# Prometheus architecture

圖 1 ：Prometheus官方文檔架構圖

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。

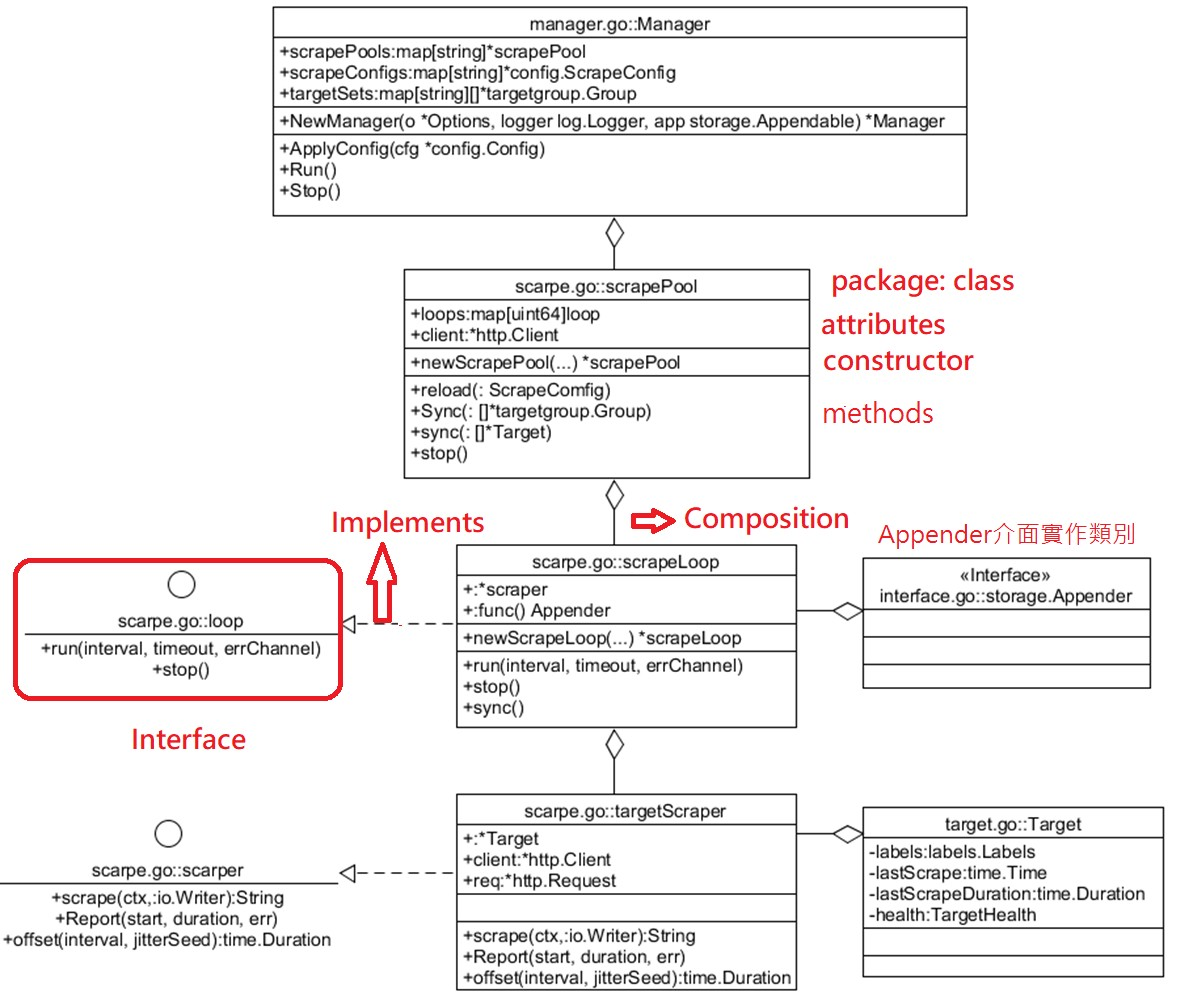


圖 2：Retrieval System Class Diagram

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。

NewManager() 是Manager的建構子

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。

newScrapePool()是scrapePool建構子。

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裡，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scarpe()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。

