

問題(目的)與背景

Prometheus 是一個監控系統和警報框架(System Monitoring and Alarm Framework)，其可以處理數以百萬的監控指標、每秒可以處理數十萬的數據點，並將他們集中到一個時間序列資料庫(Time-Series Database)中，也能夠詳細列出監控的電腦資源。這些數據能在事前及時預警故障的發生及事後追查問題來源。

Prometheus 將所有記錄下來的資料經過資訊處理以後，在內建的 web UI 中以圖形或資料集的方式呈現，也可以配合外部的 GUI(e.g. Grafana)和警示系統，提供偵錯、視覺化警告及良好的擴充能力。

Prometheus 在 CAP 中捨棄了 consistency，因為每個節點是自主的，不同 node 可能有不同檔案，也因此有著去中心化的特點。Prometheus 並不適合 100% 準確度的情境，但適合監控以機器為主或高度動態的服務導向架構。

在什麼場合用來解決什麼問題

Prometheus 適用於高度動態的系統，例如雲端的微服務，其中通訊軟體 Line 在後端也使用了 Prometheus^[1]，用以幫助收集一定時間內量化的指標數據，例如每秒查詢率(QPS)、每秒事務(TPS)等，這些數據可以幫助監控 Line Today, Line 購物，Line 音樂，Line 旅遊等。Line 的 SRE 團隊(Site Reliability Engineering，網站可靠性工程)除了使用系統監控警報框架 Prometheus 搭配 Kubernetes 的 CRD(Custom Resource Definitions，自定義資源，可在不修改 Kubernetes 的程式碼上擴展其功能)自動抓取指標，也使用監控服務 Thanos 作為資料收集的後端儲存方案，達到長期儲存資料的效果。

沒有他又如何

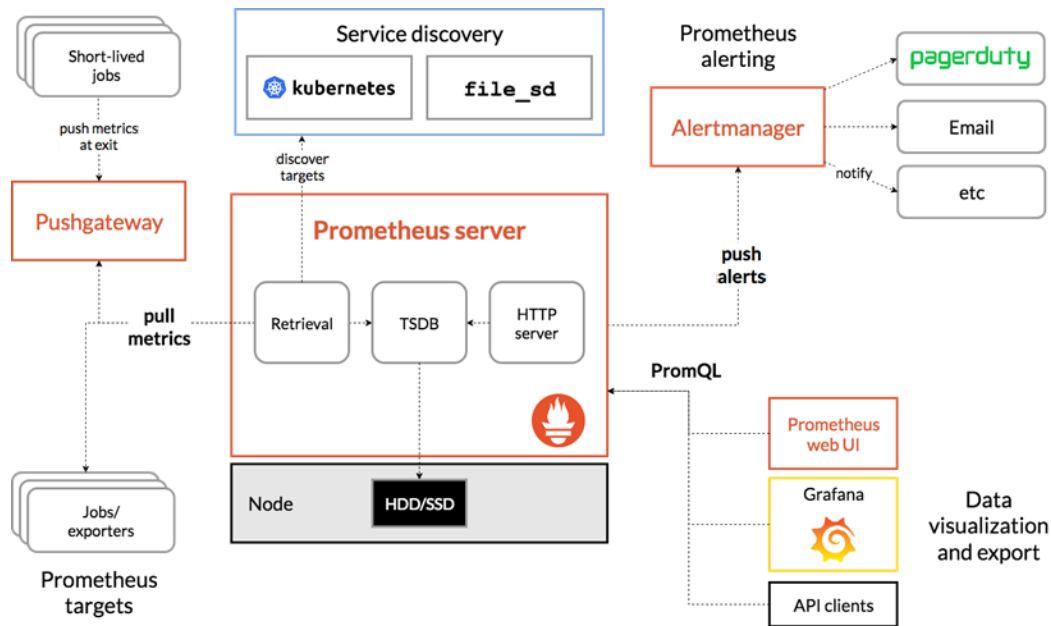
若無法使用 Prometheus 會失去什麼優勢呢？使用市面上的其它監控系統 ELK 做為比較的對象，Prometheus 可以做到蒐集指標並設置警報，而 ELK 只能對蒐集到的資料作分析和搜索，Prometheus 可以使用 http pull method 取得資料，ELK 則只能透過各處的 log 取得，在數據儲存方面，Prometheus 用的是命名的時間序列，而 ELK 使用各種不同的資料類形像是 int string bool 等等。

