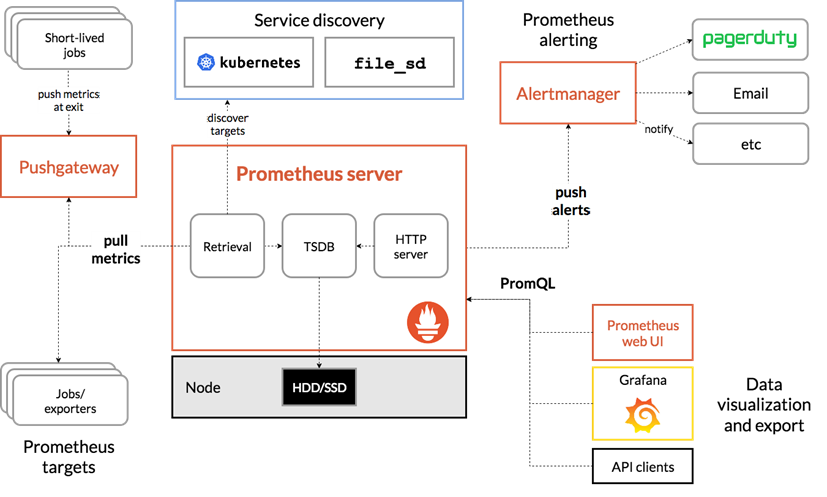
問題(目的)與背景

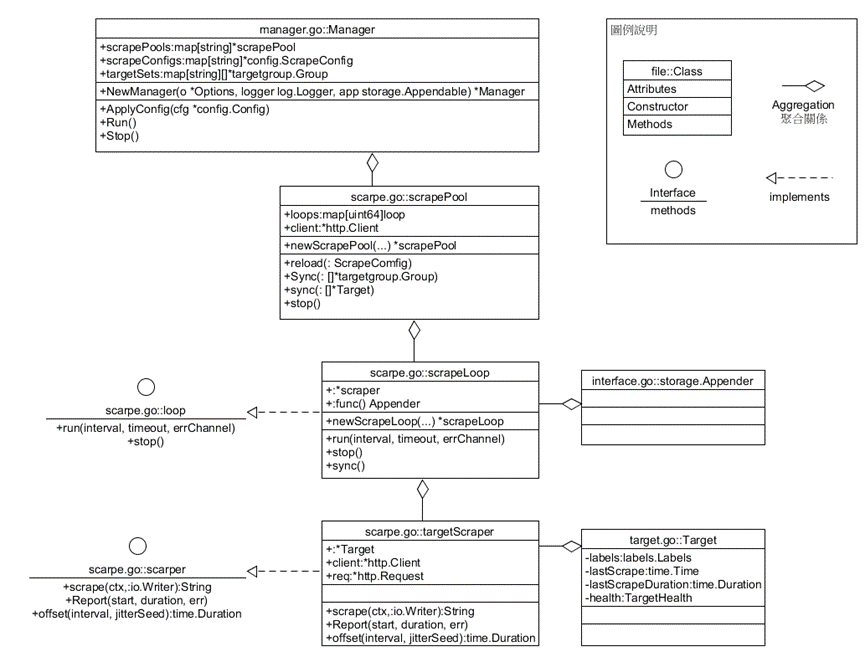
Prometheus是一個監控聚合器(Monitoring Aggregator)，它可以處理數以百萬的監控指標、每秒可以處理數十萬的數據點，並將他們集中到一個時間序列(Time-Series)資料庫中，也能夠詳細列出電腦資源。在nagios等較舊型的監控系統中，常需要第三方的後台程式協助溝通，而這些第三方程式通常需要複雜的環境配置，也導致擴充能力低下，但Prometheus不需要額外的第三方程式協助溝通，因此配置簡單。另外，在nagios的警示只有0正常、1警告、2錯誤、3未知錯誤這四種狀態提示，而Prometheus將所有紀律下來的資料經過資訊處理以後，在內建的web UI中以圖形或資料集的方式呈現，也可以配合外部的GUI和警示系統，提供偵錯及視覺化警告，提供良好的擴充能力。

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，也因此有者去中心化的特點。如果使用情境需要100%準確度那Prometheus並不適合，但適合以機器為主的監控和監控高度動態的服務導向架構。

架構解析



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](#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。



