序三

如果說《啊哈！圖解演算法》是演算法界的入門書，內容太少看得不過癮，那麼這本《演算法的樂趣》或許可以帶你一起進階一起飛。當我剛拿到書的目錄的時候，我就很期待，因為終於有一本演算法書可以有系統地和大家談一談這些我也很想分享的偉大演算法。

暴力盲目的搜尋演算法往往讓電腦顯得很笨甚至有點癡呆，如果你想設計一個「狡猾」的程式，那麼本書中的搜尋剪枝、A*尋徑、賽局樹以及遺傳演算法等將給你帶來啟發。快速傅立葉變換，這麼霸氣而高深的名字，其實在我們生活中的應用隨處可見，家中的Wi-Fi、智慧手機、電話、路由器等幾乎所有內置電腦系統的東西都會以各種方式使用這個演算法。RLE 資料壓縮演算法，在文件、視訊、音樂、資料儲存、雲端計算、資料庫等幾乎所有應用中都有著廣泛的運用。壓縮演算法令系統更有效，成本更低。再來說密碼學演算法中非常重要的 RSA 演算法，如果沒有這些演算法，網際網路就會變得不安全，電子交易就不會如此可信。

好玩的演算法還有很多很多，曆法與二十四節氣的計算、華容道、井字棋、黑白棋、五子棋以及俄羅斯方塊……你會驚訝地發現，再簡單不過的事情背後，都藏著演算法的神奇背影。不妨將本書放在案頭慢慢品讀，你將能看到演算法如何深入我們的日常生活，如何重塑我們的世界。

你準備好了嗎？接下來，這個世界演算法將接管一切。

啊哈磊
《啊哈！演算法》作者

碁峰 www.gotop.com.tw

前言

程式設計師與演算法，這是一個永恆的話題，無論在哪個網路論壇，只要出現此類主題的帖子，一定會看到兩種針鋒相對的觀點的「激烈碰撞」。其實泡過論壇的人都知道，兩種觀點「激烈辯論」的慘烈程度往往可以上升到互相問候先人的高度，即使是技術論壇也不例外。在準備此書之前，我在部落格的「演算法系列」專欄已經陸陸續續地寫了有一年多的時間，在此期間，不斷有讀者問我：「程式設計師必須會演算法嗎？」我實在不想讓我的部落格成為噴滿各種口水的是非之地，所以一般不正面回答，只是籠統地說些「各行各業情況都不盡相同」之類的話，避免爭議。

程式設計師對演算法通常懷有複雜的感情，演算法很重要是大家的共識，但是是否每個程式設計師都必須學演算法是主要的分歧點。本書是想重新定義程式設計師對演算法的理解，並不想透過說教的方式給出到底是學還是不學的結論。很多人可能覺得像人工智慧、視訊與音訊處理以及資料搜尋與挖掘這樣高大上的內容才能稱為演算法，往往覺得演算法深不可測。但是這些其實都不是具體的演算法，而是一系列演算法的集合，這裡面既有各種大名鼎鼎的演算法，比如神經網路、遺傳演算法、離散傅立葉變換演算法以及各種插值演算法，也有不起眼的排序和機率計算的演算法。你必須深入地瞭解它們，才會領略到演算法的實質——解決問題。忽視這一點，片面地或抽象地理解演算法，就會使對演算法的理解變得形而上學。在我的部落格裡就有人留言質疑：「窮舉也算是演算法？」且不說搜尋和列舉是演算法的基礎設計模式之一，單就那麼多的 NPC 問題（比如著名的漢密爾頓迴路問題，至今還沒有找到多項式時間的演算法），實際上，從只有窮舉演算法和其他隨機搜尋演算法才能求解這一點看，任何人都不能小看它。

狹隘的演算法定義會將自己局限在一個小角落裡，因而錯過了整個色彩繽紛的演算法世界。本書將帶你開啟一段演算法之旅，在這裡，你將會看到各種建構演算法的基礎方法，比如貪婪法、分治法、動態規劃法…等等，也可以透過一個個示例看到如何應用這些演算法來解決實際問題。透過對「愛因斯坦的思考題」、「三個水桶等分水」、「妖怪與和尚過河問題」等趣味智力題的電腦求解演算法設計，你可以領會到演算法設計的三個關鍵問題，以及對這些問題的處理方法，為以後解決這樣的問題提供舉一反三的基礎。