架構解析

^[1] <https://www.ithome.com.tw/news/149317>

圖 1：Prometheus 官方文檔架構圖

靜態與動態結構解析



此處只專解析資料如何 pull 進 Server 的架構，Retrieval system 主要的功能是以 pull 拉取的方式取得指定的監控目標 target 資訊後儲存的模組，定義在 Scrape 模組中，包含三個檔案 manager.go、scrape.go、target.go(圖 2)。每一個目標 target 有一個與之對應的循環 loop，每個 loop 內部執行 Http Get 請求拉取數據。每一個 job 有一個與之對應的 scrape pool，通過一些控制參數，執行周期性數據採集及結束等運作。

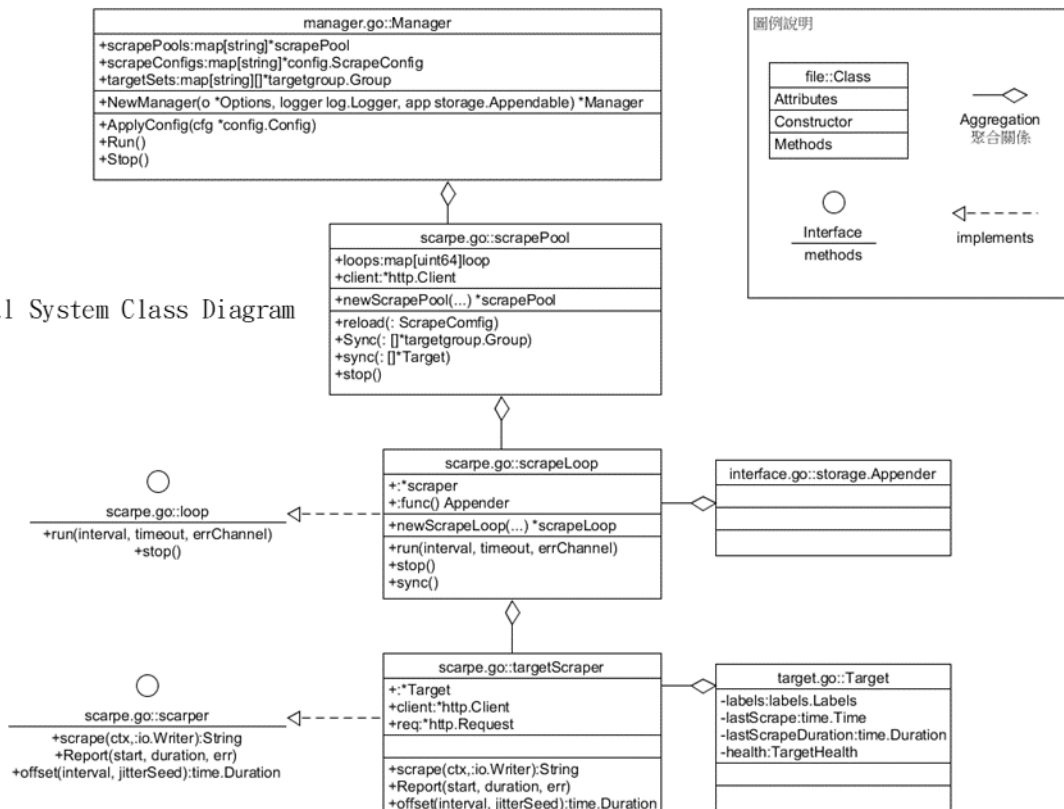


圖 2：Retrieval System Class Diagram

manager.go 包含

Manager 類別:由 discovery manager 取得數據採集目標，記錄在 scrape pools 之中，並負責啟動及停止週期性的資料採集任務。Manager 類別主要的屬性有管理目標資料採集的 scrapePools，採集設定 scrapeConfig，目標集合 targetSets。

