尝试了一下这两个系统，希望能找到两者的差别，为以后选型提供一些帮助。

首先，说道时间序列数据库不得不说老牌的 rrdtools 和 [graphite](https://graphite.readthedocs.io/)，这些经典老系统工作的非常好，除了有人嫌弃它们在巨大规模情景下不 scale，嫌弃它们部署不方便外。于是有了 OpenTSDB，Prometheus，InfluxDB 等这些后起之秀。

监控系统

OpenTSDB

[OpenTSDB](http://opentsdb.net/)：基于 Hadoop and HBase 的时间序列数据库，它最先提出了为 metric 增加 tag（key-value 键值对） 的方法来实现更方便和强大的查询语法，InfluxDB 的设计和查询语法受它的启发很大。OpenTSDB 基于 Hadoop 和 HBase 的实现了变态的横向扩展能力，但是也因为这两个依赖，对于不熟悉 Hadoop 这套系统的团队来说，OpenTSDB 的维护成本很高，于是有人搞出了 InfluxDB。

InfluxDB

[InfluxDB](https://influxdata.com/)：InfluxData 公司使用 golang 实现的时间序列数据库，InfluxDB 的口号之一就是：From the ground up，没有任何外部依赖，就一个可执行文件，丢到服务器上就可以运行，对运维非常之友好。语法的设计很大程度受到 OpenTSDB 的启发。虽然项目初期标榜了自带集群功能，可以非常轻松地实现横向扩展，但是在在 InfluxDB 1.0 之后集群功能被删除，取而代之的是通过 Relay 模式实现高可用，官方文档上挂出如下说明，[但是0.9版本的集群使用说明在官网上仍然能访问到，估计未来被删除的可能性非常大](https://docs.influxdata.com/influxdb/v0.9/guides/clustering/)。

*Note: Clustering is now a commerial product called InfluxEnterprise. More information can be found here.*

Prometheus

[Prometheus](http://prometheus.io/)：SoundCloud 开源的监控系统，已经提交给开源社区独立运营。并且和 k8s 一样都为[Cloud Native Computing Foundation](https://cncf.io/) 的成员，虽然目前这个 Foundation 只有 k8s 和 Prometheus 两个项目。Prometheus 和上面两者最大的区别可以理解成：上面两者仅仅是数据库，而 Prometheus 是一个监控系统，它不仅仅包含了时间序列数据库，还有全套的抓取、检索、绘图、报警的功能。[官方也对这种区别做了详细的描述](https://prometheus.io/docs/introduction/comparison/)。

它很大程度收到了 Google 内部的 Borgmon 系统的启发，基于拉（pull）模式实现的监控系统。在《Site Reliability Engineering》一书中有这句话提到 Prometheus，当然我不会告诉你原文其实还提到了 Bosun 和 Riemann 这两个监控系统，这是为什么这面这句话末尾有个省略号：

*Even though Borgmon remains internal to Google, the idea of treating time-series data as a data source for generating alerts is now accessible to everyone through those open source tools like Prometheus […]»  
— Site Reliability Engineering: How Google Runs Production Systems (O’Reilly Media)*

不过 InfluxData 公司也推出了整套的围绕时间序列数据库的解决方案：TICK，功能覆盖了数据获取（Telegraf ）、存储和查询（InfluxDB）、图表绘制（Chronograf ）、报警（Kapacitor ）。这套解决方案和 Elastic 公司的做法特别像：围绕着 ElasticSearch 核心功能，收购了 Logstash，Kibana，又搞出了了 Beat、Watcher 等外围服务打造完整的功能完备的全文检索解决方案。

扯得有点远，回到文章的核心内容：InfluxDB 和 Prometheus 的区别是啥。目前主要区别在于：前者仅仅是一个数据库，它被动的接受客户的数据插入和查询请求。而后者是完整的监控系统，能抓取数据、查询数据、报警等功能。

Push vs Pull

到此，我们知道 Prometheus 是基于 pull 的，InfluxDB 是基于 push 的。关于 push 和 pull 之前写过 [ansible 和 puppet 的对比](https://www.gaott.info/which-one-ansible-or-puppet/)，但是在监控系统上，又有了微妙的差别。

首先，Push 和 Pull 描述的是数据传输的方式，它不影响传输的内容。换言之，只要是 push 能够携带的信息，pull 肯定也能携带同样的信息，比如 ”CPU 利用率 30%“ 这样的监控数据，不管是 pull 还是 push，传输的内容还是这些，不会因为传输模式改变导致消息体积暴长，因此两种方式消耗的网络带宽不会差别很大。

gtt 认为 Push 和 Pull 的主要区别在：

发起者不同

pull 的发起人是监控系统，它依次轮询被监控目标，所以如果目标在防火墙内或者 NAT 之后，则 Pull 方式行不通。并且，对于批处理（batch）类型的任务，因为可能整个处理时间小于轮询的间隔时间，因此监控系统会捕捉不到这类任务的数据。

为了解决这两个问题 Prometheus 提供了 pushgateway\_exporter 组件来支持 push 模式的监控需求。

push 要求发起人是被监控目标，所以它可以突破防火墙限制，即使目标躲在 NAT 之后，仍然能顺利将数据推送出来，对于批处理类型的任务也能比较从容的发出数据。

另外，有人说“push 模式下监控系统是单点，会有单点故障和性能瓶颈而 pull 模式则没有。”

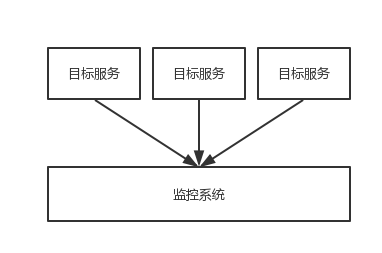
这里 gtt 不同意，因为 pull 解决单点故障的方法是增加另外一个监控系统，本质上是是通过数据冗余提高可靠性，那么 push 为什么不能推送到两个监控系统上呢，这样也能做到数据冗余。

对于性能瓶颈，这点更不成立，因为不管是 push 还是 pull，影响的只是传输方式，对传输的数据内容没有影响，占用的带宽是一样的。那么唯一区别是并发度可能不一样，push 模式下，目标服务可能在某个时间内集中向监控系统推送数据，导致瞬间并发请求很大，类似 DDos 攻击。相反地，pull 以此轮询目标服务，能够按照自己能够承受的并发度处理监控数据，避免了监控数据短时间内爆发的情况。但解决办法也有，在 push 模式下，给监控服务加上请求处理队列，超过监控系统负载的请求暂存在队列中，这样监控系统就能按照自己的节奏来处理数据，防止被队友给DDos。

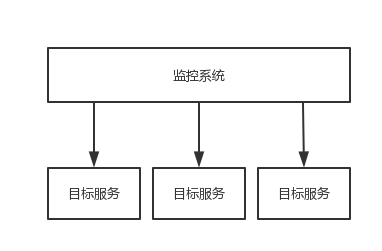
所以单点问题、性能问题不是两种模式的本质区别。

逻辑架构不同

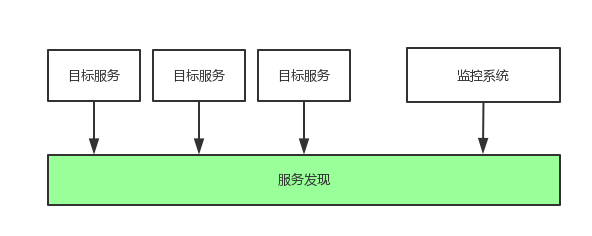
push 要求被监控目标知道监控系统的地址（IP或者域名），所以这部分信息需要设置在目标服务中，换言之，目标服务依赖监控系统。监控系统如果地址改变，所有目标服务都需要做相应的改动。而一旦产生依赖意味着监控系统故障，可能会影响到目标服务正常运行，当然在编程时可以做一些规避，但是逻辑上仍然是目标服务依赖监控系统。架构图下图所示：

[](https://www.gaott.info/wp-content/uploads/2016/09/%E7%9B%91%E6%8E%A7%E7%B3%BB%E7%BB%9F.png)

pull 要求监控系统知道所有目标服务的地址，目标服务对监控系统是不知情的。所以监控系统依赖目标服务，每次新增加一个目标服务，对监控系统做配置修改。从这点区别上看，pull 模式更加符合逻辑架构。为了自动化处理目标服务的增加和删除，Prometheus 支持从服务发现系统中动态获取目标服务的地址，省去了大规划微服务部署情况下复杂的配置需求。逻辑架构如下图所示：

[](https://www.gaott.info/wp-content/uploads/2016/09/%E7%9B%91%E6%8E%A7%E7%B3%BB%E7%BB%9F-2.png)

鸡贼的人应该发现了，那岂不是服务发现系统被所有目标服务依赖了？是的，服务发现系统和监控系统在逻辑架构上处于不同地位，在有服务发现的架构中，如果目标服务没有被“发现”，它实际上是不能正常提供服务的，所以必须依赖服务发现系统，而相反，目标服务在没有监控系统的情况下仍然可以正常运行。逻辑架构如下图所示：

[](https://www.gaott.info/wp-content/uploads/2016/09/%E7%9B%91%E6%8E%A7%E7%B3%BB%E7%BB%9F-3.png)

因为不依赖监控系统，即使没有部署监控服务，人工判断目标服务是否正常也非常容易，只要模拟监控系统访问目标服务的某个接口即可，所以 pull 模式下的监控更向白盒，你可以很轻松的获取到所有信息。相反的，push 模式依赖于一个成型的监控服务，没有监控服务就完全不知道目标服务运行状况如何，这点比较让人难以接受。

查询语法

到此，我们知道 Prometheus 是基于 pull 模式获取数据，InfluxDB 是基于 push 模式获取数据。现在关注两者在数据查询上的区别。

比如获取磁盘 IO 时间的数据：

时间戳 metric: 值 tag

1475216224 disk\_io\_time:10 type="sda"

1475216224 disk\_io\_time: 30 type="sdb"

1475216224 disk\_io\_time: 11 type="sdc"

1475216224 disk\_io\_time: 18 type="sde"

基本查询

作为基本的时间序列数据库，两者对数据的基本获取都很简单。

InfluxDB:

SELECT mean("value") FROM "disk\_io\_time" WHERE $timeFilter GROUP BY time($interval), "instance" fill(null)

Prometheus:

disk\_io\_time

基本的算数计算

两者的差别也不大：

InfluxDB:

SELECT mean("value") \*1024 FROM "disk\_io\_time" WHERE $timeFilter GROUP BY time($interval), "instance" fill(null)

Prometheus:

disk\_io\_time\*1024

计算速度

InfluxDB：

SELECT derivative(mean("value"), 10s) \*1024 FROM "disk\_io\_time" WHERE $timeFilter GROUP BY time($interval), "instance" fill(null)

Prometheus：

rate(disk\_io\_time)\*1024

维度之间的计算

这点是目前为止 gtt 发现的两者最大区别。比如我需要 sda 和 sdc 的 io 时间相加，InfluxDB 还不支持这样的语法，不过社区已经在讨论相关的实现了：[[feature request] Mathematics across measurements #3552](https://github.com/influxdata/influxdb/issues/3552)。

而 Prometheus 能够完成这个任务：

rate(disk\_io\_time{type="sda"}) + rate(disk\_io\_time{type="sdc"})

总结

整体比较下来，Prometheus 是一个靠谱的监控系统，它的设计深受到 Google 内部 Borgmon 系统的启发，并且有着优雅的查询语法，不过是基于拉（pull）模式的，需要在具体业务中做抉择。而 InfluxDB 仅仅是时间序列数据库，没有其他监控相关的功能，不过 InfluxData 公司还提供了配套的其他组件可供选择。于 Prometheus 相比，它的查询语法更加复杂，并且不支持维度之间的计算。

[devops](https://www.gaott.info/tag/devops/), [monitoring](https://www.gaott.info/tag/monitoring/)

# Prometheus 系统监控方案 二 安装与配置 - Vovolie - 博客园

http://www.cnblogs.com/vovlie/p/Prometheus\_install.html

[Prometheus 系统监控方案 二 安装与配置](http://www.cnblogs.com/vovlie/p/Prometheus_install.html)

下载Prometheus

[下载最新安装包](https://prometheus.io/download)，本文说的都是在Linux x64下面内容，其它平台没尝试过，请选择合适的下载。

* Prometheus 主程序，主要是负责存储、抓取、聚合、查询方面。
* Alertmanager 程序，主要是负责实现报警功能。
* Pushgateway 程序，主要是实现接收由Client push过来的指标数据，在指定的时间间隔，由主程序来抓取。
* \*\_exporter 这类是不同系统已经实现了的集成。

下载解压，一般默认的配置就可以运行。

tar xvfz prometheus-\*.tar.gz

cd prometheus-\*

go写的东西，直接打包成二进制包了，其本上没有别的依赖。

下面我们主要来了解一下他的配置文件，这是一个非常重要的步聚。

配置Prometheus监控自已

Prometheus 通过默认 \*[http://localhost:9090/metrics\*](http://localhost:9090/metrics*) HTTP接口暴露了自己的性能指标数据，当然也就可以配置抓取目标 targets 为自己了。Prometheus 采集自身性能数据就是一个十分好的例子了，打开解压目录下面的**prometheus.yml**文件。

# 全局配置

global:

scrape\_interval: 15s # 默认 15秒到目标处抓取数据

# 这个标签是在本机上每一条时间序列上都会默认产生的，主要可以用于 联合查询、远程存储、Alertmanger时使用。

external\_labels:

monitor: 'codelab-monitor'

# 这里就表示抓取对象的配置

# 设置抓取自身数据

scrape\_configs:

# job name 这个配置是表示在这个配置内的时间序例，每一条都会自动添加上这个{job\_name:"prometheus"}的标签。

- job\_name: 'prometheus'

# 重写了全局抓取间隔时间，由15秒重写成5秒。

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

启动Prometheus

使用刚才的配置文件启动Prometheus。

./prometheus -config.file=prometheus.yml

这时候Prometheus应该正确启动了，如果报错，请检查配置文件。（注意：yml格式是对缩进有要求的。） 使用浏览器打开该机器的\*[http://部署机器:9090\*，或者http://localhost:9090/](http://xn--:9090%2A%2Chttp-2l5wy59jkgro97g7fcny3k/localhost:9090/) 即可以看到Prometheus的graph页面了。一般等几秒抓取，就会有数据写进Prometheus里面了。

如果相验证Prometheus自己输出了什么性能数据，可以打开 <http://localhost:9090/metrics> 这个页面看看。

Prometheus默认是有多少个CPU内核就使用多少OS线程，主要是由GOMAXPROCS 这个环境变量控制的，开发GO的应该都清楚。一般默认就好了，太高的话可能会带来意想不到的后果。  
Prometheus默认大概会占用3G左右的内存，如果想调小一点，得修改配置文件，或者添加启动参数。

使用Prometheus 自带的表达式浏览器

如果想查看Prometheus都抓了些什么数据，可以使用它内置的一个浏览页面。打开\*[http://localhost:9090/graph\*，默认是在graph的页面](http://localhost:9090/graph*%EF%BC%8C%E9%BB%98%E8%AE%A4%E6%98%AF%E5%9C%A8graph%E7%9A%84%E9%A1%B5%E9%9D%A2)。  
试试在表达式 expression 输入框：

prometheus\_target\_interval\_length\_seconds

查询Promethues目标抓取间隔时间长度。执行后会返回很多条时间序列，Console 内列出的是最后一次抓到的数值。不同的标签表示不同的时间间隔和耗时%。

如果想查询99%的抓取耗时，可以这样：

prometheus\_target\_interval\_length\_seconds{quantile="0.99"}

如果想查询返回的时间序列有多少条，可以这样：

count(prometheus\_target\_interval\_length\_seconds)

暂时写到这了。

分类: [Prometheus](http://www.cnblogs.com/vovlie/category/909459.html)

标签: [Prometheus](http://www.cnblogs.com/vovlie/tag/Prometheus/)

# Prometheus 系统监控方案 一 - Vovolie - 博客园

http://www.cnblogs.com/vovlie/p/Prometheus\_CONCEPTS.html

[Prometheus 系统监控方案 一](http://www.cnblogs.com/vovlie/p/Prometheus_CONCEPTS.html)

最近一直在折腾时序类型的数据库，经过一段时间项目应用，觉得十分不错。而Prometheus又是刚刚推出不久的开源方案，中文资料较少，所以打算写一系列应用的实践过程分享一下。

Prometheus 是什么？

Prometheus是一套开源的监控&报警&时间序列数据库的组合，起始是由[SoundCloud](https://soundcloud.com/)公司开发的。随着发展，越来越多公司和组织接受采用Prometheus，社会也十分活跃，他们便将它独立成开源项目，并且有公司来运作。google SRE的书内也曾提到跟他们BorgMon监控系统相似的实现是Prometheus。现在最常见的Kubernetes容器管理系统中，通常会搭配Prometheus进行监控。

Prometheus 的优点

* 非常少的外部依赖，安装使用超简单
* 已经有非常多的系统集成 例如：docker HAProxy Nginx JMX等等
* 服务自动化发现
* 直接集成到代码
* 设计思想是按照分布式、微服务架构来实现的

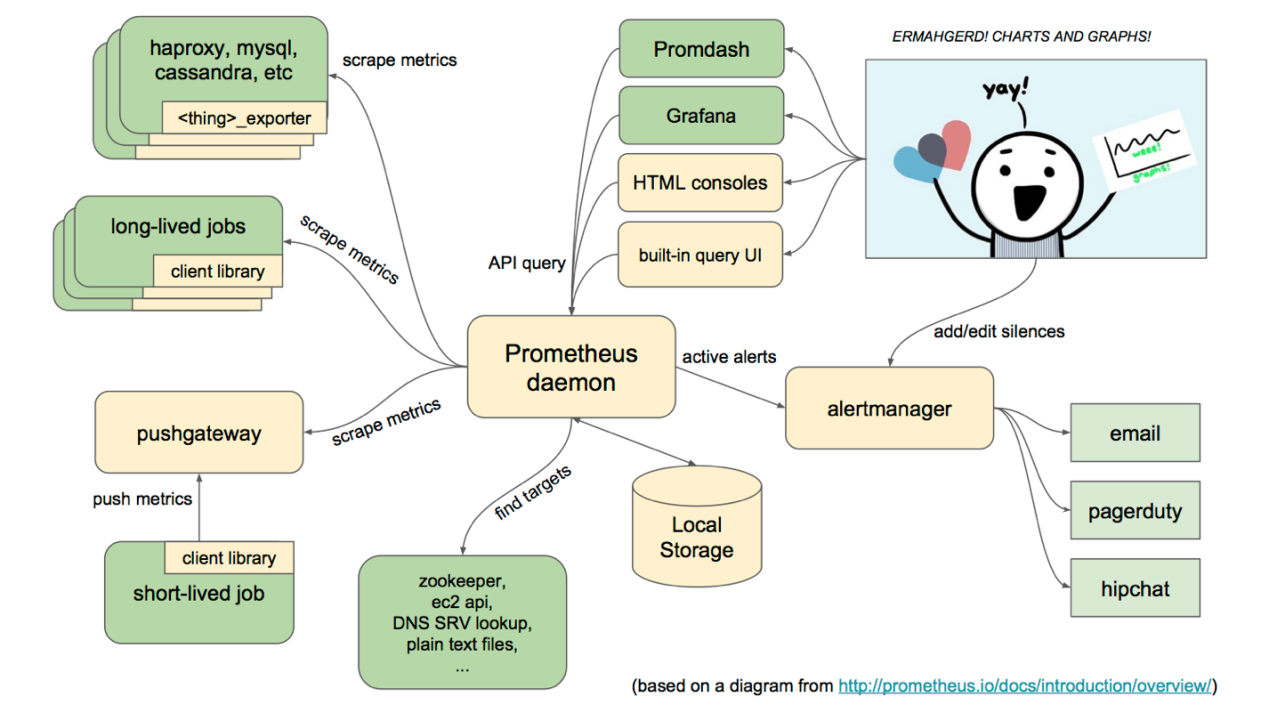
Prometheus 的特性

* 自定义多维度的数据模型
* 非常高效的存储 平均一个采样数据占 ~3.5 bytes左右，320万的时间序列，每30秒采样，保持60天，消耗磁盘大概228G。
* 强大的查询语句
* 轻松实现数据可视化

等等

相对于Graphite这种产品，还是有不少优点的。最让我觉得不错的是非常优秀的写性能和读取性能，它数据结构实现和OpenTSDB是有相似之处，有兴趣可以看看这个文档。[解密OpenTSDB的表存储优](https://yq.aliyun.com/articles/54785)

Prometheus 的系统架构



它的服务过程是这样的 Prometheus daemon 负责定时去目标上抓取 metrics(指标) 数据，每个抓取目标需要暴露一个http服务的接口给它定时抓取。  
Prometheus支持通过配置文件、文本文件、zookeeper、Consul、DNS SRV lookup等方式指定抓取目标。  
Alertmanager 是独立于Prometheus的一个组件，可以支持Prometheus的查询语句，提供十分灵活的报警方式。  
Prometheus支持很多方式的图表可视化，例如十分精美的Grafana，自带的Promdash，以及自身提供的模版引擎等等，还提供HTTP API的查询方式，自定义所需要的输出。  
PushGateway这个组件是支持Client主动推送 metrics 到PushGateway，而Prometheus只是定时去Gateway上抓取数据。