在生活中，凡是有樂趣的地方就有演算法。本書將介紹生活中無處不在的演算法。在曆法計算的章節裡，你會看到霍納法則（Horner’s rule）的使用和求解一元高次方程的牛頓反覆運算法；音訊播放機上跳動的頻譜，背後是離散傅立葉變換演算法；DOS 時代著名的 PCX 影像檔格式使用的 RLE 壓縮演算法是如此簡單，但是卻非常有效；RSA 加密演算法的光環之下是樸實的歐幾里得演算法、蒙哥馬利演算法和米勒-拉賓演算法；華容道

碁峰 www.gotop.com.tw

遊戲求解的簡單窮舉演算法中還蘊藏著對棋盤狀態的雜湊演算法……遺傳演算法神秘不可測，但是用遺傳演算法求解 0-1 背包問題只用了 60 多行程式碼。事實上，拋開對遺傳演算法的深層次研究和在各種專業領域內的擴展應用，單就演算法原理來說，它就是這麼簡單。深藍戰勝卡斯楚之後，人類棋手在與電腦的博弈中就完全處於下風，人工智慧真的這麼神奇？人工智慧確實是個神奇的領域，但就電腦下棋這件事來說，卻並不怎麼神奇，演算法的基本原理簡單得讓人難以置信，看看第 23 章你就知道了。

演算法之大，大到可以囊括宇宙萬物的運行規律，演算法之小，小到寥寥數行程式碼即可展現一個神奇的功能。演算法是瑣碎的，以至於常常被人們忽視，然而忽視演算法能力的培養所帶來的代價是巨大的，第 1 章介紹的環形佇列的例子就是一個最好的說明。我面試過很多求職者，我常常會讓他們手寫一個演算法，我的題目是這樣的：有一個由若干正整數組成的數列，數列中的每個數都不超過 32，已知數列中存在重複的數字，請給出一個演算法找出這個數列中所有重複出現的數。我期望求職者給我一個正確的演算法實作，接下來我會問這個演算法的時間複雜度是什麼，有沒有考慮過存在一個 $O(n)$ 時間複雜度的演算法。大部分求職者都知道自己的演算法是 $O(n^2)$ 時間複雜度，但是都否認存在 $O(n)$ 時間複雜度的演算法。事實上這個題目是可以有 $O(n)$ 時間複雜度的演算法的，因為大家都忽略了一個重要的條件。這個題目並不難，但是仍有將近三分之一的面試者無法給出正確的演算法，有的甚至還給我一張白紙。有人犯錯誤是正常現象，但是讓我意外的是居然有三分之一的人寫不出這個演算法，演算法設計的基本功被無視到這種地步是不正常的。

程式設計師談到演算法言必稱一些深奧的詞彙，但是這些專有名詞大部分人是用不到的，以至於人們常常認為演算法不過如此，不會又如何？這種思維變得極端就會讓人忽視演算法的基礎設計能力，這才是最要命的。在我們維護的網路裝置上，使用者的資料關係錯綜複雜，一個對線性串列進行二重迴圈都想不到的人又怎麼可能會維護這些資料？我希望程式設計師們提高基礎的演算法能力，先從培養興趣開始或許是一個不錯的切入點。

本書挑選的演算法例子，都圍繞著「趣」字展開，都是簡單且在生活中常見的演算法，可能有些是你還沒有意識到的。我上學的時候曾經做過一個 MP3 播放機程式，你可能覺得這主要就是利用一些音訊解碼演算法吧？是的，這個是主要部分，但是一個功能完整的播放程式還用了很多你想不到的演算法：為增加頻譜顯示和等化器功能，使用了離散傅立葉變換演算法；為計算頻率功率譜，使用了加權平均值演算法；為了匹配硬體輸出裝置與解碼演算法的性能差異，需要一個有多個緩衝區的佇列管理音訊資料塊，這就引入了滑動視窗演算法；為提供按照專輯名稱或作者名稱排序功能，使用了快速排序演算法；為了平滑等化器調節對音訊的影響，使用了三次樣條曲線插值演算法；為了在兩首

碁峰 www.gotop.com.tw

歌曲之間切換時壓制刺耳的雜音（透過填滿一些舒適雜訊的方式實作），還使用了正弦信號發生器演算法。這些你都沒有想到吧？其實還有更多的例子，比如大型專案管理軟體中的工作節點排序功能和關鍵路徑功能，背後支撐它們的卻是簡單的有向圖拓撲排序演算法。這是不是很有趣？生活中處處都是演算法，程式設計師又怎麼可能與演算法絕緣？