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。

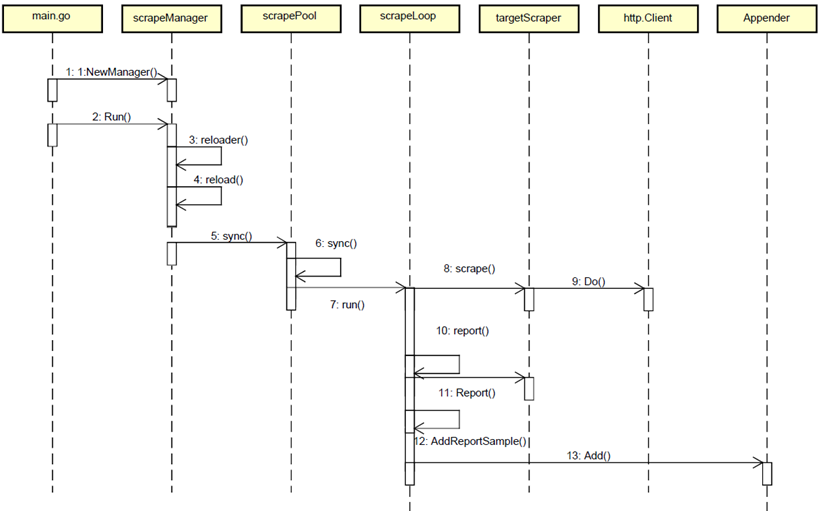
stop()方法終止所有抓取循環。

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。

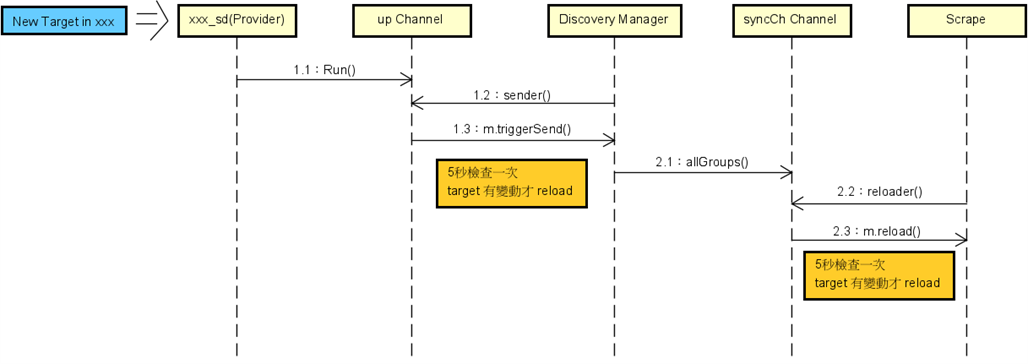


Scrape時序圖([圖3](#圖3))分析

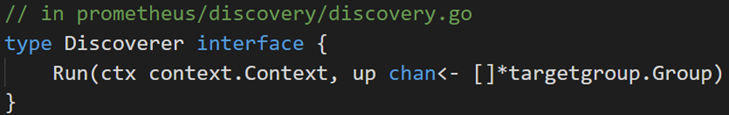
1. main.go中使用scape.NewManager建立ScrapeManager，ScapeManager中使用Map結構來儲存管理目標資料採集的scrapePools, 採集配置scrapeConfig，目標集合targetSets；Prometheus中，將一個獨立的數據來源（target）稱之爲instance。包含相同類型的instance的集合稱之爲job，Manager中ApplyConfig可抓取配置，使用job爲key查詢scrapeConfig Map結構，若job不存在，從ScrapePool中刪除，若配置被更改，清理歷史配置，啟動reload將新配置至ScrapePool。
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。

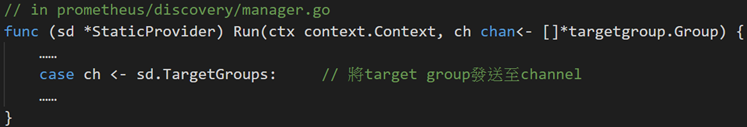
Service Discovery(服務發現，SD)

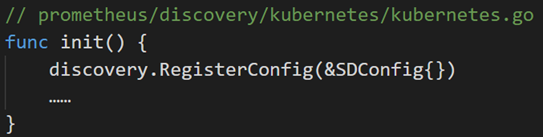
若監控的目標時常改變，則每次都需修改靜態配置後再進行reload，為了避免這種情況，Prometheus開發了動態發現(Service Discovery, SD)，能感知目標的CRUD後自動reload，而不需重新配置再重啟server。與SD相關的資訊傳遞至discovery manager，然後discovery manager會和SD系統進行通訊，當使用者下達reload handler的指令時，會調用reloadConfig ，並依次調用相關function [[[1]](#footnote-1)][[[2]](#footnote-2)][[[3]](#footnote-3)] ([圖4](#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。



