Thanos 技术内幕

袁　　胜

20200731

目录

Thanos

一、简介

二、架构

三、Prometheus

四、Thanos 组件

五、接入与开发

Thanos 简介

Thanos 简介

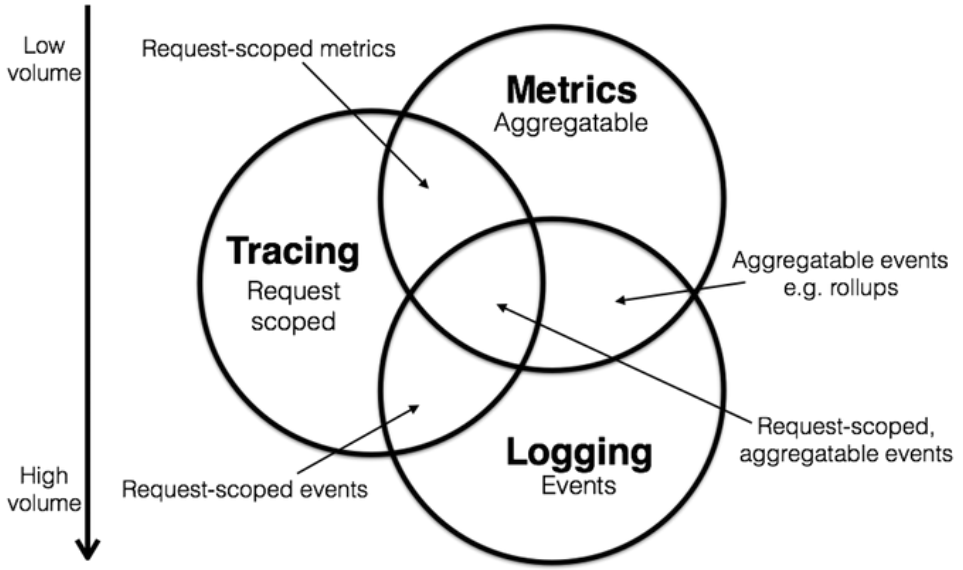
版本：0.12.2

特点：

- 兼容 Prometheus
- 可长期保留历史数据
- 组件化，可扩展
- 高可用

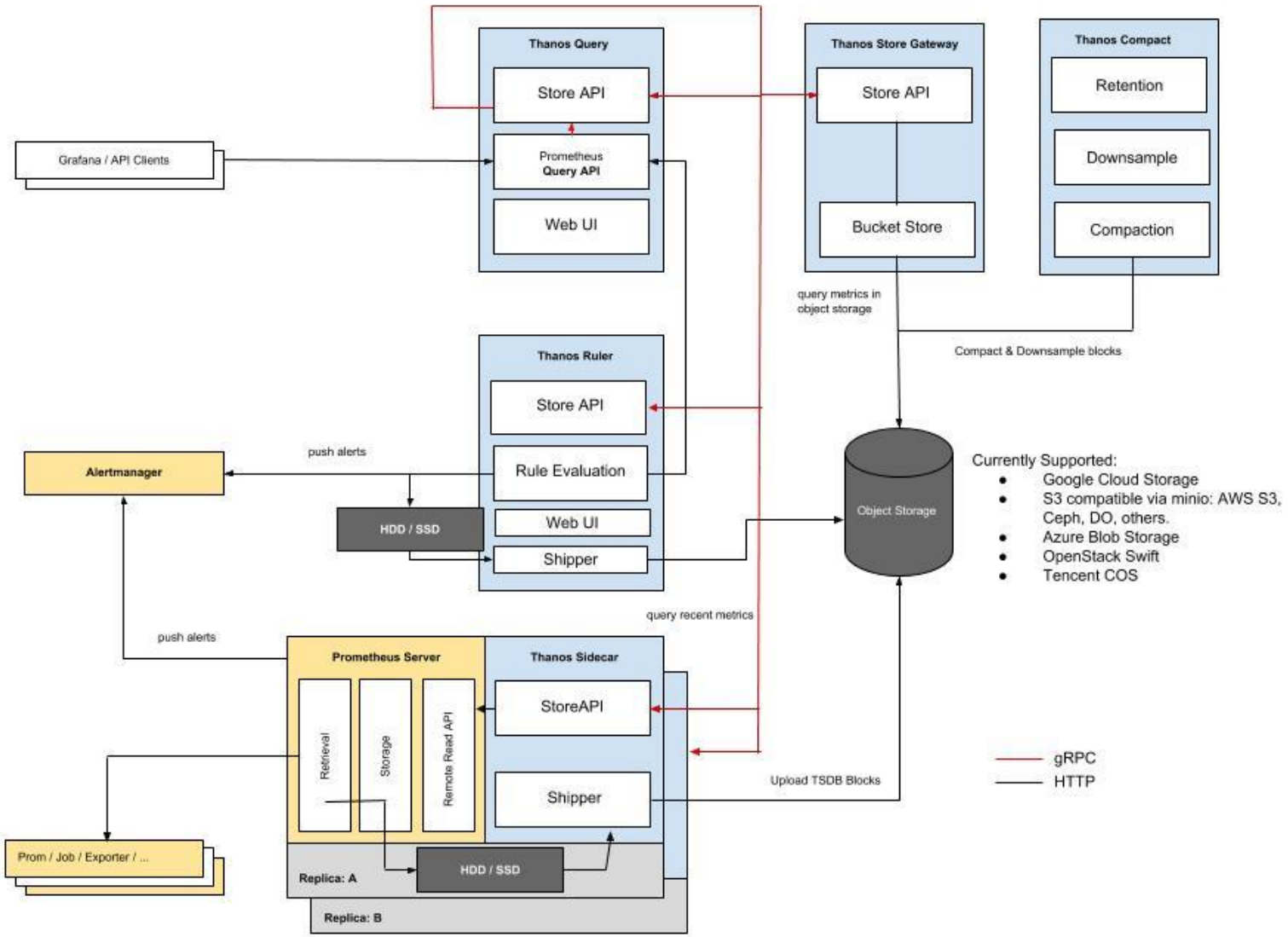
依赖：

- Prometheus v2.2.1+
- object storage (optional)