- ApplyConfig()方法使用新的設定 cfg 重置管理的目標提供者和作業配置。
- Run()方法接收並儲存目標集數據更新，並觸發抓取循環來重新加載 reloader()，重新加載為背景執行，不會阻止接收目標數據更新，reload()則為每個 job 建立對應的 scrape pool。
- Stop()方法會取消所有正在運行的抓取池和區段，直到完全結束。

scrape.go 包含

scrapePool 類別：管理目標集合的資料採集，主要屬性有執行資料拉取的 loops 以及 Http 端點的 client。

- reload()方法使用給定的抓取配置重新加載抓取池，目標狀態被保留，但所有抓取循環都使用新的抓取配置重新啟動。
- Sync(tgs []*targetgroup.Group)方法將目標群組轉換為實際的抓取目標，將當前運行的抓取器與結果集同步，並傳回所有抓取和刪除的目標。
- sync(targets []*Target)方法對一個目標列表進行同步處理，新目標啟動抓取循環，失效的目標則停止抓取循環。
- stop()方法終止所有抓取循環。

scrapeLoop 類別：實作 loop 介面的資料拉取循環物件，可以進行啟動 run()及停止 stop()呼叫，停止後不得重複使用。主要屬性有拉取資料的 Scraper 以及附加器 Appender。

targetScraper 類別：實作 scraper 介面的目標拉取物件，以 Http Get 執行請求，呼叫 scrape()抓取資料，Report()回報資料，offset()時間偏移量設置。