如果有使用过statsd的用户，则会觉得这十分相似，只是statsd是直接发送给服务器端，而Prometheus主要还是靠进程主动去抓取。

Prometheus 的数据模型

Prometheus 从根本上所有的存储都是按时间序列去实现的，相同的 metrics(指标名称) 和 label(一个或多个标签) 组成一条时间序列，不同的label表示不同的时间序列。为了支持一些查询，有时还会临时产生一些时间序列存储。

metrics name & label 指标名称和标签

每条时间序列是由唯一的 指标名称 和 一组 标签 （key=value）的形式组成。  
指标名称 一般是给监测对像起一名字，例如 http\_requests\_total 这样，它有一些命名规则，可以包字母数字\_之类的的。  
通常是以应用名称开头\_监测对像\_数值类型\_单位这样。  
例如：

1. push\_total
2. userlogin\_mysql\_duration\_seconds
3. app\_memory\_usage\_bytes

标签 就是对一条时间序列不同维度的识别了，例如 一个http请求用的是POST还是GET，它的endpoint是什么，这时候就要用标签去标记了。  
最终形成的标识便是这样了

http\_requests\_total{method="POST",endpoint="/api/tracks"}

记住，针对http\_requests\_total这个metrics name 无论是增加标签还是删除标签都会形成一条新的时间序列。  
查询语句就可以跟据上面标签的组合来查询聚合结果了。  
如果以传统数据库的理解来看这条语句，则可以考虑 http\_requests\_total是表名，标签是字段，而timestamp是主键，还有一个float64字段是值了。（Prometheus里面所有值都是按float64存储）

Prometheus 的四种数据类型

Counter

* Counter 用于累计值，例如 记录 请求次数、任务完成数、错误发生次数。
* 一直增加，不会减少。
* 重启进程后，会被重置。

例如：http\_response\_total{method="GET",endpoint="/api/tracks"} 100  
10秒后抓取 http\_response\_total{method="GET",endpoint="/api/tracks"} 100

Gauge

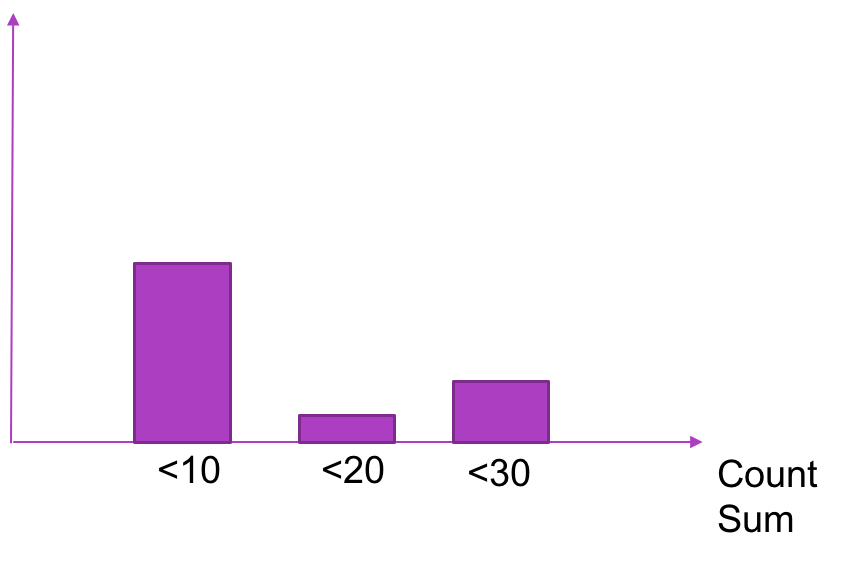
* Gauge 常规数值，例如 温度变化、内存使用变化。
* 可变大，可变小。
* 重启进程后，会被重置

例如： memory\_usage\_bytes{host="master-01"} 100 < 抓取值  
memory\_usage\_bytes{host="master-01"} 30  
memory\_usage\_bytes{host="master-01"} 50  
memory\_usage\_bytes{host="master-01"} 80 < 抓取值

Histogram

* Histogram 可以理解为柱状图的意思，常用于跟踪事件发生的规模，例如：请求耗时、响应大小。它特别之处是可以对记录的内容进行分组，提供 count 和 sum 全部值的功能。

例如：{小于10=5次，小于20=1次，小于30=2次}，count=7次，sum=7次的求和值



Summary

* Summary和Histogram十分相似，常用于跟踪事件发生的规模，例如：请求耗时、响应大小。同样提供 count 和 sum 全部值的功能。
* 例如：count=7次，sum=7次的值求值
* 它提供一个quantiles的功能，可以按%比划分跟踪的结果。例如：quantile取值0.95，表示取采样值里面的95%数据。

下一章说说Prometheus安装过程。

分类: [Prometheus](http://www.cnblogs.com/vovlie/category/909459.html)

标签: [Prometheus counter gauge histogram summary](http://www.cnblogs.com/vovlie/tag/Prometheus%20counter%20gauge%20histogram%20summary/)

# 2@使用Prometheus监控服务器 - 推酷

http://www.tuicool.com/articles/7neumi

时间 2017-03-17 05:37:27  [青蛙小白](http://www.tuicool.com/sites/ZjIBZfV)

原文  [http://blog.frognew.com/2017/02/use-prometheus-on-centos7.html](http://blog.frognew.com/2017/02/use-prometheus-on-centos7.html?utm_source=tuicool&utm_medium=referral)

主题 [Grafana](http://www.tuicool.com/topics/11040110)

Prometheus是一套开源监控系统，使用Go语言开发，是Google BorgMon监控系统的类似实现。

Prometheus的基本原理是通过HTTP协议周期性抓取被监控组件的状态，任意组件只要提供对应的HTTP接口就可以接入监控，是比较适合Docker，Kubernetes等环境的监控系统之一。输出监控信息的HTTP接口被称作exporter。

安装Prometheus

下载二进制包：

wget https://github.com/prometheus/prometheus/releases/download/v1.5.2/prometheus-1.5.2.linux-amd64.tar.gz

tar -zxvf prometheus-1.5.2.linux-amd64.tar.gz

mv prometheus-1.5.2.linux-amd64 /usr/local/prometheus

cd /usr/local/prometheus

./prometheus -version

prometheus, version 1.5.2 (branch: master, revision: bd1182d29f462c39544f94cc822830e1c64cf55b)

build user: root@a8af9200f95d

build date: 20170210-14:41:22

go version: go1.7.5

创建prometheus用户：

groupadd prometheus

useradd -g prometheus -d /var/lib/prometheus -s /sbin/nologin prometheus

/usr/lib/systemd/system/prometheus.service

**[Unit]**

Description=prometheus

After=network.target

**[Service]**

Type=simple

User=prometheus

ExecStart=/usr/local/prometheus/prometheus -config.file=/usr/local/prometheus/prometheus.yml -storage.local.path=/var/lib/prometheus

Restart=on-failure

**[Install]**

WantedBy=multi-user.target

启动：

**systemctl** enable prometheus

systemctl start prometheus

安装node\_exporter

为监控服务器CPU,内存,磁盘,I/O等信息，需要安装node\_exporter。

下载node\_exporter:

wget https://github.com/prometheus/node\_exporter/releases/download/v0.13.0/node\_exporter-0.13.0.linux-amd64.tar.gz

tar -zxvf node\_exporter-0.13.0.linux-amd64.tar.gz

mv node\_exporter-0.13.0.linux-amd64 /usr/local/prometheus/node\_exporter

/usr/lib/systemd/system/node\_exporter.service

**[Unit]**

Description=node\_exporter

After=network.target

**[Service]**

Type=simple

User=prometheus

ExecStart=/usr/local/prometheus/node\_exporter/node\_exporter

Restart=on-failure

**[Install]**

WantedBy=multi-user.target

**systemctl** enable node\_exporter

systemctl start node\_exporter

安装Grafana

/etc/yum.repos.d/grafana.repo

**[grafana]**

name=grafana

baseurl=https://packagecloud.io/grafana/stable/el/6/$basearch

repo\_gpgcheck=1

enabled=1

gpgcheck=1

gpgkey=https://packagecloud.io/gpg.key https://grafanarel.s3.amazonaws.com/RPM-GPG-KEY-grafana

sslverify=1

sslcacert=/etc/pki/tls/certs/ca-bundle.crt

yum **install** grafana

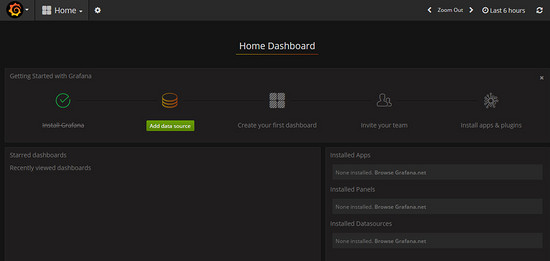
systemctl daemon-reload

systemctl **enable** grafana-**server**

systemctl **start** grafana-**server**

systemctl **status** grafana-**server**

浏览器中打开http://192.168.61.10:3000/, 默认登录用户名和密码：admin/admin



接下来在Grafana中添加Prometheus数据源：

* Name填上 **Prometheus Demo**
* Type选 **Prometheus**
* Url填上 **http://localhost:9090/**
* Access选 **direct**

参考

* [***Installing on RPM-based Linux (CentOS, Fedora, OpenSuse, RedHat)***](http://docs.grafana.org/installation/rpm/)

# Prometheus同类软件

[**服务监控系统 Prometheus**](https://www.oschina.net/p/prometheus)

https://www.oschina.net/p/prometheus

Prometheus 是一个开源的服务监控系统和时间序列数据库。 特性： 高维度数据模型 自定义查询语言 可视化数据展示 高效的存储策略 易于运维 提供各种客户端开发库 警告和报警 数据导出

Prometheus 同类软件

[监控系统 Nagios](https://www.oschina.net/p/nagios)

[分布式系统监视 zabbix](https://www.oschina.net/p/zabbix)

[网络流量监测图形分析工具 Cacti](https://www.oschina.net/p/cacti)

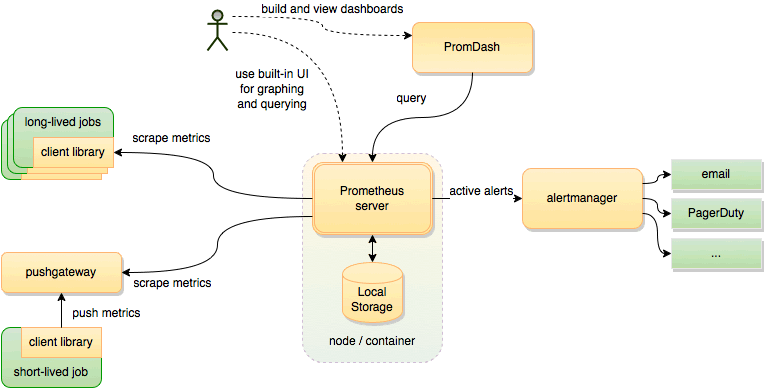
[开源视频监控系统 ZoneMinder](https://www.oschina.net/p/zoneminder)

[系统监控工具 MRTG](https://www.oschina.net/p/mrtg)

[互联网企业级监控系统 OpenFalcon](https://www.oschina.net/p/open-falcon)

[系统信息采集和监控工具 Tsar](https://www.oschina.net/p/tsar)

[Java 应用监控系统 HawtIO](https://www.oschina.net/p/hawtio)

[](https://www.oschina.net/p/prometheus)

最近更新： [Prometheus v1.0.1 发布，服务监控系统](https://www.oschina.net/news/76077/prometheus-1-0-1) 发布于 8个月前

# 使用Prometheus监控服务器性能 - 推酷

http://www.tuicool.com/articles/n2u2eiv

时间 2017-03-12 08:00:00  [CJ Ting's Blog](http://www.tuicool.com/sites/2UBVbib)

原文  [http://cjting.me/misc/2017-03-12-使用Prometheus监控服务器性能.html](http://cjting.me/misc/2017-03-12-%E4%BD%BF%E7%94%A8Prometheus%E7%9B%91%E6%8E%A7%E6%9C%8D%E5%8A%A1%E5%99%A8%E6%80%A7%E8%83%BD.html?utm_source=tuicool&utm_medium=referral)

主题 [Grafana](http://www.tuicool.com/topics/11040110)

最近一直在思考如何对线上服务做深度监控。基础的服务可用性监控很简单，定期Ping即可。但是怎样才能监控服务器的一些更加关键的数据呢？比如，每一个API Point的请求次数（QPS），最大响应时间，平均响应时间等。最终我希望实现的效果是有一个Dashboard，我可以清楚地看到各种参数曲线，对服务器的运行情况了然于胸。

绘制Dashboard不难，目前提供数据可视化的工具很多，随便选一个都能满足需要。关键问题是，怎样将整个流程打通？

* 服务器该以怎样的形式暴露出数据？
* 数据怎样被收集和存储起来？
* 存储起来的数据怎样提供给数据可视化工具？
* 怎样做到足够灵活，可以可视化自己感兴趣的任意数据？

Prometheus

像QPS和响应时间这些数据，外部工具是没办法直接拿到的，必须要服务器以某种方式将数据暴露出来。最常见的做法是写日志。比如Nginx，每一条请求对应一个日志，日志中有响应时间这个字段。通过对日志分析，我们就可以得到QPS，最大响应时间，平均响应时间等，再通过可视化工具即可绘制我们想要的Dashboard。

日志这个方法固然是可行的，但是还有更好的方法。这个方法就是 **时序数据库（Time Series Database）** 。时序数据库简单来说就是存储随时间变化的数据的数据库。什么是随时间变化的数据呢？举个简单的例子，比如，CPU使用率，典型的随时间变化的量，这一秒是50%，下一秒也许就是80%了。或者是温度，今天是20度，明天可能就是18度了。

[***Prometheus***](https://prometheus.io/) 就是一个用Go编写的时序数据库，官网对其的优点介绍的很清楚，这里就不再赘述了。总之，使用简单，功能强大。

安装

安装直接去官网下载对应的 [***安装包***](https://prometheus.io/download/) 即可。当然，如果你是Mac用户的话，brew永远不会让你失望 **brew install prometheus** 。

格式

Prometheus获取数据的策略是 **Pull** 而不是 **Push** ，也就是说，它会自己去抓取，而不用你来推送。抓取使用的是HTTP协议，在配置文件中指定目标程序的端口，路径及间隔时间即可。这也就意味着任何程序想要使用Prometheus存储数据都很简单，定义一个HTTP接口即可。

Prometheus的数据格式是简单的文本格式，可以直接阅读。其中， **#** 号开头的是注释，除此之外，每一行一个数据项，数据名在前，值在后。 **{}** 中是标签，一条数据可以有多个标签。

# HELP go\_gc\_duration\_seconds A summary of the GC invocation durations.

# TYPE go\_gc\_duration\_seconds summary

http\_request\_count{endpoint="/a"} 10

http\_request\_count{endpoint="/b"} 200

http\_request\_count(endpoint="/c") 3

配置

Prometheus使用YAML进行配置。 **global** 配置一些全局信息， **scrape\_configs** 配置具体想要抓取的目标。这段配置的含义是：启动一个叫做 **go-test** 的任务，每隔五秒钟，访问 **localhost:8888/metrics** 获取数据。

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 it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'go-test'

metrics\_path: "/metrics"

# Override the global default and scrape targets from this job every 5 seconds.

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:8888']

测试程序

我们来写一个程序测试一下Prometheus的功能。虽然可以手动返回Prometheus需要的数据，但是使用开发好的客户端会更加方便。

这里我们使用Go语言，编写一个简单的服务器和客户端。客户端会以一个稳定的速度请求服务器的 **/test** 路径，但是每两分钟会加大流量，持续30秒再回到之前的水平。服务器95%的情况下会花费50ms进行响应，还有5%的情况下会花费100ms。

这里我们定义了两个指标， **httpRequestCount** 记录HTTP的请求数， **httpRequestDuration** 记录响应时间，他们都有一个 **endpoint** 标签用于记录请求路径。这两个指标分别是 **Counter** 类型和 **Summary** 类型，Prometheus定义了四种指标类型，基本涵盖了各种用例场景，具体可以去看 [***相关文档***](https://prometheus.io/docs/concepts/metric_types/) 。简单来说，Counter类型的数据表示一个只会向上增加的数据，比如请求数。而Summary类型的数据表示一个按区间分布的数据，比如响应时间或者请求体大小。

/\*

\* @Author: CJ Ting

\* @Date: 2017-03-12 17:27:23

\* @Last Modified by: CJ Ting

\* @Last Modified time: 2017-03-12 23:49:55

\*/

**package** main

**import** (

"log"

"math/rand"

"net/http"

"time"

"github.com/prometheus/client\_golang/prometheus"

"github.com/prometheus/client\_golang/prometheus/promhttp"

)

**var** httpRequestCount = prometheus.NewCounterVec(

prometheus.CounterOpts{

Name: "http\_request\_count",

Help: "http request count",

},

[]**string**{"endpoint"},

)

**var** httpRequestDuration = prometheus.NewSummaryVec(

prometheus.SummaryOpts{

Name: "http\_request\_duration",

Help: "http request duration",

},

[]**string**{"endpoint"},

)

**func** **init**() {

prometheus.MustRegister(httpRequestCount)

prometheus.MustRegister(httpRequestDuration)

}

**func** **main**() {

http.Handle("/metrics", promhttp.Handler())

http.HandleFunc("/test", handler)

**go** **func**() {

http.ListenAndServe(":8888", nil)

}()

startClient()

doneChan := make(**chan** **struct**{})

<-doneChan

}

**func** **handler**(w http.ResponseWriter, r \*http.Request) {

start := time.Now()

path := r.URL.Path

httpRequestCount.WithLabelValues(path).Inc()

n := rand.Intn(100)

**if** n >= 95 {

time.Sleep(100 \* time.Millisecond)

} **else** {

time.Sleep(50 \* time.Millisecond)

}

elapsed := (**float64**)(time.Since(start) / time.Millisecond)

httpRequestDuration.WithLabelValues(path).Observe(elapsed)

}

**func** **startClient**() {

sleepTime := 1000

**go** **func**() {

ticker := time.NewTicker(2 \* time.Minute)

**for** {

<-ticker.C

sleepTime = 200

<-time.After(30 \* time.Second)

sleepTime = 1000

}

}()

**for** i := 0; i < 100; i++ {

**go** **func**() {

**for** {

sendRequest()

time.Sleep((time.Duration)(sleepTime) \* time.Millisecond)

}

}()

}

}

**func** **sendRequest**() {

resp, err := http.Get("http://localhost:8888/test")

**if** err != nil {

log.Println(err)

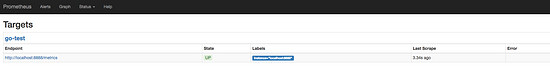
**return**

}

resp.Body.Close()

}

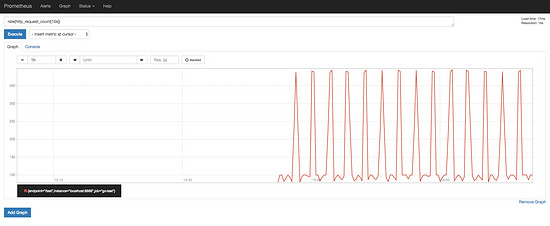
启动Prometheus **prometheus -config.file config.yml** 以后，再启动我们的测试程序 **go run test.go** 。打开Prometheus控制台 **localhost:9090/targets** 就可以看到Prometheus正在抓取数据，一切正常。



控制台

Prometheus的一个强大之处在于可以使用各种函数和操作符来查询数据。在上面的测试程序中，每个数据都带有 **endpoint** 这个标签，表示请求的路径。打开Prometheus的控制台 **http://localhost:9090/graph** ，点击 **console** 标签页，输入 **http\_request\_count{endpoint="/a"}** 就可以查询路径为 **/a** 的API Point到目前为止的总请求数。如果想看QPS的话，可以使用自带的函数 **rate** ， **rate(http\_request\_count[10s])** 表示以10s作为时间单元来统计QPS。

Prometheus的控制台自带一个简单的绘图系统，点击 **graph** 标签页，输入表达式就可以看到图表。例如，输入 **rate(http\_request\_count{endpoint="/test"}[10s])** 就可以看到我们测试程序中 **/test** 路径的QPS，从图中可以明显发现，每隔一段时间就会有一个波峰流量。

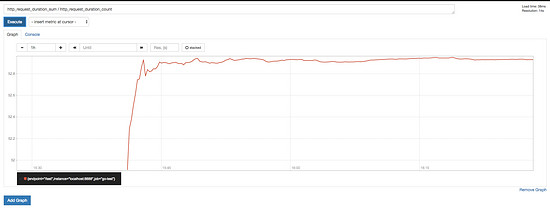


**httpRequestDuration** 是一个Summary类型的指标，比简单的Counter要复杂，会生成三个数据项。分别是 **http\_request\_duration\_sum** ，表示响应时间加在一起的总和， **http\_request\_duration\_count** ，表示响应时间的总个数，以及 **http\_request\_duration** ，表示响应时间的分布情况，这个数据项会使用 **quantile** 标签对响应时间进行分组。

