backend 課程（第三課）

by

Triton Ho 

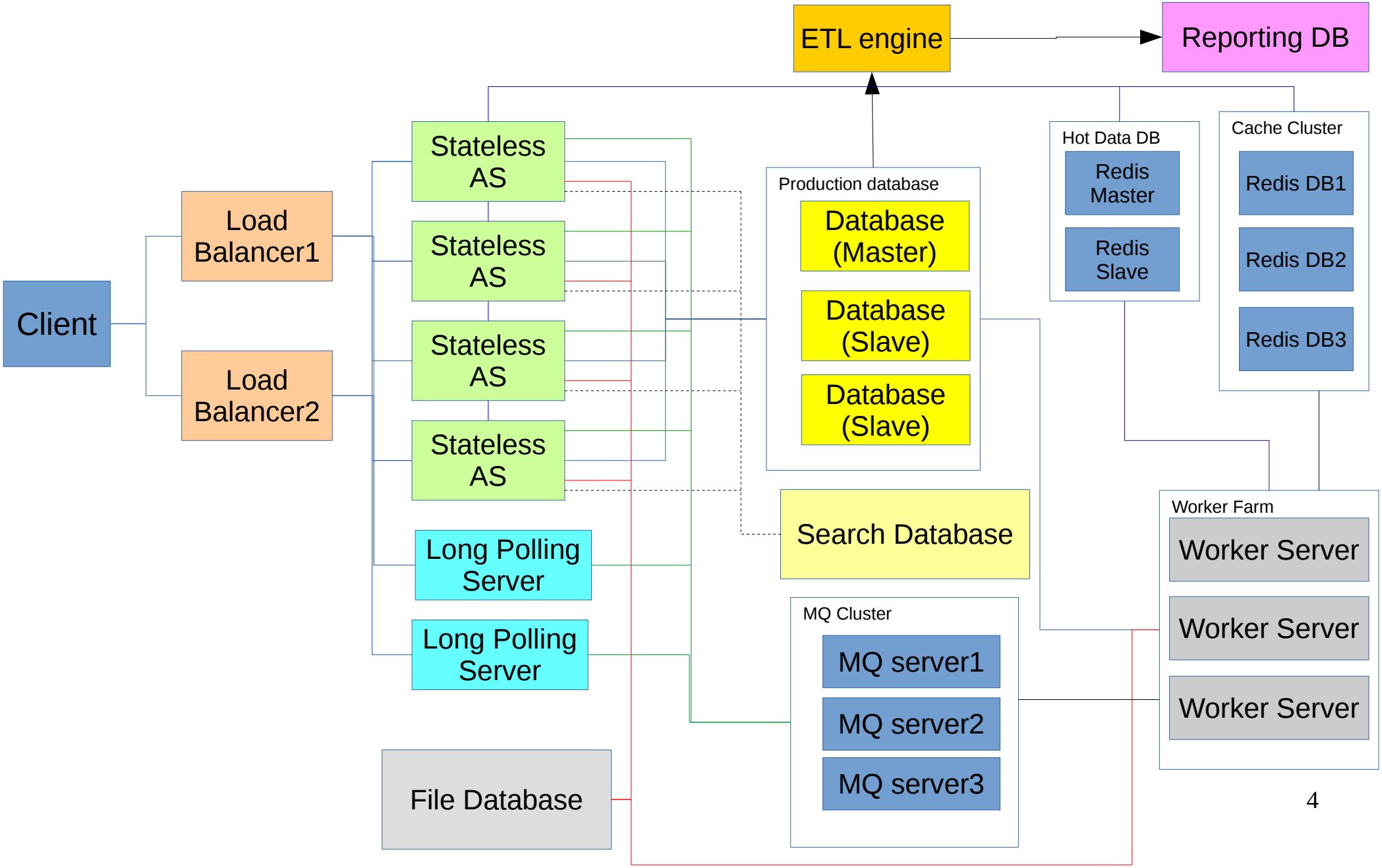
今天大綱

- 理想的 backend 系統
- Backend 架構
- 課程總結

前言

- 「工程」 = 在限的資源（錢，技術，開發時間）下，使用有效的手段，找出最理想的平衡點
- 接下來的架構，你不一定全部合用的。別單純因為別人這麼做，你便跟著做
- facebook 的架構，跟路邊八卦一樣，聽聽便好
  - facebook 一秒的流量，一般系統一年加起來也不一定有

