



Developer Student Clubs
National Kaohsiung Normal University

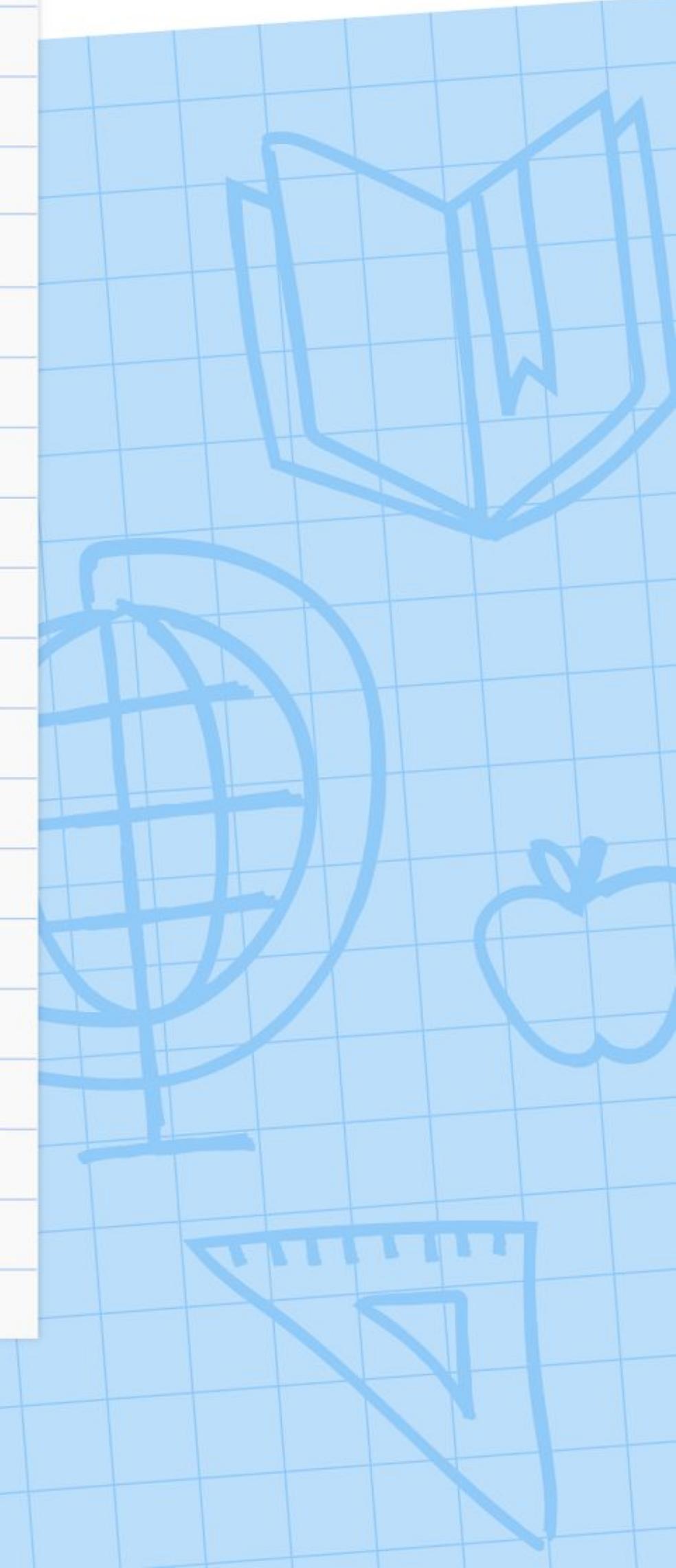
伺服器與通訊原理

對話的藝術，無侷限於人之間。

讀書會主持人：顏榕嶙(Bernie)、李淮恩

日期：09/30 燕巢場、10/14 燕巢場、11/25 燕巢場、12/09 燕巢場

```
filterByOrg = filterByOrg ? study.team_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
const filterStudies = (studies, filterByOrg, filterByStatus) =>
  studies.filter(study => filterByOrg && filterStatus || !filterByOrg && !filterStatus)
```





章節

1. 無處不在
2. 無獨有偶
3. 無邊無際
4. 無所忌憚
5. 無聲無息
6. 無可或缺
7. 無疾而終
8. 無遠弗屆
9. 無懈可擊
10. 無拘無束





Developer Student Clubs
National Kaohsiung Normal University

1

無處不在

生活的各處都是通訊的應用

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const filterPatchStatus = filterPatchStatus ? study.patch_status === filterPatchStatus : true;

function filterStudies({ studies, filterByOrg = false, filterByStatus = false, filterPatchStatus = false }) {
    return studies.filter(study => filterByOrg || filterStatus || filterPatchStatus);
}
```

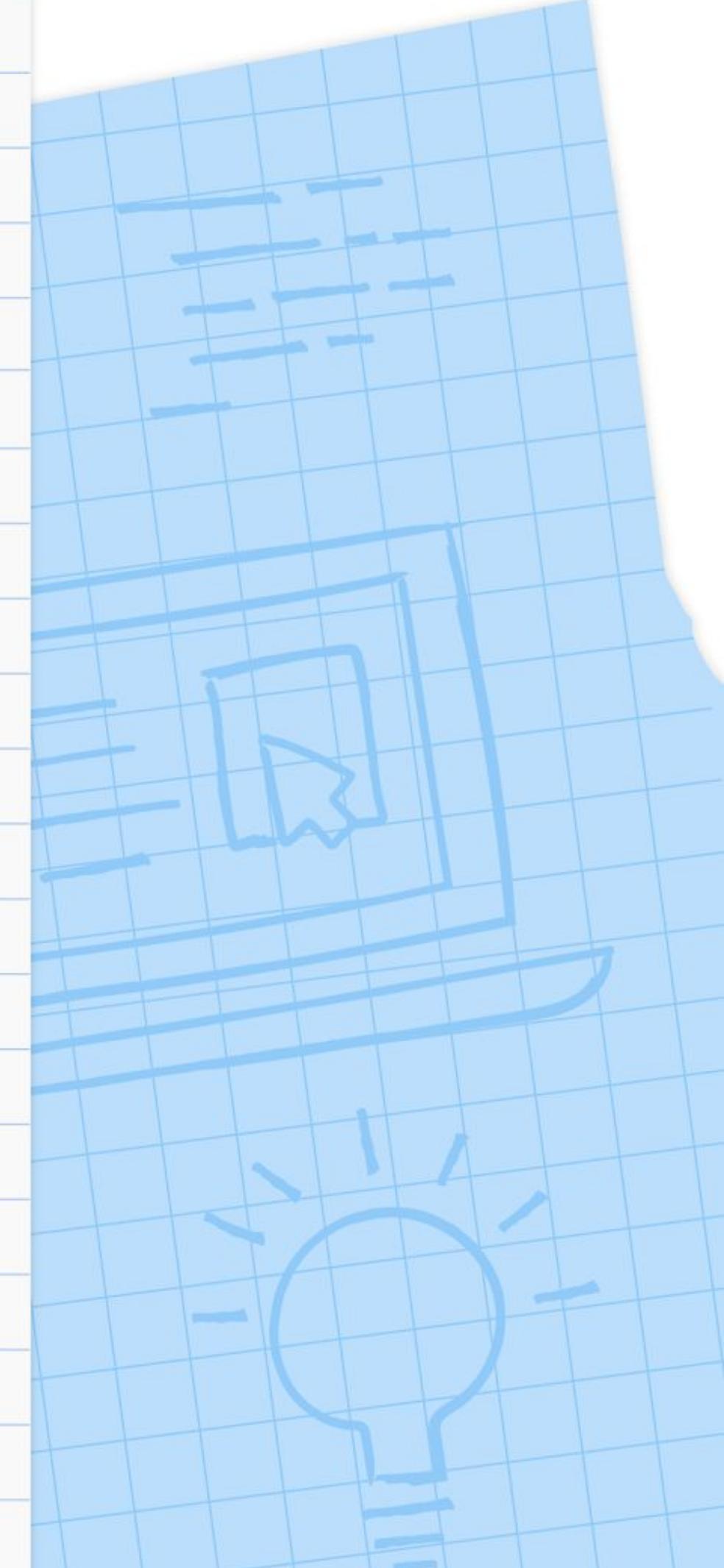


伊馮娜只會做對自己有益的事

學會伺服器和通訊原理並不能統治世界
但是能夠對這世界的運作更加清楚。

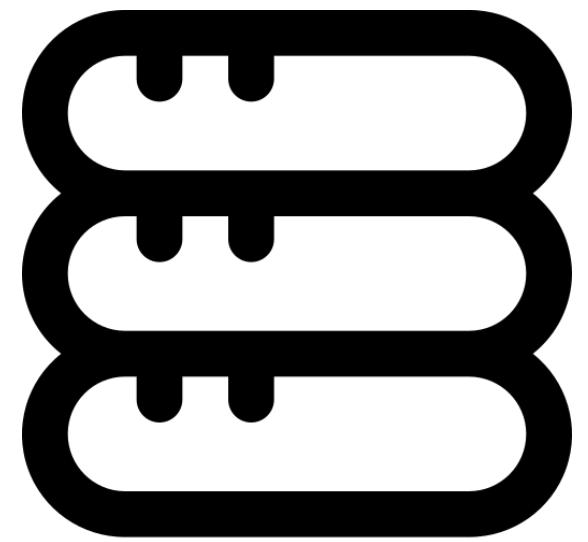
當別人還在「使用」服務時，你就已經
能夠「創造」服務了。

久而久之你會變得不可或缺。



馬上進入主題。

在分析之前，要先學會洞察



聽起來相對疏遠的概念往往其實潛藏在人間

好，對，文青裝夠了，簡單來說大家應該多少都聽過伺服器的概念

有些人甚至自己動手架過，雖然他們自己可能沒有發現

其實說到底，這原先就不是甚麼太複雜的學問



日常可見的蹤跡

網際網路真是個驚人的發明

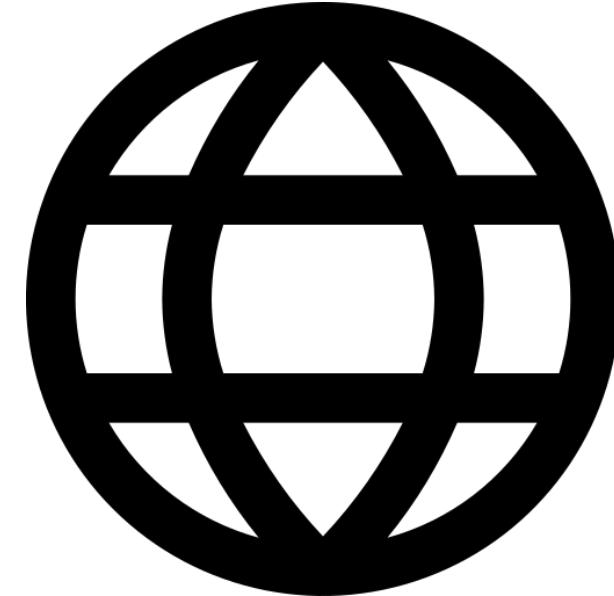
很多線上遊戲都有著一個負責應付所有玩家的伺服器
某些甚至開發讓玩家自己架設「世界」來與他們的朋友同樂

但這並不是全部。

Google、Facebook、Youtube這些平台也需要一個伺服器去管理所有資訊

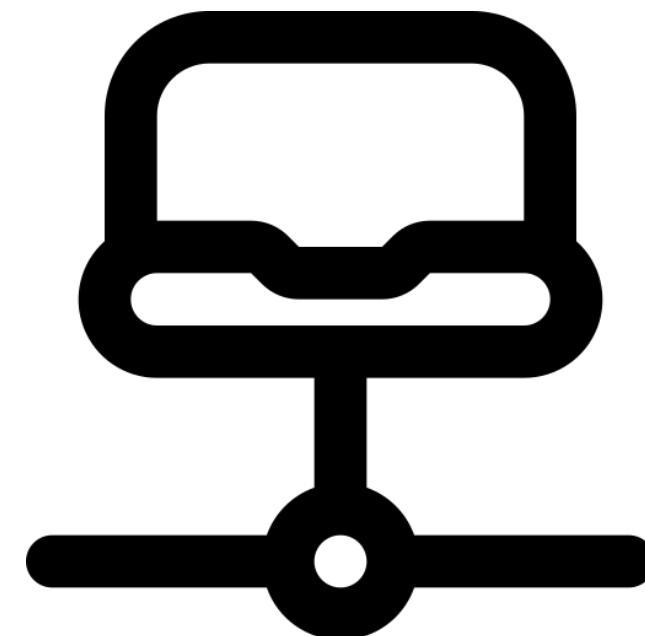
蝦皮、博客來、Amazon等等也必須要能夠雲端運算訂單數據

至此，所有你能想像的到的服務，多少都會用到通訊的技術



隱形的網路

眼不見不為證



好，所以我們知道伺服器就是一個管理各種資訊的大平台

而通訊透過**特殊的方法**可以準確地將資訊從一端傳至另一端

實體例子？你用Line跟別人聊天就是一種（端點：兩個用戶）

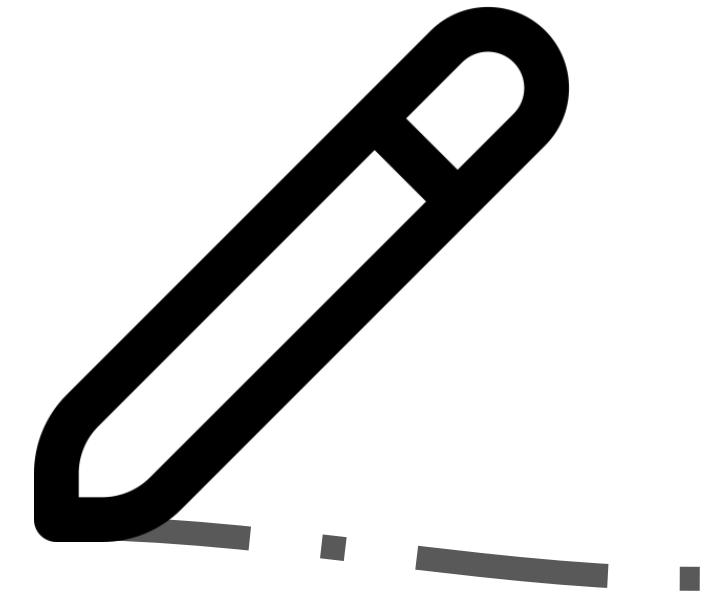
你去查詢帳戶餘額也是一種（端點：你和銀行伺服器）

你在Youtube上傳影片當然是一種（端點：你的電腦和YT伺服器）

甚麼都有，甚麼都不奇怪，想像沒了這東西世界會變怎樣

為接下來獻上伏筆

給接下來的課程帶來簡單的介紹



接下來直到期末之前我們會把所有你應該知道的丟給你們

除此之外，我們希望可能在理論以外的方面帶你們更加熟悉伺服器與通訊

因此抱緊你的電腦，準備好你們的編譯器
每一章節的最後一張投影片會是與程式實作相關的說明

隨著對課程的熟悉，你們也能夠親自操作伺服器的開發
多麼令人期待！(額...沒有嗎？)



神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

2

無獨有偶

端點與請求 - 回應模型

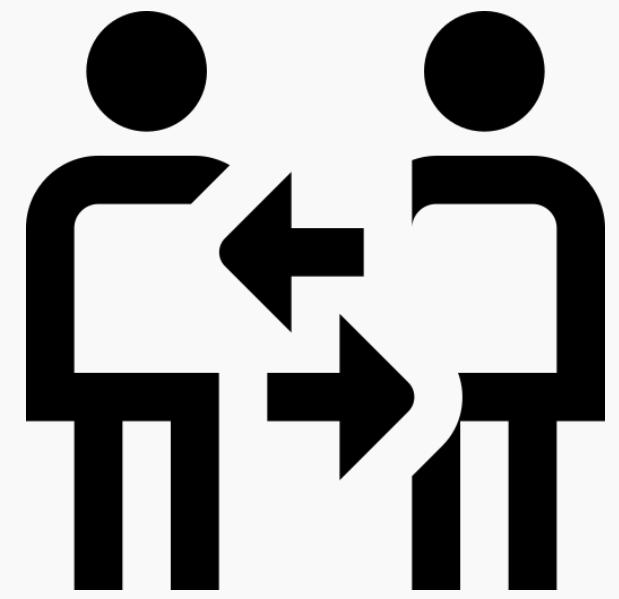
```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const filterPatchStatus = filterPatchStatus ? study.patch_status === filterPatchStatus : true;

function filterStudies({ studies, filterByOrg = false, filterByStatus = false, filterPatchStatus = false }) {
    return studies.filter(study => filterByOrg || filterStatus || filterPatchStatus);
}
```



主客觀念

既然說是通訊，那就不會是單人遊戲



位於話筒的兩端, Alice與Bob, 是我們俗稱的端點

這些端點可能有自行建立連線, 或者透過一個伺服器來導向

當一個端點向伺服器進行了請求, 那麼它將會給予回應



你是我的Server嗎

伺服器端與客戶端，兩個相依而生



通訊原理其實也就是網路基礎的一部分

在一般生活中，要建立連線就需要有至少兩個合法的端點，稱作Server和Client

- 伺服器端的作用是接受連線，並回應客戶需要的資訊
- 客戶端的作用是請求連線，並等待伺服器的回應

兩者之間是獨特的，因此會是一個封閉的網路



連結門

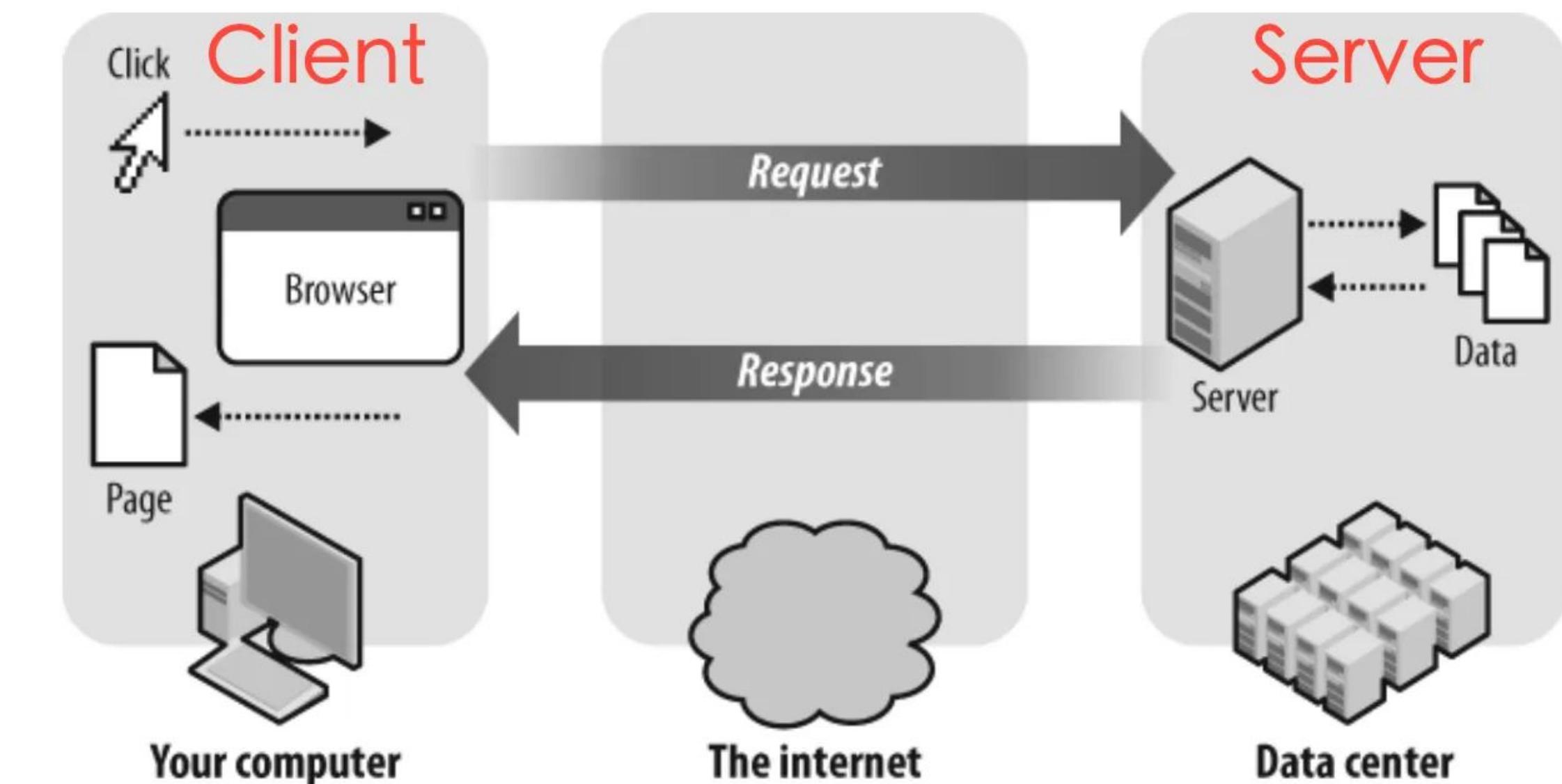
請求、回應，全自動化

一個請求 (Request) 即為客戶向伺服器傳送的資訊封包

在兩者建立了確實的連線之後，這項功能才能實際的運作 (對，不然你就是把資料丟進虛空之中)

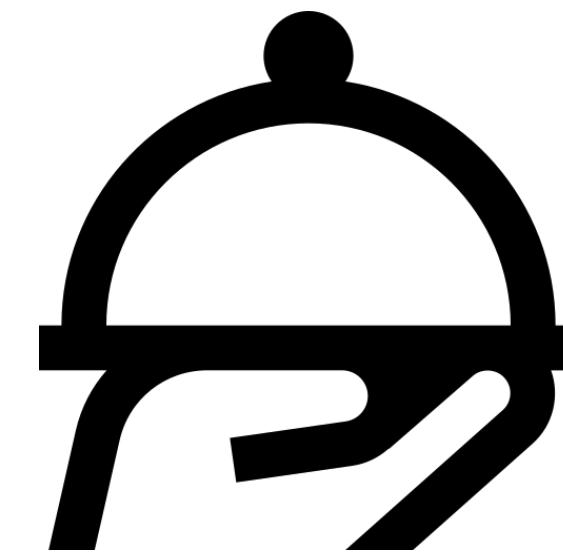
接著，伺服器會確認請求，並做出回應 (Response)

這就是著名的請求-回應模型



網路主機 (Host)

就是伺服器...運行的電腦



說到底，一個伺服器到底是怎麼被定義的

你總不能說隨便一台電腦執行一個腳本就是伺服器？還真的可以（之後會講到）。

Server，字面上來看就是Serve的機器，用來提供一或多個客戶端服務

原始定義是一整個機架電腦，但經過近年的技術發展，伺服器不再必定需要一整個處理器的資源來驅動，甚至只要是計算機都可以運行（某些部分）

因此現在我們將這種搭載伺服器腳本的電腦稱做網路主機。

封閉網路下的模樣

一個中心控管全部





神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

3

無邊無際

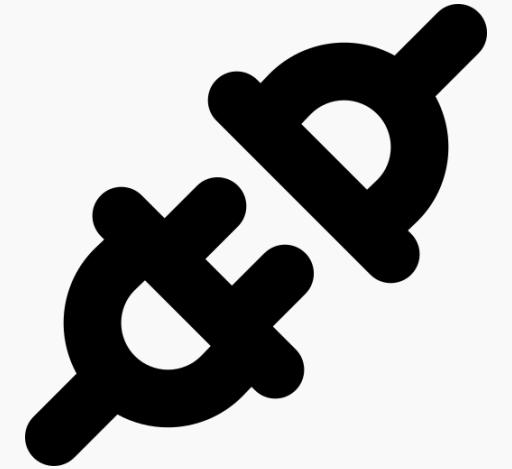
通訊協定的至關重要性

```
filterByOrg = filterByOrg ? study.team_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
const matchStatus = filterStatus ? study.status === filterStatus : true
```

