高流量心得雜談

by

Triton Ho 

前言 1

- 羅馬不是一天建成的
  - facebook 也不是第一天就支持 1M 同時在線用戶的
- 「效能不足」是快樂的問題
  - 這至少代表系統有用戶，有人在付你的薪水
  - 再三強調：別 over-engineering
- 在高流量系統.....
  - ８０％時間是清理各式各樣基本錯誤
  - １０％時間是真的動手寫高流量用 library
  - １０％時間是說服主管別幻想太多

小秘密

- 在 10K Request / Sec 之前，你的架構一般網站不會有太大差別
- 木桶理論
  - 木桶能載多少水，是最矮那片木板來決定
  - 你的系統能支持多少流量，是最弱的環節來決定
- 請花精力去肯定每一環節都做好所有基本功夫，而不是去談什麼假大空 distributed system / big data

大綱

- Auto-scaling
- Rate-limiting
- Thread pool
- Metrics 和 Logging
- 節省 IO
- Hotspot
- Cache TTL
- 災難控制
- Blue-Green Deployment

Auto-scaling

- 先說結論： Auto-scaling 是用來省錢的
- 一般用在 App. Server 層，別用在 DB 層
- 流量上升是慢慢的，不是爆發性的
  - 手遊尖峰在下班時間，但是
    - 不是所有人在同一秒下班
    - 不是所有人下班後立即上線

無法 auto-scaling 的問題

- 電商網站開賣受歡迎商品
  - 數秒內百萬用戶流入
- “Happy new year” 問題
  - 用戶在同秒發訊息
- 解決之道
  - 跟市場推廣同事合作，預先加開機器
  - 瘋搶商品改用事先登記和抽籤？

Rate-limiting

- 超過系統上限的流量就必需等待 / 拒絕
- 爛人名言：
  - 這會讓公司流失生意的，這樣子不行！
  - ~~這是你們工程師的責任啊！~~
- 我的回答：
  - 請跟那些百年牛肉麵老店老闆說，把門外排隊的客人全放進去，看看老闆砍不砍你 ~
  - 讓一千人全進一家牛肉麵小店，最終只會所有人都沒東西吃

Rate-limiting（續 1 ）

- 每一層都需要 rate-limiting
  - Load balancer
  - App. Server
  - Caching
  - Database
- 越早擋住過多的流量，就越能保護其後方系統

等待 vs 拒絕

- 電視劇經典
  - 你要等我啊，我回來故鄉後就結婚吧
  - 然後對方等不了，早便跟別人結婚生小孩了
- 同理，每一層的 Ratelimiting 都不應該有太長的等候時間
  - 因為用戶會等不了乾脆 <F5>
  - 超過等待上限的，請直接回 HTTP 503
- 另外，等待中的工作，會持續佔用系統資源
  - TCP/IP connection
  - Main Memory
  - CPU Thread

Thread pool

- ~~backend 「大神」：我在單台機器上開子 5000 個 thread 耶~~
- 爛主意：為每一個工作建立一個 thread 來跑
- 好主意：預先根據 CPU 數量和工作內容建立 worker thread pool
  - 有新工作來臨時，就把工作丟到沒在忙的 worker
  - PHP-FRM 架構 ~
  - 名字： manager-worker pattern

Thread pool（續）

- 把 thread 開開關關有很大 OS level overhead
- 一千人全跑進一家小店，只會大家都餓肚子
- 等待被派給 worker 的工作，跟等待 CPU 來執行的 thread ，其 memory 用量差很大
- 延伸思考： Database connection pool
  - Proxysql / pgpool

Metrics

- 你應該紀錄以下 Metrics
  - 系統流量
  - CPU, IO 和各種系統資源用量
  - Exception / Error 數量
  - 各式各樣的 latency
  - DB Query 的執行時間
- 當你的老闆跟你說「系統很慢」時.....
  - 你沒有數據 = 你的錯
  - CPU / IO 已經用滿了 = 老闆不肯給錢的錯

