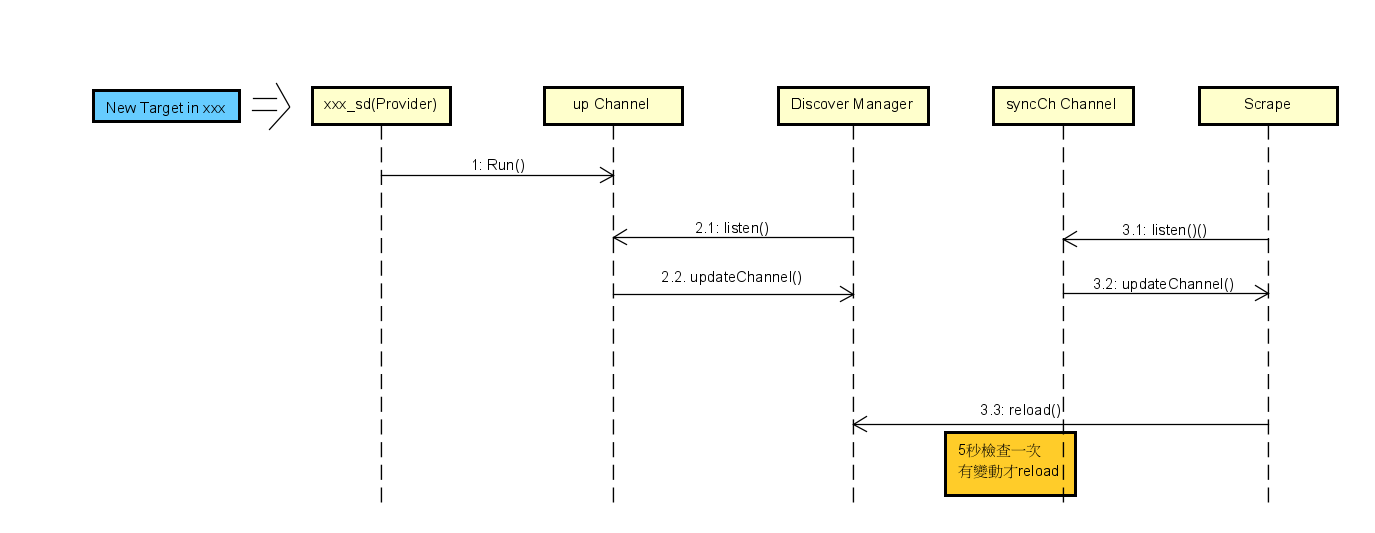
Service Discovery(服務發現，SD)

Prometheus抓取目標有靜態與動態兩種配置方法

… StaticConfigs (Job/Exporters) …由PULL組完成

若監控的目標時常改變，則每次都需修改靜態配置後再進行reload，為了避免這種情況，Prometheus開發了動態發現(Service Discovery, SD)，能感知目標的CRUD後自動reload，而不需重新配置再重啟server。取而代之，當SD的配置進行改動時(例如新增job)，prometheus需要reload來配置文件，與SD相關的資訊傳遞至discover manager，然後discover manager會和SD系統進行通訊，當使用者下達reload handler的指令時，會調用reloadConfig ，並依次調用相關function [[[1]](#footnote-1)][[[2]](#footnote-2)][[[3]](#footnote-3)]



SD機制的目的是發現target最後將資訊提供給Prometheus，適用多種平台。SD的Discoverer函式 [[[4]](#footnote-4)]，最開始會調用Run()將監控的target group發送至up channel，再用init()的discovery.RegisterConfig向discovery.Config進行註冊，接著可能將整組或是有所變動的target group(視平台而定)透過up channel發送給Manager進行處理。

// in prometheus/discovery/discovery.go

type Discoverer interface {

Run(ctx context.Context, up chan<- []\*targetgroup.Group)

}

// prometheus/discovery/kubernetes/kubernetes.go

func init() {

discovery.RegisterConfig(&SDConfig{})

