目錄

> 原作： *mostly-adequate-guide*, thank Professor Franklin Risby for his great work!
>
> llh911001 簡中翻譯版：JS函数式编程指南
>
> 繁體中文版由前簡中翻譯版透過 OpenCC 轉檔而來

關於本書

這本書的主題是函式典範（functional paradigm），我們將使用 JavaScript 這個世界上最流行的函數語言程式設計語言來講述這一主題。有人可能會覺得選擇 JavaScript 並不明智，因爲當前的主流觀點認爲它是一門命令式（imperative）的語言，並不適合用來講函數式。但我認爲，這是學習函數語言程式設計的最好方式，因爲：

- 你很有可能在日常工作中使用它

  這讓你有機會在實際的程式設計過程中學以致用，而不是在空閒時間用一門深奧的函數語言程式設計語言做一些玩具性質的專案。

- 你不必從頭學起就能開始編寫程式

  在純函數語言程式設計語言中，你必須使用 monad 才能列印變數或者讀取 DOM 節點。JavaScript 則簡單得多，可以作弊走捷徑，因爲畢竟我們的目的是學寫純函數式程式碼。JavaScript 也更容易入門，因爲它是一門混合典範的語言，你隨時可以在感覺吃力的時候回退到原有的程式設計習慣上去。

- 這門語言完全有能力書寫高階的函數式程式碼

  只需藉助一到兩個微型類庫，JavaScript 就能模擬 Scala 或 Haskell 這類語言的全部特性。雖然物件導向程式設計（Object-oriented programing）主導著業界，但很明顯這種典範在 JavaScript 裡非常笨拙，用起來就像在高速公路上露營或者穿著橡膠套鞋跳踢踏舞一樣。我們不得不到處使用 `bind` 以免 `this` 不知不覺地變了，語言裡沒有類可以用（目前還沒有），我們還發明瞭各種變通方法來應對忘記呼叫 `new` 關鍵字後的怪異行爲，私有成員只能通過閉包（closure）才能實現，等等。對大多數人來說，函數語言程式設計看起來更加自然。

以上說明，強型別的函數式語言毫無疑問將會成爲本書所示範式的最佳試驗場。JavaScript 是我們學習這種典範的一種手段，將它應用於什麼地方則完全取決於你自己。幸運的是，所有的介面都是數學的，因而也是普適的。最終你會發現你習慣了 swiftz、scalaz、haskell 和 purescript，以及其他各種數學偏向的語言。

Gitbook (更好的閱讀體驗)

- 線上閱讀
- 下載EPUB
- 下載Mobi (Kindle)

目錄

第 1 部分

第 2 部分

未來計劃

- 第 1 部分是基礎知識。這是初版草稿，所以我會及時更正發現的的錯誤。歡迎提供幫助！
- 第 2 部分講述型別類（type class），比如 functor 和 monad，最後會講到到 traversable。我希望能塞進來一些 monad transformer 相關的知識，再寫一個純函式的應用。
- 第 3 部分將開始遊走於程式設計實踐與學院學究之間。我們將學習 comonad、f-algebra、free monad、yoneda 以及其他一些範疇學概念。

第 1 章：我們在做什麼？

介紹

你好，我是 Franklin Risby 教授，很高興認識你。接下來我們將共度一段時光了，因爲我要教你一些函數語言程式設計的知識。好了，關於我就介紹到這裡，你怎麼樣？我希望你已經熟悉 JavaScript 語言了，關於物件導向也有一點點的經驗了，而且自認爲是一個合格的程式設計師。希望你沒有昆蟲學博士學位也能找到並殺死一些臭蟲（bug）。

我並不假設你之前有任何函數語言程式設計相關的知識——我們都知道假設的後果是什麼（譯者注：此處原文是“we both know what happens when you assume”，源自一句名言“When you assume you make an ASS of U and ME”，意思是“讓兩人都難堪”）。但我猜想你在使用可變狀態（mutable state）、無限制副作用（unrestricted side effects）和無原則設計（unprincipled design）的過程中已經遇到過一些麻煩。好了，介紹到此爲止，我們進入正題。

本章的目的是讓你對函數語言程式設計的目的有一個初步認識，對一個程式之所以是函數式程式的原因有一定了解，要不然就會像無頭蒼蠅一樣，不問青紅皁白地避免使用物件——這等於是在做無用功。寫程式碼需要遵循一定的原則，就像水流湍急的時候你需要天文羅盤來指引一樣。

現在已經有一些通用的程式設計原則了，各種縮寫詞帶領我們在程式設計的黑暗隧道里前行：DRY（不要重複自己，don't repeat yourself），高內聚低耦合（loose coupling high cohesion），YAGNI （你不會用到它的，ya ain't gonna need it），最小意外原則（Principle of least surprise），單一責任（single responsibility）等等。

我當然不會囉裡八嗦地把這些年我聽到的原則都列舉出來，你知道重點就行。重點是這些原則同樣適用於函數語言程式設計，只不過它們與本書的主題不十分相關。在我們深入主題之前，我想先通過本章給你這樣一種感覺，即你在敲鍵盤的時候內心就能強烈感受到的那種函數式的氛圍。

一個簡單例子

我們從一個愚蠢的例子開始。下面是一個海鷗程式，鳥群合併則變成了一個更大的鳥群，繁殖則增加了鳥群的數量，增加的數量就是它們繁殖出來的海鷗的數量。注意這個程式並不是物件導向的良好實踐，它只是強調當前這種變數賦值方式的一些弊端。

```
var Flock = function(n) {
  this.seagulls = n;
};

Flock.prototype.conjoin = function(other) {
  this.seagulls += other.seagulls;
  return this;
};

Flock.prototype.breed = function(other) {
  this.seagulls = this.seagulls * other.seagulls;
  return this;
};

var flock_a = new Flock(4);
var flock_b = new Flock(2);
var flock_c = new Flock(0);

var result = flock_a.conjoin(flock_c).breed(flock_b).conjoin(flo
ck_a.breed(flock_b)).seagulls;
//=> 32
```

我相信沒人會寫這樣糟糕透頂的程式。程式碼的內部可變狀態非常難以追蹤，而且，最終的答案還是錯的！正確答案是 `16`，但是因爲 `flock_a` 在運算過程中永久地改變了，所以得出了錯誤的結果。這是 IT 部門混亂的表現，非常粗暴的計算方式。

如果你看不懂這個程式，沒關係，我也看不懂。重點是狀態和可變值非常難以追蹤，即便是在這麼小的一個程式中也不例外。

我們試試另一種更函數式的寫法：

```
var conjoin = function(flock_x, flock_y) { return flock_x + floc
k_y };
var breed = function(flock_x, flock_y) { return flock_x * flock_
y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = conjoin(breed(flock_b, conjoin(flock_a, flock_c)),
breed(flock_a, flock_b));
//=>16
```

很好，這次我們得到了正確的答案，而且少寫了很多程式碼。不過函式巢狀有點讓人費解...（我們會在第 5 章解決這個問題）。這種寫法也更優雅，不過程式碼肯定是越直白越好，所以如果我們再深入挖掘，看看這段程式碼究竟做了什麼事，我們會發現，它不過是在進行簡單的加（ `conjoin` ）和乘（ `breed` ）運算而已。

程式碼中的兩個函式除了函式名有些特殊，其他沒有任何難以理解的地方。我們把它們重新命名一下，看看它們的眞面目。

```
var add = function(x, y) { return x + y };
var multiply = function(x, y) { return x * y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = add(multiply(flock_b, add(flock_a, flock_c)), multi
ply(flock_a, flock_b));
//=>16
```

這麼一來，你會發現我們不過是在運用古人早已獲得的知識：

```
// 結合律（assosiative）
add(add(x, y), z) == add(x, add(y, z));

// 交換律（commutative）
add(x, y) == add(y, x);

// 同一律（identity）
add(x, 0) == x;

// 分配律（distributive）
multiply(x, add(y,z)) == add(multiply(x, y), multiply(x, z));
```

是的，這些經典的數學定律遲早會派上用場。不過如果你一時想不起來也沒關係，多數人已經很久沒複習過這些數學知識了。我們來看看能否運用這些定律簡化這個海鷗小程式。

```
// 原有程式碼
add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a,
flock_b));

// 應用同一律，去掉多餘的加法操作（add(flock_a, flock_c) == flock_a）
add(multiply(flock_b, flock_a), multiply(flock_a, flock_b));

// 再應用分配律
multiply(flock_b, add(flock_a, flock_a));
```

漂亮！除了呼叫的函式，一點多餘的程式碼都不需要寫。當然這裡我們定義 `add` 和 `multiply` 是爲了程式碼完整性，實際上並不必要——在呼叫之前它們肯定已經在某個類庫裡定義好了。

你可能在想“你也太偷換概念了吧，居然舉一個這麼數學的例子”，或者“眞實世界的應用程式比這複雜太多，不能這麼簡單地推理”。我之所以選擇這樣一個例子，是因爲大多數人都知道加法和乘法，所以很容易就能理解數學可以如何爲我們所用。

不要感到絕望，本書後面還會穿插一些範疇學（category theory）、集合論（set theory）以及 lambda 運算的知識，教你寫更加複雜的程式碼，而且一點也不輸本章這個海鷗程式的簡潔性和準確性。你也不需要成爲一個數學家，本書要教給你的程式設計典範實踐起來就像是使用一個普通的框架或者 api 一樣。

你也許會驚訝，我們可以像上例那樣遵循函數式的典範去書寫完整的、日常的應用程式，有著優異效能的程式，簡潔且易推理的程式，以及不用每次都重新造輪子的程式。如果你是罪犯，那違法對你來說是好事；但在本書中，我們希望能夠承認並遵守數學之法。

我們希望去踐行每一部分都能完美接合的理論，希望能以一種通用的、可組合的元件來表示我們的特定問題，然後利用這些元件的特性來解決這些問題。相比命令式（稍後本書將會介紹命令式的精確定義，暫時我們還是先把重點放在函數式上）程式設計的那種“某某去做某事”的方式，函數語言程式設計將會有更多的約束，不過你會震驚於這種強約束、數學性的“框架”所帶來的回報。

我們已經看到函數式的點點星光了，但在眞正開始我們的旅程之前，我們要先掌握一些具體的概念。

第 2 章：一等公民的函式

第 2 章: 一等公民的函式

快速概覽

當我們說函式是"一等公民"的時候，我們實際上說的是它們和其他物件都一樣...所以就是普通公民（坐經濟艙的人？）。函式眞沒什麼特殊的，你可以像對待任何其他資料型別一樣對待它們——把它們存在數組裡，當作引數傳遞，賦值給變數...等等。

這是 JavaScript 語言的基礎概念，不過還是值得提一提的，因爲在 Github 上隨便一搜就能看到對這個概念的集體無視，或者也可能是無知。我們來看一個杜撰的例子：

```
var hi = function(name){
  return "Hi " + name;
};

var greeting = function(name) {
  return hi(name);
};
```

這裡 `greeting` 指向的那個把 `hi` 包了一層的包裹函式完全是多餘的。爲什麼？因爲 JavaScript 的函式是可呼叫的，當 `hi` 後面緊跟 `()` 的時候就會執行並返回一個值；如果沒有 `()`，`hi` 就簡單地返回存到這個變數裡的函式。我們來確認一下：

```
hi;
// function(name){
//  return "Hi " + name
// }

hi("jonas");
// "Hi jonas"
```

`greeting` 只不過是轉了個身然後以相同的引數呼叫了 `hi` 函式而已，因此我們可以這麼寫：

```
var greeting = hi;


greeting("times");
// "Hi times"
```

換句話說，`hi` 已經是個接受一個引數的函數了，爲何要再定義一個額外的包裹函式，而它僅僅是用這個相同的引數呼叫 `hi`？完全沒有道理。這就像在大夏天裡穿上你最厚的大衣，只是爲了跟熱空氣過不去，然後吃上個冰棍。眞是脫褲子放屁多此一舉。

用一個函式把另一個函式包起來，目的僅僅是延遲執行，眞的是非常糟糕的程式設計習慣。（稍後我將告訴你原因，跟可維護性密切相關。）

充分理解這個問題對讀懂本書後面的內容至關重要，所以我們再來看幾個例子。以下程式碼都來自 npm 上的模組包：

```
// 太傻了
var getServerStuff = function(callback){
  return ajaxCall(function(json){
    return callback(json);
  });
};

// 這才像樣
var getServerStuff = ajaxCall;
```

世界上到處都充斥著這樣的垃圾 ajax 程式碼。以下是上述兩種寫法等價的原因：

```
// 這行
return ajaxCall(function(json){
  return callback(json);
});

// 等價於這行
return ajaxCall(callback);

// 那麼，重構下 getServerStuff
var getServerStuff = function(callback){
  return ajaxCall(callback);
};

// ...就等於
var getServerStuff = ajaxCall; // <-- 看，沒有括號哦
```

各位，以上才是寫函式的正確方式。一會兒再告訴你爲何我對此如此執著。

```
var BlogController = (function() {
  var index = function(posts) {
    return Views.index(posts);
  };

  var show = function(post) {
    return Views.show(post);
  };

  var create = function(attrs) {
    return Db.create(attrs);
  };

  var update = function(post, attrs) {
    return Db.update(post, attrs);
  };

  var destroy = function(post) {
    return Db.destroy(post);
  };

  return {index: index, show: show, create: create, update: upda
te, destroy: destroy};
})();
```

這個可笑的控制器（controller）99% 的程式碼都是垃圾。我們可以把它重寫成這樣：

```
var BlogController = {index: Views.index, show: Views.show, crea
te: Db.create, update: Db.update, destroy: Db.destroy};
```

...或者直接全部刪掉，因爲它的作用僅僅就是把檢視（Views）和資料庫（Db）打包在一起而已。

爲何鍾愛一等公民？

好了，現在我們來看看鐘愛一等公民的原因是什麼。前面 `getServerStuff` 和 `BlogController` 兩個例子你也都看到了，雖說新增一些沒有實際用處的間接層實現起來很容易，但這樣做除了徒增程式碼量，提高維護和檢索程式碼的成本外，沒有任何用處。

另外，如果一個函式被不必要地包裹起來了，而且發生了改動，那麼包裹它的那個函式也要做相應的變更。

```
httpGet('/post/2', function(json){
  return renderPost(json);
});
```

如果 `httpGet` 要改成可以丟擲一個可能出現的 `err` 異常，那我們還要回過頭去把“膠水”函式也改了。

```
// 把整個應用裡的所有 httpGet 呼叫都改成這樣，可以傳遞 err 引數。
httpGet('/post/2', function(json, err){
  return renderPost(json, err);
});
```

寫成一等公民函式的形式，要做的改動將會少得多：

```
httpGet('/post/2', renderPost);  // renderPost 將會在 httpGet 中呼叫，想要多少引數都行
```

除了刪除不必要的函式，正確地爲引數命名也必不可少。當然命名不是什麼大問題，但還是有可能存在一些不當的命名，尤其隨著程式碼量的增長以及需求的變更，這種可能性也會增加。

專案中常見的一種造成混淆的原因是，針對同一個概念使用不同的命名。還有通用程式碼的問題。比如，下面這兩個函式做的事情一模一樣，但後一個就顯得更加通用，可重用性也更高：

```
// 只針對當前的部落格
var validArticles = function(articles) {
  return articles.filter(function(article){
    return article !== null && article !== undefined;
  });
};

// 對未來的專案友好太多
var compact = function(xs) {
  return xs.filter(function(x) {
    return x !== null && x !== undefined;
  });
};
```

在命名的時候，我們特別容易把自己限定在特定的資料上（本例中是 `articles`）。這種現象很常見，也是重複造輪子的一大原因。

有一點我必須得指出，你一定要非常小心 `this` 值，別讓它反咬你一口，這一點與物件導向程式碼類似。如果一個底層函式使用了 `this`，而且是以一等公民的方式被呼叫的，那你就等著 JS 這個蹩腳的抽象概念發怒吧。

```
var fs = require('fs');

// 太可怕了
fs.readFile('freaky_friday.txt', Db.save);

// 好一點點
fs.readFile('freaky_friday.txt', Db.save.bind(Db));
```

把 Db 繫結（bind）到它自己身上以後，你就可以隨心所欲地呼叫它的原型鏈式垃圾程式碼了。`this` 就像一塊髒尿布，我儘可能地避免使用它，因爲在函數語言程式設計中根本用不到它。然而，在使用其他的類庫時，你卻不得不向這個瘋狂的世界低頭。

也有人反駁說 `this` 能提高執行速度。如果你是這種對速度吹毛求疵的人，那你還是合上這本書吧。要是沒法退貨退款，也許你可以去換一本更入門的書來讀。

至此，我們才準備好繼續後面的章節。

第 3 章: 純函式的好處

第 3 章：純函式的好處

再次強調“純”

首先，我們要釐清純函式的概念。

> 純函式是這樣一種函式，即相同的輸入，永遠會得到相同的輸出，而且沒有任何可觀察的副作用。

比如 `slice` 和 `splice`，這兩個函式的作用並無二致——但是注意，它們各自的方式卻大不同，但不管怎麼說作用還是一樣的。我們說 `slice` 符合純函式的定義是因爲對相同的輸入它保證能返回相同的輸出。而 `splice` 卻會嚼爛呼叫它的那個陣列，然後再吐出來；這就會產生可觀察到的副作用，即這個陣列永久地改變了。

```
var xs = [1,2,3,4,5];

// 純的
xs.slice(0,3);
//=> [1,2,3]

xs.slice(0,3);
//=> [1,2,3]

xs.slice(0,3);
//=> [1,2,3]


// 不純的
xs.splice(0,3);
//=> [1,2,3]

xs.splice(0,3);
//=> [4,5]

xs.splice(0,3);
//=> []
```

在函數語言程式設計中，我們討厭這種會改變資料的笨函式。我們追求的是那種可靠的，每次都能返回同樣結果的函式，而不是像 `splice` 這樣每次呼叫後都把資料弄得一團糟的函式，這不是我們想要的。

來看看另一個例子。

```
// 不純的
var minimum = 21;

var checkAge = function(age) {
  return age >= minimum;
};


// 純的
var checkAge = function(age) {
  var minimum = 21;
  return age >= minimum;
};
```

在不純的版本中，`checkAge` 的結果將取決於 `minimum` 這個可變變數的值。換句話說，它取決於系統狀態（system state）；這一點令人沮喪，因爲它引入了外部的環境，從而增加了認知負荷（cognitive load）。

這個例子可能還不是那麼明顯，但這種依賴狀態是影響系統複雜度的罪魁禍首（http://www.curtclifton.net/storage/papers/MoseleyMarks06a.pdf ）。輸入值之外的因素能夠左右 `checkAge` 的返回值，不僅讓它變得不純，而且導致每次我們思考整個軟體的時候都痛苦不堪。

另一方面，使用純函式的形式，函式就能做到自給自足。我們也可以讓 `minimum` 成爲一個不可變（immutable）物件，這樣就能保留純粹性，因爲狀態不會有變化。要實現這個效果，必須得建立一個物件，然後呼叫 `Object.freeze` 方法：

```
var immutableState = Object.freeze({
  minimum: 21
});
```

副作用可能包括...

讓我們來仔細研究一下“副作用”以便加深理解。那麼，我們在純函式定義中提到的萬分邪惡的副作用到底是什麼？“作用”我們可以理解爲一切除結果計算之外發生的事情。

“作用”本身並沒什麼壞處，而且在本書後面的章節你隨處可見它的身影。“副作用”的關鍵部分在於“副”。就像一潭死水中的“水”本身並不是幼蟲的培養器，“死”才是生成蟲群的原因。同理，副作用中的“副”是滋生 bug 的溫床。

> 副作用是在計算結果的過程中，系統狀態的一種變化，或者與外部世界進行的可觀察的互動。

副作用可能包含，但不限於：

- 更改檔案系統
- 往資料庫插入記錄
- 傳送一個 http 請求
- 可變資料
- 列印/log
- 獲取使用者輸入
- DOM 查詢
- 訪問系統狀態

這個列表還可以繼續寫下去。概括來講，只要是跟函式外部環境發生的互動就都是副作用——這一點可能會讓你懷疑無副作用程式設計的可行性。函數語言程式設計的哲學就是假定副作用是造成不正當行爲的主要原因。

這並不是說，要禁止使用一切副作用，而是說，要讓它們在可控的範圍內發生。後面講到 functor 和 monad 的時候我們會學習如何控制它們，目前還是儘量遠離這些陰險的函式爲好。

副作用讓一個函式變得不純是有道理的：從定義上來說，純函式必須要能夠根據相同的輸入返回相同的輸出；如果函式需要跟外部事物打交道，那麼就無法保證這一點了。

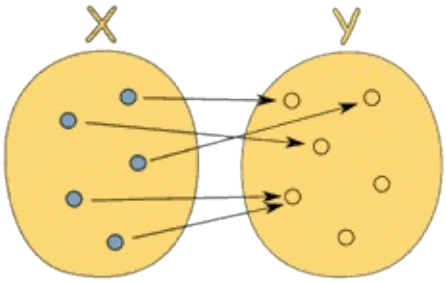
我們來仔細瞭解下爲何要堅持這種「相同輸入得到相同輸出」原則。注意，我們要複習一些八年級數學知識了。

八年級數學

根據 mathisfun.com：

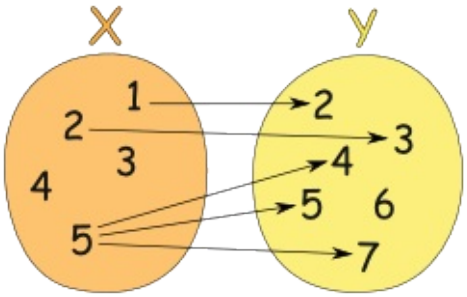
> 函式是不同數值之間的特殊關係：每一個輸入值返回且只返回一個輸出值。

換句話說，函式只是兩種數值之間的關係：輸入和輸出。儘管每個輸入都只會有一個輸出，但不同的輸入卻可以有相同的輸出。下圖展示了一個合法的從 `x` 到 `y` 的函式關係；



（http://www.mathsisfun.com/sets/function.html）

相反，下面這張圖表展示的就不是一種函式關係，因爲輸入值 `5` 指向了多個輸出：



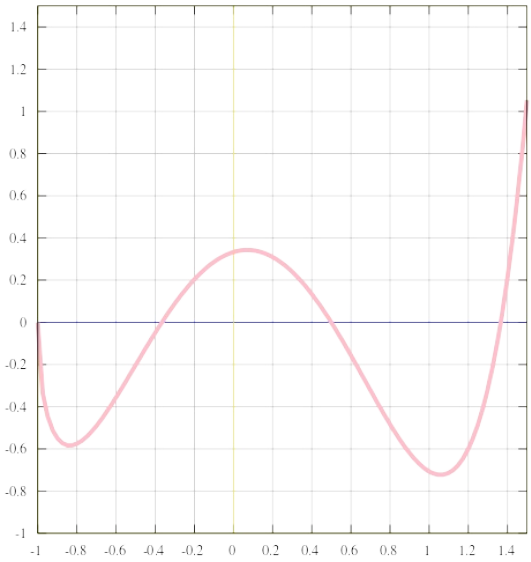
（http://www.mathsisfun.com/sets/function.html）

