backend 課程（第二課）

by

Triton Ho 

今天大綱

- 淺談系統安全
- 進階 API 設計概念
- 實戰心得

淺談系統安全

系統安全前言

- 「系統安全」是一個很嚴肅的話題
  - 如果 LOL 伺服器當機了，大家罵完後便繼續玩
  - 如果 LOL 伺服器的信用卡資料被偷，隨時賠錢賠到倒閉
- 太多太多的人，總是網上看了別人的範例後，只理解一半，然後再自行創作另一半，最終引發嚴重安全性問題
  - 前人的架構大都經過充分論證，相對上不易有漏洞
- 再說一次： Defense-in-depth
  - 多種的安全措施，讓即使其中一種安全措施被壞人攻陷後。剩下的安全措施能減低損害，增加壞人進一步入侵的困難度

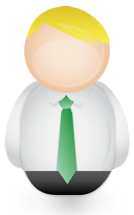
重要安全概念

- HTTPS
- Password hashing and salting
- 預防 SQL injection
- 多層資料庫權限
- Audit table
- JWT

淺談 HTTPS

- HTTPS = HTTP + TLS
- 以 2016 標準，所有網站都應該使用 HTTPS ，告別 HTTP
  - google 的搜尋排名， HTTPS 網站有加分耶
- 能有效防止各式各樣的攻擊，不過.....
  - 你使用太舊的加密方法，例如： DES
  - CNNIC

淺談 MITM



Step 1:
無知用戶連上了 Wifi
並且在 firefox 上打上
www.stupidbank.com

Step 2:
在用戶 DNS 查詢
www.stupidbank.com 時
壞人的 wiki 故意返回錯
誤的 123.123.123.123

Step 0:
壞人在街上建立了
免費的 Wifi ，等待
獵物上門

壞人的
Wifi

Step 3:
用戶對 stupidbank 的
通訊，全被帶到壞人
的 123.123.123.123

www.stupidbank.com
137.149.1.9

Step 4:
為了不被用戶發現。
壞人伺服器把從用戶
來的 Request 丟給
stupidBank ，然後把
stupidBank 的回答傳
回給用戶。
所以用戶可能看起來
一切沒有問題，而壞
人則可以在不被發現
下偷看用戶的密碼

壞人伺服器
123.123.123.123

HTTPS 流程（大幅簡化版）

Password 存放問題

- 如果用戶的密碼以明碼存到資料庫，系統管理員是一個壞人 ~
  - 壞人直接登入資料庫，然後得到受害用戶的密碼
  - 然後壞人使用用戶密碼，大大方方地登入系統，以受害用戶的身份把他的錢全換走 ~
  - 因為壞人是用正常方法登入系統，所以他的行為和其他用戶的正常行動沒有分別，極難被發現
- 在 2016 年新建立的專案，如果還在使用明碼，其工程師請切腹謝罪

One way hashing

- hash 是接受任意長度的資料，然後輸出固定長度結果的東西
  - 例子： input mod 10 便是一個簡單的 hash function
- One way 特性，就是只看 function output 和 function 本身，不能輕易找出 function input
  - 所以， input mod 10 沒有 One way 特性
    因為例如我們看到結果是 2 時，我們可以輕易舉出 input 是 12, 22, 32……
  - sha256 便是一個 one way hashing
    53d563477a60d0f25e95eb2a7b738e5ab65b18cfe8a9d0cbaca5ba948af8db89
    賭賭看：能找出產生以上 sha256 digest 的學員，我把學費退給你 ~

Password hashing

- 我們會把用戶的密碼進行 one way hashing ，然後把結果 H(password) 存放到資料庫中
- 之後用戶登入系統時，我們把用戶提供的 password 進行一次 hashing ，然後對比資料庫中的 H(password) 是否相等