```
function filterStudies({ studies, filterByOrg = false, filterByStatus = false }) {
  return studies.filter(study => {
    if (!filterByOrg) return true
    if (!filterByStatus) return true
    if (filterByOrg &amp;
```



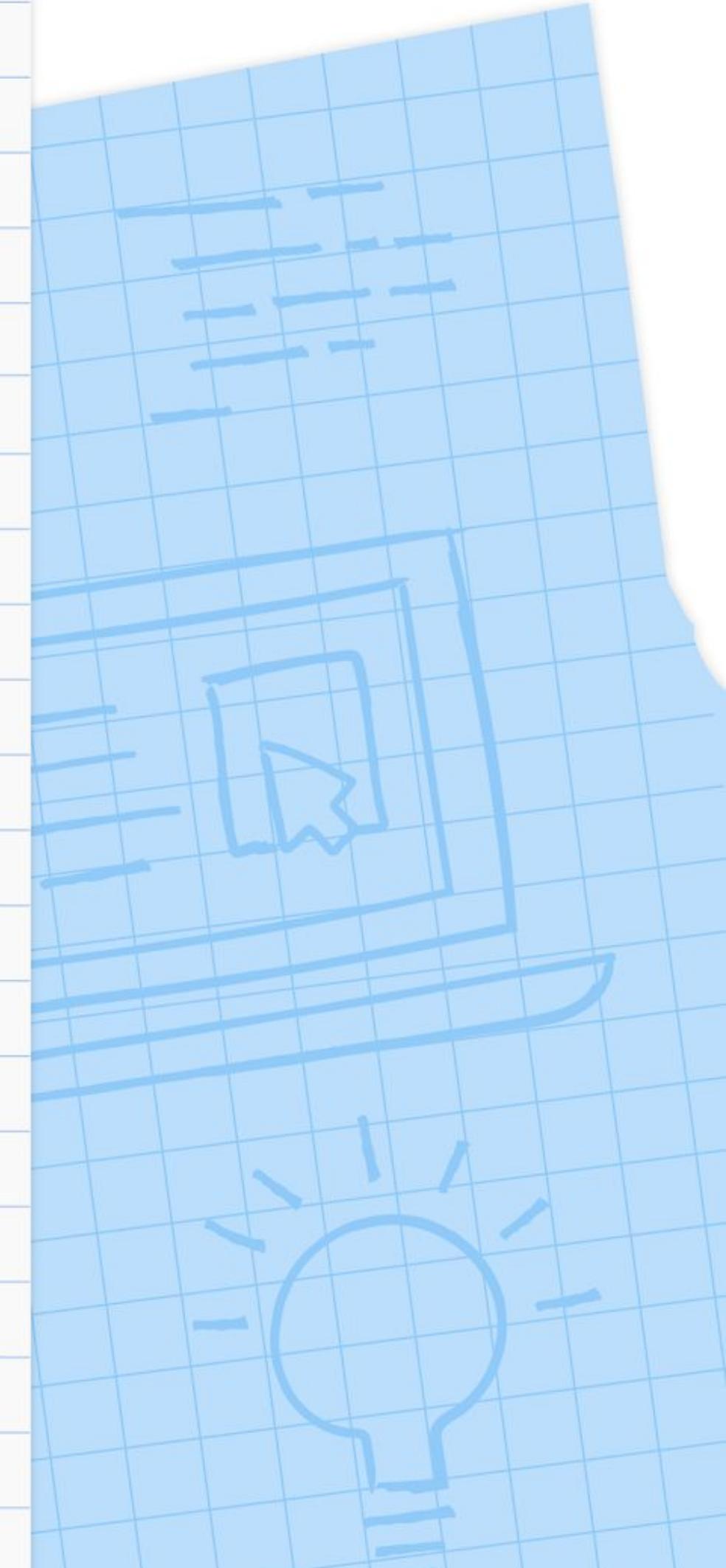
那麼，連線是什麼



我們在通訊軟體上要聯絡別人是多麼的方便，但在實際運作上，這並沒有像想像中那麼容易

為了省去不必要的麻煩，有人發明出了「協定」的概念，小至訊息的傳遞，大至網際網路上的連接都有

要學習網路技術絕對離不開一或兩個協定，請習慣。



一個程序化的流程

可靠的通訊，是資訊安全的基本



所謂的協定，聽上去很厲害對吧？

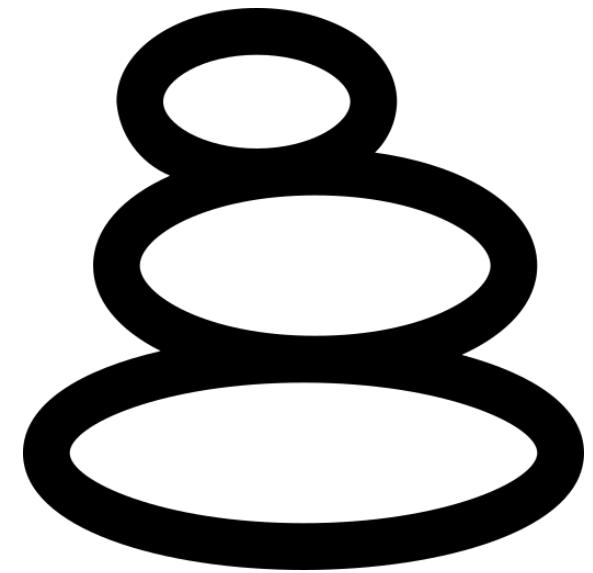
歸根究底那就只是所有人都同意的一套流程，已經廣泛運用到了全世界

這些流程在很多網路架構中都存在著，從網路的基層架設，到高層級的通訊和應用等等，全部都離不開他們自己的一套協定

放心我不會要你們記下所有東西，但在之後的課程中一些核心會反覆地用到

可愛又迷人的網路角色

OSI七層中的網路層...但現在先別了吧



我們假設一個完整的網路基礎架構，並且有一定數量的主機與客機

大家都聽過IP的概念吧，那其實就是虛擬的網路位址，就跟地址一樣，每個地址還有不同的專用信箱 - 埠口 (Port)，用來給不同的服務使用 (這是網路協定)

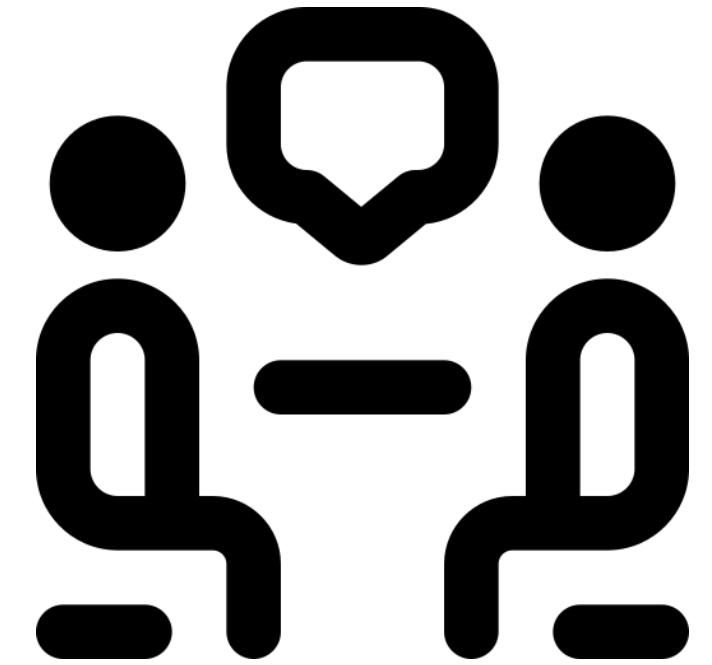
這些服務，也可以說是協定的種類，像是HTTP、HTTPS (對這兩個不一樣)和FTP等等 (這是應用協定)

但我們終究沒有要教計算機網路，因此我們先專注在通訊協定的部分吧！



基本通訊原理

可靠的通訊，是資訊安全的基本



通訊協定處於傳輸層，名符其實就是處理兩端點之間通訊的協定。

我們常見的有TCP、UDP兩種，兩者類似卻有著非常不同的運作方式。

簡單來說：

- TCP比較嚴格，會確保數據的完整性與正確性，但較為緩慢
- UDP比較快、負擔較小，並且可以一傳多，但他完全不管封包丟失

握手三次

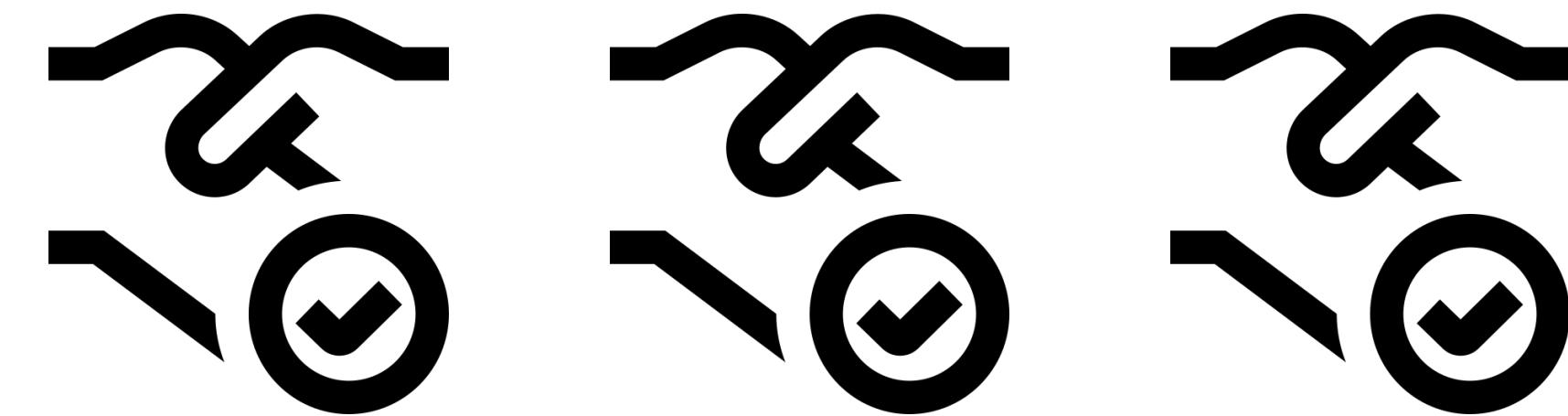
我...我們真的有這麼好嗎?

三次握手是TCP/IP協議當中建立連線的方式，有點像是Pre-通訊的感覺

在這其中，客戶發送給服務器同步請求，伺服器接收之後也做一樣的事

當兩方都確定對方確實存在時，這條連線才能夠安全的被建立起來

然後...就是熟知的資訊傳輸了。



隨便亂丟，接得到就是他的

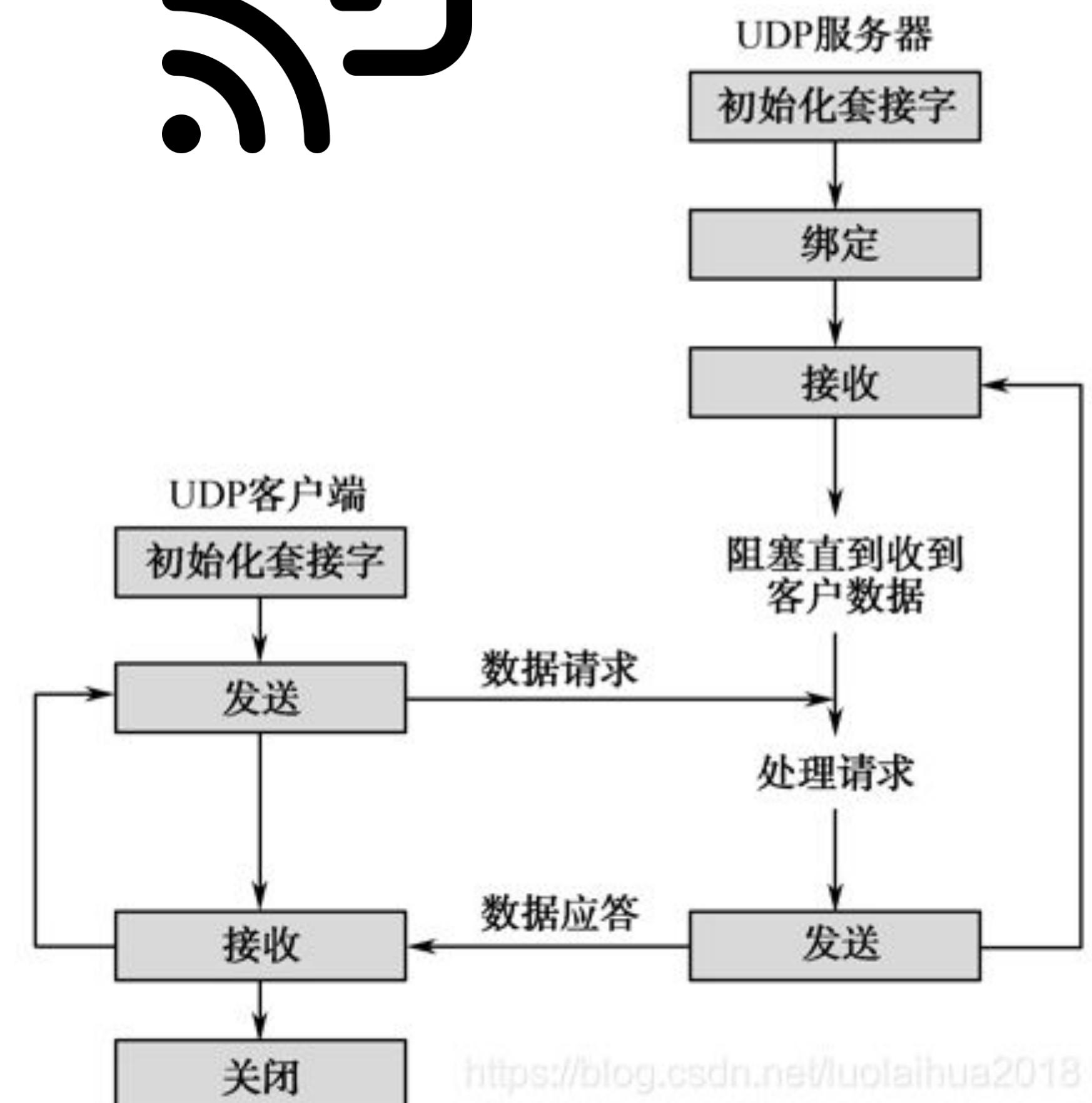
把資料包跟垃圾一樣亂丟，結果卻意外地不錯？

我們知道UDP非常的快，那是因為他的演算法很厲害嗎？是...也不是。

與TCP不同，UDP不會事先去確定一個完整的連線被確立。

只要有客戶端向UDP伺服器發送請求，他二話不說就是往客戶那裏丟，如果客戶不滿足就再丟一次。

簡單，粗暴，卻快速有效。



<https://blog.csdn.net/luolaihua2018>

兩者的差異？

不對，這很明顯吧

TCP準確且安全、UDP快速且多向，各有各的好處

在實際的運用下大概如下：

TCP：線上支付、文件下載、郵件傳輸

UDP：多人遊戲、網路通話、觀看影片

TCP 強調可靠性與數據完整性，適合需要準確傳輸的應用
而 UDP 則注重即時性，適合對延遲敏感但能容忍少量數據丟失的應用



神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

4

無所忌憚

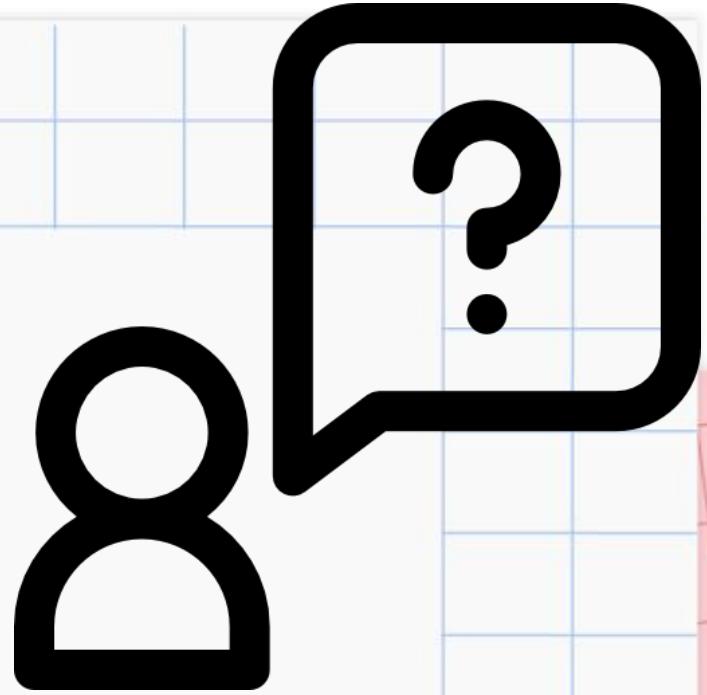
資源管理與負載平衡

```
filterByOrg = filterByOrg ? Study.team_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
const matchStatus) {
  return studies.filter(study =>
    filterByOrg && filterStatus ? study : true
  )
}

