1

建立你的第一個資料庫和資料表

chapter

SQL 的能耐不僅僅只是能從資料中擷取出知識而已。它也是一種能夠定義資料儲存結構的語言，讓我們可以組織資料之間的關係。在這套結構中，首要的主角便是資料表。

資料表其實就是一個由行列組成的表格，可以儲存資料。每一列都含有一組欄位，每個欄位則都含有特定類型的資料：最常見的莫過於數字、字元、還有日期。我們透過 SQL 來定義資料表的結構，還有每個資料表與資料庫中其他資料表可能會有的關係。此外也透過 SQL 來擷取、或是查詢資料表裡的資料。

要了解資料庫裡的資料，基礎就要從了解資料表開始。每當我接觸到一個陌生的資料庫時，頭一件事就是先看看裡面有些什麼資料表。我會觀察資料表及其欄位的名稱，試圖分辨出一些線索。資料表中是否含有文字、數字、或是兩者兼具？每個資料表有幾筆資料？

接著我會觀察資料庫中有多少個資料表。最簡單的資料庫可能只有單獨一個資料表。一個會處理到客戶資料或追蹤飛行里程的全方位應用程式，可能要處理成打甚至上百的資料表。我不只可以從資料表的數目得知需要分析的資料量，同時也提醒我每個資料表的資料之間有多少關係需要深究。

在你開始鑽研 SQL 前，我們先來看一個例子，體會一下資料表的內容會是什麼模樣。這裡有一個虛構的資料庫，是用來管理學校選課的；該資料庫裡有好幾個資料表，分別用來追蹤學生和他們所選修的課程。第一個資料表稱為 **student_enrollment**，它顯示出每堂課簽到的學生：

```
student_id    class_id      class_section    semester
----------    ----------    -------------    ---------
CHRISPA004    COMPSCI101    3                Fall 2017
DAVISHE010    COMPSCI101    3                Fall 2017
ABRILDA002    ENG101        40               Fall 2017
DAVISHE010    ENG101        40               Fall 2017
RILEYPH002    ENG101        40               Fall 2017
```

這個資料表顯示有兩名學生出席了 **COMPSCI101** 課程，另外三位則出席了 **ENG101**。但關於每名學生和課程的細節在哪呢？在本例中，這些細節都分別存放在 **students** 和 **classes** 兩個資料表裡，而且它們和 **student_enrollment** 都有關聯。這就是**關聯式資料庫**初露鋒芒的時候。

students 資料表的前幾行會像這樣：

```
student_id    first_name    last_name    dob
----------    ----------    ---------    ----------
ABRILDA002    Abril         Davis        1999-01-10
CHRISPA004    Chris         Park         1996-04-10
DAVISHE010    Davis         Hernandez    1987-09-14
RILEYPH002    Riley         Phelps       1996-06-15
```

students 資料表內有每個學生的詳盡資料，使用 **student_id** 欄位裡的資料值來識別每一名學生。這個資料值同時還擔任串起兩個資料表的獨特**鍵**（*key*）值，讓你可以把 **student_enrollment** 資料表的 **class_id** 欄位、和 **students** 資料表的 **first_name** 和 **last_name** 兩個欄位重組成以下資料列：

```
class_id      first_name    last_name
----------    ----------    ---------
COMPSCI101    Davis         Hernandez
COMPSCI101    Chris         Park
ENG101        Abril         Davis
ENG101        Davis         Hernandez
ENG101        Riley         Phelps
```

classes 資料表也有一樣的功用，它具有 class_id 欄位和幾個關於課程詳情的欄位。建構資料庫的人往往偏好把資料庫轄下的各個**實體**（*entity*）分散到各個資料表裡，以便減少重複的資料量。在本例中，我們把每位學生的名字和出生日期都一次存放在一個位置。就算學生曾出席不同的課程（就像 Davis Hernandez 一樣），我們也無須浪費資料庫空間，在 student_enrollment 資料表裡的每堂課都記錄他的名字。只需記錄他的 student_id 即可。

由於資料表是構成每個資料庫的核心組件，當你從這一章展開你的資料庫程式冒險之旅時，就要先從在新資料庫裡建立資料表開始起步。然後你還得把資料載入到資料表裡，再檢視完整的資料表是何模樣。