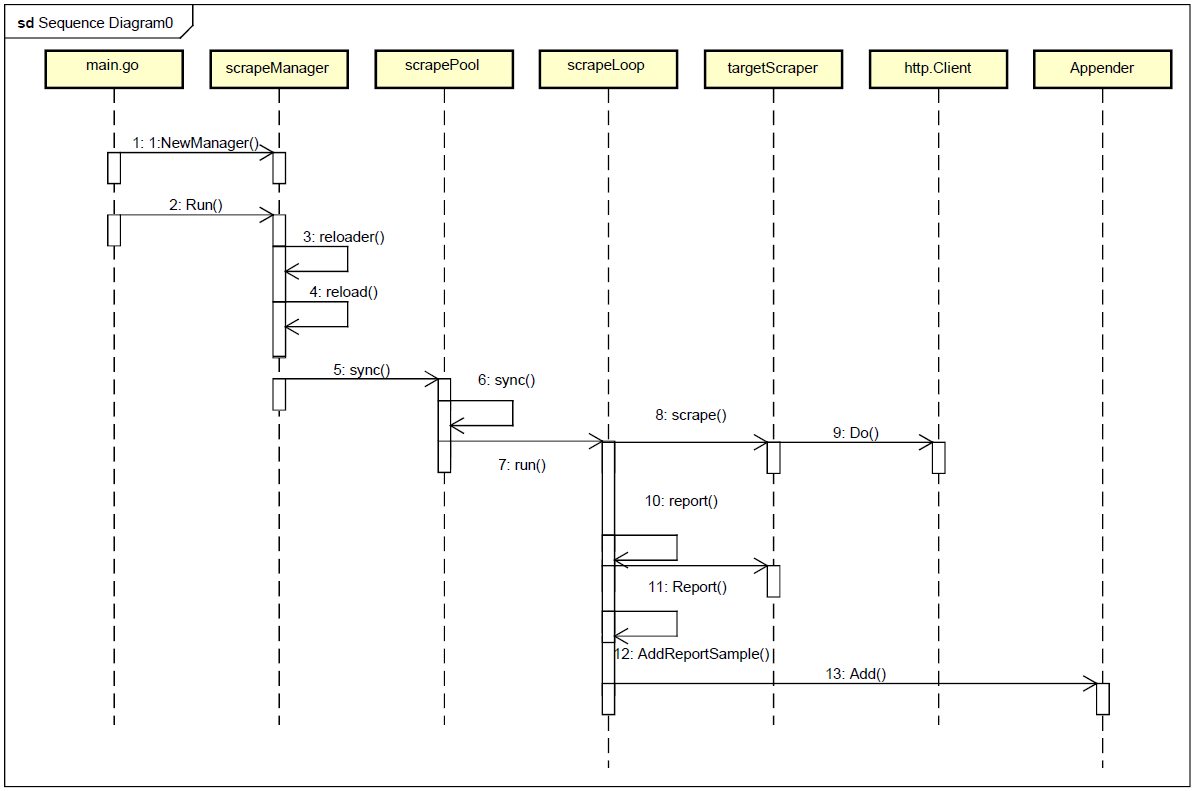


圖 3：Scrape Sequence Diagram

Scrape時序圖(圖 3)分析

1. main.go中使用scape.NewManager建立ScrapeManager，ScapeManager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）稱之爲instance。包含相同類型的instance的集合稱之爲job，Manager中ApplyConfig可抓取配置，使用job爲key查詢scrapeConfig Map結構，若job不存在，從ScrapePool中刪除，若配置被更改，清理歷史配置，啟動reload將新配置至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，爲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scarpe 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創建完成後釋放鎖，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。
10. 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。
11. 執行targetScraper的Report()方法，回報關於最後一次抓取的目標數據
12. addReportSample()方法呼叫Appender Add()方法將資料寫入儲存體。
13. 執行Appender Add()方法，storage.Appender 針對存儲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。