function filterStudies({ studies, filterByOrg, filterByStatus }) {
  const filteredStudies = filterStudies(studies, filterByOrg, filterByStatus)
  return filteredStudies
}
```



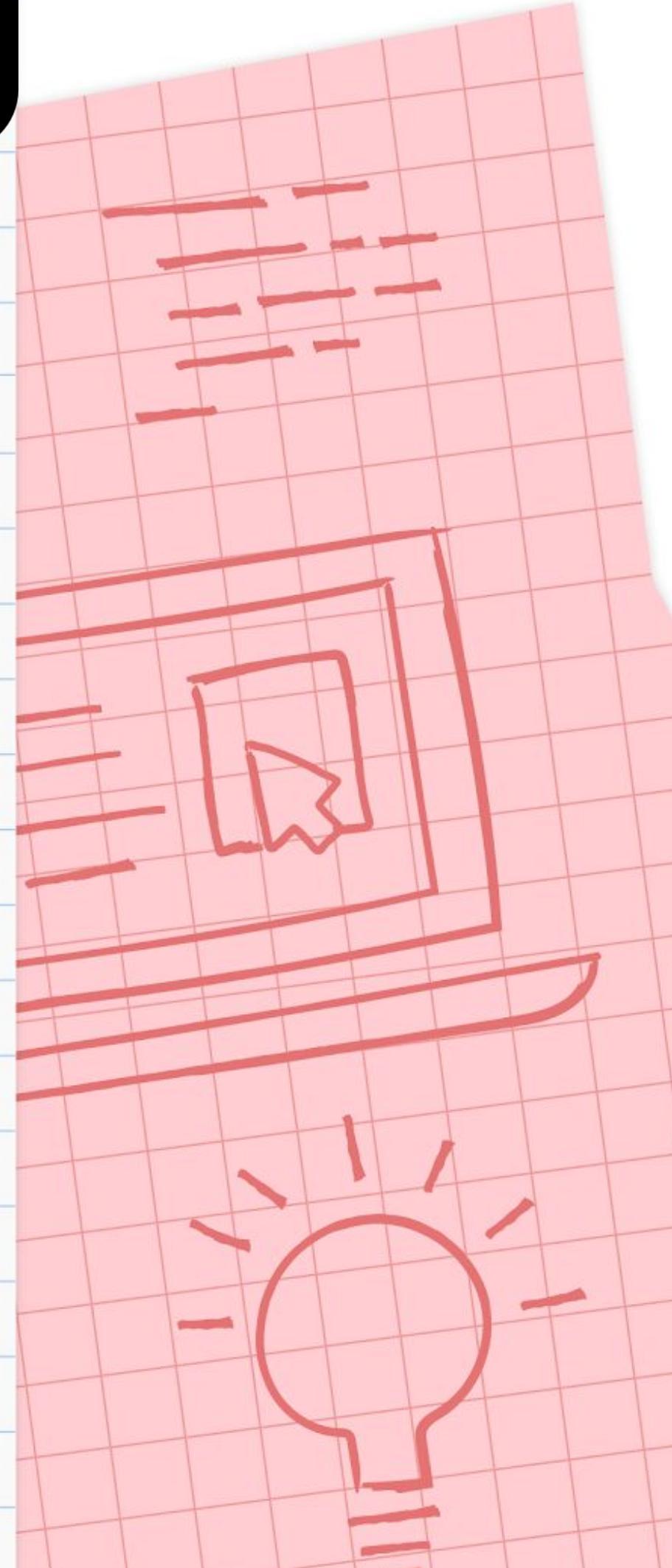
予取予求



我們知道請求 - 對應模型建立了一個伺服器與客戶端的基礎架構，而連線更確保了兩者可以進行互動

不過，是否曾設想過一個或多個客戶端在同一時間段下進行多次請求的情況呢？

這樣子是要排隊？捨棄部分？伺服器自有辦法處理



關鍵資源

監控已知籌碼是必須任務

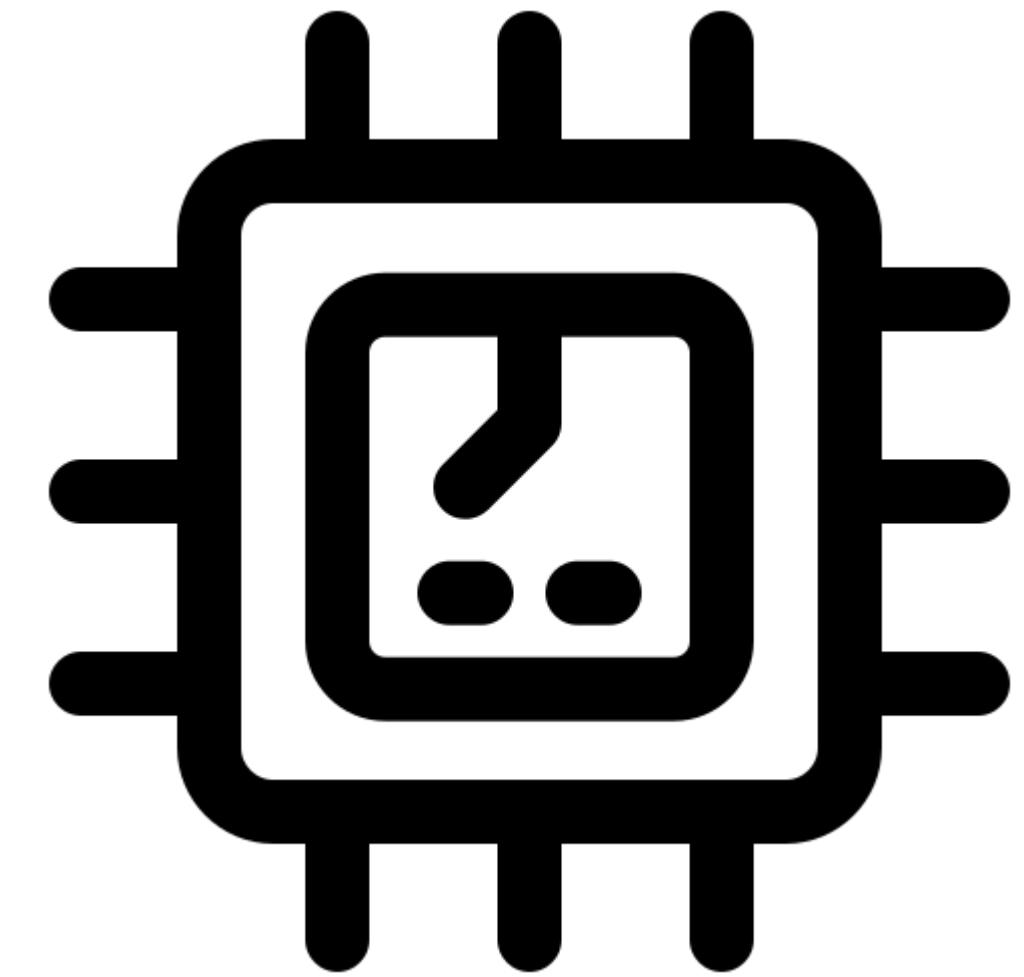
這裡沒人不知道甚麼是 CPU、RAM 等等的名詞吧，非常好

既然說我們的伺服器本體是由電腦(主機)所運行的，那些常見的電腦配件也就相當
是伺服器運行所需要的必備資源

在一個大型商業網路底下，我們也必須得投資相對應品質的硬體來給予其效能

就算只是私人網路也沒有人希望伺服器跑得比馬鈴薯還慢...

說到底，這些所謂的「資源」到底是做什麼的？

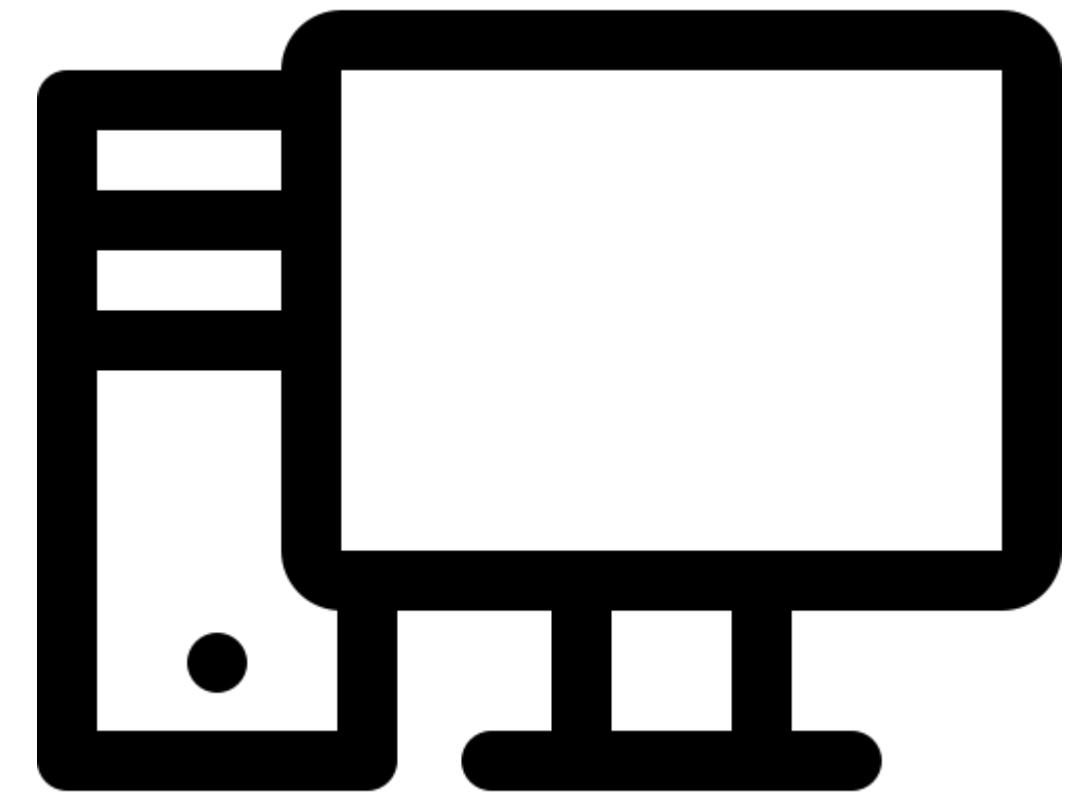


認識你的電腦

接觸計算機這麼多年，也是時候該坦誠相對了

伺服器會需要的關鍵資源主要有下者：

- **CPU**: 伺服器所使用的中央運算元件，用來進行處理與數據歸納，請求的辨識與管理也是在此處運行。
- **記憶體 (RAM)**: 用來暫時存儲數據的短期記憶，當伺服器處理請求時，數據會先儲存在記憶體中。若記憶體不足，伺服器可能會使用硬碟作為虛擬記憶體，導致性能下降。
- **硬碟**: 用來儲存持久性數據，速度比 RAM 慢，但具有持續性。
- **網路頻寬**: 影響伺服器在固定時間內能夠傳輸的最大資料量，直接影響與客戶端之間的連線品質。



打群架是不講武德

被請求之海淹沒了

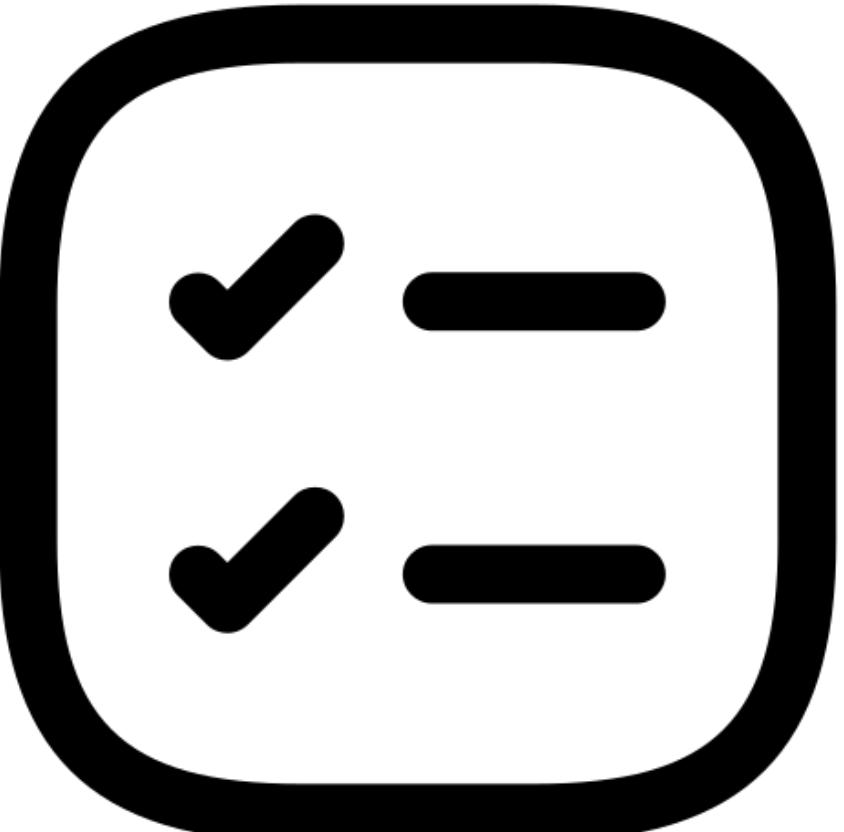
我們理解了自己的硬體，很好，接著該來應付真正的問題了

就如導論所提及的，並不是每次都存在一對一的情況，伺服器端時常會遇到一定數量的請求進入運算，這時我們就需要引入**多工處理**的概念

一個好的多工處理邏輯能夠：

- 提高伺服器響應速度
- 充分利用硬體資源

來提個大家熟悉卻又陌生的名詞吧：Thread (執行緒)





單行道不是很擠嗎？

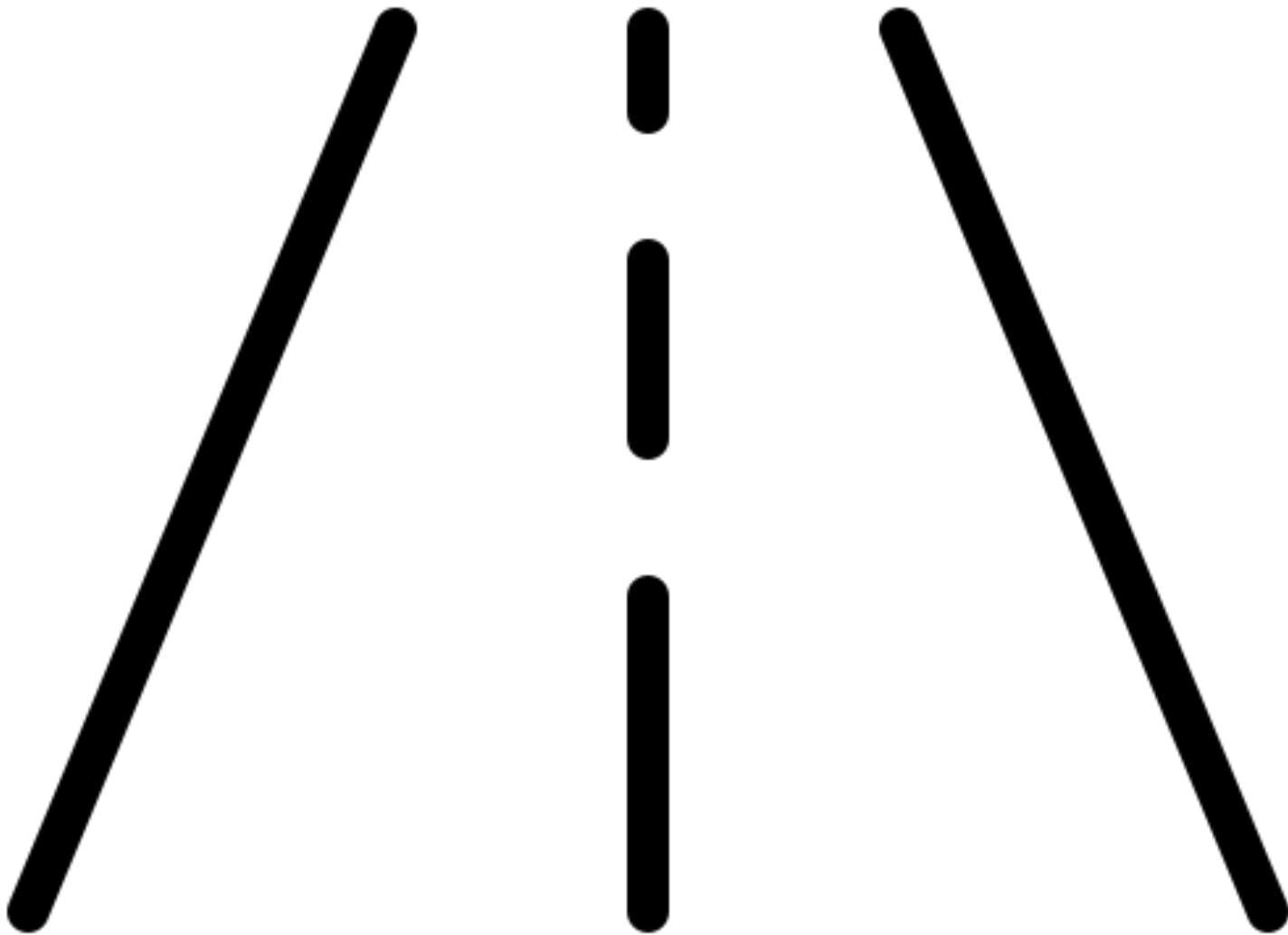
多執行緒的潛在爭議

計算機科學當中，我們把系統的運行單位成為「行程」

而將一個行程給分割成的多個單位即是執行緒

從spawn開始直至finish結束，其中存活的時段即為系統行程的實際運行區段

我們都不希望路上一直塞車，如果將一些車流負載改至別道肯定會比較好吧...?



多開無益

時間問題解決，然而空間爆炸

多執行緒著實是一個好的解決方案，大型系統運行絕對免不了它的應用

配上一些調度策略的處理，我們能夠將資源給高效率化，使得像是請求之類的系統任務能夠以更好的方式與更快的時間被解決

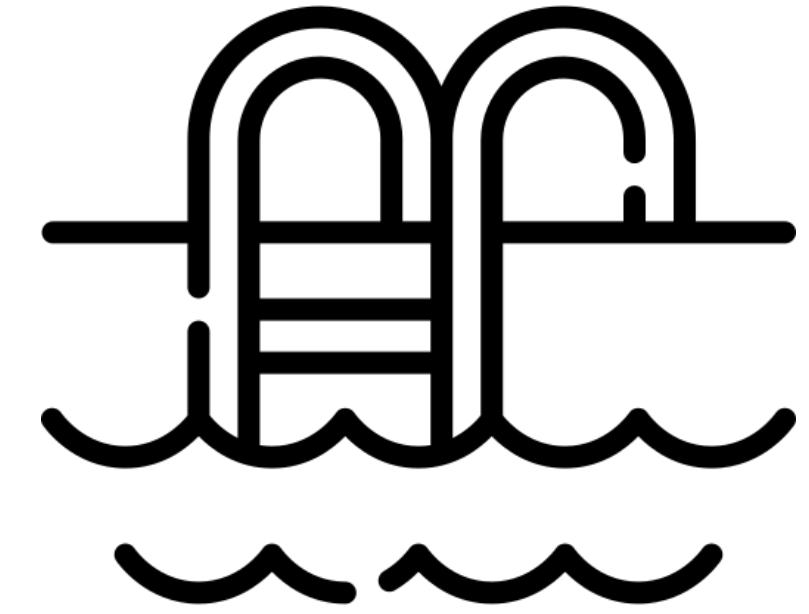
然而，它並不是絕對完美的

多個執行緒就意味著更多的資源配給，管理起來也相對要複雜得多

因此與其一味的多開執行緒，我們能夠給出更好的做法。

線織成的池子

水道創好，等待泳者的加入



與其為了一個任務而開創執行緒，我們倒不如一開始就開好一定數量的執行緒放在一個虛構的池子當中，這樣子的方式稱作...沒錯，執行緒池

透過將任務排程進佇列內，並動態指派給當前閒置的執行緒去運行

藉由此做法，我們不會不經意地消耗太多系統資源，並且可以確保行程的運行不會被擠壓在單一執行緒之上

當然，池子的建立方式，池子的伸縮性等等，甚至是HS/HA、L/F等等模式也可以加以客製化，這裡先行省略。

領域展延 - 更多伺服器

是時候進入負載平衡(Load Balancing)的一環了

直至當前為止，我們都在討論單一伺服器，單一行程的狀況。

但誰想得到應付大量請求最簡單的方法就是買更多電腦來當作伺服器？

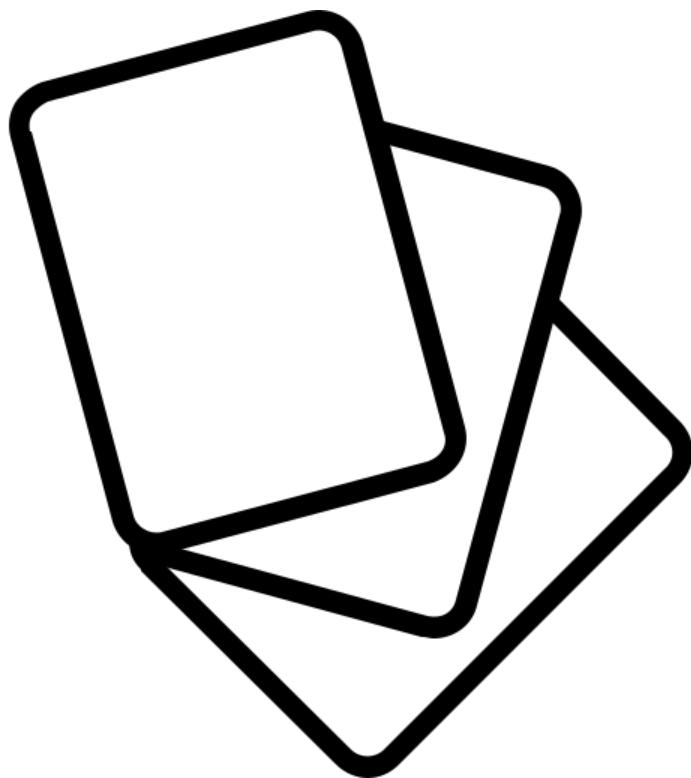
客戶端可以有多個，那麼伺服器端當然也可以的

若是要分配來自客戶端的請求至多個伺服器，我們會需要一個中央調度器，並且透過一些負載平衡的策略緒進行控管

太多名詞了？一個一個來！

發牌

根據運行邏輯來發散請求



需要負載平衡器的情況通常發生在分散式系統當中，也就是我們所謂的多伺服器應用

藉由添加像這樣的中央控制，我們能夠擴展整個系統的控制
在分配的部分，演算法（或稱策略）也有著靜態和動態之分

以下提及兩種常見的負載平衡策略：

- ❖ 輪詢法 (RR)：簡單易懂，但無法根據伺服器性能進行調整。
- ❖ 加權輪詢法 (WRR)：根據伺服器性能分配權重，性能好的伺服器獲得更多請求。

健康檢查

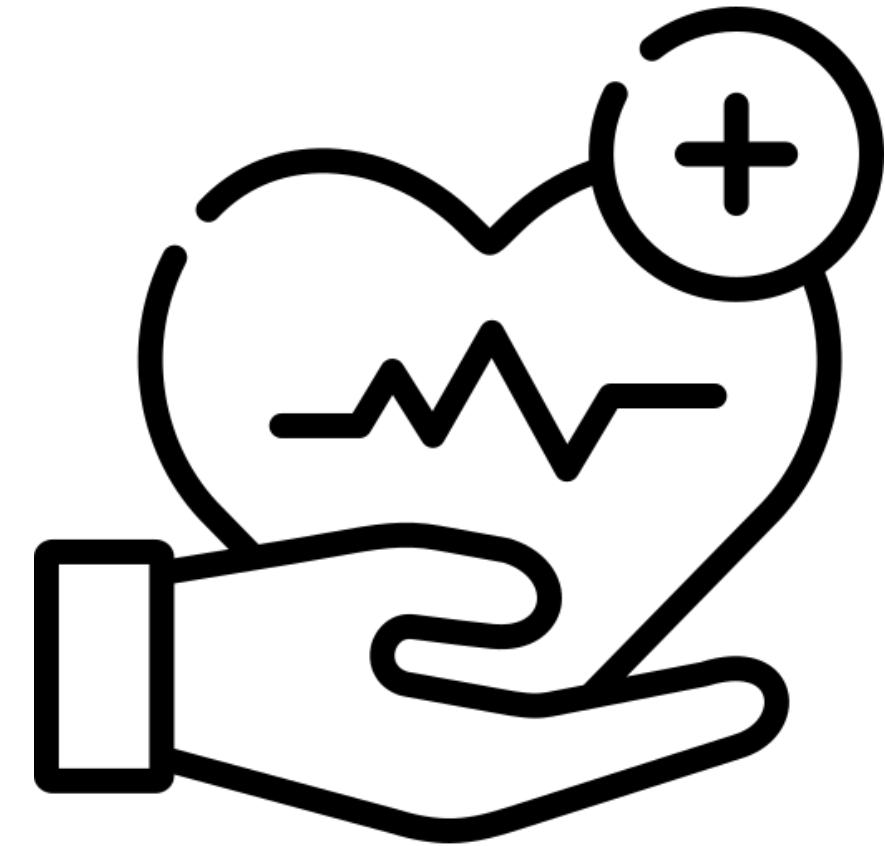
另一個令其不可或缺的理由

除了調度之外，另一項負載平衡器最知名的功能就是其防護、故障排除以及健康檢查的功能：

這麼說吧，當老闆指派工作給你，但你正在瘋狂拉肚子，只要有良知的公司應該都能想辦法讓其他人先接管你的工作，在電腦裡也是一樣，遇到錯誤時 –

負載平衡器會進行以下流程：檢測故障 -> 移除故障伺服器 -> 將請求重新分配。

除了這種重大錯誤場面，它也可以透過檢查端口、TCP連接以及HTTP請求等等的方式來確保伺服器系統保持在正常狀態，真的好厲害喔 (棒讀



累了，休息一下...

Zzz



這一章真的是意想不到的長（汗
我原本還想要講伺服器優化的說...）

久 久



神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

5

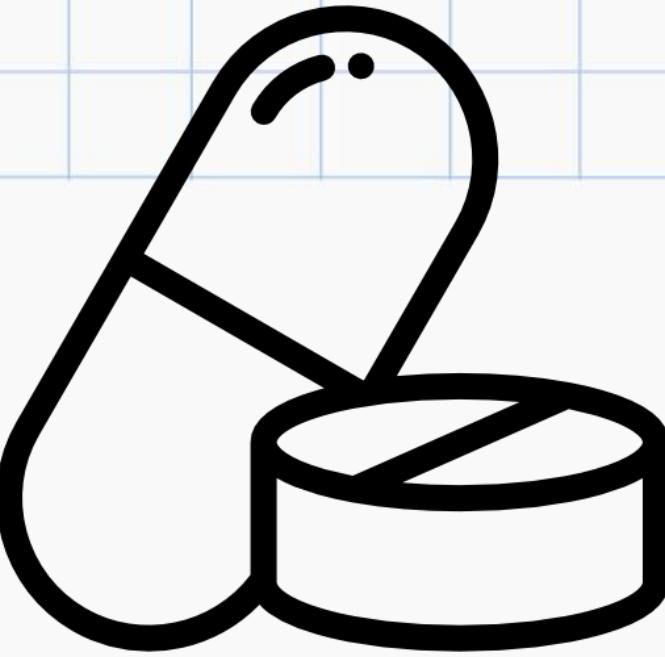
無聲無息

表現狀態與工作階段