target.go 包含

Target 類別:代表單一 HTTP 或 HTTPS 的端點。

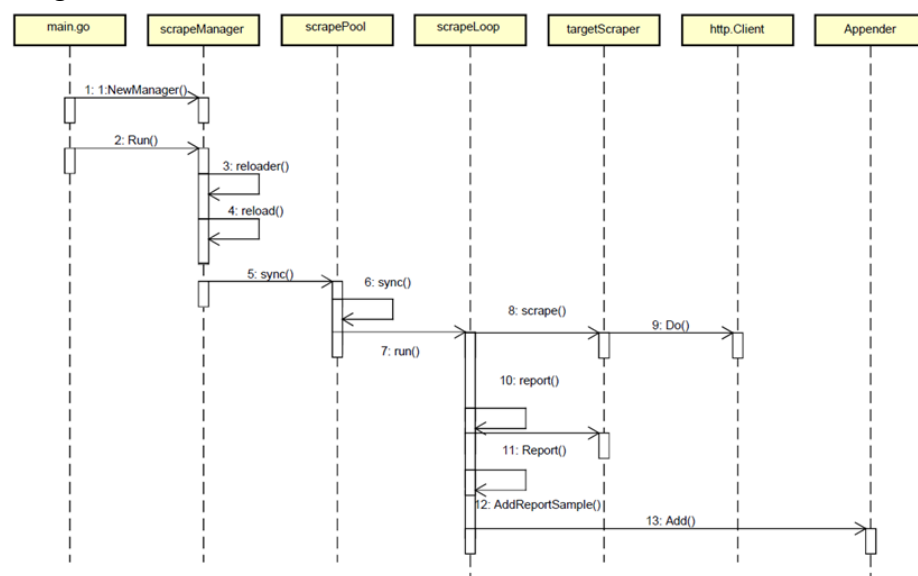


圖 3：Scrape Sequence Diagram

Scrape 時序圖(圖 3)分析

1. main.go 中使用 scrape.NewManager 建立 ScrapeManager，ScrapeManager 中使用 Map 結構來儲存管理目標資料採集的 scrapePools，採集配置 scrapeConfig，目標集合 targetSets；Prometheus 中，將一個獨立的數據來源（target）稱之為 instance。包含相同類型的 instance 的集合稱之為 job，Manager 中 ApplyConfig 可抓取配置，使用 job 為 key 查詢 scrapeConfig Map 結構，若 job 不存在，將其從 ScrapePool 中刪除，若配置被更改，清理歷史配置，啟動 reload 會將新配置新增到 ScrapePool。
2. 建立 ScrapeManager 之後，main.go 呼叫 scrapeManager.Run()以啟動 ScrapeManager，Run()方法中先執行 reloader()加載 targets，targets 更新時會觸發 reload()，加載完成後呼叫 Sync()將當前的抓取器與結果集同步。
3. Manager 的 reloader()方法用來加載 targets，reloader 的加載發生為背景執行，不會影響 target 的更新。
4. Manager 的 reload() 執行重新加載，為 targets 集合中每一個 target 資料來源生成一個對應的 scrape pool 來管理其運作，由 targetSets 中取得所有的 Job，若處理 Job 的對應 scrape pool 不存在於 Scrape_pools 之中，則讀取 Job 對應的 scrape config 配置以建立 scrape pool，並將其儲存於 Scrape_pools Map 之中，為提高效率，此流程採並行運行，需先用 sync.Mutex 鎖定，遍歷 TargetGroup 將所有 scrape pool 創建完成後將 sync.Mutex 打開，並等待其他並行工作運行完成。
5. 呼叫 Scrape Pool 的 Sync()方法，會將當前運行的抓取器與 All 集合同步，sync.Mutex 鎖定後，遍歷所有 TargetSet 群組中的 Target，有效的 Target 加入 All 集合，無效的 Target 加入 droppedTargets 集合，接著呼叫 sync()方法將 All 集合中 Target 轉換為實際的抓取器。
6. Scrape Pool 的 sync()方法所取得的 Target 的列表 all 可能重複，需將重複刪除，並為新目標啟動抓取循環，並為消失的目標停止抓取循環。完成後對每一個不重複的 Scrape Loop 呼叫 run() 方法執行抓取循環。
7. Scrape Loop 的 run() 方法執行抓取循環，依據每個 Scrape Loop 定義的時間週期，定期呼叫 scrapeAndReport()方法執行抓取。scrapeAndReport()方法會執行抓取，依序觸發執行 targetScraper 的 scrape()方法、http:Clinet 的 Do()方法、ScrapeLoop 的 report()方法。然後將結果與報告指標附加記錄到附加器 Appender 的末端。抓取動作可能使用到一個以上的 Appender，Scrape Loop 會盡可能使用較少的 appender。
8. targetScraper 的 scrape()方法透過 HTTP Get 請求來抓取數據，先建立 HTTP Request 請求物件，設定適當請求標頭，透過 http:Clinet 的 Do()方法來發起請求。
9. 執行 http:Clinet 的 Do()方法，傳回抓取資訊。ScrapeLoop 的 report()方法會呼叫 targetScraper 的 Report()方法抓取的訊息，並針對結果使用