以下針對Service Discovery的程式做解析：

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 [[[1]](#footnote-1)][[[2]](#footnote-2)][[[3]](#footnote-3)] (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。

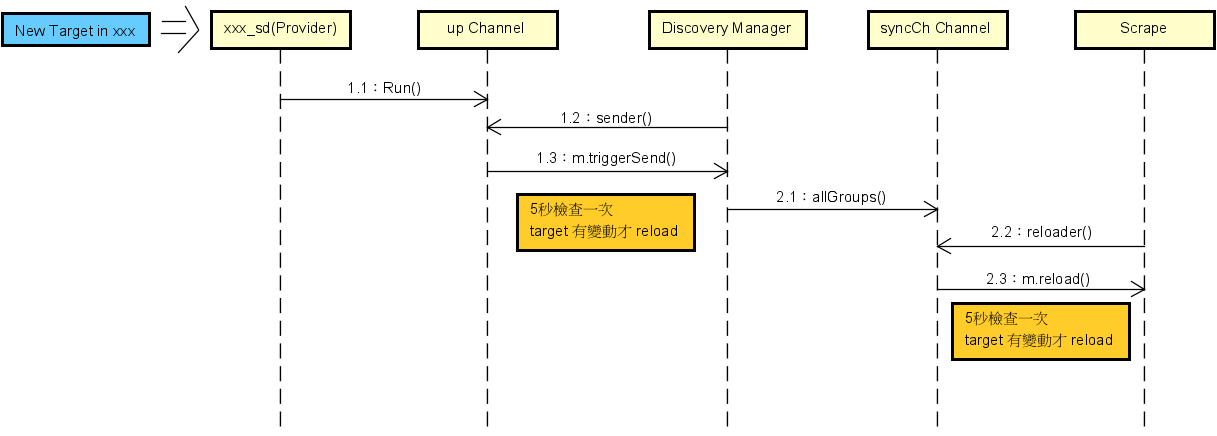


圖 4：Service Discovery Sequence Diagram

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 [[[4]](#footnote-4)]，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。

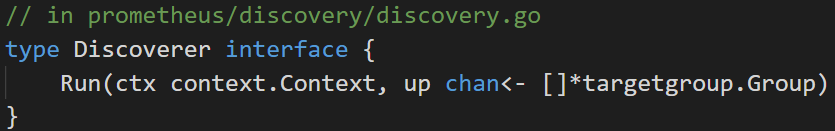


圖 5：Discoverer Interface

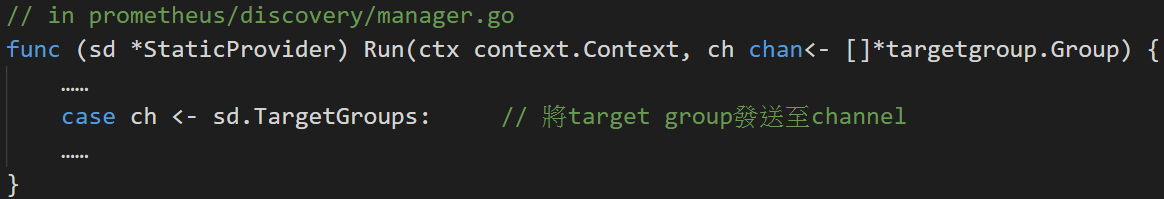


圖 6：Run()本體

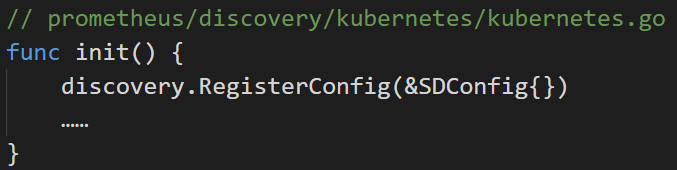


圖 7：init()內的discovery.RegisterConfig

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 [[[5]](#footnote-5)]