函式可以描述爲一個集合，這個集合裡的內容是 (輸入, 輸出) 對：`[(1,2), (3,6), (5,10)]`（看起來這個函式是把輸入值加倍）。

或者一張表：

輸入	輸出
1	2
2	4
3	6

甚至一個以 `x` 爲輸入 `y` 爲輸出的函式曲線圖：



如果輸入直接指明瞭輸出，那麼就沒有必要再實現具體的細節了。因爲函式僅僅只是輸入到輸出的對映而已，所以簡單地寫一個物件就能"執行"它，使用 `[]` 代替 `()` 即可。

```
var toLowerCase = {"A":"a", "B": "b", "C": "c", "D": "d", "E": "
e", "D": "d"};

toLowerCase["C"];
//=> "c"

var isPrime = {1:false, 2: true, 3: true, 4: false, 5: true, 6:f
alse};

isPrime[3];
//=> true
```

當然了，實際情況中你可能需要進行一些計算而不是手動指定各項值；不過上例倒是表明了另外一種思考函式的方式。（你可能會想"要是函式有多個引數呢？"。的確，這種情況表明了以數學方式思考問題的一點點不便。暫時我們可以把它們打包放到數組裡，或者把 `arguments` 物件看成是輸入。等學習 `curry` 的概念之後，你就知道如何直接爲函式在數學上的定義建模了。）

戲劇性的是：純函式就是數學上的函式，而且是函數語言程式設計的全部。使用這些純函式程式設計能夠帶來大量的好處，讓我們來看一下爲何要不遺餘力地保留函式的純粹性的原因。

追求“純”的理由

可快取性（Cacheable）

首先，純函式總能夠根據輸入來做快取。實現快取的一種典型方式是 memoize 技術：

```
var squareNumber  = memoize(function(x){ return x*x; });

squareNumber(4);
//=> 16

squareNumber(4); // 從快取中讀取輸入值爲 4 的結果
//=> 16

squareNumber(5);
//=> 25

squareNumber(5); // 從快取中讀取輸入值爲 5 的結果
//=> 25
```

下面的程式碼是一個簡單的實現，儘管它不太健壯。

```
var memoize = function(f) {
  var cache = {};

  return function() {
    var arg_str = JSON.stringify(arguments);
    cache[arg_str] = cache[arg_str] || f.apply(f, arguments);
    return cache[arg_str];
  };
};
```

值得注意的一點是，可以通過延遲執行的方式把不純的函式轉換爲純函式：

```
var pureHttpCall = memoize(function(url, params){
  return function() { return $.getJSON(url, params); }
});
```

這裡有趣的地方在於我們並沒有眞正傳送 http 請求——只是返回了一個函式，當呼叫它的時候才會發請求。這個函式之所以有資格成爲純函式，是因爲它總是會根據相同的輸入返回相同的輸出：給定了 `url` 和 `params` 之後，它就只會返回同一個傳送 http 請求的函式。

我們的 `memoize` 函式工作起來沒有任何問題，雖然它快取的並不是 http 請求所返回的結果，而是生成的函式。

現在來看這種方式意義不大，不過很快我們就會學習一些技巧來發掘它的用處。重點是我們可以快取任意一個函式，不管它們看起來多麼具有破壞性。

可移植性／自文件化（Portable / Self-Documenting）

純函式是完全自給自足的，它需要的所有東西都能輕易獲得。仔細思考思考這一點...這種自給自足的好處是什麼呢？首先，純函式的依賴很明確，因此更易於觀察和理解——沒有偷偷摸摸的小動作。

```
// 不純的
var signUp = function(attrs) {
  var user = saveUser(attrs);
  welcomeUser(user);
};

var saveUser = function(attrs) {
    var user = Db.save(attrs);
    ...
};

var welcomeUser = function(user) {
    Email(user, ...);
    ...
};

// 純的
var signUp = function(Db, Email, attrs) {
  return function() {
    var user = saveUser(Db, attrs);
    welcomeUser(Email, user);
  };
};

var saveUser = function(Db, attrs) {
    ...
};

var welcomeUser = function(Email, user) {
    ...
};
```

這個例子表明，純函式對於其依賴必須要誠實，這樣我們就能知道它的目的。僅從純函式版本的 `signUp` 的簽名就可以看出，它將要用到 `Db`、`Email` 和 `attrs`，這在最小程度上給了我們足夠多的資訊。

後面我們會學習如何不通過這種僅僅是延遲執行的方式來讓一個函式變純，不過這裡的重點應該很清楚，那就是相比不純的函式，純函式能夠提供多得多的資訊；前者天知道它們暗地裡都幹了些什麼。

其次，通過強迫“注入”依賴，或者把它們當作引數傳遞，我們的應用也更加靈活；因爲資料庫或者郵件客户端等等都引數化了（別擔心，我們有辦法讓這種方式不那麼單調乏味）。如果要使用另一個 `Db` ，只需把它傳給函式就行了。如果想在一個新應用中使用這個可靠的函式，儘管把新的 `Db` 和 `Email` 傳遞過去就好了，非常簡單。

在 JavaScript 的設定中，可移植性可以意味著把函式序列化（serializing）並通過 socket 傳送。也可以意味著程式碼能夠在 web workers 中執行。總之，可移植性是一個非常強大的特性。

指令式程式設計中“典型”的方法和過程都深深地根植於它們所在的環境中，通過狀態、依賴和有效作用（available effects）達成；純函式與此相反，它與環境無關，只要我們願意，可以在任何地方執行它。

你上一次把某個類方法拷貝到新的應用中是什麼時候？我最喜歡的名言之一是 Erlang 語言的作者 Joe Armstrong 說的這句話：“面嚮物件語言的問題是，它們永遠都要隨身攜帶那些隱式的環境。你只需要一個香蕉，但卻得到一個拿著香蕉的大猩猩...以及整個叢林”。

可測試性（**Testable**）

第三點，純函式讓測試更加容易。我們不需要僞造一個“眞實的”支付閘道器，或者每一次測試之前都要配置、之後都要斷言狀態（assert the state）。只需簡單地給函式一個輸入，然後斷言輸出就好了。

事實上，我們發現函數語言程式設計的社群正在開創一些新的測試工具，能夠幫助我們自動生成輸入並斷言輸出。這超出了本書範圍，但是我強烈推薦你去試試 *Quickcheck*——一個爲函數式環境量身定製的測試工具。

合理性（**Reasonable**）

很多人相信使用純函式最大的好處是引用透明性（referential transparency）。如果一段程式碼可以替換成它執行所得的結果，而且是在不改變整個程式行爲的前提下替換的，那麼我們就說這段程式碼是引用透明的。

由於純函式總是能夠根據相同的輸入返回相同的輸出，所以它們就能夠保證總是返回同一個結果，這也就保證了引用透明性。我們來看一個例子。

```
var Immutable = require('immutable');

var decrementHP = function(player) {
  return player.set("hp", player.hp-1);
};

var isSameTeam = function(player1, player2) {
  return player1.team === player2.team;
};

var punch = function(player, target) {
  if(isSameTeam(player, target)) {
    return target;
  } else {
    return decrementHP(target);
  }
};

var jobe = Immutable.Map({name:"Jobe", hp:20, team: "red"});
var michael = Immutable.Map({name:"Michael", hp:20, team: "green"
});

punch(jobe, michael);
//=> Immutable.Map({name:"Michael", hp:19, team: "green"})
```

`decrementHP`、`isSameTeam` 和 `punch` 都是純函式，所以是引用透明的。我們可以使用一種叫做“等式推導”（equational reasoning）的技術來分析程式碼。所謂“等式推導”就是“一對一”替換，有點像在不考慮程式性執行的怪異行爲（quirks of programmatic evaluation）的情況下，手動執行相關程式碼。我們藉助引用透明性來剖析一下這段程式碼。

首先內聯 `isSameTeam` 函式：

```
var punch = function(player, target) {
  if(player.team === target.team) {
    return target;
  } else {
    return decrementHP(target);
  }
};
```

因爲是不可變資料，我們可以直接把 `team` 替換爲實際值：

```
var punch = function(player, target) {
  if("red" === "green") {
    return target;
  } else {
    return decrementHP(target);
  }
};
```

`if` 語句執行結果爲 `false`，所以可以把整個 `if` 語句都刪掉：

```
var punch = function(player, target) {
  return decrementHP(target);
};
```

如果再內聯 `decrementHP`，我們會發現這種情況下，`punch` 變成了一個讓 `hp` 的值減 1 的呼叫：

```
var punch = function(player, target) {
  return target.set("hp", target.hp-1);
};
```

總之，等式推導帶來的分析程式碼的能力對重構和理解程式碼非常重要。事實上，我們重構海鷗程式使用的正是這項技術：利用加和乘的特性。對這些技術的使用將會貫穿本書，眞的。

並行程式碼

最後一點，也是決定性的一點：我們可以並行執行任意純函式。因爲純函式根本不需要訪問共享的記憶體，而且根據其定義，純函式也不會因副作用而進入競爭態（race condition）。

並行程式碼在服務端 js 環境以及使用了 web worker 的瀏覽器那裡是非常容易實現的，因爲它們使用了執行緒（thread）。不過出於對非純函式複雜度的考慮，當前主流觀點還是避免使用這種並行。

總結

我們已經瞭解什麼是純函數了，也看到作爲函數式程式設計師的我們，爲何深信純函式是不同凡響的。從這開始，我們將盡力以純函數式的方式書寫所有的函式。爲此我們將需要一些額外的工具來達成目標，同時也儘量把非純函式從純函式程式碼中分離。

如果手頭沒有一些工具，那麼純函式程式寫起來就有點費力。我們不得不玩雜耍似的通過到處傳遞引數來操作資料，而且還被禁止使用狀態，更別說“作用”了。沒有人願意這樣自虐。所以讓我們來學習一個叫 curry 的新工具。

第 4 章: 柯里化（curry）

第 4 章: 柯里化（curry）

不可或缺的 curry

（譯者注：原標題是“Can't live if livin' is without you”，爲英國樂隊 Badfinger 歌曲 *Without You* 中歌詞。）

我父親以前跟我說過，有些事物在你得到之前是無足輕重的，得到之後就不可或缺了。微波爐是這樣，智慧手機是這樣，網際網路也是這樣——老人們在沒有網際網路的時候過得也很充實。對我來說，函式的柯里化（curry）也是這樣。

curry 的概念很簡單：只傳遞給函式一部分引數來呼叫它，讓它返回一個函式去處理剩下的引數。

你可以一次性地呼叫 curry 函式，也可以每次只傳一個引數分多次呼叫。

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var increment = add(1);
var addTen = add(10);

increment(2);
// 3

addTen(2);
// 12
```

這裡我們定義了一個 `add` 函式，它接受一個引數並返回一個新的函式。呼叫 `add` 之後，返回的函式就通過閉包的方式記住了 `add` 的第一個引數。一次性地呼叫它實在是有點繁瑣，好在我們可以使用一個特殊的 `curry` 幫助函式（helper function）使這類函式的定義和呼叫更加容易。

我們來建立一些 curry 函式享受下（譯者注：此處原文是“for our enjoyment”，語出自聖經）。

```
var curry = require('lodash').curry;

var match = curry(function(what, str) {
  return str.match(what);
});

var replace = curry(function(what, replacement, str) {
  return str.replace(what, replacement);
});

var filter = curry(function(f, ary) {
  return ary.filter(f);
});

var map = curry(function(f, ary) {
  return ary.map(f);
});
```

我在上面的程式碼中遵循的是一種簡單，同時也非常重要的模式。即策略性地把要操作的資料（String， Array）放到最後一個引數裡。到使用它們的時候你就明白這樣做的原因是什麼了。

```
match(/\s+/g, "hello world");
// [ ' ' ]

match(/\s+/g)("hello world");
// [ ' ' ]

var hasSpaces = match(/\s+/g);
// function(x) { return x.match(/\s+/g) }

hasSpaces("hello world");
// [ ' ' ]

hasSpaces("spaceless");
// null

filter(hasSpaces, ["tori_spelling", "tori amos"]);
// ["tori amos"]

var findSpaces = filter(hasSpaces);
// function(xs) { return xs.filter(function(x) { return x.match(
/\s+/g) }) }

findSpaces(["tori_spelling", "tori amos"]);
// ["tori amos"]

var noVowels = replace(/[aeiou]/ig);
// function(replacement, x) { return x.replace(/[aeiou]/ig, repl
acement) }

var censored = noVowels("*");
// function(x) { return x.replace(/[aeiou]/ig, "*") }

censored("Chocolate Rain");
// 'Ch*c*l*t* R**n'
```

這裡表明的是一種“預載入”函式的能力，通過傳遞一到兩個引數呼叫函式，就能得到一個記住了這些引數的新函式。

我鼓勵你使用 `npm install lodash` 安裝 `lodash`，複製上面的程式碼放到 REPL 裡跑一跑。當然你也可以在能夠使用 `lodash` 或 `ramda` 的網頁中執行它們。

不僅僅是雙關語／咖哩

curry 的用處非常廣泛，就像在 `hasSpaces`、`findSpaces` 和 `censored` 看到的那樣，只需傳給函式一些引數，就能得到一個新函式。

用 `map` 簡單地把引數是單個元素的函式包裹一下，就能把它轉換成引數爲陣列的函式。

```
var getChildren = function(x) {
  return x.childNodes;
};

var allTheChildren = map(getChildren);
```

只傳給函式一部分引數通常也叫做區域性呼叫（partial application），能夠大量減少樣板檔案程式碼（boilerplate code）。考慮上面的 `allTheChildren` 函式，如果用 lodash 的普通 `map` 來寫會是什麼樣的（注意引數的順序也變了）：

```
var allTheChildren = function(elements) {
  return _.map(elements, getChildren);
};
```

通常我們不定義直接運算元組的函式，因爲只需內聯呼叫 `map(getChildren)` 就能達到目的。這一點同樣適用於 `sort`、`filter` 以及其他的高階函式（higher order function）（高階函式：引數或返回值爲函式的函式）。

當我們談論純函式的時候，我們說它們接受一個輸入返回一個輸出。curry 函式所做的正是這樣：每傳遞一個引數呼叫函式，就返回一個新函式處理剩餘的引數。這就是一個輸入對應一個輸出啊。

哪怕輸出是另一個函式，它也是純函式。當然 curry 函式也允許一次傳遞多個引數，但這只是出於減少 `()` 的方便。

總結

curry 函式用起來非常得心應手，每天使用它對我來說簡直就是一種享受。它堪稱手頭必備工具，能夠讓函數語言程式設計不那麼繁瑣和沉悶。

通過簡單地傳遞幾個引數，就能動態建立實用的新函式；而且還能帶來一個額外好處，那就是保留了數學的函式定義，儘管引數不止一個。下一章我們將學習另一個重要的工具：`組合`（compose）。

第 5 章: 程式碼組合（compose）

練習

開始練習之前先說明一下，我們將預設使用 ramda 這個庫來把函式轉爲 curry 函式。或者你也可以選擇由 losash 的作者編寫和維護的 lodash-fp。這兩個庫都很好用，選擇哪一個就看你自己的喜好了。

你還可以對自己的練習程式碼做單元測試，或者把程式碼拷貝到一個 REPL 裡執行看看。

這些練習的答案可以在本書倉庫中找到。

```
var _ = require('ramda');


// 練習 1
//==============
// 通過區域性呼叫（partial apply）移除所有引數

var words = function(str) {
  return split(' ', str);
};

// 練習 1a
//==============
// 使用 `map` 建立一個新的 `words` 函式，使之能夠操作字串陣列

var sentences = undefined;
```

```
// 練習 2
//==============
// 通過區域性呼叫（partial apply）移除所有引數

var filterQs = function(xs) {
  return filter(function(x){ return match(/q/i, x);  }, xs);
};


// 練習 3
//==============
// 使用幫助函式 `_keepHighest` 重構 `max` 使之成爲 curry 函式

// 無須改動:
var _keepHighest = function(x,y){ return x >= y ? x : y; };

// 重構這段程式碼:
var max = function(xs) {
  return reduce(function(acc, x){
    return _keepHighest(acc, x);
  }, -Infinity, xs);
};


// 彩蛋 1:
// ============
// 包裹陣列的 `slice` 函式使之成爲 curry 函式
// //[1,2,3].slice(0, 2)
var slice = undefined;


// 彩蛋 2:
// ============
// 藉助 `slice` 定義一個 `take` curry 函式，該函式呼叫後可以取出字串的前
 n 個字元。
var take = undefined;
```

第 5 章: 程式碼組合（compose）

函式飼養

這就是 `組合` （compose，以下將稱之爲組合）：

```
var compose = function(f,g) {
  return function(x) {
    return f(g(x));
  };
};
```

`f` 和 `g` 都是函式，`x` 是在它們之間通過“管道”傳輸的值。

`組合` 看起來像是在飼養函式。你就是飼養員，選擇兩個有特點又遭你喜歡的函式，讓它們結合，產下一個嶄新的函式。組合的用法如下：

```
var toUpperCase = function(x) { return x.toUpperCase(); };
var exclaim = function(x) { return x + '!'; };
var shout = compose(exclaim, toUpperCase);

shout("send in the clowns");
//=> "SEND IN THE CLOWNS!"
```

兩個函式組合之後返回了一個新函式是完全講得通的：組合某種型別（本例中是函式）的兩個元素本就該生成一個該型別的新元素。把兩個樂高積木組合起來絕不可能得到一個林肯積木。所以這是有道理的，我們將在適當的時候探討這方面的一些底層理論。

在 `compose` 的定義中，`g` 將先於 `f` 執行，因此就建立了一個從右到左的資料流。這樣做的可讀性遠遠高於巢狀一大堆的函式呼叫，如果不用組合，`shout` 函式將會是這樣的：

```
var shout = function(x){
  return exclaim(toUpperCase(x));
};
```

讓程式碼從右向左執行，而不是由內而外執行，我覺得可以稱之爲“左傾”（籲——）。我們來看一個順序很重要的例子：

```
var head = function(x) { return x[0]; };
var reverse = reduce(function(acc, x){ return [x].concat(acc); }
, []);
var last = compose(head, reverse);

last(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'uppercut'
```

`reverse` 反轉列表，`head` 取列表中的第一個元素；所以結果就是得到了一個 `last` 函式（譯者注：即取列表的最後一個元素），雖然它效能不高。這個組合中函式的執行順序應該是顯而易見的。儘管我們可以定義一個從左向右的版本，但是從右向左執行更加能夠反映數學上的含義——是的，組合的概念直接來自於數學課本。實際上，現在是時候去看看所有的組合都有的一個特性了。

```
// 結合律（associativity）
var associative = compose(f, compose(g, h)) == compose(compose(f
, g), h);
// true
```

這個特性就是結合律，符合結合律意味著不管你是把 `g` 和 `h` 分到一組，還是把 `f` 和 `g` 分到一組都不重要。所以，如果我們想把字串變爲大寫，可以這麼寫：

```
compose(toUpperCase, compose(head, reverse));

// 或者
compose(compose(toUpperCase, head), reverse);
```

因爲如何爲 `compose` 的呼叫分組不重要，所以結果都是一樣的。這也讓我們有能力寫一個可變的組合（variadic compose），用法如下：

```
// 前面的例子中我們必須要寫兩個組合才行，但既然組合是符合結合律的，我們就可以只寫一個，
// 而且想傳給它多少個函式就傳給它多少個，然後讓它自己決定如何分組。

var lastUpper = compose(toUpperCase, head, reverse);

lastUpper(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'UPPERCUT'


var loudLastUpper = compose(exclaim, toUpperCase, head, reverse)

loudLastUpper(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'UPPERCUT!'
```

運用結合律能爲我們帶來強大的靈活性，還有對執行結果不會出現意外的那種平和心態。至於稍微複雜些的可變組合，也都包含在本書的 `support` 庫裡了，而且你也可以在類似 lodash、underscore 以及 ramda 這樣的類庫中找到它們的常規定義。

結合律的一大好處是任何一個函式分組都可以被拆開來，然後再以它們自己的組合方式打包在一起。讓我們來重構重構前面的例子：

```
var loudLastUpper = compose(exclaim, toUpperCase, head, reverse)
;

// 或
var last = compose(head, reverse);
var loudLastUpper = compose(exclaim, toUpperCase, last);

// 或
var last = compose(head, reverse);
var angry = compose(exclaim, toUpperCase);
var loudLastUpper = compose(angry, last);

// 更多變種...
```

關於如何組合，並沒有標準的答案——我們只是以自己喜歡的方式搭樂高積木罷了。通常來說，最佳實踐是讓組合可重用，就像 `last` 和 `angry` 那樣。如果熟悉 Fowler 的《重構》一書的話，你可能會認識到這個過程叫做 “extract method”——只不過不需要關心物件的狀態。

pointfree

pointfree 模式指的是，永遠不必說出你的資料。咳咳對不起（譯者注：此處原文是“Pointfree style means never having to say your data”，源自 1970 年的電影 *Love Story* 裡的一句著名臺詞“Love means never having to say you're sorry”。緊接著作者又說了一句“Excuse me”，大概是一種幽默）。它的意思是說，函式無須提及將要操作的資料是什麼樣的。一等公民的函式、柯里化（curry）以及組合協作起來非常有助於實現這種模式。

```
// 非 pointfree，因爲提到了資料：word
var snakeCase = function (word) {
  return word.toLowerCase().replace(/\s+/ig, '_');
};

// pointfree
var snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

看到 `replace` 是如何被區域性呼叫的了麼？這裡所做的事情就是通過管道把資料在接受單個引數的函式間傳遞。利用 curry，我們能夠做到讓每個函式都先接收資料，然後操作資料，最後再把資料傳遞到下一個函式那裡去。另外注意在 pointfree 版本中，不需要 `word` 引數就能建構函式；而在非 pointfree 的版本中，必須要有 `word` 才能進行進行一切操作。

我們再來看一個例子。

```
// 非 pointfree，因爲提到了資料：name
var initials = function (name) {
  return name.split(' ').map(compose(toUpperCase, head)).join('.
 ');
};

// pointfree
var initials = compose(join('. '), map(compose(toUpperCase, head
)), split(' '));

initials("hunter stockton thompson");
// 'H. S. T'
```

另外，pointfree 模式能夠幫助我們減少不必要的命名，讓程式碼保持簡潔和通用。對函數式程式碼來說，pointfree 是非常好的石蕊試驗，因爲它能告訴我們一個函式是否是接受輸入返回輸出的小函式。比如，while 迴圈是不能組合的。不過你也要警惕，pointfree 就像是一把雙刃劍，有時候也能混淆視聽。並非所有的函數式程式碼都是 pointfree 的，不過這沒關係。可以使用它的時候就使用，不能使用的時候就用普通函式。

debug

組合的一個常見錯誤是，在沒有區域性呼叫之前，就組合類似 `map` 這樣接受兩個引數的函式。

```
// 錯誤做法：我們傳給了 `angry` 一個數組，根本不知道最後傳給 `map` 的是什麼東西。
var latin = compose(map, angry, reverse);

latin(["frog", "eyes"]);
// error


// 正確做法：每個函式都接受一個實際引數。
var latin = compose(map(angry), reverse);

latin(["frog", "eyes"]);
// ["EYES!", "FROG!"])
```