如下图所示， **quantile=0.5** 值为50，表示50%的请求响应时间都在50ms以下。 **quantile=0.9** 的值为54，表示90%的请求响应时间都在54ms以下。但是， **quantile=0.99** 的值为103，表示99%的请求响应时间在103ms以下。这就说明了一个问题，那就是极个别的请求耗费了大量时间。



通过使用表达式 **http\_request\_duration\_sum / http\_request\_duration\_count** 我们可以得到平均响应时间，如下图。当然，这个图的作用不大（平均数往往反映不了什么问题），不像上图那样，我们无法看出有部分请求花费了大量时间。



以上只是对数据项的最简单利用，Prometheus自带了很多函数和操作符，可以方便地对数据进行处理，具体可以参考 [***官方文档***](https://prometheus.io/docs/querying/basics/) 。

Grafana

Prometheus自带的图表是非常基础的，只能用来临时查看一下数据。如果要构建强大的Dashboard，还是需要更加专业的工具才行。这个工具就是 [***Grafana***](http://grafana.org/) 。

安装

同样是去官网下载相应的 [***安装包***](http://grafana.org/download/) 。Mac用户可以再次感受到brew的优越性。 **brew install grafana** 。

启动

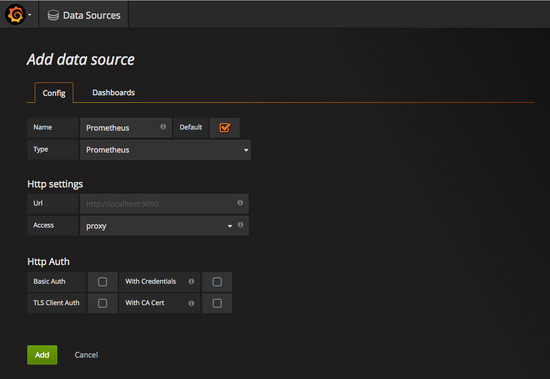
直接用默认配置就挺好的。在Mac上，启动指令如下。

$ grafana-server --config=/usr/local/etc/grafana/grafana.ini --homepath /usr/local/share/grafana cfg:default.paths.logs=/usr/local/var/log/grafana cfg:**default**.paths.data=/usr/local/var/lib/grafana cfg:default.paths.plugins=/usr/local/var/lib/grafana/plugins

Grafana默认监听在3000端口上，默认用户名和密码都是 **admin** 。

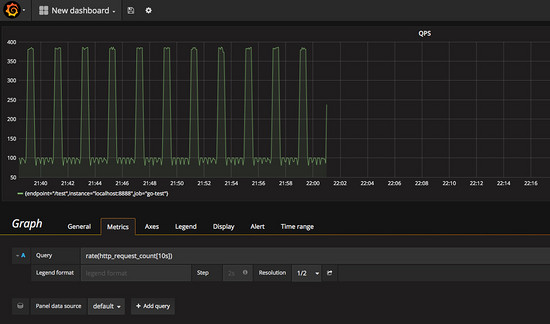
设置

输入用户名和密码以后，进入Grafana页面。第一件事是要设置数据源（Data Source），即Grafana从什么地方获取数据，选择Prometheus即可。



数据源设置好以后，接下来就是创建Dashboard了。Dashboard里面可以放置很多“组件”。比如，图表，状态值，表格，文字等等。这里我们选择 **Graph** 图表，Grafana会创建一个默认的空图表。

点击图表标题，选择 **Edit** 来编辑图表参数。最重要的参数就是 **Metrics** 标签里的 **Query**字段，这个字段定义了我们的图表到底要展示什么数据。输入 **rate(http\_request\_count{endpoint="/test"}[10s])** ，就可以看到 **/test** 路径的QPS曲线了。



同理，在Query中输入 **http\_request\_duration** 就可以得到响应时间曲线。通过使用Prometheus提供的操作符和函数，我们可以对数据进行我们想要的任意可视化，十分灵活。

在这两个工具的配合使用下，对服务器信息的监控变得非常简单。首先，服务器定义一个HTTP接口，暴露出想要监控的数据，然后使用Prometheus收集并存储这些数据，最后在Grafana中绘制这些数据。一个完整的监控方案就诞生了。

当然，在实际系统中，还缺少了一个环节，那就是报警。监控发现问题以后，需要马上报警通知相关的维护人员。这是另外一个话题了，以后再介绍。

# 2@Prometheus的架构及持久化 - davygeek - 博客园

http://www.cnblogs.com/davygeek/p/6668706.html

[Prometheus的架构及持久化](http://www.cnblogs.com/davygeek/p/6668706.html)

原文： https://my.oschina.net/go4it/blog/855598

**Prometheus是什么**

Prometheus是一个开源的系统监控和报警工具，特点是

* 多维数据模型（时序列数据由metric名和一组key/value组成）
* 在多维度上灵活的查询语言(PromQl)
* 不依赖分布式存储，单主节点工作.
* 通过基于HTTP的pull方式采集时序数据
* 可以通过push gateway进行时序列数据推送(pushing)
* 可以通过服务发现或者静态配置去获取要采集的目标服务器
* 多种可视化图表及仪表盘支持

**pull方式**

Prometheus采集数据是用的pull也就是拉模型,通过HTTP协议去采集指标，只要应用系统能够提供HTTP接口就可以接入监控系统，相比于私有协议或二进制协议来说开发、简单。

**push方式**

对于定时任务这种短周期的指标采集，如果采用pull模式，可能造成任务结束了，Prometheus还没有来得及采集，这个时候可以使用加一个中转层，客户端推数据到Push Gateway缓存一下，由Prometheus从push gateway pull指标过来。(需要额外搭建Push Gateway，同时需要新增job去从gateway采数据)

**组成及架构**

* Prometheus server 主要负责数据采集和存储，提供PromQL查询语言的支持
* 客户端sdk 官方提供的客户端类库有go、java、scala、python、ruby，其他还有很多第三方开发的类库，支持nodejs、php、erlang等
* Push Gateway 支持临时性Job主动推送指标的中间网关
* PromDash 使用rails开发的dashboard，用于可视化指标数据
* exporters 支持其他数据源的指标导入到Prometheus，支持数据库、硬件、消息中间件、存储系统、http服务器、jmx等
* alertmanager 实验性组件、用来进行报警
* prometheus\_cli 命令行工具
* 其他辅助性工具

架构图如下：

**默认配置**

docker exec -it a9bd827a1d18 less /etc/prometheus/prometheus.yml

得到

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Attach these labels to any time series or alerts when communicating with

# external systems (federation, remote storage, Alertmanager).

external\_labels:

monitor: 'codelab-monitor'

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first.rules"

# - "second.rules"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ['localhost:9090']

* scrape\_interval 这里是指每隔15秒钟去抓取数据(这里)
* evaluation\_interval 指的是计算rule的间隔

**Push Gateway**

pushgateway有单独的镜像

docker pull prom/pushgateway

对于喜欢用push模式的应用来说，可以专门搭建一个push gateway，来适配一下。

**storage**

prometheus使用了G家的LevelDB来做索引(PromSQL重度依赖LevelDB)，对于大量的采样数据有自己的存储层，Prometheus为每个时序数据创建一个本地文件，以1024byte大小的chunk来组织。

**磁盘文件**

Prometheus在storage.local.path指定的路径存储文件，默认为./data。关于chunk编码有三种

* type 0

第一代的编码格式，simple delta encoding

* type 1

目前默认的编码格式，double-delta encoding

* type 2

variable bit-width encoding，facebook的时间序列数据库Beringei采用的编码方式

**内存使用**

prometheus在内存里保存了最近使用的chunks，具体chunks的最大个数可以通过storage.local.memory-chunks来设定，默认值为1048576，即1048576个chunk，大小为1G。 除了采用的数据，prometheus还需要对数据进行各种运算，因此整体内存开销肯定会比配置的local.memory-chunks大小要来的大，因此官方建议要预留3倍的local.memory-chunks的内存大小。

As a rule of thumb, you should have at least three times more RAM available than needed by the memory chunks alone

可以通过server的metrics去查看prometheus\_local\_storage\_memory\_chunks以及process\_resident\_memory\_byte两个指标值。

* prometheus\_local\_storage\_memory\_chunks

The current number of chunks in memory, excluding cloned chunks 目前内存中暴露的chunks的个数

* process\_resident\_memory\_byte

Resident memory size in bytes 驻存在内存的数据大小

* prometheus\_local\_storage\_persistence\_urgency\_score 介于0-1之间，当该值小于等于0.7时，prometheus离开rushed模式。 当大于0.8的时候，进入rushed模式
* prometheus\_local\_storage\_rushed\_mode 1表示进入了rushed mode，0表示没有。进入了rushed模式的话，prometheus会利用storage.local.series-sync-strategy以及storage.local.checkpoint-interval的配置加速chunks的持久化。

**storage参数**

docker run -p 9090:9090 \

-v /tmp/prometheus-data:/prometheus-data \

prom/prometheus \

-storage.local.retention 168h0m0s \

-storage.local.max-chunks-to-persist 3024288 \

-storage.local.memory-chunks=50502740 \

-storage.local.num-fingerprint-mutexes=300960

**storage.local.memory-chunks**

设定prometheus内存中保留的chunks的最大个数，默认为1048576，即为1G大小

**storage.local.retention**

用来配置采用数据存储的时间，168h0m0s即为24\*7小时，即1周

**storage.local.series-file-shrink-ratio**

用来控制序列文件rewrite的时机，默认是在10%的chunks被移除的时候进行rewrite，如果磁盘空间够大，不想频繁rewrite，可以提升该值，比如0.3，即30%的chunks被移除的时候才触发rewrite。

**storage.local.max-chunks-to-persist**

该参数控制等待写入磁盘的chunks的最大个数，如果超过这个数，Prometheus会限制采样的速率，直到这个数降到指定阈值的95%。建议这个值设定为storage.local.memory-chunks的50%。Prometheus会尽力加速存储速度，以避免限流这种情况的发送。

**storage.local.num-fingerprint-mutexes**

当prometheus server端在进行checkpoint操作或者处理开销较大的查询的时候，采集指标的操作会有短暂的停顿，这是因为prometheus给时间序列分配的mutexes可能不够用，可以通过这个指标来增大预分配的mutexes，有时候可以设置到上万个。

**storage.local.series-sync-strategy**

控制写入数据之后，何时同步到磁盘，有'never', 'always', 'adaptive'. 同步操作可以降低因为操作系统崩溃带来数据丢失，但是会降低写入数据的性能。 默认为adaptive的策略，即不会写完数据就立刻同步磁盘，会利用操作系统的page cache来批量同步。

**storage.local.checkpoint-interval**

进行checkpoint的时间间隔，即对尚未写入到磁盘的内存chunks执行checkpoint操作。

**doc**

* [prometheus-configuration](https://prometheus.io/docs/operating/configuration/)
* [prometheus-storage](https://prometheus.io/docs/operating/storage/)
* [promdash](https://prometheus.io/docs/visualization/promdash/)
* [config.go](https://github.com/prometheus/prometheus/blob/master/cmd/prometheus/config.go)

# springboot输出metrics到prometheus - go4it

https://my.oschina.net/go4it/blog/858867

**搭建push gateway**

version: '2'

services:

prometheus:

build: .

ports:

- 9090:9090

volumes:

- /tmp/prometheus-data:/prometheus-data

links:

- pushgateway

pushgateway:

image: prom/pushgateway

ports:

- 9999:9091

配置从pushgateway采取数据

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Attach these labels to any time series or alerts when communicating with

# external systems (federation, remote storage, Alertmanager).

external\_labels:

monitor: 'codelab-monitor'

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first.rules"

# - "second.rules"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'push-metrics'

static\_configs:

- targets: ['192.168.99.100:9999']

**maven**

<!-- The client -->

<dependency>

<groupId>io.prometheus</groupId>

<artifactId>simpleclient</artifactId>

<version>0.0.21</version>

</dependency>

<!-- Hotspot JVM metrics -->

<dependency>

<groupId>io.prometheus</groupId>

<artifactId>simpleclient\_hotspot</artifactId>

<version>0.0.21</version>

</dependency>

<!-- Exposition servlet -->

<dependency>

<groupId>io.prometheus</groupId>

<artifactId>simpleclient\_servlet</artifactId>

<version>0.0.21</version>

</dependency>

<!-- Pushgateway exposition -->

<dependency>

<groupId>io.prometheus</groupId>

<artifactId>simpleclient\_pushgateway</artifactId>

<version>0.0.21</version>

</dependency>

<dependency>

<groupId>io.prometheus</groupId>

<artifactId>simpleclient\_dropwizard</artifactId>

<version>0.0.21</version>

</dependency>

**java config**

@Component

public class PrometheusConfig {

@Autowired

MetricRegistry dropwizardRegistry;

@Value("${spring.application.name}")

String applicationName;

@Value("${prometheus.pushgateway.host}")

String pushHost;

@Value("${prometheus.pushgateway.intervalInMillis:10000}")

long intervalInMillis;

private final CollectorRegistry prometheusRegistry = new CollectorRegistry();

@PostConstruct

public void initialize() {

DropwizardExports prometheus = new DropwizardExports(dropwizardRegistry);

prometheus.register(prometheusRegistry);

PushGateway prometheusPush = new PushGateway(pushHost);

Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {

try {

prometheus.collect();

prometheusPush.push(prometheusRegistry, applicationName);

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}, 5000, intervalInMillis, TimeUnit.MILLISECONDS);

}

}

# 部署 mesos-exporter 和 prometheus 监控 mesos task - hahp - 博客园

http://www.cnblogs.com/hahp/p/5614285.html

前几天我在mesos平台上基于 cadvisor部署了 influxdb 和 grafana，用于监控 mesos 以及 docker app 运行信息，发现这套监控系统不太适合 mesos + docker 的架构，原因是：

**1）mesos task id 和 docker container name 不一致**

cadvisor 的设计基于 docker host，没有考虑到mesos 数据中心；

cadvisor 用 docker name（docker ps能看到）来标记抓取的数据，而 mesos 用 task id（在mesos ui 或者metrics里能看到） 来标记正在运行的任务。mesos task 的类型可以是 docker 容器，也可以是非容器。mesos task id 与docker container name 的命名也是完全不一样的。

上述问题导致 cadvisor 抓取到数据后，用户难以识别属于哪个 mesos task

**2）cadvisor 和 grafana 不支持报警**

经过查询资料，发现 mesos-exporter + prometheus + alert-manager 是个很好的组合，可以解决上述问题：

mesos-exporter 是 mesosphere 开发的工具，用于导出 mesos 集群包括 task 的监控数据并传递给prometheus；prometheus是个集 db、graph、statistic 于一体的监控工具；alert-manager 是 prometheus 的报警工具

**搭建方法：**

**1. build mesos-exporter**

[?](http://www.cnblogs.com/hahp/p/5614285.html)

|  |  |
| --- | --- |
| 1  2  3 | git clone https://github.com/mesosphere/mesos\_exporter.git  cd mesos\_exporter  docker build -f Dockerfile -t mesosphere/mesos-exporter . |

**2. docker pull prometheus, alert-manager**

**3. 部署 mesos-exporter, alert-manager, prometheus**

mesos-exporter：

[?](http://www.cnblogs.com/hahp/p/5614285.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | {    "id": "mesos-exporter-slave",    "instances": 6,    "cpus": 0.2,    "mem": 128,    "args": [        "-slave=http://127.0.0.1:5051",        "-timeout=5s"    ],    "constraints": [        ["hostname","UNIQUE"],        ["hostname", "LIKE", "slave[1-6]"]    ],    "container": {      "type": "DOCKER",      "docker": {        "image": "172.31.17.36:5000/mesos-exporter:latest",        "network": "HOST"      },      "volumes": [        {          "containerPath": "/etc/localtime",          "hostPath": "/etc/localtime",          "mode": "RO"        }      ]    }  } |

请打开slave 防火墙的9110/tcp 端口

alert-manager:

[?](http://www.cnblogs.com/hahp/p/5614285.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | {    "id": "alertmanager",    "instances": 1,    "cpus": 0.5,    "mem": 128,    "constraints": [        ["hostname","UNIQUE"],        ["hostname", "LIKE", "slave[1-6]"]    ],    "labels": {      "HAPROXY\_GROUP":"external",      "HAPROXY\_0\_VHOST":"alertmanager.gkkxd.com"    },    "container": {      "type": "DOCKER",      "docker": {        "image": "172.31.17.36:5000/alertmanager:latest",        "network": "BRIDGE",        "portMappings": [          { "containerPort": 9093, "hostPort": 0, "servicePort": 0, "protocol": "tcp" }        ]      },      "volumes": [        {          "containerPath": "/etc/localtime",          "hostPath": "/etc/localtime",          "mode": "RO"        },        {          "containerPath": "/etc/alertmanager/config.yml",          "hostPath": "/var/nfsshare/alertmanager/config.yml",          "mode": "RO"        },        {          "containerPath": "/alertmanager",          "hostPath": "/var/nfsshare/alertmanager/data",          "mode": "RW"        }      ]    }  } |

prometheus：

[?](http://www.cnblogs.com/hahp/p/5614285.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | {    "id": "prometheus",    "instances": 1,    "cpus": 0.5,    "mem": 128,    "args": [        "-config.file=/etc/prometheus/prometheus.yml",        "-storage.local.path=/prometheus",        "-web.console.libraries=/etc/prometheus/console\_libraries",        "-web.console.templates=/etc/prometheus/consoles",        "-alertmanager.url=http://alertmanager.gkkxd.com"    ],    "constraints": [        ["hostname","UNIQUE"],        ["hostname", "LIKE", "slave[1-6]"]    ],    "labels": {      "HAPROXY\_GROUP":"external",      "HAPROXY\_0\_VHOST":"prometheus.gkkxd.com"    },    "container": {      "type": "DOCKER",      "docker": {        "image": "172.31.17.36:5000/prometheus:latest",        "network": "BRIDGE",        "portMappings": [          { "containerPort": 9090, "hostPort": 0, "servicePort": 0, "protocol": "tcp" }        ]      },      "volumes": [        {          "containerPath": "/etc/localtime",          "hostPath": "/etc/localtime",          "mode": "RO"        },        {          "containerPath": "/etc/prometheus",          "hostPath": "/var/nfsshare/prometheus/conf",          "mode": "RO"        },        {          "containerPath": "/prometheus",          "hostPath": "/var/nfsshare/prometheus/data",          "mode": "RW"        }      ]    }  } |

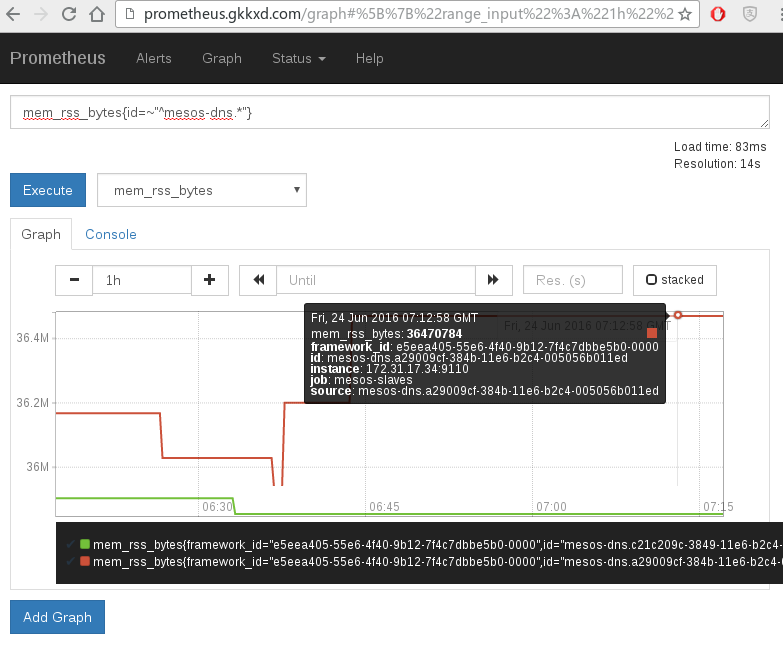
**4. prometheus 配置**

prometheus.yml

[?](http://www.cnblogs.com/hahp/p/5614285.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | # my global config  global:    scrape\_interval:     15s # By default, scrape targets every 15 seconds.    evaluation\_interval: 15s # By default, scrape targets every 15 seconds.    # scrape\_timeout is set to the global default (10s).      # Attach these labels to any time series or alerts when communicating with    # external systems (federation, remote storage, Alertmanager).    external\_labels:        monitor: 'codelab-monitor'    # Load and evaluate rules in this file every 'evaluation\_interval' seconds.  rule\_files:    # - "first.rules"    # - "second.rules"    scrape\_configs:    - job\_name: 'mesos-slaves'      scrape\_interval: 5s      metrics\_path: '/metrics'      scheme: 'http'      target\_groups:        - targets: ['172.31.17.31:9110', '172.31.17.32:9110', '172.31.17.33:9110', '172.31.17.34:9110', '172.31.17.35:9110', '172.31.17.36:9110']        - labels:            group: 'office' |

待补充 ...

