8月20号:

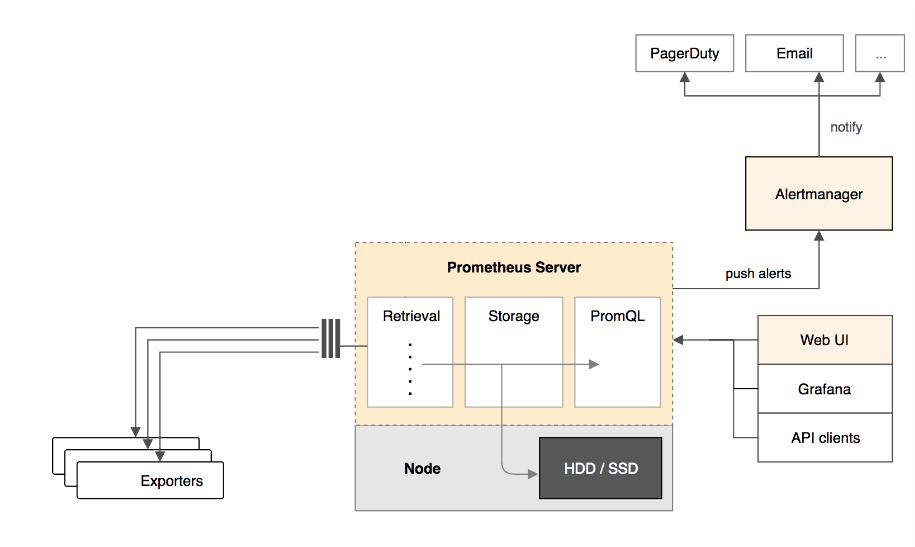
容器集群监控调研报告

报告人:刘雨生

需要完成任务：

嗯，我们产品不用granfa，需要知道有哪些指标，基本的有容器和节点的cpu，内存，磁盘，网络，pod状态等，proemtheus怎么获取这些指标值，并提供查询的api，然后prometheus监控怎么告警。

### 普罗米修斯架构图：



**Prometheus Server**

Prometheus Server 负责从 Exporter 拉取和存储监控数据，并提供一套灵活的查询语言（PromQL）供用户使用。

**Exporter**

Exporter 负责收集目标对象（host, container…）的性能数据，并通过 HTTP 接口供 Prometheus Server 获取。

**可视化组件**

监控数据的可视化展现对于监控方案至关重要。以前 Prometheus 自己开发了一套工具，不过后来废弃了，因为开源社区出现了更为优秀的产品 Grafana。Grafana 能够与 Prometheus 无缝集成，提供完美的数据展示能力。

**Alertmanager**

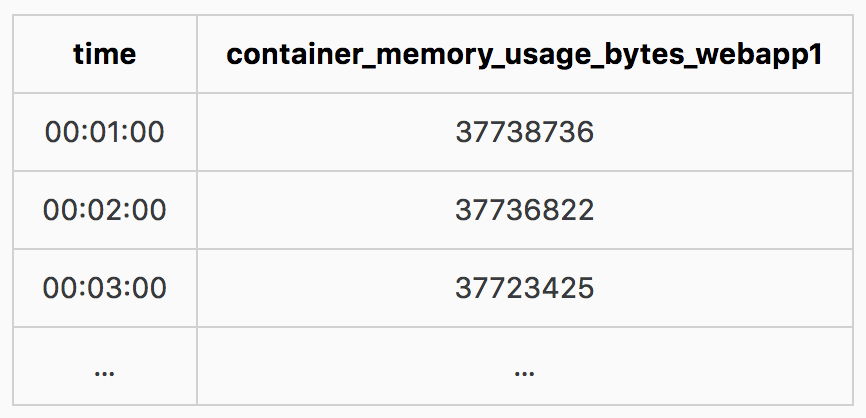
用户可以定义基于监控数据的告警规则，规则会触发告警。一旦 Alermanager 收到告警，会通过预定义的方式发出告警通知。支持的方式包括 Email、PagerDuty、Webhook 等.