Rainbow attack

- 如果壞人只需要任意一個用戶戶口便足夠
- 壞人為大量的 random string 進行 hashing 然後把 <H(s), s> 存起來。只要壞人的這個表足夠大，而系統中的用戶數量足夠多，便很有機會發生 H(s) 是相同的狀態。
- 這時，壞人只需要拿他手上 <H(s), s> 的資料表，便能拿 s 作為用戶密碼登入系統
- 註： Rainbow table 是讓壞人在建立 <H(s), s> 這個表時大幅節省空間的手法，有空請自行維基

Password salting

- 現在，我們在建立新用戶時，我們先產生一個隨機的 string (salt) ，然後把 H(salt + password) 和 salt 存到資料庫
- 當用戶登入時，系統便先從資料庫把 salt 拿出來
  然後再把對比 H(salt + password) 和資料庫中的數值，便知道用戶的密碼是否正確了
- 現在壞人需要為每一個 salt 建立獨立一張 rainbow table ，所以其需要的 CPU 和 Memory 成本是本來的 N 倍（ N = 系統中用戶數量）

預防 SQL injection

- 老實說：在 2016 還發生 SQL injection 的系統，其負責主管理應以死謝罪。
- 簡單來說：如果你的系統有這樣的程式碼，你的系統已經有 SQL injection 風險：

  q = “select * from users where username = ‘” + username + “‘ and password = ‘” + password + ”””
- 請記住： Parameter escaping 不是絕對安全的方式
  - 你不知道明天 SQL 會不會再增加特殊符號
- 只有 Parameterization 才是真正安全
  - 請善用其底層是用 Parameterization 來工作的 ORM ，輕鬆又安全 ~

多層資料庫權限

- 很多人貪圖一時方便，把擁有最高權限的 super_user 給 Application Server 使用，一旦 AS 被攻陷，壞人便得到資料庫的最高權限
- 你以為壞人只會 drop table 嗎？你未免太低估人壞的邪惡智慧了！
- 一個好的資料庫，至少有這三種用戶：admin_user ， normal_user ， readonly_user

資料庫權限： admin_user

- 不等於 super_user
- 該資料庫的擁有者，擁有該資料庫的一切權限
- 是資料庫中一切物件的擁有者，例如：
  - Tables
  - Triggers
  - Stored Procedures
- 除了要更動 database schema ，或是新減資料庫中物件，否則絕不使用
- 只能由最可信任人員持有

資料庫權限： normal_user

- 給予 Application server 使用
- 沒有建立 / 刪除資料庫物件權限
  - 這絕對重點！！！
- 沒有 truncate table 權限
- 只有 select / insert / update / delete 權限
- 只有對 stored procedure 執行的權限

資料庫權限： readonly_user

- 一般情況下，我們不應該輕易連上 production database 來除錯的。請把這看成緊急 / 最後的手段
- 當你連續爆肝了 24 小時後，你會覺得 select 和 truncate table 看起來差不多的
  - 別那麼自信拿 admin_user / normal_user 來除錯，否則你犯錯時別怪老闆無情

Audit table

- 對於敏感資料（例如銀行系統中的用戶結餘），我們會想記錄其所有的改動。
  - 例如我們想記錄 user_balances 的改動，我們便會建立 user_balances_audit 這個資料表
  - 然後在 user_balances 建立 on insert, on update, on delete trigger ，把改動自動抄到 user_balances_audit

Audit table（續）

- normal_user 不應該對 audit tables 有任何權限
- trigger 擁有者是 admin_user
- trigger 的權限要基於建立者的而不是執行者的
  - 不同的 RDBMS 具體指令會有分別，但理念是相同的
  - 例子：在 PostgreSQL ，建立 trigger function 時需要使用 SECURITY DEFINER 關鍵字

如果不用 Audit table

- 有笨蛋不喜歡用 triggers ，所以他在 Application tier 改動 user_balances 時，把改動抄送一份到 user_balances_audit
- 這樣， normal_user 便擁有對 user_balances_audit 的改動權限
  - 邪惡的系統管理員便能用 normal_user 登入資料庫，刪掉在 audit table 的犯罪證據
- 當壞人攻陷 Application server 得到 normal_user 帳戶後，壞人會直接改動 user_balances。這樣壞人的活動反而不會出現在 audit tables