建立資料庫

你在簡介章節下載而來的 PostgreSQL 程式，就是所謂的**資料庫管理系統**，也就是可以讓你定義、管理和查詢資料庫的軟體套件。當你裝好 PostgreSQL，它就會建立一個**資料庫伺服器**（亦即一個執行在你電腦上的應用程式實例），內含一個名叫 postgres 的預設資料庫。資料庫其實由一系列物件組成，如資料表、函式、使用者角色等等。根據 PostgreSQL 文件所述，預設資料庫是「刻意設計給使用者、工具程式及第三方應用程式使用的」（參閱 *https://www.postgresql.org/docs/current/static/app-initdb.html*）。在本章的習題裡，我們會留著預設資料庫不去動它，而是另外建立一個新資料庫。這樣做是為了要保留跟特定主題或應用程式有關的物件，令其集中在一起。

1. 標點符號，包括引號、小括弧、和數字運算符號
2. 數字由 0 ～ 9
3. 其他標點符號，如問號
4. 大寫字母從 A 到 Z
5. 其他標點符號，如方括弧和底線
6. 小寫字母從 a 到 z
7. 其他標點符號、特殊字元、以及擴充字母

正常來說，排序的順序不會是大問題，因為字元欄位裡通常都只有人名、地址、說明文字、和其他直截了當的文字而已。但如果你在想為何 *Ladybug* 這個字會出現在 *ladybug* 之前時，現在你有答案了。

能夠在查詢時做排序的能力，讓我們在檢視和呈現資料時都多了很多彈性。舉例來說，我們可以一次替多個欄位排序。請輸入清單 2-6 的陳述：

清單 *2-6*：用 ORDER BY 排列多個欄位

```
SELECT last_name, school, hire_date
FROM teachers
❶ ORDER BY school ASC, hire_date DESC;
```

在本例中，我們會取出教師姓氏、執教學校、以及到職日期。然後再把 `school` 欄位依升冪排列，而 `hire_date` 欄位則依降冪排列 ❶（由近至遠），藉此我們便建立了一份所有同校教師都排在一起，並按照到職時間遠近排序的名單。這會顯示出該校最資淺的教師是哪一位。結果應該會像這樣：

```
last_name    school                hire_date
---------    -------------------   ----------
Smith        F.D. Roosevelt HS     2011-10-30
Roush        F.D. Roosevelt HS     2010-10-22
Reynolds     F.D. Roosevelt HS     1993-05-22
```

於是結果中便只會有任職於 Myers Middle School 的教師：

```
last_name    school                hire_date
---------    -------------------   ----------
Cole         Myers Middle School   2005-08-01
Bush         Myers Middle School   2011-10-30
Diaz         Myers Middle School   2005-08-30
```

以上我使用了等號來當做比較運算子，藉以找出完全符合某資料值的資料列，當然你也可以用其他運算子來搭配 WHERE 作為篩選條件。表 2-1 便列舉了最常用到的比較運算子摘要清單。不同的資料庫系統可能還會有更多運算子可以使用。

表 2-1：PostgreSQL 的比較和對應運算子

運算子	功能	範例
=	等於	WHERE school = 'Baker Middle'
<> 或 !=	不等於 *	WHERE school <> 'Baker Middle'
>	大於	WHERE salary > 20000
<	小於	WHERE salary < 60500
>=	大於或等於	WHERE salary >= 20000
<=	小於或等於	WHERE salary <= 60500
BETWEEN	介於範圍內	WHERE salary BETWEEN 20000 AND 40000
IN	符合某一組資料值之一	WHERE last_name IN ('Bush', 'Roush')
LIKE	符合特定樣式（分大小寫）	WHERE first_name LIKE 'Sam%'
ILIKE	符合特定樣式（不分大小寫）	WHERE first_name ILIKE 'sam%'
NOT	反相條件	WHERE first_name NOT ILIKE 'sam%'

