Concurrency 範例

by

Triton Ho 

範例

- 我家被人 DDOS 了，我想找出 DDOS 來源的 IP
  - 100 個 apache log access file ，每個大約 5GB
  - 我想找出每小時的 top 10 IP 來源

警告

- 本文單純作範例來教 Concurrency 而已， DDOS 不是這樣子擋的
- 真的要擋 DDOS 請找專業的業者，像很有種的 cloudflare
- 以 IP 來擋，很有可能錯殺用了 NAT 的 ISP 的正常用戶
- 要分析像 apache 這種熱門的 log ，市面上有大堆軟體像 goaccess 之類的

方案

- 先寫一個小的 script ，把 access log 掃瞄一次
- 然後把每小時的 access log 丟進一個獨立的 file
- 每小時的 access log 都跑一個 goaccess 的 process 來分析
- 這樣子，你多少個 CPU 都能吃光光， 100% 真正的「平衡運算」

完

（謎之聲：……）

謎之聲：別忘記 50k NT 的石虎捐款

- 我們先忘掉這世界有一堆現有的軟體能用，要自己寫 ~
- 不准沒腦的開 1000 個 process ，要寫得優美
- 最終版本：不準讓系統資源閒置浪費掉
  - 即是說：要卡在系統的 io bandwidth 或 CPU 總效能

v1 版本

For each logFile in fileList

- Open file
  while file.GetLine() {
    line.Parsing() 去拿回 IP 和 accessTime
    把 accessTime roundtime 到最近的小時
    redis.HINCRBY(accessTime, IP, 1)
  }
  Close file

工作說明

- 程式是把 Per IP 訪問次數丟到每小時為單位的 HashSet
  - 警告：你的 Redis 會用超級大量的 Memory 的
- 全部的 access log 都跑完後，你對每一個小時 HashSet 做 HGetAll 拿回全部的 IP ，再跑一次 Sorting ，你自然拿到每小時最高的 IP

v1 解說

- logAnalyzerV1 是 Single thread 沒錯，但是.....
  - 但是你能用 tmux ，在 n CPU 的機器上跑 >=2n 個 LogAnalyzer（謎之聲：..... ）
- 例子，在 2 core machine 上
  - Process 1 的 fileList: 1.log, 5.log, 9.log……
  - Process 2 的 fileList: 2.log, 6.log, 10.log……
  - Process 3 的 fileList: 3.log, 7.log, 11.log……
  - Process 4 的 fileList: 4.log, 8.log, 12.log……

v1 解說（續）

- Redis 的操作是 blocking 的
  - 即是說： HINCRBY 在 redis 跑完前，你的 Application 的 thread 必需等待（ blocked ）
  - 等待中的 thread 是不吃 CPU 的
- 在等待 Redis 操作時， logAnalyzerV1 是不吃 CPU
- 剛才 while loop 中，最長時間的應該是 Redis 操作
  - Network 是有 latency 的.....
- 如果在 n CPU 上只跑 n 個 logAnalyzerV1 ，肯定吃不光你的 CPU

v1 缺點

- 現在是每一筆 access log 都要做 redis 操作
  - 你大部份時間都浪費在 network latency 上
  - 每一次 redis 操作都會觸發 context switching
- 結論：你的 v1 會非常非常慢
  - 跟之後 v2 來比，可以差了超過一倍速度的.....

v2 版本： RedisBulkAction

- 針對 v1 的 redis 問題，我們現在建立一個叫 RedisBulkAction 的 class

```
Class RedisBulkAction {
    private IpBuffer[2000] buffer;
    public addLog(ip, accessTime) {
        add (ip, accessTime) to buffer
        if buffer is full {
        bulk redis.HINCRBY
        }
    }
}
```

- 註： bulk redis.HINCRBY 這一部份
  你很可能會用上 redis pipelining 或是 lua scripting

v2 版本： main

re = RedisBulkAction.New()