未有 jwt 的時代： SessionId

- 用戶登入系統時，伺服器檢查用戶密碼是正確後.....
  - 隨機產生一組 SessionId
  - 把 SessionId->(UserId) 存到 Redis ，並且設定 TTL 為 30 分鐘
  - 把 SessionId 傳回用戶
- 之後用戶的所有 request ，都會把 SessionId 放在 http authorization header
  （絕對別放在 GET method 的 QueryString ）
  - 伺服器利用 SessionId 從 Redis 中找回 UserId ，便能知道發出這個 Request 的用戶身份
  - 然後，伺服器把 Redis 中該 SessionId ，重新設定 TTL 為 30 分鐘

使用 SessionId 的缺點

- 所有的 Request 都要先檢查 Redis ，去查證 SessionId 是否真確
- 萬一 Redis 當掉，所有用戶都需要登入
- 一旦壞人突破防火牆接觸到 Redis ，他便能建立受害用戶的 Session。 然後，他便能利用他所建立的 SessionId ，繞過登入程序偽裝成受害用戶。

jwt

- 請看 http://jwt.io/
- 一個 jwt 有三部份： header ， payload ， signature
- Header
  - alg ：說明這個 jwt signature 所使用的演算法（個人推薦 RS256 / RS512 ）
- Payload
  - 你喜歡放什麼也可以，一般人會至少放上 userId 和 exp
  - exp ： jwt 到期時間（ Unix Timestamp 格式 ）
- Signature
  - 把 header ， payload 和 secret(HMAC) / private key(RSA) 一起計算得出的 SHA256 / SHA512 數值
  - 因為 secret(HMAC) / private key(RSA) 只存放在 Server ，所以用戶沒法自行產生
    用戶偷改 header / payload 後，其 signature 將不會吻合

使用 jwt 流程

- 用戶登入系統時，伺服器檢查用戶密碼是正確後.....
  - 建立一個 jwt 物件， alg 為 RS256 ；在 payload 中放入 userId ， exp 為 30 分鐘後；然後以 server 的 private key 建立 signature
  - 把 jwt 傳給用戶
- 之後用戶的所有 request ，都會把 jwt 放在 http authorization header
  - 伺服器利用 public key 驗證這個 jwt 是否真確；並且檢查 exp 是否已經過期從 payload 中找回 UserId ，便能知道發出這個 Request 的用戶身份
- jwt 快到期前， client side 發出 renew jwt request ，向 server 拿到新的 jwt （ exp 為現在時間 30 分鐘後）
  - 不建議在每個 response 都傳回新的 jwt

使用 jwt 的好壞

- 好處
  - 沒有了 redis ，不用再擔心 redis 當掉後 Session 流失問題
- 壞處
  - Client side 需要定期更新 jwt ，多了一點小麻煩
  - 一旦壞人拿到 secret(HMAC) / private key(RSA) 便可以偽裝成任一用戶
    - 所以我才會建議 RS 演算法，並且把 jwt 的發行放在獨立機器

進階 API 設計概念

進階 API 設計概念前言

- 還是一句：別再為 uniform interface 吵架了
- 擁有 stateless protocol 的 backend 依然是 stateful 的
  - 如果 backend 是 stateless 的，那麼 postgreSQL 和 Redis 用來做什麼

進階 API 設計概念

- API 版本控制
- 「 middleware 」
- 「違反」 RESTful
- Idempotent API
- Optimistic lock
- Stateless protocol
- Long polling
- Asynchronous API