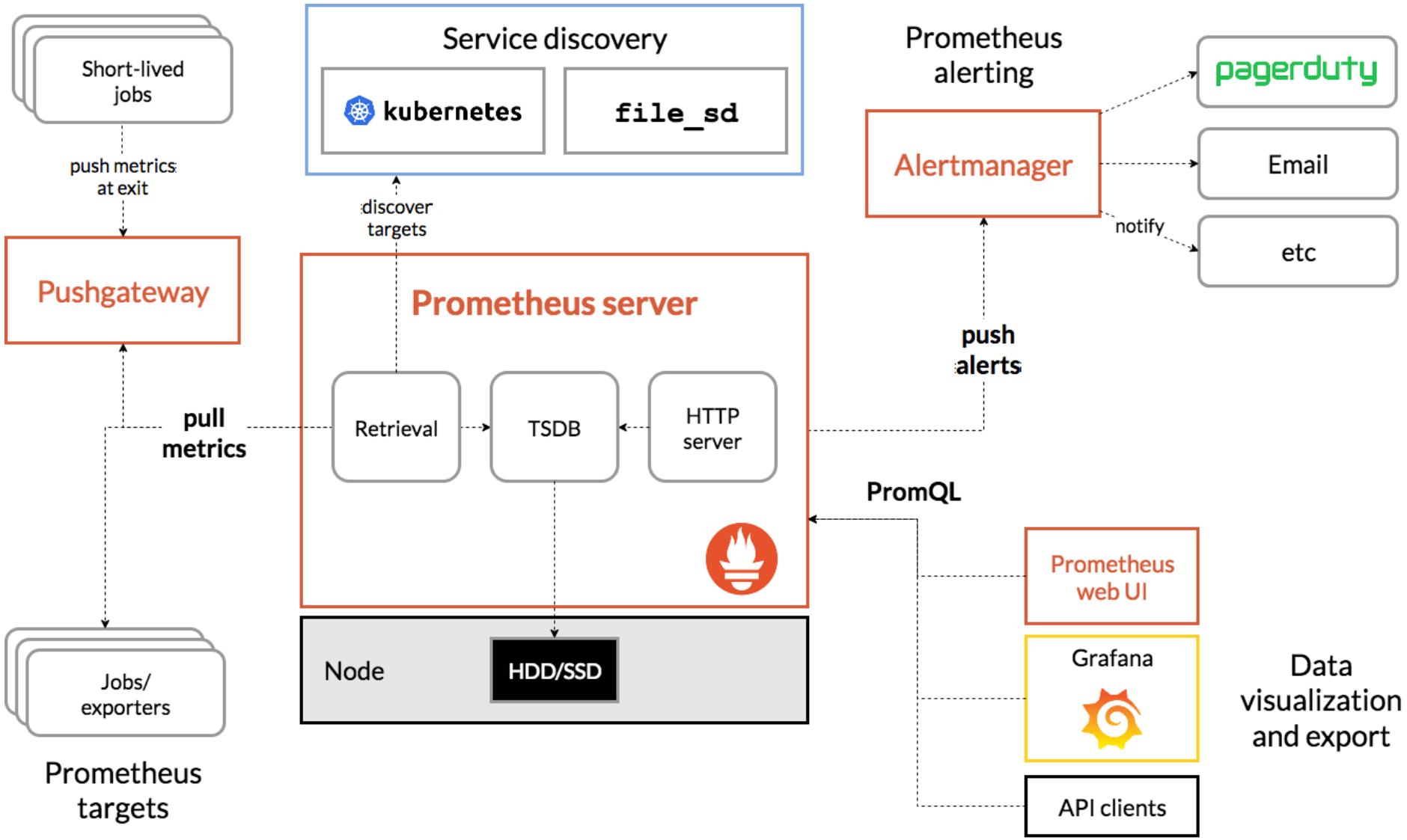
架构

Thanos


Thanos Query
Store API
Prometheus
Query API
Web UI
Grafana / API Clients
Thanos Store Gateway
Store API
Bucket Store
Thanos Compact
Retention
Downsample
Compaction
query metrics in
object storage
Compact & Downsample blocks
Thanos Ruler
Store API
Rule Evaluation
Web UI
Shipper
Alertmanager
push alerts
HDD / SSD
Object Storage
Currently Supported:
Google Cloud Storage
S3 compatible via minio: AWS S3, Ceph, DO, others.
Azure Blob Storage
OpenStack Swift
Tencent COS
query recent metrics
push alerts
Prometheus Server
Thanos Sidecar
Retrieval
Storage
Remote Read API
StoreAPI
Shipper
Upload TSDB Blocks
gRPC
HTTP
Prom / Job / Exporter / ...
Replica: A
HDD / SSD
Replica: B

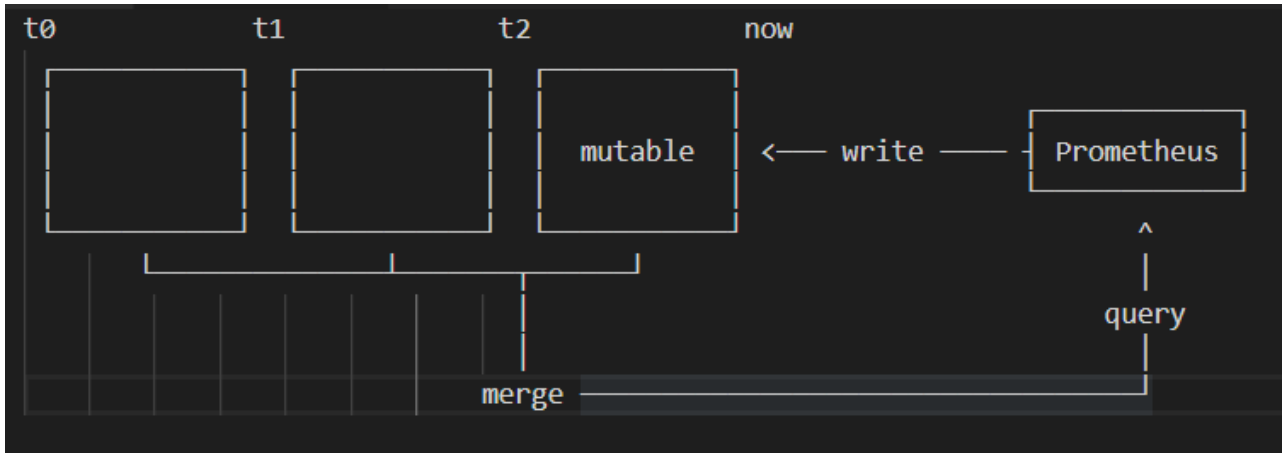
Prometheus

Thanos



Prometheus 2.x 采用自定义的存储格式将样本数据保存在本地磁盘当中。如下所示，按照两个小时为一个时间窗口，将两小时内产生的数据存储在一个块(Block)中，每一个块中包含该时间窗口内的所有样本数据(chunks)，元数据文件(meta.json)，索引文件(index)，tombstone (存储删除记录)。

```
./data
   |- 01BKGV7JBM69T2G1BGBGM6KB12 # 块
      |- meta.json  # 元数据
      |- wal        # 写入日志
        |- 000002
        |- 000001
   |- 01BKGTZQ1SYQJTR4PB43C8PD98  # 块
      |- meta.json  #元数据
      |- index   # 索引文件
      |- chunks  # 样本数据
        |- 000001
      |- tombstones # 逻辑数据
   |- 01BKGTZQ1HHWHV8FBJXW1Y3W0K
      |- meta.json
      |- wal
        |-000001
```



当前时间窗口内正在收集的样本数据，Prometheus则会直接将数据保存在内存当中。为了确保此期间如果Prometheus发生崩溃或者重启时能够恢复数据，Prometheus启动时会从写入日志(WAL)进行 replay，从而恢复数据。此期间如果通过API删除时间序列，删除记录也会保存在单独的逻辑文件当中(tombstone)。

组件 —— Thanos

Thanos

- 组件清单： Query，Store Gateway, Sidecar，Ruler，Compact。
- 设计要点：
  - 除 Compact 外，其他组件都以 gRPC 的方式暴露 Store API，用于查询，但不同组件的 Store API 有不同的实现。
  - 所有组件都暴露 HTTP 服务，提供 metrics 接口暴露各组件本身的指标，以及健康检查接口和 debug API 。
  - Sidecar 负责将 Prometheus 的 TSDB 数据块上传至 OBS 服务，并在 Prometheus 上层封装 gRPC 服务。
  - Query 组件提供兼容 promQL 的查询语言和查询接口，便于与 Grafana 等 dashboard 无缝对接。
  - Store Gateway 组件负责从 OBS 服务中查询存量数据。
  - Ruler 组件负责根据配置的规则进行告警，由于 Prometheus-Server 原生也支持类似的功能，Ruler 的核心部分直接借鉴了 Prometheus-Server 的源码实现。
  - Compact 与其他组件之间没有直接依赖关系，主要负责对 OBS 中的历史数据进行压缩和降采样。

注意：Prometheus-Server/Thanos Ruler/Doraemon 组件都实现了告警规则引擎的功能，各自的侧重点不同，在实践中，需要根据具体需求进行选型。

组件 —— Sidecar

Thanos

Sidecar 的实现主要包括如下四个部分。

- gRPC 服务：以 gRPC 接口的方式暴露 Prometheus 服务，可以认为是 Prometheus-Server http 服务的 gRPC 代理。
- http 服务：辅助性 http 接口，包括探针接口，metrics 接口，debug 接口的开放。
- Reloader：负责 watch Prometheus 的配置文件，当监听到配置修改时，调用 Prometheus-Server 的 reload API，实现 Prometheus 配置自动更新。
- Shipper：负责将 Prometheus-Server 底层存储的 TSDB 数据块上传至 OBS 服务。