```
    org = filterByOrg ? study leadOrganization === filterByOrg : true
    status = filterByStatus ? study.status === filterByStatus : true
    matchStatus) {
      Function filterStudies({ studies, filterByOrg =
        filterByStatus }) {
          return studies.filter(study => filterByOrg ? study leadOrganization === filterByOrg : true
            &amp; filterByStatus ? study.status === filterByStatus : true
            &amp; matchStatus)
        }
      }
    }
  }
}
```



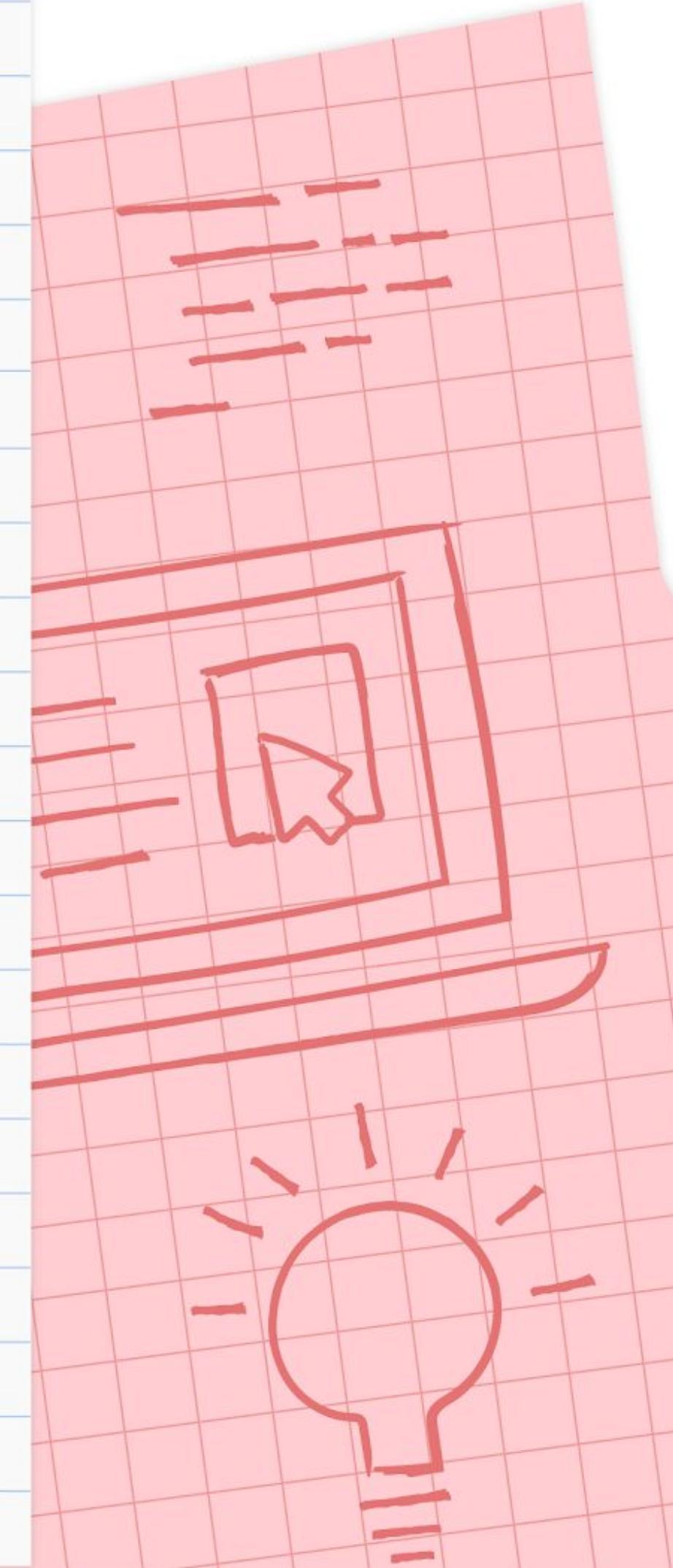
正確的人吃正確的藥



某些伺服器有固定的使用者，可能會需要存取特定個人資料等等，我們會需要去特別區分他們與普通使用者的差異

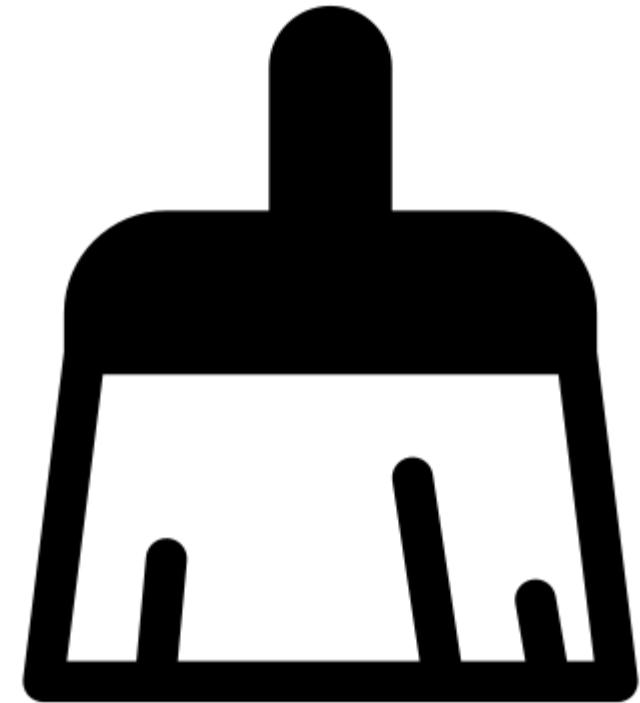
反之，另一些伺服器允許匿名使用者發送請求，在**工作階段**結束之後就會將所有資料清除

這是表現狀態的差異。



預處理，動態規劃技術

快取的特色以及他的重要性



在進入有/無狀態的解說之前，我們必須來講一下快取 (Cache) 技術

每次根據傳入的請求查詢數據時，會需要動到內部資源的運算

就算是常用數據，也需要屢次的向內部查詢，效率太低了

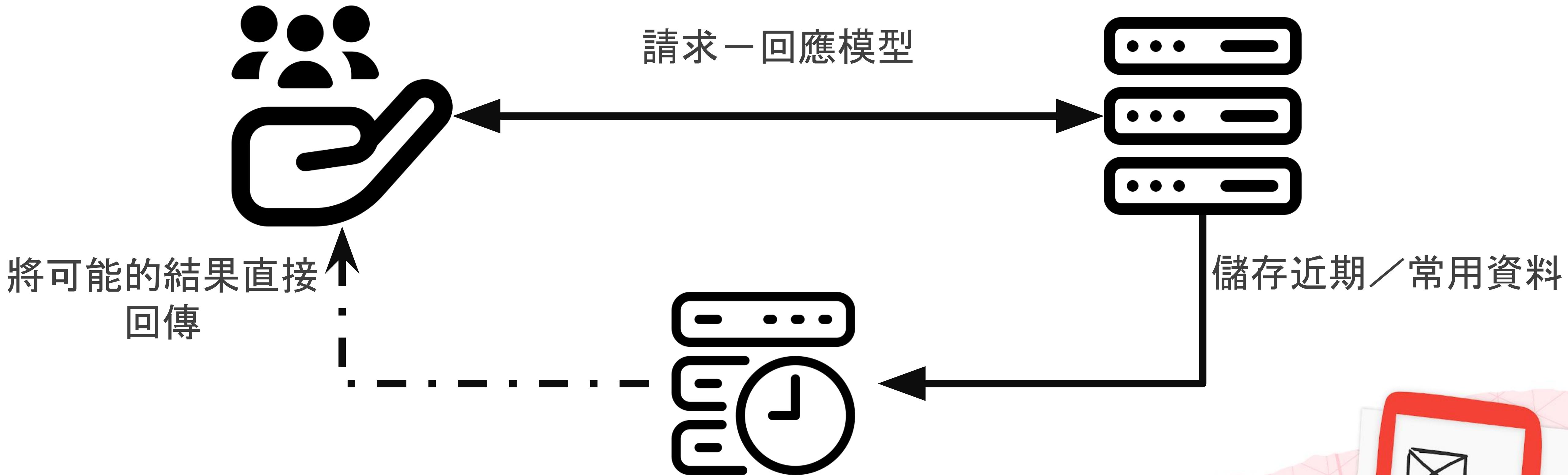
為了提升伺服器的效能，我們可以選擇去將近期或者常見的數據先保存在一個額外的記憶體空間中，以便之後方便查詢

而這，我的朋友，就是所謂的快取的應用



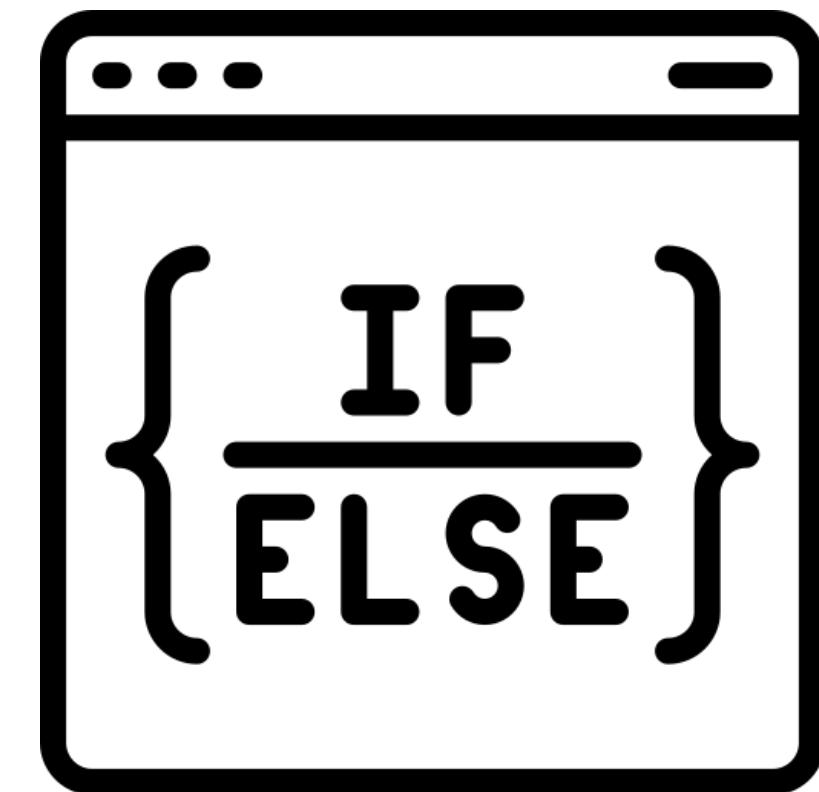
非常淺顯易懂的圖表

這是非常淺顯易懂的圖表



條件判斷式

根據不同時間的狀態而進行回饋



所謂的有/無狀態指的就是伺服器是否會依照特定情況去處理請求，就如導論所提及的一樣，而這些所謂的狀態其實就是類似條件判斷式的構造

比如說在二進制的系統，1跟0分別代表著"On"與"Off"，且兩者互斥

現在想像一個狀況，我如果給你一張紙請你依據現在的狀態寫下答案：

如果狀態是0，請寫下"Yes"，如果是1，請寫下"No"

向這樣子的表現其實就是一個有狀態伺服器的處理模式。

我需要你的身分證

「有狀態」(Stateful)的執行模式

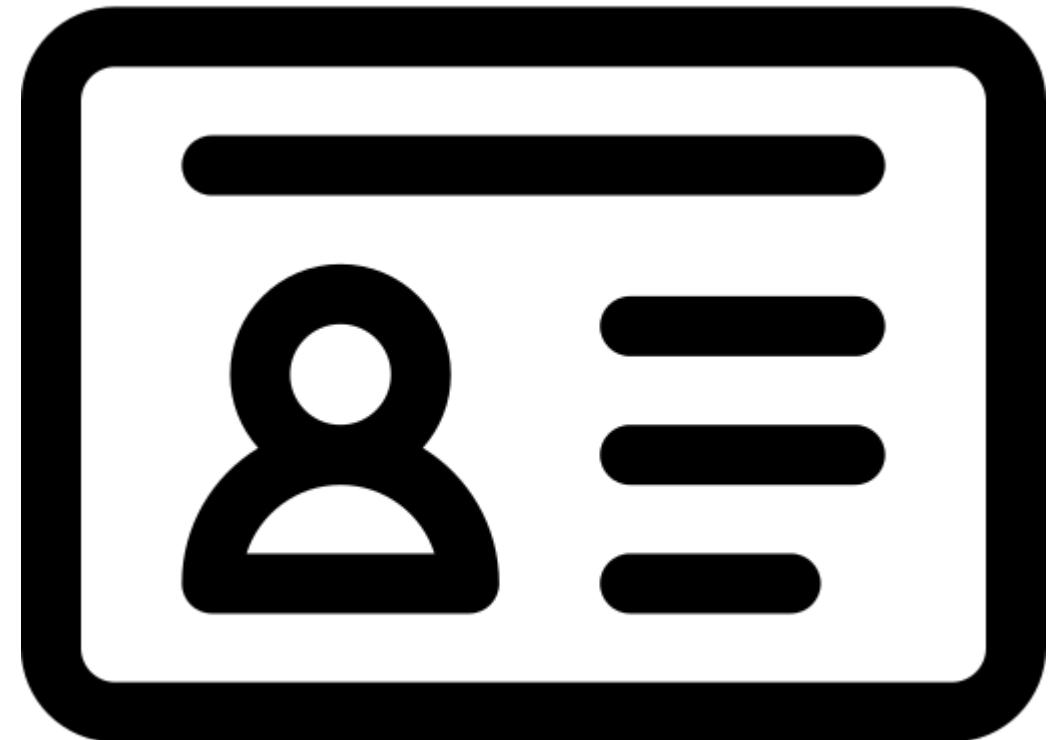
剛剛講了一大堆卻還是很模糊，那麼我們就這樣說吧：

如果網站的伺服器將你的資料存在後台，每次登入時就會利用儲存的資料來驗證你的身分的話，我們可以說這個登入服務就是**有狀態**的。

伺服器內部必須要知道：

- 誰在請求？
- 根據請求者的ID，要秀出什麼頁面給他們看？

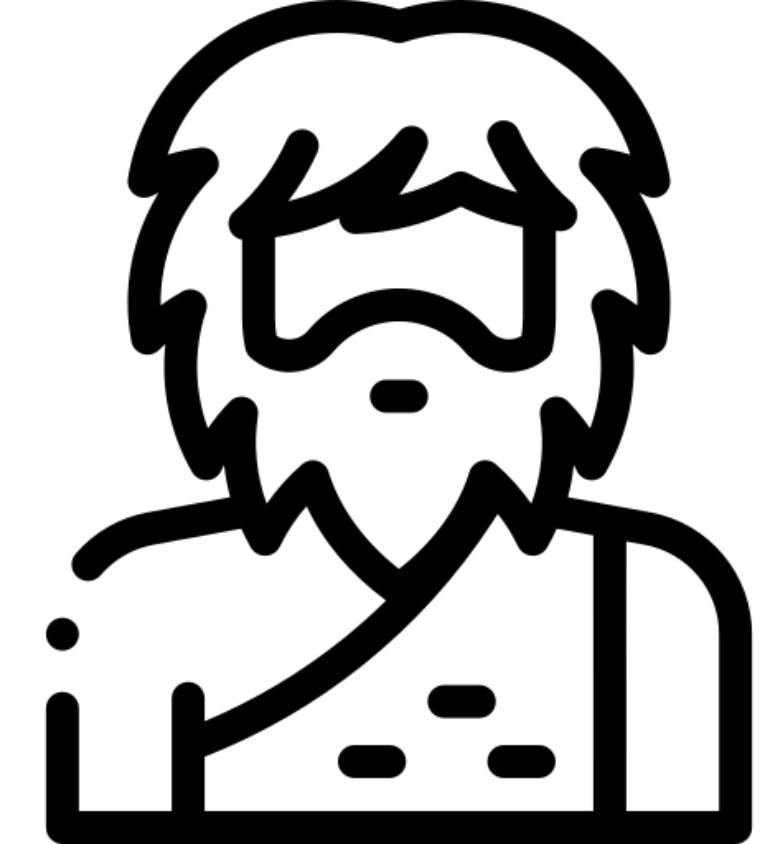
兩者之間會以**工作階段**聯繫，並透過快取或者**Cookie**的方式記錄登入使用者的資訊與偏好



我只是過路人

「無狀態」(Stateless)的執行模式

無狀態的模式，相對而言就簡單多了



我們預設來自任何客戶端的請求都是匿名(Anonymous)的，因此每個請求會彼此獨立，伺服器本身不需要花費額外資源去紀錄，並且可以輕易處理大量請求

信不信由你，當前的網路環境大多都是以這種形式在運行的，你可以去瀏覽各式網站，但只要你不登入或者接受Cookie，他們沒辦法記錄你的資料

當然IP trace以及客戶端主機identity伺服器還是會記錄

一個屬於你我的會議

不容忽視的工作階段作用

有狀態的伺服器直接在內部保存使用者資料，是相當耗費資源且可能會出現高度風險的方式，但無狀態伺服器有只能維持匿名...這該如何是好？

再次回到OSI七層的概念，這次我們向上晉升到了會議層

在這裡，使用工作階段(Session)和數位存根(Cookie)的方式，我們可以另類的「作弊」，迫使無狀態伺服器去「回憶」起以前的資料

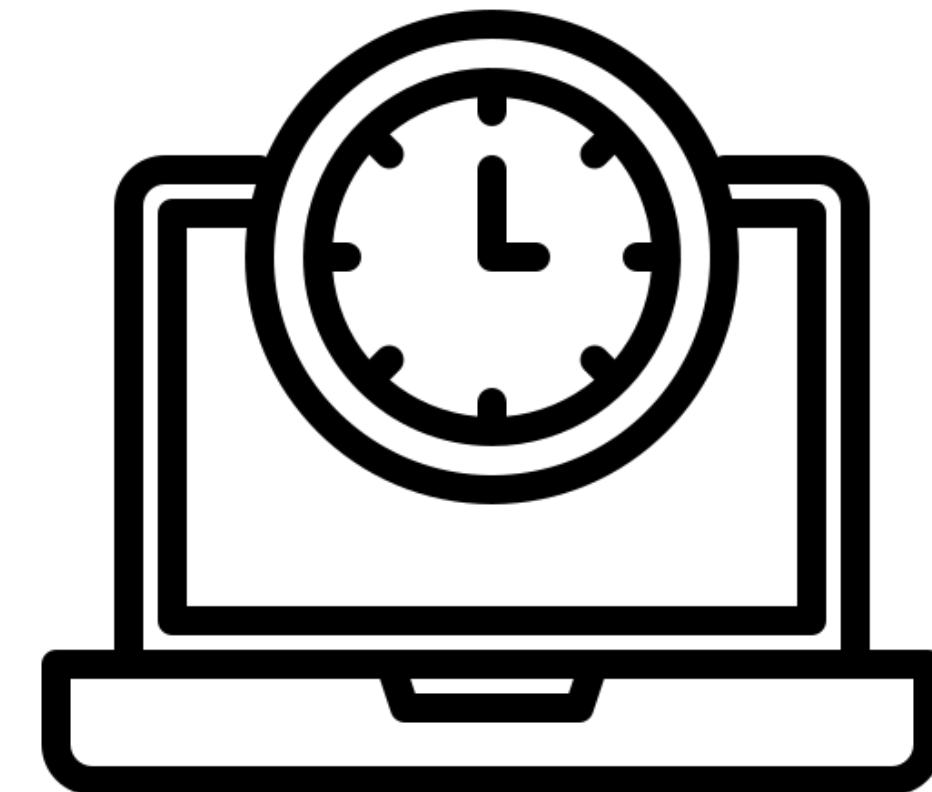
我見過你嗎？

相輔相成的兩個部分

在與客戶端建立連線後，伺服器將一個Session ID發給對方，這個ID表示著當前的工作階段，並將目前的交互資料儲存在後台**資料庫**當中

伺服器端可以透過搜尋Session ID來還原當前的工作階段

但得這麼做的話，這個ID必須要由客戶端的使用者自行提供，但無狀態伺服器會將每一次的請求訪問看作是匿名狀態，這下怎麼做呢？



暫時的餅乾

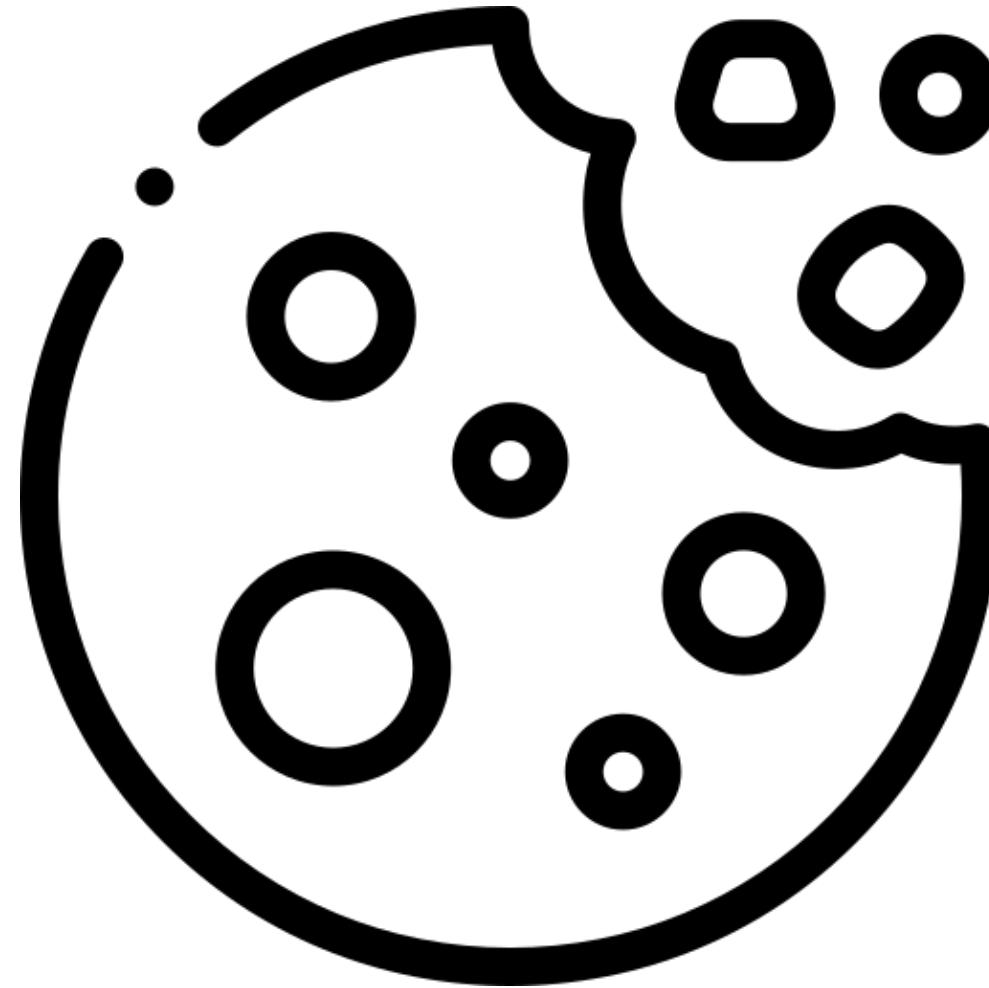
我相信根本沒有人知道這叫做數位存根

Cookie! 但不是可以吃的那種

某些網站在使用者訪問時會詢問是否接受他們的Cookie，其實就是去借用你瀏覽器的內存進行非常簡單的數據儲存

儲存什麼？當然是Session ID的部分啊

這下只要你不清除瀏覽器快取，回訪同一個網站時就能夠再次建立工作階段。





神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

6

無可或缺

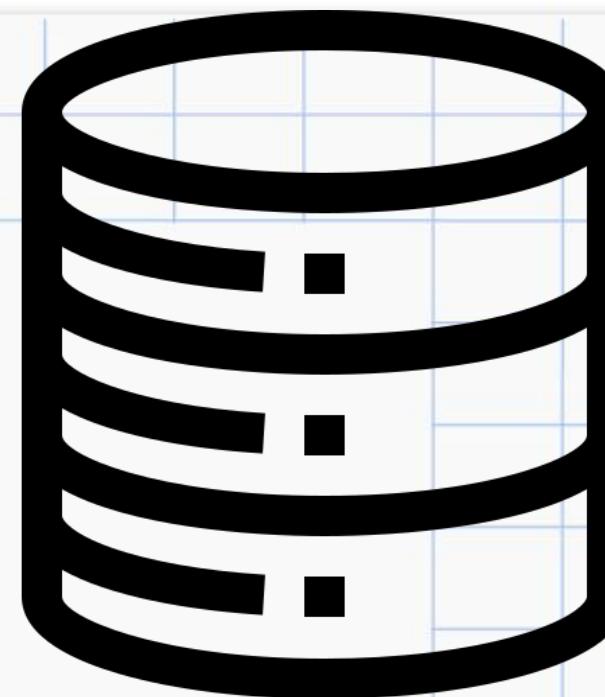
伺服器與資料庫的緊密連結

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const filterMatchStatus = filterMatchStatus ? study.match_status === filterMatchStatus : true;

function filterStudies({ studies, filterByOrg, filterByStatus, filterMatchStatus }) {
  return studies.filter(study => filterByOrg(study) & filterStatus(study) & filterMatchStatus(study));
}
```



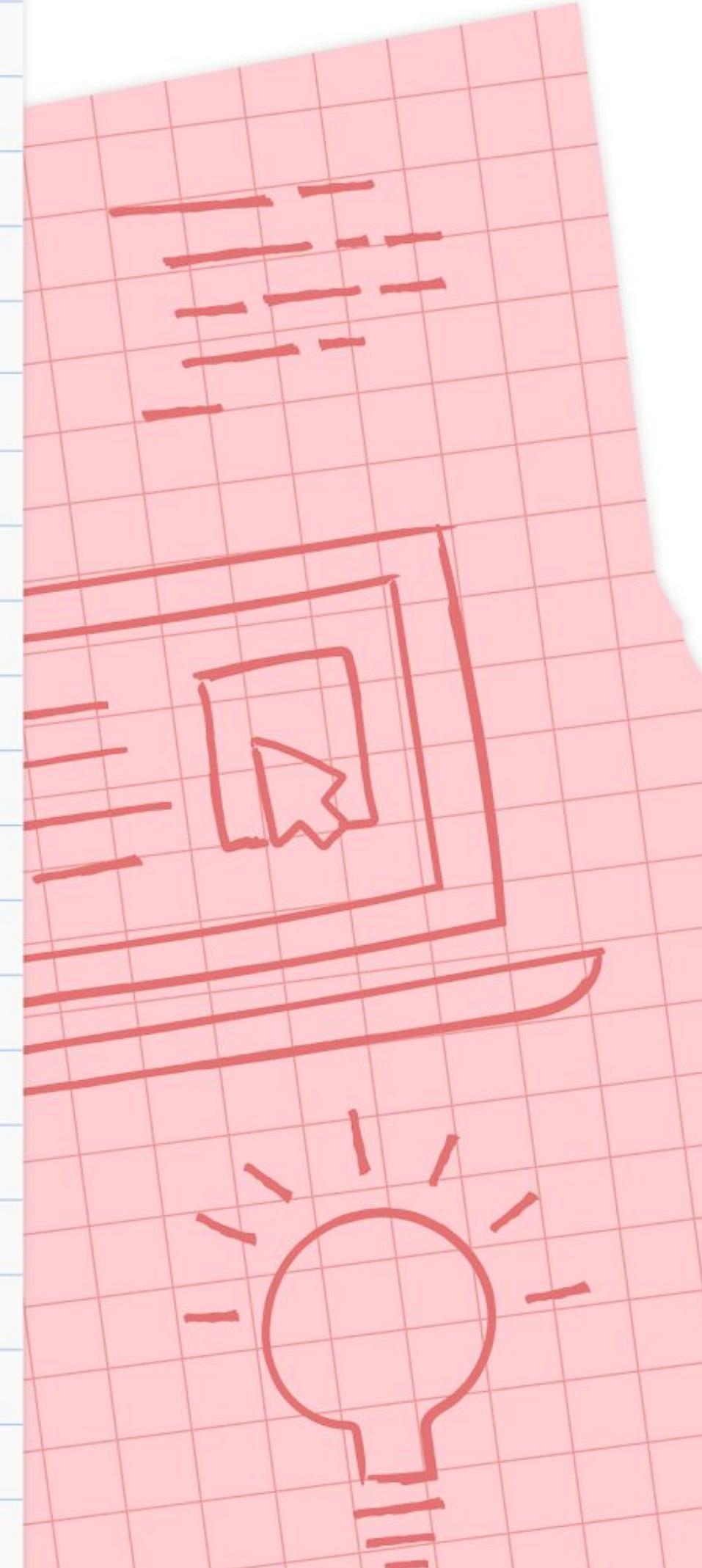
背後的功臣



其實伺服器本身就只是提供系統運算的外部接口罷了，實際上存儲數據的還得是那萬用的資料庫

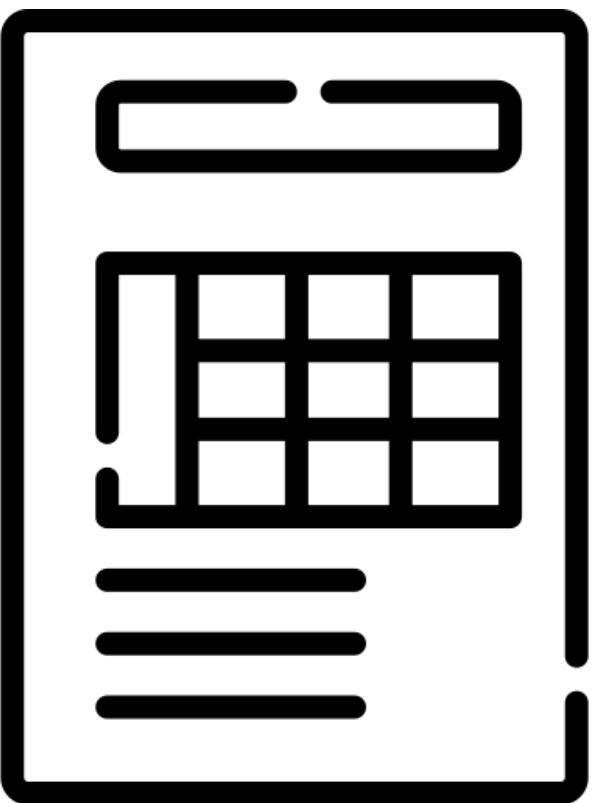
有一定規模的系統往往都需要良好的資料庫維護，畢竟這是最
重要的記憶單元

讓我們來看看伺服器本身是如何作為請求與資料調度的仲介...



非常巨大的電子表格

仍然是最完美的資料儲存格式



我想可能某些人有過接觸資料庫的經驗，這東西可能並沒有大多數想的複雜

儲存數據的方式自然有很多種，但要將資料分為成較好查詢、存取的方法可就多了，因此資料庫就是為了解決這些問題而發明的

除了可以保存大量數據，資料庫還提供了方便的查詢(Query)機制，以及提供CRUD(Create, Read, Update, Delete)操作。

經過正規化、鍵值指定後，原先雜亂的Raw Data就能夠有條有理的存放

相信我，這會是非常重要的操作



常見的資料庫樣式

Database Query Application

其他 : 系統功能 流程控制 汇出資料表 使用說明

Wurm 資料庫檢索系統

應用程式版本: 1.2.0

連線狀態: 已連線

	customerID	customerName	Address	phone	gender	annualIncome
▶	C001	Lily Johnson	123 Maple Street, Apt 101	(555) 123-1234	Female	45000
	C002	Ethan Williams	456 Oak Avenue	(555) 567-5678	Male	60000
	C003	Emma Brown	789 Elm Drive	(555) 901-9012	Female	55000
	C004	Noah Jones	321 Pine Road, Suite 2B	(555) 345-3456	Male	70000
	C005	Olivia Davis	654 Cedar Lane	(555) 789-7890	Female	80000
	C006	Liam Wilson	987 Birch Street	(555) 234-2345	Male	65000
	C007	Ava Martinez	135 Spruce Court	(555) 678-6789	Female	75000
	C008	Sophia Anderson	246 Maple Avenue	(555) 012-0123	Female	50000
	C009	Logan Taylor	579 Oak Lane	(555) 456-4567	Male	85000
	C010	Mia Thomas	864 Elm Street	(555) 890-8901	Female	48000
	C011	Mason Jackson	159 Pine Drive, Apt 3C	(555) 234-2345	Male	72000
	C012	Harper White	753 Cedar Avenue	(555) 678-6789	Female	58000

伺服器位址: 0.tcp.jp.ngrok.io
埠號: 12592
資料庫名稱: 411177034
使用者名稱: 411177034
密碼: *****
連線 斷線
執行範例查詢: 查詢一: 缺陷零件
確認查詢
加入程式列 送出查詢 清除程式列
<16:43:36> 目前佇列中有0筆查詢。
-----分隔線-----

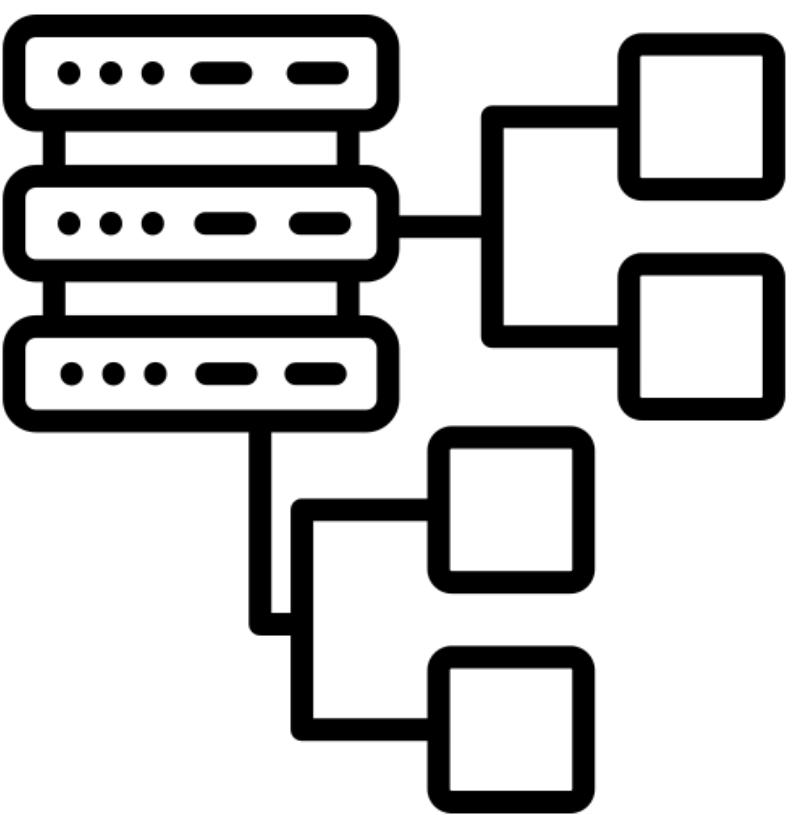
此專案的協作者們:

吳傢濶  頭像
顏榕嶧 (Bernie)  頭像

WurmUI

由Bernie與吳傢
濶開發





麻雀雖小...