再次重申一點，本書沒有任何關於演算法重要性的說教，當你看到本書時，我希望你的表情是「啊哈，原來如此！」，或者是「嗯，有意思！」，並從中獲得樂趣。

本書幾乎所有章節都有相關演算法實作和功能的程式碼，讀者可以到下列網址下載：

繁體版範例程式碼：http://books.gotop.com.tw/download/ACL045600

簡體版作者部落格：http://blog.csdn.net/orbit/

碁峰 www.gotop.com.tw

第 8 章
愛因斯坦的思考題

這是一個很有趣的邏輯推理題，傳說是愛因斯坦提出來的，他宣稱世界上只有 2%的人能解出這個題目。傳說不一定屬實，但是這個推理題還是很有意思的。題目是這樣的，據說有五個不同顏色的房間排成一排，每個房間裡分別住著一個不同國籍的人，每個人都喝一種特定品牌的飲料，抽一種特定品牌的煙，養一種寵物，沒有任意兩個人抽相同品牌的香煙，或喝相同品牌的飲料，或養相同的寵物。問題是誰在養魚作為寵物？為了尋找答案，愛因斯坦列出了以下 15 條線索。

1. 英國人住在紅色的房子裡。
2. 瑞典人養狗作為寵物。
3. 丹麥人喝茶。
4. 綠房子緊靠著白房子，在白房子的左邊。
5. 綠房子的主人喝咖啡。
6. 抽 Pall Mall 牌香煙的人養鳥。
7. 黃色房子裡的人抽 Dunhill 牌香煙。
8. 住在中間那個房子裡的人喝牛奶。
9. 挪威人住在第一個房子裡面。

碁峯
www.gotop.com.tw

10. 抽 Blends 牌香煙的人和養貓的人相鄰。

11. 養馬的人和抽 Dunhill 牌香煙的人相鄰。

12. 抽 BlueMaster 牌香煙的人喝啤酒。

13. 德國人抽 Prince 牌香煙。

14. 挪威人和住在藍房子的人相鄰。

15. 抽 Blends 牌香煙的人和喝礦泉水的人相鄰。

8.1 問題的答案

一般人很難同時記住這麼多線索，所以解決這個問題需要用紙和筆劃一些表格，一步一步慢慢推理，必要時需要一些假設進行嘗試，如果假設錯誤就推倒重來。我缺乏耐心去做這個事情，所以我一直解不出這個問題。直到有一天，我的一個聰明的朋友告訴我一個答案。我對比了一下前面提到的 15 條線索，發現這是一個正確答案。答案是住在綠色房子裡的德國人養魚作為寵物，完整的推理結果如表 8-1 所示。

我是個懶人，知道了這個問題的答案也就算了，但是我的朋友追問我一個問題，讓我不得不正視這個問題。我的朋友想知道這個問題的答案是否唯一，會不會有另外的人推導出另一個完全不同的答案。我想來想去也沒有好的辦法證明這個問題是否還有其他答案，又懶得自己推理這個問題，只好勞駕任勞任怨的電腦來做這個事情。

表 8-1 愛因斯坦思考題推理結果

房子顏色	國　籍	飲　料	寵　物	煙
黃色	挪威	水	貓	Dunhill
藍色	丹麥	茶	馬	Blends
紅色	英國	牛奶	鳥	PallMall
綠色	德國	咖啡	魚	Prince
白色	瑞士	啤酒	狗	BlueMaster

碁峯
www.gotop.com.tw

8.2　分析問題的數學模型

整個問題的描述分成兩部分，一部分是對問題基本結構的描述，比如每個人住一種顏色的房子，抽一種牌子的香煙，喝一種飲料...等等。另一部分是對線索的描述，比如英國人住在紅色的房子中。如果說基本資料結構只是定義了推理結果的一個框架，則線索就可理解為不同屬性之間的綁定關係，用來填入基本結構。因此，對本問題的建模也分成兩個部分，一部分是基本模型定義，另一部分是線索模型定義。

8.2.1　基本模型定義

這個問題的描述比較複雜，總結起來共有 5 種顏色的房子、5 種國籍、5 種飲料、5 種寵物和 5 種牌子的香煙，如何用一個數學模型同時表達這 25 個屬性呢？這 25 個屬性分成 5 種類別，仔細觀察這些屬性，會發現每個屬性都可以用「類型+值」二元組來描述。舉個例子，房子顏色是個類型，黃色就是值，組合成「黃色房子」就是一個屬性。我們首先將屬性的資料結構定義為：

```
typedef struct tagItem
{
    ITEM_TYPE type;
    int value;
}ITEM;
```