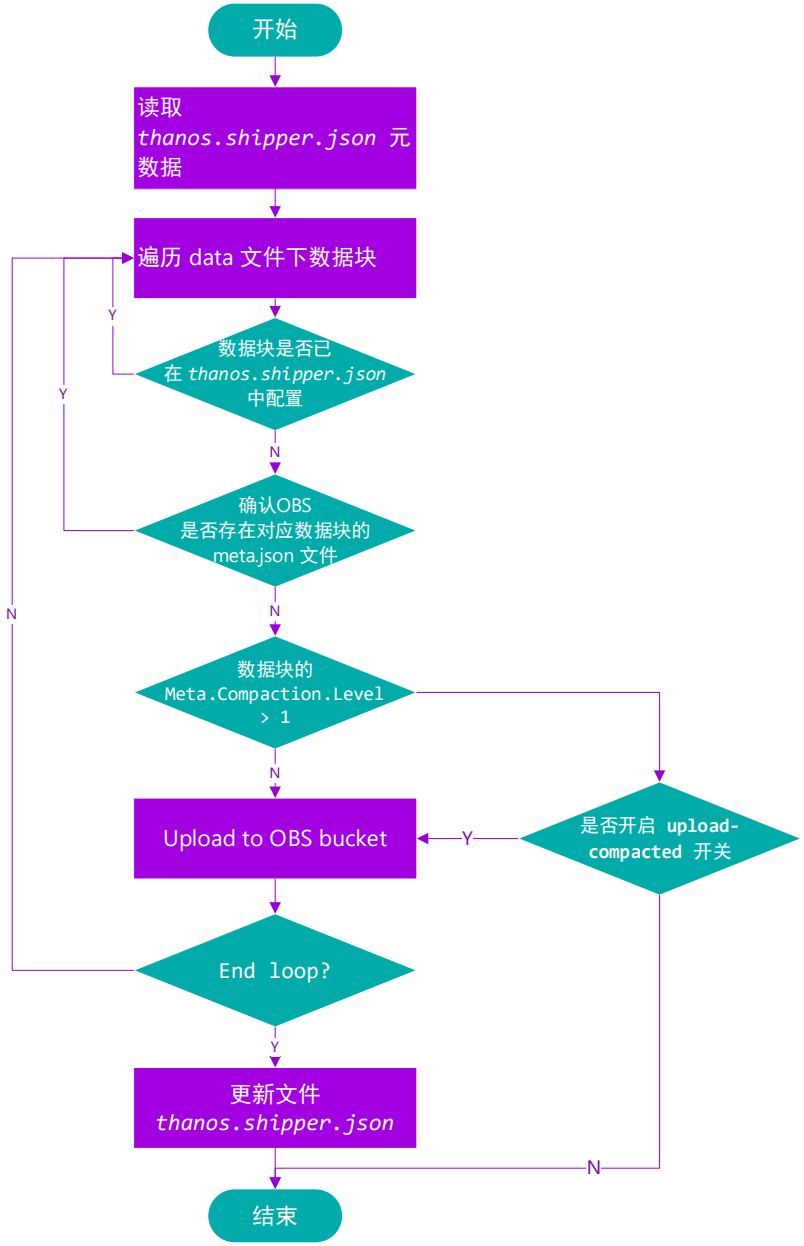
组件 —— Sidecar

Thanos

Sidecar 的核心是 Shipper。Shipper 启动后，会启动周期为 30s 的定时任务，定时执行 Shipper.Sync() 方法，该主要负责将 Prometheus-Server 的数据块同步至对象存储服务，其主要流程如流程图所示。

为了实现上述功能，要求 Prometheus-Server 的数据对 Sidecar 组件是可见的。通常，Prometheus-Server 与 Sidecar 部署在同一个 Pod 中，存储卷需要同时挂载到 Prometheus-Server 和 Sidecar 容器。

```
$ s3cmd ls s3://maa-stg-bucket
    DIR s3://maa-stg-bucket/01BKGV7JBM69T2G1BGBGM6KB12
    DIR s3://maa-stg-bucket/01BKGTZQ1SYQJTR4PB43C8PD98
    DIR s3://maa-stg-bucket/01BKGTZQ1HHWHV8FBJXW1Y3W0K
```



组件 —— Sidecar

Thanos

Sidecar 是 Thanos 架构中比较简单的组件，其参数配置如下。

配置项	描述	参考/约束
tracing.config-file	调用链追踪埋点配置。	
http-grace-period	http 服务停止前的等待时长，一般用于平滑重启服务。	
grpc-grace-period	gRPC 服务停止前的等待时长，一般用于平滑重启服务。	
prometheus.url	Prometheus-Server 访问地址。	
prometheus.ready_timeout	服务启动到 Ready 的超时时间。	
receive.connection-pool-size	Http 客户端访问 Prometheus 的连接池大小配置。	
receive.connection-pool-size-per-host	Http 客户端访问单个服务的连接池大小配置。	
tsdb.path	Prometheus-Server 数据块存储目录。	
reloader.xxx	与 Prometheus 配置文件 auto reload 相关的参数。	
objstore.config	OBS 存储服务配置。	
shipper.upload-compacted	是否开启压缩数据块上传	
min-time	定义上传数据起始时间，对于将现有的 Prometheus-Server 接入 Thanos 体系时比较有意义。	

组件 —— Store

Thanos

Store 组件全称 Store Gateway，其核心功能是为 OBS 中的存量数据提供易用的服务，其上游组件是 Query 组件。

- gRPC 服务：以 gRPC 接口的方式暴露 Store API。
- http 服务：提供 bucket-webui 页面访问接口以及辅助性 http 接口，包括探针接口，metrics 接口，debug 接口等。
- BucketStore：对接后端对象存储的核心，是 Store API 接口的底层实现。
- IndexCache：缓存索引。

组件 —— Store

Thanos

Store 组件需要将对象存储中的数据块元数据信息同步至本地，建索引。IndexCache 的大小决定了缓存大小，可以大大加速查询。在开发设计上，考虑到 Store 为高内存型组件，会通过 BytesPool 对内存的使用进行优化。

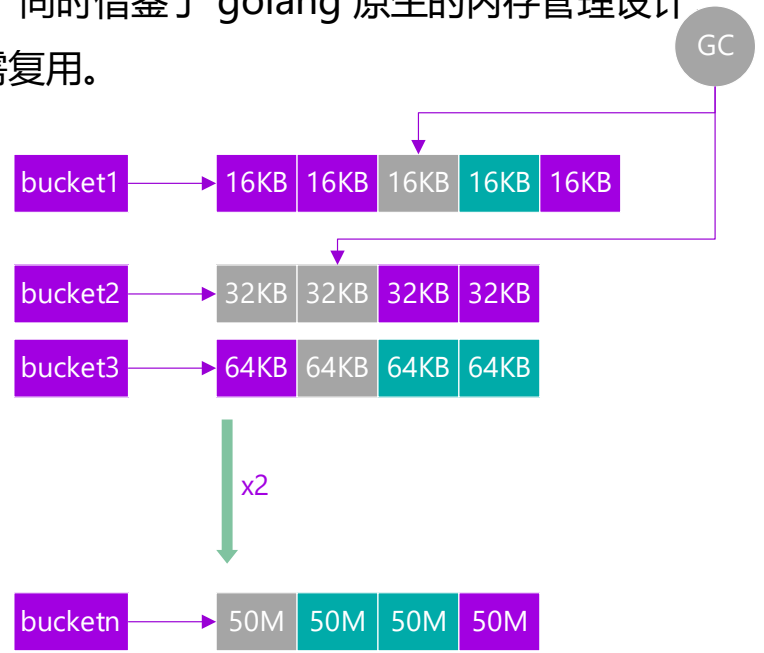
```
type IndexCache interface {
   StorePostings(ctx context.Context, blockID ulid.ULID, l labels.Label, v []byte)
   FetchMultiPostings(ctx context.Context, blockID ulid.ULID, keys []labels.Label) (hits map[labels.Label][]byte, misses []labels.Label)
   StoreSeries(ctx context.Context, blockID ulid.ULID, id uint64, v []byte)
   FetchMultiSeries(ctx context.Context, blockID ulid.ULID, ids []uint64) (hits map[uint64][]byte, misses []uint64)
}
```

```
type BytesPool interface {
   Get(sz int) (*[]byte, error)
   Put(b *[]byte)
}
```

```
type BucketedBytesPool struct {
   buckets   []sync.Pool
   sizes     []int
   maxTotal  uint64
   usedTotal uint64
   mtx       sync.Mutex

   new func(s int) *[]byte
}
```

BytesPool 底层采用 sync.Pool 优化进程对内存的使用，同时借鉴了 golang 原生的内存管理设计将 BucketedBytesPool 分成不同大小的内存区域，按需复用。

```
func (p *BucketedBytesPool) Put(b *[]byte) {

   for i, bktSize := range p.sizes {
      if cap(*b) > bktSize {
         continue
      }
      *b = (*b)[:0]
      p.buckets[i].Put(b)
      break
   }
}
```



组件 —— Store

Thanos

```
type InMemoryIndexCache struct {
   mtx sync.Mutex

   logger           log.Logger
   lru              *lru.LRU
   maxSizeBytes     uint64
   maxItemSizeBytes uint64

   curSize uint64

   evicted          *prometheus.CounterVec
   requests         *prometheus.CounterVec
   hits             *prometheus.CounterVec
   added            *prometheus.CounterVec
   current          *prometheus.GaugeVec
   currentSize      *prometheus.GaugeVec
   totalCurrentSize *prometheus.GaugeVec
   overflow         *prometheus.CounterVec
}

type LRU struct {
   size      int
   evictList *list.List
   items     map[interface{}]*list.Element
   onEvict   EvictCallback
}
```

type: IN-MEMORY

config:

max_size: 0

max_item_size: 0

```
type MemcachedIndexCache struct {
   logger    log.Logger
   memcached cacheutil.MemcachedClient

   // Metrics.
   requests *prometheus.CounterVec
   hits     *prometheus.CounterVec
}
```

type: MEMCACHED

config:

addresses: []

timeout: 0s

max_idle_connections: 0

max_async_concurrency: 0

max_async_buffer_size: 0

max_item_size: 1MiB

max_get_multi_concurrency: 0

max_get_multi_batch_size: 0

dns_provider_update_interval: 0s

组件 —— Store

Thanos

```
type LRU struct {
   size      int
   evictList *list.List
   items     map[interface{}]*list.Element
   onEvict   EvictCallback
}
```

```
type List struct {
   root Element
   len  int
}
```

```
type Element struct {
   next, prev *Element
   list *List
   Value interface{}
}
```