API 版本控制

- API= 規格，一旦定下便很很很難改變的東西
  - 要是台電跟你說明天電壓從 110V 升到 150V ，然後告訴你有電用使好，你猜會不會暴動？
- bug= 程式不合乎宣稱的規格
  - 所以 debug 是單純讓程式合乎宣稱的規格
  - Design flaw（設計錯誤）和 bug 是不同的
- 理論上 debug 不會影響到 API 使用者的，可是.....
  - 對方有可能早便知道 bug 的存在，並且作出了 workaround 去配合那個 bug
  - 對方把 bug 看成 feature 了

API 版本控制（續）

- 所以，你不應該做出 abc.com/users 這種 endpoint
  - 別做出 abc.com/user?version=1 這種笨蛋事
- 正確的 endpoint ，應該是 abc.com/v1/users
- 一旦 API 的規格改變，你應該：
  - 建立 abc.com/v2/users
  - 宣告 abc.com/v1/users 為 deprecated ，並且定合理的 grace period
  - 在 grace period 其間，新舊版本的 API 並行
  - 結束 grace period 後，刪除舊版本的 API

「 middleware 」

- 不管任何時候，都不應該把 auth token(jwt / sessionId) 放在 queryString 中
  - 否則這個 auth token 便會連同 url 一起，存到 Application Server 的 Access log 中
- 可是嘛，有一天你在為系統建立電郵身份驗證時，用戶告訴你：
  - 他們的 email 只支持 plain text
  - 他們只願意作一次 copy-and-paste ，所以需要把認證碼以 queryString 的形式跟 url 放在一起
  - ~~最好能直接讀取用戶腦電波，讓用戶 keyboard/mouse 都不用~~

「 middleware 」（ 續 ）

- 用戶會丟 /v1/emailAuth?token={token}
- 我們可以在用戶和我們的 Application Server 之間，建立一層 middleware
  - 把 /v1/emailAuth?token={token} (GET) 轉成 /v1/emailAuth ，並且把 token 放進 HTTP header ，然後再把 Request 丟給 Application Server
- 一般來說，企業客戶有很多奇怪保安要求，為每一企業客戶在主程式中增加額外的 API 只會讓程式亂得無法維護。這時應該交給 middleware 解決

「違反」RESTful

- 如果在網頁中，用戶可以在同一頁面輸入貓的資料，還有罐罐的餵食詳情
- 如果用 uniform interface 的 API ，便會有 /v1/cats 和 /v1/cats/{catId}/catFoods/{catFoodId}
  - 這會有多個的 POST request ，會造成大量的 network overhead
  - 如果一隻貓有 2 種罐罐的餵食詳情。在建立貓的記錄和第一個罐罐的記錄後，網絡斷線了，怎麼辦

「違反」RESTful（續）

- 一個 request 是否「完整」，是基於商業邏輯和用戶體驗的
- 正常的系統設計，應該是：
  - 用戶使用流程
  - （中間很多很多打架討論）
  - API 設計
- 剛才例子，/v1/catAndCatFoods (POST)
  - 雖然違反 uniform interface
  - 但是，更關鍵的 Stateless protocol 並沒有違反
  - 而且，大幅提升用戶的體驗

Idempotent API

- 有可能客戶端（邏輯上）發了一個 Request ，但是伺服器收到兩份完全相同的
  - 網絡是不穩定的嘛
- 試想一下，如果伺服器收到 2 次 Delete Request ：
  - 第一個 Req ：伺服器把物件刪除，然後返回 OK(200) / NoContent(204)
  - 第二個 Req ：伺服器無法找到物件，返回 NotFound(404)

Idempotent API（續）

- 真正致命的，是重覆的 POST request
- 如果沒有處理，用戶將會建立兩份完全相同內容的物件
- Deduplication 需要 locking 和 response 暫存，需要一個高效能的 Redis
  - 要看閣下的需求，才能決定是否值得

Optimistic lock

- 雖然名字中有 Lock ， Optimistic lock 不是 locking
- 在 UPDATE endpoint 中，我們可以：
  - select * from tableX where id = <id>
    如果 object 不存在，直接返回 NotFound(404)
  - Update tableX set attribute1 = <attribute1>,
    last_update_time = now()
    where id = <id> and last_update_time = <lastUpdateTime>
    如果 update record count = 1 ，返回 OK(200)
    如果 update record count = 0 ，返回 Conflict(409)
- 如果客戶端收到 Conflict(409) ，便知道這個資源在用戶讀取後已經被別人改動過，所以必須再讀取一次，修復之間的 inconsistency