就真的跟Excel表格很像嘛

一個資料庫的綱要 (Schema) 是相當重要的，它指示了在其中所需要遵循的格式

我們定義一張資料表 (Table) 以及其中應有的欄位 (Column)，欄位中的其中每一行 (Row) 就屬於是一筆數據 (Data)

在關聯模式資料庫中，我們還會以特別的鍵值去定義一或多個資料屬性

所謂的關聯模式，比起實體關係模式能夠更準確地描述資料，我們先以這種為主

關鍵字，以及外國人

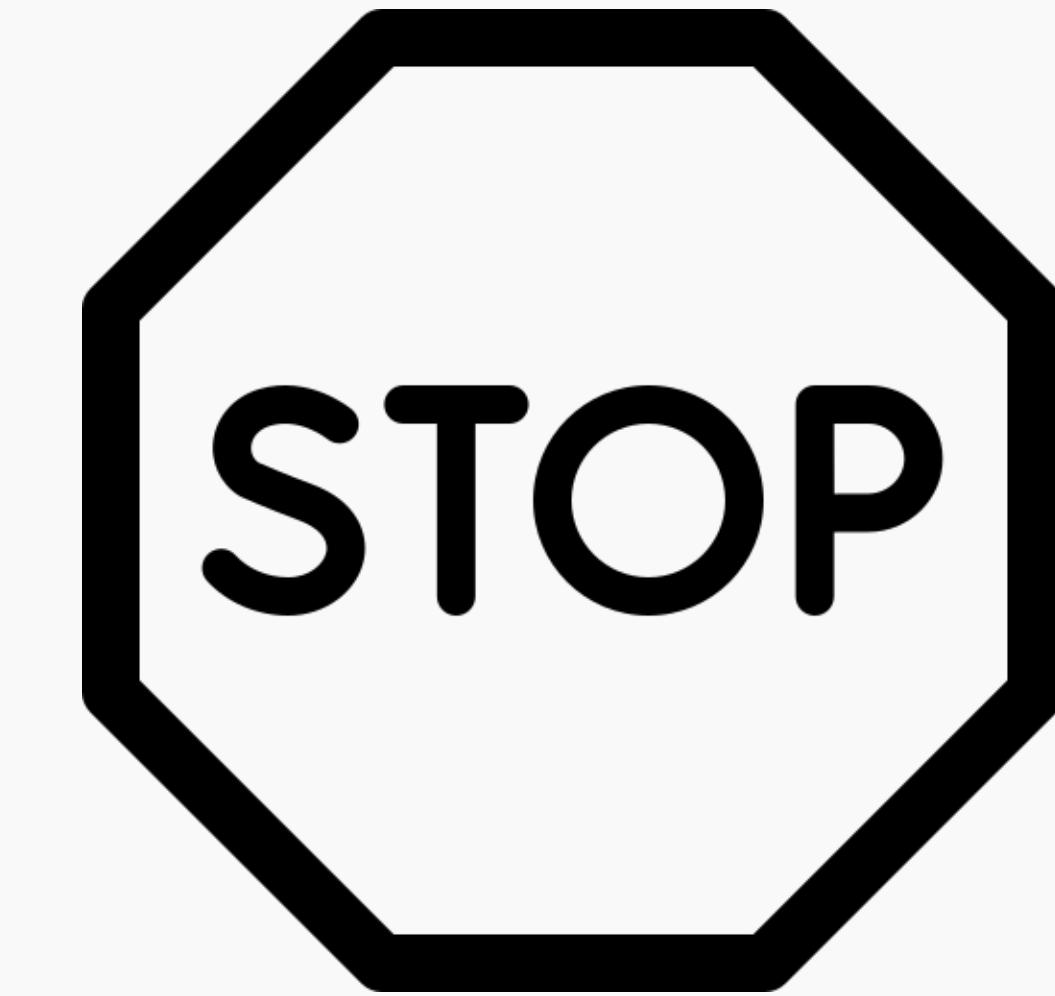
鍵值在資料庫當中的指示用途

關聯模式中四大鍵及其用途如下：

- Super key 超鍵：符合唯一性 (Uniqueness) 的關聯鍵。
- Candidate Key 候選鍵：符合唯一性以及最小性 (Minimality) 的關聯鍵。
- Primary Key 主鍵：從候選鍵中，挑選出其中一個關聯鍵，也就是最具識別意義的關聯鍵。
- Foreign Key 外鍵/外部鍵：關聯中被用來參考到其他表格主鍵的關聯鍵，就是外鍵。

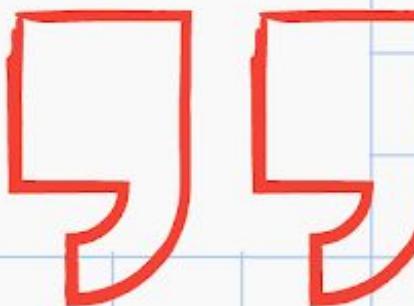


等等，停止！



回到我們的主題上！

我以為我們要教的是伺服器與通訊原理？
話說這是今天第二次看到這張簡報了。



呼叫後台，有聽到嗎？

自動化的查詢操作

所謂的SQL Queries就是外部對資料庫進行存取的操作，或稱查詢語句

我們透過使用一些語法，例如SELECT * FROM "Example Datagrid"等等，可以對相對應的資料表進行操作

而這其實就是伺服器所做的事情，它會根據收到的，來自客戶端的請求去向資料庫進行查詢，並將得到的資料予以回應。頓時變得很簡單了？



一頓操作猛如虎

我們指示在流程上加入一個新的C點而已

flowchart LR

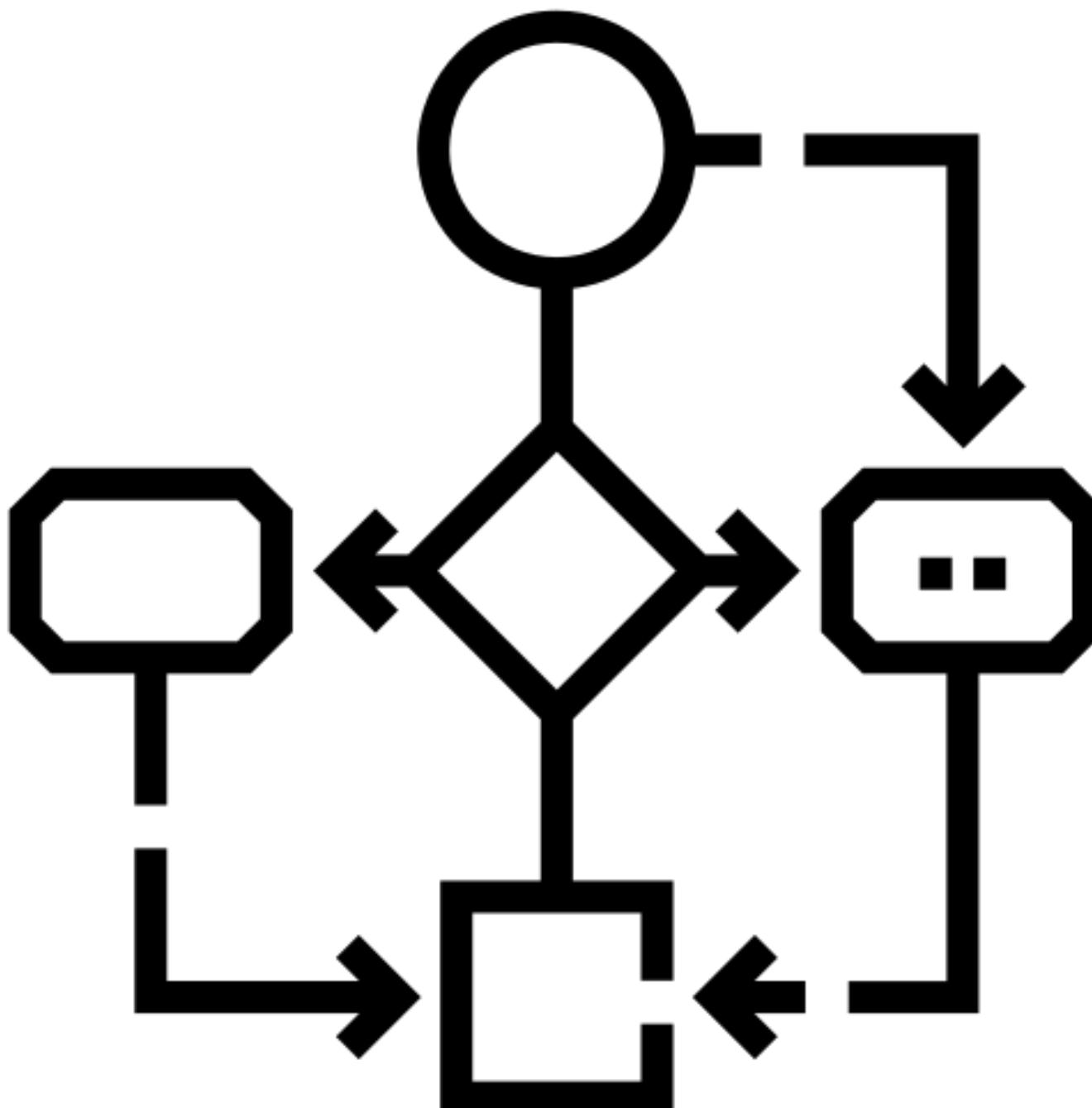
A(用戶) --> B(伺服器)

B --> C(資料庫)

C --> B

B --> A

就只是這樣的概念罷了



核心保證的ACID

酸液...? 喔好像不是

我們只想要正確、完整的資料，不接受任何的瑕疵或者假資料

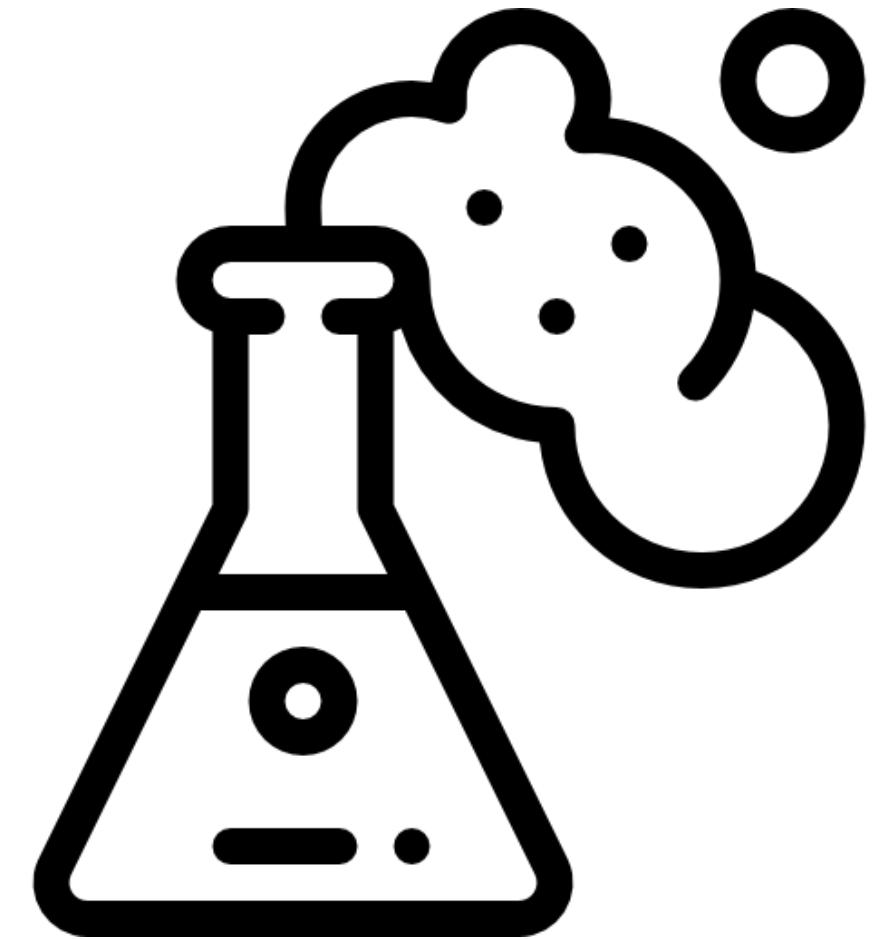
這樣的話，整體的事務交易 (Transaction) 就必須要符合ACID原則：

原子性：事務作為一個整體，要就全部執行，不然就全部不執行。

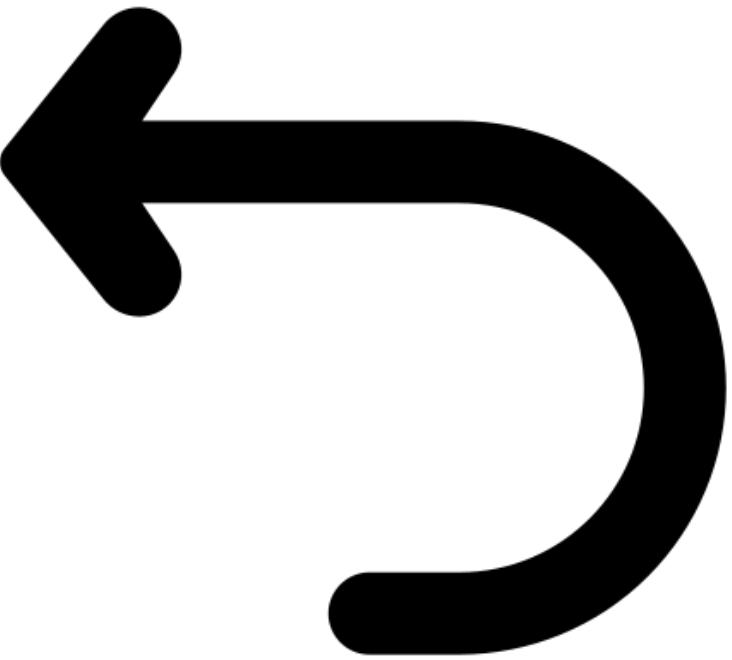
一致性：數據庫的狀態總是有效的，符合所有的完整性約束。

隔離性：多個事務同時執行時，每個事務都好像是在單獨使用資料庫，互不干擾。

持久性：一旦事務提交，其對數據庫的改變就是永久的，即使發生系統故障。



滾回家



相對而言最沒有殺傷力的排錯方法

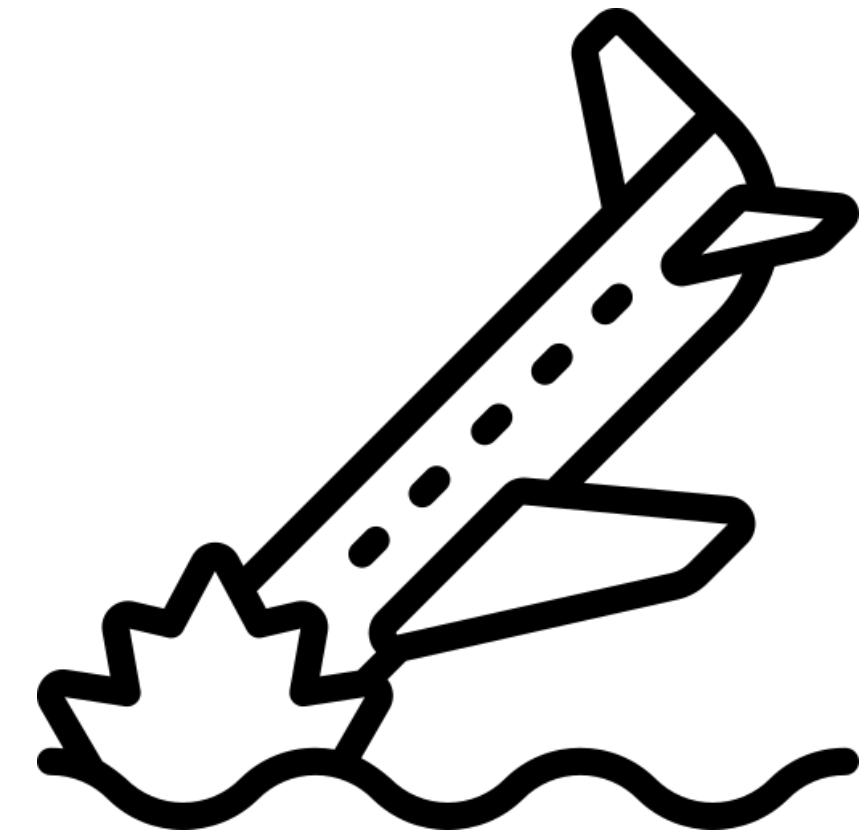
如果今天有一或多個事務交易並沒有遵守ACID原則，資料庫會進行回滾 (Rollback)，將事物給回溯到還未進行交易的時間點

這點是保障客戶端不會因為系統疏失而得到錯誤的資料

另一個應該需要注意的情況，是多個執行緒要存取同一個資料庫時，為了預防數據不一致的問題 (例：寫入跟讀取同時進行)，會有所謂的狀態鎖 (Lock) 出現

快取崩塌

稍微帶回剛才的話題



現在我們知道伺服器快取的運作實質上是在預處理資料庫中可能的常用資料，但這樣子的做法難道就沒有風險嗎？

有的，而且有些不容小覷...

例如快取失效（時限到期）、快取穿透（資料不存在）、快取雪崩（大量數據同時失效）等等，我們在之後的章節會再次提及。



神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

7

無疾而終

網路延遲與傳輸效率

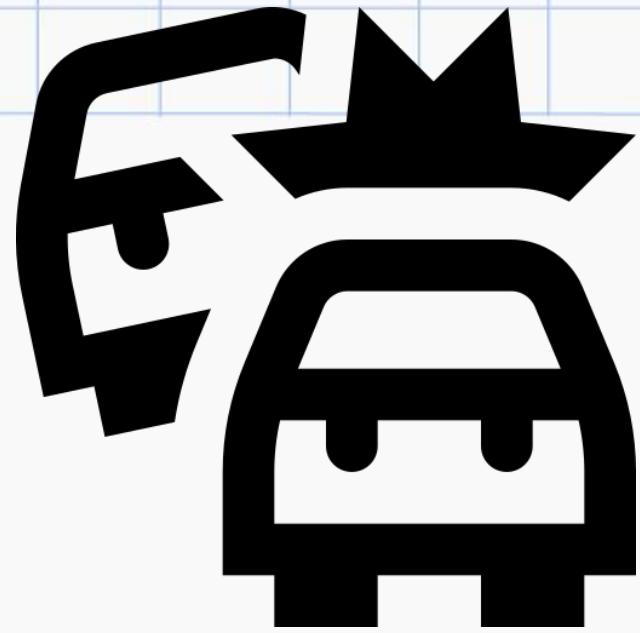
```
filterByOrg = filterByOrg ? study.lead_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
const matchStatus) {
  return studies.filter(study =>
    filterByOrg && filterStatus && study.lead_organization === filterByOrg && study.status === filterByStatus
  )
}

function filterStudies({ studies, filterByOrg, filterByStatus }) {
  const filterByOrg = filterByOrg ? study.lead_organization === filterByOrg : true
  const filterStatus = filterByStatus ? study.status === filterByStatus : true
  const matchStatus) {
  return studies.filter(study =>
    filterByOrg && filterStatus && study.lead_organization === filterByOrg && study.status === filterByStatus
  )
}
```





虛擬塞車



進入到連線與網路的部分，我們就得來提及所謂的延遲。

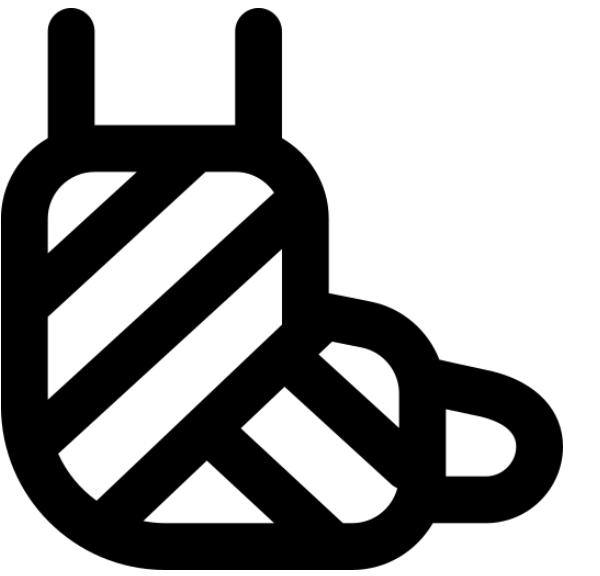
客戶端傳送請求，伺服器端回應是理想狀況，但可能會因為各種各樣的原因而被中止或者拖沓。

在這其中，我們必須先了解所謂的TTFB和RTT，以及如何去監控這些相關數值。



LAG

沒有，這次不是什麼縮寫



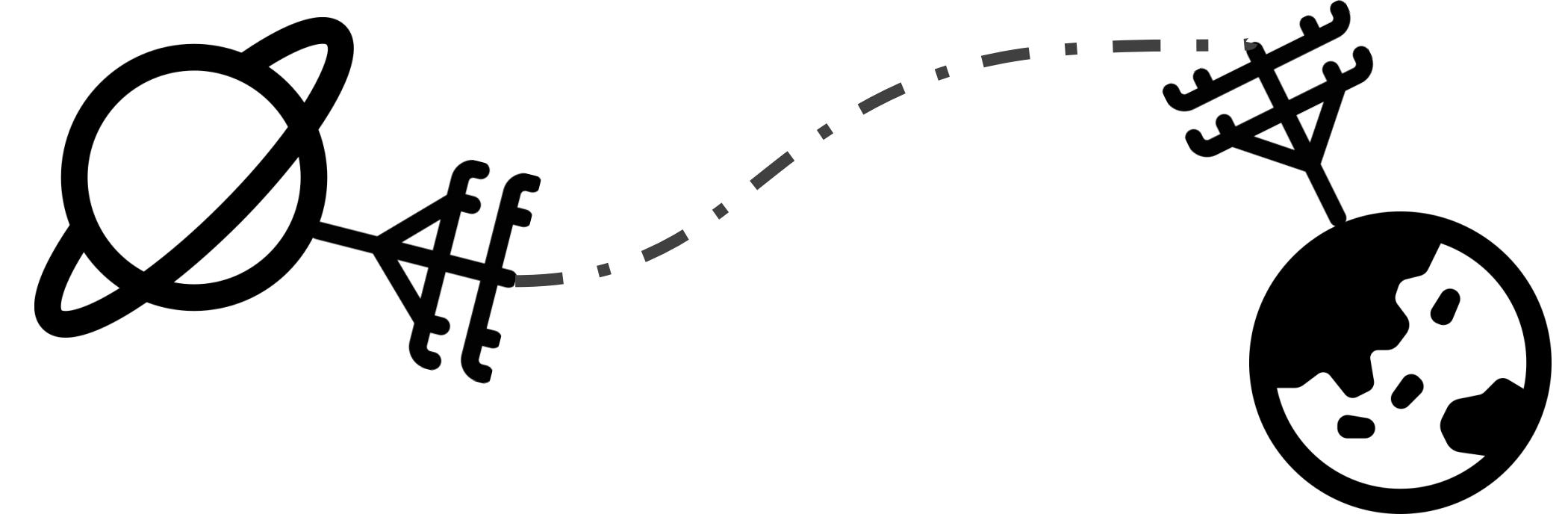
回憶一下經常出現網路延遲的時機：玩多人遊戲、上網搶周杰倫的票、中華電信在搞、要連到美國或者歐洲的伺服器等等。

俗話說知己知彼百戰百勝，我們知道延遲會使人暴躁，那又是甚麼導致了延遲呢？



也許是距離太遙遠？

物理距離也會影響虛擬通勤時間



最常見的因素就是地理位置，這應該也不難理解。

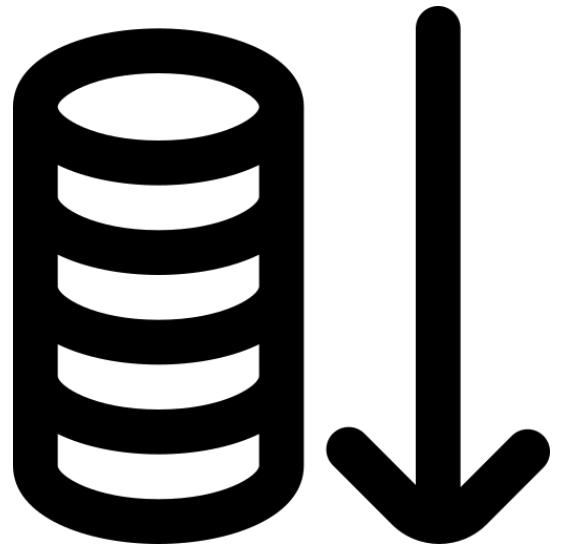
原始的位元資料透過光纖傳輸 (OSI 第一層)，經過海底電纜等等形式去傳遞到下一個節點，你總不能說這些實質的物件不存在。

另一種常見的因素是路由器的跳數，也就是網路資料經過多個路由器進行導向，導致每一個端點當中都會附加一些延遲。

也許是資源太侷限？

一條路有它的容量寬限

我們換一種角度去思考。



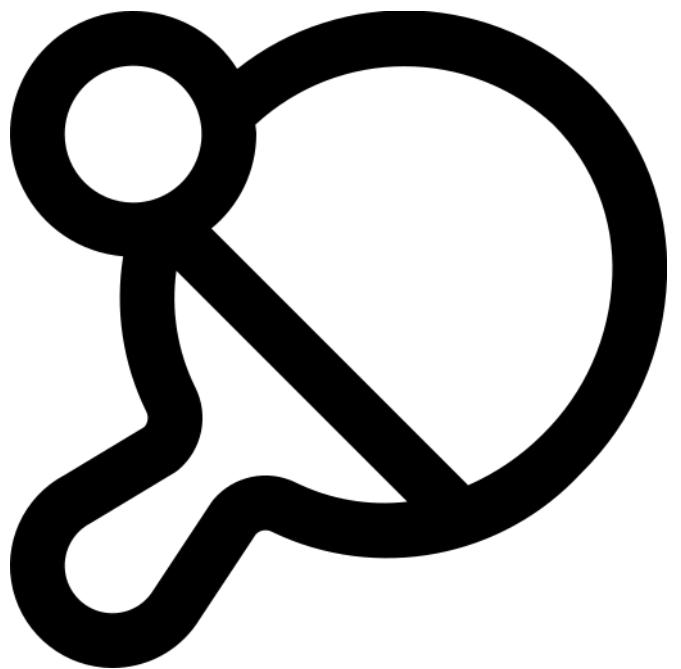
網路中的資源不是無限的，儘管有負載平衡器會幫助維持通道流暢，卻也時不時會遇到封包需要排隊，等待輪到自己的回合的情況。

另一個很像的情況是太多請求同時湧入到伺服器中，主機端沒辦法快速處理大量的請求，此時就會出現怠速運作，這種狀況超麻煩的。



實質的定義又是為何？

要找出問題解答最好的方法就是把它量化



Ping, 大家或許有聽過，如果沒有的話等等也會提到。

網路的延遲我們通常都是用毫秒 (ms) 來表示，但它代表的意義究竟是什麼呢？

這麼說吧，主要就是兩種，**第一元組時間 (TTFB)** 以及**往返時間 (RTT)**。

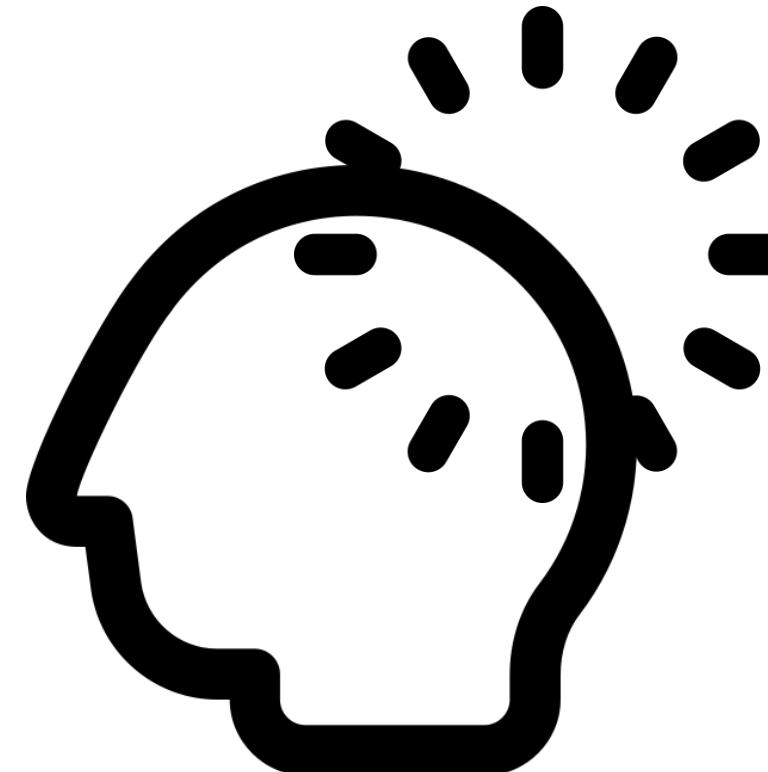
要怎麼去監控以及管理當前所用網路的佔用率，是一種非常值得學習的技能。



檢討加害人

第一元組時間的奧妙

TTFB, 聽上去很帥, 實質又是怎麼樣的?