如果在 debug 組合的時候遇到了困難，那麼可以使用下面這個實用的，但是不純的 `trace` 函式來追蹤程式碼的執行情況。

```
var trace = curry(function(tag, x){
  console.log(tag, x);
  return x;
});

var dasherize = compose(join('-'), toLower, split(' '), replace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

這裡報錯了，來 `trace` 下：

```
var dasherize = compose(join('-'), toLower, trace("after split"), split(' '), replace(/\s{2,}/ig, ' '));
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

啊！`toLower` 的引數是一個數組，所以需要先用 `map` 呼叫一下它。

```
var dasherize = compose(join('-'), map(toLower), split(' '), rep
lace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');

// 'the-world-is-a-vampire'
```

`trace` 函式允許我們在某個特定的點觀察資料以便 debug。像 haskell 和 purescript 之類的語言出於開發的方便，也都提供了類似的函式。

組合將成爲我們構造程式的工具，而且幸運的是，它背後是有一個強大的理論做支撐的。讓我們來研究研究這個理論。

範疇學

範疇學（category theory）是數學中的一個抽象分支，能夠形式化諸如集合論（set theory）、型別論（type theory）、群論（group theory）以及邏輯學（logic）等數學分支中的一些概念。範疇學主要處理物件（object）、態射（morphism）和變化式（transformation），而這些概念跟程式設計的聯絡非常緊密。下圖是一些相同的概念分別在不同理論下的形式：

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

抱歉，我沒有任何要嚇唬你的意思。我並不假設你對這些概念都瞭如指掌，我只是想讓你明白這裡面有多少重複的內容，讓你知道爲何範疇學要統一這些概念。

在範疇學中，有一個概念叫做...範疇。有著以下這些元件（component）的蒐集（collection）就構成了一個範疇：

- 物件的蒐集
- 態射的蒐集
- 態射的組合
- identity 這個獨特的態射

範疇學抽象到足以模擬任何事物，不過目前我們最關心的還是型別和函式，所以讓我們把範疇學運用到它們身上看看。

物件的蒐集

物件就是資料型別，例如 `String`、`Boolean`、`Number` 和 `Object` 等等。通常我們把資料型別視作所有可能的值的一個集合（set）。像 `Boolean` 就可以看作是 `[true, false]` 的集合，`Number` 可以是所有實數的一個集合。把型別當作集合對待是有好處的，因爲我們可以利用集合論（set theory）處理型別。

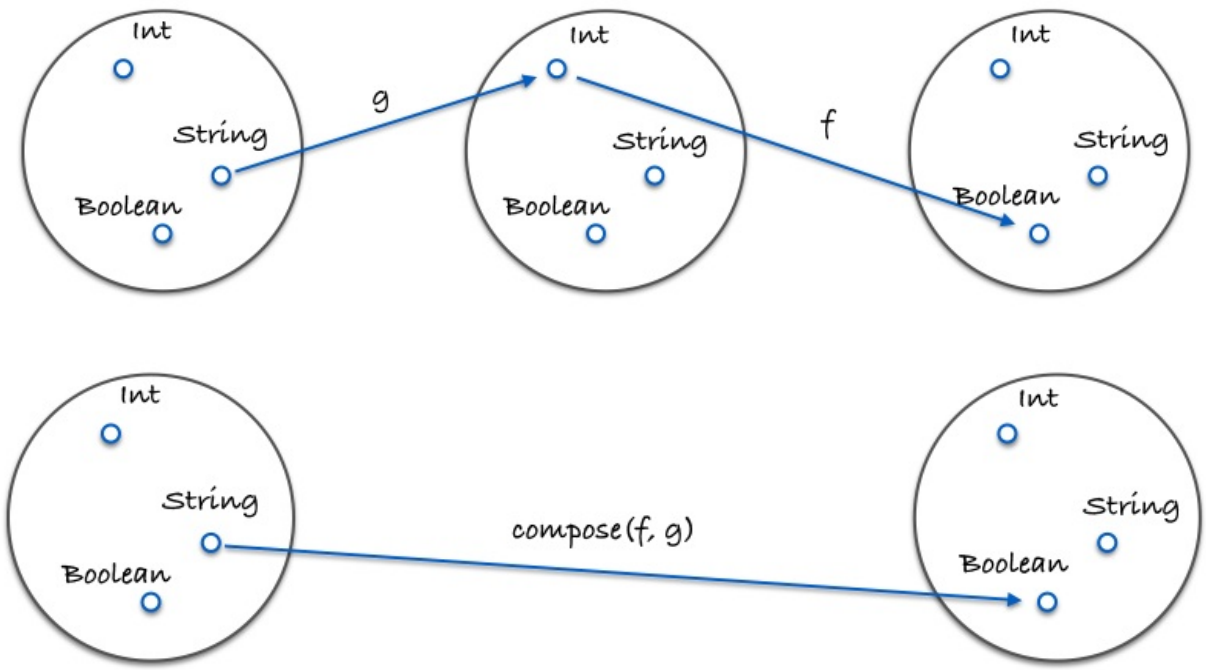
態射的蒐集

態射是標準的、普通的純函式。

態射的組合

你可能猜到了，這就是本章介紹的新玩意兒—— `組合` 。我們已經討論過 `compose` 函式是符合結合律的，這並非巧合，結合律是在範疇學中對任何組合都適用的一個特性。

這張圖展示了什麼是組合：



這裡有一個具體的例子：

```
var g = function(x){ return x.length; };
var f = function(x){ return x === 4; };
var isFourLetterWord = compose(f, g);
```

identity 這個獨特的態射

讓我們介紹一個名爲 `id` 的實用函式。這個函式接受隨便什麼輸入然後原封不動地返回它：

```
var id = function(x){ return x; };
```

你可能會問“這到底哪裡有用了？”。別急，我們會在隨後的章節中拓展這個函式的，暫時先把它當作一個可以替代給定值的函式——一個假裝自己是普通資料的函式。

`id` 函式跟組合一起使用簡直完美。下面這個特性對所有的一元函式（unary function）（一元函式：只接受一個引數的函式） `f` 都成立：

```
// identity
compose(id, f) == compose(f, id) == f;
// true
```

嘿，這就是實數的單位元（identity property）嘛！如果這還不夠清楚直白，彆著急，慢慢理解它的無用性。很快我們就會到處使用 `id` 了，不過暫時我們還是把當作一個替代給定值的函式。這對寫 pointfree 的程式碼非常有用。

好了，以上就是型別和函式的範疇。不過如果你是第一次聽說這些概念，我估計你還是有些迷糊，不知道範疇到底是什麼，爲什麼有用。沒關係，本書全書都在藉助這些知識編寫示例程式碼。至於現在，就在本章，本行文字中，你至少可以認爲它向我們提供了有關組合的知識——比如結合律和單位律。

除了型別和函式，還有什麼範疇呢？還有很多，比如我們可以定義一個有向圖（directed graph），以節點爲物件，以邊爲態射，以路徑連線爲組合。還可以定義一個實數型別（Number），以所有的實數物件，以 `>=` 爲態射（實際上任何偏序（partial order）或全序（total order）都可以成爲一個範疇）。範疇的總數是無限的，但是要達到本書的目的，我們只需要關心上面定義的範疇就好了。至此我們已經大致瀏覽了一些表面的東西，必須要繼續後面的內容了。

總結

組合像一系列管道那樣把不同的函式聯絡在一起，資料就可以也必須在其中流動——畢竟純函式就是輸入對輸出，所以打破這個鏈條就是不尊重輸出，就會讓我們的應用一無是處。

我們認爲組合是高於其他所有原則的設計原則，這是因爲組合讓我們的程式碼簡單而富有可讀性。另外範疇學將在應用架構、模擬副作用和保證正確性方面扮演重要角色。

現在我們已經有足夠的知識去進行一些實際的練習了，讓我們來編寫一個示例應用。

第 6 章: 示例應用

練習

```
require('../../support');
var _ = require('ramda');
var accounting = require('accounting');

// 示例資料
var CARS = [
    {name: "Ferrari FF", horsepower: 660, dollar_value: 700000,
in_stock: true},
    {name: "Spyker C12 Zagato", horsepower: 650, dollar_value: 6
48000, in_stock: false},
    {name: "Jaguar XKR-S", horsepower: 550, dollar_value: 132000
, in_stock: false},
    {name: "Audi R8", horsepower: 525, dollar_value: 114200, in_
stock: false},
    {name: "Aston Martin One-77", horsepower: 750, dollar_value:
1850000, in_stock: true},
    {name: "Pagani Huayra", horsepower: 700, dollar_value: 13000
00, in_stock: false}
  ];

// 練習 1:
// ============
// 使用 _.compose() 重寫下面這個函式。提示：_.prop() 是 curry 函式
var isLastInStock = function(cars) {
  var last_car = _.last(cars);
  return _.prop('in_stock', last_car);
};

// 練習 2:
// ============
// 使用 _.compose()、_.prop() 和 _.head() 獲取第一個 car 的 name
var nameOfFirstCar = undefined;


// 練習 3:
// ============
// 使用幫助函式 _average 重構 averageDollarValue 使之成爲一個組合
var _average = function(xs) { return reduce(add, 0, xs) / xs.len
gth; }; // <- 無須改動
```

```
var averageDollarValue = function(cars) {
  var dollar_values = map(function(c) { return c.dollar_value; }
, cars);
  return _average(dollar_values);
};


// 練習 4:
// ============
// 使用 compose 寫一個 sanitizeNames() 函式，返回一個下劃線連線的小寫字
串：例如：sanitizeNames(["Hello World"]) //=> ["hello_world"]。

var _underscore = replace(/\W+/g, '_'); //<-- 無須改動，並在 sanit
izeNames 中使用它

var sanitizeNames = undefined;


// 彩蛋 1:
// ============
// 使用 compose 重構 availablePrices

var availablePrices = function(cars) {
  var available_cars = _.filter(_.prop('in_stock'), cars);
  return available_cars.map(function(x){
    return accounting.formatMoney(x.dollar_value);
  }).join(', ');
};


// 彩蛋 2:
// ============
// 重構使之成爲 pointfree 函式。提示：可以使用 _.flip()

var fastestCar = function(cars) {
  var sorted = _.sortBy(function(car){ return car.horsepower },
cars);
  var fastest = _.last(sorted);
  return fastest.name + ' is the fastest';
};
```

第 6 章: 示例應用

宣告式程式碼

我們要開始轉變觀念了，從本章開始，我們將不再指示計算機如何工作，而是指出我們明確希望得到的結果。我敢保證，這種做法與那種需要時刻關心所有細節的指令式程式設計相比，會讓你輕鬆許多。

與命令式不同，宣告式意味著我們要寫表示式，而不是一步一步的指示。

以 SQL 爲例，它就沒有“先做這個，再做那個”的命令，有的只是一個指明我們想要從資料庫取什麼資料的表示式。至於如何取資料則是由它自己決定的。以後資料庫升級也好，SQL 引擎優化也好，根本不需要更改查詢語句。這是因爲，有多種方式解析一個表示式並得到相同的結果。

對包括我在內的一些人來說，一開始是不太容易理解“宣告式”這個概念的；所以讓我們寫幾個例子找找感覺。

```
// 命令式
var makes = [];
for (i = 0; i < cars.length; i++) {
  makes.push(cars[i].make);
}


// 宣告式
var makes = cars.map(function(car){ return car.make; });
```

命令式的迴圈要求你必須先例項化一個數組，而且執行完這個例項化語句之後，直譯器才繼續執行後面的程式碼。然後再直接迭代 `cars` 列表，手動增加計數器，把各種零零散散的東西都展示出來...實在是直白得有些露骨。

使用 `map` 的版本是一個表示式，它對執行順序沒有要求。而且，`map` 函式如何進行迭代，返回的陣列如何收集，都有很大的自由度。它指明的是 `做什麼` ，不是 `怎麼做` 。因此，它是正兒八經的宣告式程式碼。

除了更加清晰和簡潔之外，`map` 函式還可以進一步優化，這麼一來我們寶貴的應用程式碼就無須改動了。

至於那些說“雖然如此，但使用命令式迴圈速度要快很多”的人，我建議你們先去學學 JIT 優化程式碼的相關知識。這裡有一個非常棒的視訊，可能會對你有幫助。

再看一個例子。

```
// 命令式
var authenticate = function(form) {
  var user = toUser(form);
  return logIn(user);
};

// 宣告式
var authenticate = compose(logIn, toUser);
```

雖然命令式的版本並不一定就是錯的，但還是硬編碼了那種一步接一步的執行方式。而 `compose` 表示式只是簡單地指出了這樣一個事實：使用者驗證是 `toUser` 和 `logIn` 兩個行爲的組合。這再次說明，宣告式爲潛在的程式碼更新提供了支援，使得我們的應用程式碼成爲了一種高階規範（high level specification）。

因爲宣告式程式碼不指定執行順序，所以它天然地適合進行並行運算。它與純函式一起解釋了爲何函數語言程式設計是未來平行計算的一個不錯選擇——我們眞的不需要做什麼就能實現一個並行／併發系統。

一個函數式的 flickr

現在我們以一種宣告式的、可組合的方式建立一個示例應用。暫時我們還是會作點小弊，使用副作用；但我們會把副作用的程度降到最低，讓它們與純函式程式碼分離開來。這個示例應用是一個瀏覽器 widget，功能是從 flickr 獲取圖片並在頁面上展示。我們從寫 html 開始：

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/require.
js/2.1.11/require.min.js"></script>
    <script src="flickr.js"></script>
  </head>
  <body></body>
</html>
```

flickr.js 如下：

```
requirejs.config({
  paths: {
    ramda: 'https://cdnjs.cloudflare.com/ajax/libs/ramda/0.13.0/
ramda.min',
    jquery: 'https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/
jquery.min'
  }
});

require([
    'ramda',
    'jquery'
  ],
  function (_, $) {
    var trace = _.curry(function(tag, x) {
      console.log(tag, x);
      return x;
    });
    // app goes here
  });
```

這裡我們使用了 ramda ，沒有用 lodash 或者其他類庫。ramda 提供了 `compose` 、 `curry` 等很多函式。模組載入我們選擇的是 requirejs，我以前用過 requirejs，雖然它有些重，但爲了保持一致性，本書將一直使用它。另外，我也把 `trace` 函式寫好了，便於 debug。

有點跑題了。言歸正傳，我們的應用將做 4 件事：

1. 根據特定搜尋關鍵字構造 url
2. 向 flickr 傳送 api 請求
3. 把返回的 json 轉爲 html 圖片
4. 把圖片放到螢幕上

注意到沒？上面提到了兩個不純的動作，即從 flickr 的 api 獲取資料和在螢幕上放置圖片這兩件事。我們先來定義這兩個動作，這樣就能隔離它們了。

```
var Impure = {
  getJSON: _.curry(function(callback, url) {
    $.getJSON(url, callback);
  }),

  setHtml: _.curry(function(sel, html) {
    $(sel).html(html);
  })
};
```

這裡只是簡單地包裝了一下 jQuery 的 `getJSON` 方法，把它變爲一個 curry 函式，還有就是把引數位置也調換了下。這些方法都在 `Impure` 名稱空間下，這樣我們就知道它們都是危險函式。在後面的例子中，我們會把這兩個函式變純。

下一步是構造 url 傳給 `Impure.getJSON` 函式。

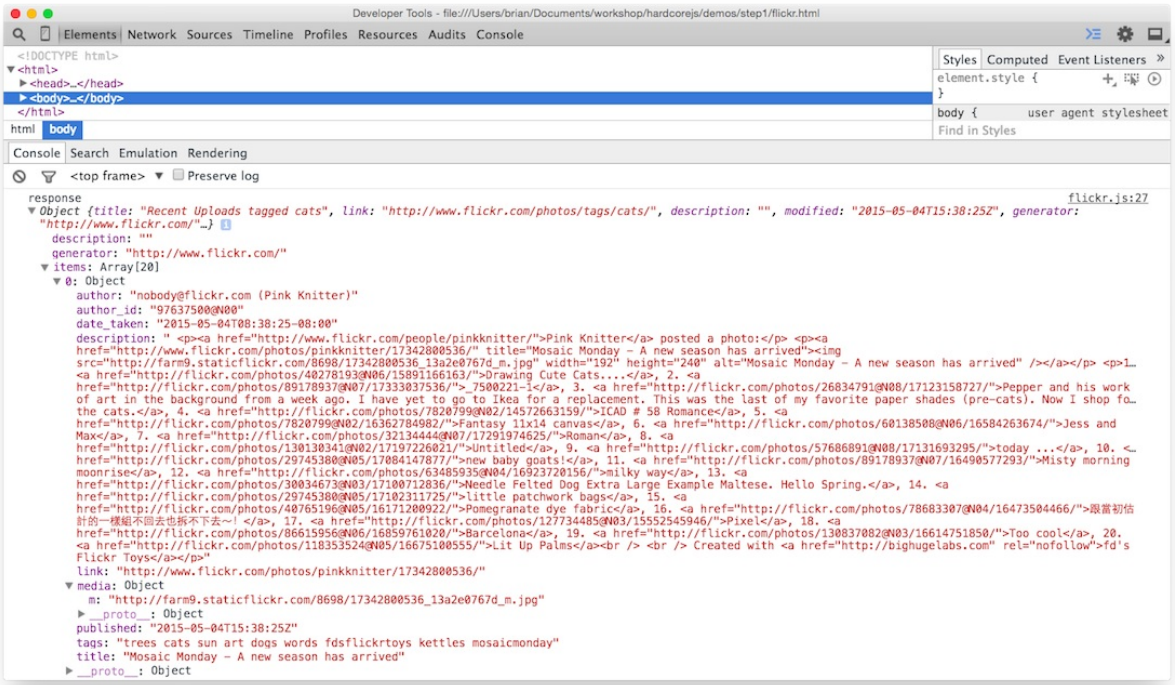
```
var url = function (term) {
  return 'https://api.flickr.com/services/feeds/photos_public.gn
e?tags=' + term + '&format=json&jsoncallback=?';
};
```

藉助 monoid 或 combinator （後面會講到這些概念），我們可以使用一些奇技淫巧來讓 `url` 函式變爲 pointfree 函式。但是爲了可讀性，我們還是選擇以普通的非 pointfree 的方式拼接字串。

讓我們寫一個 `app` 函式傳送請求並把內容放置到螢幕上。

```
var app = _.compose(Impure.getJSON(trace("response")), url);

app("cats");
```

這會呼叫 `url` 函式，然後把字串傳給 `getJSON` 函式。`getJSON` 已經區域性應用了 `trace`，載入這個應用將會把請求的響應顯示在 console 裡。



我們想要從這個 json 裡構造圖片，看起來 src 都在 `items` 陣列中的每個 `media` 物件的 `m` 屬性上。

不管怎樣，我們可以使用 ramda 的一個通用 getter 函式 `_.prop()` 來獲取這些巢狀的屬性。不過爲了讓你明白這個函式做了什麼事情，我們自己實現一個 prop 看看：

```
var prop = _.curry(function(property, object){
  return object[property];
});
```

實際上這有點傻，僅僅是用 `[]` 來獲取一個物件的屬性而已。讓我們利用這個函式獲取圖片的 src。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var srcs = _.compose(_.map(mediaUrl), _.prop('items'));
```

一旦得到了 `items`，就必須使用 `map` 來分解每一個 url；這樣就得到了一個包含所有 src 的陣列。把它和 `app` 聯結起來，列印結果看看。

```
var renderImages = _.compose(Impure.setHtml("body"), srcs);
var app = _.compose(Impure.getJSON(renderImages), url);
```

這裡所做的只不過是新建了一個組合，這個組合會呼叫 `srcs` 函式，並把返回結果設定爲 body 的 html。我們也把 `trace` 替換爲了 `renderImages`，因爲已經有了除原始 json 以外的資料。這將會粗暴地把所有的 src 直接顯示在螢幕上。

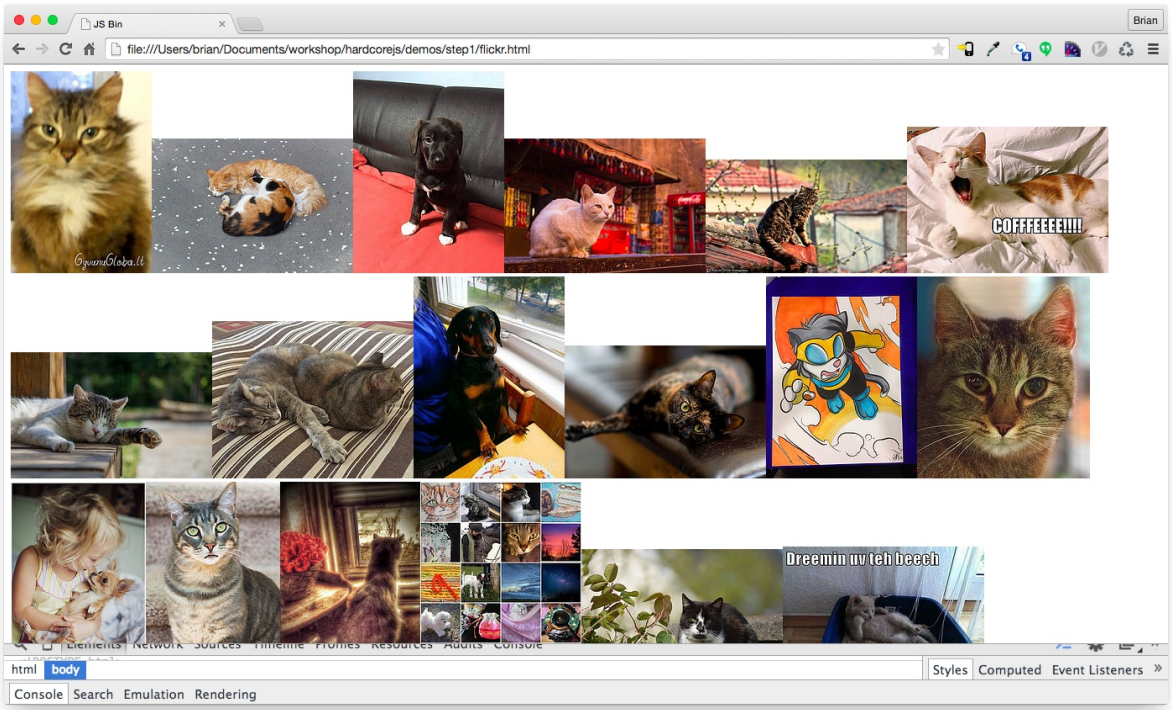
最後一步是把這些 src 變爲眞正的圖片。對大型點的應用來說，是應該使用類似 Handlebars 或者 React 這樣的 template/dom 庫來做這件事的。但我們這個應用太小了，只需要一個 img 標籤，所以用 jQuery 就好了。

```
var img = function (url) {
  return $('<img />', { src: url });
};
```

jQuery 的 `html()` 方法接受標籤陣列爲引數，所以我們只須把 src 轉換爲 img 標籤然後傳給 `setHtml` 即可。

```
var images = _.compose(_.map(img), srcs);
var renderImages = _.compose(Impure.setHtml("body"), images);
var app = _.compose(Impure.getJSON(renderImages), url);
```