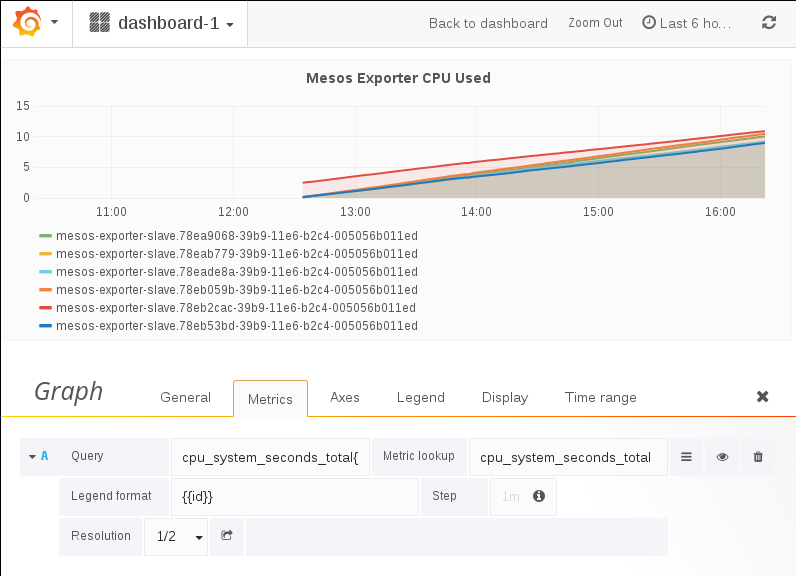


**5. 报警设置**

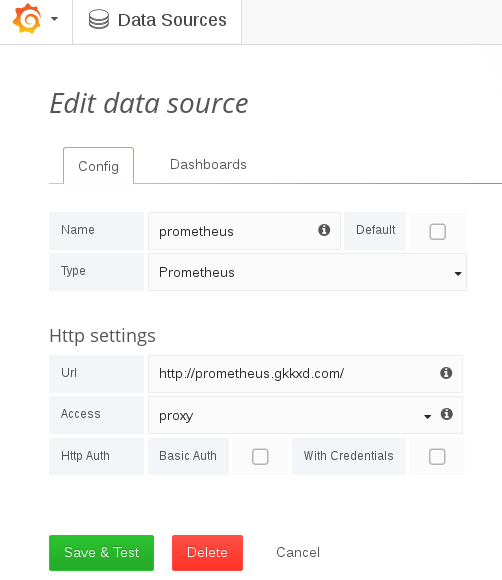
待补充 ...

**6. 与 grafana 集成**

prometheus的 graph 功能不太完善，可以与 grafana 集成，让 grafana 承担 graph 功能。



data source 设置：



**7. 附：mesos metrics 和 statics 地址**

http://master1:5050/metrics/snapshot

http://slave4:5051/metrics/snapshot

http://master1:5050/master/state.json

http://slave4:5051/monitor/statistics.json

用户可以基于上述页面的数据，编写自己的监控程序。

# 2@Prometheus及替代方案对比 - 推酷

http://www.tuicool.com/articles/2eAvIza

时间 2017-03-08 15:29:14  [addops](http://www.tuicool.com/sites/3y6R7nY)

原文  [https://addops.cn/post/comparison-to-alternatives.html](https://addops.cn/post/comparison-to-alternatives.html?utm_source=tuicool&utm_medium=referral)

主题 [系统监控](http://www.tuicool.com/topics/11000134)[数据库](http://www.tuicool.com/topics/11000064)

本文译自 [***COMPARISON TO ALTERNATIVES***](https://prometheus.io/docs/introduction/comparison/) ,翻译的比较粗浅,希望能对了解 prometheus 有所帮助.

Prometheus vs. Graphite

适用范围

Graphite 关注点是作为一个被动机制的时间序列数据库,有自己的查询语言和绘图方式.其他特性需要通过外部组件来实现.

prometheus 是一套完整的监控和趋势系统,在时序数据基础上内建主动抓取,搜索,绘图和报警系统.有丰富的官方和第三方贡献的监控收集工具( [***详见exporters***](https://prometheus.io/docs/instrumenting/exporters/) ).

数据模型

Graphite 和 Prometheus 一样存储命名后的时间序列数据.当然 prometheus 的元数据模型更加丰富.Graphite 以"."作为分割符命名监控项,而promethues 在命名监控项时通过添加Key/value标签,使得在查询时更容易进行过滤,分组和匹配.

而且当 Graphite 和 StatsD 结合使用时.一般只会存储聚合后的所有监控实例的监控数据,而不是以实例为纬度,很难深入挖掘问题.

在这里有一个例子.我们要存储 **/tracks** 接口 **POST** 请求的 **500** 返回值的数量.

在Graphite 或 StatsD 中如下:

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

在 prometheus 中相同的数据如下存储(以收集三台接口服务器为例).

**api\_server\_http\_requests\_total**{method="POST",handler="/tracks",status="500",**instance**="<sample1>"} -> 34

**api\_server\_http\_requests\_total**{method="POST",handler="/tracks",status="500",**instance**="<sample2>"} -> 28

**api\_server\_http\_requests\_total**{method="POST",handler="/tracks",status="500",**instance**="<sample3>"} -> 31

存储方式

Graphite 以 [***Whisper***](http://graphite.readthedocs.org/en/latest/whisper.html) 格式在本地磁盘存储时序数据,类似于 RRD 的形式(但 Whisper 可随机更新任意过去时间的数据,但周期固定,).每一个时序数据存储在不同的文件中.新监控数据在一定周期后会覆盖旧的数据.

Prometheus 也会为每个时序数据创建一个本地文件,但是允许按照抓取的任意周期或规则任务触发的时间存储数据.当新数据被添加进来后,旧数据依然能按需求保持任意时间.prometheus 也十分适合短时且经常变化的时序数据.

总结

prometheus 提供更丰富的数据类型和查询语言,并且十分易于部署和融入现有环境.如果想使用一个有集群解决方案并且希望存储长期数据,Graphite 可能是更好的选择.

Prometheus vs. InfluxDB

InfluxDB 是一个开源时序数据库,并且提供商业化(需付费)的集群和扩容解决方案.Prometheus 和 InfluxDB 之间有着显著的差异,同时两者也面向不同的应用场景.

适用范围

在这里需要将 [***Kapacitor***](https://github.com/influxdata/kapacitor) 和 InfluxDB 作为一个组合一起同进行对比,因为同 Prometheus 和 Alertmanager一样,这两组工具实现的功能是为了处理相同的问题.

Kapacitor 是一个混合了 prometheus 的数据处理,存储规则,报警规则,以及Alertmanager通知功能的工具.Prometheus 提供了与之相比更强大查询语言用于绘图和报警.并且Prometheus Alertmanager进一步提供了分组,去重和除噪声的功能.

数据模型和存储

和 Prometheus 相同,InfluxDB 的数据模型包含Key/Value对标签.此外 InfluxDB 还包含一个二层标签 **field** .InfluxDB支持纳秒级的时间纬度,以及float64, int64, bool, 和字符串数据类型.Promethues 作为对比只支持 float64 和有限的字符串支持,并且时间维度为纳秒级.

InfluxDB 使用自己实现的 TSM Tree 的算法， LSM Tree(源于 bigtable [***扩展阅读***](http://www.open-open.com/lib/view/open1424916275249.html) )，针对 InfluxDB 的场景做了特殊优化。比Prometheus向每个时序数据文件中增加数据的方式要更适合于事件日志.

[***Logs and Metrics and Graphs, Oh My!***](https://blog.raintank.io/logs-and-metrics-and-graphs-oh-my/) 描述了事件日志和监控指标记录的区别.

架构

prometheus 的服务器都独立运行,核心功能依赖于服务器的本地存储.开源版本的InfluxDB相同.

商业版本的 InfluxDB 提供通过设计一个分布式存储方案,使存储和查询能在不同节点上同时进行处理.

这就表明商业版本的 InfluxDB 更容易横向扩展,但也意味着你必须从最开始就要管理一个复杂的分布式存储系统.相对来说 Prometheus 的运行更佳简单,但是你也需要提前考虑好各个单独的 Prometheus 服务器的可扩展性.当然独立的服务器也能给你更好的可靠性和故障隔离.

Kapacitor 当前没有内建的分布式或冗余选项.作为对比Prometheus提供了冗余选项,配合运行prometheus冗余副本以并能使用Alertmanager的高可用模式.另外,Kapacitor可以同 Prometheus 一样通过用户手动分片进行扩容.

总结

两个系统之间有非常多的相似之处.都通过标签来支持多纬度监控指标.两者都使用基本相同的数据压缩算法.两者包括彼此之间都有广泛的集成.两者都有钩子以便将来进行扩展,例如在统计工具中进行数据分析或执行自动化操作.

InfluxDB 的优势:

* 进行事物日志监控.
* 商业化的InfluxDB集群方案,更加适合长期数据的存储.
* 副本间的数据一致性.

Prometheus 的优势:

* 主要用于监控指标收集.
* 更强大的搜索语言,报警,以及通知功能.
* 对于绘图和报警有更高的可用性和运行时间

Prometheus vs. OpenTSDB

OpenTSDB 是一个基于 Hadoop 和 HBase 的分布式时序数据库.

适用场景

适用场景的对比与 Graphite 类似.

数据模型

OpenTSDB 的数据模型与 Prometheus 几乎相同:时序数据是由一组任意的键值对来确定的.所有指标的数据都存储在一起,同时限制指标的数量.虽然有细微的差别,例如 Prometheus 允许在标签中使用任意字符,但 OpenTSDB 有部分限制.OpenTSDB同时也缺乏完善的查询语言,只能通过 API 进行简单的聚合和数学计算.

存储

OpenTSDB 的存储在 Hadoop 和 HBase 之上实现.这说明我们可以非常容易的对 OpenTSDB 进行平行扩容.但是必须接受从最开始的时候就要维护十分复杂的 Hadoop/HBase 集群.

Prometheus 初期运行简单.但是当单个节点容量达到一定限度时要提前考虑分片的问题.

总结

Prometheus 提供更丰富的查询语言,可以处理更大基数的监控指标数量并且能形成完整监控系统的一部分.如果你已经运行了 Hadoop 集群并存储长期数据,那么 OpenTSDB 是一个比较好的选择.

Prometheus vs. Nagios

Nagios 是起源于90年代名为"NetSaint"的开源监控系统

适用场景

Nagios主要是针对脚本的退出代码进行报警.能针对个别报警的停止报警功能,然而没有分组,路由和去重的功能.

Nagios 有十分多的插件.比如,perfData 插件中抽取出的几 kb 的数据可以发送到 Graphite 这样的时序数据库中,或者使用 NRPE 在远程逐级上运行检测命令.

数据模型

nagios本身没有存储,仅基于当前的状态.也有相关的插件可以存储数据,例如一些可视化插件( [***PNP4Nagios***](https://docs.pnp4nagios.org/) ).

架构

Nagios 服务器都是单点运行.所有检测配置都是基于文件的.

总结

Nagios 适合那些用黑盒探测就足够的小规模或静态系统的基础监控.

如果你想做白盒监控,或者有一个动态的或者基于云计算的环境,那么Prometheus是一个好的选择.

Prometheus vs. Sensu

[***Sensu***](https://sensuapp.org/) 一般来说是一个更现代的 nagios.

适用场景

和 nagios 的大多数场景相同,这里只描述一些有区别的.

最主要的区别是 Sensu 的客户端会自行向中心节点注册,并能同时从中心节点或本地配置文件确定要运行的检测项目.Sensu 对抽取的性能检测数据的数量没有限制.

同时 Sensu 也有一个 client socket 允许将随机检测数据推送到 Sensu.

数据模型

Sensu 和 nagios 的数据模型一样稍显粗糙.

存储

Sensu 讲数据存放在 redis 中.主要用于存放被静音的报警和客户端注册信息.

架构

Sensu 包含许多组件.使用 Rabbitmq作为消息队列,redis 用于存放当前状态,以及用于处理数据的单独服务器.

Rabbitmq 和 redis 都可以集群化.多个 Sensu 的副本可用作扩容和冗余.

总结

如果你已经在使用 Nagios,并希望使用 Sensu 的扩容和注册等高级功能.Sensu 是个不错的选择.

如果你想做白盒监控,或者有一个动态的或者基于云计算的环境,那么Prometheus是一个好的选择.

# 3@1046102779/prometheus: Prometheus官网的非官方中文手册

，旨在为大家提供一个比较容易入手的文档。翻译得不好，请大家多多包涵，并帮忙修订校正

https://github.com/1046102779/prometheus

# How to Use Prometheus to Monitor Your CentOS 7 Server | DigitalOcean

https://www.digitalocean.com/community/tutorials/how-to-use-prometheus-to-monitor-your-centos-7-server

Introduction

Prometheus is an open source monitoring system developed by SoundCloud. Like other monitoring systems, such as InfluxDB and Graphite, Prometheus stores all its data in a time series database. However, it offers a multi-dimensional data-model and a powerful query language, allowing system administrators to not only easily fine tune the definitions of their metrics, but also generate more accurate reports.

Additionally, the Prometheus project also includes PromDash (a browser-based tool that can be used to develop custom dashboards) and an experimental AlertManager capable of sending alerts via e-mail, Flowdock, Slack, HipChat and more.

In this tutorial, you will learn how to install, configure, and use the Prometheus Server, Node Exporter, and PromDash.

**Prerequisites**

To follow this tutorial, you will need:

* One 64-bit CentOS 7 Droplet
* A [non-root sudo user](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-centos-7), preferably one named **prometheus**.

**Step 1 — Installing Prometheus Server**

First, create a new directory to store all the files you download in this tutorial and move to it.

* mkdir ~/Downloads
* cd ~/Downloads

Use curl to download the latest build of the Prometheus server and time-series database from GitHub.

* curl -LO "https://github.com/prometheus/prometheus/releases/download/0.16.0/prometheus-0.16.0.linux-amd64.tar.gz"

The Prometheus monitoring system consists of several components, each of which needs to be installed separately. Keeping all the components inside one parent directory is a good idea, so create one usingmkdir.

* mkdir ~/Prometheus

Enter the directory you just created.

* cd ~/Prometheus

Use tar to extract prometheus-0.16.0.linux-amd64.tar.gz.

* tar -xvzf ~/Downloads/prometheus-0.16.0.linux-amd64.tar.gz

This completes the installation of Prometheus server. Verify the installation by typing in the following command:

* ~/Prometheus/prometheus-0.16.0.linux-amd64/prometheus -version

You should see the following message on your screen:

Prometheus output

prometheus, version 0.16.0 (branch: HEAD, revision: dcb8ba4)

build user: julius@desktop

build date: 20151009-23:51:17

go version: 1.5.1

**Step 2 — Installing Node Exporter**

Prometheus was developed for the purpose of monitoring web services. In order to monitor the metrics of your CentOS server, you should install a tool called Node Exporter. Node Exporter, as its name suggests, exports lots of metrics (such as disk I/O statistics, CPU load, memory usage, network statistics, and more) in a format Prometheus understands.

Enter the Downloads directory and use curl to download the latest build of Node Exporter which is available on GitHub.

* cd ~/Downloads && curl -LO "https://github.com/prometheus/node\_exporter/releases/download/0.11.0/node\_exporter-0.11.0.linux-amd64.tar.gz"

Create a new directory called node\_exporter inside the Prometheus directory, and get inside it:

* mkdir ~/Prometheus/node\_exporter
* cd ~/Prometheus/node\_exporter

You can now use the tar command to extract node\_exporter-0.11.0.linux-amd64.tar.gz.

* tar -xvzf ~/Downloads/node\_exporter-0.11.0.linux-amd64.tar.gz

**Step 3 — Running Node Exporter as a Service**

To make it easy to start and stop Node Exporter, let us now convert it into a service.

Use vi or any other text editor to create a unit configuration file called node\_exporter.service.

* sudo vi /etc/systemd/system/node\_exporter.service

This file should contain the path of the node\_exporter executable, and also specify which user should run the executable. Accordingly, add the following code:

/etc/init/node\_exporter.conf

[Unit]

Description=Node Exporter

[Service]

User=prometheus

ExecStart=/home/prometheus/Prometheus/node\_exporter/node\_exporter

[Install]

WantedBy=default.target

Save the file and exit the text editor.

Reload systemd so that it reads the configuration file you just created.

* sudo systemctl daemon-reload

At this point, Node Exporter is available as a service which can be managed using the systemctlcommand. Enable it so that it starts automatically at boot time.

* sudo systemctl enable node\_exporter.service

You can now either reboot your server, or use the following command to start the service manually:

* sudo systemctl start node\_exporter.service

Once it starts, use a browser to view Node Exporter's web interface, which is available athttp://your\_server\_ip:9100/metrics. You should see a page with a lot of text:

http://your\_server\_ip:9100/metrics excerpt

# HELP go\_gc\_duration\_seconds A summary of the GC invocation durations.

# TYPE go\_gc\_duration\_seconds summary

go\_gc\_duration\_seconds{quantile="0"} 0.00023853100000000002

go\_gc\_duration\_seconds{quantile="0.25"} 0.00023998700000000002

go\_gc\_duration\_seconds{quantile="0.5"} 0.00028122

. . .

**Step 4 — Starting Prometheus Server**

Enter the directory where you installed the Prometheus server:

cd ~/Prometheus/prometheus-0.16.0.linux-amd64

Before you start Prometheus, you must first create a configuration file for it called prometheus.yml.

* vi ~/Prometheus/prometheus-0.16.0.linux-amd64/prometheus.yml

Copy the following code into the file.

~/Prometheus/prometheus-0.16.0.linux-amd64/prometheus.yml

scrape\_configs:

- job\_name: "node"

scrape\_interval: "15s"

target\_groups:

- targets: ['localhost:9100']

This creates a scrape\_configs section and defines a job called node. It includes the URL of your Node Exporter's web interface in its array of targets. The scrape\_interval is set to 15 seconds so that Prometheus scrapes the metrics once every fifteen seconds.

You could name your job anything you want, but calling it "node" allows you to use the default console templates of Node Exporter.

Save the file and exit.

Start the Prometheus server as a background process.

* nohup ./prometheus > prometheus.log 2>&1 &

Note that you redirected the output of the Prometheus server to a file called prometheus.log. You can view the last few lines of the file using the tail command:

* tail ~/Prometheus/prometheus-0.16.0.linux-amd64/prometheus.log

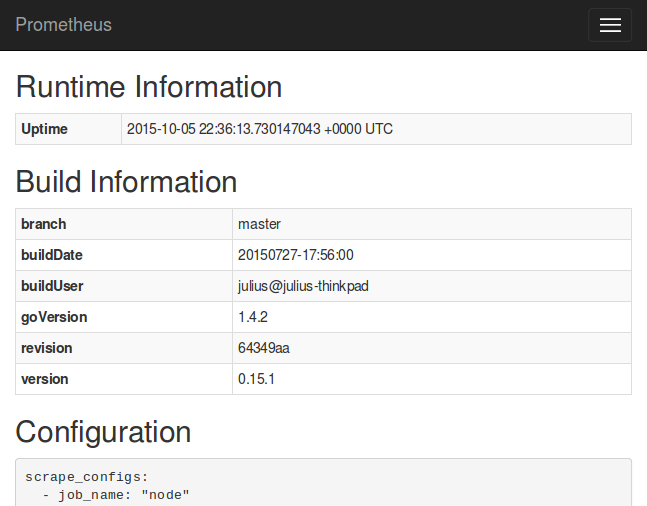
Once the server is ready, you will see the following messages in the file:

prometheus.log excerpt

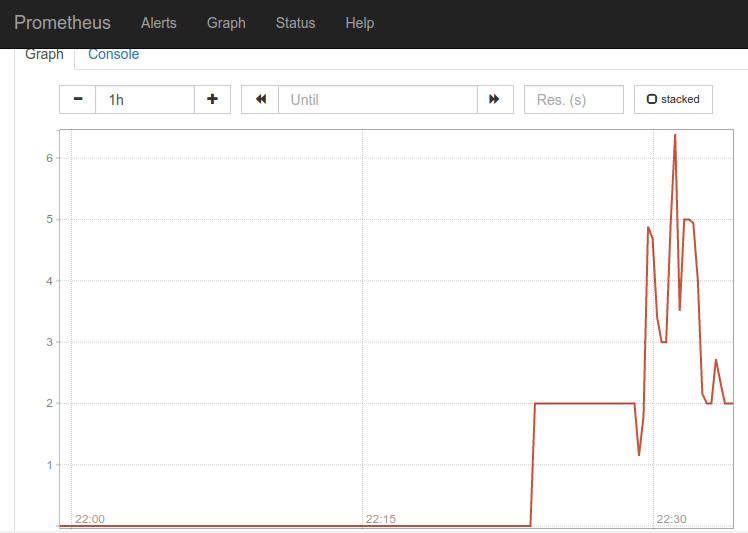
INFO[0000] Starting target manager... file=targetmanager.go line=75

INFO[0000] Listening on :9090 file=web.go line=118

Use a browser to visit Prometheus's homepage available at http://your\_server\_ip:9090. You'll see the following homepage.