簡單的說，他是指客戶端發出請求後，伺服器的第一個回應傳回用戶端所需時間...但要講詳細一點的話，他取決於兩個因素：

- 伺服器處理請求並建立回應所花費的時間
- 回應傳回到客戶端所需的时间

你可以說用這種方式去測量可以知道伺服器處理時間跟網路延遲。

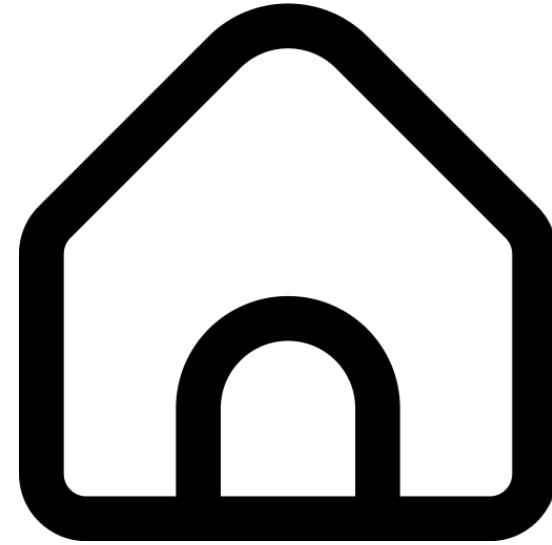
現實に歸り

這樣說吧，我們沒必要那麼專業

更常用，也更容易用的方式是找RTT，也就是往返時間的部分。

優點是他顯而易見而且又快速，Windows CMD甚至內建Ping與Traceroute指令來幫助查尋RTT（這也是等等會提到）。

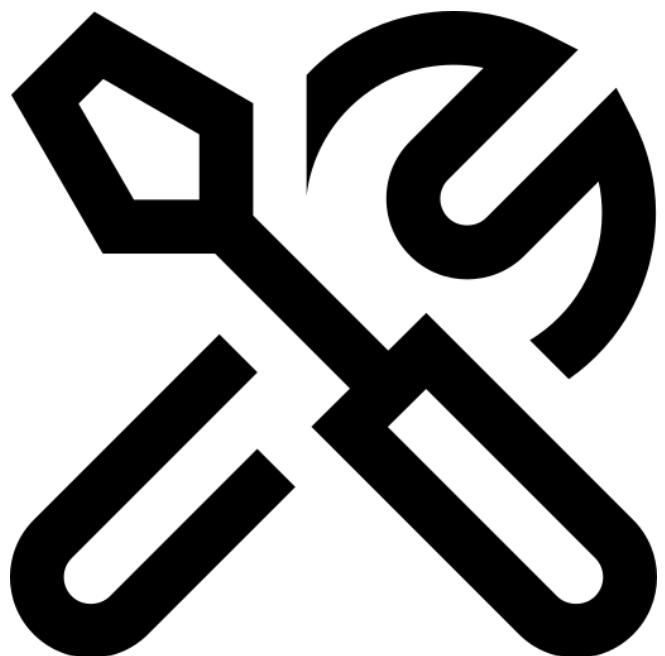
但要注意的是RTT其實只能算是部分指標，因為資料從用戶端傳輸到伺服器並返回時，可以經過不同的網路路徑。



How to 解決

我們不是原始人，我們有現代科技阿

了解了各式延遲的發生原因，我們來一一想辦法解決...



1. 地理距離引起的延遲 -> **CDN**  我們主要講這兩個就好
2. 路由器跳數 -> **BGP** 
3. 封包阻塞 -> 多工、壓縮與合併
4. 伺服器回應速度太慢 -> 分層快取

除此之外，我們還應該定期監控伺服器的回應時間



分，都分

希望你們還記得分散式系統

使用內容分發網路 (CDN)，也就是現在各種服務中最常見的技術。

只要把資料複製到世界各地的節點，就不會有距離太遠的問題了，就是基於如此簡單的概念所誕生的技術，讓大型的服務公司可以保持相當穩定的水準。

這其實也是基於分散式系統上的一個應用，好處是不需要全都依賴主機伺服器去調動所有必備資料。例如：Youtube



網路(的)郵局

調度策略，不過是動態的運作

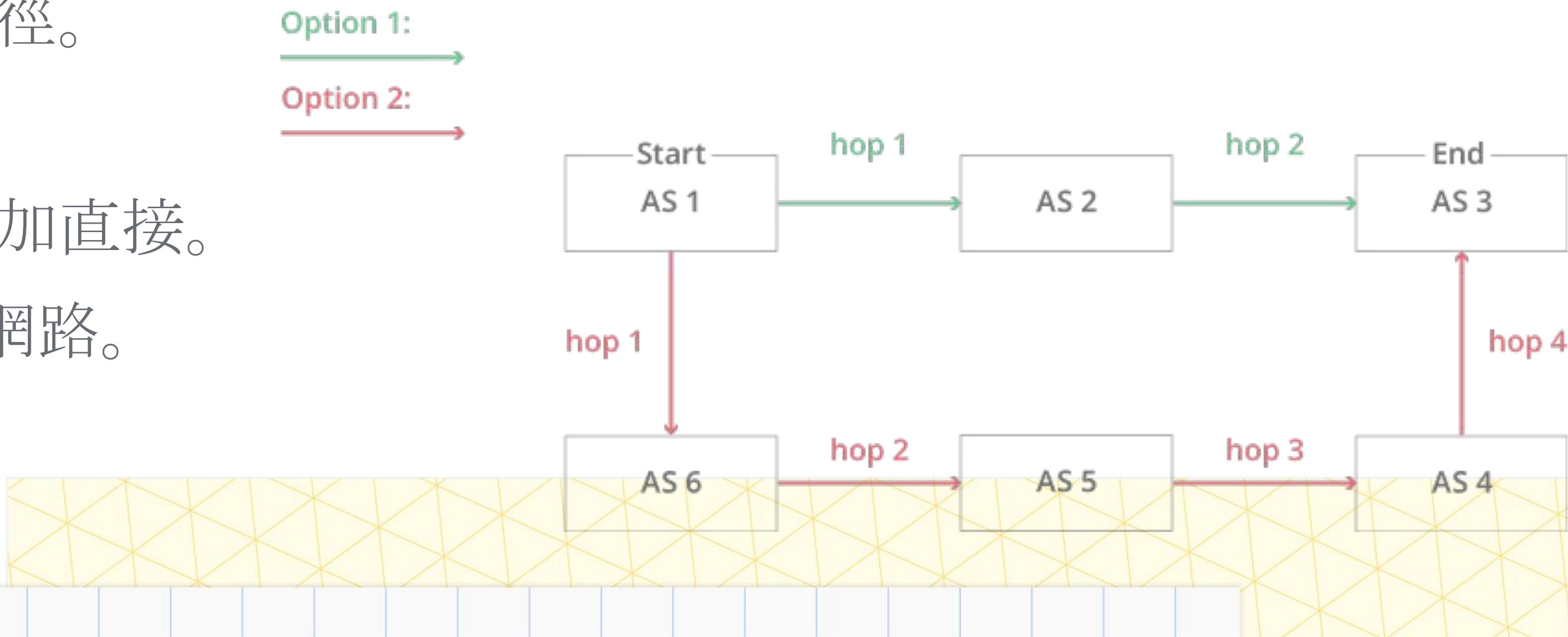


邊界閘道通訊協定 (BGP) 是一個非常厲害的網路技術。

它透過唯一的自治系統編號 (ASN) 來識別不同自治系統之間的關係，並計算出跨
越多個自治系統之間的最佳路徑。

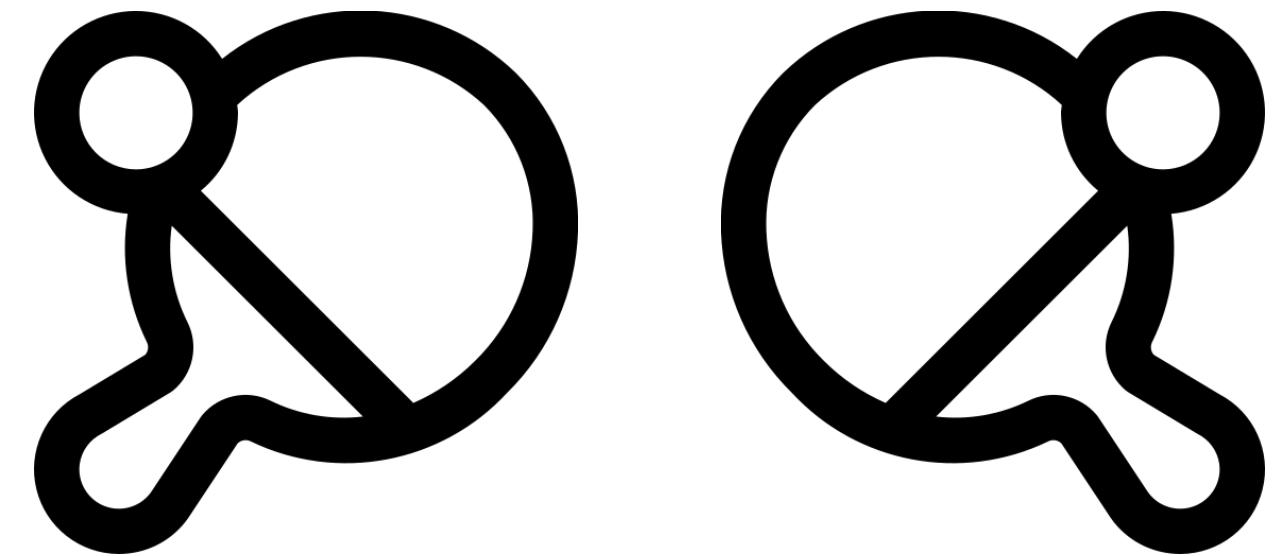
這麼做能夠讓路由器的導向更加直接。

但是當然，這依賴於自治系統網路。



乒乓

調侃了那麼久，我們來說一下Ping



跟大多數人想的不一樣，Ping這個詞本身並不代表延遲，而是一種用來測量延遲的技術，使用的是ICMP協議（對，不要問為什麼縮寫那麼多，這是資訊領域）。

他所做的事是發送一個回聲封包，然後測量回收到這個回聲的伺服器回應時間。

由於他的實作方法簡單，因此很多系統都會內建Ping的技術（就像某些遊戲，或者一些需要聯網的應用程式等等）



神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

8

無遠弗屆

應用部署 - 雲端或本地

```
filterByOrg = filterByOrg ? study.lead_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
filterMatchStatus) {
    return filterMatchStatus;
}

function filterStudies({ studies, filterByOrg,
    filterByStatus }) {
    return studies.filter(study => {
        if (filterByOrg && filterByStatus) {
            return study.lead_organization === filterByOrg && study.status === filterByStatus;
        } else if (filterByOrg) {
            return study.lead_organization === filterByOrg;
        } else if (filterByStatus) {
            return study.status === filterByStatus;
        }
        return true;
    });
}
```



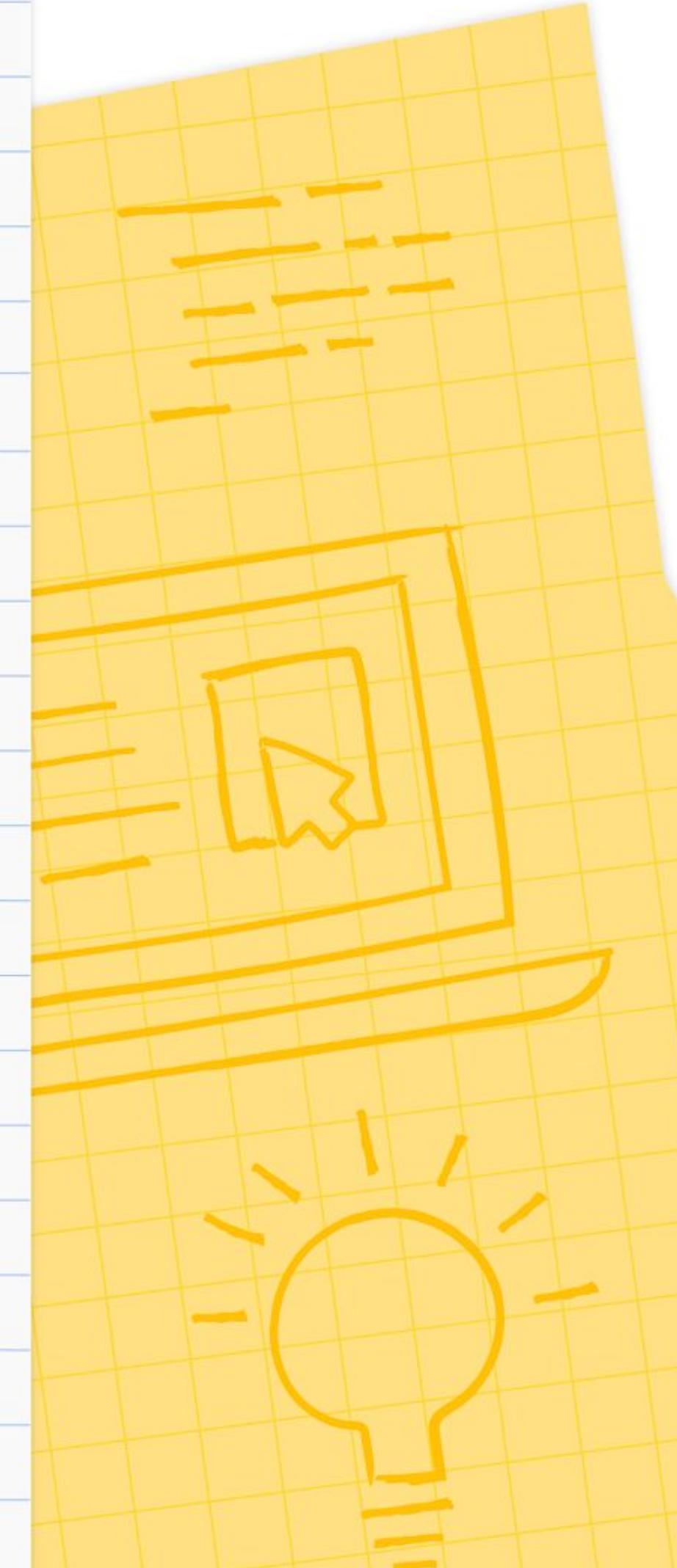


好了，啟動

我們有了裝載服務的伺服器，那麼該怎麼去讓別人使用呢？

沒錯，就單純只要打開門就好，經過了部屬程序之後，任何擁有連線方式的客戶端就可以很簡單的與你連線。

至於部屬的方法是如何，我們就等著看看吧。



搞清楚

啟動跟部屬到底有甚麼差別？

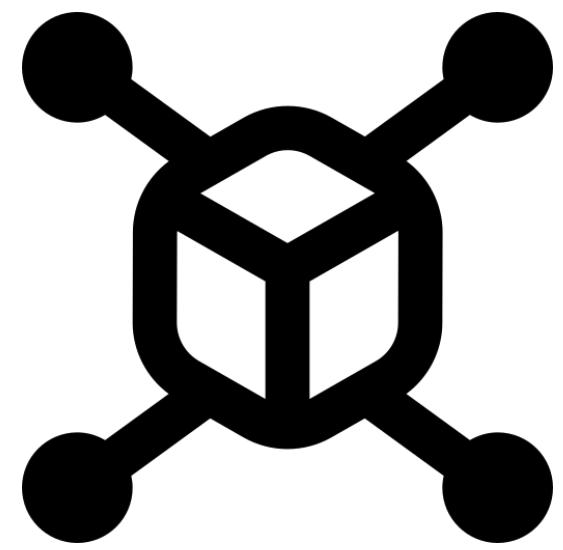
如果說伺服器部屬的關鍵概念就是讓別人可以去使用 你伺服器所提供的服務，那跟單純的啟動程式有什麼差異？

重點在可用性的部分，我們再一次的以Minecraft伺服器來舉例：

在本地電腦上啟動 Minecraft 伺服器可能很簡單，但：

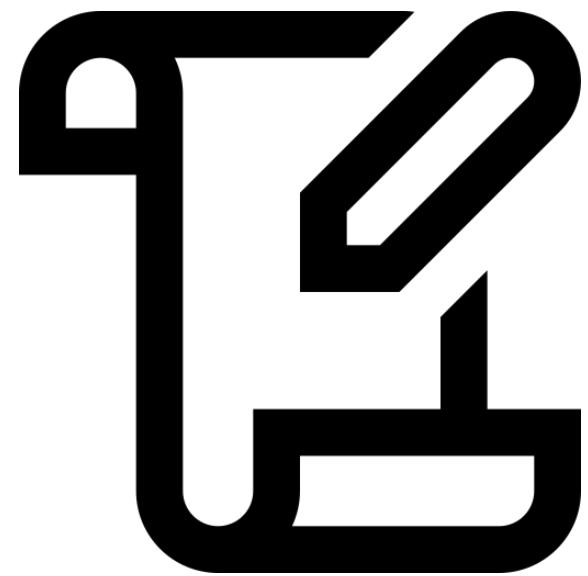
- 若朋友需要連線，你得配置伺服器IP和埠號。
- 若人數增加，你需要調整伺服器的資源分配（如記憶體）。
- 若全世界玩家都能連線，可能需要配置域名或者進行雲端託管（如 play.myserver.com）。

大致了解所謂的可用性是什麼了嗎？



關鍵要素

正確的部屬需要符合甚麼條件？



講得有些模糊，是時候來提及一些達成部屬需要遵守的一些規則：

- **持續運行**
 - 確保伺服器能 24/7 穩定運作，例如使用系統服務或啟動腳本來管理伺服器程式。
- **資料保存與備份**
 - 定期備份伺服器資料(如地圖與玩家進度)以防止損壞或丟失。
- **安全性**
 - 設定防火牆，阻止未授權的訪問。
 - 定期更新伺服器程式，修復漏洞。



看看周遭

為什麼系統環境配置這麼重要？

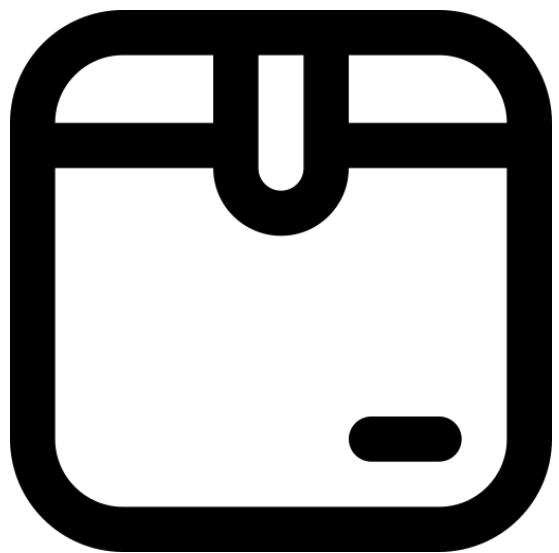
你能在一台沒裝Python的電腦上執行Python應用程式嗎？

或者說，你能夠在Window電腦上用Mac的功能嗎？(不考慮虛擬機)

伺服器也相同，配置環境是至關重要的，尤其是在大量依賴外部資料集的情況下，你可不會想要系統裡面的指向路徑錯亂，因而導致服務整體癱瘓。

我們花一點時間講一下虛擬環境和映像檔的概念好了。





裝起來，隔離起來 Pt.1

虛擬的環境其實就是最好的隔離

物理爆破的方式大家都清楚，直接在系統上安裝程式的依賴資料集來讓程式可以跑得動，簡單暴力，而且可行。

然後這種作法在專案多起來之後（或者伺服器需要品質控管的時候）就顯得有些不安全了，畢竟你在某個專案中用的某個package版本可能在其他專案中不同。

如果一開始就將專案/伺服器的執行環境設定在一個虛擬環境當中，就能夠有效的避免這種情況。

裝起來，隔離起來 Pt.2

依賴打包，輕鬆自在

如果我跟你說有方法可以將整個server應用程式給包裝起來，給他拍個照，然後用你從來沒想過的方式去快速部屬呢？

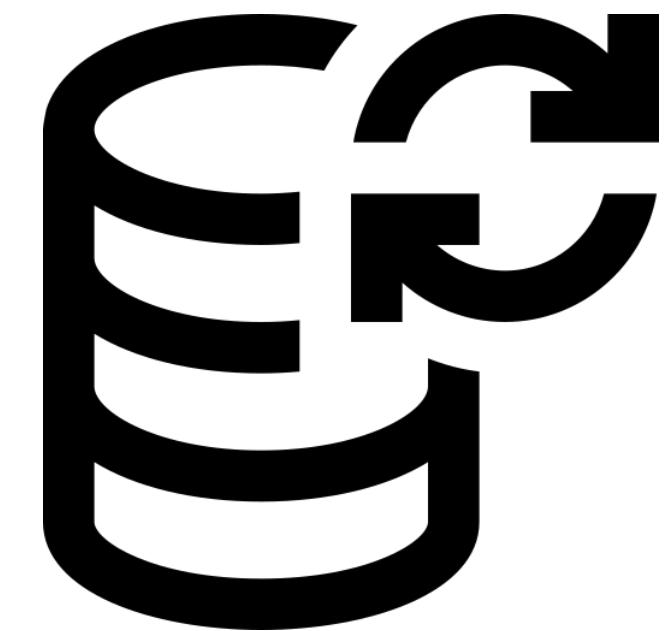
這就是image檔案跟container的應用，我們通常會用Docker來做這些事。

比起虛擬環境還要更快，更輕便（隔離性次之就是了），能夠在幾秒鐘的時間內就架設好一個微服務，多好用？（Docker Image 的應用可能會在下學期講到）



考量伺服器的位置

在本地端跑嗎？還是雲端化呢？



這可能是最需要考慮的點，可以看看下列的比較：

- **本地部署 (On-premises) :**

- 將伺服器設置在自己管理的硬體設備上，適合需要完全控制的企業。
- 優點：高控制權、無需依賴外部服務。
- 缺點：成本高（硬體、運維）、擴展性有限。

- **雲端部署 (Cloud Deployment) :**

- 使用雲服務提供商（如 AWS、Google Cloud、Azure）來托管伺服器。
- 優點：彈性擴展、低初始成本、快速部署。
- 缺點：需要依賴外部基礎設施，長期運行可能有成本考量。



兩者的常見範例

想一下，本地跟雲端常用在哪些情況

本地部屬：高師大單登、ZeroJudge (對相信我，我知道他的主機在哪)