※ != 並非標準 ANSI SQL 運算子，但 PostgreSQL 和其他幾種資料庫系統卻都認得。

```
dept    location    first_name    last_name    salary
----    --------    ----------    ---------    ------
Tax     Atlanta     Nancy         Jones        62500
Tax     Atlanta     Lee           Smith        59300
IT      Boston      Soo           Nguyen       83000
IT      Boston      Janet         King         95000
```

但來的東西卻不如預期。相反地，該機構送來一份從薪資系統中傾卸出來的資料：成打的 CSV 檔案，每個都代表資料庫中的某個資料表。你讀過解釋資料佈局的文件（記得一定須要到這類文件！），也了解了每個資料表欄位的含義。其中兩個符合你的需求：一個叫做 `employees`，另一個則是 `departments`。

利用清單 6-1 的程式碼，我們可以建立這些資料表、插入資料值、並檢視如何結合兩者的資料。請用先前建立的 `analysis` 資料庫來練習和執行程式碼，然後用基本的 `SELECT` 陳述、或是在 pgAdmin 裡點選資料表名稱，再選取 **View/Edit Data ▸ All Rows** 來觀察。

清單 6-1：建立 `departments` 和 `employees` 兩個資料表

```
CREATE TABLE departments (
    dept_id bigserial,
    dept varchar(100),
    city varchar(100),
 ❶ CONSTRAINT dept_key PRIMARY KEY (dept_id),
 ❷ CONSTRAINT dept_city_unique UNIQUE (dept, city)
);

CREATE TABLE employees (
    emp_id bigserial,
    first_name varchar(100),
    last_name varchar(100),
    salary integer,
 ❸ dept_id integer REFERENCES departments (dept_id),
 ❹ CONSTRAINT emp_key PRIMARY KEY (emp_id),
 ❺ CONSTRAINT emp_dept_unique UNIQUE (emp_id, dept_id)
);

INSERT INTO departments (dept, city)
VALUES
```

結合資料表的查詢語法，與基本的 SELECT 類似。差別只在於查詢中還要指定以下事項：

- 利用 SQL 的 JOIN ... ON 陳述指定要結合的資料表和欄位
- 利用關鍵字 JOIN 的變化來進行不同方式的結合

我們先來看一下 JOIN ... ON 的整體語法，然後再研究各種結合方式。要結合範例中的 employees 和 departments，並觀察兩者當中所有相關的資料，先寫出像是清單 6-2 裡的查詢：

清單 *6-2*：結合 employees 和 departments 資料表

```
❶ SELECT *
❷ FROM employees JOIN departments
❸ ON employees.dept_id = departments.dept_id;
```

範例裡，你在 SELECT 陳述中使用了星號萬用字元，藉以挑出兩個資料表的所有欄位 ❶。接著關鍵字 JOIN ❷ 會放在來源的兩個資料表中間。最後，你要以關鍵字 ON 指定用來結合資料表的欄位 ❸。每個資料表的欄位都必須以資料表名稱加上點號，再加上含有鍵值的欄位名稱來指定。兩個資料表欄位名稱之間則是一個等號。

當你執行以上查詢時，結果就會含有來自兩個資料表的全部欄位，而且兩邊的 dept_id 欄位值是一致的。事實上，dept_id 區塊還出現了兩次，因為你是挑選兩者的全部欄位：

```
emp_id   first_name    last_name    salary   dept_id   dept_id   dept   city
------   ----------   ---------    ------   -------   -------   ----   -------
     1   Nancy        Jones        62500          1         1   Tax    Atlanta
     2   Lee          Smith        59300          1         1   Tax    Atlanta
     3   Soo          Nguyen       83000          2         2   IT     Boston
     4   Janet        King         95000          2         2   IT     Boston
```