最終系統架構圖 ~


ETL engine
Reporting DB
Client
Load Balancer1
Load Balancer2
Stateless AS
Stateless AS
Stateless AS
Stateless AS
Production database
Database (Master)
Database (Slave)
Database (Slave)
Hot Data DB
Redis Master
Redis Slave
Cache Cluster
Redis DB1
Redis DB2
Redis DB3
Search Database
Long Polling Server
Long Polling Server
MQ Cluster
MQ server1
MQ server2
MQ server3
Worker Farm
Worker Server
Worker Server
Worker Server
File Database

WTF!? 我只是想寫一個小網站而已

所以才叫你們不要看見別人的架構很漂亮
便立即閉上眼睛跟著做啊

理想的 backend 系統

理想的 backend 系統

- 越少模組越好
- 沒有 Single Point of Failure
- 對 Surge 有抵抗性
- 不需要即時人員支援

越少模組

- 越少模組，便代表
  - 你需要監控和管理的模組便越少
  - 代表你的團隊需要掌握的語言 / 技術便越少

通用 vs 專精？

- 例子：是否使用 ElasticSearch ？
  - 你需要 Tokenization 嗎？
  - 能不能先在 RDBMS 做 searching ？

安全 vs 快速開發？

- 例子：應該把呼叫 GCM 放到獨立 worker 嗎
  - 多了一個 worker ，便多了一個 worker 要管
  - 如果使用 worker ，便需要建立 MQ server
  - 可是有了 worker ， GCM service 發生什麼事也影響不了我的主系統耶

沒有 Single Point of Failure

- 先說一下：如果 backend 某一部份當掉，很很很難完全不讓 client 知道的
  - 即使 Synchronous replication 也好；還未 Committed 的 transaction 狀態在 RDBMS Master 當掉後便流失了
  - 如果 LoadBalancer 當掉，在 IP 重新指派 / 該 LoadBalancer 的用戶都會沒法連線
- 已死 / 發瘋的模組，不可能主動說自己死掉 / 瘋掉的
  - 由監察系統停用 / 換掉該模組，不可避免會有時間差的
  - 最重要一點：別引起全面 + 長時間（ > 數分鐘）的停止服務
    - ~~只要能把問題推給用戶連線品質，便不是問題~~

沒有 Single Point of Failure（續）

- 每一系統模組 (System Component)，都應該有 standby instance
- 一旦監察系統發現某一模組當掉，便把其停用，然後讓 standby instance 接替
- 一旦某模組當掉，你的設計應該讓系統停在 Consistent 狀態
  - 除非你為了效能，而故意犧牲 Consistency
- 不過， standby instance 只能保護單一隨機性的災難
  - 例如老鼠引起電源短路，讓單一伺服器當機

軟體災難

- 由軟體錯誤 (bug) 引起的災難，很多時候會擴散，最終引起整個系統崩潰
  - 假設 json library 有 bug ，在接收超過 20KB 的 json 後便引發 stackoverflow 當掉
  - 用戶 A 現在把 21KB 的 json request 傳到 Application Server 1 AS1 當掉， load balancer 停用 AS1
  - 用戶等不到 response（ AS 當掉了嘛）重新再傳送 Request Requert 被分配到 AS2 ，讓 AS2 當掉， load balancer 停用 AS2
  - 用戶 retry 十次，便有 11 台 AS 當掉。你猜用戶 retry 比較快，還是你加開 AS 的速度比較快