Stateless protocol

- 學術上的 Stateless protocol ，現實中 100% 實踐的後果，只會帶來更大的麻煩
  - 每次都要把完整狀態在客戶端和伺服器之間丟來丟去，很浪費頻寬
  - 在大老二中，你能把整個牌局的狀態都丟給客戶端嗎？
  - 在伺服器，你要額外檢查客戶端是否誠實，沒有偷改數據
- 最簡單一句： 100% 純正的 Stateless protocol ，會要求你每個 request 都把 username 和 password 都加在 request 中耶

Stateless protocol（續）

- 真正重要的：
  - Idempotency
  - Consistency
- 即使你的 Request 需要引用存在伺服器的數據，只要你能肯定這份數據在過程中沒被改動過，其效果便跟 100% 純正的 stateless protocol 沒有分別
  - Optimistic lock 或 StepId 是你的好幫手

Long polling

- 傳統的 Server-Client 模式，都是 client 向 server 請求，然後 server 作出回答
- 有些應用，是需要 Server 主動把迅息傳給 Client 的
- 20 年前，大家看網上即時籃球比賽，那時大部份網站還沒有 long-polling……
  - 為了知道自己心愛球隊是否得分，大家只好不停按 <F5>
  - 結果便消耗掉~~大量~~鍵盤上網數據量

Long polling（續）

- 如果在 IOS / android ，能使用 APNS / GCM 便優先使用
  - 在 APNS / GCM ，所有的程式只會共享同一個 TCP 連線，省下大量電力和數據量
- 長期沒有使用的 TCP 連線，有可能斷掉了 Server / Client 也不知道。一旦有訊息要推送時 , 便要等待 Client 發現斷線了，重建連線才能推送
  - 在流動環境特別明顯，小心 TCP 連線「變爛」的問題
  - 之前經驗，放置一段時間不使用的 android ，其 GCM 的延遲可能達到 15-30 分鐘
  - 有沒有想過：一些寫得很爛的軟體為何會快速用掉手機電池？

Asynchronous API

- Async API 不會帶來系統效能的提升
  - 因為 Async API 要動用 MQ 和 long polling ，反而要用更多資源
  - 相反， Async API 是用在會消耗大量資源的工作，讓你的系統不會當掉

如果沒有 Async API

- 老闆想要看過去 12 個月的統計數據報表，這需要讀取 50GB 的數據
  - 問題 1 ：如果這份報表需要 2 小時才能產生，你的伺服器的 session timeout 只有 30 分鐘，怎麼辦？
  - 問題 2 ：生產報表其間，老闆的瀏覽器便不能關
    - ~~如果老闆使用某 I 字開頭的軟體，在等了 1 1 9 分鐘才當掉.....~~
  - 問題 3 ：如果老闆們現在想看的是 1 0 份報告，而這 1 0 份報告都同時查詢資料庫。那麼，老闆便要等 2 0 小時。

Async API 例子

- 老闆說：我想看報表 X ，並且留下他的 email
- Application Server 檢查老闆提供的報表參數和 email 是否正確。如正確，則把以上資料丟進 MQ ，然後告訴老闆 Accepted(202), 叫他安心等待
- Report Worker 從 MQ 拿到報表參數和 email ，然後生產報表
  - 如果 Report Worker 正在正生產另一老闆的報表 Y ，則先把報表 Y 的工作做完，然後才會向 MQ 查詢
    所以系統中同一時間只有一份報表正在生產
- 做完報表後， Report worker 把報表電郵給老闆