其他监控方案参考

Zabbix、Graphite、Nagios

**普罗米修斯监控数据存储方案:**

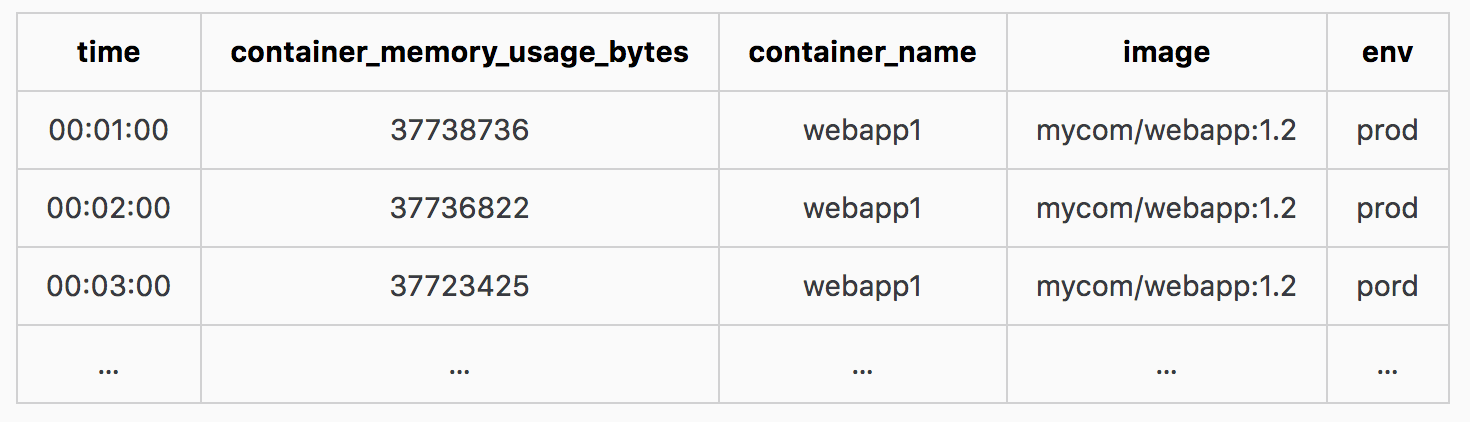
**传统数据采集方案。**



**Prometheus 的解决方案。**

Prometheus 只需要定义一个全局的指标 container\_memory\_usage\_bytes，然后通过添加不同的维度数据来满足不同的业务需求。

比如对于前面 webapp1 的三条取样数据，转换成 Prometheus 多维数据将变成：



**Prometheu数据模型分析**:

后面三列 container\_name、image、env 就是数据的三个维度。

想象一下，如果不同 env（prod、test、dev），不同 image（mycom/webapp:1.2、mycom/webapp:1.3）的容器，它们的内存使用数据中标注了这三个维度信息，那么将能满足很多业务需求，比如：

计算 webapp2 的平均内存使用情况：avg(container\_memory\_usage\_bytes{container\_name=“webapp2”})

计算运行 mycom/webapp:1.3 镜像的所有容器内存使用总量：sum(container\_memory\_usage\_bytes{image=“mycom/webapp:1.3”})

**Prometheus 数据模型的优势：**

通过维度对数据进行说明，附加更多的业务信息，进而满足不同业务的需求。同时维度是可以动态添加的，比如再给数据加上一个 user 维度，就可以按用户来统计容器内存使用量了。

Prometheus 丰富的查询语言能够灵活、充分地挖掘数据的价值。前面示例中的 avg、sum、by 只是查询语言中很小的一部分功能，已经为我们展现了 Prometheus 对多维数据进行分片、聚合的强大能力。

**Prometheus 容器集群快速搭建：（基于docker分布式容器进行部署方案）**

**组件部署分析:**

**Prometheus Server 整个监控杆系统大脑**

**Grafana 系统数据展示**

**cAdvisor 系统数据采集**

**Prometheus Server**

Prometheus Server 本身也将以容器的方式运行在master 上

**Exporter**

我们将使用：

Node Exporter，负责收集 host 硬件和操作系统数据。它将以容器方式运行在所有 host 上。

cAdvisor，负责收集容器数据。(它将以容器方式运行在所有 host 上)

**Grafana**

显示多维数据，Grafana 本身也将以容器方式运行在 host 192.168.56.103 上.

步骤1:

**运行 Node Exporter(每一个监控节点都需要运行以下命令)**

**#docker run -d -p 9100:9100 \**

**-v "/proc:/host/proc" \**

**-v "/sys:/host/sys" \**

**-v "/:/rootfs" \**

**prom/node-exporter \**

**--path.procfs /host/proc \**

**--path.sysfs /host/sys \**

**--collector.filesystem.ignored-mount-points "^/(sys|proc|dev|host|etc)($|/)"**

**--net=host \ 参数可选**

**验证:**

**通过访问**Node Exporter:9100 进行数据验证

步骤2:

**运行 cAdvisor**

**#docker run \**

**--volume=/:/rootfs:ro \**

**--volume=/var/run:/var/run:rw \**

**--volume=/sys:/sys:ro \**

**--volume=/var/lib/docker/:/var/lib/docker:ro \**

**--publish=8080:8080 \**

**--detach=true \**

**--name=cadvisor \**

**google/cadvisor:版本号**

**--net=host \ 参数可选**

**验证:**

**通过访问**Node Exporter:8080 进行数据验证

步骤3:

**运行 Prometheus Server（主节点运行普罗米修斯）**

**#docker run -d -p 9090:9090 \**

**-v /root/prometheus.yml:/etc/prometheus/prometheus.yml \**

**--name prometheus \**

**--net=host \**

**prom/prometheus**

**--net=host \ 参数可选**

这样 Prometheus Server 可以直接与 Exporter 和 Grafana 通信。

prometheus.yml 是 Prometheus Server 的配置文件，需要作出的修改是

**最重要的配置是：**

static\_configs:

-targets:['localhost:9090','localhost:8080','localhost:9100','192.168.56.102:8080','192.168.56.102:9100']

验证:需要通过访问master:9090 查看返回的数据和界面

步骤4:

**运行 Grafana（主节点运行普罗米修斯）**

docker run -d -i -p 3000:3000 \

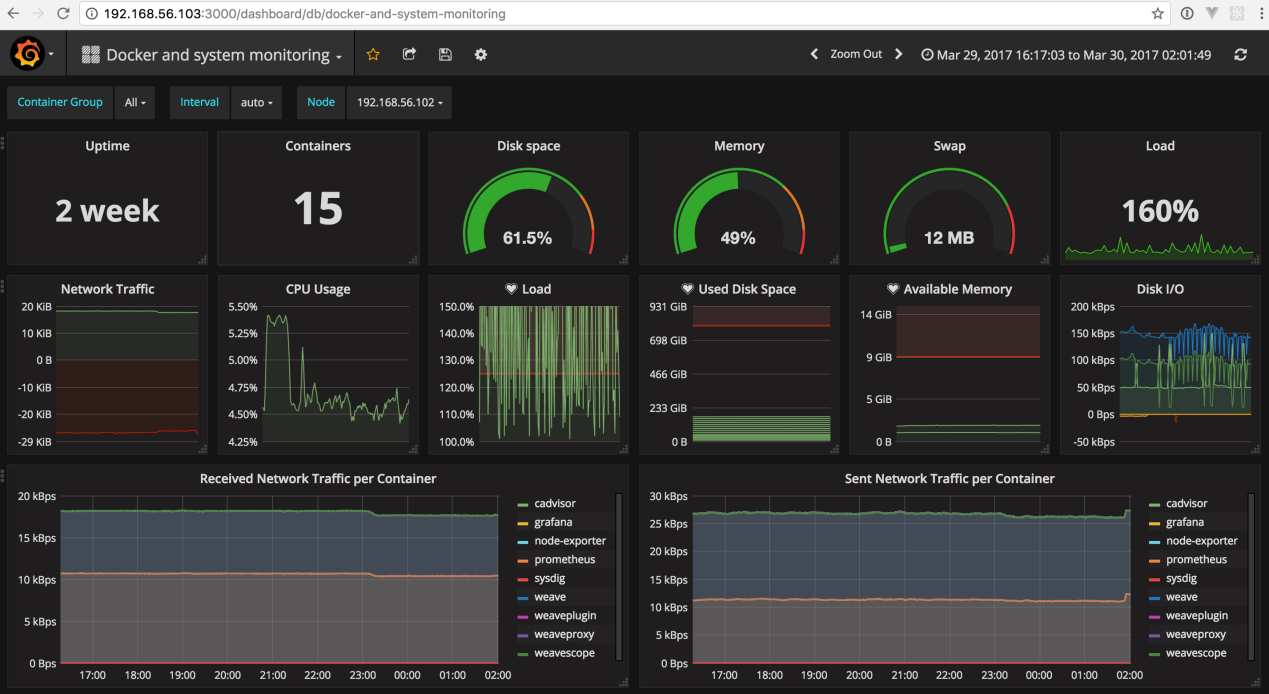
-e "GF\_SERVER\_ROOT\_URL=http://grafana.server.name" \

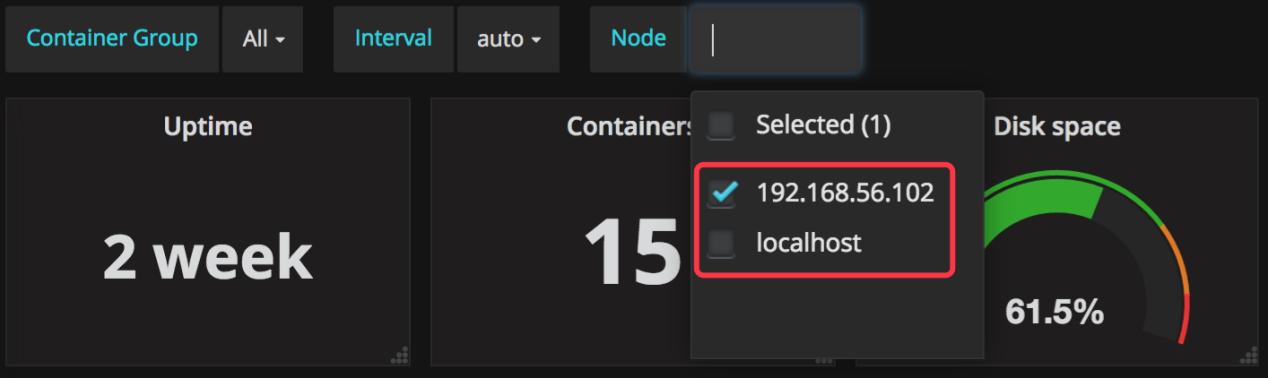
-e "GF\_SECURITY\_ADMIN\_PASSWORD=secret" \

grafana/grafana

**--net=host \ 参数可选**

验证:需要通过访问master:3000 配置界面



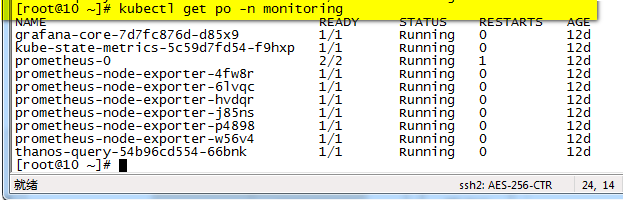


监控数据类型:

时间 cpu 内存 swap pod node 还有不少信息，我订制了比较少的信息展示。

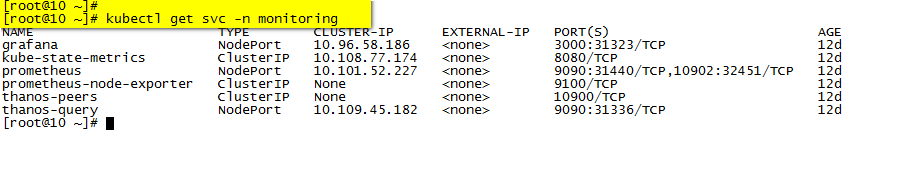
使用命令查看监控的情况:

#kubectl get pod -n monitoring



使用容器地开放端口和状态信息：

[root@10 ~]# kubectl get svc -n monitoring



系统对应地开放端口:（目前并未完全关联起来）

Grafana 10.96.58.186 3000:31323

Prometheus 10.101.52.227 9090:31440,10902:32451

Prometheus-node-exporter none 9100

Thanos-peers none 10900

Thanos-query 10.109.45.182 9090:31336

容器云平台监控工具方案的比较:

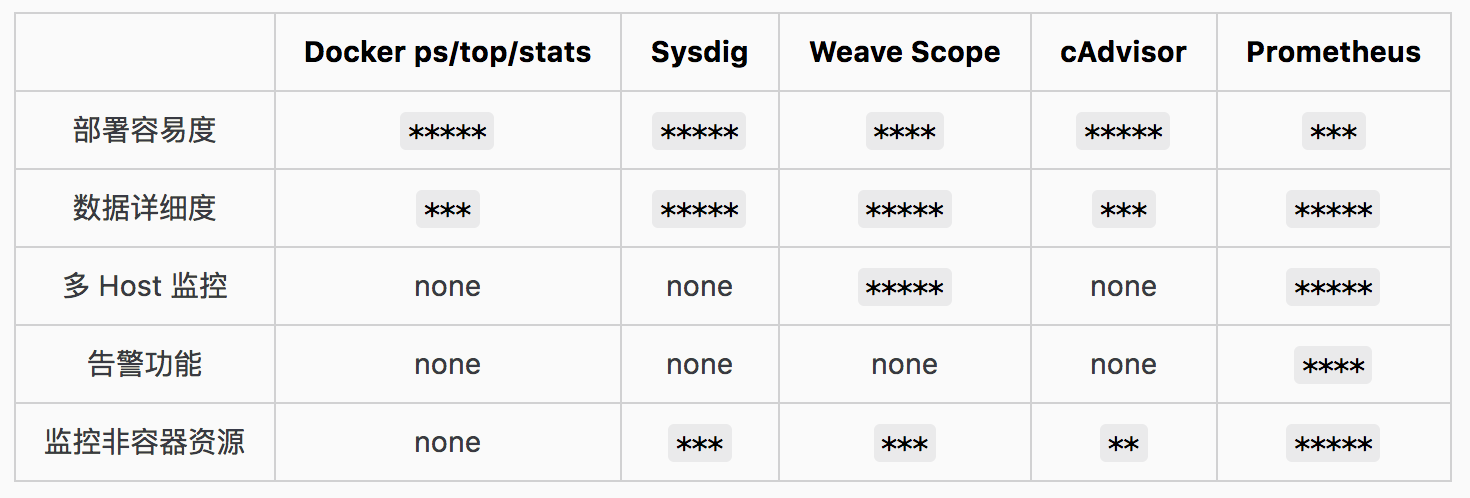
1 ps/top/stats

2 Sysdig

3 Weave Scope

4 cAdvisor

5 Prometheus



选择普罗米修斯的原因:

Prometheus 的数据模型和架构决定了它几乎具有无限的可能性。Prometheus 和 Weave Scope 都是优秀的容器监控方案。除此之外，Prometheus 还可以监控其他应用和系统，更为综合和全面。

K8s集群监控方案选型:

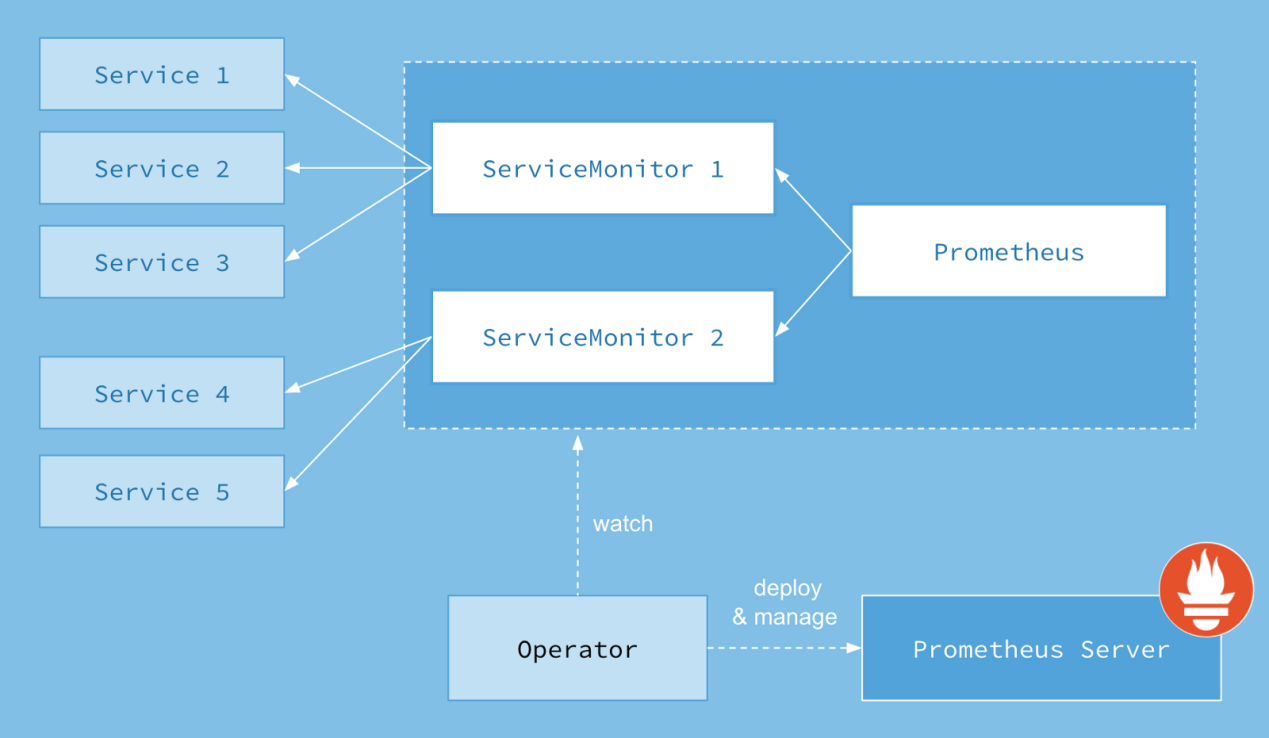
1 Weave Scope

2 Heapster 是 Kubernetes 原生的集群监控方案

3 Prometheus Operator监控方案

Prometheus Operator 是 CoreOS 开发的基于 Prometheus 的 Kubernetes 监控方案，是目前功能最全面的开源方案

**Prometheus Operator 架构**

****

#### Operator

Operator 即 Prometheus Operator，在 Kubernetes 中以 Deployment 运行。其职责是部署和管理 Prometheus Server，根据 ServiceMonitor 动态更新 Prometheus Server 的监控对象。

#### Prometheus Server

Prometheus Server 会作为 Kubernetes 应用部署到集群中。为了更好地在 Kubernetes 中管理 Prometheus，CoreOS 的开发人员专门定义了一个命名为 Prometheus 类型的 Kubernetes 定制化资源。我们可以把 Prometheus看作是一种特殊的 Deployment，它的用途就是专门部署 Prometheus Server。

#### **Service**

这里的 Service 就是 Cluster 中的 Service 资源，也是 Prometheus 要监控的对象，在 Prometheus 中叫做 Target。每个监控对象都有一个对应的 Service。比如要监控 Kubernetes Scheduler，就得有一个与 Scheduler 对应的 Service。当然，Kubernetes 集群默认是没有这个 Service 的，Prometheus Operator 会负责创建。

#### **ServiceMonitor**

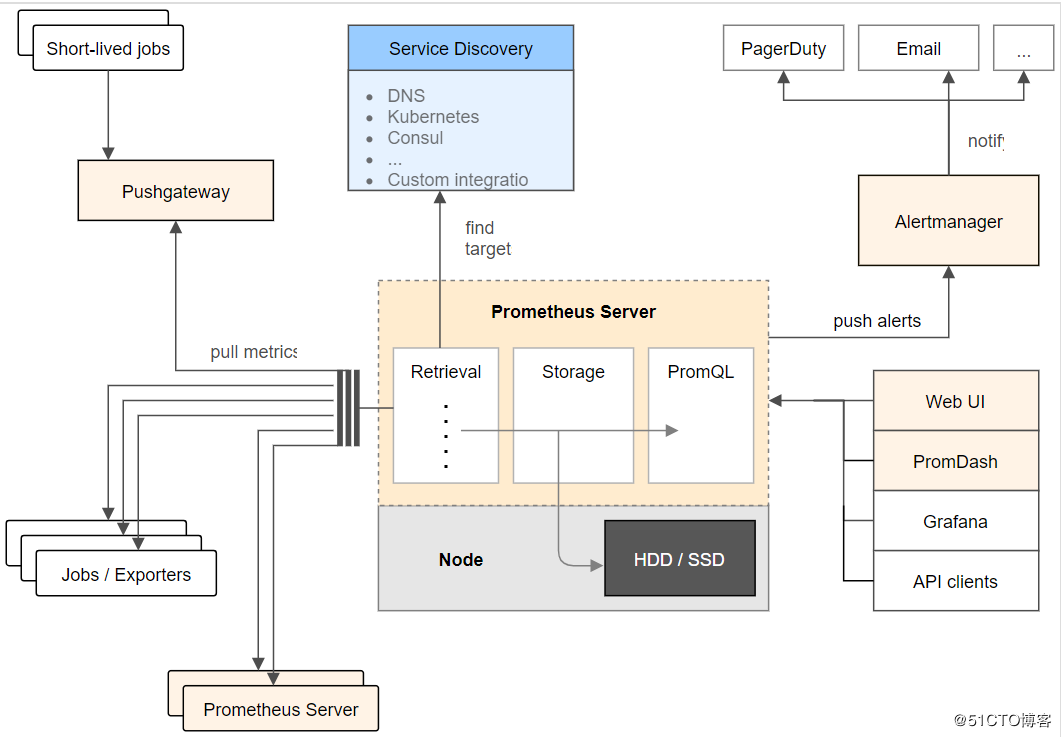
Operator 能够动态更新 Prometheus 的 Target 列表，ServiceMonitor 就是 Target 的抽象。比如想监控 Kubernetes Scheduler，用户可以创建一个与 Scheduler Service 相映射的 ServiceMonitor 对象。Operator 则会发现这个新的 ServiceMonitor，并将 Scheduler 的 Target 添加到 Prometheus 的监控列表中。