軟體災難（續）

- 可幸的是：常用的主流 library 有致命性 bug 可能性不高
  - 呃，寫這頁的時候便剛剛出現 SSLv2 security loophole……算了，至少有大家一起陪伴爆肝
- 一般來說嘛， business logic tier 是系統熱門崩潰地方 ~
  - 例子：你沒使用 ORM ，然後 DB Session 你沒有 commit / rollback
  - 想系統不當 + 安全，請寫好一點的程式碼

對 Surge 有抵抗性

- 系統流量爆發，常常是不可預測的
  - 911 時，大家都電話報平安
- 動態加開 AS 會有幫助，但是加開的速度不一定能追上流量成長
- READ 能用 Cache 支撐， WRITE 不行
- 吃大量資源的 endpoint ，可以考慮轉用 async 模式
- 別在 AS 上執行 async API 的工作，會害 AS 當掉的

不需要即時人員支援

- 使用 Load Balancer 下，一台 AS 當掉便自動被停用，只讓系統損失 1/n 的運算能力
- 監察系統發現 Master RDBMS 當掉後，應該懂得自動把 Slave DB 提升成 Master 並進行切換
- 交給 Job Farm 的工作，在本來的 Worker Instance 當掉後，會自動由其他 Worker 重做

Backend 架構

Backend 重要模組

- DNS
- Load Balancer
- Application Server
- Long Polling Server
- Main DB
- Cache Cluster
- Hot Data DB
- Search DB
- Report DB
- File DB
- Message Queue
- Worker Farm
- Cron job Works

DNS

- ~~某巨型防火牆最喜歡動手腳的東西~~
- 用戶不會記得 122.198.6.1 這種鬼東西的，他們只會記得 www.yourcompany.com
- 對大型網站，會有多個 data center 的， DNS 會回答該地區最近的 data center 的 IP
- 對大型網站，一個 domain name 可能會對應多個 IP
  - 單一 load balancer 有其物理極限的（頻寬 / CPU ）

Load Balancer(LB)

- IP 是稀有資源，每台 AS 都有自己的 public IP 不太現實
- 一般 LB 都有 Health Check 功能，一旦某台 AS 掛掉便立即停用
  - 所以 AS 常常會有 /healthcheck 的 endpoint
- 通常， LB 會把 HTTPS 轉成 HTTP ，然後才把 Request 轉給 AS
- 高階的 LB 能根據 url（甚至是 Request body ）作 routing

Load Balancer 模式

- 如果你的 AS 架構沒大問題， round robin 跟 least conn 沒什麼差別
- 如果你的 AS 不是 stateless 的.....
  - source 只能用在非流動用戶上（手機 IP 會改）
  - sticky 需要 LB 解讀 Request 並抽取像 SessionId 的變數，然後再根據之前歷史派到特定 AS 上。對 LB 有極高性能要求

Application Server(AS)

- 好的 AS ，應該是 Stateless 的
  - 這樣才能閉上眼睛，視流量來加開 / 關掉 AS
  - AS 當掉了也完全不會痛 ~
  - LB 便不用昂貴的 sticky mode 了
- AS 中，不應該跑會吃大量資源的工作
  - LB 才能使用 round robin / random 模式
  - AS 才不會「凍結」掉

Long Polling Server(LPS)

- 如果你是要把 Message 推送到手機上，請使用 GCM / APNS ，別自行來架 Server
- 除非你的系統流量很大，用 AS 處理 Long Polling 便好
- 使用獨立 LPS 的好處
  - 你能改用專精負責 Long Polling 的語言
  - 你不用擔心 AS 的 execution worker pool 被用光光

Main DB

- 別輕易使用 noSQL
  - 詳情請看 < RDBMS 課程（先修課） >
- RAID 只能保護單一 SSD 故障
  - 電源短路 / ~~打翻泡麵~~時，所有 SSD 都會一起陣亡
- Master/Slave Replication 保護不了壞人刪除數據
- 只有以離線儲存的備份數據，才是真正安全的

Main DB（續）

- 大部份時間，是系統瓶頸所在
  - 別以為換成 noSQL 便能安心.....
- CAP 理論的同義：
  - < 效能，數據安全性，低成本 >
    三者最多你只能要兩種