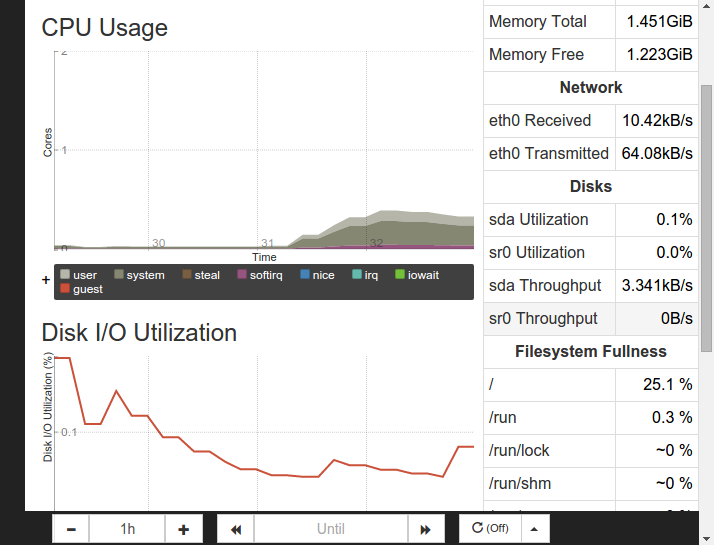


To make sure that Prometheus is scraping data from Node Exporter, click on the **Graph** tab at the top of the page. On the page that opens, type in the name of a metric (like **nodeprocsrunning**, for example) in the text field that says **Expression**. Then, press the blue **Execute** button. Click **Graph** (next to **Console**) just below, and you should see a graph for that metric:



Prometheus has console templates that let you view graphs of a few commonly used metrics. These console template are accessible only if you set the value of job\_name to node in Prometheus's configuration.

Visit http://your\_server\_ip:9090/consoles/node.html to access the Node Console and click on your server, **localhost:9100**, to view its metrics:



**Step 5 — Installing PromDash**

Though the Prometheus server allows you to view graphs and experiment with expressions, it is generally used only for debugging purposes or to run one-off queries. The preferred way to visualize the data in Prometheus's time-series database is to use PromDash, a tool that allows you to create custom dashboards which are not only highly configurable but also better-looking.

Enter the Prometheus directory:

* cd ~/Prometheus

PromDash is a Ruby on Rails application whose source files are available on GitHub. In order to download and run it, you need to install Git, Ruby and a few build tools. Use yum to do so.

* sudo yum install git ruby ruby-devel sqlite-devel zlib-devel gcc gcc-c++ automake patch

You can now use the git command to download the source files.

* git clone https://github.com/prometheus/promdash.git

Enter the promdash directory.

* cd ~/Prometheus/promdash

PromDash depends on several Ruby gems. In order to automate the installation of those gems, you should install a gem called bundler.

* gem install bundler

You can now use the bundle command to install all the Ruby gems that PromDash requires. As we will be configuring PromDash to work with SQLite3 in this tutorial, make sure you exclude the gems for MySQL and PostgreSQL using the --without parameter:

* bundle install --without mysql postgresql

You might have to wait for a few minutes for this command to complete. Once done, you should see the following messages:

Bundle output

. . .

Your bundle is complete!

Gems in the groups mysql and postgresql were not installed.

Use `bundle show [gemname]` to see where a bundled gem is installed.

**Step 6 — Setting Up the Rails Environment**

Create a directory to store the SQLite3 databases associated with PromDash.

* mkdir ~/Prometheus/databases

PromDash uses an environment variable called DATABASE\_URL to determine the name of the the database associated with it. Type in the following so that PromDash creates a SQLite3 database calledmydb.sqlite3 inside the databases directory:

* echo "export DATABASE\_URL=sqlite3:$HOME/Prometheus/databases/mydb.sqlite3" >> ~/.bashrc

In this tutorial, you will be running PromDash in production mode, so set the RAILS\_ENV environment variable to production.

* echo "export RAILS\_ENV=production" >> ~/.bashrc

Apply the changes we made to the .bashrc file.

* . ~/.bashrc

Next, create PromDash's tables in the SQLite3 database using the rake tool.

* rake db:migrate

Because PromDash uses the Rails Asset Pipeline, all the assets(CSS files, images and Javascript files) of the PromDash project should be precompiled. Type in the following to do so:

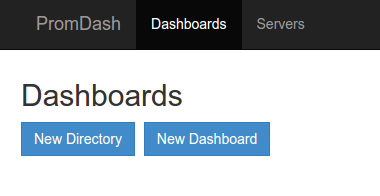
* rake assets:precompile

**Step 7 — Starting and Configuring PromDash**

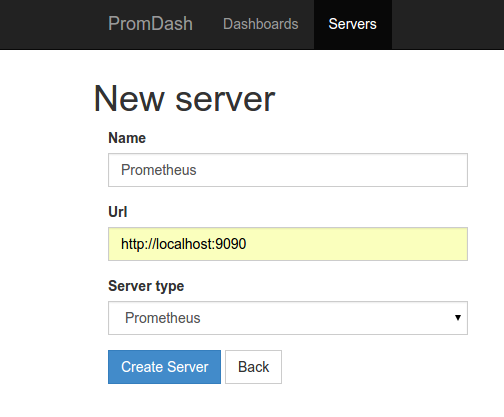
PromDash runs on Thin, a light-weight web server. Start the server as a daemon by typing in the following command:

* bundle exec thin start -d

Wait for a few seconds for the server to start and then visit http://your\_server\_ip:3000/ to view PromDash's homepage.



Before you start creating your custom dashboards, you should let PromDash know the URL of your Prometheus server. You can do so by clicking on the **Servers** tab at the top. Click **New Server**, then in the form, give any name to your Prometheus server. Set the **Url** field to http://your\_server\_ip:9090 and the**Server type** field to **Prometheus**.



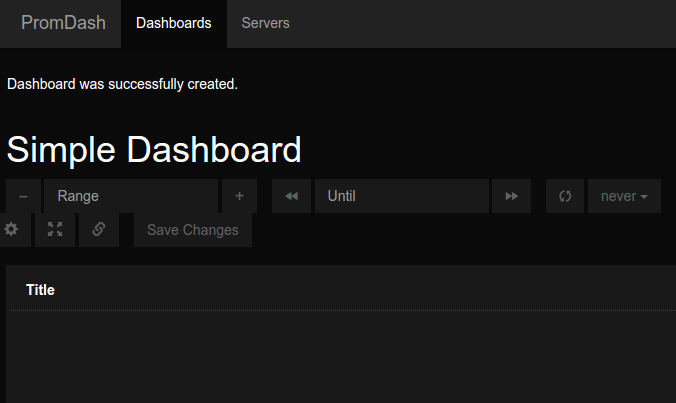
Finally, click on **Create Server** to complete the configuration. Your page will say **Server was successfully created.** and you can click back to **Dashboards** in the top menu.

**Step 8 — Creating a Dashboard**

Because a Promdash dashboard should belong to a Promdash directory, first create a new directory by clicking on the **New Directory**. In the form that shows up, give a name to your directory, like **My Dashboards**, then click **Create Directory**.

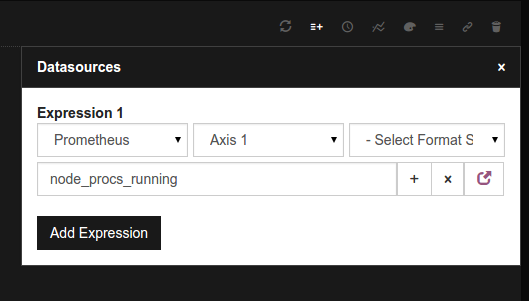
Once you submit the form, you will be taken back to the homepage. Click on the **New Dashboard** button now to create a new dashboard. In the form shown, give a name to your dashboard, like **Simple Dashboard**and select the directory you just created from the drop-down menu.

After submitting the form, you will be able to see the new dashboard.



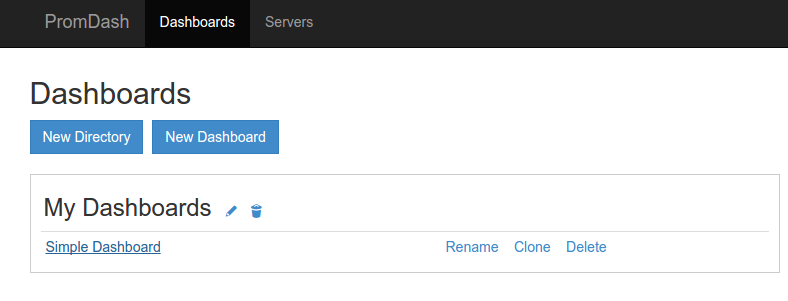
Your dashboard already has one graph, but it needs to be configured. Hovering over the graph's header (which says **Title**) will reveal various icons that let you configure the graph. To change its title, you can click on the **Graph and Axis Settings** icon (fourth from the left) and type in a new title in the **Graph Title**field.

Click on the **Datasources** icon, which is the second to the left, to add one or more expressions to the graph. Click **Add Expression**, and in the field that says **Enter Expression**, enter **nodeprocsrunning**.



Now click on the **Refresh** icon (the leftmost one) in the graph's header to update the graph. Your dashboard now contains one fully configured graph. You can add more graphs by clicking on the **Add Graph** button at the bottom.

After making all the changes, make sure you click on the **Save Changes** button on the right to make your changes permanent. The next time you visit PromDash's homepage, you will be able to see a link to your dashboard:



**Conclusion**

You now have a fully functional Prometheus ecosystem running on your CentOS 7 server, and you can use PromDash to create monitoring dashboards that suit your requirements.

Even though you installed all the components on a single CentOS machine, you can easily monitor more machines by installing only Node Exporter on each of them, and adding the URLs of the new Node Exporters to the targets array of prometheus.yml.

You can learn more about Prometheus by referring to its [documentation](http://prometheus.io/docs/introduction/overview/).

# Prometheus 和 Grafana 监控系统指南 - 推酷

http://www.tuicool.com/articles/vEVjai

时间 2016-11-22 06:13:28  [互联网技术和架构](http://www.tuicool.com/sites/qqamaa)

原文  [https://blog.eood.cn/prometheus-grafana-monitoring](https://blog.eood.cn/prometheus-grafana-monitoring?utm_source=tuicool&utm_medium=referral)

主题 [Grafana](http://www.tuicool.com/topics/11040110)

Prometheus 是源于 Google Borgmon 的一个开源监控系统，用 Golang 开发。被很多人称为下一代监控系统。

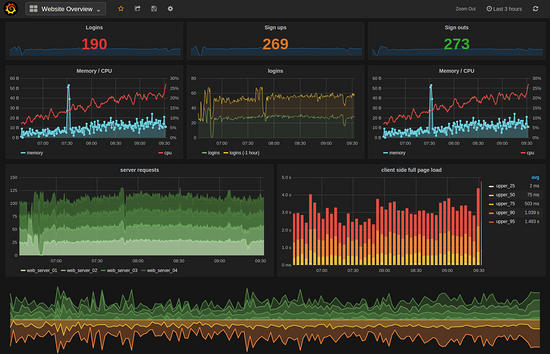
[***Prometheus***](https://prometheus.io/) 基本原理是通过 HTTP 协议周期性抓取被监控组件的状态，这样做的好处是任意组件只要提供 HTTP 接口就可以接入监控系统，不需要任何 SDK 或者其他的集成过程。这样做非常适合虚拟化环境比如 VM 或者 Docker 。

Prometheus 应该是为数不多的适合 Docker、Mesos 、Kubernetes 环境的监控系统之一。

输出被监控组件信息的 HTTP 接口被叫做 exporter 。目前互联网公司常用的组件大部分都有 exporter 可以直接使用，比如 Varnish、Haproxy、Nginx、MySQL、Linux 系统信息 (包括磁盘、内存、CPU、网络等等)。

[***Grafana***](https://grafana.net/) 是一个开源的图表可视化系统，简单说图表配置比较方便、生成的图表比较漂亮。

但是 Prometheus 还比较新，要用在生产环境还需要解决一系列的问题，比如和 Grafana 集成相关资料并不多，这篇文章简单介绍了这些问题的解决方法。



1. Prometheus 的查询系统

Prometheus 提供了一个简单的查询界面 http://127.0.0.1:9090/graph

在这里可以查询各个 exporter 的基础信息，比如 node\_load1, haproxy\_backend\_http\_responses\_total, nginx\_http\_requests\_total 等等图标和标签：

这些信息分两种，一种是状态信息，比如 node\_load1, 表示组件当前的状态。

查询 node\_load1 会输出每台服务器的 CPU 负载图表和标签：

node\_load1{**alias**="web",**instance**="y.y.y.x:9100",job="linux"}

node\_load1{**alias**="web",**instance**="y.y.y.y:9100",job="linux"}

node\_load1{**alias**="db",**instance**="y.y.y.y:9100",job="linux"}

...

{} 内部的 Key-Value 对是查询过滤的维度，这样可以方便的根据标签合并计算，比如按业务类型聚合、或者按服务器类型聚合、或者按机架聚合。

node\_load1 是不加任何过滤条件的查询会输出所有机器的信息。

node\_load1{**alias**="web"} 只输出 **alias**="web" 的机器信息。

node\_load1{**alias**="web",**instance**="y.y.y.x:9100"} 只输出 **alias**="web" 并且 **instance**="y.y.y.x:9100" 的机器信息。

另外 = (相等) 还可以换成 =~ 正则匹配，

更多信息可以查看 prometheus 查询官方文档，(虽然例子并不是完整)：https://prometheus.io/docs/querying/basics/

这类信息可以直接绘制到 Grafana 系统中。

另外一种是累加值信息，比如 haproxy\_backend\_http\_responses\_total，node\_network\_transmit\_bytes

node\_network\_transmit\_bytes{**alias**="web",device="lo",**instance**="x.x.x.x:9100",job="linux"}

node\_network\_transmit\_bytes{**alias**="web",device="eth1",**instance**="x.x.x.x:9100",job="linux"}

node\_network\_transmit\_bytes{**alias**="web",device="eth2",**instance**="x.x.x.x:9100",job="linux"}

node\_network\_transmit\_bytes{**alias**="web",device="eth3",**instance**="x.x.x.x:9100",job="linux"}

这类信息实际上更加常用，需要再次处理才能转换成我们熟悉的类似于 QPS 之类的 Rate 曲线才能输出到 Grafana 系统中，转换方式如下：

rate(**nginx\_http\_requests\_total**{instance="$node", status="200"}[5m])

这个 Query 是输出某台服务器 Nginx 200 请求的 QPS 。

更多例子见文章最后的小密圈。

完整文档可以参考：

https://prometheus.io/docs/querying/operators/

2. Prometheus 的安装和启动，这里参考 percona 的一篇文章：

wget https://github.**com**/prometheus/prometheus/releases/download/0.17.0rc2/prometheus-0.17.0rc2.linux-amd64.tar.gz

mkdir /**opt**/prometheus

tar zxf prometheus-0.17.0rc2.linux-amd64.tar.gz -C /**opt**/prometheus --strip-components=1

**cat** << EOF > /**opt**/prometheus/prometheus.yml

global:

scrape\_interval: 5s

evaluation\_interval: 5s

scrape\_configs:

- job\_name: linux

target\_groups:

- targets: ['x.x.x.x:9100']

labels:

alias: db1

- job\_name: mysql

target\_groups:

- targets: ['x.x.x.x:9104']

labels:

alias: db1

EOF

**cd** /**opt**/prometheus

./prometheus

即可完成。

3. Node exporter (Linux 服务器基础信息)的安装和启动：

wget https://github.com/prometheus/node\_exporter/releases/download/0.12.0rc3/node\_exporter-0.12.0rc3.linux-amd64.tar.gz

tar zxf node\_exporter-0.12.0rc3.linux-amd64.tar.gz -C /opt/prometheus\_exporters

cd /opt/prometheus\_exporters

./node\_exporter

即可完成。

至此，你已经可以在 Prometheus 查询到 Node 相关的图表了。/

可以发现要把这些用在自己的系统中，需要自己做 init 启动管理脚本或者打包成 RPM 。

关于如何非常方便的快速打包， **互联网架构小密圈 Roundabout** 里有一个非常实用的工具链接。

4. Grafana 的安装和启动 (这里用比较稳定的 2.6.0 版本)

yum install https://grafanarel.s3.amazonaws.com/builds/grafana-2.6.0-1.x86\_64.rpm

打 2.6.0 补丁

sed -i 's/step\_input:""/step\_input:c.target.step/; s/ HH:MM/ HH:mm/; s/,function(c)/,"templateSrv",function(c,g)/; s/expr:c.target.expr/expr:g.replace(c.target.expr,c.panel.scopedVars)/' /usr/share/grafana/**public**/app/plugins/datasource/prometheus/query\_ctrl.js

sed -i 's/h=a.interval/h=g.replace(a.interval, c.scopedVars)/' /usr/share/grafana/**public**/app/plugins/datasource/prometheus/datasource.js

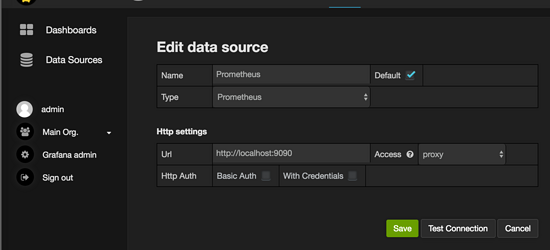
/etc/init.d/grafana-server start

至此安装完成。

浏览器打开 http://127.0.0.1:3000 ，输入默认用户名密码 (admin/admin) 可以进入 Grafana 。

然后配置数据源：

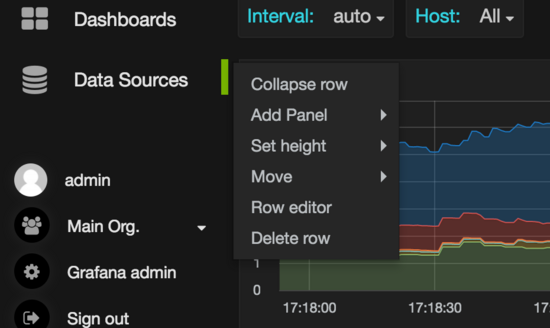
Prometheus： URL： http://127.0.0.1:9090/



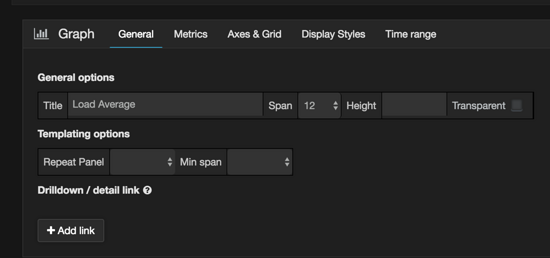
即可完成 Prometheus 和 Grafana 的对接。

5. Grafana 图表的创建和编辑

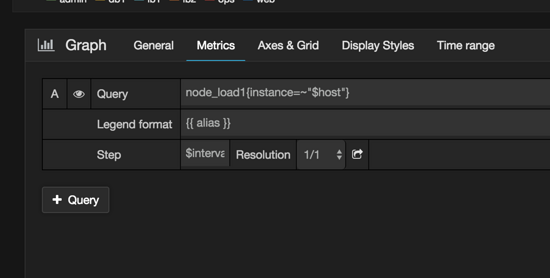
在界面上点击绿色色块，可以添加图表 Graph:



点击现有图表的标题可以复制和编辑, 这里可以编辑图表标题和宽度、高度：

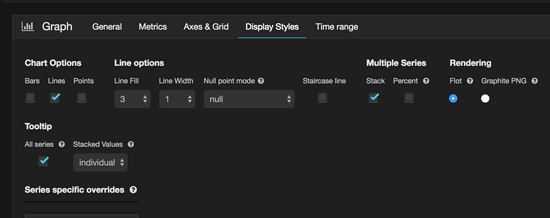


这里可以编辑 Prometheus 的查询语句：



Grafana 支持 Y 轴单位的选择，比如流量、吞吐量、磁盘使用量等等：

Grafana 支持堆叠图和曲线、柱状图各种漂亮的图表：



6. 常见问题：

6.1. 关于高可用

很多人担心这种抓取的模式的高可用问题，比如和目标节点的通讯异常或者某次的采集丢失怎么办？Prometheus 的模型和计算方式是采样然后聚合计算，假如某些样本没有采集到则这些样本不参与计算。

6.2. Prometheus 的Grafana 图表模板比较少：

Grafana 并没有太多的配置好的图表模板，除了 Percona 开源的一些外，很多需要自行配置。

Grafana 提供了一些例子，但是还是很少：https://grafana.net/dashboards

这里准备了几个常用的组件 Linux Node、Varnish、Nginx、Memcache、Haproxy、MySQL 的简单模板例子 (可以直接导入 Grafana 2.6.0 使用)和 Exporter 安装方法可以从 **互联网架构小密圈 Roundabout** 查看和下载:

prometheus\_20161121.txt

prometheus\_20161121.zip

# Prometheus 初探 - 推酷

http://www.tuicool.com/articles/RvmueeR

时间 2017-03-08 00:03:26  [addops](http://www.tuicool.com/sites/3y6R7nY)

原文  [https://addops.cn/post/Prometheus-first-exploration.html](https://addops.cn/post/Prometheus-first-exploration.html?utm_source=tuicool&utm_medium=referral)

主题 [Graphite](http://www.tuicool.com/topics/11040076)[数据库](http://www.tuicool.com/topics/11000064)

首先-什么是 TSDB (Time Series Database):

我们可以简单的理解为.一个优化后用来处理时间序列数据的软件,并且数据中的数组是由时间进行索引的.