LRU 缓存的实现为 harshcorp 公司开源的 LRU 实现，使用的数据结构为双向链表和哈希表。HASH 表用于实现缓存的 O(1) 查询，双向链表用于实现 LRU。

写：写入一条数据，意味着将数据写入在 HASH 表中，并将其插入到 List 最前面。

读：读取一条数据，意味着将数据在 HASH 表中进行查找，并在 List 中将其向前移动一位。

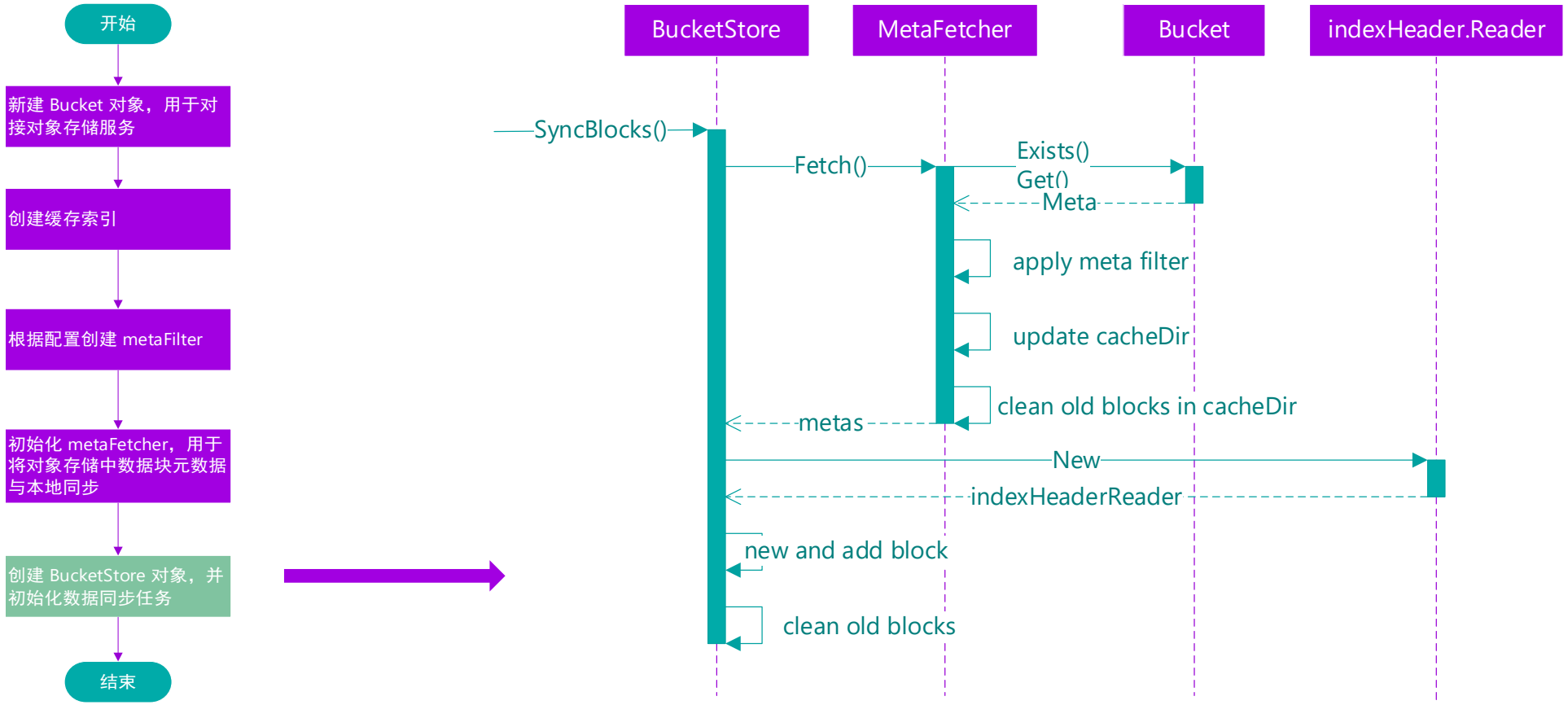
更新：更新 HASH 表中的数据，并将其在 List 中向前移动一位。

删除：删除 HASH 表中的数据，并将其在 List 中删除。

总结：通过 HASH 表 + 双向链表的设计，使 CRUD 操作都达到了 O(1) 时间复杂度，同时是纯内存操作，整个 索引缓存的效率非常高。另外，也支持 Memcached，但尚不支持 Redis 。

组件 —— Store

BucketStore 是 Store Gateway 组件的核心数据结构，它实现了 gRPC 查询服务接口。Store Gateway 启动后，除启动 gRPC 以及 HTTP 服务外，最重要的就是初始化 BucketStore。初始化 Bucket 包括初始化相关的数据结构以及初始化数据同步任务。