Cache Cluster

- 對小型系統，單獨一台大記憶體的機器作 caching ，也許比較方便
- 對大型系統，單獨一台機器作 Caching 不合成本效益，所以要用多台機器
- 要知道物件放在那台機器上，最簡單的方法：
  - MD5(key) mod n
- 為 cache cluster 增加 / 刪減機器時，小心別引發 total cache miss

Hot Data DB

- 跟 cache 是不同的
- 暫時性的，丟失了也死不了的數據
  - 像 LOL 的對戰狀態
  - Deduplication 用數據
- 在為了效能犧牲 Consistency 的情況下，延後寫入 main DB 的數據
- 一般來說： Hot Data 數據量不會太多，所以不太需要 Clustering
- 但是，建議作 Replication 去保護資料

Search DB

- 看老闆的錢包深度，沒錢別額外架 Search DB
- 數據儲存結構有特別為搜尋作特化
- 有 tokenization 和 auto correction 等等幫助搜尋的重要功能
- 很多時候： Search DB 的數據跟 Main DB 的會有時差

Report DB

- 只要看到帳單，很多老闆不介意在 Main DB(Slave) 上面等一下的
  - 錢包不是非常非常深的，別輕易說 Data warehousing
- 專門的 Report DB ，很可能採用 denormalized schema
  - 需要高度專業性
  - 要很多很多的錢
- Random Sampling 是否能解決問題？
- 別輕視傳統而且相對低成本的方法
  - 例如：直接上門拜訪客戶，問一下他們需要什麼

File DB

- 有人說： File 應該以 BLOB 物件放到 Main DB ，但是
  - File 一般建立後極少被改動
  - RDBMS 一般用上系統中最高級的 SSD ，而 File 一般不需要這種效能
- 延伸思考：在檔案名字中額外加上 MD5 ，許多能解決很多 File caching 問題

Message Queue(MQ) 前言

- 當你懂得使用 MQ 後，你的視野會變得開闊
  - 至少不會再跟別人吵什麼語言最高效能
    - 直接寫 x86 Machine Code 一定最快！
- 別以為大型系統才需要用上 MQ ，他的應用遠比你想像中大
- 有些 MQ 不保證絕對性的 FIFO 和 no-duplicate ，使用上請注意（例子： Amazon SQS ）
- 如果你的應用比較簡單， Redis 某程度上也能當作 MQ 使用

Message Queue 名詞

- Message
  - 一般來說，會裝有以 Json / XML 格式的工作內容
  - 會有 MessageId
- Producer
  - 訊息生產者，一般來說是 AS 要建立工作
  - 一個 Queue 能有多於一個 Producer
- Consumer
  - 訊息消耗者，一般來說是 Worker 要執行工作
  - Queue 中沒有 Message 時， Consumer 會被 blocking ，不佔用 CPU
  - 一個 Queue 能有多於一個 Consumer
  - 一份訊息，在同一時間內只會讓一個 Consumer 收到

Async tasks 工作流程

1.現在 Queue 上沒有任何 Message ，所以眾多聆聽著這個 Queue 的 Worker 都被 blocked ，並且不佔用 CPU

2.AS 接收到 async API request(例子：老闆要看大型報表)

3.AS 以報表參數和老闆的 email 建立 Message ，丟到 MQ 後便能安心不管，並且對用戶回答 HTTP StatusAccepted

4.其中一個 Worker 從 Queue 上接收到 Message ，並且執行工作

5.該 Worker 工作完成後，把 Message 從 Queue 上刪除

以 Redis 當作 MQ 使用

- 把 Redis 順便用來作（簡單版） MQ 使用，便不用架設 rabbitMQ 了
- List 能作為簡單版的 Queue
  - 建立 Message 時，使用 LINSERT
  - Worker 使用 BRPOPLPUSH 來等待和接收 Message
  - Worker 完成工作後，使用 LREM 來刪除工作