时间序列数据库的特点:

* 大部分时间都是写入操作
* 写入操作几乎是顺序添加;大多数时候数据到达后都以时间排序.
* 写操作很少写入很久之前的数据,也很少更新数据.大多数情况在数据被采集到数秒或者数分钟后就会被写入数据库.
* 删除操作一般为区块删除,选定开始的历史时间并指定后续的区块.很少单独删除某个时间或者分开的随机时间的数据.
* 数据一般远远超过内存大小,所以缓存基本无用.系统一般是 IO 密集型
* 读操作是十分典型的升序或者降序的顺序读,
* 高并发的读操作十分常见.

常见的时间序列数据库:

部分常见 TSDB 官网如下:

| **TSDB** | **官网** |
| --- | --- |
| influxDB | [***https://influxdata.com/***](https://influxdata.com/) |
| RRDtool | [***http://oss.oetiker.ch/rrdtool/***](http://oss.oetiker.ch/rrdtool/) |
| Graphite | [***http://graphite.readthedocs.org/en/latest/***](http://graphite.readthedocs.org/en/latest/) |
| OpenTSDB | [***http://opentsdb.net/***](http://opentsdb.net/) |
| Kdb+ | [***http://kx.com/***](http://kx.com/) |
| Druid | [***http://druid.io/***](http://druid.io/) |
| KairosDB | [***http://kairosdb.github.io/***](http://kairosdb.github.io/) |
| Prometheus | [***https://prometheus.io/***](https://prometheus.io/) |

关于 Prometheus

Prometheus是什么?

[***Prometheus***](https://github.com/prometheus) 是由 SoundCloud 开发的开源监控报警系统和时序列数据库(TSDB).自2012年起,许多公司及组织已经采用 Prometheus,并且该项目有着非常活跃的开发者和用户社区.现在已经成为一个独立的开源项目核,并且保持独立于任何公司,Prometheus 在2016加入 CNCF ( [***Cloud Native Computing Foundation***](https://cncf.io/) ), 作为在 kubernetes 之后的第二个由基金会主持的项目.

prometheus 的特点

和其他监控系统相比，Prometheus的特点包括：

* 多维数据模型（时序列数据由metric名和一组key/value组成）
* 在多维度上灵活的查询语言(PromQl)
* 不依赖分布式存储，单主节点工作.
* 通过基于HTTP的pull方式采集时序数据
* 可以通过中间网关进行时序列数据推送(pushing)
* 目标服务器可以通过发现服务或者静态配置实现
* 多种可视化和仪表盘支持

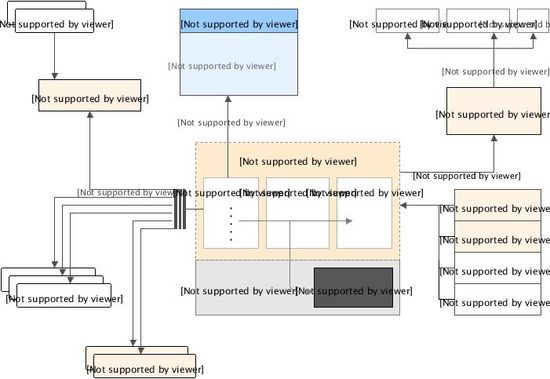
prometheus 相关组件

Prometheus生态系统由多个组件组成，其中许多是可选的：

* Prometheus 主服务,用来抓取和存储时序数据
* client library 用来构造应用或 exporter 代码 (go,java,python,ruby)
* push 网关可用来支持短连接任务
* 可视化的dashboard (两种选择,promdash 和 grafana.目前主流选择是 grafana.)
* 一些特殊需求的数据出口(用于HAProxy, StatsD, Graphite等服务)
* 实验性的报警管理端(alartmanager,单独进行报警汇总,分发,屏蔽等 )

promethues 的各个组件基本都是用 golang 编写,对编译和部署十分友好.并且没有特殊依赖.基本都是独立工作.

prometheus 的架构



部署及配置

promethues 官方给出了多重部署方案,包括但不限于 docker 容器,ansible,chef,saltstack 等.

其实官方已经给了预编译的二进制文件.如果没有修改代码的特殊需求,直接拿到二进制文件进行部署也是可以的.

下载地址: [***https://prometheus.io/download/***](https://prometheus.io/download/)

部署方式十分简单

**tar** xvfz prometheus-\*.tar.gz

cd prometheus-\*

在 prometheus 目录下有一个名为 prometheus.yml 的主配置文件.其中包含大多数标准配置及 prometheus 的自检控配置,配置文件如下.

**global**:

**scrape\_interval**: 15s # 默认抓取间隔, 15秒向目标抓取一次数据

# 和外部系统交互时每一条从本机获取的数据都会打上如下标签

**external\_labels**:

**monitor**: 'codelab-monitor'

# 这里是抓去 promethues 自身的配置

**scrape\_configs**:

# job name 会以标签`job=<job\_name>`添加到每一条由该配置抓去到的时序数据

- **job\_name**: 'prometheus'

# 覆盖默认抓取间隔

**scrape\_interval**: 5s

**static\_configs**:

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

# 添加两个线上抓取实例

- **job\_name**: 'ceph'

**scrape\_interval**: 5s

**static\_configs**:

- **targets**: ['k0140v:9128']

**labels**:

**group**: 'shbt' # 会对该配置生成的时序数据添加一条 `group=<group\_name>`的标签

- **targets**: ['ceph01:9128']

**labels**:

**group**: 'bjyt'

- **job\_name**: 'openstack'

**scrape\_interval**: 30s

**static\_configs**:

#target 可以使用 "," 分割,添加多个目标

- **targets**: ['openstack185:9128', 'openstack194:9128']

**labels**:

**group**: "bjyt"

然后我们通过该配置文件启动 promethues

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

prometheus 本身是自带 exporter 的,我们通过请求 [***http://localhost:9090/metrics***](http://localhost:9090/metrics) 可以查看从 exporter 中能具体抓到哪些数据

监控数据

数据样本

和大多数 TSDB 类似,promethus 支持的数据样本非常简单:

* 一个 float64 的值
* 一个毫秒精度的时间戳

标识

标识监控数据的方式十分简单,给一个监控项名称和一些标签,时序数据经常使用这种标识方法

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

举例来说, 一个时间序列的监控名称为 **api\_http\_requests\_total** 标签为 **method="POST"**和 **handler="/message"** ,那么监控数据的标识如下:

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

这种标识方式与 [***OpenTSDB***](http://opentsdb.net/) 相同.

监控结果

我们实际收集到的数据大多如下:

ceph\_osd\_avail\_bytes{osd="osd.0"} 3.205084244e+12

ceph\_osd\_avail\_bytes{osd="osd.1"} 2.892447332e+12

ceph\_osd\_avail\_bytes{osd="osd.10"} 3.21853432e+12

ceph\_osd\_avail\_bytes{osd="osd.100"} 3.062200424e+12

ceph\_osd\_avail\_bytes{osd="osd.101"} 3.126474844e+12

ceph\_osd\_avail\_bytes{osd="osd.102"} 3.079620512e+12

ceph\_pool\_available\_bytes{pool="backups"} 1.30342001987587e+14

ceph\_pool\_available\_bytes{pool="images"} 1.30342001987587e+14

ceph\_pool\_available\_bytes{pool="rbd"} 1.30342001987587e+14

ceph\_pool\_available\_bytes{pool="rgw-test"} 1.30342001987587e+14

ceph\_pool\_available\_bytes{pool="test\_crush"} 1.30342001987587e+14

ceph\_pool\_available\_bytes{pool="vms"} 1.30342001987587e+14

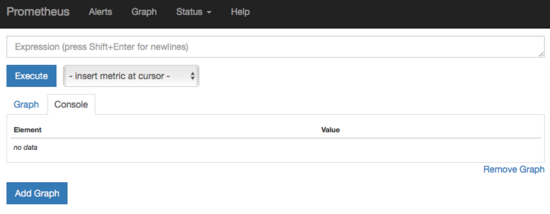
ceph\_pool\_available\_bytes{pool="vmscache"} 2.301547932444e+12

###监控数据类型

* Counter：计数器是一个累加的度量类型，记录单调递增的数据.一般用于记录服务的请求数，任务完成数，错误发生数.
* Gauge：量表用于记录可任意变大变小的数值,如cpu idle,内存使用率,磁盘IO等。
* Historgram：直方图用于度量数据中值的分布情况，如请求时间或响应大小。
* Summary: 用于度量数据累价值或总数.

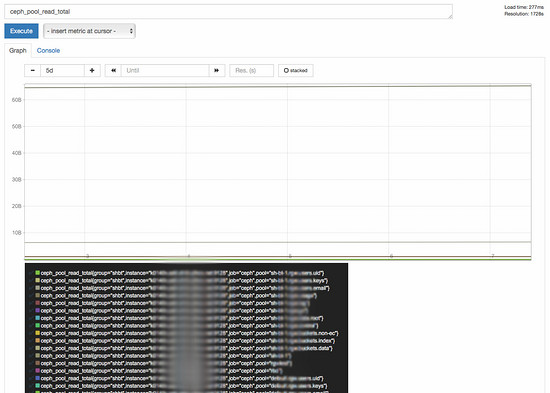
自带 dashboard及查询语句

prometheus 自带一个比较简单的dashboard 可以查看表达式搜索结果,报警配置,prometheus 配置,exporter 状态等



我们以一些真实数据为例看一下表达式及查询结果.

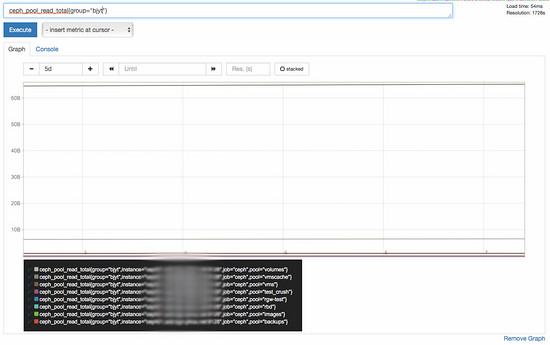
1. 我们可直接查询监控项ceph\_pool\_read\_total并绘图如下



2. 我们也可以在查询语句中通过添加一组标签,并用 **{}** 阔起来,来细化查询.

例如我们只想查看 bjyt 这个 group 各个 pool 全局读取的数据.

ceph\_pool\_read\_total{group="bjyt"}



另外，也可以也可以将标签值反向匹配，或者对正则表达式匹配标签值:

* =：选择相等的字符串标签
* !=：选择不相等的字符串标签
* =~：选择匹配正则表达式的字符串标签（或子标签）
* !=：选择不匹配正则表达式的字符串标签（或子标签）

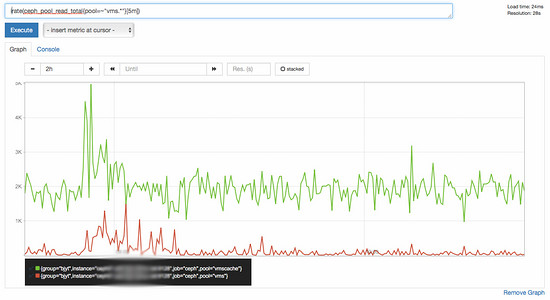
例如

ceph\_pool\_read\_total{pool=~"vms.\*"}

会如上图但只查询出 pool 名为 vms 及 vmscahe 的数据.

3. 如果我们以时间窗口来作为筛选纬度计算各个 pool 读IO的真实速率,可以用以下语句查询

irate(**ceph\_pool\_read\_total**{pool=~"vms"}[1m])



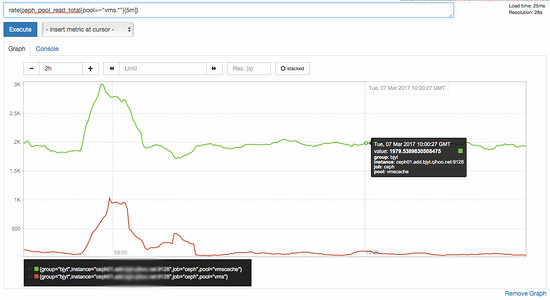
这里的 irate() 为 promethues 的查询函数.与之对应的是rate().

这两个函数在 promethues 中经常用来计算增量或者速率,在使用时需要指定时间范围如[1m]

* irate(): 计算的是给定时间窗口内的每秒瞬时增加速率.
* rate(): 计算的是给定时间窗口内的每秒的平均值.

如果还是以前面的监控项进行查询但是以 rate() 计算速率的话,绘制的结果如下:

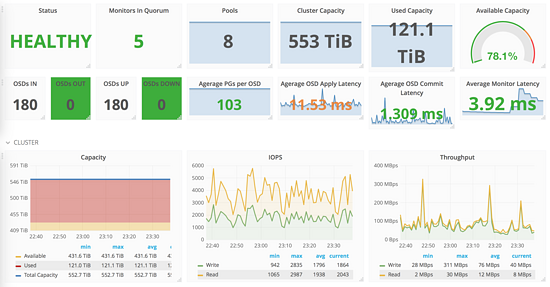
rate(**ceph\_pool\_read\_total**{pool=~"vms"}[1m])



promethues 支持的函数还有很多,具体的函数说明可详见官方文档 [***Functions***](https://prometheus.io/docs/querying/functions/)

如果在线上使用的话,可以将 Prometheus 和 grafana 相结合.可以进行十分丰富的监控结果展示.

比如 ceph 单机房集群的监控可以是这样的:



先简单介绍这么多.

后面会通过几篇文章详细介绍一些常用的查询方法, exporter的编写及使用方法,以及 promethues 如何结合 grafana使用和promethues 是如何进行报警的.

参考文献 :

* [***http://liubin.org/blog/2016/02/18/tsdb-intro/***](http://liubin.org/blog/2016/02/18/tsdb-intro/)
* [***https://www.xaprb.com/blog/2014/06/08/time-series-database-requirements/***](https://www.xaprb.com/blog/2014/06/08/time-series-database-requirements/)
* [***https://www.xaprb.com/blog/2014/03/02/time-series-databases-influxdb/***](https://www.xaprb.com/blog/2014/03/02/time-series-databases-influxdb/)
* [***https://zhuanlan.zhihu.com/p/24811652***](https://zhuanlan.zhihu.com/p/24811652)

# Scaling with discovery on Docker Swarm with Consul, Registrator and HAProxy with Prometheus monitoring and ELK log aggregation

http://sirile.github.io/2015/07/28/scaling-with-discovery-on-docker-swarm-with-consul-registrator-and-haproxy-with-prometheus-monitoring-and-elk-log-aggregation.html

**Update on 5.8.2015!** The scripts have been updated so that the examples can also be run in Amazon Virtual Private Cloud. More information can be found from the [second part of the post](http://sirile.github.io/2015/08/05/part-2-scaling-in-amazon-aws-vpc-with-docker-docker-machine-consul-registrator-haproxy-elk-and-prometheus.html).

**Update on 6.10.2015!** The scripts have been quite extensively updated and cleaned up for presentation in OpenSlava 2015. I’ll describe the new functionality in a separate blogpost later, but have updated this so that the commands should work. Major addition has been the setting up of private registry both locally and in AWS and loading the images there. The startService.sh command first tried to use the local registry and if the image can’t be found there, it’s downloaded from the Docker hub. This also supports the experimental overlay network version of the demo.

**Update on 3.4.2016!** As I had changed the scripts so that the startService.sh tries to fetch the image from external if it can’t be found from the local I updated also this post. Thanks for notifying @Laxman-SM.

**General**

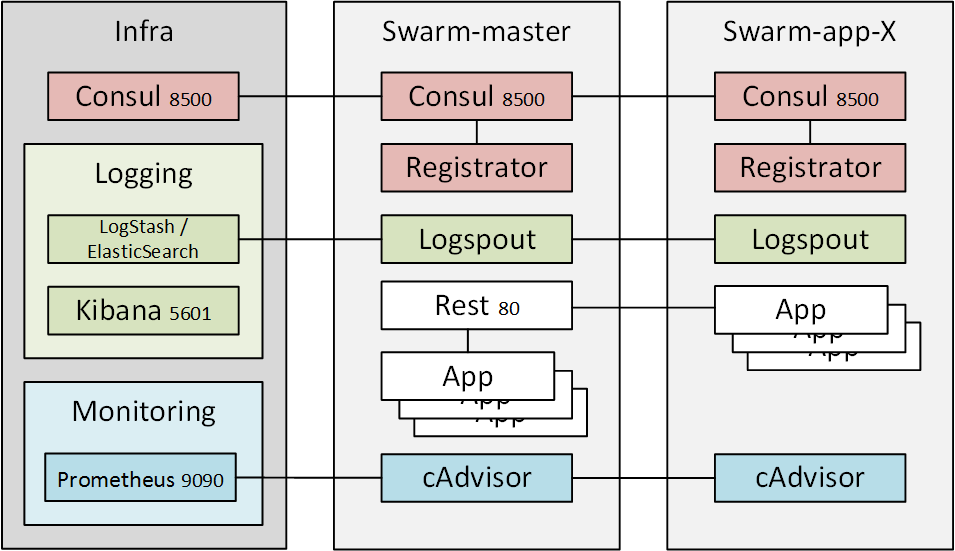
After playing around quite a lot with automatic scaling using[Registrator triggered HAProxy configuration generation](http://sirile.github.io/2015/05/18/using-haproxy-and-consul-for-dynamic-service-discovery-on-docker.html) I wanted to combine it with Docker Swarm. At the same time I wanted to see if [log aggregation using ELK-stack](http://sirile.github.io/2015/06/26/elasticsearch-logstash-kibana-and-logspout-on-docker.html) (ElasticSearch, LogStash and Kibana) could be made pluggable and add Prometheus based metrics collection as an option.

There are a few frameworks that achieve pretty much the same result, but I wanted to build from scratch to get to know how the components work. The end result is a handful of shell scripts that can be used to set up the Docker Swarm and add or remove logging and monitoring to all the nodes. If new nodes are added, just by running the script again they are joined to the monitoring and log aggregation system.

A three node swarm set-up with logging and monitoring running five instances of a test service can be done with:

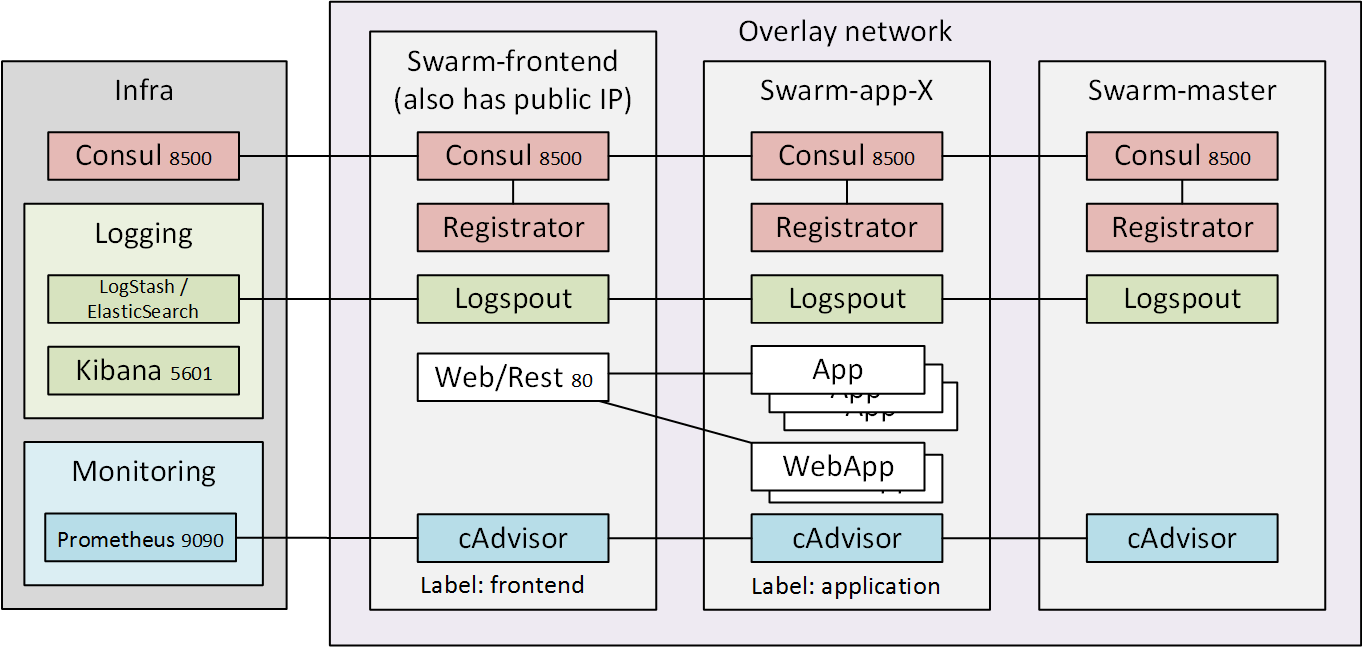
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | git clone https://github.com/SirIle/docker-multihost.git && cd docker-multihost/swarm  scripts/setupRegistry.sh  scripts/createInfraNode.sh  for i in {0..2}; do scripts/createSwarmNode.sh $i; done  scripts/addLogging.sh  scripts/addMonitoring.sh  for i in {1..5}; do ./startService.sh hello/v1 node-image-test; done |

This automatically starts a HAProxy which acts as the rest endpoint and directs traffic with the url pattern of /hello/v1 to the application containers. HAProxy itself could be started on any of the swarm nodes. If wanted this can be controlled with labels. The scripts display the public IP address of the endpoint. As Consul is used, joining the local Consul server to the one controlling the swarm is also an option and then finding the application becomes just <http://rest.service.consul/hello/v1>.