實戰心得

實戰心得前言

- 這個世界沒有 golden hammer
  < 輕鬆 + 高效 + 便宜 > = 騙子
  - 你看每個衛生棉廣告，他們都會說有什麼新科技，性能有什麼大突破的
- 要輕鬆只有一個方法：你的知識和實戰經驗讓你懂得迴避地雷，看穿騙子文宣

實戰心得

- 金流系統和 2pc
- Worker 與第三方系統
- Dependency injection
- 環境變數
- 多線程問題
- Non-local Caching
- Performance vs Consistency

金流系統

- 每月帳單
  - 你收了用戶雙份的錢，用戶會砍你
  - 你沒收到用戶的錢，老闆會把你五馬分屍
- 向用戶收錢的是第三方的系統（ Paypal / Stripe / 銀行）
  user_payments 是在我方的 RDBMS
- 問題是：怎讓「收錢」和 update user_payments 放在同一 atomic operation ？

沒有 2PC 的例子 1

1. 會計系統建立 user_payments ，其 status 為 Initial
2. Payment Worker 向金流系統收錢
3. Payment Worker 把 user_payments 從 Initial 改成 Success

如果第二步和第三步之間， Payment Worker 當掉了呢？

沒有 2PC 的例子 2

1. 會計系統建立 user_payments ，其 status 為 Initial
2. Payment Worker 把 user_payments 從 Initial 改成 Success
3. Payment Worker 向金流系統收錢

如果第二步和第三步之間， Payment Worker 當掉了呢？

淺談 2PC

- 你有物件 A 和物件 B ，你希望對 A 和 B 的改動只能最終一起成功 / 一起失敗。
  - Voting Phase
    - 把物件 A 和物件 B 上鎖，並且準備物件改動後的狀態
    - 一旦失敗， Coordinator 便會執行 Rollback
  - Completion phase
    - 改動物件，並且為物件解鎖
    - 即使當掉，之後執行的 Coordinator 也會繼續 Commit 行動
  - Coordinator
    - 一個當掉後也能重新起動的東西
    - 一般來說 2pc 會為交易設定時間。過時後， Coordinator 便會作出 Rollback / Commit
- 現實世界中， 2PC 會跟教科書中寫的不完全相同。
  不過，只要有 Voting Phase 和 Completion phase ，便算是 2PC。

2PC 和金流系統範例

1.會計系統建立 user_payments ，其 status 為 Initial

2.Payment Worker 呼叫 Stripe API ，建立 Stripe.Charge 的物件，這個 Charge 的 captured 屬性為 false ，所以現在還未向用戶的信用卡收款。

3.Payment Worker 把第一步 Stripe.Charge 物件的 id 存進資料庫
update user_payments set stripeChargeId = @Stripe.Charge.Id where id = @id;

4.Payment Worker 呼叫 Stripe API ，把第一步建立的 Stripe.Charge 的 captured 屬性改為 true ，向用戶的信用卡收款。

5.Payment Worker 把 user_payments 從 Initial 改成 Success

- 以上例子中：
  - 第 2 和第 3 步屬於 Voting Phase ，第 4 和第 5 步屬於 Completion Phase
    - Payment Worker ，還有 7 天後 Stripe 自動把未完成交易自動 Rollback 的 Worker ，都是 Coordinator 一分子

Worker 與第三方系統

- 我們沒有跟第三方系統結拜
  別讓他們系統死掉時，我們系統也一起當掉
- 例子：你使用 GCM 服務時，你的 HTTP POST 沒有設定 timeout
  - 如果 google GCM 發神經，他接收了你的 POST request 後一直也不 response 呢？
- 如果你使用 Worker 來發 GCM ，會當掉的也只有你的 GCM Worker ，而不是整個系統
- 即使 GCM 當掉了也好，這段時間的 Message 停留在 MQ 中，等待 GCM 恢復了便能重新傳送，不會有訊息丟失
- 延伸思考？

