系統設計與分析 SAD 113-2

第 14 週課程：資料庫選型 & OpenAPI

助教：葉又銘、顧寬証　|　教授：盧信銘

上週回顧 (Week 13)

- **Docker 進階**：Registry、映像管理
- **Docker Compose**：多容器協調、服務依賴
- **Docker Swarm & Stack**：集群部署、零停機更新
- **SSH 遠端部署**：SSH 金鑰、`scp` 傳輸、指令自動化
- **Cloud Native** 概念：容器、微服務、CI/CD、DevOps

> ▶ **實作回顧**：使用 Docker Compose 部署 Todo App，並透過 SSH 上線至遠端伺服器。

本週內容 (Week 14)

1. **資料庫選型**：理論、實務與案例
2. **進階資料庫設計概念**
3. **OpenAPI (Swagger)**：Design-First API 開發
4. **實作時間**：撰寫並測試 OpenAPI 文件
5. **總結與期末提醒**

資料庫選型 (Database Selection)

資料庫不只有 Relational Database！

上學期很多人應該都有修過資料庫，但我們僅停留在 Relational Database...

但其實還有超多類型的資料庫，分別負責各種資訊系統開發場景！

- **關聯式 (SQL)**：PostgreSQL、MySQL、MariaDB、Oracle、SQL Server
- **文件型 (Document)**：MongoDB、CouchDB、RavenDB
- **鍵值型 (Key-Value)**：Redis、Memcached、DynamoDB
- **列式 (Column-Family)**：Cassandra、HBase、ScyllaDB
- **圖形 (Graph)**：Neo4j、ArangoDB、JanusGraph
- **時序 (Time-Series)**：InfluxDB、TimescaleDB、Prometheus
- **搜尋引擎 (Search)**：Elasticsearch、Solr、Meilisearch

資料庫是系統設計中最主要的一環

在軟體工程師的職涯中，系統設計面試是常見的挑戰。

深入理解不同類型資料庫的特性和應用場景，不僅能幫助你在面試中脫穎而出，更能為你提供構建可擴展、高效能系統的關鍵知識。

選擇合適的資料庫架構，往往是決定系統成敗的重要因素。

例如：設計一個 TikTok 這樣的影音社交平台，需要同時運用：

- **關聯式資料庫**：用戶資料、關係、權限管理
- **文件型資料庫**：動態內容、評論、互動記錄
- **搜尋引擎**：影片標題、描述、標籤的全文檢索
- **快取系統**：熱門影片、推薦列表的即時存取

> 所以資料庫遠比你想像的還要複雜！

為什麼資料庫選型重要？

資料庫會影響各種資源的運用與效能，例如：

- **資料怎麼存**：怎麼使用越少的資源存取越多的資料？
- **跑得快不快**：怎麼存取決定回應速度與可擴展性？
- **好不好開發**：是否有開發體驗與複雜度？
- **花錢多不多**：還要考慮學習、維運與雲端成本...
- **未來好不好改**：因為資料量通常龐大，遷移代價通常極高...

> **重點**：沒有「最好」，只有「最適合」你需求的資料庫。

SQL vs NoSQL：兩大陣營

特性	SQL (關聯式)	NoSQL (非關聯式)
資料結構	Schema-on-Write	Schema-on-Read / Flexible
一致性模型	ACID 強一致	BASE 最終一致
優勢	複雜查詢、交易可靠	高擴展、彈性、效能
常見應用	銀行、訂單系統	CMS、快取、大數據

> Schema-on-Write：寫入時定義結構，確保資料一致性
> Schema-on-Read：讀取時才解析結構，提供彈性

思考

如果你要設計一個社群媒體 App...

1. Threads Post?
2. Instagram Story?

> 你會選擇哪一種資料庫？

SQL (關聯式資料庫)

- **核心概念**：表格 (Table) + 關聯 (Relation)
- **ACID 保證**
  - **A**tomicity　*原子性*
  - **C**onsistency　*一致性*
  - **I**solation　*隔離性*
  - **D**urability　*持久性*
- **強項**：需要強一致 & 複雜關聯 (銀行、ERP)
- **代表**：PostgreSQL、MySQL、Oracle、SQL Server

> 相信大家都老熟了...

NoSQL (非關聯式資料庫)

- **核心**：Not Only SQL，多樣資料模型
- **BASE 模型** (最終一致)
  - Basically Available
  - Soft State
  - Eventually Consistent
- **優勢**：高擴展、彈性、特定場景高效能
- **代表類型**：Key-Value、Document、Column-Family、Graph

> 你會發現 NoSQL 的資料庫更能拓展系統的規模，但相對的也會犧牲掉一些穩定性

NoSQL 四大類型概覽

1. **Key-Value Stores**
2. **Document Stores**
3. **Column-Family Stores**
4. **Graph Databases**

1️⃣ Key-Value Stores（KV）

- 就是一個 Key 對應一個 Value，有點像 JSON、Map、Object 一樣，非常簡單
- `Key → Value` 直接映射，操作極速
- 典型場景：**快取**、**Session**
- 代表：Redis、DynamoDB、Memcached