緣由：本地端的部屬可以讓局域內的用戶更容易地去使用服務，並且可以限制用戶來源，以防止外部的風險用戶進入並嘗試攻擊。

雲端部屬：Facebook、Youtube

緣由：為了提供全球性的共通服務，雲端化可以最小化資源運用



總而言之，部屬的流程：

總結以上的部分，我們來看看一個完整的流程

選擇架設位置：本地 / 雲端



處理應用程式嵌入：可以用工作區或者是伺服器映像檔



配置伺服器參數：如IP、埠號、負載平衡等等



監控伺服器流量：可以使用第三方監控軟體





神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

9

無懈可擊

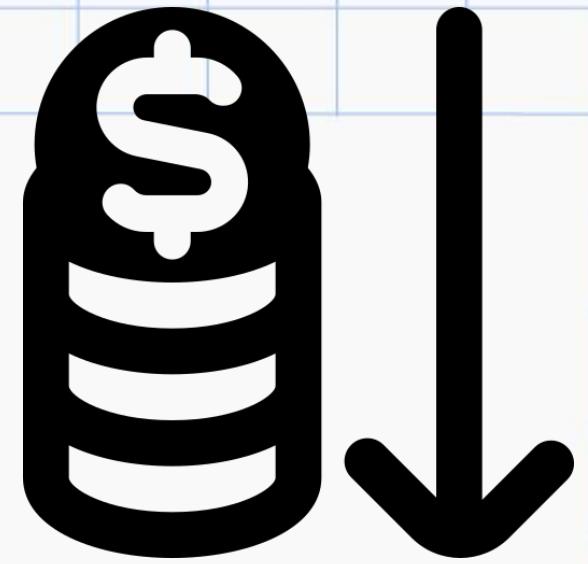
資安顧慮

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterByStatus = filterByStatus ? study.status === filterByStatus : true;
const filterStudies = studies.filter(study => filterByOrg(study) && filterByStatus(study));
```





不怕一萬，只怕萬一



到目前為止已經把所有重要的伺服器架設知識都理解的差不多了，你可能會這麼想...

錯了！我們還沒進入最重要的一個部分！也就是所謂的安全控管！

沒錯，雖然稱不上是跟淮恩的課的史詩級連動，但是我們來認識一下架設伺服器可能會遇到的資安危機！



攻擊的價值

別人哪有那麼閒去攻擊你

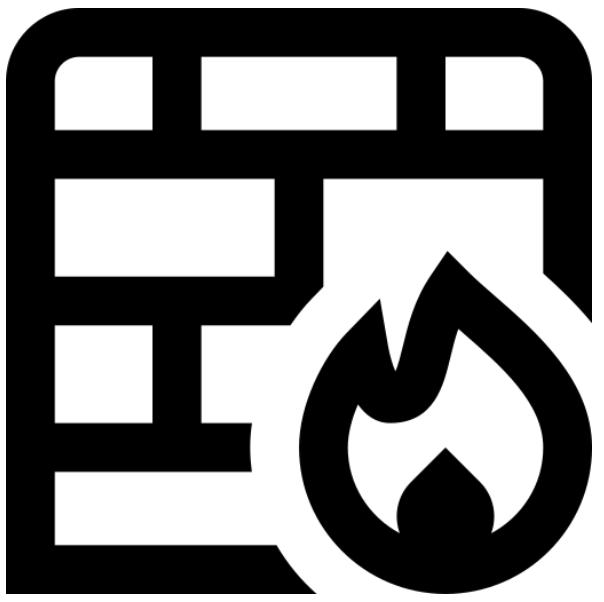
為甚麼駭客攻擊一般都是針對本機伺服器而不是其他周邊設備，這當然意味著作為核心一般的本機有著可能連你都不清楚的重要性。

實話實說，不會有人去攻擊一個沒有價值的伺服器，所以要嘛你已經有一定的規模，要嘛有一定程度的敏感資訊去引誘他人攻擊。

當然，跟現實生活一樣會有外患內憂的存在，後者比較偏向軟硬體的維護問題。

先來講一些好玩的。

 Google Developer Student Clubs



襲擊篇 - 汗染快取

快取危機，隆重回歸

還記得快取吧，就是一個臨時的記憶體，用來加速伺服器的資料存取。

今天一個攻擊者可以透過惡意操控伺服器端快取的方式，將惡意數據給注入其中，使得伺服器在查詢資料時獲得錯誤且危險的數據。

這種快取汙染容易發生在DNS上，讓原先正常的網址被導向到惡意的網站IP。

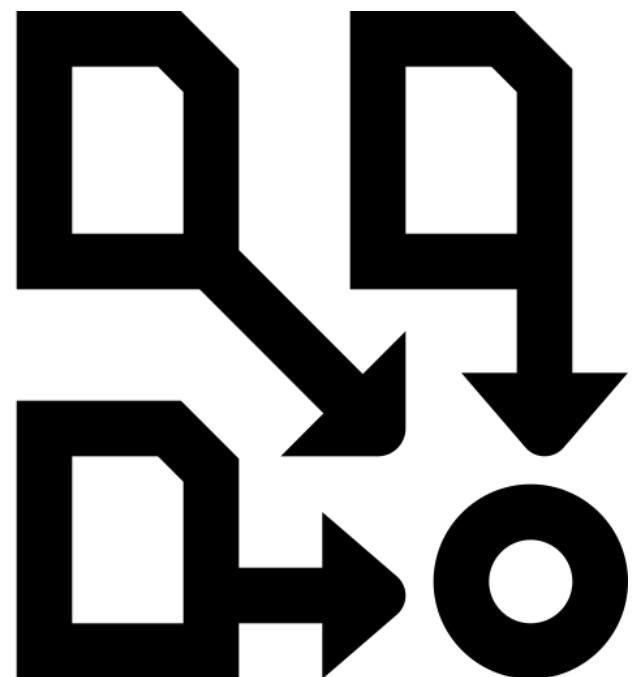
在HTTP上的汙染也可能會導致使用者得到錯誤的資訊，甚至是病毒。



襲擊篇 - 癱瘓伺服器

拒絕服務攻擊，分散式殭屍網路

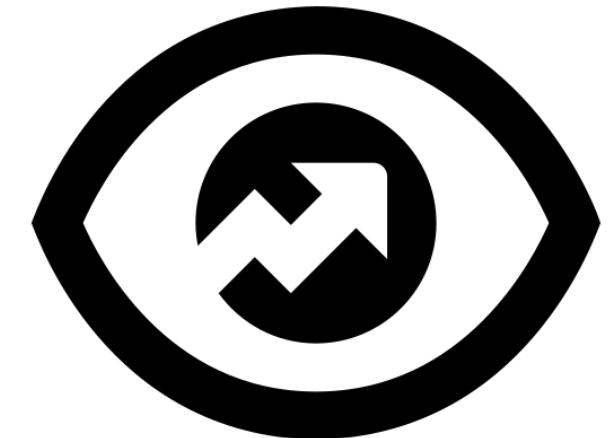
一定或多或少有聽過DoS這個詞，或者說DDoS比較熟悉？



DoS: Denial of Service 攻擊非常淺顯易懂，使用大量的請求封包來堵塞伺服器的回應能力，以達到癱瘓伺服器的目標。

而DDoS更加恐怖，使用大量設備對同一個伺服器進行DoS，已達成類似殭屍網路的功能，會更加難以應對，並且容易進入長期的Downtime。

襲擊篇 - 不被歡迎的第三者



你會知道間諜程式的可怕性

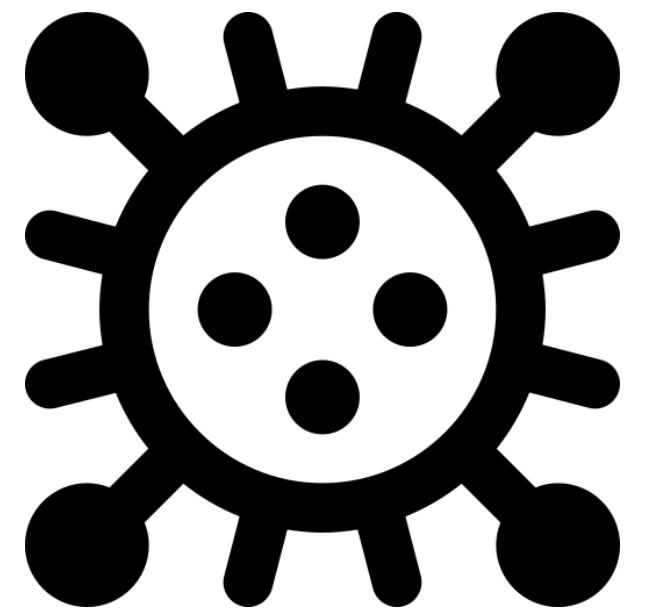
中間人攻擊 (MITM) 名副其實就是在一個兩端建立好的連線中插入間諜程式來干擾對話行程，以從中獲取敏感或者有價值的資訊。

常見的形式包含且不僅限: Wi-Fi攻擊、DNS偽造以及TLS剝離等等，三者都是竊取資訊的方式，只是方法不同罷了。

這種攻擊最可怕的是，如果太過隱密，可能在抽出間諜程式之後都還沒人察覺。

襲擊篇 - 假疫苗注入

被人操縱SQL語句可不是能夠輕鬆解決的問題



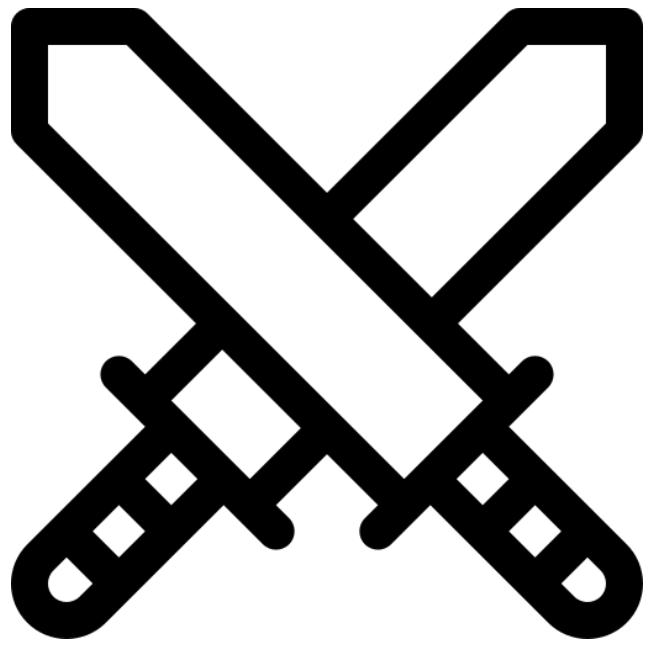
在資料庫的章節我們有提到過伺服器端去查詢資料的方式是使用SQL語句去對伺服器進行查詢，但這也代表所有資料庫用的查詢方法都相同。

假如今天有人在文字輸入框內使用惡意的資料庫語法，他就可能用這種方式來竊取資料庫當中的敏感資訊，甚至是刪除整個資料庫。

事實上，這種做法是最容易且耗費最少資源的攻擊，卻能帶來最可怕的後果。

反攻的號角

從被動狀態轉為主動



我們舉了幾個常見的攻擊例子，不過當然也不只侷限於那些，像是暴力試錯、勒索病毒等等形式也不算稀有。

了解的問題的根源之後，我們該來討論一下現在的社會中是如何去有效解決/預防這些可能的攻擊模式的。

畢竟，真正有防禦能力的也還得是伺服器端本地。



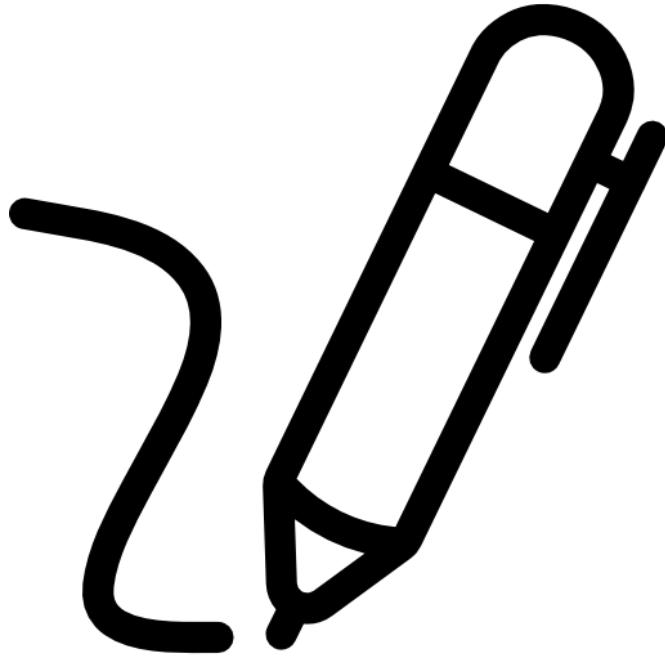
反擊篇 - 簽名檔

即使是快取 – 正因為是快取

防止快取污染的方式固然多，但在大多數情況下我們只要做一件事就可以了：加入校驗程序，也就是所謂的簽名檔。

快取雖然是暫時性的資料，卻容易被外部使用者竊取，透過簽名檔的信任程序能夠確保其資料無法被輕易取用，也能夠大挫駭客們的士氣。

而像是DNS竄改等等的情況，也可以使用DNSSEC去防禦。





反擊篇 - 流量檢測

識別與限制，防禦與監測

DoS和DDoS固然可怕，卻很容易去預防，畢竟是流量攻擊的概念。

一般來說比較進階的伺服器架構會使用WAF和IDS/IPS去識別以及封鎖流量，根據防禦程式的判決去找出問題來源的IP，並trace回到攻擊者的電腦。

不過更加簡單的方法是限制單一IP的流量上限，以防止請求溢出，而且這種方式也可以很容易地去觀測異常流量。

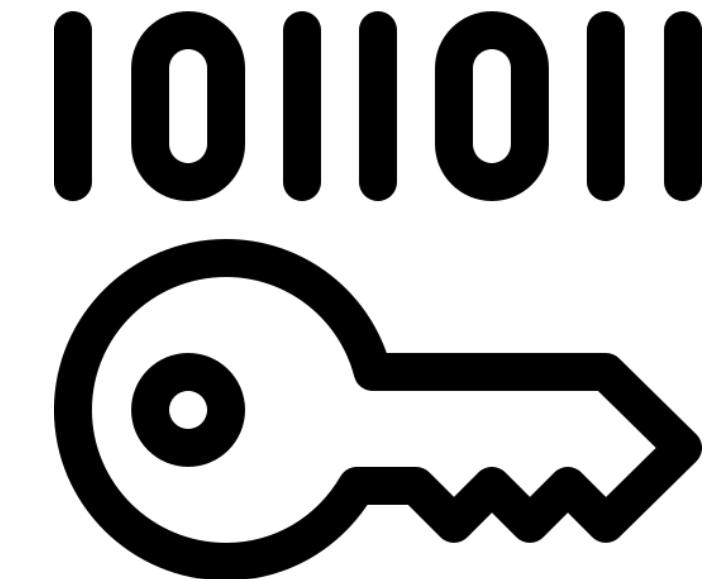
反擊篇 - 加密數據

憑證訂選與端對端加密

MITM的起源是連接之中的脆弱點，也就是未被保護的數據和進入位置，因此攻擊一旦開始就很難被防禦。

為了保護連接工作階段，我們會用使用加密協議及數位憑證去進行端點的驗證，並且確立一個安全的連線，這樣即使攻擊者能夠插入間諜，也只能得到加密過的資料，且沒有鑰匙的情況下要解密幾乎不可能。

對於一些降級攻擊，也可以使用HSTS去做防禦。



反擊篇 - 參數化查詢

杜絕任何第三方的查詢語句



SQLi能造成的影响深不可测，幸好这算是容易预防的情况。

我们知道骇客能够使用SQL语句来绕过检测并存取资料库，但毕竟还是外部的输入，我们只要能够去侦测甚至格式化它们就可以了。

像是使用ORM去防止注入漏洞、进行输入验证等等，甚至直接限制使用者的资料库存取权限，都能够避免灾难性的危机。

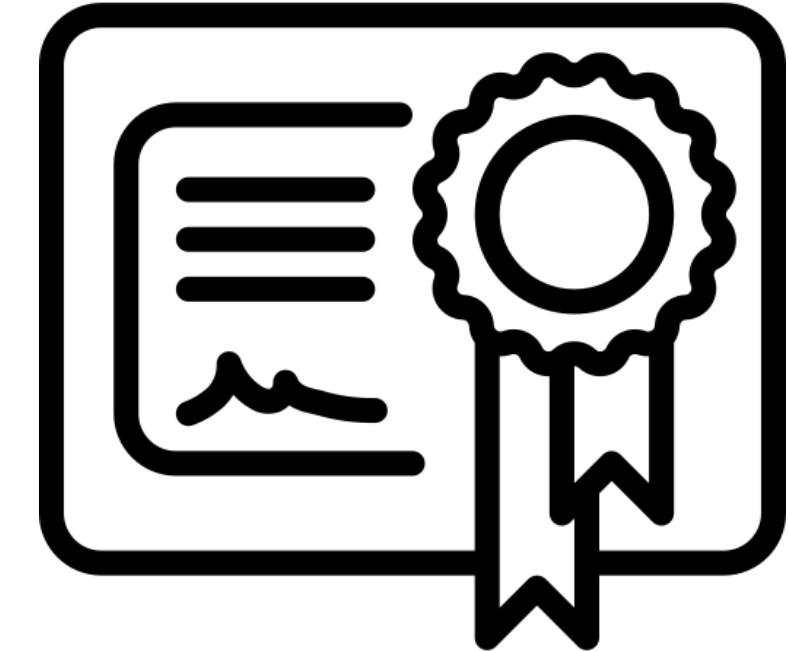
憑證管理

屬於你的各式各樣的鑰匙

有時候最大的敵人往往是你自己，尤其是在憑證外洩是如此容易的情況下。

架設伺服器服務時，我們經常去使用外部API、雲端部屬服務等等，此時會有各式各樣的憑證需要管理，儲存在本機端。

如果就這樣推上GitHub，就相當於你公布你的密碼給大家看，因此做適當的憑證管理是相當重要的，一般會使用隱藏的JSON或者.env檔案去處理。



良善的維護程序

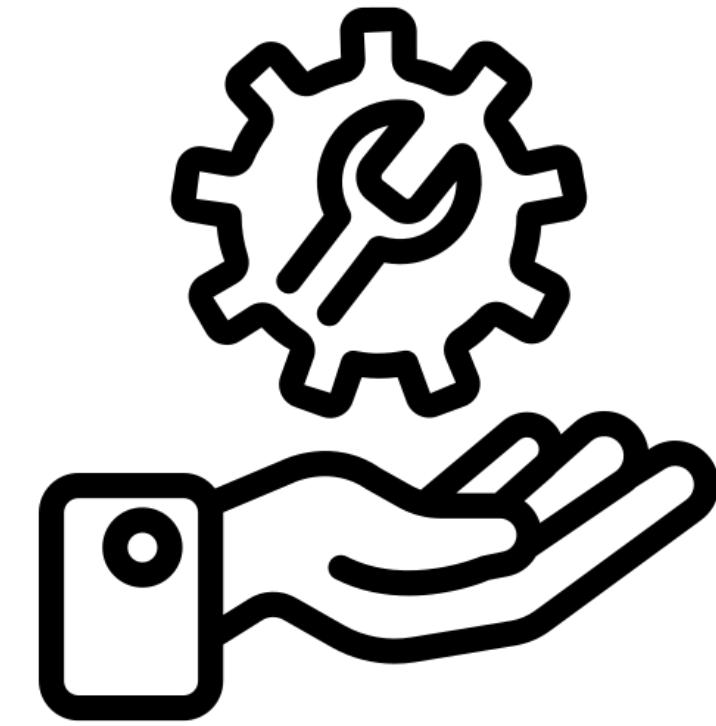
永續經營的關鍵

總結一下，進入尾聲吧。

要確立一個伺服器服務的經營，並不是在部屬之後就置之不理，而是要定期定期的去進行維護，並且透過軟硬體的添置去增強其資安強度。

適當的使用防火牆、SSL加密等等方式去保護進出的流量。

定期更新、進行滲透測試等去補強系統弱點。



通訊的藝術！ 不侷限於人之間

哇，沒想到還點題了呢！
(伊馮娜認為已經對伺服器的概念了解透徹了)





神奇的紫色通道





Developer Student Clubs
National Kaohsiung Normal University

10

無拘無束

那麼，該你們了

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterByStatus = filterByStatus ? study.status === filterByStatus : true;
const filterStudies = studies.filter(study => filterByOrg(study) && filterByStatus(study));
```





還有嗎?? (驚恐)

先不要驚慌害怕，我們的伺服器與通訊原理基本上都上完了，今天主要是會小小的介紹一些相關訊息。

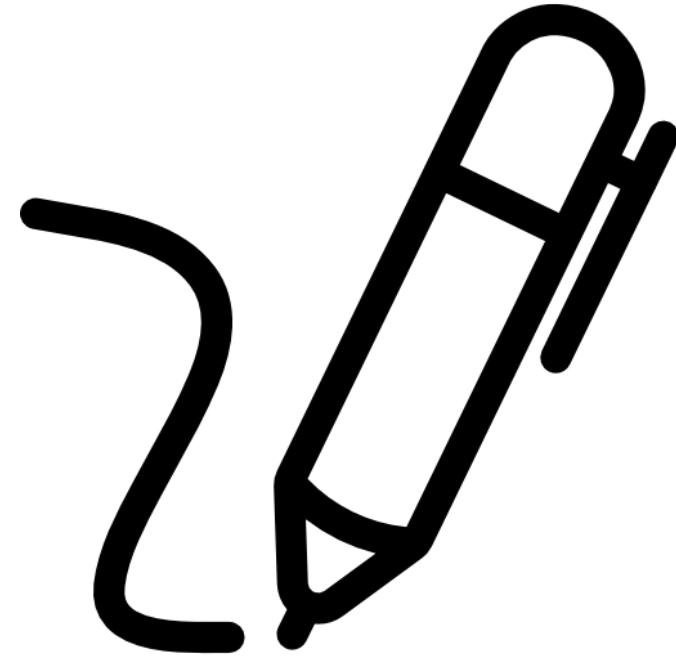
實作上工程師們都是怎麼架伺服器的、用甚麼架構等等。

一直用Python開發很丟人的，我們就是要多元學習。



技術堆疊

先來介紹一些先人的智慧



一個伺服器的架設大致有這些領域要去注意：前端、後端、資料庫、部署、版本控制，每一個都是截然不同的領域。

換句話說，開發的方法不同，使用不同的工具，卻要將其完全串聯起來。

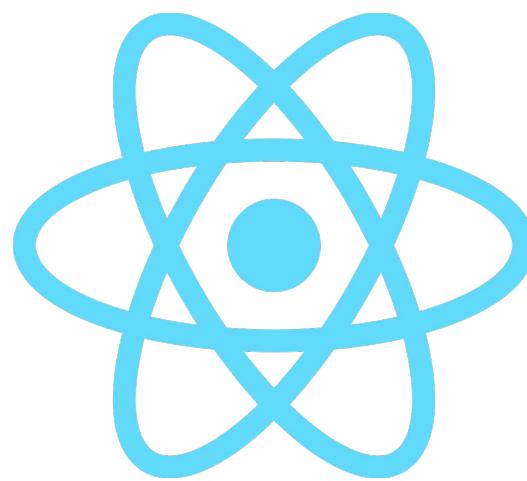
軟體工程師的職缺很多啊，不是所有人都想要花時間去做這種事，你也不可能奢求一個人能夠同時顧好所有這些領域...

它的樣子

前端工具

前端就是你的門面，使用者第一步會去注意到的部分。

React (JS) - 一步一腳印的元件設計，適合快速小型的使用者介面。



Angular (TS) - 全方位框架，由Google爸爸所開發，規模較大的專案所使用。



Vue (TS) - 容易上手，操作靈活，而且支援漸進式框架。



它的內在 後端工具

後端就是你的操作核心，也就是所有資訊的應對之處。

Node.js (JS) - 具有JavaScript的NPM生態系，並且適合處理高併發設計。



Django (PY) - 內建功能巨多，並且有自帶的資安防護網，輕易杜絕小型攻擊。



Flask (PY) - 架設速度極快，帶有自定義系統，與其他工具相性頗高。



它的知識

資料庫工具

資料庫是所有資訊的存放位置，也就是記憶的地方。

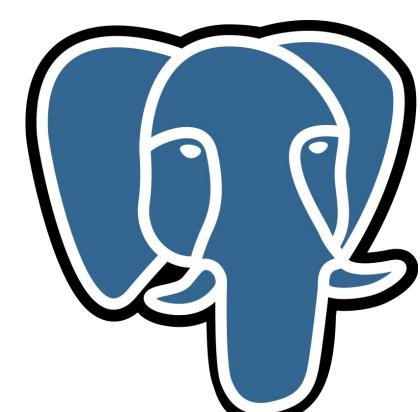
MySQL - 最簡易、好用，結構化語言很好學，而且方便快速。



MariaDB - MySQL的姊妹，與其完全相容，查詢優化更高效且容易擴展。



PostgreSQL - 關聯式資料庫，安全、強大，通常是大型專案所使用。

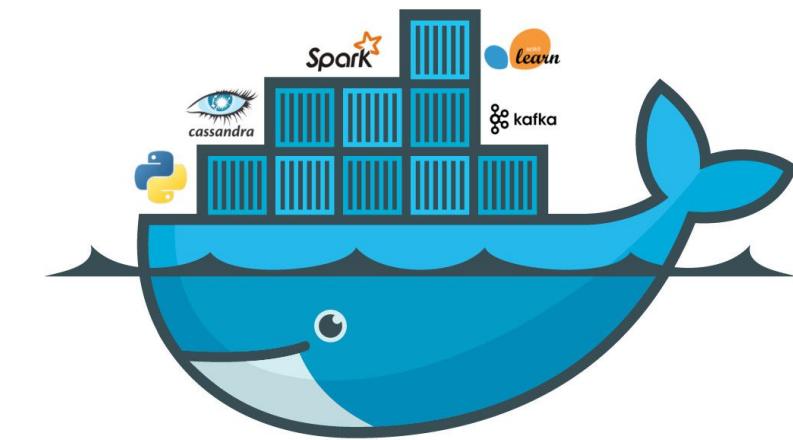


它的位置

部署工具

前面有說過，部屬一個伺服器才能讓他對外可見。

Docker (Go) - 輕量容器化技術，快速部屬並且容易移植。



Kubernetes (Go) - 能夠自動化多個容器的部屬，並隨時或定期的去監控。



kubernetes

Jenkins (Java) - CI/CD工具，支援各種插件，開放原始碼社群。



Jenkins

它的歷史

版本控管工具

不要省略版本控管，這東西會要了你的命。

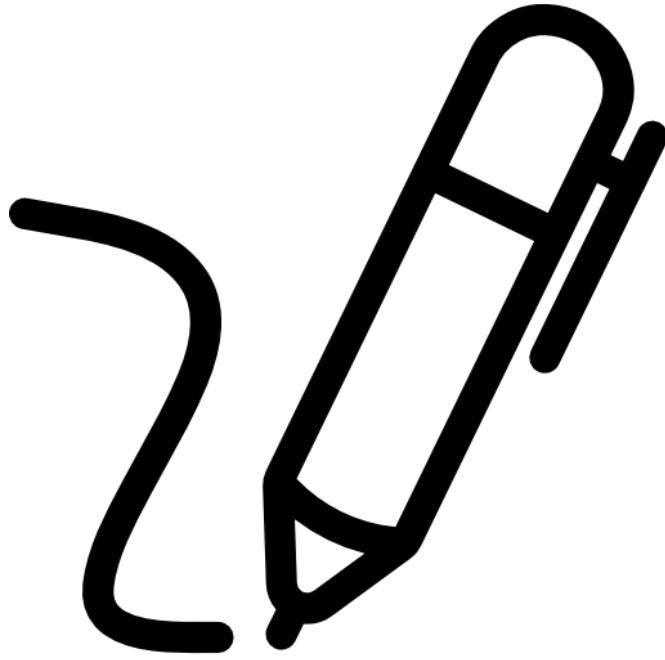
Git, 對我就只會講這一個，因為像是GitHub/GitLab等等存放庫平台都是用這個架構在處理的，讓人容易與他人協作、創建分支與遠端操作。