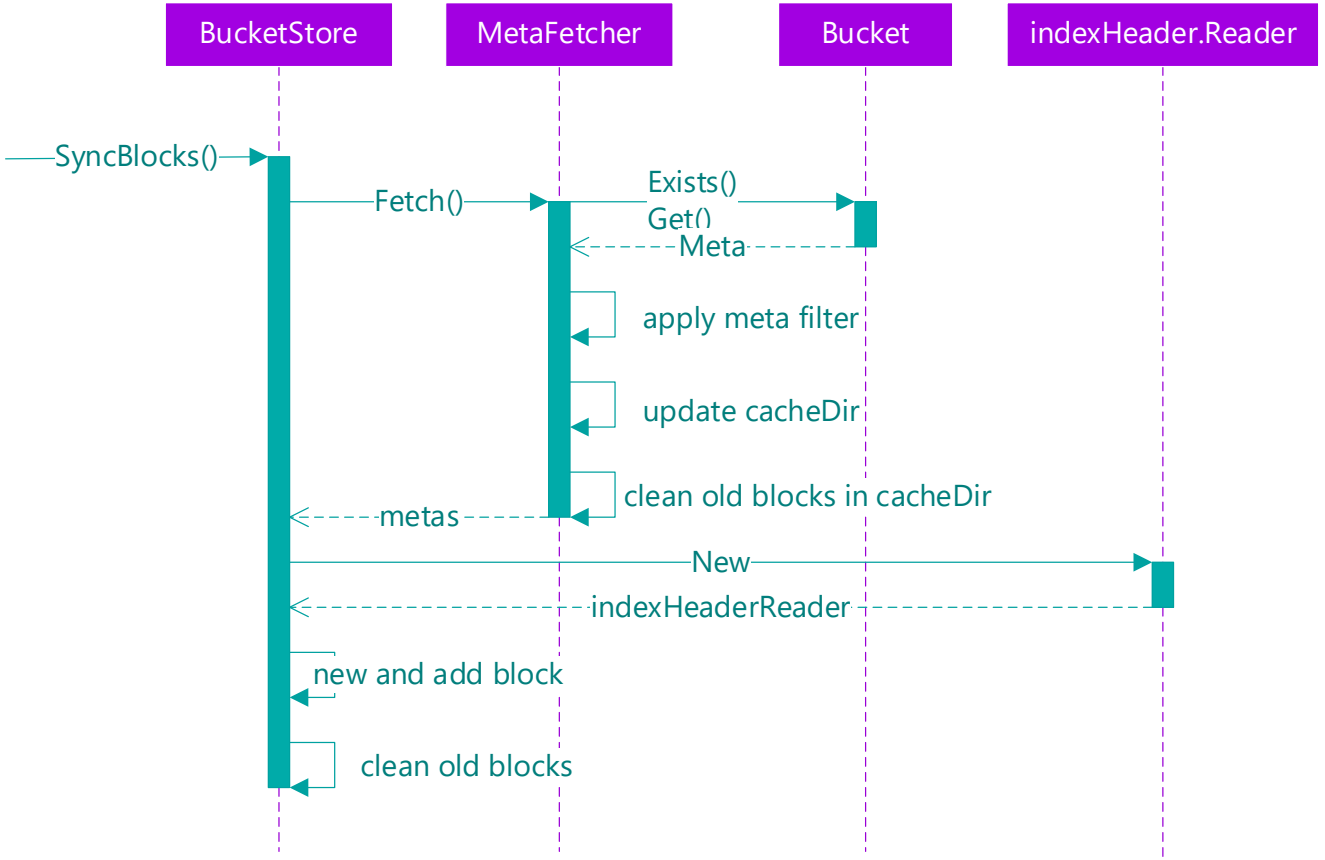
组件 —— Store

Thanos

Store 组件如何对 OBS 中的原始数据块进行过滤？

Store 组件从 OBS bucket 中同步元数据后，会通过 Filter 对元数据进行过滤。Filter 是控制数据同步的主要方式。

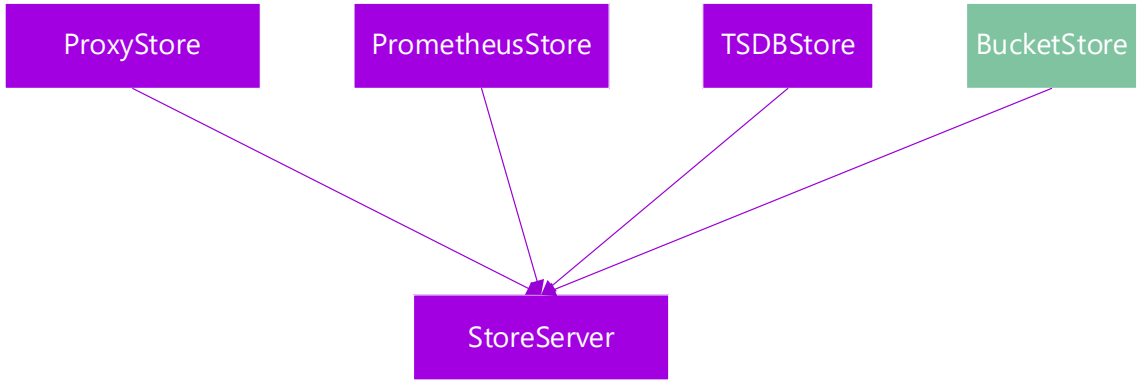
- 通过 min-time, max-time 参数设置 TimeFilter, 控制同步某段时间的数据块。
- 通过 relabel-config 参数设置 MetaFilter，丢弃带有特定标签的数据块。
- 开发者也可以很方便的实现 Filter 接口进行扩展，开发自己的 Filter。

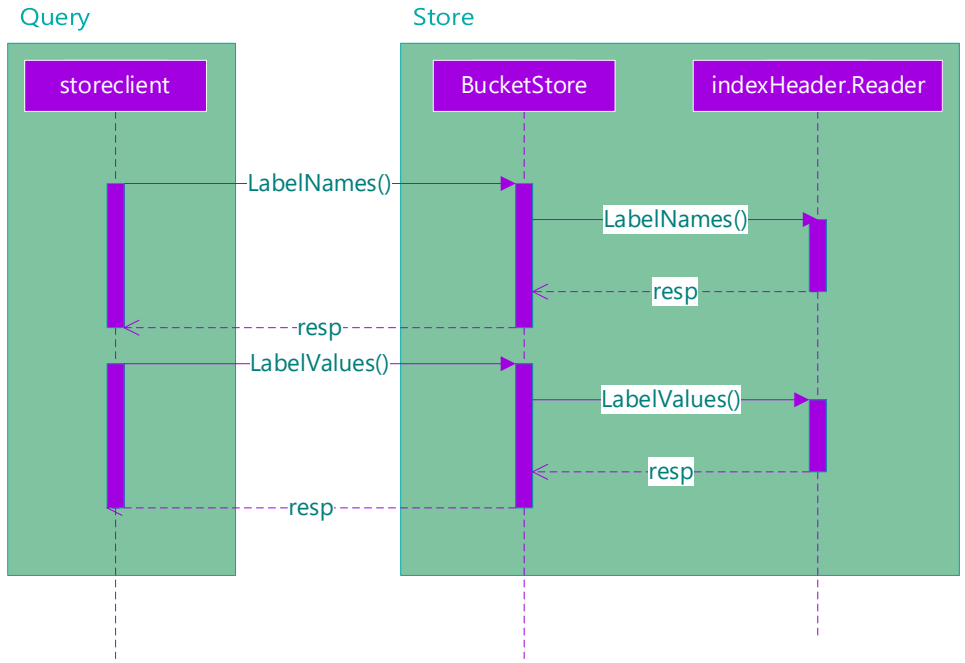


组件 —— Store

Thanos

Store 组件时整个 Thanos 体系中内存资源需求最高的组件，因为它要将对象存储中的数据块拉取到本地，并加载到内存中，对外提供查询服务。Store 组件中 BucketStore 实现了如下的 Store API 接口。





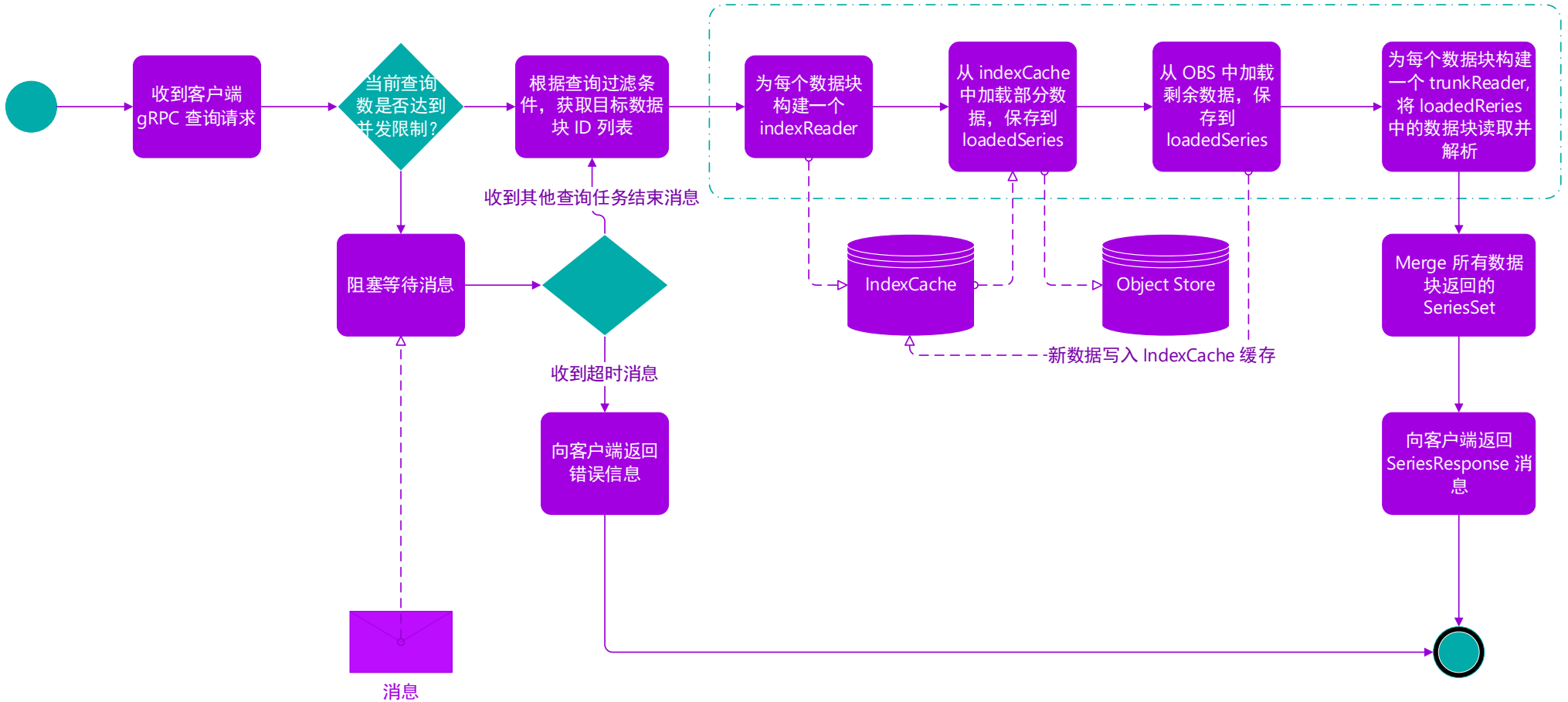

```
type StoreServer interface {

   Info(context.Context, *InfoRequest) (*InfoResponse, error)

   Series(*SeriesRequest, Store_SeriesServer) error

   LabelNames(context.Context, *LabelNamesRequest) (*LabelNamesResponse, error)

   LabelValues(context.Context, *LabelValuesRequest) (*LabelValuesResponse, error)
}
```