ITEM_TYPE 是個列舉類型的量，可以是房子顏色、國籍、飲料類型、寵物類型和香煙牌子五種類型之一，value 是 type 對應的值。value 的取值範圍是 0～4，根據 type 的不同，0～4 代表的意義也不相同。如果 type 對應的是房子顏色，則 value 取值 0～4 分別代表藍色、紅色、綠色、黃色和白色，如果 type 對應的是飲料類型，則 value 取值 0～4 分別代表茶、水、咖啡、啤酒和牛奶。

如果任由這 25 個屬性離散存在，會給設計演算法帶來困難，一般演算法建模都會用各種資料結構將這些屬性組織起來。觀察一下表 8-1 列出的推理結果，我們發現這 25 個屬性在兩個維度上都存在關係，可以按照類型組織，也可以按照同一推理之間的關係組織，是一個矩陣式關係。根據題目描述，每個人住在一種顏色的房子中，喝一種飲料，養一種寵物，抽一種牌子的香煙，這些關係是固定的，一個人不會同時養兩種寵物或喝兩種飲料。我們將這種固定的關係稱為組（group），一個組中包含一種顏色的房子、一個國籍的人、一種飲料、一種寵物和一種牌子的香煙，他們之間的關係是固定的。既然是這樣，可以將 group 資料結構設計為：

碁峰 www.gotop.com.tw

```
typedef struct tagGroup
{
    ITEM items[GROUPS_ITEMS];
}GROUP;
```

這樣的設計中規中矩，但是會為演算法實作帶來麻煩，存取每種屬性都要走訪 items，透過每個 items 的 type 屬性確定要存取的類型。比如要查詢或設定房子的顏色，需要走訪 items，找到 items[i].type== type_house 的那個屬性進行操作。

在本書中我們多次提到在設計資料結構和演算法是利用陣列足標的技巧，這裡又是一個例子。考慮到上面的麻煩，需要修改 GROUP 的設計，不妨將每種類型在 GROUP 中的位置固定，然後直接利用資料足標進行存取。比如將房子顏色類型固定為陣列第一個元素，將國籍固定為陣列第二個元素，以此類推，這樣 GROUP 定義中可以不需要屬性的類型資訊（類型資訊已經由陣列足標表達），只需要一個值資訊即可：

```
typedef struct tagGroup
{
    int itemValue[GROUPS_ITEMS];
}GROUP;
```

與此同時，需要對 ITEM_TYPE 列舉類型做值綁定，以便和陣列足標對應，綁定值如下：

```
typedef enum tagItemType
{
    type_house = 0,
    type_nation = 1,
    type_drink = 2,
    type_pet = 3,
    type_cigaret = 4
}ITEM_TYPE;
```

使用這種定義資料結構的方式，不僅可以減少設計演算法實作的麻煩，還可以提高演算法執行效率。比如現在要查看一個 GROUP 綁定組中房子的顏色是否是藍色，就可以這樣編寫程式碼：

```
if(group.itemValue[type_house] == COLOR_BLUE)
```

8.2.2 線索模型定義

接下來考慮一下如何對線索建立數學模型。線索模型的意義在於判斷一個列舉結果是否正確，如果某個列舉結果能夠符合全部 15 條線索，那這個結果就是最終的正確結果。因此，線索資料結構的定義非常關鍵，如果定義不好，不僅演算法實作會遇

碁峰 www.gotop.com.tw

到很大的麻煩，而且影響演算法實作的效率。即使最後設計出了演算法實作，也是到處都是長長的 if…else 分支，本書中多次強調，程式碼中長長的 if…else 分支結構意味著出現了不良設計。

先分析一下這 15 條線索，大致可以分成三類：第一類是描述某些屬性之間具有固定綁定關係的線索，比如，「丹麥人喝茶」和「住綠房子的人喝咖啡」...等等，線索 1、2、3、5、6、7、12、13 可歸為此類；第二類是描述某些屬性類型所在的「組」所具有的相鄰關係的線索，比如，「養馬的人和抽 Dunhill 牌香煙的人相鄰」和「抽 Blends 牌香煙的人和養貓的人相鄰」...等等，線索 10、11、14、15 可歸為此類；第三類就是不能描述屬性之間固定關係或關係比較弱的線索，比如，「綠房子緊靠著白房子，在白房子的左邊」和「住在中間那個房子裡的人喝牛奶」...等等。