任務完成！



下面是完整程式碼：

```
requirejs.config({
  paths: {
    ramda: 'https://cdnjs.cloudflare.com/ajax/libs/ramda/0.13.0/
ramda.min',
    jquery: 'https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/
jquery.min'
  }
});

require([
    'ramda',
    'jquery'
  ],
  function (_, $) {
    ////////////////////////////////////////////
    // Utils

    var Impure = {
      getJSON: _.curry(function(callback, url) {
        $.getJSON(url, callback);
      }),
```

```
      setHtml: _.curry(function(sel, html) {
        $(sel).html(html);
      })
    };

    var img = function (url) {
      return $('<img />', { src: url });
    };

    var trace = _.curry(function(tag, x) {
      console.log(tag, x);
      return x;
    });

    //////////////////////////////////////////

    var url = function (t) {
      return 'https://api.flickr.com/services/feeds/photos_publi
c.gne?tags=' + t + '&format=json&jsoncallback=?';
    };

    var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

    var srcs = _.compose(_.map(mediaUrl), _.prop('items'));

    var images = _.compose(_.map(img), srcs);

    var renderImages = _.compose(Impure.setHtml("body"), images)
;

    var app = _.compose(Impure.getJSON(renderImages), url);

    app("cats");
  });
```

看看，多麼美妙的宣告式規範啊，只說做什麼，不說怎麼做。現在我們可以把每一行程式碼都視作一個等式，變數名所代表的屬性就是等式的含義。我們可以利用這些屬性去推導分析和重構這個應用。

有原則的重構

上面的程式碼是有優化空間的——我們獲取 url map 了一次，把這些 url 變爲 img 標籤又 map 了一次。關於 map 和組合是有定律的：

```
// map 的組合律
var law = compose(map(f), map(g)) == map(compose(f, g));
```

我們可以利用這個定律優化程式碼，進行一次有原則的重構。

```
// 原有程式碼
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var srcs = _.compose(_.map(mediaUrl), _.prop('items'));

var images = _.compose(_.map(img), srcs);
```

感謝等式推導（equational reasoning）及純函式的特性，我們可以內聯呼叫 `srcs` 和 `images` ，也就是把 map 呼叫排列起來。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var images = _.compose(_.map(img), _.map(mediaUrl), _.prop('item
s'));
```

把 `map` 排成一列之後就可以應用組合律了。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var images = _.compose(_.map(_.compose(img, mediaUrl)), _.prop('
items'));
```

現在只需要迴圈一次就可以把每一個物件都轉爲 img 標籤了。我們把 map 呼叫的 compose 取出來放到外面，提高一下可讀性。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var mediaToImg = _.compose(img, mediaUrl);

var images = _.compose(_.map(mediaToImg), _.prop('items'));
```

總結

我們已經見識到如何在一個小而不失眞實的應用中運用新技能了，也已經使用過函數式這個“數學框架”來推導和重構程式碼了。但是異常處理以及程式碼分支呢？如何讓整個應用都是函數式的，而不僅僅是把破壞性的函式放到名稱空間下？如何讓應用更安全更富有表現力？這些都是本書第 2 部分將要解決的問題。

第 7 章: Hindley-Milner 型別簽名

Hindley-Milner 型別簽名

初識型別

剛接觸函數語言程式設計的人很容易深陷型別簽名（type signatures）的泥淖。型別（type）是讓所有不同背景的人都能高效溝通的元語言。很大程度上，型別簽名是以 “Hindley-Milner” 系統寫就的，本章我們將一起探究下這個系統。

型別簽名在寫純函式時所起的作用非常大，大到英語都不能望其項背。這些簽名輕輕訴說著函式最不可告人的祕密。短短一行，就能暴露函式的行爲和目的。型別簽名還衍生出了 “自由定理（free theorems）” 的概念。因爲型別是可以推斷的，所以明確的型別簽名並不是必要的；不過你完全可以寫精確度很高的型別簽名，也可以讓它們保持通用、抽象。型別簽名不但可以用於編譯時檢測（compile time checks），還是最好的文件。所以型別簽名在函數語言程式設計中扮演著非常重要的角色——重要程度遠遠超出你的想象。

JavaScript 是一種動態型別語言，但這並不意味著要一味否定型別。我們還是要和字串、數值、布林值等等型別打交道的；只不過，語言層面上沒有相關的整合讓我們時刻謹記各種資料的型別罷了。別擔心，既然我們可以用型別簽名生成文件，也可以用註釋來達到區分型別的目的。

JavaScript 也有一些型別檢查工具，比如 Flow，或者它的靜態型別方言 TypeScript。由於本書的目標是讓讀者能夠熟練使用各種工具去書寫函數式程式碼，所以我們將選擇所有函數式語言都遵循的標準型別系統。

神祕的傳奇故事

從積塵已久的數學書，到浩如煙海的學術論文；從每週必讀的部落格文章，到原始碼本身，我們都能發現 Hindley-Milner 型別簽名的身影。Hindley-Milner 並不是一個複雜的系統，但還是需要一些解釋和練習才能完全掌握這個小型語言的要義。

```
//  capitalize :: String -> String
var capitalize = function(s){
  return toUpperCase(head(s)) + toLowerCase(tail(s));
}

capitalize("smurf");
//=> "Smurf"
```

這裡，`capitalize` 接受一個 `String` 並返回了一個 `String`。先別管實現，我們感興趣的是它的型別簽名。

在 Hindley-Milner 系統中，函式都寫成類似 `a -> b` 這個樣子，其中 `a` 和 `b` 是任意型別的變數。因此，`capitalize` 函式的型別簽名可以理解爲“一個接受 `String` 返回 `String` 的函式”。換句話說，它接受一個 `String` 型別作爲輸入，並返回一個 `String` 型別的輸出。

再來看一些函式簽名：

```
//  strLength :: String -> Number
var strLength = function(s){
  return s.length;
}

//  join :: String -> [String] -> String
var join = curry(function(what, xs){
  return xs.join(what);
});

//  match :: Regex -> String -> [String]
var match = curry(function(reg, s){
  return s.match(reg);
});

//  replace :: Regex -> String -> String -> String
var replace = curry(function(reg, sub, s){
  return s.replace(reg, sub);
});
```

`strLength` 和 `capitalize` 類似：接受一個 `String` 然後返回一個 `Number` 。

至於其他的，第一眼看起來可能會比較疑惑。不過在還不完全瞭解細節的情況下，你儘可以把最後一個型別視作返回值。那麼 `match` 函式就可以這麼理解：它接受一個 `Regex` 和一個 `String` ，返回一個 `[String]` 。但是，這裡有一個非常有趣的地方，請允許我稍作解釋。

對於 `match` 函式，我們完全可以把它的型別簽名這樣分組：

```
//  match :: Regex -> (String -> [String])
var match = curry(function(reg, s){
  return s.match(reg);
});
```

是的，把最後兩個型別包在括號裡就能反映更多的資訊了。現在我們可以看出 `match` 這個函式接受一個 `Regex` 作爲引數，返回一個從 `String` 到 `[String]` 的函式。因爲 curry，造成的結果就是這樣：給 `match` 函式一個 `Regex` ，得到一個新函式，能夠處理其 `String` 引數。當然了，我們並非一定要這麼看待這個過程，但這樣思考有助於理解爲何最後一個型別是返回值。

```
//  match :: Regex -> (String -> [String])

//  onHoliday :: String -> [String]
var onHoliday = match(/holiday/ig);
```

每傳一個引數，就會彈出型別簽名最前面的那個型別。所以 `onHoliday` 就是已經有了 `Regex` 引數的 `match` 。

```
//  replace :: Regex -> (String -> (String -> String))
var replace = curry(function(reg, sub, s){
  return s.replace(reg, sub);
});
```

但是在這段程式碼中，就像你看到的那樣，爲 `replace` 加上這麼多括號未免有些多餘。所以這裡的括號是完全可以省略的，如果我們願意，可以一次性把所有的引數都傳進來；所以，一種更簡單的思路是： `replace` 接受三個引數，分別是

`Regex` 、 `String` 和另一個 `String` ，返回的還是一個 `String` 。

最後幾點：

```
//  id :: a -> a
var id = function(x){ return x; }

//  map :: (a -> b) -> [a] -> [b]
var map = curry(function(f, xs){
  return xs.map(f);
});
```

這裡的 `id` 函式接受任意型別的 `a` 並返回同一個型別的資料。和普通程式碼一樣，我們也可以在型別簽名中使用變數。把變數命名爲 `a` 和 `b` 只是一種約定俗成的習慣，你可以使用任何你喜歡的名稱。對於相同的變數名，其型別也一定相同。這是非常重要的一個原則，所以我們必須重申： `a -> b` 可以是從任意型別的 `a` 到任意型別的 `b` ，但是 `a -> a` 必須是同一個型別。例如， `id` 可以是 `String -> String` ，也可以是 `Number -> Number` ，但不能是 `String -> Bool` 。

相似地， `map` 也使用了變數，只不過這裡的 `b` 可能與 `a` 型別相同，也可能不相同。我們可以這麼理解： `map` 接受兩個引數，第一個是從任意型別 `a` 到任意型別 `b` 的函式；第二個是一個數組，元素是任意型別的 `a` ； `map` 最後返回的是一個型別 `b` 的陣列。

型別簽名的美妙令人印象深刻，希望你已經被它深深折服。型別簽名簡直能夠一字一句地告訴我們函式做了什麼事情。比如 `map` 函式就是這樣：給定一個從 `a` 到 `b` 的函式和一個 `a` 型別的陣列作爲引數，它就能返回一個 `b` 型別的陣列。 `map` 唯一的明智之舉就是使用其函式引數呼叫每一個 `a` ，其他所有操作都是噱頭。

辨別型別和它們的含義是一項重要的技能，這項技能可以讓你在函數語言程式設計的路上走得更遠。不僅論文、部落格和文件等更易理解，型別簽名本身也基本上能夠告訴你它的函式性（functionality）。要成爲一個能夠熟練讀懂型別簽名的人，你得勤於練習；不過一旦掌握了這項技能，你將會受益無窮，不讀手冊也能獲取大量資訊。

這裡還有一些例子，你可以自己試試看能不能理解它們。

```
//  head :: [a] -> a
var head = function(xs){ return xs[0]; }

//  filter :: (a -> Bool) -> [a] -> [a]
var filter = curry(function(f, xs){
  return xs.filter(f);
});

//  reduce :: (b -> a -> b) -> b -> [a] -> b
var reduce = curry(function(f, x, xs){
  return xs.reduce(f, x);
});
```

`reduce` 可能是以上簽名裡讓人印象最爲深刻的一個，同時也是最複雜的一個了，所以如果你理解起來有困難的話，也不必氣餒。爲了滿足你的好奇心，我還是試著解釋一下吧；儘管我的解釋遠遠不如你自己通過型別簽名理解其含義來得有教益。

不保證解釋完全正確...（譯者注：此處原文是“here goes nothing”，一般用於人們在做沒有把握的事情之前說的話。）注意看 `reduce` 的簽名，可以看到它的第一個引數是個函式，這個函式接受一個 `b` 和一個 `a` 並返回一個 `b` 。那麼這些 `a` 和 `b` 是從哪來的呢？很簡單，簽名中的第二個和第三個引數就是 `b` 和元素爲 `a` 的陣列，所以唯一合理的假設就是這裡的 `b` 和每一個 `a` 都將傳給前面說的函式作爲引數。我們還可以看到， `reduce` 函式最後返回的結果是一個 `b` ，也就是說， `reduce` 的第一個引數函式的輸出就是 `reduce` 函式的輸出。知道了 `reduce` 的含義，我們才敢說上面關於型別簽名的推理是正確的。

縮小可能性範圍

一旦引入一個型別變數，就會出現一個奇怪的特性叫做 *parametricity*（http://en.wikipedia.org/wiki/Parametricity ）。這個特性表明，函式將會以一種統一的行爲作用於所有的型別。我們來研究下：

```
// head :: [a] -> a
```

注意看 `head`，可以看到它接受 `[a]` 返回 `a`。我們除了知道引數是個 `陣列`，其他的一概不知；所以函式的功能就只限於操作這個陣列上。在它對 `a` 一無所知的情況下，它可能對 `a` 做什麼操作呢？換句話說，`a` 告訴我們它不是一個 `特定` 的型別，這意味著它可以是 `任意` 型別；那麼我們的函式對每一個可能的型別的操作都必須保持統一。這就是 *parametricity* 的含義。要讓我們來猜測 `head` 的實現的話，唯一合理的推斷就是它返回陣列的第一個，或者最後一個，或者某個隨機的元素；當然，`head` 這個命名應該能給我們一些線索。

再看一個例子：

```
// reverse :: [a] -> [a]
```

僅從型別簽名來看，`reverse` 可能的目的是什麼？再次強調，它不能對 `a` 做任何特定的事情。它不能把 `a` 變成另一個型別，或者引入一個 `b`；這都是不可能的。那它可以排序麼？答案是不能，沒有足夠的資訊讓它去爲每一個可能的型別排序。它能重新排列麼？可以的，我覺得它可以，但它必須以一種可預料的方式達成目標。另外，它也有可能刪除或者重複某一個元素。重點是，不管在哪種情況下，型別 `a` 的多型性（polymorphism）都會大幅縮小 `reverse` 函式可能的行爲的範圍。

這種“可能性範圍的縮小”（narrowing of possibility）允許我們利用類似 Hoogle 這樣的型別簽名搜尋引擎去搜索我們想要的函式。型別簽名所能包含的資訊量眞的非常大。

自由定理

型別簽名除了能夠幫助我們推斷函式可能的實現，還能夠給我們帶來自由定理（free theorems）。下面是兩個直接從 Wadler 關於此主題的論文 中隨機選擇的例子：

```
// head :: [a] -> a
compose(f, head) == compose(head, map(f));

// filter :: (a -> Bool) -> [a] -> [a]
compose(map(f), filter(compose(p, f))) == compose(filter(p), map
(f));
```

不用寫一行程式碼你也能理解這些定理，它們直接來自於型別本身。第一個例子中，等式左邊說的是，先獲取陣列的 `頭部`（譯者注：即第一個元素），然後對它呼叫函式 `f`；等式右邊說的是，先對陣列中的每一個元素呼叫 `f`，然後再取其返回結果的 `頭部`。這兩個表示式的作用是相等的，但是前者要快得多。

你可能會想，這不是常識麼。但根據我的調查，計算機是沒有常識的。實際上，計算機必須要有一種形式化方法來自動進行類似的程式碼優化。數學提供了這種方法，能夠形式化直觀的感覺，這無疑對死板的計算機邏輯非常有用。

第二個例子 `filter` 也是一樣。等式左邊是說，先組合 `f` 和 `p` 檢查哪些元素要過濾掉，然後再通過 `map` 實際呼叫 `f`（別忘了 `filter` 是不會改變陣列中元素的，這就保證了 `a` 將保持不變）；等式右邊是說，先用 `map` 呼叫 `f`，然後再根據 `p` 過濾元素。這兩者也是相等的。

以上只是兩個例子，但它們傳達的定理卻是普適的，可以應用到所有的多型性型別簽名上。在 JavaScript 中，你可以藉助一些工具來宣告重寫規則，也可以直接使用 `compose` 函式來定義重寫規則。總之，這麼做的好處是顯而易見且唾手可得的，可能性則是無限的。

型別約束

最後要注意的一點是，簽名也可以把型別約束爲一個特定的介面（interface）。

```
// sort :: Ord a => [a] -> [a]
```

胖箭頭左邊表明的是這樣一個事實：`a` 一定是個 `Ord` 物件。也就是說，`a` 必須要實現 `Ord` 介面。`Ord` 到底是什麼？它是從哪來的？在一門強型別語言中，它可能就是一個自定義的介面，能夠讓不同的值排序。通過這種方式，我們不僅能夠獲取關於 `a` 的更多資訊，瞭解 `sort` 函式具體要幹什麼，而且還能限制函式的作用範圍。我們把這種介面宣告叫做型別約束（type constraints）。

```
// assertEqual :: (Eq a, Show a) => a -> a -> Assertion
```

這個例子中有兩個約束：`Eq` 和 `Show`。它們保證了我們可以檢查不同的 `a` 是否相等，並在有不相等的情況下打印出其中的差異。

我們將會在後面的章節中看到更多型別約束的例子，其含義也會更加清晰。

總結

Hindley-Milner 型別簽名在函數語言程式設計中無處不在，它們簡單易讀，寫起來也不復雜。但僅僅憑簽名就能理解整個程式還是有一定難度的，要想精通這個技能就更需要花點時間了。從這開始，我們將給每一行程式碼都加上型別簽名。

第 8 章: 特百惠

特百惠

（譯者注：特百惠是美國家居用品品牌，代表產品是塑料容器。）

強大的容器

我們已經知道如何書寫函數式的程式了，即通過管道把資料在一系列純函式間傳遞的程式。我們也知道了，這些程式就是宣告式的行爲規範。但是，控制流（control flow）、異常處理（error handling）、非同步操作（asynchronous actions）和狀態（state）呢？還有更棘手的作用（effects）呢？本章將對上述這些抽象概念賴以建立的基礎作一番探究。

首先我們將建立一個容器（container）。這個容器必須能夠裝載任意型別的值；否則的話，像只能裝木薯布丁的密封塑料袋是沒什麼用的。這個容器將會是一個物件，但我們不會爲它新增物件導向觀念下的屬性和方法。是的，我們將把它當作一

個百寶箱——一個存放寶貴的資料的特殊盒子。

```
var Container = function(x) {
  this.__value = x;
}

Container.of = function(x) { return new Container(x); };
```

這是本書的第一個容器，我們貼心地把它命名爲 `Container`。我們將使用 `Container.of` 作爲構造器（constructor），這樣就不用到處去寫糟糕的 `new` 關鍵字了，非常省心。實際上不能這麼簡單地看待 `of` 函式，但暫時先認爲它是把值放到容器裡的一種方式。

我們來檢驗下這個嶄新的盒子：

```
Container.of(3)
//=> Container(3)


Container.of("hotdogs")
//=> Container("hotdogs")


Container.of(Container.of({name: "yoda"}))
//=> Container(Container({name: "yoda" }))
```

如果用的是 node，那麼你會看到打印出來的是 `{__value: x}`，而不是實際值 `Container(x)`；Chrome 打印出來的是正確的。不過這並不重要，只要你理解 `Container` 是什麼樣的就行了。有些環境下，你也可以重寫 `inspect` 方法，但我們不打算涉及這方面的知識。在本書中，出於教學和美學上的考慮，我們將把概念性的輸出都寫成好像 `inspect` 被重寫了的樣子，因爲這樣寫的教育意義將遠遠大於 `{__value: x}`。

在繼續後面的內容之前，先澄清幾點：

- `Container` 是個只有一個屬性的物件。儘管容器可以有不止一個的屬性，但大多數容器還是隻有一個。我們很隨意地把 `Container` 的這個屬性命名爲 `__value`。

- `__value` 不能是某個特定的型別，不然 `Container` 就對不起它這個名字了。
- 資料一旦存放到 `Container`，就會一直待在那兒。我們可以用 `.__value` 獲取到資料，但這樣做有悖初衷。

如果把容器想象成玻璃罐的話，上面這三條陳述的理由就會比較清晰了。但是暫時，請先保持耐心。

第一個 functor

一旦容器裡有了值，不管這個值是什麼，我們就需要一種方法來讓別的函式能夠操作它。

```
// (a -> b) -> Container a -> Container b
Container.prototype.map = function(f){
  return Container.of(f(this.__value))
}
```

這個 `map` 跟陣列那個著名的 `map` 一樣，除了前者的引數是 `Container a` 而後者是 `[a]`。它們的使用方式也幾乎一致：

```
Container.of(2).map(function(two){ return two + 2 })
//=> Container(4)


Container.of("flamethrowers").map(function(s){ return s.toUpperC
ase() })
//=> Container("FLAMETHROWERS")


Container.of("bombs").map(concat(' away')).map(_.prop('length'))
//=> Container(10)
```

爲什麼要使用這樣一種方法？因爲我們能夠在不離開 `Container` 的情況下操作容器裡面的值。這是非常了不起的一件事情。`Container` 裡的值傳遞給 `map` 函式之後，就可以任我們操作；操作結束後，爲了防止意外再把它放回它所屬的

`Container`。這樣做的結果是，我們能連續地呼叫 `map`，執行任何我們想執行的函式。甚至還可以改變值的型別，就像上面最後一個例子中那樣。

等等，如果我們能一直呼叫 `map`，那它不就是個組合（composition）麼！這裡邊是有什麼數學魔法在起作用？是 *functor*。各位，這個數學魔法就是 *functor*。

> functor 是實現了 `map` 函式並遵守一些特定規則的容器型別。

沒錯，*functor* 就是一個簽了合約的介面。我們本來可以簡單地把它稱爲 `Mappable`，但現在爲時已晚，哪怕 *functor* 一點也不 *fun*。functor 是範疇學裡的概念，我們將在本章末尾詳細探索與此相關的數學知識；暫時我們先用這個名字很奇怪的介面做一些不那麼理論的、實用性的練習。

把值裝進一個容器，而且只能使用 `map` 來處理它，這麼做的理由到底是什麼呢？如果我們換種方式來問，答案就很明顯了：讓容器自己去運用函式能給我們帶來什麼好處？答案是抽象，對於函式運用的抽象。當 `map` 一個函式的時候，我們請求容器來執行這個函式。不誇張地講，這是一種十分強大的理念。

薛定諤的 **Maybe**

說實話 `Container` 挺無聊的，而且通常我們稱它爲 `Identity`，與 `id` 函式的作用相同（這裡也是有數學上的聯絡的，我們會在適當時候加以說明）。除此之外，還有另外一種 functor，那就是實現了 `map` 函式的類似容器的資料型別，這種 functor 在呼叫 `map` 的時候能夠提供非常有用的行爲。現在讓我們來定義一個這樣的 functor。

```
var Maybe = function(x) {
  this.__value = x;
}

Maybe.of = function(x) {
  return new Maybe(x);
}

Maybe.prototype.isNothing = function() {
  return (this.__value === null || this.__value === undefined);
}

Maybe.prototype.map = function(f) {
  return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this.__v
alue));
}
```

`Maybe` 看起來跟 `Container` 非常類似，但是有一點不同：`Maybe` 會先檢查自己的值是否爲空，然後才呼叫傳進來的函式。這樣我們在使用 `map` 的時候就能避免惱人的空值了（注意這個實現出於教學目的做了簡化）。