ServiceMonitor 也是 Prometheus Operator 专门开发的一种 Kubernetes 定制化资源类型。

#### **Alertmanager**

除了 Prometheus 和 ServiceMonitor，Alertmanager 是 Operator 开发的第三种 Kubernetes 定制化资源。我们可以把 Alertmanager 看作是一种特殊的 Deployment，它的用途就是专门部署 Alertmanager 组件。

普罗米修斯整体架构:



**Thanos 结合普罗米修斯**

**监控存储平面解决方案**

**我们开发Thanos的目的**

在一定的集群规模下，一些问题将在负载超出一个普通的Prometheus集群承载能力后不断被暴露出来。我们如何能够以一个经济可靠的方式来存储PB级别的历史数据？我们能够不牺牲查询响应时间便做到这一点吗？我们能够通过一个单一的查询接口访问到不同Prometheus服务器上的所有指标数据吗？再者，我们能否以某种方式合并通过Prometheus高可用集群采集到的复制数据吗？

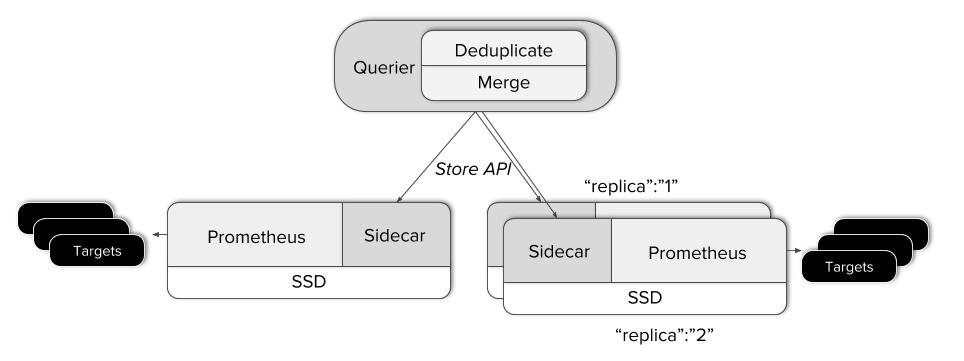
**Thanos的架构**

在上一节中我们列举了一些我们预期的目标，我们来盘点一下这些清单，看看Thanos是如何解决这些问题的。

全局视图

为了能够在现有的Prometheus集群之上收获一个全局视图，我们需要将中央查询层和所有服务器互联。Thanos Sidecar组件即是担任这样的角色，它会被部署到每一台正在运行的Prometheus服务端一侧。它充当的是一个代理服务器，通过Thanos规范化的基于gRPC的Store API提供Prometheus的本地数据，它也允许通过标签和时间段来选择时间序列数据。

在另外一端运行的是一个可以横向扩展并且无状态的Querier组件，在应答PromQL查询时比标准的Prometheus HTTP API做的事情稍微多一些。Querier、Sidecar 以及其他的 Thanos组件通过gossip协议通信。



1 当Querier收到一个请求时，它会向相关的Store API服务器（这里即是我们的Sidecar）发送请求，并从他们的Prometheus服务器获取时间序列数据。

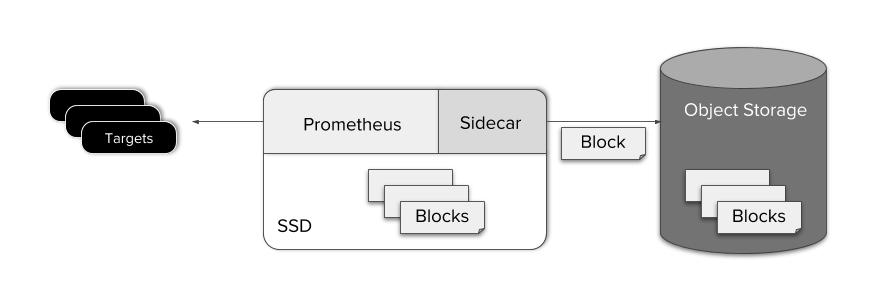
2 它将这些响应的数据聚合在一起，并对它们执行PromQL查询。 它可以聚合不相交的数据也可以针对Prometheus的高可用组进行数据去重。

它通过将完全分离的Prometheus部署整合到我们数据的一个全局视图的方式解决了我们遇到的问题里的核心难点。事实上，Thanos可以就这样部署，并且可以按需使用这些功能。根本不需要现有的Prometheus服务器做任何改动！

**不受限的保留数据!**

然而，迟早有一天，我们会希望保留超出Prometheus常规保留时间外的一些数据。为了实现这一点，我们决定使用对象存储系统来备份我们的历史数据。它在每个云甚至大多数本地数据中心上均有广泛应用，并且极具成本效益。此外，几乎所有对象存储解决方案均可以通过众所周知的S3 API进行访问。