對於第一類具有綁定關係的線索，其數學模型可以這樣定義：

```
typedef struct tagBind
{
    ITEM_TYPE first_type;
    int first_val;
    ITEM_TYPE second_type;
    int second_val;
}BIND;
```

first_type 和 first_val 是一個綁定關係中前一個屬性的類型和值，second_type 和 second_val 是綁定關係中後一個屬性的類型和值。以線索 6：「綠房子的主人喝咖啡」為例，first_type 就是 type_house，first_val 就是 COLOR_GREEN（COLOR_GREEN 是個整數型常數），second_type 就是 type_drink，second_val 就是 DRINK_COFFEE（DRINK_COFFEE 是個整數型常數）。線索 1、2、3、5、6、7、12、13 就可以儲存在 binds 陣列中：

```
const BIND binds[] =
{
    { type_house, COLOR_RED, type_nation, NATION_ENGLAND },
    { type_nation, NATION_SWEDEND, type_pet, PET_DOG },
    { type_nation, NATION_DANMARK, type_drink, DRINK_TEA },
    { type_house, COLOR_GREEN, type_drink, DRINK_COFFEE },
    { type_cigaret, CIGARET_PALLMALL, type_pet, PET_BIRD },
    { type_house, COLOR_YELLOW, type_cigaret, CIGARET_DUNHILL },
    { type_cigaret, CIGARET_BLUEMASTER, type_drink, DRINK_BEER },
    { type_nation, NATION_GERMANY, type_cigaret, CIGARET_PRINCE }
};
```

對於第二類描述元素所在的「組」具有相鄰關係的線索，其數學模型可以這樣定義：

碁峰 www.gotop.com.tw

```
typedef struct tagRelation
{
    ITEM_TYPE type;
    int val;
    ITEM_TYPE relation_type;
    int relation_val;
}RELATION;
```

type 和 val 是某個「組」內的某個屬性的類型和值，relation_type 和 relation_val 是與該屬性所在的「組」相鄰的「組」中與之有關係的屬性的類型和值。以線索 10「抽 Blends 牌香煙的人和養貓的人相鄰」為例，type 就是 type_cigaret，val 就是 CIGARET_BLENDS（CIGARET_BLENDS 是個整數型常數），relation_type 是 type_pet，relation_val 是 PET_CAT（PET_CAT 是個整數型常數）。線索 10、11、14、15 就可以儲存在 relations 陣列中：

```
const RELATION relations[] =
{
    { type_cigaret, CIGARET_BLENDS, type_pet, PET_CAT },
    { type_pet, PET_HORSE, type_cigaret, CIGARET_DUNHILL },
    { type_nation, NATION_NORWAY, type_house, COLOR_BLUE },
    { type_cigaret, CIGARET_BLENDS, type_drink, DRINK_WATER }
};
```

對於第三類線索，無法建立統一的數學模型，只能在列舉演算法進行過程中直接使用它們過濾掉一些不符合條件的組合結果。比如線索 8「住在中間那個房子裡的人喝牛奶」，就是對每個飲料類型組合結果直接判斷 groups[2].itemValue[type_drink] 的值是否等於 DRINK_MILK，如果不滿足這個線索就不再繼續下一個元素類型的列舉。再比如線索 4「綠房子緊靠著白房子，在白房子的左邊」，就是在對房子類型進行組合排列時，將綠房子和白房子看成一個整體進行排列組合的列舉，得到的結果直接符合了線索 4 的要求。

8.3 演算法設計

和其他窮舉類演算法一樣，本問題的窮舉演算法也包含兩個典型過程，一個是對所有結果的窮舉過程，另一個是對結果的證確定判定過程。這兩個過程的演算法設計與之前的資料結構設計息息相關，本節就分別介紹一下這兩個過程的演算法設計方法。