```
Maybe.of("Malkovich Malkovich").map(match(/a/ig));
//=> Maybe(['a', 'a'])

Maybe.of(null).map(match(/a/ig));
//=> Maybe(null)

Maybe.of({name: "Boris"}).map(_.prop("age")).map(add(10));
//=> Maybe(null)

Maybe.of({name: "Dinah", age: 14}).map(_.prop("age")).map(add(10
));
//=> Maybe(24)
```

注意看，當傳給 `map` 的值是 `null` 時，程式碼並沒有爆出錯誤。這是因爲每一次 `Maybe` 要呼叫函式的時候，都會先檢查它自己的值是否爲空。

這種點記法（dot notation syntax）已經足夠函數式了，但是正如在第 1 部分指出的那樣，我們更想保持一種 pointfree 的風格。碰巧的是，`map` 完全有能力以 curry 函式的方式來“代理”任何 functor：

```
//  map :: Functor f => (a -> b) -> f a -> f b
var map = curry(function(f, any_functor_at_all) {
  return any_functor_at_all.map(f);
});
```

這樣我們就可以像平常一樣使用組合，同時也能正常使用 `map` 了，非常振奮人心。ramda 的 `map` 也是這樣。後面的章節中，我們將在點記法更有教育意義的時候使用點記法，在方便使用 pointfree 模式的時候就用 pointfree。你注意到了麼？我在型別標籤中偷偷引入了一個額外的標記：`Functor f =>`。這個標記告訴我們 `f` 必須是一個 functor。沒什麼複雜的，但我覺得有必要提一下。

用例

實際當中，`Maybe` 最常用在那些可能會無法成功返回結果的函式中。

```
//  safeHead :: [a] -> Maybe(a)
var safeHead = function(xs) {
  return Maybe.of(xs[0]);
};

var streetName = compose(map(_.prop('street')), safeHead, _.prop(
'addresses'));

streetName({addresses: []});
// Maybe(null)

streetName({addresses: [{street: "Shady Ln.", number: 4201}]});
// Maybe("Shady Ln.")
```

`safeHead` 與一般的 `_.head` 類似，但是增加了型別安全保證。引入 `Maybe` 會發生一件非常有意思的事情，那就是我們被迫要與狡猾的 `null` 打交道了。`safeHead` 函式能夠誠實地預告它可能的失敗——失敗眞沒什麼可恥的——

然後返回一個 `Maybe` 來通知我們相關資訊。實際上不僅僅是通知，因爲畢竟我們想要的值深藏在 `Maybe` 物件中，而且只能通過 `map` 來操作它。本質上，這是一種由 `safeHead` 強制執行的空值檢查。有了這種檢查，我們才能在夜裡安然入睡，因爲我們知道最不受人待見的 `null` 不會突然出現。類似這樣的 API 能夠把一個像紙糊起來的、脆弱的應用升級爲實實在在的、健壯的應用，這樣的 API 保證了更加安全的軟體。

有時候函式可以明確返回一個 `Maybe(null)` 來表明失敗，例如：

```
//  withdraw :: Number -> Account -> Maybe(Account)
var withdraw = curry(function(amount, account) {
  return account.balance >= amount ?
    Maybe.of({balance: account.balance - amount}) :
    Maybe.of(null);
});

//  finishTransaction :: Account -> String
var finishTransaction = compose(remainingBalance, updateLedger);
// <- 假定這兩個函式已經在別處定義好了

//  getTwenty :: Account -> Maybe(String)
var getTwenty = compose(map(finishTransaction), withdraw(20));


getTwenty({ balance: 200.00});
// Maybe("Your balance is $180.00")

getTwenty({ balance: 10.00});
// Maybe(null)
```

要是錢不夠，`withdraw` 就會對我們嗤之以鼻然後返回一個 `Maybe(null)`。`withdraw` 也顯示出了它的多變性，使得我們後續的操作只能用 `map` 來進行。這個例子與前面例子不同的地方在於，這裡的 `null` 是有意的。我們不用 `Maybe(String)`，而是用 `Maybe(null)` 來發送失敗的訊號，這樣程式在收到訊號後就能立刻停止執行。這一點很重要：如果 `withdraw` 失敗

了，`map` 就會切斷後續程式碼的執行，因爲它根本就不會執行傳遞給它的函式，即 `finishTransaction`。這正是預期的效果：如果取款失敗，我們並不想更新或者顯示賬户餘額。

釋放容器裡的值

人們經常忽略的一個事實是：任何事物都有個最終盡頭。那些會產生作用的函式，不管它們是傳送 JSON 資料，還是在螢幕上列印東西，還是更改檔案系統，還是別的什麼，都要有一個結束。但是我們無法通過 `return` 把輸出傳遞到外部世界，必須要執行這樣或那樣的函式才能傳遞出去。關於這一點，可以借用禪宗公案的口吻來敘述：“如果一個程式執行之後沒有可觀察到的作用，那它到底運行了沒有？”。或者，執行之後達到自身的目的了沒有？有可能它只是浪費了幾個 CPU 週期然後就去睡覺了...

應用程式所做的工作就是獲取、更改和儲存資料直到不再需要它們，對資料做這些操作的函式有可能被 `map` 呼叫，這樣的話資料就可以不用離開它溫暖舒適的容器。諷刺的是，有一種常見的錯誤就是試圖以各種方法刪除 `Maybe` 裡的值，好像這個不確定的值是魔鬼，刪除它就能讓它突然顯形，然後一切罪惡都會得到寬恕似的（譯者注：此處原文應該是源自聖經）。要知道，我們的值沒有完成它的使命，很有可能是其他程式碼分支造成的。我們的程式碼，就像薛定諤的貓一樣，在某個特定的時間點有兩種狀態，而且應該保持這種狀況不變直到最後一個函式爲止。這樣，哪怕程式碼有很多邏輯性的分支，也能保證一種線性的工作流。

不過，對容器裡的值來說，還是有個逃生口可以出去。也就是說，如果我們想返回一個自定義的值然後還能繼續執行後面的程式碼的話，是可以做到的；要達到這一目的，可以藉助一個幫助函式 `maybe`：

```
//  maybe :: b -> (a -> b) -> Maybe a -> b
var maybe = curry(function(x, f, m) {
  return m.isNothing() ? x : f(m.__value);
});

//  getTwenty :: Account -> String
var getTwenty = compose(
  maybe("You're broke!", finishTransaction), withdraw(20)
);


getTwenty({ balance: 200.00});
// "Your balance is $180.00"

getTwenty({ balance: 10.00});
// "You're broke!"
```

這樣就可以要麼返回一個靜態值（與 `finishTransaction` 返回值的型別一致），要麼繼續愉快地在沒有 `Maybe` 的情況下完成交易。`maybe` 使我們得以避免普通 `map` 那種命令式的 `if/else` 語句：`if(x !== null) { return f(x) }`。

引入 `Maybe` 可能會在初期造成一些不適。Swift 和 Scala 使用者知道我在說什麼，因爲這兩門語言的核心庫裡就有 `Maybe` 的概念，只不過僞裝成 `Option(al)` 罷了。被迫在任何情況下都進行空值檢查（甚至有些時候我們可以確定某個值不會爲空），的確讓大部分人頭疼不已。然而隨著時間推移，空值檢查會成爲第二本能，說不定你還會感激它提供的安全性呢。不管怎麼說，空值檢查大多數時候都能防止在程式碼邏輯上偷工減料，讓我們脫離危險。

編寫不安全的軟體就像用蠟筆小心翼翼地畫彩蛋，畫完之後把它們扔到大街上一樣（譯者注：意思是彩蛋非常易於尋找。來源於復活節習俗，人們會藏起一些彩蛋讓孩子尋找），或者像用三隻小豬警告過的材料蓋個養老院一樣（譯者注：來源於“三隻小豬”童話故事）。`Maybe` 能夠非常有效地幫助我們增加函式的安全性。

有一點我必須要提及，否則就太不負責任了，那就是 `Maybe` 的“眞正”實現會把它分爲兩種型別：一種是非空值，另一種是空值。這種實現允許我們遵守 `map` 的 parametricity 特性，因此 `null` 和 `undefined` 能夠依然被 `map` 呼叫，

functor 裡的值所需的那種普遍性條件也能得到滿足。所以你會經常看到 `Some(x) / None` 或者 `Just(x) / Nothing` 這樣的容器型別在做空值檢查，而不是 `Maybe`。

“純”錯誤處理

說出來可能會讓你震驚，`throw/catch` 並不十分“純”。當一個錯誤丟擲的時候，我們沒有收到返回值，反而是得到了一個警告！拋錯的函式吐出一大堆的 0 和 1 作爲盾和矛來攻擊我們，簡直就像是在反擊輸入值的入侵而進行的一場電子大作戰。有了 `Either` 這個新朋友，我們就能以一種比向輸入值宣戰好得多的方式來處理錯誤，那就是返回一條非常禮貌的訊息作爲迴應。我們來看一下：

```
var Left = function(x) {
  this.__value = x;
}

Left.of = function(x) {
  return new Left(x);
}

Left.prototype.map = function(f) {
  return this;
}

var Right = function(x) {
  this.__value = x;
}

Right.of = function(x) {
  return new Right(x);
}

Right.prototype.map = function(f) {
  return Right.of(f(this.__value));
}
```

`Left` 和 `Right` 是我們稱之爲 `Either` 的抽象型別的兩個子類。我略去了建立 `Either` 父類的繁文縟節，因爲我們不會用到它的，但你瞭解一下也沒壞處。注意看，這裡除了有兩個型別，沒別的新鮮東西。來看看它們是怎麼執行的：

```
Right.of("rain").map(function(str){ return "b"+str; });
// Right("brain")

Left.of("rain").map(function(str){ return "b"+str; });
// Left("rain")

Right.of({host: 'localhost', port: 80}).map(_.prop('host'));
// Right('localhost')

Left.of("rolls eyes...").map(_.prop("host"));
// Left('rolls eyes...')
```

`Left` 就像是青春期少年那樣無視我們要 `map` 它的請求。`Right` 的作用就像是一個 `Container`（也就是 Identity）。這裡強大的地方在於，`Left` 有能力在它內部嵌入一個錯誤訊息。

假設有一個可能會失敗的函式，就拿根據生日計算年齡來說好了。的確，我們可以用 `Maybe(null)` 來表示失敗並把程式引向另一個分支，但是這並沒有告訴我們太多資訊。很有可能我們想知道失敗的原因是什麼。用 `Either` 寫一個這樣的程式看看：

```
var moment = require('moment');

//  getAge :: Date -> User -> Either(String, Number)
var getAge = curry(function(now, user) {
  var birthdate = moment(user.birthdate, 'YYYY-MM-DD');
  if(!birthdate.isValid()) return Left.of("Birth date could not
be parsed");
  return Right.of(now.diff(birthdate, 'years'));
});

getAge(moment(), {birthdate: '2005-12-12'});
// Right(9)

getAge(moment(), {birthdate: '20010704'});
// Left("Birth date could not be parsed")
```

這麼一來，就像 `Maybe(null)`，當返回一個 `Left` 的時候就直接讓程式短路。跟 `Maybe(null)` 不同的是，現在我們對程式爲何脫離原先軌道至少有了一點頭緒。有一件事要注意，這裡返回的是 `Either(String, Number)`，意味著我們這個 `Either` 左邊的值是 `String`，右邊（譯者注：也就是正確的值）的值是 `Number`。這個型別簽名不是很正式，因爲我們並沒有定義一個眞正的 `Either` 父類；但我們還是從這個型別那裡瞭解到不少東西。它告訴我們，我們得到的要麼是一條錯誤訊息，要麼就是正確的年齡值。

```
//  fortune :: Number -> String
var fortune  = compose(concat("If you survive, you will be "), a
dd(1));

//  zoltar :: User -> Either(String, _)
var zoltar = compose(map(console.log), map(fortune), getAge(mome
nt()));

zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 10"
// Right(undefined)

zoltar({birthdate: 'balloons!'});
// Left("Birth date could not be parsed")
```

如果 `birthdate` 合法，這個程式就會把它神祕的命運列印在螢幕上讓我們見證；如果不合法，我們就會收到一個有著清清楚楚的錯誤訊息的 `Left`，儘管這個訊息是穩穩當當地待在它的容器裡的。這種行爲就像，雖然我們在拋錯，但是是以一種平靜溫和的方式拋錯，而不是像一個小孩子那樣，有什麼不對勁就鬧脾氣大喊大叫。

在這個例子中，我們根據 `birthdate` 的合法性來控制程式碼的邏輯分支，同時又讓程式碼進行從右到左的直線運動，而不用爬過各種條件語句的大括號。通常，我們不會把 `console.log` 放到 `zoltar` 函式裡，而是在呼叫 `zoltar` 的時候才 `map` 它，不過本例中，讓你看看 `Right` 分支如何與 `Left` 不同也是很有幫助的。我們在 `Right` 分支的型別簽名中使用 `_` 表示一個應該忽略的值（在有些瀏覽器中，你必須要 `console.log.bind(console)` 才能把 `console.log` 當作一等公民使用）。

我想借此機會指出一件你可能沒注意到的事：這個例子中，儘管 `fortune` 使用了 `Either`，它對每一個 functor 到底要幹什麼卻是毫不知情的。前面例子中的 `finishTransaction` 也是一樣。通俗點來講，一個函式在呼叫的時候，如果被 `map` 包裹了，那麼它就會從一個非 functor 函式轉換爲一個 functor 函式。我們把這個過程叫做 *lift*。一般情況下，普通函式更適合操作普通的資料型別而不是容器型別，在必要的時候再通過 *lift* 變爲合適的容器去操作容器型別。這樣做的好處是能得到更簡單、重用性更高的函式，它們能夠隨需求而變，相容任意 functor。

`Either` 並不僅僅只對合法性檢查這種一般性的錯誤作用非凡，對一些更嚴重的、能夠中斷程式執行的錯誤比如檔案丟失或者 socket 連線斷開等，`Either` 同樣效果顯著。你可以試試把前面例子中的 `Maybe` 替換爲 `Either`，看怎麼得到更好的反饋。

此刻我忍不住在想，我僅僅是把 `Either` 當作一個錯誤訊息的容器介紹給你！這樣的介紹有失偏頗，它的能耐遠不止於此。比如，它表示了邏輯或（也就是 `||`）。再比如，它體現了範疇學裡 *coproduct* 的概念，當然本書不會涉及這方面的知識，但值得你去深入瞭解，因爲這個概念有很多特性值得利用。還比如，它是標準的 sum type（或者叫不交併集，disjoint union of sets），因爲它含有的所有可能的值的總數就是它包含的那兩種型別的總數（我知道這麼說你聽不懂，沒關係，這裡有一篇非常棒的文章講述這個問題）。`Either` 能做的事情多著呢，但是作爲一個 functor，我們就用它處理錯誤。

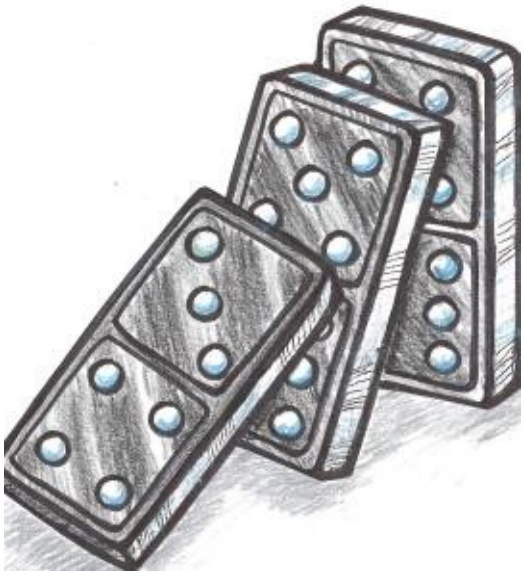
就像 `Maybe` 可以有個 `maybe` 一樣，`Either` 也可以有一個 `either`。兩者的用法類似，但 `either` 接受兩個函式（而不是一個）和一個靜態值爲引數。這兩個函式的返回值型別一致：

```
//  either :: (a -> c) -> (b -> c) -> Either a b -> c
var either = curry(function(f, g, e) {
  switch(e.constructor) {
    case Left: return f(e.__value);
    case Right: return g(e.__value);
  }
});

//  zoltar :: User -> _
var zoltar = compose(console.log, either(id, fortune), getAge(mo
ment()));

zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 10"
// undefined

zoltar({birthdate: 'balloons!'});
// "Birth date could not be parsed"
// undefined
```

終於用了一回那個神祕的 `id` 函式！其實它就是簡單地複製了 `Left` 裡的錯誤訊息，然後把這個值傳給 `console.log` 而已。通過強制在 `getAge` 內部進行錯誤處理，我們的算命程式更加健壯了。結果就是，要麼告訴使用者一個殘酷的事實並像算命師那樣跟他擊掌，要麼就繼續執行程式。好了，現在我們已經準備好去學習一個完全不同型別的 functor 了。

王老先生有作用...

（譯者注：原標題是“Old McDonald had Effects...”，源於美國兒歌“Old McDonald Had a Farm”。）



在關於純函式的的那一章（即第 3 章）裡，有一個很奇怪的例子。這個例子中的函式會產生副作用，但是我們通過把它包裹在另一個函式裡的方式把它變得看起來像一個純函式。這裡還有一個類似的例子：

```
//  getFromStorage :: String -> (_ -> String)
var getFromStorage = function(key) {
  return function() {
    return localStorage[key];
  }
}
```

要是我們沒把 `getFromStorage` 包在另一個函式裡，它的輸出值就是不定的，會隨外部環境變化而變化。有了這個結實的包裹函式（wrapper），同一個輸入就總能返回同一個輸出：一個從 `localStorage` 裡取出某個特定的元素的函式。就這樣（也許再高唱幾句讚美聖母的讚歌）我們洗滌了心靈，一切都得到了寬恕。

然而，這並沒有多大的用處，你說是不是。就像是你收藏的全新未拆封的玩偶，不能拿出來玩有什麼意思。所以要是能有辦法進到這個容器裡面，拿到它藏在那兒的東西就好了...辦法是有的，請看 `IO` ：

```
var IO = function(f) {
  this.__value = f;
}

IO.of = function(x) {
  return new IO(function() {
    return x;
  });
}

IO.prototype.map = function(f) {
  return new IO(_.compose(f, this.__value));
}
```

`IO` 跟之前的 functor 不同的地方在於，它的 `__value` 總是一個函式。不過我們不把它當作一個函式——實現的細節我們最好先不管。這裡發生的事情跟我們在 `getFromStorage` 那裡看到的一模一樣：`IO` 把非純執行動作（impure action）捕獲到包裹函式裡，目的是延遲執行這個非純動作。就這一點而言，我們認爲 `IO` 包含的是被包裹的執行動作的返回值，而不是包裹函式本身。這在 `of` 函式裡很明顯：`IO(function(){ return x })` 僅僅是爲了延遲執行，其實我們得到的是 `IO(x)`。

來用用看：

```
//  io_window_ :: IO Window
var io_window = new IO(function(){ return window; });

io_window.map(function(win){ return win.innerWidth });
// IO(1430)

io_window.map(_.prop('location')).map(_.prop('href')).map(split(
'/'));
// IO(["http:", "", "localhost:8000", "blog", "posts"])


//  $ :: String -> IO [DOM]
var $ = function(selector) {
  return new IO(function(){ return document.querySelectorAll(sel
ector); });
}

$('#myDiv').map(head).map(function(div){ return div.innerHTML; }
);
// IO('I am some inner html')
```

這裡，`io_window` 是一個眞正的 `IO`，我們可以直接對它使用 `map`。至於 `$`，則是一個函式，呼叫後會返回一個 `IO`。我把這裡的返回值都寫成了概念性的，這樣就更加直觀；不過實際的返回值是 `{ __value: [Function] }`。當呼叫 `IO` 的 `map` 的時候，我們把傳進來的函式放在了 `map` 函式裡的組合的最末端（也就是最左邊），反過來這個函式就成爲了新的 `IO` 的新 `__value`，並繼續下去。傳給 `map` 的函式並沒有執行，我們只是把它們壓到一個“執行棧”的最末端而已，一個函式緊挨著另一個函式，就像小心擺放的多米諾骨牌一樣，讓人不敢輕易推倒。這種情形很容易叫人聯想起“四人幫”（譯者注：《設計模式》一書作者）提出的命令模式（command pattern）或者佇列（queue）。

花點時間找回你關於 functor 的直覺吧。把實現細節放在一邊不管，你應該就能自然而然地對各種各樣的容器使用 `map` 了，不管它是多麼奇特怪異。這種僞超自然的力量要歸功於 functor 的定律，我們將在本章末尾對此作一番探索。無論如何，我們終於可以在不犧牲程式碼純粹性的情況下，隨意使用這些不純的值了。

好了，我們已經把野獸關進了籠子。但是，在某一時刻還是要把它放出來。因爲對 `IO` 呼叫 `map` 已經積累了太多不純的操作，最後再執行它無疑會打破平靜。問題是在哪裡，什麼時候開啓籠子的開關？而且有沒有可能我們只執行 `IO` 卻不讓不純的操作弄髒雙手？答案是可以的，只要把責任推到呼叫者身上就行了。我們的純程式碼，儘管陰險狡詐詭計多端，但是卻始終保持一副清白無辜的模樣，反而是實際執行 `IO` 併產生了作用的呼叫者，背了黑鍋。來看一個具體的例子。

```
////// 純程式碼庫: lib/params.js ///////

//  url :: IO String
var url = new IO(function() { return window.location.href; });

//  toPairs =  String -> [[String]]
var toPairs = compose(map(split('=')), split('&'));

//  params :: String -> [[String]]
var params = compose(toPairs, last, split('?'));

//  findParam :: String -> IO Maybe [String]
var findParam = function(key) {
  return map(compose(Maybe.of, filter(compose(eq(key), head)), p
arams), url);
};

////// 非純呼叫程式碼: main.js ///////

// 呼叫 __value() 來執行它！
findParam("searchTerm").__value();
// Maybe(['searchTerm', 'wafflehouse'])
```

lib/params.js 把 `url` 包裹在一個 `IO` 裡，然後把這頭野獸傳給了呼叫者；一雙手保持的非常乾淨。你可能也注意到了，我們把容器也“壓棧”了，要知道建立一個 `IO(Maybe([x]))` 沒有任何不合理的地方。我們這個“棧”有三層 functor（ `Array` 是最有資格成爲 mappable 的容器型別），令人印象深刻。

有件事困擾我很久了，現在我必須得說出來：`IO` 的 `__value` 並不是它包含的值，也不是像兩個下劃線暗示那樣是一個私有屬性。`__value` 是手榴彈的彈栓，只應該被呼叫者以最公開的方式拉動。爲了提醒使用者它的變化無常，我們把它重新命名爲 `unsafePerformIO` 看看。

```
var IO = function(f) {
  this.unsafePerformIO = f;
}

IO.prototype.map = function(f) {
  return new IO(_.compose(f, this.unsafePerformIO));
}
```

看，這就好多了。現在呼叫的程式碼就變成了 `findParam("searchTerm").unsafePerformIO()`，對應用程式的使用者（以及本書讀者）來說，這簡直就直白得不能再直白了。

`IO` 會成爲一個忠誠的伴侶，幫助我們馴化那些狂野的非純操作。下一節我們將學習一種跟 `IO` 在精神上相似，但是用法上又千差萬別的型別。

非同步任務

回撥（callback）是通往地獄的狹窄的螺旋階梯。它們是埃舍爾（譯者注：荷蘭版畫藝術家）設計的控制流。看到一個個巢狀的回撥擠在大小括號搭成的架子上，讓人不由自主地聯想到地牢裡的靈薄獄（還能再低點麼！）（譯者注：靈薄獄即 limbo，基督教中地獄邊緣之意）。光是想到這樣的回撥就讓我幽閉恐怖症發作了。不過別擔心，處理非同步程式碼，我們有一種更好的方式，它的名字以“F”開頭。

這種方式的內部機制過於複雜，複雜得哪怕我唾沫橫飛也很難講清楚。所以我們就直接用 Quildreen Motta 的 Folktale 裡的 `Data.Task`（之前是 `Data.Future`）。來見證一些例子吧：

```
// Node readfile example:
//======================

var fs = require('fs');

//  readFile :: String -> Task(Error, JSON)
var readFile = function(filename) {
  return new Task(function(reject, result) {
    fs.readFile(filename, 'utf-8', function(err, data) {
      err ? reject(err) : result(data);
    });
  });
};

readFile("metamorphosis").map(split('\n')).map(head);
// Task("One morning, as Gregor Samsa was waking up from anxious
 dreams, he discovered that
// in bed he had been changed into a monstrous verminous bug.")