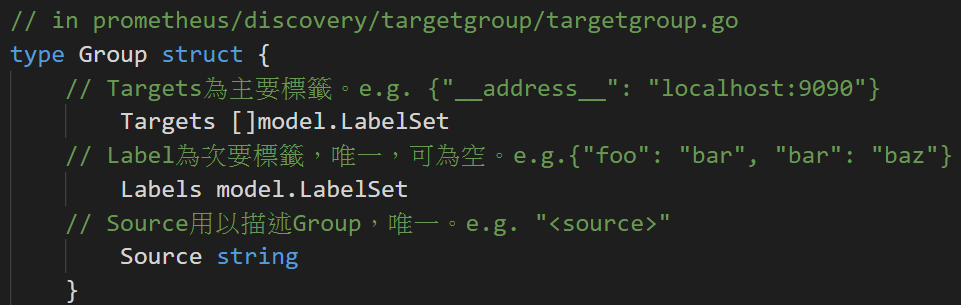


圖 8：Group struct

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。

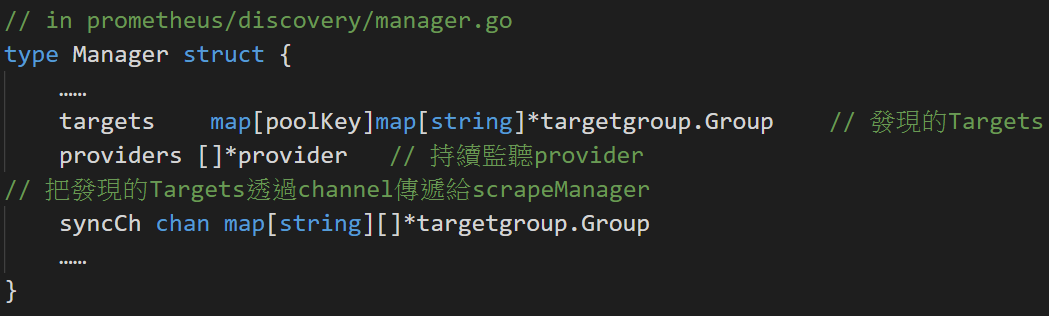


圖 9：Manager

如此在discovery manager裡就可以拿到所有targets的資訊，接著discovery manager透過allGroups() [[[6]](#footnote-6)]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。