碁峰 www.gotop.com.tw

8.3.1　窮舉所有的組合結果

前面幾章也多次介紹窮舉法解決問題，但都是一維線性組合的列舉，本題則有些特殊，需要對不同類型的元素分別用窮舉法進行列舉，因此不是簡單的線性組合。這個演算法採用的窮舉方法是對不同類型的元素分別進行列舉，然後再按照組的關係組合在一起，這個組合不是線性關係的組合，而是類似階乘的幾何關係的組合。具體思維就是按照 group 中的元素順序，首先對房子根據顏色組合進行窮舉，每得到一組房子顏色組合後，就在此基礎上對住在房子裡的人的國籍進行窮舉，在房子顏色和國籍的組合結果基礎上，在對飲料類型進行窮舉，以此類推，直到窮舉完最後一種類型得到完整的窮舉組合。

這個演算法和普通的組合窮舉演算法不同，需對五種類型的屬性分別列舉，但每種類型的列舉都有一些特殊情況，如 8.2.2 節描述的第三類線索。這類情況無法統一處理，需在列舉演算法中進行處理。以列舉房子顏色的演算法為例，這裡需要處理線索 4「綠房子緊靠著白房子，在白房子的左邊」這種特殊情況，請看演算法實作：

```
void EnumHouseColors(GROUP *groups, int groupIdx)
{
    if(groupIdx == GROUPS_COUNT) /*遞迴終止條件*/
    {
        ArrangeHouseNations(groups);
        return;
    }

    for(int i = COLOR_BLUE; i <= COLOR_YELLOW; i++)
    {
        if(!IsGroupItemValueUsed(groups, groupIdx, type_house, i))
        {
            groups[groupIdx].itemValue[type_house] = i;
            if(i == COLOR_GREEN) //應用線索(4)：綠房子緊靠著白房子，在白房子的左邊；
            {
                groups[++groupIdx].itemValue[type_house] = COLOR_WHITE;
            }

            EnumHouseColors(groups, groupIdx + 1);
            if(i == COLOR_GREEN)
            {
                groupIdx--;
            }
        }
    }
}
```

這是一個典型的線性列舉，只是在列舉結束的時候繼續呼叫 ArrangeHouseNations() 函數繼續對房間內住的人的國籍進行列舉。既然綠色房子在白色房子左邊，那麼每次

碁峰 www.gotop.com.tw

列舉中只要有綠色房子，就直接將其右邊（表現在資料結構中就是下一個組索引）的組中的房子顏色設定成白色。當然，列舉的範圍就變成從 COLOR_BLUE 到 COLOR_YELLOW 四種顏色，沒有 COLOR_WHITE，因為 COLOR_WHITE 和 COLOR_GREEN 兩種顏色直接做了綁定。

對線索 9「挪威人住在第一個房子裡面」的特殊處理展現在 ArrangeHouseNations() 函數中，請看 ArrangeHouseNations() 函數的實作，非常簡單吧，這就是資料結構設計帶來的便利。

```
void ArrangeHouseNations(GROUP *groups)
{
    /*應用規則(9)：挪威人住在第一個房子裡面；*/
    groups[0].itemValue[type_nation] = NATION_NORWAY;
    EnumHouseNations(groups, 1); /*從第二個房子開始*/
}
```

依次完成 5 種屬性的列舉，就得到一個類似表 8-1 的完整組合結果，一共有多少種這樣的組合結果呢？我們來簡單計算一下。首先是對房子顏色進行窮舉。因為是 5 種顏色的不重複組合，因此應該有 5！= 120 個顏色組合結果，但是根據線索 4「綠房子緊靠著白房子，在白房子的左邊」，相當於綠房子和白房子有穩定的綁定關係，實際就只有 4！= 24 個顏色組合結果。接下來對 24 個房子顏色組合結果中的每一個結果再進行住戶國籍的窮舉，理論上國籍也有 5！= 120 個結果，但是根據線索 9「挪威人住在第一個房子裡面」，相當於固定第一個房子住得人始終是挪威人，因此就只有 4！= 24 個國籍組合結果。窮舉完房子顏色和國籍後就已經有 24×24 = 576 個組合結果了，接下來需要對這 576 個組合結果中的每一個結果再進行飲料類型的窮舉，理論上飲料類型也有 5！= 120 個結果，但是根據線索 8「住在中間那個房子裡的人喝牛奶」，相當於固定了一個飲料類型，因此也只有 4！= 24 個飲料組合類型。窮舉完飲料類型後就得到了 576×24 = 13824 個組合結果，接下來對 13824 個組合結果中的每一個結果再進行寵物種類的窮舉，這一步沒有線索可用，共有 5！= 120 個結果。窮舉完寵物種類後就得到了 13824×20 = 1658880 個組合結果，最後對 1658880 個組合結果中的每一個結果再進行香煙品牌的窮舉，這一步依然沒有線索可用，共有 5！= 120 個結果。窮舉完香煙品牌後就得到了全部組合共 1658880×120 = 199065600 個結果。有將近 2 億個組合結果，看來出現多個正確答案的可能性很大喲。