For each logFile in fileList

- Open file
  while file.GetLine() {
    line.Parsing() 去拿回 IP 和 accessTime
    把 accessTime roundtime 到最近的小時
    re.addLog(IP, accessTime)
  }
  Close file

re.flush() // 別忘記離開前把剩餘作 flushing

v2 解說

- 現在每 2000 筆資料才會做一次 redis bulk action
  - 所以每 2000 次才會有一次 network latency
  - 所以每 2000 次才會有一次 context switching
- v1 你需要 >2n 個 process ， v2 你可能要 ~1.2n 就夠了
- 如果你是很偶然才做一次這樣子的分析，停在 v2 就夠了
  - 謎之聲：人家是捐了 5 萬耶.....

v2 小缺點

- 每一個 process 拿獨立的 fileList ，如果你的 accessLog file 是有特定 pattern 的
  - 像 1.log, 5.log, 9.log 的數據量特別少
- 那會有一些 process 很快就跑完，然後你要等待最慢的 process
- 現在有 1.2n 個 thread ，有沒有辦法壓得更低？
  - 在用光 n CPU 前提下，越少的 thread ，代表越少的

V3 前言

- 先說一下 v2 的 1.2n 是怎來的
- 每 2000 筆資料
  - Redis 操作用了 2ms，不吃 CPU
  - 其他的（ FileIO, Line Parsing ） 10ms，吃 CPU
- 所以，每一個 thread 只有 5 / 6 時間吃 CPU
- 反向來算，如果要用光 n CPU
  那就需要 n / (5 / 6) = 1.2n 這麼多的 thread
- 所以，理想上：做 unbuffered blocking IO 操作的

V3 前言（續）

- Redis 操作所用時間，很受 network latency 影響的
- 你涉及 Redis 操作 (blocking IO) 的 thread ，其 ThreadNum 就跟 network latency 相關 =.=
- 所以，更理想做法：把 blocking IO 的工作以獨立的 thread 群來工作

v3 架構

- Single process, multi threading
  - 謎之聲：你終於到正題了
- 有二種 thread
  - Importer
    - 從 file 中一行一行拿出來，然後做 Parsing
  - Exporter ：
    - 把 (IP + AccessTime) 以 BulkAction 手段寫到 Redis
- Importer 和 Exporter 之間資料流動以 channel 串起來
- 在 n CPU 環境，大約是 n 個 Importer + 2 到 3 個 Exporter

淺談 stream

- 未談 channel 前，先來談一下 Linux 的 stream 吧
- 所謂的 stream ，就是 FIFO 地讓資料從一個 process 流到另一個 process
- process 的 stdin ，就是一個 stream
  - data ：用戶的 keyboard input
    sender ： terminal ，像 bash 這個 process
    receiver ： process 自己
- blocking 會出現在：
  - stream 的 buffer 滿了， sender 要丟資料進 stream
  - stream 是空的， receiver 想要從 stream 拿資料

channel

- 跟 stream 相以，但是可以有多於一個 sender 和 receiver
  - 註 1 ：有些 implementation 不保證 FIFO 的
  - 註 2 ：有些 implementation 不保證沒有 starvation
- 也是會用上 mutex 來防止 race condition
- 看起來用 ring buffer 再配 mutex 就能自己寫出來
  - 能用 library 就千萬絕對別自己寫
  - 涉及 concurrent 的底層架構有太多出錯的可能性了
  - 提外話：用 Redis SETNX 作 locking library 的，沒問題的少於 10%
    ……

智障架構

- 現在有 1000 個 log file
- 智障主意：每一個 file 都開一個 Importer thread
  - 這樣子 file descriptor 會不夠用！
  - local file 還不一定有事，像 Amazon S3 這種 non-local file ，死定了
  - 1000 個 thread 的 context switching ，讓你效能大降
- 江湖傳說：某「架構師」，在 4 core 機器上開了 5000 threads
  - 如有雷同，實屬不幸

智障架構（續）

- 你的 worker threads 數量，應該只跟 CPU core 數量有關
- 跟你的 tasks 的數量不應有關係
- 喵的！別告訴我 green thread 就沒 context switching
  - 你知道： green thread 會吃到 process 的 global memory 嗎？
  - process 的 global memory ，是遠遠遠遠小於你 physical memory 的
    - 具體有多少看你底層 green thread runtime 和 OS 設定
- 無限制地建立 green thread ，最終就是 Out of ~~Mana~~ Memory 了

