程式語言的特性本質（一）靜態語言與動態語言的信任抉擇

iThome 網站首載：[程式語言的特性本質（一）靜態語言與動態語言的信任抉擇]

單就撰寫程式本身來說，程式開發就是以程式語言表述如何解決問題的過程，不同語言擁有各自的表述元素，決定了解決問題時的描述方式，勾勒出不同開發生態與成效，因此選用何種程式語言，一直都是熱門爭議焦點。現今語言發展多樣化，雖然語法巧妙各有不同，然而主流程式語言特性不出幾類，像是靜態語言與動態語言、原型語言與類別語言、函數式語言與物件導向語言等，在決定語言前，先辨識這些特性的本質是否適用於需求，才不至於一頭栽入語法細節的混亂。

辨識靜、動型態系統

型態系統是對底層位元組的抽象化，例如"這些位元組是個字串"，型態讓開發者免於處理底層細節，而可以使用高階型態來描述與操作，例如"把這個字串轉為大寫"，而不是"對這些位元組作某運算"。有了型態系統，開發者只要瞭解型態，就可得知如何以高階操作處理資料，開發者選擇程式語言的第一步，就是如何從語法得知型態資訊，也就是決定選用靜態語言或動態語言。

具體來說，靜態語言是指變數是否帶有資料型態，反之則為動態語言。

靜態語言的變數本身帶有型態資訊。例如底下的Java範例中，`text`宣告為`String`，就僅可參考`String`實例，若嘗試參考至其它型態，則會引發編譯錯誤：

```
String text = "programmer";
text = {'p', 'r', 'o'}; // 編譯錯誤
```

動態語言的變數只用來參考資料，本身不帶有型態資訊。例如底下的Python範例中，`text`可參考至`str`實例，也可參考至`list`實例：

```
text = 'programmer'
text = ['p', 'r', 'o']
```

單看這兩個例子，似乎可從宣告變數時是否撰寫型態資訊，來決定語言為動態或靜態，然而有些靜態語言具有型態推論（Type inference）特性，可依程式前後文判定變數型態。例如

ala是靜態語言，但底下範例宣告text時並沒有撰寫型態資訊，但text確實是String型態：

```
val text = "Hello"
```

雖然沒有具體寫出String，但可從"Hello"判斷text是String型態，其實它的完整語法是：

```
val text: String = "Hello"
```

靜態語言的優缺點

靜態語言的優點是可避免執行時期型態錯誤、提供重構輔助與更多的文件形式。

靜態語言明確要求程式中提供型態資訊，因而可透過編譯器或工具，在程式執行前（編譯時期）就檢查出型態錯誤。例如底下的Java範例：

```
// doQuack() 可以傳入什麼資料？
void doQuack(Duck duck) {
    // duck 可以作些什麼？
}
```

在實作doQuack()方法時，若嘗試透過duck變數操作Duck沒有定義的方法或屬性，或者呼叫doQuack()方法時傳入非Duck物件，都會引發編譯錯誤。由於變數本身帶有型態資訊，可據此為基礎來設計相關重構時的輔助操作，例如提取方法（Extract method）時，根據提取的程式片段前後文資訊，在自動產生的方法中建立對應型態的參數與傳回型態。

型態資訊也可作為一種文件形式。以上例來說，在沒有揭露doQuack()方法的實作內容下，doQuack()使用者可從參數型態明確得知doQuack()方法接受Duck實例，doQuack()實作者亦可查閱Duck的API文件，可得知傳入物件的可操作方法或屬性，如果有個方法接受多個引數，從參數的型態也可得知呼叫方法時引數的傳入順序；另一種文件形式則可透過工具來達成，例如在編輯器中顯示Duck可用方法清單與相關文件檢索資訊。

靜態語言的缺點是程式語法繁瑣、彈性不足，只能檢查出執行時期前的簡單錯誤。

由於變數在宣告時必須指定型態資訊，使得程式碼撰寫起來特別囉嗦。有些靜態語言可使用類型推論來解決語法繁瑣問題，先前的Scala示範就是一例，然而開發者必須熟悉語言的類型推論框架，這又容易造成語法複雜或程式碼維護上的額外負擔。

靜態語言另一個問題就是較缺乏彈性。例如Java陣列元素必須是同種型態，若陣列中要放置異質元素，必須有類別繼承或介面實作的型態關係。例如：

```
Object[] person = {1, "programmer", new Date()};
```