碁峰 www.gotop.com.tw

8.3.2　利用線索判定結果的正確性

根據 8.2.2 節的分析，一共有三類線索，其中第三類線索已經融入到列舉過程中了，因此判斷結果的正確性只需要用第一類線索和第二類線索進行過濾即可。第一類線索是同一 GROUP 內的屬性之間的綁定關係，用來描述的是一個「組」內兩種屬性之間的固定關係。對這類線索的判斷的方法就是走訪全部的「組」，找到 BIND 資料中的 first_type 和 first_val 標識的屬性所在的組 group，然後檢查 group 組中類型為 second_type 的屬性的值是否等於 second_val。如果 group 中類型為 second_type 對應屬性的值與 second_val 的值不一致就直接返回檢查失敗，否則就說明目前的組合結果滿足此 BIND 資料對應的線索，然後對下一個 BIND 資料重複上述檢查過程，直到檢查完 binds 陣列中所有線索對應的 BIND 資料。圖 8-1 是用綁定關係線索對結果檢查的流程圖。

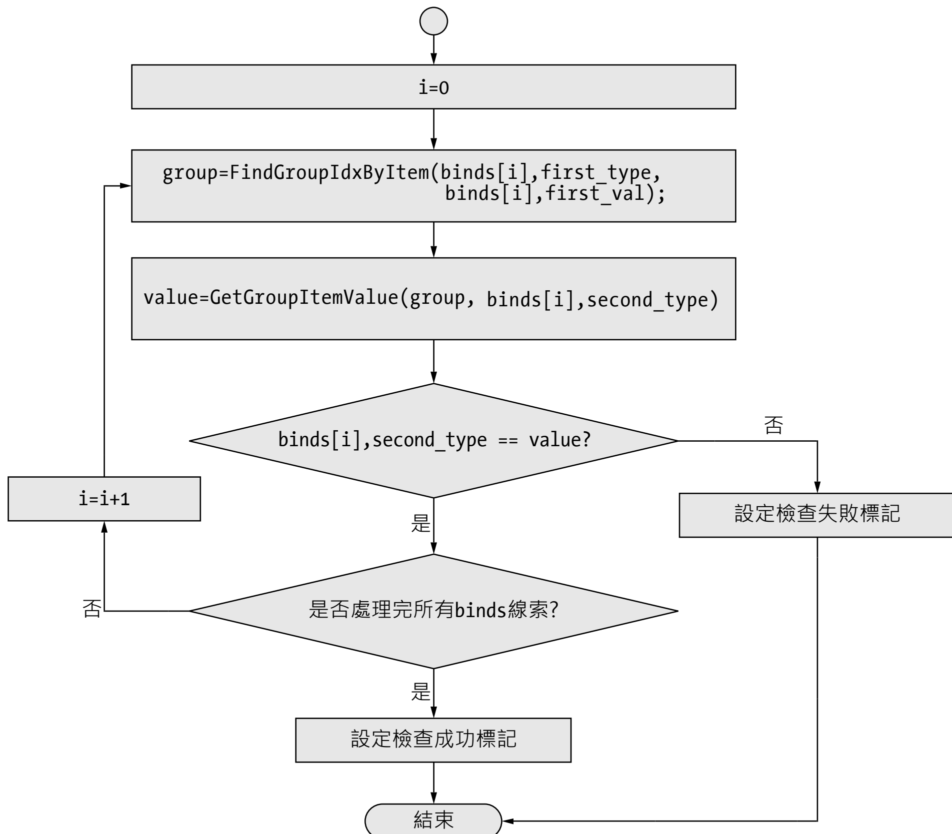

圖 8-1　綁定關係線索檢查的流程圖

碁峯 www.gotop.com.tw

第二類線索是「組」之間的相鄰關係線索，描述的是相鄰的兩個組之間的屬性的固定關係，判斷的方法就是走訪全部的「組」，找到 RELATION 資料中的 type 和 val 標識的元素所在的組 group，然後分別檢查與 group 相鄰的兩個組（第一個組和最後一個組只有一個相鄰的組）中類型為 relation_type 的元素對應的值是否等於 relation_val，如果相鄰的組中沒有一個能滿足 RELATION 資料就表示目前組合結果不滿足線索，直接返回檢查失敗。相鄰的組中只要一個組中的元素滿足 RELATION 資料描述的關係就表示目前組合結果符合 RELATION 資料對應的線索，需要對下一個 RELATION 資料重複上述檢查過程，直到檢查完 relations 陣列中的全部線索對應的 RELATION 資料。圖 8-2 是用「組」相鄰關係對結果檢查的流程圖。