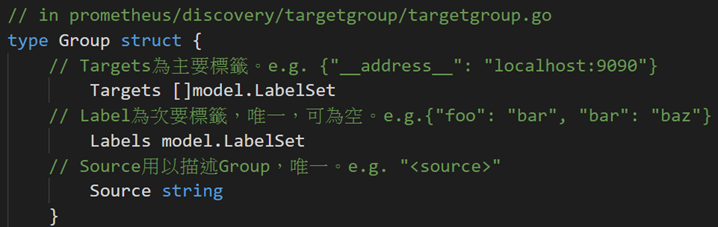
SD機制的目的是發現provider(e.g. DNS, Azure, Kubernetes)的target並最終將資訊提供給Prometheus監控。SD的Discoverer介面([圖5](#圖5))提供給各Provider介接 [[[4]](#footnote-4)]，最開始會調用Run()([圖6](#圖6))將監控的target group發送至up channel，再用init()的discovery.RegisterConfig([圖7](#圖7))進行註冊，接著可能將整組或是有更動(視provider而定)的target group透過up chan發送給discovery Manager處理。







Group([圖8](#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)]



Discovery Manager

Manager([圖9](#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。

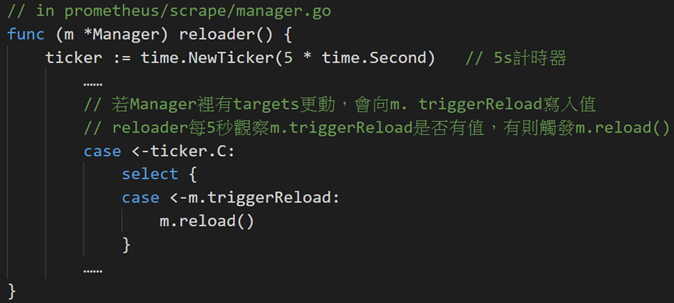


如此在discovery manager裡就可以拿到所有targets的資訊，接著discovery manager透過allGroups() [[[6]](#footnote-6)]傳入syncCh，scrape manager會持續監聽syncCh，一旦有新message傳入，scrape就會reload變動的target group([圖10](#圖10))，之後