LabelNames() 和 LabelValues() 主要用于获取元数据，其调用链比较简单。

组件 —— Store

Thanos

Store 组件的时序数据查询由 Store API 的 Series 接口实现，这也是 Store API 中最复杂的接口。



Thanos

参数配置

配置项	描述	参考/约束
data-dir	本地缓存数据目录，包括两类数据	
index-cache-size	仅对内存实现的 IndexCache 有效，默认大小 250MB。如果缓存数据超过这里的配置限制，会触发 LRU 机制进行内存回收。	对 Memcached 实现的索引缓存无需配置。
index-cache.config	IndexCache 配置文件，可以为 Memory 类型或者 Memcached 类型	如果使用 Memcached，需要另外搭建服务。
chunk-pool-size	内存缓存数据大小，默认 2GB ，可根据实际情况配置，底层是 sync.Pool 实现，可以提高内存利用率	
store.grpc.series-sample-limit	限制 Series() 接口查询返回的最大数据量	
store.grpc.series-max-concurrency	限制 Series() 接口的并发请求数，默认值为 20。对于超过配置的调用请求，会排队等待，排队时间计入超时计算	
sync-block-duration	SyncBlocks 定时任务调度周期，默认 3m	
block-sync-concurrency	Store 组件会定时调用 SyncBlocks 与 OBS 进行数据同步，如果由多个数据块需要同时从 OBS 同步到本地，会启动多个协程并发同步。block-sync-concurrency 参数控制这里的并发度，超过并发度的任务会等待	
min-time，max-time	设置 Store 对外暴露的数据时间范围，默认不设限制	
grpc-xxx	grpc 服务相关配置	
http-xxx	http 服务相关配置	
selector.relabel-config	OBS 中数据块过滤过滤规则，与 Prometheus-Server relabel 格式兼容。	

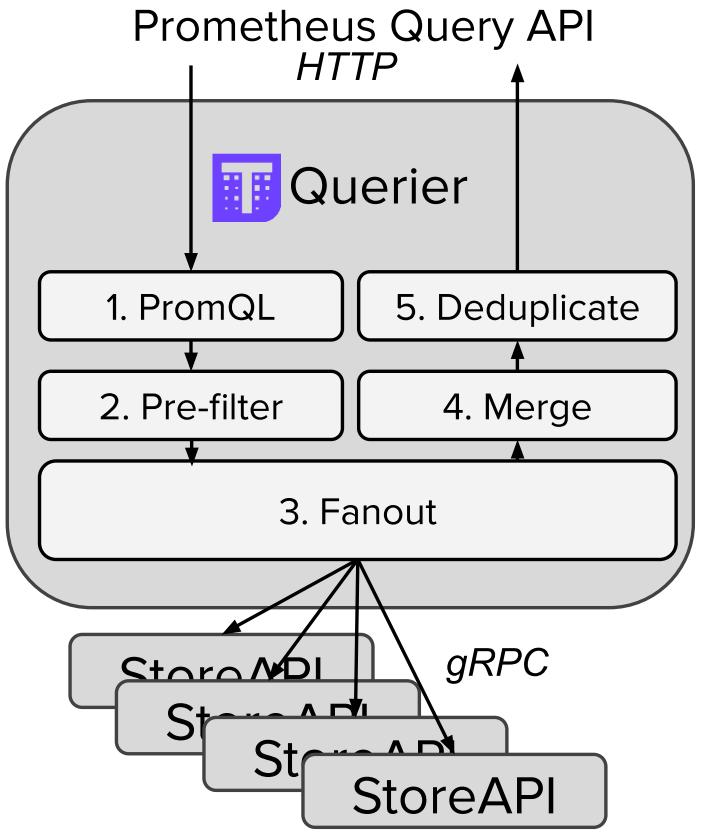
组件 —— Query

Thanos

Query 组件核心功能是对外提供兼容 Prometheus Query API 的查询接口，其下游组件要求以 gRPC 服务的方式支持 Store API。也就是，Query 组件与其下游组件之间通过 gRPC 协议进行交互。

- 数据去重

Query 组件启动时，默认会根据 query.replica-label 字段做重复数据的去重，你也可以在页面上勾选 deduplication 来决定。Query 的结果会根据你的 query.replica-label 的 label 选择副本中的一个进行展示。如果 0，1，2 三个副本都返回了数据，且值不同，Thanos 会基于打分机制，选择更为稳定的 replica 数据。去重会对查询效率有一定影响，是否开启需要根据实际情况评估。



组件 —— Query

Thanos

- API 设计

Query 组件为了便于对接前端，在接口设计上有一套与 Prometheus-Server 相同的 API 接口设计。

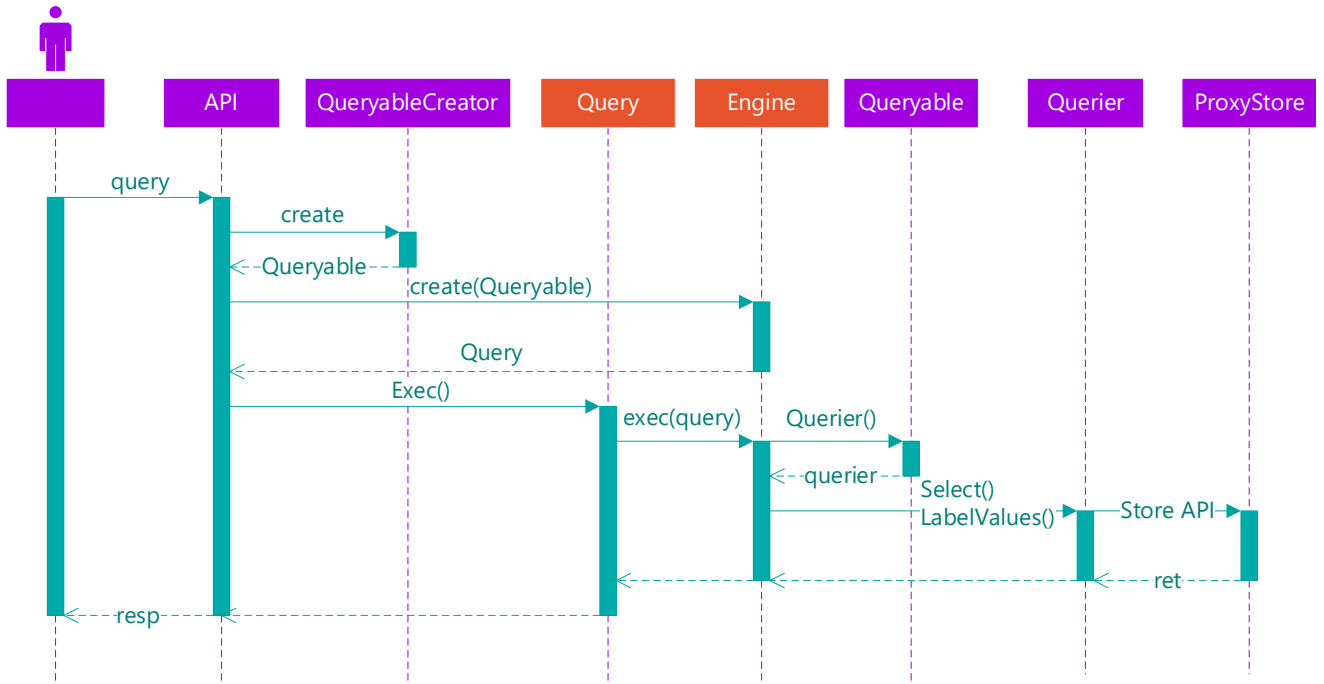
```
// 携带 query&time 参数，查询某个指标在某个时间点所有满足条件的值
r.Get("/query", instr("query", api.query))
r.Post("/query", instr("query", api.query))

// 携带 query&start&end@step 参数，查询某个指标在一段时间范围内所
有满足条件的值
r.Get("/query_range", instr("query_range", api.queryRange))
r.Post("/query_range", instr("query_range", api.queryRange))

// 如果 ${:name} == "__name__" 接口返回所有指标清单
r.Get("/label/:name/values", instr("label_values",
api.labelValues))

// series 与 influxdb 中的 series 概念类似，返回满足条件的指标
series 集合
r.Get("/series", instr("series", api.series))
r.Post("/series", instr("series", api.series))

// 返回整个 prometheus 数据中所有标签集合
r.Get("/labels", instr("label_names", api.labelNames))
r.Post("/labels", instr("label_names", api.labelNames))
```

组件 —— Query

Thanos

- API 实现

Query 查询接口的实现流程比较复杂，这里以 query 接口为例说明。Query 接口通过实现 Prometheus 的 Queryable 接口完成了对查询功能的改写。 Queryable 只有一个方法 Querier()，用于返回一个可用的 querier 对象。Querier 接口由 Prometheus 定义，thanos 通过实现接口 Querier 的方式，完成了对底层数据 ProxyStore 的访问，并保证了对 promQL 的支持。




```
// A Queryable handles queries against a storage.
type Queryable interface {
   // Querier returns a new Querier on the storage.
   Querier(ctx context.Context, mint, maxt int64) (Querier,
error)
}

// Querier provides reading access to time series data.
type Querier interface {
   // Select returns a set of series that matches the given
label matchers.
   Select(*SelectParams, ...*labels.Matcher) (SeriesSet,
Warnings, error)

   // LabelValues returns all potential values for a label
name.
   LabelValues(name string) ([]string, Warnings, error)

   // LabelNames returns all the unique label names present in
the block in sorted order.
   LabelNames() ([]string, Warnings, error)

   // Close releases the resources of the Querier.
   Close() error
}
```