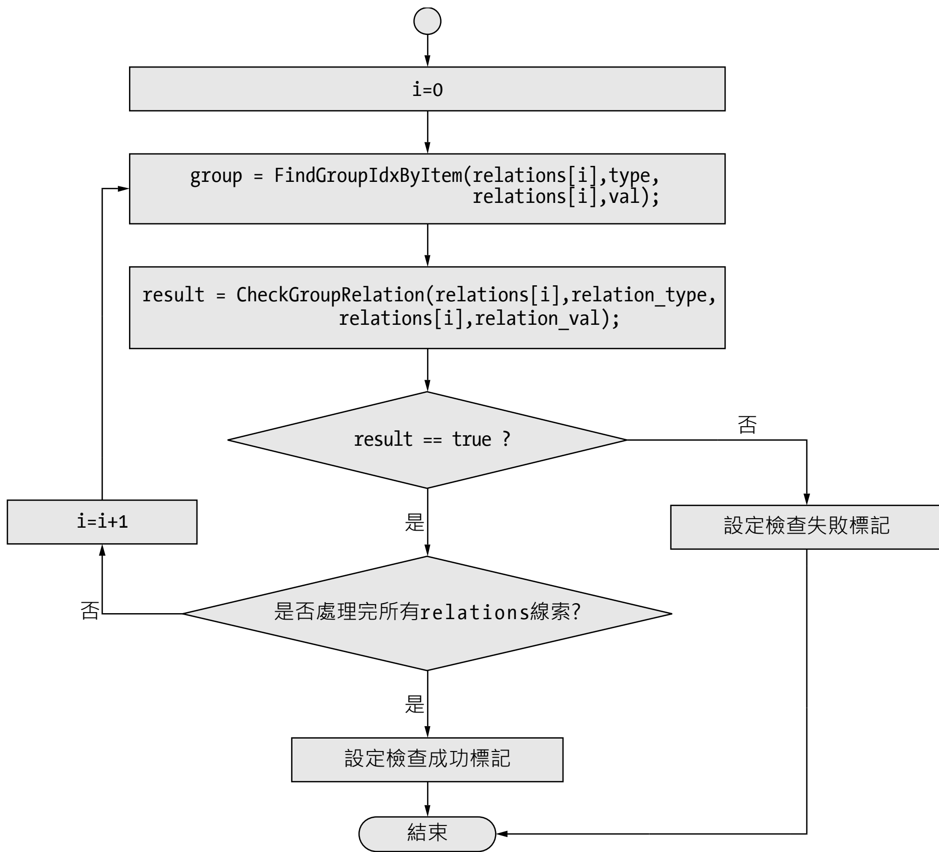

圖 8-2　「組」相鄰關係線索檢查流程圖

碁峯 www.gotop.com.tw

根據圖 8-1 和圖 8-2 所示的流程圖可以看出，對這兩類線索進行檢查的演算法實作非常簡單。得益於我們的資料結構設計，檢查演算法只需要走訪 `binds` 陣列和 `relations` 陣列即可，避免了寫很多 `if…else` 分支。這兩個檢查的具體演算法實作程式碼在本書的配書程式碼中，此處就不再列出。

8.4　總結

雖然有將近 2 億個組合結果，但是令人驚訝的是，竟然只有一組結果能透過所有的線索檢查，就是 8.1 節列出的答案。結果有了，答案真的是唯一的，有點出乎預料，但是也說明了這個問題的難度。

本問題的窮舉演算法是一個比較另類的窮舉演算法，可能因為其結果是二維關係的原因吧，與之前介紹的線性窮舉演算法稍有不同。透過本演算法，大家可以瞭解一下多個維度窮舉的一般方法，就是對每個維度分別窮舉，然後再按照關係組合窮舉結果。

8.5　參考資料

[1] Levitin A. 演算法設計與分析基礎. 潘彥譯. 北京：清華大學出版社，2007

[2] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[3] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

碁峰 www.gotop.com.tw