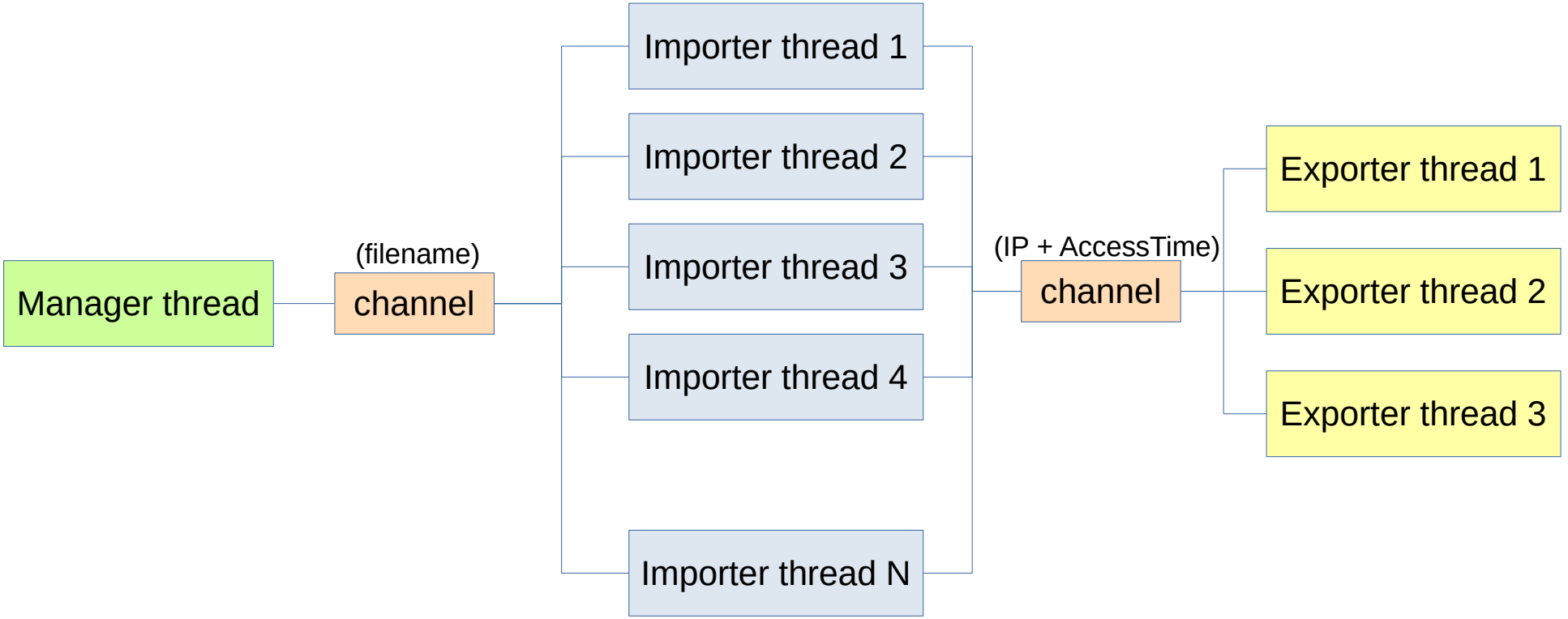
Manager Worker model

- 你的 worker threads，在 n CPU 上就應該是 n 個，不管你有多少個工作
- 你的 worker threads 需要 channel 來接收工作
  做完手上的工作才從 channel 拿新的一個，直到 channel 關掉

```
While inChannel is not closed {
    Get task from channel
    Process task
    Put result to outChannel
}
return;
```

- 所以，你需要一個 manager thread 來把工作丟進 channel
  丟完後別忘記關掉 channel

v3 架構圖


Manager thread
(filename)
channel
Importer thread 1
Importer thread 2
Importer thread 3
Importer thread 4
Importer thread N
(IP + AccessTime)
channel
Exporter thread 1
Exporter thread 2
Exporter thread 3

V3 Manager Thread

```
For each filename in fileList {
    put filename into fileNameChannel
}

close fileNameChannel
```