圖 10：scrape監聽syncCh

每5s監聽m.triggerReload信號，執行m.reload() [[[7]](#footnote-7)]加載targets(圖 11)。

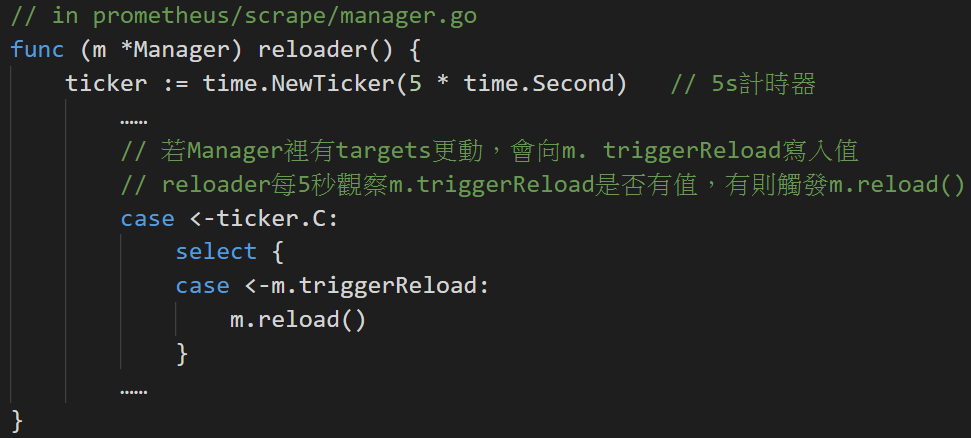


圖 11：Manager

TSDB

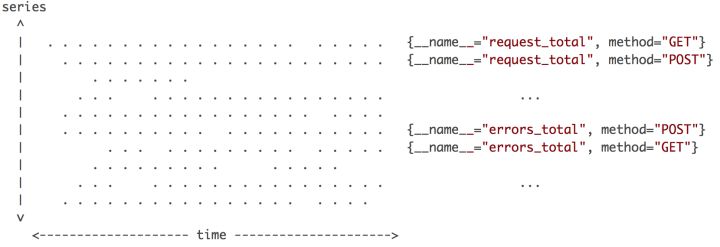
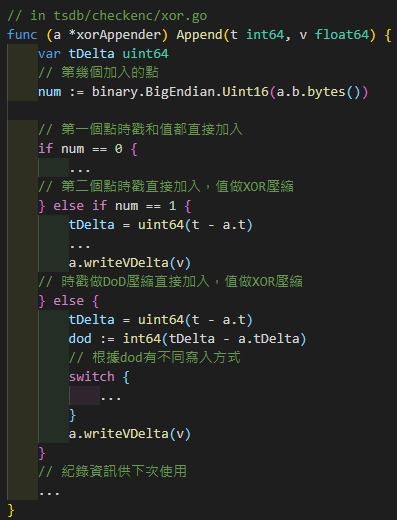