……

}

Group是一群target的list，其擁有共同的LabelSet，LabelSet是一組名稱與值的map [[[5]](#footnote-5)]

type Group struct {

// Targets為主要標籤。e.g. {"\_\_address\_\_": "localhost:9090"}

Targets []model.LabelSet

// Label為次要標籤，唯一，可為空。e.g.{"foo": "bar", "bar": "baz"}

Labels model.LabelSet

// Source用以描述Group，唯一。e.g. "<source>"

Source string

}

Discover Manager

Manager會處理由up管道送來的資料，並透過poolKey和Source快速找到對應的target group陣列。

// in prometheus/discovery/manager.go

type Manager struct {

……

targets map[poolKey]map[string]\*targetgroup.Group // 發現的Targets

providers []\*provider // 持續監聽 provider

// 把發現的Targets透過channel傳遞給scrapeManager

syncCh chan map[string][]\*targetgroup.Group

……

}

type poolKey struct {

setName string // 對應配置文件的 job name

provider string // 提供者的類型(DNS, Azure, kubernetes等)

}

如此在Manager裡就可以拿到所有targets的資訊，供scrape元件使用，scrape主要功能是pull target， discover manager透過syncCh channel傳入scrape manager [[[6]](#footnote-6)]。

scrape manager會持續監聽syncCh channel裡的數據，一旦有新message傳入，scrape就會reload變動的target group。

// in prometheus/scrape/manager.go

func (m \*Manager) Run(tsets <-chan map[string][]\*targetgroup.Group) error {

go m.reloader()

……

// 透過syncCh獲取被監控的targets資料

case ts := <-tsets:

m.updateTsets(ts) // 將targets存儲到scrapeManager.targetSets

// 若有targets更動，發送至m.triggerReload並觸發reloader

select {

case m.triggerReload <- struct{}{}:

……

}

……

}

每5s觸發任務監聽m.triggerReload信號，執行m.reload() [[[7]](#footnote-7)]加載targets

// in prometheus/scrape/manager.go

func (m \*Manager) reloader() {

ticker := time.NewTicker(5 \* time.Second) // 5s計時器

……

// 若Manager裡有targets更動，會向m. triggerReload寫入值

// reloader每5秒觀察m.triggerReload是否有值，有則觸發m.reload()

case <-ticker.C:

select {

case <-m.triggerReload:

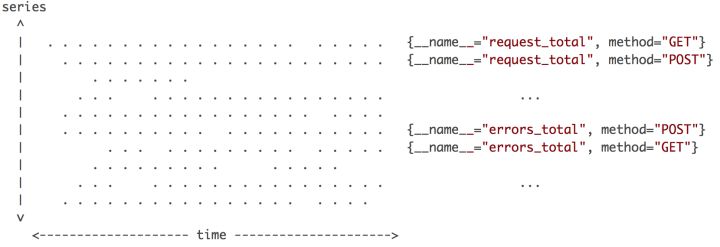
m.reload()