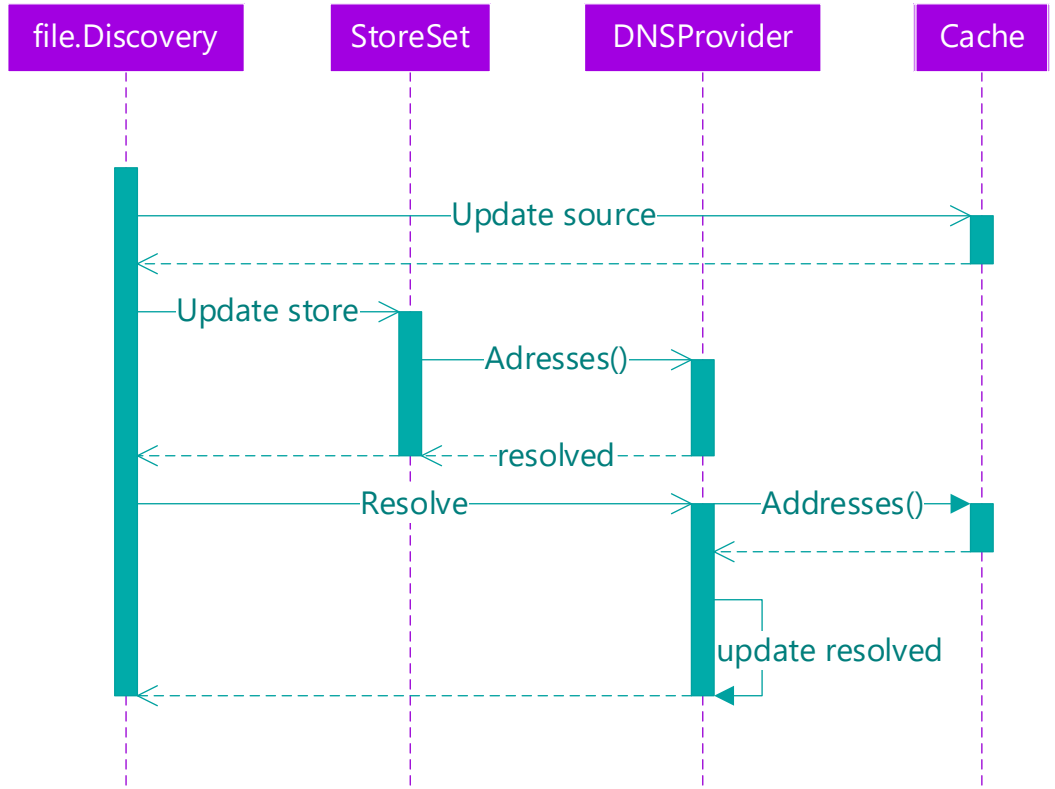
组件 —— Query

Thanos

Query 组件启动的核心包括两部分，一部分是前面提到的 API 服务，另一部分是启动后端 Store 配置动态刷新任务，详情如下。

1. 如果开启 SD 功能，Query 组件启动后启动后台任务监听 SDFile。如果 SDFile 有变化，监听的 goroutine 会发送更新消息。
2. 收到消息的 Discovery 首先会更新 Cache 中缓存的数据。
3. 更新 StoreSet 中的 stores 信息及其状态。 StoreSet 中的 stores 信息服务于 ProxyStore 。
4. 更新 DNSProvider 中的缓存，该缓存保存了域名向 IP 的映射。

注意：除了上面的流程外，Query 还会启动两个定时任务：一个以 5s 的周期定时执行步骤 3，一个以 dnsSDInterval 配置的时间为周期定时执行步骤 4.



思考：上面的流程是源码实现，但似乎有些不合理。

组件 —— Query

Thanos

- 服务发现相关配置

Query 需要查询的后端存储服务通常会包括多个，这些服务的地址可以以多种方式提供给 Query 组件，与之相关的配置项如下。

配置项	描述	参考/约束
store	静态配置 Store API Server 的地址，一般需要配置多个。	
store-strict	与 store 参数作用类似，不同的是这里的地址即使健康检查失败也不会被剔除。	
store.sd-files	SD 配置文件，文件中配置 Store API Server 地址。如果文件更新，Query 会自动 reload。	
store.sd-interval	自动 reload SD 配置文件的周期，默认 5 分钟。	
store.sd-dns-interval	DNS 刷新时间。	
store.sd-dns-resolver	DNS 解析器，取值可以为 golang/miekgdns。golang 为 golang net 包中自带的 DNS 解析实现，miekgdns 为 thanos 自己实现的 DNS 解析。	
store.unhealthy-timeout	健康检查结果为不健康的 store 会在 unhealthy-timeout 时间后，在 store UI 页面中更新。	
store.response-timeout	Store API 查询超时时间。	

组件 —— Query

Thanos

- 参数配置

配置项	描述	参考/约束
query.timeout	promQL 解析引擎的查询超时时间，默认 2m。	
query.max-concurrent	控制 Engine 对 promQL 最大并发查询数，默认20。对于超过 20 的并发查询，将会排队等待执行，排队时间计入 query.timeout 的计时时间。	
query.replica-label	默认的 replicaLabels 参数，如果 API 访问请求中指定了该参数，默认参数会被覆盖掉。	
query.auto-downsampling	降采样查询默认参数，一般用于前端展示。如果 API 访问请求中指定了该参数，默认参数会被覆盖掉。	
query.partial-response	是否配置只返回部分数据，默认开启。如果 API 访问请求中指定了该参数，这里的配置会被覆盖。	
query.default-evaluation-interval	promQL 引擎中的数据处理相关参数，默认值为 1m。	
grpc-client-xxx	Query 组件作为客户但访问其他 gRPC 服务端的配置。	
grpc-server-xxx	Query 组件启动 Store API gRPC 服务相关配置。如果 Query 组件存在级联调用，会用到。	
tracing.config-file	调用链追踪埋点配置。	需要单独的分布式调用追踪服务支持。