所以就算是資料分散在兩個資料表裡，而且兩者各自都有自己的一群欄位，各位仍然可以同時查詢兩者，並把相關聯的資料一起拉出來。在「在

為了說明起見，清單 7-6 展示了兩個虛構資料庫的資料表，以便追蹤車輛活動：

清單 *7-6*：外部鍵值範例

```
CREATE TABLE licenses (
    license_id varchar(10),
    first_name varchar(50),
    last_name varchar(50),
  ❶ CONSTRAINT licenses_key PRIMARY KEY (license_id)
);

CREATE TABLE registrations (
    registration_id varchar(10),
    registration_date date,
  ❷ license_id varchar(10) REFERENCES licenses (license_id),
    CONSTRAINT registration_key PRIMARY KEY (registration_id, license_id)
);

❸ INSERT INTO licenses (license_id, first_name, last_name)
  VALUES ('T229901', 'Lynn', 'Malero');

❹ INSERT INTO registrations (registration_id, registration_date, license_id)
  VALUES ('A203391', '3/17/2017', 'T229901');

❺ INSERT INTO registrations (registration_id, registration_date, license_id)
  VALUES ('A75772', '3/17/2017', 'T000001');
```

第一個 licenses 資料表，跟我們先前製作的 natural_key_example 資料表很像，也以駕駛人獨有的 license_id ❶ 為自然型主要鍵值。第二個 registrations 資料表，則是用來追蹤車輛註冊的。單獨一個駕照號碼可能可以對應到多筆車輛註冊記錄，因為一名有照駕駛人有可能會在數年之間有過幾次換車記錄。此外，單一車輛也可能註冊給多位有照駕駛人，從而建立各位在第 6 章學過的多對多對應關係。

以下是 SQL 表達關聯的方式：在宣告 registrations 資料表時，我們替 license_id 欄位加上了關鍵字 REFERENCES，再加上它參照的資料表名稱和欄位，藉此將之指定為外部鍵值 ❷。

```
    first_name varchar(50),
    last_name varchar(50),
    email varchar(200),
  ❶ CONSTRAINT email_unique UNIQUE (email)
);

INSERT INTO unique_constraint_example (first_name, last_name, email)
VALUES ('Samantha', 'Lee', 'slee@example.org');

INSERT INTO unique_constraint_example (first_name, last_name, email)
VALUES ('Betty', 'Diaz', 'bdiaz@example.org');

INSERT INTO unique_constraint_example (first_name, last_name, email)
❷ VALUES ('Sasha', 'Lee', 'slee@example.org');
```

在這個資料表裡，contact_id 扮演了代理型主要鍵值，可以識別出每一筆資料的獨特性。但我們還有一個 email 欄位，是每個人的聯絡重點。我們可以預料這個欄位裡一定只會有獨特的電子信箱郵址，但這些郵址卻可能隨時變化。因此我們引用了 UNIQUE ❶ 來保證，當任何時候我們新增或更新任何人的電子信箱郵址時，都不會輸入一筆已經存在的資料。如果我們嘗試插入一欄已經存在的電子信箱郵址 ❷，資料庫就會拋回錯誤：

```
ERROR:  duplicate key value violates unique constraint "email_unique"
DETAIL:  Key (email)=(slee@example.org) already exists.
```
譯註 4

錯誤訊息再一次證明資料庫確實有在替我們把關。

約束條件 NOT NULL

在第 6 章裡，各位已經學過 NULL 這個特殊的 SQL 資料值，它代表該筆資料中的某欄位沒有內容、或是狀況不明。各位也已經知道主要鍵值中是不允許有 NULL 出現的，因為主要鍵值必須要能獨一無二地識別資料表中的每一筆資料。但除了主要鍵值以外，其實還有別的欄位也會不希望有空資

譯註 4　錯誤內容為：重複的鍵值違背了獨特性約束條件 "email_unique"。
詳情：鍵值 (email)=(slee@example.org) 已經存在。

max() 和 min() 的用法類似：你要以 SELECT 陳述加上函式名稱，並附上要處理的欄位名稱。清單 8-6 就使用了 max() 和 min() 來計算 2014 年資料表中的 visits 欄位。visits 欄位記錄了某圖書館機構及其分館的當年度造訪人數。執行程式碼，我們再來研究其輸出。

清單 *8-6*：用 max() 和 min() 找出造訪人數最多和最少的圖書館

```
SELECT max(visits), min(visits)
FROM pls_fy2014_pupld14a;
```