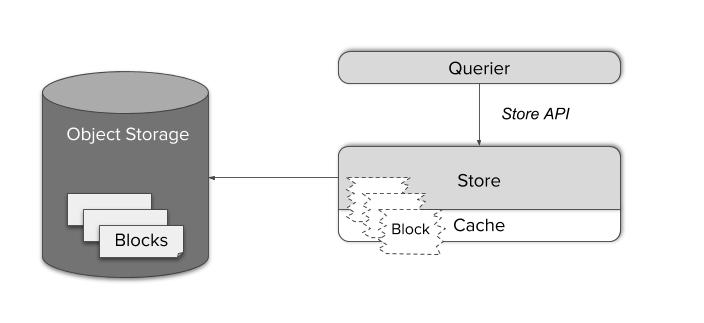
Prometheus的存储引擎大约每两个小时将其最近的内存数据写入磁盘。持久化的数据块包含固定时间范围内的所有数据，并且是不可变的。这一点很有用，因为这样一来，Thanos Sidecar便可以简单地监听Prometheus的数据目录变化，然后在新的数据块出现时将它们上传到对象存储桶中。



在指标数据块写入磁盘后通过Sidecar上传到对象存储的一个额外好处在于它可以保持 ”scraper“（由Prometheus和其配套的Thanos Sidecar组成）足够的轻量。这简化了运维，成本，和系统设计。

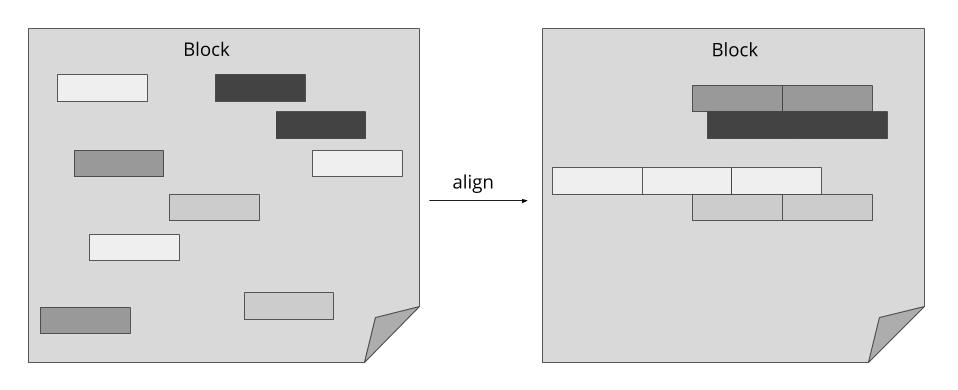
备份我们的数据很容易。 那么再次从对象存储中查询数据呢？

Thanos Store组件充当了一个对象存储里面数据的数据检索代理。就像Thanos Sidecar那样，它也会加入到gossip集群里并且实现Store API。这样一来现有的Querier就可以把它也当成是类似Sidecar那样的另外一个时间序列数据的数据源 —— 不需要任何特殊处理。



时间序列的数据块由一些大文件组成。按需下载它们的话会显得效率低下，而且在本地缓存它们需要巨大的内存和磁盘空间。

与之相反的是，Store Gateway知道如何处理Prometheus存储引擎的数据格式。通过智能的查询计划并且仅缓存必要索引部分的数据块的方式，它能够将一些复杂查询降级成一个针对对象存储里的文件最小数量的HTTP范围请求（HTTP range request）。通过这种方式，它可以在不影响响应时间的同时将原始请求的数量降低四到六个数量级，从整体上看，很难区分这和查询本地SSD上的数据有何差异。

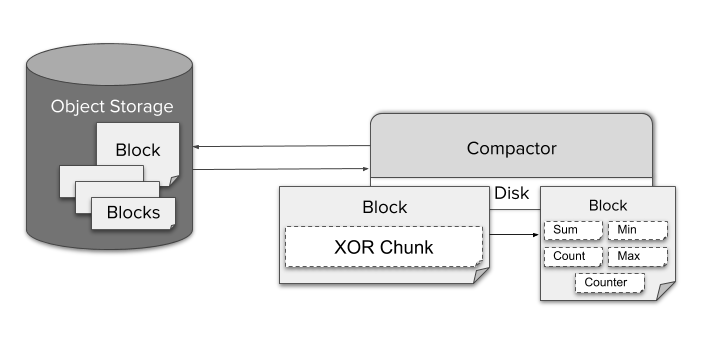


Thanos Querier通过利用Prometheus存储格式的特点（可在块文件中共存相关数据）显著降低了针对对象存储产品的单次请求成本。考虑到这一点，我们可以将多个字节的提取工作聚合到一个最小数量的批量调用中。

**压缩和降准采样（Compaction & downsampling）**

在时间序列的一个新数据块成功上传到对象存储的那一刻起，我们便可以把它当做是Storage Gateway上一条可用的“历史”数据。

然而，经过一段时间后，来自单一数据源（即附带运行Sidecar的Prometheus实例）的这些数据块逐渐累积，却没能充分发挥索引的全部潜能。为了解决这个问题，我们引入了一个单独的单例组件，名为Compactor。它会简单地将Prometheus的本地压缩机制应用到对象存储里的历史数据，并且可以作为一个简单的定时批处理任务执行。