Dependency injection

- 理想的開發環境：在溫暖陽光下，享受海風優雅地工作 ~
  - ~~下次我去旅行絕對不帶電腦！~~
- 當你在沒有網絡的環境，但你現在要 debug 的程式碼，卻要使用 Amazon S3 Service ？
- 所以， AWS 物件應該是由外部提供給 business logic module ，而不是 business logic 中自行建立的。
  當測試 / 除錯時，便能提供一個無需網絡的 MockAWS 物件給 business logic module
- 如果你想做自動化測試， Mocking 幾乎是不能避免的

環境變數

- 像資料庫用戶名字，密碼，你絕對不應該放在程式碼中
  - 一旦 git commit 和 git push 後，你的笨事便全球皆知，永遠留傳 ~
- Dev, UAT, staging, Production 他們都應該有自己的資料庫、Redis、 MQ 設定的。你不應該為每個環境也改一下程式碼
  - 人類是最大的 bug source ，每次人類的改動總會帶來笨蛋錯誤
- 一旦程式碼在 staging 環境測試完成後，其 executable / package 應該是不作任何改動地，由 deployment tool 放到 Production 上

Non-local Caching

- 直接存取放於本機的 Redis ，一定會是最快的
  - 省下內部網絡之間頻寬還有延遲
- 不過，隨系統用戶量變多，你需要 cache 的東西也越來越多的。但是，本機記憶體有物理上限的
- 如果你的 Caching 只使用本機的 Redis ，隨系統流量加大，你的 Cache miss 只會越來越大

多線程問題

- 如果是 Worker ：可以考慮
- 如果是 Application Server ：強烈不建議
- 留在 Application Server 的，都是用不多資源的工作，再為其拆碎進行多線程的利益太少
- 你的 web container / library 底層早已為你替不同的 Request 進行多線程管理，你不需要再自己動手

Performance vs Consistency

- Consistency 最精準的翻譯不是「正確」，是「一致性」
- 想想看：如果現在你在做遊戲活動，要派一萬件史詩級裝備~~（每人有持一把的菜刀能有多史詩？）~~
- 現在有人來領取~~史詩級~~裝備時
  - Begin transaction
  - Update campaigns
    set remaining_items = remaining_items – 1
    where id = @id and remaining_items > 0
  - Insert into user_items……
  - Commit transaction

放棄 Consistency-1

- 現在為了追求效能，改寫成：
  - Update campaigns
    set remaining_items = remaining_items – 1
    where id = @id and remaining_items > 0
  - Insert into user_items……
- 少了 begin TX 和 Commit TX ，所以 Application Server 對 RDBMS 的活動從 4 次變成了 2 次了 ~
- 不過，有機會少於 1 萬人得到史詩級裝備

放棄 Consistency-2

- 改寫成：
  - READ campaigns record from REDIS cache
  - Insert into user_items……
  - Update campaigns
    set remaining_items = remaining_items – 1
    where id = @id and remaining_items > 0
- 這樣， RDBMS 的活動維持是 2
- 有機會大於 1 萬人得到~~史詩級~~裝備

放棄 Consistency-3

- 改寫成：
  - READ campaigns record, campaign_item_delta from REDIS cache
  - Insert into user_items……
  - INCR campaign_item_delta in redis
- 然後，每小時執行一次的 Worker ，以 campaign_item_delta 的值更新 RDBMS 中的 Campaigns 記錄
- RDBMS 的活動大約是 ~1。不過，你需要額外寫一個 Worker

Performance vs Consistency 總結

- 除非你明確知道你系統流量非常龐大，系統開發一律以 Consistency 優先
  - 老闆說未來一百萬個同時在線用戶？能信的嗎？
- 追求效能一般都需要更複雜的系統架構，無可避免會拖慢開發追度
  - 系統效能不好可以之後慢慢改善
  - 慢了把產品推出市場，很可能永遠地輸了
  - 追求效能時，很容易犯錯讓系統反而變慢
    把產品上線，有人流了，自然有錢找專家來好好研究
- 最後一句：沒有用戶的系統，不管背後技術多強，只是垃圾

完