V3 Importer Thread

- While fileNameChannel is not closed {
  get fileName from fileNameChannel
  Open file
  while file.GetLine() {

    line.Parsing() 去拿回 IP 和 accessTime
    把 accessTime roundtime 到最近的小時
    put (IP, accessTime) into accessLogChannel

  }
  Close file

}
atomic.Decrease(importerCounter – 1)

V3 Exporter Thread

- var buffer
  While accessLogChannel is not closed {
      get (IP, accessTime) from fileNameChannel
      Put (IP, accessTime) into buffer
      if length(buffer) >= 2000 {
          flush buffer into redis
      }
  }
  flush remaining buffer into redis
  atomic.Decrease(exporterCounter – 1)

Thread counter 解說

- 問題： main thread 怎知道所有 worker thread 都完成了工作，可以讓 process 結束？
- 答案：很簡單，每個 worker group 都用一個 int counter 來記錄在生的 thread， thread 要結束前把 counter 減一就好
- 因為可能多個 thread 同時結束，所以 counter 要用上支持 atomic operation 的 library 來操作
- Main thread 對這些 counter ，進行 blocking wait ，直到這 counter 歸零
  - 很可能你要一些 atomic library
- 細心讀者會問：為何 manager thread 不用這種 counter ？
  - Importer thread 要結束，其 input channel 要先關掉吧
  - 其 input channel 關掉，代表其 manager thread 已經結束了
  - 所以， main thread 只需要等 importer thread 結束就可以

V3 main thread

```
Var filenameChannel, accessLogChannel
start manager thread
var importerCounter = n // n = CPUNumber
start importer thread * n
var exporterCounter = 3
start exporter thread * 3
atomic.WaitUntilZero(&importerCounter)
close accessLogChannel
atomic.WaitUntilZero(&exporterCounter)
return
```

Main thread 解說

- 一個 importer thread 結束時，他沒法知道是否還有別的 importer 還在工作的
- 所以， accessLogChannel 不能在 importer 內關掉
- 所以， main thread 要等所有 importer thread 都結束，才可以關掉 accessLogChannel
- 另外一句：
  - 在比較大的軟體， main thread 只負責把 modules 建立起來，他不應該做事的

明明 file io 都是 blocking IO

- 謎之聲：明明 file io 是 blocking IO ，為何你沒把 read file 動作和 Parsing 分開二個 stage ，由不同的 thread 來負責？
- 我實實在在地告訴你： Linux 的 file IO 是 buffered
  - 不是你呼叫 ReadLine() 時， OS 才會替你把 data 從 HDD 拿出來
  - Linux 在你打開 file 時，就背後替你建立了 buffer ，然後自動地預先把 data 從 HDD 放到 buffer 中
  - 除非你做 Seek() 操作，或你每行超過了 buffer 上限，否則你的 ReadLine 大都是 memory-only 操作，時間接近零

延伸一句：別有過多的 stage

- 智障：
  - 只有 Importer 和 Exporter 這二種 worker thread 不夠「潮流」
  - 我們要把 Line Parsing 拆成獨立的 stage ，由不同的 worker thread 來工作
- 喵的！ single responsibility principle 不是這麼用的

別有過多的 stage

- 在 Importer 完成 Parsing ，這代表資料留在同一 thread 的 local variable 上
- channel 底層是有 mutex 保護的，這代表：你使用 channel ，就有一點點效能損失在 channel 上
- 另外一個重點：使用 channel ，就代表資料要從 thread A 先抄到 channel 的 memory ，然後再從 channel 的 memory 抄到 Thread B 的 local variable
- 你的工作有越多的 stage ，就要越多的 channel 把 stages 連接起來，就越多

結論

- 這篇主要談了 Application layer 的 concurrency
  - 如果是底層 library（像 caching ），是另一個思維
  - 底層 library 經常使用大量的 mutex 的
- 你應該重點思考：怎建剛剛好的 thread ，來用光 n CPU
- channel 機制是 concurrency 的核心
  - Akka, Erlang 的 Actor Model 也是相以的

最後一句：

祝願台灣石虎

能見證人類離開太陽系那一天

完