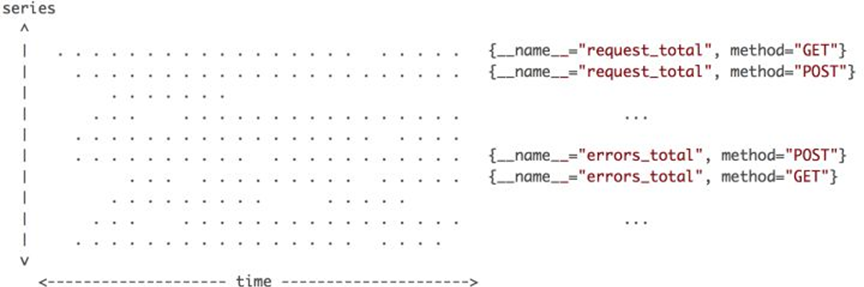
scrape就會開始pull target進入TSDB。

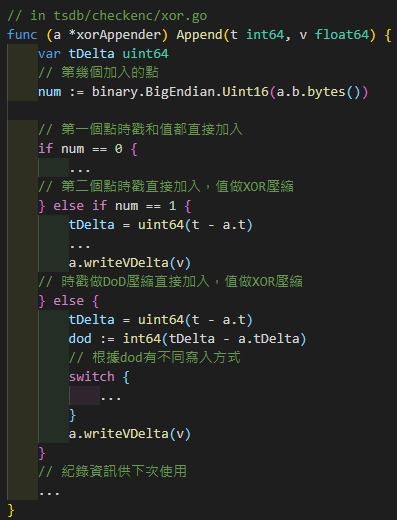


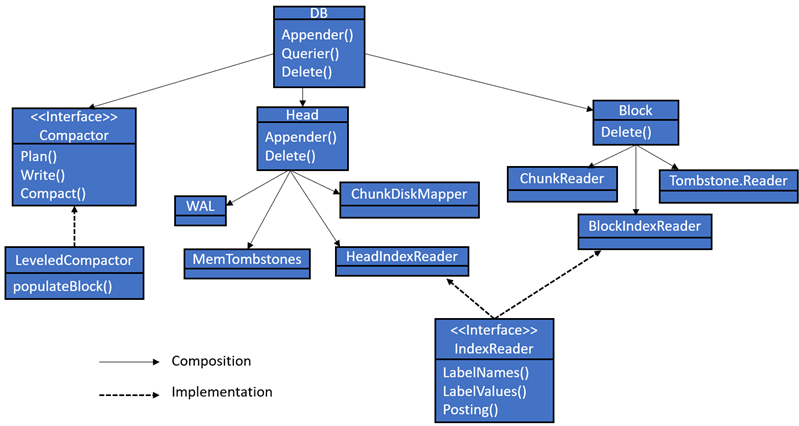
每5s監聽m.triggerReload信號，執行m.reload() [[[7]](#footnote-7)]加載targets([圖11](#圖11))。

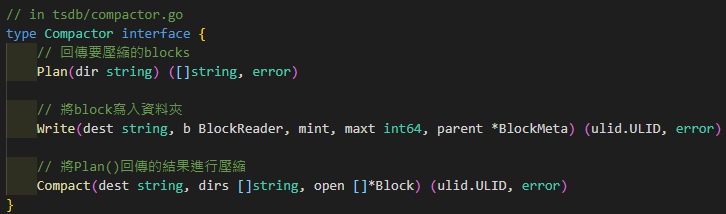


TSDB



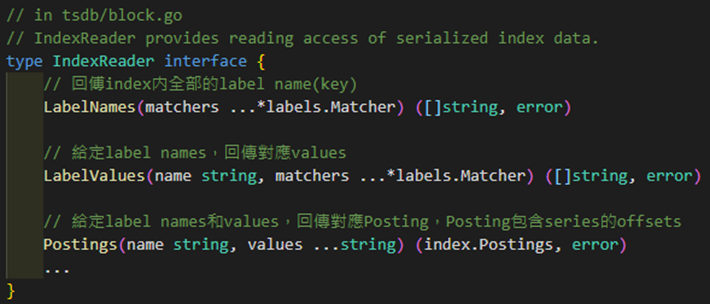
在tsdb中，同一時間會有多個series監控紀錄指定資源，一個series由數據點的list和label的list組成。數據點由一個時戳和一個值組成，而label為一key-value pair。將一個新的數據點加入series，而在加入時會先將時戳和值分別用DoD壓縮和XOR壓縮再加到series內([圖13](#圖13))。因為tsdb有以下特性，一、 相鄰數據點的時戳差距變化， 即使有浮動也僅在小範圍內(採樣間隔固定)；二、 相鄰數據點的value變化也很小，甚至有相當比例為0。因此採用此種壓縮能有效壓縮series的大小。



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

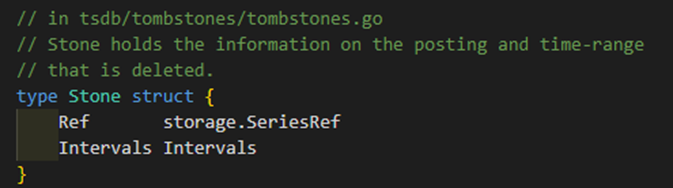
建立資料時，為避免資料因為意外崩潰導致數據丟失，除了經過前述壓縮後寫進head block，也會將未壓縮的資料寫入至預寫式日誌(Write-ahead Logging)中。當寫入head block超過寫入時間時，會將其持久化成一個level 0 block；當同一level的block夠多，Compactor就會壓縮成level更高的block。

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



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](#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。