圖 12：series 紀錄

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。

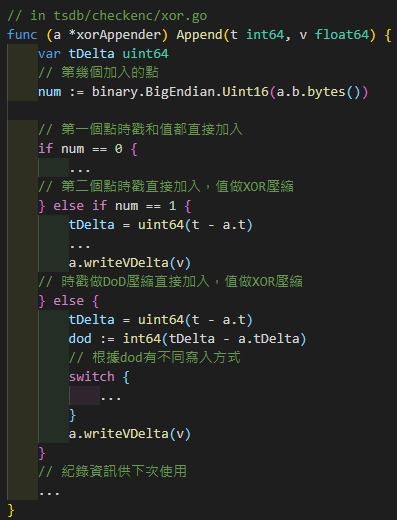


圖 13：數據點加入series

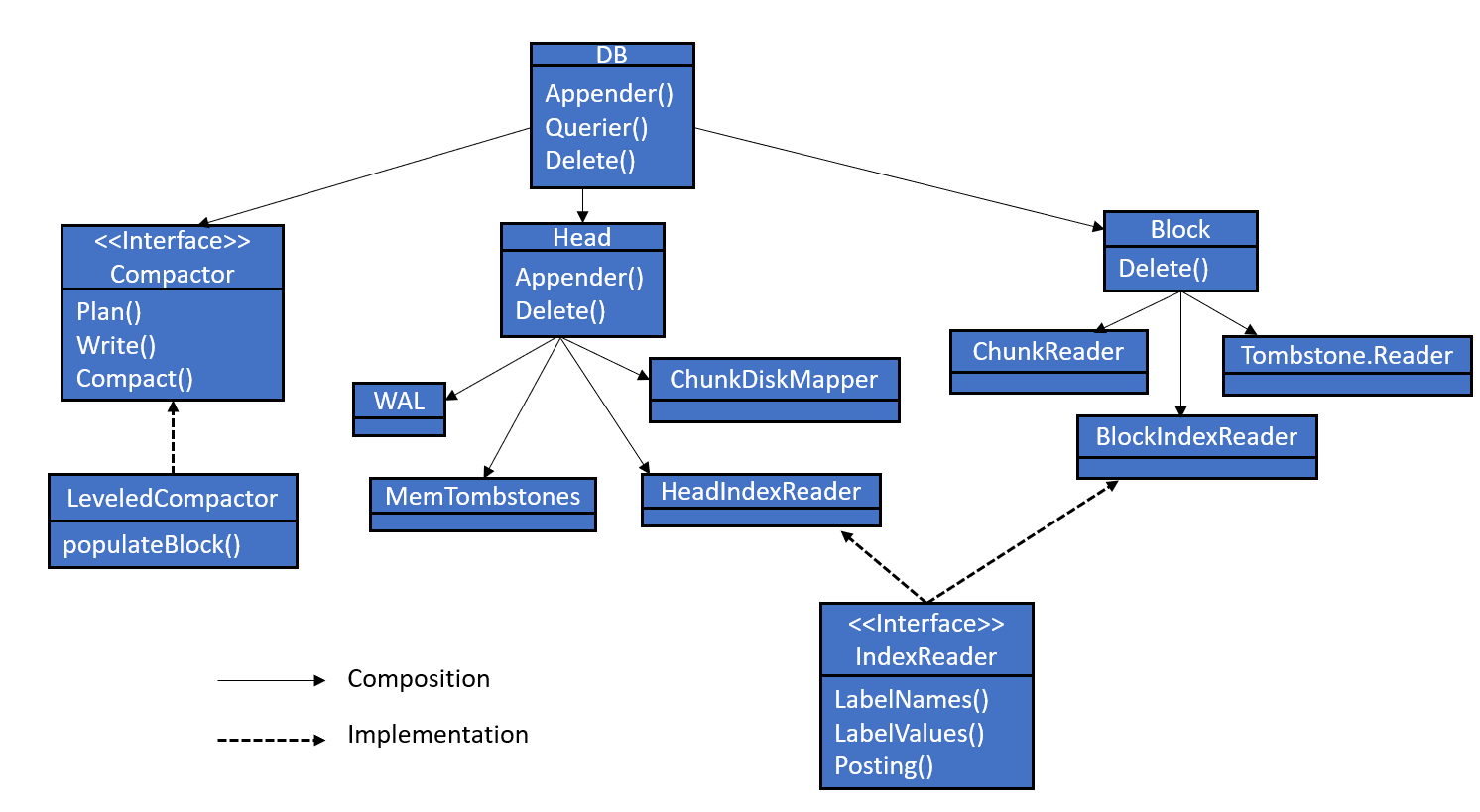


圖 14：TSDB架構

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。

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。

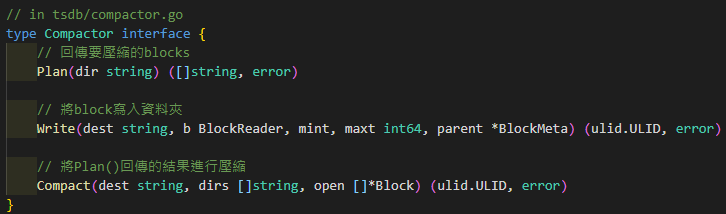


圖 15：Compactor Interface

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。

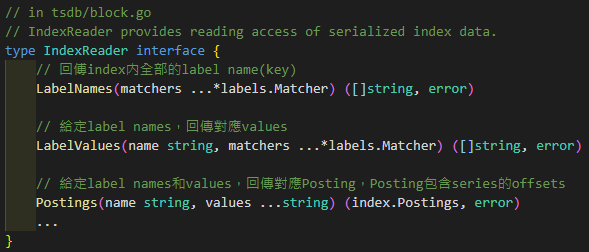


圖 16：IndexReader Interface

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。

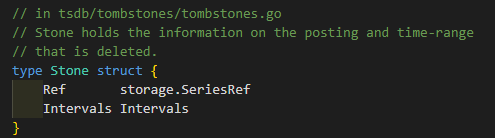
刪除資料時，刪除紀錄會保存在目標所擁有的tombstone files(圖 17)，而非立即從block刪除。當整個block的資料都超過保留時間後，整個block就會被丟棄。

圖 17：Stone struct

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/6555cc68caf8d8f323056e497ae7bb1e32a81667/discovery/manager.go#L381-L399> [↑](#footnote-ref-6)
7. [] <https://github.com/prometheus/prometheus/blob/063319087c122b3b296cc630d93f577dac31fd1c/scrape/manager.go#L188-L216> [↑](#footnote-ref-7)