为了生成降准采样的数据，Compactor会持续不断地将序列数据聚合到5分钟和一小时的分辨率。针对用TSDB的XOR压缩编码的每个原始数据块，它将会存储几个不同类型的聚合数据，例如，单一数据块的最小值，最大值，或者总和。这使得Querier能够针对给定的PromQL查询自动选择合适的聚合数据。

从用户的角度来看，使用降准采样后的数据不需要做什么特殊配置。当用户缩小（zoom in）和拉大（zoom out）时，Querier会自动在不同的分辨率和原始数据之间进行切换。或者，用户也可以通过在查询参数中指定一个自定义的“step”来直接控制它。

由于每GB的存储成本是微乎其微的，在默认情况下，Thanos将会始终在存储里维持原始的以及五分钟和一小时分辨率的数据，因而无需删除原始数据。

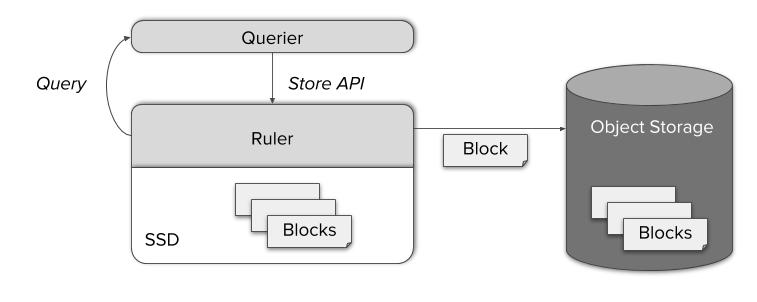
**记录规则（recording rule）**

即便使用了Thanos，记录规则依然是监控堆栈的重要组成部分。 它们减少了查询的复杂度，延迟和成本。它们还为用户提供了方便的查看指标数据重要的聚合后视图的快捷方式。Thanos建立在普通的Prometheus实例之上，因此，在现有的Prometheus服务器里保留记录规则（recording rule）和告警规则（alert rule）是完全有效的。但是，对于下面这些情况来说，这可能不太够用：

1 全局的告警和规则（例如：当三个集群中的两个以上的集群里的服务宕机时发出告警）

2 超出单台Prometheus本地保留数据的规则；

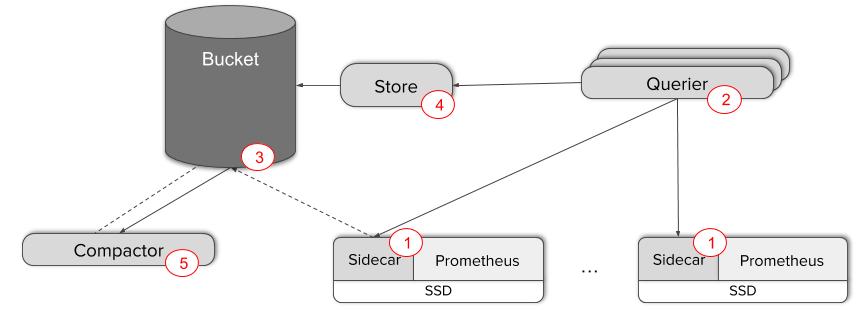
3 希望将所有规则和告警存储在一个地方



针对上述这些情况，Thanos提供了一个名为Ruler的单独组件，它会基于Thanos Querier执行规则并作出告警。通过公开众所周知的Store API，查询节点得以访问新计算出的指标数据。随后，它们也会备份到对象存储并可以通过Store Gateway访问。

**Thanos的力量**

Thanos非常灵活，它可以根据用户的使用场景进行不同的设置。在对普通的Prometheus实际做迁移时这一点尤其有用。我们不妨通过一个简单例子快速回顾一下从Thanos的组件里学到的东西。下面这个例子讲解了如何将自己普通的Prometheus迁移到我们闪亮的'无限保留指标'的世界：



1 将Thanos Sidecar添加到你的Prometheus服务端 - 例如，在Kubernetes pod里运行一个相邻容器；

2 部署一些Thanos Querier的副本以启用数据浏览功能。与此同时，用户可以轻松地在Scraper和Querier之间配置一个gossip集群。使用thanos\_cluster\_members指标来确认所有组件都已连接。

值得一提的是，仅靠上述单独的这两步就足以实现一个从潜在的Prometheus高可用复本获取结果的全局视图和无缝的数据去重！只需将仪表盘连接到Querier HTTP端点或直接使用Thanos UI即可。

但是，如果你想要实现指标数据的备份和长期保留，我们需要完成额外的下面三个步骤：

1 创建一个AWS S3或GCS存储桶。你只需要简单地配置一下Sidecar即可完成数据的备份。如今你也可以将本地保留策略配至最低。

2 部署一个Store Gateway然后将它加入到现有的gossip集群。做到这一点的话我们的查询便也可以访问到备份好的数据！

3 部署Compactor，通过应用压缩和降准采样来提升长期数据查询的响应能力。

只需要五步，我们便可以将Prometheus的实例组装成一个强大的监控系统，为用户提供全局视图，无限制的保留数据并且拥有潜在的指标高可用。