查詢結果如下：

```
max         min
--------    ---
17729020     -3
```

嗯，有點意思。最大值超過了 1 千 7 百 70 萬人次，對於大都會圖書館系統來說是很合理的人數，但最小值是 -3 ？表面上看起來這像是一個錯誤，但顯然圖書館調查的作者採用了一個有問題、但很常見的資料採集慣例：利用負數或某種人為的高數值來作為指標。

在本例中，調查設計者採用負數來代表以下狀況：

1. 資料值 -1 代表該問題「未作答」。
2. 資料值 -3 代表「不適用」，通常用於該圖書館正值休館期間、或已永久性閉館。

當我們在探索資料時，必須知道這種負值資料的存在，並加以排除，因為把帶有負值的欄位納入總和計算，會導致不正確的結果。我們可以用 WHERE 子句將其過濾掉。此時發現這一點是件好事，總比已經深入分析後才發覺要好吧！

當你執行查詢時，其結果頂端顯示出有 10 個州或地區，在 2009 到 2014 年的圖書館造訪人數其實是上升的。其他則呈現下跌。敬陪末座的奧克拉荷馬州，跌幅甚至高達百分之 35 ！

```
stabr    visits_2014    visits_2009    pct_change
-----    -----------    -----------    ----------
GU            103593          60763         70.49
DC           4230790        2944774         43.67
LA          17242110       15591805         10.58
MT           4582604        4386504          4.47
AL          17113602       16933967          1.06
AR          10762521       10660058          0.96
KY          19256394       19113478          0.75
CO          32978245       32782247          0.60
SC          18178677       18105931          0.40
SD           3899554        3890392          0.24
MA          42011647       42237888         -0.54
AK           3486955        3525093         -1.08
ID           8730670        8847034         -1.32
NH           7508751        7675823         -2.18
WY           3666825        3756294         -2.38
--snip--
RI           5259143        6612167        -20.46
NC          33952977       43111094        -21.24
PR            193279         257032        -24.80
GA          28891017       40922598        -29.40
OK          13678542       21171452        -35.39
```

這份有用的資料應該會讓分析師引往另一個方向，調查是什麼原因造成這樣的變化，尤其是變化最劇烈的州。資料分析所帶來的新疑問有時跟它的答案一樣多，但這也是過程的一部分。這時致電一位熟知內情的人士，更能探索出結果的來龍去脈。有時他們可以提出合情合理的解釋。有時專家則會說「這不對勁啊。」這時你就會回頭去找資料供應人或是文件，看看你是否忽略了資料中的某個代碼或是細微線索。

在清單 9-4 裡，我們利用了「用 NULL 找出內容從缺的資料列」一節中介紹過的技巧，並對 st 欄位加上 WHERE 子句和關鍵字 IS NULL，藉此找出哪幾筆資料的州別代碼是從缺的：

清單 *9-4*：使用 IS NULL 找出沒有資料的 st 欄位

```
SELECT est_number,
       company,
       city,
       st,
       zip
FROM meat_poultry_egg_inspect
WHERE st IS NULL;
```

查詢傳回三筆資料，它們的 st 欄位確實空無一物：

```
est_number          company                          city      st    zip
-----------------   ------------------------------   ------    --    -----
V18677A             Atlas Inspection, Inc.           Blaine          55449
M45319+P45319       Hall-Namie Packing Company, Inc                  36671
M263A+P263A+V263A   Jones Dairy Farm                                 53538
```

如果我們想精確地計算每一州的業者數目，這些錯失的資料值就會造成不正確的結果。為了判斷糟糕資料的起源，應該儘快檢視一下從 *https://www.data.gov/* 下載的原始檔案。除非你的檔案範圍高達 gigabyte 等級，不然你應該可以用文字編輯器打開 CSV 檔，用肉眼逐列檢視。如果檔案太大，也許就該改用 grep（在 Linux 和 macOS 上）或是 findstr（在 Windows 上）之類的工具檢視原始檔。在上例中，我們確實證實了這一點，CSV 檔中這幾筆資料的確沒有州別，因此錯誤源於資料本質不佳，而非匯入時造成的錯誤。

資料訪查到此，我們已經得知需要把漏失的州別放到 st 欄位裡，以便清理資料表。我們這就來看看資料中還有哪些問題存在，並列出需要清理的作業項目。