addReportSample()方法做處理或錯誤回報。

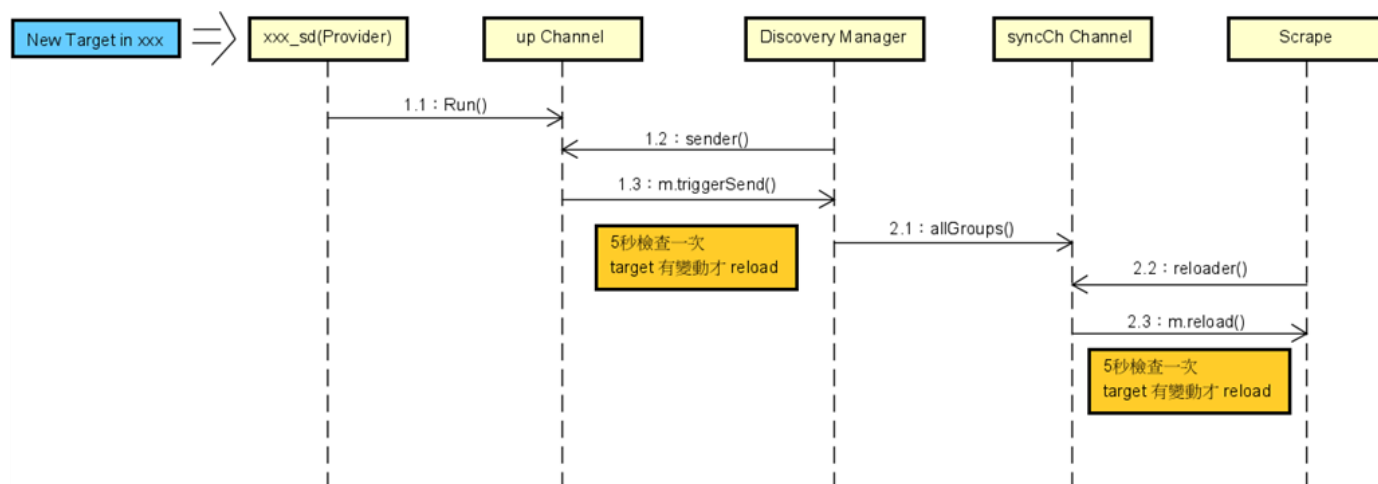
10. 執行 targetScraper 的 Report()方法，回報關於最後一次抓取的目標數據
11. addReportSample()方法呼叫 Appender Add()方法將資料寫入儲存體。
12. 執行 Appender Add()方法，storage.Appender 針對儲存體提供資料批次附加的邏輯。必須通過調用 Commit 或 Rollback 來完成，之後不得重用。

以下僅針對 Service Discovery 的程式碼進行解讀：

Service Discovery(服務發現，SD)

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。與 SD 相關的資訊傳遞至 discovery manager，然後 discovery 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 ^{[2][3][4]} (圖 4)。但當 SD 的配置進行改動時(例如新增 job)，prometheus 還是需要手動 reload 配置文件並重啟 server。

圖 4：Service Discovery Sequence Diagram



SD 機制的目的是發現 provider(e.g. DNS, Azure, Kubernetes)的 target 並最終將資訊提供給 Prometheus 監控。SD 的 Discoverer 介面(圖 5)提供給各 Provider 介接 ^[5]，最開始會調用 Run()(圖 6)將監控的 target group 發送至 up channel，再用 init()的 discovery.RegisterConfig(圖 7)進行註冊，接著可能將整組或是有更動(視 provider 而定)的 target group 透過 up chan 發送給 discovery Manager 處理。

^[2] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L874-L909>

^[3] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L1159-L1201>

^[4] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/cmd/prometheus/main.go#L662-L748>

^[5] <https://github.com/prometheus/prometheus/tree/main/discovery>


```
// in prometheus/discovery/discovery.go
type Discoverer interface {
    Run(ctx context.Context, up chan<- []*targetgroup.Group)
}
```

圖 5：Discoverer Interface

```
// in prometheus/discovery/manager.go
func (sd *StaticProvider) Run(ctx context.Context, ch chan<- []*targetgroup.Group) {
    .....
    case ch <- sd.TargetGroups: // 將target group發送至channel
    .....
}
```

圖 6：Run()本體

```
// prometheus/discovery/kubernetes/kubernetes.go
func init() {
    discovery.RegisterConfig(&SDConfig{})
    .....
}
```

圖 7：init()內的 discovery.RegisterConfig

Group(圖 8)是一群 target 的 list，其擁有共同的 LabelSet，LabelSet 是一組名稱與值的 map^[6]

```
// in prometheus/discovery/targetgroup/targetgroup.go
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
}
```

圖 8：Group struct

Discovery Manager

Manager(圖 9)會處理由 up Chan 送來的資料(監聽與獲取資料方式與 scrape 處理 sync Chan 雷同，故此略)，並透過 poolKey(對應配置文件的 job name)和 provider 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
    .....
}
```

圖 9：Manager

如此在 discovery manager 裡就可以拿到所有 targets 的資訊，接著 discovery manager 透過 allGroups()^[7]傳入 syncCh，scrape manager 會持續監聽 syncCh，一旦有新 message 傳入，scrape 就會 reload 變動的 target group(圖 10)，之後 scrape 就會開始 pull target 進入 TSDB。

^[6] <https://github.com/prometheus/common/blob/840c039c5fcce8204ed656bd75b084d2e9d80c1d/model/labelset.go#L28>

^[7] <https://github.com/prometheus/prometheus/blob/6555cc68caf8d8f323056e497ae7bb1e32a81667/discovery/manager.go#L381-L399>