// jQuery getJSON example:
//=======================

//  getJSON :: String -> {} -> Task(Error, JSON)
var getJSON = curry(function(url, params) {
  return new Task(function(reject, result) {
    $.getJSON(url, params, result).fail(reject);
  });
});

getJSON('/video', {id: 10}).map(_.prop('title'));
// Task("Family Matters ep 15")

// 傳入普通的實際值也沒問題
Task.of(3).map(function(three){ return three + 1 });
// Task(4)
```

例子中的 `reject` 和 `result` 函式分別是失敗和成功的回撥。正如你看到的，我們只是簡單地呼叫 `Task` 的 `map` 函式，就能操作將來的值，好像這個值就在那兒似的。到現在 `map` 對你來說應該不稀奇了。

如果熟悉 promise 的話，你該能認出來 `map` 就是 `then`，`Task` 就是一個 promise。如果不熟悉你也不必氣餒，反正我們也不會用它，因爲它並不純；但剛才的類比還是成立的。

與 `IO` 類似，`Task` 在我們給它綠燈之前是不會執行的。事實上，正因爲它要等我們的命令，`IO` 實際就被納入到了 `Task` 名下，代表所有的非同步操作——`readFile` 和 `getJSON` 並不需要一個額外的 `IO` 容器來變純。更重要的是，當我們呼叫它的 `map` 的時候，`Task` 工作的方式與 `IO` 幾無差別：都是把對未來的操作的指示放在一個時間膠囊裡，就像家務列表（chore chart）那樣——眞是一種精密的拖延術。

我們必須呼叫 `fork` 方法才能執行 `Task`，這種機制與 `unsafePerformIO` 類似。但也有不同，不同之處就像 `fork` 這個名稱表明的那樣，它會 fork 一個子程序執行它接收到的引數程式碼，其他部分的執行不受影響，主執行緒也不會阻塞。當然這種效果也可以用其他一些技術比如執行緒實現，但這裡的這種方法工作起來就像是一個普通的非同步呼叫，而且 event loop 能夠不受影響地繼續運轉。我們來看一下 `fork`：

```
// Pure application
//====================
// blogTemplate :: String

//  blogPage :: Posts -> HTML
var blogPage = Handlebars.compile(blogTemplate);

//  renderPage :: Posts -> HTML
var renderPage = compose(blogPage, sortBy('date'));

//  blog :: Params -> Task(Error, HTML)
var blog = compose(map(renderPage), getJSON('/posts'));


// Impure calling code
//====================
blog({}).fork(
  function(error){ $("#error").html(error.message); },
  function(page){ $("#main").html(page); }
);

$('#spinner').show();
```

呼叫 `fork` 之後，`Task` 就趕緊跑去找一些文章，渲染到頁面上。與此同時，我們在頁面上展示一個 spinner，因爲 `fork` 不會等收到響應了才執行它後面的程式碼。最後，我們要麼把文章展示在頁面上，要麼就顯示一個出錯資訊，視 `getJSON` 請求是否成功而定。

花點時間思考下這裡的控制流爲何是線性的。我們只需要從下讀到上，從右讀到左就能理解程式碼，即便這段程式實際上會在執行過程中到處跳來跳去。這種方式使得閱讀和理解應用程式的程式碼比那種要在各種回撥和錯誤處理程式碼塊之間跳躍的方式容易得多。

天哪，你看到了麼，`Task` 居然也包含了 `Either`！沒辦法，爲了能處理將來可能出現的錯誤，它必須得這麼做，因爲普通的控制流在非同步的世界裡不適用。這自然是好事一樁，因爲它天然地提供了充分的“純”錯誤處理。

就算是有了 `Task`，`IO` 和 `Either` 這兩個 functor 也照樣能派上用場。待我舉個簡單例子向你說明一種更復雜、更假想的情況，雖然如此，這個例子還是能夠說明我的目的。

```
// Postgres.connect :: Url -> IO DbConnection
// runQuery :: DbConnection -> ResultSet
// readFile :: String -> Task Error String

// Pure application
//=====================

//  dbUrl :: Config -> Either Error Url
var dbUrl = function(c) {
  return (c.uname && c.pass && c.host && c.db)
    ? Right.of("db:pg://"+c.uname+":"+c.pass+"@"+c.host+"5432/"+
c.db)
    : Left.of(Error("Invalid config!"));
}

//  connectDb :: Config -> Either Error (IO DbConnection)
var connectDb = compose(map(Postgres.connect), dbUrl);

//  getConfig :: Filename -> Task Error (Either Error (IO DbConn
ection))
var getConfig = compose(map(compose(connectDB, JSON.parse)), rea
dFile);


// Impure calling code
//=====================
getConfig("db.json").fork(
  logErr("couldn't read file"), either(console.log, map(runQuery
))
);
```

這個例子中，我們在 `readFile` 成功的那個程式碼分支裡利用了 `Either` 和 `IO`。`Task` 處理非同步讀取檔案這一操作當中的不“純”性，但是驗證 config 的合法性以及連線資料庫則分別使用了 `Either` 和 `IO`。所以你看，我們依然在同步地跟所有事物打交道。

例子我還可以再舉一些，但是就到此爲止吧。這些概念就像 `map` 一樣簡單。

實際當中，你很有可能在一個工作流中跑好幾個非同步任務，但我們還沒有完整學習容器的 api 來應對這種情況。不必擔心，我們很快就會去學習 monad 之類的概念。不過，在那之前，我們得先檢查下所有這些背後的數學知識。

一點理論

前面提到，functor 的概念來自於範疇學，並滿足一些定律。我們先來探索這些實用的定律。

```
// identity
map(id) === id;

// composition
compose(map(f), map(g)) === map(compose(f, g));
```

同一律很簡單，但是也很重要。因爲這些定律都是可執行的程式碼，所以我們完全可以在我們自己的 functor 上試驗它們，驗證它們是否成立。

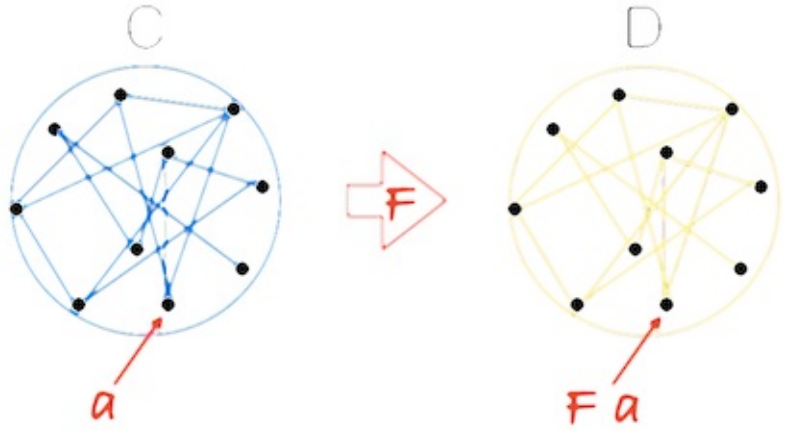
```
var idLaw1 = map(id);
var idLaw2 = id;

idLaw1(Container.of(2));
//=> Container(2)

idLaw2(Container.of(2));
//=> Container(2)
```

看到沒，它們是相等的。接下來看一看組合。

```
var compLaw1 = compose(map(concat(" world")), map(concat(" cruel"
)));
var compLaw2 = map(compose(concat(" world"), concat(" cruel")));

compLaw1(Container.of("Goodbye"));
//=> Container('Goodbye cruel world')

compLaw2(Container.of("Goodbye"));
//=> Container('Goodbye cruel world')
```

在範疇學中，functor 接受一個範疇的物件和態射（morphism），然後把它們對映（map）到另一個範疇裡去。根據定義，這個新範疇一定會有一個單位元（identity），也一定能夠組合態射；我們無須驗證這一點，前面提到的定律保證這些東西會在對映後得到保留。

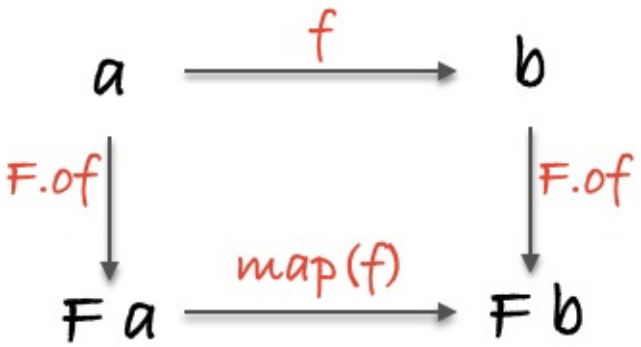
可能我們關於範疇的定義還是有點模糊。你可以把範疇想象成一個有著多個物件的網路，物件之間靠態射連線。那麼 functor 可以把一個範疇對映到另外一個，而且不會破壞原有的網路。如果一個物件 `a` 屬於源範疇 `C`，那麼通過 functor `F` 把 `a` 對映到目標範疇 `D` 上之後，就可以使用 `F a` 來指代 `a` 物件（把這些字母拼起來是什麼？！）。可能看圖會更容易理解：



比如，`Maybe` 就把型別和函式的範疇對映到這樣一個範疇：即每個物件都有可能不存在，每個態射都有空值檢查的範疇。這個結果在程式碼中的實現方式是用 `map` 包裹每一個函式，用 functor 包裹每一個型別。這樣就能保證每個普通的型別和函式都能在新環境下繼續使用組合。從技術上講，程式碼中的 functor 實際上

是把範疇對映到了一個包含型別和函式的子範疇（sub category）上，使得這些 functor 成爲了一種新的特殊的 endofunctor。但出於本書的目的，我們認爲它就是一個不同的範疇。

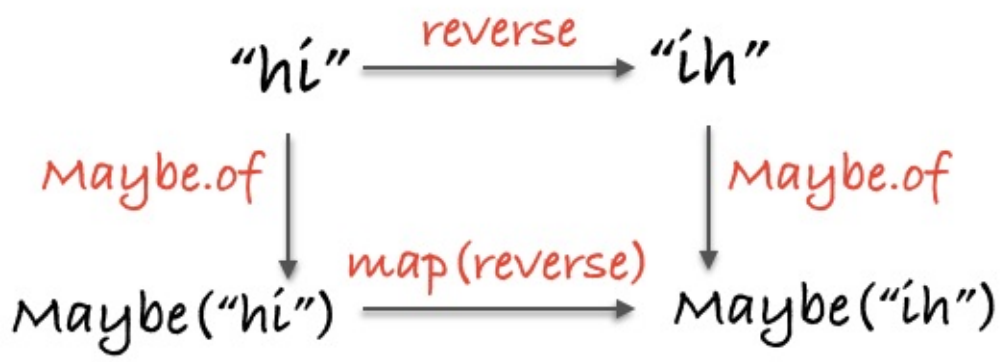
可以用一張圖來表示這種態射及其物件的對映：



這張圖除了能表示態射藉助 functor `F` 完成從一個範疇到另一個範疇的對映之外，我們發現它還符合交換律，也就是說，順著箭頭的方向往前，形成的每一個路徑都指向同一個結果。不同的路徑意味著不同的行爲，但最終都會得到同一個資料型別。這種形式化給了我們原則性的方式去思考程式碼——無須分析和評估每一個單獨的場景，只管可以大膽地應用公式即可。來看一個具體的例子。

```
//  topRoute :: String -> Maybe(String)
var topRoute = compose(Maybe.of, reverse);

//  bottomRoute :: String -> Maybe(String)
var bottomRoute = compose(map(reverse), Maybe.of);


topRoute("hi");
// Maybe("ih")

bottomRoute("hi");
// Maybe("ih")
```

或者看圖：



根據所有 functor 都有的特性，我們可以立即理解程式碼，重構程式碼。

functor 也能巢狀使用：

```
var nested = Task.of([Right.of("pillows"), Left.of("no sleep for
 you")]);

map(map(map(toUpperCase)), nested);
// Task([Right("PILLOWS"), Left("no sleep for you")])
```

`nested` 是一個將來的陣列，陣列的元素有可能是程式丟擲的錯誤。我們使用 `map` 剝開每一層的巢狀，然後對陣列的元素呼叫傳遞進去的函式。可以看到，這中間沒有回撥、`if/else` 語句和 `for` 迴圈，只有一個明確的上下文。的確，我們必須要 `map(map(map(f)))` 才能最終執行函式。不想這麼做的話，可以組合 functor。是的，你沒聽錯：

```
var Compose = function(f_g_x){
  this.getCompose = f_g_x;
}

Compose.prototype.map = function(f){
  return new Compose(map(map(f), this.getCompose));
}

var tmd = Task.of(Maybe.of("Rock over London"))

var ctmd = new Compose(tmd);

map(concat(", rock on, Chicago"), ctmd);
// Compose(Task(Maybe("Rock over London, rock on, Chicago")))

ctmd.getCompose;
// Task(Maybe("Rock over London, rock on, Chicago"))
```

看，只有一個 `map`。functor 組合是符合結合律的，而且之前我們定義的 `Container` 實際上是一個叫 `Identity` 的 functor。identity 和可結合的組合也能產生一個範疇，這個特殊的範疇的物件是其他範疇，態射是 functor。這實在太傷腦筋了，所以我們不會深入這個問題，但是讚歎一下這種模式的結構性含義，或者它的簡單的抽象之美也是好的。

總結

我們已經認識了幾個不同的 functor，但它們的數量其實是無限的。有一些值得注意的可迭代資料型別（iterable data structure）我們沒有介紹，像 tree、list、map 和 pair 等，以及所有你能說出來的。eventstream 和 observable 也都是 functor。其他的 functor 可能就是拿來做封裝或者僅僅是模擬型別。我們身邊到處都有 functor 的身影，本書也將會大量使用它們。

用多個 functor 引數呼叫一個函式怎麼樣呢？處理一個由不純的或者非同步的操作組成的有序序列怎麼樣呢？要應對這個什麼都裝在盒子裡的世界，目前我們工具箱裡的工具還不全。下一章，我們將直奔 monad 而去。

第 9 章: Monad

練習

```
require('../../support');
var Task = require('data.task');
var _ = require('ramda');

// 練習 1
// ==========
// 使用 _.add(x,y) 和 _.map(f,x) 建立一個能讓 functor 裡的值增加的函式

var ex1 = undefined



//練習 2
// ==========
// 使用 _.head 獲取列表的第一個元素
var xs = Identity.of(['do', 'ray', 'me', 'fa', 'so', 'la', 'ti',
'do']);

var ex2 = undefined



// 練習 3
// ==========
// 使用 safeProp 和 _.head 找到 user 的名字的首字母
var safeProp = _.curry(function (x, o) { return Maybe.of(o[x]);
});

var user = { id: 2, name: "Albert" };

var ex3 = undefined


// 練習 4
// ==========
// 使用 Maybe 重寫 ex4，不要有 if 語句
```

```
var ex4 = function (n) {
  if (n) { return parseInt(n); }
};

var ex4 = undefined



// 練習 5
// ==========
// 寫一個函式，先 getPost 獲取一篇文章，然後 toUpperCase 讓這片文章標題變爲大寫

// getPost :: Int -> Future({id: Int, title: String})
var getPost = function (i) {
  return new Task(function(rej, res) {
    setTimeout(function(){
      res({id: i, title: 'Love them futures'})
    }, 300)
  });
}

var ex5 = undefined



// 練習 6
// ==========
// 寫一個函式，使用 checkActive() 和 showWelcome() 分別允許訪問或返回錯誤

var showWelcome = _.compose(_.add( "Welcome "), _.prop('name'))

var checkActive = function(user) {
 return user.active ? Right.of(user) : Left.of('Your account is not active')
}

var ex6 = undefined
```

```
// 練習 7
// ==========
// 寫一個驗證函式，檢查引數是否 length > 3。如果是就返回 Right(x)，否則就返回
// Left("You need > 3")

var ex7 = function(x) {
  return undefined // <--- write me. (don't be pointfree)
}



// 練習 8
// ==========
// 使用練習 7 的 ex7 和 Either 構造一個 functor，如果一個 user 合法就儲存它，否則
// 返回錯誤訊息。別忘了 either 的兩個引數必須返回同一型別的資料。

var save = function(x){
  return new IO(function(){
    console.log("SAVED USER!");
    return x + '-saved';
  });
}

var ex8 = undefined
```

Monad

pointed functor

在繼續後面的內容之前，我得向你坦白一件事：關於我們先前建立的容器型別上的 `of` 方法，我並沒有說出它的全部實情。眞實情況是，`of` 方法不是用來避免使用 `new` 關鍵字的，而是用來把值放到預設最小化上下文（default minimal context）中的。是的，`of` 沒有眞正地取代構造器——它是一個我們稱之爲 *pointed* 的重要介面的一部分。

> *pointed functor* 是實現了 `of` 方法的 functor。

這裡的關鍵是把任意值丟到容器裡然後開始到處使用 `map` 的能力。

```
IO.of("tetris").map(concat(" master"));
// IO("tetris master")

Maybe.of(1336).map(add(1));
// Maybe(1337)

Task.of([{id: 2}, {id: 3}]).map(_.prop('id'));
// Task([2,3])

Either.of("The past, present and future walk into a bar...").map
(
  concat("it was tense.")
);
// Right("The past, present and future walk into a bar...it was
tense.")
```

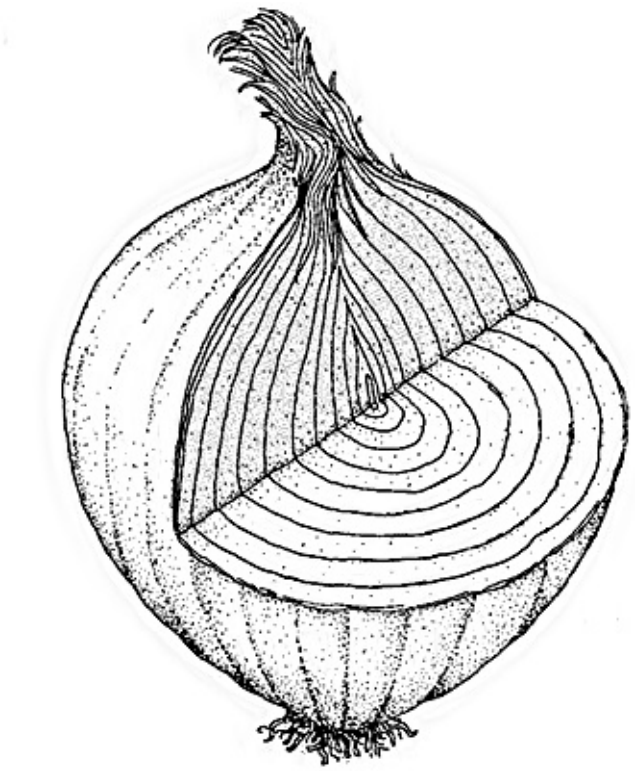
如果你還記得，`IO` 和 `Task` 的構造器接受一個函式作爲引數，而 `Maybe` 和 `Either` 的構造器可以接受任意值。實現這種介面的動機是，我們希望能有一種通用、一致的方式往 functor 裡填值，而且中間不會涉及到複雜性，也不會涉及到對構造器的特定要求。“預設最小化上下文”這個術語可能不夠精確，但是卻很好地傳達了這種理念：我們希望容器型別裡的任意值都能發生 `lift`，然後像所有的 functor 那樣再 `map` 出去。

有件很重要的事我必須得在這裡糾正，那就是，`Left.of` 沒有任何道理可言，包括它的雙關語也是。每個 functor 都要有一種把值放進去的方式，對 `Either` 來說，它的方式就是 `new Right(x)`。我們爲 `Right` 定義 `of` 的原因是，如果一個型別容器可以 `map`，那它就應該 `map`。看上面的例子，你應該會對 `of` 通常的工作模式有一個直觀的印象，而 `Left` 破壞了這種模式。

你可能已經聽說過 `pure`、`point`、`unit` 和 `return` 之類的函數了，它們都是 `of` 這個史上最神祕函式的不同名稱（譯者注：此處原文是“international function of mystery”，源自惡搞《007》的電影 *Austin Powers: International Man of Mystery*，中譯名《王牌大賤諜》）。`of` 將在我們開始使用 monad 的時候顯示其重要性，因爲後面你會看到，手動把值放回容器是我們自己的責任。

要避免 `new` 關鍵字，可以藉助一些標準的 JavaScript 技巧或者類庫達到目的。所以從這裡開始，我們就利用這些技巧或類庫，像一個負責任的成年人那樣使用 `of`。我推薦使用 `folktale`、`ramda` 或 `fantasy-land` 裡的 functor 例項，因爲它們同時提供了正確的 `of` 方法和不依賴 `new` 的構造器。

混合比喻



你看，除了太空墨西哥卷（如果你聽說過這個傳言的話）（譯者注：此處的傳言似乎是說一個叫 Chris Hadfield 的宇航員在國際空間站做墨西哥卷的事，視訊連結），monad 還被喻爲洋蔥。讓我以一個常見的場景來說明這點：

```
// Support
// ===========================
var fs = require('fs');

//  readFile :: String -> IO String
var readFile = function(filename) {
  return new IO(function() {
    return fs.readFileSync(filename, 'utf-8');
  });
};