**Target architecture**

As Docker overlay networking matures I’ll move towards target architecture where I have front-end nodes that expose ports outside and application nodes that are only reachable through the overlay networking. At the moment this set-up doesn’t yet work and registrator doesn’t understand overlay networking services at all so Consul based service discovery can’t be used with it. Docker Swarm labels can be used to distinguish different roles and so direct the applications to specific swarm members, but that isn’t used in this example as all the nodes can work in all the roles.



**Cloud service providers**

Everything should work also when running against a cloud based service providers, only change should be in the createInfraNode.sh and createSwarmNode.sh where instead of using the VirtualBox driver another driver (with the required extra configuration like AWS token) should be used.

**Prerequisites**

The following tools need to be installed for the scripts to work. Everything has been tested on Mac, but should work also on Linux. For Windows tweaking would be necessary, but all the tools have been ported, so just tweaking the scripts should be enough.

* VirtualBox (5.0.4)
* Docker Machine (0.4.1)
* Docker (1.8.2)

**Docker Compose**

I tried using Docker Compose to define the nodes, but as it doesn’t yet [support](https://github.com/docker/compose/issues/1377) [variable](https://github.com/docker/compose/issues/495) [expansion](https://github.com/docker/compose/issues/64) passing the required information for the services isn’t at the moment possible. I may explore it again later as there have been a lot of requests for that functionality.

**Components**

The basic architecture consists of an infra node which runs the Consul master. In addition it can run the Prometheus server doing the performance metrics aggregation and the ELK stack for log aggregation and browsing.

Docker Swarm nodes are used to host the applications and also the HAProxy which does load-balancing between different application containers. The first created node (or node id 0) is called swarm-master and the Swarm commands are executed through it. It is the only required server, but more instances can be created and they’re automatically joined into the swarm.

There are also a few helpful functions which make finding and controlling containers in the swarm easier.

**Log aggregation**

Log aggregation is done by running [ELK-stack](https://www.elastic.co/webinars/introduction-elk-stack) consisting of ElasticSearch, LogStash and Kibana on the infra node and [logspout](https://github.com/gliderlabs/logspout) on the swarm nodes.

**Monitoring**

I used the [article at CenturyLink labs](https://labs.ctl.io/monitoring-docker-services-with-prometheus/) as the starting point in adding [Prometheus](http://prometheus.io/) as the runtime monitoring information collector. All the swarm nodes have [cAdvisor](https://github.com/google/cadvisor) running which can automatically export the runtime information for Prometheus. Prometheus configuration is re-written if the script is run again and Prometheus is signaled so that it will reload the configuration on the fly.

**Getting the code**

The scripts can be checked out with

|  |  |
| --- | --- |
| 1 | git clone https://github.com/SirIle/docker-multihost.git && cd docker-multihost/swarm |

**Creating the Swarm**

**Set up private registry**

A private registry is created with

|  |  |
| --- | --- |
| 1 | scripts/setupRegistry.sh |

This also pulls, tags and pushes specific images to the registry.

**Create infra node**

Infra node is created with the script

|  |  |
| --- | --- |
| 1 | scripts/createInfraNode.sh |

It first checks if infra node hasn’t been already created quitting if it has and then starts the Consul server.

**Create swarm master**

Swarm master is the first node and can be created by giving the command a 0 as an argument.

|  |  |
| --- | --- |
| 1 | scripts/createSwarmNode.sh 0 |

The command line for starting the Swarm master has the corresponding flag set, but otherwise the node itself is almost the same as for normal nodes. The Consul running on infra is used as the discovery back-end and so the infra node needs to be running.

**NB!** If you only start this one node, please add “front-end” after the command as the startService.sh script starts the HAProxy on a that node.

**Optional: create zero or many swarm members**

More swarm members can be created at any time by running the same script with a different id

|  |  |
| --- | --- |
| 1  2 | scripts/createSwarmNode.sh 1 front-end  scripts/createSwarmNode.sh 2 back-end |

The started nodes automatically start Registrator and Consul and join the Consul running on infra node.

**Start service(s)**

I created two simple containers that can be used to demonstrate scaling called [sirile/node-test](https://github.com/SirIle/node-test) and[sirile/scala-boot-test](https://github.com/SirIle/scala-boot-test). They can both be found from the Docker Hub, so there is no need to build them locally. Node-test is 23.31 MB and scala-boot-test 191 MB (as the JRE is rather large) so downloading them shouldn’t take too long. They both output the hostname, current timestamp and the implementation language, Javascript and Node respectively.

Any service can be used as long as it exposes the service through port 80 so that registrator picks it up and HAProxy configuration is written correctly. The real port that is visible outside is controlled by Docker and HAProxy can then direct traffic there based on the given url identifier.

When a service is started using the script it checks that at least one HAProxy container is running and if not, starts it and shows the address of the HAProxy through which the service is available.

|  |  |
| --- | --- |
| 1  2  3  4 | $ ./startService.sh hello/v1 sirile/node-test  \*\* Starting image sirile/node-test with the name hello/v1 \*\*  a1aaa0691a548aa9cc6db024537f83dc0c2f08d7344f1e1e41a69dc28b91db5f  \*\* Service available at http://192.168.99.102/hello/v1 \*\* |

**Using Consul for service discovery in the containers**

Consul has been specified as the default DNS server for the containers. Registered services can be looked up normally, so after starting a container you can do this (based on the previous example):

|  |  |
| --- | --- |
| 1  2  3  4 | $ docker exec -it 16f25b75b4fdee72be6d992c4c7f39f01604c2b4cd5c2a062b0779d031f403f0 ping consul.service.consul  PING consul.service.consul (192.168.99.100): 56 data bytes  64 bytes from 192.168.99.100: seq=0 ttl=63 time=0.208 ms  64 bytes from 192.168.99.100: seq=1 ttl=63 time=0.434 ms |

**Add log aggregation**

ELK based log aggregation system can be added with the script

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ ./addLogging.sh  \*\* Starting LogBox and Kibana on infra \*\*  decc99d53c5033f1758b4b63973daff3b63e1a3923454647ba662fd86ea37d08  c5e9f612143a84550921fafbf6b7a122451f3463ee852c66268eed7881bfeb2f  \*\* Servers in the swarm: swarm-1 swarm-0 \*\*  \*\* Starting logspout on swarm-1  2ca445fd69c85383ea2485004e3ad0584e94910281981dcbfc8f7b7788e48fc0  \*\* Starting logspout on swarm-0  ad5adf8a329eb4243f87652df8da04f722bfec9fa2ee96062ba6f2a773339b2e  \*\* Logging system started, Kibana is available at http://192.168.99.100:5601 \*\* |

This starts the LogBox container on infra node which consists of LogStash and ElasticSearch. It also starts Kibana that acts as the UI and tells the address where it can be found, for example[http://192.168.99.100:5601](http://192.168.99.100:5601/).

LogSpout is started on all swarm nodes and the log traffic is directed to LogStash running on infra node. If the script is run again, it checks if the required containers are running on all servers and starts them as needed, for example after a new swarm node has been added.

**Stopping the log aggregation**

LogBox and all the LogSpout instances can be removed with the script

|  |  |
| --- | --- |
| 1 | scripts/rmLogging.sh |

**Add monitoring**

Prometheus based monitoring and the corresponding cAdvisor containers can be started with

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ scripts/addMonitoring.sh  \*\* Servers in the swarm: swarm-1 swarm-0 \*\*  \*\* Starting cAdvisor on swarm-1  3f6a32670668d0a54d91df02d6107fc4a8225c3fb2f4637045622130a3b1ed83  \*\* Starting cAdvisor on swarm-0  8a379f5385b9cc9b99d8eadf473bc647db5c313f1df17e1536199b193f0a668d  \*\* Starting Prometheus on infra \*\*  aff2bb500671d7b81d91331845abdb61466f4638888cc552c299869a728a3010  Prometheus ui can be found at http://192.168.99.100:9090 |

Then the cAdvisor instances can be directly accessed from port 8080, for example[http://192.168.99.101:8080](http://192.168.99.101:8080/) and the Prometheus UI can be accessed from the infra server at port 9090, for example [http://192.168.99.100:9090](http://192.168.99.100:9090/).

**Using Consul for local service discovery**

One of the functions in docker-functions.sh can also return an external IP for a container running a given image. Remember to point the docker client to swarm master.

|  |  |
| --- | --- |
| 1  2  3  4 | $ source scripts/docker-functions.sh  $ dock --swarm swarm-0  $ dockip rest  192.168.99.102 |

Things can be made even more simple by joining a local Consul to the Consul that is running on infra node and using it as the local DNS server.

**Setting up resolver configuration**

OS X can use resolver to enable Consul DNS based service discovery locally. To set it up (following this[blog](http://passingcuriosity.com/2013/dnsmasq-dev-osx/)) the following steps are needed:

Create directory /etc/resolver

|  |  |
| --- | --- |
| 1 | sudo mkdir -p /etc/resolver |

Create a file called /etc/resolver/consul with the contents:

|  |  |
| --- | --- |
| 1  2 | nameserver 127.0.0.1  port 8600 |

Now the domain .consul is resolved with the DNS server running locally on port 8600 which is the default port for Consul DNS.

**Starting a local Consul and joining it to infra Consul**

First you need to install Consul locally, which can be done through [brew](http://brew.sh/) with brew install consul. Then you can start a local Consul instance that joins to the Consul server running on infra node with

|  |  |
| --- | --- |
| 1 | consul agent -data-dir=/tmp/consul -join $(docker-machine ip infra) |

**Testing the discovery**

Now the HAProxy based endpoint should we reachable at [http://rest.service.consul](http://rest.service.consul/) and the example service should reply

|  |  |
| --- | --- |
| 1  2 | $ curl rest.service.consul/hello/v1  {"hostname":"e736179c1932","time":"2015-07-28T10:07:05.433Z","language":"javascript"} |

**Scripts and files**

In this section I’ll explain the different scripts quickly. The scripts directory looks like

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ ls -lF  total 72  -rwxr--r-- 1 ilkka.anttonen 562225435 1449 Jul 27 14:56 addLogging.sh\*  -rwxr--r-- 1 ilkka.anttonen 562225435 2220 Jul 28 10:23 addMonitoring.sh\*  -rwxr--r-- 1 ilkka.anttonen 562225435 727 Jul 28 10:41 createInfraNode.sh\*  -rwxr--r-- 1 ilkka.anttonen 562225435 1841 Jul 27 13:55 createSwarmNode.sh\*  -rwxr--r-- 1 ilkka.anttonen 562225435 1228 Jul 27 14:54 docker-functions.sh\*  -rw-r--r-- 1 ilkka.anttonen 562225435 984 Jul 28 10:23 prometheus.yml  -rwxr--r-- 1 ilkka.anttonen 562225435 286 Jul 27 14:24 rmLogging.sh\*  -rwxr--r-- 1 ilkka.anttonen 562225435 286 Jul 27 15:30 rmMonitoring.sh\* |

**docker-functions.sh**

This file contains functions that make life with containers in a swarm easier. The functions are also used in the other scripts to prevent duplication of code. Easiest way is to run source docker-functions.sh.

**dock**

This is a shortcut that is used to point the docker command line client to a given docker-machine controlled node. For example to point to infra the command is dock infra and to point to swarm-master the command is dock --swarm swarm-master.

**dockpsi**

This function returns ps information for all containers running a given image. For example to query all the containers on a swarm that run an image containing the string ‘node’:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ dockpsi node  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  f495c4fbe5b8 sirile/node-test "node /app.js" 2 hours ago Up 2 hours 192.168.99.102:32778->80/tcp swarm-app-1/dreamy\_euclid  e8d148835aab sirile/node-test "node /app.js" 2 hours ago Up 2 hours 192.168.99.101:32777->80/tcp swarm-master/suspicious\_yalow  86cb5b75ef42 sirile/node-test "node /app.js" 2 hours ago Up 2 hours 192.168.99.102:32777->80/tcp swarm-app-1/cranky\_feynman  50f9e99d8144 sirile/node-test "node /app.js" 2 hours ago Up 2 hours 192.168.99.101:32776->80/tcp swarm-master/insane\_lalande  f8942be4f752 sirile/node-test "node /app.js" 2 hours ago Up 2 hours 192.168.99.102:32776->80/tcp swarm-app-1/pensive\_ptolemy |

**dockrm**

This function removes all the instances of a given image. For example if the image is sirile/node-test you can run dockrm node to remove all running containers that run that image.

**addLogging.sh and addMonitoring.sh**

These scripts check that the infra node is running, then start the main containers there. LogBox for logging and Prometheus for monitoring. Then they loop through the swarm nodes (with a bit of awk and xargs trickery) and start the required services there if they are not already running. In the case ofaddMonitoring.sh, it also rewrites the Prometheus configuration file prometheus.yml and sends a SIGHUP to the running Prometheus container if it exists so that the configuration is reloaded.

**createInfraNode.sh and createSwarmNode.sh**

These scripts create the infra node and swarm nodes (including the master) respectively. ThecreateSwarmNode.sh checks that infra node is running and if it isn’t it errors out. It also checks that swarm-master has been created if application node is being created.

**rmLogging.sh and rmMonitoring.sh**

These scripts remove containers that take care of logging or monitoring.

**startService.sh**

This script is used to start services. It checks that at least one instance of HAProxy image is running and starts it if needed.

**Comments**

Want to leave a comment? Visit [this post's issue page on GitHub](https://github.com/sirile/sirile.github.io/issues/9) (you'll need a GitHub account).

# 2@Advanced Service Discovery in Prometheus 0.14.0 | Prometheus

https://prometheus.io/blog/2015/06/01/advanced-service-discovery/

Posted at: June 1, 2015 by Fabian Reinartz, Julius Volz

This week we released Prometheus v0.14.0 — a version with many long-awaited additions and improvements.

On the user side, Prometheus now supports new service discovery mechanisms. In addition to DNS-SRV records, it now supports [Consul](https://www.consul.io/) out of the box, and a file-based interface allows you to connect your own discovery mechanisms. Over time, we plan to add other common service discovery mechanisms to Prometheus.

Aside from many smaller fixes and improvements, you can now also reload your configuration during runtime by sending a SIGHUP to the Prometheus process. For a full list of changes, check the [changelog for this release](https://github.com/prometheus/prometheus/blob/master/CHANGELOG.md#0140--2015-06-01).

In this blog post, we will take a closer look at the built-in service discovery mechanisms and provide some practical examples. As an additional resource, see [Prometheus's configuration documentation](https://prometheus.io/docs/operating/configuration).

**Prometheus and targets**

For a proper understanding of this blog post, we first need to take a look at how Prometheus labels targets.

There are various places in the configuration file where target labels may be set. They are applied in the following order, with later stages overwriting any labels set by an earlier stage:

1. Global labels, which are assigned to every target scraped by the Prometheus instance.
2. The job label, which is configured as a default value for each scrape configuration.
3. Labels that are set per target group within a scrape configuration.
4. Advanced label manipulation via [relabeling](https://prometheus.io/docs/operating/configuration/#relabel_config).

Each stage overwrites any colliding labels from the earlier stages. Eventually, we have a flat set of labels that describe a single target. Those labels are then attached to every time series that is scraped from this target.

Note: Internally, even the address of a target is stored in a special \_\_address\_\_ label. This can be useful during advanced label manipulation (relabeling), as we will see later. Labels starting with \_\_ do not appear in the final time series.

**Scrape configurations and relabeling**

Aside from moving from an ASCII protocol buffer format to YAML, a fundamental change to Prometheus's configuration is the change from per-job configurations to more generalized scrape configurations. While the two are almost equivalent for simple setups, scrape configurations allow for greater flexibility in more advanced use cases.

Each scrape configuration defines a job name which serves as a default value for the job label. The job label can then be redefined for entire target groups or individual targets. For example, we can define two target groups, each of which defines targets for one job. To scrape them with the same parameters, we can configure them as follows:

scrape\_configs:

- job\_name: 'overwritten-default'

scrape\_interval: 10s

scrape\_timeout: 5s

target\_groups:

- targets: ['10.1.200.130:5051', '10.1.200.134:5051']

labels:

job: 'job1'

- targets: ['10.1.200.130:6220', '10.1.200.134:6221']

labels:

job: 'job2'

Through a mechanism named [relabeling](http://prometheus.io/docs/operating/configuration/#relabel_config), any label can be removed, created, or modified on a per-target level. This enables fine-grained labeling that can also take into account metadata coming from the service discovery. Relabeling is the last stage of label assignment and overwrites any labels previously set.

Relabeling works as follows:

* A list of source labels is defined.
* For each target, the values of those labels are concatenated with a separator.
* A regular expression is matched against the resulting string.
* A new value based on those matches is assigned to another label.

Multiple relabeling rules can be defined for each scrape configuration. A simple one that squashes two labels into one, looks as follows:

relabel\_configs:

- source\_labels: ['label\_a', 'label\_b']

separator: ';'

regex: '(.\*);(.\*)'

replacement: '${1}-${2}'

target\_label: 'label\_c'

This rule transforms a target with the label set:

{

"job": "job1",

"label\_a": "foo",

"label\_b": "bar"

}

...into a target with the label set:

{

"job": "job1",

"label\_a": "foo",

"label\_b": "bar",

"label\_c": "foo-bar"

}

You could then also remove the source labels in an additional relabeling step.

You can read more about relabeling and how you can use it to filter targets in the [configuration documentation](https://prometheus.io/docs/operating/configuration#relabel_config).

Over the next sections, we will see how you can leverage relabeling when using service discovery.

**Discovery with DNS-SRV records**

Since the beginning, Prometheus has supported target discovery via DNS-SRV records. The respective configuration looked like this:

job {

name: "api-server"

sd\_name: "telemetry.eu-west.api.srv.example.org"

metrics\_path: "/metrics"

}

Prometheus 0.14.0 allows you to specify multiple SRV records to be queried in a single scrape configuration, and also provides service-discovery-specific meta information that is helpful during the relabeling phase.

When querying the the DNS-SRV records, a label named \_\_meta\_dns\_srv\_name is attached to each target. Its value is set to the SRV record name for which it was returned. If we have structured SRV record names liketelemetry.<zone>.<job>.srv.example.org, we can extract relevant labels from it those names:

scrape\_configs:

- job\_name: 'myjob'

dns\_sd\_configs:

- names:

- 'telemetry.eu-west.api.srv.example.org'

- 'telemetry.us-west.api.srv.example.org'

- 'telemetry.eu-west.auth.srv.example.org'

- 'telemetry.us-east.auth.srv.example.org'

relabel\_configs:

- source\_labels: ['\_\_meta\_dns\_srv\_name']

regex: 'telemetry\.(.+?)\..+?\.srv\.example\.org'

target\_label: 'zone'

replacement: '$1'

- source\_labels: ['\_\_meta\_dns\_srv\_name']

regex: 'telemetry\..+?\.(.+?)\.srv\.example\.org'

target\_label: 'job'

replacement: '$1'

This will attach the zone and job label to each target based on the SRV record it came from.

**Discovery with Consul**

Service discovery via Consul is now supported natively. It can be configured by defining access parameters for our Consul agent and a list of Consul services for which we want to query targets.

The tags of each Consul node are concatenated by a configurable separator and exposed through the\_\_meta\_consul\_tags label. Various other Consul-specific meta labels are also provided.

Scraping all instances for a list of given services can be achieved with a simple consul\_sd\_config and relabeling rules:

scrape\_configs:

- job\_name: 'overwritten-default'

consul\_sd\_configs:

- server: '127.0.0.1:5361'

services: ['auth', 'api', 'load-balancer', 'postgres']

relabel\_configs:

- source\_labels: ['\_\_meta\_consul\_service']

regex: '(.\*)'

target\_label: 'job'

replacement: '$1'

- source\_labels: ['\_\_meta\_consul\_node']

regex: '(.\*)'

target\_label: 'instance'

replacement: '$1'

- source\_labels: ['\_\_meta\_consul\_tags']

regex: ',(production|canary),'

target\_label: 'group'

replacement: '$1'

This discovers the given services from the local Consul agent. As a result, we get metrics for four jobs (auth,api, load-balancer, and postgres). If a node has the production or canary Consul tag, a respective grouplabel is assigned to the target. Each target's instance label is set to the node name provided by Consul.

A full documentation of all configuration parameters for service discovery via Consul can be found on the[Prometheus website](https://prometheus.io/docs/operating/configuration#relabel_config).

**Custom service discovery**

Finally, we added a file-based interface to integrate your custom service discovery or other common mechanisms that are not yet supported out of the box.

With this mechanism, Prometheus watches a set of directories or files which define target groups. Whenever any of those files changes, a list of target groups is read from the files and scrape targets are extracted. It's now our job to write a small bridge program that runs as Prometheus's side-kick. It retrieves changes from an arbitrary service discovery mechanism and writes the target information to the watched files as lists of target groups.

These files can either be in YAML:

- targets: ['10.11.150.1:7870', '10.11.150.4:7870']

labels:

job: 'mysql'

- targets: ['10.11.122.11:6001', '10.11.122.15:6002']

labels:

job: 'postgres'

...or in JSON format:

[

{

"targets": ["10.11.150.1:7870", "10.11.150.4:7870"],

"labels": {

"job": "mysql"

}

},

{

"targets": ["10.11.122.11:6001", "10.11.122.15:6002"],

"labels": {

"job": "postgres"

}

}

]

We now configure Prometheus to watch the tgroups/ directory in its working directory for all .json files:

scrape\_configs:

- job\_name: 'overwritten-default'

file\_sd\_configs:

- names: ['tgroups/\*.json']

What's missing now is a program that writes files to this directory. For the sake of this example, let's assume we have all our instances for different jobs in a single denormalized MySQL table. (Hint: you probably don't want to do service discovery this way.)