例如：當你需要實作 API 限流 (Rate Limiting) 時，可以將使用者的 IP 位址或 Session ID 作為 Key，存取次數作為 Value，透過 Redis 這類 Key-Value 資料庫來實現高效能的用量控制。

2 Document Stores

- 就是 JSON/BSON 文件，Schema-Flexible
- 典型場景：CMS、logs
- 代表：MongoDB、CouchDB、Firebase

> JSON 是純文字格式，而 BSON 是二進制格式，提供更高效的序列化與更多資料類型支援

其實很多應用程式，都有一個叫 Store 的東西，用來存一些簡單的資訊，例如使用者設定

3 Column-Family Stores

- Row Key + Column Families，寫入吞吐高
- 典型場景：大數據分析、時間序列
- 代表：Cassandra、HBase、Bigtable

> 把資料分類存放，查詢複雜度就會降低、更有效率！

4 Graph Databases

- Nodes + Edges 表示關係，遍歷高效
- 典型場景：社交網路、推薦引擎、詐欺偵測
- 代表：Neo4j、Amazon Neptune

5 Vector Databases

- 專門處理向量資料
- 典型場景：推薦系統、圖像搜尋、自然語言處理、**AI**、**RAG**
- 代表：Faiss、Annoy、HNSW

選型考量 Checklist

1. 資料模型 & 關聯複雜度
2. 讀寫比例
3. 一致性 vs 可用性
4. 擴展策略 (Vertical / Horizontal)
5. 團隊熟悉度 & 生態
6. 成本、合規、安全需求
7. 專用功能：全文、GIS、圖遍歷、時間序列、向量搜尋

進階資料庫設計概念

> 提升效能、擴展與可靠性。

概念 1｜Sharding

將資料分散到多台伺服器上，就像把工作分配給多個團隊，每個團隊負責處理一部分資料，提升整體效能和容量

> 通常一個資料庫服務很難撐住所有流量，所以需要分散到多台伺服器上

概念 2｜Replication

資料複製機制，分為同步和非同步兩種模式。同步確保資料即時一致，非同步則提供更好的效能。主要用於提升系統可用性和讀取效能

> CDN 的原理就是這樣，把資料複製到多個地方，讀取時就近讀取

概念 3｜Caching

快取策略，就像把常用資料放在記憶體中，減少存取硬碟的次數。常見策略：

- Cache-Aside：先查快取，沒有才查資料庫
- Read-Through：讀取時自動更新快取
- Write-Through：寫入時同步更新快取
- Write-Back：先寫入快取，之後再同步到資料庫

概念 4｜CAP Theorem

分散式系統的三個核心特性，但只能同時滿足其中兩個：

- 一致性：所有節點看到相同的資料
- 可用性：系統持續回應請求
- 分區容錯：網路故障時仍能運作

概念 5｜Indexing

資料庫的索引機制，幫助快速定位資料。常見類型：

- B-Tree：平衡樹結構，適合範圍查詢
- Hash：雜湊表，適合精確匹配
- Full-Text：全文檢索
- Geo：地理空間索引

> 索引能提升查詢速度，但需要權衡維護成本

Discord 資料庫演進 — 一段擴展故事

背景與挑戰

- 2015-2016：Discord 用戶暴增，每天要處理的訊息量越來越大
- 原本用的 MongoDB 資料庫撐不住了：寫入速度變慢，延遲變高
- 最重要的是要確保全球用戶都能即時聊天，不能卡頓

階段一：改用 Cassandra

- **為什麼要換**：需要更好的擴展能力，能處理更多寫入
- **怎麼做**：用伺服器 ID 來分散資料，讓每台伺服器負擔變小
- **新問題**：讀取變慢了，而且資料一致性需要調整

階段二：改用 ScyllaDB

- **為什麼要換**：Cassandra 用 Java 寫的，記憶體回收會造成延遲
- **ScyllaDB**：用 C++ 重寫的版本，功能一樣但效能更好，伺服器數量可以減少一半以上
- **好處**：延遲降低 5-10 倍，硬體成本省了 30% 以上

其他優化方法

- **用 Redis 存狀態**：記錄誰在線上，快取常用資料
- **用 Rust 寫中間層**：統一管理資料流向，聰明地分散負載
- **大量使用快取**：減少直接讀取資料庫的次數
- **批次寫入**：把多筆資料一次寫入，減少即時壓力

學到的經驗

1. **不同資料庫各司其職**：根據資料特性選擇最適合的資料庫
2. **慢慢改，持續監控**：一步一步升級，風險比較小
3. **自己寫工具補強**：開發中間層和分片工具來解決限制
4. **持續優化**：系統擴展是永無止境的，要不斷改進

RESTful API 是什麼？

在我們深入探討 OpenAPI 之前，先來了解一下 RESTful API 的基本概念。

REST 核心原則

REST 是一種設計 Web 服務的架構風格，主要原則有：

- **資源 (Resources)**：用 URI 來標識資源，如 `/users`、`/products/123`
- **表述 (Representations)**：用 JSON 或 XML 等格式傳輸資源
- **狀態轉移 (State Transfer)**：透過操作資源來改變伺服器狀態
- **統一介面 (Uniform Interface)**：使用標準的 HTTP 方法來操作資源
- **無狀態 (Stateless)**：每次請求都是獨立的，伺服器不保存客戶端狀態