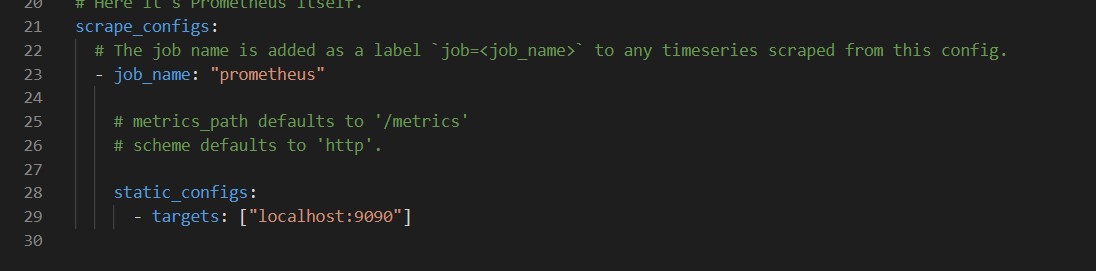
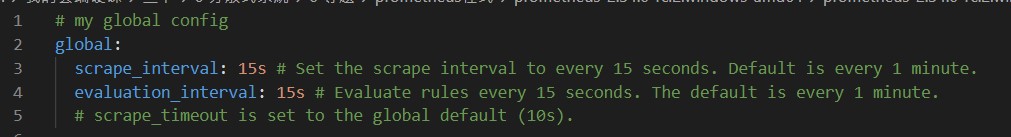
此設計如何解決問題(簡單擴充、大量資料集、資料模組化)

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，就可以達到簡單的高可擴充性和避免傳統push方法，需要耗費大量資源去聯繫每個client的缺點，並且使用metric和tsdb來將資料整合，使用metric能夠讓資料模組化，因而更好的使用圖形或

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

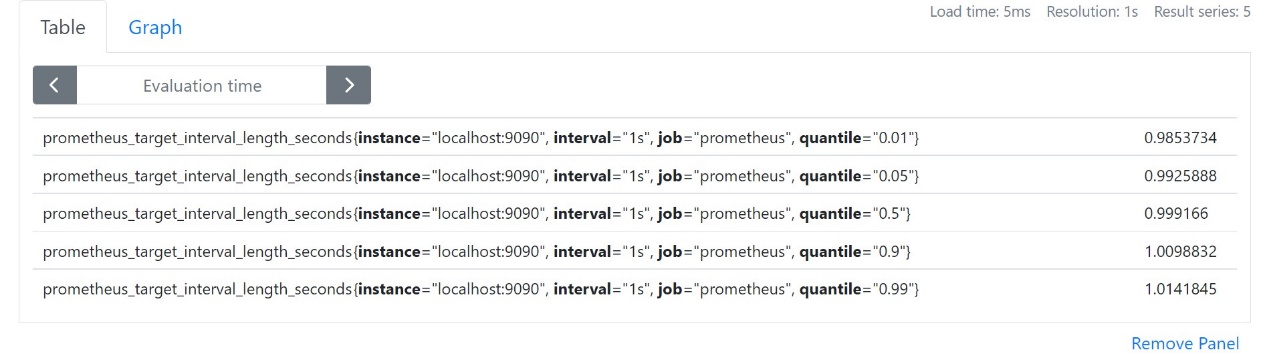
這次小實作採用的是windows系統的server，因為還在期中，所以我們還沒對prometheus的深度用法進行了解，像是docker、API等等的用法。

Windows的server架設非常簡單，直接到官方網站下載server的exe檔案就可以了，打開後他會執行server，預設的位址會是<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這個檔案中更改scrape\_interval: 15s和- targets: ["localhost:9090"]。

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。

這裡以prometheus\_target\_interval\_length\_seconds 為例子，輸入完呈現這樣



按下執行後，下方就會呈現出本地端電腦的資源

附上網址的差異(前&後)



心得

老實說prometheus的server真的非常簡單，能查詢的東西也非常多，在promql裡就有很多像是cpu\_time、http\_request\_total等等，還可以運用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GUI介面，可以讓使用者看出資料的分布情形。

評論

Pros

* base-PULL，無須安裝代理。
* 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。
* 針對單節點儲存使其操作簡單。
* 可視化和監控警報對使用者很友善。
* 可以詳細列出電腦資源的遠端監控系統，擁有部署簡單、去中心化的特點，配合外部的GUI和警示系統，提供偵錯及視覺化警告。

Cons

* 內建的query採用PromQL語言，其需綁定Grafana，需要額外進行Grafana [[[8]](#footnote-8)]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 [[[9]](#footnote-9)]，大規模使用需要第三方component(e.g. Thanos, Cortex, etc.)才能實現 [[[10]](#footnote-10)]，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

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)
8. [] <https://grafana.com/> [↑](#footnote-ref-8)
9. [] <https://prometheus.io/docs/prometheus/latest/storage/> [↑](#footnote-ref-9)
10. [] <https://www.youtube.com/watch?v=3pTG_N8yGSU> [↑](#footnote-ref-10)