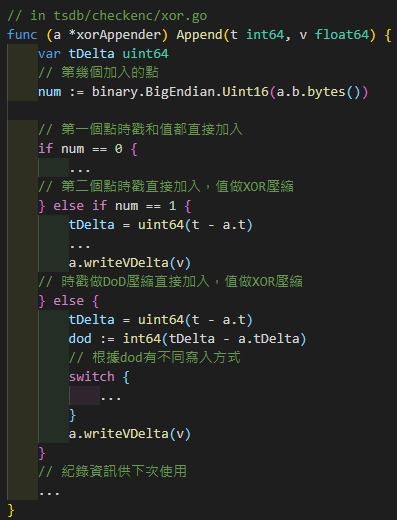
}

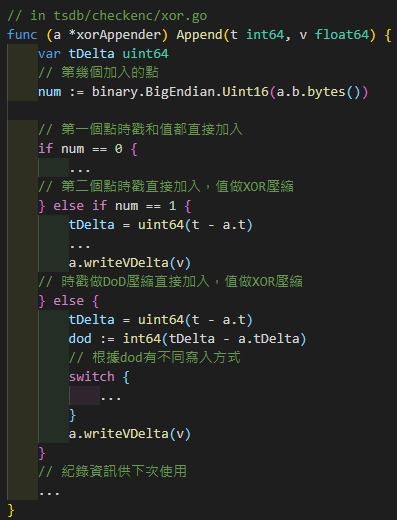
……

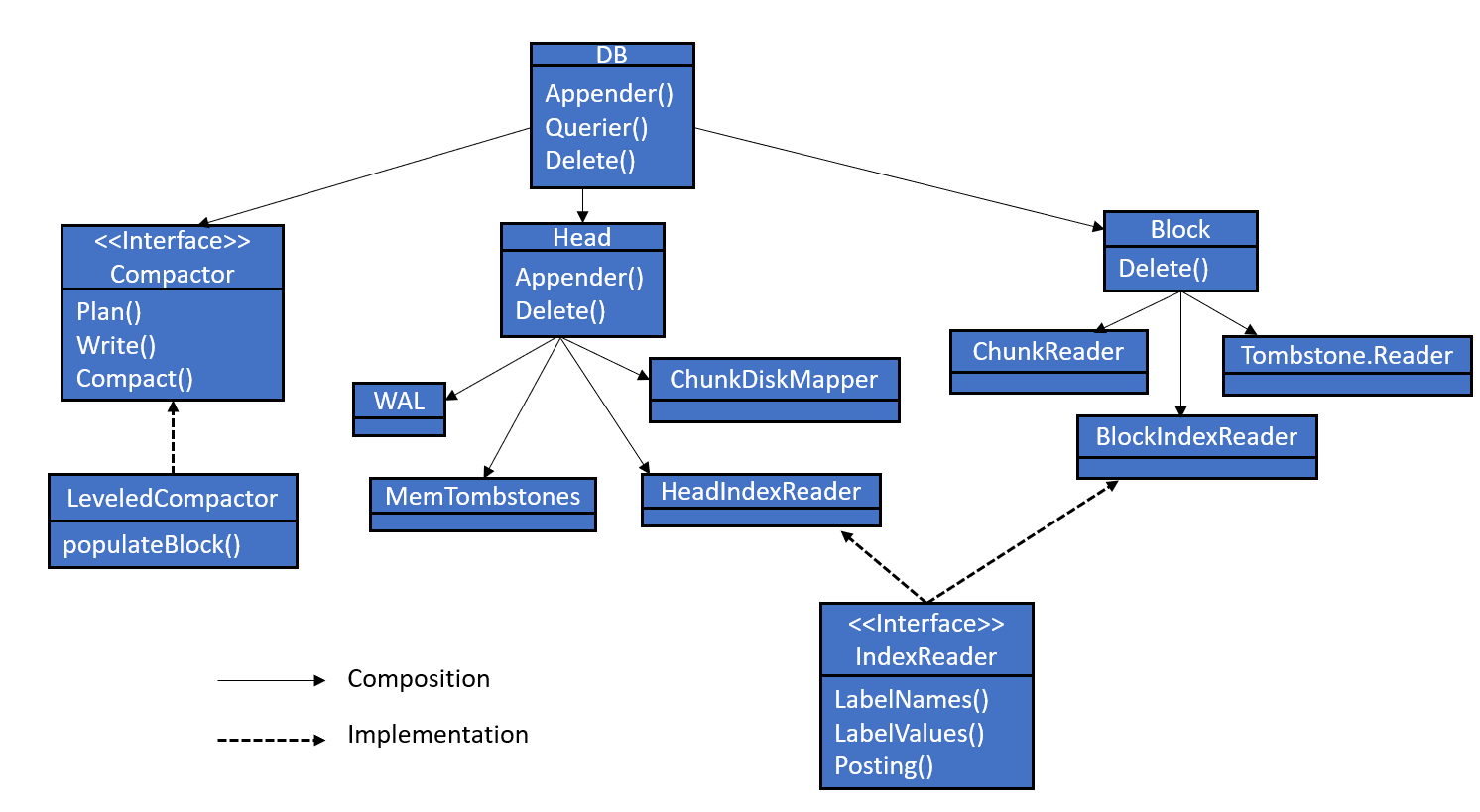
}。

TSDB

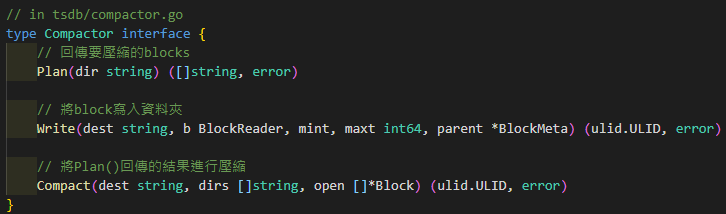


在tsdb中，同一時間會有多個series監控紀錄指定資源，一個series由數據點的list和label的list組成。數據點由一個時戳和一個值組成，而label為一key-value pair。將一個新的數據點加入series，而在加入時會先將時戳和值分別用DoD壓縮和XOR壓縮再加到series內。因為tsdb有以下特性，一、 相鄰數據點的時戳差距變化， 即使有浮動也僅在小範圍內(採樣間隔固定)；二、 相鄰數據點的value變化也很小，甚至有相當比例為0。因此採用此種壓縮能有效壓縮series的大小。

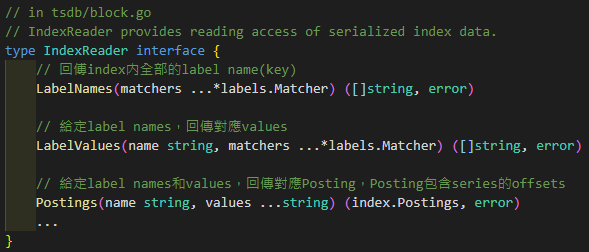




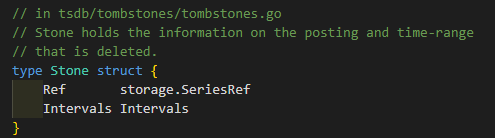
整個DB採用類似LSM algorithm(Log Structured Merge-tree)，主要由Head, Block, Compactor物件組成。Compactor會在下文詳述；Head指的是正在寫入的block，儲存在memory中，且因為在tsdb中有「越近期資料，越容易被查找」的特性，因此僅將Head存進memory既能減少空間也能加快查訪速度；Block是已經持久化且無法更動(immutable)的block。不管是head或是block，都以更小的chunk為單位保存在disk。

建立資料時，經過前述壓縮後除了寫到head block外，也會將未壓縮的資料寫到預寫式日誌(Write-ahead Logging)中，避免資料因為意外因素崩潰導致數據丟失。當head block的超過寫入時間，會將其持久化成一個level 0 block，當同一level的block夠多，就會壓縮(compact)成level更高的block。

讀取時， BlockQuerier根據不同結構產生headIndexReader或是blockIndexReader，index是倒序索引(inverted index)的資料結構，將給定的label name映射到目標所在chunk的offset，減少查找所需時間。



更新資料時，若目標在block中，因為block無法做更動，所以只能寫在head，在讀取時會以新的為主，而壓縮block時若有對同一資料進行寫入，會只有新的資料寫入至壓縮後的block。

刪除資料時，刪除紀錄會保存在目標所擁有的tombstone files，而非立即從block刪除。當整個block的資料都超過保留時間後，整個block就會被丟棄。

1. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L874-L909> [↑](#footnote-ref-1)
2. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L1159-L1201> [↑](#footnote-ref-2)
3. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L662-L748> [↑](#footnote-ref-3)
4. [] <https://github.com/prometheus/prometheus/tree/main/discovery> [↑](#footnote-ref-4)
5. [] <https://github.com/prometheus/common/blob/840c039c5fcce8204ed656bd75b084d2e9d80c1d/model/labelset.go#L28> [↑](#footnote-ref-5)
6. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L839-L859> [↑](#footnote-ref-6)
7. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/scrape/manager.go#L188-L216> [↑](#footnote-ref-7)