```
// 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{}{}:
            .....
        }
    }
}
```

圖 10：scrape 監聽 syncCh

每 5s 監聽 m.triggerReload 信號，執行 m.reload()^[8]加載 targets(圖 11)。

```
// 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()
        }
    }
    .....
}
```

圖 11：Manager

TSDB

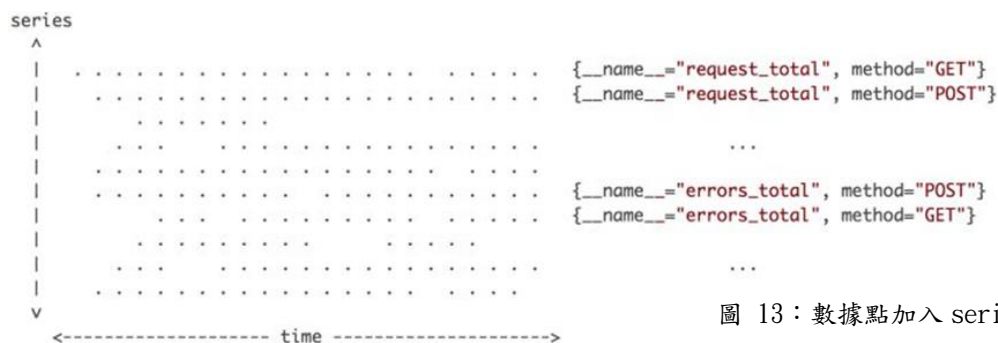


圖 12：series 紀錄

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 壓縮再加入到 series 內(圖 13)。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。

```
// 第一個點時戳和值都直接加入
if num == 0 {
    ...
    // 第二個點時戳直接加入，值做XOR壓縮
} else if num == 1 {
    tDelta = uint64(t - a.t)
    ...
    a.writeVDelta(v)
    // 時戳做DoD壓縮直接加入，值做XOR壓縮
} else {
    tDelta = uint64(t - a.t)
    dod := int64(tDelta - a.tDelta)
    // 根據dod有不同寫入方式
    switch {
    ...
    }
    a.writeVDelta(v)
}
// 紀錄資訊供下次使用
...
}
```

圖 13：數據點加入 series

[8] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/scrape/manager.go#L188-L216>

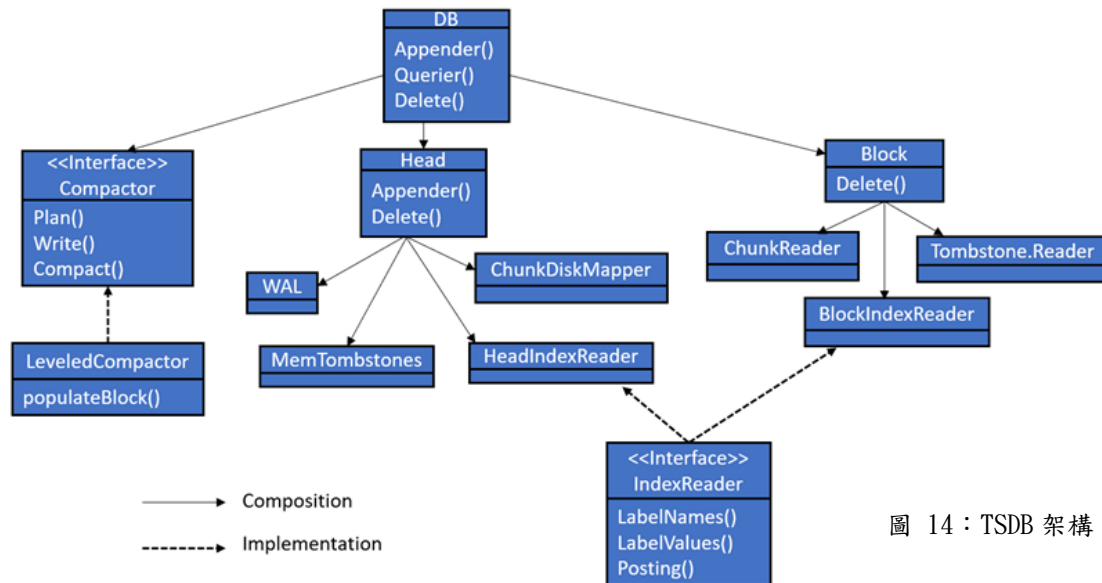


圖 14：TSDB 架構

整個 DB(圖 14)採用類似 LSM algorithm(Log Structured Merge-tree)，主要由 Head, Block, Compactor(圖 15)物件組成。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。

圖 15：Compactor Interface