//  print :: String -> IO String
var print = function(x) {
  return new IO(function() {
    console.log(x);
    return x;
  });
}

// Example
// ===========================
//  cat :: IO (IO String)
var cat = compose(map(print), readFile);

cat(".git/config")
// IO(IO("[core]\nrepositoryformatversion = 0\n"))
```

這裡我們得到的是一個 `IO`，只不過它陷進了另一個 `IO`。要想使用它，我們必須這樣呼叫：`map(map(f))`；要想觀察它的作用，必須這樣：`unsafePerformIO().unsafePerformIO()`。

```
//  cat :: String -> IO (IO String)
var cat = compose(map(print), readFile);

//  catFirstChar :: String -> IO (IO String)
var catFirstChar = compose(map(map(head)), cat);

catFirstChar(".git/config")
// IO(IO("["))
```

儘管在應用中把這兩個作用打包在一起沒什麼不好的，但總感覺像是在穿著兩套防護服工作，結果就形成一個稀奇古怪的 API。再來看另一種情況：

```
//  safeProp :: Key -> {Key: a} -> Maybe a
var safeProp = curry(function(x, obj) {
  return new Maybe(obj[x]);
});

//  safeHead :: [a] -> Maybe a
var safeHead = safeProp(0);

//  firstAddressStreet :: User -> Maybe (Maybe (Maybe Street) )
var firstAddressStreet = compose(
  map(map(safeProp('street'))), map(safeHead), safeProp('address
es')
);

firstAddressStreet(
  {addresses: [{street: {name: 'Mulburry', number: 8402}, postco
de: "WC2N" }]}
);
// Maybe(Maybe(Maybe({name: 'Mulburry', number: 8402})))
```

這裡的 functor 同樣是巢狀的，函式中三個可能的失敗都用了 `Maybe` 做預防也很乾淨整潔，但是要讓最後的呼叫者呼叫三次 `map` 才能取到值未免也太無禮了點——我們和它才剛剛見面而已。這種巢狀 functor 的模式會時不時地出現，而且是 monad 的主要使用場景。

我說過 monad 像洋蔥，那是因爲當我們用 `map` 剝開巢狀的 functor 以獲取它裡面的值的時候，就像剝洋蔥一樣讓人忍不住想哭。不過，我們可以擦乾眼淚，做個深呼吸，然後使用一個叫作 `join` 的方法。

```
var mmo = Maybe.of(Maybe.of("nunchucks"));
// Maybe(Maybe("nunchucks"))

mmo.join();
// Maybe("nunchucks")

var ioio = IO.of(IO.of("pizza"));
// IO(IO("pizza"))

ioio.join()
// IO("pizza")

var ttt = Task.of(Task.of(Task.of("sewers")));
// Task(Task(Task("sewers")));

ttt.join()
// Task(Task("sewers"))
```

如果有兩層相同型別的巢狀，那麼就可以用 `join` 把它們壓扁到一塊去。這種結合的能力，functor 之間的聯姻，就是 monad 之所以成爲 monad 的原因。來看看它更精確的完整定義：

> monad 是可以變扁（flatten）的 pointed functor。

一個 functor，只要它定義個了一個 `join` 方法和一個 `of` 方法，並遵守一些定律，那麼它就是一個 monad。`join` 的實現並不太複雜，我們來爲 `Maybe` 定義一個：

```
Maybe.prototype.join = function() {
  return this.isNothing() ? Maybe.of(null) : this.__value;
}
```

看，就像子宮裡雙胞胎中的一個吃掉另一個那麼簡單。如果有一個 `Maybe(Maybe(x))`，那麼 `.__value` 將會移除多餘的一層，然後我們就能安心地從那開始進行 `map`。要不然，我們就將會只有一個 `Maybe`，因爲從一開始就沒有任何東西被 `map` 呼叫。

既然已經有了 `join` 方法，我們把 monad 魔法作用到 `firstAddressStreet` 例子上，看看它的實際作用：

```
//  join :: Monad m => m (m a) -> m a
var join = function(mma){ return mma.join(); }

//  firstAddressStreet :: User -> Maybe Street
var firstAddressStreet = compose(
  join, map(safeProp('street')), join, map(safeHead), safeProp('
addresses')
);

firstAddressStreet(
  {addresses: [{street: {name: 'Mulburry', number: 8402}, postco
de: "WC2N" }]}
);
// Maybe({name: 'Mulburry', number: 8402})
```

只要遇到巢狀的 `Maybe`，就加一個 `join`，防止它們從手中溜走。我們對 `IO` 也這麼做試試看，感受下這種感覺。

```
IO.prototype.join = function() {
  return this.unsafePerformIO();
}
```

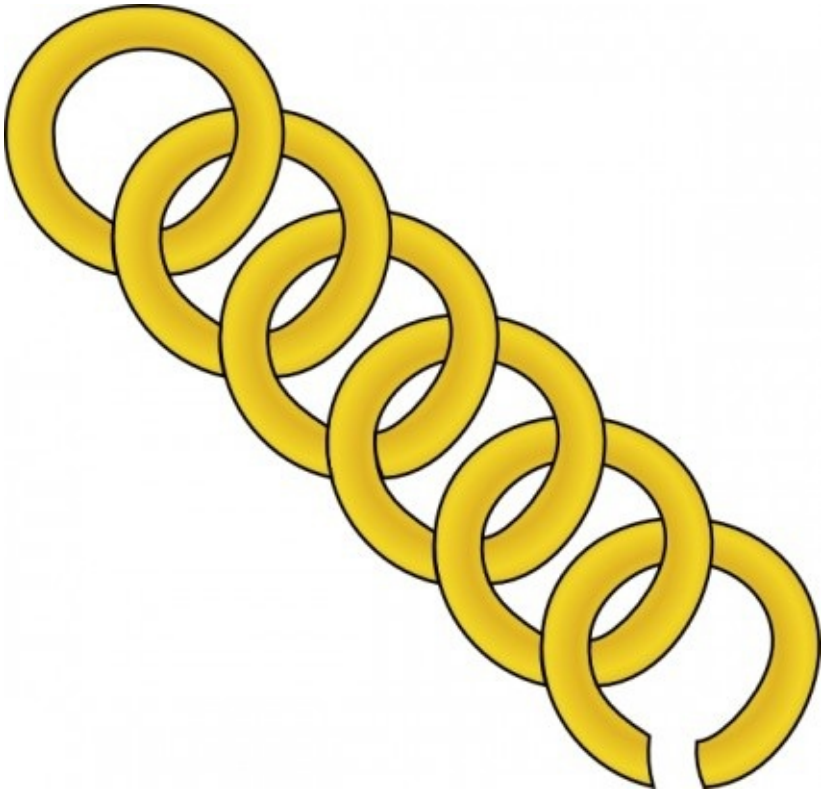
同樣是簡單地移除了一層容器。注意，我們還沒有提及純粹性的問題，僅僅是移除過度緊縮的包裹中的一層而已。

```
//  log :: a -> IO a
var log = function(x) {
  return new IO(function() { console.log(x); return x; });
}

//  setStyle :: Selector -> CSSProps -> IO DOM
var setStyle = curry(function(sel, props) {
  return new IO(function() { return jQuery(sel).css(props); });
});

//  getItem :: String -> IO String
var getItem = function(key) {
  return new IO(function() { return localStorage.getItem(key); }
);
};

//  applyPreferences :: String -> IO DOM
var applyPreferences = compose(
  join, map(setStyle('#main')), join, map(log), map(JSON.parse),
 getItem
);


applyPreferences('preferences').unsafePerformIO();
// Object {backgroundColor: "green"}
// <div style="background-color: 'green'"/>
```

`getItem` 返回了一個 `IO String`，所以可以直接用 `map` 來解析它。`log` 和 `setStyle` 返回的都是 `IO`，所以必須要使用 `join` 來保證這裡邊的巢狀處於控制之中。

chain 函式

（譯者注：此處標題原文是“My chain hits my chest”，是英國歌手 M.I.A 單曲 *Bad Girls* 的一句歌詞。據說這首歌有體現女權主義。）



你可能已經從上面的例子中注意到這種模式了：我們總是在緊跟著 `map` 的後面呼叫 `join`。讓我們把這個行爲抽象到一個叫做 `chain` 的函式裡。

```
//  chain :: Monad m => (a -> m b) -> m a -> m b
var chain = curry(function(f, m){
  return m.map(f).join(); // 或者 compose(join, map(f))(m)
});
```

這裡僅僅是把 map/join 套餐打包到一個單獨的函式中。如果你之前瞭解過 monad，那你可能已經看出來 `chain` 叫做 `>>=`（讀作 bind）或者 `flatMap`；都是同一個概念的不同名稱罷了。我個人認爲 `flatMap` 是最準確的名稱，但本書還是堅持使用 `chain`，因爲它是 JS 裡接受程度最高的一個。我們用 `chain` 重構下上面兩個例子：

```
// map/join
var firstAddressStreet = compose(
  join, map(safeProp('street')), join, map(safeHead), safeProp('
addresses')
);

// chain
var firstAddressStreet = compose(
  chain(safeProp('street')), chain(safeHead), safeProp('addresse
s')
);



// map/join
var applyPreferences = compose(
  join, map(setStyle('#main')), join, map(log), map(JSON.parse),
 getItem
);

// chain
var applyPreferences = compose(
  chain(setStyle), chain(log), map(JSON.parse), getItem
);
```

我把所有的 `map/join` 都替換爲了 `chain`，這樣程式碼就顯得整潔了些。整潔固然是好事，但 `chain` 的能力卻不止於此——它更多的是龍捲風而不是吸塵器。因爲 `chain` 可以輕鬆地巢狀多個作用，因此我們就能以一種純函數式的方式來表示序列（sequence）和變數賦值（variable assignment）。

```
// getJSON :: Url -> Params -> Task JSON
// querySelector :: Selector -> IO DOM


getJSON('/authenticate', {username: 'stale', password: 'crackers'
})
  .chain(function(user) {
    return getJSON('/friends', {user_id: user.id});
});
// Task([{name: 'Seimith', id: 14}, {name: 'Ric', id: 39}]);


querySelector("input.username").chain(function(uname) {
  return querySelector("input.email").chain(function(email) {
    return IO.of(
      "Welcome " + uname.value + " " + "prepare for spam at " +
email.value
    );
  });
});
// IO("Welcome Olivia prepare for spam at olivia@tremorcontrol.n
et");


Maybe.of(3).chain(function(three) {
  return Maybe.of(2).map(add(three));
});
// Maybe(5);


Maybe.of(null).chain(safeProp('address')).chain(safeProp('street'
));
// Maybe(null);
```

本來我們可以用 `compose` 寫上面的例子，但這將需要幾個幫助函式，而且這種風格怎麼說都要通過閉包進行明確的變數賦值。相反，我們使用了插入式的 `chain` 。順便說一下， `chain` 可以自動從任意型別的 `map` 和 `join` 衍生出來，就像這樣： `t.prototype.chain = function(f) { return`

`this.map(f).join(); }`。如果手動定義 `chain` 能讓你覺得效能會好點的話（實際上並不會），我們也可以手動定義它，儘管還必須要費力保證函式功能的正確性——也就是說，它必須與緊接著後面有 `join` 的 `map` 相等。如果 `chain` 是簡單地通過結束呼叫 `of` 後把值放回容器這種方式定義的，那麼就會造成一個有趣的後果，即可以從 `chain` 那裡衍生出一個 `map`。同樣地，我們還可以用 `chain(id)` 定義 `join`。聽起來好像是在跟魔術師玩德州撲克，魔術師想要什麼牌就有什麼牌；但是就像大部分的數學理論一樣，所有這些原則性的結構都是相互關聯的。fantasyland 倉庫中提到了許多上述衍生概念，這個倉庫也是 JavaScript 官方的代數資料結構（algebraic data types）標準。

好了，我們來看上面的例子。第一個例子中，可以看到兩個 `Task` 通過 `chain` 連線形成了一個非同步操作的序列——它先獲取 `user`，然後用 `user.id` 查詢 `user` 的 `friends`。`chain` 避免了 `Task(Task([Friend]))` 這種情況。

第二個例子是用 `querySelector` 查詢幾個 input 然後建立一條歡迎資訊。注意看我們是如何在最內層的函式裡訪問 `uname` 和 `email` 的——這是函數式變數賦值的絕佳表現。因爲 `IO` 大方地把它的值借給了我們，我們也要負起以同樣方式把值放回去的責任——不能辜負它的信任（還有整個程式的信任）。`IO.of` 非常適合做這件事，同時它也解釋了爲何 pointed 這一特性是 monad 介面得以存在的重要前提。不過，`map` 也能返回正確的型別：

```
querySelector("input.username").chain(function(uname) {
  return querySelector("input.email").map(function(email) {
    return "Welcome " + uname.value + " prepare for spam at " +
email.value;
  });
});
// IO("Welcome Olivia prepare for spam at olivia@tremorcontrol.n
et");
```

最後兩個例子用了 `Maybe`。因爲 `chain` 其實是在底層呼叫了 `map`，所以如果遇到 `null`，程式碼就會立刻停止執行。

如果覺得這些例子不太容易理解，你也不必擔心。多跑跑程式碼，多琢磨琢磨，把程式碼拆開來研究研究，再把它們拼起來看看。總之記住，返回的如果是“普通”值就用 `map`，如果是 `functor` 就用 `chain`。

這裡我得提醒一下，上述方式對兩個不同型別的巢狀容器是不適用的。functor 組合，以及後面會講到的 monad transformer 可以幫助我們應對這種情況。

炫耀

這種容器程式設計風格有時也能造成困惑，我們不得不努力理解一個值到底嵌套了幾層容器，或者需要用 `map` 還是 `chain`（很快我們就會認識更多的容器型別）。使用一些技巧，比如重寫 `inspect` 方法之類，能夠大幅提高 debug 的效率。後面我們也會學習如何建立一個“棧”，使之能夠處理任何丟給它的作用（effects）。不過，有時候也需要權衡一下是否值得這樣做。

我很樂意揮起 monad 之劍，向你展示這種程式設計風格的力量。就以讀一個檔案，然後就把它直接上傳爲例吧：

```
// readFile :: Filename -> Either String (Future Error String)
// httpPost :: String -> Future Error JSON

//  upload :: String -> Either String (Future Error JSON)
var upload = compose(map(chain(httpPost('/uploads'))), readFile)
;
```

這裡，程式碼不止一次在不同的分支執行。從型別簽名可以看出，我們預防了三個錯誤——`readFile` 使用 `Either` 來驗證輸入（或許還有確保檔名存在）；`readFile` 在讀取檔案的時候可能會出錯，錯誤通過 `readFile` 的 `Future` 表示；檔案上傳可能會因爲各種各樣的原因出錯，錯誤通過 `httpPost` 的 `Future` 表示。我們就這麼隨意地使用 `chain` 實現了兩個巢狀的、有序的非同步執行動作。

所有這些操作都是在一個從左到右的線性流中完成的，是完完全全純的、宣告式的程式碼，是可以等式推導（equational reasoning）並擁有可靠特性（reliable properties）的程式碼。我們沒有被迫使用不必要甚至令人困惑的變數名，我們的 `upload` 函式符合通用介面而不是特定的一次性介面。這些都是在一行程式碼中完成的啊！

讓我們來跟標準的命令式的實現對比一下：

```
//  upload :: String -> (String -> a) -> Void
var upload = function(filename, callback) {
  if(!filename) {
    throw "You need a filename!";
  } else {
    readFile(filename, function(err, contents) {
      if(err) throw err;
      httpPost(contents, function(err, json) {
        if(err) throw err;
        callback(json);
      });
    });
  }
}
```

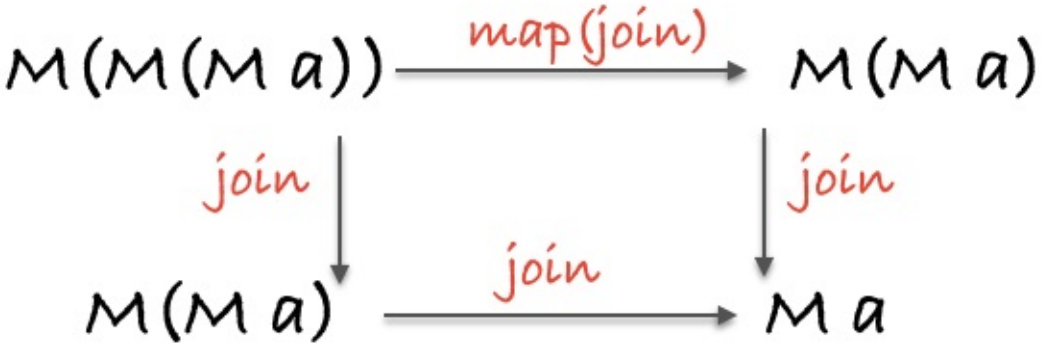
看看，這簡直就是魔鬼的算術（譯者注：此處原文是“the devil's arithmetic”，爲美國 1988 年出版的歷史小說，講述一個猶太小女孩穿越到 1942 年的集中營的故事。此書亦有同名改編電影，中譯名《穿梭集中營》），我們就像一顆彈珠一樣在變幻莫測的迷宮中穿梭。無法想象如果這是一個典型的應用，而且一直在改變變數會怎樣——我們肯定會像陷入瀝青坑那樣無所適從。

理論

我們要看的第一條定律是結合律，但可能不是你熟悉的那個結合律。

```
// 結合律
compose(join, map(join)) == compose(join, join)
```

這些定律表明了 monad 的巢狀本質，所以結合律關心的是如何讓內層或外層的容器型別 `join`，然後取得同樣的結果。用一張圖來表示可能效果會更好：

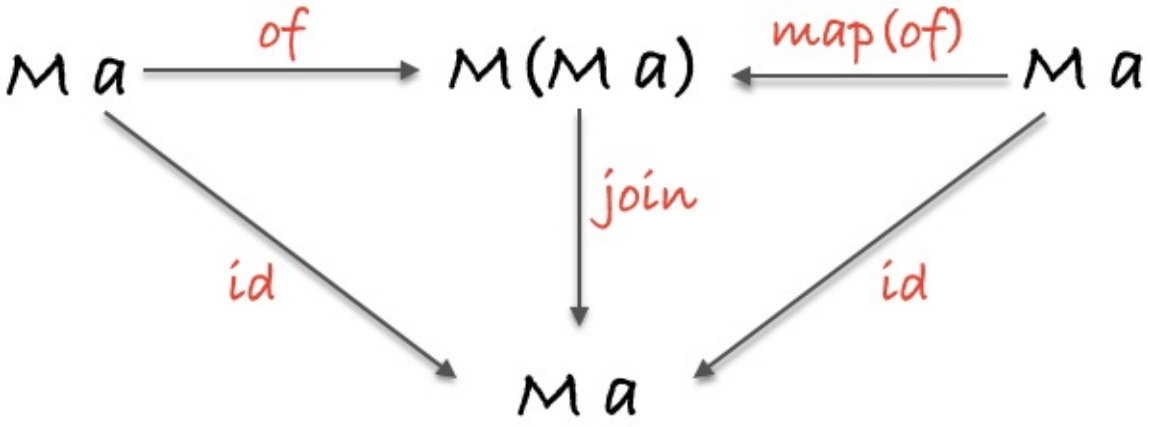


從左上角往下，先用 `join` 合併 `M(M(M a))` 最外層的兩個 `M`，然後往左，再呼叫一次 `join`，就得到了我們想要的 `M a`。或者，從左上角往右，先開啓最外層的 `M`，用 `map(join)` 合併內層的兩個 `M`，然後再向下呼叫一次 `join`，也能得到 `M a`。不管是先合併內層還是先合併外層的 `M`，最後都會得到相同的 `M a`，所以這就是結合律。值得注意的一點是 `map(join) != join`。兩種方式的中間步驟可能會有不同的值，但最後一個 `join` 呼叫後最終結果是一樣的。

第二個定律與結合律類似：

```
// 同一律 (M a)
compose(join, of) == compose(join, map(of)) == id
```

這表明，對任意的 monad `M`，`of` 和 `join` 相當於 `id`。也可以使用 `map(of)` 由內而外實現相同效果。我們把這個定律叫做“三角同一律”（triangle identity），因爲把它圖形化之後就像一個三角形：



如果從左上角開始往右，可以看到 `of` 的確把 `M a` 丟到另一個 `M` 容器裡去了。然後再往下 `join`，就得到了 `M a`，跟一開始就呼叫 `id` 的結果一樣。從右上角往左，可以看到如果我們通過 `map` 進到了 `M` 裡面，然後對普通值 `a` 呼叫 `of`，最後得到的還是 `M (M a)`；再呼叫一次 `join` 將會把我們帶回原點，即 `M a`。

我要說明一點，儘管這裡我寫的是 `of`，實際上對任意的 monad 而言，都必須要使用明確的 `M.of`。

我已經見過這些定律了，同一律和結合律，以前就在哪兒見過...等一下，讓我想想...是的！它們是範疇遵循的定律！不過這意味著我們需要一個組合函式來給出一個完整定義。見證吧：

```
var mcompose = function(f, g) {
  return compose(chain(f), chain(g));
}

// 左同一律
mcompose(M, f) == f

// 右同一律
mcompose(f, M) == f

// 結合律
mcompose(mcompose(f, g), h) == mcompose(f, mcompose(g, h))
```

畢竟它們是範疇學裡的定律。monad 來自於一個叫 “Kleisli 範疇”的範疇，這個範疇裡邊所有的物件都是 monad，所有的態射都是聯結函式（chained funtions）。我不是要在沒有提供太多解釋的情況下，拿範疇學裡各式各樣的概念來取笑你。我的目的是涉及足夠多的表面知識，向你說明這中間的相關性，讓你在關注日常實用特性之餘，激發起對這些定律的興趣。

總結

monad 讓我們深入到巢狀的運算當中，使我們能夠在完全避免回撥金字塔（pyramid of doom）情況下，爲變數賦值，執行有序的作用，執行非同步任務等等。當一個值被困在幾層相同型別的容器中時，monad 能夠拯救它。藉助

“pointed” 這個可靠的幫手，monad 能夠借給我們從盒子中取出的值，而且知道我們會在結束使用後還給它。

是的，monad 非常強大，但我們還需要一些額外的容器函式。比如，假設我們想同時執行一個列表裡的 api 呼叫，然後再蒐集返回的結果，怎麼辦？是可以使用 monad 實現這個任務，但必須要等每一個 api 完成後才能呼叫下一個。合併多個合法性驗證呢？我們想要的肯定是持續驗證以蒐集錯誤列表，但是 monad 會在第一個 `Left` 登場的時候停掉整個演出。

下一章，我們將看到 applicative functor 如何融入這個容器世界，以及爲何在很多情況下它比 monad 更好用。

第 10 章: Applicative Functor

練習

```
// 練習 1
// ==========
// 給定一個 user，使用 safeProp 和 map/join 或 chain 安全地獲取 sreet
 的 name

var safeProp = _.curry(function (x, o) { return Maybe.of(o[x]);
});
var user = {
  id: 2,
  name: "albert",
  address: {
    street: {
      number: 22,
      name: 'Walnut St'
    }
  }
};

var ex1 = undefined;