Worker Farm

- 同樣的 Worker ，請在多於一台 Worker Server 上建立 running instance
  - 你不想 AM0300 只因為一台 Worker Server 當掉，你便要爬起來撲火吧
  - 一個 Message 同一時間只會一個 Worker 收到，不用擔心多個 Worker 重覆工作問題
  - 你可以視 Queue 中的剩餘 Message 量，動態決定增減 Worker Server

Cron job workers

- 跟 async task worker 有點不同，他們只在預定時間睡醒，把工作做完後便死亡
- 有時候 cron job 是不能迴避的
- 為了不用深夜撲火，你還是應該在多於一台主機上執行
  - 為了一份工作不被多個 Worker 執行，你有可能需要 execution control 機制
  - Database-as-IPC 是很嚴重的 anti-pattern ，但是如果有很少量的 data exchange ，還算可以接受

理想的 Cron job

- 即使當掉，也無需 data cleanup ，單純重跑便好
- 同時間多個 Worker 執行時，也懂得自動分工，不會出錯

課程總結

課程總結

- 到了現在，你應該懂得 AS 內的主要模組，有勇氣對別人寫好的 framework 進行 hacking
- 你應該對大型 backend 系統有初步性的認識，懂得正確使用 MQ
- 你應該再花 100 小時去消化課堂所教的東西
  請對所有人的說話抱持合理的懷疑

廢話時間

以買蛋糕作比喻

- 你去離家五分鐘路程的地方買 100NT 的蛋糕，對方打算以紙盒包裝時.....
  - 為了預防回家時預上 10 級大地震震壞你的蛋糕，所以你要求以六軸穩定器來包裝，即使遇上 10000G 的加速度也震不壞 ~
  - 你擔心外星人攻打台北時，其 UFO 的主砲會你的蛋糕順便炸掉，所以你要求包裝至少能防禦原子彈直擊的 ~
- 店員：別鬧了

請不要當奧客（無誤）

- 你不會以 100 億 NT 的包裝去裝只值 100NT 的蛋糕同理，你不應該追求一個「完美 framework 」，去 over-engineering
- 設計架構時，請優先從需求優先度來思考
  - 你的蛋糕能在 9 級地震沒事也好，那你呢？
- 請記住：最大的風險不是系統全面崩潰，而是沒有客戶使用

身為架構師的雜感

- 具同理心的溝通，比什麼也重要
  - 維持系統效能 / 程式碼整潔度固然重要，但團隊士氣更加重要
  - 人才是公司最重要的資產，而不是系統架構
  - 絕對別建立遵守不了的規定
- 架構師是前線衝鋒的戰車，用來突破所有技術關卡
  - 空談一堆名詞而不身先士卒的傢伙，很易死於 ~~friendly fire~~ 流彈的

工作優先度

- 沒人使用的系統，才是沒有問題的
- 工作有 2 個象限
  - 會惡化 / 不會惡化
  - 昂貴 / 不昂貴
    （指付出時間 / 金錢 / 嘗試解決時的潛在風險）
- （會惡化 + 不昂貴）一定最優先做
- 當你得到老闆的信任下（這句超級超級重要）
  - 先做（會惡化 + 昂貴）工作，然後才做（不會惡化 + 不昂貴）工作

Punctuated Equilibrium

- 物種的演化不是一步一步漸漸來的（ phyletic gradualism ）
- 科學家相信：物種大部份時間幾乎沒有什麼變化，然後在短時間內高速改變（ Punctuated Equilibrium ）

變革需要氣力

- 大部份團隊，都不想離開舒適圈（ comfort zone ）
  - 現有東西已經經過實戰考驗，其安全性和效能都是可以預計的
  - 使用新東西需要學習成本
  - 新東西需要時間踩雷
- 要帶領團隊離開現在的 Stasis ，在任何時候都是高度困難的
  - 想導入新東西，可以找小型獨立專案 / 模組來先實驗
  - 動手證明使用新東西後會更好，人們便自然會跟隨
  - 要引入新東西，請先看清你的「能量」如何

完