Logging

- 俗語：有人之處就有 bug……
  - ~~台灣石虎一天未肯替大家寫程式前，請接受軟體有 bug 這事實~~
- 如果你覺得可以寫出 100% 肯定沒 bug 的系統
  - 你可以上來當講師了
- 沒錢就單純 logrotate 時存到 AWS S3
- 有錢就建 ELK
- DB 的 status dump 也是關鍵！

System Monitoring

- 監測系統是否正常（像是太多 HTTP 5XX ）
  - 應該放在 Metrics ，不應放在 log
  - Metrics 和 log 的數據量絕對有差， delay 也有差
- Metrics 是用來告知系統是否出事
- log 是讓人們重組案情的

System Tuning

- 名句： premature optimization is the root of all evil
- 如果你沒有足夠的 Metrics ，你就沒法知道系統慢在那裡
- 案例分享：
  - 某公司發現，好像使用 Redis 越多的 Endpoint ，就好像越慢
  - 加了 Metrics 還有一堆實驗後發現：真正慢的不是 Redis ，而是把資料丟進 Redis 前的 json encoding / decoding library

節省 IO

- App Server 可以輕易加開機器， DB / cache 層不是
- DB / cache 主要卡的，是 IO
- Compression 能有效減少 network IO 和 storage IO
- 如果你只要 Page 1 第 1-10 筆資料，別智障去拿整個 Result Set

Hotspot

- 如果你需要 Consistency ，不管你是 RDBMS 還是 noSQL ，一份資料只會存放在一個 master node 內
- 如果那一份資料有非常大量的 Update ，存了那一份資料的 master node 就會有大量流量
- 重點： hotspot 是卡在 single thread 效能，你沒法以加開機器來解決的

Local caching

- 如果所有的 READ 都是從 Redis 拿資料
  - 如果有一份資料是大量使用者需要的
  - 這會變成 READ 版本的 hotspot
- Write 的 hotspot
  - 大部份都要放棄 Consistency
- Read 的 hotspot ，有機會能以 MVCC 來解
  - 反向思考：什麼是「即時」？
  - Snapshot 是不會被改動的，所以能放在 local cache

Cache TTL

- 笨蛋：反正 Redis 滿了會自動把舊的 cache 丟掉，不設定 TTL 也沒差囉 ~
  - 沒設 TTL ， Redis Cache Eviction 是在 Peak hours 的，呵呵
  - 而且是在有新資料寫入時強迫先 Eviction ，呵呵呵
  - 然後 Redis 是 Single Thread ，單一工作緩慢是會卡到之後全部的工作
- LRU 不是萬能的.....

災難控制

- 再三地說：有人之處就有 bug
- 單一巨型 binary 最大問題
  - 一個 while(1) loop bug 就系統全滅
- 別用 main process 來接第三方系統
  - 你真的信什麼 SLA 99.99% 嗎
- 延伸思考
  - 怎保護 Redis / Database ，減少 Application bug 的災難度？

延伸思考： Micro-service

- 好主意：根據商業邏輯做垂直切割
  - service 獨立生存
  - service 能獨立提供用戶某一面向的功能
- 壞主意：根據內部架構做水平切割
  - 用戶 → ServA → ServB → Database
  - ServB 一旦死掉，依賴 ServB 的 service 會全滅
- 一份工作要用上的 component 越多，它的 Availability 就會越低
- 人家 FNAAG 的 Micro-service 架構，是系統太太複雜下的無奈之作
  - 員工人數眾多也是主因

Blue-Green Deployment

- 永不在 server 上跑 do-release-upgrade
  - upgrade 會帶來未知風險
- 如果你要從 18.04LTS 升級到 20.04LTS ，請直接開新的 20.04 機器
- 你應該使用 ansible 來替你的 server 安裝東西
  - 人類錯誤永遠是 production 最大風險
  - 可重複的行動，能讓你輕易建立 staging 和各式各樣測試環境
- 升級後別立即把舊機器關掉

完