詳細介紹可以講一整節課，下學期有機會的話就講講看吧。



它的歷史

版本控管工具



不要省略版本控管，這東西會要了你的命。

Git, 對我就只會講這一個，因為像是GitHub/GitLab等等存放庫平台都是用這個架構在處理的，讓人容易與他人協作、創建分支與遠端操作。

詳細介紹可以講一整節課，下學期有機會的話就講講看吧。

工作流

其實就跟其他大部分專案開發差不多

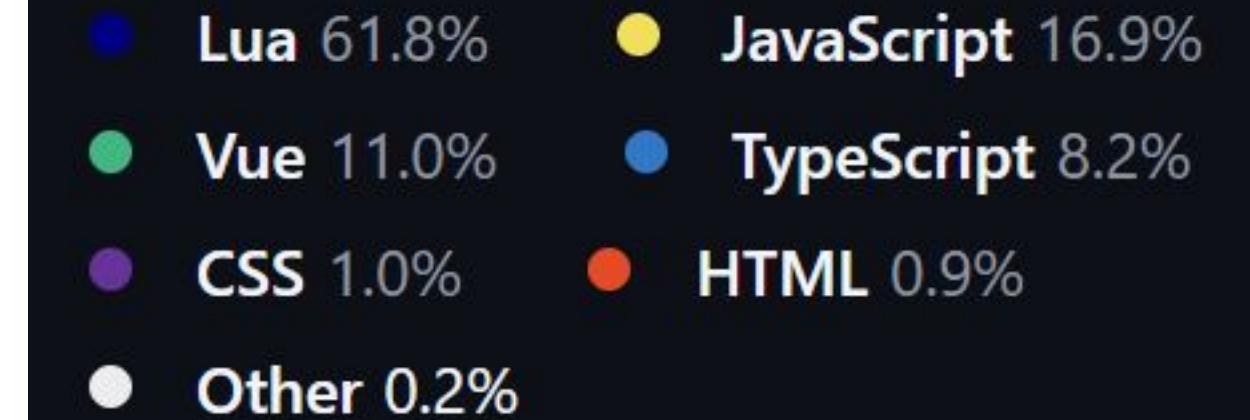
某些人會執著於一種程式語言去做開發，認為這樣比較統一而且好管理，我也是的確不能否認，但那只能說是最理想的狀況。

大型專案，尤其是那些需要多模組化的，多種程式語言合併開發不可避免。

一個好的工作流能夠協調工作程序，不管是單人或者多人開發上都很重要。

重點是整合的部分啦，要怎麼把不同架構的半成品融入到最終成品之內。

Languages

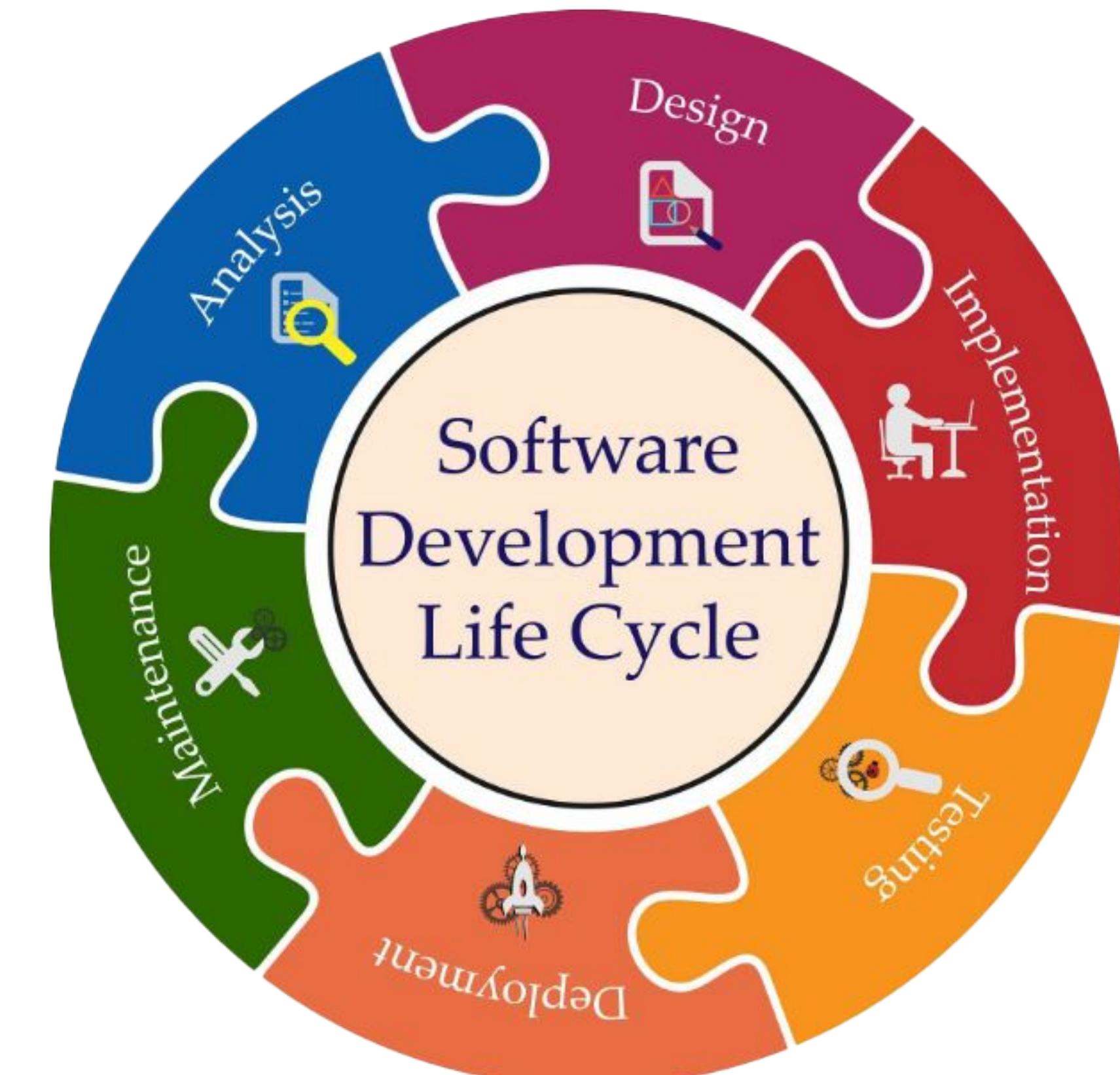


倒數第二張投影片

果不其然還是要水時間

有參加我的自動化測試讀書會的都已經看過這張圖至少兩次了，我也不會完全從頭講起。

這裡我只是想要傳達一個很重要的概念，聽不聽隨便你：你可以用任何方法去開發自己的專案，不管是軟體還是其他方面都好，沒人逼迫你遵循業界定則，但它們的存在有著他們的價值，而這些價值並不是每個人都能輕易模仿的。



就醬了~



Discord



GitHub

感謝你們的參與!

有任何其他問題可以到我們的DC群詢問，也可以直接寄件給講師們！
今天的內容都放在我們的GitHub頁面上了，有興趣的人可以去看看～

Bernie: ptyc4076@gmail.com

吳稼澂: charlie930320@gmail.com



Developer Student Clubs
National Kaohsiung Normal University

?

鏡像空間

實作環節，大家都期待的

```
    study = filterByOrg ? study : organization.filterByOrg();
    status = filterByStatus ? study.status == filterByStatus : true;
    if (!status) return;
    matchStatus);
}

function filterStudiesIf(studies, filterByOrg, filterByStatus) {
    let filteredStudies = studies.filter(study => {
        if (filterByOrg) {
            return study.org === filterByOrg;
        }
        if (filterByStatus) {
            return study.status === filterByStatus;
        }
        return true;
    });
    return filteredStudies;
}
```





編譯環境

從Python的安裝開始，我們並不會用到其他編譯器

[Python 官網](#)

這堂課演示的是使用Python進行簡易Socket程式的開發，因此希望每個人都能夠事先安裝好相對應的工具。

接下來的幾張簡報將會實行程式的實際編寫，因此如果在哪些部分有所疑問，歡迎提出。

程式語言

開始來寫一點東西？

可以的話，我希望不用花太多時間來介紹Python的使用，因此我大概用幾句話 (+螢幕分享) 來總結

他是動態型別 (變數不需要指派資料型態)、程序式、結構化、模組化、反射式、物件導向...

反正就是很厲害，也很好上手就是了

```
n = input()  
print(n)
```

導入必備資源

好在socket是Python內建的函式庫

由於要模擬伺服器與客戶端的連結，我們使用
socket這個小東西來做為今天的主角

下面的threading也很快就會用到了...

我們需要分成兩個檔案，一個模擬伺服器，一個
模擬客戶端，兩者都要導入函式庫！

```
# -*- coding: utf-8 -*-
import socket
import threading
```

開始有點東西

就如之前所說的，伺服器會監聽連線請求

設定串接類型
(不重要)

設定為TCP連線

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0', 12345)) # 修改為伺服器要監聽的 IP 和埠號
server_socket.listen(1)
```

我們的伺服器始終保持一個接收的狀態，這裡的
0.0.0.0指的是localhost，也就是你的本機IP

埠號的部分可以隨便填 (某些除外...)

客戶端也大同小異

就只是幾個字的差異罷了

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('0.0.0.0', 12345) # 修改為伺服器的 IP 和埠號
client_socket.connect(server_address)
```

客戶端的部分，由於我們要連到預先設置好的伺服器，我們需要填入一模一樣的IP和埠號

接著就是連線的部分...

處理訊息的發送

瞬間麻煩起來了？

```
while True:  
    message = input("客戶端: ")  
    if message.lower() == 'exit':  
        break  
    client_socket.send(message.encode())
```

處理發送的數據，我們首先須要讓使用者能夠輸入訊息

隨後，我們把訊息經過編碼，傳送到socket當中（此時伺服器還沒有接收到）

圖中的判斷式讓使用者得以終止程式

處理訊息的接收

等一下，怎麼圖片愈來愈大了??

伺服器的接收就只是把socket上的新東西給抓下來

你可能有注意到這一整個流程在方法當中，這就是為什麼呢？

```
def receive_messages():
    while True:
        try:
            message = client_socket.recv(1024).decode()
            if message:
                print(f"\r客戶端: {message}\n伺服器: ", end='')
        except:
            print("\n客戶端已斷開連線")
            break
```

執行緒的關鍵作用

讓程式不會塞車

```
def main():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('0.0.0.0', 12345) # 修改為伺服器的 IP 和埠號
    client_socket.connect(server_address)

    print("已連接到伺服器")

    threading.Thread(target=receive_messages, args=(client_socket,)).start()

while True:
    message = input("客戶端: ")
    if message.lower() == 'exit':
        break
    client_socket.send(message.encode())

client_socket.close()
```

發送和接受沒有辦法同時進行，否則整個程式會大塞車

因此，是時候用到我們的Threading了

透過將整個接收訊息的程式丟進另一條執行緒裡，我們成功地達到多工處理

記得把資源回收

讓程式不會塞車

```
def main():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('0.0.0.0', 12345) # 修改為伺服器的 IP 和埠號
    client_socket.connect(server_address)

    print("已連接到伺服器")

    threading.Thread(target=receive_messages, args=(client_socket,)).start()

    while True:
        message = input("客戶端: ")
        if message.lower() == 'exit':
            break
        client_socket.send(message.encode())

    client_socket.close()
```

在使用者寫上exit之後，整個程式應該就會結束了

我們不能忘記要把socket給關起來，以免造成不必要的資源浪費

聲明自己是主程式

我們沒有要讓程式模組化運行（至少現在還沒）

在剛剛的兩張截圖當中，你會發現我把整個
程式用main方法給包了起來

當然，你也可以直接零散的做，但是我會建
議把這個當作是一種習慣

右側的這個判斷式比較特別，我應該會直接
用講的

```
if __name__ == "__main__":
    main()
```

全部套起來

那麼，我們的運行邏輯當然也能夠打包

如果我們把整個程式邏輯給直接放在一個方法之內，程式的可讀性就變更高了

我們只需要在main函式當中去呼叫他就好

```
def handle_server(client_socket):
    def receive_messages():
        while True:
            try:
                message = client_socket.recv(1024).decode()
                if message:
                    print(f"\r伺服器: {message}\n你: ", end="")
            except:
                print("\n與伺服器的連線已中斷")
                break

    threading.Thread(target=receive_messages).start()

    while True:
        message = input("你: ")
        if message.lower() == 'exit':
            break
        client_socket.send(message.encode())

    client_socket.close()
```



就變這樣

簡單明瞭不是嗎？

目前為止的程式下載

```
def main():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('127.0.0.1', 12345) # 修改為伺服器的 IP 和埠號
    client_socket.connect(server_address)

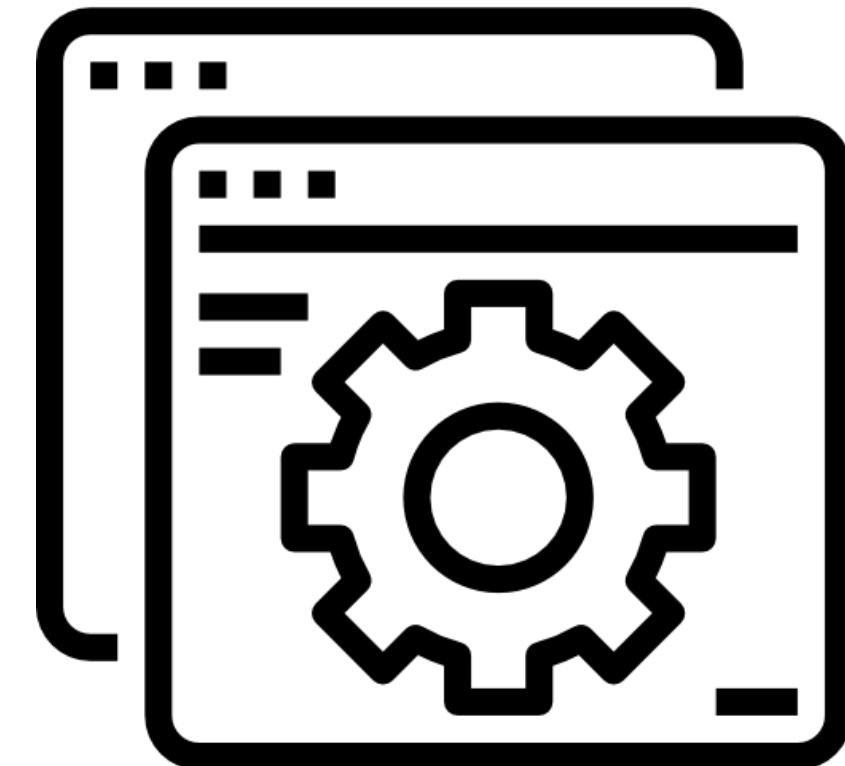
    print("已連接到伺服器")

    handle_server(client_socket)

if __name__ == "__main__":
    main()
```

接下來的部分...

我還沒有達成我一開始的承諾



接下來，我們把這個小小練習給推到極限吧！

我們用Python製作視窗介面，並且將我們的Socket邏輯給放進去

如果我想得沒錯的話，一個小型的對話窗口就這樣產生了

但前端開發可不是件容易的事...我們先從認識工具開始吧

TK不是team kill

Python中最原汁原味的視窗製作工具



在Python當中要實作視窗設計通常有兩種套件可以使用: Tkinter和PyQt, 前者是內建的函式庫, 不需要安裝, 後者會需要pip install。

差異在哪裡? PyQt是一個比較完善的架構, 也支援很多更為進階的元件種類, 而Tkinter比較原始, 卻也比較好上手。

在今天的練習當中, 我們會優先注重在Tkinter的實作上。

而至於怎麼實作, 就先從認識整體架構開始。

根基(主要窗口)

但Tkinter不支援多執行緒的主窗口

如果對上學期的課程還有一些印象的話，一個視窗的根本就是元件之間的交互。

我們需要有一個主要的窗口來放置這些元件，但！我們可是要實作伺服器以及客戶端呢！

基於Tkinter的神奇限制，一次只能存在一個主窗口，因此這種交互我們必須用Toplevel(頂層視窗)來實現。

```
import tkinter as tk
from UI.client_window import ClientWindow
from UI.server_window import ServerWindow

def run_application():
    root = tk.Tk()
    root.withdraw() # 隱藏主視窗

    # 創建兩個獨立的窗口
    client_root = tk.Toplevel(root)
    client_root.title("客戶端")
    ClientWindow(client_root)

    server_root = tk.Toplevel(root)
    server_root.title("伺服器端")
    ServerWindow(server_root)

    root.mainloop()
```

```
class ServerWindow:  
    def __init__(self, master):  
        self.master = master  
        self.master.title("伺服器端")  
        self.master.configure(bg="#2C2C2C")  
  
        custom_font = font.Font(family="Microsoft JhengHei", size=10)  
  
        self.ip_label = tk.Label(master, text="IP位址:", bg="#2C2C2C", fg='white')  
        self.ip_label.grid(row=0, column=0, padx=5, pady=5, sticky='w')  
  
        self.ip_entry = tk.Entry(master, bg='#424242', fg='white', font=custom_f  
        self.ip_entry.grid(row=0, column=1, padx=5, pady=5, sticky='ew')  
  
        self.port_label = tk.Label(master, text="埠號:", bg="#2C2C2C", fg='white')  
        self.port_label.grid(row=1, column=0, padx=5, pady=5, sticky='w')  
  
        self.port_entry = tk.Entry(master, bg='#424242', fg='white', font=custom_f  
        self.port_entry.grid(row=1, column=1, padx=5, pady=5, sticky='ew')  
  
        self.listen_button = tk.Button(master, text="開始聆聽", command=self.start)  
        self.listen_button.grid(row=2, column=0, columnspan=2, padx=5, pady=10,
```

元件大家族 (續)

總有種似曾相似的感覺...?

Label! Button! 這些曾經用過的名詞又再次回來了，不同的是Python並沒有一個設計界面可供使用。

沒錯，要回歸最原始的形式去逐行寫程式碼...然後布置每一個元件。

其中style、佈局還有各式屬性都要自己處理，真的要那麼死人嗎？

真的那麼做的話今天上不完

好家在，我已經幫各位把介面都弄好了！

我感受到大家的恐懼了，因此我們不會強迫你們去從頭到尾寫一個完整的視窗應用程式。

不過，基於我們的重點是伺服器的架設實作，裡面的所有關鍵功能都還是需要依賴各位的小手手。

接下來會一個一個帶過所有需要處理的部分。

首先，幫我創一個資料夾，並在裡面加入兩個.py檔案。



程式碼請由此獲得↑

初始化

Logic跟UI一樣都是用Class的部分去實作

第一件也是最重要的事情就是導入依賴的函式庫，
以及建立類別的建構子。

作為一個伺服器，我們需要有一個socket用來聆聽
可能的客戶端請求，以及另一個socket用來記錄已
經連線的客戶端。

UI的部分直接使用提供的程式即可。
客戶端的部分基本差不多而且只需一個socket。

```
✓import socket
| import threading
| from datetime import datetime

✓class ServerLogic:
| ✓def __init__(self, ui):
| | self.ui = ui
| | self.server_socket = None
| | self.client_socket = None
```

主要的聆聽程式

做為一個server，我們要做的是準備連線

這個部分看起來多，其實就是跟之前差不多的邏輯罷了。

我們在每一個階段最後都使用Thread去處理下一個部分。

開始聆聽 -> 處理連線 -> 接收訊息 這樣子的程序

```
def start_listening(self, ip, port):
    self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server_socket.bind((ip, port))
    self.server_socket.listen(1)
    self.ui.display_message("伺服器正在聆聽...\n")
    threading.Thread(target=self.accept_connection).start()

def accept_connection(self):
    try:
        self.client_socket, addr = self.server_socket.accept()
        self.ui.display_message(f"<已建立與 {addr[0]} 的連線>\n")
        threading.Thread(target=self.receive_message).start()
    except:
        self.ui.display_message(f"<連線已中斷: {'伺服器' if not self.ui.connectio
        self.ui.close_connection()
```

訊息處理

麻煩的是跟UI做聯合，好在這部份現在不會頭痛

接收以及發送訊息大致上也和上次做得差不多，只是這次要能夠及時的顯現在視窗上。

我有一個display_message方法專門拿來做這件事，所以不用驚慌！

try/except主要是要處理斷線。

```
def receive_message(self):
    try:
        while True:
            message = self.client_socket.recv(1024).decode()
            if message:
                timestamp = datetime.now().strftime('%H:%M:%S')
                self.ui.display_message(f"{timestamp} - 客戶端: {message}")
    except:
        self.ui.display_message(f"<連線已中斷: {'伺服器' if not self.ui.connection else '客戶端'}>")
        self.ui.close_connection()

def send_message(self, message):
    if message and self.client_socket:
        timestamp = datetime.now().strftime('%H:%M:%S')
        self.ui.display_message(f"{timestamp} - 伺服器: {message}")
        self.client_socket.send(message.encode())
```

申請連線

這裡比起伺服器要簡單多了

客戶端的部分不需要聆聽連線，而且處理訊息的部分也和伺服器一樣。

因此，我們要額外做的事就只有進行連線的程式，這個部分也跟之前講到的差不多。

然後無法連線的部分一樣也要顧。

```
def connect_to_server(self, ip, port):
    try:
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client_socket.connect((ip, port))
        self.ui.display_message(f"<已建立與 {ip} 的連線>\n")
        threading.Thread(target=self.receive_message).start()
    except:
        self.ui.display_message("無法連線，因為目標電腦拒絕連線。\\n")
        self.ui.close_connection()
```

處理正確的關閉

這個部分等等會再講到

```
def close_connection(self):
    if self.client_socket:
        self.client_socket.close()
    if self.server_socket:
        self.server_socket.close()
```

我們加入一個斷開連線的機制來讓整體使用上更加方便，而且在除錯上也比較容易。

主要是要跟UI的程式進行連接，我們馬上會提到。

這個方法所做的事就只是把socket給關閉而已，注意客戶端的程式只有一個socket。

介面上的程式實作

其實我已經幫你們做完了 (?)

```
def close_connection(self):
    self.server_logic.close_connection()
    self.connection_state = False

    self.ip_entry.config(state='normal')
    self.port_entry.config(state='normal')

    self.listen_button.config(text="開始聆聽", command=self.start_listening)

def display_message(self, message):
    self.chat_box.config(state='normal')
    self.chat_box.insert(tk.END, message + "\n")
    self.chat_box.config(state='disabled')

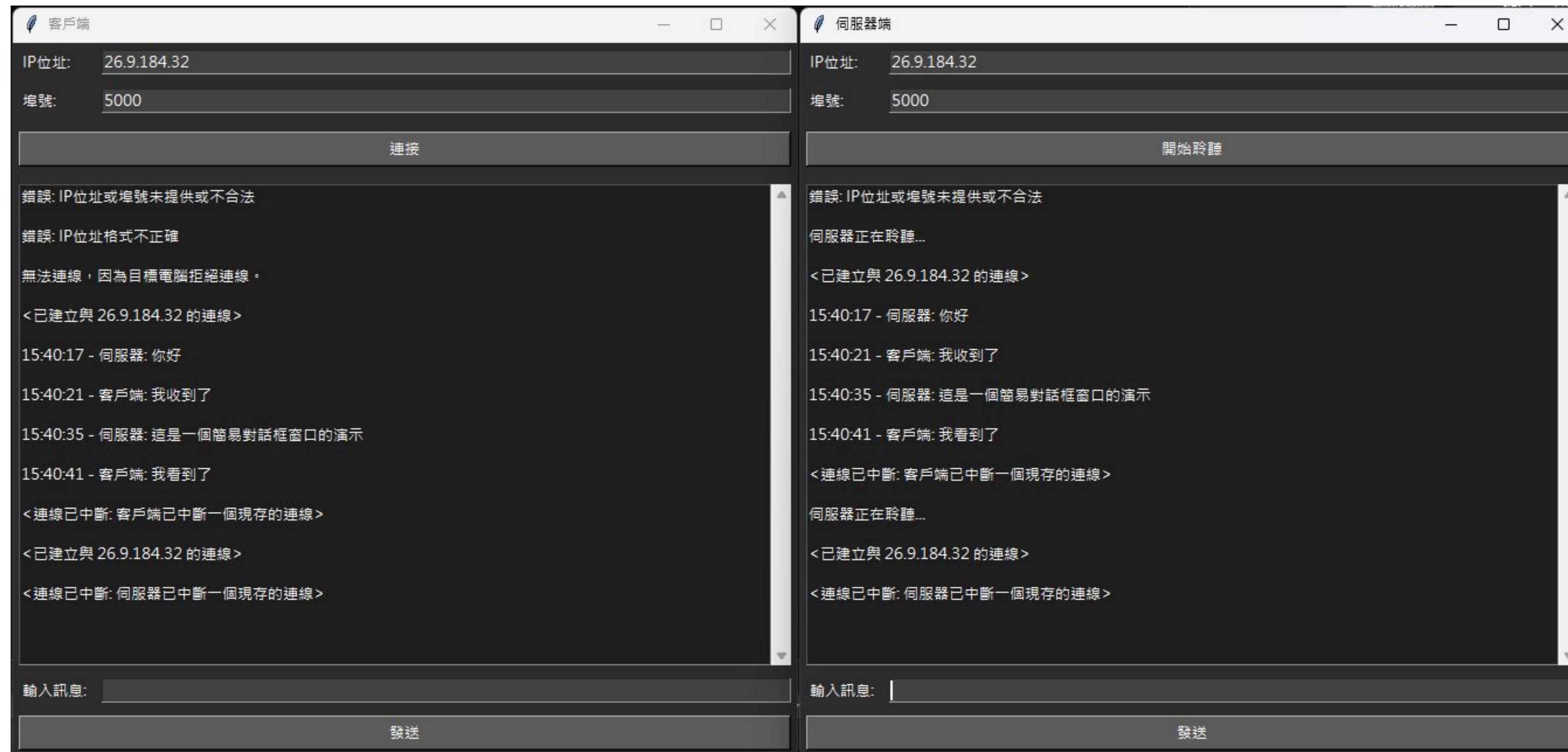
def send_message(self, event=None):
    message = self.message_entry.get()
    if message:
        self.server_logic.send_message(message)
        self.message_entry.delete(0, tk.END)
```

雖然不需要你們自己動手寫，但是我還是得講一下視窗介面裡的邏輯。

那個連線/聆聽的按鈕同時也是用來斷開連線的，而且我們有一個connection_state變數來確認當前連線狀況。

因此，如果自己斷開或者對方斷開，在程式當中都可以偵測。

對話窗口看上去的樣子



結語: 進步空間

很多部分都還可以去處理，並且精進

之後是我們的實作演示課，也就是我們這個系列的最後一堂課。

在今天的課程中我們並沒有太過注重在講解程式的部分，因為我希望各位可以自己去加以研究，進步，然後弄出一個與眾不同的應用程式。

下一次會介紹一些其他的伺服器工具（例如 Flask），以及其他的操作平台。

會預留大概半個到一個小時左右去讓大家展示。