HTTP 方法與 RESTful API

RESTful API 常見使用 HTTP 方法來表達對資源的操作：

HTTP 方法	CRUD 操作	描述	是否冪等 (Idempotent)
`GET`	Read	讀取資源	是
`POST`	Create	新增資源	否
`PUT`	Update	更新或取代整個資源	是
`PATCH`	Update	部分更新資源	否 (通常)
`DELETE`	Delete	刪除資源	是

> **冪等性**：多次相同請求，結果應相同 (e.g., GET, PUT, DELETE)。
> 也就是我這次 GET 跟下次 GET 結果應該一樣，不會因為我 GET 一次就變成別的東西

為何選擇 RESTful API？

- **簡單**：基於 HTTP 標準
- **可擴展**：無狀態設計
- **靈活**：支援多種資料格式
- **通用**：跨平台支援
- **整合**：與 Web 技術相容

> RESTful API 是設計 Web App 的強大框架。為確保大型 API 的一致理解，OpenAPI (Swagger) 應運而生。

RESTful API 設計：常見的 好與壞 實踐

1. URI 設計

❌ `/getUsers`, `/createProduct`
✅ `/users`, `/products`

❌ `/users/1/updateEmail`
✅ `PUT /users/1`

> URI 用名詞，HTTP 方法表示操作

2. 格式與狀態碼

❌ 大小寫混用, 底線
✅ 小寫及連字號 (`/user-settings`)

3. 狀態碼

❌ 錯誤都回 200 OK
✅ 用正確狀態碼 (400, 404, 500)

> 快速理解結果

4. 請求與回應

❌ GET 用 Request Body
✅ 用 Query String

> GET 要冪等且可快取

5. 單複數

❌ 單複數混用 (`/user/1` , `/products`)
✅ 統一用複數 (`/users/1` , `/products`)

> 提高一致性

6. 進階設計

❌ 回傳過多/過少資料
✅ 提供篩選、分頁、欄位選擇

> 優化效能

討論

那如果我的 API 其實很複雜，不只是 CRUD？例如推薦系統、搜尋引擎、訂單系統...

一定要用 RESTful API 嗎？

> 當然不一定，像是 GraphQL、gRPC 等，都是現在有很常見的選擇

助教認為：

1. 如果是前端與後端溝通，用 RESTful API 或 GraphQL 是比較好的選擇
2. 如果是後端與後端溝通，用 gRPC 或 RESTful API 是比較好的選擇

OpenAPI (Swagger)

OpenAPI 是什麼？

- **OpenAPI Spec (OAS)**：以 YAML/JSON 描述 REST API 的標準格式。
- **核心理念**：API = 合約 → 人機皆可讀。
- **Swagger**：OAS 的工具家族 (Editor、UI、Codegen ...)。

> 就把它當成是一個設計 API 的工具吧！

為何使用 OpenAPI？

1. **標準化 & 共識**：同一份合約，減少溝通誤差。
2. **自動化**：生成 SDK / Server Stub / 測試腳本，節省重複工。
3. **互動式文件**：Swagger UI / Redoc 可即時 Try-It。
4. **Design-First**：先設計、再開發，降低返工。

直接來看範例！

https://editor.swagger.io/

文件結構總覽

Section	作用
`openapi`	版本號 (ex. 3.0.0)
`info`	標題、版本、描述、聯絡人
`servers`	API Base URLs
`paths`	各端點 (Endpoint) 及 HTTP 方法
`components`	共用資料模型 (schemas)、參數、回應、Security
`security`	全域安全設定 (OAuth2、API Key...)

YAML 範例片段

```yaml
openapi: 3.0.0
info:
  title: Todo API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /todos:
    get:
      summary: List all todos
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/TodoList'
```

```yaml
components:
  schemas:
    Todo:
      type: object
      properties:
        id: { type: string, readOnly: true }
        title: { type: string }
        isCompleted: { type: boolean, default: false }
    TodoList:
      type: array
      items:
        $ref: '#/components/schemas/Todo'
```

Design-First vs Code-First API 開發

Design-First 優點

- 團隊先達成 API 設計共識
- 前端可提前開發
- 文件即合約，減少溝通成本
- 支援自動化測試與程式碼生成

Code-First 優點

- 開發速度快，適合快速迭代
- 程式碼即文件，減少重複工作
- AI 輔助提升開發效率與文件生成品質
- 適合小型團隊或原型開發
- 框架自動生成文件

總結 (Week 14)

- **資料庫選型**：理解 SQL vs NoSQL 差異、常見資料庫與選型要點。
- **進階概念**：Sharding、Replication、Caching、CAP、Indexing。
- **案例**：Discord 擴展之路。
- **OpenAPI**：設計優先、工具生態與實作流程。

期末專案提醒

AMA (Ask Me Anything)

> 有任何問題都可以問我，我會盡量回答

Q & A

> 感謝聆聽 — 有任何問題歡迎提出！