```

// in tsdb/compactor.go
type Compactor interface {
    // 回傳要壓縮的blocks
    Plan(dir string) ([]string, error)

    // 將block寫入資料夾
    Write(dest string, b BlockReader, mint, maxt int64, parent *BlockMeta) (ulid.ULID, error)

    // 將Plan()回傳的結果進行壓縮
    Compact(dest string, dirs []string, open []*Block) (ulid.ULID, error)
}
  
```

建立資料時，為避免資料因為意外崩潰導致數據丟失，除了經過前述壓縮後寫進 head block，也會將未壓縮的資料寫入至預寫式日誌(Write-ahead Logging)中。當寫入 head block 超過寫入時間時，會將其持久化成一個 level 0 block；當同一 level 的 block 夠多，Compactor 就會壓縮成 level 更高的 block。

讀取時，BlockQuerier 根據不同結構產生 headIndexReader 或是 blockIndexReader(圖 16)，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。

圖 16：IndexReader Interface

```
// in tsdb/block.go
// IndexReader provides reading access of serialized index data.
type IndexReader interface {
    // 回傳index內全部的label name(key)
    LabelNames(matchers ...*labels.Matcher) ([]string, error)

    // 給定label names，回傳對應values
    LabelValues(name string, matchers ...*labels.Matcher) ([]string, error)

    // 給定label names和values，回傳對應Posting，Posting包含series的offsets
    Postings(name string, values ...string) (index.Postings, error)
    ...
}
```

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(圖 17)，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。

圖 17：Stone struct

```
// in tsdb/tombstones/tombstones.go
// Stone holds the information on the posting and time-range
// that is deleted.
type Stone struct {
    Ref          storage.SeriesRef
    Intervals Intervals
}
```

此設計如何在某個背景下解決問題

為了可以觀察監控時的數據，使其不再是黑箱作業，以及簡單的提供網路大範圍監控，採用了 exporter 和 pull 的機制，先在需要被監控的機器上安裝好 exporter，接著在 server 端進行 pull，就可以簡化 client 端的安裝複雜度(簡單擴充性)和避免傳統 push 方法需要耗費大量資源去聯繫每個 client 的缺點。此外使用 metric 和 tsdb 來將資料整合，使用 metric 能夠讓資料模組化，因此能夠使用圖形或資料集的方式呈現，而 tsdb 則讓資料能夠更長時間的存在磁碟當中，並且使資料擁有時間性。

技術實作

我們尚未對 docker、API 等等的用法深度用法進行了解，所以這次實作採用的是 windows 系統的 server。Windows 的 server 架設非常簡單，首先到官網下載 server 的 exe 檔案，下載後雙擊執行，預設的位址是 <http://localhost:9090>，這代表 Prometheus 會定時抓取 local 端的資源進行紀錄，抓取時間預設為 15s 抓取一次，如果想要更改 port 或是抓取資料的間隔，可以在 prometheus.yml 的檔案中更改 scrape_interval: 15s(圖 18)和- targets: ["localhost:9090"] (圖 19)。

```
1 # my global config
2 global:
3   scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
4   evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
5   # scrape_timeout is set to the global default (10s).
```

圖 18