// 練習 2
// ==========
```

```
// 使用 getFile 獲取檔名並刪除目錄，所以返回值僅僅是檔案，然後以純的方式列印檔案

var getFile = function() {
  return new IO(function(){ return __filename; });
}

var pureLog = function(x) {
  return new IO(function(){
    console.log(x);
    return 'logged ' + x;
  });
}

var ex2 = undefined;



// 練習 3
// ==========
// 使用 getPost() 然後以 post 的 id 呼叫 getComments()
var getPost = function(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res({ id: i, title: 'Love them tasks' });
    }, 300);
  });
}

var getComments = function(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res([
        {post_id: i, body: "This book should be illegal"},
        {post_id: i, body: "Monads are like smelly shallots"}
      ]);
    }, 300);
  });
}
```

```
var ex3 = undefined;


// 練習 4
// ==========
// 用 validateEmail、addToMailingList 和 emailBlast 實現 ex4 的型別簽名

//  addToMailingList :: Email -> IO([Email])
var addToMailingList = (function(list){
  return function(email) {
    return new IO(function(){
      list.push(email);
      return list;
    });
  }
})([]);

function emailBlast(list) {
  return new IO(function(){
    return 'emailed: ' + list.join(',');
  });
}

var validateEmail = function(x){
  return x.match(/\S+@\S+\.\S+/) ? (new Right(x)) : (new Left('invalid email'));
}

//  ex4 :: Email -> Either String (IO String)
var ex4 = undefined;
```

Applicative Functor

應用 applicative functor

考慮到其函數式的出身，**applicative functor** 這個名稱堪稱簡單明瞭。函數式程式設計師最爲人詬病的一點就是，總喜歡搞一些稀奇古怪的命名，比如 `mappend` 或者 `liftA4`。誠然，此類名稱出現在數學實驗室是再自然不過的，但是放在其他任何語境下，這些概念就都像是扮作達斯維達去汽車餐館搞怪的人。（譯者注：此處需要做些解釋，1. 汽車餐館（drive-thru）指的是那種不需要顧客下車就能提供服務的地方，比如麥當勞、星巴克等就會有這種 drive-thru；2. 達斯維達（Darth Vader）是《星球大戰》系列主要反派角色，在美國大眾文化中的有著廣泛的影響力，其造型是很多人致敬模仿的物件；3. 由於 2 的緣故，美國一些星戰迷會扮作 Darth Vader 去 drive-thru 點單，YouTube 上有不少這種搞怪視訊；4. 作者使用這個“典故”是爲了說明函數式裡很多概念的名稱有些“故弄玄虛”，而 applicative functor 是少數比較“正常”的。）

無論如何，applicative 這個名字應該能夠向我們表明一些事實，告訴我們作爲一個介面，它能爲我們帶來什麼：那就是讓不同 functor 可以相互應用（apply）的能力。

然而，你可能爲會問了，爲何一個正常的、理性的人，比如你自己，會做這種“讓不同 functor 相互應用”的事？而且，“相互應用”到底是什麼意思？

要回答這些問題，我們可以從下面這個場景講起，可能你已經碰到過這種場景了。假設有兩個同類型的 functor，我們想把這兩者作爲一個函式的兩個引數傳遞過去來呼叫這個函式。簡單的例子比如讓兩個 `Container` 的值相加：

```
// 這樣是行不通的，因爲 2 和 3 都藏在瓶子裡。
add(Container.of(2), Container.of(3));
//NaN

// 使用可靠的 map 函式試試
var container_of_add_2 = map(add, Container.of(2));
// Container(add(2))
```

這時候我們建立了一個 `Container`，它內部的值是一個區域性呼叫的（partially applied）的函式。確切點講就是，我們想讓 `Container(add(2))` 中的 `add(2)` 應用到 `Container(3)` 中的 `3` 上來完成呼叫。也就是說，我們想把一個 functor 應用到另一個上。

巧的是，完成這種任務的工具已經存在了，即 `chain` 函式。我們可以先 `chain` 然後再 `map` 那個區域性呼叫的 `add(2)`，就像這樣：

```
Container.of(2).chain(function(two) {
  return Container.of(3).map(add(two));
});
```

只不過，這種方式有一個問題，那就是 monad 的順序執行問題：所有的程式碼都只會在前一個 monad 執行完畢之後才執行。想想看，我們的這兩個值足夠強健且相互獨立，如果僅僅爲了滿足 monad 的順序要求而延遲 `Container(3)` 的建立，我覺得是非常沒有必要的。

事實上，當遇到這種問題的時候，要是能夠無需藉助這些不必要的函式和變數，以一種簡明扼要的方式把一個 functor 的值應用到另一個上去就好了。

瓶中之船

`ap` 就是這樣一種函式，能夠把一個 functor 的函式值應用到另一個 functor 的值上。把這句話快速地說上 5 遍。

```
Container.of(add(2)).ap(Container.of(3));
// Container(5)

// all together now
Container.of(2).map(add).ap(Container.of(3));
// Container(5)
```

這樣就大功告成了，而且程式碼乾淨整潔。可以看到，`Container(3)` 從巢狀的 monad 函式的牢籠中釋放了出來。需要再次強調的是，本例中的 `add` 是被 `map` 所區域性呼叫（partially apply）的，所以 `add` 必須是一個 curry 函式。

可以這樣定義一個 `ap` 函式：

```
Container.prototype.ap = function(other_container) {
  return other_container.map(this.__value);
}
```

記住，`this.__value` 是一個函式，將會接收另一個 functor 作爲引數，所以我們只需 `map` 它。由此我們可以得出 applicative functor 的定義：

> applicative functor 是實現了 `ap` 方法的 pointed functor

注意 `pointed` 這個前提，這是非常重要的一個前提，下面的例子會說明這一點。

講到這裡，我已經感受到你的疑慮了（也或者是困惑和恐懼）；心態開放點嘛，`ap` 還是很有用的。在深入理解這個概念之前，我們先來探索一個特性。

```
F.of(x).map(f) == F.of(f).ap(F.of(x))
```

這行程式碼翻譯成人類語言就是，map 一個 `f` 等價於 `ap` 一個值爲 `f` 的 functor。或者更好的譯法是，你既可以把 `x` 放到容器裡然後呼叫 `map(f)`，也可以同時讓 `f` 和 `x` 發生 lift（參看第 8 章），然後對他們呼叫 `ap`。這讓我們能夠以一種從左到右的方式編寫程式碼：

```
Maybe.of(add).ap(Maybe.of(2)).ap(Maybe.of(3));
// Maybe(5)

Task.of(add).ap(Task.of(2)).ap(Task.of(3));
// Task(5)
```

細心的讀者可能發現了，上述程式碼中隱約有普通函式呼叫的影子。沒關係，我們稍後會學習 `ap` 的 pointfree 版本；暫時先把這當作此類程式碼的推薦寫法。通過使用 `of`，每一個值都被輸送到了各個容器裡的奇幻之地，就像是在另一個平行世界裡，每個程式都可以是非同步的或者是 null 或者隨便什麼值，而且不管是什麼，`ap` 都能在這個平行世界裡針對這些值應用各種各樣的函式。這就像是在一個瓶子中造船。

你注意到沒？上例中我們使用了 `Task`，這是 applicative functor 主要的用武之地。現在我們來看一個更深入的例子。

協調與激勵

假設我們要建立一個旅遊網站，既需要獲取遊客目的地的列表，還需要獲取地方事件的列表。這兩個請求就是相互獨立的 api 呼叫。

```
// Http.get :: String -> Task Error HTML

var renderPage = curry(function(destinations, events) { /* rende
r page */  });

Task.of(renderPage).ap(Http.get('/destinations')).ap(Http.get('/
events'))
// Task("<div>some page with dest and events</div>")
```

兩個請求將會同時立即執行，當兩者的響應都返回之後，`renderPage` 就會被呼叫。這與 monad 版本的那種必須等待前一個任務完成才能繼續執行後面的操作完全不同。本來我們就無需根據目的地來獲取事件，因此也就不需要依賴順序執行。

再次強調，因爲我們是使用區域性呼叫的函式來達成上述結果的，所以必須要保證 `renderpage` 是 curry 函式，否則它就不會一直等到兩個 `Task` 都完成。而且如果你碰巧自己做過類似的事，那你一定會感激 `applicative functor` 這個異常

簡潔的介面的。這就是那種能夠讓我們離“奇點”（singularity）更近一步的優美程式碼。

再來看另外一個例子。

```
// 幫助函式：
// ==============
//  $ :: String -> IO DOM
var $ = function(selector) {
  return new IO(function(){ return document.querySelector(select
or) });
}

//  getVal :: String -> IO String
var getVal = compose(map(_.prop('value')), $);

// Example:
// ===============
//  signIn :: String -> String -> Bool -> User
var signIn = curry(function(username, password, remember_me){ /*
 signing in */  })

IO.of(signIn).ap(getVal('#email')).ap(getVal('#password')).ap(IO
.of(false));
// IO({id: 3, email: "gg@allin.com"})
```

`signIn` 是一個接收 3 個引數的 curry 函式，因此我們需要呼叫 `ap` 3 次。在每一次的 `ap` 呼叫中，`signIn` 就收到一個引數然後執行，直到所有的引數都傳進來，它也就執行完畢了。我們可以繼續擴充套件這種模式，處理任意多的引數。另外，左邊兩個引數在使用 `getVal` 呼叫後自然而然地成爲了一個 `IO`，但是最右邊的那個卻需要手動 `lift`，然後變成一個 `IO`，這是因爲 `ap` 需要呼叫者及其引數都屬於同一型別。

lift

（譯者注：此處原標題是“Bro, do you even lift?”，是一流行語，發源於健身圈，指質疑別人的健身方式和效果並顯示優越感，後擴散至其他領域。再注：作者書中用了不少此類俚語或俗語，有時並非在使用俚語的本意，就像這句，完全就是爲了好

玩。另，關於 lift 的概念可參看第 8 章。）

我們來試試以一種 pointfree 的方式呼叫 applicative functor。因爲 `map` 等價於 `of/ap` ，那麼我們就可以定義無數個能夠 `ap` 通用函式。

```
var liftA2 = curry(function(f, functor1, functor2) {
  return functor1.map(f).ap(functor2);
});

var liftA3 = curry(function(f, functor1, functor2, functor3) {
  return functor1.map(f).ap(functor2).ap(functor3);
});

//liftA4, etc
```

`liftA2` 是個奇怪的名字，聽起來像是破敗工廠裡挑剔的貨運電梯，或者僞豪華汽車公司的個性車牌。不過你要是眞正理解了，那麼它的含義也就不證自明瞭：讓那些小程式碼塊發生 lift，成爲 applicative functor 中的一員。

剛開始我也覺得這種 2-3-4 的寫法沒什麼意義，看起來又醜又沒有必要，畢竟我們可以在 JavaScript 中檢查函式的引數數量然後再動態地構造這樣的函式。不過，區域性呼叫（partially apply） `liftA(N)` 本身，有時也能發揮它的用處，這樣的話，引數數量就固定了。

來看看實際用例：

```
// checkEmail :: User -> Either String Email
// checkName :: User -> Either String String

//  createUser :: Email -> String -> IO User
var createUser = curry(function(email, name) { /* creating... */
 });

Either.of(createUser).ap(checkEmail(user)).ap(checkName(user));
// Left("invalid email")

liftA2(createUser, checkEmail(user), checkName(user));
// Left("invalid email")
```

`createUser` 接收兩個引數，因此我們使用的是 `liftA2` 。上述兩個語句是等價的，但是使用了 `liftA2` 的版本沒有提到 `Either` ，這就使得它更加通用靈活，因爲不必與特定的資料型別耦合在一起。

我們試試以這種方式重寫前一個例子：

```
liftA2(add, Maybe.of(2), Maybe.of(3));
// Maybe(5)

liftA2(renderPage, Http.get('/destinations'), Http.get('/events'
))
// Task("<div>some page with dest and events</div>")

liftA3(signIn, getVal('#email'), getVal('#password'), IO.of(false
));
// IO({id: 3, email: "gg@allin.com"})
```

操作符

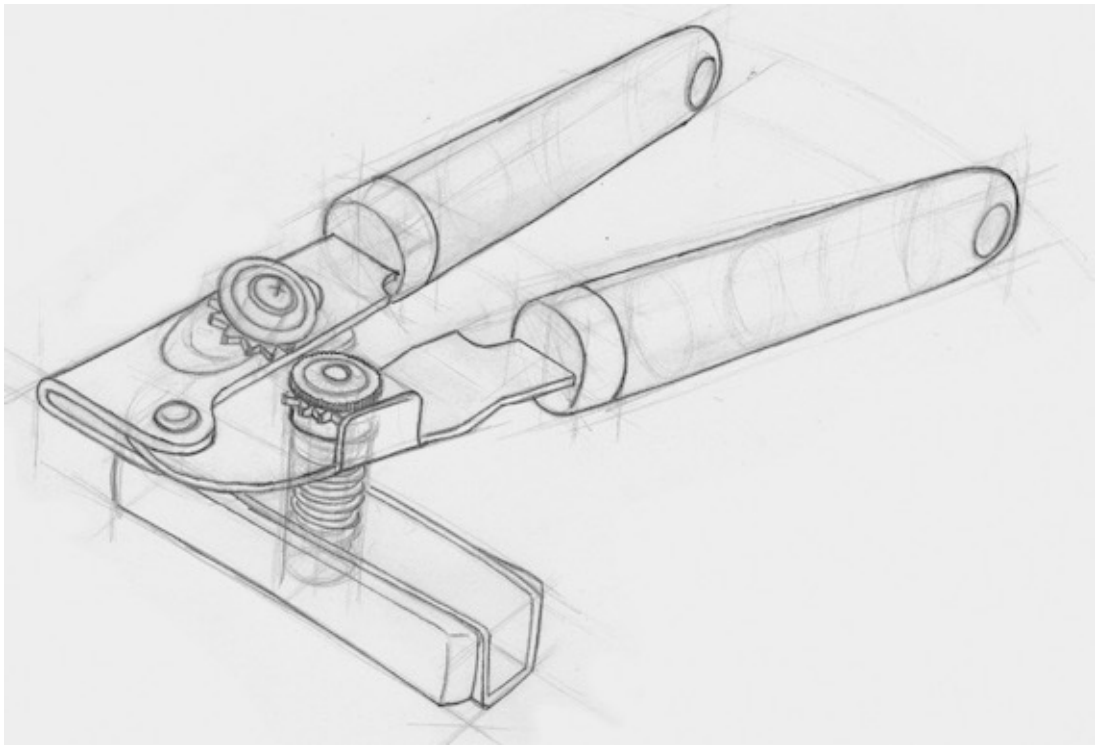
在 haskell、scala、PureScript 以及 swift 等語言中，開發者可以建立自定義的中綴操作符（infix operators），所以你能看到到這樣的語法：

```
-- haskell
add <$> Right 2 <*> Right 3
```

```
// JavaScript
map(add, Right(2)).ap(Right(3))
```

`<$>` 就是 `map` （亦即 `fmap` ），`<*>` 不過就是 `ap` 。這樣的語法使得開發者可以以一種更自然的風格來書寫函數式應用，而且也能減少一些括號。

免費開瓶器



我們尚未對衍生函式（derived function）著墨過多。不過看到本書介紹的所有這些介面都互相依賴並遵守一些定律，那麼我們就可以根據一些強介面來定義一些弱介面了。

比如，我們知道一個 applicative 首先是一個 functor，所以如果已經有一個 applicative 例項的話，毫無疑問可以依此定義一個 functor。

這種完美的計算上的大和諧（computational harmony）之所以存在，是因爲我們在跟一個數學“框架”打交道。哪怕是莫扎特在小時候就下載了 ableton（譯者注：一款專業的音樂製作軟體），他的鋼琴也不可能彈得更好。

前面提到過，`of/ap` 等價於 `map`，那麼我們就可以利用這點來定義 `map`：

```
// 從 of/ap 衍生出的 map
X.prototype.map = function(f) {
  return this.constructor.of(f).ap(this);
}
```

monad 可以說是處在食物鏈的頂端，因此如果已經有了一個 `chain` 函式，那麼就可以免費得到 functor 和 applicative：

```
// 從 chain 衍生出的 map
X.prototype.map = function(f) {
  var m = this;
  return m.chain(function(a) {
    return m.constructor.of(f(a));
  });
}

// 從 chain/map 衍生出的 ap
X.prototype.ap = function(other) {
  return this.chain(function(f) {
    return other.map(f);
  });
};
```

定義一個 monad，就既能得到 applicative 也能得到 functor。這一點非常強大，相當於這些“開瓶器”全都是免費的！我們甚至可以審查一個數據型別，然後自動化這個過程。

應該要指出來的一點是，`ap` 的魅力有一部分就來自於並行的能力，所以通過 `chain` 來定義它就失去了這種優化。即便如此，開發者在設計出最佳實現的過程中就能有一個立即可用的介面，也是很好的。

爲啥不直接使用 monad？因爲最好用合適的力量來解決合適的問題，一分不多，一分不少。這樣就能通過排除可能的功能性來做到最小化認知負荷。因爲這個原因，相比 monad，我們更傾向於使用 applicative。

向下的巢狀結構使得 monad 擁有序列計算、變數賦值和暫緩後續執行等獨特的能力。不過見識到 applicative 的實際用例之後，你就不必再考慮上面這些問題了。

下面，來看看理論知識。

定律

就像我們探索過的其他數學結構一樣，我們在日常編碼中也依賴 applicative functor 一些有用的特性。首先，你應該知道 applicative functor 是“組合關閉”（closed under composition）的，意味著 `ap` 永遠不會改變容器型別（另一個勝過 monad

的原因）。這並不是說我們無法擁有多種不同的作用——我們還是可以把不同的型別壓棧的，只不過我們知道它們將會在整個應用的過程中保持不變。

下面的例子可以說明這一點：

```
 var tOfM = compose(Task.of, Maybe.of);

 liftA2(_.concat, tOfM('Rainy Days and Mondays'), tOfM(' always
get me down'));
 // Task(Maybe(Rainy Days and Mondays always get me down))
```

你看，不必擔心不同的型別會混合在一起。

該去看看我們最喜歡的範疇學定律了：同一律（identity）。

同一律（identity）

```
// 同一律
A.of(id).ap(v) == v
```

是的，對一個 functor 應用 `id` 函式不會改變 `v` 裡的值。比如：

```
var v = Identity.of("Pillow Pets");
Identity.of(id).ap(v) == v
```

`Identity.of(id)` 的“無用性”讓我不禁莞爾。這裡有意思的一點是，就像我們之前證明了的，`of/ap` 等價於 `map`，因此這個同一律遵循的是 functor 的同一律：`map(id) == id`。

使用這些定律的優美之處在於，就像一個富有激情的幼兒園健身教練讓所有的小朋友都能愉快地一塊玩耍一樣，它們能夠強迫所有的介面都能完美結合。

同態（homomorphism）

```
// 同態
A.of(f).ap(A.of(x)) == A.of(f(x))
```

同態就是一個能夠保持結構的對映（structure preserving map）。實際上，functor 就是一個在不同範疇間的同態，因爲 functor 在經過對映之後保持了原始範疇的結構。

事實上，我們不過是把普通的函式和值放進了一個容器，然後在裡面進行各種計算。所以，不管是把所有的計算都放在容器裡（等式左邊），還是先在外面進行計算然後再放到容器裡（等式右邊），其結果都是一樣的。

一個簡單例子：

```
Either.of(_.toUpper).ap(Either.of("oreos")) == Either.of(_.toUpp
er("oreos"))
```

互換（interchange）

互換（interchange）表明的是選擇讓函式在 `ap` 的左邊還是右邊發生 lift 是無關緊要的。

```
// 互換
v.ap(A.of(x)) == A.of(function(f) { return f(x) }).ap(v)
```

這裡有個例子：

```
var v = Task.of(_.reverse);
var x = 'Sparklehorse';

v.ap(Task.of(x)) == Task.of(function(f) { return f(x) }).ap(v)
```

組合（composition）

最後是組合。組合不過是在檢查標準的函式組合是否適用於容器內部的函式呼叫。

```
// 組合
A.of(compose).ap(u).ap(v).ap(w) == u.ap(v.ap(w));
```

```
var u = IO.of(_.toUpper);
var v = IO.of(_.concat("& beyond"));
var w = IO.of("blood bath ");

IO.of(_.compose).ap(u).ap(v).ap(w) == u.ap(v.ap(w))
```

總結

處理多個 functor 作爲引數的情況，是 applicative functor 一個非常好的應用場景。藉助 applicative functor，我們能夠在 functor 的世界裡呼叫函式。儘管已經可以通過 monad 達到這個目的，但在不需要 monad 的特定功能的時候，我們還是更傾向於使用 applicative functor。

至此我們已經基本介紹完容器的 api 了，我們學會了如何對函式呼叫 `map` 、 `chain` 和 `ap` 。下一章，我們將學習如何更好地處理多個 functor，以及如何以一種原則性的方式拆解它們。

Chapter 11: Traversable/Foldable Functors

練習

```
require('./support');
var Task = require('data.task');
var _ = require('ramda');

// 模擬瀏覽器的 localStorage 物件
var localStorage = {};



// 練習 1
// ==========
// 寫一個函式，使用 Maybe 和 ap() 實現讓兩個可能是 null 的數值相加。

//  ex1 :: Number -> Number -> Maybe Number
var ex1 = function(x, y) {
```

```
};


// 練習 2
// ==========
// 寫一個函式，接收兩個 Maybe 爲引數，讓它們相加。使用 liftA2 代替 ap()。


//  ex2 :: Maybe Number -> Maybe Number -> Maybe Number
var ex2 = undefined;



// 練習 3
// ==========
// 執行 getPost(n) 和 getComments(n)，兩者都執行完畢後執行渲染頁面的操作。（引數 n 可以是任意值）。

var makeComments = _.reduce(function(acc, c){ return acc+"<li>"+c+"</li>" }, "");
var render = _.curry(function(p, cs) { return "<div>"+p.title+"</div>"+makeComments(cs); });

//  ex3 :: Task Error HTML
var ex3 = undefined;



// 練習 4
// ==========
// 寫一個 IO，從快取中讀取 player1 和 player2，然後開始遊戲。

localStorage.player1 = "toby";
localStorage.player2 = "sally";

var getCache = function(x) {
  return new IO(function() { return localStorage[x]; });
}
var game = _.curry(function(p1, p2) { return p1 + ' vs ' + p2; }
);
```

```
//  ex4 :: IO String
var ex4 = undefined;




// 幫助函式
// ====================

function getPost(i) {
  return new Task(function (rej, res) {
    setTimeout(function () { res({ id: i, title: 'Love them futu
res' }); }, 300);
  });
}

function getComments(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res(["This book should be illegal", "Monads are like space
 burritos"]);
    }, 300);
  });
}
```