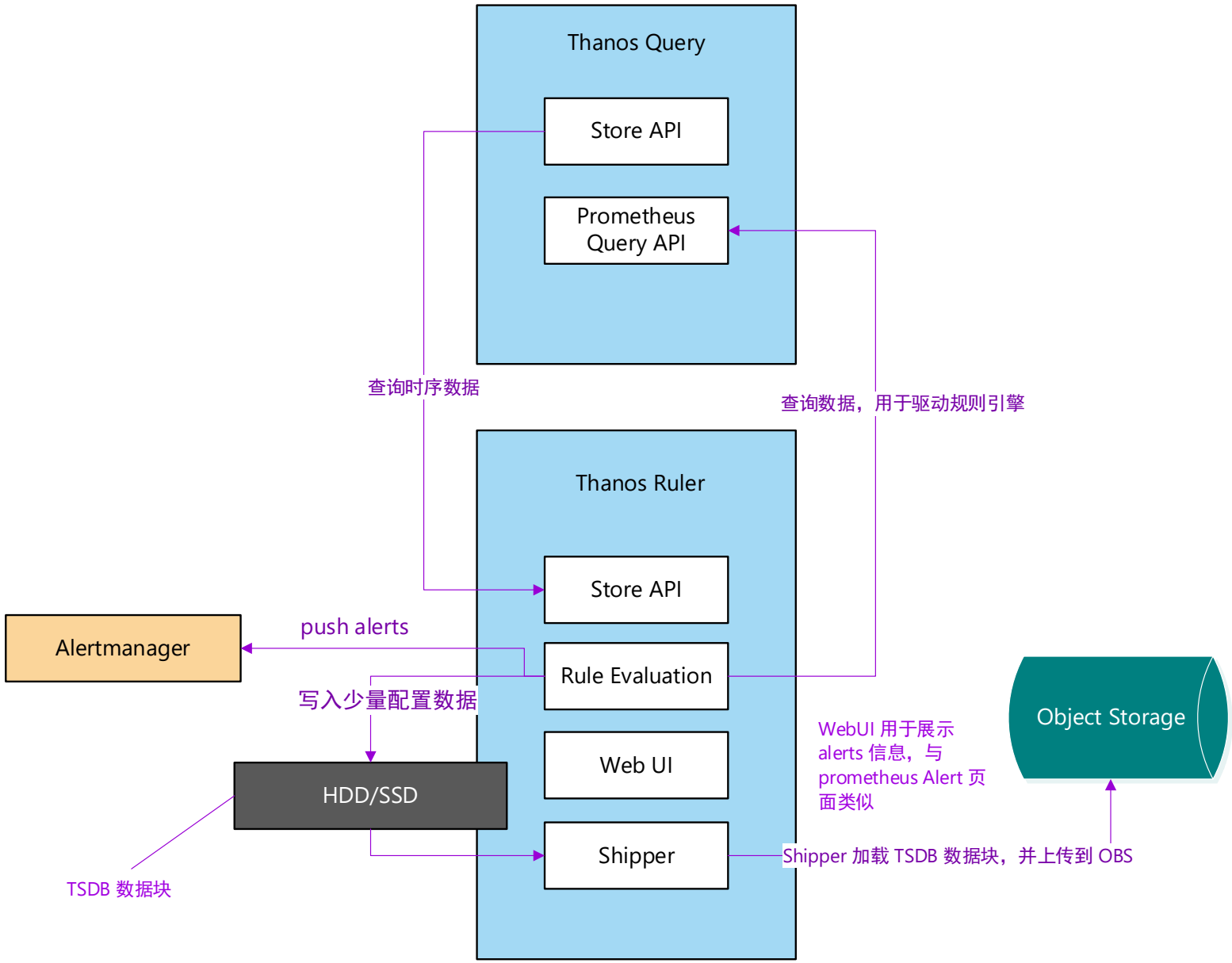
组件 —— Ruler

在实践中，Ruler 组件因为存在一些风险， Thanos 官方并不推荐使用，这里只作简单介绍。

- Ruler 组件本身即是数据源，它的实现大量借助了 Prometheus，这个数据源直接在 Prometheus 的 TSDB 基础上实现。
- 与 Prometheus-Server 不同的是，Ruler 的规则引擎并不基于本地数据实现，而是基于网络查询（比如，Prometheus-Server/Thanos Query 的 Query API），如果查询失败（超时等异常出现），规则引擎将会由失效的风险。
- Ruler 组件同样实现了 Store API 接口，提供 gRPC 的查询服务。
- 与 Store 组件类似，Ruler 组件提供了简单的 UI 页面用于查看基本的信息。
- Ruler 组件向下游组件（比如 Alertmanager）推送告警，其推送告警的 API 格式与 Prometheus-Server 相同。
- Ruler 组件提供 HTTP 服务，除了用于访问 UI 页面外，还用于暴露 metrics， debug 接口等。
- Ruler 支持对 Alertmanager 的自动发现。

通常，我们可以选择部署 Sidecar 模式的 Thanos ，或者使用 Ruler 方式进行部署，官方更推荐 Sidecar 模式，Ruler 一般在一些特殊需要的场景下使用。

组件 —— Ruler


Thanos Query
Store API
Prometheus Query API
查询时序数据
查询数据，用于驱动规则引擎
Thanos Ruler
Store API
Alertmanager
push alerts
Rule Evaluation
写入少量配置数据
Web UI
HDD/SSD
Shipper
TSDB 数据块
WebUI 用于展示 alerts 信息，与 prometheus Alert 页面类似
Object Storage
Shipper 加载 TSDB 数据块，并上传到 OBS

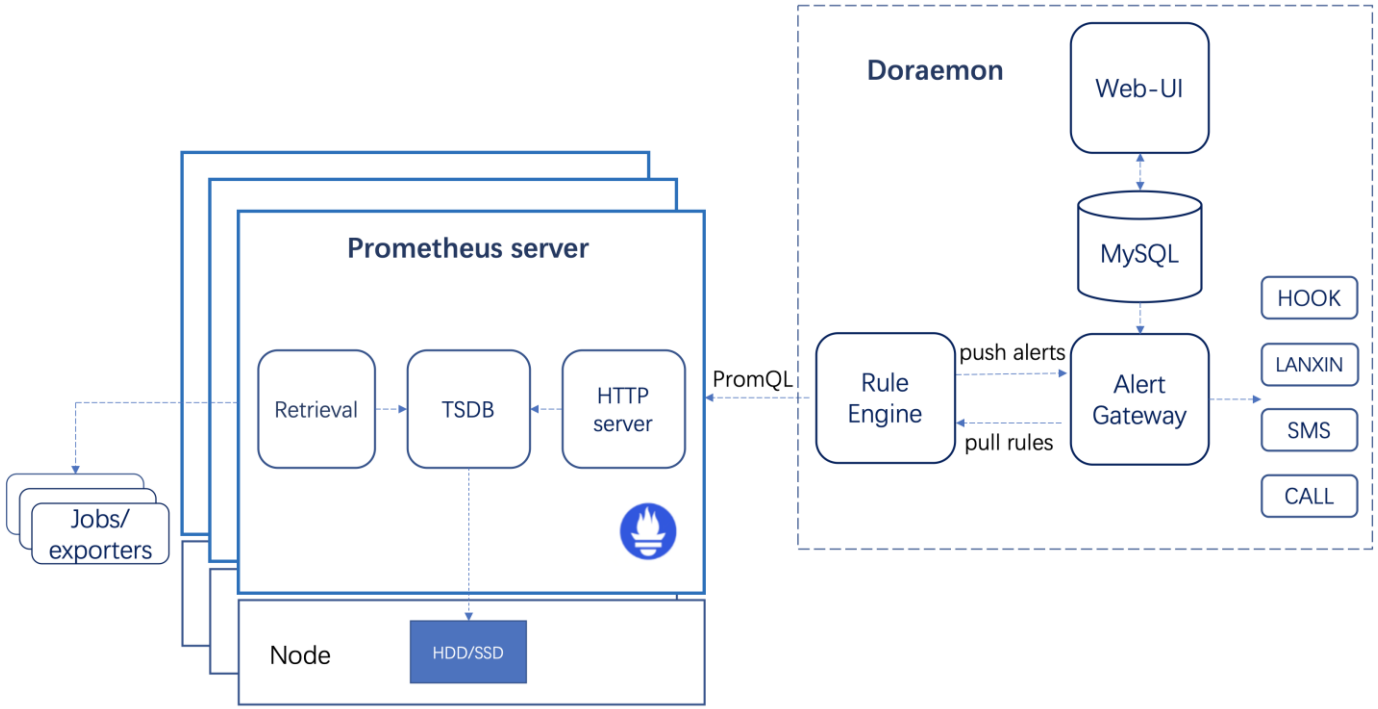
组件 —— OBS

Thanos

Thanos 支持多种对象存储服务，用来保存历史数据。其中 S3/Blob/GCS 为稳定版本，其他几种目前都仅为 Beta 版。

- Azure 对象存储服务 Blob
- 腾讯云对象存储服务 COS
- Google 对象存储 GCS
- 阿里云对象存储 OSS
- Amazon S3 存储
- 本地文件系统
- OpenStack Swift 存储

组件 —— Alertmanager 与 Doraemon



360 开源的 Doraemon 解决了 Alertmanager 配置维护困难的问题，其核心包括 3 部分，alert-gateway/rule-engine/frontend，采用 MySQL 保存元数据。

特性	Alertmanager	Doraemon
邮件告警	支持	不支持
电话告警	不支持	支持
IM 告警	支持企业微信等	支持钉钉
后端存储	插件式	Mysql
高可用	支持	不支持
水平扩展	支持	不支持
规则引擎	不支持	支持
WebUI	不支持	支持

Metric Types

- Counter
- Gauge
- Histogram
- Summary

Gauge 表示一个可以上下波动的值。

Counter 是一个单调递增的计数器。它不允许counter值下降，但是它可以被重置为 0（例如：客户端服务重启）。

Summary 通过时间滑动窗口抽样观察（通常是要求持续时间），并提供对其分布、频率和总和的即时观察。

Histogram 允许时间的可聚合分布，如：请求延迟。每个bucket中都会有一个count值, 表示累加的样本数量值。

Best practice

Metric 名称

- 一个度量指标名称有一个与单位所属的域相关的（单个单词）应用程序前缀。前缀有时成为客户端库的命名空间。对于特定于应用程序的度量指标，前缀通常是应用程序名称本身。
  - http_request_duration_seconds (所有HTTP请求)
    - 应该有一个描述单位的后缀，复数形式，node_memory_usage_bytes。
    - 累计数使用 total 作为后缀。
    - 适当避免使用 summary，对性能要求较高。

标签

- 使用标签来区分正在测量的事物特征。
  - api_http_requests_total - 区分请求类型： type=“created| update | delete”
  - api_request_duration_seconds - 区分请求阶段：stage="extract | transform | load“
- 不要将标签名称放在度量指标名称下面，因为这会导致冗余，如果响应的标签被聚合，会导致混淆。
- 不要使用标签来存储具有高基数（许多不同标签值）的维度，例如用户ID，电子邮件地址或其他无限制的值集合。

Questions

Thanos

Thank U !