```
20 # Here it's Prometheus itself.
21 scrape_configs:
22   # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
23   - job_name: "prometheus"
24
25     # metrics_path defaults to '/metrics'
26     # scheme defaults to 'http'.
27
28   static_configs:
29     - targets: ["localhost:9090"]
30
```

圖 19

Server 架設好之後，就可以連上 <http://localhost:9090> 去做查詢，Prometheus 的查詢方法是使用他們創建好的 API 叫做 promql，因為 prometheus 有網頁的 UI 介面，所以可以直接在搜尋欄中使用 promql 選擇我們想要查詢的東西，如果對網頁通訊比較熟悉的使用者，也可以直接使用網址來做搜尋(圖 22)。這裡以 prometheus_target_interval_length_seconds 為例子(圖 20)

Execute

圖 20

按下執行後，下方就會呈現出本地端電腦的資源(圖 21)

Table	Graph	Load time: 5ms Resolution: 1s Result series: 5
Evaluation time		
<div>< Evaluation time ></div>		
prometheus_target_interval_length_seconds{instance="localhost:9090", interval="1s", job="prometheus", quantile="0.01"}		0.9853734
prometheus_target_interval_length_seconds{instance="localhost:9090", interval="1s", job="prometheus", quantile="0.05"}		0.9925888
prometheus_target_interval_length_seconds{instance="localhost:9090", interval="1s", job="prometheus", quantile="0.5"}		0.999166
prometheus_target_interval_length_seconds{instance="localhost:9090", interval="1s", job="prometheus", quantile="0.9"}		1.0098832
prometheus_target_interval_length_seconds{instance="localhost:9090", interval="1s", job="prometheus", quantile="0.99"}		1.0141845

Remove Panel

圖 21

附上網址的差異

localhost:9090/graph?g0.expr=&g0.tab=0&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h

localhost:9090/graph?g0.expr=prometheus_target_interval_length_seconds&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=5m

圖 22

使用心得

Prometheus 的 server 操作並不複雜，能查詢的東西也非常多，例如：
cpu_time、http_request_total 等等，也可以運用 promql 來選擇要查詢的區間，像是瞬時、區間、純量等等，除此之外，Prometheus 還提供了基本的 Web UI 圖形介面，讓使用者查看資料的分布情形。

評論

Pros

- base-PULL，無須安裝代理，部署簡單。
- Prometheus 只要添加 label 就能輕易抓取 K8S cluster(其可用於部署和管理多機器內的多個 container)的指標。
- 支援許多客戶端 library 和第三方 Exporter。
- 去中心化，針對單節點儲存使其操作簡單。
- 可視化(GUI)和監控警報對初學者較容易上手。
- 使用 metric 讓資料更有模組性。

Cons

- 內建的 query 採用 PromQL 語言，其需綁定 Grafana，需要額外進行 Grafana^[9]的設置。
- 不支援 clustered storage，對於監控指標的數量有先天限制。
- 監控系統的通病，CAP 中的可用性(availability)優先於一致性(consistency)，可能導致部分副本數據丟失。
- 可擴展性(scalability)差，不適合儲存龐大或長期資料^[10]，大規模使用需要第三方 component(e.g. Thanos, Cortex, etc.)才能實現^[11]，這些第三方 component 也都有各自的問題(不展開討論)。
- 儲存指標而非儲存 log，需要第三方的 log 傳送和分析工具才能保存 log(e.g. Loki, ELK stack, etc.)。

結語

Prometheus 的框架非常龐大，在架構解析途中也時常迷失自我，這時就會想起老師上課教導的方法 —— 記得想要追尋的功能 -> 不斷的提問 -> 猜測 -> 驗證想法，同時配合一些官方文檔，最後才得以成功完成這份報告。這次的報告不僅讓我們了解到 Prometheus 的一些基本操作與內容，也學會如何針對大型系統的特定功能進行描述與解讀，並為期末專案做前期鋪墊。

^[9] <https://grafana.com/>

^[10] <https://prometheus.io/docs/prometheus/latest/storage/>

^[11] https://www.youtube.com/watch?v=3pTG_N8yGSU