._ 刊用到Java中Object類別為所有物件父類別，以解決異質陣列問題，然而要操作特定型態時就得進行轉型，這又造成囉嗦的語法。例如：

```
Integer id = (Integer) person[0];
String name = (String) person[1];
Date loginTime = (Date) person[2];
```

雖然靜態語言可於執行程式前檢查出型態錯誤，但沒有辦法檢查像是陣列邊界、除零、無窮遞迴等執行時期問題，單元測試依舊是必要的，這也是靜態語言反對者常持有的論調：「既然都要依賴更全面性的單元測試，為何要忍受靜態語言帶來的困擾？」

動態語言的優缺點

動態語言的優點是語法簡潔、具有較高的彈性。

動態語言的變數在使用者不需要指定型態資訊，最直接效益就是節省打字的功夫。例如Python在宣告函式時就簡潔許多：

```
def doquack(duck):
    duck.quack()
```

變數不需要型態資訊，所以只要宣告duck名稱，傳回型態也無需宣告，相較於靜態語言來說，著實簡潔許多。既然變數不需要型態資訊，開發者操作變數時也無需思考型態問題，而只要思考變數參考的物件擁有哪些行為，也是動態語言界流行的鴨子型態（Duck typing），這樣的特性在日後較易應付事先沒有設想到的型態，例如doquack()一開始是設計給Duck型態的物件使用，若有個物件不是Duck型態，但確實擁有quack()方法，那麼要使用doquack()呱呱叫一下也是可以的。

語言簡潔與彈性的例子，也可以在底下範例中看出：

```
person = [1, 'programmer', datetime.now()]
id = person[0]
name = person[1]
logintime = person[2]
```

list中可以是各種物件，將list中的物件指定給變數也沒有轉型的問題。

動態語言的缺點是型態錯誤在執行時期才會呈現出來，效能表現較不理想，編輯輔助工具較為缺乏，依賴慣例或實體文件來得知API使用方式。

靜態語言是在程式編譯為可執行形式的過程中檢查型態錯誤，動態語言則常採邊剖析邊執行的方式，型態錯誤的程式碼在執行時期才會發現，以doquack()為例，若human參考的物件

沒有quack()方法，那doquack(human)的呼叫，得在執行時期才發現沒有quack()行為而錯誤。動態語言若要發掘出型態錯誤，得依賴覆蓋率更全面的單元測試。由於採邊剖析邊執行的方式，執行時期檢查型態必然造成效能負擔。

由於沒有型態資訊，編輯上的輔助或重構工具通常不足。例如：

```
# 可以傳入什麼物件？
def dosome(x, y):
    x.? # 有哪些行為？
    y.? # 有哪些行為？
```

只從這個函式定義，編輯工具無從得知x、y有哪些方法或屬性可以操作，也就無法作出提示。dosoome()呼叫者又怎麼知道傳遞的引數順序呢？或是x、y參考的物件得具有哪些方法？動態語言界常使用命名慣例作為補救，例如x也許慣例上都接受X型態，而y慣例上都接受Y型態，然而通常還是得查詢文件才能瞭解使用細節。

信任型態約束或慣例約束

靜態語言與動態語言思考時的基礎其實相同，都是物件應當擁有何種行為，靜態語言將行為具體為某型態所擁有，動態語言的行為由物件自身負責。

以先前Java的doQuack()方法及Python的doquack()函式為例，都是要求傳入物件必須擁有quack()方法；靜態語言假設開發者有可能犯錯，傳入不具quack()方法的物件，因此要求以Duck型態定義quack()方法，在編譯時期藉由Duck型態約束可傳入物件，從而避免執行時期檢查或遭遇型態錯誤的負擔；動態語言假設開發者都遵守約定慣例，明白傳入物件該擁有的行為，因而免除型態約束以換取簡潔與彈性。

從信任的角度來看，選擇靜態語言代表信任它會檢查出開發者可能犯下的型態錯誤，由於型態帶來的約束是強制遵守且不能有例外，對成員來自不同文化或水準不一的團隊而言，靜態語言在語法層面採用型態作為約束工具，對型態錯誤的發掘有一定的保障作用；相反地，選擇動態語言代表採用慣例作為約束工具，相信開發者會為自己的程式負責，因而將行為交由物件本身擁有，對於成員有著一致默契、文準與文化的團隊而言，選擇動態語言可獲得較大的彈性來面對需求。