Every 30 seconds, we read all instances from the MySQL table and write the resulting target groups into a JSON file. Note that we do not have to keep state whether or not any targets or their labels have changed. Prometheus will automatically detect changes and applies them to targets without interrupting their scrape cycles.

import os, time, json

from itertools import groupby

from MySQLdb import connect

def refresh(cur):

# Fetch all rows.

cur.execute("SELECT address, job, zone FROM instances")

tgs = []

# Group all instances by their job and zone values.

for key, vals in groupby(cur.fetchall(), key=lambda r: (r[1], r[2])):

tgs.append({

'labels': dict(zip(['job', 'zone'], key)),

'targets': [t[0] for t in vals],

})

# Persist the target groups to disk as JSON file.

with open('tgroups/target\_groups.json.new', 'w') as f:

json.dump(tgs, f)

f.flush()

os.fsync(f.fileno())

os.rename('tgroups/target\_groups.json.new', 'tgroups/target\_groups.json')

if \_\_name\_\_ == '\_\_main\_\_':

while True:

with connect('localhost', 'root', '', 'test') as cur:

refresh(cur)

time.sleep(30)

While Prometheus will not apply any malformed changes to files, it is considered best practice to update your files atomically via renaming, as we do in our example. It is also recommended to split larger amounts of target groups into several files based on logical grouping.

**Conclusion**

With DNS-SRV records and Consul, two major service discovery methods are now natively supported by Prometheus. We've seen that relabeling is a powerful approach to make use of metadata provided by service discovery mechanisms.

Make sure to take a look at the new [configuration documentation](https://prometheus.io/docs/operating/configuration/) to upgrade your Prometheus setup to the new release and find out about other configuration options, such as basic HTTP authentication and target filtering via relabeling.

We provide a [migration tool](https://github.com/prometheus/migrate/releases) that upgrades your existing configuration files to the new YAML format. For smaller configurations we recommend a manual upgrade to get familiar with the new format and to preserve comments.

# 利用prometheus, Grafana, postgres\_exporter 搭建简单简单监控 - MtrS的个人页面

https://my.oschina.net/innovation/blog/829707

prometheus是一个优秀的服务树监控组件， Grafana 是一个图形化展示界面 postgres\_exporter 是一个postgres服务器状态收集组件，服务于prometheus, prometheus 官方提供了mysql的服务器状态的手机组件，postgres的却没有提供，好在拥有万能的github

<https://github.com/wrouesnel/postgres_exporter>

安装流程:

**安装prometheus**

下载prometheus, 可以到[官网](https://prometheus.io/download/)下载编译好的二进制文件,也可以到github 下载源码自己编译:

wget https://github.com/prometheus/prometheus/releases/download/v1.4.1/prometheus-1.4.1.linux-amd64.tar.gz

#编辑配置文件

nvim prometheus.yml

下面的文件为我的配置文件，第二个job 为postgres的状态收集任务.

#cat prometheus.yml

# cd ../prometheus-1.4.1.linux-amd64

16:56:17  ⚡  /opt/pkg/prometheus-1.4.1.linux-amd64 

# cat prometheus.yml

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Attach these labels to any time series or alerts when communicating with

# external systems (federation, remote storage, Alertmanager).

external\_labels:

monitor: 'codelab-monitor'

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first.rules"

# - "second.rules"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'postgres'

static\_configs:

- targets: ['127.0.0.1:9187']

labels:

instance: db1

启动prometheus

./prometheus

**安装postgres\_exporter**

github 上有几个组件，自己可以任选几个

[包括慢查询,表，数据库,buffers 的统计信息](https://github.com/mc2soft/postgresql_exporter) 个人选用的为:

<https://github.com/wrouesnel/postgres_exporter>

包括buffers,rows 统计等的信息，同样可以自行编译或者使用github release的二进制

wget https://github.com/wrouesnel/postgres\_exporter/releases/download/v0.1.1/postgres\_exporter

运行时命令行，最后一个参数log.level 可以不配置, queries.yaml 使用源码里面的 [queries.yaml](https://raw.githubusercontent.com/wrouesnel/postgres_exporter/master/queries.yaml)

需在监控的数据库上执行一下以下命令(不使用数据库超级管理员的权限的时候,注意更改密码),

CREATE USER postgres\_exporter PASSWORD 'password';

ALTER USER postgres\_exporter SET SEARCH\_PATH TO postgres\_exporter,pg\_catalog;

CREATE SCHEMA postgres\_exporter AUTHORIZATION postgres\_exporter;

CREATE FUNCTION postgres\_exporter.f\_select\_pg\_stat\_activity()

RETURNS setof pg\_catalog.pg\_stat\_activity

LANGUAGE sql

SECURITY DEFINER

AS $$

SELECT \* from pg\_catalog.pg\_stat\_activity;

$$;

CREATE FUNCTION postgres\_exporter.f\_select\_pg\_stat\_replication()

RETURNS setof pg\_catalog.pg\_stat\_replication

LANGUAGE sql

SECURITY DEFINER

AS $$

SELECT \* from pg\_catalog.pg\_stat\_replication;

$$;

CREATE VIEW postgres\_exporter.pg\_stat\_replication

AS

SELECT \* FROM postgres\_exporter.f\_select\_pg\_stat\_replication();

CREATE VIEW postgres\_exporter.pg\_stat\_activity

AS

SELECT \* FROM postgres\_exporter.f\_select\_pg\_stat\_activity();

GRANT SELECT ON postgres\_exporter.pg\_stat\_replication TO postgres\_exporter;

GRANT SELECT ON postgres\_exporter.pg\_stat\_activity TO postgres\_exporter;

监控

#配置 数据库数据源信息

export DATA\_SOURCE\_NAME=postgresql://postgres:postgres@127.0.0.1:5433/postgres?sslmode=disable

#启动监控命令

./postgres\_exporter -extend.query-path queries.yaml -log.level debug

该程序会开启服务器的 :9187 端口

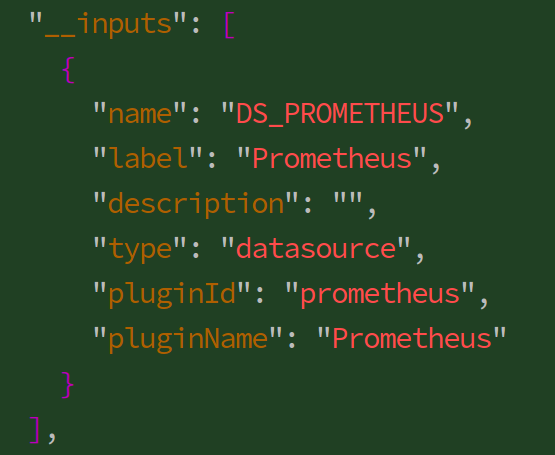
**安装grafana**

下载grafana

下载postgres\_expoter的 的dasboard

wget https://grafana.net/api/dashboards/455/revisions/1/download

但是该模板存在小问题， 需要编辑 原先的 \_\_inputs -> name -> 是不对的



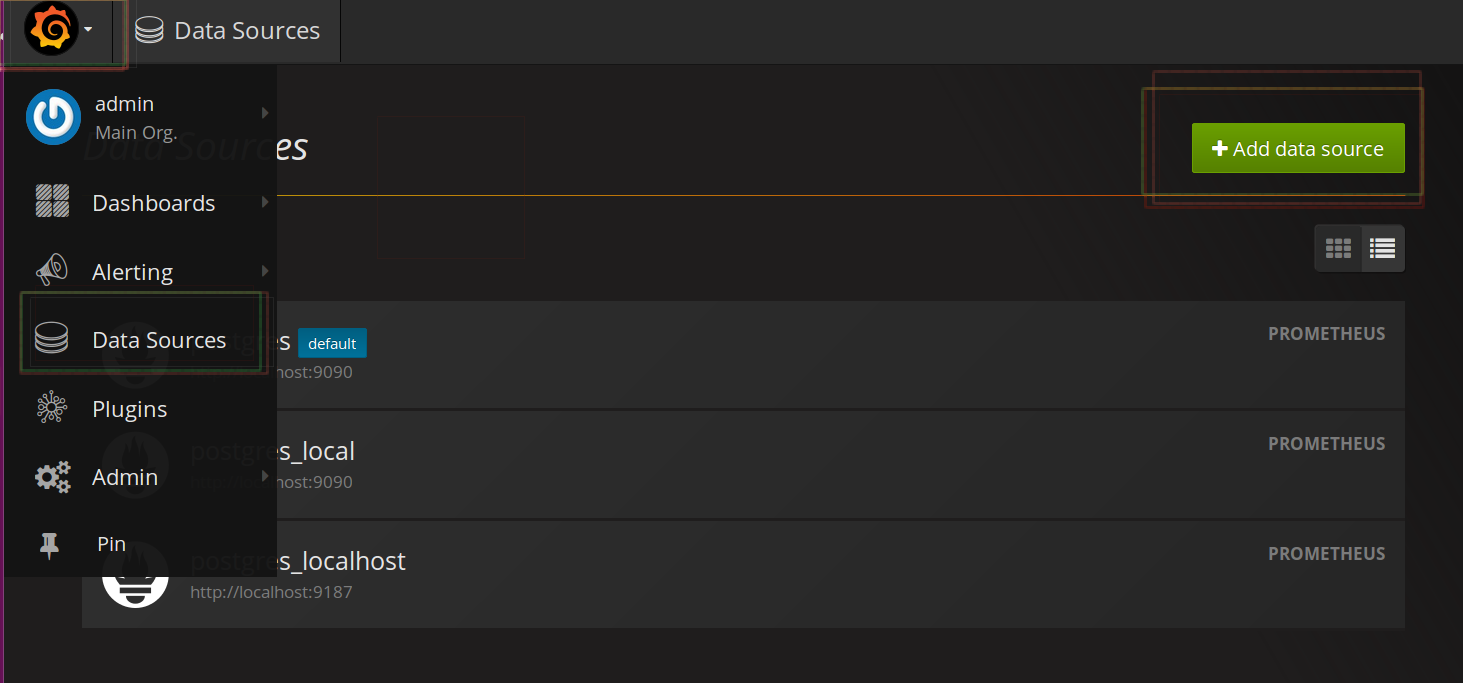
启动grafana

https://grafana.net/api/dashboards/455/revisions/1/download

进入grafana 的页面 为<http://ip:3000/>

默认用户名，密码分别为 admin admin

点击左侧 data sources, -> add source



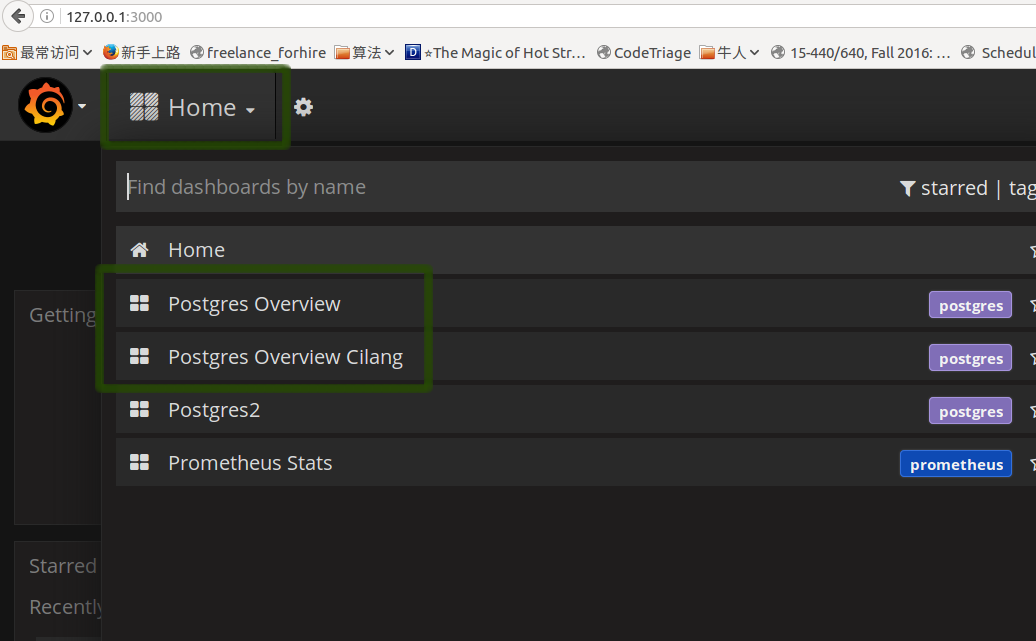
填写 prometheus 的主机端口

postgres\_local

[http://127.0.0.1:9090](http://127.0.0.1:9090/)

选择dashboard -> import -> 导入刚才下载的 postgres\_exporter

在host 选项中填写第一步 prometheus 中job instance 中的主机参数，就可以拥有基本的图形话界面了。





# 利用 Prometheus 监控 NodeJS 应用 | HiMySQL - leopku 润物细无声

http://www.himysql.com/post/monitoring-node-apps-with-prometheus/

**利用 Prometheus 监控 NodeJS 应用**

Posted on Tue, Mar 28, 2017 | [#NodeJS](http://www.himysql.com/tags/nodejs) [#Prometheus](http://www.himysql.com/tags/prometheus) [#monitor](http://www.himysql.com/tags/monitor)

Unless otherwise indicated, the text of documents in this site is available under the [Creative Commons Attribution 3.0 Unported License](https://creativecommons.org/licenses/by/3.0/deed), or any later version. Copyright 2009 - 2017 leopku.

[「欢快的搭建好 Prometheus」](https://askmike.org/articles/setting-up-prometheus-on-ubuntu-16-04/)之后，记录我们 nodejs 应用的指标就是一件更欢快的事了。

脑洞主要是在应用中引入一个外部的 node 模块，这个模块提供了对 Prometheus [指标类型(metric types)](https://prometheus.io/docs/concepts/metric_types/)的支持。本文中用到的是 Counter 和 Gauge 。这个模块通过一个轻量的 express 服务暴露统计数据，以便 Prometheus 来抓取。

**记录应用数据**

每个应用都可能有许多有意思的指标。比如我要对 [Realtime Bitcoin Globe](https://blocks.wizb.it/) 添加 websocket 服务，它将关于比特币交易消息中继到连接的 web 客户端。通过 Prometheus 我们可以很容易获取下面的指标：

* 服务发送了多少个 websocket 消息？
* 服务有多少活跃的 websocket 连接？
* 新建链接？
* 内存占用？

这些统计必须通过指标来实现。同时记录的指标包括：中继延迟（从接收发送到Websocket客户端的事务需要多长时间？）、交易细节（记录经过系统的交易）或者数据库查询（执行时长、查询类型、缓存命中）等。记录的内容无尽可能。

通过 Prometheus 类似的稳固的监控系统，可以更直观地了解应用当前的状态。这些（收集来的）数据在出状况、衡量某项优化效果（或想找出可优化点）时帮你大忙。

**干货**

这里采用的是一个很轻量的 NPM 包 [prometheus-client](http://npmjs.com/package/prometheus-client) ，刚好能满足我们的需求。

**monitor.js**

var Prometheus = require('prometheus-client');

client = new Prometheus();

module.exports = {

connections: client.newCounter({

namespace: 'blocks\_wizbit',

name: 'new\_connections',

help: 'The number of new connections.'

}),

messages: client.newCounter({

namespace: 'blocks\_wizbit',

name: 'send\_ws\_messages',

help: 'The number of send websocket messages.'

}),

connectedUsers: client.newGauge({

namespace: 'blocks\_wizbit',

name: 'active\_connections',

help: 'The number of active connections.'

}),

memoryUsage: client.newGauge({

namespace: 'blocks\_wizbit',

name: 'memory',

help: 'The current used memory.'

})

}

client.listen(9095);

最后一行通过模块自带的 express 服务监听在 9095 端口。

现在可以在任何想要监控的地方引入 monitor.js，比如在处理所有 websocket 服务逻辑的地方：

const monitor = require('path/to/monitor.js');

server.on('connection', e => {

monitor.connections.increment();

// 其它逻辑

});

再比如监控活跃用户：

setInterval(() => {

monitor.connectedUsers.set({

period: '10sec'

}, server.connections());

}, 10000);

**创建图表**

前面用了 promdash （看[如何安装](https://askmike.org/articles/setting-up-prometheus-on-ubuntu-16-04/#configuring-promdash)博客），留意其中查询的部分：

服务发送了多少个 websocket 消息（每秒）？

rate(blocks\_wizbit\_send\_ws\_messages[1m])

有多少个活跃 websocket 连接？

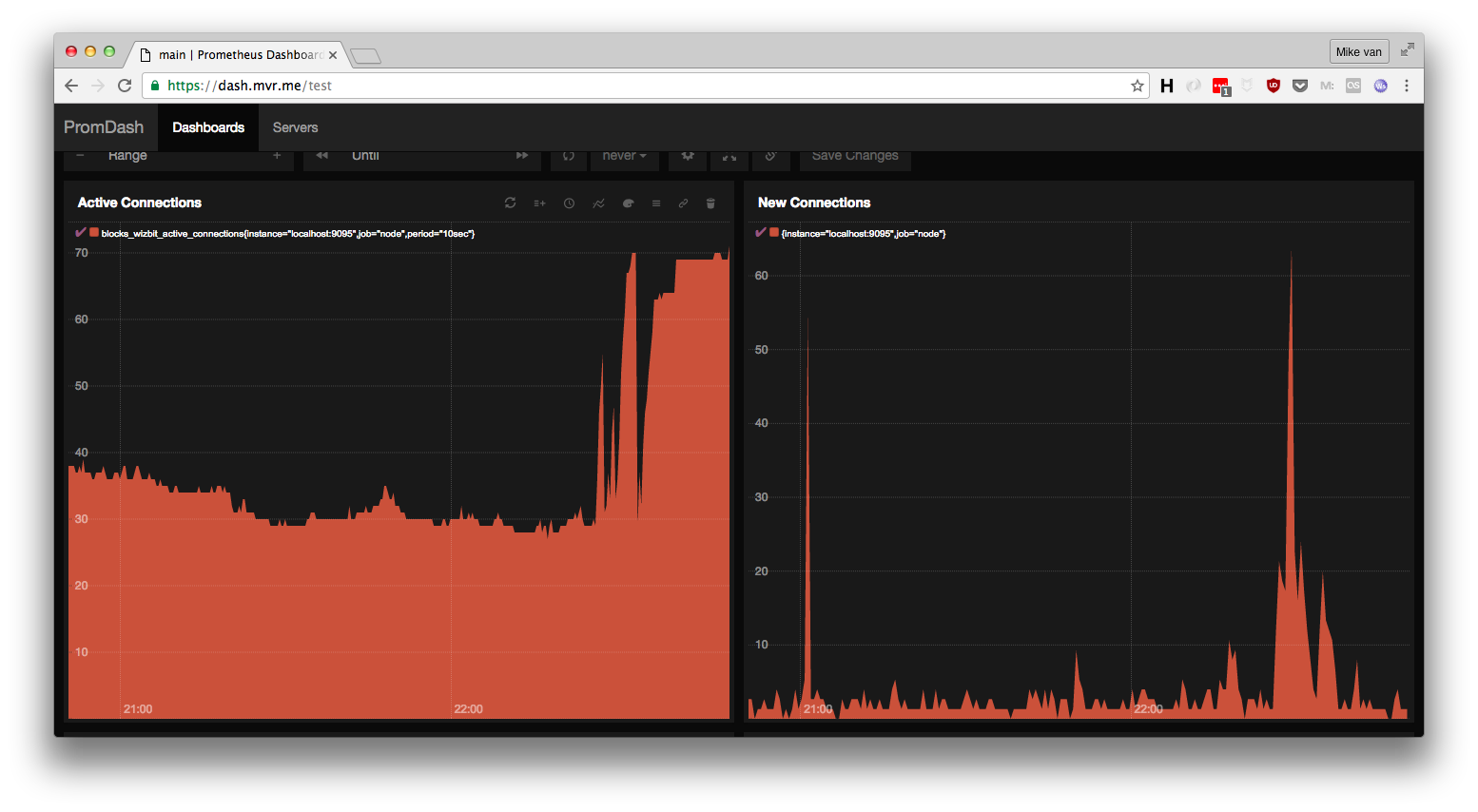
blocks\_wizbit\_active\_connections

新建了多少连接（每分钟）？

rate(blocks\_wizbit\_new\_connections[1m]) \* 60

内存占用（单位 mb）

blocks\_wizbit\_memory / 1024 / 1024

最终图表长这样： 

*本文根据*[*askmike*](mailto:askmike@mvr.me)*的《Monitoring nodejs apps with Prometheus》所译，整个译文带有我们自己的理解与思想，如果译得不好或有不对之处还请同行朋友指点。如需转载此译文，需注明英文出处：*[*https://askmike.org/articles/monitoring-node-apps-with-prometheus/*](https://askmike.org/articles/monitoring-node-apps-with-prometheus/)*。*